Extensible Stochastic Search Using Steady State Grammatical Evolution With Applications in System Identification, Controls, and Circuit Design

by

Alan Christianson

A thesis submitted to the Graduate Office

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

SOUTH DAKOTA SCHOOL OF MINES AND TECHNOLOGY

RAPID CITY, SOUTH DAKOTA

2010

Prepared by:

Alan Christianson, Degree Candidate

Approved by:

Dr. Jeff McGough, Major Professor

Dr. Kyle Riley, Chair, Department of Math and Computer Science

---

Dr. John Helsdon, Dean, Graduate Education

Abstract

Certain classes of problems have no direct method for finding solutions, requiring individuals to make educated guesses based on experience and knowledge of the problem domain. Automation using computers is often impractical when creativity is needed, causing the process to be much slower and error prone than computerized solution methods for other problems. An approach that combines the knowledge and creativity of a human with the speed and accuracy of a computer would be far preferable to traditional, manual methods employed by a person.

Grammatical Evolution is a creative, computer-based search method which seeks to evolve solutions to difficult problems with the aid of a domain expert who clearly defines and directs the set of solutions to investigate. A definition of possible expressions is devised based on the user's intuition and experience; the search is allowed to explore the space of possible expressions made within these constraints. A study of this approach was performed, with applications in control theory, system identification, and circuit design. In each case, GE was able to discover highly suitable solutions to the problems presented. The results show GE to be capable of serving as a powerful search tool, particularly for problems of a mathematical nature.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computers are immensely useful tools which allow the automation of tasks that were formerly performed by humans. Computationally intensive tasks can be performed at speeds which are orders of magnitude faster than those achievable by humans working by hand. In spite of the great computational power of the computer, it still depends on instructions from humans in order to solve a problem. There exists a class of problems in which clear, closed-form solutions do not exist. Some of these problems are addressed by humans working by hand, employing guess-and-test approaches and intuition rather than using a clearly-defined algorithm. The strength of computers is the ability to perform many computations in a short period of time, but some problems have solution spaces so large that exhaustive searches are impractical, meaning that some direction is required if a computer will be used to perform the search.

Lacking the intuition of a well-trained human, the software in question must either be supplied with heuristics which approximate the human's intuition or it must incorporate some sort of directed search algorithm to drive it towards better solutions. A group of approaches collectively known as evolutionary algorithms are an example of the latter which have shown the ability to solve the type of problems at which humans excel but computers have traditionally lagged. Although EAs are only one of a number of effective techniques for solving complex problems, this research focuses exclusively on a specific EA variant.

The field of mathematics contains numerous examples of problems which either have no clear approach for finding solutions or require symbolic computation which can be difficult to implement in software. The following document details the results of research using an evolutionary algorithm known as Grammatical Evolution to find solutions to problems which are otherwise difficult to solve using conventional techniques. Chapter 2

discusses the original biological basis for the research being described. Chapter 3 covers the biologically- inspired search algorithms on which this research is based. A full description of the approach used and the reasons it was chosen is found in chapter 4. The remaining chapters describe the specific problems and the results of this investigation.

Chapter 2

Biology

Before discussing the details of Grammatical Evolution (GE), it's necessary to provide some background on the concepts behind GE. GE is based on other, older forms of Evolutionary Algorithms (EA), which in turn are based on the scientific theories of evolution and genetics.

2.1 Genetics and Molecular Biology

Some of the most groundbreaking research done in the 20th century is related to molecular biology and genetics. Scientists discovered the nature of the human genome, the collection of genetic data contained within human cells. This research has led to greater understanding in numerous fields, including computing. Molecular biology and genetics are complicated topics about which entire books have been written. Since the algorithms used in this research are based on a highly simplified genetic model, a general overview of the topic will be presented in order to put the research in context.

The cells in the human body (and the bodies of other living creatures) contain data stored in strands of deoxyribonucleic acid, or DNA. DNA can be thought of as the blueprint of a living creature, encoding the data needed to construct all of the cells within the organism. It is via replication of DNA that new cells are created in living organisms, both to sustain life in an organism and to bring about new life by way of sexual reproduction. The genetic material of two individuals is combined during reproduction to produce one or more offspring which inherit genes from both parents but are genetically distinct from them due to the process by which the parents' genes are combined. Random mutations in the offspring's genes increase the differences between parent and child, contributing to the diversity of a population.

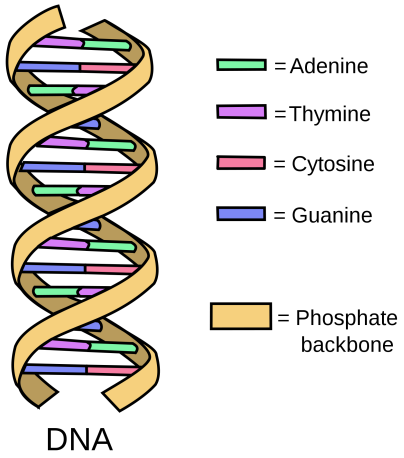


Figure 2.1: A representation of a strand of DNA showing the combination of base pairs which connect the phosphate backbone [21].

DNA consists of long strings of molecules known as nucleotides. These molecules, in turn, are composed of sugars, phosphate groups, and molecules known as nucleobases, or more commonly, bases. The bases which comprise molecules of DNA consist of adenine, guanine, thymine, and cytosine; abbreviated A,G,T, and C respectively [2]. Single units of DNA are known as chromosomes; a complete set of chromosomes needed to define a living organism is known as its genome. Each chromosome in turn consists of many subunits known as codons, sequences of three nucleotides which together encode a specific amino acid. It is these amino acids which are the basic building blocks of proteins. It's worth noting that this genetic code contains redundancy in that there exist 64 distinct codons but only 20 amino acids, meaning that several different codons specify a given amino acid. This feature of the genetic code is known as degeneracy.

The genetic composition of an individual is known as its genotype; the observable traits which are determined by the individual's genotype are known as the phenotype. For example, a certain gene in a human's genome might be responsible for a specific trait such as eye color. A specific variant of a gene is known as an allele. Differing alleles correspond to differences in an individual's phenotype. The general process by which the genetic code creates an observable trait in an individual is known as gene expression and is illustrated in Figure 2.2. Information encoded in nucleic acid flows into proteins, but information never moves back from proteins into nucleic acids, a process Crick describes as *The Central Dogma of Molecular Biology* [15]. A pair of DNA strands is transcribed into a single strand of ribonucleic acid (RNA), with only one of the DNA strands actually being used in the process.

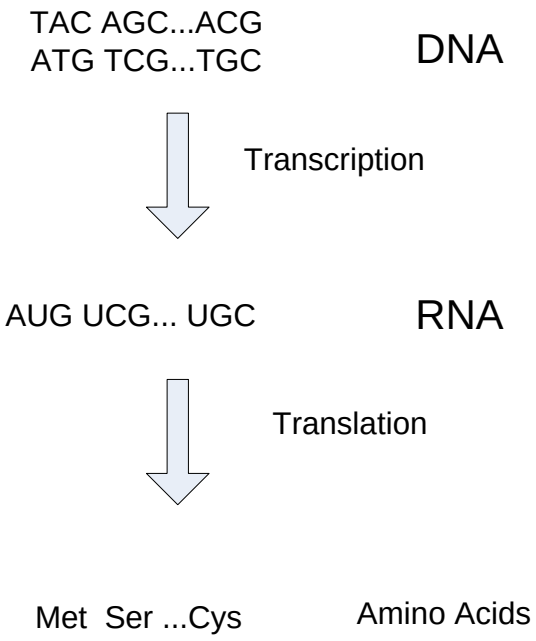


Figure 2.2: An illustration of the process of biological gene expression, showing the DNA being transcribed into RNA, which is in turn translated into amino acids, the building blocks of proteins. The amino acids Methionine, Serine, and Cysteine are shown as the result of translation.

The resulting strand of RNA matches the corresponding strand of DNA, with the exception of thymine in the DNA being replaced by uracil (U) in the RNA. Following transcription, the RNA is used to create amino acids in a process called translation. These amino acids are joined together and the resulting proteins fold, forming complex 3-dimensional structures in combination with other amino acids to create a functional component of the organism [4].

2.2 Evolution

Just as the details of genetic encoding and gene expression provide the basis for representation of individuals in EA systems, a discussion of biological evolution serves to explain the model used to create effective search algorithms. Evolution, put simply, refers to a change in allele frequency over time. The great body of knowledge surrounding evolution began with Darwin's seminal work *On the Origin of Species by Means of Natural Selection* [17] and grew throughout the 20^{th} century to become a comprehensive theory describing the means by which species arise, change, split, and die off over time [24].

The forces of natural selection and genetic drift drive evolution, causing changes in allele frequency in a population. Natural selection refers to the process by which heritable traits which aid an organism in surviving and reproducing become more prevalent in a population over time. Certain adaptations make an organism better suited to its environment than other individuals, increasing its chances to produce more offspring which share its genes. Genetic drift also refers to a change is allele frequency, but unlike natural selection, it is not a result of some selective pressure but is instead a result of random sampling. Over time, a given allele will occur more frequently in a population than in previous generations, due solely to chance. This change may be beneficial, harmful, or have no effect at all on the fitness of the individuals who exhibit the trait.

Over time, these forces combine to create individuals that are well adapted to their environment. Once scientists began to understand the way in which organisms become suited to their environment and how incredibly complex traits arose, some sought to replicate this process artificially, in order to solve complex problems in a similar manner. This desire led to a great variety of approaches in computing, collectively known as Evolutionary Computation (EC). A subset of EC which focuses on search and optimization using an evolving population of individuals is known as Evolutionary Algorithms (EA).

Chapter 3

Evolutionary Algorithms

Scientists and mathematicians saw in Darwinian evolution an immensely powerful tool for creating complex structures. Some sought to replicate evolution in a computer algorithm, thereby evolving solutions to problems. Though numerous approaches to this problem were created, four major branches of EA encompass most of these. Evolutionary Programming (EP), Evolution Strategies (ES), Genetic Algorithms (GA), and Genetic Programming (GP) can be considered the most prominent variations of EA. Some background on each of these approaches is given in the following sections, but more detailed discussions are reserved for Sections 3.3 and 3.4 which cover GAs and GP.

3.1 Evolutionary Programming

In the 60's Lawrence Fogel developed an EA variant while pursuing his Ph.D at UCLA. This approach, based largely on mutation, is known as evolutionary programming [20]. The algorithm is relatively simple, as illustrated in Figure 3.1. Individuals are represented in a problem-specific manner, meaning that for one problem the search might operate on vectors of floating point numbers, while another problem would require ordered lists or other structures which are suited to a specific domain. An initial population of N individuals is created and evaluated. Offspring are created by cloning each individual in the current generation. These offspring are mutated, then evaluated for fitness. From the N parents and $2N$ offspring, N are chosen to survive into the next generation, typically using a stochastic method based on fitness. This process of cloning, mutation, evaluation, and selection for survival continues for some specified number of generations. Other variations of EA follow a similar algorithm.

```
initialize population
evaluate population
for G generations
{
    for each parent
    {
        clone parent
        mutate cloned individual
        evaluate individual
    }

    select N individuals to survive
}
```

Figure 3.1: Pseudo-code describing the algorithm used in EP.

3.2 Evolution Strategies

Rechenberg [64] and Schwefel [71] are both credited with developing the approach known as evolution strategies. Several popular variations of ES exist, resulting in a common notation used to categorize different approaches: $(\mu/\rho+,\lambda)$. The μ represents the number of individuals in the population, ρ is the number of individuals who actually participate in procreation, and λ is the number of offspring which are created. A variant of ES is either $(\mu/\rho,\lambda)$-ES or $(\mu/\rho+\lambda)$-ES. In the case of the former, the μ most fit offspring are chosen to survive into the next generation. In the latter form, the parents are also eligible for survival, meaning that the μ most fit individuals from both the parents and the offspring survive [6]. Unlike EP, ES sometimes utilizes a recombination operator in addition to mutation. In this regard, ES is similar to a GA.

3.3 Genetic Algorithms

One of the most popular techniques inspired by biological evolution is the Genetic Algorithm (GA). This technique was invented in the 1970's at the University of Michigan by John Holland [27], though it is based on other, earlier forms of artificial evolution. A GA mimics biological evolution in order to search for solutions to a given problem. The canonical GA consists of a population of chromosomes, with each chromosome representing an individual within the population. The standard representation for the chromosome, or genotype of an individual, is a binary string; integer strings are sometimes used instead.

The process generally starts with the creation of a population of randomly gener-

ated individuals. At this point a generation loop of some sort begins, during which a series of steps are repeated to simulate reproduction within the population, as shown in Figure 3.2.

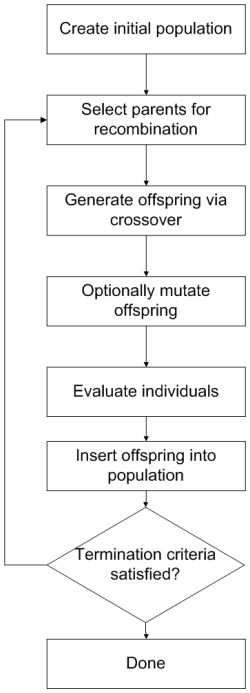


Figure 3.2: The operation of a traditional GA.

3.3.1 Evaluation

In each generation, the quality of each individual is evaluated by means of a fitness function. This function evaluates an individual and assigns some numeric value to it, representing the quality of the individual, based on how well it solves a given problem. This fitness value is used to compare individuals and to determine if a satisfactory solution has been found for the current problem. Fitness functions are problem-specific, and may serve as only an approximate measure of the true fitness of an individual. Depending on the problem, the objective may be to either maximize or minimize the value returned by a fitness function or to satisfy one or more constraints.

3.3.2 Selection

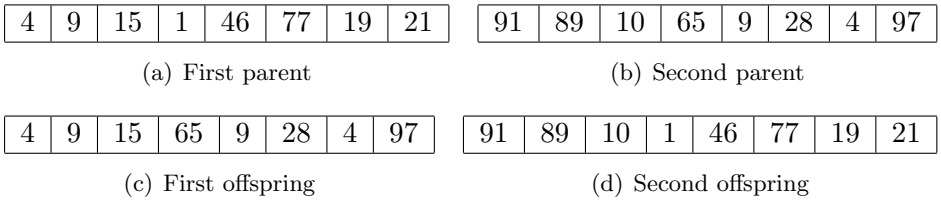
After the individuals have been evaluated to measure their fitness, a subset of the population is selected for use in recombination. The method used to select individuals

varies depending on the specific implementation. One common selection method is known as tournament selection. In tournament selection, K individuals are chosen at random from the population. From this group a single individual is chosen, often using some probabilistic method, with more fit individuals having a greater chance of being chosen than less fit ones. In other cases, the most fit individual from this subset is always selected. The tournament size, or number of members participating in the tournament, can be varied to adjust the selection pressure, with larger tournament sizes decreasing the probability of selection for low-fitness individuals [50]. A variation of selection created for this research usually selects the most fit competitor in a tournament, but incorporates a random tournament winner probability. This value, which can be adjusted for a given run, determines the probability w of a randomly chosen competitor winning the tournament rather than the most fit competitor. This variation serves as yet another means of adjusting the selection pressure of the search algorithm.

3.3.3 Recombination

Once a pair of individuals has been selected for reproduction, some crossover operation is used to combine their genomes, thereby generating offspring. The basic idea of crossover is to swap portions of the genomes of two individuals in order to create two new individuals, analogous to the process of sexual reproduction in nature in which two organisms mate, combining genetic material to create offspring. As with most aspects of genetic algorithms, there exist a number of variations of crossover. A basic one-point crossover consists of selecting a cut point within the genome and swapping all codons after the cut point between the two parents. Figures 3.3(a) and 3.3(b) show the genomes of two parents which were selected for use in crossover. Figures 3.3(c) and 3.3(d) show the results of a one-point crossover using these individuals as parents and with the cut point between the third and fourth codon. The first child's genome consists of the first three codons from the first parent, with the remaining five codons corresponding to the last five of the second parent. The other offspring is constructed using the remaining pieces of the first and second parent.

This technique is simple and ensures that the genotype retains a fixed length, a quality which is required for many problems to which GAs are applied. A two-point crossover involves selecting a beginning and an ending cut point. The codons that fall between these two points are swapped between parents. Another variation on genetic crossover is called uniform crossover. This scheme involves iterating through the individual codons in



(a) First parent (b) Second parent

(c) First offspring (d) Second offspring

Figure 3.3: Two offspring generated by a basic one-point crossover

a set of parents, swapping the codons in each position with a probability of p, commonly 0.5 [76]. Other techniques select different cut points in each parent, causing a different number of codons to be swapped between parents. This form of crossover is only appropriate when the problem allows for a variable length genome.

3.3.4 Mutation

After the new individuals have been created using some crossover operator, a mutation often takes place. A mutation consists of an alteration of one or more codons within a genome, as shown in Figure 3.4. As with crossover, a variety of mutation operations exist, but the basic premise remains the same. One standard mutation operator generates a random value for each codon and uses a probability of mutation to determine if that codon should be modified or not. This operator is used to maintain diversity within a population and allows new areas of the search space to be investigated and local minima to be escaped.

(a) Original genome (b) Mutated genome

Figure 3.4: A genome before and after mutation. The third codon is replaced with a new value.

3.3.5 Replacement

After the new individuals have been generated, their ability to solve the problem is evaluated using the fitness function. At this point, some replacement strategy is applied to merge the new individuals into the population. In some cases the entire population is replaced by newly-created individuals. In other cases, some subset of the population is replaced, often with more fit offspring replacing less fit individuals in the population. This process of selection, recombination, evaluation, and replacement continues until some termination criterion is met. Typically, these criteria include a maximum number of generations,

a target fitness value for an individual, or some indication that little or no further progress is likely.

One variant of GA, introduced by Gilbert Syswerda [76], is known as a steady state GA. The difference between the steady state GA and a traditional GA lies in the population model. As previously mentioned, each generation in a traditional GA sees most or all of the current population replaced by newly created individuals. Elitism is often employed, meaning that one or more of the most fit individuals in a population survive into the next generation. In the case of the steady state GA, only a single individual is added to the population at each step, giving a continuously updated pool of individuals. One member of the current population is replaced (sometimes conditionally) by the new individual. Various methods exist for determining the individual to be replaced; often the least fit individual in the population is replaced. Other approaches have been investigated, including random replacement, but Syswerda suggests that this approach performs poorly [77]. The remaining individuals remain in the population. At the next step, it's possible that the newest member of the population will be selected as a parent for the next offspring, allowing the algorithm to immediately take advantage of progress which has been made in the preceding generation. Replacement of an existing member of the population may be contingent on a number of things. Uniqueness of the genotype is one common condition, as research has shown improved performance in GAs, particularly ones using a steady state approach, when some mechanism is used to avoid introduction of duplicate genotypes[66, 19].

Some implementations of steady state GA make use of a form of mutation based local search. When an offspring is created, it is cloned some number of times, with the number N being referred to as the clutch size. Each individual in the clutch then undergoes mutation and evaluation, after which one individual is chosen to join the general population. The clutch size is typically small; a clutch size of 5 has been experimentally shown to be optimal for some problems [74]. Steady state GAs have been shown to outperform generational GAs on a number of problems, both in combination with GE [14] and without [78],[65].

3.3.6 Problems With GA

A basic genetic algorithm is fairly simple to implement and use, but it requires a problem-specific encoding to be applied to give meaning to the codons (usually bits) which constitute an individual. Depending on the problem being addressed, this encoding can be complicated, particularly if a variable-length genome is used. Certain problems which

consist of fixed-form expressions requiring only parameter optimization are particularly well-suited for a standard GA. Figure 3.5 shows a simple encoding which maps the codon values in the genome to coefficients of a fixed-form polynomial. The first codon (4) is the coefficient for the x^5 term, the next codon is the coefficient for the x^4 term, and so on. Other problems which require the form of an expression to be evolved, such as the search

4	9	1	7	15

$$4x^5 + 9x^4 + x^3 + 7x^2 + 15x$$

Figure 3.5: An example of a fixed encoding for a linear genome. The codon values are mapped to the coefficients of a polynomial.

for the quadratic formula, are difficult to map to a vector of integers, particularly one of fixed length. GE addresses this by treating the individual as the genotype from which a phenotype will be expressed via the grammar. This approach more closely matches the successful model of biological evolution on which the entire field of Evolutionary Computing is based.

A further benefit of GE is that the grammar serves as an abstraction between the final representation of the individual and the underlying search mechanism, allowing one to easily experiment with alternate approaches, such as Differential Evolution [63] and Particle Swarm Optimization [33]. In fact, GE has been used in combination with DE [12] and PSO [56] with satisfactory results.

3.4 Genetic Programming

3.4.1 Overview

Another EA variant which uses a tree-based structure rather than a linear genome is known as Genetic Programming. This form was popularized by John Koza [37, 36, 38] and is one of the most frequently used (and modified) types of EA. Rather than encoding an individual as a binary vector, the representation used in GP is a tree which, when evaluated in order, produces a complete program for solving a problem (see Figure 3.6). Traditional GP utilizes a form of crossover but no mutation; other variants may include a mutation operator as well. In terms of search, GP operates in much the way that a GA does, mimicking the population dynamics in living organisms.

As with a GA, an initial population is created by building trees, with non-leaf nodes

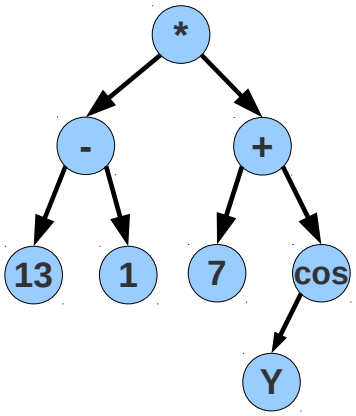


Figure 3.6: A tree-based representation of an individual used in GP.

chosen at random from a set of functions and leaf nodes chosen from a set of arguments. In the case of mathematical expressions, an individual would be represented by a tree with non-leaf nodes selected from a set such as $\{+, -, \div, \star, \cos\}$ and leaf nodes consisting of numbers and variables such as x and y. Multiple initialization algorithms exist for GP, many of which carefully control the depth of the trees to properly set the stage for successful search [46].

Some selection operator (often tournament selection) is used to choose two parents to which a crossover operator is applied. This operator swaps subtrees between the parents, creating two new individuals as shown in Figure 3.7. One of these offspring is inserted into the population; the other is typically discarded. In Koza's traditional GP, there is a certain probability that the crossover will be foregone and one of the parents will instead be copied directly into the next generation (cloning).

Traditional forms of GP are designed based on the requirement that all nodes return a value and all operators and functions are able to handle any input value which is returned from another node. This property is known as closure and it's important because GP operators which exhibit this property will not produce invalid individuals when applied to existing valid individuals. For example, the tree shown in Figure 3.6 is composed of mathematical operators $(+, -, \star, \div)$ as well as the function cos(). Each of these elements can accept floating point numbers as input and each returns floating point numbers as

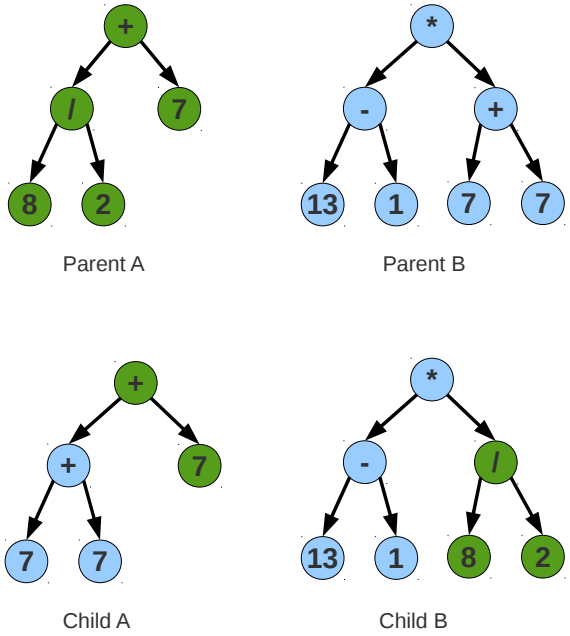


Figure 3.7: An illustration of basic GP crossover. Subtrees from each parent are swapped, creating two offspring.

output, allowing them to be composed and swapped freely. This constraint ensures closure for the crossover operator. If a mixture of data types and functions which handle and return a proper subset of valid data types are used, a sophisticated crossover operator is required, one which is type aware to ensure that type constraints are honored. One such variant of GP is Montana's Strongly-Typed Genetic Programming [51], or STGP. In STGP each variable and constant has an associated type (e.g. float, $N \times N$ matrix). Similarly, each function has associated types for its arguments and its return value. Such an approach increases the complexity of both the base code and the problem-specific definitions used when applying GP to a given problem. Alternatives which avoid such complexity are desirable.

3.4.2 Problems With GP

A number of problems were encountered during the initial exploration of genetic programming for finding function inverses. Several of these problems stem from the lack of constraints imposed on the creation of new individuals via GP crossover. GP crossover using variable length expressions can result in a condition known as bloat, in which indi-

viduals (usually represented as trees) grow continually larger as the evolution progresses. Bloat increases both runtime (due to increased computation needed to evaluate very large individuals) and memory usage (as more memory is required to store the large individuals). The increased computation and memory requirements slow the process and decrease the practical population size and number of generations. Some research has also shown that larger solutions tend to generalize less well than solutions with a more compact representation [67], [35]. GP implementations whose evaluation is based solely on the fitness of individuals are plagued by expression bloat [45], [75]. The simplest solution to address bloat is to set a maximum size for individuals, as in the case of Dynamic Maximum Tree Depth [72]. Individuals which exceed this limit can either be truncated such that they no longer exceed it or their fitness can be set to a very poor value to ensure that the individual is removed from the population through the usual operation of the algorithm. This doesn't address the problem so much as it tries to minimize the negative effects of it. The undesirable behavior which causes bloat still occurs, but is simply limited at some artificial threshold.

The cause of bloat in evolved expressions is usually due to the presence of introns, expressions whose inclusion or lack thereof has no effect on the fitness of an individual [75], [3]. Introns are introduced during recombination, persisting and spreading throughout the population as highly fit individuals containing them are chosen as parents for reproduction. Various approaches exist to deal with this, including ones such as ADATE [54] which tries search smaller solutions first. Other approaches attempt to evaluate individuals based on both their fitness and their size [47] . These approaches require a fine-tuned function which relates these two values to address the tradeoff between length and fitness. Approaches which employ adaptive parsimony pressure assign a penalty based on the sizes of the best individuals in recent generations. Multi-objective methods exist which attempt to treat size and fitness as separate objectives rather than addressing them with a single weighted function [62]. All of these approaches require additional work and tuning to balance the size and fitness values for individuals, but some have been shown to successfully improve the performance of GP by promoting the creation of smaller, more highly fit individuals.

During the initial phases of research in which a traditional GP approach was used, bloat was addressed first by setting a maximum length for individuals and later using simple parsimony to penalize individuals based on length. These approaches seem to have limited the size of individuals at the expense of fitness and diversity, a finding that is consistent with the results of other researchers. An approach which more naturally controls

the representation of individuals is desirable, hence the use of GE.

Chapter 4

Grammatical Evolution

Due to both the shortcomings of traditional GP and GA approaches and to the desirable features outlined below, the author selected Grammatical Evolution (GE) as the means by which a number of problems would be solved. GE is a technique devised by Connor Ryan, JJ Collins, and Michael O'Neill at the University of Limerick in Ireland. Research has shown GE to be effective on problems such as symbolic regression [60], credit ratings for bonds [11], stock market index trading [57], and edge detection for image processing [53].

GE can be thought of both as a variation on GP and as a layer of abstraction added to another search algorithm such as a GA. The notable features of GE are the use of a grammar to define and constrain the generated candidate solutions and the method of gene expression in which the genotype is mapped to a phenotype.

4.1 Grammars

A formal grammar is a means of specifying a language's syntax, consisting of a set of rules which govern how a string is created and a start symbol. Chomsky identified and described a hierarchy of languages and their associated grammars [13], including the context-free languages and grammars used in GE. Grammars are an important tool in the field of computing and are often used to describe the syntax of programming languages.

There exist four components which may be combined in a 4-tuple (N,T,P,S) to describe a given grammar. The components are a set of non-terminals (N), a set of terminals (T), a set of production rules (P), and a start symbol (S). Non-terminals are symbols (typically enclosed within angle brackets) which can refer to other non-terminals or to terminals within the grammar. Terminals are actual symbols which comprise the language

being described by the grammar and which cannot be changed into something else by a production rule. Production rules define the translation of one symbol (non-terminal) into another. A start symbol is a single non-terminal which serves as the first non-terminal to which production rules may be applied.

The standard notation used to describe a formal grammar is known as Backus-Naur Form (BNF) [7]. BNF consists of a series of production rules with non-terminals on the left side and either a terminal or another non-terminal on the right side, as shown in Figure 4.1. The start symbol is traditionally the first non-terminal shown.

```
<expr>    ::= <expr><op><expr>
          |   (<expr><op><expr>)
          |   <const>
          |   <var>

<op>      ::= +
          |   -
          |   *
          |   /

<var>     ::= X

<const>   ::= 1
          |   2
          |   3
          |   4
```

Figure 4.1: A sample grammar in Backus-Naur form.

The non-terminal <op>in the sample grammar can be replaced with one of four terminals, corresponding to the basic arithmetic operators for addition, subtraction, multiplication, and division. An example of this substitution is shown in Figure 4.3.

4.2 Genotype to Phenotype Mapping

GE is an evolutionary algorithm which separates the genotype and the phenotype, an approach that works more like the biological processes that EAs mimic than GA or GP. This separation has been shown to be beneficial in work by Banzhaf [10], in that the mapping process allows for fully investigating the genotype search space while still generating valid individuals. The translation from genotype to phenotype, which is analogous

to gene expression in biological systems, is performed by selecting production rules from a context free grammar. This grammar is represented in Backus Naur Form (BNF) and is used to define the phenotype of the individuals which will comprise to population. It is this phenotype which is evaluated using a fitness function, rather than the genotype.

The basic individual in GE is represented by its genotype, typically a vector of integers. Combined with a problem-specific grammar, each individual can be evaluated after undergoing a genotype to phenotype mapping. This process is performed by starting at the first codon, or integer, in the genotype and using this value to select a production rule from the grammar. Each non-terminal on the left hand side of the BNF representation can be expanded into one or more values (either a terminal or another non-terminal) on the right hand side. These expressions on the right hand side can be assigned numbers which are specific to the non-terminal on the left side to which they belong. Thus, for each non-terminal on the left there exist N rules, which can be assigned numbers 0 through N-1. These numbers are used along with the value of a given codon to build the phenotype of a given individual. The canonical GE algorithm selects a production rule by applying the modulo operation to the codon, yielding a number between 0 and N-1. This number selects the production rule to apply to the current non-terminal.

As with the genetic coding in living organisms, the encoding used in GE is degenerate, meaning that different sets of codons will map to the same phenotype, or a portion thereof. The effect of degeneracy in evolutionary algorithms has been studied by Weicker [79] and O'Neill and Ryan [55], the latter pair noting that the resulting frequency of neutral mutations acts as protection from destructive mutation events. In other words, degeneracy limits the impact of mutation since a mutation from one multiple of R to another multiple of R, where R is the number of production rules for a given non-terminal, has no effect on the phenotype. This benefit is similar to the protection from mutation afforded by introns, but without the associated increase in expression length and the corresponding growth in memory and computation resources.

Starting at the first codon in the genome, a production rule is selected to translate the start symbol into either another non-terminal or a terminal. At this point the next codon is used on the first remaining non-terminal, again selecting a production rule to apply. This process continues until either no non-terminals remain or some limit has been reached. What the limit is depends on whether codons are reused. GE utilizes a mechanism known as wrapping to allow a relatively compact genome to apply more production rules than there exist codons within it. Without wrapping, the process of expressing the genes

would end when the last codon had been used. If non-terminals remained, the expression would be invalid and there would be no usable phenotype. When wrapping is used, the process starts over at the first codon upon reaching the end of the genome. Wrapping typically occurs a limited number of times, preventing the generation of extremely long individuals and infinite expansion. Wrapping has been shown to improve the performance of GE on a number of problems [58].

```
A) <expr>    ::= <expr><op><expr>  (0)
          |    (<expr><op><expr>) (1)
          |    <const> (2)
          |    <var> (3)

B) <op>      ::= + (0)
          |    - (1)
          |    * (2)
          |    / (3)

C) <var>     ::= X (0)

D) <const>   ::= 1 (0)
          |    2 (1)
          |    3 (2)
          |    4 (3)
```

Figure 4.2: An annotated grammar showing production rules with their associated indices. The non-terminals are labeled A-D, and each production rule is shown with its numeric offset, from 0 to N-1, where N is the number of rules for a given non-terminal.

Figure 4.3(b) shows the full genotype to phenotype mapping using the grammar in Figure 4.2 and the genome in Figure 4.3(a). The process begins with the start symbol being mapped to <expr>. Beginning at the first codon in the genotype, a rule is selected to replace the first non-terminal in the expression with something else. The codon's value (4) maps to zero when taken modulo four, which is the number of rules for the non-terminal. The expression is expanded to <expr><op><expr>. The second codon is used to evaluate the first non-terminal in the next expression. This process repeats as shown, until the final expression $2 + (X * 4)$ is found. This represents the phenotype of the individual and is used in the fitness function to evaluate and rank the individual.

Although the abstraction of the phenotype from the genotype can be useful, it also imposes some limitations on the flexibility of the user. Hybrid approaches, which are

4	10	1	8	5	3	17	2	6	3

(a) A sample genome consisting of integer codons.

Codon	Rule	Replacement	Expression
4	0	<*expr*> ⇒ <*expr*><*op*><*expr*>	<expr><op><expr>
10	2	<*expr*> ⇒ <*const*>	<const><op><expr>
1	1	<*const*> ⇒ 2	2 <op><expr>
8	0	<*op*> ⇒ +	2 + <expr>
5	1	<*expr*> ⇒ (<*expr*><*op*><*expr*>)	2 + (<expr><op><expr>)
3	3	<*expr*> ⇒ <*var*>	2 + (<var><op><expr>)
17	0	<*var*> ⇒ *X*	2 + (X <op><expr>)
2	2	<*op*> ⇒ ∗	2 + (X * <expr>)
6	2	<*expr*> ⇒ <*const*>	2 + (X * <const>)
3	3	<*const*> ⇒ 4	2 + (X * 4)

(b) An example of genotype to phenotype mapping.

Figure 4.3: An example of genotype to phenotype mapping showing each non-terminal being translated using the grammar in Figure 4.2 and the genome in (a).

common in GA and GP applications, are difficult to implement using GE because operations on the phenotype cannot be easily carried back to the genotype, making it difficult to return information to the population. For example, it's common to use a form of local search as a mutation operator, allowing a given form to be retained while parameters or constants are optimized. Any local search which operates on the phenotype in a GA or GP is not easily translated for use with GE. A more general local search can be performed by making changes to the genotype and then performing the mapping to phenotype followed by evaluation, but this search cannot focus on particular structural components of the phenotype as one would with GP.

4.3 GA Operators In the Context of GE

Although the search algorithm used in GE is typically a standard GA, the application of standard crossover and mutation operators on the genotype causes very different results on the generated phenotypes. For example, if the first codon in Figure 4.3(a) was changed from 4 to 3, a mapping using the grammar in Figure 4.2 would result in the expression X, rather than the expression $2 + (X * 4)$, the creation of which is described in Figure 4.3(b). Thus, a more careful look at the genetic operators is warranted.

4.3.1 Crossover in GE

The nature of the mapping from genotype to phenotype in GE means that a given codon or series of codons in the genotype has no intrinsic meaning, and is interpreted only in the context of a complete mapping. Thus, a group of codons which coded for a given string in a parent could very well code for an entirely different string when swapped into a child. Keijzer et al. [32] refer to this property as intrinsic polymorphism and go on to label the basic one point crossover *ripple crossover*, as a change in one position in the genotype goes on to impact the expression of all the remaining codons. The phenotype resulting from the creation of a new individual using ripple crossover may bear little resemblance to either of the parents, depending on the grammar, cut point, and the specific codons of which it is composed. Still, Ryan, Majeed, and Azad found that GE creates and uses building blocks [68] (short collections of genes which are combined to create fit individuals) in much the same way that Holland and others have observed in traditional GAs [27].

As there exist numerous variations of crossover, the question of choosing one arises. Banzhaf, Conrads, Francone, and Nordin developed a variant known as homologous crossover which seeks to more closely model the recombination which occurs in organisms [22]. Specifically, this approach attempts to align the crossover points, allowing a segment from one parent to be swapped only with a segment in the same position in the other parent. This is analogous to the way that biological crossover occurs, with DNA from both parents being structurally similar and aligning on the same boundaries. O'Neill and Ryan investigated crossover operators and compared the basic one point crossover with a number of alternatives [59]. O'Neill and Ryan's research suggested that the basic one point crossover used in GE provided some of the benefits of a more complicated homologous crossover while demonstrating more consistent performance than other variants. For these reasons, a one point crossover operator was used for this research.

4.3.2 Mutation in GE

The mutation operator in a GA modifies the value of a single codon; in the case of a binary genotype, this amounts to flipping a bit. For an integer genome, like the one used with GE, the value of a given codon is replaced with a randomly chosen value in the allowed range. The result of mutation on the corresponding phenotype can be much more profound than the change in the genotype, as a different production rule being selected in position J of the genome can potentially change the subsequent production rules as well, due to the context specific nature of GE gene expression. Mutation in GE might seem to be have a

great destructive potential, particularly when the mutation occurs near the beginning of the genome. Research by Ryan and O'Sullivan [61], as well as subsequent work by Ryan et al. [68], found that the omission of mutation had a detrimental effect on the performance of GE.

4.3.3 Conditional Replacement

As discussed in Section 3.3.5, GAs commonly incorporate a de-duplication method to maintain diversity within the population. The steady state GA used in this research makes replacement of an existing individual conditional on the newly created offspring being more highly fit than an existing individual within the population. Additionally, the new individual will only join the population if an individual with an identical genotype does not already exist. If the new individual's genotype is found to exist in the population, a randomly initialized individual is created using the same approach by which the initial population was created. This new individual is inserted into the population instead of the duplicate.

A number of possible definitions of duplication exist for this implementation of GE. Clearly, identical genotypes imply identical phenotypes and therefore identical fitness. However, given the degenerate decoding used in GE, it's possible that two very different genotypes can give rise to identical phenotypes. A more sophisticated de-duplication operation might therefore compare the phenotypes of individuals which have identical fitness values to see if the phenotypes match. It's not clear that this modification would be beneficial, as one genotype may contribute to more highly fit offspring than another genotype which yields the same phenotype. In the case of mathematical expressions such as the targets of symbolic regression and the search for Lyapunov functions, individuals might have different genotypes and different phenotypes and yet be equivalent expressions which would have the same fitness for any set of inputs. Consider that the expressions $3x+9$ and $3(x+3)$ take different forms but are equivalent expressions. De-duplication to this level is neither practical nor desirable.

Chapter 5

Symbolic Regression and System Identification

5.1 Problem Overview

A common problem in mathematics, controls, and various other fields is finding a model to describe a collection of data. Certain systems (e.g. mechanical systems, electronics) and processes (e.g. behavior of financial markets) can be observed and measured, but not easily modeled. Numerical regression is one common approach to modeling such a system, but this depends on knowing a priori what form the solution will take. Given a functional form, parameters are optimized to match the available data. Such numerical methods can be useful but may be impractical due to high computational cost or due to an inability to select a proper model to use as the base for optimization. A more flexible approach involves evolving both the structure and the parameters of the target expression.

Given that symbolic regression is a well-known application of evolutionary algorithms, and given previous work in symbolic regression using GE, this problem may seem neither interesting nor novel. Symbolic regression was used as a means of testing the usefulness of the GE implementation, to verify previously published results, and as a bridge between sampled data and the work in finding Lyapunov functions, described in section 6.1. Given a small number of sampled data points, this approach can be used to discover a pair of equations which describe the system being observed, as show in section 5.3. This system of equations can in turn be used, as shown in section 6.2, to evolve Lyapunov functions which define the behavior of the system in question. Taken together, these approaches can provide new information about an observed system.

5.2 GE Approach to Symbolic Regression

5.2.1 Grammar Tuning

Grammar tuning is an important part of solving problems using GE. Small changes can have large effects on the performance of the search. Consider two similar grammars, differing only in their treatment of exponents. In the first grammar, exponentiation is allowed as an operation on an expression by being included in the set of binary operators. In the second grammar, exponentiation is treated separately and is only allowed as an operation on a variable. In the first case, expressions can be nested, allowing the creation of nested exponents, e.g. $(X^4)^{23X}$, something that is likely undesirable for many applications due to the likelihood of overflow in floating point computations. In some cases a user will know that expressions of this form are outside the desired search space for a given problem. The latter restricts exponentiation such that the expressions evaluate to values which are reasonable for the given problem. In addition to the choice of grammar, there exist a number of other tunable parameters which can greatly affect the performance of the search.

5.2.2 General GA parameters and Settings

A basic GA alone has a number of settings which can be altered to change the behavior of the search. The generational model, mutation rate, crossover rate, population size, number of generations, and selection method can all be manipulated in ways that change the performance characteristics of a given search.

The crossover rate (C) defines the probability that a new individual will be created via recombination of the genomes of two parent individuals. Typically there exist one or more alternatives to crossover which are used with probability $1.0 - C$. For this implementation, the alternative varies depending on whether a generational or steady state approach is used. In the case of the generational approach, the most fit individuals from the current generation are allowed to survive into the next generation. For steady state search, the alternate method is to create one randomly, in the same way that the initial population is created. This approach is used to maintain diversity in the population, though the new individuals are unlikely to have high fitness. Crossover rates tend to be high, and most runs utilized a crossover rate of 0.9. Section 5.5.1 provides a justification for the choice for crossover rates.

As discussed in Section 4.3.2, mutation is an important part of the search algorithm, serving to maintain diversity and to explore more of the search space than is possible

with crossover alone. In the context of this research, the mutation rate refers to the probability of a given codon being replaced with a randomly generated integer. Each codon in a newly created genome is subject to mutation; high mutation rates tend to cause multiple codon alterations in a given individual. The fact that the encoding is degenerate (discussed in Section 4.2) means that GE is more resilient to mutations; many mutations in a given codon may not change the resulting phenotype. As such, mutation rates vary between 0.05 and 0.1 for most runs.

Another choice to be made relates to the structure of the genome and the type of crossover operator used. A fixed-length genome, common in GAs, sets a firm limit on the size of the corresponding phenotypes which are created with GE, while at the same time limiting the search space of genotypes. Further, a fixed-length genome requires a crossover operator that doesn't change the length of the genomes. A variable-length genome, on the other hand, allows for a greater degree of exploration and discovery and allows a larger variety of crossover operators. Since GE does not require genomes to have a fixed length, the initial population is created with lengths randomly selected between 20 and 40, and the single-point crossover operator allows offspring to be created with different lengths than either parent. This approach is consistent with O'Neill and Ryan's work on GE [60].

5.3 GE Approach to Finding Vector Fields

The approach to evolving the system of equations corresponding to a given vector field can be considered as a pair of symbolic regression problems. Given a finite number of samples from a vector field $\{(x, y), <u, v>\}$, the goal is to find the corresponding functions $F(x, y)$ and $G(x, y)$ such that $u = F(x, y)$ and $v = G(x, y)$ for all samples from the vector field. First, one considers the regression problem of finding the function $F(x, y)$ for which $F(x_n, y_n) = u_n$ for all samples. Then, using the same (x, y) pairs, a second symbolic regression problem is tackled to find $G(x, y)$ which produces the v component of the sample vector. Taken together, these evolved expressions describe the sampled system.

The vector field problem was approached by decomposing it into two symbolic regression problems. A fixed grid of (x, y) values is used to evaluate candidate expressions with the intention of finding one which matches the target expression $F(x, y) = u$ over that set of (x, y) values. A second regression run is then performed using the same (x, y) values, with the target being $G(x, y) = v$.

In order to simulate sampling error in real-world data, Gaussian noise was added to some regression runs to evaluate the performance of this approach under more realistic

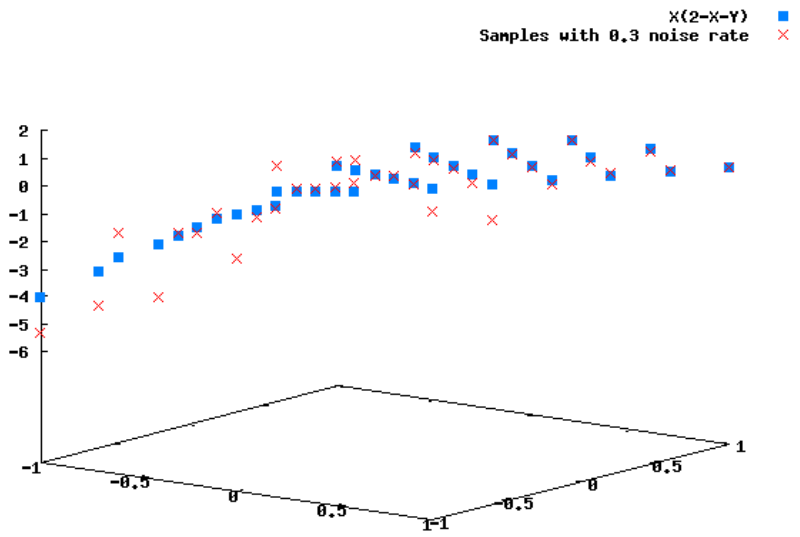


Figure 5.1: A graph showing sample points on the interval [-1:1] for the function $x(2-x-y)$ and the same points after being perturbed with Gaussian noise with a noise rate of 0.3.

conditions. A modified Box-Muller transform was used to generate random, normally-distributed values by which the samples were perturbed. The independent, normally distributed variables z_1, z_2 are generated using a pair of uniformly distributed random numbers U_1, U_2 [23]:

$$z_1 = \sqrt{-2\ln(U_1)} * \sigma * \cos(2\pi U_2)$$

$$z_2 = \sqrt{-2\ln(U_1)} * \sigma * \sin(2\pi U_2)$$

The distribution in this case is defined with a mean $\mu = 0$ and a standard deviation $\sigma = v_i * q$ given by the product of the sample value and the noise rate. For each sample point, either z_1 or z_2 is used, with the choice made randomly. Figure 5.1 shows a graph of these samples and the perturbed values used in the regression run.

5.4 Inverse Regression

By modifying the fitness function used for symbolic regression, it's possible to instead search for the inverse of a given function. Given a function $F(x)$, the inverse of F is given by $G(y)$ where $G(F(x)) = F(G(x)) = x, \forall x \in D$. For example, if $F(x) = 2x + 8$,

Setting	Value
Genome length	Variable
Generational model	Steady state
Number of samples	36
Sample delta	0.4
Sample interval	-1 to 1
Noise rate	0.0

Table 5.1: Settings for regression runs.

then $G(y) = (y - 8)/2$ is the inverse of $F(x)$, $F^{-1}(x)$. In order to evaluate the fitness of an individual, the fitness function is given by

$$\sum_{i=0}^{s} |x_n - G(F(x_n))| + |x_n - F(G(x_n))|$$

where s is the number of sample points. With the exception of the different fitness function, the approach to finding inverses is identical to the basic symbolic regression; even the same grammars are used for many runs. Results for inverse regression are covered in Section 5.5.4.

5.5 Results

In addition to testing the ability of GE to perform symbolic regression on a series of expressions, additional tests were performed to understand the effect of parameter changes and to discover general guidelines for settings to achieve the best performance. Additionally, tests were performed to measure the ability of this GE approach to operate in the presence of noise in the input data.

5.5.1 Parameter Survey

A common target expression used in symbolic regression work is the quartic polynomial, $x^4+x^3+x^2+x$ [44, 60, 37]. Population size (n), number of generations (g), crossover rate (c), mutation rate (m), random tournament winner rate (w), and grammar were varied. An initial survey was performed to investigate the effect of varying the crossover and mutation rates along with the grammar and generational model.

Crossover rates of 0.7, 0.8, and 0.9 were tested along with mutation rates of 0.05, 0.08, and 0.12. Two different grammars were used, shown in Figures 5.2 and 5.5. Table 5.2 shows the fixed parameters used in the first stage of the survey.

The performance of a given configuration was evaluated both in terms of the number of successful runs (i.e. an exact or equivalent expression was found) and the mean

Setting	Value
N	200
G	200
Target	$x^4 + x^3 + x^2 + x$
W	0.02

Table 5.2: Fixed settings used to investigate crossover and mutation rates

```
<expr> ::= (<expr> <op> <expr>)
         | (<expr> <op> <var>)
         | <var>
         | <fun>(<var>)
         | (<var> <op> <expr>)
<op>   ::= +
         | -
         | /
         | *
<fun>  ::= sin
         | cos
<var>  ::= X
```

Figure 5.2: Reg5.bnf: A simple grammar used for symbolic regression.

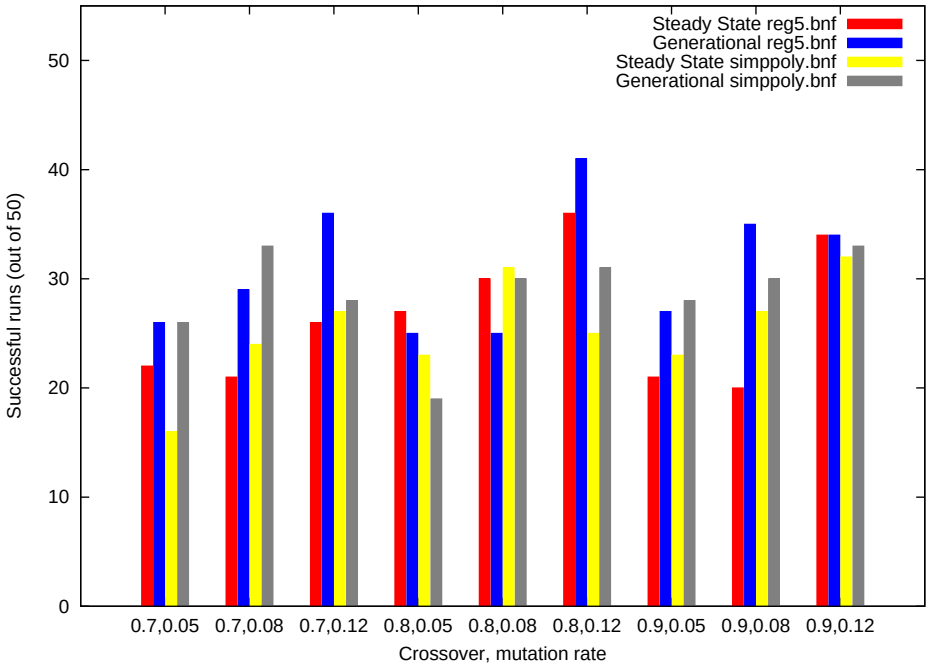


Figure 5.3: A chart showing the number of successful runs using various settings for crossover and mutation rates.

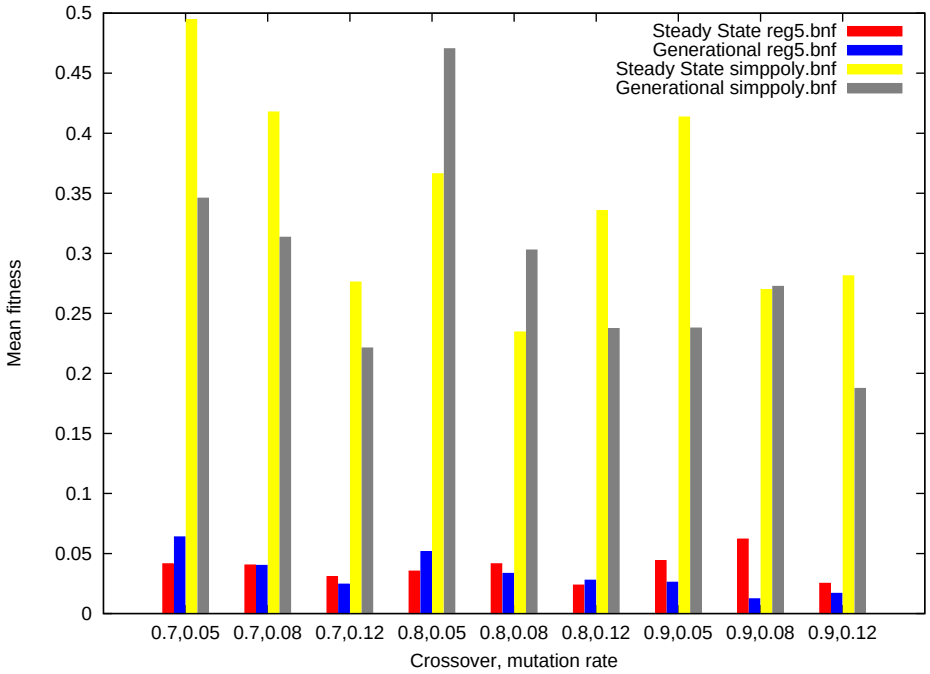


Figure 5.4: A chart showing the mean fitness over a set of runs using various settings for crossover and mutation rates.

fitness over the set of runs. In some cases, these two metrics show very different pictures, an issue also noted by Luke and Panait [48]. Figure 5.3 shows the number of successful runs at each configuration. The clearest trend in this chart is the relationship between higher mutation rates and greater success. At each crossover rate, improvement is seen as the mutation rate increases, regardless of grammar or generational model. Changes in other settings show less obvious changes in the success rate. In most cases the generational approach outperforms the steady state approach, a surprising result given previous findings on other problems and published results from other researchers. The grammar in Figure 5.2 usually outperformed the one shown in Figure 5.5. The picture is quite different when mean fitness is used as the metric for evaluating settings.

Figure 5.4 shows the mean fitness at each configuration, with lower values indicating higher fitness and therefore better performance. The mean fitness chart looks nothing like the success rate chart. The difference between the two grammars is startling, with the mean fitness of the simppoly grammar being four to ten times as high as the fitness for runs using reg5. Though the success rates were similar, the solutions which were not successful were far less fit with simppoly.bnf. No clear trend was exhibited regarding the generational model; each approach outperformed the other in several configurations. Again we do see

```
<expr> ::= <expr><op><expr>
         | (<expr><op><expr>)
         | <const>
         | <var>
<op>   ::= +
         | -
         | *
<var>  ::= X
<const> ::= 1
         | 2
         | 3
         | 4
```

Figure 5.5: Simppoly.bnf: A grammar for polynomial expressions used in symbolic regression.

an improvement in the fitness of solutions with higher mutation rates, confirming the belief that other aspects of the algorithm created a tendency towards stagnation, requiring higher mutation rates to maintain diversity within the population.

Figure 5.7 shows the mean fitness with a variety of settings for w when using a generational approach. Moderate values for w resulted in the best (lowest) mean fitness, with the best configuration being $c = 0.9, m = 0.08$, and $w = 0.05$. Runs with these crossover and mutation rates also did very well using $w = 0.02$ and $w = 0.1$. Figure 5.6 shows the number of successful runs with a variety of settings for w when using a generational approach. It appears that a moderate value for w works best, with 0.05 and 0.1 having the highest success rate.

The steady state approach is affected differently by changes in w. Figure 5.8 shows the mean fitness with a variety of settings for w when using a steady state approach. Although the generational runs with $w = 0.05$ resulted in very low mean fitness (recall that lower is better) , the steady state runs with this setting yielded the worst fitness values. Figure 5.9 shows the number of successful runs with a variety of settings for w when using a steady state approach. For steady state approaches, the most runs are successful when the random tournament winner function is disabled entirely ($w = 0$).

5.5.2 Performance Characteristics With Added Noise

Experiments were done to discover the ability of this approach to operate on noisy data, as other researchers have shown the ability of various GP implementations to operate effectively in the presence of noise [16, 18, 31]. As mentioned in Section 5.3, Gaussian noise was added to the sampled data, with a zero mean and the standard deviation given by the

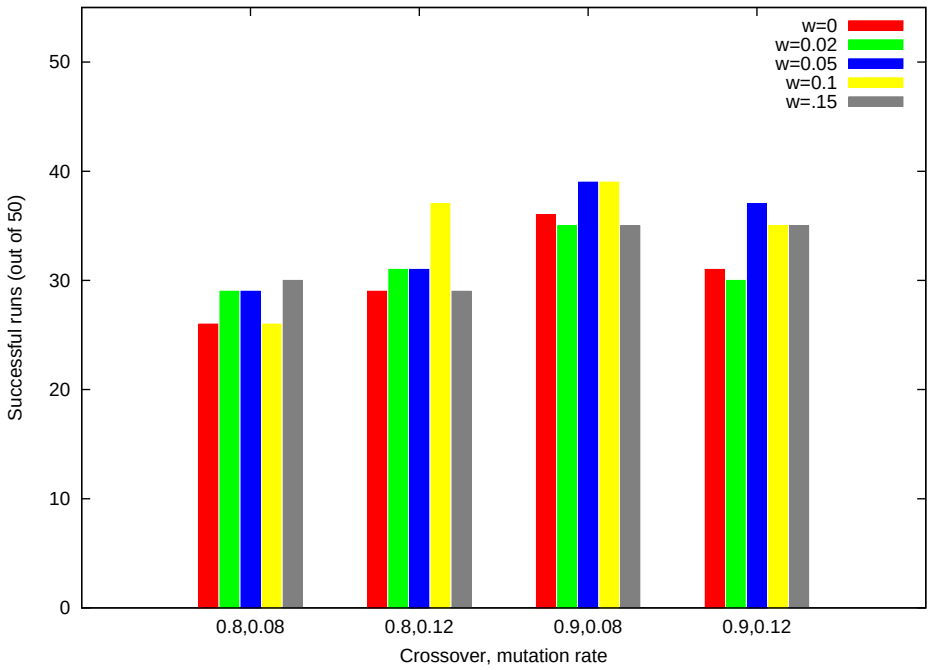


Figure 5.6: A chart showing the number of successful runs with various random tournament winner rates with a generational approach.

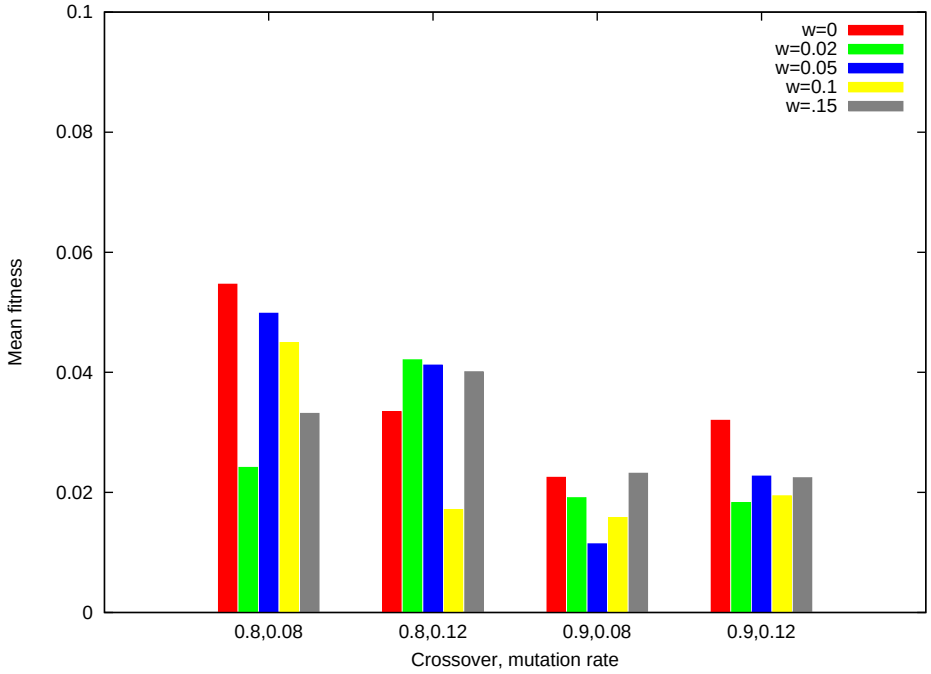


Figure 5.7: A chart showing the mean fitness with various random tournament winner rates with a generational approach.

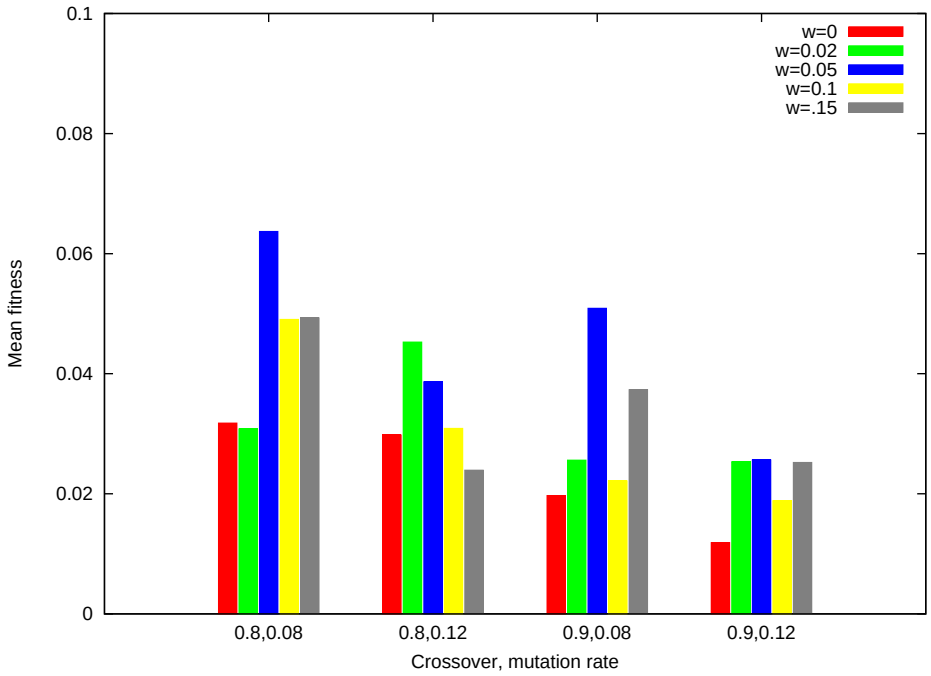


Figure 5.8: A chart showing the mean fitness with various random tournament winner rates with a steady state approach.

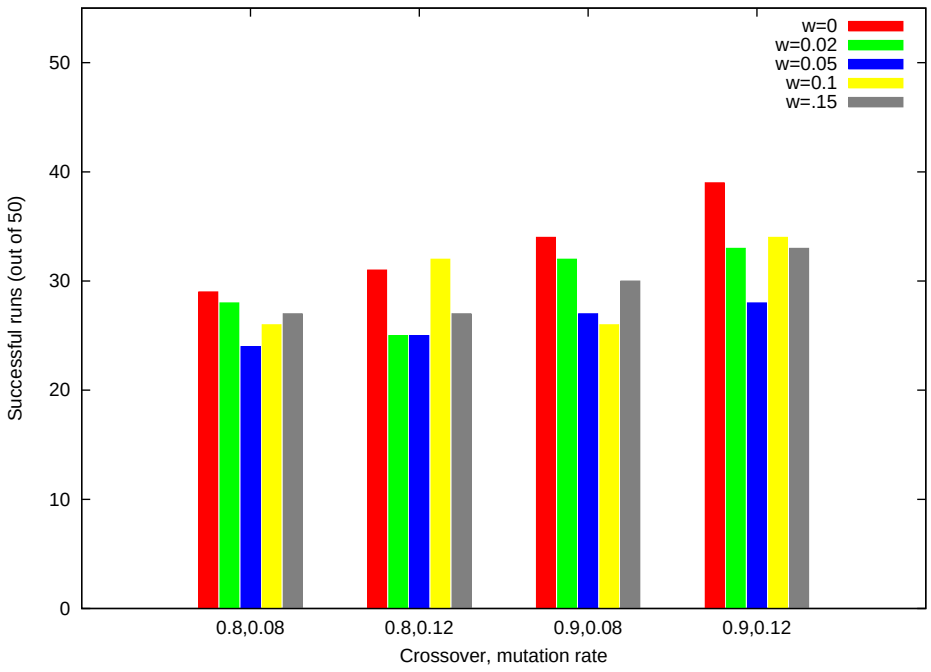


Figure 5.9: A chart showing the number of successful runs with various random tournament winner rates with a steady state approach.

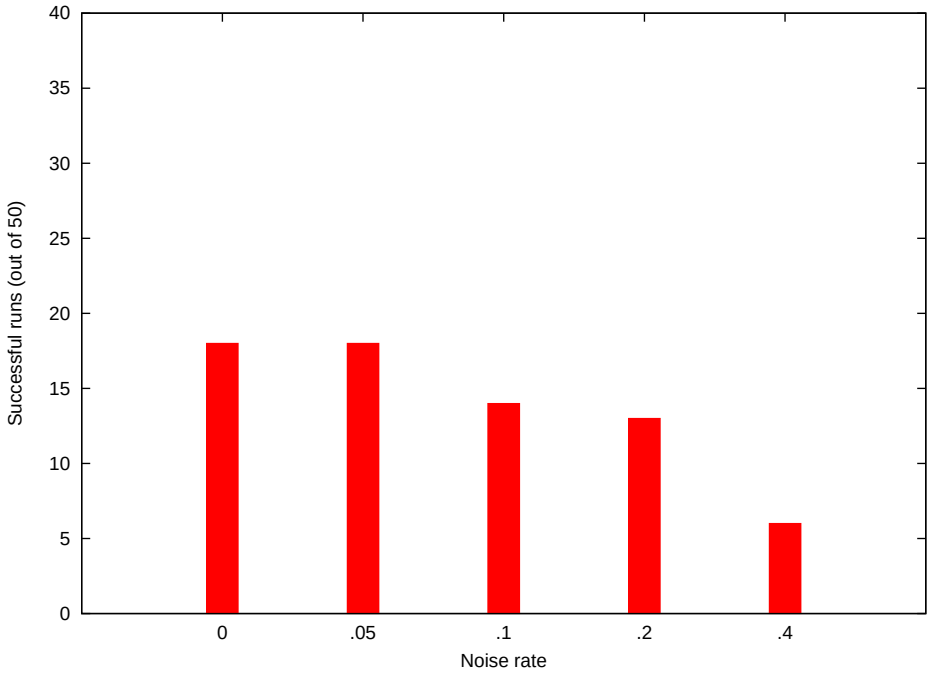


Figure 5.10: A chart showing the number of successful runs with differing amounts of Gaussian noise added to the input data.

product of each sample value and an adjustable noise rate q with $0 < q < 1.0$. Since the actual target expression is known, it was possible to calculate an adjusted fitness using the actual sample points rather than the ones to which noise was added. As shown in Figures 5.10 and 5.11, an increasing amount of noise did have a negative effect on the number of perfect solutions and the mean fitness values, although the difference is negligible for a noise rate of 0.05. It's also worth noting that when comparing the noiseless runs with the high noise runs (.4 noise rate), the mean adjusted fitness increased by a factor of only 2.4 while the raw fitness increased by a factor of 5.1 and the number of perfect solutions found without noise is three times the number found with the highest added noise. This suggests that this approach does not overfit the noisy data and instead more closely matches the underlying function being sought. A detailed listing of these results is shown in table B.5.

5.5.3 Additional Regression Results

Regression on the expression $-0.5y(1.0 - x^2 + .1x^4) - x$ proved to be more challenging than some simpler expressions. Perfect solutions were not found, but quite a few solutions had excellent fitness. The expression $(y^4/2(x^2 - y^4)/y^1) - ((x^1 - y^9) + y^3)$ was found with a raw fitness of 0.104. For 30 iterations a mean fitness of 1.25 resulted. A

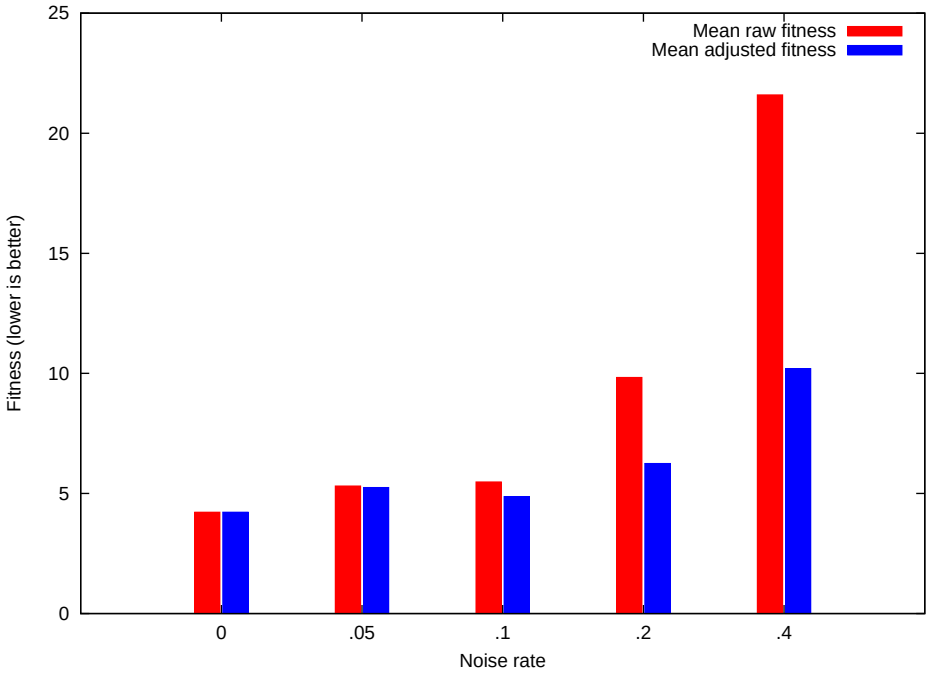


Figure 5.11: A chart showing the mean fitness for runs with differing amounts of Gaussian noise added to the input data.

matching run using a generational approach, rather than steady state, also resulted in a mean fitness of 1.25. These results were generated using the grammar shown in Figure A.1. This function was included because it, along with the trivial to evolve function $F(x, y) = y$ were an example system used by Grosman and Lewin in a study using GP to find Lyapunov functions [26].

Regression was also performed on the vector field data for the system described in Section 6.4.3. The $F(x, y) = y$ component is of course trivial to evolve, so the runs of interest are for the $G(x, y) = -\sin(x) - y$ expression. Using the grammar shown in Figure A.2, a 27/50 success rate was achieved using a steady state approach. If the definition of success is altered from requiring an exact solution to requiring the raw fitness to be less than 0.01, then the success rate becomes 32/50. The mean raw fitness over the set of 50 runs was 0.1, demonstrating that nearly all solutions were highly fit. A wide variety of functional forms were discovered, including the following list of expressions which are equivalent to

the target expression of $G(x, y) = -\sin(x) - y$:

$$\begin{aligned} G(x,y) &= (x - \sin(x) - x - y), \\ G(x,y) &= ((x - y) * 1.0 - x) - \sin(x), \\ G(x,y) &= (\sin(((x - x) - x)) - y) - \sin((y - y)), \\ G(x,y) &= (\sin(x * 1.0 - (2.0 * x)) - y), \\ G(x,y) &= (\sin((\sin(y - y) - x)) - y). \end{aligned}$$

A notable expression that's very nearly equivalent to the target is $(\sin((\cos(4.0 - \sin(1.0)) * x)) - y)$. The subexpression $\cos(4.0 - \sin(1.0))$ very closely approximates the -1 term needed on the $\sin(x)$ part of the expression. The ability to create needed constants is useful when such constants aren't part of the language used to build expressions. The grammar used for this problem contained neither negative numbers nor the unary negation operator.

5.5.4 Inverse regression

A number of target expressions were used in searching for inverses. The function $F(x) = 1/(x + 1)$ was used as an initial test of this variation on regression. Using the basic grammar shown in Figure 4.1, F^{-1} was easily found. The evolved expressions varied a great deal in their initial form, despite simplifying to the same function: $G(y) = 1/y - 1$. One evolved inverse was $(y/y/y) - 1$; a much longer equivalent expression was $(((y - y/y)/(4 - (1 + y))/y * ((y/3) - 1)) + ((y - y/y)/(4 - (1 + y))/y * ((y/3) - 1)) + ((y - y/y)/(4 - (1 + y))/y * ((y/3) - 1)))$. Similarly, a variety of different but equivalent inverses were found for the function $F(x) = \sqrt{x + 1}$, including

$$\begin{aligned} G(y) &= y + (y * y) - y/1/1/y - y, \\ G(y) &= ((y * y) - 2 + 1), \\ G(y) &= ((1 * y) * y - 1). \end{aligned}$$

These expressions all simplify to $y^2 - 1$. The behavior and performance of the inverse search appears to be nearly the same as that of the symbolic regression search.

Chapter 6

Lyapunov Functions

The field of control theory deals with influencing and modeling the behavior of dynamical systems, with applications in electronics, physics, economics, aeronautics, and a variety of other fields [5]. In many cases it is necessary to gain an understanding of the dynamics of a given system; numerous approaches exist to aid in this understanding. One method of gaining understanding about such is system is to discover something known as a Lyapunov function.

6.1 Problem Overview

One of the basic questions faced when analyzing a nonlinear dynamical system is concerned with stability at a rest point of the system. Put simply, a stable rest point is one for which all solutions of the system that start near a rest point R stay near R. A related property, asymptotic stability, means that all solutions that start near R converge to R [5]. Thus, any method of showing stability for a given dynamical system provides a great deal of information about the behavior of that system.

There is no direct method for finding a Lyapunov function, but a given function can be shown to be a Lyapunov function by verifying that it meets a series of well-defined conditions. For a given system there may be 0, 1, or many Lyapunov functions. For the purpose of this research the systems are limited to two dependent variables, with systems of the form

$$\frac{dx}{dt} = F(x, y), \quad \frac{dy}{dt} = G(x, y).$$

The functions F and G are assumed to be continuously differentiable in an open set $U \subset \mathbb{R}^2$. If there is a point (a, b) in U such that $F(a, b) = 0$ and $G(a, b) = 0$, then (a, b) is a rest point for this system and it is at this point that we wish to investigate the stability properties.

Consider the conditions shown in Figure 6.1 with $\dot{L}(x,y) = \frac{\partial L}{\partial x} F(x,y) + \frac{\partial L}{\partial y} G(x,y)$. A

$$
\begin{aligned}
&(i) \quad L(a,b) = 0 \\
&(ii) \quad L(x,y) > 0,\ \forall (x,y) \in U \setminus \{(a,b)\}, \\
&(iii) \quad \dot{L}(x,y) \leq 0,\ \forall (x,y) \in U \setminus \{(a,b)\},
\end{aligned}
$$

Figure 6.1: Conditions for a Lyapunov function for a system of two dependent variables.

candidate Lyapunov function is a function $L(x,y) : \mathbb{R}^2 \rightarrow \mathbb{R}$ which is locally positive-definite, i.e. one which satisfies conditions *i* and *ii*. If a candidate $L(x,y)$ is found whose time derivative is negative semidefinite, i.e. one which satisfies condition *iii*, then the rest point (a,b) is stable. If $\dot{L}(x,y) < 0$ then the rest point is asymptotically stable and solutions which start near this rest point will eventually converge to it.

A number of traditional approaches exist which attempt to find valid Lyapunov functions without the use of an evolutionary approach. A somewhat naive approach is to attempt to evaluate the energy in the system [70, 49]. This approach does work in certain situations but often fails to find usable candidates. Another approach which has shown some success uses a sum of squares decomposition [1, 69]. This approach is limited to systems and candidates which are polynomials.

Since a function can be evaluated by testing whether or not it satisfies these three conditions, a fitness function can be created and an EA approach can be used to search for Lyapunov functions. The idea of evolving candidate Lyapunov functions is not novel; GP has been applied to this problem by a number of researchers [26, 25, 8, 52]. Other research was done using a GA with fixed functional forms [73]. The goal of this research was to improve on previous results by taking advantage of GE's highly tunable nature and resistance to bloat.

6.2 GE Approach to Finding Lyapunov Functions

In applying GE to the problem of finding Lyapunov functions, the process consisted of selecting a system of equations $F(x,y), G(x,y)$ along with a rest point for the system and attempting to evolve a function which satisfies the previously mentioned conditions for a certain region surrounding the rest point. A regular grid of P points surrounding the rest point (a,b) is generated (see Figure 6.2) and it is at each of these points where the fitness function is applied to each candidate to evaluate its fitness as a possible Lyapunov function for the system in question. The size and range of the grid depend on the system of

equations and specific rest point being examined, as other rest points in a system may also act as attractors and should therefore not be contained in the set U for the selected rest point. The evaluation consists of verifying that the three conditions outlined in Figure 6.1

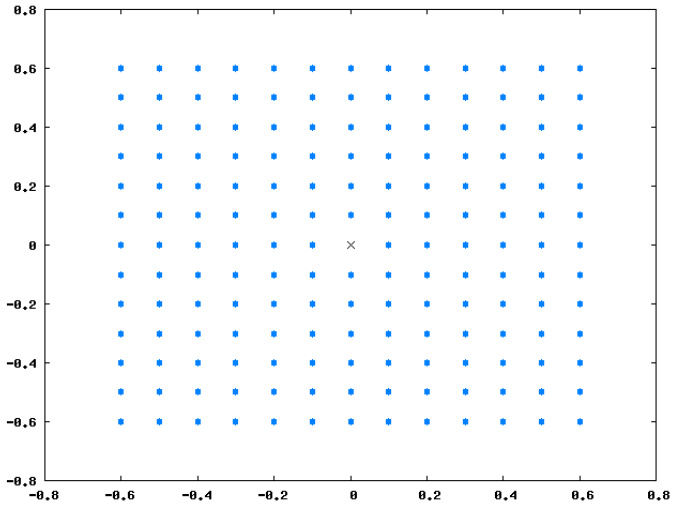


Figure 6.2: A grid of sample points in the region surrounding the rest point, which is marked with an X.

hold over the sample points in U. Penalties are assessed (as shown in Figure 6.3) when any of these conditions fail to hold.

Condition	Penalty	Value
Undefined operation	Q_{eval}	300.0
$L(a,b) \neq 0$	Q_1	100.0
$L(x,y) \leq 0$	Q_2	1.0
$\dot{L}(x,y) > 0$	Q_3	1.0
$\dot{L}(x,y) = 0$	Q_4	0.33

Figure 6.3: Penalties which are applied to adjust the fitness of a candidate Lyapunov function for failure to satisfy required conditions.

The first step is a single evaluation of the function at (a,b) and comparison of the result to 0. This step, verifying that the function has an equilibrium point at the rest point, contains an optimization that takes advantage of the fact that a function which fails to satisfy this condition can be shifted so that it does satisfy it. If a candidate fails to satisfy the first condition i.e. $L(a,b) = Z$ where $Z \neq 0$, then we can instead use $L - Z$ as our candidate since it will satisfy this condition. For example, given a candidate $L(x,y) = x^2 + y^2 + 5$ and a rest point $(a,b) = (0,0)$, we see that this candidate fails condition i, with $L(0,0) = 5$. With

the optimization enabled, we would shift L to create $L_{new} = x^2 + y^2$ which does evaluate to 0 at the rest point. L_{new} would then be used in lieu of L for the remainder of the evaluation process. This optimization was made optional, and runs were done both with and without this modification. This setting is discussed in more detail in Section 6.4.2.

The second step checks the candidate at each sample point in U, the neighborhood surrounding the rest point in which we are interested. A fixed penalty Q_2 is applied to the fitness of a candidate for each sample point at which the second condition is not satisfied. The third step again checks the candidate at each of the sample points, using a three point central finite difference approximation for the partial derivatives:

$$\frac{\partial L}{\partial x} \approx \frac{L(x + \Delta x, y) - L(x - \Delta x, y)}{2\Delta x},$$
$$\frac{\partial L}{\partial y} \approx \frac{L(x, y + \Delta y) - L(x, y - \Delta y)}{2\Delta y}.$$

Given the limitations of floating point math in this context and the imprecise nature of the problems being addressed, the values $\Delta x = \Delta y = 10^{-7}$ were found to perform adequately. In this third step, two separate penalties are applied in order to distinguish between stability and asymptotic stability. In the case that $\dot{L} > 0$, the same penalty $Q_3 = Q_2$ is applied to the fitness value. If instead $\dot{L} = 0$, a small penalty is applied ($Q_4 = Q_3/3.0$). A discussion of the results of varying these parameters is found in Section 6.3.1.

It's important to note that, due to floating point error and the discretization of the sampled regions, the fitness is at best an approximation. A fitness value of zero does not imply that the candidate is indeed a Lyapunov function for the system and sampling region in question. Rather, highly fit candidates are functions which should be analyzed by the user to determine if they are indeed useful for understanding the system.

6.3 Run Parameters and Details

The GE approach to finding Lyapunov functions includes numerous parameters and settings which require investigation and optimization of their own. A complete parameter survey would require unreasonable resources and time, but an investigation of a subset of these settings led to better performance and deserves some discussion.

6.3.1 Lyapunov Evaluation Settings

Because expressions are created by combining variables, operators, and constants without understanding their meaning, it's possible that evaluation of these expressions may

result in undefined operations, such as division by zero. This problem in particular has been addressed in GP by the use of a protected division operator [37]. Instead of treating a division by zero as an error, the protected division operator returns 1 as the result of the operation. Though this operator causes evaluation to return incorrect results, for certain applications it proves useful by preventing the complete rejection of an individual due to a single bad operation. For the purpose of Lyapunov function discovery, this modified operator was not used, and division by zero was penalized in the same way that any other invalid or undefined operations were.

Another optional setting involves the assessment of a penalty for candidates in which $\dot{L}(x, y) = 0$, described in Figure 6.3. The usage of this setting depends on the system being investigated and may be altered based on the type of stability being sought and quality of results being generated. If one wishes to use a Lyapunov function to show that the rest point is asymptotically stable, then the time derivative of the candidate function should be negative definite. As such, candidates for which the time derivative is only negative semidefinite should be assigned a lower fitness value; most runs performed utilized $Q_4 = 0.33$ as the penalty which is assessed for each sample point at which $\dot{L}(x, y) = 0$.

In some cases, however, it is still possible to show asymptotic stability when the time derivative is negative semidefinite by applying LaSalle's theorem [34]. Disabling this penalty can allow acceptable candidates to be evaluated as being highly fit, allowing the user to decide if the function allows an argument based on LaSalle's theorem. Further, since the algorithm is designed to continue searching until either the maximum number of generations are completed or a sufficiently fit individual is found, runs will not terminate early on discovery of a candidate with a negative semidefinite time derivative unless this penalty is disabled. Discussion on the impact of changes to this setting is found in Section 6.4.

Since the evaluation of candidate Lyapunov functions is performed by checking a series of conditions over a set of sample points, an important question relates to the granularity of the sampling grid and its effect on the performance of the search. The selection of appropriate values for Δx and Δy on the sample grid requires a balance between precision and computational load. A choice to set $\Delta x = \Delta y = 0.1$ allows a fairly high degree of coverage while limiting the number of sample points to be on the order of 1000 for most regions of interest on the interval $[-2.0 : 2.0]$. Following the termination of a given run, a validation step using a much finer sampling grid can be used to check for failures that were missed in the initial evaluation.

6.4 Results

A number of systems were used to test the ability of GE to find a Lyapunov function. In order to visualize the resulting functions, phase space plots were created to show the points in a region surrounding U at which all three conditions were satisfied. The blue circles show points at which all conditions are met and the red squares indicate the points at which $\dot{L}(x, y) = 0$. Several subsections follow, detailing the results for different systems of equations. Candidate Lyapunov functions are shown in their generated form, without simplification.

6.4.1 System A

One system for which Lyapunov functions were sought is given by

$$\begin{aligned} F(x, y) &= y, \\ G(x, y) &= -x(1 + xy). \end{aligned}$$

The standard guess $x^2 + y^2$ was quickly discovered; a plot of the satisfied sample points is shown in Figure 6.4. This result confirms that the GE approach to finding Lyapunov functions works, discovering the same functions as a human would for a given system. When simple solutions such as $x^2 + y^2$ exist, the GE-based search quickly discovers them. Other systems present more of a challenge; the following sections describe more novel candidates which were evolved.

6.4.2 System B

Another system of interest is given by

$$\begin{aligned} F(x, y) &= y, \\ G(x, y) &= -x - x^2 y. \end{aligned}$$

This system served as basis for investigating several modifications to the settings used in this approach. The optimization described in Section 6.2 allows functions which fail to satisfy the first condition ($L(a, b) = 0$) to be transformed into L_{new}, a function which does satisfy this condition. Tests were performed to investigate the behavior of GE search both with and without this optimization. The same type of expressions were discovered with the optimization and without, with all 30 iterations finding equivalent expressions similar to the candidate shown in Figure 6.4.

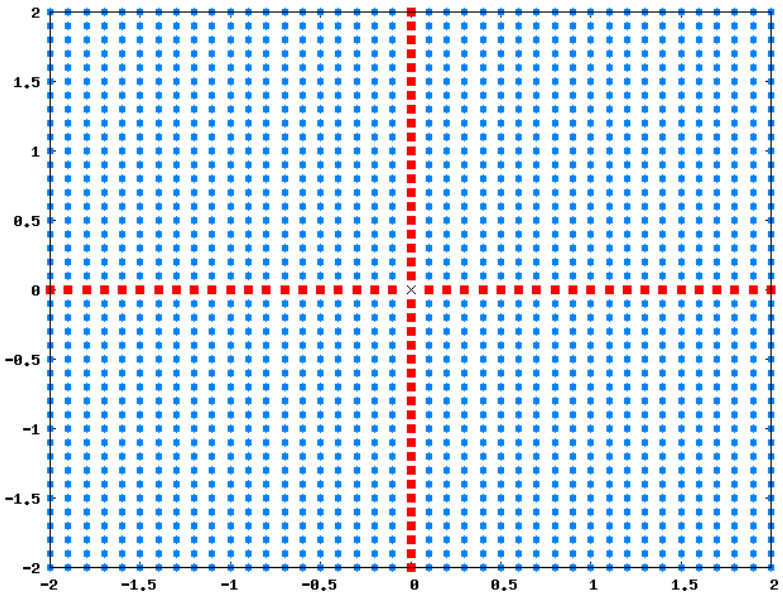


Figure 6.4: Graph of the Lyapunov function $x^2 + y^2$ for System A.

Additional tests were performed to investigate the effect of disabling the penalty for $\dot{L}(x, y) = 0$, equivalent to setting $Q_4 = 0$ rather than the default value 0.33 shown in Figure 6.3. Disabling this penalty has different results depending on the nature of the system being investigated. In the case of system B, functions of the form $x^2 + y^2$ satisfy the conditions everywhere, with the exception of the x-axis and y-axis where $\dot{L}(x, y) = 0$ as shown in Figure 6.4. Disabling this penalty for such a system generates the same functions but results in faster runtimes due to ideal fitness values being assigned for these candidates. When the penalty is enabled, the search continues until the maximum number of generations is reached, as the fitness values are nonzero on account of the penalty assigned on the axes. The mean number of generations of the steady state search needed to find an ideally fit solution with the L_{new} remapping was 100; without this remapping the mean was 149, suggesting a moderate gain in performance when using this optimization.

6.4.3 System C: Damped Pendulum

A system describing a damped pendulum is given by

$$\begin{aligned} F(x, y) &= y, \\ G(x, y) &= -\sin(x) - y. \end{aligned}$$

Figure 6.5 shows an interesting and unusual function which was evolved for this

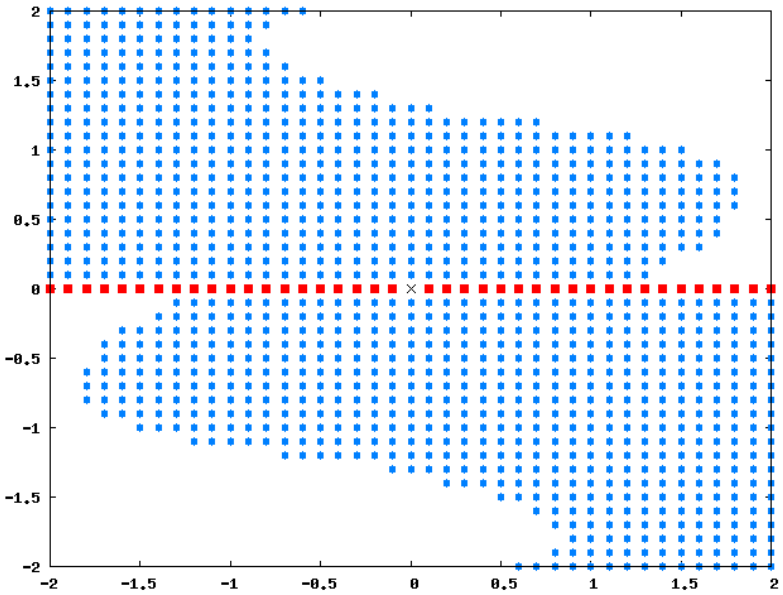


Figure 6.5: Candidate $V(x, y) = (y^2 + 4 * \sin(1 * y^2) + 4 * x^2)$ for System C

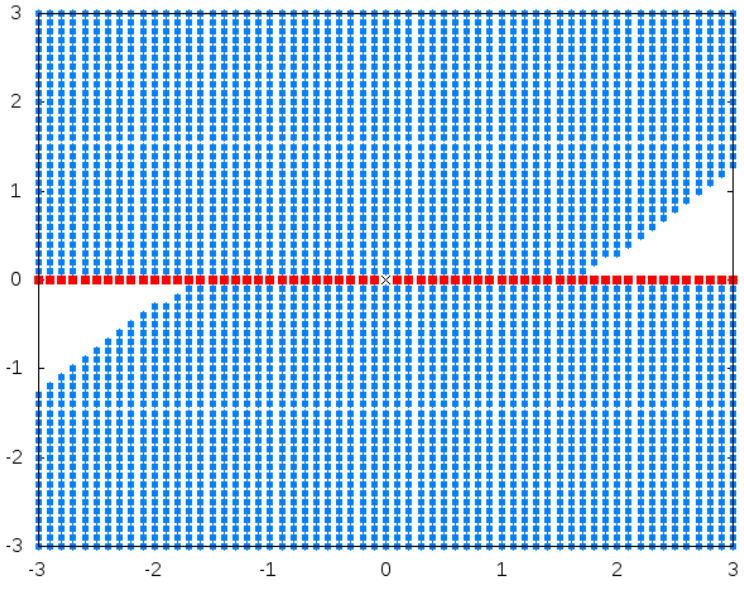


Figure 6.6: Candidate $V(x, y) = 2 * 3 * \text{sqrt}((x^2 + y^2)) + y^2$ for System C

system, $(y^2 + 4 * \sin(1 * y^2) + 4 * x^2)$. This expression was evolved during a run in which the region of interest was $[-1 : 1]$; in this region the evolved expression slightly outperforms the standard $x^2 + y^2$ in terms of the fitness function being used. When the area of interest was expanded to [-3:3], another candidate was found which has greater coverage than $x^2 + y^2$: $2 * 3 * \text{sqrt}((x^2 + y^2)) + y^2$, shown in Figure 6.6.

6.4.4 System D: Nonlinear system with tangent

Several results of interest came from testing on a nonlinear system given by

$$\begin{aligned} F(x, y) &= -\tan(x) + y^2, \\ G(x, y) &= -y + x. \end{aligned}$$

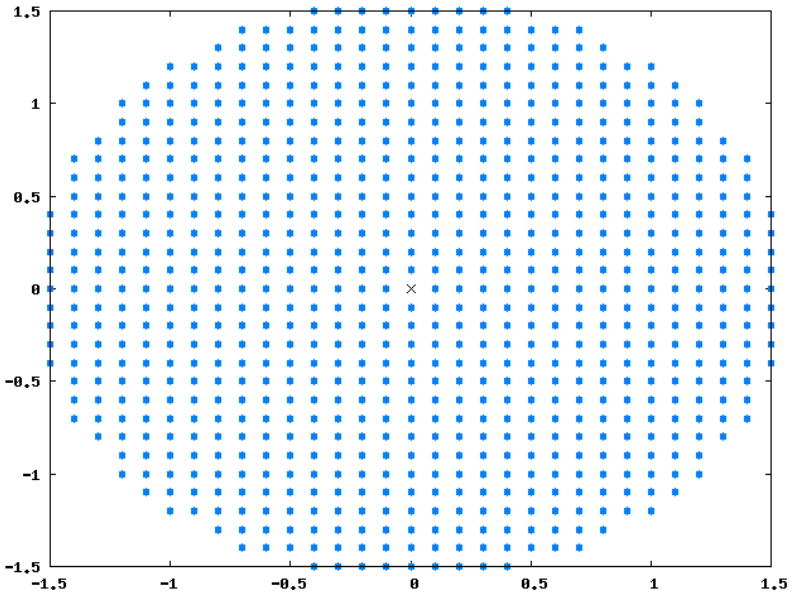


Figure 6.7: Graph of the Lyapunov function $2 * \tan(\text{sqrt}((y^2 + x^2)))$. System D.

Figure 6.7 shows the results for the candidate $2 * \tan(\text{sqrt}((y^2 + x^2)))$. This candidate satisfies the conditions for a Lyapunov function in a roughly circular region surrounding the rest point, utilizing the square root and tangent functions to create an expression that might not be found by traditional approaches.

Figure 6.8 shows another unusual construct in the use of $\text{sqrt}(y^2)$ to form the absolute value of y as a component of the candidate. Finally, Figure 6.9 shows a third candidate, $(y^2 + \text{sqrt}(x^2) - \cos(x^2) - x^2)$. This candidate, while not generating a simply connected domain, demonstrates both a novel functional form and a large region in which

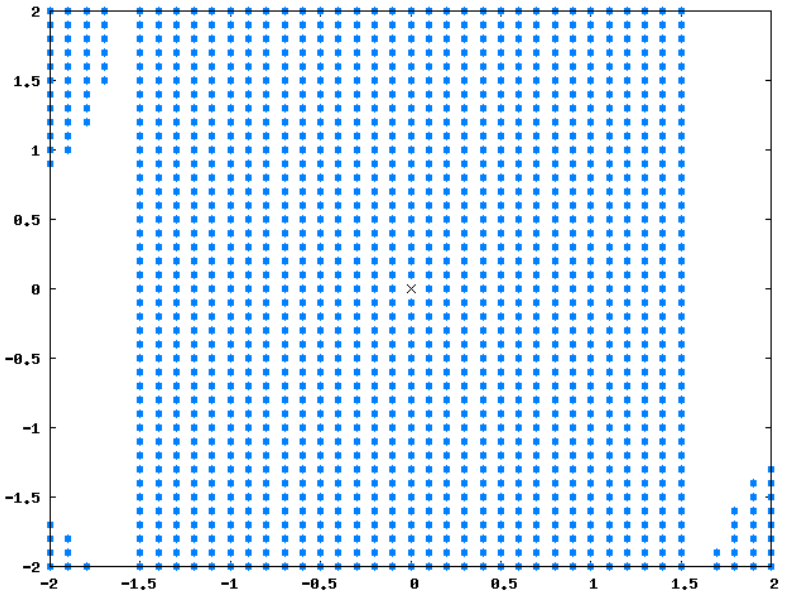


Figure 6.8: Graph of the Lyapunov function $(x^2 + (y^2 * \text{sqrt}(y^2)))$ for System D.

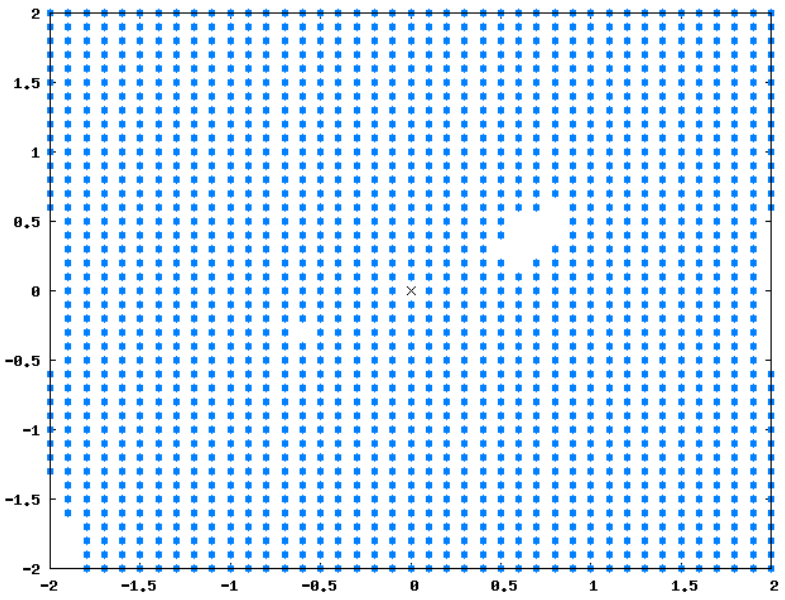


Figure 6.9: Graph of the Lyapunov function $(y^2 + \text{sqrt}(x^2) - cos(x^2) - x^2)$ for System D.

the majority of sampled points satisfy the criteria for a Lyapunov function. This sort of result is occasionally found because the fitness function seeks to minimize the number of sample points at which the Lyapunov criteria fail to hold, rather than attempting to grow a region in which all points satisfy the conditions. An interesting variation on this approach would involve iterative runs, with the sampling region starting very small and being grown as maximally fit solutions are found for the current region. It's not clear whether this approach would be an improvement over the existing one.

Chapter 7

Circuit Design

Electronic circuits come in many forms, but generally consist of a combination of interconnected components. Circuits can be digital, analog, or a combination of the two. Digital circuits are composed of logic gates which control the discrete-valued output of the circuit. Analog circuits are made up of components such as resistors and capacitors and control a signal which varies continuously, rather than taking on discrete values like the digital circuit does. Designing analog circuits is the third problem for which GE was used, one which is quite different from the mathematical problems previously described. The goal of studying this problem is to discover if the power of GE search shown in the regression and Lyapunov function research is also shown in a different class of problem.

7.1 Problem Overview

As mentioned previously, analog circuits consist of a number of interconnected components including resistors, capacitors, transistors, and inductors. For the purpose of this research, the components were limited to resistors, capacitors, and inductors.

In order to gain an understanding of the behavior of a circuit, tests must be run, either by building the circuit and actually applying a current to it, or by simulating these tests using a software package such as SPICE. SPICE was originally developed at the University of California, Berkeley to accept text descriptions of electronic components and their connections and build a mathematical model of the circuit, allowing simulations of the circuit's behavior [9]. These textual descriptions are known as netlists and have become the de facto standard for circuit simulation packages. Numerous variants of SPICE have since been created, including the one used in this research, ngspice.

The idea of evolving circuits is not new; in fact, circuit design is one of the most

well known applications of Genetic Programming. Koza has done a great deal of work on the subject of designing electronic circuits using GP [39, 41, 40], with excellent results on a number of different circuits. Other work on evolving circuits has been carried out by other researchers, including Kruiskamp's use of a GA to evolve operational amplifiers [42, 43]. Given the exceptional results in this area owing to a great deal of research over more than a decade, the goal of this work is not to meet or exceed the work of Koza but rather to investigate the feasibility of evolving analog circuits using GE and to provide a basis for future investigation.

7.2 GE Approach to Analog Circuit Design

7.2.1 Netlists

```
LPF.CIR - SIMPLE RC LOW-PASS FILTER
*
* Voltage source
VS    1    0    AC    1    SIN(0VOFF 1VPEAK   2KHZ)
* Begin variable section which defines the components
* and their interconnections
*
R1    1    2    1K
C1    2    0    0.032UF
*
* End variable section
*
* ANALYSIS
.AC     DEC     5 10 10MEG
.PRINT    AC     VM(2) VP(2)
.END
```

Figure 7.1: A sample netlist describing a low-pass filter and defining an AC analysis to be performed.

In order to use ngspice to simulate analog circuits, netlist representations of the candidate circuits were generated. For a given run, all candidates shared some portions of the netlist description, including the definition of the voltage source and the simulation commands which define the analysis to be performed and the desired output. An example netlist is shown in Figure 7.1. This netlist describes a simple, two component circuit along with instructions for simulating its behavior. The components consist of a $1k\Omega$ $(1,000\,Ohm)$ resistor and a 32 nF $(3.2 * 10^{-8}$ Farad) capacitor joined in series. An AC voltage source is

defined, with a magnitude of 1 V. In this example, the source is sinusoidal with no offset, an amplitude of 1 V, and a frequency of 2 kHz. The analysis section defines the test to be run and the desired output data. The voltage magnitude and phase at node 2 are printed.

The part of the netlist which distinguishes one circuit from another is the listing of components and their connections. All circuits share certain portions, including the analysis instructions which are used by ngspice to simulate the circuit and generate the output samples. These samples are compared to the samples generated by the target netlist.

7.2.2 Evaluation

As mentioned in section 7.1, a tool called ngspice was used to simulate circuits described by evolved netlists. The grammars define the phenotypes (netlists) in such a way that only the list of RLC components, their values, and the interconnections are allowed to vary. The input voltage and the instructions that define the analysis and output data are the same in all individuals, allowing a uniform set of output samples to be used in the fitness function. Much like the fitness functions used in symbolic regression problems, the raw fitness is calculated as the sum of squared error, with the error given as the difference between the output samples of the target circuit and those of the candidates. In this case, the output samples correspond to voltage and phase at a given node.

After the genotype to phenotype mapping was performed, a post-processing step took place to facilitate proper formatting, since netlists require line breaks between component listings and other entries, and because each component is required to be identified by a unique name (e.g. R1, R2,...RN for resistors). Line breaks and unique numbering for components were generated from markers ($NL and $CHANGEME respectively) within the generated phenotype to create a usable netlist.

7.2.3 Grammars

Selecting the grammar for this problem was a particularly interesting task since any circuit which didn't connect the input node to the output node would fail to generate any output. The challenge was to constrain the phenotypes such that the netlists corresponded to complete circuits which generate some output while still yielding an adequate search space for candidate circuits. Ideally, the grammar would define a circuit consisting of the voltage source, V, the output node O, and some sub-circuit S such that V is connected to S which is in turn connected to O. Similarly S should be connected between V and O to ensure that the complete circuit is closed.

```
<expr> ::= <heading>$NL<vs>$NL<components>$NL<end>

<heading> ::= EVOLVED.CIR - Evolved circuit $NL
<vs>         ::= VS   1   0   AC   1   SIN(0VOFF 1VPEAK    2KHZ) $NL

<components> ::= <basecomp>
             | <basecomp> <components>

<basecomp>   ::= <component> <node> <node> <val> $NL


<component> ::= R$CHANGEME
             |  C$CHANGEME
             |  L$CHANGEME

<val>       ::= <C>
             |  <C><C>
             |  0.<C>
             |  <val>E<C>
             |  <val>E-<C>

<node>     ::=  0
             |   1
             |   2
             |   3
             |   4
             |   5
             |   6
             |   7
             |   8
             |   9

<C>         ::= 1
             |   2
             |   3
             |   4
             |   5
             |   6
             |   7
             |   8
             |   9

<end>       ::= .AC    DEC   5  10  10MEG$NL.PRINT   AC   VM(2) VP(2)
```

Figure 7.2: Free1: A flexible grammar for a simple AC circuit.

One (naive) approach to evolving an RLC circuit is to allow any combination of resistors, capacitors, and inductors of varying values to be connected in any way. A grammar which provides this level of freedom is shown in Figure 7.2. Without any structure imposed on the netlist (and the corresponding circuit), most of the generated circuits were open between the two measured nodes, meaning that no current could flow. A constant corresponding to the worst possible fitness is assigned to these candidates, since the circuit fails to function. The power of a grammar based structure for phenotypes is wasted if basic domain knowledge isn't used to design a grammar which limits the search space appropriately. Nonetheless, this approach was tested in order to provide a baseline metric for generic search.

A second approach attempts to impose a general structure on the phenotype to ensure that the generated circuits are closed, i.e. current flows between the two nodes being considered. This approach allows all generated circuits to be evaluated and ranked according to the proximity of their output to that of the target circuit. The search space is limited, perhaps excessively, by the static grammar. The maximum number of nodes must be selected a priori and encoded in the grammar, limiting the breadth of the search being performed. Constructing a grammar for this approach is difficult without the use of a tool for generating it.

A third approach decouples the grammar from the phenotype and instead uses a rules-based, constructive grammar. The grammar encodes rules or instructions, which are in turn evaluated to build the circuit. One rule creates a new component (R,L, or C), using two existing nodes as its endpoints. Another rule adds a new node to the list of available nodes. The purpose of this approach is to combine the benefits of the second approach with the flexibility of the first. In other words, to generate valid, closed circuits without imposing a static structure via the grammar. The process of building the circuit consists of defining the two nodes between which the voltage will be measured, then adding additional components in between them such that a complete circuit exists. Any number of additional nodes might be added, with any number of components connecting two of those nodes. The additional level of abstraction more closely matches the biological gene expression process upon which EAs are based. This approach was not investigated in this research but remains as a promising direction for future work.

7.3 Results

As mentioned in Section 7.2.3, a number of possible approaches exist for structuring the search for RLC circuits. The first, naive approach allows for maximal freedom in the structure of the evolved circuits. Predictably, this approach fails to perform well, owing to the large number of individuals which fail to form a closed circuit. Without a closed path between in the input voltage source and the output node, no current flows and no measurement is possible. These individuals are assigned an extremely high value by the fitness function, corresponding to very poor fitness.

```
<expr> ::= <heading>$NL<vs>$NL<components>$NL<end>

<heading> ::= EVOLVED.CIR - Evolved circuit $NL
<vs>         ::= VS   1   0   AC   1   SIN(0VOFF 1VPEAK    2KHZ) $NL

<components> ::= <basecomp>
             | <basecomp> <adcomp>

<basecomp>   ::= <component> 2 0 <val> $NL<component> 1 2 <val> $NL

<adcomp>     ::= <component> 2 0 <val> $NL
             | <component> 1 2 <val> $NL

<component> ::= R$CHANGEME
             |  C$CHANGEME

<val>       ::= <C>
            |  <C><C>
            |  0.<C>
            |  <val>E<C>
            |  <val>E-<C>

<C>         ::= 1
            |   2
            |   3
            |   4
            |   5
            |   6
            |   7
            |   8
            |   9

<end>       ::= .AC   DEC   5  10  10MEG$NL.PRINT   AC   VM(2) VP(2)
```

Figure 7.3: LP1.bnf, A grammar defining an embryonic circuit for simple RC filter.

As Koza notes in his work on circuit evolution [41], a basic structure he calls an embryonic circuit must be created based on the user's knowledge of the problem domain. This embryonic circuit defines the general structure of the evolved circuit, narrowing the scope of variations between evolved individuals to a subset of the components which comprise the circuit. Figure 7.2 shows a very non-restrictive grammar which defines an embryonic circuit in a very general way, allowing resistors, capacitors, and inductors to be added in any configuration using up to 10 nodes. No guarantee of closed circuits is made, so many candidates created with this grammar will describe nonsensical circuits. A series of runs were performed using the grammar to gain a baseline for comparison to more well-designed grammars.

Figure 7.3 shows a grammar used to define the embryonic circuit, the allowable variation within it, and the commands to be used in simulating the circuit to analyze the output voltage at node 2. This grammar is highly tuned, ensuring that a component, either a resistor or a capacitor, connects node 2 to node 0 and node 1 to node 2. It allows additional components to connect these nodes as well. The type of component, the associated value (either resistance or capacitance), and the existence of additional parallel components between the nodes are allowed to vary.

```
LP.CIR - SIMPLE RC LOW-PASS FILTER
*
VS  1   0   AC   1  SIN(0VOFF 1VPEAK 2KHZ)
*
R1  1   2   1K
C1  2   0   0.032UF
*
* ANALYSIS
.AC     DEC     5 10 10MEG
.PRINT    AC     VM(2) VP(2)
.END
```

Figure 7.4: The netlist for a simple RC circuit. This circuit consists of a resistor and a capacitor in series, producing a low-pass filter.

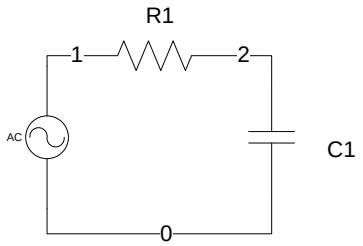


Figure 7.5: Circuit diagram of a simple low-pass filter described in lp.cir.

7.3.1 RC Low-pass Filter

A few basic analog circuits were selected as targets for evolutionary search, some of which were inspired by examples provided by Banzhaf [9]. The first circuit used as a target is a simple low-pass filter consisting of one resistor and one capacitor; its netlist is shown in

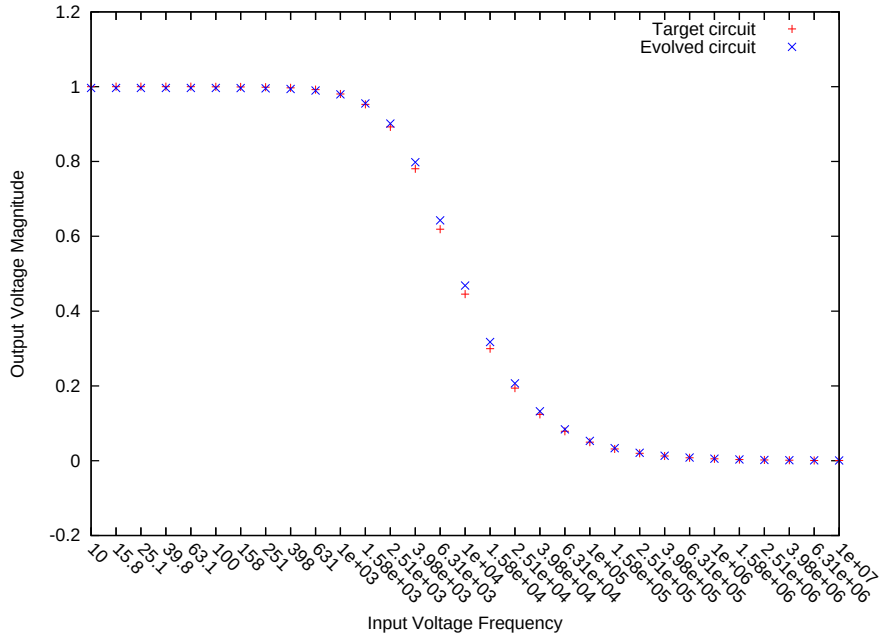


Figure 7.6: A plot of the voltage at node 2 on the target circuit and the least fit evolved circuit for a set of runs.

Figure 7.4. A 30 run set was executed using the Free1 grammar, shown in Figure 7.2. This grammar provides a great deal of freedom in the form of the evolved netlists; the search results were rather poor due to the high rate of invalid circuits being created. Only one out of 30 circuits was found to exactly match the target voltages, and only 5/30 had a raw fitness less than 8.7×10^{-5}.

A 30 run set executed using the LP1.bnf grammar shown in Figure 7.3 performed extremely well, with 12/30 producing perfect matches for the target filter and 29/30 having a raw fitness less than 8.7×10^{-5}. Since plots of the most fit individuals would be of little interest, Figure 7.6 shows a plot of the voltage values for the target circuit compared with the worst evolved circuit from this run set. Even this individual managed to closely match the desired voltages.

7.3.2 Wien Bridge Oscillator Sub-circuit

Another target circuit chosen is part of a larger circuit known as a Wien bridge oscillator, used to generate sine waves for various applications. The target circuit consists of two capacitors and two resistors and the analysis is performed on a range of frequencies between 10 Hz and 1000 Hz with 20 samples between 10 and 100 and another 20 samples between 100 and 1000, logarithmically spaced. The netlist and diagram are shown in Figures

7.7 and 7.8.

One of the best solutions found using the grammar in Figure A.6 is shown in netlist format in Figure 7.9. Figures 7.10 and 7.11 show the voltage magnitude and phase for both the target and this evolved circuit; they're almost identical. Testing was performed using three different grammars, the first two of which are shown in Figures A.6 and A.7.

```
WIEN.CIR - PART OF WIEN BRIDGE
*
V      1    0    AC  1
R1     1    3    2K
C2     3    2    796N
R3     2    0    2K
C4     2    0    796N
.AC    DEC  20   10 1000
.PRINT AC VM(2)  VP(2)
.END
```

Figure 7.7: The netlist for part of a wien bridge circuit.

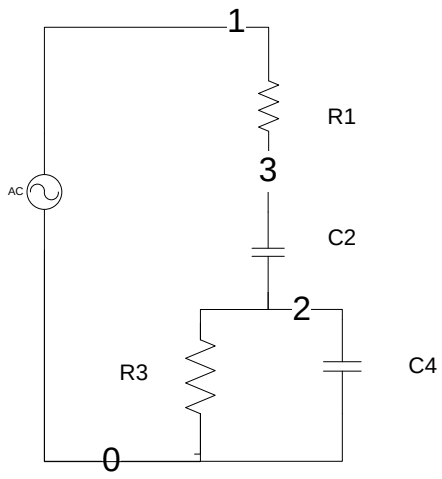


Figure 7.8: Circuit diagram for part of a wien bridge, defined in wien.cir.

```
EVOLVED.CIR - Evolved circuit

VGEN   1   0   AC   1
R1 1 3 80E-6
C2 2 0 20

C3 2 3 18
R4 2 0 90E-6

.AC    DEC  20   10  1000
.PRINT   AC   VM(2) VP(2)
```

Figure 7.9: An evolved netlist for the Wien bridge target circuit. The number and type of components is the same as in the target, and the values are very close to matching one another.

The first one, named wien2.bnf, is structured such that every generated netlist will have a component (R,L, or C) connected to the voltage source and another connecting the output node (2) to ground (0). This helps limit and direct the search, as someone

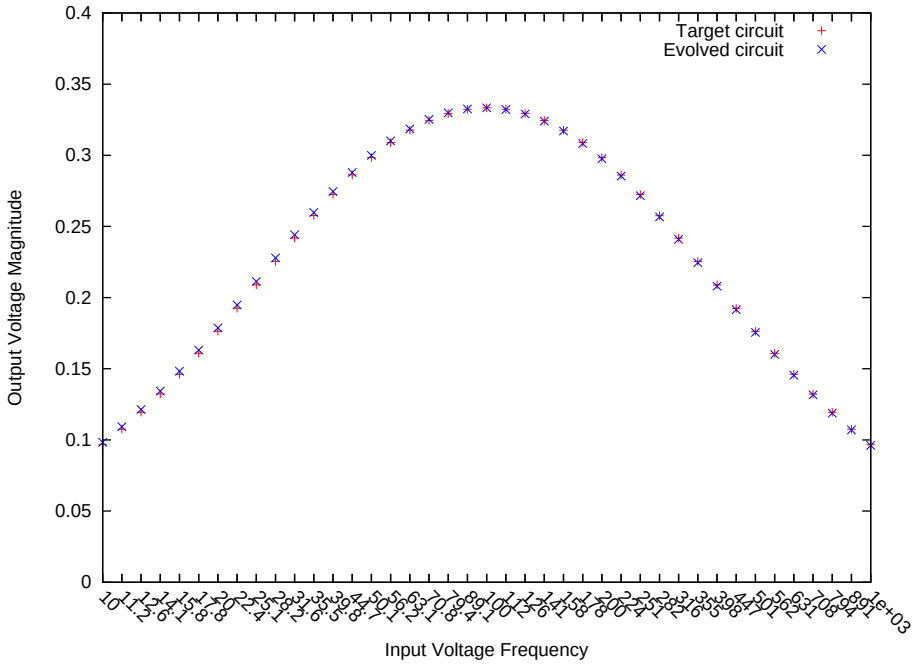


Figure 7.10: Comparison of output voltage magnitude for evolved and target circuits for the Wien bridge problem. The evolved netlist is shown in Figure 7.9.

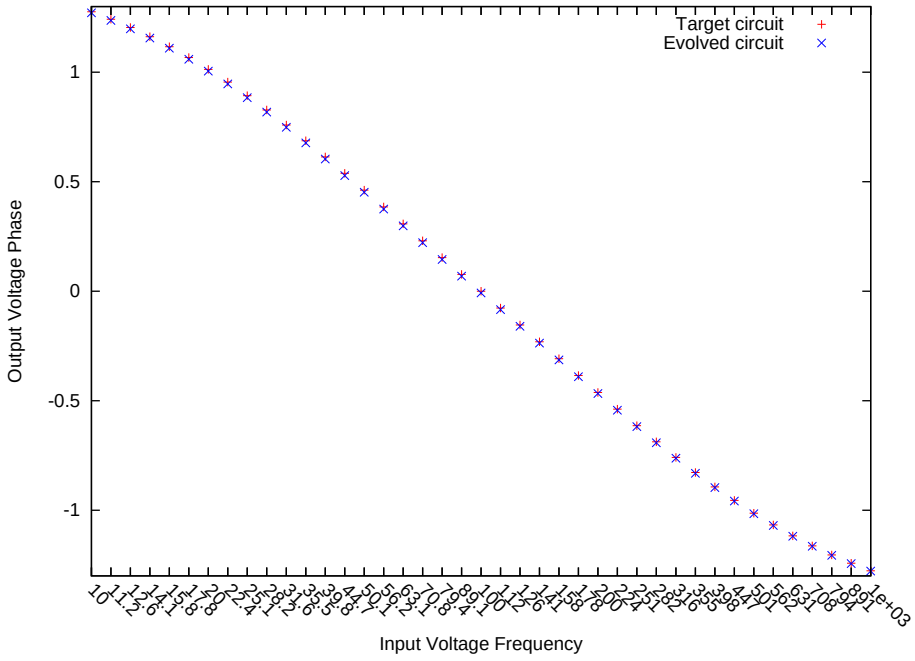


Figure 7.11: Comparison of output voltage phase for evolved and target circuits for the Wien bridge problem. The evolved netlist is shown in Figure 7.9.

designing such a circuit would clearly expect components to be connected in that way. The second grammar, wien3.bnf, attempts to encode more domain knowledge by restricting the values for resistance, inductance, and capacitance to eliminate values which are far outside the reasonable range. For example, the values for capacitance are allowed to vary between 1×10^{-19} and 9.9 farad. This range is still too large, allowing for unreasonably small and large values for the problem domain, but it improves greatly over the range of values allowed by wien2.bnf. Another grammar, wien4.bnf, was created as a duplicate of wien3.bnf, with the exception that only resistors and capacitors are used. Since the target circuit utilizes only resistors and capacitors, one might expect that runs utilizing this grammar would outperform those using wien3.bnf. Surprisingly, the mean fitness using wien4.bnf was actually worse than for runs using either of the other two grammars. Details on these runs are shown in Table C.2.

7.3.3 Parallel Resonant Circuit

```
PRTANK.CIR - RLC RESONANT CIRCUIT
*
R1      1   2   1K
R2      3   0   3
VGEN    1   0   AC  1
L1      2   3   1M
C1      2   0   0.2533U
*
* ANALYSIS
.AC     LIN 21  5000    15000
.PRINT  AC  VM(2) VP(2)
.END
```

Figure 7.12: The netlist for an RLC circuit. This defines a parallel resonant circuit using inductors, resistors, and capacitors.

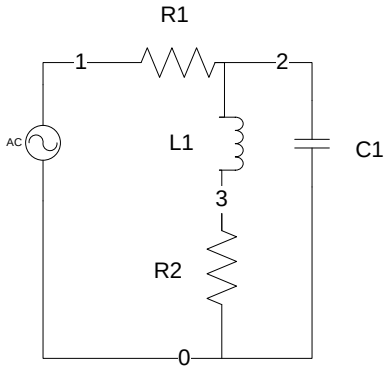


Figure 7.13: Circuit diagram of a parallel resonant circuit defined in prtank.cir.

The circuit described by the netlist shown in Figure 7.12 and illustrated in Figure 7.13 is an RLC resonant circuit with a frequency of 10 kHz. Analysis is performed with an AC voltage source at frequencies between 5000 and 15000 Hz at 500 Hz steps.

One evolved solution found using the grammar in Figure A.5 is shown in Figures 7.14 and 7.15. These plots compare the output voltage magnitude and phase with the

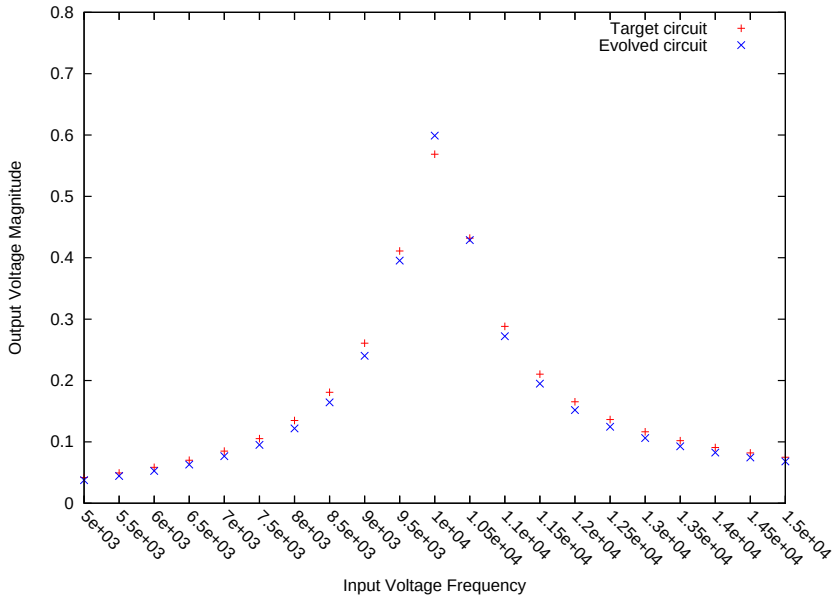


Figure 7.14: Comparison of output voltage magnitude for evolved and target circuits for the PRTANK problem. The evolved netlist is shown in Figure 7.16

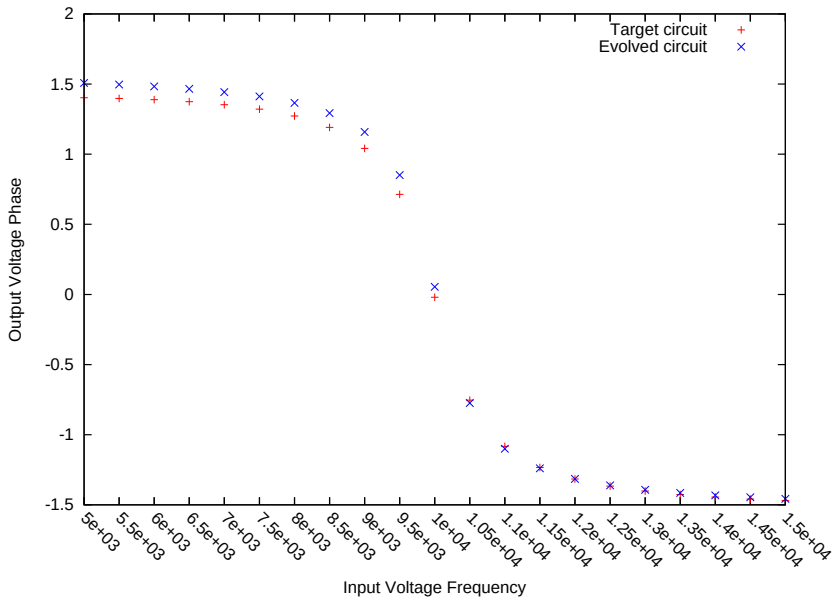


Figure 7.15: Comparison of output voltage phase for evolved and target circuits for the PRTANK problem. The evolved netlist is shown in Figure 7.16.

desired values. This solution matches the target circuit quite well over the range of sampled frequencies; its netlist description is given in Figure 7.16. Note the unusual notation for the resistance value of element R7. The structure of the grammar was such that repeated exponents were possible, and the parser used by ngspice appears able to cope with this. Testing suggests that the trailing characters are truncated, meaning that the value used for R7 is 0.8 Ω. Although this notation is strange and possibly invalid for use with other analysis software, the fitness function reported expressions using this notation to be highly fit.

```
EVOLVED.CIR - Evolved circuit

VGEN   1   0   AC   1
L1 2 0 0.7
R2 1 2 4
C3 0 3 64
L4 2 3 36E-7
C5 2 0 0.7E-4
C6 0 3 0.8
R7 1 3 8E-1E6
R8 2 3 6

.AC    LIN  21   5000  15000
.PRINT   AC   VM(2) VP(2)
```

Figure 7.16: Netlist for evolved circuit approximating PRTANK.CIR

This work constitutes only a preliminary investigation of circuit design using GE, but it demonstrates GE's ability to find circuits which produce the desired outputs. Additionally, these results show that GE is capable of evolving structures very different from the mathematical expressions found in previous experiments. Further study of GE's applications in designing circuits is warranted, with the goal of performing competitively with GP.

Chapter 8

Conclusions

Grammatical Evolution can be successfully applied to a variety of problems with relatively minor changes needed to transition from one problem domain to another. This research has shown a successful approach to finding mathematical functions in cases where direct approaches cannot be used, including finding Lyapunov functions. The quality and novelty of the evolved Lyapunov candidates shows this approach to be effective and useful. Additional results in symbolic regression, inverse regression, and circuit design demonstrate flexibility. The variation in performance shown when different grammars are used demonstrates the ability of a user to leverage domain-specific knowledge to guide the search.

Although the problems being investigated seem quite different and use different grammars to describe the evolved expressions, at its core the approach is the same. When desired outputs are known for a given set of inputs, it's possible to measure how well an expression is able to produces the desired outputs. In all three problem domains, a function which produces the desired output is sought. The form the function takes differs, with a netlist appearing quite different from a polynomial expression, but fundamentally they are the same. Thus, rather than being a conglomeration of unrelated experiments, this research can be viewed as a single cohesive study, with different aspects of the problem being investigated.

8.1 Further Work

A wide variety of directions are available for additional research using a GE search to solve problems. Given the sometimes profound effect on fitness caused by relatively minor changes in the grammar, a great deal of additional study and testing should take place to investigate improvements owing to the use of more appropriate grammars. An iterative

approach might be tested, in which the user starts with a basic grammar, performs a series of runs, then creates a modified grammar using feedback from the most recent set of runs. Such an approach would be something of a guided search, combining the power of evolution with the intuition and domain knowledge of a user.

Another area ripe for further study is modification or replacement of the underlying search algorithm. Steady state and generational genetic algorithms were used in this research, with a variety of adjustable settings such as mutation rate, crossover type and rate, and genotype length. The nature of GE is such that the search algorithm can be changed with very little modification to the rest of the system.

One promising possible alternative is Hierarchical Fair Competition (HFC) [28]. HFC is a structured, multi-population evolution model in which subpopulations are organized based on fitness levels. Research using HFC has shown a resistance to premature convergence and an ability to maintain diversity within a population without a loss of effectiveness [29, 30]. Combining GE and HFC could improve on the results described here.

Many problem-specific changes also deserve consideration. In the case of the Lyapunov functions, variations on the fitness function could certainly alter the performance of the search. For example, instead of using a fixed penalty for each of the conditions being tested, a variable penalty could be applied based on the degree to which a condition failed. When a value fails a requirement to be strictly negative at a given point, the penalty assessed could be calculated by the magnitude of the value. An approach like this might provide a better assessment of a candidate and could serve to direct the search towards more fit expressions. In the case of the circuit design problems, a more knowledgeable user could encode more specific structure into the grammars, based on experience and theoretical knowledge exceeding that of the author. Indeed, there exists a great deal of work left for future research in this area.

8.2 Final Thoughts

Scientific and technological knowledge continues to grow at a fantastic rate, radically changing the lives of billions of people. All the discoveries and innovations to date are the result of hard work and human ingenuity. Although this is likely to continue, our technology is reaching the point that we need tools to manage the complexity and scale of the things we build. Our technology has also reached the point that we can efficiently automate huge amounts of computation and tackle highly complex problems using computers. For these reasons it is important to find ways of combining the knowledge, intelligence, and

intuition of human scientists, engineers, and other experts with the immense computational power of machines to tackle increasingly complex problems and to continue to develop more powerful tools. GE may very well serve as one means of doing so and may prove to be part of the solution to tomorrow's challenges.

Bibliography

[1] A. Papachristodoulou and S. Prajna. On the Construction of Lyapunov Functions using the Sum of Squares Decomposition. In *IEEE Conf. on Dec. and Cont. (CDC)*, 2002.

[2] Bruce Alberts, Alexander Johnson, Julian Lewis, Martin Raff, Keith Roberts, and Peter Walter. *Molecular Biology of the Cell, Fourth Edition*. Garland, 2002.

[3] David Andre and Astro Teller. A study in program response and the negative effects of introns in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 12–20, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[4] C Anfinsen. The formation and stabilization of protein structure. *The Biochemical Journal*, 128:737–749, July 1972.

[5] Karl J. Astrom and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, illustrated edition edition, April 2008.

[6] Thomas Bäck and Hans-Paul Schwefel Günter Rudolph. Evolutionary programming and evolution strategies: similarities and differences. In *Proceedings of the 2nd Annual Conference on Evolutionary Programming*, pages 11–22, La Jolla, CA, 1993.

[7] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. Mccarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithm language ALGOL 60. *Commun. ACM*, 6(1):1–17, January 1963.

[8] Carl Banks. Searching for lyapunov functions using genetic programming. Technical report, Virginia Polytechnic Institute and State University, Blacksburg, VA.

[9] Walter Banzhaf. *Computer-Aided Circuit Analysis Using PSpice*. Prentice Hall, New Jersey, 1992.

[10] Wolfgang Banzhaf. Genotype-phenotype-mapping and neutral variation – A case study in genetic programming. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature III*, volume 866 of *LNCS*, pages 322–332, Jerusalem, 9-14 October 1994. Springer-Verlag.

[11] Anthony Brabazon and Michael O'Neill. Bond issuer credit rating with grammatical evolution. In *Proceedings of Applications of Evolutionary Computation EvoIASP*, pages 270–279, Coimbra, Portugal, 2004. Springer.

[12] Anthony Brabazon and Michael O'Neill. Grammatical differential evolution. In *Proceedings of International Conference on Artificial Intelligence*, 2006.

[13] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, January 2003.

[14] Michael O'Neill Connor Ryan. Grammatical evolution: A steady state approach. *Late Breaking Papers, Genetic Programming*, pages 180–185, 1998.

[15] Francis Crick. Central dogma of molecular biology. *Nature*, 227:561–563, 1970.

[16] Luis E. Da Costa, Jacques-Andre Landry, and Yan Levasseur. Treating noisy data sets with relaxed genetic programming. In Nicolas Monmarché, El-Ghazali Talbi, Pierre Collet, Marc Schoenauer, and Evelyne Lutton, editors, *Artificial Evolution*, volume 4926 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2007.

[17] Charles Darwin. *On the Origin of Species by Means of Natural Selection*. John Murray, London, 1859.

[18] Ivanoe De Falco, Antonio Della Cioppa, Domenico Maisto, Umberto Scafuri, and Ernesto Tarantino. Parsimony doesn't mean simplicity: Genetic programming for inductive inference on noisy data. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 351–360, Valencia, Spain, 11 - 13 April 2007. Springer.

[19] Larry J. Eshelman and J. David Schaffer. Preventing premature convergence in genetic algorithms by preventing incest. In Richard K. Belew and Lashon B. Booker, editors,

Proc. of the Fourth Int. Conf. on Genetic Algorithms, pages 115–122, San Mateo, CA, 1991. Morgan Kaufmann.

[20] Lawrence J. Fogel, Alvin J. Owens, and Michael J. Walsh. *Artificial Intelligence Through simulated Evolution.* John Wiley and Sons, New York, 1966.

[21] Forluvoft. Dna simple2. `http://commons.wikimedia.org/wiki/File:DNA_simple2.svg`. Released into public domain by Wikipedia user Forluvoft.

[22] Frank D. Francone, Markus Conrad, Wolfgang Banzhaf, and Peter Nordin. Homologous crossover in genetic programming. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proc. of the Genetic and Evolutionary Computation Conf. GECCO-99*, pages 1021–1026, San Francisco, CA, 1999. Morgan Kaufmann.

[23] George and Mervin E. Muller. A note on the generation of random normal deviates. *The Annuls of Mathematical Statistics*, 29(2):610–611, 1958.

[24] Stephen Jay Gould. *The structure of evolutionary theory.* Harvard University Press, Cambridge, 2002.

[25] B. Grosman and D. R. Lewin. Lyapunov-based stability analysis automated by genetic programming. In *IEEE International Symposium on Computer-Aided Control Systems Design, 2006*, pages 766–771, Munich, Germany, 4-6 October 2006. IEEE.

[26] Benyamin Grosman and Daniel Lewin. Automatic generation of lyapunov functions using genetic programming. In *Proceedings of the 16th IFAC Word Congress*, 2005.

[27] John H. Holland. *Adaptation in Natural and Artifical Systems.* University of Michigan Press, Ann Arbor, MI, 1975.

[28] Jianjun Hu, Erik Goodman, Kisung Seo, Zhun Fan, and Rondal Rosenberg. The hierarchical fair competition framework for sustainable evolutionary algorithms. *Evolutionary Computation*, 13(2):241–277, Summer 2005.

[29] Jianjun Hu, Erik D. Goodman, Kisung Seo, Zhun Fan, and Ronald C. Rosenberg. HFC: A continuing EA framework for scalable evolutionary synthesis. In *Proceedings of the 2003 AAAI Spring Symposium - Computational Synthesis: From Basic Building Blocks to High Level Functionality*, pages 106–113, Stanford, California, 24March 2003. AAAI press.

[30] Jianjun Hu, Erik D. Goodman, Kisung Seo, and Min Pei. Adaptive hierarchical fair competition (AHFC) model for parallel evolutionary algorithms. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 772–779, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[31] Janine H. Imada and Brian J. Ross. Using feature-based fitness evaluation in symbolic regression with added noise. In Marc Ebner, Mike Cattolico, Jano van Hemert, Steven Gustafson, Laurence D. Merkle, Frank W. Moore, Clare Bates Congdon, Christopher D. Clack, Frank W. Moore, William Rand, Sevan G. Ficici, Rick Riolo, Jaume Bacardit, Ester Bernado-Mansilla, Martin V. Butz, Stephen L. Smith, Stefano Cagnoni, Mark Hauschild, Martin Pelikan, and Kumara Sastry, editors, *GECCO-2008 Late-Breaking Papers*, pages 2153–2158, Atlanta, GA, USA, 12-16 July 2008. ACM.

[32] Maarten Keijzer, Conor Ryan, Michael O'Neill, Mike Cattolico, and Vladan Babovic. Ripple crossover in genetic programming. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 74–86, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[33] J. Kennedy and R. C. Eberhart. Particle swarm optimization. In *Proc. of the IEEE Int. Conf. on Neural Networks*, pages 1942–1948, Piscataway, NJ, 1995. IEEE Service Center.

[34] Hassan K. Khalil. *Nonlinear Systems*. Prentice Hall, New Jersy, 2002.

[35] Kenneth E. Kinnear, Jr. Generality and difficulty in genetic programming: Evolving a sort. In Stephanie Forrest, editor, *Proc. of the Fifth Int. Conf. on Genetic Algorithms*, pages 287–294, San Mateo, CA, 1993. Morgan Kaufmann.

[36] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programms*. MIT Press, Cambridge, MA, 1994.

[37] John R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, 1992.

[38] John R. Koza, David Andre, Forrest H Bennett III, and Martin Keane. *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman, April 1999.

[39] John R. Koza, David Andre, Forrest H Bennett III, and Martin A. Keane. Design of a 96 decibel operational amplifier and other problems for which a computer program evolved by genetic programming is competitive with human performance. In Mitsuo Gen and Weixuan Zu, editors, *Proceedings of l996 Japan-China Joint International Workshop on Information Systems*, pages 30–49, Ashikaga, 4-16 October 1996.

[40] John R. Koza, Forrest H Bennett III, David Andre, Martin A. Keane, and Frank Dunlap. Automated synthesis of analog electrical circuits by means of genetic programming. *IEEE Transactions on Evolutionary Computation*, 1(2):109–128, July 1997.

[41] John R. Koza, Forrest H Bennett III, Jason Lohn, Frank Dunlap, Martin A. Keane, and David Andre. Automated synthesis of computational circuits using genetic programming. In *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*, pages 447–452, Indianapolis, 13-16 April 1997. IEEE Press.

[42] Wim Kruiskamp. *Analog design automation using genetic algorithms and polytopes*. PhD thesis, Technical University of Eindhoven, 1996.

[43] Wim Kruiskamp and Domine Leenarts. Darwin: Cmos op-amp synthesis by means of a genetic algorithm. In *Proceedings of the 32nd ACM/IEEE Conference on Design Automation*, pages 433–438, 1995.

[44] W. B. Langdon. Quadratic bloat in genetic programming. In Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 451–458, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[45] W. B. Langdon and R. Poli. Fitness causes bloat. Technical Report CSRP-97-09, University of Birmingham, School of Computer Science, Birmingham, B15 2TT, UK, 24 February 1997.

[46] Sean Luke. *Essentials of Metaheuristics*. 2009. available at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[47] Sean Luke and Liviu Panait. Fighting bloat with nonparametric parsimony pressure. In Juan J. Merelo-Guervos, Panagiotis Adamidis, Hans-Georg Beyer, Jose-Luis

Fernandez-Villacanas, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VII*, number 2439 in Lecture Notes in Computer Science, LNCS, pages 411–421, Granada, Spain, 7-11 September 2002. Springer-Verlag.

[48] Sean Luke and Liviu Panait. Is the perfect the enemy of the good? In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 820–828, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[49] Horacio J. Marquez. *Nonlinear Control Systems: Analysis and Design.* Wiley, New Jersy, 2003.

[50] Brad L. Miller and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.

[51] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[52] Kuan Luen Ng and Rolf Johansson. Evolving programs and solutions using genetic programming with application to learning and adaptive control. *Journal of Intelligent and Robotic Systems*, 35(3):289–307, November 2002.

[53] Marianne O'Driscoll, Stephen McKenna, and J. J. Collins. Synthesising edge detectors with grammatical evolution. In Alwyn M. Barry, editor, *GECCO 2002: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*, pages 137–140, New York, 8 July 2002. AAAI.

[54] Roland Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–81, March 1995.

[55] M. O'Neill and C. Ryan. Genetic code degeneracy: Implications for grammatical evolution and beyond. In D. Floreano, J.-D. Nicoud, and F. Mondada, editors, *Advances in Artificial Life*, volume 1674 of *LNAI*, page 149, Lausanne, 13-17 September 1999. Springer Verlag.

[56] Michael O'Neill, Anthony Brabazon, and Catherine Adley. The automatic generation of programs for classification problems with grammatical swarm. In *Proceedings of the*

2004 IEEE Congress on Evolutionary Computation, pages 104–110, Portland, Oregon, 20-23 June 2004. IEEE Press.

[57] Michael O'Neill, Anthony Brabazon, Conor Ryan, and J. J. Collins. Evolving market index trading rules using grammatical evolution. In Egbert J. W. Boers, Stefano Cagnoni, Jens Gottlieb, Emma Hart, Pier Luca Lanzi, Günther R. Raidl, Robert E. Smith, and Harald Tijink, editors, *Applications of evolutionary computing: Proc. EvoWorkshops 2001*, pages 343–352, Berlin, 2001. Springer.

[58] Michael O'Neill and Conor Ryan. Under the hood of grammatical evolution. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proc. of the Genetic and Evolutionary Computation Conf. GECCO-99*, pages 1143–1148, San Francisco, CA, 1999. Morgan Kaufmann.

[59] Michael O'Neill and Conor Ryan. Crossover in grammatical evolution: A smooth operator? In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 149–162, Edinburgh, 15-16 April 2000. Springer-Verlag.

[60] Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, volume 4 of *Genetic programming*. Kluwer Academic Publishers, 2003.

[61] John O'Sullivan and Conor Ryan. An investigation into the use of different search strategies with grammatical evolution. In James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 268–277, Kinsale, Ireland, 3-5 April 2002. Springer-Verlag.

[62] Liviu Panait and Sean Luke. Alternative bloat control methods. In Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund Burke, Paul Darwen, Dipankar Dasgupta, Dario Floreano, James Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andy Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 630–641, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[63] Kenneth Price and Rainer Storn. Differential evolution. *Dr. Dobb's Journal*, 1997.

[64] Ingo Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* frommann-holzbog, Stuttgart, 1973. German.

[65] Alex Rogers and Adam Prügel-Bennett. Modelling the dynamics of a steady-state genetic algorithm. In *Foundations of Genetic Algorithms (FOGA-5). Preliminary Version of the Proceedings*, pages 161–171. Leiden, 1998.

[66] Simon Ronald. Duplicate genotypes in a genetic algorithm. In *Proceedings of IEEE International Conference on Evolutionary Computation*, 1998.

[67] Justinian Rosca. Generality versus size in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 381–387, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[68] Conor Ryan, Hammad Majeed, and Atif Azad. A competitive building block hypothesis. In Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund Burke, Paul Darwen, Dipankar Dasgupta, Dario Floreano, James Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andy Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 654–665, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[69] S. Prajna, A. Papachristodoulou, and F. Wu. Nonlinear control synthesis by sum of squares optimization: a lyapunov-based approach. In *Asian Conf. Cont. (ASCC)*, 2004.

[70] Shankar Sastry. *Nonlinear Systems: Analysis, Stability, and Control.* Springer, New York, 1999.

[71] Hans-Paul Schwefel. *Evolutionsstrategie und numerische Optimierung.* PhD thesis, Technische Universität Berlin, Berlin, Germany, 1975. German.

[72] Sara Silva and Jonas Almeida. Dynamic maximum tree depth. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary*

Computation – GECCO-2003, volume 2724 of *LNCS*, pages 1776–1787, Chicago, 12-16 July 2003. Springer-Verlag.

[73] Carin Smith. Evolving Lyapunov Functions. Technical report, South Dakota School of Mines and Technology, Rapid City, SD, USA, May 2006.

[74] Jim E. Smith and Terence C. Fogarty. Self adaptation of mutation rates in a steady state genetic algorithm. In *Proc. of 1996 IEEE Conf. on Evolutionary Computation*, pages 318–323, New York, 1996. IEEE Press.

[75] P. W. H. Smith. Controlling code growth in genetic programming. In Robert John and Ralph Birkenhead, editors, *Advances in Soft Computing*, pages 166–171, De Montfort University, Leicester, UK, 2000. Physica-Verlag.

[76] Gilbert Syswerda. Uniform crossover in genetic algorithms. In James D. Schaffer, editor, *Proc. of the Third Int. Conf. on Genetic Algorithms*, pages 2–9, San Mateo, CA, 1989. Morgan Kaufmann.

[77] Gilbert Syswerda. A study of reproduction in generational and steady-state genetic algorithms. In Gregory J. Rawlins, editor, *Foundations of genetic algorithms*, pages 94–101. Morgan Kaufmann, San Mateo, CA, 1991.

[78] Frank Vavak and Terence C. Fogarty. A comparative study of steady state and generational genetic algorithms for use in nonstationary environments. In *Proc. of the Society for the Study of Artificial Intelligence and Simulation of Behaviour Workshop on Evolutionry Computing 96*, pages 301–307. University of Sussex, 1996.

[79] Karsten Weicker and Nicole Weicker. Burden and benefits of redundancy. In *Proceedings of the Sixth Workshop on Foundations of Genetic Algorithms*, 2000.

Appendix A

Listing of Grammars

This appendix contains a listing of additional grammars used in this research.

```
<expr> ::= <expr><op><expr>
         | (<expr><op><expr>)
         | <var>^<const>
         | <const>*<expr>
<op>   ::= +
         | -
         | *
         | /
<var>  ::= X
         | Y
<const> ::= 1.0
         | 2.0
         | 3.0
         | 4.0
         | 5.0
         | 6.0
         | 7.0
         | 8.0
         | 9.0
```

Figure A.1: Reg4, a grammar used in regression on the system $-0.5Y(1.0 - X^2 + .1X^4) - X$

```
<expr> ::= <expr> <op> <expr>
         | (<expr> <op> <var>)
         | <var>
         | <const>
         | <fun>(<expr>)
<op>   ::= +
         | -
         | /
         | *
<fun>  ::= sin
         | cos
<var>  ::= X
         | A
<const> ::= 1.0
         |  2.0
         |  3.0
         |  4.0
```

Figure A.2: Reg16, a grammar used in symbolic regression runs.

```
<expr> ::= <expr><op><expr>
         | (<expr><op><expr>)
         | <var>^2
         | <const>*<expr>
         | <fun>(<expr>)
<fun>  ::= sin
         | cos
         | tan
         | sqrt
<op>   ::= +
         | -
         | *
         | /
<var>  ::= x
         | y
<const> ::= 1
         | 2
         | 3
         | 4
```

Figure A.3: Lyap8, a grammar used to search for Lyapunov functions for some systems.

```
<expr>    ::= <expr><op><expr>
          |   (<expr><op><expr>)
          |   <var>^<const>
          |   <const>*<expr>

<op>      ::= +
          |   -
          |   *

<var>     ::= X
          |   Y
<const>   ::= 1.0
          |   2.0
          |   3.0
          |   4.0
```

Figure A.4: A simple polynomial grammar used to define the search space for Lyapunov functions

```
<expr> ::= <heading>$NL<vs>$NL<components>$NL<end>

<heading> ::= EVOLVED.CIR - Evolved circuit $NL
<vs>          ::= VGEN   1   0   AC   1

<components> ::= <basecomp>
              | <basecomp> <adcomps>

<basecomp>   ::= <component> 2 0 <val> $NL<component> 1 2 <val> $NL

<adcomps>    ::= <adcomp>
              | <adcomp> <adcomps>

<adcomp>     ::= <component> 2 0 <val> $NL
              | <component> 1 2 <val> $NL
              | <component> 0 3 <val> $NL
              | <component> 1 3 <val> $NL
              | <component> 2 3 <val> $NL

<component> ::= R$CHANGEME
              |  C$CHANGEME
              |  L$CHANGEME

<val>        ::= <C>
              |  <C><C>
              |  0.<C>
              |  <val>E<C>
              |  <val>E-<C>

<C>          ::= 1
              |   2
              |   3
              |   4
              |   5
              |   6
              |   7
              |   8
              |   9

<end>        ::= .AC  LIN  21 5000  15000 $NL.PRINT  AC VM(2) VP(2)
```

Figure A.5: Premb1.bnf: A grammar used to evolve small RLC circuits.

```
<expr> ::= <heading>$NL<vs>$NL<base>$NL<components>$NL<end>

<heading> ::= EVOLVED.CIR - Evolved circuit $NL
<vs>          ::= VGEN   1   0   AC   1

<base>        ::= <component> 1 3 <val>$NL<component> 2 0 <val>$NL

<components> ::= <adcomp>
              | <adcomp> <adcomp>
              | <adcomp> <components>

<adcomp>      ::= <component> 2 0 <val> $NL
              | <component> 1 2 <val> $NL
              | <component> 0 3 <val> $NL
              | <component> 1 3 <val> $NL
              | <component> 2 3 <val> $NL

<component> ::= R$CHANGEME
              |  C$CHANGEME
              |  L$CHANGEME

<val>         ::= <num>
               | <num>E<num>
               | <num>E-<num>

<num>         ::= <C>
              |  <C><C>

<C>           ::= 1
              |   2
              |   3
              |   4
              |   5
              |   6
              |   7
              |   8
              |   9
              |   0

<end>         ::= .AC DEC  20  10  1000 $NL.PRINT AC  VM(2) VP(2)
```

Figure A.6: Wien2.bnf: A structured grammar used on the Wien Bridge problem.

```
<expr> ::= <heading>$NL<vs>$NL<base1>$NL<base2><components>$NL<end>
<heading> ::= EVOLVED.CIR - Evolved circuit $NL
<vs>         ::= VGEN   1   0   AC   1

<base1>       ::= R$CHANGEME 1 <node> <rval> $NL
             |   C$CHANGEME 1 <node> <cval> $NL
             |   L$CHANGEME 1 <node> <lval> $NL

<base2>       ::= R$CHANGEME 2 0 <rval> $NL
             |   C$CHANGEME 2 0 <cval> $NL
             |   L$CHANGEME 2 0 <lval> $NL

<components> ::= <adcomp>
             | <adcomp> <adcomp>
             | <adcomp> <components>

<adcomp>     ::= R$CHANGEME <node> <node> <rval> $NL
             |   C$CHANGEME <node> <node> <cval> $NL
             |   L$CHANGEME <node> <node> <lval> $NL

<node>       ::= 0
             |   1
             |   2
             |   3
             |   4
<rval>       ::= <val>
<cval>       ::= <num>E-<pow>
<lval>       ::= <num>E-<pow>
<val>        ::= <num>
             | <num>E<pow>
             | <num>E-<pow>

<num>       ::= <C>
            |  <C><C>
<pow>       ::= <C>
            |   1<C>
<C>         ::= 1
            |   2
            |   3
            |   4
            |   5
            |   6
            |   7
            |   8
            |   9
            |   0
<end>       ::= .AC  DEC  20  10  1000 $NL.PRINT  AC  VM(2) VP(2)
```

Figure A.7: Wien3.bnf: A grammar used on the Wien Bridge problem which uses more reasonable values for components.

Appendix B

Data From Regression Parameter Surveys

Additional data from regression testing is shown here.

C	M	Mean fitness	Successful	Mean runtime (s)
0.7	0.05	0.0638	24	6.640
0.7	0.08	0.0165	28	7.260
0.7	0.12	0.0323	24	7.240
0.8	0.05	0.0391	27	7.880
0.8	0.08	0.0410	30	6.960
0.8	0.12	0.0367	31	6.220
0.9	0.05	0.0505	16	11.480
0.9	0.08	0.0293	31	8.720
0.9	0.12	0.0251	32	6.720

Table B.1: Results from a parameter survey for the expression $x^4 + x^3 + x^2 + x$. A steady state approach was used, with $n = 200, g = 200$ and reg5.bnf for the grammar.

C	M	Mean fitness	Successful	Mean runtime (s)
0.7	0.05	0.4239	22	6.500
0.7	0.08	0.2734	27	6.180
0.7	0.12	0.3374	23	6.440
0.8	0.05	0.4434	23	6.800
0.8	0.08	0.3327	25	6.580
0.8	0.12	0.3111	25	7.080
0.9	0.05	0.3562	19	7.700
0.9	0.08	0.3361	23	7.760
0.9	0.12	0.2676	25	7.140

Table B.2: Results from a parameter survey for the expression $x^4 + x^3 + x^2 + x$. A steady state approach was used, with $n = 200, g = 200$ and simppoly.bnf for the grammar.

C	M	Mean fitness	Successful	Mean runtime (s)
0.7	0.05	0.0315	34	2.500
0.7	0.08	0.0384	29	3.100
0.7	0.12	0.0203	37	2.520
0.8	0.05	0.0533	19	4.200
0.8	0.08	0.0356	26	4.320
0.8	0.12	0.0193	36	2.700
0.9	0.05	0.0477	26	3.540
0.9	0.08	0.0169	38	2.360
0.9	0.12	0.0288	40	2.240

Table B.3: Results from a parameter survey for the expression $x^4+x^3+x^2+x$. A generational model was used, with $n = 200, g = 200$ and reg5.bnf for the grammar.

C	M	Mean fitness	Successful	Mean runtime (s)
0.7	0.05	0.4277	23	2.660
0.7	0.08	0.2501	28	2.900
0.7	0.12	0.3023	25	3.660
0.8	0.05	0.4807	20	3.120
0.8	0.08	0.2575	27	2.860
0.8	0.12	0.1842	34	2.820
0.9	0.05	0.2517	30	2.460
0.9	0.08	0.2404	29	2.700
0.9	0.12	0.2351	28	3.720

Table B.4: Results from a parameter survey for the expression $x^4+x^3+x^2+x$. A generational model was used, with $n = 200, g = 200$ and simppoly.bnf for the grammar.

Noise rate	Perfect results (out of 50)	Mean raw fitness	Mean adjusted fitness
0	18	4.22	4.22
0.05	18	5.31	5.25
0.1	14	5.48	4.87
0.2	13	9.83	6.25
0.4	6	21.6	10.20

Table B.5: Results of regression on $Y(3 - X^2 - Y)$ showing the effect of added Gaussian noise

Appendix C

Additional Circuit Data

Additional data on circuit evolution is included here.

Grammar	Perfect solutions (out of 30)	Mean fitness
Free1	1	6.54
LP1	12	9.7×10^{-5}

Table C.1: Performance comparison for different grammars describing a simple low-pass filter

N	G	C	M	Gen Model	Grammar	Mean Fitness	Best Fitness
200	100	0.9	0.1	SSGA	wien2.bnf	0.035	.000001
200	100	0.9	0.1	SSGA	wien3.bnf	0.038	.0001
200	100	0.9	0.1	SSGA	wien4.bnf	0.049	.0001

Table C.2: Selected run results from testing on a Wien bridge circuit. Each row summarizes a set of 30 runs with the given configuration.

Appendix D

Code Listing

The full source code developed in the course of this research is available on the web at http://painfulconsciousness.com/thesisge.tgz.

Vita

Alan Christianson was born in Rapid City, SD in 1980. After graduating from St. Thomas More High School in 1999, he began his undergraduate study at South Dakota School of Mines and Technology. While pursuing his first degree, Alan was inducted into Tau Beta Pi, a national engineering honor society. After earning a Bachelor of Science degree in computer science in 2004, Alan went on to work as a software engineer for Dakota Legal Software, then later took another position doing research and development for Golden West Internet Solutions. He soon returned to South Dakota School of Mines and Technology to pursue a Master of Science degree, also in computer science.