Gene Expression Programming for Symbolic Regression

Asher A. Hensley

1 Introduction

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

2 Genetic Programming

Genetic Programming has its foundations in Genetic Algorithms (GA), which is where most discussions on this subject begin. This is a little confusing because the GP algorithm is also a genetic algorithm because any algorithm based on a population of candidate solutions evolving and adapting to a problem, commonly referred to as a "fitness landscape", can technically be called a genetic algorithm. However, the class of algorithms covered by GAs are a specific type of genetic algorithm. To clarify this, we will refer to all algorithms based on population evolution as evolutionary algorithms, *not* genetic algorithms. The name genetic algorithms will be reserved for the class of algorithms called GAs, which we will not be explicitly covering. Instead we would like to merely point out that GAs were introduced by Holland in 1975 [4], and the interested reader is encourage to review [14].

Genetic programming can be traced back to the work of Koza [15, 16] in which symbolic regression was only one aspect. Genetic programming in general is based on using code snippets to write computer programs to solve a particular problem. Although there are several publications in this area, an excellent treatment of GP theory can be found in [5]. At first this topic may seem challenging, however the concept is actually quite simple and is described by [4] in literally 2 pages. The remainder of this section briefly describes the GP algorithm and sets the stage for the GEP algorithm.

2.1 The Population of Solutions

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

The terminal set is comprised of elements that end a branch on the parse tree. This is usually the independent variables of the data set and/or constants. For example, if we want to discover a function of the form $f(x, y) = z$, then the terminal set would be $T = \{x, y\}$, where the variables x and y represent the "data" we have measured. The variable z is also measured, and the problem is to find the mapping $(x, y) \mapsto z$. Another approach is to assume there is no explicit dependent variable and to try and find the implicit equation $f(x, y, z) = 0$. This has been successfully done by Schmidt in [20,22,23] for variety of physics problems based on pendulums and masses on springs.

2.2 Numerical Constants

Aside from independent variables, constants can also be included in the terminal set. This can either be known constants such as π, or randomly generated constants. This is one of the tricky areas in genetic programming which Koza has called *“a skeleton in the GP closet"* [3]. For example if we are trying to discover a function with a large constant like 1034.59, this can be very difficult without large constants in the terminal set. One common approach is to use the Ephemeral Random Constants technique, which is slightly out of scope here, but the interested reader is welcome to consult [15,16]. There are some shortcomings to this approach which have been addressed in [1,19,24], however the constants problem is still considered open in the GP community [2,3].

2.3 Solution Fitness

Once the population is initialized using the function and terminal sets, each candidate solution is tested against this fitness criterion. For symbolic regression problems this is typically the Mean Square Error (MSE) between the output of the candidate solution and the dependent variable. This provides a score for each member of the population which allows the solutions to be ranked. Naturally, some solutions will be terrible, so it common to employ a survival threshold which solutions must meet otherwise they are removed completely from the population. The algorithm then evolves the remaining solutions using the genetic operators: replication, mutation, crossover, and insertion to make the next generation. The next generation is then scored based on the fitness criteria, a new generation is created, and so on until convergence. The computer’s representation of each candidate solution can vary depending on the given language, but it has been recognized that the LISP language offers a convenient representation [4].

2.4 Shortcomings

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

3 Gene Expression Programming

Gene Expression Programming (GEP) is an extension to Genetic Algorithms (GA) and Genetic Programming (GP) introduced by Candida Ferreira. Several papers have been published in this area [6,8,9,11,12] which have ultimately led to the comprehensive book [13]. The major contribution of GEP is that is it uses fixed length linear chromosomes to encode parse trees of varying size using the Karva language. GAs use fixed length linear chromosomes also, however the resulting parse trees are also a fixed size which seriously impedes their ability to maneuver the search space. As we have seen, GPs vary the parse tree size by manipulating the parse trees directly, however they often result in illegal expressions and still lack the ability the efficiently search the solution space. Additionally, the GP parse trees must be bounded by additional rules otherwise they will grow without bound and will typically generate "bloated" solutions. GEP overcomes these problems by using a fixed length representation which can generate variable length solutions, with the addition of multiple new reproduction operations.

GEP can also be used for many types of problems outside the scope of this study such as decision tree induction, design of neural networks, and combinatorial optimization [7, 10, 13]. Here we will just focus on function discovery via symbolic regression according to our implementation which may slightly differ from Ferreira's implementation. So instead of referring the reader to [13] and only presenting results, the next section is a self contained introduction to the GEP algorithm. The reader is encouraged to consult [13] or [6] if there are any areas that are unclear or if other GEP applications are of interest.

3.1 Evolutionary Process

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

The steps for the GEP algorithm are as follows:

1. Initialize Population
2. Score Solutions Using Fitness Criterion
3. Test For Convergence
4. Replication
5. Mutation
6. 1-Pt Recombination
7. 2-Pt Recombination
8. Gene Recombination
9. Insertion Sequence Transpose
10. Root Insertion Sequence Transpose
11. Inversion
12. Go To Step 2

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

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

3.2 Solution Structure

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

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

where m_a is the maximum arity of the function library. For example, if the head size is 5 and the maximum arity is 2, then the tail size would be 6. The chromosome map for this single gene example with head elements H and tail elements T would then be written as

$$(H, H, H, H, H, T, T, T, T, T, T) \tag{2}$$

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

$$p(H = +) = p(H = -) = \cdots = p(H = a) = p(H = b) = \frac{1}{6} \tag{3}$$

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

$$p(T = a) = p(T = b) = \frac{1}{2} \tag{4}$$

By doing so, every chromosome will always result in a legal expression [6].

3.3 Gene Expression

As an example, assume the following was generated:

$$(H, H, H, H, H, T, T, T, T, T, T) = (+, a, \times, +, b, a, b, a, b, b, a) \tag{5}$$

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

$$(+, a, \times, +, b, a, b, a, b, b, a) \mapsto a + b(a + b) \tag{6}$$

Depending on the arguments on the Karva list, not every element will necessarily be expressed in the parse tree. In this example, the last 4 elements were not expressed because there were not enough empty arguments. However this can change when we start introducing the evolution operators. As we will see, minor changes due to mutation and recombination can act as switches causing previously unexpressed parts of a gene to become active. This is the reason for Equation 1; in the event that every H element is from the function set, there will be enough terminals to satisfy every argument.

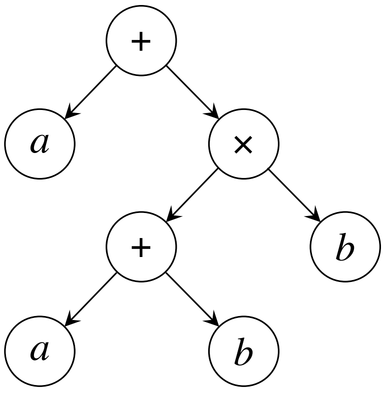


Figure 1: Parse tree representation of $\{+, a, \times, +, b, a, b, a, b, b, a\}$

3.4 Replication

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

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

$$\Phi_{MS}(f) = K\left(1 + \frac{1}{N}\sum_{k=1}^{N}\left(y^{(k)} - f\left(\mathbf{x}^{(k)}\right)\right)^2\right)^{-1} \tag{7}$$

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

3.5 The Founder Effect

Before discussing roulette wheel selection with elitism, we should mention our implementation of the GEP algorithm includes a founder threshold. The founder threshold only applies to the initial population, and is simply the minimum number of candidate solutions that meet the survival threshold (or minimum fitness score). At initialization, the algorithm has no knowledge of the environment and typically gets low fitness scores. This stage is critical in determining the trajectory of the algorithm, and the founder threshold helps guarantee that good candidates will be available in the population, thus allowing the algorithm to begin with good momentum. In [13] Ferreira uses $m = 1$ and argues that "this does not hinder the evolutionary process", however we have left this as a design parameter which can be varied. If the initial population does not meet the founder threshold, new populations are generated until the founder threshold is met.

3.6 Roulette Wheel Selection and Elitism

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

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

$$p(n) = x_n^{fit} \left(\sum_{k=1}^{M} x_k^{fit} \right)^{-1} \tag{8}$$

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

$$\mathbf{g} = P^{-1}(\mathbf{u}) \tag{9}$$

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

$$P(n) = \sum_{k=1}^{n} p(k) \tag{10}$$

This is a standard result in probability theory, a proof of which can be found in any standard text such as [18].

3.7 Mutation

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

$$N_m = round(R_m(M-1)) \tag{11}$$

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

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

$$a + b(a+b) \mapsto \frac{a+b}{b} + ab \tag{12}$$

3.8 Recombination

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

In each case, the number of selected solutions is computed using

$$N_k = 2 \times round(R_k(M-1)), k = 1, 2 \tag{13}$$

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

$$\begin{pmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \end{pmatrix} = \begin{pmatrix} +, a, \times, +, b, a, b, a, b, b, a \\ -, +, a, b, \times, b, b, a, a, a, b \end{pmatrix} \tag{14}$$

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

$$\begin{pmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{P}_{11} & \mathbf{P}_{12} \\ \mathbf{P}_{21} & \mathbf{P}_{22} \end{pmatrix} = \left(\begin{array}{c|c} +, a, \times & +, b, a, b, a, b, b, a \\ -, +, a & b, \times, b, b, a, a, a, b \end{array} \right) \tag{15}$$

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

$$\begin{pmatrix} \mathbf{Q}_1 \\ \mathbf{Q}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{P}_{11} & \mathbf{P}_{22} \\ \mathbf{P}_{21} & \mathbf{P}_{12} \end{pmatrix} = \left(\begin{array}{c|c} +, a, \times & b, \times, b, b, a, a, a, b \\ -, +, a & +, b, a, b, a, b, b, a \end{array} \right) \tag{16}$$

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

$$\begin{pmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{P}_{11} & \mathbf{P}_{12} & \mathbf{P}_{13} \\ \mathbf{P}_{21} & \mathbf{P}_{22} & \mathbf{P}_{23} \end{pmatrix} \tag{17}$$

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

$$\begin{pmatrix} \mathbf{Q}_1 \\ \mathbf{Q}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{P}_{11} & \mathbf{P}_{22} & \mathbf{P}_{13} \\ \mathbf{P}_{21} & \mathbf{P}_{12} & \mathbf{P}_{23} \end{pmatrix} \tag{18}$$

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

3.9 Random Numerical Constants

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

$$(H, H, H, H, H, T, T, T, T, T, T, C, C, C, C, C, C) \tag{19}$$

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

$$(+, \times, ?, ?, b, a, ?, b, b, ?, a, 9, 4, 0, 7, 2, 4) \tag{20}$$

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

$$(+, \times, z_{10}, z_5, b, a, z_1, b, b, z_8, a) \mapsto z_{10} + z_5 b \tag{21}$$

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

3.10 Multiple Genes

For complex problems, single gene systems are typically inadequate so we must at some point turn to multi gene systems. Multi gene systems are quite simply single gene systems linked together by arithmetic operators. These linking operators can be evolvable, however for our implementation we have chosen to use the addition operator $\{+\}$, although it can be easily changed in the configuration script. For example, if we want to extend the template in Equation 19 to a 3 gene system, the chromosome map would be written as:

$$(\mathbf{G}_1, \mathbf{G}_2, \mathbf{G}_3) = (\mathbf{H}_5, \mathbf{T}_6, \mathbf{C}_6, \mathbf{H}_5, \mathbf{T}_6, \mathbf{C}_6, \mathbf{H}_5, \mathbf{T}_6, \mathbf{C}_6) \tag{22}$$

where

$$\mathbf{H}_5 = (H, H, H, H, H) \tag{23}$$

$$\mathbf{T}_6 = (T, T, T, T, T, T) \tag{24}$$

$$\mathbf{C}_6 = (C, C, C, C, C, C) \tag{25}$$

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

3.11 Transpose

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

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

$$(+, \times, -, ?, b, a, ?, \mathbf{b}, \mathbf{b}, ?, a, 9, 4, 0, 7, 2, 4) \tag{26}$$

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

$$(+, \times, \mathbf{b}, \mathbf{b}, -, ?, b, a, ?, \mathbf{b}, \mathbf{b}, ?, a, 9, 4, 0, 7, 2, 4) \tag{27}$$

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

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

$$(+, \times, \mathbf{b}, \mathbf{b}, -, a, ?, \mathbf{b}, \mathbf{b}, ?, a, 9, 4, 0, 7, 2, 4) \tag{28}$$

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

$$b\mathbf{Z}_{10,2} + (a - \mathbf{Z}_{5,2}) \mapsto b + b(a - \mathbf{Z}_{10,2}) \tag{29}$$

3.12 Inversion

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

$$(+, \times, -, ?, b, a, ?, b, b, ?, a, 9, 4, 0, 7, 2, 4) \tag{30}$$

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

$$(-, \times, +, ?, b, a, ?, b, b, ?, a, 9, 4, 0, 7, 2, 4) \tag{31}$$

Observe how the first 3 elements have been flipped from left to right. This implementation differs slightly from that of [13], where the starting point of the inversion sequence can be anywhere in the head of the gene.

3.13 Criticism

As with all evolutionary algorithms, the GEP algorithm has a lot of moving parts which makes it difficult to define its theoretical properties exactly. Finding the optimal model configuration for a given problem is challenging and is considered an open problem in evolutionary computing [3]. The GEP algorithm has been criticized by Oltean in [17] where it is pointed out that the GEP algorithm is more sensitive to the algorithm configuration (head size, number of genes, etc.) than other similar approaches. Additionally the linking operator for multi gene systems tends to constrain the search space, and the use of transcendental functions tend to overcomplicate the search space [17].

4 Experiments

In this section we study the GEP algorithm performance through a series of numerical experiments. The objective is to learn symbolic equations from data. We have done a total of 7 experiments ranging from simple to complex which we will briefly discuss before presenting the results.

4.1 Experiment 1: A Simple Equation

In this experiment we have attempted to learn the following simple equation:

$$f(x) = \frac{1}{x^2 + 1}, \qquad x \in [-10, +10] \tag{32}$$

4.2 Experiment 2: A Simple Equation + Noise

This is a repeat of Experiment 1 with Gaussian noise $\mathbf{n}$:

$$f(x) = \frac{1}{x^2 + 1} + \mathbf{n}, \qquad x \in [-10, +10] \tag{33}$$

4.3 Experiment 3: Unnecessary Constants

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

$$f(x) = \frac{1}{x^2+1} + \mathbf{n}, \qquad x \in [-10, +10] \tag{34}$$

4.4 Experiment 4: Necessary Constants

Here we have repeated Experiment 3 with an actual numerical constant π in the numerator:

$$f(x) = \frac{\pi}{x^2+1} + \mathbf{n}, \qquad x \in [-10, +10] \tag{35}$$

4.5 Experiment 5: Information Removal

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

$$(1) \qquad f(x) = \frac{\pi}{x^2+1} + \mathbf{n}, \qquad x \in [-8, +10] \tag{36}$$

$$(2) \qquad f(x) = \frac{\pi}{x^2+1} + \mathbf{n}, \qquad x \in [-6, +10] \tag{37}$$

$$(3) \qquad f(x) = \frac{\pi}{x^2+1} + \mathbf{n}, \qquad x \in [-4, +10] \tag{38}$$

$$(4) \qquad f(x) = \frac{\pi}{x^2+1} + \mathbf{n}, \qquad x \in [-2, +10] \tag{39}$$

$$(5) \qquad f(x) = \frac{\pi}{x^2+1} + \mathbf{n}, \qquad x \in [-1, +10] \tag{40}$$

$$(6) \qquad f(x) = \frac{\pi}{x^2+1} + \mathbf{n}, \qquad x \in [0, +10] \tag{41}$$

$$(7) \qquad f(x) = \frac{\pi}{x^2+1} + \mathbf{n}, \qquad x \in [1, +10] \tag{42}$$

$$(8) \qquad f(x) = \frac{\pi}{x^2+1} + \mathbf{n}, \qquad x \in [2, +10] \tag{43}$$

$$(9) \qquad f(x) = \frac{\pi}{x^2+1} + \mathbf{n}, \qquad x \in [4, +10] \tag{44}$$

$$(10) \qquad f(x) = \frac{\pi}{x^2+1} + \mathbf{n}, \qquad x \in [6, +10] \tag{45}$$

$$(11) \qquad f(x) = \frac{\pi}{x^2+1} + \mathbf{n}, \qquad x \in [8, +10] \tag{46}$$

4.6 Experiment 6: Simplified Viscosity of Hydrogen

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

$$\mu(T) = 0.636\frac{T^{3/2}}{T+1}, \qquad T \in [-5 \times 10^5, 10 \times 10^6] \tag{47}$$

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

4.7 Experiment 7: Imaginary Viscosity of Hydrogen

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

$$\mu(T) = 0.636 \frac{T^{3/2}}{T+1}, \qquad T \in [-20, +20] \tag{48}$$

5 Experiment 1: A Simple Equation

5.1 Setup

For this first experiment, we have asked the algorithm to learn:

$$f(x) = \frac{1}{x^2+1} \tag{49}$$

based on 200 random x values from the uniform distribution $U(-10, +10)$ as shown in Figure 2 and configured the GEP algorithm as shown in Table 1.

5.2 Results

We ran 10 independent runs, all of which converged to Equation 49 within 20 generations as shown by the learning curves in Figure 3.

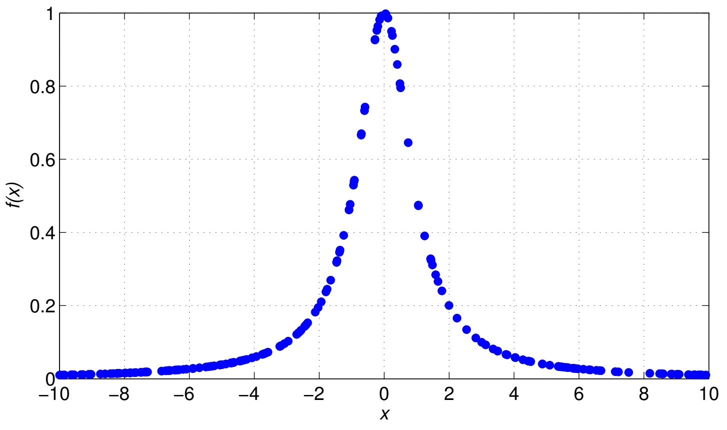


Figure 2: Experiment 1 Data

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

Table 1: Experiment 1 Algorithm Configuration

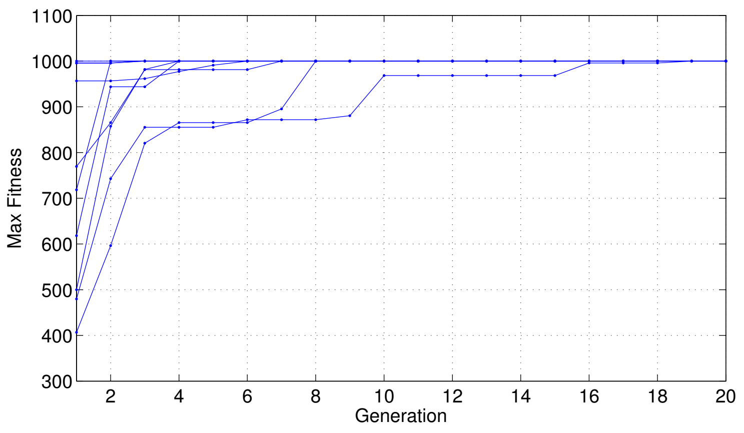


Figure 3: Experiment 1 Learning Curves

6 Experiment 2: A Simple Equation + Noise

6.1 Setup

In this experiment, we have repeated Experiment 1 with noise to study the GEP algorithm sensitivity:

$$f(x) = \frac{1}{x^2+1} + \mathbf{n} \tag{50}$$

where is $\mathbf{n} \sim N(\mu, \sigma)$ (i.i.d.) with $\mu = 0$ and $\sigma = 0.1$. We used the same 200 x values from Experiment 1 as well as the same algorithm parameters (see Table 1) aside from the convergence threshold. As we have seen when noise is present, a perfect fitness score is unreachable, so the convergence threshold τ has been compensated using,

$$\tau = \left\lfloor \frac{K}{1+\mathbb{E}(\mathbf{n}^2)} \right\rfloor \tag{51}$$

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

6.2 Results

We did 10 independent runs using the data shown in Figure 4 and the algorithm again converged to Equation 49 in every case. What came as a surprise was the algorithm converged roughly twice as fast with the addition of noise as shown in Figure 5.

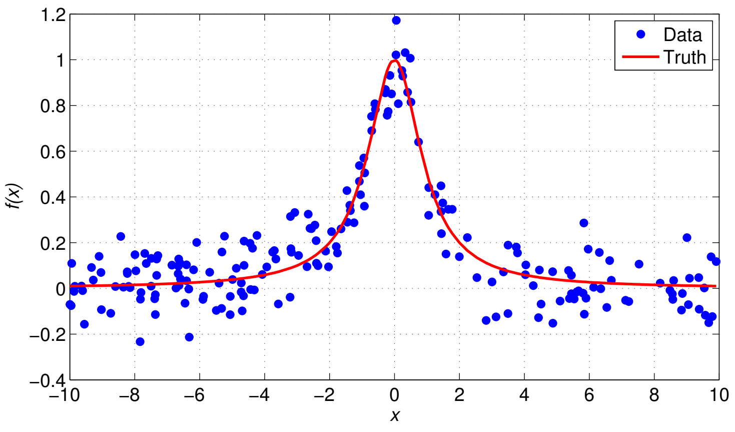


Figure 4: Experiment 2 Data

7 Experiment 4-3: Unnecessary Constants

7.1 Setup

So far we have not used any Random Numerical Constants (RNC) which has substantially reduced the search space. In this experiment we have repeated Experiment 2 with the addition of RNCs even though the formula we are trying to find does not have any numerical constants other than the number 1. The algorithm parameters are the same as those shown in Table 1, however Random Numerical Constants have been set to 'On'.

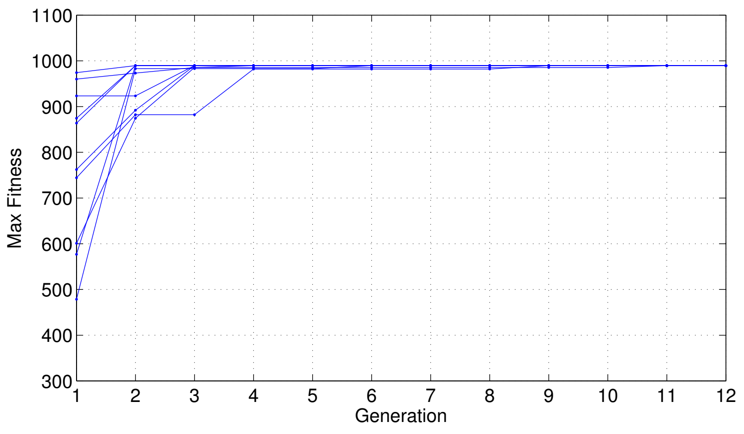


Figure 5: Experiment 2 Learning Curves

7.2 Results

With the addition of RNCs to the configuration, the GEP algorithm had a significantly more difficult time converging. It found the correct solution 2 out of 10 times, while the others were close approximations. Because we have limited the number of generations to 100, it's very likely the algorithm did not have enough time to converge. This argument is supported by the learning curves shown in Figure 6, which did not achieve maximum fitness as fast as Experiments 1 and 2. With additional of RNCs, there appears to be many more local fitness maximums which are giving the algorithm some trouble finding the correct solution.

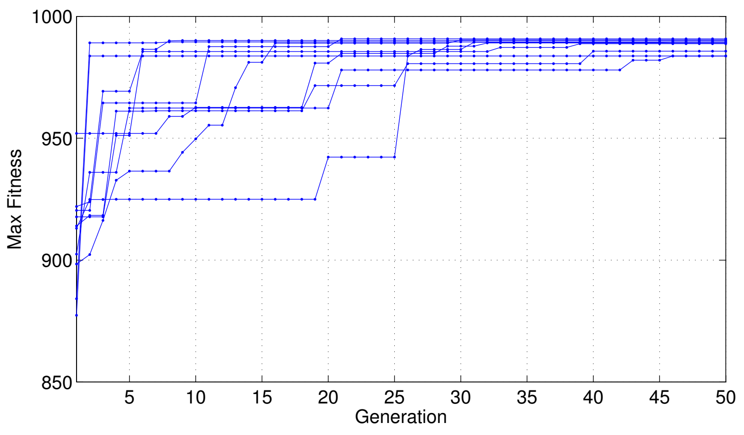


Figure 6: Experiment 3 Learning Curves

7.3 Algorithm Solutions

Recall, the true data generating formula is given by:

$$f(a) = \frac{1}{a^2 + 1} \tag{52}$$

The top 10 solutions returned by GEP algorithm were:

$$f_1(a) = \frac{1.1}{a^2 + 0.16\,a + 1.1} \tag{53}$$

$$f_2(a) = \frac{0.16}{0.16\,a^2 + 0.16} + 0.025 \tag{54}$$

$$f_3(a) = \frac{1.8}{1.8\,a^2 + 1.8} \tag{55}$$

$$f_4(a) = \frac{1.8}{2.1\,a^2 + 1.9} \tag{56}$$

$$f_5(a) = \frac{0.47}{0.22\,a^2 + 0.56} \tag{57}$$

$$f_6(a) = \frac{a}{a^3 + 1.1\,a} \tag{58}$$

$$f_7(a) = \frac{0.86}{1.1\,a^2 - 0.16\,a + 1.1} + 0.063 \tag{59}$$

$$f_8(a) = \frac{0.068}{0.03\,a^2 + 0.093} \tag{60}$$

$$f_9(a) = \frac{0.063}{0.063\,a^2 + 0.063} \tag{61}$$

$$f_{10}(a) = \frac{1.8}{2.6\,a^2 + 1.8} \tag{62}$$

8 Experiment 4: Necessary Constants

8.1 Setup

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

$$f(x) = \frac{\pi}{x^2 + 1} \tag{63}$$

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

8.2 Results

Upon analyzing the learning curves plotted in Figure 7, we can see the convergence behavior is roughly the same as Experiment 3, but by allowing an extra 100 generations, the algorithm was able to fine tune the solutions and reach the correct solution (approximately) on every run. The solutions generated on each run were as follows:

8.3 Algorithm Solutions

$$f_1(a) = \frac{1.9}{0.62\,a^2 + 0.59} \tag{64}$$

$$f_2(a) = \frac{1.9}{0.69\,a^2 + 0.58} \tag{65}$$

$$f_3(a) = \frac{1.9}{0.62\,a^2 + 0.61} \tag{66}$$

$$f_4(a) = \frac{3.1}{0.96\,a^2 + 0.97} \tag{67}$$

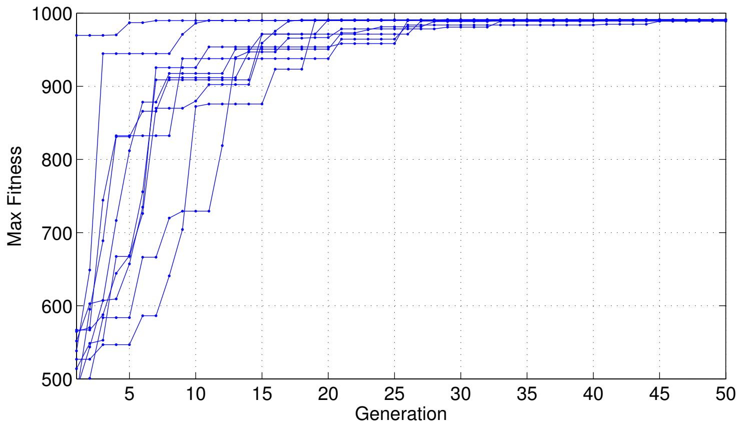


Figure 7: Experiment 4 Learning Curves

$$f_5(a) = \frac{3.1}{0.96\,a^2 + 0.96} \tag{68}$$

$$f_6(a) = \frac{1.9}{0.62\,a^2 + 0.6} \tag{69}$$

$$f_7(a) = \frac{3.2}{a^2 + 1.0} \tag{70}$$

$$f_8(a) = \frac{1.9}{0.62\,a^2 + 0.59} \tag{71}$$

$$f_9(a) = \frac{3.1}{a^2 + 0.97} \tag{72}$$

$$f_{10}(a) = \frac{1.8}{0.57\,a^2 + 0.59} \tag{73}$$

9 Experiment 5: Information Removal

9.1 Setup

So far we have seen the GEP algorithm find the correct solution to a pair of simple equations under a variety of conditions (i.e. with/without noise, with/without RNCs, etc.). This is promising evidence that it is possible for the GEP algorithm to find the true formula of a simple system explicitly. In this experiment, we study the effects of information removal. So far we have randomly sampled the independent variable on the interval of $[-10, +10]$, which for Equation 49 is essentially the entire region where dependent variable is non-zero. It's reasonable to conclude the reason for the algorithm's success is because there are few solutions, aside from the true solution, that adequately fit the data. To study how this process breaks down, we have rerun Experiment 4 eleven times and continuously decreased the independent variable sample interval to determine if there's a correlation with the algorithm's ability to find the correct solution.

9.2 Results

The results for this experiment are shown in Table 2 and Figures 8 through 18. In each case, 200 samples for the independent variable were randomly drawn from $U(x_{min}, x_{max})$. The score for each run was determined by counting the number of solutions of the form:

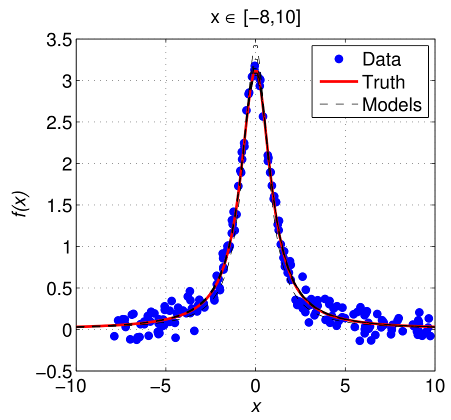
run	x_{min}	x_{max}	score	#solutions
1	-8	$+10$	10/10	1
2	-6	$+10$	9/10	2
3	-4	$+10$	10/10	1
4	-2	$+10$	8/10	3
5	-1	$+10$	8/10	3
6	0	$+10$	5/10	4
7	$+1$	$+10$	2/10	5
8	$+2$	$+10$	0/10	4
9	$+4$	$+10$	0/10	5
10	$+6$	$+10$	0/10	7
11	$+8$	$+10$	0/10	6

Table 2: Experiment 5 Results

$$f(x) = \frac{c_1}{c_2\, x^2 + c_3} \tag{74}$$

at convergence or the 200th generation, whichever came first. The constants c_k were ignored when comparing solutions. We have assumed that if the solution is of the correct form, the true formula would have been found eventually.

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



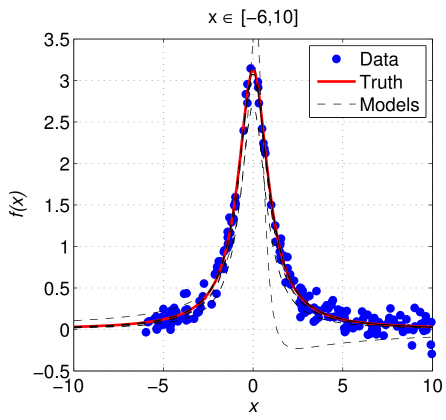
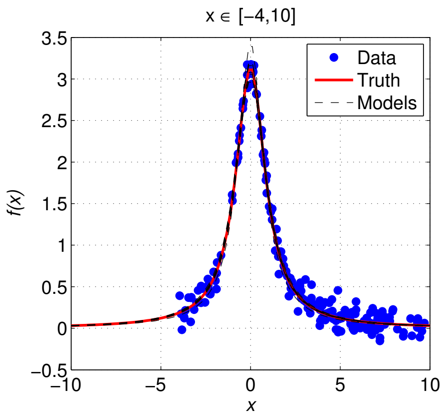


Figure 8: Experiment 5 Results from run 1 (left)

Figure 9: Experiment 5 Results from run 2 (right)



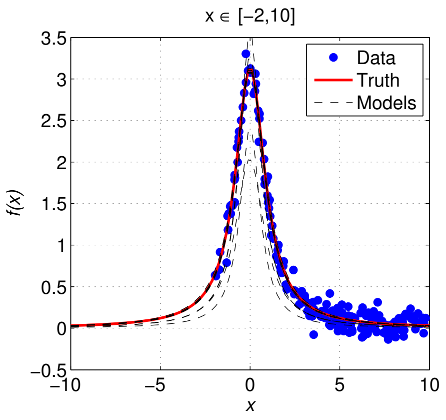
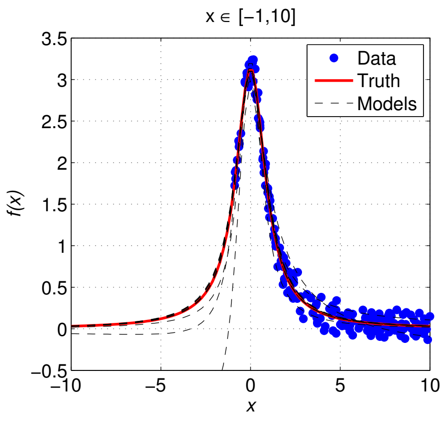


Figure 10: Experiment 5 Results from run 3 (left)

Figure 11: Experiment 5 Results from run 4 (right)



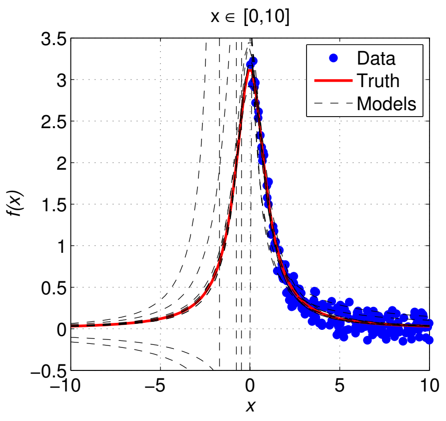
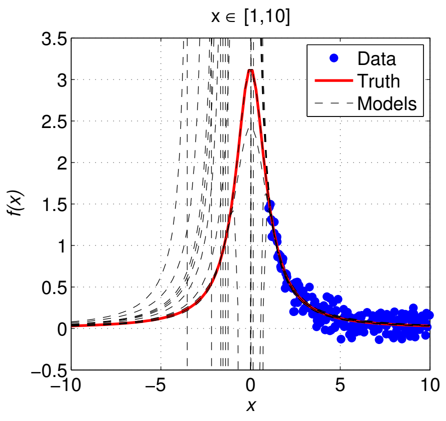


Figure 12: Experiment 5 Results from run 5 (left)

Figure 13: Experiment 5 Results from run 6 (right)



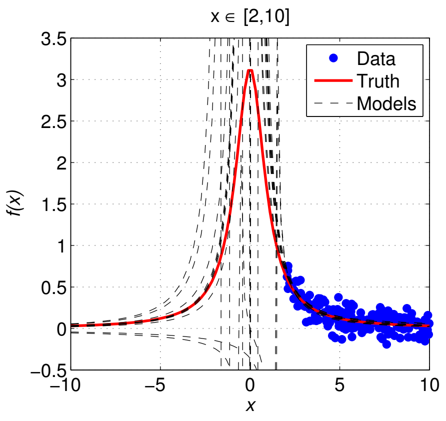
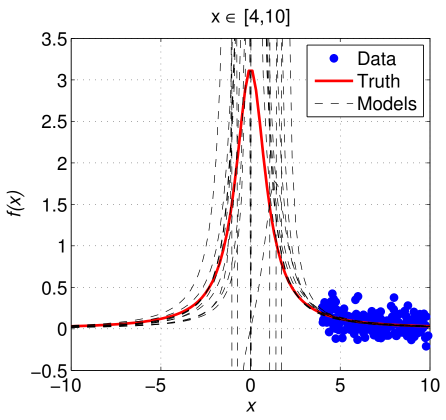


Figure 14: Experiment 5 Results from run 7 (left)

Figure 15: Experiment 5 Results from runs 8 (right)



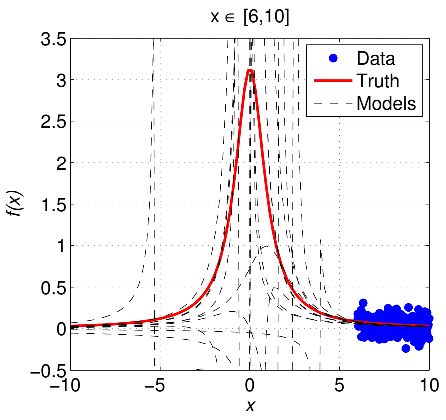


Figure 16: Experiment 5 Results from runs 9 (left)

Figure 17: Experiment 5 Results from runs 10 (right)

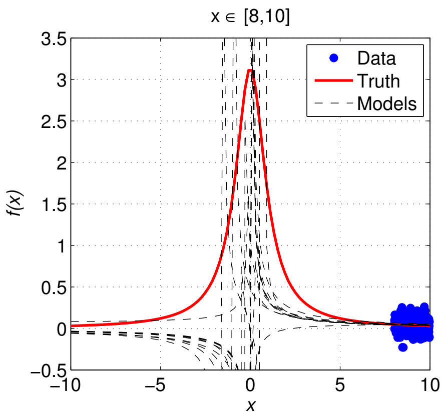


Figure 18: Experiment 5 Results from run 11

10 Experiment 6: Simplified Viscosity of Hydrogen

10.1 Setup

In this experiment we attempted to learn the Hydrogen Viscosity equation by sampling on the interval $[-5 \times 10^5, 10 \times 10^6]$ degrees Kelvin. Instead of random sampling we have taken 200 linearly spaced samples with noise $\sigma = 0.01$ to simplify the problem (see Figure 19). Recall, the hydrogen viscosity equation is written as,

$$\mu(T) = \lambda \frac{T^{3/2}}{T + C} \tag{75}$$

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

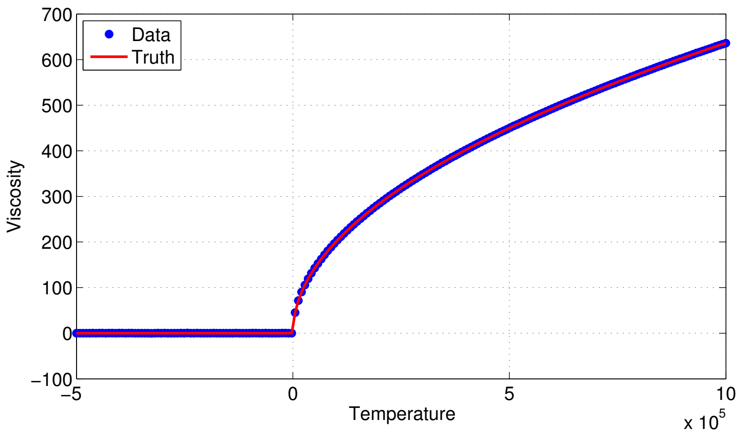


Figure 19: Experiment 6 Data

Note that this is the real part of the training data which is zero for negative temperatures. The algorithm parameters are shown in Table 3. We have added the function $\Re(\sqrt{\star})$ to the function set to avoid complex outputs.

10.2 Results

Unfortunately the algorithm failed to find the correct solution on every run. The learning curves are shown in Figure 20 followed by the algorithm solutions, sorted in order of fitness. The convergence behavior here was not consistent which suggests the training data does have the correct features for the algorithm to succeed. Upon examining the most fit solution, $f_1(a)$ we found that it actually exceeded the convergence threshold, meaning

$$\mathbb{E}(\mathbf{n}^2) \approx \frac{1}{M} \sum_{k=1}^{M} \|f_1(T^{(k)}) - \mu(T^{(k)})\|^2 \tag{76}$$

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

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

Table 3: Experiment 6 Algorithm Configuration

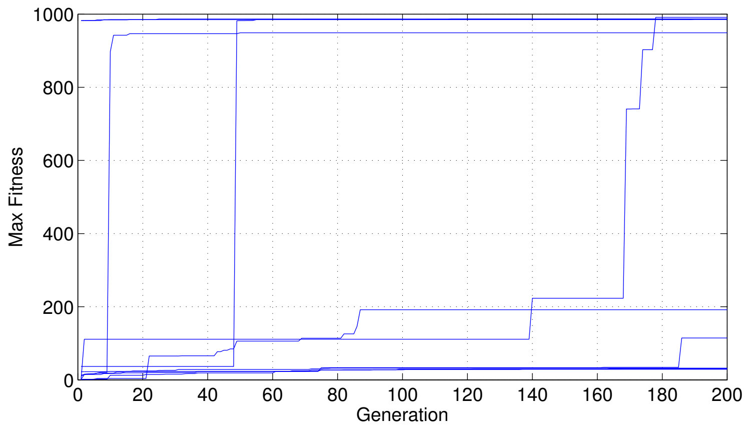


Figure 20: Experiment 6 Learning Curves

10.3 Algorithm Solutions

$$f_1(a) = 0.64\, a\, \Re\left(\sqrt{\frac{1}{a}}\right) \tag{77}$$

$$f_2(a) = \Re\big(\sqrt{0.4\, a - 1.7}\big) - 0.049 \tag{78}$$

$$f_3(a) = 0.65\, \Re\big(\sqrt{0.95\, a - 2.3}\big) \tag{79}$$

$$f_4(a) = \Re\big(\sqrt{0.4\, a - 1.4}\big) - 0.024 \tag{80}$$

$$f_5(a) = \Re\big(\sqrt{0.41\, a - 2.8}\big) - 0.27 \tag{81}$$

$$f_6(a) = 0.64\, \Re\big(\sqrt{a}\big) - 2.5 \tag{82}$$

$$f_7(a) = 0.64\,\Re\big(\sqrt{a}\big) + 0.024 \tag{83}$$

$$f_8(a) = 0.65\,\Re\left(\sqrt{a - 1.0\,\Re\big(\sqrt{a}\big) + 0.024}\right) - 6.6 \tag{84}$$

$$f_9(a) = 0.65\,\Re\big(\sqrt{a}\big) - 6.8 \tag{85}$$

$$f_{10}(a) = 0.65\,\Re\big(\sqrt{a}\big) - 5.3 \tag{86}$$

11 Experiment 7: Imaginary Viscosity of Hydrogen

11.1 Setup

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

$$\Phi_{MS}(f) = K\left(1 + \frac{1}{M}\sum_{k=1}^{M}(\Re(e^{(k)})^2 + \Im(e^{(k)})^2)\right)^{-1} \tag{87}$$

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

$$\tau = \left\lfloor \frac{K}{1 + \mathbb{E}(\mathbf{n}_R^2) + \mathbb{E}(\mathbf{n}_I^2)} \right\rfloor = 980 \tag{88}$$

11.2 Results

The resulting solutions are shown in Figures 21 through 22[2] and the learning curves are shown in Figure 23, followed the solutions for each run sorted by fitness. Out of the 10 runs we did, $f_1(a)$, $f_2(a)$, and $f_4(a)$ found the true solution (approximately). On the other hand $f_3(a)$ was not the correct solution form, but was able to score a higher fitness than $f_4(a)$. The pole at $T = 1$ has clearly provided a unique feature the algorithm can use to find the correct solution.

11.3 Algorithm Solutions

$$f_1(a) = \frac{\sqrt{a}}{\frac{1.5}{a} + 1.5} \tag{89}$$

$$f_2(a) = \frac{\sqrt{a}}{\frac{1.5}{a} + 1.5} \tag{90}$$

$$f_3(a) = 0.62\sqrt{\frac{a^2}{a + 1.1}} \tag{91}$$

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

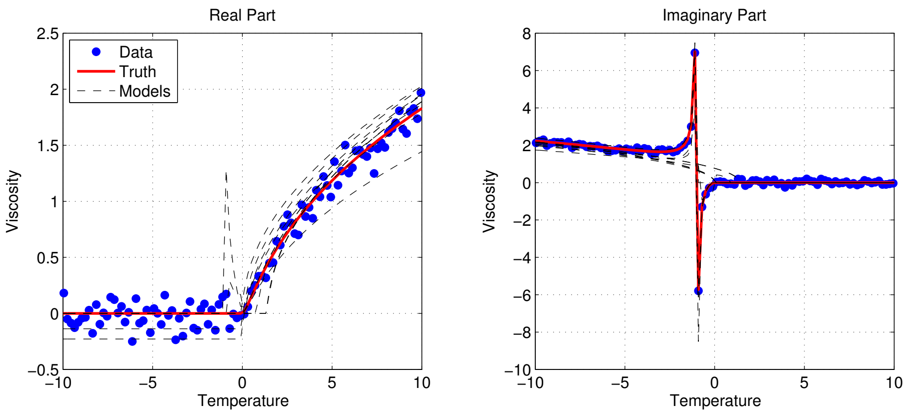


Figure 21: Experiment 7 Results (real part)

Figure 22: Experiment 7 Results (imaginary part)

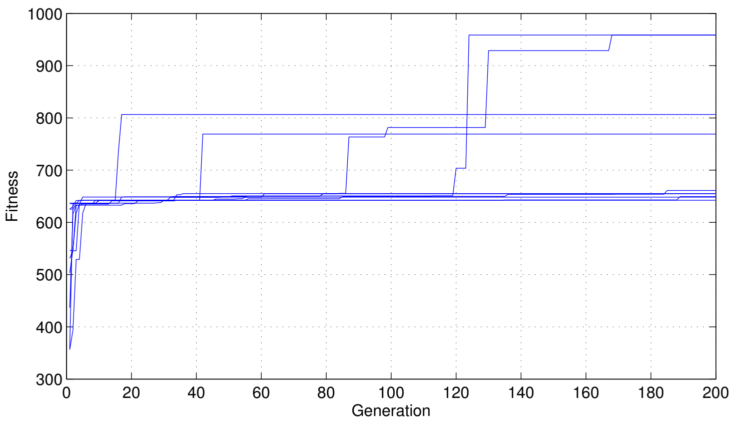


Figure 23: Experiment 7 Learning Curves

$$f_4(a) = \frac{a^{\frac{3}{2}}}{2.0\,a + 1.9} \tag{92}$$

$$f_5(a) = \sqrt{0.35\,a - 0.35\left(\frac{a}{a + 0.74}\right)^{\frac{1}{4}}} \tag{93}$$

$$f_6(a) = \sqrt{0.42\,a - 0.58} \tag{94}$$

$$f_7(a) = \sqrt{0.42\,a - 0.59} \tag{95}$$

$$f_8(a) = 0.68\,\sqrt{a} - 0.14 \tag{96}$$

$$f_9(a) = 0.045\,\sqrt{244.0\,a} - 0.23 \tag{97}$$

$$f_10(a) = 0.059\,\sqrt{122.0\,a} \tag{98}$$

12 Conclusion

In these experiments we discovered that

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

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

At this point, the question is: can additional information be extracted from a data set which lacks the correct features? Schmidt used partial derivative ratios for implicit models of the form $f(\mathbf{x}, y) = 0$ as way to detect trivial solutions (i.e. $\mathbf{x} - \mathbf{x} = 0$) in dynamical systems [20, 22, 23] which, in a sense, guided the search down the correct path. Here we have the problem of detecting decoy solutions, which approximate the true solution very closely, but are incorrect. While this seems like this would be a problem with the data set, it may be possible to constrain the search space to penalize decoys.

For all the experiments in this chapter we have been using single gene systems with a head size of 10 (tail size of 11), and function sets with the correct operators. By doing this, we have already constrained the search space. This configuration was not chosen to fit the problem, but it's possible we were lucky for the simple equation we chose. Selecting the optimal algorithm configuration is a common problem in evolutionary computing, which has also been addressed by Schmidt in [21] for the problem of domain alphabet learning.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[19] C. Ryan and M. Keijzer. “An analysis of diversity of constants of genetic programming,". *In Genetic Programming, Proceedings of EuroGPO2003 (Essex, 14-16 Apr. 2003), C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, Eds., vol. 2610 of LNCS, Springer*, 2003.

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

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

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

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

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