AUTOMATED DISCOVERY OF NUMERICAL APPROXIMATION FORMULAE VIA GENETIC PROGRAMMING

by

Matthew J. Streeter

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

_____

May 2001

APPROVED:

_____

Dr. Lee A. Becker, Major Advisor

_____

Dr. Micha Hofri, Head of Department

Abstract

This thesis describes the use of genetic programming to automate the discovery of numerical approximation formulae. Results are presented involving rediscovery of known approximations for Harmonic numbers and discovery of rational polynomial approximations for functions of one or more variables, the latter of which are compared to Padé approximations obtained through a symbolic mathematics package. For functions of a single variable, it is shown that evolved solutions can be considered superior to Padé approximations, which represent a powerful technique from numerical analysis, given certain tradeoffs between approximation cost and accuracy, while for functions of more than one variable, we are able to evolve rational polynomial approximations where no Padé approximation can be computed. Furthermore, it is shown that evolved approximations can be iteratively improved through the evolution of approximations to their error function. Based on these results, we consider genetic programming to be a powerful and effective technique for the automated discovery of numerical approximation formulae.

ACKNOWLEDGEMENTS

The author wishes to thank Prof. Lee Becker and Prof. Micha Hofri of Worcester Polytechnic Institute for valuable advice and feedback received during the course of this project.

CONTENTS

LIST OF TABLES AND FIGURES

1 INTRODUCTION

Numerical approximation formulae are useful in two primary areas: firstly, approximation formulae are used in industrial applications in a wide variety of domains to reduce the amount of time required to compute a function to a certain degree of accuracy (Burden and Faires 1997), and secondly, approximations are used to facilitate the simplification and transformation of expressions in formal mathematics. The discovery of approximations used for the latter purpose typically requires human intuition and insight, while approximations used for the former purpose tend to be polynomials or rational polynomials obtained by a technique from numerical analysis such as Padé approximants (Baker 1975; Bender and Orszag 1978) or Taylor series. Genetic programming (Koza 1989, Koza 1990a, Koza 1992) provides a unified approach to the discovery of approximation formulae which, in addition to having the obvious benefit of automation, provides a power and flexibility that potentially allows for the evolution of approximations superior to those obtained using existing techniques from numerical analysis. In this thesis, we discuss a number of experiments and techniques which demonstrate the ability of genetic programming to successfully discover numerical approximation formulae, and provide a thorough comparison of these techniques with traditional methods.

1.1 INTRODUCTION TO GENETIC ALGORITHMS

Genetic algorithms represent a search technique in which simulated evolution is performed on a population of entities or objects, with the goal of ultimately producing an individual or instance which satisfies some specified criterion. Specifically, genetic algorithms operate on a population of individuals, usually represented as bit strings, and apply selection, recombination, and mutation operators to evolve an individual with maximal fitness, where "fitness" is measured in some domain-dependent fashion as the extent to which a given individual represents a solution to the problem at hand. Genetic algorithms are a powerful and practical method of search, supported both by successful real-world applications and a solid theoretical foundation. Genetic algorithms have been applied to a wide variety of problems in many domains, including problems involving robotic motion planning (Eldershaw and Cameron 1999), modelling of spatial interactions (Diplock 1996), optimization of database queries (Gregory 1998), and optimization of control systems (Desjarlais 1999).

As an example application for genetic algorithms, suppose we wished to find a real value x which satisfies the quadratic equation $x^2 + 2x + 1 = 0$, and did not have access to the quadratic formula or any applicable technique from numerical analysis. We might choose to represent possible solutions to this equation as a set or population (initially generated at random) of single-precision IEEE floating point numbers, each of which requires 32 bits of storage. The fitness measure for an individual with x-value x_I could be defined as the squared difference between $x_I^2 + 2x_I + 1$ and the target value of 0. As in nature, the more fit individuals of the population will be more likely to reproduce and will tend to have a larger number of children. A "child" can be produced from two IEEE floating point numbers by generating a random bit position *n* (0<=n<=31), copying the leftmost *n* bits from one parent, and taking the remaining 32-*n* bits from the other, a process referred to as *single-point crossover*. As the process continues over many generations, it is likely that an individual will eventaully be evolved which satisfies the equation exactly (in this case, an individual with x-value x_I=-1).

This search method turns out to be highly flexible and efficient. In general, any search problem for which an appropriate representation and fitness measure can be defined may be attempted by a genetic algorithm.

In *Adaptation in Natural and Artificial Systems*, John Holland laid the foundation for genetic algorithms. The "Holland GA" employs fitness-proportionate reproduction, crossover, and (possibly) mutation. Pseudo code for this algorithm is given below:

1. Initialize a population of randomly created individuals

2. Until an individual is evolved whose fitness meets some pre-established criterion:

2.1. Assign each individual in the population a fitness, based on some domain-specific fitness function.

2.2. Set the "child population" to the empty set.

2.3. Until the size of the child population equals that of the parent population:

2.3.1. Select two members of the parent population, with the probability of a member being selected being proportionate to its fitness (the same member may be selected twice).

2.3.2. Breed these two members using a crossover operation to produce a child.

2.3.3. (Possibly) mutate the child, according to some pre-specified probability.

2.3.4. Add the new child to the child population.

2.4. Replace the parent population with the child population.

The theoretical underpinnings of this algorithm are given in the Schema theorem (Holland 1975), which establishes the near mathematical optimality of this algorithm under certain circumstances.

A schema is a set of bit-strings defined by a string of characters, with one character corresponding to each bit. The characters may be '1', indicating that a `1` must appear in the corresponding bit-position, '0', indicating that a `0` must appear in the corresponding bit position, or '*', indicating that either value may appear.

For example, the an individual encoded by the bit string:

`10101101000010011110101100011111`

would be an instance of the schemata `1*****************************`, `*01*****************************`, and `***0*********************11111`, but not of the schema `0*****************************`. Since we can create a schema of which an individual is an instance using two possible characters for each bit position ('*' and the character which actually occurs in that bit position), each individual will be a member of 2^{32} different schemata. Following the definition of the fitness of an individual, the fitness of a schema can be defined as the average fitness of all individuals which are instances of that schema. For example, the schema `1*...*` (1 followed by 31 *'s), which in our representation denotes the set of all negative IEEE floating-point numbers, might be expected to have a different fitness than the schema `0*...*`, which denotes the set of all positive numbers.

The Schema theorem establishes that the straightforward operation of fitness-proportionate reproduction, as employed in the genetic algorithm given above, causes the number of instances of a particular schema that are present in a population to grow (and shrink) at a rate proportional to the schema's fitness, which is "mathematically near-optimal when the process is viewed as a set of multi-armed slot machine problems requiring an optimal allocation of trials" (Koza 1989). Thus, despite their superficial appearance as simply a heuristically guided multiple hill-climbing or "beam search", genetic algorithms implicitly process a great deal of information concerning similar individuals not actually present in the evolving population, a phenomenon referred to as the *implicit parallelism* of genetic algorithms.

1.2 INTRODUCTION TO GENETIC PROGRAMMING

John Koza, in his seminal paper "Hierarchical Genetic Algorithms Operating on Populations of Computer Programs" (Koza 1989), made note of the fact that in Holland's work and in many of his students', chromosomes are defined as fixed-length bit-strings. While bit strings are an adequate representation for many applications, they are not particularly suited to representing computer programs in a manner that is robust and amenable to the crossover operation. In his 1989 paper, Koza presented the possibility of representing programs evolved under a GA as parse trees, or more specifically as symbolic expressions in the LISP programming language ("LISP S-expressions"), and performing crossover by exchanging subtrees.

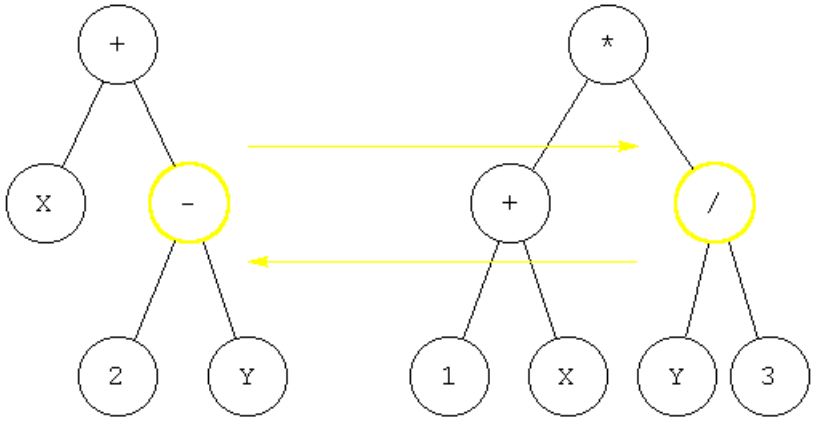

Figure 1.1: Parental LISP Expressions.

Figure 1.1 illustrates two parent LISP expression trees, corresponding to the LISP expressions `(+ X (- 2 Y))` and `(* (+ 1 X) (/ Y 3))`, respectively, which correspond to the mathematical expressions $x+2-y$ and $x+y/3$, respectively. Crossover has been performed at the highlighted points with the left expression tree taken as the "base" parent and the right tree as the "contributing" parent to produce the child LISP expression `(+ X (/ Y 3))` (which corresponds to the mathematical expression $x+y/3$), as illustrated in Figure 1.2.

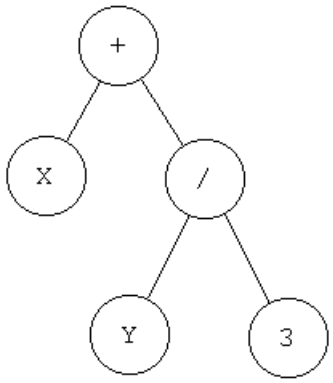

Figure 1.2: Child LISP Expression.

Associated with these trees is a *function set* which specifies the primitive operators from which the trees may be composed (in this case, {*,+,/,-}) and a *terminal set* which specifies the elements that may appear as leaf nodes in the tree (in this case the variables X and Y, and some number of available integers). The programs evolved in the course of a run of genetic programming can be simply arithmetic expressions, as in the example here, Turing-complete programs (evolved by including appropriate looping, conditional, and memory I/O operators in the function set), or programs of an entirely different nature (such as instructions to grow an electrical circuit from a given circuit embryo). The fitness function employed in genetic programming is some measure of the extent to which a given program successfully solves the problem at hand.

As an example genetic programming application, consider the problem of *symbolic regression* or function identification, where, given a set of data points, one wishes to find a function in symbolic form which most accurately models the data. As an example, given a set of pairs (x,y) taken from the function curve f(x) = x^4 + x^3 + x^2 + x, we might attempt to rediscover, via genetic programming, the function (in symbolic form) which generated the points. An appropriate function set for this problem might consist of the arithmetic operators {*,+,/,-}, while an appropriate terminal set could consist simply of the single independent variable {X}. Fitness could be defined as the squared difference between the output of an individual expression for a given x-value and the desired y-value given on the curve, averaged over the points in the available training data. To discover a more complex function with real-valued coefficients, for example f(x) = 3.14159*x^4 + x^3 + x^2 + x, we could make use of the terminal set {X, *R*}, where *R* is the so-called "random numeric terminal" which takes on a specific random value in a pre-specified range when first inserted into an individual expression in the initial population. Specifically, a set of "ephemeral random constants" are created through instantiations of *R* in the initial population, which can be then combined in arithmetic-performing subtrees to generate constants of (practically) arbitrary value.

Two major parameters: the population size and number of generations to run, and five minor parameters are involved in initiating a run of genetic programming. These parameters and their meanings are given in Table 1.1.

Table 1.1: GP Parameters.

PARAMETER	MEANING
Population size	Number of individuals in each generation
Num. generations	Number of generations to run (including initial random generation)
Crossover fraction	Fraction of population which is reproduced using crossover (with parents selected in proportion to fitness)
Fitness proportionate reproduction fraction	Fraction of population which is reproduced using fitness-proportionate reproduction (cloning individuals selected in proportion to fitness)
Probability of internal crossover point selection	Probability that subtree swap points will be selected as internal (rather than leaf) nodes
Max. depth of individuals created by crossover	Maximum depth of trees created by crossover operation
Max. depth of randomly generated individuals	Maximum depth of randomly generated trees (in generation 0)

Genetic programming has been successfully applied to problems in such diverse areas as the design of minimal sorting networks (Koza 1999), automatic parallelization of programs (Walsh and Ryan 1996), image analysis (Howard and Roberts 1999), and design of complex structures such as Lindenmayer systems (Jacob 1996), cellular automata (Andre, Bennett, and Koza 1996), industrial controllers (Koza, Keane, Bennett, Yu, Mydlowec, and Stiffelman 1999), wire antannae (Comisky, Yu, and Koza 2000), and electrical circuits (Koza 1999). The Schema Theorem can be generalized to apply to GP trees as well as GA bit-strings, as has been done by (O'Reilly and Oppacher 1994).

1.3 EVALUATING APPROXIMATIONS

In formal mathematics, the utility or value of a particular approximation formula is difficult to analytically define, and depends perhaps on its syntactic simplicity, as well as the commonality or importance of the function it approximates. In industrial applications, in contrast, the value of an approximation is uniquely a function of the computational cost involved in calculating the approximation and the approximation's associated error. In the context of a specific domain, one can imagine a utility function which assigns value to an approximation based on its error and cost. We define a reasonable utility function to be one which always assigns lower (better) scores to an approximation a_1 which is *unequivocally superior* to an approximation a_2, where a_1 is defined to be unequivocally superior to a_2 iff. neither its cost nor error is greater than that of a_2, and at least one of these two quantities is lower than the corresponding quantity of a_2. Given a set of approximations for a given function (obtained through any number of approximation techniques), one is potentially interested in any approximation which is not *unequivocally inferior* (defined in the natural way) to any other approximation in the set. In the terminology of multi-objective optimization, this subset is referred to as a Pareto front (Goldberg 1989). As an example, if we were given the following set of points in the (*error*,*cost*) plane:

$$\{(0.1,10),\ (0.3,7),\ (0.6,8),\ (1.5,1),\ (2.0,2)\}$$

the Pareto front would be:

$$\{(0.1,10),\ (0.3,7),\ (1.5,1)\}$$

since the point (0.6,8) is dominated by (or is unequivocally inferior to) the point (0.3,7), and the point (2.0,2) is dominated by the point (1.5,1), while the remaining points (0.1,10), (0.3,7) and (1.5,1) are not dominated by any other points in the set.

Thus, the Pareto front contains the set of approximations which could be considered to be the most valuable under some reasonable utility function.

1.4 USING GENETIC PROGRAMMING TO DISCOVER NUMERICAL APPROXIMATION FORMULAE

The problem of functional approximation using genetic programming differs from the problem of symbolic regression in only two respects. First, it is desirable in evolving an approximation to not only obtain a function which models a set of data points accurately, but which also does so cheaply, i.e. a function which can be computed in a relatively small amount of time. Such a solution could be obtained either by limiting the available function set in such a way that the search space contains only approximations to the target function (f.e. evolving an approximation to the function ln(x) using the function set {*,+,/,-}), or by somehow incorporating the cost of an expression into the fitness function, so that the evolutionary process is guided toward simpler expressions which presumably will only be able to approximate the data; the latter approach would be similar to work involving the "minimum description length principle" (Lam 1998), "parsimony pressure" (Soule 1998), and "Occam's Razor" (Zhang 1995). Secondly, it is desirable in evolving approximations for the system to return not simply a single most-accurate approximation, but a set of approximations exhibiting various trade-offs between error and cost. This is simply a matter of bookkeeping, and can be accomplished in a natural way by returning the Pareto front of the entire population history, maintained and updated iteratively as the population evolves.

1.5 RELATED WORK

The problem of function approximation is closely related to the problem of function identification or symbolic regression, which has been extensively studied by numerous sources including (Koza 1992; Andre and Koza 1996; Chellapilla 1997; Luke and Spector 1997; Nordin 1997; Ryan, Collins, and O'Neill 1998). Notably, the economics exchange equation (M=PQ/V) (Koza 1990b) and Kepler's third law (Koza 1990a) have been rediscovered from empirical data through GP symbolic regression. Approximation of specific functions has been performed by (Keane, Koza, and Rice 1993), who use genetic programming to find an approximation to the impulse response function for a linear time-invariant system, and by (Blickle and Thiele 1995), who derive three analytic approximation formulae for functions concerning performance of various selection schemes in genetic programming. Regarding general techniques for the approximation of arbitrary functions, (Moustafa, De Jong, and Wegman 1999) use a genetic algorithm to evolve locations of mesh points for Lagrange interpolating polynomials.

1.6 SUMMARY OF REPORT

This report consists of a description of the GP system used in the experiments presented in this report (Section 2), a description of an unsuccessful attempt to optimize the parameters used for these experiments (Section 3), a description of a number experiments involving automated discovery or attempted discovery of numerical approximation formulae

through genetic programming (Sections 4 - 9), an outline of possible future work and extensions to this thesis (Section 10), and a final summary and conclusions (Section 11). The experimental section consists of three successful experiments and two unsuccessful or only partially successful experiments. Section 4 presents an experiment involving rediscovery of the first three terms, or variations thereupon, of the asymptotic expansion for the Harmonic number series. Section 5 presents experiments involving the evolution of rational polynomial approximations to the common functions ln(x), sqrt(x), arcsinh(x), exp(-x), and tanh(x) which, given certain trade-offs between cost and error, are superior to Padé approximations. Section 6 presents an experiment involving the successful evolution of accurate rational polynomial approximations to a two-dimensional surface defined by a function of more than one variable to which the Padé approximation technique cannot be applied. Section 7 presents successful experiments involving refinement of evolved approximations through approximation of their error function, then discusses other ways in which approximations could be refined using the genetic programming technique. Section 8 presents a partially successful experiment involving rediscovery of neural network activation functions. Section 9 describes an attempt at piecewise approximation of functions of one or two variables, which meets with only limited success.

2 OUR GENETIC PROGRAMMING SYSTEM

The genetic programming system used for all experiments described in this thesis was written by the author in C++, and, excluding module tests and utilities, comprises approximately 11,000 lines of code. The code is made up of essentially two subsystems: a general "GA framework" suitable for evolving any entity under a genetic algorithm, and a set of classes representing the specific individuals being evolved: namely, mathematical expressions represented as program trees. A brief discussion of system architecture, a discussion of the assignment of costs to primitive functions, illustration of the system's output, and a discussion of the consistency of this system with other GP systems are given in this section. Full specification of available command-line options and parameter settings is given in Appendix B.

2.1 GA FRAMEWORK

The code used in this system is designed around a general GA framework. In an approach somewhat similar to that of GAlib (Wall 2000), abstract classes are provided for individuals and populations, and problem-specific subclasses of these are created for specific applications. The nature of the actual entity being evolved is transparent to both the genetic algorithm and to any higher-level operators which regulate the course of a run. The framework is designed to be highly flexible and extensible for use in a variety of applications. In addition to the genetic programming experiments described in this thesis, this GA framework has been used to evolve linear equations (in a module test) and rules for rule-based learning (in a machine learning course).

2.2 GP REPRESENTATION

Individual programs in this system are represented as a tree of dynamically allocated nodes, with each node either an input, a primitive function, or a real-valued constant. Facilities are provided to generate, breed, and evaluate these trees. Fitness is determined by comparing the output of an individual program with the correct output, as specified in an external file containing a number of *training samples*, or sets of inputs along with their associated outputs. This system makes use of 23 primitive functions explicitly coded into the program; these functions and their meanings are given in Table 2.1. For somewhat historical reasons (see Appendix B), each function was coded to be capable of receiving any number of parameters as input, though this ability is not exploited in the experiments described here.

Table 2.1: Primitive Functions.

PRIMITIVE FUNCTION	MEANING
`Sigmoid`	Sigmoid (1/(1+exp(-x))) of sum of inputs (standard neural network activation function)
`Product (*)`	Product of inputs
`ReciprocalProduct`	Reciprocal of product of inputs, or 10^6 if product of inputs is 0
`ReciprocalSum`	Reciprocal of sum of inputs, or 10^6 if sum of inputs is 0
`Cos`	Cosine of sum of inputs
`Ln`	Natural logarithm of sum of inputs, or 0 if sum of inputs is non-positive
`Sqrt`	Square root of sum of inputs, or 0 if sum of inputs is negative

`Sum (+)`	Sum of inputs
`TanH`	Hyperbolic tangent of sum of inputs
`Max`	Maximum input
`Min`	Minimum input
`Sign`	Sign of sum of inputs (-1 for negative, 1 for non-negative)
`Abs`	Absolute value of sum of inputs
`Sin`	Sine of sum of inputs
`Exp`	Exponential (e^x) of sum of inputs
`Div (/)`	1st input divided by 2nd input . . . divided by last input, or 10^6 if any input other than the 1st is 0
`Subtract (-)`	1st input minus 2nd input . . . minus last input
`GreaterThan`	1 if 1st input is > 2nd input, 0 otherwise
`GreaterThanOrEqual`	1 if 1st input is >= 2nd input, 0 otherwise
`LessThan`	1 if 1st input is < 2nd input, 0 otherwise
`LessThanOrEqual`	1 if 1st input is <= 2nd input, 0 otherwise
`IfThenElse`	If 1st input is non-zero, returns 2nd input; returns 3rd input otherwise
`Split`	If 1st input is non-negative, returns 2nd input; returns 3rd input otherwise

2.3 PRIMITIVE FUNCTION COSTS

In order to compute the Pareto front for the entire population history of a run or set of runs (as described in section 1.4), it is necessary to assign each primitive function a cost, so that the total cost of an expression can be computed based upon the primitive functions it employs. We will take the total cost of an expression to be the sum of the costs of all primitive functions used within its nodes, excluding nodes subtrees which involve only constant expressions, which we consider to have zero cost. As examples, the cost of the expression `(+ 3 5)` under this procedure would be 0, the cost of the expression `(* (+ 3 5) X)` would be equal to the cost of the '`*`' operator, and the cost of the expression `(* (+ 3 X) X)` would be equal to the cost of the '`*`' operator plus the cost of the '`+`' operator.

Initially, an attempt was made to assign costs to primitive functions based on hardware timing data, i.e. for the cost of a primitive function to be proportional to the CPU time needed to compute that function, on the average case. Toward this end, a timing procedure was written which sat in a tight loop repeatedly polling the clock, and counted the number of times a given function could be executed during a single clock tick (equal to 1/1000 of a second on the system on which the test was run). This was repeated for every primitive function, and the results averaged over many clock ticks to give an indication of the relative time requirements of each primitive function. This procedure was repeated for various numbers of input arguments to the functions (various arities), so that a series of points were generated for each function in the (*arity*, *time*) plane, where time is represented in clock ticks. A linear least-squares fit was calculated for each of these sets of points, generating coefficients M and B for each function such that:

$$[\textit{function's execution time}] \approx \mathbf{M}*[\textit{number of input arguments}] + \mathbf{B}$$

This timing test was run on a single-user personal computer running Microsoft Windows 95, with no other programs running concurrently. Nevertheless, the timing data produced by this test was highly erratic. It was not uncommon for the number of times which a function could be executed during a single clock tick to vary by several orders of magnitude from one clock tick to another. Even when the results were averaged over tens of thousands of clock ticks, there was still a great deal of variability in the relative times assigned to the various functions. In an attempt to remedy this problem, the amount of time required to compute a function was taken as the inverse of the *maximum*, rather than average, number of times the function was executed during a single clock tick. In a multitasking operating system, the amount of computation that can be performed in a clock tick is limited by the amount of processor time that happens to be devoted to the relevant task (in this case, the timing test) during that tick, but presumably can never exceed a certain maximum (i.e. when 100% of the processor time is devoted to the task). This approach made the output of the timing test much more stable. Using this approach, the timing test was run for one hour (which comprises 3.6 million clock ticks) on a Pentium II-233 MHz system, and on an older Pentium 60 MHz system for reference. The results of these experiments are given in Table 2.2 (for the Pentium II-233 processor) and Table 2.3 (for the Pentium-60 processor). To facilitate meaningful cross-processor comparison, the coefficients M and B in each of these tables have been normalized so that the M value for the *Sum* function is equal to 1.0. At the time these experiments were run, only 13 of the eventual 23 primitive functions were coded in the system.

Table 2.2: Pentium II-233 Timing Data.

PRIMITIVE FUNCTION	M	B
`Sigmoid`	0.607312	9.118956
`Product (*)`	1.350015	4.132117
`ReciprocalProduct`	1.336272	5.798273
`ReciprocalSum`	1.904643	2.245144
`Cos`	0.798725	8.971738
`Ln`	1.179137	3.906359
`Sqrt`	0.776423	3.942328
`Sum (+)`	1.000000	2.076896
`TanH`	1.020568	5.199593
`Max`	1.114182	2.035016
`Min`	1.174219	1.540112
`Sign`	0.945316	1.003580
`Abs`	1.132322	0.225641

Table 2.3: Pentium-60 Timing Data.

PRIMITIVE FUNCTION	M	B
`Sigmoid`	0.905280	8.787455
`Product (*)`	2.238005	4.475859
`ReciprocalProduct`	2.106955	5.681528

ReciprocalSum	1.927736	3.895167
Cos	1.036731	9.678981
Ln	1.326693	3.191963
Sqrt	0.942754	4.568197
Sum (+)	1.000000	3.836034
TanH	0.868542	7.711673
Max	1.396215	0.625175
Min	1.162758	2.451946
Sign	1.077272	2.789748
Abs	1.042189	1.577727

The coefficients given in these two tables for the various primitive functions exhibit considerable variation. Between the two values of M listed for the *Sigmoid* function, for example, there is approximately a 50% difference, as is roughly the case for the *Product* and *ReciprocalProduct* functions. Also, the relative ranking of the functions is not consistent; in Table 2.3, the *TanH* function with 1 argument is more expensive to compute than the *ReciprocalProduct* function, while in Table 2.2, the reverse is true. Finally, the coefficients given in these tables do not significantly distinguish between simple functions such as *Sum* and *Sign*, intermediate functions such as *Product* and *Division*, and more complex functions such as *Sqrt*, *Ln*, and *Sin*. For these reasons, the hardware timing approach to cost assignment was abandoned, and a simpler hand-coded set of costs used in its place. The values of B were (somewhat arbitrarily) set to 1 for the arithmetic functions involving division and/or multiplication, 0.1 for the functions sum and subtraction function, and 10 for any more complex function such as *Exp*, *Cos*, or *Ln*. The values of M were uniformly set to 0 since the variable arity feature of the system was not used, the leaving the B coefficient as the single determinant of cost for a function. The final costs, as defined by this coefficient, are given in Table 2.4 for the 23 functions finally used in this system.

Table 2.4: Final Assigned Costs of Primitive Functions.

PRIMITIVE FUNCTION	COST
Sigmoid	10
Product	1
ReciprocalProduct	1
ReciprocalSum	1
Cos	10
Ln	10
Sqrt	10
Sum	.1
TanH	10
Max	.1
Min	.1
Sign	.1
Abs	.1

Sin	10
Exp	10
Div	1
Subtract	.1
GreaterThan	.1
GreaterThanOrEqual	.1
LessThan	.1
LessThanOrEqual	.1
IfThenElse	.1
Split	.1

2.4 PROGRAM OUTPUT

The output produced by this genetic programming system is designed to be highly informative and extensive. At the conclusion of each run or set of runs executed by the system, an HTML summary is generated which can be viewed using a web browser, and which gives the full parameter settings used for the run(s), the best-of-run individual for each run, and the set of individuals Pareto front for the union of the entire population histories of each run (we refer to these individuals as "candidate solutions"). Additionally, links are provided in this HTML summary to plot data representing either (a) the fitness, error, cost, adjusted cost, and adjusted error curves for the best-of-generation individual in each generation, for experiments involving a single run or (b) plots representing convergence probability curves, expected individuals-to-be-processed curves, and individual effort curves (to be defined later) for experiments involving multiple runs. The plot data is output to a text file which can be read by the Gnuplot plotting program (Williams and Kelley 1998). Also provided is a link to the set of candidate solutions, represented in a textual form which can be pasted into a worksheet using the Maple symbolic mathematics package (Heal, Hansen, and Rickard 1998).

Figures 2.1 through 2.7 illustrate the output of the system for a simple problem involving symbolic regression of the quadratic polynomial f(x) = x^2 + x. Figure 2.1 gives the HTML summary for a single-run experiment with this problem; Figures 2.2 through 2.6 give the associated fitness, error, cost, adjusted error, and adjusted cost curves, respectively. Figure 2.7 is a screen shot of the candidate solutions generated as a result of this experiment pasted into the Maple symbolic mathematics package, where they can be further analyzed. For a full explanation of the parameters given at the top of the HTML summary in Figure 2.1, see Appendix B.

FGP run started Thursday, Feb 22 2001, 03:59:49 PM; ended Thursday, Feb 22 2001, 03:59:55 PM

Parameters

Problem name: f(x) = x^2 + x (50s,1i,1o)
Training set file: FGPExample_50.ts
Generation limit: 50
Fitness limit: 0.999
Hit ratio limit: None
Hit range: 0.01
Initial rand. seed: 0
GA: Standard
Population size: 500
Fitness-proportionate reproduction fraction: 0.1
Error metric: Absolute, total sum
Error multiplier: 1
Adjusted error power: 1
Function set: {Product, Ln, Sum, Exp, Div, Subtract}
Random numeric terminal: On
Occam's razor: Off
Rand. tree uniform depth: Off
Rand. tree default arity: On
Random weight initialization: Off
Weight combination: Averaged
Gaussian weight mutation: 0
Pr[Function mutation]: 0
Pr[Subtree mutation]: 0
Crossover: SingleSubtreeSwap
Biased choice of crossover point: On
Pr[Internal crossover point]: 0.9
Max. nodes per output: [No Limit]
Max. rand. tree depth: 5
Max. tree depth: 16

Results

Best-of-run individual (generation 3):
O1 = Sum(i0, Product(i0, i0))
Raw error: 2.51227e-10
Adj. error: 1
Raw cost: 2.2
Adj. cost: 0.3125
Fitness: 1
Total nodes: 5
Hits: 50 of 50

GA was run for 3 generations.
90% of best fitness was achieved after 3 generations.

Candidate solutions: (ordered by 1*error + 0*cost)

1. Raw error = 2.51227e-10; raw cost = 2.2 *(run 0, generation 2)*
O1 = Sum(i0, Product(i0, i0))

2. Raw error = 17.9351; raw cost = 2 *(run 0, generation 2)*
O1 = Product(Exp(Product(0.322123, Exp(Ln(2.63634)))), i0)

3. Raw error = 21.138; raw cost = 0.4 *(run 0, generation 0)*
O1 = Sum(i0, Subtract(i0, Div(Exp(Product(2.53258, -2.7337)), Exp(Ln(1.82638)))))

4. Raw error = 45.1888; raw cost = 0.2 *(run 0, generation 0)*
O1 = Sum(0.404523, i0)

5. Raw error = 53.1089; raw cost = 0 *(run 0, generation 0)*
O1 = i0

Cost/error/fitness data is available here.

Figure 2.1: Example HTML Summary.

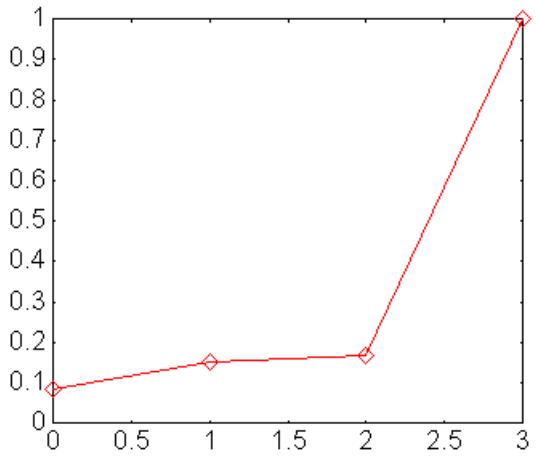

Figure 2.2: Fitness Curve.

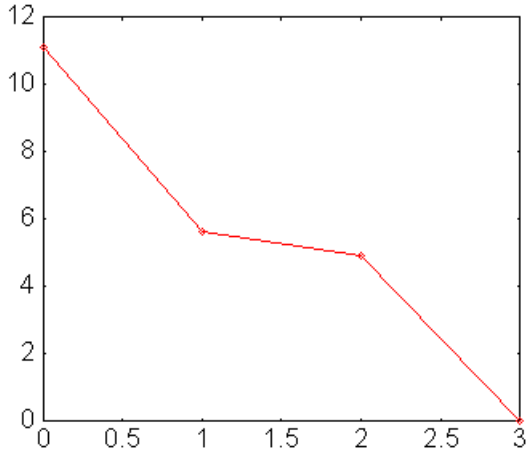

Figure 2.3: Error Curve.

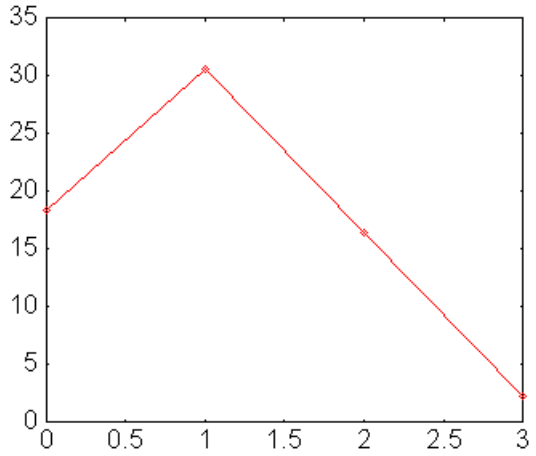

Figure 2.4: Cost Curve.

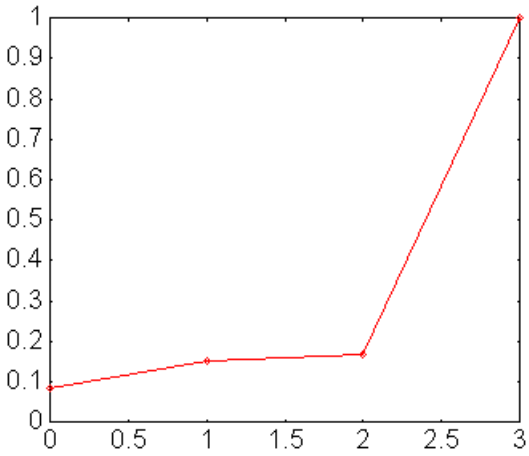

Figure 2.5: Adjusted Error Curve.

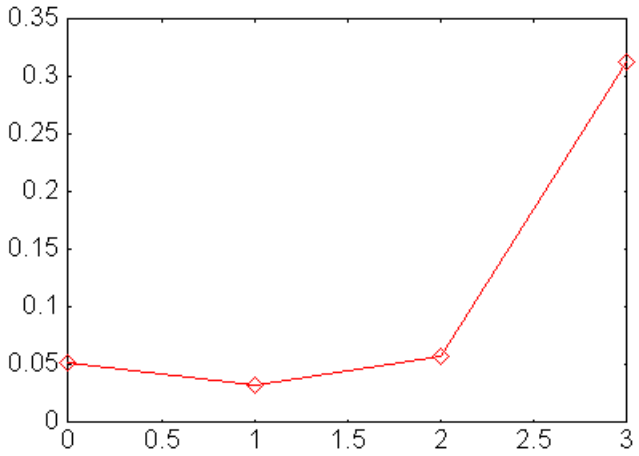

Figure 2.6: Adjusted Cost Curve.

```
> c0:=[(x)+((x)*(x))];
  c1:=[(exp((0.322123)*(exp(ln(2.63634)))))*(x)];
  c2:=[(x)+((x)-((exp((2.53258)*(-2.7337)))/(exp(ln(1.82638)))))];
  c3:=[(0.404523)+(x)];
  c4:=[x];
  candidate_solution_set:=[c0,c1,c2,c3,c4];
```

$$c0 := [x + x^2]$$

$$c1 := [2.337836081\,x]$$

$$c2 := [2\,x - .0005390782552]$$

$$c3 := [.404523 + x]$$

$$c4 := [x]$$

$$candidate\_solution\_set := [[x + x^2], [2.337836081\,x], [2\,x - .0005390782552], [.404523 + x], [x]]$$

```
>
```

Figure 2.7: Candidate Solutions Imported into Maple.

For experiments involving multiple runs, a different set of plot curves are generated. The *convergence probability curve* gives the fraction of runs $P(g)$ such that the run converged, or reached a pre-specified level of fitness or error, at or before generation g. The *individual effort curve* (Koza 1992) gives the number of individuals $I(g)$ which must be processed when executing multiple independent runs, each lasting up to g+1 generations (the initial random generation plus g subsequent generations), for a 99% probability of converging in at least one of the runs. The formula for $I(g)$ is given by:

$$I(g) = \log(0.01)/\log(1.0\text{-}P(g)\text{+}0.5) \ *(g\text{+}1)*([\textit{population size}])$$

Finally, we introduce a function $s(g)$ to denote the expected number of individuals to be processed before finding a solution (i.e. converging) when executing multiple independent runs, each lasting up to g generations. This is given by the formula:

$$s(g) = (1/P(g))*(1+g)*([population\ size])$$

This formula is justified by the fact that a run involving g generations (after the initial random generation) will involve the processing of $(1+g)*([population\ size])$ individuals, combined with the observation that given a fixed convergence probability $P(g)$, the number of indepdendent runs required to find a solution follows a geometric distribution, so that its expected value is $(1/P(g))$. The formula is actually a slight overestimate, since it assumes $(1+g)*([population\ size])$ individuals will be processed in each independent run, whereas if converge occurs earlier than generation g in a particular run, fewer than this number of individuals will need to be processed. This caveat is present in Koza's formula for $I(g)$ as well (as he acknowledges), and will be ignored for the purposes of our analyses.

Figures 2.8 through 2.10 represent the output of the system for an experiment involving 10 runs against the same symbolic regression problem discussed in regard to figures 2.1 through 2.7. The figures give the convergence probability ($P(g)$), individual effort ($I(g)$), and expected number of individuals-to-be-processed ($s(g)$) curves, respectively.

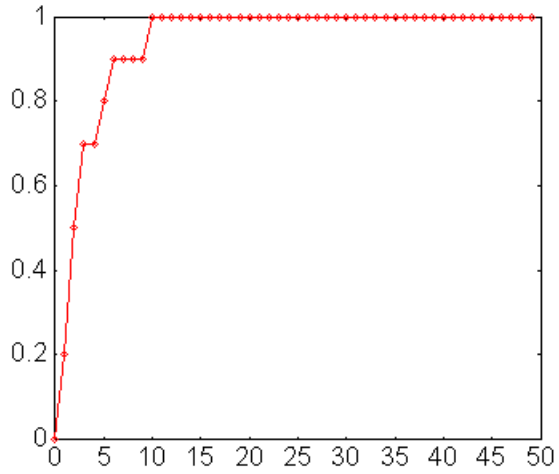

Figure 2.8: Convergence Probability Curve.

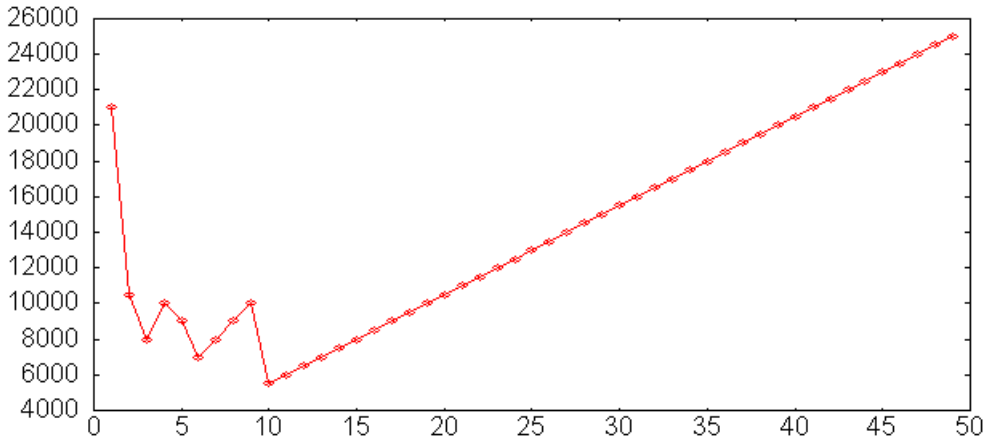

Figure 2.9: Individual Effort Curve.

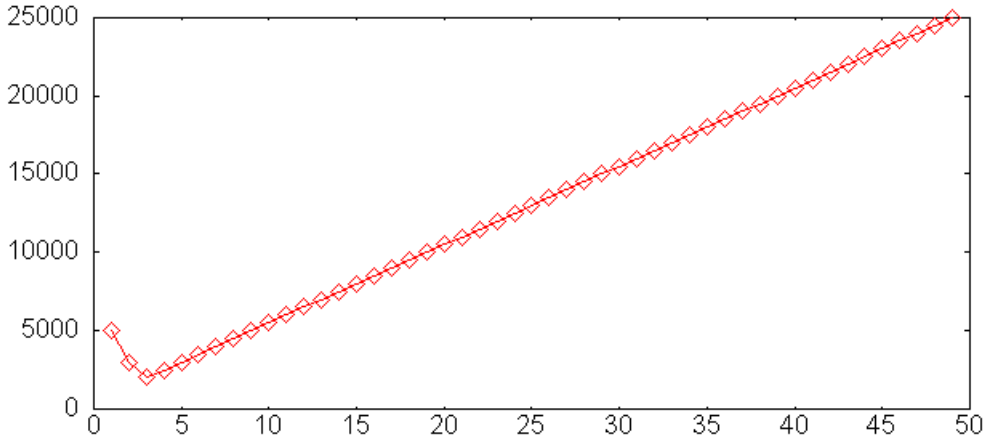

Figure 2.10: Expected Number of Individuals to be Processed.

2.5 CONSISTENCY WITH OTHER GENETIC PROGRAMMING SYSTEMS

The system used for the experiments described in this paper was designed to be configurable to be functionally equivalent to the GP described in (Koza 1992). To verify this equivalence, an experiment described in *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza 1992), performed using Koza's LISP genetic programming kernel, was replicated using the author's GP system. The experiment involved symbolic regression of the quartic polynomial f(x) = x^4 + x^3 + x^2 + x. The parameter settings used for this experiment are given in Table 2.5. The function set was the arithmetic function set {*,+,/,-}. An individual's error was calculated as the sum of its absolute error for all training points. Training data was taken 20 points with *x*-values drawn at random from the interval [-1,1]. Adjusted fitness (based upon which fitness-proportionate selection is conducted) was calculated according to the formula 1/(1+[*error*]).

Table 2.5: Parameter Settings for Reproduction of Symbolic Regression Experiment.

PARAMETER	VALUE
Population size	500
Num. generations	51
Crossover fraction	90%
Fitness proportionate reproduction fraction	10%
Probability of internal crossover point selection	90%
Max. depth of individuals created by crossover	5
Max. depth of randomly generated individuals	16

Table 2.6: Results for Reproduction of Symbolic Regression Experiment.

SEED	# GENS.	SEED	# GENS.	SEED	# GENS.	SEED	# GENS.	SEED	# GENS.
0	-	*10*	-	*20*	4	*30*	-	*40*	-
1	-	*11*	-	*21*	-	*31*	-	*41*	-
2	-	*12*	-	*22*	-	*32*	-	*42*	18
3	-	*13*	-	*23*	19	*33*	-	*43*	-
4	-	*14*	32	*24*	-	*34*	-	*44*	27
5	-	*15*	-	*25*	-	*35*	-	*45*	-
6	23	*16*	-	*26*	9	*36*	-	*46*	-
7	-	*17*	-	*27*	-	*37*	-	*47*	-
8	-	*18*	-	*28*	-	*38*	24	*48*	-
9	16	*19*	-	*29*	-	*39*	24	*49*	40

On 11 of the 50 independent runs executed as part of this experiment, a perfect solution is found within 50 generations, for a 22% probability of convergence. Based on 295 independent runs, Koza reports a 23% probability of convergence. The two figures are sufficiently close to lead us to believe that the systems are consistent.

3 OPTIMIZING GP PARAMETERS

There are many parameters involved in executing a run of genetic programming, each of which can have a potentially important effect on the efficacy of the system at finding a solution. It would be desirable, in evolving approximations through genetic programming, to come up with a consistent set of parameter settings which could be expected to perform well over this general domain. No standard set of benchmark problems for genetic programming in general or symbolic regression in particular (Feldt, O'Neill, Ryan, Nordin, and Langdon 2000), so the choice of problems used in experiments to obtain such a set of parameters must inherently be somewhat ad hoc. This section presents an attempt to optimize GP parameters for the problem of finding approximations to functions.

3.1 EXPERIMENTS WITH INITIAL TEST SUITE

This section describes experiments run against a suite of 28 functions designed to test various aspects of GP performance. The complete set of functions is given in the leftmost column of Table 3.1.

Function 1 was intended as a trivial test of the ability of GP to discover the identity function. Functions 2-6 attempted to test the ability of GP to discover constants, both as coefficients of a simple polynomial (or alone) and nested inside the argument a non-linear function. Functions 7-14 attempted to determine ability of GP to find polynomials of various degree, including polynomials which can be easily factored and those that cannot. Functions 15-17 were designed to given an indication as to the effect of noise on GP performance. Functions 18-26 attempted to determine the ability of GP to exploit building blocks. Function 27 was designed to provide the opportunity to rediscover a known approximation formula for Harmonic numbers, namely that $H_n \approx \ln(n) + \gamma$ (see section 4). Finally, function 28 was designed to provide an opportunity to rediscover Newton's telescopic factoring method, namely that a polynomial such as x^4 + x^3 + x^2 + x + 1 can be calculated using a smaller number of multiplications by nesting the multiplications telescopically, i.e. x(x(x(x +1) + 1) + 1) + 1.

Two experiments, each involving 3 independent runs on each of these 28 problems, were run using the author's GP system: an initial experiment conducted using the "factory" settings of the library, including the use of weights, variable-arity functions, and other non-standard features (see Appendix B), and a second experiment using the standard GP settings (section 2.5). First experiment was conducted using the function set {*Sigmoid*, *Product*, *ReciprocalProduct*, *ReciprocalSum*, *Cos*, *Ln*, *Sqrt*, *Sum*, *TanH*, *Max*, *Min*, *Sign*, *Abs*}, which was the set of all functions coded into the library at the time. The second experiment (in an attempt to ease problem difficulty) used the reduced function set {*Product*, *ReciprocalSum*, *Cos*, *Ln*, *Sqrt*, *Sum*}. The first experiment used 101 generations (the initial random generation plus 100 subsequent generations) for each run, while the second experiment used 51 generations for each run. The results of these two experiments are presented in Table 3.1. For each of the 28 functions, the number of generations needed to discover the function (including the initial random generation) in each of the three independent runs for the two sets of parameter settings is given under the appropriate column, and a dash (-) character is used to denote that the function was not ever discovered in the corresponding run (after 51 or 101 generations). In the case of functions such as

f(x) = 3.14159, a run was often terminated when a solution was found that was sufficiently close (though not exactly equal) to the target function. An asterisk (*) symbol is used to denote such inexact convergence.

Table 3.1: Experiments with Initial Test Suite.

FUNCTION	INITIAL EXPERIMENT	STANDARD GP
1. f(x) = x	11, 82, 28	1, 1, 1
2. f(x) = 3.14159	32, 32, 88*	-, 24*, -
3. f(x) = 0.5667*x + 3.14159	-, -, -	-, -, -
4. f(x) = 0.8x^2 + 0.5667*x + 3.14159	-, -, -	-, -, -
5. f(x) = sin(0.8x^2 + 0.5667*x + 3.14159)	-, -, -	-, -, -
6. f(x) = sqrt(0.8x^2 + 0.5667*x + 3.14159)	-, -, -	-, -, -
7. f(x) = (x+1)^2	-, -, -	31, -, -
8. f(x) = (x+1)^3	-, -, -	-, -, -
9. f(x) = (x+1)^4	-, -, -	-, -, -
10. f(x) = (x+1)^5	-, -, -	-, -, -
11. f(x) = (x+1)(x+2)	-, -, -	-, -, -
12. f(x) = (x+1)(x+2)(x+3)	-, -, -	-, -, -
13. f(x) = (x+1)(x+2)(x+3)(x+4)	-, -, -	-, -, -
14. f(x) = (x+1)(x+2)(x+3)(x+4)(x+5)	-, -, -	-, -, -
15. f(x) = (x+1)^3 + GaussRand(StdDev=0.1)	-, -, -	-, -, -
16. f(x) = (x+1)^3 + GaussRand(StdDev=1.0)	-, -, -	-, -, -
17. f(x) = (x+1)^3 + GaussRand(StdDev=10.0)	-, -, -	-, -, -
18. f(x) = sin(x^2+x+1)	-, -, -	-, -, -
19. f(x,y) = sin(x^2+x+1) + sin(y^2+y+1)	-, -, -	-, -, -
20. f(x,y,z) = sin(x^2+x+1) + sin(y^2+y+1) + sin(z^2+z+1)	-, -, -	-, -, -
21. f(x) = ln(x)	-, -, -	1, 1, 1
22. f(x) = (x-7)/(x+3)	-, -, -	-, -, -
23. f(x) = ln(x) + sin(x^2+x+1)	-, -, -	-, -, -
24. f(x) = ln(x) + (x-7)/(x+3)	-, -, -	-, -, -
25. f(x) = sin(x^2+x+1) + (x-7)/(x+3)	-, -, -	-, -, -
26. f(x) = ln(x) + sin(x^2+x+1) + (x-7)/(x+3)	-, -, -	-, -, -
27. f(x) = H_x	-, -, -	38*, -, -
28. f(x) = x^4 + x^3 + x^2 + x + 1	-, -, -	-, -, -

* denotes inexact convergence

It should be apparent from examination of table 3.1 that the selected test suite is too hard. Only the simplest of the 28 functions are consistently discovered, and the vast majority of functions are not discovered at all. Though it might be possible to discover any one of these functions using a higher number of independent runs, it is likely that the number of runs required, the number of problems in the test suite, and the resulting computation time required would make the test

suite unusable. Here we have 23 problems, representing 138 independent runs, which were never solved under either set of parameter settings. Since the experiments reported here took 6 and 8 hours to complete, respectively, and since many such experiments would have to be performed to arrive at an optimum set of parameter settings, this test suite is largely impractical for our purposes.

3.2 EXPERIMENTS WITH REVISED TEST SUITE

My next set of experiments were conducted against a revised test suite consisting of the functions:

1. f(x) = ln(x) + cos(x)
2. f(x) = ln(x) + cos(x) + sqrt(x)
3. f(x) = ln(x) + cos(x) + sqrt(x) + x^2
4. f(x) = ln(x)*cos(x)
5. f(x) = ln(x)*cos(x)*sqrt(x)
6. f(x) = ln(x)*cos(x)*sqrt(x)*(x^2)

Each of these six functions involves additive or multiplicative combinations of the simple building block functions *ln(x)*, *cos(x)*, *sqrt(x)*, and *x^2*. This was intended to provide a smooth gradation from simple to difficult problems, and to provide problems which were simple enough that they could be solved on consistent basis using my system.

This section presents the results of two experiments run against this smaller test suite using the standard GP parameters, using population sizes of 250 and 500, and given in tables 3.2 and 3.3, respectively. For all functions, 25 points were taken as training data, each of whose *x*-values were the absolute value of a real number drawn at random from a Gaussian distribution with mean 0 and standard deviation 10. 100 independent runs were executed in each experiment for each function. As described in section 2.4, our GP system generated curves for the probability of convergence, individual effort, and expected number of individuals-to-be-processed as a function of the generation limit *g*. In this analysis, we let *G* denote the value of *g* which minimizes the individual effort $I(g)$, and let *p* denote the probability of convergence at or before generation *G* (i.e. $p \equiv P(G)$). We calculate 99% confidence intervals for *p* according to the formula:

$$[N\%\ \textit{confidence interval for}\ p] = p \pm Z_N * \mathrm{sqrt}([p*(1-p)]/n)$$

which is statistically valid when $n*p*(1-p) >= 5$ (Mitchell 1997). Here *n* denotes the number of indepdent runs (100 in our case), and Z_N denotes the appropriate confidence interval coefficient, which for an N=99% confidence interval is equal to 2.58.

We let *s* denote the expected number of individuals to be processed before finding a solution when executing multiple independent runs with a generation limit of *G* (i.e. $s \equiv s(G)$), and let I denote the corresponding individual effort (i.e. $I \equiv I(G)$). We take *s* to be the more natural measure of problem difficulty than *I*, since it indicates the average-case workload imposed on GP for a particular problem. Note, however, that while *I* is the minimum value of $I(g)$, *s* is not necessarily the minimum value of $s(g)$, a slight caveat which will be ignored for the purposes of this analysis.

Confidence intervals have been calculated for *s* as well, and are given in brackets to the right of the calculated value in table 3.2 and 3.3. In cases where the convergence probabilities were such that the confidence interval formula was not statistically valid (see above), question marks (??) appear in the corresponding places.

In summary,

G = Optimal generation limit (for minimal value of *I(g)*)

p = Probability of convergence at generation *G*.

s = Expected number of individuals to process before finding a solution = $(1/p)*(1+G)*([\textit{population size}])$

I = Number of individuals which must be processed for 99% probability of finding a solution

$= I(G) = \lceil \log(0.01)/\log(1.0-p+0.5) * (G+1)*([\textit{population size}])$

Table 3.2: Experiments with Revised Test Suite Using Population Size = 250.

PROBLEM	G	p	s	I
1. f(x) = ln(x) + cos(x)	5	0.28 ± 0.116	5357 [3788,9146]	21000
2. f(x) = ln(x) + cos(x) + sqrt(x)	20	0.13 ± 0.087	40385 [24194,122093]	173250
3. f(x) = ln(x) + cos(x) + sqrt(x) + x^2	14	0.02 ± *??*	187500 [*??*,*??*]	855000
4. f(x) = ln(x)*cos(x)	5	0.65 ± 0.123	2308 [1940,2846]	6000
5. f(x) = ln(x)*cos(x)*sqrt(x)	8	0.12 ± 0.084	18750 [11029,62500]	81000
6. f(x) = ln(x)*cos(x)*sqrt(x)*(x^2)	48	0.11 ± 0.081	111364 [64136,422414]	490000

Table 3.3: Experiments with Revised Test Suite Using Population Size = 500.

PROBLEM	G	p	s	I
1. f(x) = ln(x) + cos(x)	7	0.66 ± 0.122	6061 [5115,7434]	16000
2. f(x) = ln(x) + cos(x) + sqrt(x)	10	0.15 ± 0.092	36667 [22727,94828]	154000
3. f(x) = ln(x) + cos(x) + sqrt(x) + x^2	18	0.06 ± 0.061	158333 [78512,+∞]	703000
4. f(x) = ln(x)*cos(x)	6	0.86 ± .090	4070 [3684,4545]	7000
5. f(x) = ln(x)*cos(x)*sqrt(x)	9	0.14 ± 0.090	35714 [21739,100000]	155000
6. f(x) = ln(x)*cos(x)*sqrt(x)*(x^2)	1	0.01 ± *??*	100000 [*??*,*??*]	458000

In general, the corresponding confidence intervals for s in tables 3.2 and 3.3 overlap. The only meaningful conclusions that can be drawn from these tables is that, with 99% confidence, the function f(x)=ln(x)*cos(x) is between 1.29 and 2.34 times easier to find with a population size of 250 than with one of 500 (see table entries for function 4), when these population sizes are combined with all other parameter settings used in these experiments.

3.3 IMPRACTICALITY OF OPTIMIZING GP PARAMETERS IN THIS MANNER

As stated in the previous paragraph, tables 3.2 and 3.3 provide a meaningful conclusion regarding only one of the six functions on which the experiments were conducted. Using the formula for confidence intervals, we can calculate the number of additional runs which would have be required to prevent each set of corresponding confidence intervals from overlapping, assuming that the same convergence probabilities continued to be observed. Using this information, we can estimate the amount of time that would be required to draw meaningful conclusions concerning all six functions in this test suite. Table 3.4 gives the minimum number of runs for each function which would be required to produce definitive results for this experiment.

Table 3.4: Number of Runs Required to Produce Definitive Results for Experiments with Revised Test Suite.

PROBLEM	MINIMUM RUNS REQUIRED
1. f(x) = ln(x) + cos(x)	2475
2. f(x) = ln(x) + cos(x) + sqrt(x)	17566
3. f(x) = ln(x) + cos(x) + sqrt(x) + x^2	26794
4. f(x) = ln(x)*cos(x)	33
5. f(x) = ln(x)*cos(x)*sqrt(x)	475
6. f(x) = ln(x)*cos(x)*sqrt(x)*(x^2)	99953

As illustrated in this table, only function 4 provided a meaningful result within the 100 independent runs allowed for each function in this experiment. The number of runs required for every other function are substantially larger. Since executing 100 independent runs took an average of about 2 hours per problem, determining a "winner" for problem 1 would take just under 50 hours, while determining a winner for problem 2 would take over 2 weeks. Even if all this computation time were expended, it would only give information concerning one parameter change applied to a small set of problems, and would only be a valid indicator of the significance of that parameter change when it occurs in conjunction with all other parameter settings used for these experiments. Thus, it does not seem to be practical to try to find an exact optimal set of parameters for a general test suite of symbolic regression problems.

4 REDISCOVERY OF HARMONIC NUMBER APPROXIMATIONS

One commonly used quantity in mathematics is the Harmonic number, defined as:

$$H_n \equiv \sum_{i=1}^{n} 1/i$$

This series can be approximated using the asymptotic expansion (Gonnet 1984):

$$H_n = \gamma + \ln(n) + 1/(2n) - 1/(12n^2) + 1/(120n^4) - \ldots$$

where γ is Euler's constant ($\gamma \approx 0.57722$).

Using the system described in section 2, and the function set {+, *, *ReciprocalSum*, *Ln*, *Sqrt*, *Cos*}, the authors attempted to rediscover some of the terms of this asymptotic expansion, where the *ReciprocalSum* function is simply a reciprocal function (since we are not using the variable-arity feature of the system), and *Sqrt* and *Cos* are included as extraneous functions.

All parameter settings used in this experiment are the same as those given for the symbolic regression experiment in Section 2.5, including a population size of 500 and generation limit of 50. The first 50 Harmonic numbers (i.e. H_n for 1<=n<=50) were used as training data. 50 independent runs were executed, producing a single set of candidate approximations. Error was calculated as the sum of absolute error for each training instance. The set of evolved approximations returned by the genetic programming system (which represent the Pareto front for the population histories of all independent runs) is given in Table 4.1. For the purpose of analysis, each approximation was simplified using the Maple symbolic mathematics package. This simplified form, as well as the cost and error associated with each approximation and the run and generation in which it originated, are given in the table. For the evolved approximations, a LISP-like notation is used, with the functions *ReciprocalSum*, *Ln*, *Sqrt*, and *Cos* denoted by `RCP`, `RLOG`, `SQRT`, and `COS`, respectively.

Table 4.1: Evolved Harmonic Number Approximations.

EVOLVED LISP EXPRESSION				
SIMPLIFIED MAPLE EXPRESSION	**ERROR**	**COST**	**RUN**	**GEN.**
1. `(+ (RCP (RCP (+ (RLOG X) (RCP (SQRT (* 4.67956 (RLOG 1.90146)))))))) (RCP (+ (+ (SQRT (+ (+ (RCP (RCP (+ (RLOG X) (RCP (SQRT (* 4.67956 (RLOG 1.90146)))))))) (RCP (+ (+ (+ (RCP X) X) (RLOG 1.90146)) X))) (* X (+ X (SQRT (RLOG -0.455794)))))) (RLOG -0.455794)) X)))`				
ln(x)+.5766598187+1/(sqrt(ln(x)+.5766598187+1/(1/x+2*x+.6426220121)+x^2)+x)	0.0215204	39.1	22	32
2. `(+ (RCP (RCP (+ (RLOG X) (RCP (SQRT (* 4.67956 (RLOG 1.90146)))))))) (RCP (+ (+ X (RCP (+ (+ (RLOG 1.90146) (RLOG (RCP (+ (RLOG X) (RCP (SQRT (* 4.67956 (RLOG 1.90146)))))))))))) (+ (RCP (SQRT (* 4.67956 (RLOG 1.90146)))) X)))) X)))`				
ln(x)+.5766598187+1/(2*x+1/(1.219281831 + ln(1/(ln(x)+.5766598187))+x))	0.0229032	35.8	22	35

3. (+ (RCP (RCP (+ (RLOG X) (RCP (SQRT (* 4.67956 (RLOG 1.90146)))))) (RCP (+ (+ X (RCP (+ (+ (RLOG 1.90146) (RLOG (RCP (RCP (+ (+ (+ (SQRT (* 4.67956 (RLOG 1.90146))) X) (RLOG -0.455794)) X))))) (RLOG 1.90146)))) X)))				
ln(x)+.5766598187+1/(2*x+1/(1.285244024 + ln(1.734124639+2*x)))	0.0264468	26.9	22	37
4. (+ (RCP (RCP (+ (RLOG X) (RCP (SQRT (* 4.67956 (RLOG 1.90146)))))) (RCP (+ (+ X (RCP (+ (RCP (+ (RCP (RLOG 1.90146)) (RCP (SQRT (* 4.67956 (RLOG 1.90146)))))) (+ (+ (+ (RCP 4.67956) (RLOG X)) (RCP (+ (* 4.67956 (RLOG 1.90146)) X)) 1.90146)))) X)))				
ln(x)+.5766598187+1/(2*x+1/(2.584025920 + ln(x)+1/(3.007188263+x)))	0.0278816	25.9	22	49
5. (+ (RCP (RCP (+ (RLOG X) (RCP (SQRT (* 4.67956 (RLOG 1.90146)))))) (RCP (+ (+ X (RCP (+ (+ (SQRT (RLOG (RLOG 1.90146))) (+ (RCP (SQRT (* 4.67956 (RLOG 1.90146)))) (RCP X))) X))) X)))				
ln(x)+.5766598187+1/(2*x+1/(.5766598187 + 1/x+x))	0.0286254	15.7	22	36
6. (+ (RCP (RCP (+ (RLOG X) (RCP (SQRT (* 4.67956 (RLOG 1.90146)))))) (RCP (+ (+ X (RCP (+ (SQRT (* 4.67956 (RLOG 1.90146))) (SQRT (RLOG (* 4.67956 (RLOG 1.90146)))))) X)))				
ln(x)+.5766598187+1/(2*x+.3592711879)	0.0293595	13.4	22	37
7. (+ (+ (RLOG X) (RCP (SQRT (* 4.67956 (RLOG 1.90146))))) (RCP (+ (+ X (RCP (+ (+ (RLOG 1.90146) (RLOG (+ (RCP (RCP (SQRT (* 4.67956 (RLOG 1.90146))))) (RCP (SQRT (* 4.67956 (RLOG 1.90146)))))))) (SQRT 1.90146)))) X)))				
ln(x)+.5766598187+1/(2*x+.3497550998)	0.0297425	11.4	22	42
8. (+ (RLOG (+ (+ X (RLOG (RLOG (+ 2.04489 (COS (RLOG (+ (RCP -2.35221) (COS (RCP 2.04489)))))))))) (RCP 2.04489))) (COS (RLOG 2.59758)))				
ln(x+.5022291180)+.5779513609	0.0546846	10.3	40	28
9. (+ (RLOG (+ X (RCP 2.04489))) (COS (RLOG 2.59758)))				
ln(x+.4890238595)+.5779513609	0.0653603	10.2	40	21
10. (+ (SQRT (COS (+ (* (COS (* (RCP (RLOG 3.90774)) (RLOG (+ (COS (RLOG (RLOG -2.11142))) (RLOG 3.90774))))) (RLOG (+ 0.777459 (COS (RLOG 3.52657))))) (RCP -0.786615)))) (RLOG X))				
0.5965804779+ln(x)	1.44089	10.1	49	49
11. (+ (* (RCP (RCP -4.34843)) (+ (RCP -1.02527) (RCP X))) (RLOG (SQRT (SQRT 0.316019))))				
3.953265289-4.348430001/x	20.2786	2.2	3	1
12. (+ (RLOG 3.97427) 2.43614)				
3.815981083	31.0297	0	10	4

An analysis of this set of candidate solutions follows. For comparison, Table 4.2 presents the error values associated with the asymptotic expansion when carried to between 1 and 4 terms.

Table 4.2: Accuracy of Asymptotic Expansion.

TERMS	EXPRESSION	ERROR
1	0.57722	150.559
2	0.57722 + ln(n)	2.12094
3	0.57722 + ln(n) + 1/(2n)	0.128663
4	0.57722 + ln(n) + 1/(2n) - 1/(12n^2)	0.00683926

Candidate approximation 12, the cheapest approximation in the set, is simply a constant, while candidate approximation 11 is a simple rational polynomial. Candidate approximation 10 represents a variation on the first two terms of the asymptotic expansion, with a slightly perturbed version of Euler's constant which gives greater accuracy on the 50 supplied training instances. Candidate solutions 8 and 9 represent slightly more costly variations on the first two terms of the asymptotic expansion which provide increased accuracy over the training data. Similarly, candidate solutions 6 and 7 are slight variations on the first three terms of the asymptotic expansion, tweaked as it were to give greater accuracy on the 50 training points. Candidate solutions 2-5 can be regarded as more complicated variations on the first three terms of the asymptotic expansion, each giving a slight increase in accuracy at the cost of a slightly more complex computation. Candidate solution 1 represents a unique and unexpected approximation which has the greatest accuracy of all evolved approximations, though it is unequivocally inferior to the first four terms of the asymptotic expansion has presented in Table 2.

Candidate approximations 1-7 all make use of the constant 0.5766598187 as an approximation to Euler's constant, which was evolved using the LISP expression:

```
(RCP(SQRT(* 4.67956 RLOG(1.90146))))
```

This approximation is accurate to two decimal places. Candidate approximations 8 and 9 make use of the slightly less accurate approximation of 0.5779513609, evolved using the LISP expression:

```
(COS(LN 2.59758))
```

Note that in this experiment, pure error-driven evolution has produced a rich set of candidate approximations exhibiting various trade-offs between accuracy and cost. Also note that with the exception of the first candidate approximation, which uses the `SQRT` function, the `SQRT` and `COS` functions were used only in the creation of constants, so that these extraneous functions did not provide a significant obstacle to the evolution of the desired approximations. Thus, this experiment represents a partial rediscovery of the first three terms of the asymptotic expansion for H_n.

5 DISCOVERY OF RATIONAL POLYNOMIAL APPROXIMATIONS FOR KNOWN FUNCTIONS

5.1 INTRODUCTION

By limiting the set of available functions to the arithmetic function set {*,+,/,-}, it is possible to evolve rational polynomial approximations to functions, where a rational polynomial is defined as the ratio of two polynomial expressions. Since approximations evolved with the specified function set use only arithmetic operators, they can easily be converted to rational polynomial form by hand, or by using a symbolic mathematics package such as Maple. Approximations evolved in this manner can be compared to approximations obtained through other techniques such as Padé approximations by comparing their Pareto fronts. In the section, we present the results of such a comparison for five common mathematical functions: the natural logarithm ln(x), the square root sqrt(x), the hyperbolic arcsine arcsinh(x), the negative exponential exp(-x), and the hyperbolic tangent tanh(x). The functions are approximated over the interval [0,100], with the exception of ln(x), which (since ln(0) is not defined) is approximated over the interval [1,100]. The functions were selected to be common, aperiodic functions whose calculation was sufficiently complex to warrant the use of approximation. The intervals were chosen to be relatively large due to the fact that Padé approximations are weaker over larger intervals, and we wished to construct examples for which the genetic technique might be most applicable.

5.2 COMPARISON WITH PADÉ APPROXIMATIONS

The Padé approximation technique is parameterized by the value about which the approximation is centered, the degree of the numerator in the rational polynomial approximation, and the degree of the denominator. Using the Maple symbolic mathematics package, we calculated all Padé approximations whose numerator and denominator had a degree of 20 or less, determined their associated error and cost, and calculated their (collective) Pareto front for each of the three functions being approximated. The center of approximation was taken as the leftmost point on the interval for all functions except the square root, whose center was taken as x=1 since the necessary derivatives of sqrt(x) are not defined for x=0. Error was calculated using a Riemann integral with 1000 points. For simplicity, the cost of Padé approximations was taken only as the minimum number of multiplications/divisions required to compute the rational polynomial, as calculated by a separate Maple procedure.

The Maple procedure written to compute the cost of an approximation operated by first putting the approximation in continued-fraction form (known to minimize the number of necessary multiplications/divisions), counting the number of multiplications/divisions required to compute the approximation in this form, and then subtracting for redundant multiplications. As an example of a redundant multiplication, the function $f(x)=x^2+x^3$ when computed literally requires 3 multiplications (1 for x^2, 2 for x^3), but need be computed using only 2, since in the course of computing x^3 one naturally computes x^2.

For consistency, the candidate approximations evolved through the genetic programming technique were also evaluated (subsequent to evolution) using the Riemann integral and Maple cost procedure, and the Pareto front for this set of approximations was recomputed using the new cost and error values. Finally, it should be noted that a Padé

approximation with denominator of degree zero is identical to the Taylor series whose degree is that of the numerator, so that the Pareto fronts reported here effectively represent the potentially best (under some reasonable utility function) members of a set of 20 Taylor series and 380 uniquely Padé approximations.

5.3 AVOIDING DIVISION BY ZERO

In evolving approximations to functions, it is desirable to avoid evolving expressions which divide by zero when evaluated at some point in the target interval. Toward this end, a number of informal experiments (not reported here) were conducted to determine the way in which division by zero could best be discouraged. Initially, the division operators was defined to simply return 1 when division by zero was attempted, so that the expression `(/ X X)` would always evaluate to 1. This tended to produce many expressions which divided by zero, which of course, when evaluated through Maple, were assigned infinite error. We attempted to remedy this by assigning a high fitness penalty (10^6) to any approximation which attempted to divide by zero on any of the training examples. This approach, however, tended to stifle evolutionary progress, so that the resulting expressions were not as accurate. As a final solution, we defined the division operator to simply return the high value 10^6 if division by zero was attempted, which seemed to provide a good compromise between the loss of accuracy introduced by the fitness-penalty approach and the loss of valid approximations imposed by the original definition of our division operator.

5.4 RESULTS

All experiments involving rational polynomial approximations were performed using same settings as described for the symbolic regression experiment in section 2.5, but with a generation limit of 101 (informal experiments indicate that accurate rational polynomial approximations take a while to evolve). Note that the `/` function listed in the function set was defined to be a protected division operator which returns the value 10^6 if division by zero is attempted. In analyzing evolved approximations via Maple, any approximation which performed division by zero was discarded. To reduce the execution time of these experiments, we employed the technique suggested as a possible optimization in (Koza 1990a) of using only a subset of the available training instances to evaluate individuals at each generation. In our experiments, the subset is chosen at random for the initial generation, and selected as the subset of examples on which the previous best-of-generation individual performed the worst for all subsequent generations. The subset is assigned a fixed size for all generations; for all experiments reported in this section, the subset size was 25. Training data consisted of 100 points, uniformly spaced over the interval of approximation. Each of the three experiments reported was completed in approximately 4-5 hours on a 600 MHz Pentium III system.

The complete process through which candidate approximations were obtained and evaluated is illustrated in tables 5.1 through 5.3 below for approximations to the function ln(x). Table 5.1 presents the evolved candidate approximations calculated (as usual) as the Pareto front of the entire population history for all 50 independent runs. Table 5.2 presents the evaluation through Maple of each of these approximations. The cost and error values given in this table are those obtained through a Maple cost procedure and a Riemann integral, respectively. Table 5.3 presents the Pareto front, with respect to the new cost and error values, for the approximations in table 5.2.

Table 5.1: Evolved Approximations for ln(x).

EVOLVED LISP EXPRESSION			
ERROR	**COST**	**RUN**	**GENERATION**
1. (% (- (- 0.0682089 (+ X X)) (% X (+ 2.68242 (- (- X X) (- (% X (+ (- (% 1.17237 (+ X X)) (% -3.40938 -1.76199)) (* X 0.221107))) (- 4.6585 (+ (* X 0.221107) (* -1.43422 (- (% 1.17237 3.42463) (% 3.72341 X))))))))))) (- (* (% X -1.43422) 1.9866) (- 4.6585 X)))			
4.248	11.5	7	20
2. (% (- (- 0.0682089 (+ X X)) (% X (+ (% X 3.79818) (- (- (* (% X -1.43422) 1.9866) (- (+ (- 4.6585 X) X) X)) (% 2.91803 -3.57418))))) (- (* (% X -1.43422) 1.9866) (- (+ (- 4.6585 X) X) X)))			
7.02533	8.3	7	10
3. (% -1.32282 (* (% -0.927 X) (- (- (- -0.498825 3.39076) (% X -3.6227)) (- (- -0.498825 3.39076) 4.13266))))			
7.68202	4.2	21	60
4. (+ (% (+ (+ (* X 2.32688) (* X 0.879086)) (- (+ -3.79666 X) (+ 0.417035 2.38792))) (+ X (- (+ (* (+ 3.55251 (- (* 0.68041 -2.30094) (* 0.68041 -2.30094))) 3.06574) -1.6747) -3.63491))) 0.694754)			
9.01302	3.6	14	9
5. (+ (% (+ (+ (* X 2.32688) (* X 0.879086)) (+ -3.79666 X)) (+ X (- (+ (* (+ 3.55251 (- (* 0.68041 -2.30094) (* 0.68041 -2.30094))) 3.06574) -1.6747) -3.63491))) 0.694754)			
11.0388	3.5	14	43
6. (% (- (- 0.0682089 (+ X X)) -3.40938) (- (* (% X -1.43422) 1.9866) (- 4.6585 X)))			
12.5878	3.5	7	51
7. (% (- (- 4.6585 X) X) (- (* (% X -1.43422) 1.9866) (- 4.6585 X)))			
13.5293	3.4	7	52
8. (% (+ X -0.306864) (* (- (% (+ X (- 3.99442 -0.306864)) (* (- 3.99442 -0.306864) (- 3.99442 -0.306864))) -0.306864) 3.99442))			
14.7673	3.3	32	43
9. (- 3.70144 (% (- (- (* -4.36491 (+ -4.11252 -3.17591)) -3.66451) (+ X 3.70144)) (+ X (- 3.33522 -3.66451))))			
15.6657	1.4	37	12
10. (+ 4.70397 (% (* (+ (+ 4.70397 (* -1.64571 -3.62819)) 4.8471) -1.87643) (+ X 2.82952)))			
28.5238	1.2	5	61
11. (- 4.15036 (% (- (- (- -0.888241 -3.75332) -1.65822) (- (- -0.888241 (- -0.888241 -3.75332)) (- -0.888241 -3.75332))) X))			
43.7394	1.1	9	9
12. (- X (+ X -3.91812))			
66.7454	0.2	36	0
13. (- 3.89462 (* (- -2.99371 4.85595) 0.00228889))			
66.7454	0	19	5

Table 5.2: Maple Evaluation of Approximations for ln(x).

SIMPLIFIED MAPLE EXPRESSION	COST	ERROR
1. (.0682089-2*x-x/(7.831903406-.221107*x-x/(.586185/x-1.934959903+.221107*x)-5.34018909/x))/(-.385143144*x-4.6585)	11	3.994089354
2. (.0682089-2*x-x/(-.1218591501*x-3.842080570))/(-.385143144*x-4.6585)	6	6.798897089
3. 1.426990291*x/(4.132660+.2760372098*x)	3	7.436110884
4. (4.205966*x-6.601615)/(x+12.85128201)+.694754	2	8.743267301
5. (4.205966*x-3.79666)/(x+12.85128201)+.694754	2	10.64746639
6. (3.4775889-2*x)/(-.385143144*x-4.6585)	3	12.39955600
7. (4.6585-2*x)/(-.385143144*x-4.6585)	3	13.24060063
8. (x-.306864)/(.2159024101*x+2.154401281)	2	14.61975837
9. 3.70144-(31.77641099-x)/(x+6.99973)	3	15.58935522
10. 4.70397-29.12598131/(x+2.82952)	1	26.93968611
11. 4.15036-11.141698/x	1	39.72303025
12. 3.91812	0	64.55919780
13. 3.912587008	0	64.56236430

Table 5.3: Final Evolved Approximations for ln(x).

EVOLVED APPROXIMATION	COST	ERROR
(2) (.0682089-2*x-x/(-.1218591501*x-3.842080570))/(-.385143144*x-4.6585)	6	6.798897089
(3) 1.426990291*x/(4.132660+.2760372098*x)	3	7.436110884
(4) (4.205966*x-6.601615)/(x+12.85128201)+.694754	2	8.743267301
(10) 4.70397-29.12598131/(x+2.82952)	1	26.93968611
(12) 3.91812	0	64.55919780

Figure 5.1 (below) presents the Pareto fronts for Padé approximations and for genetically evolved approximations of the function ln(x), evaluated over the intervals [1,100]. In this figure, the dashed line connects points corresponding to Padé approximations, while the solid line connects points corresponding to genetically evolved approximations. All Padé approximations not accounted for in computing the Pareto front represented by the dashed line (i.e. all Padé approximation whose numerator or denominator has a degree larger than 20) must involve at least 20 multiplications/divisions, if only to compute the various powers of x: x, x^2, x^3, . . . x^{21}. For this reason, a dashed horizontal line at cost=20 is drawn in the figure, so that the horizontal line, combined with the dashed lines representing the Pareto front for Padé approximations with numerator and denominator of degree at most 20, represents the best case Pareto front for all Padé approximations of any degree. This same convention is followed in Figures 5.2 through 5.5, which represent the Pareto fronts for approximations to sqrt(x), arcsinh(x), exp(-x), and tanh(x), respectively, all evaluated over the interval [0,100]. The final evolved approximations (obtained after processing through Maple) for each of these functions are presented in Tables 5.4 through 5.7. For brevity, only the final evolved approximations, and not the original candidate approximations or the cost and error data obtained by evaluating them through Maple, are

given for these functions. For a full set of tables (such as that given for ln(x)) for each of these functions, see Appendix A.

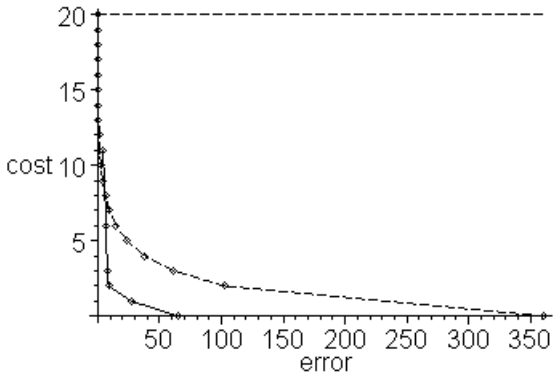

Figure 5.1: Pareto Fronts for Approximations of ln(x).

Table 5.4: Final Evolved Approximations for sqrt(x).

EVOLVED APPROXIMATION	COST	ERROR
1. x/(x/(4.78576+x/(9.17981+x/(15.39292+.04005697704*x)))+1.48335)	5	2.591348148
2. (x+.06288503787)/((x-9.04049)/(.05822627334*x+8.30072)+4.32524)+.795465	3	3.123452980
3. x/(5.5426193+.06559635887*x)+1.48335	2	8.935605674
4. .07262106112*x+3.172308452	1	32.95322345
5. 7.011926	0	195.5193204

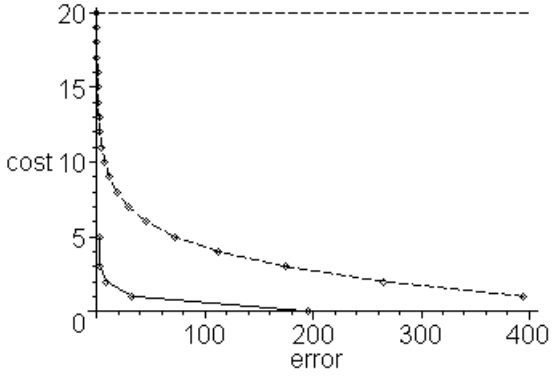

Figure 5.2: Pareto Fronts for Approximations of sqrt(x).

Table 5.5: Final Evolved Approximations for arcsinh(x).

EVOLVED APPROXIMATION	COST	ERROR
1. 1.86636*(1.277853316*x/((.3868816181*(-2.90216-x)/(-4.88586-x)+1.02145)*(-1.122792357-.3868816181*x))-.03522759767*(-1.122792357-.3868816181*x)*(x+4.86602)*(x-.269326)/(.0840785+x)+4.83551*x)/(9.684284+2.08151*x)	17	3.361399200
2. 1.86636*(.07017092454*x^2/((2*x+4.86602)*(3.111694208+4.83551*x))-.03539134480*(.2502505059-.3868816181*x)*(x+4.86602)*(x-.269326)/(.0840785+x)+	15	3.533969225

4.83551*x)/(9.684284+2.08151*x)		
3. 1.86636*(.0840785-.03522759767*(-1.122792357-.3868816181*x)*(x-.269326)+ 4.83551*x)/(9.684284+2.08151*x)	7	3.804858563
4. 2.46147/(.4180284579-4.28068*1/(-2.299172064-.7261005920*x))	3	6.596080331
5. 4.466119361*x/(18.01575130+x)+1.32282	2	7.581253733
6. 3.30409+.02369172723*x	1	25.83927515
7. 4.600931145	0	68.51916981

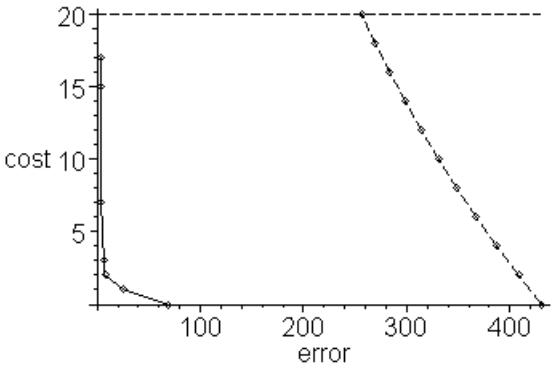

Figure 5.3: Pareto Fronts for Approximations of arcsinh(x).

Table 5.6: Final Evolved Approximations for exp(-x).

EVOLVED APPROXIMATION	COST	ERROR
1. -3.00226/(-3.00226-x^2*(x+3.70418))	4	.1651951367
2. -3.00226/(-3.00226-4.66735*x^2)	3	.2530555679
3. .950804/(x+.9878317412+x^2)	2	.2992106079
4. 0	0	1.050833194

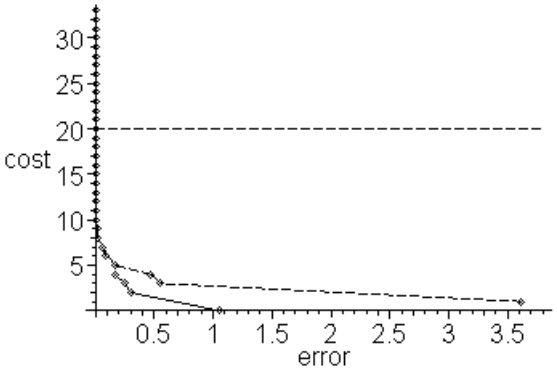

Figure 5.4: Pareto Fronts for Approximations of exp(-x).

Table 5.7: Final Evolved Approximations for tanh(x).

EVOLVED APPROXIMATION	COST	ERROR
1. x/(x+.3867602520/(x+.247353))	2	.2020989978
2. x/(.00961333+x)	1	.6362665411
3. 1.000000000	0	.7439807922

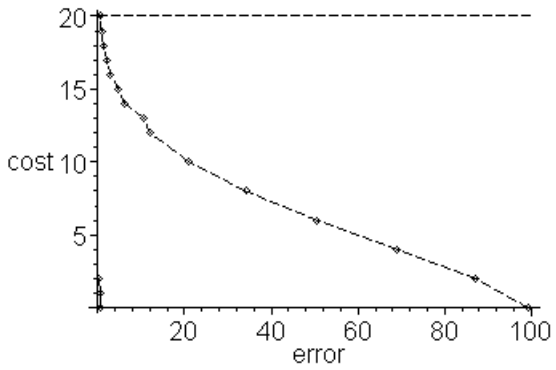

Figure 5.5: Pareto Fronts for Approximations of tanh(x).

For each of these experiments, we are interested in the genetically evolved approximations which lie to the interior of the Pareto fronts for Padé approximations, and thus are superior to Padé approximations given certain trade-offs between error and cost. As can be seen through examination of tables 5.3 through 5.7, we are able to evolve such approximations for all five functions examined in these experiments. For ln(x), we are able to obtain 5 approximations which lie to the interior of the Pareto front for Padé approximations, for sqrt(x) we are able to obtain 5 such approximations, for arcsinh(x), 7 such approximations, for exp(-x), 4 such approximations, and for tanh(x), we are able to obtain 3 such approximations, all exhibiting various trade-offs between error and cost. As can be seen from Figure 5.3, arcsinh(x) proved to be a particularly difficult function for Padé approximations to model over the given interval.

6 APPROXIMATING FUNCTIONS OF MORE THAN ONE VARIABLE

For some functions of more than one variable, it is possible to obtain a polynomial or rational polynomial approximations using techniques designed to approximate functions of a single variable; this can be done by nesting and combining approximations. For example, to obtain a rational polynomial approximation for the function f(x,y)=ln(x)*sin(y), one could compute a Padé approximation for ln(x) and a Padé approximation for cos(x) and multiply the two together. To compute a rational polynomial approximation for a more complex function such as f(x,y)=cos(ln(x)*sin(y)), one could again compute two Padé approximations and multiply them together, assign the result to an intermediate variable z, and compute a Padé approximation for cos(z). However, for functions which meet certain conditions, there is no way to compute a polynomial or rational polynomial approximation using techniques designed to compute approximations for functions of a single variable. Specifically, there is no way to use Padé approximations or Taylor series (or any similar technique) to obtain an approximation to a function which:

- Employs a non-unary operator which is not in the set {*, +, /, -}

  -and-

  - at least two of the arguments to the operator are variables.

-or-

- Contains a function of more than one variable, with at least two of its arguments being variables, which cannot be reduced to an expression which fails to meet the two criteria above.

For the function $f(x)=x^y$, for example, there is no way to use Padé approximations or Taylor series to obtain an approximation, since the variables x and y are inextricably entwined by the exponentiation operator.

In contrast, the genetic programming approach can be used on any function for which data points can be generated. To test the ability of genetic programming to evolve rational polynomial approximations for the type of function just described, an experiment was conducted to evolve approximations of the function $f(x)=x^y$ over the area 0<=x<=1, 0<=y<=1. Parameter settings were the same as described in the section on Harmonic numbers, including the generation limit of 51. Training data consisted of 100 (three dimensional) points chosen at random from the given rectangle. As in the previous section, a subset of 25 examples was used to evaluate the individuals of each generation.

The approximations returned by the genetic programming system were further evaluated through Maple. As in the previous section, a Maple procedure was used to calculate the minimum number of multiplications/divisions necessary to compute the approximation, while the error was evaluated using a double Riemann integral with 10000 points. The Pareto front for this set of approximations was then recomputed using the new cost and error values. The results of this evaluation are presented in Table 6.1.

Table 6.1: Final Evolved Approximations for x^y.

EXPRESSION	COST	ERROR
1. x/(y^2+x-x*y^3)	4	.03643611691
2. x/(y^2+x-x*y^2)	3	.04650160477
3. x/(y+x-x*y)	2	.04745973920
4. x*y-y+.989868	1	.05509570980
5. x+.13336555	0	.1401316648

The most accurate approximation evolved as a result of this experiment was $x/(y^2+x-xy^3)$. Figures 4 and 5 present graphs for the target surface $f(x)=x^y$ and for this approximation, respectively. Visually, the evolved surface is quite similar to the target function.

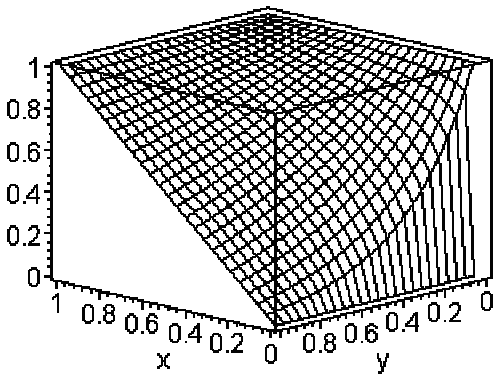

Figure 6.1: $f(x)=x^y$.

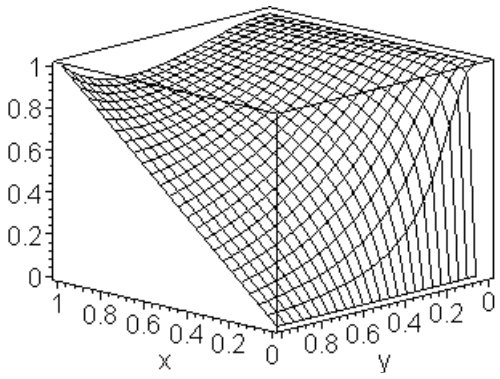

Figure 6.2: $x/(y^2+x-xy^3)$.

7 REFINING APPROXIMATIONS

In this section, experiments are presented involving the refinement of evolved approximations through approximation of their error function. We then discuss alternative ways in which genetic programming could be used for the refinement of approximations.

7.1 APPROXIMATING ERROR FUNCTION OF EVOLVED APPROXIMATIONS

Table 7.1 represents the result of evaluation through Maple of a set of candidate approximations to the function sin(x) over the interval $[0,\pi/2]$. The experiment was performed using the same GP parameters as for the symbolic regression experiment of section 2.5, and with 50 uniformly spaced points taken as training data. We can attempt to refine some of the approximations evolved in this experiment by approximating their error function, i.e. evolving approximations to the function sin(x)-a(x), where a(x) is an existing approximation to sin(x). An approximation a'(x) evolved in this way can be used to create a "refined" approximation a(x)+a'(x) which presumably will be able to more accurately model the target function sin(x).

For reference, the Pareto front for the approximations in Table 7.1 is given in Table 7.2, as calculated from the Maple cost and error data in the manner of section 5.

Table 7.1: Maple Evaluation of Approximations for sin(x).

SIMPLIFIED MAPLE EXPRESSION	COST	ERROR
1. x-x^2*(x+.2210019201*x/(-.883747041-3.021766978*x+2.16636/ (4.91791+x))) /(4.91791+x)	8	.3419994310e-3
2. x-x^2*(x+.2210019201*x/(-3.021766978*x-.6682312438))/(4.91791+x)	7	.4174535818e-3
3. x-x^2*(x-.06687879829)/(4.91791+x)	4	.001030416793
4. x^2/(-1.61855*(x+3.93155)/(x^2/(-10.50606654+x)+x)+2.562407433)+x	5	INF
5. x-x^2*(x+.3275553404*x/(x-x*(4.91791+x)))/(4.91791+x)	8	INF
6. x-x^2*(x-.06858002070)/(4.91791+x)	4	.001337163993
7. x-.137486*x^2*(.137486+x)	3	.002441754164
8. -.155492*x^3+x	3	.006453608840
9. -.3508427242*(-3.37336+x)*x	2	.02458904516
10. 2*x/(1.44734+x)	2	.03794022488
11. .0840785+.739616*x	1	.08566001170
12. .810419*x	1	.1118010031
13. x-.0758385	0	.2023284751
14. x	0	.2332524535

Table 7.2: Final Evolved Approximations for sin(x).

EVOLVED APPROXIMATION	COST	ERROR
(1) x-x^2*(x+.2210019201*x/(-.883747041-3.021766978*x+2.16636/ (4.91791+x))) /(4.91791+x)	8	.3419994310e-3
(2) x-x^2*(x+.2210019201*x/(-3.021766978*x-.6682312438))/(4.91791+x)	7	.4174535818e-3
(3) x-x^2*(x-.06687879829)/(4.91791+x)	4	.001030416793
(7) x-.137486*x^2*(.137486+x)	3	.002441754164
(9) -.3508427242*(-3.37336+x)*x	2	.02458904516
(11) .0840785+.739616*x	1	.08566001170
(13) x-.0758385	0	.2023284751

The graphs for each of the candidate approximations in Table 7.1 were examined manually. Candidate approximations (3), (7), and (8) were chosen for refinement due to the simple shapes of their error functions, and the simplicity of the formulae they represent. Figures 7.1, 7.2, and 7.3 graph the error functions for candidate approximations (3), (7), and (8), respectively.

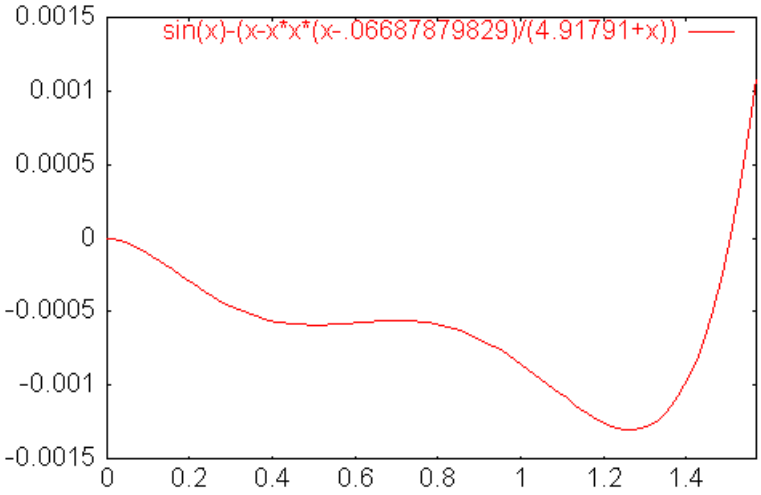

Figure 7.1: Error Function for Candidate Approximation (3).

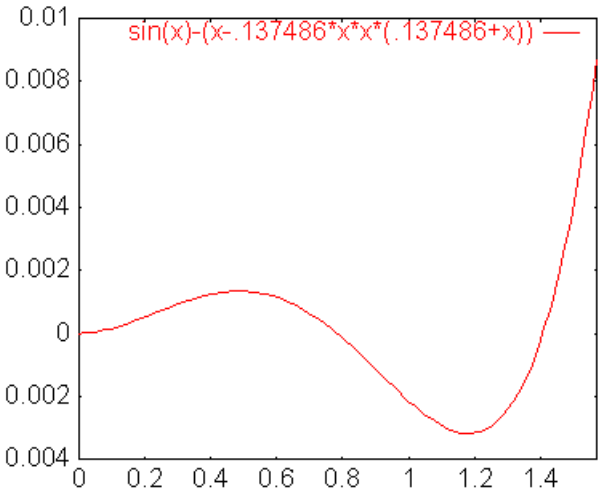

Figure 7.2: Error Function for Candidate Approximation (7).

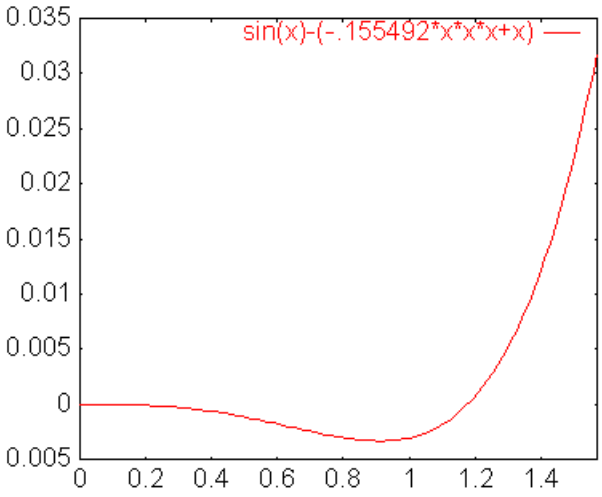

Figure 7.3: Error Function for Candidate Approximation (8).

Each of these approximations was refined using the parameter settings of section 2.5, again with 50 uniformly spaced points taken as training data. Additionally, for these experiments we use a slightly modified version of the standard adjusted fitness formula 1/(1+[*error*]) designed to maintain selection pressure when error values are small. We note that although an approximation which attains an error of 0.1 is twice as accurate as one with an error of 0.2, the standard formula will assign it an adjusted fitness which is just over 9% greater. Since we are intend to approximate functions which have very small values, it is likely that the use of this standard formula would tend to hinder evolutionary progress. To avoid this problem, we introduce an *error multiplier*, so that the adjusted fitness formula becomes 1/(1+[*error multiplier*][*error*]). In the previous example, this causes the approximation with an accuracy of 0.1 to have a fitness which is nearly twice (~1.99 times) that of the approximation whose accuracy is 0.2, which is more appropriate. For the three refinement experiments reported in this paper, the error multiplier was set to 1000.

Tables 7.3, 7.4, and 7.5 present the final evolved approximations for the error functions of candidate solutions (3), (7), and (8), respectively. Note that in these tables, both the cost of the evolved approximation and the implicit cost of the approximation being refined are indicated; an entry "3 (+4)" in the cost column indicates an approximation of cost 3 which is being used to refine an approximation of cost 4.

Table 7.3: Final Evolved Approximations for Refinement of Candidate Approximation 3 for sin(x).

EVOLVED APPROXIMATION	COST	ERROR
1. -.2932035419*x/(151.1461467*x+191.8603473)	3 (+4)	.3095090765e-3
2. -.8756705834e-3*x	1 (+4)	.3462600904e-3
3. -.5786124498e-3	0 (+4)	.4676456530e-3

Table 7.4: Final Evolved Approximations for Refinement of Candidate Approximation 7 for sin(x).

EVOLVED APPROXIMATION	COST	ERROR
1. (-1.4624515+x)*x*(-.6015476503e-2+.005000840892*x)/(2*x+1.249727)	5 (+3)	.002153461540
2. -.001581017185*x+.001566928741	1 (+3)	.002333705786
3. .3290962388e-3	0 (+3)	.002416170399

Table 7.5: Final Evolved Approximations for Refinement of Candidate Approximation 8 for sin(x).

EVOLVED APPROXIMATION	COST	ERROR
1. .001955402411*(x-1.1681)*x^4*(x+4.198426)	6 (+3)	.001728764493
2. .01587790457*(x^2-x)*x	4 (+3)	.002729466161
3. .00686666*x^2-.00686666*x	3 (+3)	.004986344588
4. -.5786124498e-3	0 (+3)	.006343650815

The experiment involving refinement of candidate approximation 3 (Table 7.3) produced three useful new approximations as a result. Approximation (1) in table 7.3, with an estimated integral error of .3095090765e-3 and cost of 7 multiplications/divisions, is unequivocally superior to approximations (1) (error .3419994310e-3, cost 8) and (2) (error .4174535818e-3, cost 7) from table 7.2, and thus expands the Pareto front created in our original experiment. Approximation (2) from table 7.3, with error .3462600904e-3 and cost 5, is also unequivocally superior to approximation (2) from table 7.2. Approximation (3) (error .4676456530e-3, cost 4) is unequivocally superior to approximation (6) from table 7.2, which has an error of .001030416793 and a cost of 4. Thus, all three approximations evolved as the final result of this experiment expand upon the Pareto front for the original set of evolved approximations represented in table 7.2.

The experiment involving refinement of candidate approximation 7 (Table 7.4) produced one new useful approximation: refined approximation (3) from this table (error .002416170399, cost 3) is unequivocally superior to candidate approrixmation 7 (error .002441754164, cost 3) from the original experiment, representing a slight improvement in accuracy.

No improvement on the Pareto front from table 7.2 was produced in the experiment involving refinement of candidate approximation 8 (Table 7.5).

In Table 7.6, we present the Pareto front for all the approximations given in tables 7.2-7.5 together, which, due to the selective nature of the Pareto front operator, is equal to the Pareto front for the union of the entire population histories of these four experiments, each of which involves 50 independent runs on a population of 500 individuals for 50 generations, for a total of 5 million individuals being processed. For clarity, approximations involving refinement have been written in brackets in the form [a(x)]+[a'(x)], where a(x) is the original approximation and a'(x) is the approximation to its error function. Four of the seven approximations in this Table 7.6 are the results of refinement of candidate approximations from the experiment reported in Table 7.1. Final refined approximations 1-3 were created through refinement of the original candidate approximation 3 (x-x^2*(x-.06687879829)/(4.91791+x)); final

approximation 4 is a refinement of original candidate approximation 7 (x-.137486*x^2*(.137486+x)), and final approximations 5-7 are from the original experiment. Had more candidate approximations been refined in this manner, or had refinement been applied multiple times (i.e. refinements of refinements), we might expect to see additional improvement on the original Pareto front. It is not clear from the results of these experiments what the limits to such potential improvements might be.

Table 7.6: Final Refined Approximations for sin(x).

EVOLVED APPROXIMATION	COST	ERROR
1. [x-x^2*(x-.06687879829)/(4.91791+x)] + [-.2932035419*x/(151.1461467*x+ 191.8603473)]	7	.3095090765e-3
2. [x-x^2*(x-.06687879829)/(4.91791+x)] + [-.8756705834e-3*x]	5	.3462600904e-3
3. [x-x^2*(x-.06687879829)/(4.91791+x)] + [-.5786124498e-3]	4	.4676456530e-3
4. [x-.137486*x^2*(.137486+x)] + [.3290962388e-3]	3	.002416170399
5. -.3508427242*(-3.37336+x)*x	2	.02458904516
6. .0840785+.739616*x	1	.08566001170
7. x-.0758385	0	.2023284751

7.2 OTHER POSSIBLE APPROACHES TO REFINEMENT OF APPROXIMATIONS

The idea of creating and refining approximations using genetic programming could be applied in three possible ways: refining evolved approximations via genetic programming, refining evolved approximations using a technique from numerical analysis such as Padé approximations or Taylor series, or refining approximations obtained through a numerical analysis technique using genetic programming. Only the first of these approaches has been considered in this section. The second of these approaches, if effective, could be of particular use if incorporated on-the-fly in the evaluation of individuals. One can imagine a rather different approach to the problem in which all evolving approximations are refined to a certain specified degree of accuracy by adding terms based on the Padé approximations or Taylor series for their error function, and fitness is taken simply as the cost of the refined expression. This would allow for the evolution of a hybrid GP/Padé or GP/Taylor approximation which attempted to minimize approximation cost given a certain desired minimum level of accuracy.

8 ATTEMPTED REDISCOVERY OF NEURAL NETWORK ACTIVATION FUNCTIONS

Neural networks (Bishop 1995) are a powerful and popular model of computation used in numerous applications in the field of maching learning (Cottrell 1990; Pomerleau 1993, Müller, Hemberger, and Baier 1997), whose creation, like that of genetic algorithms, was originally inspired by the study of biological structures in nature. Specifically, artificial neural networks attempt to model the network of neurons and synapses that exists in the human brain, using a directed acyclic graph where nodes correspond to neurons, and weighted edges correspond to synapses of various strengths. Of key importance to the operation of a neural network is the activation function which maps neuron input to output, which in early neural networks, called *perceptrons*, was simply defined as:

$$p(x) = \begin{cases} 0 \text{ if } x \leq 0 \\ 1 \text{ if } x > 0 \end{cases}$$

This function, combined with a shifted split point made possible by the use of special "bias" nodes, allows a neuron to "fire" (emit a 1) if its input reaches a certain threshold level, and to remain silent otherwise. The function works well for this purpose, though it is not differentiable. When the powerful *error backpropagation* algorithm (Bryson and Ho 1969) was introduced as an efficient and effective method of finding a set of weights or synapse strengths for a neural network, a differentiable function was required due to the nature of the weight updates performed during the training process. For this reason the "sigmoid" function was introduced, defined as:

$$sigmoid(x) = 1/(1+e^{-x})$$

This function accurately models the perceptron switching function, but is somewhat computationally expensive to calculate. If a rational polynomial approximation could be evolved which approximated the perceptron switching function with similar accuracy, it would presumably be of value to the neural network community. Toward this end, an experiment was conducted using the arithmetic function set {*,+,/,-} to evolve rational polynomial approximations to the perceptron switching function p(x). Parameter settings are the same as in other experiments involving rational polynomial approximations to functions of a single variable reported in section 5. The results of this experiment are given in table 8.1.

Table 8.1. Rational Polynomial Approximations for Perceptron Switching Function.

EVOLVED LISP EXPRESSION						
SIMPLIFIED MAPLE EXPRESSION	**ERROR**	**COST**	**RUN**	**GEN.**	**LIM $x \to -\infty$**	**LIM $x \to +\infty$**

1. (% (- (% (- 1.64235 (% -0.0636311 (% (+ -3.97977 4.4467) (+ -4.41404 X)))) (+ (% (* 1.63259 (* X (% X X))) -4.29563) (% -1.03717 0.680105))) (+ -4.33287 (% (+ X -1.16535) (+ -2.91406 2.18955)))) (- (% X -3.38984) (- (+ (% -2.40043 X) (% X X)) (% X (* -0.218665 -1.57796)))))						
((1.040824674+.1362754588*x)/(-.3800583384 *x-1.52501452)+2.724403588 +1.380243199 *x)/(2.603176691*x+2.40043/x-1)	46.8089	13.9	7	3	.53021495	.53021495
2. (% (+ X (% X X)) (- (- (+ 1.87551 (* X X)) (+ X (* 1.29322 -0.223548))) X))						
(x+1)/(2.164606745+x^2-2*x)	46.9252	3.5	40	4	0	0
3. (% X (+ -4.09024 (+ X (+ (% 4.62065 X) X))))						
x/(-4.09024+2*x+4.62065/x)	47.9418	2.3	28	0	.5	.5
4. (% (% X (+ (+ -4.5996 X) X)) 4.96521)						
.2014013506*x/(-4.5996+2*x)	49.1347	2.2	36	1	.1007006753	.1007006753
5. (% -0.12711 (+ X (- 2.97693 4.36766)))						
-.12711/(x-1.39073)	49.248	1.1	44	5	0	0
6. (% (% 0.206153 (* 2.94641 (- (* 2.07724 (+ -0.130467 2.61345)) -4.19858))) X)						
.007478093203/x	50 (≈)	1	23	8	0	0
7. (+ -1.73421 (% -3.37397 -1.56758))						
.418133102	50	0	0	0	.418133102	.418133102

The approximations evolved as a result of this experiment are not very accurate. An approximation which always outputs 0 (or for that matter any constant value between 0 and 1) would have an error of 50 over the given training set, while the best approximation given in table 8.1 has an error of just under 47. As can be seen from the graphs in Figure 8.1 below, the evolved approximations model the target function extremely poorly. This figure shows the perceptron switching function (red dotted line), the sigmoid function (green line), and four evolved approximations (using blue lines of various styles). From the solid line to the most sparsely dotted, the blue lines correspond to candidate solutions 1, 2, 3, and 4, respectively.

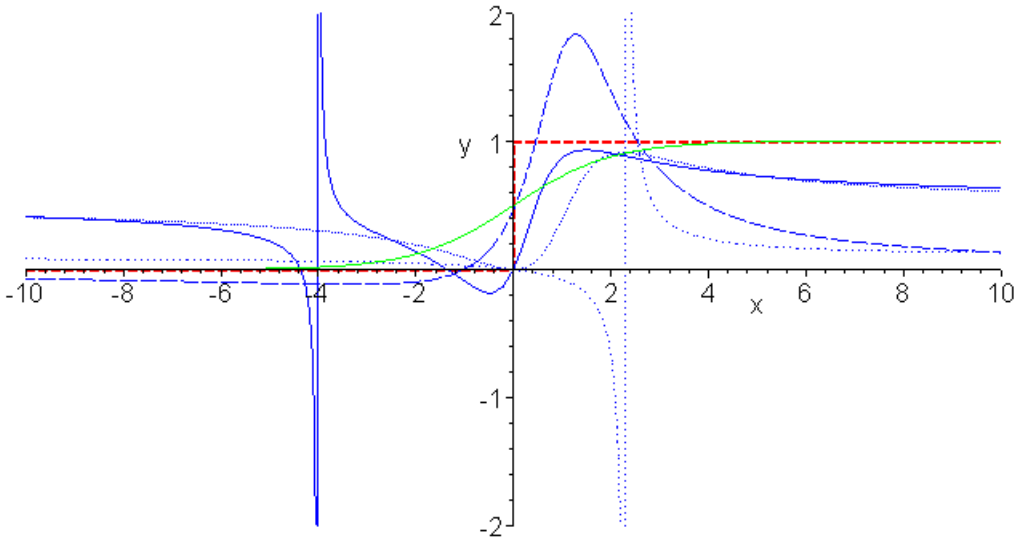

Figure 8.1: Plot of Rational Polynomial Approximations for Perceptron Switching Function

After observing the poor results of this initial experiment, the experiment was repeated using the function set {*,+,/,-,EXP} in the hopes of rediscovering the sigmoid function or perhaps a slightly more accurate variation upon it. The results of this experiment are presented in table 8.2.

Table 8.2. Approximations for Perceptron Switching Function Using Function Set {*,+,/,-,EXP}.

EVOLVED LISP EXPRESSION						
SIMPLIFIED MAPLE EXPRESSION	**ERROR**	**COST**	**RUN**	**GEN.**	**LIM x→-∞**	**LIM x→+∞**
1. `(% (% X X) (EXP (EXP (- (* 1.54927 4.28159) (EXP (- X (% -1.67409 X))))))))`						
1/exp(exp(6.633338939-exp(x+1.67409/x)))	33.0013	33.2	27	7	.850013e-330	1
2. `(EXP (* -4.69207 (EXP (- (% -0.365154 X) (- (- X -1.59932) -0.0349437)))))`						
exp(-4.69207*exp(-.365154/x-x-1.6342637))	33.8325	22.3	49	0	0	1
3. `(EXP (* -4.69207 (EXP (- (% -0.365154 X) (- X -0.0349437)))))`						
exp(-4.69207*exp(-.365154/x-x-.0349437))	35.6592	22.2	49	0	0	1
4. `(EXP (- (- X X) (EXP (* -1.07685 X))))`						
exp(-exp(-1.07685*x))	39.3448	21.2	3	1	0	1
5. `(EXP (- X (+ (EXP (- -0.0965911 X)) X)))`						
exp(-exp(-.0965911-x))	39.4363	20.3	46	0	0	1
6. `(% X (+ X (% (* (EXP (- -2.23502 X)) (EXP 3.96146)) (- (% X X) (- -4.83398 -1.19465)))))`						
x/(x+11.32361129*exp(-2.23502-x))	40.7289	14.3	42	0	0	1
7. `(% X (- X (* (+ -4.10886 X) (% (EXP (- (- X X) X)) (+ X -0.860775)))))`						
x/(x-(-4.10886+x)*exp(-x)/(x-.860775))	41.2916	13.5	0	12	0	1
8. `(% X (- X (% (EXP (* -3.78872 X)) (- X (- X X)))))`						
x/(x-exp(-3.78872*x)/x)	41.9689	13.3	21	24	0	1
9. `(% X (- (* (% -4.1818 (* -1.2804 3.06116)) (+ X 2.08762)) (% X (EXP (- X 1.40645)))))`						
x/(1.06691928*x+2.227322027-x/exp(x-1.40645))	43.3873	13.3	27	0	0	.9372780291
10. `(% X (- X (EXP (* -4.89288 X))))`						
x/(x-exp(-4.89288*x))	43.9838	12.1	35	38	0	1
11. `(% X (- X (EXP (- (- X X) X))))`						
x/(x-exp(-x))	46.33	11.3	41	27	0	1
12. `(% (EXP -1.21876) (+ (EXP (- (* 3.2873 -3.61293) X)) (EXP (% (+ 4.494 -4.93683) 2.5573))))`						
.2955964794*1/(exp(-11.87678479-x)+.8410004351)	47.8246	11.2	44	0	0	.3514819578
13. `(% (% X X) (+ (+ -1.83004 X) (% 1.74093 X)))`						
1/(-1.83004+x+1.74093/x)	48.1367	3.2	27	4	0	0
14. `(% (- X -1.13086) (+ (* X X) 2.8576))`						
(x+1.13086)/(x^2+2.8576)	48.3269	2.2	2	0	0	0

15. (% X (- X (- (EXP -4.85809) X)))						
x/(2*x-.007765301462)	49.1459	1.2	34	14	.5	.5
16. (% -4.47325 (- X (EXP 3.88363)))						
-4.47325/(x-48.6003144)	49.5743	1.1	17	41	0	0
17. (EXP -3.31538)						
.03632024426	50	0	7	0	.0363202443	.0363202443

The results of this second experiment are substantially more accurate than those of the first, as is especially apparent when examining the approximations' graphs, as presented in Figure 8.2 below. The same conventions have been used in this figure as were used in figure 8.1, namely a dotted red line for the perceptron switching function being modeled, a green line for the sigmoid function, and four blue lines (of various styles) for four of the evolved approximations. The blue lines correspond to candidate approximations 1, 2, 4, and 5, respectively. These approximations were chosen because, among the most accurate approximations, they had the most interesting graphs, and the graphs from which it was most visually apparent that the approximation was more accurate than the sigmoid function in modeling the perceptron switching function. The solid blue line, which corresponds to candidate 1, may be difficult to see because it is actually *coincident* with the heavier dotted red line. Likewise, the graph for candidate approximation 2 is clearly closer to the perceptron switching function than is the sigmoid function for all values of x. The graphs of candidate approximations 4 and 5, while further from the red line than the graph of the sigmoid function for some values of x, are clearly more accurate overall. Furthermore, as indicated in Table 8.2, each of these candidate approximations has the desired limits as x tends to positive and negative infinity.

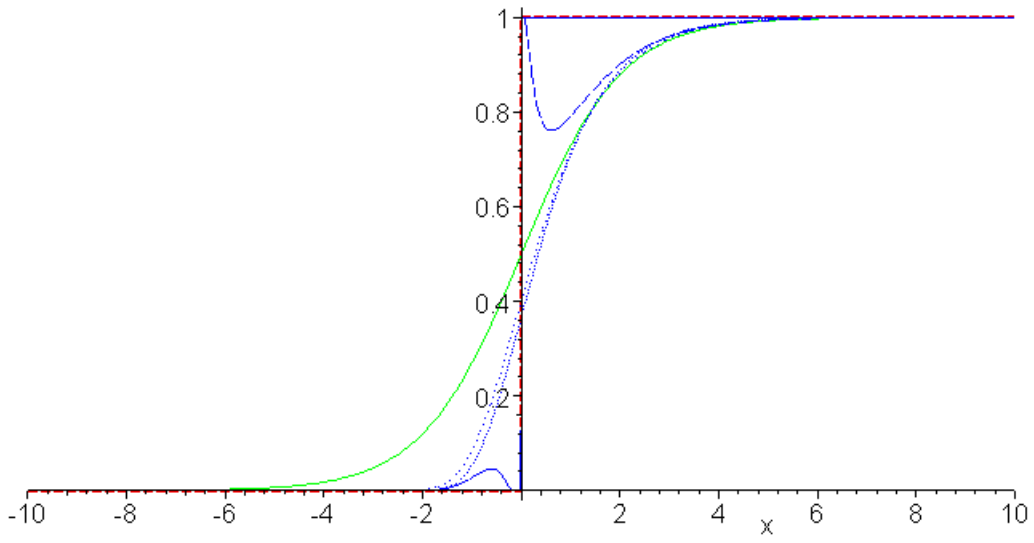

Figure 8.2: Plot of Approximations for Perceptron Switching Function Evolved Using Function Set {*,+,/,-,EXP}.

Though this experiment illustrates the power of genetic programming to evolve a variety of accurate an interesting approximations to the target function exhibits various trade-offs between cost and error, the evolved approximations are unfortunately not of great practical use. The sigmoid function, like the four evolved approximations we have been

discussing, can be made to arbitrarily accurately model the perceptron switching function simply by multiplying its argument by a large value (and thus horizontally squashing the graph). For example, were we to graph *sigmoid*(1000000*x) = $1/(1+e^{-1000000*x})$, we would see that the result is virtually indistinguishable from the target function. In a purely academic sense, candidate approximation 4 may be the most interesting because it is a slight variation on exp(-exp(-*x*)), which is a simple and somewhat elegant expression.

9 ATTEMPTED PIECEWISE APPROXIMATION OF FUNCTIONS

9.1 INTRODUCTION AND PRELIMINARY WORK

It is possible to evolve piecewise rational polynomial approximations to functions in a manner similar to that described in Section 5 by adding appropriate operators to the function set to allow for the insertion of split points or conditions. One such way of doing this would be to add the functions *IfThenElse*, *GreaterThan*, and *GreaterThanOrEqual* to the arithmetic function set {*,+,/,-}, where the three functions are defined as:

IfThenElse(i_1, i_2, i_3, . . .) = i_2 if $i_1 \neq 0$

i_3 otherwise

GreaterThan(i_1, i_2, . . .) = 1 if $i_1 > i_2$

0 otherwise

GreaterThanOrEqual(i_1, i_2, . . .) = 1 if $i_1 >= i_2$

0 otherwise

Alternatively, since all approximations will be evaluated using a fixed set of training examples (and thus there is a finite set of x-values which any splitting condition can attempt to separate) either the *GreaterThan* or *GreaterThanOrEqual* function may be omitted, without any loss (as far as evaluation against the training data is concerned) in expressive power. Preliminary experiments were performed involving piecewise rational approximation of the function ln(x) over the interval [1,100] with all three of the above functions added to the arithmetic function set, and with only *IfThenElse* and *GreaterThan* added to the arithmetic function set. An even more compact function set for evolving piecewise rational polynomial approximations can be obtained using the *Split* function, defined as:

Split(i_1, i_2, i_3, . . .) = i_2 if $i_1 >= 0$

i_3 otherwise

If this function is used in evolving approximations to a function of a single variable x, and if the argument i_1 is a linear expression involving x, the function introduces a single split point, so that, for example, *Split*(x,x,-x) is equal to $|x|$, while *Split*(x-4,x-4,4-x) is equal to $|x-4|$. Table 9.1 presents the results of experiments with piecewise rational polynomial approximation of ln(x) over the interval [0,100] using each of the function sets discussed, as well as a control experiment (from Section 5) involving non-piecewise rational polynomial approximation to this same function. For each function set, the error of the best approximation evolved over the course of 50 runs is given. Excluding the function set, the parameters for all experiments are the same, and are the same as those given in section 5.

No claim is made as to the statistical validity of these experiments in establishing the relative value of different possible function sets in evolving piecewise rational polynomial approximations; they are merely intended to help gain an intuitive idea of the relative effectiveness of different possible function sets, which is at least slightly better than the understanding that one could obtain having performed no experiments at all.

Table 9.1: Error of Best Evolved Piecewise Approximation to ln(x) Using Various Function Sets.

FUNCTION SET	ERROR OF BEST EVOLVED APPROXIMATION
{*, +, /, -, *IfThenElse*, *GreaterThan*, *GreaterThanOrEqual*}	9.39709
{*, +, /, -, *IfThenElse*, *GreaterThan*}	9.8963
{*, +, /, -, *Split*}	7.32875
{*, +, /, -}	3.994089354

The function set in which only the *Split* function is added to the set of arithmetic operators seems the most natural choice for experiments involving piecewise approximation of functions, both intuitively and based on the very little empircal evidence we have presented. Sections 9.2 and 9.3 present the results of such experiments for functions of a single variable and for functions of more than one variable, respectively.

9.2 PIECEWISE RATIONAL POLYNOMIAL APPROXIMATIONS OF FUNCTIONS OF A SINGLE VARIABLE

Our first experiment involving piecewise rational polynomial approximation of functions was performed against the function:

$$f(x) = \exp(-x\text{^}2) + \exp(-(50-x)\text{^}2) + \exp(-(100-x)\text{^}2)$$

which is designed to be especially amenable to piecewise approximation. The function is designed to have three distinct peaks, as illustrated in Figure 9.1 below.

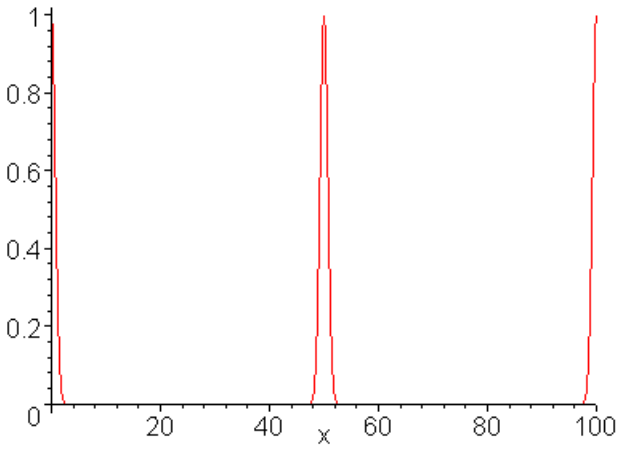

Figure 9.1: Graph of Three-Peaks Function.

Presumably, genetic programming would be employ the Split function to create two or three split points: one for each peak at the left and right ends of the interval, and (possibly) one in the center to approximate the middle peak using two separate functions. To test this hypothesis, an experiment was conducted involving 50 indepedent runs in the manner of Section 5, with parameter settings the same as in that section. The results of this experiment are presented in Table 9.2. For comparison, the same experiment was also run using only the arithmetic function set (i.e. with the Split function removed) to evolve non-piecewise rational polynomial approximations. The results of this experiment are presented in Table 9.3.

Table 9.2: Evolved Piecewise Rational Polynomial Approximations for Three-Peaks Function.

EVOLVED LISP EXPRESSION				
SIMPLIFIED MAPLE EXPRESSION	**ERROR**	**COST**	**RUN**	**GEN.**
1. (% (SPLIT (- (+ (+ 0.457015 (+ X (% (* (- 4.33287 X) (+ 4.04538 3.27998)) (- X 3.23603)))) (SPLIT X (% (- X (- 0.529038 X)) (SPLIT 2.10837 (SPLIT (% -4.97436 4.64202) X (* X 1.75619)) (- X X))) (% -0.629749 3.77529))) (* (% (- (- (% -2.44621 X) (+ X X)) (* -4.45311 (* -2.73309 -0.610828))) (% -0.122837 (+ (% X X) (- 1.65029 -0.797296)))) (* 0.865963 1.15467))) (% X (* -1.26576 (* X X))) (% (- X X) (SPLIT X (- (- 1.1623 3.43745) (* X -2.48009)) (- X X)))) (+ (% X X) (- (* X (+ (% X X) (- (* X (* -1.26576 (* X 1.75619))) (- X (+ X -1.91794))))) (SPLIT X (- X X) X))))				
piecewise([-.7900391859/x, 0 <= 209.0882933-55.12716696*x+7.32536*(4.33287-x)/(x-3.23603)+piecewise([.5694144711*(2*x-.529038)/x, 0 <= x],[-.1668081128, otherwise])-68.64941854/x],[0, otherwise])/(1+x*(-.91794-2.222915054*x^2)-piecewise([0, 0 <= x],[x, otherwise]))	2.16313	23.9	35	21

2. (% (SPLIT (- (+ (+ 0.457015 (+ X (% (* (- 4.33287 X) (+ 4.04538 3.27998)) (- X 3.23603)))) (SPLIT X (% (- X (- (% -1.6744 0.87878) X)) (* (SPLIT X X 3.05933) (- (- X (+ X -0.0929289)) (- X (SPLIT (- (- X X) -2.58934) X (- (% -2.44621 X) (+ X X))))))) (% -0.629749 3.77529))) (- X X)) (% X (* -1.26576 (* X X))) (% (- X X) (SPLIT X (- (- 1.1623 3.43745) (* X -2.48009)) (- X X)))) (+ (% X X) (- (* X (+ (% X X) (- (* X (* -1.26576 (* X 1.75619))) (- X (+ X -1.91794))))) (SPLIT X (- X X) X))))				
piecewise([-.7900391859/x, 0 <= .457015+x+7.32536*(4.33287-x)/(x-3.23603)+piecewise([10.76091507*(2*x+1.905368807)/piecewise([x, 0 <= x], [3.05933, otherwise]), 0 <= x],[-.1668081128, otherwise])],[0, otherwise])/(1+x*(-.91794-2.222915054*x^2)-piecewise([0, 0 <= x],[x, otherwise]))	2.16575	20.3	35	26
3. (% (% 1.26728 X) (+ (* (+ X (* X (SPLIT X X -3.83145))) (* X 3.21009)) (% X (% X -2.97967))))				
1.26728/(x*(3.21009*(x+x*piecewise([x, 0 <= x],[-3.83145, otherwise]))*x-2.97967))	2.166	7.3	12	65
4. (% (% -1.51509 (* (* X (- -0.299539 (SPLIT X X 2.84051))) (* 3.50093 X))) (+ (% -0.305643 (SPLIT (% -1.10858 -3.75942) (+ X X) 4.11466)) X))				
-.4327678645/(x^2*(-.299539-piecewise([x, 0 <= x],[2.84051, otherwise]))* (-.1528215/x+x))	2.19819	6.5	3	21
5. (% (% 1.26728 X) (+ (* (+ X (SPLIT X 3.60683 -3.211)) (* X X)) (+ (SPLIT X 3.12342 X) (% -4.45433 X))))				
1.26728/(x*((x+piecewise([3.60683, 0 <= x],[-3.211, otherwise]))*x^2+piecewise([3.12342, 0 <= x],[x, otherwise])-4.45433/x))	2.19898	5.5	12	33
6. (% (% 1.26728 X) (+ (* (SPLIT X X -3.83145) (* X 3.21009)) (% (- X X) X)))				
.3947802087/(x^2*piecewise([x, 0 <= x],[-3.83145, otherwise]))	2.24712	5.3	12	64
7. (% (% 1.26728 X) (+ (* (+ X (SPLIT X 3.60683 -3.211)) (+ X 0.112461)) (+ (SPLIT X 3.12342 X) (% -4.45433 X))))				
1.26728/(x*((x+piecewise([3.60683, 0 <= x],[-3.211, otherwise]))*(x+.112461)+piecewise([3.12342, 0 <= x],[x, otherwise])-4.45433/x))	2.27206	4.7	12	34
8. (% (% (- 2.03024 1.13544) (* X X)) (+ (% -3.93216 (+ (+ X X) X)) (SPLIT X 3.81191 4.9234)))				
.8948/(x^2*(-1.31072/x+piecewise([3.81191, 0 <= x],[4.9234, otherwise])))	2.31721	4.4	16	98
9. (% (% (- 2.03024 1.13544) (* X X)) (+ (% (% 4.30723 1.71773) (+ X X)) X))				
.8948/(x^2*(1.253756411/x+x))	2.31792	4.2	16	98
10. (% (% (% (* 2.03269 (% -0.220191 -1.38447)) X) X) (% X (SPLIT -3.27937 4.96887 X)))				
.3232861989/(x^2)	2.38838	4.1	31	76
11. (% (% (% 1.89657 4.83367) (* X X)) (% X X))				
.3923664627/(x^2)	2.4118	4	31	30
12. (% (% 2.03024 (+ X X)) (+ (% X X) (+ X X)))				
1.015120000/(x*(1+2*x))	2.44672	3.3	16	24
13. (% (+ -4.34812 4.73968) (- (- (+ (* X X) (SPLIT (SPLIT X X 1.27186) (+ 1.25111 -2.25089) -4.79247)) (SPLIT X (- -1.38874 X) (+ X -2.21244))) X))				
.39156/(x^2-.99978-piecewise([-1.38874-x, 0 <= x],[x-2.21244, otherwise])-x)	2.46785	2.8	4	94
14. (% 0.587939 (- (- (+ (* X X) X) -0.578478) (- X X)))				
.587939/(x^2+x+.578478)	2.56907	2.4	12	30

15. (% (- (SPLIT (- 0.688955 X) -2.23533 X) X) (SPLIT (SPLIT X -4.59014 X) X -2.3986))				
-.4169098641*piecewise([-2.23533, 0 <= .688955-x],[x, otherwise])+ .4169098641*x	2.61334	1.5	45	1
16. (- (SPLIT (% X -0.0553911) (+ (- X X) 0.864132) (SPLIT X X X)) X)				
piecewise([.864132, 0 <= -18.0534418*x],[piecewise([x, 0 <= x],[x, otherwise]), otherwise])-x	2.68114	1.5	15	69
17. (SPLIT (+ 1.28986 X) (- (SPLIT (% X -0.0553911) (+ -0.0553911 0.864132) X) X) X)				
piecewise([piecewise([.8087409, 0 <= -18.0534418*x],[x, otherwise])-x, 0 <= 1.28986+x],[x, otherwise])	2.73653	1.4	15	86
18. (- X (SPLIT (* X -1.31916) (+ (% -2.76605 2.20664) X) X))				
x-piecewise([-1.253512127+x, 0 <= -1.31916*x],[x, otherwise])	2.79878	1.3	8	4
19. (- X (SPLIT (+ X -1.56301) X (+ (- (SPLIT X X 3.03339) 2.19413) 1.54775)))				
x-piecewise([x, 0 <= x-1.56301],[piecewise([x, 0 <= x],[3.03339, otherwise])-.64638, otherwise])	2.80952	0.6	17	11
20. (SPLIT (+ (+ -0.934324 X) (+ (+ X (% -1.76778 -2.99677)) X)) (- X X) 1.4153)				
piecewise([0, 0 <= -.3444288796+3*x],[1.4153, otherwise])	2.96057	0.6	22	57
21. (SPLIT (+ (- X 1.7687) (SPLIT 4.02097 X -2.12455)) (- X X) (* -0.124973 -2.74895))				
piecewise([0, 0 <= 2*x-1.7687],[.3435445284, otherwise])	3.20173	0.5	45	79
22. (SPLIT (+ -0.578478 X) (- X X) (% 0.694754 2.33543))				
piecewise([0, 0 <= -.578478+x],[.2974844033, otherwise])	3.24779	0.3	12	6
23. (- -2.15964 -2.15964)				
0	3.54527	0	17	58

Table 9.3: Evolved Rational Polynomial Approximations for Three-Peaks Function.

EVOLVED LISP EXPRESSION				
SIMPLIFIED MAPLE EXPRESSION	**ERROR**	**COST**	**RUN**	**GEN.**
1. (% (% (- 1.133 -0.999634) (+ (* (* X X) (* 1.72414 -3.65444)) X)) (% (* (% 0.489669 (% 4.28159 X)) (- X -2.6223)) (% (* X (% (% 4.65361 (% X 0.084994)) (+ (* (* X X) 0.084994) X))) -0.963927)))				
-7.651611353/((-6.300766182*x^2+x)*x*(x+2.6223)*(.084994*x^2+x))	2.16563	15.3	47	38
2. (% (% X (* (+ (% X -0.434431) (* -1.53798 X)) (+ (+ X X) (* (* -3.96634 X) X)))) (% (- (* X X) (% X 1.56423)) X))				
-.2604274151*x/((2*x-3.96634*x^2)*(x^2-.6392921757*x))	2.16873	10.4	33	6
3. (% (% (% (* 2.33695 -1.60756) X) (* X X)) (- X (+ (- X (% -2.52007 X)) (* (- (- 3.85037 -2.4218) -1.70888) X))))				
-3.756787342/(x^3*(-7.98105*x-2.52007/x))	2.18699	6.3	18	71
4. (% (% (% 0.385601 (+ (* X -3.63643) X)) X) (- (% (* 1.08692 3.04956) X) (* X 3.67092)))				
-.1462587666/(x^2*(3.314627755/x-3.67092*x))	2.21662	6.2	35	86
5. (% (% (- X -4.59197) (* X (* X (- X -4.09421)))) (+ X (+ X (% X X))))				

(x+4.59197)/(x^2*(x+4.09421)*(2*x+1))	2.23272	5.4	27	34
6. (% (% (% (% 0.771661 -0.686514) (* X X)) X) (- (% (+ X -0.807978) X) 3.44417))				
-1.124028061/(x^3*((x-.807978)/x-3.44417))	2.24471	5.2	28	82
7. (% (% (% 0.441145 X) X) (+ X (% (- X X) X)))				
.441145/(x^3)	2.30284	4.2	31	77
8. (% (% (% 0.441145 X) X) (+ X (% X X)))				
.441145/(x^2*(x+1))	2.35169	4.1	31	36
9. (% (% 0.682546 X) (+ (+ X X) (% (- X X) X)))				
.341273/(x^2)	2.38172	3.3	41	66
10. (% (% 0.441145 X) (+ X (% (% 0.441145 2.01376) X)))				
.441145/(x*(x+.2190653305/x))	2.41646	3.1	31	45
11. (% 0.124973 (+ (- -4.60906 X) (+ (- 4.57976 -0.145116) (* X (+ X X)))))				
.124973/(.115816-x+2*x^2)	2.52221	2.4	10	12
12. (* (% 0.0392163 (- (- -0.0227363 X) X)) (+ -3.9993 3.39534))				
-.02368507655/(-.0227363-2*x)	2.60907	2.2	45	16
13. (% -4.35698 (+ -3.71548 (* (% (- 3.97794 -4.26359) (* -0.105136 0.0492874)) X)))				
-4.35698/(-3.71548-1590.451702*x)	2.72272	2.1	4	3
14. (% (% 4.80377 (* (+ 3.25709 (+ 1.82516 3.97305)) 4.6823)) (+ 0.103916 X))				
.1132974478/(.103916+x)	2.93413	1.1	23	11
15. (- -3.92605 -3.92605)				
0	3.54527	0	5	16

The best approximation in Table 9.2 achieves ever-so-slightly higher accuracy than that of Table 9.3, indicating that the inclusion of the Split function has at least in some way helped improve performance on this problem. However, none of the approximations in Table 9.2 do exactly what was intended of them. When graphed, it is clear that none of the approximations even attempt to account for all three peaks. Moreover, of the 22 evolved candidate approximations, only approximations 17, 20, 21, and 22 employ the *Split* function as the root node of their LISP expression, which is the only way in which meaningful "split points" can be introduced. Candidate approximation 17 uses the expression `(+ 1.28986 X)`, which introduces a split point at x=-1.28986, which is outside the interval over the which the target function is being approximated. Candidate approximations 20, 21, and 22 model the first peak by using a high constant value for one part of the function, and a low constant value for the rest, introducing split points at x=.3444288796, x=1.7687, and x=.578478, respectively. Though this behavior is on the right track, it is not exactly what we had hoped for. For this reason, a second set of experiments was conducted using a modified "two peaks" function. It was hoped that this simpler nature of this function would provide a smoother learning surface for the GP algorithm. Also, this function was defined in a way that did not require squaring the x-values (as was done with the three peaks function), so that the peaks are broader and therefore a more significant source of error, thereby giving a greater fitness advantage to functions which attempted to approximate them. The two-peaks function is defined as:

$f(x) = \exp(-x) + \exp(-(100-x))$

A graph of this function is given in Figure 9.2.

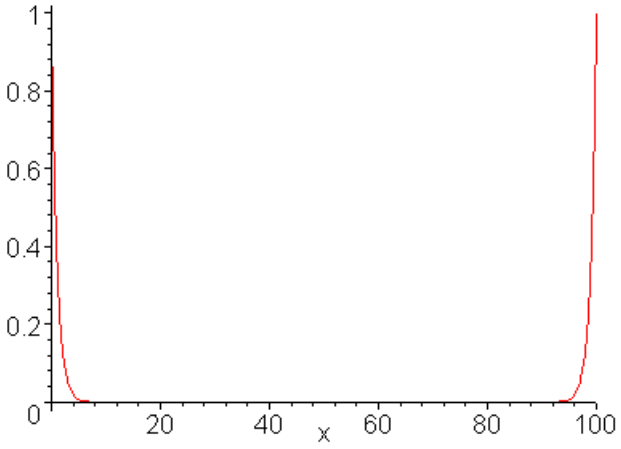

Figure 9.2: Graph of Two-Peaks Function.

Tables 9.4 and 9.5 present the results of the experiments conducted with the two-peaks target function, using the arithmetic function set with the *Split* function added, and using the arithmetic function set alone, respectively.

Table 9.4: Evolved Piecewise Rational Polynomial Approximations for Two-Peaks Function.

EVOLVED LISP EXPRESSION				
SIMPLIFIED MAPLE EXPRESSION	**ERROR**	**COST**	**RUN**	**GEN.**
1. (% (- (SPLIT (% (- (SPLIT (- (SPLIT (+ -1.38874 X) (* (- (+ -2.16361 0.870846) (+ X (+ (- X X) X))) (- X (+ -1.40309 X))) (SPLIT (% (% X X) (- 3.16309 -2.9574)) (SPLIT X -2.57225 X) (% X -2.63237))) (- X X)) X X) (- X X)) (+ (* (- X X) -3.38618) (% X (* (+ X -4.39787) X)))) (- (% (- (+ X 1.51387) (- 2.71203 (+ X 1.51387))) X) -1.18305) (% X X)) (- 0.945921 (SPLIT (% (- X X) (- 3.16309 -2.9574)) (SPLIT X -2.57225 X) (* (- (- (% (% (- X (% (% X -2.63237) (- X (* -1.9805 X)))) (- (% 2.39982 2.92718) -4.40153)) -0.314188) -1.18305) (- 2.71203 (+ X X))) (SPLIT (* (* (- (* X 0.0511185) (- 2.71203 (+ X X))) (+ (+ X -2.87652) (% X -0.093234))) X) X (* (- X (- (SPLIT (* X X) (SPLIT (% (- X X) (- 3.16309 -2.9574)) (SPLIT X -2.57225 X) (SPLIT (* (- 3.48628 -3.75149) X) (+ (- 3.16309 -0.662709) (SPLIT X -0.0630207 X)) (SPLIT X 4.10611 (% (+ 1.44215 (% X X)) (- X X))))) (% X X)) (- X (% X -2.63237)))) X)))))) (+ (* (+ (+ X 1.51387) (* (- (* (SPLIT (SPLIT -1.18305 2.30125 X) (- X X) X) 0.0511185) (- 2.71203 (+ X 1.51387))) (SPLIT (* (- 3.48628 -3.75149) X) (* (- X (% (% X -2.63237) (+ 0.963317 -1.9805))) X) (- -0.233924 X)))) -3.38618) (% X (* X X))))				
[Division by zero]	0.611535	45.6	32	93

2. (% (- (SPLIT (* (SPLIT (SPLIT -1.18305 2.30125 X) (- X X) X) 0.0511185) (% X X) (% X X)) (- 0.945921 (SPLIT (% (- X X) (- 3.16309 -2.9574)) (SPLIT X -2.57225 X) (- X X)))) (+ (* (+ (+ X 1.51387) (* (- (* (SPLIT (SPLIT -1.18305 2.30125 X) (* (- (* (SPLIT (SPLIT -1.18305 2.30125 X) (- X X) X) 0.0511185) (- 2.71203 (+ (SPLIT -0.580004 -2.15323 X) X))) (SPLIT (SPLIT (- X X) X (SPLIT X -2.57225 X)) X (+ (* (+ (+ X 1.51387) (* X X)) -3.38618) (+ (- 3.48628 (% X (* X X))) (- 3.16309 -2.9574))))) X) 0.0511185) (- 2.71203 (+ X 1.51387))) (SPLIT (* (- 3.48628 -3.75149) X) (* (- X (% (% X -2.63237) (+ 0.963317 -1.9805))) X) (% 1.57064 -0.093234)))) -3.38618) (% X (* X X))))				
(piecewise([1, 0 <= .0511185*piecewise([0, 0 <= x],[x, otherwise])],[1, otherwise])-.945921+piecewise([-2.57225, 0 <= x],[x, otherwise]))/(-3.38618*x-5.126236317-3.38618*(.0511185*piecewise([(.0511185*piecewise([0, 0 <= x],[x, otherwise])-2.71203+2*x)*piecewise([x, 0 <= x],[-3.38618*x+4.480533683-3.38618*x^2-1/x, otherwise]), 0 <= x],[x, otherwise]) -1.19816+x)*piecewise([.6265315029*x^2, 0 <= 7.23777*x],[-16.84621490, otherwise])+1/x)	0.613495	23.6	32	98
3. (% (- (SPLIT X (% X X) (% X X)) (- 0.945921 (SPLIT (% (- X X) (- 3.16309 -2.9574)) (SPLIT X -2.57225 X) (* (- (- (* (+ X 3.14234) X) -1.18305) (- 2.71203 (+ X X))) (- 3.16309 -0.662709))))) (+ (* (+ (+ X 1.51387) (* (- (* (SPLIT (SPLIT -1.18305 2.30125 X) (- X X) X) 0.0511185) (- 2.71203 (+ X 1.51387))) (SPLIT (* (- 3.48628 -3.75149) X) (* (- X (% (% X -2.63237) (+ 0.963317 -1.9805))) X) (- -0.233924 X)))) -3.38618) (% X (* X X))))				
(piecewise([1, 0 <= x],[1, otherwise])-.945921+piecewise([-2.57225, 0 <= x],[x, otherwise]))/(-3.38618*x-5.126236317-3.38618*(.0511185*piecewise([0, 0 <= x],[x, otherwise])-1.19816+x)*piecewise([.6265315029*x^2, 0 <= 7.23777*x],[-.233924-x, otherwise])+1/x)	0.625353	17.3	32	92
4. (% (- (SPLIT (- X (* (+ (+ X 1.51387) (* (- (SPLIT 1.81967 3.43959 X) (- 2.71203 (+ X X))) (SPLIT (* (- 3.48628 -3.75149) X) X (- -0.233924 X)))) -3.38618)) (% X X) (% X X)) (- 0.945921 (SPLIT 3.49727 (SPLIT X -2.57225 X) (+ -2.31895 X)))) (+ (* (+ (+ X 1.51387) (* (- (* (+ 0.963317 -1.9805) 0.0511185) (- 2.71203 (+ X 1.51387))) (* X X))) -3.38618) (% X (* X X))))				
(piecewise([1, 0 <= 4.38618*x+5.126236317+3.38618*(.72756+2*x)*piecewise([x, 0 <= 7.23777*x],[-.233924-x, otherwise])],[1, otherwise])-.945921+piecewise([-2.57225, 0 <= x],[x, otherwise]))/(-3.38618*x-5.126236317-3.38618*(-1.250156869+x)*x^2+1/x)	0.637342	13.2	32	84
5. (% (% (% (- (% -4.75036 (SPLIT X 1.88131 1.51784)) (% (- X (% X X)) (+ (SPLIT 2.82617 X X) (% X X)))) (* 2.4868 X)) X) (- (% (+ -2.6455 X) X) X))				
.4021232106*(-4.75036/piecewise([1.88131, 0 <= x],[1.51784, otherwise])-(x-1)/(x+1))/(x^2*((-2.6455+x)/x-x))	0.63978	9.7	2	22
6. (% (SPLIT -4.41527 (* -0.992004 0.566576) (SPLIT (% X X) (% (% -3.23359 1.57704) X) (* -3.64986 X))) (+ (* X (+ (% (% -3.23359 1.57704) X) (* X -1.40706))) (% -1.96829 X)))				
-2.050417237/(x*(x*(-2.050417237/x-1.40706*x)-1.96829/x))	0.63984	8.4	28	46
7. (% (% (SPLIT -4.46471 -0.345317 4.12626) (+ X X)) (+ (+ (* (SPLIT 1.4388 X X) X) X) (% (SPLIT 1.61702 (+ 2.35191 X) (SPLIT -3.47682 X 2.21397)) X)))				
2.063130000/(x*(x^2+x+(2.35191+x)/x))	0.646409	4.7	1	39
8. (% (% -0.337077 X) (- (SPLIT X (% (% (+ 1.43941 -1.91916) (SPLIT X -2.85119 -1.80105)) (+ X X)) X) X))				
-.337077/(x*(piecewise([-.239875/(piecewise([-2.85119, 0 <= x],[-1.80105, otherwise])*x), 0 <= x],[x, otherwise])-x))	0.706085	4.4	12	78
9. (% (% -0.337077 X) (- (% (% 0.179296 X) (+ X X)) X))				
-.337077/(x*(.089648/(x^2)-x))	0.709234	4.2	12	57

10. (% (% -0.337077 X) (- (% 0.1854 (+ X X)) X))				
-.337077/(x*(.0927/x-x))	0.709357	3.2	12	85
11. (% (% -0.337077 X) (- (% -0.337077 X) X))				
-.337077/(x*(-.337077/x-x))	0.830719	3.1	12	18
12. (% (- (% 3.69564 2.38304) -0.599536) (- -0.438093 (- (- (- (+ (SPLIT 4.67009 2.62566 1.4156) (* -2.21183 2.36198)) X) (SPLIT X (* (SPLIT 0.708792 X X) (+ X X)) (SPLIT 4.67009 2.62566 1.4156))) X)))				
2.150345051/(2.160545223+2*x+piecewise([2*x^2, 0 <= x],[2.62566, otherwise]))	0.875509	2.7	22	12
13. (% (% 3.70724 2.9986) (- (* X X) (- (SPLIT 2.97632 -1.1272 X) (SPLIT (% 3.70724 2.9986) X X))))				
1.236323618/(x^2+1.1272+x)	1.04709	2.4	33	23
14. (% (+ (- X X) (+ (SPLIT (- 0.86993 X) (+ X 0.81164) (- X X)) (SPLIT X (- X X) (- X X)))) (+ X (- 0.86993 X)))				
1.149517777*piecewise([x+.81164, 0 <= .86993-x],[0, otherwise])+ 1.149517777*piecewise([0, 0 <= x],[0, otherwise])	1.23096	2.2	19	46
15. (% (- (SPLIT (- 0.688955 X) -2.23533 X) X) (SPLIT (SPLIT X -4.59014 X) X -2.3986))				
-.4169098641*piecewise([-2.23533, 0 <= .688955-x],[x, otherwise])+ .4169098641*x	1.23202	1.5	45	1
16. (% (+ 0.63921 -0.657826) (SPLIT (- (% 1.90847 2.17307) X) (+ 0.63921 -0.657826) X))				
-.018616/piecewise([-.018616, 0 <= .8782367802-x],[x, otherwise])	1.26034	1.2	5	57
17. (SPLIT (SPLIT -3.11914 X 4.67284) (SPLIT (- 0.86993 X) (- 0.86993 X) (- X X)) (% 2.07846 2.52098))				
piecewise([.86993-x, 0 <= .86993-x],[0, otherwise])	1.29402	0.6	19	96
18. (SPLIT (- 0.86993 X) (+ X 0.81164) (- X X))				
piecewise([x+.81164, 0 <= .86993-x],[0, otherwise])	1.35231	0.4	19	46
19. (SPLIT (+ X -1.64998) (- X X) 0.637684)				
piecewise([0, 0 <= x-1.64998],[.637684, otherwise])	1.42819	0.3	29	0
20. (- (SPLIT (+ -0.191504 X) X 0.693228) X)				
piecewise([x, 0 <= -.191504+x],[.693228, otherwise])-x	1.47073	0.3	2	1
21. (- (* -1.87399 1.96921) (* -1.87399 1.96921))				
0	2.16395	0	31	68

Table 9.5: Evolved Rational Polynomial Approximations for Two-Peaks Function.

EVOLVED LISP EXPRESSION SIMPLIFIED MAPLE EXPRESSION	ERROR	COST	RUN	GEN.
1. (% (* 0.695669 -1.64602) (+ (* (+ (+ X X) (- (+ (+ X X) (- 0.774102 X)) (% (% (- X (% X X)) (* X -1.64602)) (- (% (* (% (* (- X X) X) (- (+ (+ X X) (+ X X)) X)) -4.80651) (* (- -3.93582 1.68722) (* 2.84936 X))) (- -3.93582 (- X X)))))) (* (* 2.84936 X) (% (- (* (% (* 0.695669 X) (- (% (- -3.93582 (- -3.93582 (% X 4.42778))) (% (% (% 4.49705 (- X X)) (+ 2.40349 X)) (- (* (- X (* (- X X) X)) X) (* 0.695669 -1.64602)))) (- -3.93582 (% (% (% -3.93582 (* 0.695669 -1.64602)) (- 2.84936 X)) (% (* 0.695669 -1.64602) (- (% 4.49705 (% X 2.98334)) (* 0.695669 -1.64602))))))) -4.80651) 1.68722) (- (% -3.93582 (* 0.695669 -1.64602)) (- -3.93582 (- 1.68722 X)))))) (* 0.695669 -1.64602)))				
[Division by zero]	0.614663	33.1	25	96
2. (% (% (- X -3.81893) X) (+ (+ -3.76308 (+ (- (% (% 1.96249 -3.13532) (- X -3.81893)) (* X (- (* (+ X -2.02139) (% (* (* X X) -3.13532) X)) (* X X)))) (- (% 4.30509 X) (+ -3.76308 (+ -3.76308 (- 3.53114 (+ X 4.97314))))))) 4.93927))				
(x+3.81893)/(x*(10.14435-.6259297297/(x+3.81893)-x*(-3.13532*(x-2.02139)*x-x^2)+4.30509/x+x))	0.614685	11.3	24	60
3. (% (% 3.93002 X) (+ (% (- X (+ (% 2.56737 (% -1.32649 2.35435)) (+ (- X 1.69881) X))) X) (+ (% -2.32597 (- (% -2.622 (% 2.59026 0.887936)) X)) (- (+ 2.59026 X) (* -2.91101 (% (* (+ (- (% 2.56737 (% -1.32649 2.35435)) (* 0.0117496 X)) (+ X 3.93002)) (+ (+ X X) (% -1.32649 2.35435))) 2.21335))))))				
3.93002/(x*((-x+6.255563206)/x-2.32597/(-.8988164091-x)+2.59026+x+1.315205458*(-.626733206+.9882504*x)*(2*x-.5634209017)))	0.632158	9.4	37	59
4. (% (% 4.80377 (* (% (+ (* 4.89746 4.21323) X) (* -2.16086 (* 2.07724 (+ -0.130467 2.61345)))) X)) (- (% (- -1.18122 -2.16086) (* X (+ X (- X 3.05658)))) (+ 4.81017 (* X X))))				
-53.53887720/((20.63412540+x)*x*(.97964/(x*(2*x-3.05658))-4.81017-x^2))	0.633389	7.5	23	95
5. (% 2.39189 (+ (+ (+ (% (% 0.451827 3.1933) -4.30113) (* (+ (* X X) (* X X)) X)) (+ X 3.38496)) (+ -0.956603 X)))				
2.39189/(2.395460485+2*x^3+2*x)	0.648203	4.7	28	71
6. (% -3.00226 (- -3.00226 (* (* (+ (% X (- (+ X X) (% -4.30296 -3.85739))) (+ X 2.61834)) X) X)))				
-3.00226/(-3.00226-(x/(2*x-1.115510747)+x+2.61834)*x^2)	0.660497	4.5	30	53
7. (% (+ (- -1.37379 -3.94192) (+ 0.651112 -1.53127)) (+ (+ (- X (- X X)) -2.32139) (+ (- X -3.95962) (* X (* X X)))))				
1.687972/(2*x+1.63823+x^3)	0.661793	3.6	34	6
8. (% -3.00226 (- -3.00226 (* (* (+ X 3.70418) X) X)))				
-3.00226/(-3.00226-(x+3.70418)*x^2)	0.667633	3.2	30	12
9. (% (% (* 0.603198 0.487533) X) (+ X (% (% -1.29353 4.86206) X)))				
.2940789305/(x*(x-.2660456679/x))	0.738883	3.1	22	20
10. (% -0.275124 (- (- X (* X X)) (- X -0.271767)))				
-.275124/(-x^2-.271767)	0.884859	2.3	42	38
11. (% 0.357524 (- (* X X) (% -1.66433 4.13755)))				
.357524/(x^2+.4022501239)	0.938956	2.1	35	5
12. (% -0.275124 (- (- X X) (- X -0.271767)))				

-.275124/(-x-.271767)	1.62963	1.3	42	13
13. (% -0.275124 (- -0.0401318 (- X -0.271767)))				
-.275124/(-.3118988-x)	1.74011	1.2	42	31
14. (% (% 4.80377 (* (+ 3.25709 (+ 1.82516 3.97305)) (* 2.07724 (+ -0.130467 2.61345)))) (+ (+ -0.0151067 0.218665) X))				
.102853468/(.2035583+x)	1.75921	1.1	23	1
15. (- (+ (* -1.28956 0.906858) -1.31764) (+ (* -1.28956 0.906858) -1.31764))				
0	2.16395	0	3	77

As in the previous set of experiments, the most accurate approximation evolved using the function set with the *Split* function included is ever-so-slightly more accurate than that evolved using only the arithmetic function set, as indicated in tables 9.4 and 9.5, respectively. Of the 21 candidate approximations given in Table 9.4, only approximations 17, 18, and 19 use the *Split* function as the root node of their LISP expression tree, and thus are the only functions that truly make use of split points in approximating the target function. Approximations 17 and 18 use a combination of a linear and constant curve, split at the point x=.86993. Approximation 19 uses a combination of two constant curves, split at the point x=1.64998. As before, all of the evolved approximations attempt to model only the first peak.

9.3 PIECEWISE RATIONAL POLYNOMIAL SURFACE APPROXIMATIONS

The piecewise approximation technique described in the previous subsection can also be applied to functions of more than one variable. To test the feasability of piecewise rational polynomial surface approximation in this manner, an experiment was conducted involving the evolution of approximations to a hemispherical surface defined over the rectangular interval 0<= x<= 1, 0<=y<=1, centered at the point (0.5, 0.5), and with a radius of 0.5. Training data was taken as 100 points selected at random from within this interval. Any point whose coordinates were outside the hemisphere was considered to have a zero function value. Parameter settings were the same as those described in section 5, including the use of a randomly chosen training subset of size 25. The candidate approximations evolved as a result of this experiment are presented in Table 9.6.

Table 9.6: Evolved Rational Polynomial Approximations for Hemispherical Surface.

EVOLVED LISP EXPRESSION				
SIMPLIFIED MAPLE EXPRESSION	**ERROR**	**COST**	**RUN**	**GEN.**
1. (* Y (+ (* -2.17917 (- (* X X) X)) (- 0.897397 Y)))				
y*(-2.17917*x^2+2.17917*x+.897397-y)	9.62479	3.3	10	14
2. (% X (+ (* X (% X Y)) Y))				
x/(x^2/y+y)	10.404	3.1	4	18
3. (- Y (SPLIT Y (* (SPLIT -4.31455 (- X Y) Y) (* Y Y)) (- X Y)))				
y-piecewise([y^3, 0 <= y],[x-y, otherwise])	10.7249	2.5	15	99
4. (% (- (SPLIT X Y -4.32585) (* Y Y)) (% -1.82363 -3.71456))				
2.036904416*piecewise([y, 0 <= x],[-4.32585, otherwise])-2.036904416*y^2	11.2113	2.2	22	71
5. (* Y (- 1.15711 Y))				

y*(1.15711-y)	12.2005	1.1	46	1
6. (SPLIT (+ (SPLIT Y X 0.825678) (- (SPLIT Y (SPLIT (+ Y Y) X Y) 1.13819) (+ 1.74856 -1.34632))) (+ 1.74856 -1.34632) X)				
piecewise([.40224, 0 <= piecewise([x, 0 <= y],[.825678, otherwise])+ piecewise([piecewise([x, 0 <= 2*y],[y, otherwise]), 0 <= y],[1.13819, otherwise])-.40224],[x, otherwise])	13.8525	0.7	5	88
7. (SPLIT (- 0.406964 X) (+ 0.0483718 X) 0.406964)				
piecewise([.483718e-1+x, 0 <= .406964-x],[.406964, otherwise])	13.9652	0.3	25	64
8. (SPLIT (- X (% 1.52852 3.35658)) (% 1.52852 3.35658) X)				
piecewise([.4553801786, 0 <= x-.4553801786],[x, otherwise])	15.2826	0.2	11	10
9. (* -2.17917 (% 0.0831629 -0.635853))				
.2850125687	15.6818	0	20	37

These approximations are not very accurate. The best evolved approximation has an average difference of approximately 0.096 from the target surface, where the hemisphere itself is only of radius 0.5. When graphed, the best evolved surfaces does not particularly resemble a hemisphere. Of the 9 evolved candidate approximations, only approximations (6), (7), and (8) employ split points, or rather split lines, in the manner we desire. Candidate approximation (6) uses a nested split condition which, for all points in the given interval, has only one outcome. Candidate approximations (7) and (8) create split lines at x=.406964 and x=.4553801786, which represents a somewhat reasonable attempt to cut the hemisphere in half. In both cases, however, the approximations do not take advantage of these split lines in the manner we would expect.

9.4 3-D SURFACE GENERATION

Were the experiments in section 9.3 to prove were successful, the technique would have possible applications for 3-D graphics rendering, in which it is important to represent complex surfaces in a compact and easily computable form. Many existing rendering applications make use of a polygon mesh for this purpose; the evolution of piecewise rational polynomial approximations represents a possibility for a degree of smoothness and accuracy not possible using any reasonable number of polygons.

One way to improve upon the results presented in this section might be to employ a constrained syntax (and corresponding constrained crossover operator) wherein the *Split* function can only appear (a) at the root or (b) as the second or third argument to another *Split* function. This would force split points to be employed in evolving individuals in the manner that is desired for the final evolved piecewise approximations.

10 FUTURE WORK

The work presented in this thesis suggests a number of possible extensions. First, as stated in the section on refinement of evolved approximations through approximation of their error function (Section 7.1), refinement in this way could be applied iteratively to produce refinements of refined approximations, and the limits to which this iteration is effective could be examined. As discussed in section 7.2, refinement of approximations could also be applied in at least two ways not considered in this report: refining evolved approximations using a technique from numerical analysis such as Padé approximations or Taylor series, and refining approximations obtained through a numerical analysis technique using genetic programming. Were the former approach to prove effective, it could be incorporated on-the-fly in the evaluation of individuals. One can imagine a rather different approach to the problem in which all evolving approximations are refined to a certain specified degree of accuracy by adding terms based on the Padé approximations or Taylor series for their error function, and fitness is taken simply as the cost of the refined expression. This provides a second possible extension. Third, although our experiments involving GP parameter optimization proved ineffective, there is no reason to think that the default set of parameters we adapted from (Koza 1992) are optimal for this class of problems, so that alteration of parameter settings (whether based on more suggestive empirical evidence or intuitive judgment) represents one possible potential for improvement on the results presented in this paper. These results could also presumably be improved by using additional computational power and memory, and by employing a genetic programming system which allows for automatically defined functions (Koza 1994) and architecture-altering operations (Koza, Andre, Bennett, and Keane 1999).

Perhaps the ideal application of this technique would be to perform the equivalent of conducting the Harmonic number experiment prior to 1734, the year that Leonhard Euler established the limiting relation

$$\lim x \to \infty \; H_n - \ln(n) \equiv \gamma$$

which defines Euler's constant (Eulero 1734). Such a result would represent "discovery" of an approximation formula in the truest sense, and would be a striking and exciting application of genetic programming.

11 SUMMARY AND CONCLUSIONS

11.1 SUMMARY

This report has described the author's original genetic programming system and a number of experiments that were carried out using it. We have discussed an unsuccessful attempt to optimize GP parameters for the problem of discovering approximations to functions, and have presented several successful or partially successful experiments involving the application of genetic programming to the automated discovery of numerical approximation formulae. In particular, we have presented an experiment involving rediscovery of the first three terms of the asymptotic expansion for H_n. We have also presented positive results in applying genetic programming to the discovery of rational polynomial approximations to common functions, and have shown that evolved approximations can compare favorably with Padé under certain trade-offs between cost and error. We have demonstrated the application of this technique to the discovery of a rational polynomial surface approximation for a function to which the Padé approximation technique cannot be applied. We have also shown that evolved approximations can be refined through genetic programming via approximation of their error functions, and have suggested other ways in which refinement of approximations through genetic programming might be attempted. We have had partial success in an experiment involving attempted discovery or rediscovery of neural network activation functions, and have likewise met with partial success in attempting to evolve piecewise approximations to functions through this technique.

11.2 CONCLUSIONS

This thesis has shown that genetic programming is capable of rediscovering approximation formulae for Harmonic numbers, and of evolving rational polynomial approximations to functions which, under some reasonable utility functions, are superior to Padé approximations. For common mathematical functions of a single variable approximated over a relatively large interval, it has been shown that genetic programming can provide a set of rational polynomial approximations whose Pareto front lies in part to the interior of the Pareto front for Padé approximations to the same function. Moreover, we have shown that the Pareto front for evolved approximations can be improved by evolving approximations to the error function of existing approximations on the Pareto front. Though it has not been demonstrated explicitly in this paper, one would expect that genetic programming would also be able to expand upon the Pareto front for approximations to functions of more than one variable obtained by combining and nesting Padé approximations. Furthermore, for at least one function of more than one variable, genetic programming has been shown to provide a way to evolve rational polynomial approximations where the Padé approximation technique cannot be applied. Based upon these results, the author regards the genetic programming approach described in this thesis as a powerful, flexible, and effective technique for the automated discovery of approximations to functions.

APPENDIX A: EXTENDED RESULTS FOR RATIONAL POLYNOMIAL APPROXIMATION OF FUNCTIONS

This appendix presents tables showing the results of experiments which, for the sake of brevity, were omitted from the sections describing the experiments themselves. Tables A.1 through A.12 present the evolved approximations, the Maple evaluation of these approximations, and the approximations on the Pareto front with respect to the new cost and error values, in that order, for the functions sqrt(x), arcsinh(x), exp(-x), and tanh(x). Each group of three tables (f.e. table A.1 though A.3, tables A.4 through A.6) contains data precisely analogous to that presented for the function ln(x) in tables 5.1 through 5.3 of section 5. Tables A.13 through A.17 present the Padé approximations (on the Pareto front) for each of the five functions ln(x), sqrt(x), arcsinh(x), exp(-x), and tanh(x), respectively. Tables A.18 through A.20 present the evolved approximations, Maple evaluation, and revised Pareto front for approximations to the multi-variable function $f(x,y) = x^y$, expanding on the results presented in section 6. Tables A.21 through A.30 expand upon the results presented in section 7 for refinement of evolved approximations. Table A.21 presents the full set of approximations evolved for the function sin(x). Tables A.22 through A.30 present the evolved approximations, Maple evaluation, and revised Pareto fronts, in that order, for approximations to the error functions of candidate solutions (3), (7), and (8), respectively, from Table A.21.

Table A.1: Evolved Approximations for sqrt(x).

EVOLVED LISP EXPRESSION			
ERROR	**COST**	**RUN**	**GENERATION**
1. `(* (+ (+ (- -4.67254 1.53035) (% X -4.7351)) (% (* (+ (+ (+ (% (% (+ (+ 2.45445-3.86196) (- X (+ X -4.13358))) -4.13358) (- (+ X X) (% (* 1.61611 (* 1.61611 X)) (- (- (+ X X) -2.73919) (* 4.4055 (% X (* 1.61611 X))))))) (+ 2.45445 -3.86196)) (+ (% (- (+ X X) -2.73919) X) (% X -4.7351))) 4.96521) (% X (+ (- -4.67254 1.53035) (% X -4.7351)))) (% (+ X X) (+ (+ (% (- (- X (+ X (+ (% (% X -4.7351) -4.13358) (+ (% X X) -3.86196)))) (* (% X X) (- X -0.917234))) -4.7351) (+ (- -4.67254 -2.73919) -0.399945)) -4.13358)))) (% X (+ (% (* 1.61611 (* 1.61611 X)) (+ -1.98202 0.83636)) -0.399945)))`			
1.54819	29.7	30	83
2. `(* (+ (+ (- -4.67254 1.53035) (% X -4.7351)) (% (* (+ (+ (+ (% (% (% -2.90277 (* 1.61611 X)) -4.13358) (- (+ X X) (% (* 1.61611 (* 1.61611 X)) (% X X)))) (+ 2.45445 -3.86196)) (+ (% (- (+ X X) -2.73919) X) (% X -4.7351))) 4.96521) (% X (+ (- -4.67254 1.53035) (% X -4.7351)))) (% (+ X X) (+ (+ (% (- (- X (+ X (+ (% (% X -4.7351) -4.13358) (+ (% X X) -3.86196)))) (* (% X X) (- X -0.917234))) -4.7351) (+ (- -4.67254 -2.73919) -0.399945)) -4.13358)))) (% X (+ (% (* 1.61611 (* 1.61611 X)) (+ -1.98202 0.83636)) -0.399945)))`			
1.57657	29.1	30	99
3. `(* (+ (+ (- -4.67254 1.53035) (% X -4.7351)) (% (* (+ (+ (+ (% (% (% (- (+ X X) -2.73919) X) -4.13358) (- (+ X X) (% (* 1.61611 (* 1.61611 X)) (- (- -4.67254 1.53035) (+ X X))))))) (+ 2.45445 -3.86196)) (+ (% (- (+ X X) -2.73919) X) (% X -4.7351))) 4.96521) (% X (+ (- -4.67254 1.53035) (% X -4.7351)))) (% (+ X X) (+ (+ (% (- (- X (+ X (+ (% (% X -4.7351) -4.13358) (+ (% X X) -3.86196)))) (* (% X X) (- X -0.917234))) -4.7351) (+ (- -4.67254 -2.73919) -0.399945)) -4.13358)))) (% X (+ (% (* 1.61611 (* 1.61611 X)) (+ -1.98202 0.83636)) -0.399945)))`			
1.60098	27.5	30	96

4. (* (+ (+ (- -4.67254 1.53035) (% X -4.7351)) (% (* (+ (+ (+ (% (+ (% (% X -4.7351) -4.13358) (+ 2.45445 -3.86196)) (- (+ X X) (% X -4.7351))) (+ 2.45445 -3.86196)) (+ (% (- (+ X X) -2.73919) X) (% X -4.7351))) 4.96521) (% X (+ (- -4.67254 1.53035) (% X -4.7351)))) (% (+ X X) (+ (+ (% (- (- X (+ X (+ (% (% X -4.7351) -4.13358) (+ (% X X) -3.86196)))) (* (% X X) (- X -0.917234))) -4.7351) (+ (- -4.67254 -2.73919) -0.399945)) -4.13358)))) (% X (+ (% (* 1.61611 (* 1.61611 X)) (+ -1.98202 0.83636)) -0.399945)))			
1.6085	25.2	30	91
5. (* (+ (+ (- -4.67254 1.53035) (% X -4.7351)) (% (* (+ (+ (+ (% (% (- -4.67254 -2.73919) -4.13358) (- (+ X X) (% (* 1.61611 (* 1.61611 X)) (+ -1.98202 0.83636)))) (+ 2.45445 -3.86196)) (+ (% (- (+ X X) -2.73919) X) (% X -4.7351))) 4.96521) (% X (+ (- -4.67254 1.53035) (% X -4.7351)))) (% (+ X X) (+ (+ (% (- (- X (+ X (+ X (+ (% (% X -4.7351) -4.13358) (+ (% X X) -3.86196)))) (* (% X X) (- X -0.917234))) -4.7351) (+ (- -4.67254 -2.73919) -0.399945)) -4.13358)))) (% X (+ (% (* 1.61611 (* 1.61611 X)) (+ -1.98202 0.83636)) -0.399945)))			
1.76722	25.1	30	65
6. (* (+ (+ (- -4.67254 1.53035) (% X -4.7351)) (% (* (+ (+ (+ (% (% (% (- (+ X X) -2.73919) X) -4.13358) (% (- -4.67254 1.53035) -4.13358)) (+ 2.45445 -3.86196)) (+ (% (- (* 1.61611 (* 1.61611 X)) -2.73919) X) (% X -4.7351))) 4.96521) (% X (+ (- -4.67254 1.53035) (% X -4.7351)))) (% (+ X X) (+ (+ (% (- (- X (+ X (+ (% (% X -4.7351) -4.13358) (+ 2.45445 -3.86196)))) (* (% X X) (- X -0.917234))) -4.7351) (+ (- -4.67254 -2.73919) -0.399945)) -4.13358)))) (% X (+ (% (* 1.61611 (* 1.61611 X)) (+ -1.98202 0.83636)) -0.399945)))			
1.7839	24.9	30	73
7. (* (+ (+ (- -4.67254 1.53035) (% X -4.7351)) (% (* (+ (+ (+ (% (% (* 1.61611 X) -4.13358) (- (+ X X) (- -4.67254 1.53035))) (+ 2.45445 -3.86196)) (+ (% (- (+ X X) -2.73919) X) (% X -4.7351))) 4.96521) (% X (+ (- -4.67254 1.53035) (% X -4.7351)))) (% (+ X X) (+ (+ (% (- (- X (+ X (+ (% (% X -4.7351) -4.13358) (+ (% X X) -3.86196)))) (* (% X X) (- X -0.917234))) -4.7351) (+ (- -4.67254 -2.73919) -0.399945)) -4.13358)))) (% X (+ (% (* 1.61611 (* 1.61611 X)) (+ -1.98202 0.83636)) -0.399945)))			
1.79371	24.1	30	77
8. (* (+ (+ (- -4.67254 1.53035) (% X -4.7351)) (% (* (+ (+ (+ -1.98202 0.83636) (+ (% (- (+ X X) -2.73919) X) (% X -4.7351))) 4.96521) (% X (+ (- -4.7351 1.53035) (% X -4.7351)))) (% (+ X X) (+ (+ (% (- (- X (+ X (+ (% (% X -4.7351) -4.13358) (+ 2.45445 -3.86196)))) (* (% X X) (- X -0.917234))) -4.7351) (+ (- -4.67254 -2.73919) -0.399945)) -4.13358)))) (% X (+ (% (* 1.61611 (* 1.61611 X)) (+ -1.98202 0.83636)) -0.399945)))			
1.79654	19.7	30	87
9. (% X (+ (% X (+ 4.78576 (% X (+ (+ (+ 4.78576 2.9107) (% X (+ (+ (+ 4.78576 2.9107) (+ 4.78576 2.9107)) (% X (+ (+ (+ 4.78576 (+ 4.78576 2.9107)) (+ 4.78576 2.9107)) 4.78576))))) 1.48335)))) 1.48335))			
2.43853	5.5	21	73
10. (- (% (- X (% -0.577258 (+ (- 4.89013 (% -0.577258 4.15036)) 4.15036))) (- (% (- X (+ 4.89013 4.15036)) (+ (+ (% (% X (+ 4.7766 -0.795465)) 4.31394) 4.15036) 4.15036)) -4.32524)) -0.795465)			
3.63081	4.7	10	61
11. (- (% X (- (% (- X (+ 4.89013 4.15036)) (+ (+ (% (% X (+ 4.7766 -0.795465)) 4.31394) 4.15036) 4.15036)) -4.32524)) -0.795465)			
3.84913	4.5	10	63
12. (% X (+ (% X (+ 4.78576 (% X (+ (+ (+ 4.78576 2.9107) (% X (+ (+ (+ 4.78576 2.9107) (+ 4.78576 2.9107)) 2.9107))) 1.48335)))) 1.48335))			

4.10542	4.4	21	87
13. (- (% X (- (- (% X (+ (+ (+ (- (+ (* -1.38783 (- 2.91162 3.84304)) 1.7861) -0.222938) (- 2.91162 -4.27366)) (% X (+ (+ (+ (+ 0.587939 3.55647) (- (+ 3.8345 (+ 0.587939 3.55647)) -1.89444)) (- 2.91162 -4.27366)) (+ 0.587939 3.55647)))) 1.7861)) -1.89444) -4.27366)) -1.89444)			
7.76912	3.5	17	59
14. (% X (+ (% X (+ (+ (+ -0.0300607 (% X (+ (+ (+ 4.64171 (+ (+ 4.78576 2.9107) 1.48335)) (+ -0.0300607 -0.0300607)) 1.48335))) 4.08933) 1.48335)) 1.48335))			
7.82567	3.4	21	16
15. (+ (% X (+ (+ (+ -0.0300607 (% X (+ (+ (+ 4.64171 (+ (+ 4.78576 2.9107) 1.48335)) (+ -0.0300607 -0.0300607)) 1.48335))) 4.08933) 1.48335)) 1.48335)			
9.67331	2.4	21	14
16. (% (+ (+ 4.82238 (* -2.11447 -1.0741)) X) (+ 4.82238 (* (+ (* 0.0779748 (* 0.0779748 (* 3.05109 0.0621052))) X) 0.0621052)))			
11.4792	2.3	45	90
17. (- (% X (+ (- 2.91162 -4.27366) (% X (+ (- 2.91162 -4.27366) (- (+ (- 2.91162 -4.27366) (- 0.587939 -1.89444)) -1.89444))))) -1.89444)			
14.7712	2.2	17	44
18. (% X (+ (% X (+ (+ (+ 3.8345 0.821711) (* -1.38783 -4.61577)) (+ 3.8345 0.821711))) 3.84304))			
26.1004	2.1	17	7
19. (- (% (+ X (+ (* 2.73003 -0.974914) (* 2.05283 (* 2.05283 (+ 3.50856 2.80145)))))) (+ 4.26939 (+ 4.68993 4.81079))) -1.43452)			
34.3854	1.2	31	27
20. (+ (% X (* -4.81658 -2.70287)) (- 1.08234 -1.84835))			
34.9276	1.1	25	9
21. (% X (- (+ 3.8345 (+ 0.587939 3.55647)) -0.395367))			
124.992	1	17	27
22. (+ (- (- 4.89013 -0.467086) -4.32524) (- 4.89013 (- 3.23542 -4.32524)))			
197.533	0	10	12

Table A.2: Maple Evaluation of Approximations for sqrt(x).

SIMPLIFIED MAPLE EXPRESSION	COST	ERROR
1. (-6.20289-.2111887817*x+1/2*(-.6594937076/(2*x-2.611811532*x/(2*x+.013199814))+3.5577+(2*x+2.73919)/x-.2111887817*x)*(.2219786301*x-6.877579315)/(-6.20289-.2111887817*x))*x/(-2.279744018*x-.399945)	17	INF
2. (-6.20289-.2111887817*x+1/2*(-.7102278571/(x^2)+3.5577+(2*x+2.73919)/x-.2111887817*x)*(.2219786301*x-6.877579315)/(-6.20289-.2111887817*x))*x/(-2.279744018*x-.399945)	13	INF
3. (-6.20289-.2111887817*x+1/2*(-.2419210466*(2*x+2.73919)/(x*(2*x-2.611811532*x/(-6.20289-2*x)))+3.5577+(2*x+2.73919)/x-.2111887817*x)*(.2219786301*x-6.877579315)/(-6.20289-.2111887817*x))*x/(-2.279744018*x-.399945)	20	INF
4. (-6.20289-.2111887817*x+1/2*(.4522454203*(.05109101111*x-1.40751)/x+3.55770+(2*x+2.73919)/x-.2111887817*x)*(.2219786301*x-6.877579315)/(-6.20289-.2111887817*x))*x/(-2.279744018*x-.399945)	15	INF

5. (-6.20289-.2111887817*x+1/2*(.1092864558/x+3.55770+(2*x+2.73919)/x-.2111887817*x)*(.2219786301*x-6.877579315)/(-6.20289-.2111887817*x))*x/(-2.279744018*x-.399945)	13	INF
6. (-6.20289-.2111887817*x+1/2*(-.1612151755*(2*x+2.73919)/x+3.55770+(2.611811532*x+2.73919)/x-.2111887817*x)*(.2219786301*x-6.570415791)/(-6.20289-.2111887817*x))*x/(-2.279744018*x-.399945)	15	INF
7. (-6.20289-.2111887817*x+1/2*(-.3909710227*x/(2*x+6.20289)+3.55770+(2*x+2.73919)/x-.2111887817*x)*(.2219786301*x-6.877579315)/(-6.20289-.2111887817*x))*x/(-2.279744018*x-.399945)	15	INF
8. (-6.20289-.2111887817*x+1/2*(3.81955+(2*x+2.73919)/x-.2111887817*x)*(.2219786301*x-6.570415791)/(-6.26545-.2111887817*x))*x/(-2.279744018*x-.399945)	12	INF
9. x/(x/(4.78576+x/(9.17981+x/(15.39292+.04005697704*x)))+1.48335)	5	2.591348148
10. (x+.06288503787)/((x-9.04049)/(.05822627334*x+8.30072)+4.32524)+.795465	3	3.123452980
11. x/((x-9.04049)/(.05822627334*x+8.30072)+4.32524)+.795465	3	3.343211605
12. x/(x/(4.78576+x/(9.17981+.05463400136*x))+1.48335)	4	4.291955444
13. x/(x/(12.27307062+.03945170494*x)+6.16810)+1.89444	3	6.729391394
14. x/(x/(5.5426193+.06559635887*x)+1.48335)	3	8.090631361
15. x/(5.5426193+.06559635887*x)+1.48335	2	8.935605674
16. (7.093532227+x)/(4.822451552+.0621052*x)	2	10.76103179
17. x/(7.18528+.05334078966*x)+1.89444	2	13.78128841
18. x/(.06362000603*x+3.84304)	2	26.36889260
19. .07262106112*x+3.172308452	1	32.95322345
20. .07681323648*x+2.93069	1	33.68753043
21. .1194133081*x	1	126.0548802
22. 7.011926	0	195.5193204

Table A.3: Final Evolved Approximations for sqrt(x).

EVOLVED APPROXIMATION	COST	ERROR
(9) x/(x/(4.78576+x/(9.17981+x/(15.39292+.04005697704*x)))+1.48335)	5	2.591348148
(10) (x+.06288503787)/((x-9.04049)/(.05822627334*x+8.30072)+4.32524)+.795465	3	3.123452980
(15) x/(5.5426193+.06559635887*x)+1.48335	2	8.935605674
(19) .07262106112*x+3.172308452	1	32.95322345
(22) 7.011926	0	195.5193204

Table A.4: Evolved Approximations for arcsinh(x).

EVOLVED LISP EXPRESSION			
ERROR	**COST**	**RUN**	**GENERATION**
1. (* (% (+ (* (% X 4.67803) (% (% (% X (+ (% (+ 4.45067 -1.31733) X) X)) (+ (% (% (% X (+ 0.0840785 X)) X) (% (% (+ (+ (- (% X X) 3.61873) (+ 4.82299 -1.77664)) (+ -4.75433 -2.74957)) (% (- -2.90216 X) 2.58477)) (% (+ 0.0840785 X) (* (+ -3.84457 4.86602) (+ X X))))) (% (% (% X (+ 0.0840785 X)) X) (% (% X (% (- -2.90216 X) 2.58477)) (% (* (+ (* (+ -3.84457 4.86602) (+ 0.711234 4.15006)) 4.86602) (+ 1.61763 3.61873)) (* (+ -3.84457 4.86602) (+ 4.82299 -1.77664))))))) (% (% (% X (+ 0.0840785 X)) X) (% (% (+ (+ (- (% X X) 3.61873) (% (+ 0.0840785 X) (+ 0.0840785 X))) (+ -4.75433 -2.74957)) (% (- -2.90216 X) 2.58477)) (% (* (+ X 4.86602) (+ X -0.269326)) (* (+ -3.84457 4.86602) (+ 4.82299 -1.77664))))))) (+ (% (% (% X (+ 0.0840785 X)) X) (% (% (+ (+ (- (% X X) 3.61873) (% (+ 0.0840785 X) (+ 0.0840785 X))) (+ -4.75433 -2.74957)) (% (- -2.90216 X) 2.58477)) (% (* (+ X 4.86602) (+ X -0.269326)) (* (+ -3.84457 (+ 0.711234 4.15006)) (+ 4.82299 -1.77664))))) (* 4.83551 X))) (+ (- (+ 0.711234 4.15006) (* -2.08151 X)) 4.82299)) 1.86636)			
3.12475	48.3	40	52
2. (* (% (+ (* (% X 4.67803) (% (% (% X (+ (% (+ 4.45067 -1.31733) (* (% X 4.67803) (+ (% (* X -2.85394) (* (+ -3.84457 4.86602) (+ 4.82299 -1.77664))) (- (- -0.678274 -0.843684) (+ (- 1.86636 2.55089) 1.33137))))) X)) (+ -4.75433 -2.74957)) (% (% (+ (% (+ 4.45067 -1.31733) (- -4.88586 X)) (- (% (- (+ 1.61763 3.61873) X) -3.5345) (+ X X))) (% (+ 0.711234 4.15006) 2.58477)) (% (* (- X 0.271157) (% (+ (+ (- (% X X) 3.61873) (% (% X (+ 0.0840785 X)) (+ 0.0840785 X))) (+ -4.75433 -2.74957)) (% (- -2.90216 X) 2.58477))) (- (- -0.678274 -0.843684) (+ (- 1.86636 2.55089) 1.33137)))))) (+ (% (% (% X (+ 0.0840785 X)) X) (% (% (+ (+ (- (% X X) 3.61873) (% (+ 0.0840785 X) (+ 0.0840785 X))) (+ -4.75433 -2.74957)) (% (- -2.90216 X) 2.58477)) (% (* (+ X 4.86602) (+ X -0.269326)) (* (+ -3.84457 4.86602) (+ 4.82299 -1.77664))))) (* 4.83551 X))) (+ (- (+ 0.711234 4.15006) (* -2.08151 X)) 4.82299)) 1.86636)			
3.15516	37.7	40	41
3. (* (% (+ (* (% X 4.67803) (% (+ -3.84457 4.86602) (% (% (+ (% (% (- -2.90216 X) 2.58477) (- -4.88586 X)) (+ -3.84457 4.86602)) (% (+ 0.711234 4.15006) 2.58477)) (% (* (+ 4.82299 -1.77664) (+ -3.84457 4.86602)) (% (- -2.90216 X) 2.58477))))) (+ (% (% (% X (+ 0.0840785 X)) X) (% (% (+ (+ (- (% X X) 3.61873) (% (+ 0.0840785 X) (+ 0.0840785 X))) (+ -4.75433 -2.74957)) (% (- -2.90216 X) 2.58477)) (% (* (+ X 4.86602) (+ X -0.269326)) (* (+ -3.84457 4.86602) (+ 4.82299 -1.77664))))) (* 4.83551 X))) (+ (- (+ 0.711234 4.15006) (* -2.08151 X)) 4.82299)) 1.86636)			
3.37734	24.7	40	76
4. (* (% (+ (* (% X 4.67803) (% (% (% X (+ (+ X 4.86602) X)) (+ (* (+ -3.84457 4.86602) (+ 4.82299 -1.77664)) (* 4.83551 X))) (+ 4.82299 -1.77664))) (+ (% (% (% X(+ 0.0840785 X)) X) (% (% (+ (+ (- (% X X) 3.61873) (% (+ 0.0840785 X) (+ 0.0840785 X))) (+ -4.75433 -2.74957)) (% (- (+ (- 1.86636 2.55089) 1.33137) X) 2.58477)) (% (* (+ X 4.86602) (+ X -0.269326)) (* (+ -3.84457 (+ 0.711234 4.15006)) (+ 4.82299 -1.77664))))) (* 4.83551 X))) (+ (- (+ 0.711234 4.15006) (* -2.08151 X)) 4.82299)) 1.86636)			
3.51899	21.6	40	84

5. (* (% (+ (* (% X 4.67803) (% (% (+ (% (- -2.90216 X) 2.58477) (+ -3.84457 4.86602)) (% (+ 0.711234 4.15006) 2.58477)) X)) (+ (% (% (% X X) X) (% (% (+ (+ (- (% X X) 3.61873) (% (+ 0.0840785 X) (+ 0.0840785 X))) (+ -4.75433 -2.74957)) (% (- -2.90216 X) 2.58477)) (% (* (+ X 4.86602) (+ X -0.269326)) (* (+ -3.84457 4.86602) (+ 4.82299 -1.77664))))) (* 4.83551 X))) (+ (- (+ 0.711234 4.15006) (* -2.08151 X)) 4.82299)) 1.86636)			
3.64576	20.4	40	78
6. (* (% (+ (% X (- -2.90216 X)) (+ (% (% (% X X) X) (% (% (+ (+ (- (% X X) 3.61873) (% (+ 0.0840785 X) (+ 0.0840785 X))) (+ -4.75433 -2.74957)) (% (- -2.90216 X) 2.58477)) (% (* (+ X 4.86602) (+ X -0.269326)) (* (+ -3.84457 4.86602) (+ 4.82299 -1.77664))))) (* 4.83551 X))) (+ (- (+ 0.711234 4.15006) (* -2.08151 X)) 4.82299)) 1.86636)			
3.68765	16.3	40	66
7. (* (% (+ 0.0840785 (+ (% (% (% X (+ 0.0840785 X)) X) (% (% (+ (+ (- (% X X) 3.61873) (% X X)) (+ -4.75433 -2.74957)) (% (- -2.90216 X) 2.58477)) (% (* (+ 0.0840785 X) (+ X -0.269326)) (* (+ -3.84457 4.86602) (+ 4.82299 -1.77664))))) (* 4.83551 X))) (+ (- (+ 0.711234 4.15006) (* -2.08151 X)) 4.82299)) 1.86636)			
3.8012	15.1	40	88
8. (% 4.97223 (% (- (% (- -3.74569 X) (- (% (% (* -0.818354 -4.4705) (* X (% (+ -0.52385 0.528428) (- (% (* X (+ (- -3.74569 X) (+ 4.90081 1.16779))) (- (- (% X (- (- 0.528428 -0.231788) 3.59493)) (- (+ (- -3.74569 X) X) X)) (* -0.818354 -4.4705))) 3.59493)))) (- (- -3.74569 X) (+ (* -0.818354 -4.4705) (+ 4.90081 1.16779)))) 3.59493)) (- (+ (- (- (+ -0.52385 0.528428) (+ (- X -4.40397) (% (- (+ (- (+ -0.52385 0.528428) 2.97479) X) X) (* X (+ -4.4705 (+ (- (- -3.74569 X) 2.97479) 1.16779)))))) 2.97479) X) X)) X))			
4.55309	14.5	25	99
9. (% 4.97223 (% (- (% (- -3.74569 X) (- (% (% (* -0.818354 -4.4705) (* X (% (+ -0.52385 0.528428) (- (% (* X (+ (- -3.74569 X) -2.99371)) (- (- (% X (- 4.90081 3.59493)) (- (- -3.74569 X) 2.97479)) (+ X (+ 4.90081 1.16779)))) 3.59493)))) (- (* 1.16779 -1.42933) (+ (- X -4.40397) -2.99371))) 3.59493)) (- (+ (- (- (- -3.74569 X) 2.97479) X) X)) X))			
4.67254	11.8	25	60
10. (% 2.46147 (- (% -1.91549 -4.5822) (% 4.28068 (- (% -1.0744 (% (* (+ -4.38017 (* X -1.84927)) (* (- (% 4.28068 X) (% 4.28068 (- (+ (- X X) (- (% -1.91549 (% -1.0744 -2.10685)) (% 4.28068 (% -1.91549 -4.5822)))) X))) 0.243995)) (+ -4.38017 (- -1.0744 (- (- (- (% -1.91549 -4.5822) (% -1.0744 -2.10685)) -4.74944) X))))) X))))			
4.69637	10	24	73
11. (% 4.97223 (% (- (% (- -3.74569 X) (- (% (% (* -0.818354 -4.4705) (* X (- 0.528428 -0.231788))) (+ -0.52385 0.528428)) 3.59493)) (- (+ (- (- -3.74569 X) 2.97479) X) X)) X))			
6.187	6.7	25	80
12. (% 2.46147 (- (% -1.91549 -4.5822) (% 4.28068 (- (% -1.0744 (% (% -4.21445 (- (% 4.28068 (% -1.0744 -2.10685)) X)) -1.0744)) X))))			
7.0933	5.3	24	20
13. (% 2.46147 (- (% -1.91549 -4.5822) (% 4.28068 (- (% -1.0744 (% -4.12046 (- -4.12046 (- (% 2.46147 (% -1.0744 -2.10685)) X)))) X))))			
7.19888	4.4	24	48
14. (% 2.46147 (- (% -1.91549 -4.5822) (% 4.28068 (- (% -1.0744 (% 4.28068 (- (% 4.28068 (% -1.0744 -2.10685)) X))) X))))			

7.44273	4.3	24	30
15. (- (% (+ 4.78576 (% -1.32282 4.13846)) (% (+ 4.00327 (+ (+ 4.78576 (+ 4.78576 -0.498825)) (+ 4.64171 (+ (+ X (% (- 4.16868 -1.845) -4.68993)) (% -1.32282 (% -4.17173 4.98383)))))) X)) -1.32282)			
8.15916	2.6	21	21
16. (- (+ -3.45485 (% 3.07154 (- (% (+ (+ (+ 4.75799 (+ (+ 4.75799 (- -3.64345 -4.6292)) 0.849788)) 0.849788) 0.849788) X) -0.718253))) -4.95209)			
9.28266	2.3	8	79
17. (% X (- (% (- X (* -0.0599689 (- (- 4.13846 0.320597) -1.79861))) (+ 1.83035 (- 4.13846 0.320597))) -1.79861))			
11.0621	2.2	43	67
18. (% X (- (% X (- (- 4.13846 0.320597) -1.79861)) -1.79861))			
11.2001	2.1	43	56
19. (- 4.16318 (% (- (- (+ (+ (- 4.65697 1.66738) (- 4.16318 (% -0.418256 (* -1.19282 -3.41975)))) (+ (+ (+ (- (+ 3.76675 (- 3.76675 1.66738)) (- 0.937071 0.376751)) (* -1.19282 -3.41975)) (- 4.16318 (% -0.418256 (- (- 0.937071 0.376751) (- 4.65697 1.66738))))) (- 4.65697 1.66738))) (+ -3.13746 -1.29597)) X) (+ (+ (* -1.19282 -3.41975) (- 4.65697 1.66738)) X)))			
18.4133	1.3	42	92
20. (+ 4.98077 (% (+ (% (% 4.98077 3.68709) 3.68709) (% (% (- (+ 4.98077 3.68709) 1.18305) (% 0.31663 3.68709)) 3.68709)) (- (- -2.22068 (% (- (+ 4.98077 3.68709) 2.26096) 3.68709)) X)))			
26.3929	1.2	22	90
21. (+ 3.30409 (% X (% (+ -0.0279244 (* 2.25852 -0.509507)) -0.0279244)))			
27.2391	1.1	14	70
22. (- (- 3.03095 (+ (+ (- -2.40593 X) X) 1.64968)) (% 3.51802 (- (+ -3.52412 3.55678) (+ (- 1.64968 -1.94357) (% -2.91498 -3.78842)))))			
70.3609	0.5	9	11
23. (* -3.88546 (* -0.38316 3.09046))			
70.3609	0	5	78

Table A.5: Maple Evaluation of Approximations for arcsinh(x).

SIMPLIFIED MAPLE EXPRESSION	COST	ERROR
1. 1.86636*(-6.068117333*x^2*(.0840785+x)/((3.13334/x+x)*(-.06917479528*(-1.122792357-.3868816181*x)/x+16.54460068*(-1.122792357-.3868816181*x)/(x*(.0840785+x)))*(-1.122792357-.3868816181*x)*(x+4.86602)*(x-.269326))-.0353913448*(-1.122792357-.3868816181*x)*(x+4.86602)*(x-.269326)/(.0840785+x)+ 4.83551*x)/(9.684284+2.08151*x)	26	INF
2. 1.86636*(.1112876041*x^2*(x-.271157)*(-10.12263+x/((.0840785+x)^2))/((14.65785852/(x*(-.9171659582*x-.481430))+x)*(3.13334/(-4.88586-x)-1.481499505-1.717074551*x)*(-1.122792357-.3868816181*x))-.03522759767*(-1.122792357-.3868816181*x)*(x+4.86602)*(x-.269326)/(.0840785+x)+4.83551*x)/(9.684284+2.08151*x)	24	INF
3. 1.86636*(1.277853316*x/((.3868816181*(-2.90216-x)/(-4.88586-x)+1.02145)*(-1.122792357-.3868816181*x))-.03522759767*(-1.122792357-.3868816181*x)*(x+4.86602)*(x-.269326)/(.0840785+x)+4.83551*x)/(9.684284+2.08151*x)	17	3.361399200
4. 1.86636*(.07017092454*x^2/((2*x+4.86602)*(3.111694208+4.83551*x))-.03539134480*(-.2502505059-.3868816181*x)*(x+4.86602)*(x-.269326)/(.0840785+x)+4.83551*x)/	15	3.533969225

(.2502505059-.3868816181*x)*(x+4.86602)*(x-.269326)/(.0840785+x)+4.83551*x)/(9.684284 +2.08151*x)		
5. 1.86636*(-.0115185554+4.7915371*x-.03522759767*(-1.122792357-.3868816181*x)*(x+4.86602)*(x-.269326)/x)/(9.684284+2.08151*x)	9	INF
6. 1.86636*(x/(-2.90216-x)-.03522759767*(-1.122792357-.3868816181*x)*(x+4.86602)*(x-.269326)/x+4.83551*x)/(9.684284+2.08151*x)	11	INF
7. 1.86636*(.0840785-.03522759767*(-1.122792357-.3868816181*x)*(x-.269326)+4.83551*x)/(9.684284+2.08151*x)	7	3.804858563
8. 4.97223*x/((-3.74569-x)/(799.1375179*(x*(2.32291-x)/(.6472307259*x+.087238443)-3.59493)/(x*(-13.47274156-x))-3.59493)+7.374182+x-2.970212/(x*(-10.02319-x)))	15	INF
9. 4.97223*x/((-3.74569-x)/(799.1375179*(x*(-6.73940-x)/(.7657671455*x+.65188)-3.59493)/(x*(-3.079417281-x))-3.59493)+6.72048+x)	12	INF
10. 2.46147/(.4180284579-4.28068/(-1.0744*(-10.11208284+x)/((-4.38017-1.84927*x)*(1.044464517/x-1.044464517/(-x-13.99635368)))-x))	10	INF
11. 4.97223*x/((-3.74569-x)/(1051.197972/x-3.59493)+6.72048+x)	5	INF
12. 2.46147/(.4180284579-4.28068/(-2.299172064-.7261005920*x))	3	6.596080331
13. 2.46147/(.4180284579-4.28068/(-2.332984737-.7392524136*x))	3	6.705450066
14. 2.46147/(.4180284579-4.28068/(-2.106850000-.7490118392*x))	3	6.963261097
15. 4.466119361*x/(18.01575130+x)+1.32282	2	7.581253733
16. 1.49724+3.07154/(13.051094/x+.718253)	2	INF
17. x/(.1770471475*x+1.858241906)	2	11.18858617
18. x/(.1780476822*x+1.79861)	2	11.32660567
19. 4.16318-(28.05427784-x)/(7.068736195+x)	3	18.48691726
20. 4.98077+24.00535693/(-3.958337611-x)	2	26.11701008
21. 3.30409+.02369172723*x	1	25.83927515
22. 4.599669224	0	68.51967458
23. 4.600931145	0	68.51916981

Table A.6: Final Evolved Approximations for arcsinh(x).

EVOLVED APPROXIMATION	COST	ERROR
(3) 1.86636*(1.277853316*x/((.3868816181*(-2.90216-x)/(-4.88586-x)+1.02145)*(-1.122792357-.3868816181*x))-.03522759767*(-1.122792357-.3868816181*x)*(x+4.86602)*(x-.269326)/(.0840785+x)+4.83551*x)/(9.684284+2.08151*x)	17	3.361399200
(4) 1.86636*(.07017092454*x^2/((2*x+4.86602)*(3.111694208+4.83551*x))-.03539134480*(.2502505059-.3868816181*x)*(x+4.86602)*(x-.269326)/(.0840785+x)+4.83551*x)/(9.684284+2.08151*x)	15	3.533969225
(7) 1.86636*(.0840785-.03522759767*(-1.122792357-.3868816181*x)*(x-.269326)+4.83551*x)/(9.684284+2.08151*x)	7	3.804858563
(12) 2.46147/(.4180284579-4.28068*1/(-2.299172064-.7261005920*x))	3	6.596080331
(15) 4.466119361*x/(18.01575130+x)+1.32282	2	7.581253733
(21) 3.30409+.02369172723*x	1	25.83927515
(23) 4.600931145	0	68.51916981

Table A.7: Evolved Approximations for exp(-x).

EVOLVED LISP EXPRESSION			
ERROR	**COST**	**RUN**	**GENERATION**
1. (% (% 2.40715 X) (- (* (- (% (* (* (% (% 2.40715 X) (+ (% (- (+ (- X (% 2.40715 X)) -4.14823) X) (* (* -3.43654 (- X 4.21812)) (- X X))) (+ X X))) (+ -4.8178 2.60949)) (- (* (- X (% 2.40715 (% 2.40715 X))) (- (% (+ -3.64345 (- X 1.16077)) (% 2.73614 1.98782)) (% 3.77285 X))) X)) (- X (% (* -3.1402 (+ (% 2.40715 X) (% 4.82208 0.155187))) (- X (% 2.40715 X))))) (* X X)) (- -0.558031 (% X 2.39586))) (+ (- X (% 2.40715 X)) -4.14823)))			
0.0151717	26	9	92
2. (% (% 2.40715 X) (- (* (- (% (* (* (% (% 2.40715 X) (+ (% (- (% 2.40715 X) X) (* (* -3.43654 (- X 4.21812)) (- X X))) (+ X X))) (+ -4.8178 2.60949)) (- (* (- X (% 2.40715 (% 2.40715 X))) (- (% (+ -3.64345 X) (% 2.73614 1.98782)) (% 3.77285 X))) X)) (- X (% (* -3.1402 (+ (% 2.40715 X) (% 4.82208 0.155187))) (- X (% 2.40715 X))))) (* X X)) (- -0.558031 (% X 2.39586))) (+ (- X (% 2.40715 X)) -4.14823)))			
0.0151717	25.7	9	94
3. (% (% 2.40715 X) (- (* (- (% (* (* (% (% 2.40715 X) (+ (% (- (+ X -4.14823) X) (* (* -3.43654 (- X 4.21812)) (- X X))) (+ X X))) (+ -4.8178 2.60949)) (- (- 0.558031 (% X 2.39586))) (- X (% (* -3.1402 (+ (% 2.40715 X) (% 4.82208 0.155187))) (- X (% 2.40715 X))))) (* X X)) (- -0.558031 (% X 2.39586))) (+ (- X (% 2.40715 X)) -4.14823)))			
0.0151729	20.5	9	95
4. (% (% 2.40715 X) (- (* (- (% (- (- X -2.36961) (% X (% 2.73614 1.98782))) (- X (- (- X (- -2.1569 (* X X))) (* X X)))) (* X X)) (- (% (- (- X X) X) (* 3.02149 (+ 1.48305 4.21812))) (% X 2.39586))) (+ (- X (% 2.40715 X)) -4.14823)))			
0.0208874	12.3	9	87
5. (% (% 2.40715 X) (- (* (- (- (- X X) X) (* X X)) (- (% (- X X) (+ (- X -0.23484) (% (- X (- X (% (% -1.36494 (+ X X)) X))) (+ -3.64345 X)))) (% X 2.39586))) (+ (- X (% 2.40715 X)) -4.14823)))			
0.0278936	11.4	9	34
6. (% (% (* 3.49696 (+ (+ 2.89727 4.49675) -1.63442)) (* X 3.86624)) (+ (+ (+ 2.89727 4.49675) -1.63442) (% (* X (+ (+ (+ (- X -4.4058) -1.63442) (* X (* X X))) (+ (- (% (+ (+ 2.89727 4.49675) -1.63442) 4.49675) X) (% X (% X 3.49696))))) X)))			
0.0318482	9.7	33	77
7. (% (% 1.53188 X) (+ (% (- (% 2.56737 (- (+ X -3.27265) X)) (% (- X 0.848262) (+ (- X X) (% -0.619373 -2.65526)))) X) (+ (* X X) (- 3.45882 -1.10736))))			
0.0343202	6.8	37	61
8. (% (% 1.53188 X) (+ (% (+ 4.96979 (- (% (- X X) 2.21335) X)) X) (+ (* X X) (- X 1.73177))))			
0.0372532	5.6	37	81
9. (% (% 4.61425 (+ X X)) (- (+ (% (+ X X) (* X X)) (+ X (* X X))) -2.37358))			
0.0623418	5.5	3	5
10. (% (% 1.53188 X) (+ (% (+ 4.96979 -2.65526) X) (+ (* X X) (% X 1.39454))))			
0.0623977	5.2	37	84
11. (% (% 1.53188 X) (+ (% 2.21335 X) (+ (* X X) X)))			

0.0635599	4.2	37	86
12. (% (% 1.53188 X) (+ (% (+ 4.96979 (% -4.0286 2.93756)) X) (* X X)))			
0.0687091	4.1	37	74
13. (% -3.00226 (- -3.00226 (* (* X (+ X 3.70418)) X)))			
0.0857489	3.2	30	53
14. (% -3.00226 (- -3.00226 (* (* X 4.66735) X)))			
0.187809	3.1	30	7
15. (% (+ (- (- X -0.950804) X) (- X X)) (+ (- X (- (% -3.54274 3.58638) (* X X))) (- X X)))			
0.30584	2.8	43	99
16. (% (- (- X -0.950804) X) (+ (- X (- (- X X) (* X X))) (- X -0.950804)))			
0.338316	2.7	43	77
17. (% 2.70226 (+ (+ (- -1.37379 -3.94192) (* X (+ (+ X X) X))) X))			
0.358178	2.4	34	66
18. (% -3.00226 (- -2.90277 (* (+ X (+ X 3.70418)) X)))			
0.41648	2.3	30	70
19. (% 2.39189 (+ (% X (% (- -3.16797 -3.21253) 3.1933)) (- 2.34153 X)))			
0.617717	2.2	28	14
20. (% -4.35698 (+ -3.71548 (* (% (- 3.97794 -4.26359) (* -0.105136 0.0492874)) X)))			
0.753557	2.1	4	2
21. (% -0.275124 (- (- -0.16831 (+ (* -0.0126652 -4.77813) (- X X))) (+ (+ (+ X X) (- X (- X X))) X)))			
0.837032	1.9	42	6
22. (% (+ -0.442976 0.119175) (+ (+ -0.442976 0.119175) (- (- -4.83276 X) (+ (* 1.17573 -4.04508) X))))			
0.991187	1.4	11	12
23. (% -0.0892666 (+ -0.0889615 X))			
1.06148	1.1	32	45
24. (- X X)			
1.58198	0.1	0	0
25. (% (* (% (% -1.61824 -1.65487) -4.85687) (% 0.00167852 -2.90094)) (* 4.99176 -4.14487))			
1.58254	0	13	9

Table A.8: Maple Evaluation of Approximations for exp(-x).

SIMPLIFIED MAPLE EXPRESSION	COST	ERROR
1. [division by zero]	NA	NA
2. [division by zero]	NA	NA
3. [division by zero]	NA	NA
4. 2.40715/(x*(-.4754383461*(-.1267999337*x-1.098618387-x^2)*x-x+2.40715/x+4.14823))	9	INF

5. 2.40715/(x*(-.4173866587*(-x-x^2)*x-x+2.40715/x+4.14823))	9	INF
6. 5.209477637/(x*(13.30877616+x^3))	4	INF
7. 1.53188/(x*((2.852017392-4.287012834*x)/x+x^2+4.56618))	5	INF
8. 1.53188/(x*((4.96979-x)/x+x^2+x-1.73177))	5	INF
9. 2.307125/(x*(2/x+x+x^2+2.37358))	4	INF
10. 1.53188/(x*(2.31453/x+x^2+.7170823354*x))	5	INF
11. 1.53188/(x*(2.21335/x+x+x^2))	4	INF
12. 1.53188/(x*(3.598379714/x+x^2))	4	INF
13. -3.00226/(-3.00226-x^2*(x+3.70418))	4	.1651951367
14. -3.00226/(-3.00226-4.66735*x^2)	3	.2530555679
15. .950804/(x+.9878317412+x^2)	2	.2992106079
16. .950804/(2*x+x^2+.950804)	3	.4031475900
17. 2.70226/(2.56813+3*x^2+x)	3	.3685369319
18. -3.00226/(-2.90277-(2*x+3.70418)*x)	4	.3928167835
19. 2.39189/(70.66292640*x+2.34153)	2	.9034893384
20. -4.35698/(-3.71548-1590.451702*x)	2	.9608621607
21. -.275124/(-.2288259721-4*x)	2	.9072783614
22. -.323801/(-.400639092-2*x)	2	1.016436073
23. -.0892666/(-.0889615+x)	1	2.412875333
24. 0	0	1.050833194
25. -.5630484060e-5	0	1.051396243

Table A.9: Final Evolved Approximations for exp(-x).

EVOLVED APPROXIMATION	COST	ERROR
(13) -3.00226/(-3.00226-x^2*(x+3.70418))	4	.1651951367
(14) -3.00226/(-3.00226-4.66735*x^2)	3	.2530555679
(15) .950804/(x+.9878317412+x^2)	2	.2992106079
(24) 0	0	1.050833194

Table A.10: Evolved Approximations for tanh(x).

EVOLVED LISP EXPRESSION			
ERROR	**COST**	**RUN**	**GENERATION**
1. `(% X (- X (% -1.12415 (* (- X (% -1.12415 (+ 2.30583 (% -1.12415 (* (% (* -1.33717 4.95727) (+ (+ (% X X) (* X X)) (+ (+ X 4.43846) X))) (* (- X 2.36839) (% X (- X (% -1.12415 (* (+ -0.588549 3.49513) -1.12415)))))))))) (% X (% -1.12415 (- (* (% -1.12415 (- X 4.43846)) (% -1.12415 X)) (* (+ (* X X) X) (% X X)))))))))`			
0.00867654	20.2	7	40
2. `(% X (- X (% -1.12415 (* (* X (+ X (% X (- X (% -1.12415 (% X (* (% X (% X X)) (% X X)))))))) (- (- X -1.12415) (% X (* X -3.47255)))))))`			

0.0153367	13.5	7	36
3. (% X (- X (% -1.12415 (* (% (* X -3.47255) (% X (- X (% (- (+ -0.588549 3.49513) 0.0422681) (- X (- (* -1.12415 (* (+ (% -1.12415 1.02802) 4.43846) (+ X 4.43846))) 1.02802)))))) (% (* X -3.47255) (+ -0.588549 3.49513))))))			
0.0183741	11.5	7	48
4. (% X (- X (% -1.12415 (* (* X X) (+ (% -1.12415 (- X (* -0.410321 X))) 4.43846)))))			
0.0194832	6.3	7	26
5. (% X (- X (% -1.12415 (% (% X (% -1.12415 X)) -0.247353))))			
0.0200782	5.1	7	14
6. (% X (- X (% -1.12415 (* (* X X) (+ (% -1.12415 (* -1.12415 -1.12415)) 4.43846)))))			
0.0227701	4.1	7	25
7. (% X (- X (% -1.12415 (* (+ X 3.02667) X))))			
0.0807519	3.2	7	1
8. (% X (- X (% -1.12415 (* X (+ (% -1.12415 (+ -0.588549 3.49513)) 4.43846)))))			
0.149682	3.1	7	41
9. (% X (- X (% (% -1.12415 (+ -0.588549 3.49513)) (- X -0.247353))))			
0.178682	2.2	7	94
10. (% X (- X (% (% -1.12415 (+ -0.588549 3.49513)) X)))			
0.232949	2.1	7	98
11. (% (% -2.81732 -2.81732) (% X X))			
0.2801	2	42	20
12. (% X (+ 0.00961333 X))			
0.293169	1.1	14	0
13. (% -1.12415 -1.12415)			
1.2801	0	7	8

Table A.11: Maple Evaluation of Approximations for tanh(x).

SIMPLIFIED MAPLE EXPRESSION	COST	ERROR
1. x/(x-1.263713223/((x+1.12415/(2.30583+.1695879798*(5.43846+x^2+2*x)*(x-.3440468372)/((x-2.36839)*x)))*x*(1.263713223/((x-4.43846)*x)-x^2-x)))	15	INF
2. x/(x+1.12415/(x*(x+x/(x+1.12415))*(x+1.412122815)))	5	INF
3. x/(x+.2709628052/(x*(x-2.8643129/(4.760225707*x+17.71763140))))	5	INF
4. x/(x+1.12415/(x^2*(-.7970880388/x+4.43846)))	4	INF
5. x/(x+.3125832568/(x^2))	2	INF
6. x/(x+.3167602122/(x^2))	2	INF
7. x/(x+1.12415/((x+3.02667)*x))	3	INF
8. x/(x+.2774514574/x)	2	INF
9. x/(x+.3867602520/(x+.247353))	2	.2020989978
10. x/(x+.3867602520/x)	2	INF

11. 1.000000000	0	.7439807922
12. x/(.00961333+x)	1	.6362665411
13. 1.000000000	0	.7439807922

Table A.12: Final Evolved Approximations for tanh(x).

EVOLVED APPROXIMATION	COST	ERROR
(9) x/(x+.3867602520/(x+.247353))	2	.2020989978
(12) x/(.00961333+x)	1	.6362665411
(11) 1.000000000	0	.7439807922

Table A.13: Padé Approximations for ln(x).

PADÉ APPROXIMATION			
NUM. DEGREE	**DENOM. DEGREE**	**COST**	**ERROR**
1. 7.195479314-840/(x+262.25-6187.270833/(x+65.82178044-464.6336465/(x+26.86478638-98.63535114/(x+14.31857879-31.86167580/(x+8.765143228-12.91833467/(x+5.833382508-6.027585821/(x+4.100445023-3.086957251/(x+2.992082715-1.685328467/(x+2.240710135-.9615615789/(x+1.708037584-.5649290196/(x+1.316769388-.3376800239/(x+1.020961319-.2031352219/(x+.7919155112-.1216292021/(x+.6109585685-.07157321772/(x+.4655160166-.04071151301/(x+.3468695802-.02183686918/(x+.2488207316-.01058314632/(x+.1668650236-.004238105654/(x+.09766592337-.001080858601/(x+.03871113631))))))))))))))))))))))			
20	20	20	.02853873033
2. 7.095479314-760/(x+237.25-5061.4375/(x+59.51030772-379.412099/(x+24.26070364-80.31337637/(x+12.90842827-25.83853792/(x+7.883468062-10.42028827/(x+5.230698048-4.828767506/(x+3.662669133-2.451726674/(x+2.659779930-1.324160627/(x+1.979908898-.7453928174/(x+1.497926205-.4305892287/(x+1.143891751-.2519191513/(x+.8762334325-.1474051333/(x+.6689843365-.08508106294/(x+.5052480909-.04760744756/(x+.3736468966-.02517899989/(x+.2662922546-.01205529562/(x+.1775757736-.004776701678/(x+.1034223341-.00120693593/(x+.04081521903)))))))))))))))))))			
19	19	19	.04415040140
3. 6.990216156-684/(x+213.5-4096.527778/(x+53.51440922-306.4420261/(x+21.78682566-64.64962313/(x+11.56878593-20.70049939/(x+7.045877335-8.295503452/(x+4.658148527-3.812873532/(x+3.246782796-1.915954692/(x+2.344093098-1.021337292/(x+1.732148606-.5654858853/(x+1.298321366-.3198297183/(x+.9796590809-.1820574485/(x+.7387431821-.1027156593/(x+.5522011787-.05638788011/(x+.4048249104-.02934375737/(x+.2863734948-.01385572791/(x+.1897468912-.005424657303/(x+.1098978142-.001356425099/(x+.04316090146)))))))))))))))))))			
18	18	18	.06838417802
4. 6.879105045-612/(x+191.-3276.444444/(x+47.83408505-244.4941360/(x+19.44315256-51.37607256/(x+10.29965189-16.35772290/(x+6.252371179-6.505771106/(x+4.115734088-2.960983383/(x+2.852786172-1.469226382/(x+2.045022396-.7706606099/(x+1.497429462-.4179248891/(x+1.109223310-.2300561503/(x+.8240716778-.1263086050/(x+.6084909494-.06779695611/(x+.4415665507-.03462236193/(x+.3096897660-.01608928630/(x+.2036969864-.006213643587/(x+.1172356751-.001535501025/(x+.04579228207)))))))))))))))))))			
17	17	17	.1060544270
5. 6.761457986-544/(x+169.75-2585.9375/(x+42.46933535-192.4038364/(x+17.22968448-40.23881201/(x+9.101026317-12.72509946/(x+5.50294977-5.014893134/(x+3.603454925-2.25517236/(x+2.480679476-1.101674979/(x+1.762568074-.5662589344/(x+1.275751761-.2990000385/(x+.9306323925-.1588107352/(x+.6771299935-.08298033185/(x+.4854773379-.04144602127/(x+.3370813173-.01890603201/(x+.2198440262-.007187656418/(x+.1256199049-.001752516492/(x+.04876487426)))))))))))))))))			
16	16	16	.1646975729

6. 6.636457986-480/(x+149.75-2010.604167/(x+37.4201603-149.0712345/(x+15.14642164-30.99803512/(x+7.972909422-9.722248325/(x+4.797613348-3.788682442/(x+3.121311304-1.678511585/(x+2.130463012-.8039818724/(x+1.496730486-.4025868266/(x+1.06711593-.2052076481/(x+.7625491552-.103772234/(x+.5388347455-.05047436551/(x+.36970337-.0225266987/(x+.2387470882-.008409238668/(x+.1352906562-.002018993011/(x+.05214955084))))))))))))))			
15	15	15	.2561330921
7. 6.503124653-420/(x+131-1536.888889/(x+32.68656015-113.4611372/(x+13.19336430-23.4280418/(x+6.91530151-7.273517056/(x+4.136362246-2.794962951/(x+2.669303602-1.215068002/(x+1.802137217-.5673766142/(x+1.247510155-.2744250566/(x+.8715226266-.1332501403/(x+.6049744625-.06275596268/(x+.4091871578-.02728647405/(x+.2611709609-.009969690564/(x+.1465673704-.002351180223/(x+.05603824353)))))))))))))			
14	14	14	.3989340828
8. 6.360267510-364/(x+113.5-1152.083333/(x+28.26853526-84.60305096/(x+11.37051285-17.31723818/(x+5.928203007-5.307981487/(x+3.519196941-2.003569588/(x+2.247432367-.8499043759/(x+1.495702743-.3836369166/(x+1.014907892-.1768806074/(x+.688972907-.08003605055/(x+.4579097995-.03371380078/(x+.2881895378-.01200647251/(x+.1598843084-.002772589493/(x+.06055238008)))))))))))))			
13	13	13	.6223473410
9. 6.206421356-312/(x+97.25-844.3263889/(x+24.16608613-61.59118167/(x+9.677867863-12.46813670/(x+5.011614532-3.759445739/(x+2.946118136-1.386348300/(x+1.855698427-.5690793019/(x+1.211160605-.2450886595/(x+.7989250102-.1053866826/(x+.5194686031-.04268003932/(x+.3213579887-.01473421108/(x+.1758469811-.003318264412/(x+.06585572416)))))))))))			
12	12	12	.9725312518
10. 6.039754690-264/(x+82.25-602.6041667/(x+20.37921348-43.58443496/(x+8.115430139-8.697356044/(x+4.165537006-2.566442227/(x+2.417126906-.9171560499/(x+1.494103085-.3596472083/(x+.9485124742-.1446059/(x+.5995638136-.05570272146/(x+.3630132285-.01850291786/(x+.195325316-.00404229436/(x+.07217454891)))))))))))			
11	11	11	1.522516178
11. 5.857936508-220/(x+68.5-416.75/(x+16.90791842-29.806416/(x+6.68320093-5.835621167/(x+3.389971865-1.672231661/(x+1.932224971-.5718608334/(x+1.162648517-.2096583726/(x+.7077613116-.07561089364/(x+.4168287673-.0239144348/(x+.2196146873-.005031722076/(x+.07983053395)))))))))))			
10	10	10	2.388179787
12. 5.657936508-180/(x+56.-277.4444444/(x+13.75220264-19.54542956/(x+5.381182217-3.727763318/(x+2.684921452-1.024803066/(x+1.49141525-.3283416962/(x+.8613386367-.1081589513/(x+.4889129373-.03207414498/(x+.2507300412-.006433906953/(x+.08929682283)))))))))			
9	9	9	3.754030750
13. 5.435714286-144/(x+44.75-176.2152778/(x+10.91206897-12.15448001/(x+4.209377312-2.232720056/(x+2.050389827-.576873813/(x+1.094703092-.1664887800/(x+.5901812671-.04519105296/(x+.2919808073-.008514552903/(x+.1012987298))))))))			
8	8	8	5.915219562
14. 5.185714286-112/(x+34.75-105.4375/(x+8.387522229-7.051271395/(x+3.167792123-1.223535292/(x+1.486384618-.2858896840/(x+.7420994263-.06820343293/(x+.3491948425-.01179295492/(x+.1170067613)))))))			
7	7	7	9.347034102
15. 4.9-84/(x+26.-58.33333333/(x+6.178571429-3.718207483/(x+2.256438126-.5873594093/(x+.9929218647-.1140250534/(x+.4336306516-.01739854729/(x+.1384379289))))))			
6	6	6	14.82276169
16. 4.566666667-60/(x+18.5-28.97222222/(x+4.285234899-1.702391984/(x+1.475340423-.2254495628/(x+.5700416197-.02818344391/(x+.1693830584)))))			
5	5	5	23.62201190
17. 4.166666667-40/(x+12.25-12.27083333/(x+2.707555178-.6156291106/(x+.8245647361-.05317066656/(x+.2178800857))))			

4	4	4	37.92481404
18. 3.666666667-24/(x+7.25-3.993055556/(x+1.445652174-.1344255409/(x+.3043478261)))			
3	3	3	61.64347749
19. 3-12/(x+3.5-.75/(x+.5))			
2	2	2	102.5125906
20. 0			
0	0	0	361.2882544

Table A.14: Padé Approximations for sqrt(x).

PADÉ APPROXIMATION			
NUM. DEGREE	**DENOM. DEGREE**	**COST**	**ERROR**
1. 41-22960/(x+671.8-5340.845714/(x+74.2-442.0779221/(x+28.23076923-96.27541689/(x+14.71040724-31.41140047 /(x+8.915966387-12.79783309/(x+5.902857143-5.987617391/(x+4.136551724-3.071612136/(x+3.012539185- 1.678790418/(x+2.253071253-.958556418/(x+1.715886618-.5634690144/(x+1.32195122-.3369421079/ (x+1.024489796-.2027523993/(x+.7943781286-.1214278779/(x+.6127110228-.0714672904/(x+.4667817084- .04065662268/(x+.3477931904-.02180946493/(x+.2494983278-.01057043386/(x+.1673615247-.004233016791 /(x+.09802526241-.00107944004/(x+.03896103896))))))))))))))))))			
20	20	20	.09118538392
2. 39.-1976/(x+607.8-4368.96/(x+67.08888889-360.9876543/(x+25.4957265-78.39053254/(x+13.26244344- 25.47295575/(x+8.019607843-10.32290872/(x+5.293333333-4.796660870/(x+3.695172414-2.439492156/ (x+2.67816092-1.318996416/(x+1.990990991-.7430465809/(x+1.504943968-.4294658069/(x+1.148509485- .2513616899/(x+.8793650794-.1471226789/(x+.6711590296-.08493707202/(x+.5067858325-.04753483471/ (x+.3747483463-.02514359635/(x+.2670870113-.01203921222/(x+.1781493868-.004770382934/(x+.1038316458- .001205204581/(x+.04109589041)))))))))))))))))			
19	19	19	.1398690501
3. 37-16872/(x+547-3536/(x+60.33333333-291.5555556/(x+22.8974359-63.10059172/(x+11.88687783-20.4072118/ (x+7.168067227-8.21781161/(x+4.714285714-3.787439614/(x+3.275862069-1.906349754/(x+2.360501567- 1.017328712/(x+1.742014742-.5636905096/(x+1.304548451-.3189855259/(x+.9837398374-.1816482489/ (x+.7414965986-.1025146489/(x+.5541008856-.05628965719/(x+.4061569017-.0292971501/(x+.2873166523- .01383505133/(x+.1904161412-.005416703465/(x+.110367893-.001354287165/(x+.04347826087)))))))))))))))))			
18	18	18	.2154105542
4. 35-14280/(x+489.4-2828.068571/(x+53.93333333-232.6118326/(x+20.43589744-50.14394836/(x+10.58371041- 16.12558732/(x+6.361344538-6.444683136/(x+4.165714286-2.941155556/(x+2.87862069-1.461820073/ (x+2.059561129-.7676124462/(x+1.506142506-.4165838544/(x+1.114700066-.2294400535/(x+.8276422764- .1260190225/(x+.6108843537-.06766058573/(x+.4432036966-.03455965641/(x+.3108242304-.01606221376/ (x+.2044866264-.006203475291/(x+.1177805801-.001532826148/(x+.04615384615)))))))))))))))))			
17	17	17	.3328325129
5. 33-11968/(x+435.-2232/(x+47.88888889-183.0482604/(x+18.11111111-39.27272727/(x+9.352941176-12.54416318/ (x+5.599439776-4.967657238/(x+3.647619048-2.24/(x+2.503448276-1.096083771/(x+1.775339603-.5639977487/ (x+1.283374283-.2980277575/(x+.9353988134-.1583774885/(x+.6802168022-.08278507767/(x+.4875283447- .04135948778/(x+.3384674625-.01886982826/(x+.2207878186-.00717442512 8/(x+.1262582686-.001749119746/ (x+.04918032787)))))))))))))))))			
16	16	16	.5156804731
6. 31-9920/(x+383.8-1735.36/(x+42.2-141.8181818/(x+15.92307692-30.2528241/(x+8.194570136-9.583682389/ (x+4.882352941-3.752860412/(x+3.16-1.667153623/(x+2.150344828-.7998670298/(x+1.507836991-.4009597441/ (x+1.073710074-.2045288532/(x+.7666446935-.1034821466/(x+.5414634146-.05035131123/(x+.3714285714- .02247707594/(x+.2398921833-.008391666968/(x+.1360476663-.002014605893/(x+.05263157895)))))))))))))))))			

15	15	15	.8009340032
7. 29-8120/(x+335.8-1326.445714/(x+36.86666667-107.9365079/(x+13.87179487-22.86390533/(x+7.108597285-7.169550173/(x+4.210084034-2.768411713/(x+2.702857143-1.206786473/(x+1.819310345-.5644415464/(x+1.257053292-.273298549/(x+.8771498771-.1327992679/(x+.608437706-.06257456004/(x+.4113821138-.02721644950/(x+.262585034-.009945793341/(x+.1474778591-.002345402383/(x+.05660377358))))))))))))			
14	14	14	1.246791792
8. 27-6552/(x+291.-994.2857143/(x+31.88888889-80.47971781/(x+11.95726496-16.89940828/(x+6.095022624-5.23183391/(x+3.582633053-1.984422755/(x+2.276190476-.844057971/(x+1.510344828-.3816245381/(x+1.022988506-.1761392729/(x+.6936936937-.07975662030/(x+.460777851-.03361147796/(x+.2899728997-.01197304842/(x+.1609977324-.002764805445/(x+.0612244898)))))))))))			
13	13	13	1.945031218
9. 25-5200/(x+249.4-728.64/(x+27.26666667-58.58585859/(x+10.17948718-12.16654115/(x+5.153846154-3.705263158/(x+3.-1.372997712/(x+1.88-.5651169082/(x+1.223448276-.243778741/(x+.8056426332-.1049320181/(x+.5233415233-.04252402090/(x+.3236651285-.01468586613/(x+.1772357724-.003307493540 /(x+.06666666667))))))))))			
12	12	12	3.040643795
10. 23-4048/(x+211-520/(x+23-41.45454545/(x+8.538461538-8.486282948/(x+4.285067873-2.529229649/(x+2.462184874-.9082333162/(x+1.514285714-.3571014493/(x+.9586206897-.1438124097/(x+.605015674-.05545187950/(x+.3660933661-.01843007248/(x+.1970995386-.004026907061/(x+.07317073171)))))))))))			
11	11	11	4.763007369
11. 21-3080/(x+175.8-359.5885714/(x+19.08888889-28.34696168/(x+7.034188034-5.69338354/(x+3.488687783-1.647787288/(x+1.969187675-.5662188598/(x+1.179047619-.2081391304/(x+.715862069-.07517931802/(x+.421107628-.02379894462/(x+.2219492219-.005008869874/(x+.08108108108)))))))))))			
10	10	10	7.475223130
12. 19-2280/(x+143.8-239.36/(x+15.53333333-18.58585859/(x+5.666666667-3.636363636/(x+2.764705882-1.009652158/(x+1.521008403-.325036193/(x+.8742857143-.1073468599/(x+.4951724138-.03187875843/(x+.2539184953-.006398293788/(x+.09090909091))))))))))			
9	9	9	11.75167010
13. 17-1632/(x+115-152/(x+12.33333333-11.55555556/(x+4.435897436-2.177514793/(x+2.113122172-.5682025132/(x+1.117647059-.1647597254/(x+.6-.04483091787/(x+.2965517241-.008455542344/(x+.1034482759))))))))			
8	8	8	18.49833684
14. 15-1120/(x+89.4-90.92571429/(x+9.488888889-6.701940035/(x+3.341880342-1.192899408/(x+1.533936652-.2814787835/(x+.7591036415-.06745642574/(x+.3561904762-.01168695652/(x+.12)))))))			
7	7	7	29.13522737
15. 13-728/(x+67-50.28571429/(x+7-3.532467532/(x+2.384615385-.5723507262/(x+1.027149321-.1121835731/(x+.4453781513-.01718582170/(x+.1428571429))))))			
6	6	6	45.86177149
16. 11-440/(x+47.8-24.96/(x+4.866666667-1.616161616/(x+1.564102564-.2194728349/(x+.592760181-.0276816609/(x+.1764705882)))))			
5	5	5	72.01547528
17. 9-240/(x+31.8-10.56/(x+3.088888889-.5836139169/(x+.8803418803-.05164066703/(x+.2307692308))))			
4	4	4	112.4900312
18. 7-112/(x+19-3.428571429/(x+1.666666667-.1269841270/(x+.3333333333)))			
3	3	3	174.0621286
19. 5-40/(x+9.4-.64/(x+.6))			
2	2	2	265.2268546

20. 3-8/(x+3)			
1	1	1	394.6880113

Table A.15: Padé Approximations for arcsinh(x).

PADÉ APPROXIMATION			
NUM. DEGREE	**DENOM. DEGREE**	**COST**	**ERROR**
1. 89.05078273/(x+235.6852216/(x+16.81451427/(x+18.41805995/(x+3.826223223/(x+6.905604077/(x+1.407336121/(x+3.925515866/(x+.6050173677/(x+2.699119231/(x+.2751416523/(x+2.054260408/(x+.1251850398/(x+1.662231028/(x+.05363735774/(x+1.401551874/(x+.01954987495/(x+1.217313711/(x+.004633124087/(x+1.081080442/x))))))))))))))))))))			
20	20	20	256.2274138
2. 78.645922/(x+194.938632/(x+13.33875825/(x+15.33368144/(x+2.96708953/(x+5.830156259/(x+1.050592423/(x+3.365351488/(x+.4276918656/(x+2.343586126/(x+.1802203341/(x+1.801404587/(x+.07303251146/(x+1.469625134/(x+.02554178286/(x+1.248049596/(x+.005864955859/(x+1.090908074/x))))))))))))))))))			
18	18	18	269.5718297
3. 68.44032165/(x+157.9099196/(x+10.27195781/(x+12.52808551/(x+2.216169965/(x+4.847984955/(x+.744827727/(x+2.85012065/(x+.280613585/(x+2.013778709/(x+.1051849876/(x+1.565256791/(x+.03477941921/(x+1.288882969/(x+.007662541124/(x+1.103446567/x))))))))))))))))			
15	17	16	283.6586549
4. 58.45716257/(x+124.6252734/(x+7.611707503/(x+10.00268142/(x+1.57340465/(x+3.959228601/(x+.4904189929/(x+2.379549468/(x+.1640427822/(x+1.709477624/(x+.05010600850/(x+1.345754519/(x+.01043406307/(x+1.119996965/x))))))))))))))			
14	15	14	298.5407985
5. 48.72581666/(x+95.11569074/(x+5.35525492/(x+7.759119351/(x+1.038859802/(x+3.16396096/(x+.2878960606/(x+1.953240309/(x+.07825337505/(x+1.430449952/(x+.01503467969/(x+1.142852073/x))))))))))))			
12	12	12	314.2775354
6. 39.28473977/(x+69.41876945/(x+3.499411962/(x+5.799369798/(x+.612845477/(x+2.462114544/(x+.1379878827/(x+1.570630820/(x+.02350508316/(x+1.176473049/x)))))))))			
9	10	10	330.9355171
7. 30.18652253/(x+47.58163373/(x+2.040443387/(x+4.125843526/(x+.2961561984/(x+1.853315118/(x+.04165063345*1/(x+1.230965452/x)))))))			
8	8	8	348.5894762
8. 21.50760255/(x+29.66614223/(x+.9739583066/(x+2.741577146/(x+.09058445769/(x+1.336508727/x)))))			
5	6	6	367.3213224
9. 13.36976321/(x+15.75941395/(x+.2950030564/(x+1.650501031/x)))			
4	4	4	387.2126918
10. 6/(x+6/x)			
0	3	2	408.3097844
11. 0			
0	0	0	430.5634944

Table A.16: Padé Approximations for exp(-x).

PADÉ APPROXIMATION			
NUM. DEGREE	**DENOM. DEGREE**	**COST**	**ERROR**
1. .3379030567e16/(x^14+266*x^13+37226*x^12+3634176*x^11+277053756*x^10+.1751545270e11*x^9+ .9526926876e12*x^8+.4568500161e14*x^7+.1964805622e16*x^6+.7673271864e17*x^5+.2746275673e19*x^4+.9069 534716e20*x^3+.2777884108e22*x^2+.7920385906e23*x+.2107767957e25+.5244598280e26/(x-23.29162812+ 34.27332657/(x-18.72162239+55.41237735/(x-15.17375246+96.73438973/(x-17.04913701+129.1906262/(x- 25.15164139+53.75436362/(x-26.61221863)))))))			
6	20	33	.3155118435e-8
2. -.4827186524e15/(x^13+307*x^12+49107*x^11+5432925*x^10+465826791*x^9+.3289683918e11*x^8 +.1986453634e13*x^7+.1051597219e15*x^6+.4967151662e16*x^5+.2120430535e18*x^4+.8259515526e19*x^3+.295 6892079e21*x^2+.9782283846e22*x+.3002981787e24+.8580082290e25/(x-26.65116526+46.726719/(x-21.14886938+ 68.61628909/(x-17.30688964+88.4428926/(x-15.38043474+122.0928983/(x-18.62224461+112.8159629/(x- 23.29241839+52.82061669/(x-24.59797798))))))))			
7	20	32	.3967064108e-8
3. .6033983155e14/(x^12+348*x^11+62628*x^10+7743432*x^9+737527176*x^8+.5754178310e11*x^7+ .3819540247e13*x^6+.2212540017e15*x^5+.1138726840e17*x^4+.5276039859e18*x^3+.2222498603e20*x^2+.8575 906631e21*x+.3048729591e23+.1002914835e25/(x-30.54170496+64.62167350/(x-23.70280148+89.35647487/(x- 19.15586981+102.4180937/(x-16.20482134+118.583517/(x-15.63293313+133.9463347/(x-18.58068135+106.9238447/ (x-21.4951373+53.39519202/(x-22.68605063)))))))))			
8	20	31	.6181992607e-8
4. -.6704425728e13/(x^11+389*x^10+77789*x^9+10629657*x^8+1113440877*x^7+.9510934753e11*x^6+ .6883428860e13*x^5+.4330899545e15*x^4+.2412461203e17*x^3+.1205793924e19*x^2+.5462646740e20*x+.226051 5291e22+.8595582280e23/(x-35.10184308+90.69461009/(x-26.52686704+119.1126424/(x-21.00854263+128.1254093 /(x-17.3482099+133.8854021/(x-15.27416469+142.4529034/(x-15.44706934+140.5733955/(x-17.7426086+ 105.9001079 /(x-19.74367372+54.55068219/(x-20.80702101)))))))))			
9	20	30	.1177683005e-7
5. .3016991578e13/(x^11+331*x^10+56590*x^9+6638590*x^8+599119120*x^7+.4422853720e11*x^6+ .2773777521e13*x^5+.1515692532e15*x^4+.7346360241e16*x^3+.3199739805e18*x^2+.1264628846e20*x+.456886 5507e21+.1517283631e23/(x-30.65413562+69.60606105/(x-23.27529068+92.93341247/(x-18.55076220+103.0944879 /(x-15.60134586+113.7145435/(x-14.81707454+121.9195831/(x-16.91353238+99.03430732/(x-19.46230071+ 51.14317306/(x-20.72555800)))))))))			
8	19	29	.2300520244e-7
6. -.1270312243e13/(x^11+277*x^10+39877.*x^9+3960001*x^8+303918373*x^7+.1915502573e11*x^6+ .1029096048e13*x^5+.4831082243e14*x^4+.2016458354e16*x^3+.7577818566e17*x^2+.2587764657e19*x+.808542 7106e20+.2323066651e22/(x-26.51598705+52.5930233/(x-20.25118227+72.12961641/(x-16.30540342 +84.78058093 /(x-14.41485477+99.49805581/(x-15.84693435+91.40769125/(x-19.03085938+48.06863374/(x-20.63477876))))))))			
7	18	28	.4727853791e-7
7. .1587890304e12/(x^10+314*x^9+50858*x^8+5644176*x^7+481169552*x^6+.3350171670e11*x^5+ .1978309108e13*x^4+.1016069220e15*x^3+.4620098654e16*x^2+.1883971269e18*x+.6955736685e19+.2341872470 e21/(x-30.86668459+75.83577523/(x-22.83621850+97.21934197/(x-17.89246763+104.0459647/(x-14.89758665+ 109.6770707/(x-13.90870331+112.2940580/(x-15.33271816+91.79514603/(x-17.49501305+48.77217498/(x- 18.77060810)))))))))			
8	18	27	.8992298410e-7
8. -.7057290240e11/(x^10+262*x^9+35622*x^8+3335688*x^7+241004484*x^6+.1427485097e11*x^5+ .7193834651e12*x^4+.3161575170e14*x^3+.1232748876e16*x^2+.4317645726e17*x+.1370691416e19+.3970181245 e20/(x-26.55209451+56.61000809/(x-19.77683617+74.38012731/(x-15.69474910+83.39143794/(x-13.71718720+ 91.82061775/(x-14.52718799+83.03097931/(x-17.06766840+45.42836132/(x-18.66427663))))))))			
7	17	26	.1746363296e-6

9. .2905943040e11/(x^10+214*x^9+23958*x^8+1859976*x^7+112062852*x^6+5562511704*x^5+.2359073836e12*x^4+.8755582135e13*x^3+.2891256063e15*x^2+.8594588766e16*x+.2319014091e18+.5712497706e19/(x-22.57446878+41.54237987/(x-16.97213841+57.07810151/(x-13.85849667+69.13921334/(x-13.64543594+72.32550196/(x-16.41896237+42.35362834/(x-18.53049784)))))))			
6	16	25	.3597665847e-6
10. -4151347200/(x^9+247*x^8+31607*x^7+2780701*x^6+188403467*x^5+.1044410998e11*x^4+.4915631124e12*x^3+.2013050700e14*x^2+.7296048172e15*x+.2368916950e17+.6950843831e18/(x-26.68240338+61.71921888/(x-19.27772822+77.16663776/(x-15.00940005+82.40608786/(x-12.91114952+85.69993953/(x-13.22102651+75.89417429/(x-15.19073716+42.74282374/(x-16.70755516))))))))			
7	16	24	.6816921402e-6
11. 1816214400/(x^9+201*x^8+21096*x^7+1532376*x^6+86202576*x^5+3986255376*x^4+.1571221446e12*x^3+.5405837041e13*x^2+.1650131690e15*x+.4520077303e16+.1119887134e18/(x-22.52788158+44.56059684/(x-16.45541136+57.87537982/(x-13.21025600+65.42760281/(x-12.62381743+65.00497464/(x-14.62720292+39.22894232/(x-16.55543071)))))))			
6	15	23	.1315990430e-5
12. -726485760/(x^9+159*x^8+13347*x^7+782529*x^6+35803899*x^5+1355331465*x^4+.4396744724e11*x^3+.1250540773e13*x^2+.3166675751e14*x+.7213168449e15+.1487784575e17/(x-18.74579328+31.64308337/(x-13.95566393+44.18248883/(x-12.13820226+51.73022467/(x-13.82080925+35.69236626/(x-16.33953128))))))			
5	14	22	.2722076548e-5
13. 121080960/(x^8+188*x^7+18416*x^6+1245696*x^5+65099280*x^4+2789437440*x^3+.1015975843e12*x^2+.3220267617e13*x+.9025634355e14+.2261519858e16/(x-22.56617616+48.49259977/(x-15.8956353+59.09771966/(x-12.4653612+62.5196819/(x-11.54910738+59.08736459/(x-12.91573187+36.24267655/(x-14.60798809)))))))			
6	14	21	.5125322882e-5
14. -51891840/(x^8+148*x^7+11536*x^6+626416*x^5+26472880*x^4+922915840*x^3+.2748693472e11*x^2+.7152571014e12*x+.1650559256e14+.3410656023e15/(x-18.61002568+33.62472046/(x-13.39225409+43.34817153/(x-11.32262219+46.72731334/(x-12.29170854+32.22744018/(x-14.38338950))))))			
5	13	20	.9828439143e-5
15. 19958400/(x^8+112*x^7+6712*x^6+283840*x^5+9441136*x^4+261363712*x^3+6227253952*x^2+.1304243603e12*x+.2433487744e13+.4076028844e14/(x-15.07922806+23.06364739/(x-11.39813757+32.39064328/(x-11.4849561+27.67635272/(x-14.03767827)))))			
4	12	19	.2045873862e-4
16. -3991680/(x^7+137*x^6+9857*x^5+492585*x^4+19096865*x^3+608659465*x^2+.1651046271e11*x+.3896645245e12+.8116260427e13/(x-18.54453712+36.32632200/(x-12.76171007+42.97562258/(x-10.41769993+42.82383590/(x-10.80409343+29.15810958/(x-12.47195945))))))			
5	12	18	.3814642508e-4
17. 1663200/(x^7+103*x^6+5658*x^5+218562*x^4+6616536*x^3+166059144*x^2+3571617168*x+.6719708866e11+.1119654685e13/(x-14.84633646+23.95694966/(x-10.74048048+30.04403425/(x-10.26121929+24.36625208/(x-12.15196376)))))			
4	11	17	.7256117626e-4
18. -604800/(x^7+73*x^6+2913*x^5+83337*x^4+1898241*x^3+36332649*x^2+602981073*x+8840603961+.1156056275e12/(x-11.67434860+15.80182740/(x-9.686751938+18.49195211/(x-11.63889946))))			
3	10	16	.1525600013e-3
19. 151200/(x^6+94*x^5+4694*x^4+164152*x^3+4479032*x^2+100840624*x+1935316784+.3228823629e11/(x-14.66092498+25.39230136/(x-9.995163108+28.35016956/(x-9.020467172+21.65170216/(x-10.32344473)))))			
4	10	15	.2801684915e-3
20. -60480/(x^6+66*x^5+2370*x^4+60720*x^3+1232460*x^2+20911320*x+305858520.+3923640000/(x-11.32588770+15.58112710/(x-8.767089391+16.06221865/(x-9.907022912))))			
3	9	14	.5273112295e-3

21. 20160/(x^6+42*x^5+1002*x^4+17616*x^3+252108*x^2+3093624*x+33559992.+327105216*1/(x-8.77182285 +9.052064812/(x-9.22817715)))			
2	8	13	.001127671514
22. -6720/(x^5+59*x^4+1883*x^3+42621*x^2+759567*x+11241129+142315623/(x-10.99125091+15.79963295/(x-7.780805673+14.22416024/(x-8.227943415))))			
3	8	12	.002019591062
23. 2520/(x^5+37*x^4+772*x^3+11788*x^2+145624*x+1534168+14202880/(x-8.235472242+8.055447177/(x-7.764527758)))			
2	7	11	.003744171624
24. -720/(x^5+19*x^4+223*x^3+2041*x^2+16087*x+116929+823543*1/(x-7))			
1	6	10	.008278633905
25. 360/(x^4+32*x^3+572*x^2+7416*x+77192.+680512/(x-7.677419355+7.458896982/(x-6.322580645)))			
2	6	9	.01414324321
26. -120/(x^4+16*x^3+156*x^2+1176*x+7656+46656/(x-6.))			
1	5	8	.02560401406
27. 24/(x^4+4*x^3+12*x^2+24*x+24)			
0	4	7	.06181188285
28. -24/(x^3+13*x^2+101*x+601+3125/(x-5))			
1	4	6	.09403937231
29. 6/(x^3+3*x^2+6*x+6.)			
0	3	5	.1631035181
30. -6/(x^2+10*x+58+256/(x-4))			
1	3	4	.4735915028
31. 2/(x^2+2*x+2)			
0	2	3	.5509853262
32. 1/(x+1)			
0	1	1	3.614624695

Table A.17: Padé Approximations for tanh(x).

PADÉ APPROXIMATION			
NUM. DEGREE	**DENOM. DEGREE**	**COST**	**ERROR**
1. 210/(x+14734.5/(x+2967.734043/(x+1286.960862/(x+726.9322691/(x+472.6286465/(x+335.9707735/(x+ 254.2655889/(x+201.6932413/(x+166.0076055/(x+140.7574087/(x+122.2209024/(x+108.0254996/(x+96.45065982/(x+ 86.06565456/(x+75.60664115/(x+64.05685729/(x+50.80117549/(x+35.64978680/(x+18.67238462/x))))))))))))))))))			
20	20	20	.5121115434
2. .005263157895*x+63.49736842/(x+2431.732376/(x+1055.813704/(x+597.4003635/(x+389.2818623/(x+ 277.4877829/(x+210.6923566/(x+167.7513297/(x+138.6227721/(x+117.9891245/(x+102.7318404/(x+90.78276254/(x+ 80.54957489/(x+70.66306440/(x+59.97950613/(x+47.73386987/(x+33.62136374/(x+17.66634661/x)))))))))))))))			
20	18	19	.8371640142
3. 171/(x+9775/(x+1971.930435/(x+857.3943584/(x+486.1049699/(x+317.5835556/(x+227.1038940/(x+173.0846554/ (x+138.3861453/(x+114.8471005/(x+98.10393592/(x+85.52488613/(x+75.26502146/(x+65.82368834/(x+55.93116109/ (x+44.66690997/(x+31.58987447/(x+16.65940867/x)))))))))))))))))			

18	18	18	1.273144625
4. .006535947712*x+51.16339869/(x+1580.728155/(x+688.4454927/(x+391.2361174/(x+256.3812231/(x+184.0197159/(x+140.8529431/(x+113.1394686/(x+94.30287423/(x+80.76426990/(x+70.26690475/(x+61.12268764/(x+51.92823873/(x+41.60487/(x+29.55560114/(x+15.6514374/x)))))))))))))))			
18	16	17	2.062490667
5. 136/(x+6187.5/(x+1250.925455/(x+545.8811374/(x+311.0789036/(x+204.5825408/(x+147.4767507/(x+113.4358191/(x+91.57036282/(x+76.61727612/(x+65.62197097/(x+56.60134189/(x+47.99212049/(x+38.55467577/(x+27.51933395/(x+14.64231188/x)))))))))))))			
16	16	16	2.923681344
6. .008333333333*x+40.1625/(x+975.7222222/(x+426.7866601/(x+244.0134118/(x+161.1551314/(x+116.756761/(x+90.29839218/(x+73.25138368/(x+61.41521914/(x+52.30974437/(x+44.15008789/(x+35.52636445/(x+25.48266648/(x+13.63195531/x)))))))))))			
16	14	15	4.805264414
7. 105/(x+3692/(x+748.7183099/(x+328.4187224/(x+188.5145796/(x+125.1261795/(x+91.18069965/(x+70.92954787/(x+57.76263119/(x+48.31004474/(x+40.43649524/(x+32.53398878/(x+23.44841038/(x+12.62039080/x)))))))))))			
14	14	14	6.173809808
8. .01098901099*x+30.49450549/(x+563.9135135/(x+248.2052128/(x+143.1519803/(x+95.58177545/(x+70.10687288/(x+54.83741010/(x+44.68302301/(x+36.89452526/(x+29.59669047/(x+21.42116147/(x+11.60783479/x)))))))))))			
14	12	13	10.68949113
9. 78/(x+2040.5/(x+415.7075472/(x+183.7451364/(x+106.5894536/(x+71.66577154/(x+52.92775529/(x+41.54204953/(x+33.57935691/(x+26.74002784/(x+19.40805323/(x+10.59484852/x)))))))))))			
12	12	12	11.93716932
10. 55/(x+1017/(x+208.6902655/(x+93.33564411/(x+54.98688548/(x+37.56927881/(x+27.95791615/(x+21.4151885/(x+15.47171167/(x+8.573109797/x))))))))))			
10	10	10	21.08701846
11. 36/(x+437.5/(x+90.86/(x+41.39302663/(x+24.86059394/(x+17.01277348/(x+11.79432026/(x+6.579285692/x)))))))			
8	8	8	34.06781061
12. 21/(x+150/(x+31.8/(x+14.81320755/(x+8.708534906/(x+4.678257547/x)))))			
6	6	6	50.52902829
13. 10/(x+34.5/(x+7.456521739/(x+3.043478261/x)))			
4	4	4	69.09094069
14. 3/(x+3/x)			
2	2	2	87.09031056
15. 0			
0	0	0	99.25601921

Table A.18: Evolved Approximations for x^y.

EVOLVED LISP EXPRESSION			
ERROR	**COST**	**RUN**	**GENERATION**
1. `(% X (+ (* Y Y) (- X (* (* (* Y Y) X) Y))))`			
3.36649	5.2	37	33
2. `(% X (+ (* Y Y) (- X (* X (* Y Y)))))`			

4.32633	4.2	37	15
3. (% X (+ Y (- X (* Y (% X 0.98529)))))			
4.93578	3.2	4	11
4. (% X (+ Y (- X (* Y X))))			
4.99067	2.2	4	4
5. (- (- (* X Y) Y) -0.989868)			
6.42354	1.2	12	5
6. (* (* -0.228126 -3.54549) (+ 0.286111 X))			
12.5833	1.1	32	9
7. (+ (- X -0.00503555) 0.12833)			
13.2616	0.2	7	33
8. (+ 0.13657 X)			
13.2626	0.1	42	3
9. X			
18.7783	0	0	0

Table A.19: Maple Evaluation of Approximations for x^y.

SIMPLIFIED MAPLE EXPRESSION	COST	ERROR
1. x/(y^2+x-x*y^3)	4	.03643611691
2. x/(y^2+x-x*y^2)	3	.04650160477
3. x/(y+x-1.014929615*x*y)	3	.04676595598
4. x/(y+x-x*y)	2	.04745973920
5. x*y-y+.989868	1	.05509570980
6. .2314118560+.8088184517*x	1	.1275586202
7. x+.13336555	0	.1401316648
8. .13657+x	0	.1401367539
9. x	0	.1948507892

Table A.20: Final Evolved Approximations for x^y.

EXPRESSION	COST	ERROR
(1) x/(y^2+x-x*y^3)	4	.03643611691
(2) x/(y^2+x-x*y^2)	3	.04650160477
(4) x/(y+x-x*y)	2	.04745973920
(5) x*y-y+.989868	1	.05509570980
(7) x+.13336555	0	.1401316648

Table A.21: Evolved Approximations for sin(x).

EVOLVED LISP EXPRESSION			
ERROR	**COST**	**RUN**	**GENERATION**
1. (- X (* (% (* X (+ X (* (% -2.36473 (* (* 4.36827 0.735038) -3.33247)) (% X (+ (% (* -4.60814 -0.66393) (% X X)) (+ -4.96673 (+ (+ (* X -2.81365) (% (+ (- X (+ 4.91791 X)) X) -4.80499)) (% (* (% X (+ 4.91791 X)) 2.16636) X)))))))) (+ 4.91791 X)) X))			
0.0101724	13.1	10	24
2. (- X (* (% (* X (+ X (* (% -2.36473 (* (* 4.36827 0.735038) -3.33247)) (% X (+ (* X -2.81365) (% (+ (* 4.36827 0.735038) X) -4.80499)))))) (+ 4.91791 X)) X))			
0.0123257	7.5	10	13
3. (- X (* (% (* X (+ X (* (% -2.36473 (* (* 4.36827 0.735038) -3.33247)) (% X (* 4.36827 (% X -1.32191)))))) (+ 4.91791 X)) X))			
0.0322861	7.2	10	9
4. (- (* X (% X (+ (* (% (- X -3.93155) (- (* X (% X (+ (* 2.72729 -3.8522) X))) (- (- X X) X))) -1.61855) (* -1.58315 -1.61855)))) (- (- X X) X))			
0.0328475	6.9	41	39
5. (- X (* (% (* X (+ X (* (% -2.36473 (* 2.16636 -3.33247)) (% X (- X (* X (+ 4.91791 X))))))) (+ 4.91791 X)) X))			
0.0377116	6.5	10	23
6. (- X (* (% (* X (+ X (% -2.36473 (* (* 4.36827 -2.36869) -3.33247)))) (+ 4.91791 X)) X))			
0.0424079	3.3	10	42
7. (- X (* X (* 0.137486 (* X (+ 0.137486 X)))))			
0.0737534	3.2	39	12
8. (+ (* (* (* -0.155492 X) X) X) X)			
0.190762	3.1	9	6
9. (* (% (+ -3.37336 X) -2.85028) X)			
0.779006	2.1	17	4
10. (% (+ X X) (+ 1.44734 X))			
1.18729	1.2	42	3
11. (+ 0.0840785 (* X 0.739616))			
2.65241	1.1	40	6
12. (* X 0.810419)			
3.42891	1	5	1
13. (- X 0.0758385)			
6.2441	0.1	3	3
14. X			
7.15614	0	0	0

Table A.22: Evolved Approximations for Refinement of Candidate Approximation 3 for sin(x).

EVOLVED LISP EXPRESSION			
ERROR	**COST**	**RUN**	**GENERATION**
1. (% (* (% X 4.43754) 0.0270089) (- (+ (% (* (% (% X 4.43754) (* (% X 4.43754) 0.0270089)) 0.0270089) (- (* (% X (+ X 0.709708)) (+ X -1.13941)) (* (+ (% -3.34071 (+ (* (- X 2.86187) (* 1.02924 1.02405)) (+ (- X X) -1.13941))) (+ X 4.79949)) (- (+ (- (- X X) 2.86187) (% (* X 0.0270089) (% (* (* (% X 4.43754) 0.0270089) 0.0270089) 0.0270089))) X)))) (- -3.74172 1.82455)) (* 3.01508 (% X (+ X 0.709708)))))			
0.00799294	23.6	8	41
2. (% (* (% X 4.43754) 0.0270089) (- (+ (% X (- -0.598621 X)) (- -3.74172 1.82455)) (* (+ (* (* (- X (+ (- X 2.86187) (% X (- -2.3867 X)))) (% X (% X (* X (* (+ -4.55351 (* (- X X) -2.9519)) (% X 3.01508)))))) -1.92953) (+ (% X 4.43754) 4.79949)) (% X (* (- X (% 2.80328 X)) (* (- -2.3867 X) 4.13633))))))			
0.00824729	20.3	8	13
3. (% (* (% X 4.43754) 0.0270089) (- (+ (% X (- -0.598621 X)) (- -3.74172 1.82455)) (* (+ (* (* (- X (+ (- X X) (% X (- -2.3867 X)))) (+ X -4.55351)) -1.92953) (+ (% X 4.43754) 4.79949)) (% X (* (- X (% 2.80328 X)) (* (- -2.3867 X) 4.13633))))))			
0.00841626	14.2	8	11
4. (% (% (% -0.12711 (- (+ X -4.05148) (% (% (% 2.77215 (- (+ (% (- -2.40776 (- (+ X -2.40776) -1.59749)) X) (- (- X -0.463118) -1.59749)) X)) 4.72747) (% (- (- X -0.463118) -1.59749) (% (* (+ X -4.05148) (- X -0.463118)) 4.72747))))) 4.72747) (% (- -2.40776 (* (- (- X -0.463118) -1.59749) 2.77215)) X))			
0.00876727	13.7	44	49
5. (% (% X -3.4106) (* (* (- (* (* (- X (% X X)) 1.5389) 2.51305) (* (+ X 2.83135) (+ -3.62423 -1.99454))) -3.99167) -3.99167))			
0.00888704	8.3	19	20
6. (% (% X -3.4106) (* (* (- (- X X) (* (+ X 2.83135) (+ -3.62423 (* X (+ -3.44874 X))))) -3.99167) -3.99167))			
0.0089408	6.5	19	36
7. (% (% (% -0.12711 (- (+ X -4.05148) X)) 4.72747) (% (- -2.40776 (+ -3.30348 (* (- (- X -0.463118) -1.59749) 2.77215))) X))			
0.00895232	5.6	44	15
8. (% (% (% -0.12711 (- (+ X -4.05148) (- X X))) 4.72747) (% (- -2.40776 (* (+ (+ X X) X) 2.77215)) X))			
0.00897132	5.6	44	0
9. (% (% X -3.4106) (* (* (- (* (+ X 2.83135) (+ X 2.83135)) (+ -3.62423 -1.99454)) -3.99167) -3.99167))			
0.00903882	5.3	19	28
10. (% (% X -3.4106) (* (* (- (% 3.31751 -4.20133) (* (+ X 2.83135) (+ -3.62423 -1.99454))) -3.99167) -3.99167))			
0.00915817	5.2	19	13
11. (% (* X 0.0270089) (- (+ (- X X) (+ X 0.709708)) (* (+ (+ X 0.709708) (+ X 4.79949)) 4.43754)))			
0.0091977	3.7	8	34

12. (% (% (% -0.12711 -2.57591) 4.72747) (% (- -2.40776 (* (- (- X -0.463118) -1.59749) 2.77215)) X))			
0.0094937	3.3	44	14
13. (* (* 4.05911 X) (% (% -4.10947 (% (+ 3.27723 1.21174) (% 0.00167852 -2.90094))) -2.45537))			
0.00989481	2	13	12
14. (% 0.00167852 -2.90094)			
0.0144338	0	13	2

Table A.23: Maple Evaluation of Approximations for Refinement of Candidate Approximation 3 for sin(x).

SIMPLIFIED MAPLE EXPRESSION	COST	ERROR
1. .006086457814*x/(.9999999999/(x*(x-1.13941)/(x+.709708)-(-3.34071/(1.053993222*x-4.155801582)+x+4.79949)*(-x+1.575669997))-5.56627-3.01508*x/(x+.709708))	12 (+4)	.4332379048e-3
2. .006086457814*x/(x/(-.598621-x)-5.56627-(2.914063359*(2.86187-x/(-2.3867-x))*x^2+.2253500814*x+4.79949)*x/((x-2.80328/x)*(-9.872178811-4.13633*x)))	17 (+4)	INF
3. .006086457814*x/(x/(-.598621-x)-5.56627-(-1.92953*(x-x/(-2.3867-x))*(x-4.55351)+.2253500814*x+4.79949)*x/((x-2.80328/x)*(-9.872178811-4.13633*x)))	16 (+4)	INF
4. -.02688753181*x/((x-4.05148-.1240392589*(x-4.05148)*(x+.463118)/(((-1.59749-x)/x+2.060608)*(x+2.060608)))*(-8.120074467-2.77215*x))	10 (+4)	INF
5. -.2932035419*x/(151.1461467*x+191.8603473)	3 (+4)	.3095090765e-3
6. .01840178500*x/((x+2.83135)*(-3.62423+x*(-3.44874+x)))	4 (+4)	.3104470014e-3
7. .006636471563*x/(-4.816594467-2.77215*x)	3 (+4)	.3133582952e-3
8. -.02688753181*x/((x-4.05148)*(-2.40776-8.31645*x))	4 (+4)	.3124598540e-3
9. -.2932035419*x/(15.93342939*(x+2.83135)^2+89.52627505)	4 (+4)	.3162295205e-3
10. -.2932035419*x/(240.8986528+89.52627505*x)	3 (+4)	.3200857218e-3
11. .0270089*x/(-7.87508*x-23.73757849)	3 (+4)	.3215765352e-3
12. .01043807113*x/(-8.120074467-2.77215*x)	3 (+4)	.3329046160e-3
13. -.8756705834e-3*x	1 (+4)	.3462600904e-3
14. -.5786124498e-3	0 (+4)	.4676456530e-3

Table A.24: Final Evolved Approximations for Refinement of Candidate Approximation 3 for sin(x).

EVOLVED APPROXIMATION	COST	ERROR
(5) -.2932035419*x/(151.1461467*x+191.8603473)	3 (+4)	.3095090765e-3
(13) -.8756705834e-3*x	1 (+4)	.3462600904e-3
(14) -.5786124498e-3	0 (+4)	.4676456530e-3

Table A.25: Evolved Approximations for Refinement of Candidate Approximation 7 for sin(x).

EVOLVED LISP EXPRESSION			
ERROR	**COST**	**RUN**	**GENERATION**
1. (% (* (+ -1.00848 X) (* (- (* X 3.71792) (- (* (+ 1.66616 -3.72433) (* (* (* (% (* (+ (+ X -1.34144) (+ 1.66616 -3.72433)) (- X (* (- -3.37184 2.99127) 4.06827))) (- (* X X) 4.335)) (* (+ (- (% (* -3.24366 -0.254067) (- X 3.65749)) X) (* X (* (- -4.22849 -0.734123) X))) 4.06827)) (- -4.22849 -0.734123)) 4.06827)) X)) (+ X -1.34144))) (+ (+ (+ (- (- 0.534227 -4.4058) X) (* X (* X X))) (+ (+ (% (* 1.66616 (- (* X X) (* (% (+ (* X X) X) X) X))) (* (% (* -0.653554 X) (* (* 1.66616 X) X)) X)) (- -4.22849 -0.734123)) (+ (- (- -3.37184 2.99127) (- (* X X) (+ X -0.652638))) (% (- X X) (- X X))))) (+ (% 3.47682 X) (* (* (+ X (+ 1.66616 -3.72433)) (% X 1.66616)) (* (+ (* -0.653554 X) (+ 1.66616 -3.72433)) (- X X))))))			
0.0598397	38.9	33	47
2. (% (* (+ -1.00848 X) (* (- (* X 3.71792) (- (* (+ 1.66616 -3.72433) (* (* (* (% (* (+ (+ X -1.34144) (+ 1.66616 -3.72433)) (- X (* (- -3.37184 2.99127) 4.06827))) (- (* X X) 4.335)) (* (+ (- (% (* -3.24366 -0.254067) (- X 3.65749)) X) (* X (* (- -4.22849 -0.734123) X))) 4.06827)) (- -4.22849 -0.734123)) 4.06827)) X)) (+ X -1.34144))) (+ (+ (+ (- (- 0.534227 -4.4058) X) (* X (* X X))) (+ (+ (+ X (- X X)) (- -4.22849 -0.734123)) (+ (- (- -3.37184 2.99127) (- (* X X) (+ X -0.652638))) (% (- X X) (- X X))))) (+ (% 3.47682 X) (* (* (+ X (+ 1.66616 -3.72433)) (% X 1.66616)) (* (+ (* -0.653554 X) (+ 1.66616 -3.72433)) (- X X))))))			
0.0599625	27.9	33	48
3. (% (* (* (+ (+ -1.37349 -0.0889615) X) X) (* (+ (+ 3.02423 X) (- -3.37764 0.849483)) (% (* 0.017243 2.94671) (* 2.10318 4.83093)))) (- X (- -2.45262 (+ (+ 3.02423 X) (- -3.37764 0.849483)))))			
0.064597	4.7	10	2
4. (% (* (% (% (+ 3.57631 0.320292) -1.65487) -4.85687) (% 0.00167852 -2.90094)) (- 3.54793 (% (- 1.91092 -3.61141) X)))			
0.0673435	2.1	13	2
5. (% (+ X -0.991089) (% (% 3.76949 1.15009) (* -0.105136 0.0492874)))			
0.0706563	1.1	4	15
6. (* X (% 0.00167852 -2.90094))			
0.0721997	1	13	13
7. (* (* -0.105136 0.0492874) (% 0.136265 -2.1456))			
0.0732281	0	4	2

Table A.26: Maple Evaluation of Approximations for Refinement of Candidate Approximation 7 for sin(x).

SIMPLIFIED MAPLE EXPRESSION	COST	ERROR
1. [division by zero]	-	-
2. [division by zero]	-	-
3. (-1.4624515+x)*x*(-.6015476503e-2+.005000840892*x)/(2*x+1.249727)	5 (+3)	.002153461540
4. -.2805133023e-3/(3.54793-5.52233/x)	2 (+3)	INF
5. -.001581017185*x+.001566928741	1 (+3)	.002333705786

6. -.5786124498e-3*x	1 (+3)	.002408273886
7. .3290962388e-3	0 (+3)	.002416170399

Table A.27: Final Evolved Approximations for Refinement of Candidate Approximation 7 for sin(x).

EVOLVED APPROXIMATION	COST	ERROR
(3) (-1.4624515+x)*x*(-.6015476503e-2+.005000840892*x)/(2*x+1.249727)	5 (+3)	.002153461540
(5) -.001581017185*x+.001566928741	1 (+3)	.002333705786
(7) .3290962388e-3	0 (+3)	.002416170399

Table A.28: Evolved Approximations for Refinement of Candidate Approximation 8 for sin(x).

EVOLVED LISP EXPRESSION			
ERROR	**COST**	**RUN**	**GENERATION**
1. `(* (+ (- X X) (* (* 0.709708 (% (* (* (+ X -1.1681) -1.2566) X) (% -0.641957 X))) -1.92953)) (* (* (* (* (- (* (- X X) 0.0270089) X) X) 0.0270089) 0.0270089) (- X (+ -4.39817 0.199744))))`			
0.0530266	12.6	8	33
2. `(% (* (% (- X (* X X)) (+ (* -4.32371 3.2934) (% X (- X (* X X))))) X) (% 4.12778 (- (% X (+ X X)) (% (+ X (* X X)) (* (* 1.23981 -2.11264) (* 0.814997 2.9162))))))`			
0.0634081	10.6	2	42
3. `(% (* (% (- X (* X X)) (+ (* -4.32371 3.2934) (- X X))) X) (% (* (- X (+ X (% 2.82861 X))) (% 2.82861 X)) (- (% X (+ X X)) X)))`			
0.0658891	9.7	2	49
4. `(% (* (% (- X (* X X)) (+ (* -4.32371 3.2934) (* (% 1.66402 -3.22352) -0.876644))) X) (% 4.12778 (- (% X (+ X X)) (% (* (+ X -0.00808741) (* X (+ X -0.00808741))) (* (* 1.23981 -2.11264) (* 0.814997 2.9162))))))`			
0.0672362	9.5	2	36
5. `(% (* (% (- X (* X X)) (+ (* -4.32371 3.2934) (% 1.66402 -3.22352))) X) (% 4.12778 (- (% X (+ X X)) (% (+ X (* X X)) (* (* 1.23981 -2.11264) (* 0.814997 2.9162))))))`			
0.0709168	8.4	2	32
6. `(% (* (% (- X (* X X)) (+ (* -4.32371 3.2934) (- -2.43797 (+ X (- X X))))) X) (+ X (% 2.82861 X)))`			
0.0746054	5.6	2	12
7. `(% (* (% (- X (* X X)) (+ (* -4.32371 3.2934) (- -2.43797 (+ X X)))) X) (+ X (% 2.82861 X)))`			
0.0772754	5.5	2	11
8. `(% (% (% (- (* X X) X) (% (* -4.84191 -0.625477) (- (- X X) X))) (* 2.87072 1.65181)) (* -2.09281 2.09555))`			
0.0825382	5.3	18	3
9. `(* (% (* -1.27766 X) (+ (+ 1.32771 X) (% (+ -2.85607 -4.13785) 2.40959))) (* (+ (+ 3.02423 X) (- -3.37764 0.849483)) (% (* 0.017243 2.94671) (* 2.10318 (% -4.05393 -0.376446)))))`			

0.10046	4.4	10	6
10. (% -0.00747703 (% (+ -4.34385 (- -4.31425 (* X (* X -3.53755)))) X))			
0.141391	4.2	20	13
11. (* (- (* X X) X) 0.00686666)			
0.146896	2.1	18	3
12. (% 0.00167852 -2.90094)			
0.187177	0	13	2

Table A.29: Maple Evaluation of Approximations for Refinement of Candidate Approximation 8 for sin(x).

SIMPLIFIED MAPLE EXPRESSION	COST	ERROR
1. .001955402411*(x-1.1681)*x^4*(x+4.198426)	6 (+3)	.001728764493
2. .2422609732*(x-x^2)*x*(1/2+.1606371725*x^2+.1606371725*x)/(-14.23970651+x /(x-x^2))	10 (+3)	INF
3. .008777135782*(x-x^2)*x^3*(1/2-x)	7 (+3)	.002080725728
4. -.01757147646*(x-x^2)*x*(1/2+.1606371725*(x-.00808741)^2*x)	7 (+3)	.002171372526
5. -.01641788490*(x-x^2)*x*(1/2+.1606371725*x^2+.1606371725*x)	7 (+3)	.02306290529
6. (x-x^2)*x/((-16.67767651-x)*(x+2.82861/x))	7 (+3)	INF
7. (x-x^2)*x/((-16.67767651-2*x)*(x+2.82861/x))	7 (+3)	INF
8. .01587790457*(x^2-x)*x	4 (+3)	.002729466161
9. -1.27766*x*(-.002698529816+.002243366464*x)/(-1.574825286+x)	4 (+3)	.005078799840
10. -.00747703*x/(-8.65810+3.53755*x^2)	4 (+3)	.03303209138
11. .00686666*x^2-.00686666*x	3 (+3)	.004986344588
12. -.5786124498e-3	0 (+3)	.006343650815

Table A.30: Final Evolved Approximations for Refinement of Candidate Approximation 8 for sin(x).

EVOLVED APPROXIMATION	COST	ERROR
(1) .001955402411*(x-1.1681)*x^4*(x+4.198426)	6 (+3)	.001728764493
(8) .01587790457*(x^2-x)*x	4 (+3)	.002729466161
(11) .00686666*x^2-.00686666*x	3 (+3)	.004986344588
(12) -.5786124498e-3	0 (+3)	.006343650815

APPENDIX B: CODE DOCUMENTATION

The genetic programming system used for the experiments described in this thesis was originally conceived, prior to the initiation of work on this thesis, as a library for "generalized neural networks", intended as a new approach to function approximation and classification wherein the topology of a neural network could be modified by a genetic algorithm, and the neural network would be allowed to use a variety of mathematical operators as the activation functions for its neurons. For this reason, some relics of neural network terminology and orientation remain in the code and parameters for this system, now seen as a "functional genetic programming" system. This appendix gives the full set of command-line options available for the system along with an explanation of each of the corresponding parameters, as well as a brief description of each of the modules in the code.

FGP COMMAND-LINE OPTIONS

This section gives the command-line options available for the *FGP* executable. Each of the options can be used directly on the command line or as a line in a configuration file (specified using the *-File* option), with the exception of the *-ReportName* and *-NoReport* options, which can only be used from the command line.

AdjustedErrorPower [R] - Raise 1/(1+[error]) to the power of [R] when computing adjusted error.

AverageSampleError [On|Off] - Divide by number of samples when computing error (On) or do not perform this division (Off).

BiasedChoiceOfCrossoverPoint [On|Off] - Use fixed probability of choosing internal (vs. leaf) crossover point (On) or uniform probability for all points (Off). (The -PrInternalCrossover option is used to specify the probability of choosing an internal point).

CandidateSolutionErrorWeight [R] - Order candidate solutions according to weighted sum of error and cost with [R] as the error weight and 1-[R] as the weight for cost.

Crossover [Scramble|SingleSubtreeSwap|Meta] - Set crossover type.

ErrorMetric [RMS|Absolute|Max] - Set formula for error calculations.

ErrorMultiplier [R] - Multiply error by [R] before computing adjusted error.

ExcludeFN [Primitive function name] - Exclude specified function from function set.

File | *-f* [filename] - Use configuration file [filename].

FitnessLimit [R] - Run until best fitness is >= [R] (where [R] >= 0).

FitnessProportionateReproductionFraction [R] - Set fraction of population which will undergo fitness-proportionate reproduction each generation (for standard GA only).

FunctionSet "{"[Primitive function name]","[Primitive function name]. . ."}" - Specify set of available primitive functions, f.e. -FunctionSet {Product,Sum,Subtract,Div}.

GA ES([mu][,|+][lambda]) | Standard - Set genetic algorithm to Evolution Strategy with [mu] parents, [lambda] children, and "+" or "," recombination strategy, or to standard (Holland-style) GA.

GaussianWeightMutation [R] - Set Gaussian weight mutation rate (standard deviation) to [R].

GenerationLimit | -g [N] - Run up to [N] generations.

HitRange [R] - FGP output will be considered a "hit" if difference from correct value is < [R].

HitRatioLimit [R] - Run until hit ratio for best FGP is >= [R]. Note that [R]>1 can be used for no limit.

HTMLSummaryHeaderFooter [On|Off] - Turn HTML summary header/footer generation on or off ('Off' causes only <TABLE> to be generated).

IncludeFN [Primitive function name] - Include specified function in function set.

MaxNodesPerOutput [N] - Set maximum number of nodes per output.

MaxRandTreeDepth [N] - Set maximum (zero-based) depth of randomly generated trees.

MaxTreeDepth [N] - Set maximum (zero-based) depth for all trees.

NoRandomize - Do not seed random number generator using clock value.

NoReport - Turn off report generation.

NumRuns [N] - Execute [N] runs.

OccamsRazor [On|Off] - Turn "Occam's Razor" (incorporation of expression cost into fitness function) on or off.

PopulationSize [N] - Set population size (for standard GA only).

PrFunctionMutation [R] - Set probability of primitive function mutation to [R].

PrInternalCrossoverPoint [R] - Set probability of choosing internal (vs. leaf) crossover point to [R].

PrSubtreeRecursion [R] - Set subtree recursion probability to [R] (for breeding trees via "scramble" crossover operator).

PrSubtreeMutation [R] - Set probability of subtree mutation to [R].

Randomize - Seed random number generator using clock value.

RandomNumericTerminal [On|Off] - Turn use of random numeric terminal on or off.

RandSeed [N] - Set random number generator seed value to [N].

RandTreeDefaultArity [On|Off] - Generate random initial trees using default arity for each function (On) or random arity (Off).

RandTreeUniformDepth [On|Off] - Create initial random trees with each branch reaching same depth (On) or with branches of varying depth within each tree (Off).

RandWeightInit [On|Off] - Turn random weight initialization on or off.

ReportName [filename without extension] - Set report filename prefix ([report name].html, [report name].gnuplot and [report name]_MapleCandidateSolutions.txt will be generated).

SurvivorSelection [BestN|Roulette] - Set survivor selection scheme (applicable to Evolution Strategy GA only).

TrainingSet | *-t* [filename] - Use [filename] as training set.

VirtualTrainingSetSize [N] - Use virtual training set of size [N]

WeightCombination [Averaged|Discrete] - Set weight combination type.

MODULE DESCRIPTIONS

The following are descriptions of each module of the author's GP system. The modules are divided into core library files, module tests, the main driver program, and utilities.

Core Library Files

CConvexFitnessHull.cpp - Class which keeps track of which points in the (error,cost) plane are on the convex hull for the population history.

CFunctionalGP.cpp - Class implementing functional genetic programs (functions of one or more variables which can be randomly generated, bred, etc.).

CFunctionalGP_Environment.cpp - Class implementing functional genetic program "environment", which is basically a repository of global variables shared by all FGPs in an evolving population.

CFunctionalGP_Node.cpp - Class for an individual node in an FGP tree.

CFunctionalGP_Population.cpp - Class derived from CPopulation (in GA.cpp) with special overloaded function to maintain convex fitness hull.

CFunctionSet.cpp - Class implementing set of primitive functions from which FGPs are constructed.

CLinkedList.hpp / *CLinkedListT.hpp* - Linked list template class.

CSampleSet.cpp - Class for loading and using training sets.

GA.cpp - Contains "GA framework" within which FGPs are evolved, with base classes for individuals (CIndividual), populations (CPopulation), environments (CEnvironment), and genetic algorithms (CGeneticAlgorithm).

Rand.cpp - Routines for generating random numbers.

SelectionSchemes.cpp - Functions to select individuals according to various selection schemes.

Util.cpp - Various utility functions.

Module Tests

CConvexFitnessHullTest.cpp - test for CConvexFitnessHull.cpp.

CFunctionalGP_EnvironmentTest.cpp - test for CFunctionalGP_Environment.cpp.

CFunctionalGP_NodeTest.cpp - test for CFunctionalGP_Node.cpp.

CFunctionalGPTest.cpp - test for CFunctionalGP.cpp.

CFunctionSetTest.cpp - test for CFunctionSet.cpp.

CLinkedListTest.cpp - test for CLinkedList template class.

CSampleSetTest.cpp - test for CSampleSet.cpp.

GATest.cpp - test for GA.cpp.

RandTest.cpp - test for Rand.cpp.

SelectionSchemesTest.cpp - test for SelectionSchemes.cpp.

UtilTest.cpp - test for Util.cpp.

TestUtils.cpp - Common utility functions shared by various module tests.

Main driver program

CFGP_CommandLine.cpp - Class to process command-line options.

Main.cpp - Main driver program which accepts command-line options, configures FGP appropriately, executes runs, and generates HTML and Gnuplot-compatible summary data.

Utilities

ClosedFormFinder.cpp - Obsolete program which attempts to co-evolve a series to be approximated and the approximating function itself.

FnTimer.cpp - Program to time various primitive functions and assign costs accordingly.

PrintAsLISPExpression.cpp - Takes an FGP expression as input and outputs a LISP expression.

PrintAsMaple.cpp - Takes an FGP expression as input and outputs a Maple expression.

RandExpression.cpp - Generates a random FGP expression.

Simplify.cpp - Takes an FGP expression as input and outputs it in simplified form (currently just evaluates constant expressions).

TGen.cpp - Generates various training sets.

REFERENCES

D. Andre, F. H. Bennett III, and J. R. Koza (1996). Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (eds.) *Genetic Programming 1996: Proceedings of the First Annual Conference*. Cambridge, MA: The MIT Press.

D. Andre and J. R. Koza (1996). Parallel genetic programming: A scalable implementation using the transputer network architecture. In P. J. Angeline and K. E. Kinnear, Jr. (eds.), *Advances in Genetic Programming 2*, 317-338. Cambridge, MA: MIT Press.

G. A Baker (1975). *Essentials of Padé Approximants*. New York: Academic Press.

C. M. Bender and S. A. Orszag (1978). *Advanced Mathematical Methods for Scientists and Engineers*. New York: McGraw-Hill.

C. M. Bishop (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.

T. Blickle and L. Thiele (1995). A comparison of selection schemes used in genetic algorithms. TIK-Report 11, TIK Institut fur Technische Informatik und Kommunikationsnetze, Computer Engineering and Networks Laboratory, ETH, Swiss Federal Institute of Technology.

A. E. Bryson and Y.-C. Ho (1969). *Applied Optimal Control and Estimation*. Blaisdell.

R. L. Burden and J. D. Faires (1997). *Numerical Analysis*. Pacific Grove, CA: Brooks/Cole Publishing Company.

K. Chellapilla (1997). Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation* **1**(3):209-216.

W. Comisky, J. Yu, and J. R. Koza (2000). Automatic synthesis of a wire antenna using genetic programming. In *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*, 179-186.

G. W. Cottrell (1990). Extracting features from faces using compression networks: Face, identity, emotion and gender recognition using holons. In *Connection Models: Proceedings of the 1990 Summer School*. San Mateo, CA: Morgan Kaufmann.

L. M. Desjarlais, M. Akbarzadeh-T and C. W. Wright (1999). Control system optimization using genetic algorithms within the SoftLab tooklit. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, 2:1774. San Mateo, CA: Morgan Kaufmann.

G. Diplock (1996). The application of evolutionary computing techniques to spatial interaction modelling. PhD thesis, Leeds University, UK

C. Eldershaw and S. Cameron (1999). Real-world applications: Motion planning using GAs. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, 2:1776. San Mateo, CA: Morgan Kaufmann.

L. Eulero (1734). De progressionibus harmonicus observationes. In *Comentarii academiæ scientarum imperialis Petropolitanæ* **7**(1734):150-161.

R. Feldt, M. O'Neill, C. Ryan, P. Nordin, and W. Langdon. (2000). GP-Beagle: A benchmarking problem repository for the genetic programming community. In *Late Breaking Papers at the GECCO'2000 Conference*. San Mateo, CA: Morgan Kaufmann.

D. E. Goldberg (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

G. H. Gonnet (1984). *Handbook of Algorithms and Data Structures*. London: Addison-Wesley.

M. Gregory. (1998). Genetic algorithm optimisation of distributed database queries. In *Proceedings of the 1998 IEEE World Congress on Computational Intelligence*, 271-276. IEEE Press.

K. M. Heal, M. L. Hansen, and K. M. Rickard (1998). *Maple V Learning Guide*. New York: Springer-Verlag.

J. H. Holland (1975). *Adaptation in Natural and Artificial Systems*. Cambridge, MA: The MIT Press

D. Howard and S. C. Roberts (1999). A staged genetic programming strategy for image analysis. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, 2:1047-1052. San Mateo, CA: Morgan Kaufmann.

C. Jacob (1996). Evolving evolution programs: Genetic programming and L-systems. In J. R. Koza, D. E. Goldberg, D. B.Fogel, and R. L. Riolo (eds.), *Genetic Programming 1996: Proceedings of the First Annual Conference*, 107-115. Cambridge, MA: The MIT Press.

M. A. Keane, J. R. Koza, and J. P. Rice (1993). Finding an impulse response function using genetic programming. In *Proceedings of the 1993 American Control Conference*, 3:2345-2350.

J. R. Koza (1989). Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann. 1:768-774.

J. R. Koza (1990a). Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Stanford University Computer Science Department technical report STAN-CS-90-1314.

J. R. Koza (1990b). A genetic approach to econometric modeling. Presented at *Sixth World Congress of the Econometric Society*, Barcelona, Spain.

J. R. Koza (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press.

J. R. Koza (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press.

J. R. Koza, D. Andre, F. H Bennett III, and M. Keane. (1999). *Genetic Programming 3: Darwinian Invention and Problem Solving*. San Mateo, CA: Morgan Kaufman.

J. R. Koza, M. A. Keane, F. H. Bennett III, J. Yu, W. Mydlowec, and O. Stiffelman (1999). Automatic creation of both the topology and parameters for a robust controller by means of genetic programming. In *Proceedings of the 1999 IEEE International Symposium on Intelligent Control, Intelligent Systems, and Semiotics*, 344-352. Piscataway, NJ: IEEE Press.

W. Lam, M. L. Wong, K. S. Leung, and P. S. Ngan. (1998). Discovering probabilistic knowledge from databases using evolutionary computation and minimum description length principle. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, (eds.), *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 786-794. San Mateo, CA: Morgan Kaufmann.

S. Luke and L. Spector (1997). A comparison of crossover and mutation in genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (eds.), *Genetic Programming 1997: Proceedings of the Second Annual Conference*, 240-248. San Mateo, CA: Morgan Kaufmann.

T. M. Mitchell. (1997). *Machine Learning*. New York: McGraw-Hill.

R. E. Moustafa, K. A. De Jong, and E. J. Wegman (1999). Using genetic algorithms for adaptive function approximation and mesh generation. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, 1:798. San Mateo, CA: Morgan Kaufmann.

R. Müller, H.-H. Hemberger, and K. Baier (1997). Engine control using neural networks: A new method in engine management systems. In *MECCANICA*, **32**(5): 423-430.

P. Nordin (1994). A compiling genetic programming system that directly manipulates the machine code. In *Advances in Genetic Programming*, 311-331. Cambridge, MA: The MIT Press

P. Nordin (1997). *Evolutionary Program Induction of Binary Machine Code and its Applications*. PhD thesis, der Universitat Dortmund am Fachereich Informatik.

U. M. O'Reilly and F. Oppacher (1994). The troubling aspects of a building block hypothesis for genetic programming. Santa Fe Institute Working Paper 94-02-001.

D. A. Pomerleau (1993). Knowledge-based training of artificial neural networks for autonomous robot driving. In *Robot Learning*, 19-43. Boston: Kluwer Academic Publishers.

C. Ryan, J. J. Collins, and M. O'Neill (1998). Grammatical evolution: Evolving programs for an arbitrary language. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty (eds.), *Proceedings of the First European Workshop on Genetic Programming*, 1391:83-95. New York: Springer-Verlag.

T. Soule and J. A. Foster. (1998). Effects of code growth and parsimony pressure on populations in genetic programming. In *Evolutionary Computation*, **6**(4):293-309.

M. Wall (2000). GAlib: Matthew's Genetic Algorithms Library. http://lancet.mit.edu/ga/

P. Walsh and C. Ryan (1996). Paragen: A novel technique for the autoparallelisation of sequential programs using genetic programming. In J.R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (eds.) *Genetic Programming 1996: Proceedings of the First Annual Conference*. Cambridge, MA: The MIT Press.

T. Williams, C. Kelley (1999). Gnuplot: An Interactive Plotting Program. http://theochem.ki.ku.dk/on_line_docs/gnuplot/gnuplot_1.html

B. Zhang and H. Mühlenbein. (1995). Balancing accuracy and parsimony in genetic programming. In *Evolutionary Computation*, **3**(1):17-38.