A Baseline Symbolic Regression Algorithm

Michael F. Korns

Korns Associates, 98 Perea Street, Makati 1229, Manila Philippines
mkorns@korns.com.

Abstract.

Recent advances in symbolic regression (SR) have promoted the field into the early stages of commercial exploitation. This is the expected maturation history for an academic field which is progressing rapidly. The original published symbolic regression algorithms in (Koza 1992) have long since been replaced by techniques such as pareto front, age layered population structures, and even age pareto front optimization. The lack of specific techniques for optimizing embedded real numbers, in the original algorithms, has been replaced with sophisticated techniques for optimizing embedded constants. Symbolic regression is coming of age as a technology.

As the discipline of Symbolic Regression (SR) has matured, the first commercial SR packages have appeared. There is at least one commercial package on the market for several years *http://www.rmltech.com/*. There is now at least one well documented commercial symbolic regression package available for Mathmatica *www.evolved-analytics.com*. There is at least one very well done open source symbolic regression package available for free download *http://ccsl.mae.cornell.edu/eureqa*. Yet, even as the sophistication of commercial SR packages increases, there have been glaring issues with SR accuracy even on simple problems (Korns 2011). The depth and breadth of SR adoption in industry and academia will be greatly affected by the demonstrable accuracy of available SR algorithms and tools.

In this chapter we develop a complete public domain algorithm for modern symbolic regression which is reasonably competitive with current commercial SR packages, and calibrate its accuracy on a set of of previously published sample problems. This algorithm is designed as a baseline for further public domain research on SR algorithm simplicity and accuracy. No claim is made placing this baseline algorithm on a par with commercial packages - especially as the commercial offerings can be expected to relentlessly improve in the future. However this baseline is a great improvement over the original published algorithms, and is an attempt to consolidate the latest published research into a simplified baseline algorithm of similar speed and accuracy.

The baseline algorithm presented herein is called Age Weighted Pareto Opimization. It is an almalgamation of recent published techniques in pareto front optimization (Kotanchek 2008), age layered population structures (Hornby 2006), age fitness pareto optimization (Schmidt 2010), and specialized embedded abstract constant optimization (Korns 2010). The complete pseudo code for the baseline algorithm is presented in this paper. It is developed step by step as enhancements to the original published SR algorithm (Koza 1992) with justifications for each enhancement. Before-after speed and accuracy comparisons are made for each enhancement on a series of previously published sample problems.

Key words: Abstract Expression Grammars, Grammar Template Genetic Programming, Genetic Algorithms, Particle Swarm, Symbolic Regression.

1 Introduction

The discipline of Symbolic Regression (SR) has matured significantly in the last few years. There is at least one commercial package on the market for several years *http://www.rmltech.com/*. There is now at least one well documented commercial symbolic regression package available for Mathematica *www.evolved-analytics.com*. There is at least one very well done open source symbolic regression package available for free download *http://ccsl.mae.cornell.edu/eureqa*. In addition to our own ARC system (Korns 2010), currently used internally for massive (million row) financial data nonlinear regressions, there are a number of other mature symbolic regression packages currently used in industry including (Smits 2010) and (Castillo 2010). Plus there is an interesting work in progress by (McConaghy 2009).

Yet, despite the increasing sophistication of commercial SR packages, there have been serious issues with SR accuracy even on simple problems (Korns 2011). Clearly the perception of SR as a *must use* tool for important problems or as an *interesting heurism* for shedding light on some problems, will be greatly affected by the demonstrable accuracy of available SR algorithms and tools. The depth and breadth of SR adoption in industry and academia will be greatest if a very high level of accuracy can be demonstrated for SR algorithms.

In this chapter we develop a simple, easy to implement, public domain algorithm for modern symbolic regression which is reasonably competitive with current commercial SR packages. This algorithm is meant to be a baseline for further public domain research on provable SR algorithm accuracy. It is called Constant Swarm with Operator Weighted Pruning, and is inspired by recent published techniques in pareto front optimization (Kotanchek 2008), age layered population structures (Hornby 2006), age fitness pareto optimization (Schmidt 2010), and specialized embedded abstract constant optimization (Korns 2010). The complete pseudo code for the baseline algorithm is presented in this paper. It is developed as a series of step by step as enhancements to a simple brute force GP algorithm with justifications for each enhancement. Before-after speed and accuracy comparisons are made on a series of previously published sample problems.

Of course, as commercial packages improve, many market-competitive features and techniques will be developed outside the public domain. This is a natural process within the development of a promising new technology. No claim is made placing this baseline algorithm on a par with commercial packages - especially as the commercial offerings can be expected to relentlessly improve in the future and not necessarily within the public domain. This baseline is an attempt to consolidate the latest published research into a simplified baseline algorithm for further research on SR speed and accuracy.

Before continuing with the details of our baseline algorithm, we proceed with a basic introduction to general nonlinear regression. Nonlinear regression is the mathematical problem which Symbolic Regression aspires to solve. The

canonical generalization of nonlinear regression is the class of Generalized Linear Models (GLMs) as described in (Nelder 1972). A GLM is a linear combination of **I** basis functions B_i; i = 1,2, I, a dependent variable y, and an independent data point with M features x = <x_1, x_2, x_3, x_m>: such that

- (*E1*) y = γ(x) = c_0 + $\Sigma c_i B_i$(x) + **err**

As a broad generalization, GLMs can represent any possible nonlinear formula. However the format of the GLM makes it amenable to existing linear regression theory and tools since the GLM model is linear on each of the basis functions B_i. For a given vector of dependent variables, Y, and a vector of independent data points, X, symbolic regression will search for a set of basis functions and coefficients which minimize **err**. In (Koza 1992) the basis functions selected by symbolic regression will be formulas as in the following examples:

- (*E2*) B_1 = x3
- (*E3*) B_2 = x1+x4
- (*E4*) B_3 = sqrt(x2)/tan(x5/4.56)
- (*E5*) B_4 = tanh(cos(x2*.2)*cube(x5+abs(x1)))

If we are minimizing the least squared error, LSE, once a suitable set of basis functions B have been selected, we can discover the proper set of coefficients C deterministically using standard univariate or multivariate regression. The value of the GLM model is that one can use standard regression techniques and theory. Viewing the problem in this fashion, we gain an important insight. Symbolic regression does not add anything to the standard techniques of regression. The value added by symbolic regression lies in its abilities as a search technique: how quickly and how accurately can SR find an optimal set of basis functions B. The immense size of the search space provides ample need for improved search techniques. In basic Koza-style tree-based Genetic Programming (Koza 1992) the genome and the individual are the same Lisp s-expression which is usually illustrated as a tree. Of course the tree-view of an s-expression is a visual aid, since a Lisp s-expression is normally a list which is a special Lisp data structure. Without altering or restricting basic tree-based GP in any way, we can view the individuals not as trees but instead as s-expressions such as this depth 2 binary tree s-exp: *(/ (+ x2 3.45) (* x0 x2))*, or this depth 2 irregular tree s-exp: *(/ (+ x4 3.45) 2.0)*.

In basic GP, applied to symbolic regression, the non-terminal nodes are all operators (implemented as Lisp function calls), and the terminal nodes are always either real number constants or features. The maximum depth of a GP individual is limited by the available computational resources; but, it is standard practice to limit the maximum depth of a GP individual to some manageable limit at the start of a symbolic regression run.

Given any selected maximum depth k, it is an easy process to construct a maximal binary tree s-expression U_k, which can be produced by the GP system without violating the selected maximum depth limit. As long as we are reminded that each f represents a function node while each t represents a terminal node, the construction algorithm is simple and recursive as follows.

- (U_0): t
- (U_1): (f t t)
- (U_2): (f (f t t) (f t t))
- (U_3): (f (f (f t t) (f t t)) (f (f t t) (f t t)))
- (U_k): (f U_{k-1} U_{k-1})

The basic GP symbolic regression system (Koza 1992) contains a set of functions F, and a set of terminals T. If we let t ∈ T, and f ∈ F ∪ ξ, where $\xi(a,b) = \xi(a) = a$, then any basis function produced by the basic GP system will be represented by at least one element of U_k. In fact, U_k is isomorphic to the set of all possible basis functions generated by the basic GP system *to a depth of k*.

Given this formalism of the search space, it is easy to compute the size of the search space, and it is easy to see that the search space is huge even for rather simple basis functions. For our use in this chapter the function set will be the following functions: F = **(+ - * / abs sqrt square cube quart cos sin tan tanh log exp max min** ξ**)**. The terminal set is the features **x0** thru **xm** and the real constant **c**, which we shall consider to be 2^{264} in size. Where ‖F‖ = 18, M=20, and k=0 , the search space is S0 = M+2^{264} = 20+2^{264} = 1.84x10^{19}. Where k=1, the search space is S1 = ‖F‖*S0*S0 = 6.12x10^{39}. Where k=2, the search space grows to S2 = ‖F‖*S1*S1 = 6.75x10^{80}. For k=3, the search space grows to S3 = ‖F‖*S2*S2 = 8.2x10^{162}. Finally if we allow three basis functions B=3, then the final size of the search space is S3*S3*S3 = 5.53x10^{487}.

Clearly even for three simple basis functions, with only 20 features and very limited depth, the size of the search space is already very large; and, the presence of real constants accounts for a significant portion of that size. For instance, without real constants, S0 = 20, S3 = 1.054x10^{19}, and with B=3 the final size of the search space is 1.054x10^{57}.

It is our contention that since real constants account for such a significant portion of the search space, symbolic regression would benefit from special constant evolutionary operations. Since basic GP does not offer such operations, we investigate the enhancement of symbolic regression with swarm intelligence algorithms specifically designed to evolve real constants.

1.1 Example Test Problems

In this chapter we list the example test problems which we will address. All of these test problems are no more than three grammar nodes deep (Note:

in problem P10, *quart(x)* = x^4). All test problems reference no more than five input features. Some are easily solved with current Symbolic Regression techniques. Others are not so easily solved.

- (*P1*): y = 1.57 + (24.3*x3)
- (*P2*): y = 0.23 + (14.2*((x3+x1)/(3.0*x4)))
- (*P3*): y = -5.41 + (4.9*(((x3-x0)+(x1/x4))/(3*x4)))
- (*P4*): y = -2.3 + (0.13*sin(x2))
- (*P5*): y = 3.0 + (2.13*log(x4))
- (*P6*): y = 1.3 + (0.13*sqrt(x0))
- (*P7*): y = 213.80940889 - (213.80940889*exp(-0.54723748542*x0))
- (*P8*): y = 6.87 + (11*sqrt(7.23*x0*x3*x4))
- (*P9*): y = ((sqrt(x0)/log(x1))*(exp(x2)/square(x3)))
- (*P10*): y = 0.81 + (24.3*(((2.0*x1)+(3.0*square(x2)))/((4.0*cube(x3))+(5.0*quart(x4)))))
- (*P11*): y = 6.87 + (11*cos(7.23*x0*x0*x0))
- (*P12*): y = 2.0 - (2.1*(cos(9.8*x0)*sin(1.3*x4)))
- (*P13*): y = 32.0 - (3.0*((tan(x0)/tan(x1))*(tan(x2)/tan(x3))))
- (*P14*): y = 22.0 - (4.2*((cos(x0)-tan(x1))*(tanh(x2)/sin(x3))))
- (*P15*): y = 12.0 - (6.0*((tan(x0)/exp(x1))*(log(x2)-tan(x3))))

As a discipline, our goal is to demonstrate that *all* of the 10^{162} possible test problems can be solved after a reasonable number of individuals have been evaluated. This is especially true since we have limited these 10^{162} possible test problems to target functions which are univariate, reference no more than five input features, and which are no more than three grammar nodes deep. On the hopeful side, if the Symbolic Regression community can achieve a demonstration of absolute accuracy, then the same rigorous statistical inferences can follow a Symbolic Regression as now follow a Linear Regression, which would be a significant advancement in scientific technique.

For the sample test problems, we will use only statistical best practices out-of-sample testing methodology. A matrix of independent variables will be filled with random numbers. Then the model will be applied to produce the dependent variable. These steps will create the training data. A symbolic regression will be run on the training data to produce a champion estimator. Next a matrix of independent variables will be filled with random numbers. Then the model will be applied to produce the dependent variable. These steps will create the testing data. The estimator will be regressed against the testing data producing the final LSE and R-Square scores for comparison.

2 Brute Force Elitist GP Symbolic Regression

In basic GP symbolic regression (Koza 1992), a Lisp s-expression is manipulated via the evolutionary techniques of mutation and crossover to produce a new s-expression which can be tested, as a basis function candidate in a GLM.

Basis function candidates that produce better fitting GLMs are promoted. Mutation inserts a random s-expression in a random location in the starting s-expression. For example, mutating s-expression (E4) we obtain s-expression (E4.1) wherein the sub expression "tan" has been randomly replaced with the sub expression "**cube**". Similarly, mutating s-expression (E5) we obtain s-expression (E5.1) wherein the sub expression "cos(x2*.2)" has been randomly replaced with the sub expression "**abs(x2+x5)**".

- (*E4*) B_3 = sqrt(x2)/tan(x5/4.56)
- (*E4.1*) B_5 = cos(x2)/**cube**(x5/4.56)
- (*E5*) B_4 = tanh(cos(x2*.2)*cube(x5+abs(x1)))
- (*E5.1*) B_6 = tanh(**abs(x2+ x5)***cube(x5+abs(x1)))

Crossover combines portions of a mother s-expression and a father s-expression to produce a child s-expression. Crossover inserts a randomly selected sub expression from the father into a randomly selected location in the mother. For example, crossing s-expression (E5) with s-expression (E4) we obtain child s-expression (E5.2) wherein the sub expression "cos(x2*.2)" has been randomly replaced with the sub expression "**tan(x5/4.56)**". Similarly, again crossing s-expression (E5) with s-expression (E4) we obtain another child s-expression (E5.3) wherein the sub expression "x5+abs(x1)" has been randomly replaced with the sub expression "**sqrt(x2)**".

- (*E4*) B_3 = sqrt(x2)/tan(x5/4.56)
- (*E5*) B_4 = tanh(cos(x2*.2)*cube(x5+abs(x1)))
- (*E5.2*) B_7 = tanh(**tan(x5/4.56)***cube(x5+abs(x1)))
- (*E5.3*) B_8 = tanh(cos(x2*.2)*cube(**sqrt(x2)**))

These mutation and crossover operations are the main tools of basic GP, which functions as described in Algorithm A1, randomly creating a population of candidate basis functions, mutating and crossing over those basis functions repeatedly while consistently promoting the most fit basis functions. The winners being the collection of basis functions which receive the most favorable least square error in a GLM with standard regression techniques.

Our core baseline is the brute force elitist GP algorithm outlined in Algorithm (A1). It has been selected, as our baseline, because it is very simple, has very few parameters, and is easy to implement. It is intended for use as a public domain baseline SR algorithm for further research in SR accuracy. The inputs are a vector of N training points, X, a vector of N dependent variables, Y, and the number of generations to train, G. Each point in X is a member of RM = $<x_1, x_2, x_3, x_m>$. The fitness score is the root mean squared error divided by the standard deviation of Y, NLSE.

```
Algorithm A1: Brute force elitist GP Symbolic Regression
1   Function: symbolicRegression(X,Y,G)
2   Input: X // N vector of independent M-featured training points
3   Input: Y // N vector of dependent variables
4   Input: G // Number of generations to train
5   Output: champ // Champion s-expression individual
6   Parameters: maxPopSize maxIslandCount
    Summary: Brute force elitist GP searches for a champion
             s-expression by randomly growing and scoring a
             large number of candidate s-expressions, then
             iteratively creating and scoring new candidate
             s-expressions via mutation and crossover. After
             each iteration, the population of candidate
             s-expressions is truncated to those with the best
             fitness score. After the final iteration, the
             champion is the s-expression with the best
             fitness score.

7   Function: mutateSExp(me)
    Summary: mutateSExp randomly alters an input s-expression
             by replacing a randomly selected sub expression
             with a new randomly grown sub expression.
8     me = copy(me)
9     set L = number of nodes in me // me is a list of Lisp Pairs
10    set n = random integer between 0 and L
11    set me[n] = s  // Replaces nth node with s
12    return me
13  end fun mutateSExp

14  Function: crossoverSExp(mom,dad)
    Summary: crossoverSExp randomly alters a mom input
             s-expression by replacing a randomly selected
             sub expression in mom with a randomly selected
             sub expression from dad.
15    dad = copy(dad)
16    mom = copy(mom)
17    set Ld = number of nodes in dad // dad is a list of Pairs
18    set Lm = number of nodes in mom // mom is a list of Pairs
19    set n = random integer between 0 and Lm
20    set m = random integer between 0 and Ld
21    set mom[n] = dad[m]  // Replaces nth node with mth node
22    return mom
23  end fun crossoverSExp

24  Function: optimizeConstants(me)
```

```
    Summary: Optimize any embedded constants in me
25    return mutateSExp(me)
26  end fun optimizeConstants

27  Function: insertLambda(population,lambda)
    Summary: inserts the specified lambda into the specified
             population unordered
28    if (population.length <= 0) then set population = new vector of length 0
29    set population.last = lambda
30    return lambda
31  end fun insertLambda

32  Function: populationPruning(inPopulation,outPopulation,islands)
    Summary: Copies the input population into the output population.
             Adds random individuals to the output population.
             Sorts the output population in ascending order of
             fitness score. Truncates the output population to
             the maximum population size. Always organizes the
             population into a single island.
33    set outPopulation = new vector of length 0
34    set maxIslandCount = 1
35    if (inPopulation.length <= 0) then
36      // Initialize with random individuals
37      set K = 5*maxPopSize
38      for k from 0 until K do // Initialize population
39        set lambda = generate random individual
40        do evaluate lambda and set its fitness score
41        set lambda.island = 0
42        insertLambda(outPopulation,lambda)
43      end for k
44    else
45      // Copy and add a few more random individuals
46      set outPopulation = copy(inPopulation)
47      set K = maxPopSize/10
48      for k from 0 until K do // Initialize population
49        set lambda = generate random individual
50        do evaluate lambda and set its fitness score
51        set lambda.island = 0
52        insertLambda(outPopulation,lambda)
53      end for k
54    end if
55    sort outPopulation in ascending order by fitness score
56    truncate outPopulation to length of maxPopSize
57    set inPopulation = outPopulation
58    // Alway return island index with only one island
```

```
  set islands = new vector of length 1
  set islands[0] = inPopulation
  champ = inPopulation.first
  return champ
end fun populationPruning

Main:
set maxPopSize = 5000
set maxIslandCount = 1
if (G <= 0) then populationPruning(inPopulation,outPopulation,islands)
set P = inPopulation.length
set champ = inPopulation.first
for g from 0 until G do // Main evolution loop
  set P = inPopulation.length
  // Everyone gets mutated and crossed over
  for p from 0 until P do
    set lambda = optimizeConstants(inPopulation[p])
    insertLambda(outPopulation,lambda)
    set lambda = mutateSExp(inPopulation[p])
    insertLambda(outPopulation,lambda)
    set dad = inPopulation[p]
    // Cross over partner must be from the same island
    set K = dad.island
    set i = random integer between 0 and islands[K].length
    set mom = island[K][i]
    set lambda = crossoverSExp(dad,mom)
    insertLambda(outPopulation,lambda)
  end for p
  populationPruning(inPopulation,outPopulation,islands)
 set champ = inPopulation.first
end for g
return champ
end fun symbolicRegression
```

As a base line for symbolic regression, our brute force elitist algorithm leaves us far from our accuracy goal. In fact we quickly demonstrate that there are large sets of test problems which are intractable with our baseline algorithm.

The results of our brute force elitist algorithm on the test problems are shown in Table 1.

Table 1. Results for Brute Force Elitist GP Symbolic Regression

Test	*WFFs*	*Train-NLSE*	*Train-TCE*	*Test-NLSE*	*Test-TCE*
P01	14K	0.00	0.00	0.00	0.00
P02	96K	0.00	0.00	0.00	0.00
P03	74M	0.00	0.00	0.00	0.00
P04	34K	0.00	0.00	0.00	0.00
P05	94K	0.00	0.00	0.00	0.00
P06	12K	0.00	0.00	0.00	0.00
P07	82M	0.00	0.00	0.00	0.00
P08	1M	0.00	0.00	0.00	0.00
P09	71G	0.01	0.00	0.97	0.37
P10	85G	0.83	0.39	1.00	0.49
P11	81G	0.99	0.46	1.00	0.49
P12	89G	0.99	0.48	1.04	0.50
P13	85G	0.81	0.30	1.00	0.93
P14	83G	0.53	0.17	1.53	0.47
P15	1M	0.00	0.00	0.00	0.00

(*Note: the number of individuals evaluated before finding a solution is listed in the Well Formed Formulas (WFFs) column*)

3 Premature Convergence and Complexity Pruning

Our brute force elitist symbolic regression program, suffers from the same problem of bloat which plagues other unconstrained GP systems (Smits 2005). The population is increasingly dominated by basis functions which have grown impossibly long. Furthermore, the population quickly becomes dominated by large numbers of basis functions clustering in the most promising areas preventing the exploration of other potentially promising areas. This creates a population malaise known as premature convergence.

The problems of bloat and premature convergence in GP systems have been discussed heavily in the recent literature. A number of techniques have been developed to combat bloat, the most notable of which is pareto front optimization (Kotanchek 2008). Additionally a number of techniques have been developed to combat premature convergence, the most notable of which are age layered population structures (Hornby 2006), and age fitness pareto optimization (Schmidt 2010). Each of these techniques attempt to keep the population evenly distributed over a wide variety of different possible solutions in both length and complexity.

To address this issue in our simple baseline algorithm, we can apply the concept of a pareto front forcing a multi objective optimization between fitness and complexity via simple complexity pruning. Our symbolic regression system will be enhanced to prune the population such that basis functions with similar operator complexity are clustered together in islands. Each complexity island will be pruned when it gets too large. This pruning activity assures

that the computational resources will be spread across individuals in many different complexity islands. The population will never be allowed to cluster in one or two areas preventing the exploration of other promising areas. We call this simple but effective technique Operator Weighted Pruning.

The changes and enhancements for operator weighted population pruning are shown in Algorithm (A2).

```
Algorithm A2: populationPruning
1   Function: populationPruning(inPopulation,outPopulation,islands)
2   Parameters: maxPopSize maxIslandCount maxIslandSize
3   Replaces: A1.populationPruning lines 32 thru 63
4   Summary: Copies the input population into the output population.
             Adds random individuals to the output population.
             Sorts the output population in ascending order of
             fitness score. Computes the weighted complexity score of
             each individual and assigns each individual to a complexity
             island. Eliminates all non-dominant individuals in each
             complexity island. Truncates the output population to
             the maximum population size. Always organizes the
             population into multiple separate islands by complexity.

5   Function: weightedComplexity(me,depth)
6   Summary: Returns a complexity score similar to the pareto front length
             algorithm but with each function operator having a different
             weight. For A = (+ (cos x2) x1), B = (- (log x2) x4), and
             C = (+ 22.3 (abs x0)) these all have the same pareto front
             length of 5. However with operator weighted pareto complexity
             the complexities all change because the operators are different
             where the weighted lengths are A = 13, B = 11, and C = 5.
             This tends to group arithmetic operators with other arithmetic
             operators and trancendential operators with other trancendential
             operators etc.
7     if (me is a list) then
8       // me is a function call with the operator first
9       set operator = me[0]
10      set weight = operator.weight
11      set complexity = 0
12      set N = me.length
13      for n from 1 until N do
14        set complexity = complexity + weightedComplexity(me[n],weight*(depth+1))
15      end for n
16    else
17      // me is a single element
18        set complexity = depth + 1
19    end if
```

```
20    return complexity
21  end fun weightedComplexity

22    set maxPopSize = 5000
23    set maxIslandCount = 500
24    set maxIslandSize = integer(maxPopSize / maxIslandCount)
25    set outPopulation = new vector of length 0
26    if (inPopulation.length <= 0) then
27      // Initialize with random individuals
28      set K = 5*maxPopSize
29      for k from 0 until K do // Initialize population
30        set lambda = generate random individual
31        set lambda = convertToAEG(lambda)
32        do evaluate lambda and set its fitness score
33        set lambda.weight = empty
34        insertLambda(outPopulation,lambda)
35      end for k
36    else
37      // Copy and add a few more random individuals
38      set outPopulation = copy(inPopulation)
39      set K = maxPopSize/10
40      for k from 0 until K do // Initialize population
41        set lambda = generate random individual
42        do evaluate lambda and set its fitness score
43        set lambda.weight = empty
44        insertLambda(outPopulation,lambda)
45      end for k
46    end if
47    sort outPopulation in ascending order by fitness score
48    set N = outPopulation.length
49    // Compute weighted complexity range
50    for n from 0 until N do
51      set lambda = outPopulation[n]
52      if (lambda.weight is empty) then set lambda.weight = weightedComplexity(lambda,0)
53      weight = lambda.weight
54      if  (weight < low) then low = weight
55      if  (weight > high) then high = weight
56    end for n
57    range = (high - low)
58    set islandCounts = new vector of length maxIslandCount
59    set inPopulation = new vector of length 0
60    set islands = new vector of length maxIslandCount
61    // Alway return island structure with one island for each complexity partition
62    // Prune all non-dominant individuals in each pareto front complexity island
63    for n from 0 until N do
```

```
64      set lambda = outPopulation[n]
65      set weight = lambda.weight
66      set lambda.island = island = integer(maxIslandCount * ((weight - low) / range))
67      set islandCounts[island] = islandCounts[island] + 1
68      if  (islandCounts[island] <= maxIslandSize) then
69        set inPopulation.last = lambda
70        if (islands[island].length = 0) then set islands[island] = new vector of length
71        set islands[island].last = lambda
72      end if
73    end for n
74    champ = inPopulation.first
75    return champ
76  end fun populationPruning
```

The operator weights for the pareto front population pruning are shown in Table 2.

Table 2. Operator Weights Table

Operator	Weight	Operator	Weight	Operator	Weight
ξ	00	+	01	-	01
*	02	/	02	sqrt	03
square	02	cube	02	quart	02
log	04	exp	04	cos	05
sin	05	tan	06	tanh	06
max	07	min	07	abs	01

4 Constant Optimization Issues

A theoretical issue with basic GP symbolic regression is the poor optimization of embedded constants under the mutation and crossover operators. Notice that in basis functions (E4) and (E5) there are real constants embedded inside the formulas. These embedded constants, 4.56 and .2, are quite important. That is to say that basis function (E4) behaves quite differently than basis function (E4.2) while basis function (E5) behaves quite differently than basis function (E5.4).

- (*E4*) B_3 = sqrt(x2)/tan(x5/4.56)
- (*E4.2*) B_9 = sqrt(x2)/tan(x5)
- (*E5*) B_4 = tanh(cos(x2*.2)*cube(x5+abs(x1)))
- (*E5.4*) B_{10} = tanh(cos(x2)*cube(x5+abs(x1)))

The behavior can be quite startling. For instance, if we generate a set of random independent variables for <x1, x2, x3, xm> and we set the dependent variable, y = sqrt(x2)/tan(x5/4.56), then a regression on y = sqrt(x2)/tan(x5) returns a very bad LSE. In fact the bad regression fit continues until one regresses on y = sqrt(x2)/tan(x5/4.5). It is only until one regresses on y = sqrt(x2)/tan(x5/4.55) that we get a reasonable LSE with an R-Square of .56. Regressing on y = sqrt(x2)/tan(x5/4.555) achieves a better LSE with an R-Square of .74. Of course regressing on y = sqrt(x2)/tan(x5/4.56) returns a perfect LSE with an R-Square of 1.0. Clearly, in many cases of embedded constants, there is a very small neighborhood, around the correct embedded constant, within which an acceptable LSE can be achieved. In standard Koza-style symbolic regression (Koza 1992), the mutation and crossover operators are quite cumbersome in optimizing constants. As standard GP offers no constant manipulation operators per se, the mutation and crossover operators must work doubly hard to optimize constants. For instance, the only way to optimize the embedded constant in s-expression (E5) would be to have a series of mutations or crossovers which resulted in an s-expression with multiple iterative additions and subtractions as follows (Koza 1992).

- (*E4*) B_3 = sqrt(x2)/tan(x5/4.56)
- (*E4.2*) B_3 = sqrt(x2)/tan(x5/(1.0+3.2))
- (*E4.3*) B_3 = sqrt(x2)/tan(x5/((1.0+3.2)+.3))
- (*E4.4*) B_3 = sqrt(x2)/tan(x5/(((1.0+3.2)+.3)+.07))
- (*E4.5*) B_3 = sqrt(x2)/tan(x5/((((1.0+3.2)+.3)+.07)-.01))

Characteristically, the repeated mutation and crossover operations which finally realize an optimized embedded constant also greatly bloat the resulting basis function with byzantine operator sequences (Poli 2009). On the other hand swarm intelligence techniques are quite good at optimizing vectors of real numbers. So the challenge is how to collect the embedded constants found in a GP s-expression into a vector so they can be easily optimized by swarm intelligence techniques. Recent advances in symbolic regression technology including Abstract Expression Grammars (AEGs) (Korns 2010) and (Korns 2011) can be used to control bloat, specify complex search constraints, and expose the embedded constants in a basis function so they are available for manipulation by various swarm intelligence techniques suitable for the manipulation of real numeric values. This presents an opportunity to combine standard genetic programming techniques together with swarm intelligence techniques into a seamless, unified algorithm for pursuing symbolic regression. The focus of this chapter will be an investigation of swarm intelligence techniques, used in connection with AEGs, which can improve the speed and accuracy of symbolic regression search, especially in cases where embedded numeric constants are an issue hindering performance.

Adding Abstract Expression Grammars to standard GP Symbolic Regression (Korns 2010) evolves the GLM's basis functions as AEG individuals. Our

simple modified elitist GP Algorithm (2) is outlined below. The inputs are a vector of N training points, X, a vector of N dependent variables, Y, and the number of generations to train, G. Each point in X is a member of RM = <x1,x2,,xm>. The fitness score is the root mean squared error divided by the standard deviation of Y, NLSE.

Clearly even for three simple basis functions, with only 20 features and very limited depth, the size of the search space is already very large; and, the presence of real constants accounts for a significant portion of that size. For instance, without real constants, S0 = 20, S3 = $1.054 \text{x} 10^{19}$, and with B=3 the final size of the search space is $1.05 \text{x} 10^{57}$.

It is our contention that since real constants account for such a significant portion of the search space, symbolic regression would benefit from special constant evolutionary operations. Since basic GP does not offer such operations, we investigate the enhancement of symbolic regression with swarm intelligence algorithms specifically designed to evolve real constants.

5 Abstract Constants

In standard Koza-style tree-based Genetic Programming (Koza 1992) the genome and the individual are the same Lisp s-expression which is usually illustrated as a tree. Of course the tree-view of an s-expression is only a visual aid, since a Lisp s-expression is normally a list which is a special Lisp data structure. Without altering or restricting standard tree-based GP in any way, we can view the individuals not as trees but instead as s-expressions.

- (*E6*) **depth 0 binary tree s-exp**: 3.45
- (*E7*) **depth 1 binary tree s-exp**: (+ x2 3.45)
- (*E8*) **depth 2 binary tree s-exp**: (/ (+ x2 3.45) (* x0 x2))
- (*E9*) **depth 2 irregular tree s-exp**: (/ (+ x2 3.45) 2.0)

Up until this point we have not altered or restricted standard GP in any way; but, now we are about to make a slight alteration so that the standard GP s-expression can be made swarm friendly. Let us use the following s-expression.

- (*E10*) (* (/ (- x0 3.45) (+ x0 x2)) (/ (- x5 1.31) (* x0 2.1)))

In this individual (E10), the real constants are embedded within the s-expression and are inconvenient for swarm algorithms. So we are going to add an annotation to the individual (E10). We are going to add enumerated constant nodes, and we are going to add a constant chromosome vector creating a new individual (E11). The individual (E11) will now have three components: an abstract s-expression (E11), the original s-expression (E11.1), and a constant chromosome (E11.2) as follows.

- (*E11*) (* (/ (- x0 c[0]) (+ x0 x2)) (/ (- x5 c[1]) (* x0 c[2])))
- (*E11.1*) **s-exp**: (* (/ (- x0 3.45) (+ x0 x2)) (/ (- x5 1.31) (* x0 2.1)))
- (*E11.2*) **c**: <3.45 1.31 2.1>

Individual (E11) evaluates to the exact same value as (E10). Each real number constant in (E10) has been replaced with an indexed vector reference of the type c[i], where c is a vector of real numbers containing the same real numbers originally found in (E10). While this process adds some annotation overhead to (E10), it does expose all of the real number constants in a vector which is swarm intelligence friendly.

At this point let us take a brief pause. Examine the original s-expression (E10) also (E11.1) and compare it to the new annotated abstract version (E11). Walk through the evaluation process for each version. Satisfy yourself that the concrete s-expression (E11.1) and the abstract annotated (E11) both evaluate to exactly the same interim and final values.

We have made no restrictive or destructive changes in the original individual (10). Slightly altered to handle the new constant vector references and the new chromosome annotations, any standard GP system will behave as it did before. Prove it to yourself this way. Take the annotated individual (E11), and replace each indirect reference with the proper value from the constant vector. This converts the abstract annotated (E11) back into the concrete s-expression (E11.1). Let your standard GP system operate on (E11.1) any way it wishes to produce a new individual (E11'.1). Now convert (E11'.1) back into an abstract annotated version with the same process we used to annotate (E10).

Furthermore, if we have a compiled a machine register optimized version, γ(x), of (E10), we do not even have to perform expensive recompilation in order to change a value in the constant chromosome. We need only alter the values in the constant chromosome and re-evaluate the already compiled and optimized γ(x).

Armed with these newly annotated individuals, let's take a fresh look at how we might improve the standard process of genetic programming during a symbolic regression run. Consider the following survivor population in a standard GP island.

- (*E12.1*) (* (/ (- x0 3.45) (+ x0 x2)) (/ (- x5 1.31) (* x0 2.1)))
- (*E12.2*) (cos (/ (- x4 2.3) (min x0 x2)))
- (*E12.3*) (* (/ (- x0 5.15) (+ x0 x2)) (/ (- x5 -2.21) (* x0 9.32)))
- (*E12.4*) (sin (/ (- x4 2.3) (min x0 x2)))
- (*E12.5*) (sin (/ (- x4 2.3) (avg x0 x2)))
- (*E12.6*) (* (/ (- x0 3.23) (+ x0 x2)) (/ (- x5 -6.31) (* x0 7.12)))
- (*E12.7*) (* (/ (- x0 2.13) (+ x0 x2)) (/ (- x5 3.01) (* x0 2.12)))

First of all, the GP mutation and crossover operators do not have any special knowledge of real numbers. They have a difficult time isolating and optimizing numeric constants. But the situation gets worse.

As generation after generation of training has passed, the surviving individuals in the island population have become specialized in common and predictable ways. Individuals (E12.2), (E12.4), and (E12.5) are all close mutations of each other. Evolution has found a form that is pretty good and is trying to search for a more optimal version. GP is fairly good at exploring the search space around these individuals.

However, (E12.1), (E12.3), (E12.6), and (E12.7) are all identical forms with the exception of the values of their embedded numeric constants. As time passes, the survivor population will become increasingly dominated by variants of (E12.1) and in time its progeny may crowd out all other survivors. GP has a difficult time exploring the search space around (E12.1) largely because the form is already optimized - it is the constant values which need additional optimization.

In swarm friendly AEG enhanced symbolic regression system, the individuals (E12.1), (E12.3), (E12.6), and (E12.7) are all viewed as constant homeomorphs and they are stored in the survivor pool as one individual with another annotation: a swarm constant pool as follows.

- (*E13.1*) (* (/ (- x0 c[0]) (+ x0 x2)) (/ (- x5 c[1]) (* x0 c[2])))
- (*E13.1.1*) **s-exp**: (* (/ (- x0 3.45) (+ x0 x2)) (/ (- x5 1.31) (* x0 2.1)))
- (*E13.1.2*) **c**: <3.45 1.31 2.1>
- (*E13.1.3*) **Swarm Constant Pool**
- (*E13.1.3.1*) **pool[0]**: <3.45 1.31 2.1>
- (*E13.1.3.2*) **pool[1]**: <5.15 -2.21 9.32>
- (*E13.1.3.3*) **pool[2]**: <3.23 -6.31 7.12>
- (*E13.1.3.4*) **pool[3]**: <2.13 3.01 2.12>
- (*E13.2*) (cos (/ (- x4 2.3) (min x0 x2)))
- (*E13.3*) (sin (/ (- x4 2.3) (min x0 x2)))
- (*E13.4*) (sin (/ (- x4 2.3) (avg x0 x2)))

The AEG enhanced SR system has combined the individuals (E12.1), (E12.3), (E12.6), and (E12.7) into a single constant homeomorphic canonical version (E13.1) with all of the constants stored in a swarm constant pool inside the individual. Now the GP island population does not become dominated inappropriately. Plus, we are free to apply swarm intelligence algorithms to the constants inside (E13.1) without otherwise hindering the GP algorithms in any way.

6 Constant Optimization

Abstract Expression Grammar GP can be used with particle swarm [3] which evolves the GLM's basis functions as AEG individuals. In Algorithm (4)

swarm evolution is seamlessly merged with standard GP and our AEG particle swarm algorithm is outlined in Algorithm (3) below.

Our Particle Swarm (PSO) algorithm has also been modified to fit within the larger framework of an evolving GP environment. Therefore, the evolutionary loop is in the GP algorithm and has been removed from the PSO algorithm. Instead the PSO algorithm is repeatedly called from the main GP loop during evolution. Furthermore, we must execute the PSO algorithm on all AEG individuals with a non-empty constant pool; therefore, care must be taken such that any one AEG individual does not monopolize the search process.

The PSO algorithm gets its inspiration from the clustering behavior of birds or insects as they fly in formation. There is the concept of an individual swarm member called a particle, the current position of each particle, the best position ever visited by each particle, a velocity for each particle, and the best position every visited by any particle (the global best). In our case, each particle will be one of the constant vectors in our AEG individual's constant pool. A fitness value will be assigned to each constant by wrapping the AEG individual around the constant vector and scoring.

Each AEG <aexp,sexp,c,pool> stores the population of PSO individuals in its constant pool and the current most fit champion as its constant vector c. However, implementing the PSO algorithm requires adding a few new items to our AEG individual. Let aeg be an AEG individual in our system. The best position ever visited by any particle will be designated as **aeg.best** (global best). The best position ever visited by each particle, i, will be designated as **aeg.pool[i].best** (local best). The velocity of each particle, i, will be designated as **aeg.pool[i].v**. The score of a constant vector, c, will be designated as c.fitness. And, of course, each particle, i, is nothing more than one of the constant vectors in the AEG individual's constant pool aeg.pool[i].

The changes and enhancements for constant optimization are shown in Algorithm (A3).

```
Algorithm A3: Particle Swarm Constant Optimization
1   Function: optimizeConstants(me)
2   Parameters: WL, WG, WV, maxPoolSize
    Replaces: A1.optimizeConstants lines 24 thru 26
    Summary: Particle Swarm constant optimization optimizes a
             pool of vectors, in an AEG formatted individual,
             by randomly selecting a pair of constant vectors
             from the pool of constant vectors. A new vector
             is produced when the pair of vectors, together
             with the global best vector, are randomly nudged
             closer together based upon their previous
             approaching velocities. The new vector is scored.
             After scoring, the population of vectors is truncated
```

```
            to those with the best scores.

3   Function convertToAEG(me)
4   Input: me // Koza-style s-expression individual
5   Output: out // AEG annotated individual
    Summary: Converts an s-expression individual into an AEG individual.
             AEG Conversion removes all of the constants from an input s-expression
             and places them in a vector where swarm intelligence algorithms can
             easily optimize them. The output is a constant vector and the original
             s-expression modified to refer indirectly into the constant vector
             instead of referencing the constants directly.
6     set out = <aexp,sexp,c,pool>  // empty AEG individual
7     set out.aexp = me
8     set out.sexp = me
9     set out.c = <..empty vector of reals..>
10    set out.pool = <..empty vector of vectors..>
11    set N  = out.aexp.length
12    for n from 0 until N do
13      if out.aexp[n] is a real number constant then
14        set r = out.aexp[n]
15        set k = out.c.length
16        set out.c[k] = r
17        set out.aexp[n] = "c[k]" // replace r with c indexed reference
18      end if
19    end for
20    set out.pool[0] = out.c
21    return out
22  end fun convertToAEG

23  Function convertToSExp(me)
24  Input: me // AEG formatted individual
25  Output: out // Koza-style s-expression individual
    Summary: Converts an AEG formatted individual into an s-expression individual.
              All AEG constant vector references, like "c[k]", are replaced with
              the actual constant values in the constant vector.
              AEG formatted individuals are structured as: <aexp,sexp,c,pool>
26    set out = copy(me.aexp)
27    set N  = out.length
28    for n from 0 until N do
29      if out[n] is a constant "c[k]" style vector reference then
30        set r = me.c[k]
31        set out[n] = r
32      end if
33    end for
34    return out
```

```
35  end fun convertToSExp

36  Function: insertLambda(population,lambda) // aeg = <aexp,sexp,c,pool>
    Replaces: A1.insertLambda lines 27 thru 31
    Summary: Accepts an input individual (lambda) and converts it into
             AEG format. It then searches the population of AEG individuals
             for a constant homeomorphic AEG (an AEG with matching form and
             constant locations  although the value of the constants may
             be different). If a constant homeomorphic AEG is found, the
             input lambda is merged with the existing AEG version already
             in the population; otherwise, the input lambda is inserted in
             at the end of the population.
37    // Convert everything to AEG format
38    if lambda is not in AEG format then set lambda = convertToAEG(lambda)
39    P = population.length
40    for p from 0 until P do // Search population
41      set w = population[p]
42      if (w.aexp = lambda.aexp) then
43        set w.pool = append(w.pool,lambda.pool)
44        sort w.pool inascending order by fitness score
45        truncate w.pool to the maxPoolSize most fit constant vectors
46        set w.c = w.pool.first
47        set w.sexp = convertToSExp(w)
48        return population
49      end if
50    end for p
51    set population.last = lambda
52    return population
53  end fun insertLambda

54  Function: mutateSExp(me) // me = <aexp,sexp,c,pool>
    Replaces: A1.mutateSExp lines 7 thru 13
    Summary: mutateSExp randomly alters an input s-expression by replacing
             a randomly selected sub expression with a new randomly grown
             sub expression.
55    me = copy(me.sexp)
56    set L = number of nodes in me // me is a list of Lisp Pairs
57    set s = generate random s-expression
58    set n = random integer between 0 and L
59    set me[n] = s  // Replaces nth node with s
60    set me = convertToAEG(me)
61    return me
62  end fun mutateSExp

63  Function: crossoverSExp(dad,mom)
```

```
    Replaces: A1.crossoverSExp lines 14 thru 23
    Summary: crossoverSExp randomly alters a mom input s-expression by
             replacing a randomly selected sub expression in mom with a
             randomly selected sub expression from dad.
64    dad = copy(dad.sexp)
65    mom = copy(mom.sexp)
66    set Ld = number of nodes in dad // dad is a list of Pairs
67    set Lm = number of nodes in mom // mom is a list of Pairs
68    set n = random integer between 0 and Ld
69    set m = random integer between 0 and Lm
70    set dad[n] = mom[m]  // Replaces nth node with mth node
71    set dad = convertToAEG(dad)
72    return dad
73  end fun crossoverSExp

74  Main:
75  vars (Ic starts at 0)
76  set J = me.pool.length
77  if (J<=0) then return me end if
78  i = Ic
79  c = copy(me.pool[i])
80  v = copy(me.pool[i].v)
81  if (v = null) then
82    set v = random velocity vector
83    set me.pool[i].v = v
84  end if
85  lbest = me.pool[i].best
86  if (lbest = null) then
87    set lbest = c
88    set me.pool[i].best = lbest
89  end if
90  gbest = me.best
91  if (gbest = null) then
92    set gbest = c
93    set me.best = gbest
94  end if
95  // Compute the velocity weight parameters
96  maxg = maximum generations in the main GP search
97  g = current generation count in the main GP search
98  WL = .25 + ((maxg - g)/maxg) // local weight
99  WG = .75 + ((maxg - g)/maxg) // global weight
100 WV = .50 + ((maxg - g)/maxg) // velocity weight
101 I = c.length
102 set r1 = random number from 0 to 1.0
103 set r2 = random number from 0 to 1.0
```

```
104 // Update the particle's velocity and position
105 for i from 0 until I do
106   set lnudge = (WL*r1*(lbest[i]-c[i]))
107   set gnudge = (WG*r2*(gbest[i]-c[i]))
108   set v[i] = (WV*v[i])+lnudge+gnudge
109   set c[i] = c[i]+v[i]
110 end for i
111 // Score the new particle position
112 set me.c = c
113 do evaluate me and set my fitness score
114 // Update the best particle positions
115 if (c.fitness > lbest.fitness) then lbest = c end if
116 if (c.fitness > gbest.fitness) then gbest = c end if
117 me.best = gbest
118 set me.pool.last = c
119 set me.pool.last.best = lbest
120 set me.pool.last.v = v
121 // Enforce elitist constant pool
122 sort me.pool ascending by fitness score
123 truncate me.pool to the maxPoolSize most fit constant vectors
124 set me.c = me.pool.first
125 set me.sexp = convertToSExp(me)
126 // Enforce iterative search of constant pool
127 set Ic = Ic + 1
128 if (Ic>=maxPoolSize) then set Ic = 0 end if
129 return me
130 end fun optimizeConstants
```

The results with the enhancements for constant optimization and operator weighted population pruning are shown in Table 3.

Table 3. Results with Weighted Pareto Front Pruning

Test	*WFFs*	*Train-NLSE*	*Train-TCE*	*Test-NLSE*	*Test-TCE*
P01	14K	0.00	0.00	0.00	0.00
P02	96K	0.00	0.00	0.00	0.00
P03	74M	0.00	0.00	0.00	0.00
P04	34K	0.00	0.00	0.00	0.00
P05	94K	0.00	0.00	0.00	0.00
P06	12K	0.00	0.00	0.00	0.00
P07	82M	0.00	0.00	0.00	0.00
P08	1M	0.00	0.00	0.00	0.00
P09	2G	0.00	0.00	0.00	0.00
P10	35G	0.00	0.00	0.00	0.00
P11	21G	0.00	0.00	0.00	0.00
P12	90G	0.98	0.44	1.08	0.52
P13	42G	0.00	0.00	0.00	0.00
P14	88G	0.06	0.04	1.06	0.45
P15	1M	0.00	0.00	0.00	0.00

(*Note: the number of individuals evaluated before finding a solution is listed in the Well Formed Formulas (WFFs) column*)

7 Conclusion

In a previous paper (Korns 2011), significant accuracy issues were identified for state of the art SR systems. In this paper we attempt to lay the ground work for an eventual *proof* of SR accuracy.

A simplified brute force base line symbolic regression algorithm has been constructed, inspired by the work in (Koza 1992). Simplified enhancements for controlling bloat, premature convergence, and embedded constant optimization have been added. The base line algorithm is largely non-parametric and easy to implement. The enhanced base line system achieves accuracy roughly competitive with current state of the art SR systems on a set of previously published test problems.

No attempt has been made to make this base line algorithm commercially competitive. It has not been adapted to multi-tasking or to cloud computing. In fact every effort has been made to simplify the base line algorithm for easy implementation and ease of understanding.

Demonstrations of SR accuracy in the form of a challenge, such as "We'll pay a cash prize for any test problem our SR system cannot solve - *up to a certain complexity*", will be part of the marketing campaigns of the best commercial SR packages. No public domain base line algorithm is needed for simple commercial accuracy challenges.

However, to "prove" SR system accuracy for test problems *up to a certain complexity* will require a well understood public domain SR algorithm. *Proving* SR accuracy will provide much greater academic and industrial penetration benefits than mere open commercial challenges.

This base line algorithm provides an opening opportunity for SR community joint development and enhancement. To be useful, it will have to be duplicated and enhanced by multiple researchers in the SR community. And finally, the final base line algorithm will have to take on a form convenient for SR theoreticians to "prove" accuracy *up to a certain complexity.*

The opportunity is unprecedented. If the symbolic regression community is able to offer accuracy, even within the favorable minimalist assumptions of this chapter, and if that accuracy is vetted or confirmed by an independent body (distinct from the SR community), then symbolic regression will realize its true potential. SR could be yet another machine learning technique (such as linear regression, support vector machines, etc.) to offer a foundation from which hard statistical assertions can be launched. Furthermore, we would finally have realized the original dream of returning not just accurate coefficients but accurate formulas as well.

The challenge is significant. It is unlikely that any single research team, working alone, will be sufficient to meet the challenge. We will have to band together as a community, developing standardized test problems, and standardized grammars. More importantly we will have to reach out to the theoreticians in our GP discipline and in the mathematical and statistical communities to establish some body of conditions whereby independent communities will be willing to undertake the task of confirming SR accuracy. And, of course, first we, in the SR community, will have to work together to achieve such accuracy.

Our further research directions will include enhancements of simplified versions of the elastic net opening initialization concepts in (McConaghy 2011). We will also explore simple statistical analyses and modified learning techniques based upon an analysis of the complexity pareto front which is surfaced in the island structure resulting from operator weight pruning. Every effort will be made to keep the enhanced base line algorithm relatively nonparametric and easy to implement.

What is clear is that if we, in the symbolic regression community, wish to continue making the claim that we return *accurate formulas*; and, if we wish to win the respect of other academic disciplines, then we will have to solve our accuracy issues.

References

1. Gregory S Hornby (2006). Age-Layered Population Structure For reducing the Problem of Premature Convergence, in *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation.* ACM Press, New York.
2. Michael Korns (2010). Abstract Expression Grammar Symbolic Regression, in *Genetic Programming Theory and Practice VIII.* Springer, New York. Kaufmann Publishers, San Francisco California.
3. Michael Korns (2011). Accuracy in Symbolic Regression, in *Genetic Programming Theory and Practice IX.* Springer, New York. Kaufmann Publishers, San Francisco California.
4. Mark Kotanchek, Guido Smits, and Ekaterina Vladislavleva (2008). Trustable Symbolic Regression Models: Using Ensambles, Interval Arithmetic and Pareto Fronts to Develop Robust and Trust-Aware Models, in *Genetic Programming Theory and Practice V.* Springer, New York.
5. John R Koza (1992). Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge Massachusetts.
6. John R Koza (1994). Genetic Programming II: Automatic Discovery of Reusable Programs. The MIT Press, Cambridge Massachusetts.
7. John R Koza, Forrest H Bennett III, David Andre, Martin A Keane (1999). Genetic Programming III: Darwinian Invention and Problem Solving. Morgan
8. McConaghy, Trent, (2011). FFX: Fastm Scalable, Deterministic Symbolic Regression Technology, in *Genetic Programming Theory and Practice IX.* Springer, New York.
9. J.A., Nelder, and R. W. Wedderburn (1972). Generalized linear Models, in *Journal of the Royal Statistical Society*, Series A, General, 135:370-384.
10. Poli, Riccardo, McPhee, Nicholas, Vanneshi, Leonardo, (2009). Analysis of the Effects of Elitism on Bloat in Linear and Tree-based Genetic Programming, in *Genetic Programming Theory and Practice VI.* Springer, New York.
11. Guido Smits, and Mark Kotanchek (2005). Pareto-Front Exploitation in Symbolic Regression, in *Genetic Programming Theory and Practice II.* Springer, New York.
12. Michael Schmidt, Hod Lipson (2010). Age-Fitness Pareto Optimization, in *Genetic Programming Theory and Practice VI.* Springer, New York.