Extreme Accuracy in Symbolic Regression

Michael F. Korns

Analytic Research Foundation, 98 Perea Street, Makati 1229, Manila Philippines
mkorns@korns.com.

Abstract.

Although recent advances in symbolic regression (SR) have promoted the field into the early stages of commercial exploitation, the poor accuracy of SR is still plaguing even the most advanced commercial packages today. Users expect to have the correct formula returned, especially in cases with zero noise and only one basis function with minimally complex grammar depth. Poor accuracy is a hinderence to greater academic and industrial acceptance of SR tools.

In a previous paper, the poor accuracy of Symbolic Regression was explored, and several classes of test formulas, which prove intractable for SR, were examined. An understanding of why these test problems prove intractable was developed. In another paper a baseline Symbolic Regression algorithm was developed with specific techniques for optimizing embedded real numbers constants. These previous steps have placed us in a position to make an attempt at vanquishing the SR accuracy problem.

In this chapter we develop a complex algorithm for modern symbolic regression which is extremely accurate for a large class of Symbolic Regression problems. The class of problems, on which SR is extremely accurate, is described in detail. A definition of extreme accuracy is provided, and an *informal argument* of extreme SR accuracy is outlined in this chapter. Given the critical importance of accuracy in SR, it is our suspicion that in the future all commercial Symbolic Regression packages will use this algorithm or a substitute for this algorithm.

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 2005) and (Kotanchek 2008). Plus there is another commercially deployed regression package which handles up to 50 to 10,000 input features using specialized linear learning (McConaghy 2011).

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 (Korns 2012) we developed a simple, easy to implement, public domain baseline algorithm for modern symbolic regression which is reasonably competitive with current commercial SR packages. This algorithm was 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).

In this chapter we enhance the previous baseline with a complex multi-island algorithm for modern symbolic regression which is extremely accurate for a large class of Symbolic Regression problems. The class of problems, on which SR is extremely accurate, is described in detail. A definition of extreme accuracy is provided, and an *informal argument* of extreme SR accuracy is outlined in this chapter.

Before continuing with the details of our extreme accuracy 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 = 0,1, I, a dependent variable y, and an independent data point with M features x = $<x_0, x_1, x_2, , x_{M-1}>$: such that

- (*E1*) $y = \gamma(x) = c_0 + \Sigma c_i B_i(x) + \mathbf{err}$

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*) $B_0 = x_3$
- (*E3*) $B_1 = x_1+x_4$
- (*E4*) $B_2 = \text{sqrt}(x_2)/\tan(x_5/4.56)$
- (*E5*) $B_3 = \tanh(\cos(x_2*.2)*\text{cube}(x_5+\text{abs}(x_1)))$

If we are minimizing the normalized least squared error, NLSE (Korns 2012), 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: *(/ (+ x_2 3.45) (* x_0 x_2))*, or this depth 2 irregular tree s-exp: *(/ (+ x_4 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 $\in$ T, and f $\in$ F $\cup$ ξ, where *ξ(a,b) = ξ(a) = a*, then any basis function produced by the basic GP system will be represented by at least one element of U_k. Adding the ξ function allows U_k to express all possible basis functions generated by the basic GP system *to a depth of k*. **Note to the reader**, the ξ function performs the job of a pass-through function. The ξ function allows a *fixed-maximal-depth* expression in U_k to express trees of varing depth, such as might be produced from a GP system. For instance, the varying depth GP expression *x_2 + (x_3 - x_5) = ξ(x_2,0.0) + (x_3 - x_5) = +(ξ(x_2,0.0) -(x_3 x_5))* which is a *fixed-maximal-depth* expression in U_2.

In addition to the special pass through function ξ, in our system we also make additional slight alterations to improve coverage, reduce unwanted errors, and restrict results from wandering into the complex number range. All unary functions, such as *cos*, are extended to ignore any extra arguments so that, for all unary functions, *cos(a,b) = cos(a)*. The *sqroot* and *ln* functions are extended for negative arguments so that *sqroot(a) = sqroot(abs(a))* and *ln(a) = ln(abs(a))*.

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 = **(+ - * / cos sin tan tanh sqroot square cube quart exp ln ξ)**. The terminal set is the features $\mathbf{x}_0$ thru $\mathbf{x}_{M-1}$ and the real constant **c**, which we shall consider to be 2^{18} in size.

During the writing of (Korns 2010), (Korns 2011), and (Korns 2012) a high level regression search language was developed called **RQL**. RQL was inspired by the database search language SQL. Therefore RQL is analogous to SQL but not similar to SQL. The algorithm included in this paper is primarily presented in RQL. A very brief, but hopefully sufficient, description of RQL follows.

Regression Query Language **RQL** is a high level Symbolic Regression search language, and consists of **one or more search clauses** which together make up a symbolic regression request. Each search clause represents an independent evolutionary island in which a separate symbolic regression search is performed.

- (*A1*) **search** goal **where** island(breeder,strategy,popsize,pool,serial)
  ...constraints...
  ...events...

It is assumed that the champions from each independent search island will be accumulated into a final list of champions from which the best champion will become the answer to the entire search process. The search **goal** specifies the area to be searched. For example, a common goal is *universal(3,1,t)* which searches all single (1) regression champions from all possible basis functions of depth (3) where the terminals are both (t) variables (*containing features*) or abstract constants (*containing real numbers*). The goal *universal(3,1,t)* is also known as $U_3(1)$ throughout this chapter.

Another search goal example might be $f_0(v_0,f_1(v_1,c_0))$ which searches for a function with two arguments where the second argument is also a function with two argument, the second of which is a constant. The abstract function variables f_0 thru f_K are meant to contain one concrete function from the set $F \cup \xi$ unless otherwise constrained. The abstract feature variables v_0 thru v_J are meant to contain one concrete feature from the set x_0 thru x_{M-1} unless otherwise constrained. The abstract constant variables c_0 thru c_L are meant to contain one real number, of size 2^{cbit}, unless otherwise constrained. The *constraints*, located anywhere after the **where** keyword, are in the form of limitations on variable and function variable coverage such as $f_0(cos,sin,tan,tanh)$ or $v_0(x_0,x_3,x_{10})$ or $c_0(3.45)$.

The **island** keyword sets up the parameters of the evolutionary search island. We use only two **breeders**: *pareto* which implements a typical pareto front algorithm and also understands *onfinal* and *onscore* events, and *smart* which implements a focused elitist algorithm and also understands *onfinal* and *onscore* events. We use only one population operator **strategy** *standard* which implements typical elitist mutation and crossover operators, plus standard swarm operators for optimizing embedded constants, see the baseline algorithm (Korns 2012). The population size **popsize**, constant pool size **pool**, and number of serial iterations per generation **serial** vary with each search specification.

Three other *constraint* and *event* clauses may appear anywhere after the **where** keyword. These are the **isolate** *constraint* clause, and the **onscore** and **onfinal** *events*. Each of these will be explained, with brief descriptions and actual examples, as we detail specific regression search requests required for the extreme accuracy algorithm.

Incidentally any reasonable pareto front implementation, any reasonable elitist implementation, any reasonable standard set of population operators, and any reasonable set of swarm optimizers for embedded constants will work with this extreme accuracy algorithm. The key to implementing this extreme accuracy algorithm lies in the number of independent search island requests, and exactly what is searched for in each independent island. Which brings us to the core issues involved in the pursuit of extreme accuracy.

When searching for a single regression champion with one simple basis function of depth 3 i.e. *universal(3,1,t)* also known as $U_3(1)$, one encounters a number of difficult problems. Many of the simple forms covered in $U_3(1)$, such as *cos(x_{10})*, cannot be easily discovered by evolutionary methods. This is because getting close to the champion does not necessarily convey a fitness improvement. For instance where cos(x_{10}) is the correct champion, it is not clear that cos(x_8) cos(x_9) cos(x_{11}) provide any fitness improvement which evolutionary search methods might exploit. Another easily understood pair of examples can be shown where the correct champion is square(x_{10}+3.427). Any trial champion such as cube(x_{10}+c_0) will have its fitness improved as c_0 approaches 3.427. Unfortunately this convenient fitness improvement does not occur when the correct champion is cos(x_{10}+3.427) and the trial champion is cube(x_{10}+c_0) or even cos(x_{10}+c_0).

So the obvious answer is to search *universal(3,1,t)* serially for every possible value of functions, variables, and embedded constants. This is fine when the number of functions and variables are small and when the number of bits (*cbit*) used to represent embedded constants is small. **However Symbolic Regression is of greatest value when the number of functions, features, and cbits is large.** In our work in this chapter we have the number of functions $\|F\|$ = 15, the number of features $\|V\|$ = 100, and *cbit* = 18. The size of *universal(3,1,t)* can be computed with the following formula $\|F\|^7*(\|V\|+2^{cbit})^8$. Therefore $15^7*(100+2^{18})^8$. = 3.82E+51 which is larger than the estimated number of stars in our universe.

Since serial search of *universal(3,1,t)* is not possible in reasonable time, pursuit of extreme accuracy requires us to move on to the more complex algorithm presented in this chapter. This extreme accuracy algorithm relies on three strategies. **First**, carving out the smaller subsets of *universal(3,1,t)* which can be shown to require serial search and demonstrating these areas are small enough to be serially searched in practical time. **Second**, carving out the larger subsets of *universal(3,1,t)* which are tractable for evolutionary search and demonstrating these larger areas are responsive to evolutionary search in practical time. **Third**, for those remaining areas too large for serial search and too unresponsive for evolutionary search, we use algebraic manipulations and mathematical regression equivalences to reduce these problems spaces to equivalent spaces which can be solved.

Our core assertion in this chapter is that the algorithm will find extremely accurate champions for all of the problems in $\mathbf{U}_2\mathbf{(1)}$ and in $\mathbf{U}_1\mathbf{(3)}$.

1.1 Example Test Problems

In this section we list the example test problems which we will address. All of these test problems lie in the domain of either $\mathbf{U}_2\mathbf{(1)}$ or $\mathbf{U}_1\mathbf{(3)}$ where the function set F = **(+ - * / cos sin tan tanh sqroot square cube quart exp ln ξ)**, and the terminal set is the features $\mathbf{x}_0$ thru $\mathbf{x}_{M-1}$ plus the constant **c** with **cbit** = 18. Our test will reference 100 features. Our core assertion is that the algorithm will find extremely accurate champions for all of these problems and for **all similar problems** in practical time.

- (*T1*): $y = 1.57 + (14.3*x_3)$
- (*T2*): $y = 3.57 + (24.33/x_3)$
- (*T3*): $y = 1.687 + (94.183*(x_3*x_2))$
- (*T4*): $y = 21.37 + (41.13*(x_3/x_2))$
- (*T5*): $y = -1.57 + (2.3*((x_3*x_0)*x_2))$
- (*T6*): $y = 9.00 + (24.983*((x_3*x_0)*(x_2*x_4)))$
- (*T7*): $y = -71.57 + (64.3*((x_3*x_0)/x_2))$
- (*T8*): $y = 5.127 + (21.3*((x_3*x_0)/(x_2*x_4)))$
- (*T9*): $y = 11.57 + (69.113*((x_3*x_0)/(x_2+x_4)))$
- (*T10*): $y = 206.23 + (14.2*((x_3*x_1)/(3.821-x_4)))$
- (*T11*): $y = 0.23 + (19.2*((x_3-83.519)/(93.821-x_4)))$
- (*T12*): $y = 0.283 + (64.2*((x_3-33.519)/(x_0-x_4)))$
- (*T13*): $y = -2.3 + (1.13*\sin(x_2))$
- (*T14*): $y = 206.23 + (14.2*(\exp(\cos(x_4))))$
- (*T15*): $y = -12.3 + (2.13*\cos(x_2*13.526))$
- (*T16*): $y = -12.3 + (2.13*\tan(95.629/x_2))$
- (*T17*): $y = -28.3 + (92.13*\tanh(x_2*x_4))$
- (*T18*): $y = -222.13 + (-0.13*\tanh(x_2/x_4))$
- (*T19*): $y = -2.3 + (-6.13*\sin(x_2)*x_3)$
- (*T20*): $y = -2.36 + (28.413*\ln(x_2)/x_3)$
- (*T21*): $y = 21.234 + (30.13*\cos(x_2)*\tan(x_4))$
- (*T22*): $y = -2.3 + (41.93*\cos(x_2)/\tan(x_4))$
- (*T23*): $y = .913 + (62.13*\ln(x_2)/\mathrm{square}(x_4))$
- (*T24*): $y = 13.3 + (80.23*x_2) + (1.13*x_3)$
- (*T25*): $y = 18.163 + (95.173/x_2) + (1.13/x_3)$
- (*T26*): $y = 22.3 + (62.13*x_2) + (9.23*\sin(x_3))$
- (*T27*): $y = 93.43 + (71.13*\tanh(x_3)) + (41.13*\sin(x_3))$
- (*T28*): $y = 36.1 + (3.13*x_2) + (1.13*x_3) + (2.19*x_0)$
- (*T29*): $y = 17.9 + (2.13*x_2) + (1.99*\sin(x_3)) + (1.13*\cos(x_3))$
- (*T30*): $y = -52.183 + (9.13*\tanh(x_3)) + (-11.13*\sin(x_3)) + (14.3*\ln(x_3))$

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 between -100 and +100. Then the model will be applied to produce the dependent variable. These steps will create the training data (each matrix row is a *training example* and each matrix column is a *feature*). 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 between -100 and +100. Then the model will be applied to produce the dependent variable. These steps will create the testing data. The fitness score is the root mean squared error divided by the standard deviation of Y, NLSE. The estimator will be evaluated against the testing data producing the final NLSE and R-Square scores for comparison.

For the purposes of this algorithm, ***extremely accurate*** will be defined as any champion which achieves a normalized least squares error (NLSE) of

.0001 or less on the **testing data** under conditions where both the training data and testing data were constructed with zero noise.

All timings quoted in this chapter were performed on a Dell XPS L521X Intel i7 quad core laptop with 16Gig of RAM, and 1Tb of hard drive, manufactured in Dec 2012 (our test machine). Each test problem was trained against 10,000 training examples with 100 features per example, and tested against 10,000 testing examples with 100 features per example. Noise was NOT introduced into any of the test problems, so an exact answer was always theoretically possible.

2 General Search Island

The extremely accurate algorithm begins with an RQL search command which sets up a blanket search of a user specified depth and breadth

- (*S0*) **search** universal(D,B,t) **where** island(pareto,standard,100,100,200) op(ξ,+,-,*,/,cos,sin,tan,tanh,sqroot,square,cube,quart,exp,ln)

For our purposes herein we will set the expression depth $D = 4$ and the number of basis functions $B = 3$. But any user specified depth and number of basis functions can be accommodated.

Search command (S0) assumes that one has an SR system at least as capable as the baseline algorithm in (Korns 2012), a reasonably competent implementation of pareto front breeding with *onfinal* and *onscore* event handling, a reasonably competent implementation of standard population operators with mutation, crossover, and swarm optimizers for the constant pools. The survivor population size will be 100. The constant pool size will be 100. Each generation 200 serial iterations will be made in universal(4,3,t). This island search is independent of all other search commands in the algorithm.

Search (S0) will provide the same breadth and depth of search and will be as accurate as any existing commercial package - depending upon the implementation of *pareto* and *standard*. That is the good news. The bad news is that (S0) will not be extremely accurate because of the issues already mentioned. The size of the (S0) search space is $(15^{15}*(100+2^{18})^{16})^3$ = 10.0E+312. So the serial search at 200 iterations per generation will take longer than the age of the universe to complete. Meaning that we can't count on serial search and evolutionary search is powerful but not extremely accurate.

Therefore, if we wish to achieve extreme accuracy on $U_2(1)$ and $U_1(3)$, additional search commands will have to be added to the algorithm. These search commands will be executed independently and asynchronously from search (S0). Taken as a whole, general search (S0) together with the specialized searches to be added will constitute the entire extreme accuracy algorithm. The additional specialized search commands will carve out subsets of $U_2(1)$

and $U_1(3)$ which are amenable to serial search in practical time, will carve out the subsets which are tractable for evolutionary search, and using algebraic manipulations and mathematical regression equivalences will carve out the subsets which can be solved with complex search commands.

There will be 24, *which for cloud deployment can be expanded into 80 searches*, of these additional RQL search commands in the algorithm. Each RQL search command sets up a search island independent of all other search islands. This allows the algorithm to be easily distributed across multiple computers or in a cloud environment. The champions from each island are gathered together with the most fit champion being the answer to this RQL query.

The algorithm's claim of extreme accuracy is suported by what might be called an *informal argument* rather than a formal proof. A brief sketch of the informal arguments will accompany each of the 24 RQL commands with, hopefully, enough information and examples to allow the reader to understand the basic reasoning supporting the claim of extreme accuracy.

3 $U_1(3)$ Search Island

The RQL search command covering the space $U_1(3)$ is thankfully fairly straightforward and the space responds very well to evolutionary search.

- (*S1*) **search** regress($f_0(v_0,v_1)$,$f_1(v_2,v_3)$,$f_2(v_4,v_5)$) **where** island(smart,standard,10,25,200) op(ξ,+,-,*,/,cos,sin,tan,tanh,sqroot,square,cube,quart,exp,ln,inv)

Search command (S1) performs multiple regressions with three basis functions, each of which is in U_1 and looks like f(t,t). Each of the variables v_0 thru v_5 contain a single feature from the set x_0 thru x_{99}. Each of f_0, f_1, and f_2 are function variables contains functions from the set $\mathbf{F} \cup \xi \cup$ inv. From the terms, t, all embedded constants can be eliminated because they cancel out of the basis function and enhance the regression coefficient for the basis function as shown in the following examples.

- (*E6*) regress(c_0+v_0) = a+b*(c_0+v_0) = a+(b*c_0)+b*v_0 = regress(v_0)
- (*E7*) regress(c_0/v_0) = a+b*(c_0/v_0) = a+(b*c_0)/v_0 = regress(inv(v_0))
- (*E8*) regress(cos(c_0)) = a+b*cos(c_0) = c_1

Since we can eliminate all of the embedded constants from each term in U_1, we are left with *regress($f_0(v_0,v_1)$,$f_1(v_2,v_3)$,$f_2(v_4,v_5)$)* as our search goal.

4 Search Island S2

The RQL search command covering the space $f_0(f_1(v_0,v_1))$ in U_2 is necessary because large portions of this space do not respond well to evolutionary methods.

- (*S2*) **search** regress($f_0(f_1(v_0,v_1))$) **where** island(smart,standard,10,25,200)
  f_0(ξ,inv,cos,sin,tan,tanh,sqroot,square,cube,quart,exp,ln)
  f_1(ξ,+,-,*,/,inv,cos,sin,tan,tanh,sqroot,square,cube,quart,exp,ln)

Search command (S2) performs single regressions where each of the variables v_0 thru v_1 contain single features from the set x_0 thru x_{99}. The search space size is 12*16*100*100 = 1.92M. At 200 serial iterations per generation, this search will require a maximum of 9,600 generations. On our test machine, each generation requires .00012hrs. So the maximum time required for this search island to complete is 1.152hrs.

5 Search Island S3

The RQL search command covering the space $f_0(f_1(c_0,v_0))$ in U_2 allows the algorithm to carve out a large space which responds very well to evolutionary search methods.

- (*S3*) **search** regress($f_0(f_1(c_0,v_0))$) **where** island(smart,standard,10,25,200)
  f_0(inv,sqroot,square,cube,quart,exp,ln)
  f_1(rdiv,*,+,-,rsub)

Search command (S3) performs single regressions where the variable v_0 contains features from the set x_0 thru x_{99}. The function rdiv is defined as rdiv(c_0,v_0) = v_0/c_0. The function rsub is defined as rsub(c_0,v_0) = v_0-c_0. The search space size is $7*5*2^{18}*100$ = 917M. All of the unary functions are mostly monotonic. The binary operators are all monotonic. This very large search space responds well to evolutionary methods.

6 Search Islands S4 thru S15

The RQL search command also covering the space $f_0(f_1(c_0,v_0))$ in U_2 but where f_0 are all of the unary functions allows the algorithm to isolate the space which do not respond well to evolutionary search methods. With these search islands the algorithm carves out a series of difficult spaces to be searched serially.

- (*S4*) **search** regress($\cos(c_0+v_0)$) **where** island(smart,standard,10,25,2000)
- (*S5*) **search** regress($\cos(c_0*v_0)$) **where** island(smart,standard,10,25,2000)
- (*S6*) **search** regress($\cos(c_0/v_0)$) **where** island(smart,standard,10,25,2000)
- (*S7*) **search** regress($\sin(c_0+v_0)$) **where** island(smart,standard,10,25,2000)
- (*S8*) **search** regress($\sin(c_0*v_0)$) **where** island(smart,standard,10,25,2000)
- (*S9*) **search** regress($\sin(c_0/v_0)$) **where** island(smart,standard,10,25,2000)
- (*S10*) **search** regress($\tan(c_0+v_0)$) **where** island(smart,standard,10,25,2000)
- (*S11*) **search** regress($\tan(c_0*v_0)$) **where** island(smart,standard,10,25,2000)

- (*S12*) **search** regress(tan(c_0/v_0)) **where** island(smart,standard,10,25,2000)
- (*S13*) **search** regress(tanh(c_0+v_0)) **where** island(smart,standard,10,25,2000)
- (*S14*) **search** regress(tanh(c_0*v_0)) **where** island(smart,standard,10,25,2000)
- (*S15*) **search** regress(tanh(c_0/v_0)) **where** island(smart,standard,10,25,2000)

Search commands (S4) thru (S15) perform single regressions where the variable v_0 contains features from the set x_0 thru x_{99}. The reverse argument order f_0(v_0,c_0) and the binary operators rdiv and rsub, from S(3), do not have to be searched because these trigonometric functions all share the following properties.

- (*E9*) regress(cos(v_0+c_0)) = regress(cos(c_0+v_0))
- (*E10*) regress(cos(v_0-c_0)) = regress(cos(c_1-v_0))
- (*E11*) regress(cos(v_0/c_0)) = regress(cos(c_1*v_0))

The search space size, for each of these islands, is 2^{18}*100 = 26.2144M. At 2000 serial iterations per generation, this search will require a maximum of 13,107 generations. On our test machine, each generation requires .0001998hrs. So the maximum time required for this search island to complete is 2.619hrs.

Taken together searches (S3) thru (S15) cover the entire space of f_0(f_1(c_0,v_0)) and f_0(f_1(v_0,c_0)) where f_0(+,-,*,/) and f_1(inv,cos,sin,tan,tanh,sqroot,square,cube,quart,exp,ln).

7 Search Islands S16

The RQL search command covering the space f_1(f_0(v_0),f_2(v_1,v_2)) in U_2 allows the algorithm to carve out a large space with both evolutionary and serial search methods. Search (S5) performs single regressions where the variables v_0, v_1, and v_2 contain single features from the set x_0 thru x_{99}.

- (*S16*) **search** regress(f_1(f_0(v_0),f_2(v_1,v_2))) **where** island(smart,standard,10,25,400)
  f_0(ξ,inv,cos,sin,tan,tanh,sqroot,square,cube,quart,exp,ln)
  f_1(+,-,rsub,*,/,rdiv)
  f_2(+,-,*,/)

This space is so large that, in its cloud version, it must be carved up into 24 separate island searches in order to achieve results in practical time. The **where** clause for each search (S5.1) thru (S5.24) island contains the following.

- **where**
  island(smart,standard,10,25,400)
  f_0(ξ,inv,cos,sin,tan,tanh,sqroot,square,cube,quart,exp,ln)

The space is carved up by expanding the f_1 and f_2 functions as shown below.

- (*S16.1*) **search** regress(f_0(v_0)/(v_1/v_2)) **where** ...as above...
- (*S16.2*) **search** regress(f_0(v_0)/(v_1*v_2)) **where** ...as above...

- (*S16.3*) **search** regress($f_0(v_0)/(v_1+v_2)$) **where** ...as above...
- (*S16.4*) **search** regress($f_0(v_0)/(v_1-v_2)$) **where** ...as above...
- (*S16.5*) **search** regress($f_0(v_0)*(v_1/v_2)$) **where** ...as above...
- (*S16.6*) **search** regress($f_0(v_0)*(v_1*v_2)$) **where** ...as above...
- (*S16.7*) **search** regress($f_0(v_0)*(v_1+v_2)$) **where** ...as above...
- (*S16.8*) **search** regress($f_0(v_0)*(v_1-v_2)$) **where** ...as above...
- (*S16.9*) **search** regress($(v_1/v_2)/f_0(v_0)$) **where** ...as above...
- (*S16.10*) **search** regress($(v_1*v_2)/f_0(v_0)$) **where** ...as above...
- (*S16.11*) **search** regress($(v_1+v_2)/f_0(v_0)$) **where** ...as above...
- (*S16.12*) **search** regress($(v_1-v_2)/f_0(v_0)$) **where** ...as above...
- (*S16.13*) **search** regress($f_0(v_0)+(v_1/v_2)$) **where** ...as above...
- (*S16.14*) **search** regress($f_0(v_0)+(v_1*v_2)$) **where** ...as above...
- (*S16.15*) **search** regress($f_0(v_0)+(v_1+v_2)$) **where** ...as above...
- (*S16.16*) **search** regress($f_0(v_0)+(v_1-v_2)$) **where** ...as above...
- (*S16.17*) **search** regress($f_0(v_0)-(v_1/v_2)$) **where** ...as above...
- (*S16.18*) **search** regress($f_0(v_0)-(v_1*v_2)$) **where** ...as above...
- (*S16.19*) **search** regress($f_0(v_0)-(v_1+v_2)$) **where** ...as above...
- (*S16.20*) **search** regress($f_0(v_0)-(v_1-v_2)$) **where** ...as above...
- (*S16.21*) **search** regress($(v_1/v_2)-f_0(v_0)$) **where** ...as above...
- (*S16.22*) **search** regress($(v_1*v_2)-f_0(v_0)$) **where** ...as above...
- (*S16.23*) **search** regress($(v_1+v_2)-f_0(v_0)$) **where** ...as above...
- (*S16.24*) **search** regress($(v_1-v_2)-f_0(v_0)$) **where** ...as above...

The search space size, for each of these islands, is 100*100*100*12 = 12M. At 400 serial iterations per generation, this search will require a maximum of 30,000 generations. On our test machine, each generation requires .00021hrs. So the maximum time required for this search island to complete is 6.3hrs.

Most often the evolutionary search finds the correct answer in far less time. For instance, even in the case of this difficult target *y = (1.57 + (2.13*(cos(x99)*(x98/x97))))*, the evolutionary search normally finds the target in less than half of the maximum serial time.

Taken together searches (S16.1) thru (S16.24) cover the entire space of (S16).

8 Search Islands S17

The RQL search command covering the space $f_1(f_0(v_0,v_1),f_2(v_2,v_3))$ in U_2 allows the algorithm to carve out a large space with both evolutionary and serial search methods. Search (S17) performs single regressions where the variables v_0, v_1, v_2, and v_3 contain single features from the set x_0 thru x_{99}. This is by far the largest and most costly search required to cover U_2.

- (*S17*) **search** regress($f_0(f_1(v_0,v_1),f_2(v_2,v_3))$) **where** island(smart,standard,10,25,4000) $f_0(*,/)$

f_1(+,-,*,/)
f_2(+,-,*,/)

The reason that we can eliminate the + and - operators from f_0 is precisely because those expansions are linear in two basis functions and will be solved independently by search island (S1). Therefore we do not have to expand them here. Nevertheless, even this reduced space is so large that it must be carved up into 32 separate island searches in order to achieve results in practical time. The space is carved up by expanding the f_0, f_1, and f_2 functions as shown below.

- (*S17.1*) **search** regress((v_0+v_1)*(v_2+v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.2*) **search** regress((v_0+v_1)*(v_2-v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.3*) **search** regress((v_0+v_1)*(v_2*v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.4*) **search** regress((v_0+v_1)*(v_2/v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.5*) **search** regress((v_0+v_1)/(v_2+v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.6*) **search** regress((v_0+v_1)/(v_2-v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.7*) **search** regress((v_0+v_1)/(v_2*v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.8*) **search** regress((v_0+v_1)/(v_2/v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.9*) **search** regress((v_0-v_1)*(v_2+v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.10*) **search** regress((v_0-v_1)*(v_2-v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.11*) **search** regress((v_0-v_1)*(v_2*v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.12*) **search** regress((v_0-v_1)*(v_2/v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.13*) **search** regress((v_0-v_1)/(v_2+v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.14*) **search** regress((v_0-v_1)/(v_2-v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.15*) **search** regress((v_0-v_1)/(v_2*v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.16*) **search** regress((v_0-v_1)/(v_2/v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.17*) **search** regress((v_0*v_1)*(v_2+v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.18*) **search** regress((v_0*v_1)*(v_2-v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.19*) **search** regress((v_0*v_1)*(v_2*v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.20*) **search** regress((v_0*v_1)*(v_2/v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.21*) **search** regress((v_0*v_1)/(v_2+v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.22*) **search** regress((v_0*v_1)/(v_2-v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.23*) **search** regress((v_0*v_1)/(v_2*v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.24*) **search** regress((v_0*v_1)/(v_2/v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.25*) **search** regress((v_0/v_1)*(v_2+v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.26*) **search** regress((v_0/v_1)*(v_2-v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.27*) **search** regress((v_0/v_1)*(v_2*v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.28*) **search** regress((v_0/v_1)*(v_2/v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.29*) **search** regress((v_0/v_1)/(v_2+v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.30*) **search** regress((v_0/v_1)/(v_2-v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.31*) **search** regress((v_0/v_1)/(v_2*v_3)) **where** island(smart,standard,10,25,4000)
- (*S17.32*) **search** regress((v_0/v_1)/(v_2/v_3)) **where** island(smart,standard,10,25,4000)

The search space size, for each of these islands, is 100*100*100*100 = 100M. At 4000 serial iterations per generation, this search will require a maximum of 25,000 generations. On our test machine, each generation requires .009hrs. So the maximum time required for this search island to complete is 225hrs = 9.375days.

Most often the evolutionary search finds the correct answer in far less time. For instance, even in the case of this difficult target *y = (1.57 + (2.13*((x97-x96)/(x98+x99))))*, the evolutionary search normally finds the target in less than a quarter of the maximum serial time.

Taken together searches (S17.1) thru (S17.32) cover the entire space of (S17).

9 Search Island S18

The RQL search command covering the space $f_0(f_1(v_0),f_2(v_1))$ in U_2 allows the algorithm to carve out a large space with both evolutionary and serial search methods. Search (S18) performs single regressions where the variables v_0, and v_1 contain single features from the set x_0 thru x_{99}.

- (*S18*) **search** regress($f_0(f_1(v_0),f_2(v_1))$) **where** island(smart,standard,10,25,200)
  f_0(+,-,*,/)
  f_1(ξ,inv,cos,sin,tan,tanh,sqroot,square,cube,quart,exp,ln)
  f_2(ξ,inv,cos,sin,tan,tanh,sqroot,square,cube,quart,exp,ln)

The search space size, for each of this island, is 100*100*12*12*4 = 5.76M. At 200 serial iterations per generation, this search will require a maximum of 28,800 generations. On our test machine, each generation requires .000135hrs. So the maximum time required for this search island to complete is 3.9hrs.

Most often the evolutionary search finds the correct answer in far less time. For instance, even in the case of this difficult target *y = (1.57 + (2.13*(ln(x98)/quart(x99))))*, the evolutionary search normally finds the target in less than half of the maximum serial time.

10 Search Island S19

The RQL search command covering the space $f_0(f_1(v_0),f_2(c_0,v_1))$ in U_2 allows the algorithm to carve out a large space where evolutionary search and serial search methods are both intractable. The formal RQL search command is.

- (*E12*) **search** regress($f_1(f_0((v_0),f_2(c_0,v_1)))$) **where** island(smart,standard,10,25,200)
  f_0(ξ,inv,cos,sin,tan,tanh,sqroot,square,cube,quart,exp,ln)
  f_1(+,-,*,/,rsub,rdiv)
  f_2(+,-,*,/,rsub,rdiv)

If we are to try E12 by evolutionary search we run into trouble with test problems such as *y = cos(v_0)/(c_0+v_1)*. If we try serial search we see that the size of this space is 100*100*12*6*6*2^{18} = 1.13T which at 200 iterations per generation will require 5,662,310,400 generations. On our test computer each generation requires .00021hrs. So we will finish testing all possible serial combinations in approximately 1,189,085hrs = 49,545days = 135yrs.

Since searching for (E12) is not practical under any approach, we take a giant leap and search for the the following.

- (*S19*) **search** regress(f_0(v_0),1/f_0(v_0),v_1,1/v_1,f_0(v_0)*v_1,f_0(v_0)/v_1,v_1/f_0(v_0),1/(v_1*f_0(v_0)))
  where island(smart,standard,10,25,200)
  f_0(ξ,inv,cos,sin,tan,tanh,sqroot,square,cube,quart,exp,ln)
  onfinal(regress($poly$))

There is remarkable difference between (E12) and (S19). Search (E12) performs single regression; while, search (S19) performs multiple regressions where the variables v_0, and v_1 contain features from the set x_0 thru x_{99} (*the* **onfinal** *clause simply eliminates all zero or near zero coefficient terms from the final answer and converts to simpler form if possible*). Showing/Learning how these two searches relate to each other will require a bit of simple *regression math* and will help wake us up.

First, notice that search (E12) can be expanded into the following 36 single regression cases with the following equivalent regressions. Notice that all except the three *bolded* cases can be expanded into equivalent simpler regressions.

- (*E12.1*) regress(f_0(v_0)+(c_0+v_1)) = regress(f_0(v_0),v_1)
- (*E12.2*) regress(f_0(v_0)-(c_0+v_1)) = regress(f_0(v_0),v_1)
- (*E12.3*) **regress(f_0(v_0)/(c_0+v_1)) = regress(f_0(v_0)/(c_0+v_1))**
- (*E12.4*) regress(f_0(v_0)*(c_0+v_1)) = regress(f_0(v_0),f_0(v_0)*v_1)
- (*E12.5*) regress((c_0+v_1)-f_0(v_0)) = regress(f_0(v_0),v_1)
- (*E12.6*) regress((c_0+v_1)/f_0(v_0)) = regress(1/f_0(v_0),v_1/f_0(v_0))
- (*E12.7*) regress(f_0(v_0)+(c_0-v_1)) = regress(f_0(v_0),v_1)
- (*E12.8*) regress(f_0(v_0)-(c_0-v_1)) = regress(f_0(v_0),v_1))
- (*E12.9*) **regress(f_0(v_0)/(c_0-v_1)) = regress(f_0(v_0)/(c_0-v_1))**
- (*E12.10*) regress(f_0(v_0)*(c_0-v_1)) = regress(f_0(v_0),f_0(v_0)*v_1)
- (*E12.11*) regress((c_0-v_1)-f_0(v_0)) = regress(f_0(v_0),v_1)
- (*E12.12*) regress((c_0-v_1)/f_0(v_0)) = regress(1/f_0(v_0),v_1/f_0(v_0))
- (*E12.13*) regress(f_0(v_0)+(c_0*v_1)) = regress(f_0(v_0),v_1)
- (*E12.14*) regress(f_0(v_0)-(c_0*v_1)) = regress(f_0(v_0),v_1)
- (*E12.15*) regress(f_0(v_0)/(c_0*v_1)) = regress(f_0(v_0)/v_1)
- (*E12.16*) regress(f_0(v_0)*(c_0*v_1)) = regress(f_0(v_0)*v_1)
- (*E12.17*) regress((c_0*v_1)-f_0(v_0)) = regress(f_0(v_0),v_1)
- (*E12.18*) regress((c_0*v_1)/f_0(v_0)) = regress(v_1/f_0(v_0))
- (*E12.19*) regress(f_0(v_0)+(c_0/v_1)) = regress(f_0(v_0),1/v_1)
- (*E12.20*) regress(f_0(v_0)-(c_0/v_1)) = regress(f_0(v_0),1/v_1)

- (*E12.21*) regress($f_0(v_0)/(c_0/v_1)$) = regress($f_0(v_0)*v_1$)
- (*E12.22*) regress($f_0(v_0)*(c_0/v_1)$) = regress($f_0(v_0)/v_1$)
- (*E12.23*) regress($(c_0/v_1)-f_0(v_0)$) = regress($f_0(v_0),1/v_1$)
- (*E12.24*) regress($(c_0/v_1)/f_0(v_0)$) = regress($1/(f_0(v_0)*v_1)$)
- (*E12.25*) regress($f_0(v_0)+(v_1-c_0)$) = regress($f_0(v_0),v_1$)
- (*E12.26*) regress($f_0(v_0)-(v_1-c_0)$) = regress($f_0(v_0),v_1$)
- (*E12.27*) **regress($f_0(v_0)/(v_1-c_0)$) = regress($f_0(v_0)/(v_1-c_0)$)**
- (*E12.28*) regress($f_0(v_0)*(v_1-c_0)$) = regress($f_0(v_0),f_0(v_0)*v_1$)
- (*E12.29*) regress($(v_1-c_0)-f_0(v_0)$) = regress($f_0(v_0),v_1$)
- (*E12.30*) regress($(v_1-c_0)/f_0(v_0)$) = regress($1/f_0(v_0),v_1/f_0(v_0)$)
- (*E12.31*) regress($f_0(v_0)+(v_1/c_0)$) = regress($f_0(v_0),v_1$)
- (*E12.32*) regress($f_0(v_0)-(v_1/c_0)$) = regress($f_0(v_0),v_1$)
- (*E12.33*) regress($f_0(v_0)/(v_1/c_0)$) = regress($f_0(v_0)/v_1$)
- (*E12.34*) regress($f_0(v_0)*(v_1/c_0)$) = regress($f_0(v_0)*v_1$)
- (*E12.35*) regress($(v_1/c_0)-f_0(v_0)$) = regress($f_0(v_0),v_1$)
- (*E12.36*) regress($(v_1/c_0)/f_0(v_0)$) = regress($v_1/f_0(v_0)$)

Eliminating the three bolded cases and collecting all the equivalent regression terms from the right hand side of equations (E12.1) thru (E12.36) we arrive at *regress($f_0(v_0),1/f_0(v_0),v_1,1/v_1,f_0(v_0)*v_1,f_0(v_0)/v_1,v_1/f_0(v_0),1/(v_1*f_0(v_0))$)* which is equivalent to search (S19). Addressing the bolded cases (E12.3), (E12.9), and (E12.27) is more complicated and will be left to the following section.

The search space size, for island (S19), is 100*100*12 = 120,000. At 200 serial iterations per generation, this search will require a maximum of 600 generations. On our test machine, each generation requires .000435hrs. So the maximum time required for this search island to complete is 0.261hrs.

Most often the evolutionary search finds the correct answer in far less time. For instance, even in the case of this difficult target *y = 1.57 + (2.13*(ln(x98)*(23.583-x99)))*, the evolutionary search normally finds the target in less than a third of the maximum serial time.

11 Search Island S20

The previous search island (S19) addressed the RQL search command covering the space $f_0(f_1(v_0),f_2(c_0,v_1))$ see (E12); however, three more complicated regression cases (E12.3), (E12.9), and (E12.27) were left to this section. If we are to try these three cases by evolutionary search we run into trouble with test problems such as $y = cos(v_0)/(c_0-v_1)$. If we try serial search we see that the size of each of these spaces is $100*100*12*2^{18}$ = 31.5B which at 200 iterations per generation will require 157,286,400 generations. On our test computer each generation requires .000135hrs. So we will finish testing all possible serial combinations in approximately 21,233hrs = 9,830days = 2.4yrs.

Since searching for (E12.3), (E12.9), and (E12.27) is not practical under any approach, we take a giant leap and search for the the following.

- (*S20*) **search** regress(v_1*y,v_1,f_0(v_0)) **where** island(smart,standard,10,25,200)
  isolate(true)
  f_0(ξ,inv,cos,sin,tan,tanh,sqroot,square,cube,quart,exp,ln)
  onscore(0.0,600,regress(1.0/f_0(c_0,\$$v_1$\$),\$$f_0$\$(\$$v_0$\$)/f_0(c_0,\$$v_1$\$))
  where
  island(smart,standard,10,25,200)
  f_0(+,-,rsub)
  c_0(1.0/\$w0\$))

Not only is there is a big difference between (E12.3), (E12.9), (E12.27), and (S20); but, the search goal in (S20) contains the term ***y*** which is the dependent variable in the regression. While this may seem invalid in the domain of any basic regression course taught in university, we will show below why it is perfectly valid in this context.

First, the ***isolate(true)*** clause in (S20) keeps any of the champions from the first **where** clause from reaching the final solution list. While there is nothing wrong with using ***y*** during training (***y*** is used repeatedly during fitness calculation while training). Allowing ***y*** to appear in the final solution causes problems because ***y*** will not be available during testing. So only solutions containing the features x_0 thru x_{99} are appropriate for final champions.

Second, the ***onscore*** clause erases all champions except the top champion and proceeds to convert the top champion into the form specified in the ***onscore*** goal and **where** clauses. The ***onscore*** clause is triggered when the first search achieves a fitness score of 0.0 or when the number of training generations reaches 600. This initiates a completely new search for *regress(1.0/f_0(c_0,\$$v_1$\$),\$$f_0$\$(\$$v_0$\$)/f_0(c_0,\$$v_1$\$))* which does not contain the inappropriate ***y*** term. Therefore the term ***y*** is used only during training and setup but never in the final solution.

In order to understand how the two cascading searches in (S20) relate to the three difficult cases (E12.3), (E12.9), and (E12.27), we must observe the following regression equivalence chains in the situation where there is zero noise.

- (*E12.9*) **regress(f_0(v_0)/(c_0-v_1))** $->$
- (*E12.9.1*) y = a0+b0(f_0(v_0)/(c_0-v_1)) $->$
- (*E12.9.2*) y(c_0-v_1) = a0(c_0-v_1) + b0f_0(v_0) $->$
- (*E12.9.3*) c_0y - v_1y = a0(c_0-v_1) + b0f_0(v_0) $->$
- (*E12.9.4*) c_0y = a0(c_0-v_1) + b0f_0(v_0) + v_1y $->$
- (*E12.9.5*) y = (a0(c_0-v_1)/c_0) + (b0f_0(v_0)/c_0) + (v_1y/c_0) $->$
- (*E12.9.6*) y = a0 - (a0/c_0)*v_1 + (b0/c_0)*f_0(v_0) + (1/c_0)*v_1y $->$
- (*E12.9.7*) **regress(v_1*y,v_1,f_0(v_0)))** $->$
- (*E12.9.8*) y = a1 + w0*v_1y - w1v_1 + w2*f_0(v_0) $->$
- (*E12.9.9*) w0=(1/c_0) $->$

- (*E12.9.10*) **c_0=(1/w0)** $->$
- (*E12.9.11*) regress(f_0(v_0)/((1/w0)+v_1)) $->$
- (*E12.9.12*) **regress(f_0(v_0)/(c_0+v_1))**

Similar equivalence chains are true for (E12.3) and (E12.27). Taken all three together, we see that the answers to (E12.3), (E12.9), and (E12.27) will be either *regress(f_0(v_0)/((1/w0)+v_1))*, *regress(f_0(v_0)/((1/w0)-v_1))*, or *regress(f_0(v_0)/(v_1-(1/w0)))* which is exactly what search island (S20) proposes.

Let's use this actual example, suppose the target formula is *y = .913 + (62.13*ln(x_2)/(x_4-23.451))*. The first search in (S20) *regress(v_1*y,v_1,f_0(v_0)))* discovers that the champion *y = 0.913-(0.039*x_4)+(0.0426421*(x_4*y)-(2.65*ln(x_2)));* achieves a fitness score of 0.0. This low fitness score triggers the **onscore** clause (otherwise the onscore clause would be triggered by more than 600 generations passing). The **onscore** clause substitutes for the items enclosed in $ sign pairs and searches for the following goal *regress(1.0/f_0(c_0,x_4),ln(x_2)/f_0(c_0,x_4)) where f_0(+,-,rsub) c_0(23.451)* (**since (1/.0426421) = 23.451**). The final answer is *y = 0.913+(62.13*(ln(x_2)/(x_4-23.451)))* with a final fitness score of 0.0.

The search space size, for island (S20), is 100*100*12 = 120,000. At 200 serial iterations per generation, this search will require a maximum of 600 generations. On our test machine, each generation requires .0003hrs. So the maximum time required for this search island to complete is 0.18hrs and is followed immediately by a brief search for the onscore goal.

Most often the evolutionary search finds the correct answer in far less time. For instance, even in the case of this difficult target *y = .913 + (62.13*ln(x_2)/(x_4-23.451))*, the evolutionary search normally finds the target in less than a half of the maximum serial time.

12 Search Island S21

The RQL search command covering the space f_1(f_0(c_0,v_0),f_2(v_1,v_2)) in U_2 allows the algorithm to carve out a large space where evolutionary search and serial search methods are both intractable. The formal RQL search command is.

- (*E13*) **search** regress(f_1(f_0(c_0,v_0),f_2(v_1,v_2))) **where** island(smart,standard,10,25,200)
  f_0(+,*,/)
  f_1(+,-,*,/,rsub,rdiv)
  f_2(+,-,*,/)

If we are to try E13 by evolutionary search we run into trouble with test problems such as *y = (v_1*v_2)/(c_0+v_0)*. If we try serial search we see that the size of this space is 100*100*100*3*6*4*2^{18} = 18.87T which at 200 iterations per generation will require 94,371,840,000 generations. On our test computer each generation requires .00021hrs. So we will finish testing all possible serial combinations in approximately 19,818,086hrs = 825,753days = 2,256yrs.

Since searching for (E13) is not practical under any approach, we take a giant leap and search for the the following.

- (*S21*) **search** regress(v_0,v_1,v_2,v_1*v_2,v_1/v_2,1.0/v_0,v_0*v_1,v_0*v_2, v_0*v_1*v_2,(v_0*v_1)/v_2,v_1/v_0,v_2/v_0,(v_1*v_2)/v_0,v_1/(v_0*v_2), 1/(v_1+v_2),v_0/(v_1+v_2),1/(v_1-v_2),v_0/(v_1-v_2),1/(v_1*v_2),v_0/(v_1*v_2), v_2/v_1,(v_0*v_2)/v_1,1/(v_0*(v_1+v_2)),1/(v_0*v_1*v_2),v_2/(v_0*v_1)) **where** island(smart,standard,10,25,200) onfinal(regress($poly$))

Of course there is a big difference between (E13) and (S21). Search (E13) performs single regression; while, search (S21) performs multiple regressions where the variables v_0, v_1, and v_2 contain features from the set x_0 thru x_{99} (*the* **onfinal** *clause simply eliminates all zero or near zero coefficient terms from the final answer and converts to simpler form if possible*). Showing/Learning how these two searches relate to each other will require a bit of simple *regression math* and our close attention.

First, notice that search (E13) can be expanded into the following 72 single regression cases with the following equivalent regressions. Notice that all except the three *bolded* cases can be expanded into equivalent simpler regressions.

- (*E13.1*) regress((c_0+v_0)+(v_1+v_2)) = regress(v_0,v_1,v_2)
- (*E13.2*) regress((c_0+v_0)+(v_1-v_2)) = regress(v_0,v_1,v_2)
- (*E13.3*) regress((c_0+v_0)+(v_1*v_2)) = regress(v_0,v_1*v_2)
- (*E13.4*) regress((c_0+v_0)+(v_1/v_2)) = regress(v_0,v_1/v_2)
- (*E13.5*) regress((c_0+v_0)-(v_1+v_2)) = regress(v_0,v_1,v_2)
- (*E13.6*) regress((c_0+v_0)-(v_1-v_2)) = regress(v_0,v_1,v_2)
- (*E13.7*) regress((c_0+v_0)-(v_1*v_2)) = regress(v_0,v_1*v_2)
- (*E13.8*) regress((c_0+v_0)-(v_1/v_2)) = regress(v_0,v_1/v_2)
- (*E13.9*) regress((c_0+v_0)*(v_1+v_2)) = regress(v_1,v_2,v_0*v_1,v_0*v_2)
- (*E13.10*) regress((c_0+v_0)*(v_1-v_2)) = regress(v_1,v_2,v_0*v_1,v_0*v_2)
- (*E13.11*) regress((c_0+v_0)*(v_1*v_2)) = regress(v_1*v_2,v_0*v_1*v_2)
- (*E13.12*) regress((c_0+v_0)*(v_1/v_2)) = regress(v_1/v_2,(v_0*v_1)/v_2)
- (*E13.13*) regress((c_0+v_0)/(v_1+v_2)) = regress(1/(v_1+v_2),v_0/(v_1+v_2))
- (*E13.14*) regress((c_0+v_0)/(v_1-v_2)) = regress(1/(v_1+v_2),v_0/(v_1+v_2))
- (*E13.15*) regress((c_0+v_0)/(v_1*v_2)) = regress(1/(v_1*v_2),v_0/(v_1*v_2))
- (*E13.16*) regress((c_0+v_0)/(v_1/v_2)) = regress(v_2/v_1,(v_0*v_2)/v_1)
- (*E13.17*) regress((v_1+v_2)-(c_0+v_0)) = regress(v_0,v_1,v_2)
- (*E13.18*) regress((v_1-v_2)-(c_0+v_0)) = regress(v_0,v_1,v_2)
- (*E13.19*) regress((v_1*v_2)-(c_0+v_0)) = regress(v_0,v_1*v_2)
- (*E13.20*) regress((v_1/v_2)-(c_0+v_0)) = regress(v_0,v_1/v_2)
- (*E13.21*) **regress((v_1+v_2)/(c_0+v_0)) = regress((v_1+v_2)/(c_0+v_0))**
- (*E13.22*) **regress((v_1-v_2)/(c_0+v_0)) = regress((v_1-v_2)/(c_0+v_0))**

- (*E13.23*) **regress((v_1*v_2)/(c_0+v_0)) = regress((v_1*v_2)/(c_0+v_0))**
- (*E13.24*) **regress((v_1/v_2)/(c_0+v_0)) = regress((v_1/v_2)/(c_0+v_0))**
- (*E13.25*) regress((c_0*v_0)+(v_1+v_2)) = regress(v_0,v_1,v_2)
- (*E13.26*) regress((c_0*v_0)+(v_1-v_2)) = regress(v_0,v_1,v_2)
- (*E13.27*) regress((c_0*v_0)+(v_1*v_2)) = regress(v_0,v_1*v_2)
- (*E13.28*) regress((c_0*v_0)+(v_1/v_2)) = regress(v_0,v_1/v_2)
- (*E13.29*) regress((c_0*v_0)-(v_1+v_2)) = regress(v_0,v_1,v_2)
- (*E13.30*) regress((c_0*v_0)-(v_1-v_2)) = regress(v_0,v_1,v_2)
- (*E13.31*) regress((c_0*v_0)-(v_1*v_2)) = regress(v_0,v_1*v_2)
- (*E13.32*) regress((c_0*v_0)-(v_1/v_2)) = regress(v_0,v_1/v_2)
- (*E13.33*) regress((c_0*v_0)*(v_1+v_2)) = regress(v_0*v_1,v_0*v_2)
- (*E13.34*) regress((c_0*v_0)*(v_1-v_2)) = regress(v_0*v_1,v_0*v_2)
- (*E13.35*) regress((c_0*v_0)*(v_1*v_2)) = regress(v_0*v_1*v_2)
- (*E13.36*) regress((c_0*v_0)*(v_1/v_2)) = regress((v_0*v_1)/v_2)
- (*E13.37*) regress((c_0*v_0)/(v_1+v_2)) = regress(v_0/(v_1+v_2))
- (*E13.38*) regress((c_0*v_0)/(v_1-v_2)) = regress(v_0/(v_1+v_2))
- (*E13.39*) regress((c_0*v_0)/(v_1*v_2)) = regress(v_0/(v_1*v_2))
- (*E13.40*) regress((c_0*v_0)/(v_1/v_2)) = regress((v_0*v_2)/v_1)
- (*E13.41*) regress((v_1+v_2)-(c_0*v_0)) = regress(v_0,v_1,v_2)
- (*E13.42*) regress((v_1-v_2)-(c_0*v_0)) = regress(v_0,v_1,v_2)
- (*E13.43*) regress((v_1*v_2)-(c_0*v_0)) = regress(v_0*v_1*v_2)
- (*E13.44*) regress((v_1/v_2)-(c_0*v_0)) = regress((v_0*v_1)/v_2)
- (*E13.45*) regress((v_1+v_2)/(c_0*v_0)) = regress(v_1/v_0,v_2/v_0)
- (*E13.46*) regress((v_1-v_2)/(c_0*v_0)) = regress(v_1/v_0,v_2/v_0)
- (*E13.47*) regress((v_1*v_2)/(c_0*v_0)) = regress((v_1*v_2)/v_0)
- (*E13.48*) regress((v_1/v_2)/(c_0*v_0)) = regress(v_1/(v_0*v_2))
- (*E13.49*) regress((c_0/v_0)+(v_1+v_2)) = regress(1/v_0,v_1,v_2)
- (*E13.50*) regress((c_0/v_0)+(v_1-v_2)) = regress(1/v_0,v_1,v_2)
- (*E13.51*) regress((c_0/v_0)+(v_1*v_2)) = regress(1/v_0,v_1*v_2)
- (*E13.52*) regress((c_0/v_0)+(v_1/v_2)) = regress(1/v_0,(v_1/v_2))
- (*E13.53*) regress((c_0/v_0)-(v_1+v_2)) = regress(1/v_0,v_1,v_2)
- (*E13.54*) regress((c_0/v_0)-(v_1-v_2)) = regress(1/v_0,v_1,v_2)
- (*E13.55*) regress((c_0/v_0)-(v_1*v_2)) = regress(1/v_0,v_1*v_2)
- (*E13.56*) regress((c_0/v_0)-(v_1/v_2)) = regress(1/v_0,(v_1/v_2))
- (*E13.57*) regress((c_0/v_0)*(v_1+v_2)) = regress(v_1/v_0,v_2/v_0)
- (*E13.58*) regress((c_0/v_0)*(v_1-v_2)) = regress(v_1/v_0,v_2/v_0)
- (*E13.59*) regress((c_0/v_0)*(v_1*v_2)) = regress(f_0(v_0),v_1)
- (*E13.60*) regress((c_0/v_0)*(v_1/v_2)) = regress(f_0(v_0),v_1)
- (*E13.61*) regress((c_0/v_0)/(v_1+v_2)) = regress(v_1/v_0,v_2/v_0)
- (*E13.62*) regress((c_0/v_0)/(v_1-v_2)) = regress(v_1/v_0,v_2/v_0)
- (*E13.63*) regress((c_0/v_0)/(v_1*v_2)) = regress(1(v_0*v_1*v_2))
- (*E13.64*) regress((c_0/v_0)/(v_1/v_2)) = regress(v_2/(v_0*v_1))
- (*E13.65*) regress((v_1+v_2)-(c_0/v_0)) = regress(1/v_0,v_1,v_2)
- (*E13.66*) regress((v_1-v_2)-(c_0/v_0)) = regress(1/v_0,v_1,v_2)
- (*E13.67*) regress((v_1*v_2)-(c_0/v_0)) = regress(1/v_0,v_1*v_2)

- (*E13.68*) regress((v_1/v_2)-(c_0/v_0)) = regress($1/v_0$,v_1/v_2)
- (*E13.69*) regress((v_1+v_2)/(c_0/v_0)) = regress(v_0*v_1,v_0*v_2)
- (*E13.70*) regress((v_1-v_2)/(c_0/v_0)) = regress(v_0*v_1,v_0*v_2)
- (*E13.71*) regress((v_1*v_2)/(c_0/v_0)) = regress($v_0*v_1*v_2$)
- (*E13.72*) regress((v_1/v_2)/(c_0/v_0)) = regress((v_0*v_1)/v_2)

Eliminating the four bolded cases and collecting all the equivalent regression terms from the right hand side of equations (E13.1) thru (E13.72) we arrive at search (S21). Addressing the bolded cases (E13.21) thru (E13.24) is more complicated and will be left to the following section.

The search space size, for island (S21), is 100*100*100 = 1,000,000. At 200 serial iterations per generation, this search will require a maximum of 5000 generations. On our test machine, each generation requires .0006hrs. So the maximum time required for this search island to complete is 3hrs.

Most often the evolutionary search finds the correct answer in far less time. For instance, even in the case of this difficult target *y = (1.57 + (2.13*(x98*x67)/(23.583-x99)))*, the evolutionary search normally finds the target in less than half of the maximum serial time.

13 Search Island S22

The previous search island (S21) addressed the RQL search command covering the space $f_1(f_0(c_0,v_0),f_2(v_1,v_2))$ see (E13); however, three more complicated regression cases (E13.21) thru (E13.24) were left to this section. If we are to try these three cases by evolutionary search we run into trouble with test problems such as $y = (v_0*v_1)/(c_0+v_2)$. If we try serial search we see that the size of each of these spaces is $100*100*100*2^{18}$ = 262.144B which at 4000 iterations per generation will require 65,536,000 generations. On our test computer each generation requires .00021hrs. So we will finish testing all possible serial combinations in approximately 41,287hrs = 1,720days = 47yrs.

Since searching for (E13.21) thru (E13.24) is not practical under any approach, we take a giant leap and search for the the following.

- (*S22*) **search** regress(v_0*y,v_0,v_1,v_2,v_1*v_2,v_1/v_2) **where** island(smart,standard,10,25,200)
  isolate(true)
  onscore(0.0,5000,regress(f_0(\$$v_1$\$,\$$v_2$\$)/f_1(c_0,\$$v_0$\$))
  where
  island(smart,standard,10,25,200)
  f_0(+,-,*,/)
  f_1(+,-,rsub)
  c_0(1.0/\$w0\$))

Clearly there is a big difference between (E13.21) thru (E13.24) and (S22), and also the search goal in (S22) contains the term ***y*** which being the dependent variable in the regression. First, the ***isolate(true)*** clause in (S22) keeps any of the champions from the first **where** clause from reaching the final solution list. While there is nothing wrong with using ***y*** during training (***y*** is used repeatedly during fitness calculation while training). Allowing ***y*** to appear in the final solution causes problems because ***y*** will not be available during testing. So only solutions containing the features x_0 thru x_{99} are appropriate for final champions.

Second, the ***onscore*** clause erases all champions except the top champion and proceeds to convert the top champion into the form specified in the ***onscore*** goal and **where** clauses. The ***onscore*** clause is triggered when the first search achieves a fitness score of 0.0 or when the number of training generations reaches 5000. This initiates a completely new search for *regress(f_0(\$$v_1$\$,\$$v_2$\$)/f_1(c_0,\$$v_0$\$))* which does not contain the inappropriate ***y*** term. Therefore the term ***y*** is used only during training and setup but never in the final solution.

In order to understand how the two cascading searches in (S22) relate to the four difficult cases (E13.21) thru (E13.24), we must observe the following regression equivalence chains in the situation where there is zero noise.

- (*E13.23*) **regress((v_1*v_2)/(c_0+v_0))** $->$
- (*E13.23.1*) y = a0+b0((v_1*v_2)/(c_0+v_0)) $->$
- (*E13.23.2*) y(c_0+v_0) = a0(c_0+v_0) + b0(v_1*v_2) $->$
- (*E13.23.3*) c_0y + v_0y = a0(c_0+v_0) + b0(v_1*v_2) $->$
- (*E13.23.4*) c_0y = a0c_0 + a0v_0 + b0(v_1*v_2) - v_0y $->$
- (*E13.23.5*) y = a0 + (a0/c_0)v_0 + (b0/c_0)(v_1*v_2) - (1/c_0)v_0y $->$
- (*E13.23.6*) **regress(v_0*y,v_0,v_1*v_2))** $->$
- (*E13.23.7*) y = a1 - w0*v_0y - w1v_0 + w2*(v_1*v_2) $->$
- (*E13.23.8*) w0=(1/c_0) $->$
- (*E13.23.9*) **c_0=(1/w0)** $->$
- (*E13.23.10*) regress((v_1*v_2)/((1/w0)+v_0)) $->$
- (*E13.23.11*) **regress((v_1*v_2)/(c_0+v_0))**

Similar equivalence chains are true for (E13.21), (E13.22) and (E13.24). Taken all three together, we see that the answers to (E13.21), (E13.22), (E13.23) and (E13.24) will be *regress(f_0(v_1,v_2)/f_1((1/w0),v_0))* where f_0(+,-,*,/) and f_1(+,-,rsub), which is exactly what search island (S22) proposes.

Let's use this actual example, suppose the target formula is *y = 1.0 + (2.0*((x_1*x_2)/(23.451+x_4))*. The first search in (S22) *regress(v_0*y,v_0,v_1,v_2,v_1*v_2,v_1/v_2))* discovers that the champion *y = y = 1-(0.0426421*(x_4*y))+(0*x_4)+(0*x_2)+(0*x_1)+(0.085289*(x_2*x_1))+(0*(x* achieves a fitness score of 0.0. This low fitness score triggers the **onscore** clause (otherwise the onscore clause would be triggered by more than 5000 generations passing). The **onscore** clause substitutes for the items enclosed in \$ sign pairs and searches for the following goal *regress(f_0(x_1,x_2)/f_1(c_0,x_4)) where f_0(+,-,*,/) f_1(+,-,rsub) c_0(23.451)* (**since (1/.0426421) = 23.451**). The final answer is *y = 1.0 + (2.0*((x_1*x_2)/(23.451+x_4)))* with a final fitness score of 0.0.

The search space size, for island (S22), is 100*100*100*12 = 720,000. At 200 serial iterations per generation, this search will require a maximum of 60,000 generations. On our test machine, each generation requires .0003hrs. So the maximum time required for this search island to complete is 18hrs and is followed immediately by a brief search for the onscore goal.

Most often the evolutionary search finds the correct answer in far less time. For instance, even in the case of this difficult target $y = 1.0 + (2.0*((x_1*x_2)/(23.451+x_4)))$, the evolutionary search normally finds the target in less than a third of the maximum serial time.

14 Search Island S23

The RQL search command covering the space $f_1(f_0(c_1,v_1),f_2(c_0,v_0))$ in U_2 allows the algorithm to carve out a large space where evolutionary search and serial search methods are both intractable. The formal RQL search command is.

- (*E14*) **search** regress($f_1(f_0(c_1,v_1),f_2(c_0,v_0))$) **where** island(smart,standard,10,25,200)
  f_0(+,-,*,/)
  f_1(+,-,*,/)
  f_2(+,-,*,/)

If we are to try E14 by evolutionary search we run into trouble with test problems such as $y = (c_1*v_1)/(c_0-v_0)$. If we try serial search we see that the size of this space is $100*100*4*4*4*2^{18}*2^{18}$ = 43.9 Quadrillion which at 200 iterations per generation will require 219,902,325,555,200 generations. On our test computer each generation requires .00021hrs. So we will finish testing all possible serial combinations in approximately 461,794,883,665hrs = 19,241,453,486days = 52,716,310yrs.

Since searching for (E14) is not practical under any approach, we take a giant leap and search for the the following.

- (*S23*) **search** regress($v_0,v_1,1.0/v_0,1.0/v_1,v_0*v_1,v_0/v_1,v_1/v_0,1/(v_0*v_1)$) **where** island(smart,standard,10,25,200)
  onfinal(regress(\$poly\$))

Of course there is a big difference between (E14) and (S23). Search (E14) performs single regression; while, search (S23) performs multiple regressions where the variables v_0, and v_1 contain features from the set x_0 thru x_{99} (*the* **onfinal** *clause simply eliminates all zero or near zero coefficient terms from the final answer and converts to simpler form if possible*). Showing/Learning how these two searches relate to each other will require a bit of simple *regression math* and our close attention.

First, notice that search (E14) can be expanded into the following 64 single regression cases with the following equivalent regressions. Notice that all except the three *bolded* cases can be expanded into equivalent simpler regressions.

- (*E14.1*) regress((c_1+v_1)+(c_0+v_0)) = regress(v_1,v_0)
- (*E14.2*) regress((c_1+v_1)+(c_0-v_0)) = regress(v_1,v_0)
- (*E14.3*) regress((c_1+v_1)+(c_0*v_0)) = regress(v_1,v_0)
- (*E14.4*) regress((c_1+v_1)+(c_0/v_0)) = regress(v_1,1/v_0)
- (*E14.5*) regress((c_1-v_1)+(c_0+v_0)) = regress(v_1,v_0)
- (*E14.6*) regress((c_1-v_1)+(c_0-v_0)) = regress(v_1,v_0)
- (*E14.7*) regress((c_1-v_1)+(c_0*v_0)) = regress(v_1,v_0)
- (*E14.8*) regress((c_1-v_1)+(c_0/v_0)) = regress(v_1,1/v_0)
- (*E14.9*) regress((c_1*v_1)+(c_0+v_0)) = regress(v_1,v_0)
- (*E14.10*) regress((c_1*v_1)+(c_0-v_0)) = regress(v_1,v_0)
- (*E14.11*) regress((c_1*v_1)+(c_0*v_0)) = regress(v_1,v_0)
- (*E14.12*) regress((c_1*v_1)+(c_0/v_0)) = regress(v_1,1/v_0)
- (*E14.13*) regress((c_1/v_1)+(c_0+v_0)) = regress(1/v_1,v_0)
- (*E14.14*) regress((c_1/v_1)+(c_0-v_0)) = regress(1/v_1,v_0)
- (*E14.15*) regress((c_1/v_1)+(c_0*v_0)) = regress(1/v_1,v_0)
- (*E14.16*) regress((c_1/v_1)+(c_0/v_0)) = regress(1/v_1,1/v_0)
- (*E14.17*) regress((c_1+v_1)-(c_0+v_0)) = regress(v_1,v_0)
- (*E14.18*) regress((c_1+v_1)-(c_0-v_0)) = regress(v_1,v_0)
- (*E14.19*) regress((c_1+v_1)-(c_0*v_0)) = regress(v_1,v_0)
- (*E14.20*) regress((c_1+v_1)-(c_0/v_0)) = regress(v_1,1/v_0)
- (*E14.21*) regress((c_1-v_1)-(c_0+v_0)) = regress(v_1,v_0)
- (*E14.22*) regress((c_1-v_1)-(c_0-v_0)) = regress(v_1,v_0)
- (*E14.23*) regress((c_1-v_1)-(c_0*v_0)) = regress(v_1,v_0)
- (*E14.24*) regress((c_1-v_1)-(c_0/v_0)) = regress(v_1,1/v_0)
- (*E14.25*) regress((c_1*v_1)-(c_0+v_0)) = regress(v_1,v_0)
- (*E14.26*) regress((c_1*v_1)-(c_0-v_0)) = regress(v_1,v_0)
- (*E14.27*) regress((c_1*v_1)-(c_0*v_0)) = regress(v_1,v_0)
- (*E14.28*) regress((c_1*v_1)-(c_0/v_0)) = regress(v_1,1/v_0)
- (*E14.29*) regress((c_1/v_1)-(c_0+v_0)) = regress(1/v_1,v_0)
- (*E14.30*) regress((c_1/v_1)-(c_0-v_0)) = regress(1/v_1,v_0)
- (*E14.31*) regress((c_1/v_1)-(c_0*v_0)) = regress(1/v_1,v_0)
- (*E14.32*) regress((c_1/v_1)-(c_0/v_0)) = regress(1/v_1,1/v_0)
- (*E14.33*) regress((c_1+v_1)*(c_0+v_0)) = regress(v_0,v_1,v_0*v_1)
- (*E14.34*) regress((c_1+v_1)*(c_0-v_0)) = regress(v_0,v_1,v_0*v_1)
- (*E14.35*) regress((c_1+v_1)*(c_0*v_0)) = regress(v_0,v_0*v_1)
- (*E14.36*) regress((c_1+v_1)*(c_0/v_0)) = regress(1/v_0,v_1/v_0)
- (*E14.37*) regress((c_1-v_1)*(c_0+v_0)) = regress(v_0,v_1,v_0*v_1)
- (*E14.38*) regress((c_1-v_1)*(c_0-v_0)) = regress(v_0,v_1,v_0*v_1)
- (*E14.39*) regress((c_1-v_1)*(c_0*v_0)) = regress(v_0,v_0*v_1)
- (*E14.40*) regress((c_1-v_1)*(c_0/v_0)) = regress(1/v_0,v_1/v_0)
- (*E14.41*) regress((c_1*v_1)*(c_0+v_0)) = regress(v_1,v_0*v_1)
- (*E14.42*) regress((c_1*v_1)*(c_0-v_0)) = regress(v_1,v_0*v_1)

- (*E14.43*) regress((c_1*v_1)*(c_0*v_0)) = regress(v_0*v_1)
- (*E14.44*) regress((c_1*v_1)*(c_0/v_0)) = regress(v_1/v_0)
- (*E14.45*) regress((c_1/v_1)*(c_0+v_0)) = regress(1/v_1,v_0/v_1)
- (*E14.46*) regress((c_1/v_1)*(c_0-v_0)) = regress(1/v_1,v_0/v_1)
- (*E14.47*) regress((c_1/v_1)*(c_0*v_0)) = regress(v_0/v_1)
- (*E14.48*) regress((c_1/v_1)*(c_0/v_0)) = regress(1/(v_0*v_1))
- (*E14.49*) **regress((c_1+v_1)/(c_0+v_0)) = regress(1/(c_0+v_0),v_1/(c_0+v_0))**
- (*E14.50*) **regress((c_1+v_1)/(c_0-v_0)) = regress(1/(c_0+v_0),v_1/(c_0+v_0))**
- (*E14.51*) regress((c_1+v_1)/(c_0*v_0)) = regress(1/v_0,v_1/v_0)
- (*E14.52*) regress((c_1+v_1)/(c_0/v_0)) = regress(v_0,v_0*v_1)
- (*E14.53*) **regress((c_1-v_1)/(c_0+v_0)) = regress(1/(c_0+v_0),v_1/(c_0+v_0))**
- (*E14.54*) **regress((c_1-v_1)/(c_0-v_0)) = regress(1/(c_0+v_0),v_1/(c_0+v_0))**
- (*E14.55*) regress((c_1-v_1)/(c_0*v_0)) = regress(1/v_0,v_1/v_0)
- (*E14.56*) regress((c_1-v_1)/(c_0/v_0)) = regress(v_0,v_0*v_1)
- (*E14.57*) **regress((c_1*v_1)/(c_0+v_0)) = regress(v_1/(c_0+v_0))**
- (*E14.58*) **regress((c_1*v_1)/(c_0-v_0)) = regress(v_1/(c_0+v_0))**
- (*E14.59*) regress((c_1*v_1)/(c_0*v_0)) = regress(v_1/v_0)
- (*E14.60*) regress((c_1*v_1)/(c_0/v_0)) = regress(v_0*v_1)
- (*E14.61*) **regress((c_1/v_1)/(c_0+v_0)) = regress(1/(v_1*(c_0+v_0)))**
- (*E14.62*) **regress((c_1/v_1)/(c_0-v_0)) = regress(1/(v_1*(c_0+v_0)))**
- (*E14.63*) regress((c_1/v_1)/(c_0*v_0)) = regress(1/v_0*v_1)
- (*E14.64*) regress((c_1/v_1)/(c_0/v_0)) = regress(v_0/v_1)

Eliminating the eight bolded cases and collecting all the equivalent regression terms from the right hand side of equations (E14.1) thru (E14.64) we arrive at search (S23). Addressing the bolded cases (E14.49), (E14.50), (E14.53), (E14.54), (E14.57), (E14.58), (E14.61), and (E14.62) is more complicated and will be left to the following section.

The search space size, for island (S23), is 100*100 = 10,000. At 200 serial iterations per generation, this search will require a maximum of 50 generations. On our test machine, each generation requires .0006hrs. So the maximum time required for this search island to complete is 0.03hrs.

Most often the evolutionary search finds the correct answer in far less time. For instance, even in the case of this difficult target *y = (1.57 + (2.13*(3.23*x67)*(23.583-x99)))*, the evolutionary search normally finds the target in less than a third of the maximum serial time.

15 Search Island S24

The previous search island (S23) addressed the RQL search command covering the space $f_1(f_0(c_1,v_1),f_2(c_0,v_0))$ see (E14); however, eight more complicated regression cases (E14.49), (E14.50), (E14.53), (E14.54), (E14.57), (E14.58), (E14.61), and (E14.62) were left to this section. If we are to try these eight cases by evolutionary search we run into trouble with test problems such as y = *(c_1*v_1)/(c_0+v_0)*. If we try serial search we see that the size of each of these spaces is 100*100*4*4*4*2^{18}*2^{18} = 43.9 Quadrillion which at 200 iterations per generation will require 219,902,325,555,200 generations. On our test computer each generation requires .00021hrs. So we will finish testing all possible serial combinations in approximately 461,794,883,665hrs = 19,241,453,486days = 52,716,310yrs.

Since searching for (E14.49), (E14.50), (E14.53), (E14.54), (E14.57), (E14.58), (E14.61), and (E14.62) is not practical under any approach, we take a giant leap and search for the the following.

- (*S24*) **search** regress(v_0*y,v_0,v_1,1/v_1) **where** island(smart,standard,10,25,200)
  isolate(true)
  onscore(0.0,50,regress(1/f_0(c_0,\$$v_0$\$),\$$v_1$\$/f_0(c_0,\$$v_0$\$),1/(\$$v_1$\$*f_0(c_0,\$$v_0$\$)))
  where
  island(10,25,200)
  f_0(+,-,rsub)
  c_0(1.0/\$w0\$)
  onfinal(regress(\$poly\$)))

Clearly there is a big difference between (E14.49), (E14.50), (E14.53), (E14.54), (E14.57), (E14.58), (E14.61), and (E14.62) and (S24), and also the search goal in (S23) contains the term ***y*** which being the dependent variable in the regression. First, the ***isolate(true)*** clause in (S24) keeps any of the champions from the first **where** clause from reaching the final solution list. While there is nothing wrong with using ***y*** during training (***y*** is used repeatedly during fitness calculation while training). Allowing ***y*** to appear in the final solution causes problems because ***y*** will not be available during testing. So only solutions containing the features x_0 thru x_{99} are appropriate for final champions.

Second, the ***onscore*** clause erases all champions except the top champion and proceeds to convert the top champion into the form specified in the ***onscore*** goal and **where** clauses. The ***onscore*** clause is triggered when the first search achieves a fitness score of 0.0 or when the number of training generations reaches 50. This initiates a completely new search for *regress(1/f_0(c_0,\$$v_0$\$),\$$v_1$\$/f_0(c_0,\$$v_0$\$),1/(\$$v_1$\$*f_0(c_0,\$$v_0$\$)))* which does not contain the inappropriate ***y*** term. Therefore the term ***y*** is used only during training and setup but never in the final solution.

In order to understand how the two cascading searches in (S24) relate to the eight difficult cases (E14.49), (E14.50), (E14.53), (E14.54), (E14.57), (E14.58), (E14.61), and (E14.62), we must observe the following regression equivalence chains in the situation where there is zero noise.

- (*E14.61*) **regress((c_1/v_1)/(c_0+v_0))** $->$
- (*E14.61.1*) y = a + b((c_1/v_1)/(c_0+v_0)) $->$
- (*E14.61.2*) y(c_0+v_0) = a(c_0+v_0) + ((bc_1)/v_1) $->$

- (*E14.61.3*) c_0y + v_0y = a(c_0+v_0) + ((bc_1)/v_1) − >
- (*E14.61.4*) c_0y = ac_0 + av_0 + ((bc_1)/v_1) - v_0y − >
- (*E14.61.5*) y = a + (a/c_0)v_0 + (bc_1/c_0)/v_1 - (1/c_0)v_0y − >
- (*E14.61.6*) **regress(v_0y,v_0,1/v_1)** − >
- (*E14.61.7*) y = a1 + w0v_0y + w1v_0 + w2/v_1 − >
- (*E14.61.8*) w0 = (1/c_0) − >
- (*E14.61.9*) **c_0=(1/w0)** − >
- (*E14.61.10*) regress(1/(v_1*((1/w0)+v_0))) − >
- (*E14.61.11*) **regress((c_1/v_1)/(c_0+v_0))**

Similar equivalence chains are true for (E14.49), (E14.50), (E14.53), (E14.54), (E14.57), (E14.58), and (E14.62). Taken all together, we see that the answers to the eight bolded cases will be *regress(1/f_0(c_0,\$$v_0$\$),\$$v_1$\$/f_0(c_0,\$$v_0$\$),1/(\$$v_1$\$*f_0(c_0,\$$v_0$\$)))* where f_0(+,-,rsub), which is exactly what search island (S24) proposes.

Let's use this actual example, suppose the target formula is *y = 1.0 + (2.0*((2.8*x_2)/(23.451+x_4))*. The first search in (S24) *regress(v_0*y,v_0,v_1,1/v_1)* discovers that the champion *y = y = 1-(0.0426421*(x_4*y))+(0*x_4)+(2.8*x_2)+(0*x_1)+(0*(1/x_2)));* achieves a fitness score of 0.0. This low fitness score triggers the **onscore** clause (otherwise the onscore clause would be triggered by more than 50 generations passing). The **onscore** clause substitutes for the items enclosed in \$ sign pairs and searches for the following goal *regress(1/f_0(c_0,x_4),x_2/f_0(c_0,x_4),1/(x_2*f_0(c_0,x_4))) where f_0(+,-,rsub) c_0(23.451)* (**since (1/.0426421) = 23.451**). The final answer is *y = 1.0 + (4.8*(x_2/(23.451+x_4)))* with a final fitness score of 0.0.

The search space size, for island (S24), is 100*100 = 10,000. At 200 serial iterations per generation, this search will require a maximum of 50 generations. On our test machine, each generation requires .0003hrs. So the maximum time required for this search island to complete is 0.015hrs and is followed immediately by a brief search for the onscore goal.

Most often the evolutionary search finds the correct answer in far less time. For instance, even in the case of this difficult target *y = 1.0 + (2.0*((2.8*x_2)/(23.451+x_4)))*, the evolutionary search normally finds the target in less than a third of the maximum serial time.

16 Accuracy Measurements

Packaging together RQL search commands from (S0) thru (S24), with searches (S16.1 thru S16.24) and (S17.1 thru S17.32) expanded for cloud deployment, we attack the 30 test problems using 81 processor units. As mentioned, each of the problems were trained and tested on 10,000 training examples with 100 features. The maximum time to complete a test problem in our cloud environment is 225 hours or 9.375 days. The results in **Table 1** demonstrate extreme accuracy on the 30 test problems.

Notice the extreme search efficiency which **Table 1** demonstrates. Our assertion is that the extreme accuracy algorithm is getting the same accuracy

Table 1. Results demonstrating extreme accuracy

Test	*WFFs*	*Train-NLSE*	*Test-NLSE*	*Extreme-NLSE*
T01	1K	0.0000	0.0000	0.0000
T02	5K	0.0000	0.0000	0.0000
T03	5K	0.0000	0.0000	0.0000
T04	5K	0.0000	0.0000	0.0000
T05	6K	0.0000	0.0000	0.0000
T06	51M	0.0000	0.0000	0.0000
T07	6K	0.0000	0.0000	0.0000
T08	2B	0.0000	0.0000	0.0000
T09	12M	0.0000	0.0000	0.0000
T10	139M	0.0000	0.0000	0.0000
T11	32M	0.0000	0.0000	0.0000
T12	6K	0.0000	0.0000	0.0000
T13	3K	0.0000	0.0000	0.0000
T14	17K	0.0000	0.0000	0.0000
T15	3M	0.0000	0.0000	0.0000
T16	1M	0.0000	0.0000	0.0255
T17	12M	0.0000	0.0000	0.0000
T18	14M	0.0000	0.0000	0.0000
T19	729K	0.0000	0.0000	0.0000
T20	22M	0.0000	0.0000	0.0000
T21	41M	0.0000	0.0000	0.0000
T22	61M	0.0000	0.0000	0.0000
T23	32M	0.0000	0.0000	0.0000
T24	2K	0.0000	0.0000	0.0000
T25	6K	0.0000	0.0000	0.0000
T26	436K	0.0000	0.0000	0.0000
T27	158K	0.0000	0.0000	0.0000
T28	2K	0.0000	0.0000	0.0000
T29	3M	0.0000	0.0000	0.0000
T30	2M	0.0000	0.0000	0.0000

(*Note1: the number of individuals evaluated before finding a solution is listed in the Well Formed Formulas (WFFs) column*) (*Note2: the fitness score of the champion on the training data is listed in the (Train-NLSE) column*) (*Note3: the fitness score of the champion on the testing data is listed in the (Test-NLSE) column*) (*Note4: the fitness score of the champion on the extreme range data is listed in the (Extreme-NLSE) column*) on $U_2(1)$ and $U_1(3)$ as if each and every single element of those sets were searched serially; and yet we are never evaluating more than a few billion candidates. Notice also the high variance in WFFs evaluated per test problem. This is the result of the random nature of evolutionary search and how much of the search burden must be carried by the serial search and mathematical treatments.

Obviously *extreme accuracy* is not the same as *absolute accuracy* and is therefore fragile under some conditions. Extreme accuracy will stop at the first estimator which achieves an NLSE of **0.0** on the training data, and *hope* that the estimator will achieve an NLSE of **.0001** or less on the testing data. Yes, an extremely accurate algorithm is guaranteed to find a perfect champion (*estimator training fitness of 0.0*) if there is one to be found; but, this perfect champion may or may not be the estimator which was used to create the testing data. For instance in the target formula *y = 1.0 + (100.0*sin(x_0)) + (.001*square(x_0))* we notice that the final term *(.0001*square(x_0))* is less significant at low ranges of x_0; but, as the absolute magnitude of x_0 increases, the final term is increasingly significant. And, this does not even cover the many issues with problematic training data ranges and poorly behaved target formulas within those ranges. For instance, creating training data in the range -1000 to 1000 for the target formula *y = 1.0 + exp(x_2*34.23)* runs into many issues where the value of **y** exceeds the range of a 64 bit IEEE real number. So as one can see the concept of *extreme acuracy* is just the beginning of the attempt to conquer the accuracy problem in SR.

In an attempt to further explore the behavior we have labeled *extreme accuracy*, An extreme training matrix of independent variables was filled with random numbers in the range [0,1]. Then an extreme testing matrix of independent variables was filled with random numbers in the range [-1,0]. The champion, which was trained on the [0,1] range, had never seen this data before and had never seen data in the range [-1,0] before. The champion's results against the extreme testing data are shown in the *Extreme-NLSE* column of **Table 1**.

It should be noted that the end user has no knowledge of RQL searches (S0) thru (S24). These searches are applied, behind the veil, when the user submits a test problem. Similarly, the end user had no knowledge of the details of the cloud deployment - nor is it necessary or desirable that the end user have such involvement.

All the extreme algorithm timings and tables of results in this paper have been performed, in a modest cloud deployment. In a modest cloud deployment, searches (S0) thru (S15), (S16.1) thru (S16.24), (S17.1) thru (S17.32), and (S18) thru (S24) are distributed across 81 processor units. On our test machine, if one allows a maximum time to complete of 9.375 days running in this modest cloud configuration, the maximum number of features which can be attempted is 100 features.

The extreme algorithm can also be delivered, in a single thread deployment, on a laptop for scientists who want to run nonlinear regression problems in the background as they perform their normal tasks. In a single thread deployment, searches (S0) thru (S24) are packaged together as a unit and run in a single process on the laptop. On our test machine, if one allows a maximum time to complete of 3.25 days running in the background, the maximum number of features which can be attempted is 25 features. If one allows a maximum time to complete of 12.5 days running in the background, the maximum number of features which can be attempted is 35 features.

The extreme algorithm can also be delivered, as a multi-thread deployment, on a workstation for scientists who want to run nonlinear regression problems in their laboratory. In a multi-thread deployment, searches (S0-S3), (S4-S15), (S16-S17), and (S18-S24) are packaged together as four units. Each unit is run on a single thread on the workstation and assigned to a single core cpu. On our test machine, if one allows a maximum time to complete of 13.02 days running on the workstation, the maximum number of features which can be attempted is 50 features.

The extreme algorithm can also be delivered, on a large cloud deployment, for scientists who want to run very large nonlinear regression problems and who have a large number of computation nodes. In a large cloud deployment, searches (S0) thru (S15), (S16.1) thru (S16.24), and (S18) thru (S24) are distributed across 49 processor units. Searches (S17.1) thru (S17.32) are further broken out, at run time, into 32 x M separate searches where $\|V\| = M$. This is done by expanding each of the searches (S17.1) thru (S17.32) into M separate searches by setting the variable v_0 into all possible M concrete values, as shown in the examples below.

- (*S17.1.1*) **search** regress($(x_0+v_1)*(v_2+v_3)$) **where** island(smart,standard,10,25,4000)
- (*S17.1.2*) **search** regress($(x_1+v_1)*(v_2+v_3)$) **where** island(smart,standard,10,25,4000)
- (*S17.1.3*) **search** regress($(x_2+v_1)*(v_2+v_3)$) **where** island(smart,standard,10,25,4000)
- (*S17.1.M*) **search** regress($(x_M+v_1)*(v_2+v_3)$) **where** island(smart,standard,10,25,4000)

On our test machine, if one allows a maximum time to complete of 11.718 days running in this large cloud configuration, the maximum number of features which can be attempted is 500 features, and one would require 16,049 = (49 + (32*500)) computation nodes. Furthermore, at 500 features, the size of the search space for which we are asserting extreme accuracy is larger than $15^7*(500+2^{18})^8 = 3.86E+051$.

17 Conclusion

In a previous paper (Korns 2011), significant accuracy issues were identified for state of the art SR systems. It is now obvious that these SR accuracy issues are due primarily to the poor surface conditions of specific subsets of the problem space. For instance, if the problem space is exceedingly choppy with little monotonicity or flat with the exception of a single point with fitness advantage, then no amount of fiddling with evolutionary parameters will address the core issue.

In this paper we lay the ground work for an enhanced algorithmic approach to SR which achieves a level of extreme accuracy. This enhanced algorithm contains a search language and an *informal argument*, suggesting a priori, that extreme accuracy will be achieved on any single isolated problem within a broad class of basic SR problems. Furthermore, maximum resource allocations and maximum timings are given for achieving extreme accuracy.

The new extreme accuracy algorithm introduces a hybrid view of SR in which advanced evolutionary methods are deployed in the extremely large spaces where serial search is impractical, and in which the intractable smaller spaces are first identified and then attacked either serially or with mathematical treatments. All academics and SR researchers are heartily invited into this newly opened *playground*, as a plethora of intellectual work awaits. Increasing SR's demonstrable range of extreme accuracy will require that new intractable subspaces be identified and that new mathematical treatments be devised.

Finally, to the extent that the reasoning in this *informal argument*, of extreme accuracy, gain academic and commercial acceptance, a climate of *belief* in SR can be created wherein SR is increasingly seen as a "**must have**" tool in the scientific arsenal.

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. Michael Korns (2012). A Baseline Symbolic Regression Algorithm, in *Genetic Programming Theory and Practice X.* Springer, New York. Kaufmann Publishers, San Francisco California.
5. 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.
6. John R Koza (1992). Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge Massachusetts.
7. John R Koza (1994). Genetic Programming II: Automatic Discovery of Reusable Programs. The MIT Press, Cambridge Massachusetts.
8. John R Koza, Forrest H Bennett III, David Andre, Martin A Keane (1999). Genetic Programming III: Darwinian Invention and Problem Solving. Morgan
9. McConaghy, Trent, (2011). FFX: Fastm Scalable, Deterministic Symbolic Regression Technology, in *Genetic Programming Theory and Practice IX.* Springer, New York.
10. J.A., Nelder, and R. W. Wedderburn (1972). Generalized linear Models, in *Journal of the Royal Statistical Society*, Series A, General, 135:370-384.
11. 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.
12. Guido Smits, and Mark Kotanchek (2005). Pareto-Front Exploitation in Symbolic Regression, in *Genetic Programming Theory and Practice II.* Springer, New York.
13. Michael Schmidt, Hod Lipson (2010). Age-Fitness Pareto Optimization, in *Genetic Programming Theory and Practice VI.* Springer, New York.