Improving Symbolic Regression with Interval Arithmetic and Linear Scaling

Maarten Keijzer

Computer Science Department, Free University Amsterdam
mkeijzer@cs.vu.nl

Abstract. The use of protected operators and squared error measures are standard approaches in symbolic regression. It will be shown that two relatively minor modifications of a symbolic regression system can result in greatly improved predictive performance and reliability of the induced expressions. To achieve this, interval arithmetic and linear scaling are used. An experimental section demonstrates the improvements on 15 symbolic regression problems.

1 Introduction

Two commonly used methods in symbolic regression are the focus of this work: the use of protected operators and of error measures. It will be shown that although protected operators avoid undefined mathematical behaviour of the function set by defining some ad-hoc behaviour at those points, the technique has severe shortcomings in the vicinity of mathematical singularities. An approach using interval arithmetic is proposed to provably induce expressions that do not contain undefined behaviour anywhere in their output range, both for training and unseen data.

Although error measures (particularly squared error measures) often satisfactorily determine the goal of a symbolic regression run, they can in many circumstances be very difficult for genetic programming. Here the use of linear scaling, prior to calculating the error measure, is examined. This scaling takes the form of a simple and efficient linear regression that is used to find the optimal slope and intercept of the expressions against the target. In effect, the use of linear scaling is a fast method to calculate two constants that otherwise would have to be found during the run of the genetic programming system. This enables the system to concentrate on the more important problem of inducing an expression that has the desired shape.

This paper demonstrates that the use of these techniques in combination provide a very effective method of performing symbolic regression that produces solutions that provably avoid undefined behaviour on both training and test set, and which performs better than a more standard approach. The runtime overhead for the two techniques is minimal: in the case of interval arithmetic a single evaluation of the tree suffices, while the linear scaling approach can be done in time linear with the number of cases. It is also demonstrates that the implementation of these techniques is, if not trivial, particularly easy to achieve in most

C. Ryan et al. (Eds.): EuroGP 2003, LNCS 2610, pp. 70–82, 2003.

tree-based genetic programming systems. Overall, the work demonstrates to the genetic programming community that it is possible to dramatically improve the reliability and performance of a symbolic regression system without resorting to exotic measures.

2 Why Protected Operators Do Not Help

In standard genetic programming, Koza identified the need for *closure* in the set of primitives: functions and terminals. Every function is defined in such a way that it can handle every possible input value. For symbolic regression, this means that some ad-hoc values need to be defined to avoid division by zero, taking the logarithm of zero or a negative number or taking the square root of a negative number[1]. For example: Koza chose to return the value 1 in the case of a division by zero to make it possible for genetic programming to synthesize constants by using x/x [10]. Likewise, exception values can be defined for other functions that have a limited range of applicability such as the logarithm and the square root. Another approach is to delete proposed solutions when they perform a mathematically unsound operation. Although this might avoid destructive operations on the training set, the practitioner still has to protect the operators when applied to unseen data. A third approach is to restrict the function set to those functions that do not have undefined behaviour, for instance the set of polynomial functions, possibly augmented with trigonometry and exponentiation. Systems based on the induction of polynomials, have been used for symbolic regression with success [6,1]. However, even though the set of polynomials can approximate any function, it does do this at the possible cost of increasing the size of the solutions[2]. By limiting the expressiveness of genetic programming one might miss out on that particular expression that combines representational clarity with low error. Especially when there are reasons to assume that particular functions are valid in the application domain, restricting oneself to polynomials might be a suboptimal choice.

Howard and Roberts [5], identified another problem with protected division. In their problem setting, the arbitrary values a division operator can induce led very quickly to a local optimum that used these values. Their solution was to sacrifice the division operator and continue with attempting to find an polynomial.

Although protection of operators at undefined values will make sure that the function that is evolved using symbolic regression will always return a numerical value on any input, it can lead to undesired phenomena in the output range of such a function. Consider for instance the functions induced by genetic programming in Figure 1. Although these functions are well-behaved on the training set, examining their range on all possible inputs reveals the use of asymptotes in

[1] Assuming that symbolic regression is performed on the set of reals instead of the set of complex numbers.

[2] As an example, consider the polynomial Taylor expansion of the exponentiation function. With every decrease in error an extra polynomial term needs to be added.

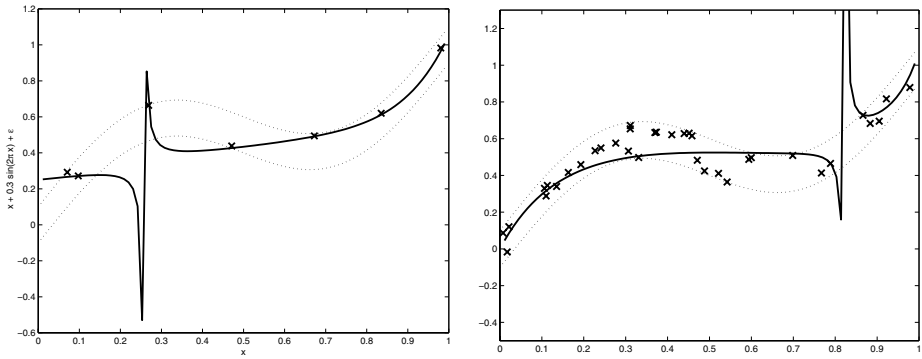

Fig. 1. Two examples where genetic programming induces an asymptote to fit the data: (a) a situation with sparse data and (b) a situation with a more dense covering. The set of training points are depicted with crosses. The function set included the protected division operator.

regions of the input space that are not covered by the training points. When using this function on unseen data, it is quite possible that a prediction will be requested for a point that lies on this asymptote. In that case the function identified by genetic programming can return arbitrarily high or low values. Note that protection only holds for the point where the actual division by zero occurs: there the function will return the ad hoc value defined by the protection rule.

3 Static Analysis through Interval Arithmetic

Interval arithmetic is a general method for calculating the bounds for an operation of arithmetic, given the bounds of the input arguments. It is often used to give reliable bounds on the value of mathematical functions calculated on hardware of finite precision, or in global optimization [2]. Given for instance that an input variable x has values in the range $[-1, 1]$, and another input variable y consists of values in the range $[0, 2]$, it is possible to deduce that the addition function $x + y$ will necessarily output values that lie in the range $[-1, 3]$. Similarly, a division by a variable that has the value 0 as a possibility, will result in infinite bounds, as division by zero is a possibility.

Sanchez used interval arithmetic to force genetic programming to output bounds as the output for cases, rather than point predictions [14] . Here we will focus on a very basic use of interval arithmetic: making sure that the mathematical function defined by a genetic programming system does not contain any undefined values. It then functions as a pre-processing step for the acceptance of a solution to be submitted to evaluation for performance. This can be viewed as a form of static analysis. For a different problem setting than symbolic regression, Johnson [8] attempts to replace evaluation over fitness cases with a more rigorous framework of static analysis techniques. A complete replacement of evaluation by static analysis does not seem to be feasible for symbolic regres-

sion due to the high dimensionality of the input space. Here static analysis in the form of interval arithmetic is used as a fast and simple test of the feasibility of solutions. To use interval arithmetic, an extra demand is placed on the definition of the terminal set, namely the theoretical range of the input variables. Many variables have such a known range: a percentage measurement is forced to lie in the interval [0,1], while a standard deviation is necessarily positive. If the theoretical range is unknown, or if it is infinite, the training data itself can be used to define these bounds. Given that each variable x is accompanied with its lower bound x_l and its upper bound x_u, all operations of arithmetic can be defined as recursive operations on the lower and upper bound of the interval. Some examples:

$$
\begin{aligned}
x + y \quad & \begin{cases} \text{lower}: \ x_l + y_l \\ \text{upper}: x_u + y_u \end{cases} \\
-x \quad & \begin{cases} \text{lower}: \ -x_u \\ \text{upper}: \ -x_l \end{cases} \\
x \times y \quad & \begin{cases} \text{lower}: \ \min(x_l y_l, x_l y_u, x_u y_l, x_u y_u) \\ \text{upper}: \max(x_l y_l, x_l y_u, x_u y_l, x_u y_u) \end{cases} \\
e^x \quad & \begin{cases} \text{lower}: \ e^{x_l} \\ \text{upper}: e^{x_u} \end{cases}
\end{aligned}
$$

And for division:

$$
1/x \qquad \begin{cases} \text{lower}: \ \min(1/x_l, 1/x_u) \\ \text{upper}: \max(1/x_l, 1/x_u) \end{cases}
$$

if $\text{sign}(x_l) = \text{sign}(x_u)$, and none of the bounds are zero.

Similar bounds can be defined for functions such as sqrt and the logarithm, whose monotonic behaviour make the definition equivalent to the application of the operators on the bounds. Checking whether the bounds fall into the well-defined area of the range is then trivial. Periodic functions such as sine and cosine take a bit more work, as the output range is defined by the exact periods that are in use in the input range[3].

By calculating the bounds recursively through a symbolic expression, possible problem with undefined values are identified at the node where the violation can occur. Here it is suggested that when this occurs, the individual is either assigned the worst possible performance value, or that it is simply deleted. As this procedure involves a single evaluation of the tree, it is a cheap method to make sure that all individuals are well-defined for all possible input values. Tree based GP systems that use a recursive or stack based approach to evaluation, could easily accommodate the evaluation of the theoretical bounds of the algorithm.

Although in mathematical notation, the implementation of interval arithmetic may seem trivial, it is found that to use the definitions straightforwardly can lead to numerically unstable results. Small roundoff errors at the start of

[3] Calculating tight bounds for these trigonometric functions is however expensive. Therefore, in the experimental section, the loose bounds of [-1,1] are always returned for the sine and cosine functions.

a big calculation can propagate in such a way that subsequent interval calculations become wrong. To counter this, a library for interval arithmetic is used that uses the underlying hardware to provably make sure that the intervals that are defined include all possible floating point numbers in the interval.

Even though the algorithm sketched above will return the theoretical range for the entire function if it is well defined, this information is not used, because the bounds are not necessarily tight. Consider for example the expression $x * x$, with bounds $x_l = -1$ and $x_u = 1$. Application of the interval arithmetic rule for multiplication will lead to an output range of $[-1, 1]$, while a tighter bound of $[0, 1]$ can be deduced. In general the effects of a multiple occurrence of a single variable in the expression, will lead to larger bounds than necessary. For this reason, it can be necessary to include the function $\mathrm{sqr}(x) = x*x$ with appropriate interval calculations next to the usual binary multiplication, to make it possible that the formula $1/(1 + \mathrm{sqr}(x))$ is not discarded when x is bounded below by a value smaller than -1.

4 Minimization of Error Can Deceive Genetic Programming

Practitioners of symbolic regression usually use absolute or squared error measures to calculate the performance of a solution. Due to the straightforward comparison of the predicted value and the target value when using error measures, a genetic programming system is forced to first get the range right before any solutions can be considered. Consider for example the two target functions $t = x^2$ and $t = 100 + x^2$. When using standard symbolic regression to find these functions, a large difference in search efficiency can be observed. Whereas the first target function is readily found (often even in the initial generation), the second target function is usually not found at all. Table 1 shows the results of experiments with these two different target functions. When floating point constants are used, genetic programming routinely converges on the average of the training data: a value of 100.37. Only in 8 cases out of the 50 runs that were performed did the particular genetic programming system find something that had a better performance than this average. The selection pressure on getting the range right is so high in this case, that the system spends most effort in finding that particular value. Once found, diversity has dropped to such a point that the additional square of the inputs is no longer found.

5 Linear Scaling

The use of linear scaling (regression) is by no means new to genetic programming. Iba, Nikolaev and others routinely use multiple linear regression for finding coefficients in polynomial regression models [6,7,12]. Hiden, McKay and other have researched many different forms of multiple linear regression for combining several expressions created by GP [11,4,3]. However, multiple linear regression is

Table 1. Number of successes of a genetic programming system on two simple problems. For each problem the system was run 50 times for 20 generations using a population size of 500 individuals. The function set consisted of simple arithmetic. The inputs consisted of 21 regularly spaced points between -1 and 1. A solution was considered a success if its mean squared error was smaller than 0.001.

Target	Success Rate
x^2	98%
$x^2 + 100$	16%

a fairly costly procedure as it involves a matrix inversion either over the covariance matrix, or over the data set itself. Another problem with multiple linear regression lies in the need to specify the number of coefficients that are used. If this number increases, the likeliness of overfitting increases as well. Topchy and Punch [16] go so far as to extend linear regression to a more general gradient search. With their approach, coefficients appearing inside non-linear function are optimized. They also use a protected operator, namely division. A full gradient descent is computationally even more expensive than multiple linear regression, therefore Topchy and Punch limit the number of gradient steps for each individual to 3 [16]. They also report that the individuals undergoing gradient search tend to increase the number of coefficients to be able to get a better performance on the training set.

Here the use of the simplest form of regression is examined: the calculation of the slope and intercept of a formula. Given that $y = \text{gp}(x)$ is the output of an expression induced by genetic programming on the input data x, a linear regression on the target values t can be performed using the equations:

$$b = \frac{\sum [(t - \bar{t})(y - \bar{y})]}{\sum [(y - \bar{y})^2]} \tag{1}$$

$$a = \bar{t} - b\bar{y} \tag{2}$$

where n is the number of cases, and $\bar{y}$ and $\bar{t}$ denote the average output and average target value respectively. These expressions calculate the *slope* and *intercept* respectively of a set of outputs y, such that the sum of squared errors between t and $a + by$ is minimized. The operations defined by Equation 1 and 2 can be done in $O(N)$ time.

After this any error measure can be calculated on the scaled formula $a + by$, for instance the mean squared error (MSE):

$$MSE(t, a + by) = \frac{1}{N} \sum_{i}^{N} (a + by - t)^2 \tag{3}$$

If a is different from 0 and b is different from 1, the procedure outlined above is guaranteed to reduce the MSE for any formula $y = \text{gp}(x)$. The cost of calculating the slope and intercept is linear in the size of the dataset. By efficiently calculating the slope and intercept for each individual, the need to

search for these two constants is removed from the genetic programming run. Genetic programming is then free to search for that expression whose *shape* is most similar to that of the target function.

To ensure numerical stability in the calculation of the slope, the variance of the outputs y is measured. If this value exceeds 10^7 or is lower than 10^{-7}, the individual gets deleted. Note that an expression that evaluates to a constant will lead to a denominator of 0 in Equation 1. In this case the expression will also get deleted.

6 Demonstration

To demonstrate the use of both proposed improvements a number of experiments are performed. To test the claim that the use of interval arithmetic removes one source of overfitting, a genetic programming system with and without interval checks is used. Two things need to be ascertained: first and foremost, it needs to be shown that the use of interval arithmetic indeed produces solutions that perform better on unseen data than solutions produced without such interval checks. Secondly, it needs to be ascertained if there is a price to pay for using interval arithmetic. It could be possible that deleting individuals that are possibly undefined, precludes a genetic programming system of performing optimally. To test the claim that the use of scaling reduces underfitting, it would be sufficient to show that it will produce better solutions on the training data. Although it seems vacuous to perform these experiments as it is a priori known that the use of scaling will reduce the training error, even for random search, it is still instructive to see the amount of improvement that can be achieved using this method. At this point, no effort is undertaken to estimate the generalization performance of the scaled and unscaled variants. The genetic programming system that is used does not use any form of regularization, and the expressions are allowed to grow to a size of 1024 nodes. Error on a separate testset is only used to estimate the likeliness of the systems to produce destructive overfitting behaviour.

These questions inspired three settings for the experiments: (a) one where both interval arithmetic and linear scaling are used; (b) one for which linear scaling is not used but interval arithmetic is; and (c) one for which linear scaling is used, but interval arithmetic is not. The use of interval arithmetic is checked by comparing the performance on both test and training set of the two systems that both use linear scaling, while the use of linear scaling is checked by comparing the training set performance of the two systems that both use interval arithmetic instead of protected operators.

Choosing a good set of problems for testing symbolic regression is difficult, especially because no established set of benchmark problems has been established. To prevent the bias inherent in an ad-hoc definition of testing functions, most problems are taken from other papers that apply or propose improvements on symbolic regression. Where possible the results produced by the improvements suggested in this paper will be compared to the results in the papers that are used. The target expressions defined below are preceded with a reference to

the originating paper. In one case there is no such reference, this is the rational function using three variables.

$$[9] : f(x) = 0.3x\sin(2\pi x) \tag{4}$$

$$[13] : f(x) = x^3 \exp^{-x} \cos(x)\sin(x)(\sin^2(x) * \cos(x) - 1) \tag{5}$$

$$[] : f(x, y, z) = \frac{30xz}{(x - 10)y^2} \tag{6}$$

$$[15] : f(x) = \sum_{i}^{x} 1/i \tag{7}$$

$$[15] : f(x) = logx \tag{8}$$

$$[15] : f(x) = \sqrt{x} \tag{9}$$

$$[15] : f(x) = \text{arcsinh}(x) \tag{10}$$

$$[15] : f(x, y) = x^y \tag{11}$$

$$[16] : f(x, y) = xy + sin((x - 1)(y - 1)) \tag{12}$$

$$[16] : f(x, y) = x^4 - x^3 + y^2/2 - y \tag{13}$$

$$[16] : f(x, y) = 6sin(x)cos(y) \tag{14}$$

$$[16] : f(x, y) = 8/(2 + x^2 + y^2) \tag{15}$$

$$[16] : f(x, y) = x^3/5 + y^3/2 - y - x \tag{16}$$

The aim of this set of experiments is to demonstrate the practical implications of the use of the two methods studied here. Being of low dimensionality does not make the problems easy however. Many of the problems above mix trigonometry with polynomials, or make the problems in other ways highly non-linear. The description of the sampling strategy and other problem specific details can be found in Table 3. It is attempted to mimic the problem setup from the originating papers as closely as possible. The genetic programming system that is used in the experiments, is a steady state algorithm, that uses subtree crossover and node mutation as its genetic operators. Parents are chosen using tournament selection. A child is created using crossover, and is subsequently mutated. The child replaces an individual that is chosen using a third independent (but inverse) tournament. Crossover points are chosen using a heuristic similar to the 90/10 rule [10] by giving the uniform node selection routine three tries to select a non-terminal. Random constants are ephemeral, but with a small twist. The system uses one byte per node, where all values not taken up by functions or variables are used as constants. After random initialization of the constants using a normal distribution with standard deviation 5, the constants are sorted. This sort order is used by node mutation to select a single increment or decrement of the index to the ephemeral constant in 50% of the cases, otherwise a uniform index mutation is used. This allows for small changes in the constant value to take place more regularly than otherwise. The parameters for the system can be found in Table 2.

Table 4 shows the amount of destructive overfitting that occurs with each of the three training setups. It is clear from the table that the approach that

Table 2. Parameter settings for the genetic programming system. Unless noted otherwise in Table 3 these settings are used for each run.

Population Size	500
Function Set	$\{ x + y, x \times y, 1/x, -x, \text{sqrt}(x) \}$
Tournament Size	5
Number of Evaluations	25,000
Maximum Genome Size	1024
Number of Runs	50

Table 3. Problem settings for the 15 problems tackled in this paper. The training and testing ranges are denoted using *[start:step:stop]* notation when the set is created using regular intervals. The notation $rnd(min, max)$ defines random (uniform) sampling in the range, while the mesh([start:step:stop]) defines regular sampling in two dimension.

Problem	Equation	range (train)	range (test)	note
1	Eq 4	x = [-1:0.1:1]	[-1:0.001:1]	
2	Eq 4	x = [-2:0.1:2]	[-2:0.001:2]	
3	Eq 4	x = [-3:0.1:3]	[-3:0.001:3]	
4	Eq 5	x = [0:0.05:10]	[0.05:0.05:10.05]	$\{\exp, \log, \sin, \cos\}$ 100000 evals
5	Eq 6	x,z = rnd(-1,1)	idem	Train: 1000 cases
		y = rnd(1,2)	idem	Test: 10000 cases
6	Eq 7	x = [1:1:50]	[1:1:120]	extrapolation
7	Eq 8	x = [1:1:100]	[1:0.1:100]	
8	Eq 9	x = [0:1:100]	[0:0.1:100]	
9	Eq 10	x = [0:1:100]	[0:0.1:100]	
10	Eq 11	x,y = rnd(0,1)	x,y = mesh([0:0.01:1])	Train: 100 cases
11	Eq 12	x,y = rnd(-3,3)	x,y = mesh([-3:0.01:3])	Train: 20 cases
12	Eq 13	x,y = rnd(-3,3)	x,y = mesh([-3:0.01:3])	Train: 20 cases
13	Eq 14	x,y = rnd(-3,3)	x,y = mesh([-3:0.01:3])	Train: 20 cases
14	Eq 15	x,y = rnd(-3,3)	x,y = mesh([-3:0.01:3])	Train: 20 cases
15	Eq 16	x,y = rnd(-3,3)	x,y = mesh([-3:0.01:3])	Train: 20 cases

Table 4. Amount of destructive overfitting over 50 runs for each of the 15 problems. For each run, the best of run result is evaluated over the test data, and the number of times the MSE exceeds 10000 is recorded.

problem	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
no interval	6	11	25			1				49	31	30	36	23	40
no scaling			2									1	15		1
interval + scaling												1	3		1

only uses protected operators is much more likely to produce expressions that generalize so badly that they are useless in prediction. Both methods that do use interval arithmetic instead of protected operators are much less susceptible to the problem of destructive overfitting. On Problem 10, genetic programming without interval checks consistently uses asymptotes to model the training data. This leads to a very poor generalization performance.

The performance over the training data for the three methods can be found in Table 5. For comparative purposes, the training error is stated in the percentage points of the Normalized Root Mean Square Error (NRMS), which is calculated as $100\% \times sqrt(N/(N-1) \times MSE)/\sigma_t$, where N is the number of cases, and σ_t

Table 5. Mean training performance of the best of run individuals produced by the three methods on 50 runs. The figures are stated in percentage points of the normalized root mean squared error, for which a value of 100 is equal to the performance of the mean target value and an error of 0 corresponds with a perfect fit. A more detailed comparison between the scaled and unscaled variant for the problems stated in boldface can be found in Figure 2.

problem	1	2	**3**	**4**	5	6	7	**8**	9	10	11	12	**13**	14	15
no interval	20	54	76	22	2	2	2	2	2	11	12	4	30	15	10
no scaling	50	78	88	46	22	49	50	56	93	54	19	4	63	91	23
interval + scaling	8	34	62	15	1	1	1	1	1	7	11	1	25	1	7

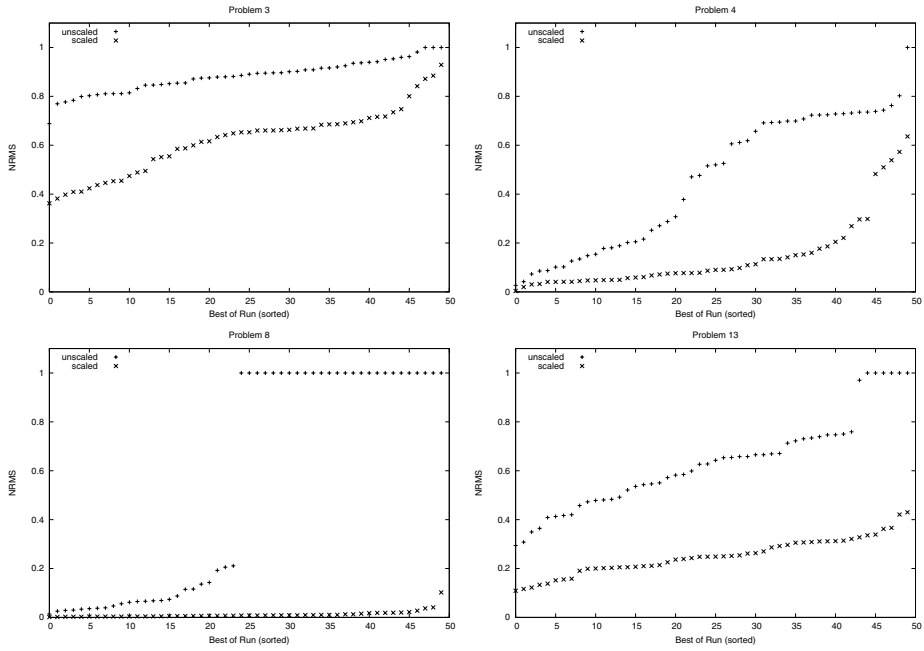

Fig. 2. Plot of the errors of the best of run individuals on the training set for 4 selected problems. Depicted are the NRMS errors for the scaled and the unscaled variant. An error of 1.0 usually means that the run has converged to a constant.

the standard deviation of the target values. This measure assigns the value 100 to an expression that performs equivalent with the mean target value. A perfect fit is obtained when the NRMS error reaches 0.

Interestingly, the comparison between the use of protected operators and interval arithmetic (that both use scaling), reveals that the exclusion of individuals that can have undefined values on unseen data actually improves the ability to fit the data. It was already observed by Howard and Roberts that the use of the division operator near the value zero is a source for premature convergence [5]. The findings in this work give additional support for this claim. The performance

Table 6. Performance difference (MSE) between standard (unscaled) GP, gradient descent GP (HGP), and scaled GP on the 5 problems taken from [16]. The first two columns of results are taken from [16] and represent the mean MSE on 10 runs, each run taking 30,000 evaluations. The results in the last two columns are performed using 50 runs, and represent the experiments performed here on 25,000 evaluations.

Problem	unscaled GP [16]	HGP [16]	unscaled GP	scaled GP
11	0.8	0.47	0.37	0.11
12	2.18	1.03	2.80	0.10
13	6.59	5.98	2.74	0.43
14	4.41	4.06	0.47	0.001
15	0.78	0.27	1.60	0.12

improvement that can be gained by using scaling is high, and in some cases (the problems taken from [15]), extreme. This is for some part caused by the ability of the unscaled genetic programming variant to rapidly converge to the mean target value as the best expression, but even then, scaled symbolic regression will find better solutions more regularly. The individual best of run results can be found in Figure 2, where the results on four problems from the four different sources are depicted. The graphs show that the performance increase by using scaling helps in general to find better solutions, and also improves the reliability in finding these solutions.

Salustowicz and Schmidhuber report that the best error achieved on problem 4 by their PIPE system was a sum of absolute errors of 1.18 using 20 runs of 100,000 evaluations in length [13]. With an equivalent effort, the scaled genetic programming system not only creates 11 individuals out of 50 runs that improve upon this figure, but the best individual practically finds the target function, in that the sum of absolute errors has dropped to 0.098 (which equates to an absolute deviation of only 0.00098 per case). Due to the random sampling employed by [16] the results on problem 11 through 15 are not directly comparable. However, in [16] a comparison is given between a standard genetic programming approach and their use of a gradient descent routine. These differences is what is used here to give an estimate of the difference between the use of gradient descent and the scaling method used here, by comparing using the unscaled genetic programming system as a baseline for comparison. The results are tabulated in Table 6. Even though the results are not directly comparable, the magnitude of the differences indicate that scaled GP performs at the very least as good as the use of gradient descent. It is however left as future work to compare simple linear scaling as is done here with more involved coefficient fitting methods as multiple linear regression and gradient descent.

7 Conclusion

The use of interval arithmetic on the theoretical range of the input values for symbolic regression, is an efficient way to induce functions that will provably avoid the use of mathematical functions at undefined values on unseen data.

This completely circumvents the need to protect mathematical operators and avoids the induction of asymptotes in areas of the input range that are not covered by the training examples. It was demonstrated that the use of protected operators do not help in avoiding the use of asymptotes. For all 15 problems tried here, the use of interval arithmetic in itself already helps in performing better.

The use of scaling prior to evaluation is an efficient way to promote solutions that have the overall shape of the target function right, but miss the appropriate scale. In effect, the scaling method is used to calculate the two linear constants for each individual in time linear with the number of cases. The improvement achieved by using this simple form of scaling is often dramatic when compared to an unscaled variant.

The experiments point out that the combination of interval arithmetic and linear scaling provides a safe and effective method of performing symbolic regression, without introducing additional parameters and also without sacrificing runtime efficiency.

References

1. J. W. Davidson, D. A. Savic, and G. A. Walters, *Method for the identification of explicit polynomial formulae for the friction in turbulent pipe flow*, Journal of Hydroinformatics 1 (1999), no. 2, 115–126.
2. Eldon Hansen, *Global optimization using interval analysis*, Dekker, New York, 1992.
3. Hugo Hiden, Ben McKay, Mark Willis, and Gary Montague, *Non-linear partial least squares using genetic programming*, Genetic Programming 1998: Proceedings of the Third Annual Conference (University of Wisconsin, Madison, Wisconsin, USA) (John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, eds.), Morgan Kaufmann, 22-25 July 1998, pp. 128–133.
4. Hugo Hiden, Mark Willis, Ming Tham, Paul Turner, and Gary Montague, *Non-linear principal components analysis using genetic programming*, Second International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA (University of Strathclyde, Glasgow, UK) (Ali Zalzala, ed.), Institution of Electrical Engineers, 1-4 September 1997.
5. Daniel Howard and Simon C. Roberts, *Genetic programming solution of the convection-diffusion equation*, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001) (San Francisco, California, USA) (Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, eds.), Morgan Kaufmann, 7-11 July 2001, pp. 34–41.
6. Hitoshi Iba, Hugo de Garis, and Taisuke Sato, *Genetic programming using a minimum description length principle*, Advances in Genetic Programming (Kenneth E. Kinnear, Jr., ed.), MIT Press, 1994, pp. 265–284.
7. Hitoshi Iba and Nikolay Nikolaev, *Genetic programming polynomial models of financial data series*, Proceedings of the 2000 Congress on Evolutionary Computation CEC00 (La Jolla Marriott Hotel La Jolla, California, USA), IEEE Press, 6-9 July 2000, pp. 1459–1466.

8. Colin Johnson, *Deriving genetic programming fitness properties by static analysis*, Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002 (Kinsale, Ireland) (James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea G. B. Tettamanzi, eds.), LNCS, vol. 2278, Springer-Verlag, 3-5 April 2002, pp. 298–307.
9. Maarten Keijzer and Vladan Babovic, *Genetic programming, ensemble methods and the bias/variance tradeoff - introductory investigations*, Genetic Programming, Proceedings of EuroGP'2000 (Edinburgh) (Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, eds.), LNCS, vol. 1802, Springer-Verlag, 15-16 April 2000, pp. 76–90.
10. John R. Koza, *Genetic programming: On the programming of computers by means of natural selection*, MIT Press, Cambridge, MA, USA, 1992.
11. Ben McKay, Mark Willis, Dominic Searson, and Gary Montague, *Non-linear continuum regression using genetic programming*, Proceedings of the Genetic and Evolutionary Computation Conference (Orlando, Florida, USA) (Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, eds.), vol. 2, Morgan Kaufmann, 13-17 July 1999, pp. 1106–1111.
12. Nikolay Y. Nikolaev and Hitoshi Iba, *Regularization approach to inductive genetic programming*, IEEE Transactions on Evolutionary Computing 54 (2001), no. 4, 359–375.
13. R. P. Salustowicz and J. Schmidhuber, *Probabilistic incremental program evolution*, Evolutionary Computation 5 (1997), no. 2, 123–141.
14. Luciano Sanchez, *Interval-valued GA-P algorithms*, IEEE Transactions on Evolutionary Computation 4 (2000), no. 1, 64–72.
15. Matthew Streeter and Lee A. Becker, *Automated discovery of numerical approximation formulae via genetic programming*, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001) (San Francisco, California, USA) (Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, eds.), Morgan Kaufmann, 7-11 July 2001, pp. 147–154.
16. Alexander Topchy and W. F. Punch, *Faster genetic programming based on local gradient search of numeric leaf values*, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001) (San Francisco, California, USA) (Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, eds.), Morgan Kaufmann, 7-11 July 2001, pp. 155–162.