# Robust Symbolic Regression with Affine Arithmetic

Cassio L. Pennachin
Universidade Federal de Minas Gerais
Av. Antônio Carlos, 6627
Belo Horizonte, MG, Brazil
pennachin@ufmg.br

Moshe Looks
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043
madscience@google.com

João A. de Vasconcelos
Universidade Federal de Minas Gerais
Av. Antônio Carlos, 6627
Belo Horizonte, MG, Brazil
joao@cpdee.ufmg.br

## ABSTRACT

We use affine arithmetic to improve both the performance and the robustness of genetic programming for symbolic regression. During evolution, we use affine arithmetic to analyze expressions generated by the genetic operators, estimating their output range given the ranges of their inputs over the training data. These estimated output ranges allow us to discard trees that contain asymptotes as well as those whose output is too far from the desired output range determined by the training instances. We also perform linear scaling of outputs before fitness evaluation. Experiments are performed on 15 problems, comparing the proposed system with a baseline genetic programming system with protected operators, and with a similar system based on interval arithmetic. Results show that integrating affine arithmetic with an implementation of standard genetic programming reduces the number of fitness evaluations during training and improves generalization performance, minimizes overfitting, and completely avoids extreme errors on unseen test data. **Track: Genetic Programming**.

## Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program synthesis*

## General Terms

Algorithms, Design, Experimentation

## Keywords

Symbolic regression, affine arithmetic, robustness

## 1. MOTIVATION

Practical applicability of symbolic regression depends on avoiding overfitting to training data. When overfitting cannot be eliminated, we must at least minimize its probability and extent. A particularly damaging kind of overfitting is the error introduced when genetic programming induces expressions containing asymptotes on points in parameter space that are not part of the training data. Out of sample evaluation of points lying near an asymptote can result in extremely high errors, which makes prevention of this form of overfitting especially important.

Interval methods, also known as self-validated numerics, are techniques for numerical computation in which approximate results are produced with guaranteed ranges, or error bounds. They can be used to analyze expressions, such as the ones defined by program trees in a symbolic regression evolution run, and will produce output bounds for those expressions. If a program tree includes asymptotes, interval methods will detect them. This process of static analysis is used in [5] to significantly improve the generalization performance of symbolic regression, by assigning infinite error to program trees with potential asymptotes.

Since interval methods provide guaranteed bounds on the output range of a program tree, another potential use is the detection of well-formed trees which, despite having finite bounds, are guaranteed to have bad fitness. A desirable output range can be estimated from the training data, and trees whose output falls outside the desirable range can be eliminated without complete fitness evaluation. Because GP applied to symbolic regression problems typically generates many trees with very bad fitness [9], this kind of elimination can significantly boost performance. Unfortunately, the intervals output by interval arithmetic, the simplest such method and the one used in [5], tend to be too wide to be useful for this purpose.

We propose a technique similar to [5], but based on affine arithmetic, a more refined interval method that generates tighter bounds for the expressions being evolved. We hypothesize that the intervals generated by affine arithmetic are tight enough to be useful for fitness estimation as well as asymptote detection, and that symbolic regression with affine arithmetic can successfully prevent extreme overfitting. Furthermore, some programs incorrectly rejected by interval arithmetic as having potential asymptotes will be kept by our method; this can lead to greater flexibility for the practitioner, as we shall see.

## 2. SYMBOLIC REGRESSION WITH INTERVAL ARITHMETIC

The simplest and most widely adopted interval method is interval arithmetic (IA) [12]. In IA, a quantity $x$ is represented by an *interval* $\bar{x} = [\bar{x}_L, \bar{x}_U]$, where $\bar{x}_L, \bar{x}_U \in \Re$ are the interval bounds, and $\bar{x}_L \leq \bar{x}_U$. Arithmetic operations


*GECCO'10,* July 7–11, 2010, Portland, Oregon, USA.

are then based on the interval bounds. For instance:

$$\begin{aligned}
\bar{x} + \bar{y} &= [\bar{x}_L + \bar{y}_L, \bar{x}_U + \bar{y}_U] \\
\bar{x} - \bar{y} &= [\bar{x}_L - \bar{y}_U, \bar{x}_U - \bar{y}_L] \\
\bar{x} \times \bar{y} &= [min(\bar{x}_L\bar{y}_L, \bar{x}_L\bar{y}_U, \bar{x}_U\bar{y}_L, \bar{x}_U\bar{y}_U), \\
&\qquad max(\bar{x}_L\bar{y}_L, \bar{x}_L\bar{y}_U, \bar{x}_U\bar{y}_L, \bar{x}_U\bar{y}_U)] \\
e^{\bar{x}} &= [e^{\bar{x}_L}, e^{\bar{x}_U}]
\end{aligned}$$

A program tree can be analyzed by recursively applying the elementary operations of interval arithmetic, given interval representations of its inputs. For symbolic regression, these intervals can be estimated from the training data, and this analysis process will generate guaranteed bounds for the program output over this region of parameter space.

Keijzer [5] used linear scaling and interval arithmetic to improve the robustness of symbolic regression. Interval arithmetic is used for analysis of program trees, and all trees that compute functions containing asymptotes (within the region of parameter-space defined by the training data) are eliminated from the population, as they are sure to generate infinite intervals. Note, however, that due to the conservative nature of IA, some harmless expressions that do not compute functions with asymptotes may nonetheless be eliminated. For example, given $\bar{x} = [-2, 2]$, IA gives an infinite interval for the harmless expression $1/(1 + x * x)$ because the denominator straddles zero.

In his experiments with benchmark functions, the combination of interval arithmetic and linear scaling greatly reduces overfitting while at the same time improving training fitness. His technique has subsequently been applied to a real-world problem, the estimation of phytoplankton concentration in oceans from satellite spectrometer measurements [14]. In these experiments, the use of interval arithmetic imposed a small penalty in training fitness, while successfully preventing extreme overfitting in validation data. Use of scaling and other techniques without interval arithmetic always led to extreme overfitting in some runs. Interval arithmetic is used to eliminate trees with infinite intervals by [7] as well, in conjunction with model ensembles and measures to maximize the diversity of those ensembles.

The major downside of IA is that sequences of operations tend to accumulate and magnify errors[1] that can end up being quite large relative to the input ranges and exact forms of the expressions being computed. As a pathological case, consider $\bar{x} - \bar{x}$, where $\bar{x} = [-1, 1]$. The resulting interval is $[-2, 2]$, while clearly the correct value is $[0, 0]$. Repeated application of error-inducing operations during expression analysis tends to result in error propagation and magnification. Consequently, the output bound of a program tree is considered too wide to be useful in detecting trees outside the desirable output range [5].

## 3. AFFINE ARITHMETIC

Several extensions and improvements to interval arithmetic have been proposed. Affine arithmetic (AA) is an interval method originally developed for computer graphics [1], which has found diverse applications, including robust optimization [2, 3]. Affine arithmetic keeps track of correlations between quantities, in addition to the ranges of each quantity. These correlations lead to significantly reduced approximation errors, especially over long computation chains that would lead to error explosion under IA. We now provide a brief overview of affine arithmetic, recommending the above references for further details.

In AA, a quantity $x$ is represented by an *affine form* $\hat{x}$, which is a first-degree polynomial:

$$\hat{x} = x_0 + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n$$

where $x_0$ is called the central value of $\hat{x}$, the $x_i$ are finite numbers, called partial deviations, and the $\varepsilon_i$ are called *noise symbols*, whose values are unknown but assuredly lie in the interval $\mathbb{U} = [-1, 1]$. Each noise symbol corresponds to one component of the overall uncertainty as to the actual value of $x$. This uncertainty may be intrinsic to the original data, or introduced by approximations in the computation of $\hat{x}$.

The fundamental invariant of affine arithmetic ensures that there is always a single assignment of values to each $\varepsilon_i$ that makes the value of every affine form $\hat{x}$ equal to the true value of the corresponding $x$. All of these values assigned to $\varepsilon_i$s are of course constrained to lie in the interval $\mathbb{U}$.

Correlation is accounted for in AA by the fact that the same symbol $\varepsilon_i$ can be part of multiple affine forms created by the evaluation of an expression. Shared noise symbols indicate dependencies between the underlying quantities. The presence of such shared symbols allows one to determine joint ranges for affine forms that are tighter than the corresponding intervals of interval arithmetic.

Computations with AA involve defining, for each operation, a corresponding procedure that operates on affine forms and produces a resulting affine form. Therefore, an elementary function $z \leftarrow f(x, y)$ is implemented by a procedure $\hat{x} \leftarrow \hat{f}(\hat{x}, \hat{y})$, such that given

$$\begin{aligned}
\hat{x} &= x_0 + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n \\
\hat{y} &= y_0 + y_1\varepsilon_1 + \cdots + y_n\varepsilon_n
\end{aligned}$$

$\hat{f}$ produces some

$$\hat{z} = z_0 + z_1\varepsilon_1 + \cdots + z_m\varepsilon_m$$

for $m \geq n$, preserving as much information as possible about the constraints embedded in the noise symbols that have non-zero coefficients.

When $f$ is itself an affine function of its arguments, $\hat{z}$ can be obtained by simple rearrangement of the noise symbols. This gives us addition, subtraction, and negation. Formally, for any given $\alpha, \beta, \zeta \in \Re$, $z \leftarrow \alpha x + \beta y + \zeta$:

$$\hat{z} = (\alpha x_0 + \beta y_0 + \zeta) + (\alpha x_1 + \beta y_1)\varepsilon_1 + \cdots + (\alpha x_n + \beta y_n)\varepsilon_n$$

With the exception of rounding errors, the above formula captures all the information available about $x$, $y$, and $z$, so it will introduce almost no error in the uncertainty estimates. Accordingly, the above formula will produce tight bounds for operations such as $\hat{x} - \hat{x}$, and identities such as $(\hat{x} + \hat{y}) - \hat{y}$ hold according to this formula, unlike in interval arithmetic. This rule of combination can be trivially extended to any number of inputs.

In the case of $z \leftarrow f(x, y)$ when $f$ is not an affine function, $z$ is given by $f^*(\varepsilon_1, \ldots, \varepsilon_n)$, where $f^*$ is a non-affine function from $\mathbb{U}^n$ to $\Re$. This is the case for important functions such

[1]This means that, as in the example given above, the computed bounds may be wider than necessary. Since IA bounds are guaranteed to contain the actual value of an expression over the given input ranges, they can never be too narrow.

as multiplication, division, exponentiation, logarithm, and others.

In this case, $z$ cannot be expressed exactly as an affine combination of the noise symbols. Instead, we need to find some affine function

$$f^a(\varepsilon_1, \ldots, \varepsilon_n) = z_0 + z_1\varepsilon_1 + \cdots + z_n\varepsilon_n$$

which is a reasonable approximation of $f^*$ over $\mathbb{U}^n$, and then introduce an extra term $z_k\varepsilon_k$ that represents the approximation error incurred when using $f^a$. The noise symbol $\varepsilon_k$ has to be exclusive to this operation; it cannot be shared with other affine forms. The magnitude of its coefficient $z_k$ must be an upper bound on the approximation error, i.e.:

$$|z_k| \geq max\{|f^*(\varepsilon_1, \ldots, \varepsilon_n) - f^a(\varepsilon_1, \ldots, \varepsilon_n)|, \varepsilon_1, \ldots, \varepsilon_n \in \mathbb{U}\}$$

Introducing the error term $z_k\varepsilon_k$ necessarily leads to loss of information, as the noise symbol $\varepsilon_k$ is a function of $\varepsilon_1, \ldots, \varepsilon_n$, but this dependency information is not recorded. Subsequent operations will therefore assume independence between $\varepsilon_k$ and $\varepsilon_1, \ldots, \varepsilon_n$, which may introduce error.

The choice of the affine approximation $f^a$ is an implementation decision, and there are many possibilities. For purposes of simplicity and efficiency, one can consider approximations that are affine combinations of the inputs $\hat{x}$ and $\hat{y}$, so $f^a(\varepsilon_1, \ldots, \varepsilon_n) = \alpha\hat{x} + \beta\hat{y} + \zeta$. Using affine approximations gives us only $k + 1$ parameters to estimate for functions of $k$ inputs.

These parameters have to be chosen in order to minimize the approximation error term $|z_k|$. The best choice, therefore, is arguably the one that minimizes the maximum value of $|\alpha\hat{x} + \beta\hat{y} + \zeta - f(x, y)|$ over the joint range $\langle\hat{x}, \hat{y}\rangle$. This is called the *Chebychev or minmax affine approximation* of $f$ over $\langle\hat{x}, \hat{y}\rangle$; it guarantees that the volume of the joint range is minimized. This, however, does not always minimizes the range for $\hat{z}$ alone. A minimum range approximation will do that. Also, the Chebychev approximation may introduce undesirable values in the range of $\hat{z}$, such as negative values for the exponential function, which are avoided by the minimum range approximation. The choice between the Chebychev approximation and the minimum range approximation is application dependent, although the latter is usually easier to compute.

One does not necessary have or want to use the best affine approximation, however. As an example, we consider the case of multiplication. Given two affine forms $\hat{x}$ and $\hat{y}$:

$$\begin{aligned}\hat{x} \cdot \hat{y} &= \left(x_0 + \sum_{i=1}^{n} x_i\varepsilon_i\right) \cdot \left(y_0 + \sum_{i=1}^{n} y_i\varepsilon_i\right) \\ &= x_0y_0 + \sum_{i=1}^{n}(x_0y_i + y_0x_i)\varepsilon_i + \sum_{i=1}^{n} x_i\varepsilon_i \cdot \sum_{i=1}^{n} y_i\varepsilon_i\end{aligned}$$

An approximation to this function will consist of the affine terms from the above expansion plus an approximation to the quadratic term. Accordingly:

$$\hat{x} \cdot \hat{y} = A(\varepsilon_1, \ldots, \varepsilon_n) + Q(\varepsilon_1, \ldots, \varepsilon_n)$$

where

$$A(\varepsilon_1, \ldots, \varepsilon_n) = x_0y_0 + \sum_{i=1}^{n}(x_0y_i + y_0x_i)\varepsilon_i$$

$$Q(\varepsilon_1, \ldots, \varepsilon_n) = \sum_{i=1}^{n} x_i\varepsilon_i \cdot \sum_{i=1}^{n} y_i\varepsilon_i$$

While computing the best approximation to $Q(\varepsilon_1, \ldots, \varepsilon_n)$ is expensive, a simple and efficient heuristic is to replace $Q(\varepsilon_1, \ldots, \varepsilon_n)$ by a single error term $z_k\varepsilon_k$, where $\varepsilon_k$ is not shared with other forms in the computation and

$$|z_k| = \sum_{i=1}^{n} |x_i| \cdot \sum_{i=1}^{n} |y_i| \geq \left|\sum_{i=1}^{n} x_i\varepsilon_i \cdot \sum_{i=1}^{n} y_i\varepsilon_i\right|, \varepsilon_i \in \mathbb{U}.$$

This heuristic introduces an error at most four times greater than the best affine approximation [2].

Detailed presentations of the approximations for a number of non-affine operations are provided in [2], including pseudo-code and implementation discussions. For our work, we implement an affine arithmetic module in Common Lisp. This code is based on the techniques of previous implementations in C [13] (basic operations, exponentiation, and logarithm) and C++ [4] (least-squares method for trigonometric functions), as well as two improvements that are relevant for its application to genetic programming.

Firstly, we utilize a single "anonymous" error term per affine form that accounts for all non-affine error introduced in the source of a calculation, and is always non-negative. When two forms are combined via an affine operation, the magnitudes of their anonymous errors are added together to obtain the term for the new form. Likewise, when a non-affine operation is performed, the magnitude of what would otherwise be a new error term is simply added to the existing anonymous error term. Since the minima and maxima of our computations remain constant, the AA bounds are still valid. Thus, the number of terms required for our affine forms may be bounded by the number of problem parameters, independent of the size of our expressions. This is an important invariant in the context of GP, where expressions may be quite large.

Affine arithmetic typically gives tighter bounds than interval arithmetic, in some cases dramatically so, as we have shown by example. However, it can also sometimes lead to *worse* bounds. To see this, consider the product $x \times y$, where both variables have bounds $[c - r, c + r]$, where both $c \geq r \geq 0$. Based on the heuristic given above, the resulting affine form will be $c^2 + cr\varepsilon_0 + cr\varepsilon_1 + r^2\varepsilon_2$, corresponding to a lower bound of $c^2 - 2cr - r^2$ (attained when $\varepsilon_0 = \varepsilon_1 = \varepsilon_2 = -1$), while IA gives $(c - r)^2$.

While the lower bound from IA is tight, the bound from AA can actually be negative, as in the case of $c = 5$, $r = 3$ ($c^2 - 2cr - r^2 = -14$). This is unfortunately necessary to preserve the (linear) dependency information while respecting the constrained (i.e. center-symmetric and convex) affine form representation. To see how this can be problematic for our GP application, consider an expression such as $log(x * y)$; we need to be able to accept such expressions as valid when $x$ and $y$ are known to be strictly positive.

Accordingly, the second improvement to basic AA that we employ is hybridization with IA, similar to the proposal in [2]. Affine forms are extended by the presence of independent minimal and maximal values, which by default are set

to their AA minima and maxima. When a non-affine operation is computed, its minima and maxima according to IA are computed as well, and the tightest possible bounds are kept, and exploited by AA to allow expressions that would otherwise be disallowed. This is achieved by adjusting the residual error term to be the smallest needed to cover the combined (minimal) interval. The hybrid model produces tighter bounds than either AA or IA could attain independently, and is not significantly more costly that plain AA to compute.

# 4. PROPOSED SYSTEM

We utilize a simple, generational genetic programming system in our experiments, based on Koza's classic configuration [8]. This baseline GP system has protected operators for division ($rdiv(x) = 1$ if $x = 0$ else $1/x$) and logarithm ($rlog(x) = 0$ if $x = 0$ else $\log|x|$), as well as random ephemeral constants. Parents are chosen for reproduction via tournament selection. New expressions are created via subtree crossover (probability 0.9) and mutation (probability 0.1). Additionally, the top 10% of each generation are carried over to the next without modification (elitism). The initial population is generated via the ramped half-and-half method with a maximal depth of 6, and with terminals taking on ephemeral random constants with probability 0.2. Ephemeral random constants are drawn from a normal distribution with mean 0 and standard deviation 5, following [5].

This baseline system can be extended by the integration of a number of enhancements:

**Interval Arithmetic** As in [5], static analysis of expressions through interval arithmetic is performed, so trees containing asymptotes are detected and rejected.

**Affine Arithmetic for Asymptote Rejection** Same as above, but substituting affine arithmetic for interval arithmetic.

**Affine Arithmetic for Output Bounds Rejection** We utilize affine arithmetic to determine the output bounds of each expression, and those whose output range is outside a desirable range are rejected. We define the desirable range as:

$$r(Y^T) = [min(Y^T) - \alpha r,\ max(Y^T) + \alpha r]$$

where $Y^T$ are the target outputs for all training cases, $r = (max(Y^T) - min(Y^T))/2$, and $\alpha$ is a parameter.

**Linear Scaling** As in [5]. Given an expression producing outputs $Y$ for $n$ training cases $X$, and with $Y^T$ being the desired, or target, output values, a linear regression can be performed as:

$$b = \frac{\sum_{i=1}^{n}(y_i^T - \tilde{Y}^T)(y_i - \tilde{Y})}{\sum_{i=1}^{n}(y_i - \tilde{Y})}$$
$$a = \tilde{Y}^T - b\tilde{Y}$$

where $\tilde{Y}$ and $\tilde{Y}^T$ denote the average output and average desired output, respectively. Replacing outputs by $a + bY$ minimizes mean squared error w.r.t. $Y^T$. During linear regression, the variance of outputs $Y$ is measured, and the tree is discarded if that value is greater than $10^7$ or lower than $10^{-7}$. This includes expressions with constant outputs, as they lead to a division by zero in the equation for $b$ above.

**Table 2: Parameter settings for experiments. These parameters are used for all problems, except Problem 4, for which we have a population size of 2000, allow 100,000 evaluations and include functions $exp(x)$, $log(x)$, $sin(x)$, and $cos(x)$.**

| | |
|---|---|
| Population size | 500 |
| Function set | $x + y$, $x \times y$, $1/x$, $-x$, $x^2$ |
| Tournament size | 5 |
| Max fitness evaluations | 25,000 |
| Max tree depth | 17 |
| Max mutation tree size | 4 |
| $\alpha$ for output bounds | 1 |

The cost of linear scaling is $O(n)$, where $n$ is the number of fitness cases, while the cost of interval and affine arithmetic analysis is determined by the size of the program tree and the costs of individual operations. Output bounds rejection does not add any significant cost to the affine arithmetic analysis process, as the target output range is computed once before each evolution run. Note however that when output bounds rejection is combined with linear scaling, scaling must be performed first (although expressions with asymptotes may be rejected prior to scaling).

We implemented the baseline system as well as the enhancements in Common Lisp, as part of the PLOP open source project[10].

# 5. EXPERIMENTAL RESULTS

We test our system on the same set of 15 benchmark problems used in [5], summarized in Table 1. Table 2 describes the parameters used in our tests. We did not attempt to tune any of these parameters to improve performance (they are either standard in the GP literature or selected to allow our results to be directly comparable to those in [5][2]), with the exception of $\alpha$, for which larger values produced worse results (not shown), and lower values are likely too conservative due to errors in the estimated bounds. Each problem has a set of training points and a separate set of test points.

We tested a number of configurations of the system described above on each problem, performing enough runs to obtain tight confidence intervals for the performance metrics to be presented below (150). We have found that using either linear scaling or interval/affine analysis only does not lead to reliably robust results, while combining linear scaling with either analysis technique is quite productive. Consequently, we will analyze results for the following configurations:

**Base** Baseline GP system with protected operators.

**IA** Linear scaling and interval arithmetic for asymptote rejection. This configuration produced the best results

[2]The careful reader familiar with [5] will note that in that work the default function set is reported as $x + y$, $x \times y$, $1/x$, $-x$, and $sqrt(x)$. However, the experiments in that work in fact included the function $sqr(x)$ (i.e. $x^2$), and *not* $sqrt(x)$ [6]. We accordingly use the function $x^2$ here.

**Table 1: Benchmark problems. The last two columns describe how training and test data are generated. Datasets can be uniformly spaced (notation *[lower:step:upper]*) or a random sample of $n$ uniformly distributed points (notation $rnd_n(lower, upper)$). For multidimensional problems, the *[lower:step:upper]* notation denotes an $n$-dimensional mesh, and hence $((upper - lower)/step)^n$ total data points.**

| Prob. | Equation | Train | Test |
|---|---|---|---|
| 1 | | $[-1 : 0.1 : 1]$ | $[-1 : 0.001 : 1]$ |
| 2 | $f(x) = 0.3xsin(2\pi x)$ | $[-2 : 0.1 : 2]$ | $[-2 : 0.001 : 2]$ |
| 3 | | $[-3 : 0.1 : 3]$ | $[-3 : 0.001 : 3]$ |
| 4 | $f(x) = x^3e^{-x}cos(x)sin(x)(sin^2(x)cos(x) - 1)$ | $[0 : 0.05 : 10]$ | $[0.05 : 0.05 : 10.05]$ |
| 5 | $f(x, y, z) = \frac{30xz}{(x-10)y^2}$ | $x, z = rnd_{10^3}(-1, 1)$<br>$y = rnd_{10^3}(1, 2)$ | $x, z = rnd_{10^4}(-1, 1)$<br>$y = rnd_{10^4}(1, 2)$ |
| 6 | $f(x) = \sum_i^x 1/i$ | $[1 : 1 : 50]$ | $[1 : 1 : 120]$ |
| 7 | $f(x) = logx$ | $[1 : 1 : 100]$ | $[1 : 0.1 : 100]$ |
| 8 | $f(x) = \sqrt{x}$ | $[0 : 1 : 100]$ | $[0 : 0.1 : 100]$ |
| 9 | $f(x) = \text{arcsinh}(x)$ | $[0 : 1 : 100]$ | $[0 : 0.1 : 100]$ |
| 10 | $f(x, y) = x^y$ | $x, y = rnd_{100}(0, 1)$ | $[0 : 0.1 : 1]$ |
| 11 | $f(x, y) = xy + sin((x - 1)(y - 1))$ | | |
| 12 | $f(x, y) = x^4 - x^3 + y^2/2 - y$ | | |
| 13 | $f(x, y) = 6sin(x)cos(y)$ | $x, y = rnd_{20}(-3, 3)$ | $[-3 : 0.01 : 3]$ |
| 14 | $f(x, y) = 8/(2 + x^2 + y^2)$ | | |
| 15 | $f(x, y) = x^3/5 + y^3/2 - y - x$ | | |

in [5], although the baseline system used in that work is a steady-state implementation with a specialized mutation operator.

**AA** Linear scaling and affine arithmetic for asymptote rejection.

**AA+** Linear scaling and affine arithmetic for asymptote rejection, plus output bounds rejection.

We note that, when using affine arithmetic for asymptote rejection, we can remove the function $x^2$ from our vocabulary, while this function is necessary for the interval arithmetic-based configuration in a number of problems, in order to generate expressions such as $1/(1 + x^2)$. Removing the function $x^2$ does not have a significant impact on the performance metrics presented below, and is not shown. It is, however, relevant to note that equivalent results can be obtained with a simpler function vocabulary.

First, we look at how each configuration performs during training. We use the Normalized Root Mean Square Error (NRMS) as a performance measure:

$$NRMS = \frac{\sqrt{\frac{n}{n-1}MSE}}{\sigma_{Y^T}}$$

where $n$ is the number of training (or test) cases, $\sigma_{Y^T}$ is the standard deviation of desired outputs for those cases, and $MSE$ is the mean squared error. We report NRMS as percentage points; an expression that always produces the mean desired output will have an NRMS of 100, while a perfect expression will have an NRMS of 0.

Table 3 shows the training NMRS for each of the above configurations. We note that there is always some residual error at the end of the evolution in all test cases, which shows that our benchmark problems are of non-trivial difficulty. We see that configurations **IA**, **AA**, and **AA+** all significantly improve on the baseline on most problems. **IA** and **AA** obtain very similar training errors on all problems, while **AA+** results in worse error in problems 2, 3, 4 (compared to **IA**), 10, 13, and 14 (compared to **IA**).

**Table 3: Mean training NRMS and 95% confidence interval.**

| Prob. | Base | IA | AA | AA+ |
|---|---|---|---|---|
| 1 | 38±2 | 20±2 | 18±2 | 19±2 |
| 2 | 70±2 | 47±3 | 48±4 | 72±3 |
| 3 | 84±2 | 80±2 | 80±2 | 92±2 |
| 4 | 20±2 | 16±2 | 19±2 | 21±2 |
| 5 | 6±1 | 1±0 | 1±0 | 1±0 |
| 6 | 26±3 | 1±0 | 1±0 | 1±0 |
| 7 | 25±4 | 2±0 | 2±0 | 2±0 |
| 8 | 19±3 | 3±1 | 3±1 | 4±1 |
| 9 | 26±3 | 4±1 | 4±1 | 5±1 |
| 10 | 24±3 | 11±1 | 10±1 | 14±1 |
| 11 | 15±1 | 16±1 | 16±1 | 16±1 |
| 12 | 6±1 | 4±1 | 5±1 | 4±1 |
| 13 | 49±3 | 36±3 | 33±3 | 53±3 |
| 14 | 68±7 | 22±3 | 25±3 | 30±4 |
| 15 | 28±2 | 17±1 | 17±1 | 17±1 |

Using interval or affine arithmetic eliminates some expressions before fitness evaluation. When those expressions are eliminated, we do not replace them with new ones; instead, they are assigned infinitely bad fitness, and are never selected as parents. Therefore, eliminating expressions causes the algorithm to compute less than the maximum allowed

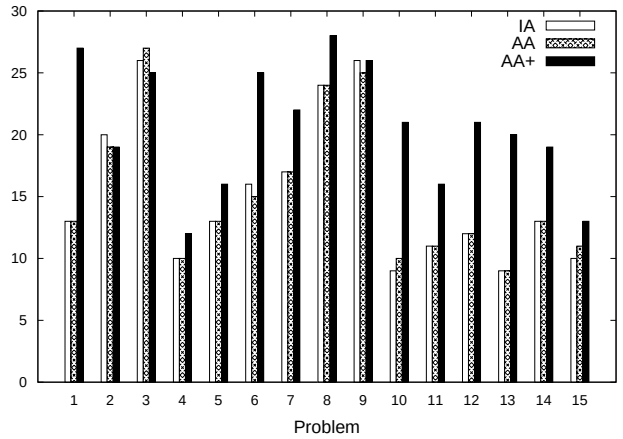


**Figure 1: Percentage of expressions rejected by each configuration.**

number of fitness evaluations. Figure 1 shows the percentage of expressions that are eliminated by interval or affine arithmetic analysis, thus reducing the necessary number of fitness evaluations during training. These values are very consistent across runs, with 95% confidence intervals nearly always within 1% of the plotted means. We see that all configurations significantly reduce the number actual fitness evaluations in comparison with the baseline, and the reduction by **AA+** is significantly higher on most problems, an expected consequence of output bounds rejection.

Now, we look at the generalization ability of each configuration. We measure the NRMS of the best evolved expression[3] in each run over the test set, and compare these values with the training NRMS values. Figure 2 shows the ratio between test and train NRMS[4] for each problem, as an indication of generalization quality. All configurations present very good generalization in problems 4 and 5. In the other problems, a clear pattern emerges. The baseline system has the worst generalization performance, followed by **IA**. **AA** and **AA+** are essentially equivalent, and superior to both **IA** and the baseline in problems 1 through 10. In the last five problems, however, **AA+** vastly outperforms all other configurations. It is also remarkable that **AA+** only has apparently poor generalization in problem 6, in which training NRMS is very low, and the mean test NRMS is less than 3%.

Figure 2 clearly shows that **AA+** has better overall generalization, and it is possible that its occasionally worse training NRMS is not a weakness, but a necessary artifact for improved performance on test data.

We have looked at test performance as it relates to training performance, but it is also important to evaluate absolute test error. For this purpose we apply two tests for overfitting. Keijzer describes an evolved expression as manifesting *destructive overfitting* when its mean squared error over the test set is greater than $10^5$. Given the number of test points and their target output values, MSE values greater than $10^5$ imply that, for at least some test points, the evolved expression produces values very far from the expected range. Using such expressions in production settings could have serious consequences. According to this definition, the baseline system suffers from overfitting in 12 of the 15 benchmark problems, with problems 11 (39%), 12 (31%), 13 (71%), and 15 (43%) being the worst ones. **IA** only suffers from overfitting in problems 1 (2%) and 11(1%); **AA** displays overfitting in problems 11 (1%) and 14 (2%), while **AA+** never suffers from such damaging errors.

Our results here for baseline GP and GP with scaling and IA agree qualitatively with those reported in [5], with the exception of Problem 10 ($f(x) = \text{arcsinh}(x)$), where a much greater rate of overfitting is reported for the configuration without IA (98%) that our highest rate (7%). A strong possibility is that this discrepancy is due to the more effective mechanism in [5] for adapting the values of numerical constants (a specialized mutation operator to allow for more frequent small changes), as otherwise the two setups are quite similar. While this operator allows the training set MSE to be minimize more effectively than by standard GP, it may also be responsible for a greater degree of overfitting on some problems (e.g. by facilitating the exact positioning of vertical asymptotes between training points).

We also consider a more restrictive definition of overfitting, based on NRMS on the unseen test set. As an NRMS value of 100% over the test set implies performance no better than constantly outputting the mean desired value, we can assume that all evolved expressions whose test set NRMS is greater than 100% are overfit. Table 4 shows the percentage of runs that produce overfit expressions according to this definition, for each problem. The baseline system produces overfit expressions in 13 of the 15 benchmark problems, and nearly always fails to generalize in problem 13. Configurations **IA** and **AA** also have some difficulty with this problem, and, less frequently, with problems 11, 14, and 15. Configuration **AA+** outperforms the alternatives on this test as well, with small failure rates in problems 13 (4%) and 15 (1%) and consistent success on the remaining problems.

We further investigate the relation between NRMS threshold and the probability that test performance will be worse than that threshold. We focus on the four hardest problems, as determined by the failure rates shown in Table 4. Figures

**Table 4: Percentage of runs displaying test NRMS greater than 100%.**

| Prob. | Base | IA | AA | AA+ |
|---|---|---|---|---|
| 1 | 13 | 2 | 0 | 0 |
| 2 | 64 | 0 | 0 | 0 |
| 3 | 72 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 |
| 6 | 18 | 0 | 0 | 0 |
| 7 | 8 | 0 | 0 | 0 |
| 8 | 23 | 0 | 0 | 0 |
| 9 | 35 | 0 | 0 | 0 |
| 10 | 18 | 0 | 0 | 0 |
| 11 | 83 | 11 | 11 | 0 |
| 12 | 44 | 0 | 0 | 0 |
| 13 | 98 | 33 | 43 | 4 |
| 14 | 84 | 10 | 8 | 0 |
| 15 | 73 | 8 | 4 | 1 |

[3]Best considering NRMS on *training data only*, of course.
[4]Test NRMS was capped at $10^6$ to minimize disruption caused by ill-formed exceptions in the baseline system

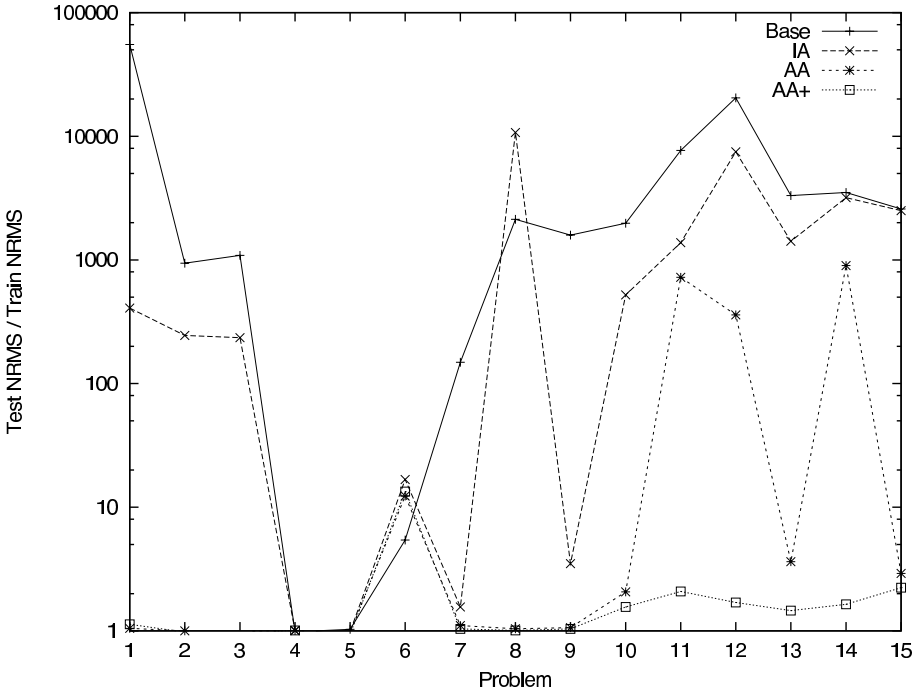


**Figure 2: Test NRMS divided by train NRMS for each problem (log scale).**

3 through 6 depict the probability that test set NRMS will be greater than a threshold for problems 11, 13, 14, and 15, for the baseline and for configurations **IA** and **AA+**. In each chart, the horizontal axis is the NRMS threshold, in logarithmic scale, with the midpoint corresponding to the values in Table 4. We can see that while the baseline GP system is always prone to overfitting, **IA** is fairly resilient. **AA+** is even more robust, and its maximal NRMS values are significantly lower than the ones obtained by other configurations on these harder problems. However, on Problems 13 and 14 it can be seen that **IA** finds solutions with very low NRMS somewhat more often than **AA+** (illustrated by the crossing of the lines). It may be that the greater proportion of expressions rejected by **AA+** (see Figure 1) necessitates a somewhat larger population size in order to find such high-quality solutions. The benefits from avoiding a larger proportion of destructively overfit solutions clearly justify this requirement, over the problems we have studied.

## 6. CONCLUSION

We have confirmed Keijzer's original idea from [5], namely that genetic programming for symbolic regression can be made significantly more robust by a combination of linear scaling before fitness evaluation and static analysis of expression bounds through interval arithmetic, with the goal of rejecting expressions that contain asymptotes. We have improved that combination even further by substituting affine arithmetic for interval arithmetic. Static analysis with affine arithmetic provides bounds tight enough to be useful for rejecting expressions whose outputs lie outside a desirable range of values. This extra method for rejecting umpromising solutions does not impact runtime efficiency compared to performing just asymptote rejection.

Experiments on benchmark functions show that a system configured with linear scaling, affine arithmetic analysis, and output bounds rejection, produces significantly better generalization than the interval arithmetic-based alternative. Our improved system is also able to solve all benchmark functions with a reduced function vocabulary.

Introducing affine arithmetic into GP opens up a host of intriguing possibilities; our work here barely scratches the surface. AA computes bounds and dependencies not just for candidate solutions, but for all of their subexpressions. This can be exploited in performing crossover, for example, by preferentially crossing subtrees with similar ranges and/or dependencies, without costing us any additional fitness evaluations (compare to the context-aware crossover operation [11]). A similar smart mutation operator can introduce medium-sized random subtrees that are nonetheless expected to behave similarly to those they replace.

Future research directions also include experiments on higher dimensional problems. We are especially interested in problems that present dependencies among the input variables, which can be mapped into affine noise symbols, potentially magnifying the usefulness of affine arithmetic analysis. We will also apply the techniques described here to supervised classification problems. Finally, we will explore the construction of model ensembles, both for regression and classification problems.

## 7. REFERENCES

[1] J. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. In *Anais do VI Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens (SIBGRAPI)*, 1993.

[2] L. H. de Figueiredo and J. Stolfi. *Self-Validated Numerical Methods and Applications*. Brazilian Mathematics Colloquium monographs. IMPA/CNPq, Rio de Janeiro, Brazil, 1997.

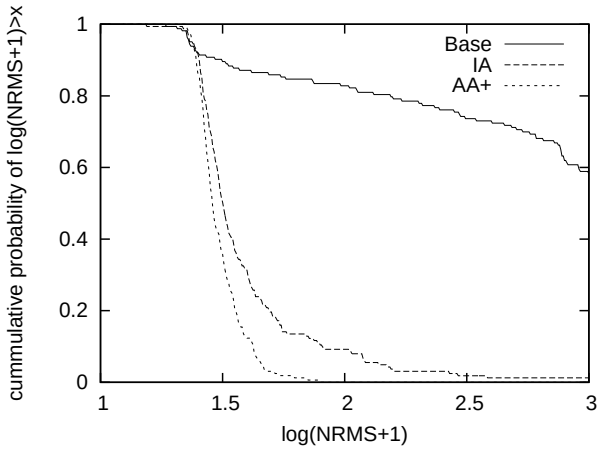


**Figure 3: Probability of test set NRMS being greater than a threshold for Problem 11.**

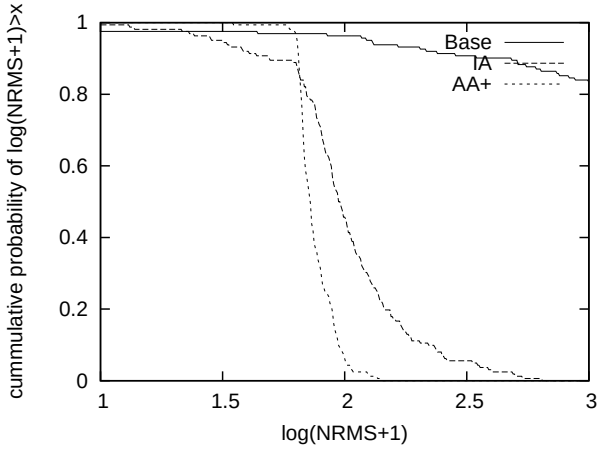


**Figure 4: Probability of test set NRMS being greater than a threshold for Problem 13.**

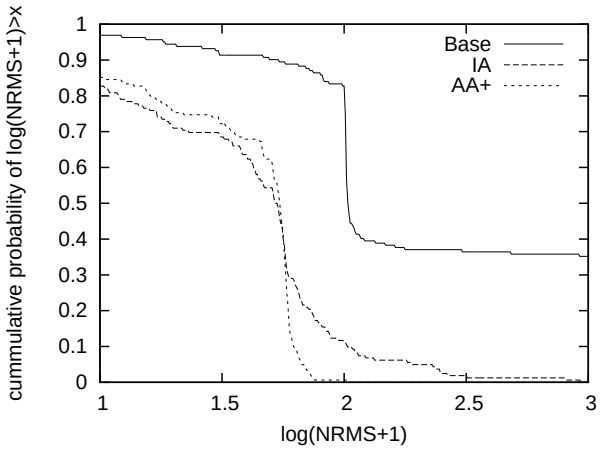


**Figure 5: Probability of test set NRMS being greater than a threshold for Problem 14.**

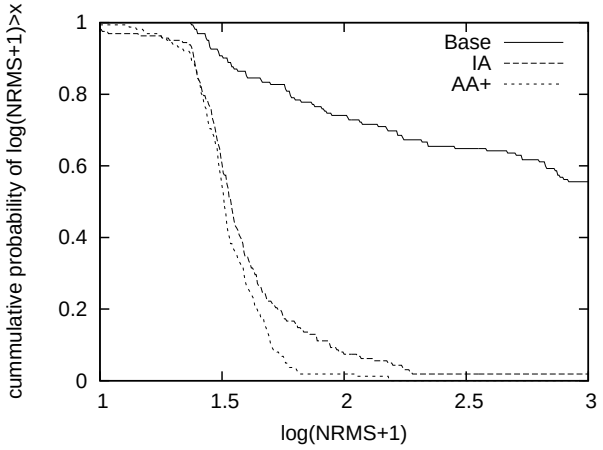


**Figure 6: Probability of test set NRMS being greater than a threshold for Problem 15.**


[3] L. H. de Figueiredo and J. Stolfi. Affine arithmetic: Concepts and applications. *Numerical Algoritms*, 37:147–158, 2004.

[4] O. Gay. Libaffa - C++ affine arithmetic library for GNU/Linux. `http://www.nongnu.org/libaffa/`, 2006.

[5] M. Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In *EuroGP: European conference on genetic programming*, 2003.

[6] M. Keijzer. Personal communication, 2010.

[7] M. Kotanchek, G. Smits, and E. Vladislavleva. Trustable symbolic regression models: using ensembles, interval arithmetic and pareto fronts to develop robust and trust-aware models. In R. Riolo, T. Soule, and B. Worzel, editors, *Genetic Programming Theory and Practice V*. Springer, 2008.

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

[9] W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[10] M. Looks. PLOP: Probabilistic Learning of Programs. `http://code.google.com/p/plop`, 2010.

[11] H. Majeed and C. Ryan. A less destructive, context-aware crossover operator for GP. In *EuroGP: European Conference on Genetic Programming*, 2006.

[12] R. Moore. *Interval Analysis*. Prentice Hall, 1966.

[13] J. Stolfi. LIBAA: An affine arithmetic library in C. `http://www.dcc.unicamp.br/~stolfi/`, 1993.

[14] G. Valigiani, C. Fonlupt, and P. Collet. Analysis of GP improvement techniques over the real-world inverse problem of ocean colour. In *EuroGP: European Conference on Genetic Programming*, 2004.