A Hybrid GP Approach for Numerically Robust Symbolic Regression

Günther R. Raidl
Department of Computer Graphics
Vienna University of Technology
Karlsplatz 13/1861, 1040 Vienna, Austria
raidl@eiunix.tuwien.ac.at

ABSTRACT

This article introduces a hybrid variant of genetic programming (GP) for doing symbolic regression. Instead of the usual interpretation of a parse tree, all top-level terms are identified and extended by multiplying them with locally optimized factors. These weighted terms are then linearly combined to form the resulting expression. When using the mean square error as fitness function, local optimization of the factors can be done efficiently by applying a robust variant of the method of least squares. Furthermore, the presented hybrid GP uses arbitrary precision arithmetic for evaluating each solution to detect major precision losses, numerical underflows, or overflows. A penalty according to the lost accuracy is added to the objective function to avoid such problems in the final solution. Various experiments indicate that the new hybrid GP finds numerically robust expressions with much smaller approximation errors faster and more reliably than traditional GP.

1 Introduction

Approximating a function using a given finite sampling of values of independent variables and associated values of dependent variables is a very important practical problem. Usually, a mathematical model is given and numerical parameters need to be found so that the resulting approximation provides a good fit for the given samples. One very common technique is linear regression, in which the model is a linear combination of given base functions and the parameters to be determined are their coefficients. Examples for base functions are polynomials (polynomial regression) or trigonometric polynomials (e.g. Fourier series). For general linearly independent base functions, the *method of least squares* (MLS) is an efficient method for determining the coefficients to get the smallest possible mean square error. Another class of approximation techniques is the great variety of neural networks in which the underlying model is a connected net of functional units, and the unknown parameters are usually the weights of connections between these units.

Over the last years, evolutionary algorithms have been recognized to be very suitable not only for optimizing parameters for given fixed approximation models as e.g. neural nets width predefined structures (Schaffer, Whitley, and Eshelman 1992) or Tensor Product Bernstein Polynomials (Raidl 1998, Raidl and Kodydek 1998), but also for adapting the models themselves, see e.g. Ahmed and De Jong (1997) and Schaffer, Whitley, and Eshelman (1992). In McKay, Willis, and Barton (1995), a genetic algorithm is presented which optimizes expressions encoded in tree structures to perform symbolic regression. A non-linear optimization method was used to get well suited values for constants in the evolved expressions. Unfortunately, these non-linear optimizations are very time-consuming. Rogers (1995) presented a commercial drug modeling tool which uses a genetic algorithm. Numerical constants within evolved polynomial models are efficiently optimized by using the MLS.

In Koza (1992), *genetic programming* (GP) was introduced which is a class of evolutionary algorithms working on executable tree structures (*parse trees*). Koza showed that GP is capable of doing *symbolic regression* (or function identification) by generating mathematical expressions approximating a given sample set very closely or in some cases even perfectly. Therefore, GP finds the entire approximation model and its (numerical) parameters simultaneously. In the following it is assumed that the reader is familiar with the basics of GP. Otherwise, see Koza (1992) and Koza (1994a) for a general introduction.

Although arbitrary numerical constants contained in functions to be approximated can principally be evolved in GP by using so-called *ephemeral random constants* $\Re$ (Koza 1992), they seem to be a weak point in the following sense: Since the needed constants must be assembled from random values and the given set of base functions $\mathcal{F}$, destination functions containing constants are usually much more demanding to GP than others. E.g. an expression corresponding to $f_1(x) = x + x^2 + x^3$ is usually evolved much faster than one corresponding to $f_2(x) = 2.71x + 3.14x^2$, assuming the set of base functions is $\mathcal{F} = \{+, -, *\}$, and the set of allowed terminals is $\mathcal{T} = \{x, \Re\}$. Similarly to McKay, Willis, and Barton (1995), Fröhlich and Hafner (1996) presented a modified GP including linear and non-linear optimization techniques to improve these numerical constants. Unfortunately, only a few steps of the time expensive non-linear optimization can be applied to each new solution to keep the total running time within limits.

Another drawback of doing symbolic regression with GP is the possibility of getting final solutions with very low approximation errors but undetected numerical problems: The closure property of GP requires that each of the base functions is able to accept, as its arguments, any value that might possibly be returned by any base function and any value that may possibly be assumed by any terminal. To use e.g. division, a protected function $\mathrm{div}(a, b)$ is usually defined which returns 1 if $b = 0$ and a/b otherwise. Similar protections must be established for many other base functions as e.g. $\log(a)$ or $\sqrt{a}$. At first sight, these protected definitions seem to solve numerical problems. But they may introduce unwanted and unexpected discontinuities in the resulting approximation. Furthermore, extremely small or high values triggering problems with numerical underflows or overflows might easily occur during the evaluation of GP solutions. E.g. including functions like e^a or a^b in $\mathcal{F}$ is dangerous in this sense. Functions must therefore also be protected against under- and overflow.

Usually, solutions for which such extreme values occur as intermediate results during evaluation will perform only poorly and get bad fitness values. But this is not always the case. Sometimes, GP leads to final solutions containing underflows or overflows which are compensated in some way (e.g. a very large value may be multiplied by a very small one). Although numerical problems do occur, only small errors might be observed for the samples used during evaluation. One way to circumvent these problems is to introduce a special value *undef* which is returned by a function in case of any error. Each function getting *undef* as one of its parameters must also return *undef*. Errors made at any time during evaluation are encountered in this way and the solution gets worst fitness and will never be selected for recombination or reproduction.

Often GP solutions also contain overlooked numerical problems which, unlike underflows, overflows, or undefined results, are not directly detectable: E.g. the subtraction of two nearly equal values may result in a substantial loss of precision. Also, determining a trigonometric function such as $\sin(a)$ for some value a much larger than 2π usually results in a major precision loss. Often such a precision loss exceeds the precision of the underlying arithmetic and the result renders absolutely meaningless, purely depending on the implementation of the arithmetic, and therefore in some sense random.

This paper introduces an improved version of GP called *hybrid genetic programming* (HGP) in which each solution is locally optimized by applying the method of least squares to find optimum coefficients for top-level terms. Furthermore, an arithmetic is used which observes the numerical precision during each step of the evaluation of a solution. Solutions are then penalized according to their loss of accuracy. Although local optimization introduces a time overhead, experimental results indicate that for many problems the algorithm converges faster to much better solutions which are numerically robust.

2 Locally Improving GP solutions

The general goal is to find a numerically robust expression $E(x)$ which minimizes the following mean square error for a given sample set $\mathcal{A}$ of independent input values x_k and dependent output values d_k ($k = 1, \ldots, |\mathcal{A}|$):

$$MSE(E, \mathcal{A}) = \frac{1}{|\mathcal{A}|} \sum_{k=1}^{|\mathcal{A}|} (E(x_k) - d_k)^2 \qquad (1)$$

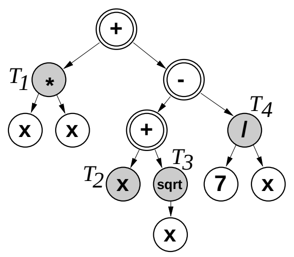
Note that limiting the dimensions of the input and output spaces to one does not restrict generality, since the approximation of a function $f : \mathbb{R}^m \to \mathbb{R}^n$ can be substituted by n independent approximations $E_j : \mathbb{R}^m \to \mathbb{R}$, $j = 1, \ldots, n$. Furthermore, the extension from one-dimensional input space to m dimensions is also straightforward in GP by including m variables instead of only one in the terminal set $\mathcal{T}$. But to keep things simple, only approximations with one-dimensional input and output spaces are considered in the following.

To improve the efficiency of GP for finding such expressions $E(x)$ including floating point constants, the parse tree of a solution is interpreted in a different, more sophisticated way than it is usually done. In a first step, all subtrees which have no other nodes than addition or subtraction as predecessors are identified. These so-called *top-level terms* $T_i(x)$ ($i = 1, \ldots, l$) are as usual interpreted as subexpressions and extended by multiplying them with independent weights w_i. The final resulting expression is the sum of all weighted top-level terms, see also the example depicted in Fig. 1:

$$E(x) = \sum_{i=1}^{l} w_i T_i(x) \qquad (2)$$

The optimum weights w_i of such a linear combination of terms leading to a minimum mean square error can be found efficiently by the well-known method of least squares, see e.g. Crow, Davis, and Maxfield (1960): Partially differentiating $MSE(E, \mathcal{A})$ successively by all the weights w_i and setting these derivatives equal to zero results in a system of linear equations. This system of linear equations can be solved easily as long as the terms T_i are linearly independent and the system is well-conditioned. To also handle critical cases containing linearly depending terms, a QR decomposition with Givens rotations is used. In this way, the MLS always gives meaningful values. More details on this efficient and robust technique can be found in Golub and Loan (1990).

To make the expression more compact and in some cases numerically more robust, some algebraic simplification rules



$(\mathcal{F} = \{+, -, *, /, \mathrm{sqrt}\}, \mathcal{T} = \{x, \Re\})$

Traditional interpretation:
$E(x) = x^2 + x + \sqrt{x} - \frac{7}{x}$

New interpretation:
$E(x) = w_1T_1 + w_2T_2 + w_3T_3 + w_4T_4 =$
$= w_1x^2 + w_2x + w_3\sqrt{x} + w_4\frac{7}{x}$

Figure 1 Interpretation of a parse tree.

are applied to the whole expression before and after performing the MLS.

Note that all the described techniques are only applied to a copy of the original parse tree generated before evaluation. The unchanged original parse tree remains the genotype which may be used during the next generation of GP for creating offspring solutions. In various experiments GP has performed poorly when the parse trees were adapted to correspond to the expressions finally obtained. One reason for the poor performance in this case might be that building blocks cannot evolve well because they are frequently destroyed.

3 Numerical Robustness

As already mentioned in the introduction, an important goal in symbolic regression is to get a solution which is numerically robust and which does not require a special underlying arithmetic to give accurate output values for given input parameters.

Unfortunately, it is very hard to decide on the robustness of a general expression for all possible input values within certain limits. In practice, only an estimation for the general robustness based on the sample set used during evaluation seems to be possible.

In contrast to other work, e.g. Koza (1992), Koza (1994a), and Fröhlich and Hafner (1996), we propose to use the special symbol *undef* in case of underflows, overflows, or undefined results instead of providing protected versions of functions. Our tests point out that GP finds solutions with small mean square errors slightly faster when using protected functions, but on the other hand it is much safer if a solution containing any kind of numerical error gets the worst possible fitness (∞) and will never be selected for reproduction or recombination. In this way, underflows or overflows cannot be compensated and will therefore not occur in a final solution. Furthermore, if the function set $\mathcal{F}$ only contains continuous and differentiable functions, the final solution is usually continuous and differentiable. In contrast, protected functions usually contain discontinuities and therefore often lead to discontinuities in the final solution.

To estimate the numerical robustness of an expression not directly leading to any numerical error during evaluation, special arbitrary precision arithmetic is used which keeps track of the real precision of a value at all points during evaluation: For each numerical value a, the number of decimal digits treated as significant and therefore known exactly is stored together with the value as precision $Prec(a)$. Since the floating point format is used to handle large values as well as values close to zero, this precision is a measure of the relative error in the value. In case of HGP, each terminal's value gets the same initial precision. Each function of the function set must be able to determine the correct precision of the resulting value considering the precision of its input values, see Fig. 2.

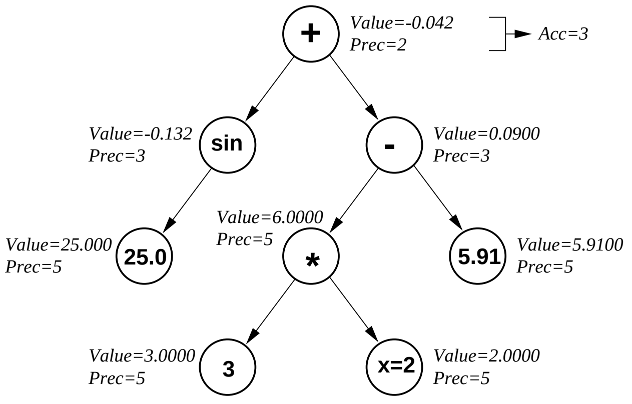


Figure 2 Evaluation with arbitrary precision arithmetic.

If the precision of a resulting value amounts to zero, no digit of the obtained value is accurate, and the entire value is therefore useless. This case might easily happen for e.g. $\sin(a)$, $a \gg 2\pi$, or when subtracting two nearly identical values. Expressions resulting in such useless values for any of the evaluated samples will immediately be penalized by getting fitness ∞ as in the case of an underflow, overflow, or undefined value.

But resulting values for which only a few digits are accurate also depend highly on the underlying arithmetic. Small mean square errors may then lead to wrong assumptions about the real quality of the found expression. To be on the safe side, a worst case absolute error should be determined and considered when calculating the mean square error.

From a specific value a and its precision $Prec(a)$, the so-called accuracy $Acc(a)$, which is the number of significant digits to the right of the decimal point and therefore a measure for the absolute error, can be obtained easily:

$$Acc(a) = Prec(a) - \lfloor \log_{10}|a| + 0.5 \rfloor \tag{3}$$

Note that according to this definition, $Acc(a)$ may also be negative.

The calculation of the mean square error can be extended as follows to take accuracy into account:

$$MSE_{Acc}(E,\mathcal{A}) = \frac{1}{|\mathcal{A}|}\sum_{k=1}^{|\mathcal{A}|}(|E(x_k)-d_k| + 10^{-Acc(E(x_k))})^2 \tag{4}$$

Worst case errors are added as *accuracy penalties* to the absolute differences between the results of the expression for the samples x_k and the aspired output values d_k. Using the mean square error determined in this way is therefore far more meaningful.

Note that instead of the described arbitrary precision arithmetic, exact interval arithmetic can be used for HGP in a very similar way. Usually, interval arithmetic is more accurate, but also more time expensive. Since only the order of magnitude of accuracy is important in this application, the faster arbitrary precision arithmetic seems to be the better choice.

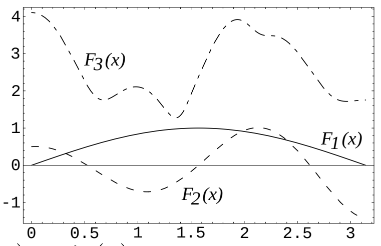
4 Implementation and Results

The proposed HGP has been implemented using Linux, the *Genetic Programming Kernel* written in C++ by Fraser and Weinbrenner (1997), and Mathematica 3.0 (Wolfram 1991). Solutions generated by the GP Kernel are sent over a pipe to a Mathematica process running in parallel. Mathematica interprets solutions by applying the MLS and algebraic simplification rules and carries out the evaluation using arbitrary precision arithmetic. Resulting fitness values are returned to the GP Kernel via a second pipe. Using a mathematics package like Mathematica is surely not the most efficient way to implement HGP, but a very flexible approach. Besides the fact that only a few statements are necessary for simplifying a given expression or applying the MLS, the capabilities of Mathematica made testing and debugging very easy. A more efficient implementation using only one C++ process is planned for the future.

Besides several other functions, those shown in Fig. 3 were used for testing the new approach and comparing it to traditional GP. The sample set $\mathcal{A}$ always consisted of 50 samples randomly chosen out of $[0,\pi]$. The function and terminal sets used for approximating the test functions were $\mathcal{F} = \{+,-,*,/,\mathrm{sqrt},\hat{}2,\hat{}3\}$ and $\mathcal{T} = \{x,\Re\}$. Various control parameters which proved to be well suited when using either GP or HGP are subsumed in Table 1. Note that accuracy penalization was used for HGP as well as GP. During evaluation with the arbitrary precision arithmetic, an initial precision of eight decimal digits was assumed for all values corresponding to terminals. Note that in HGP, large population sizes are not as useful as in traditional GP. When using HGP, a population size of 300 turned out to be a good choice for the given test problems. For traditional GP, a population size between 1000 and 5000 proved to be well suited. Concerning the CPU time for interpreting and evaluating a solution, our implementation of HGP is about 10 to 15 times slower than the traditional approach. To make a comparison of GP and HGP easy, the population size of GP was set to 4000, thus about 13 times larger than the population size of HGP. In this way, the total CPU times per generation are very similar for GP and HGP. In both cases a run was terminated after 50 generations and needed approximately 40 minutes CPU time.

Table 1 Control parameters used for GP and HGP.

Function set:	$\mathcal{F} = \{+,-,*,/,\mathrm{sqrt},\hat{}2,\hat{}3\}$
Terminal set:	$\mathcal{T} = \{x,\Re\}$, $\Re \in [-1,1]$
Fitness cases:	$\mathcal{A}$: set of 50 random samples
Fitness function:	$MSE_{Acc}(E,\mathcal{A})$
Init. precision for evaluation:	8
Error handling:	return *undef*
Population size:	GP: 4000, HGP: 300
Number of generations:	50
Initialization:	ramped half-and-half
Max. depth for initialization:	6
Selection:	Tournament selection (GP: $k=7$, HGP: $k=5$)
Elitism:	yes
Crossover probability:	90%
Reproduction probability:	10%
Max. depth for crossover:	17



$F_1(x) = \sin(x)$
$F_2(x) = e^{x/3}\cos(3x)/2$
$F_3(x) = \ln(4+2\sin(x)\sin(8x)) + e^{\cos(3x)}$
$x \in [0,\pi]$

Figure 3 Test functions.

Table 2 shows mean square errors MSE_{Acc} of final solutions from 15 performed runs per test function and per algorithm sorted according to increasing values. Note that HGP outperforms traditional GP by several orders of magnitude. The performance curves of the runs leading to median mean square errors are depicted in Fig. 6. Already the best solutions of the initial populations of HGP led to smaller errors than the final solutions of GP. Note that the used initial precision for evaluation limits the smallest possible MSE_{Acc}. This limitation can especially be observed in the HGP runs for F_1 and F_2: While the runs perform very well until the 18th generations, only small improvements are made thereafter. Considering the order of magnitude of the dependent variables d_k and our initial precision of eight digits, MSE_{Acc} can never get lower than $\approx (10^{-8})^2 = 10^{-16}$. Other experiments have shown that a higher initial precision indeed enables even smaller errors for such functions as F_1 and F_2.

For the shown typical runs the mean square errors MSE_{Acc} and MSE, their relative difference (a measure for the total numerical precision), and the structural complexity (total number of nodes) of the final solutions are depicted in Table 3.

Table 2 Sorted mean square errors MSE_{Acc} of final solutions from 15 runs for GP/HGP and F_1 to F_3.

MSE_{Acc}	$F_1(x)$ / GP	$F_1(x)$ / HGP	$F_2(x)$ / GP	$F_2(x)$ / HGP	$F_3(x)$ / GP	$F_3(x)$ / HGP
Best:	$1.77{\times}10^{-2}$	$1.38{\times}10^{-14}$	$3.17{\times}10^{-2}$	$8.72{\times}10^{-13}$	$2.34{\times}10^{-1}$	$1.94{\times}10^{-4}$
	$1.93{\times}10^{-2}$	$1.46{\times}10^{-14}$	$3.62{\times}10^{-2}$	$9.36{\times}10^{-13}$	$2.35{\times}10^{-1}$	$2.49{\times}10^{-4}$
	$3.58{\times}10^{-2}$	$1.50{\times}10^{-14}$	$3.79{\times}10^{-2}$	$1.28{\times}10^{-12}$	$2.51{\times}10^{-1}$	$2.79{\times}10^{-4}$
	$3.82{\times}10^{-2}$	$3.23{\times}10^{-14}$	$3.83{\times}10^{-2}$	$1.50{\times}10^{-12}$	$2.88{\times}10^{-1}$	$3.12{\times}10^{-4}$
	$4.98{\times}10^{-2}$	$4.38{\times}10^{-14}$	$7.85{\times}10^{-2}$	$4.32{\times}10^{-12}$	$2.97{\times}10^{-1}$	$3.86{\times}10^{-4}$
	$5.13{\times}10^{-2}$	$1.20{\times}10^{-13}$	$7.88{\times}10^{-2}$	$5.61{\times}10^{-12}$	$3.25{\times}10^{-1}$	$3.90{\times}10^{-4}$
	$5.36{\times}10^{-2}$	$1.60{\times}10^{-13}$	$8.43{\times}10^{-2}$	$8.18{\times}10^{-12}$	$5.08{\times}10^{-1}$	$4.33{\times}10^{-4}$
Median:	$6.31{\times}10^{-2}$	$1.68{\times}10^{-13}$	$9.13{\times}10^{-2}$	$8.63{\times}10^{-12}$	$5.93{\times}10^{-1}$	$6.98{\times}10^{-4}$
	$6.36{\times}10^{-2}$	$2.47{\times}10^{-13}$	$1.04{\times}10^{-1}$	$1.22{\times}10^{-11}$	$6.02{\times}10^{-1}$	$8.12{\times}10^{-4}$
	$6.44{\times}10^{-2}$	$2.79{\times}10^{-13}$	$1.11{\times}10^{-1}$	$1.32{\times}10^{-11}$	$8.71{\times}10^{-1}$	$8.23{\times}10^{-4}$
	$7.38{\times}10^{-2}$	$5.14{\times}10^{-13}$	$1.12{\times}10^{-1}$	$2.10{\times}10^{-11}$	1.01	$1.08{\times}10^{-3}$
	$1.19{\times}10^{-1}$	$6.19{\times}10^{-13}$	$1.64{\times}10^{-1}$	$2.87{\times}10^{-11}$	1.04	$1.09{\times}10^{-3}$
	$1.20{\times}10^{-1}$	$6.50{\times}10^{-13}$	$2.12{\times}10^{-1}$	$5.57{\times}10^{-11}$	1.16	$1.41{\times}10^{-3}$
	$1.69{\times}10^{-1}$	$9.25{\times}10^{-13}$	$2.25{\times}10^{-1}$	$5.73{\times}10^{-11}$	1.20	$1.55{\times}10^{-3}$
Worst:	$2.11{\times}10^{-1}$	$1.00{\times}10^{-12}$	$2.54{\times}10^{-1}$	$5.96{\times}10^{-11}$	1.59	$2.27{\times}10^{-3}$
Average:	$7.83{\times}10^{-2}$	$4.63{\times}10^{-13}$	$1.12{\times}10^{-1}$	$1.39{\times}10^{-11}$	$5.96{\times}10^{-1}$	$8.08{\times}10^{-4}$

Table 3 The two mean square errors, their absolute and relative differences, and the structural complexities $Comp$ of the final solutions from the runs shown in Fig. 6.

Function	Algorithm	MSE_{Acc}	MSE	$MSE_{Acc} - MSE$	$\frac{MSE_{Acc}-MSE}{MSE}$	$Comp$
$F_1(x)$	GP	$6.3093278{\times}10^{-2}$	$6.3093277{\times}10^{-2}$	$1.00{\times}10^{-9}$	$1.58{\times}10^{-8}$	127
	HGP	$1.6799260{\times}10^{-13}$	$1.5379202{\times}10^{-13}$	$1.42{\times}10^{-14}$	$9.23{\times}10^{-2}$	183
$F_2(x)$	GP	$9.1329357{\times}10^{-2}$	$9.1329212{\times}10^{-2}$	$1.45{\times}10^{-7}$	$1.59{\times}10^{-6}$	139
	HGP	$8.6278573{\times}10^{-12}$	$8.3136812{\times}10^{-12}$	$3.14{\times}10^{-13}$	$3.78{\times}10^{-2}$	109
$F_3(x)$	GP	$5.9343866{\times}10^{-1}$	$5.9343853{\times}10^{-1}$	$1.30{\times}10^{-7}$	$2.20{\times}10^{-7}$	114
	HGP	$6.9802702{\times}10^{-4}$	$6.7389201{\times}10^{-4}$	$2.41{\times}10^{-5}$	$3.58{\times}10^{-2}$	218

$$E(x) = 4.9510016{\times}10^{-1} + 6.2906751{\times}10^{-1} \cdot x - 6.3670445{\times}10^{-1} \cdot x^3 + 8.1338748{\times}10^{-3} \cdot x^9 + 4.8007820{\times}10^{-5} \cdot \\ \cdot x^{12} - 2.6656530{\times}10^{-2} \cdot \left(2.3612010{\times}10^{-2} - 1.9604153 \cdot x + x^3\right)^3 - 1.6270552 \cdot \sqrt{x^3 + \sqrt{x^7}} + \sqrt{x^7} \cdot \\ \cdot \left(3.009657{\times}10^{-2} \cdot (2.3612010{\times}10^{-2} - 1.9604153 \cdot x)^2 + 1.0630923 \cdot \left(-1.9604152 \cdot x + \sqrt[4]{x^7}\right)^2\right)$$

Figure 4 A simplified final solution generated by HGP for function F_2: $MSE_{Acc} = 8.63{\times}10^{-12}$.

$$E(x) = 5.6997292 - 371.86678 \cdot (2.6925{\times}10^{-1} - x)^3 - 1.007778 \cdot \sqrt{x} - 114.92842 \cdot x - 752.4797 \cdot x^3 - 661.50399 \cdot \\ \cdot \sqrt{x^7} + 2325.1778 \cdot (3.09898{\times}10^{-1} + x)^6 - 463.74047 \cdot \sqrt{x^7} \cdot (3.1746{\times}10^{-1} + x) - 1981.9457 \cdot (3.1746{\times}10^{-1} + x)^3 \cdot \\ \cdot (x^2 + x^3) - 346.04283 \cdot (x^5 + x^6)$$

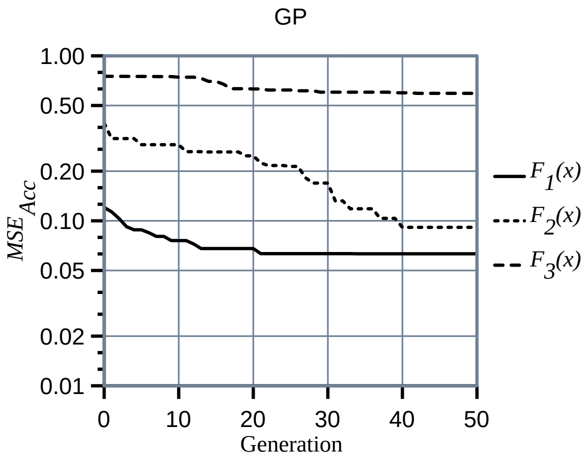
Figure 5 A simplified final solution generated by HGP without accuracy penalization for function F_2 leading to an essential precision loss when evaluated: $MSE = 1.43{\times}10^{-15}$, $MSE_{Acc} = 4.36{\times}10^{-2}$, $\frac{MSE_{Acc}-MSE}{MSE} = 3.0{\times}10^{13}$.

Usually solutions of HGP runs were slightly larger than those of GP. The final solution obtained by HGP for function F_2 is shown in Fig. 4.

Fig. 5 shows a solution generated by HGP when accuracy penalization has been disabled and MSE was used instead of MSE_{Acc} as fitness function. The large relative difference between MSE_{Acc} and MSE of $3{\times}10^{13}$ indicates an essential loss of precision. Very often, such ill-conditioned solutions with MSE values close to zero but large losses of accuracy were obtained as final results. This shows that the usage of the arbitrary precision arithmetic and MSE_{Acc} as fitness function is a substantial part of the algorithm for getting robust solutions.

5 Conclusions

Standard GP has been improved by extending the interpretation of parse trees: Top-level terms are multiplied by factors



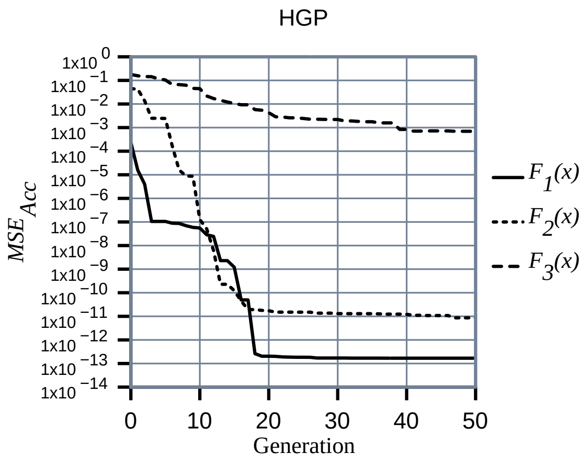


Figure 6 Typical performance curves for applying GP and HGP to the test functions F_1 to F_3.

determined by the MLS. Although only linear dependencies are locally optimized in this way, HGP usually finds much better approximations to functions involving numerical constants than traditional GP and is also faster concerning convergence speed. The MLS clearly introduces a time overhead in the evaluation process, but the described technique is far less time consuming than other, non-linear local optimization methods improving general numerical constants in GP solutions.

Numerical robustness seems to be a point often overlooked in doing symbolic regression, especially when using GP. Estimating accuracy by using arbitrary precision arithmetic during evaluation and considering the loss of accuracy in the fitness calculation guarantees numerical robustness at least for the samples used during evaluation.

6 Future Work

Our next step for accelerating HGP will be an implementation as a single process in C++. This should allow the application of HGP to larger problems. In general, more experiments should be done to gain better knowledge about well suited GP parameters. Incorporating automatically defined functions as discussed in Koza (1994a) may also be an interesting step forward in improving the abilities of HGP.

Bibliography

Ahmed M. A., De Jong, K.A. 1997. Function Approximator Design Using Genetic Algorithms. In *Proc. of the 1997 IEEE Int. Conference on Evolutionary Computation*. Indianapolis, IN, pp. 519–523.

Crow, E. L., Davis, F. A., Maxfield M. W. 1960. *Statistics Manual*. Dover Publications, New York.

Fraser, A., Weinbrenner, T. 1993–1997. *The Genetic Programming Kernel*. Version 0.5.2, GNU free software.

Fröhlich, J., Hafner, C. 1996. Extended and Generalized Genetic Programming for function Analysis. submitted to the *Journal of Evolutionary Computation*.

Golub, F. H., Loan, C. F. 1990. *Matrix Computations*. The Jones Hopkins University Press, London.

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

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

McKay, B., Willis, M. J., Barton G. W. 1995. Using a Tree Structured Genetic Algorithm to Perform Symbolic Regression. In *Proc. of the 1st Int. Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*. UK, pp. 487–492.

Raidl, G. R. 1998. Approximation with Evolutionary Optimized Tensor Product Bernstein Polynomials. In *Proc. of the Int. Conference on Artificial Intelligence in Industry: From Theory to Practice*. High Tatras, Slovakia, (to appear).

Raidl, G. R., Kodydek, G. 1998. Evolutionary Optimized Tensor Product Bernstein Polynomials versus Backpropagation Networks. In *Proc. of the Int. ICSC/IFAC Symposium on Neural Computation*. Vienna, Austria, (to appear).

Rogers, D. 1995. Development of the Genetic Function Approximation Algorithm. In *Proc. of the 6th Int. Conference on Genetic Algorithms*. Pittsburgh, PA, pp. 589–596.

Schaffer, J. D., Whitley, D., Eshelman, L. J. 1992. *Combinations of Genetic Algorithms and Neural Networks: A Survey of the State of the Art*. COGAN–92, IEEE Computer Society Press.

Wolfram, S. 1991. *Mathematica, a System for Doing Mathematics by Computer*. Addison Wesley, CA, 1991.