Faster Genetic Programming based on Local Gradient Search of Numeric Leaf Values

Alexander Topchy
Computer Science Dept.
Michigan State University
East Lansing, MI 48823
topchyal@cse.msu.edu

William Punch
Computer Science Dept.
Michigan State University
East Lansing, MI 48823
punch@cse.msu.edu

Abstract

We examine the effectiveness of gradient search optimization of numeric leaf values for Genetic Programming. Genetic search for tree-like programs at the population level is complemented by the optimization of terminal values at the individual level. Local adaptation of individuals is made easier by algorithmic differentiation. We show how conventional random constants are tuned by gradient descent with minimal overhead. Several experiments with symbolic regression problems are performed to demonstrate the approach's effectiveness. Effects of local learning are clearly manifest in both improved approximation accuracy and selection changes when periods of local and global search are interleaved. Special attention is paid to the low overhead of the local gradient descent. Finally, the inductive bias of local learning is quantified.

1 INTRODUCTION

The quest for more efficient Genetic Programming (GP) is an important research problem. This is due to the fact that a high computational complexity of GP is among its distinctive features (Poli & Page, 2000). Especially now, when variants of GP are being used on very ambitious projects (Thompson, 1998; Koza et al., 1997), the speed and efficiency of evolution are very crucial for such problems.

Numerous modifications of the basic GP paradigm (Koza, 1992) are currently known, e.g. see (Langdon, 1998) for a review. Among them, several researchers have considered GP augmentation by hill climbing, simulated annealing and other stochastic techniques. In (O'Reilly & Oppacher, 1996) crossover and mutation are used as move operators of hill climbing, while Esparcia-Alcazar & Sharman (1997) considered optimization of extra parameters (node gains) using simulated annealing. Terminal search was employed in (Watson & Parmee, 1996), but due to the associated computational expense it was limited to 2-4% of individuals. The presence of stochasticity in local learning makes it relatively slow, even though some hybrid algorithms yield overall improvement. Apparently, the full potential of local search optimization is yet to be realized.

The focus of this paper is on a local adaptation of individual programs during the GP process. We rely on gradient descent for improved generation of GP individuals. This adaptation can be performed repeatedly during the lifetime of an individual. The results of local learning may or may not be coded back into the genotype (reverse transcription) based on the modified behavior, which is reported in the literature as Lamarckian and Baldwinian learning, respectively (Hinton & Nowlan, 1987; Whitley et al., 1994). The resulting new fitness values affects the selection process in both cases, which in turn changes the global optimization performance of a GP. Such an interaction between local learning, evolution and associated phenomena without reverse transcription are also generally referred to as the Baldwin effect.

We were motivated by a number of successful applications of hybridization to neural networks (Belew et al., 1991; Zhang & Mühlenbein, 1993; Nolfi et al., 1994). Both neural networks and GP trees perform input-output mapping with a number of adjustable parameters. In this respect, terminal values (leaf coefficients) in a GP perform a similar function as weights in neural network. A form of gradient descent is usually used to adjust weights in a neural net architecture. In contrast, various terminal constants are typically random within GP trees and are rarely adjusted by gradient methods. The reasons for this are twofold: the unavailability of gradients/derivatives in some GP problems and the computational expense that is assumed to exist in computing those gradients. However, the complexity of computing derivatives is largely overestimated. In order to differentiate programs explicitly, algorithmic differentiation (Griewank, 2000) may be adopted. Algorithmic (computational) differentiation is a technique that accurately determines values of derivatives with essentially the same time complexity as found in the execution of the evaluation function itself. In fact,

gradients may often be computed as part of the function evaluation. This is especially true for trees and at least potentially true for arbitrary non-tree programs. Generalization of the method for any program is possible, given that the generated program computes numeric values, even in presence of loops, branches and intermediate variables. The main requirement is that the function be piecewise differentiable. While not always true, this is the case for a great majority of engineering design applications. Moreover, it is also known, that directional derivatives can be computed with many non-smooth functions (Griewank, 2000). Knowledge of only gradient direction, not its value, is often enough to optimize the values of parameters.

In this paper we empirically compare conventional GP with a GP coupled with terminal constant learning. The effectiveness of the approach is demonstrated on several symbolic regression problems. Arithmetic operations have been chosen as the primitives set in our GP implementation for simplicity sake. While such functions make differentiation easy, again these techniques can be adapted to more difficult problems.

Our results indicate that inexpensive differentiation along with Baldwin learning leads to a very fast form of GP. Significant improvement in accuracy was also achieved beyond that which could be achieved by either local search or more generations of GP.

The Baldwin effect is known to change the inductive bias of the algorithm (Turney, 1996). In the case of GP, where functional complexity is highly variable, it is expected that such a change of bias can be properly quantified. Two manifestations of the learning bias were observed in our experiments. Firstly, the selection process is affected by local learning since the fitness of many individuals dramatically improves during their lifetime. Secondly, changes in the functional complexity of individuals were observed in the experiments. Both the length (number of nodes) of the best evolved programs and the number of leaf coefficients were higher using local learning as opposed to regular GP.

2 LAMARCKIAN VS BALDWIN STRATEGY IN GP

Evolution rarely proceeds without phenotypic changes. As we are interested in digital evolution, two dominating strategies have been proposed which allow environmental fitness to affect genetic features. Lamarckian evolution, an alternative proposition to Darwinian approaches of the time, claimed that traits acquired from individual experience could be directly encoded into the genotype and inherited by offspring. In contrast, Baldwin claimed that Lamarckian effects could be observed where no direct transfer of phenotypic characteristic to the genotype occurred, in keeping with Darwinism. Rather, Baldwin claimed that "innate" behaviors could be

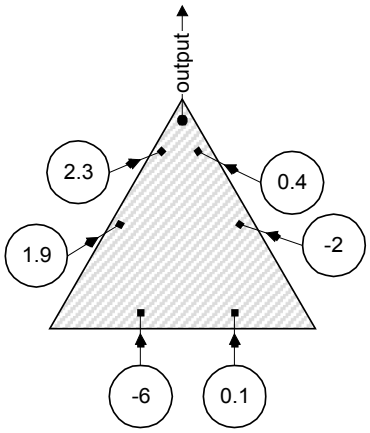


Figure 1: Sample tree with a set of random constants. In hybrid GP all these leaf coefficients are subjected to training

selected for (in a Darwinian sense) which the individual originally had to learn. In Lamarckian evolution learning affects fitness distribution as well as the underlying genotypic values, while the Baldwin effect is mediated via the fitness results only. In our case, the question is whether locally learned constants are copied back into the genotype (Lamarckian) or whether the constants are unmodified while the individual's fitness value reflects the fitness resulting from learning (Baldwin).

Real algorithmic implementations of evolution coupled with local learning are much richer than two original strategies. The researcher, usually guided by the total computational expense, may arbitrarily decide both the amount of and scheduling of learning or local adaptation of solutions. Moreover, since local learning comes with a price, it must be wisely traded off with genetic search costs. Several questions must be answered:

- What aspect of the solution should be learned beyond genetic search, as only a subset of solution parameters may be chosen for adaptation?
- Should learning be performed at every generation or should it be used as a form of fine-tuning when genetic search is converged?
- How many individuals and which of those individuals should have local learning applied to them?
- How many iterations of local learning should be done (really, how much computational cost are we willing to incur)?

Accordingly, there are many ways to introduce local learning into GP. Evolution in GP is both parametric and structural in nature. Two important features are specific to GP:

1. The fitness of the functional structure depends critically on the values of local parameters. Even very

fit structures may perform poorly due to inappropriate numeric coefficients.

2. The fitness of the individual is highly context sensitive. Slight changes in structure dramatically influence fitness and may require completely new parameters.

That is why we focus on learning numeric coefficients, so called Ephemeral Random Constants or ERC (Koza, 1992), which are traditionally randomly generated as shown in Figure 1. As explained below, the local learning algorithm -- gradient descent on the error surface in the space of the individual's coefficients, turns out to be a very inexpensive approach, so much so that every individual can do local learning in every generation.

Formally we follow the Lamarckian principle of evolution since we allow the tuned performance of individual to directly affect the genome by modifying numeric constants. At the same time, the choice between Lamarckian and Baldwin strategies in our implementation is not founded on the issue of computational complexity. In both cases the amount of the extra work is approximately the same. The main issue arises when considering the fitness values of the offspring with inherited coefficients vs. offspring with unadjusted terminals. Our experiments indicate that there is little difference between the two fitnesses when crossover is the main operator. Two factors contribute to this:

1. Crossover usually generates individuals with significantly worse fitness than their parents. The coefficients found earlier to be good for the parents are not appropriate for the offspring structures. The subsequent local learning changes fitness dramatically by updating the ERCs to more appropriate values.
2. Newly generated offspring are equally well adjusted starting from any values: earlier trained, not trained or even random.

Hence, inheritance of the coefficients does not much help the performance of the individuals created by crossover. However, if an individual is transferred to a new generation as a part of the elitist pool, i.e. unchanged by crossover or mutation, then its learned coefficients are also transferred. With respect to this structure, the use of the Baldwin strategy would be wasteful, since it requires relearning the same parameters. Thus, even though our implementation formally follows the Lamarckian strategy, we effectively observe the very same phenomena peculiar to the Baldwin effect.

3 HYBRID GP

The organization of the hybrid GP (HGP) is basically the same as that of the standard GP. The only extra activity done by the algorithm is to update the values of numeric coefficients. That is, all individuals in the population are trained using a simple gradient algorithm in every generation of the standard GP. Below we discuss the exact formulation of the corresponding optimization problem.

3.1 PROBLEM STATEMENT

The hybrid GP is intended to solve problems of a numeric nature, which may include regression, recognition, system identification or control. We will assume throughout that there are no non-differentiable nodes, such as Boolean functions. In general, given a set of N input-output pairs $(d,\mathbf{x})_i$ it is required to find a mapping $f(\mathbf{x},\mathbf{c})$ minimizing certain performance criteria, e.g. mean squared error (MSE):

Here, f is scalar function (generalization to multi-trees is

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\left(d_i - f(\mathbf{x}_i,\mathbf{c})\right)^2 \qquad (1)$$

trivial), $\mathbf{x}$ is vector of input values, $\mathbf{c}$ is vector of coefficients summed over the training samples. Of course, in GP we are interested in discovering the mapping $f(\mathbf{x},\mathbf{c})$ in the form of a program tree. That is, we seek not only coefficients $\mathbf{c}$, but also the very structure of the mapping f which is not known in advance. In our approach, finding the coefficients is done by gradient descent during the same time functional structures are evolved. Descriptions of the standard GP approach can be found elsewhere (e.g. Langdon, 1998), instead, we will focus below on details of the local learning algorithm.

3.2 LEARNING LEAF COEFFICIENTS

Minimization of MSE is done by a few iterations of a simple gradient descent. At each generation all numeric coefficients are updated several times using the rule:

$$c_k \to c_k - \alpha\frac{\partial\, MSE\,(\mathbf{c})}{\partial\, c_k}, \qquad (2)$$

where α is the learning rate, and k goes over all the coefficients at our disposal. Three important points must be discussed: how to find the derivatives, what the value of α should be, and how many iterations (steps) of descent should be used.

3.2.1 Differentiation

Using both eq. 1 and 2 we obtain:

$$\frac{\partial\, MSE(\mathbf{c})}{\partial\, c_k} = -\frac{2}{N}\sum_{i=1}^{N}\left(d_i - f(\mathbf{x}_i,\mathbf{c})\right)\frac{\partial f(\mathbf{x}_i,\mathbf{c})}{\partial\, c_k} \qquad (3)$$

Thus, an immediate goal is to differentiate any current program tree with respect to any of its leaves. The chain rule significantly simplifies computing $\partial f/\partial c$. Indeed, if $n_j(\cdot)$ denotes node functions, then:

$$\frac{\partial f(n_1(n_2(n_3(...),...),...))}{\partial c_k} = \frac{\partial f}{\partial n_1}\frac{\partial n_1}{\partial n_2}\frac{\partial n_2}{\partial n_3}\cdots\frac{\partial n_r(c_k,...)}{\partial c_k}$$

Therefore differentiation of the tree simply reduces to the product of the node derivatives on the path which starts at the given leaf and ends at the root. It is clear that each term in the product is a derivative of a node output with respect to its arguments (children). If paths from the different leaves share some common part, then corresponding sub-chains in the derivatives are also shared. Computation of such a product in practice depends on the data structure used for the program tree. In simple cases, differentiation uses single recursive postorder traversal together with the actual function evaluation. Derivatives of the program tree with respect to all its leaves can be obtained simultaneously. As soon as an entire sum in eq. 3 becomes known, i.e. derivatives in all training points obtained, one may need an extra sweep through the tree to update the coefficients. In total, the incurred overhead depends on the complexity of node derivatives and the number of leaves. For instance, in our implementation using only an arithmetic functional set, the cost of differentiation was equal to the cost of function evaluation, making the overall cost twice the standard GP cost for the same problem

3.2.2 Learning rate and number of steps

In a simple gradient descent algorithm, the proper choice of learning rate is very important. Too large a learning rate may increase error, while too small a rate may require many training iterations. It is also known that the rule in eq. 3 works better in the areas far from the vicinity of local minima (Reklaitis, 1993). Therefore we decided to make the rate as large as possible without sacrificing quality of learning. After a few trials on the test problem of symbolic regression we fixed the learning rate to the value α=0.5. The same learning rate was used for all other test problems. If the algorithm resulted in an increase in the error of an individual, the training was stopped and no update to the individual's fitness was recorded. However, this problem did not have any impact on overall quality of learning since it happened rarely, approximately 1 out of 10 successful individuals. Moreover, those individuals that had this problem showed an error rate that was typically not reduced by any subsequent application of gradient descent.

This simple local learning rule dramatically improved the fitness of individuals. Figure 2 shows the decrease in MSE for typical individuals. It is important to note that the most significant improvements happened after only the first few iterations of local learning. Note that some individuals were improved by as much as 60% or more. We decided that 3 steps of gradient descent was a good trade off between fitness gain and effort overhead. Again, the number of iterations was never altered afterwards and is used in all our experiments.

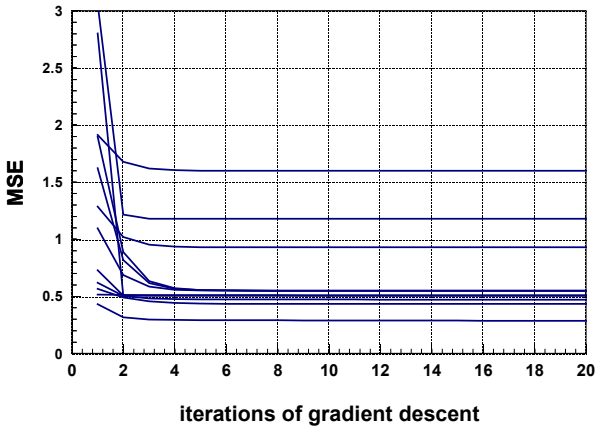


Figure 2: Local learning strongly affects fitness of individuals. Typical learning progress is illustrated using individual from test problem f_2.

4 EXPERIMENTAL DESIGN

The main goal of the empirical study is to compare the performance of the GP with and without learning. Even though an overall speed-up is very valuable, we are also interested in other effects resulting from local learning. These effects have to be properly quantified to shed light on the internal mechanisms of the interaction between learning and evolution. Three major issues are studied:

- Improvement in search speed
- Changes in fitness distribution and selection
- Changes in the functional structure of the programs

4.1 IMPLEMENTATION DETAILS

The driver GP program included following major steps:

1. Initialization of the population using the "grow" method. Starting from a set of random roots, more nodes and terminals are aggregated with equal probability until a specified number of nodes are generated. The total number of nodes in the initial population was chosen to be three times greater than the population size.
2. Fitness evaluation and training (in HGP) of each individual. Mean squared error over the given training set, as defined by eq. 1, serves as an inverse fitness function since we seek to minimize error. This stage includes parametric training in HGP given that the individual has leaf coefficients.
3. Termination criteria check. The number of function evaluations was the measure of computational effort. For instance, every individual is evaluated only once in every GP generation, but three times in every HGP generation if its parameters are trained for three steps.

4. Tournament selection (tournament size = 2) of parents. Pairs are selected at random with replacement and their number is equal to the population size. The better of the two individuals becomes a parent at the next step.
5. Crossover and reproduction. Standard tree crossover is used. Each pair of parents produces two offspring. Mutation with small probability is applied to each. In addition, elitism was always used and the best 10% of the population survive unchanged.
6. Pruning the trees with the size exceeding predefined threshold value.
7. Continue to step 2

4.2 TEST PROBLEMS

Five surface fitting problems were chosen as benchmarks.

$$f_1(x,y) = xy + \sin((x-1)(y+1))$$

$$f_2(x,y) = x^4 - x^3 + y^2/2 - y$$

$$f_3(x,y) = 6\sin(x)\cos(y)$$

$$f_4(x,y) = 8(2 + x^2 + y^2)^{-1}$$

$$f_5(x,y) = x^3/5 + y^3/2 - y - x$$

For each problem 20 random training points (fitness cases) were generated in the range [-3...3] along each axis. Figure 3 shows the desired surfaces to be evolved.

5 EXPERIMENTAL RESULTS

To compare the performance we made experiments with both hybrid and regular GP with the same effort of 30,000 function evaluations in each run. All experiments were done with a population size of 100 and the arithmetic operators {+,-,*,%-protected} as the function set with no ADFs. Initial leaf coefficients were randomly generated in the range [-1...1]. Also, the pruning threshold was set to 24 nodes. If the number of nodes in an individual grew beyond this threshold, a sub-tree beginning at some randomly chosen node was cut from the individual. Each experiment was run 10 times and the MSE value was monitored.

Our main results are shown in Figure 3 and also summarized in Table 1. The success of the hybrid GP is quite remarkable. For all the test problems, the average error of the best evolved programs was significantly smaller (1.5 to 25 times) when learning was employed. The first 20 – 30 generations usually brought most of these improvements. The gap in error levels is wide enough to require the regular GP to use hundreds more generations to achieve similar results. Certain improvements were also observed for the average population fitness, but with lesser magnitude. The similarity of each population's average fitness indicates a high diversity and that not all offspring reach small error values after local learning.

Another set of experiments included extra fine-tuning iterations performed after the regular GP terminates. Again, we run gradient optimization on the population from the last GP generation. Each individual was tuned by applying 100 gradient descent iterations. The results in Table 1 show that this approach is not effective and did not achieve the quality of result found in the HGP. This is a strong argument for Baldwin effect, namely that another factor affecting search speed-up is a change in fitness distribution which directly affects selection outcome. Learning introduces a bias that favors individuals that are more able to adapt to local learning modifications. If we would suppose that the selection bias does not occur, then the hybrid GP would be only a trivial combination of genetic search and fine tuning. However, as we see from the results this is not the case.

We attempted to measure some properties of HGP that would demonstrate this synergy between local learning and evolution.

Table 1. Performance Comparison of Hybrid and Regular GP. All data collected after 30000 f.e. and averaged over 10 experiments.

Test problem	Best MSE		Ave. MSE		Best MSE
	HGP	GP	HGP	GP	GP + fine tuning
f_1	0.009	0.26	0.47	0.80	0.233
f_2	0.075	0.761	1.03	2.18	0.31
f_3	2.32	6.22	5.98	6.59	6.21
f_4	0.64	0.76	4.06	4.41	0.76
f_5	0.097	0.36	0.27	0.78	0.30

First of all, a Baldwin effect in selection would mean that the results of some tournament selections are reversed after local learning. Indeed, local learning adaptable individuals win their tournaments due to improved fitness resulting from gradient descent. These individuals would lose the same tournament in regular GP.

Figure 4 shows both the typical and average percentage of reversed tournaments in the problem f_1. A summary of results for all the test problems is given Table 2.

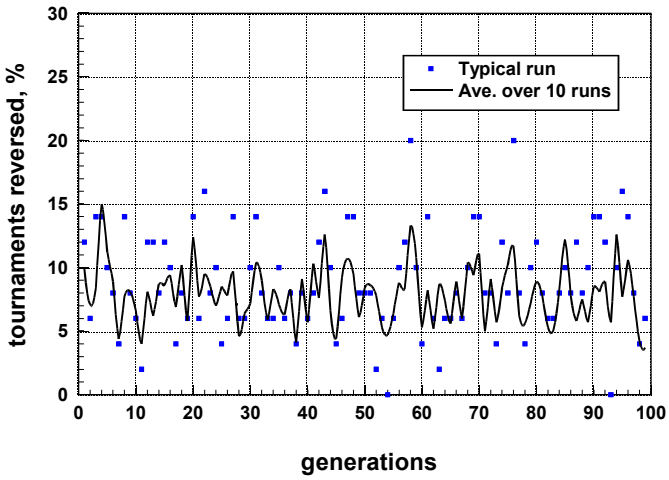


Figure 4: Comparison of Selection Process in HGP and GP. Local learning changes outcome of some tournaments used to select a mating pool.

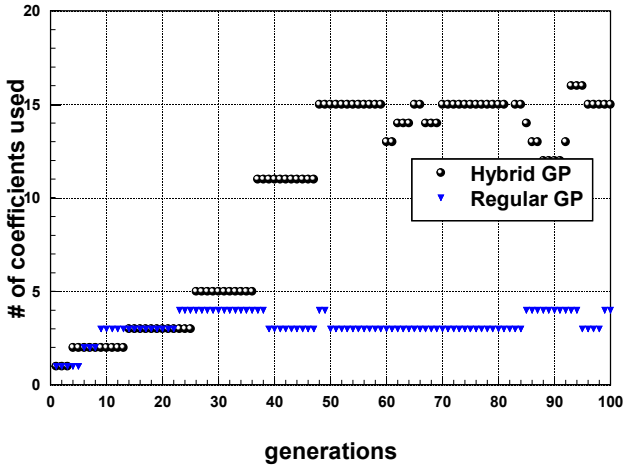


Figure 5: Typical dynamic of number of terminals (numeric coefficients) used by the best program as a function of GP generations (for the test function f1).

It was found that the average percentage of selection changes remains the same during the course of search for all test problems. Such a behavior would be expected if selection pressure pushes offspring that are very adaptable, even when older elite members are almost converged. An empirical measure of this degree of adaptability is provided by the average gain in fitness achieved by newly generated offspring. The values are given in Table 2. We do not include elite members in this statistic to emphasize magnitude of learning from scratch. The average observed drop of MSE is between 12% and 19% on all the test problems.

What exactly makes one program more adaptable than the other? Clearly, it is the functional structure of the program. For example, a program with no numeric leaves cannot learn at all using the gradient local learning method described above. Furthermore, a tree with no terminal arguments (inputs) containing only terminal constants will always produce the same output and will not benefit from local learning. Instead we have tried to understand what characteristics of adaptable programs are unique.

Table 2: Effects of local learning

Test problems	Difference in HGP selection vs. GP in each generation on average, %	Ave. MSE gain for newly generated offspring, %	Complexity of the best programs, #coefficients / #nodes after the same effort (30000 f.e.)	
			HGP	GP
f_1	7.7	16.5	16.0 / 22.4	12.2 / 21.2
f_2	7.1	12.7	16.6 / 23.0	13.7 / 21.8
f_3	8.4	15.1	17.5 / 23.5	11.8 / 20.4
f_4	7.9	18.7	17.3 / 22.9	12.4 / 21.6
f_5	7.4	15.0	17.0 / 23.1	12.9 / 21.8

We have focused on the length (number of nodes) and on the number of coefficients in the best evolved programs (remember, that length had an upper limit too). As Table 2 illustrates both values are noticeably greater for the programs evolved by HGP. This is one illustration of the inductive bias of the hybrid algorithm. More adaptive programs use more coefficients and consequently have lengthier representations. Also, the number of the terminal inputs in HGP results is slightly less, therefore such programs are easier to compute once they are extracted and simplified. Figure 5 shows typical changes in the number of coefficients for a "best" individual on a generational scale for both GP and HGP.

6 CONCLUSIONS

This paper has shown a number of important points. First, that local learning in the form of gradient descent can be efficiently included into GP search. Second, that this learning provides a substantial improvement in both final fitness and speed in reaching this fitness. Finally, the use of local learning creates a bias in the structure of the solutions, namely it prefers structures that are more readily adaptable by local learning. We feel that this approach could have significant impact on practical, engineering problems that are addressed by GP.

References

K. Balakrishnan and V. Honavar (1995). Evolutionary design of neural architectures -- a preliminary taxonomy and guide to literature, *Tech.Rep. CS TR 95-01*, Department of Computer Science, Iowa State University, Ames, IA.

R.K. Belew, J. McInerney, and N.N. Schraudolph (1991). Evolving networks: Using the Genetic Algorithm with connectionist learning. In *Proceedings of the Second Artificial Life Conference*, 511-547, Addison-Wesley.

Anna Esparcia-Alcazar and Ken Sharman (1997). Learning schemes for genetic programming, In Proceedings *Late Breaking Papers at the 1997 Genetic Programming Conference*, 57-65, Stanford University, CA.

Andreas Griewank (2000). *Evaluating derivatives: Principles and techniques of algorithmic differentiation*, SIAM, Philadelphia.

G. E. Hinton and S. J Nowlan (1987). How learning can guide evolution, *Complex Systems*, **1**, 495-502.

John R. Koza, Forrest H Bennett III, David Andre, Martin A. Keane, and Frank Dunlap (1997). Automated synthesis of analog electrical circuits by means of genetic programming, *IEEE Transactions on Evolutionary Computation*, **1**(2), 109-128, 1997.

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

William B. Langdon (1998). *Data Structures and Genetic Programming: Genetic Programming + Data Structures = Automatic Programming*, Kluwer, Boston.

S. Nolfi, J. Elman, and D. Parisi (1994). Learning and evolution in neural networks, *Adaptive Behavior*, **3**(1), 5-28.

Riccardo Poli and Jonathan Page (2000). Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code GP and demes, *Genetic Programming And Evolvable Machines*, **1**(1/2), 37-56.

Una-May O'Reilly and Franz Oppacher (1996). A comparative analysis of GP, In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, ch. 2, 23-44. MIT Press, Cambridge, MA.

G.V. Reklaitis, A. Ravindran and K.M. Ragsdell (1983). *Engineering Optimization: Methods and Applications*, Wiley, New York.

Adrian Thompson (1998). *Hardware Evolution: Automatic Design of Electronic Circuits in Reconfigurable Hardware by Artificial Evolution*, Springer-Verlag: London.

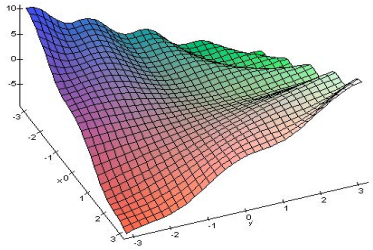
P. Turney (1996). How to shift bias: Lessons from the Baldwin effect, *Evolutionary Computation*, **4**(3), 271-295.

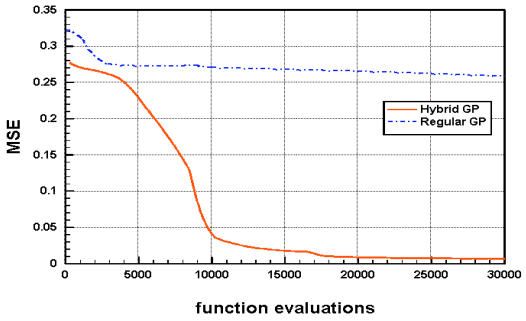
B. Zhang and H. Mühlenbein (1993). Evolving Optimal Neural Networks Using Genetic Algorithms with Occam's Razor, *Complex Systems*, **7**(3), 199 -220.

Andrew Watson and Ian Parmee (1996). Systems Identification using Genetic Programming, In *Proceedings of Int. Conf on Adaptive Computing in Engineering Design and Manufacture*, 248 - 255, ACEDC'96, University of Plymouth, UK.

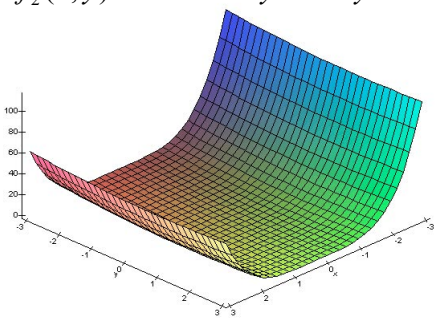
D. Whitley, S. Gordon and K. Mathias (1994) Larmarckian Evolution, The Baldwin Effect and Function Optimization.. In *Proceedings Parallel Problem Solving from Nature, PPSN III*, 6-15, Springer-Verlag.

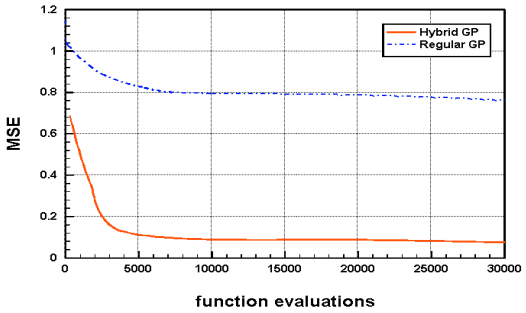
$f_1(x,y) = xy + \sin((x-1)(y+1))$



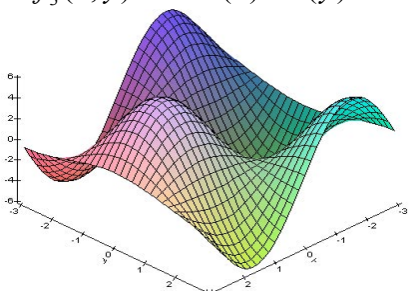


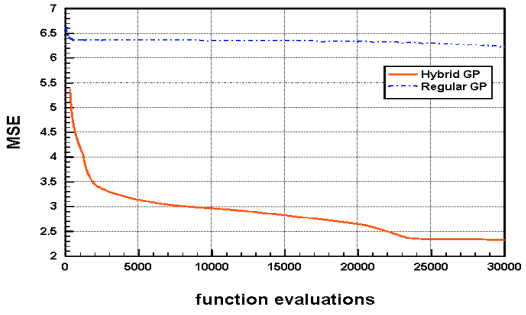
$f_2(x,y) = x^4 - x^3 + y^2/2 - y$



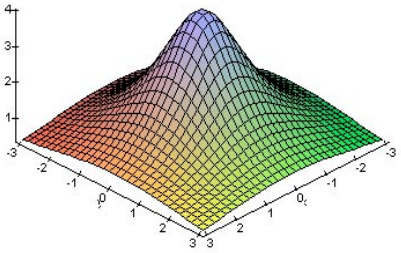


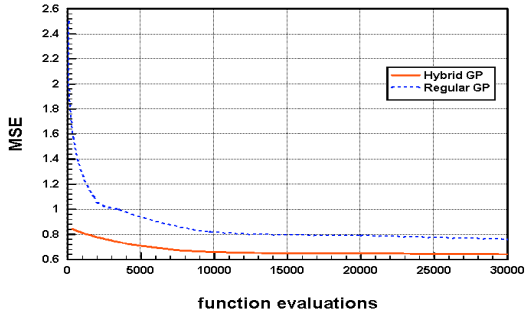
$f_3(x,y) = 6\sin(x)\cos(y)$



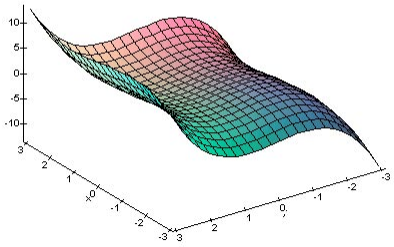


$f_4(x,y) = 8(2 + x^2 + y^2)^{-1}$





$f_5(x,y) = x^3/5 + y^3/2 - y - x$



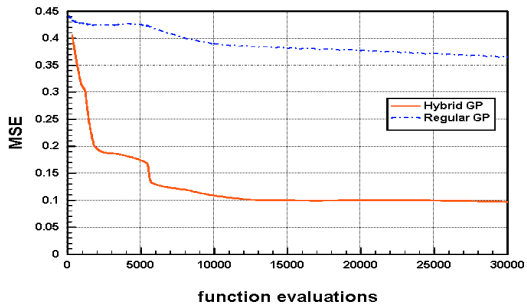


Figure 3: Surface fitting test problems and respective learning curves