Parameter Mapping: a Genetic Programming Approach to Function Optimization

Joao Carlos Figueira Pujol
Centro de Desenvolvimento da Tecnologia Nuclear (CDTN)
Rua Prof. Mario Werneck s/n, 30123 970, Belo Horizonte, Brazil
pujol@cdtn.br

Riccardo Poli
Department Computing and Electronic Systems
University of Essex, Colchester, CO4 3SQ, UK
rpoli@essex.ac.uk

September 4, 2007

Abstract

This paper describes a new approach to optimization that uses a novel representation for the parameters to be optimized. By using genetic programming, the method evolves a population of functions. The purpose of such functions is to transform initial random values of the parameters into better ones. The representation is, in principle, independent of the size of the problem being addressed. Promising results are reported, comparing the new method with differential evolution, particle swarm optimization, and genetic algorithms, on a test suite of benchmark problems.

1 Introduction

In almost any conceivable artifact or system it is possible to identify a set of quantities, called *parameters*, that control the artifact, or that determine its behavior, its shape or its function. Examples of parameters are the diameters of the pipes in a hydraulic system, the coefficients in differential equations representing a metal extrusion process, the weights of a neural network, the resistance, inductance, and capacitance of the components of an electronic circuit, the amounts of flower, sugar, and raisins used to make muffins, etc. *Parameter tuning* is the problem of setting (perhaps at design time) or adjusting an artifact's parameters in a way that guarantees acceptable performance. *Parameter optimization* is a related problem in which the objective is to set the system's parameters so as to maximise its performance.

Parameter Optimization and Tuning (POT) problems present themselves in a variety of important domains ranging from engineering to Artificial Intelligence (AI), from mathematics to the biological sciences, in areas such as function optimization, system identification, AI search, control, machine learning, design, and many others.

In order to solve POT problems using computers, one has to specify a *representation* for the parameters of the artifact/system, and then an *algorithm* that will perform the tuning or the optimization of these parameters. Many POT methods require at least one initial set of parameter values to start with and, after some appropriate processing, they produce a set of adapted parameter values (see Fig. 1). Note that often different methods will produce different parameter values, and not all methods will provide satisfactory results.

Although many algorithms exist that identify optimal sets of parameter values for specific problems, their performance (or even their applicability) is not uniform across all possible problems. Indeed, this is a well-recognised problem across all search and optimization algorithms, as identified by the No Free Lunch (*NFL*) theorem that states that "for any algorithm, any elevated performance over one class of problems is offset by performance over another class" [1]. Therefore, a general algorithm for parameter tuning that is effective and efficient in all applications cannot exist. Since an *NFL* result exists for representations [2], different parameter representations do not offer equivalent performance either.

NFL results are a clear crystallisation of a much older, very widespread piece of wisdom in AI that if one selects the right representation and search strategy, a complex problem may become much more easily solvable. This is because the right representation (for that problem) can dramatically reduce the size of the search space without correspondingly reducing the size of the solution space, while a good search strategy (for that problem) will have a favorable bias that quickly leads the search in the direction of solutions. However, deciding which representation and search strategy are most appropriate is still something of a black art. In addition, as illustrated in Fig. 2, only a tiny fraction of the huge space of all possible algorithms and representations for POT has been explored to date, and there are today a variety of important problems which cannot be solved satisfactorily with current methods. Many POT problems that will be encountered in the future will be in this category.

A lot of research has gone into search, learning, and optimization algorithms [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14], but by no means it can be claimed that a clear understanding of the space of all such algorithms has been achieved. Also, insufficient research has been devoted to the question of choosing a good representation for the parameters of the system, the simplest possible one not necessarily being the best for POT. Indeed, most of the methods of parameter tuning proposed in the literature use *simple arrays of numbers* as a representation for the system's parameters. Overall performance differences are then the result of algorithmic differences/improvements only. This is not satisfactory because major speedup is often attainable through a change in the representation of the solutions rather than in the search algorithm.

Naturally, researchers have considered the problem of representations in symbolic AI. However, many (probably most) POT problems of practical interest involve real-valued parameters and, therefore, are outside this domain. Also, in the area of evolutionary computation there has been some interest in different ways of representing parameters. In most cases the research focus has been the so-called genotype/phenotype mapping [15, 16, 17], but rarely has this research considered real-valued mappings. An exception is the work on evolution strategies, where mutation can be based on a N-dimensional Gaussian distribution, which effectively corresponds to performing a linear transformation of the original representation, followed by *1*-dimensional mutations [18].

As illustrated in Fig. 1 most POT methods require an initial set of parameter values. These values are often selected either randomly or by using some heuristic that is known to produce reasonable guesses. With very few exceptions, the transformation performed by the POT method will be iterative, involving, at each iteration, the production or informed guessing of (at least) one new set of parameter values. Iteration after iteration these values are typically closer and closer to satisfying an acceptability/optimality stopping condition.

Almost invariably this process requires the interaction between the POT method and the system/artifact whose parameters are being tuned or optimized (or an accurate model of it), and the transfer of information from the latter to the former, as shown in Fig. 3. This feedback information can take a variety of non-mutually-exclusive forms: a) it can be gradient information indicating a good direction in which to search for better solutions, b) it can be error information

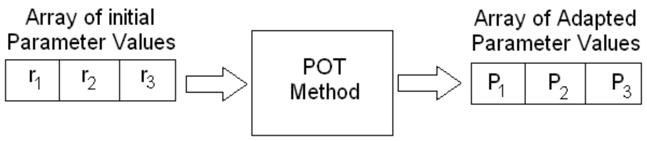

Array of initial Parameter Values
r1
r2
r3
POT Method
Array of Adapted Parameter Values
P1
P2
P3

Figure 1: Transformation performed by typical POT methods.

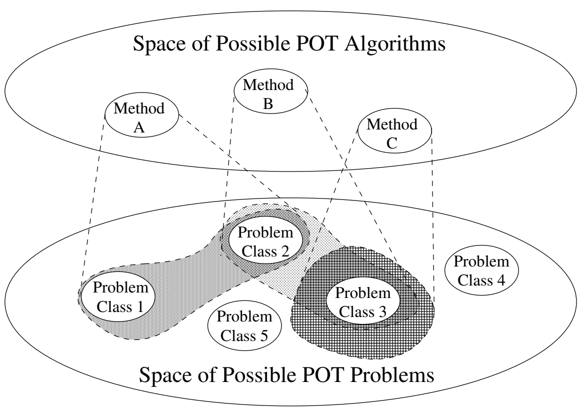


Figure 2: The space of all possible POT methods (top) is vast and largely unknown. The ellipses represent the few known POT methods currently available. Similarly the space of all possible POT problems of practical interest (bottom) is very large. Here the ellipses represent existing POT problems. The shadows cast by the methods onto the problem space represent the sets of problems for which each method provides good performance. Note the presence of problems for which no known method provides acceptable performance. Future algorithms and representations for POT might cast a shadow on (i.e. solve) such problems.

regarding for which parts of the system the current parameter values are acceptable and for which parts they are not, c) it can be a number providing an overall performance measure for the current parameter values, etc.

Inside POT methods, in some cases, the current parameter values are transformed into some different form for processing. Common transformations include rotations, translations, and scaling of the parameters' reference system, or other linear transformations such as the Fourier transform, or other orthogonal basis transformations [19]. Only very rarely non-linear transformations are performed.

In this work a new approach for parameter optimization based on Genetic Programming (GP) [20, 21] is described. The new method introduces a new parameter representation that is adaptive and independent of the number of parameters to be optimized. This work is organized as follows: in Section 2, the main ideas behind the new method are described, in Section 3 the new approach is described in detail, in Section 4 results of experiments are reported and discussed, and Section 6 concludes and provides suggestions for further research.

2 Parameter Mapping Approach

Our method to solve $POTs$ is called *parameter mapping approach* (PMA). The basic idea behind PMA is to use functions represented as computer programs, to represent both the algorithm and

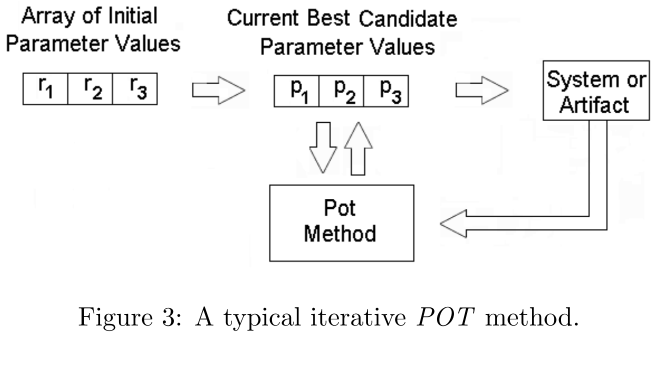

Array of Initial Parameter Values
r_1 r_2 r_3
Current Best Candidate Parameter Values
p_1 p_2 p_3
System or Artifact
Pot Method

Figure 3: A typical iterative POT method.

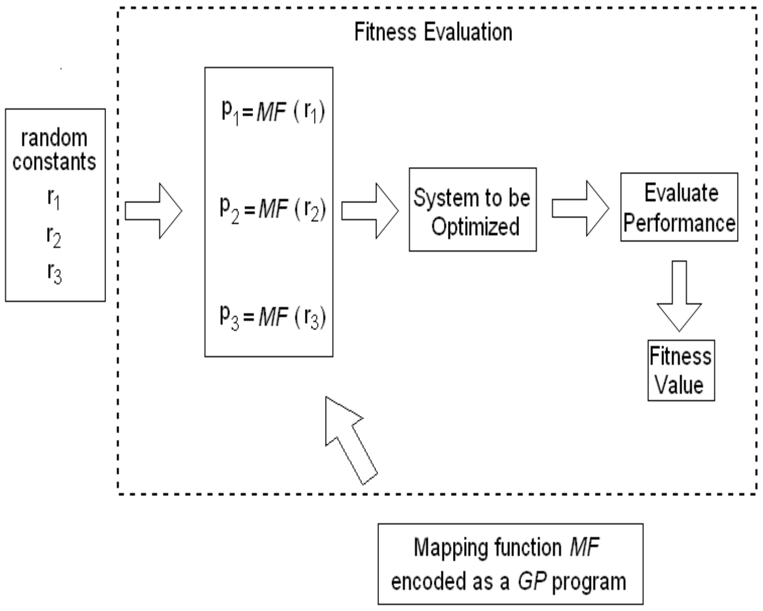


Figure 4: Fitness evaluation in *PMA* for a system with three parameters. r_1, r_2, and r_3 are random values generated initially. They are used as input to the *GP* mapping function *MF*. *MF* is applied to each one of them, producing as output the adapted values, p_1, p_2, and p_3, respectively. The adapted values are introduced into the system being optimized, the performance of the system is evaluated and a fitness value is produced. This procedure is repeated for every individual of the population by using the same random values.

the parameters to be optimized at the same time, and to optimize such programs using *GP*. In *PMA* a population of *GP* programs represents a set of trial mapping functions which transform initial sets of parameter values into adapted ones. Under the guidance of an appropriate performance measure, *GP* iteratively improves such functions, until a mapping function is evolved which can satisfactorily transform a given initial set of parameter values into a good set of values. Fitness evaluation in *PMA* is illustrated in Fig. 4. A mapping function receives as input a random number representing a raw value of a parameter, and returns as output an adapted value. This procedure is repeated for each parameter being optimized. The function to be optimized is then evaluated by using the adapted values. The random values are created in generation zero, and do not change in the course of the evolutionary process. Moreover, all individuals of the population are evaluated by using the same initial random values.

It is interesting to note that, as the same *GP* function is used to map each parameter, the result of the evolutionary process performed in *PMA* can be interpreted as a non-iterative *POT* algorithm that, given the initial set of random parameters, is able to map it into the target set of optimized ones in one pass. A second interpretation is to consider the fact that in one way or another the *GP* function is a representation for a whole set of parameter values since, given as input each component of the initial set of random values, it produces as output a corresponding optimum value. [1] So, in the first interpretation the evolutionary process performed in *PMA* is searching the space of possible non-iterative *POT* methods for a perfect direct parameter optimization algorithm. In the second interpretation, the evolutionary process performed in *PMA* is searching for a holistic representation for the target set of parameter values. Naturally, these two viewpoints are entirely equivalent.

In previous research [22, 23], an early version of *PMA* termed *weight mapping approach (WMA)*

[1]The random values act as keys for the retrieval of the values encoded (represented) in the *GP* function, in exactly the same way in which integers are used as keys to retrieve the values stored in a vector of numbers.

was applied to the training of neural networks (*NNs*). This was inspired by the observation that the process of training the weights of *NNs* is simply a (usually iterative) transformation from a set of initial (typically random) raw weights to a set of trained weights, and that that transformation is a function. This function does not have to be necessarily implemented in the iterative gradient-descent way typical of *NNs* learning rules. So we decided to use genetic programming to evolve such a function by using the output error of the neural network as fitness function. The experiments revealed that *PMA* was more effective than some traditional approaches.

WMA has the following important features that are also expected to be shared by the more general form of *PMA* introduced in this paper:

- The use of a parameter mapping function does not impose strong constraints on the value of the parameters and, consequently, *GP* can discover and use arbitrary or nearly arbitrary regularities in the parameter values appropriate for a problem.
- The representation exploits this ability in conjunction with a good set of search operators to beneficially bias the search.
- Within each individual the parameters to be optimized are not individually encoded: they are represented by a single function. Consequently, the representation does not necessarily grow quickly with the size of the problem of interest (which unavoidably would lead to the exponential growth of the search space and poor scalability).

Note that while WMA was used to train neural networks, in this paper we don't apply its generalisation, PMA, to the same task. Instead, we will use PMA for function optimization. So, in the rest of the paper we will never refer to training and test sets of examples, but simply of fitness functions that we want to optimize.

3 *PMA* in detail

3.1 Parameter encoding

To summarize, the basic idea of *PMA* is to use a set of random constants to represent raw values of the parameters being optimized. These values are then used as input to a *GP* tree. The *GP* tree encodes a function that transforms the raw values into adapted ones. This procedure is applied to every parameter being optimized, using as input the raw value of each parameter. So, for a generational system, the fitness evaluation phase in *PMA* could be pseudo-coded as follows:

```
BEGIN
  Let $p_1, \dots, p_i, \dots, p_N$ be $N$ parameters to be optimized;
  Let $r_1, \dots, r_i, \dots, r_N$ be $N$ random constants;
  Let $TSET = \{x\}$ be the GP terminal set;
  FOR every GP tree in the population DO
      Set $i = 0$;
      REPEAT
          Set $i = i + 1$;
          Assign $r_i$ to the variable $x$;
          Evaluate the GP tree using $x$ as input;
          Assign the output of the GP tree to $p_i$;
      WHILE($i \leq N$)
      Use $p_1, \dots, p_i, \dots, p_N$ to evaluate the performance
      of the system being optimized and assign a fitness
      value to the GP tree.
END
```

Note that the random constants representing raw parameter values are actually fed, one by one, into the variable x used as the input to our *GP* programs. Also, naturally, *PMA* allows the *GP* terminal set to include numerical constants or an ephemeral random generator, in addition to the variable x. So, to avoid confusion, in the following we will always talk about *GP* variables and *GP* constants, when we refer to the *GP* terminal set.

Although the raw parameters must be constant, they do not necessarily have to be random. For example, suppose we want to minimize the three-variable function $p_1^2 + p_2^2 + p_3^2$. In this case, the variables p_1, p_2, and p_3 are the parameters to be optimized, and it is only natural to use randomly generated numerical constants as their raw values. However, it is also possible to use as input to the *GP* tree the values *1*, *2*, and *3* that identify the parameters. By so doing, the mapping function encoded in the *GP* tree could be interpreted as a memory or a *hash* table, instead of a learning procedure. The "stored" adapted values would be retrieved by using the identifier as input to the *GP* tree.

Whether they are randomly chosen or assigned via some rule, the raw values used as input to the GP tree should all be different. This is because identical inputs produce identical outputs in any computer program, and, so, identical raw values are mapped to identical parameters by the function represented by a GP tree. However, in order to solve a problem different parameters may, in general, need to be set to different values.

3.2 Multivariate encoding

In WMA, a single *GP* variable was used to represent the raw values of the parameters. Although it looks natural to use a one-to-one correspondence between raw parameter values and adapted ones, it does not have to be so. A many-to-one representation, i.e., a sparse representation, can be used, where a single parameter may be encoded by more than one raw value. The only change that is required is the use of more than one *GP* variable in the terminal set. That is, if we want to use K raw values to represent each parameter, we will need K *GP* variables in the terminal set (see Fig. 5).

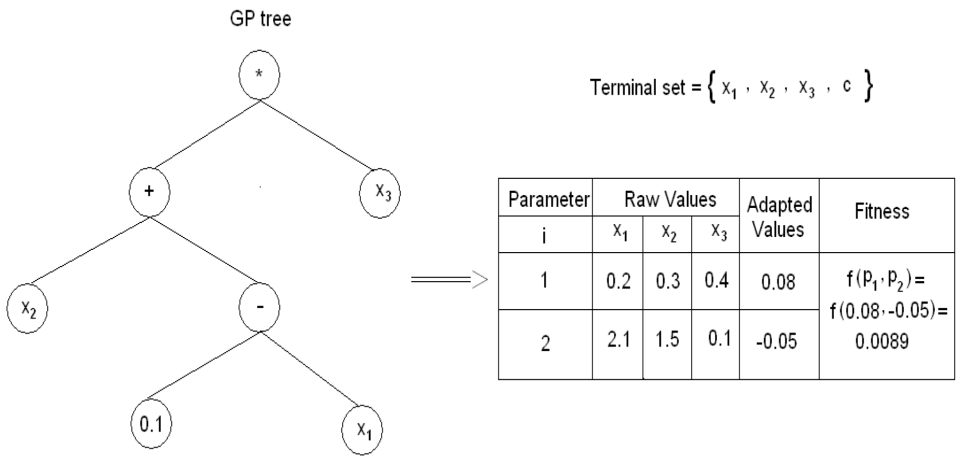
There is no restriction whatsoever as to the number of raw values used to encode each parameter being optimized. Tens or hundreds of them can be used to that purpose. However, one should be mindful of the fact that not all raw values are necessarily used. They are all present in the terminal set as *GP* variables and, depending on the initialization procedure of the *GP* population, different *GP* trees may include different *GP* variables. Moreover, in the course of the evolutionary process, some of them may disappear from the population and not be used at all. So formally, the mapping function *MF* is defined as *MF*: $\mathbb{R}^S \rightarrow \mathbb{R}$, where $S \leq K$. This increases the ability of the evolutionary process to evolve an appropriate set of values to be used as input to the *GP* trees. For example, the set of raw values may happen to include values that are close or even equal to each other. Should the adapted parameters have quite different values, a single-variable mapping function would have trouble in computing them.

The benefits of using a multivariate encoding vs. the original single-variate one are illustrated in Section 4.5.3 where we vary the number of raw values used to encode each parameter and see what effects that has on performance.

3.3 *GP* implementation

3.3.1 Terminal set

The *GP* constants are typically taken from the range [-1,+1]. If a single *GP* variable was used to represent the parameters, it would make sense to initialize the raw value of the parameters within their respective range, since these values are essentially random guesses for the corresponding adapted values. So, if p_i should be in the range $[a_i, b_i]$, we could initialize the corresponding raw values in the same range. With a multi-variable encoding this interpretation disappears, the raw values can be also initialized within the range [-1,+1], and the domain of the mapping function *MF* becomes $[-1,+1]^S$. This makes possible to abstract from the range of the parameters, by using



Parameter	Raw Values			Adapted Values	Fitness
i	x_1	x_2	x_3		
1	0.2	0.3	0.4	0.08	$f(p_1, p_2) =$ $f(0.08, -0.05) =$ 0.0089
2	2.1	1.5	0.1	-0.05	

Figure 5: Example of sparse representation. A function $f(p_1, p_2) = p_1^2 + p_2^2$ has to be minimized. Each parameter p_i is encoded by using three *GP* variables x_1, x_2, and x_3. These three variables are included in the terminal set. The terminal set also includes an ephemeral random number generator c to create *GP* constants. The *GP* tree in the picture encodes the mapping function represented by the algebraic expression $(x_2 + 0.1 - x_1) \times x_3$. To compute adapted values for parameter p_1 and p_2, raw values (randomly initialized at the beginning of each run, and kept constant thereafter) are assigned to x_1, x_2, and x_3. The adapted values produced by the *GP* tree are then used to evaluate the function *f*, which is used to compute the fitness of the individual represented by the *GP* tree.

a *change of coordinates* that maps all of them to the same range. This issue is further addressed in the description of the experiments.

3.3.2 Function set

This set included four 4-arity operators: $SUM = x \times y + u \times v$, $SUB = x \times y - u \times v$, $MUL = (x+y) \times (u+v)$, and $PDV(x+y, u+v)$, where x, y, u, and v are the arguments for these operators. *PDV(num,den)* is the protected division, which returns *num* if *den* is zero. With this choice of function set, except where divisions by 0 are involved, the function encoded in any GP tree is equivalent to a polynomial or a ratio of polynomials in the input variables.

The reason for using 4-arity operators is that this breaks the symmetry of the addition and multiplication arithmetic operators, thus reducing the possibility of the so-called *permutation* or *competing convention* problem [24, 25]. Indeed, in preliminary experiments these operators demonstrated to be superior to the ordinary arithmetic operators [26, 27]. The idea of using these operators was also inspired by neural networks. In the first two operators, the values of y and u may be seen as *weights* associated to inputs x and y, respectively.

3.3.3 Selection and crossover

A steady state form of *GP* was implemented, with tournament selection and *non-destructive headless chicken crossover* [28]. In this form of crossover, standard subtree crossover is performed between a parent selected from the population and a randomly created individual (the new individual is created by the same procedure used to create the initial population). The resulting offspring is inserted in the population if it is superior to the parent, otherwise it is discarded. This procedure is repeated a number of times equal to the size of the population.

In *tournament* selection, by changing the size of the tournament one can impose more or less selection pressure. This may affect the convergence speed of the evolutionary process considerably.

A strongly competitive strategy may quickly converge to a solution, but it is also prone to get trapped in local optima. On the other hand, a small tournament may lead to a low convergence rate. The result of a large tournament size combined with a hill-climbing replacement strategy should be the perfect formula for poor performance. However, in extensive experiments (not reported) we found that these ingredients matched nicely with *headless chicken crossover*, probably because this form of crossover is actually a macro-mutation in disguise, and it is very effective in introducing new genetic material into the evolutionary process. So, in this work, we decided to use tournament selection with large tournament sizes.

4 Experiments

4.1 Test functions

To explore its potential, *PMA* was applied to minimize a set of benchmark functions $f_i : \mathbb{R}^n \rightarrow \mathbb{R}$, the well-known De Jong set as described in [12]. This set of functions was chosen because it has been extensively used as a test suite for optimization algorithms. In addition, it includes functions of varying degrees of difficulty, so that one can compare the performance of different optimization methods. Three other functions taken from [29, 30] were also included. The whole test suite is as follows:

i. $f_1(\mathbf{x}) = \sum_{j=1}^{3} x_j^2$.

   The range of the search is $-5.12 \leq x_j \leq 5.12$. The global minimum is zero, at $\mathbf{x} = \mathbf{0}$.

ii. $f_2(\mathbf{x}) = 100 \times (x_1^2 - x_2)^2 + (1 - x_1)^2$.

   The range of the search is $-2.48 \leq x_j \leq 2.48$. The global minimum is zero, at $\mathbf{x} = \mathbf{1}$.

iii. $f_3(\mathbf{x}) = 30 + \sum_{j=1}^{5} \lfloor x_j \rfloor$.

   The range of the search is $-5.12 \leq x_j \leq 5.12$. The global minimum is zero. This value is achieved if $-5.12 \leq x_j < -5$.

iv. $f_4(\mathbf{x}) = \sum_{j=1}^{30} (x_j^4 \times j) + \nu$.

   The range of the search is $-1.28 \leq x_j \leq 1.28$. $\nu \in [0, 1)$ is a noise term defined by a random variable with a uniform distribution, and expected value $\overline{\nu} = 0.5$.

   The noise-free component of the function has a global minimum of zero at $\mathbf{x} = \mathbf{0}$. With the addition of noise, $f_4(\mathbf{0}) = \overline{\nu}$ on average.
   The standard De Jong implementation introduces Gaussian noise with zero mean and unitary standard deviation in the function. It has been argued [12] that, in this case, the minimum is ill-defined or is not defined at all, as a good set of values for the parameters may not be recognized as such by a searching algorithm. This might give an unfair advantage to a slower algorithm over a faster one. Therefore, it was suggested that the Gaussian noise should be replaced by a random noise generated by a uniform distribution and constrained

to the range [0,1). This is the form implemented here.[2]

v. $f_5(\mathbf{x}) = [0.002 + \sum_{i=1}^{25} g_i]^{-1}$

$g_i = [i + (x_1 - a_i)^6 + (x_2 - b_i)^6]^{-1}$

$a_1 = -32, a_2 = -16, a_3 = 0, a_4 = 16, a_5 = 32$
$a_i = a_{i-5},\ i = 6, \ldots, 25$
$\mathrm{b}_1 = \mathrm{b}_2 = \mathrm{b}_3 = \mathrm{b}_4 = \mathrm{b}_5 = -32$
$\mathrm{b}_i = \mathrm{b}_{i-5} + 16,\ i = 6, \ldots, 25$
The range of the search is $-65.536 \leq x_1, x_2 \leq 65.536$. The global minimum is $f_5(\mathbf{x}) \approx 0.998\ 004$, at $x_1 = x_2 = -32$.

vi. $f_6(\mathbf{x}) = \sum_{j=1}^{4} g_j$.

$$g_j = \begin{cases} 0.15 \times (h_j)^2 \times d_j & \text{if } \|x_j - z_j\| < 0.05 \\ d_j \times x_j^2 & \text{otherwise} \end{cases}$$

$z_j = \left\lfloor \left\| \frac{x_j}{0.2} \right\| + 0.499\ 99 \right\rfloor \times sign(x_j) \times 0.2$
$h_j = z_j - 0.05 \times sign(z_j)$
$d_1 = 1, d_2 = 1000, d_3 = 10, d_4 = 100$

This is the Corona's parabola [29]. The range of the search is $-1000 \leq x_j \leq 1000$. The global minimum is zero, when $\|x_j - z_j\| < 0.05$.

vii. $f_7(\mathbf{x}) = \sum_{j=1}^{10} \frac{x_j^2}{4000} - \prod_{j=1}^{10} \cos\left(\frac{x_j}{\sqrt{j+1}}\right) + 1$.

This is the Griewangk's function [29]. The range of the search is $-400 \leq x_j \leq 400$. The global minimum is zero, at $\mathbf{x} = \mathbf{0}$.

viii. $f_8(\mathbf{x}) = \sum_{j=1}^{n} \left(-x_j \times \sin\left(\sqrt{\|x_j\|}\right)\right)$.

This is the Schwefel's function [30]. The range of the search is $-500 \leq x_j \leq 500$. The global minimum is achieved at $\mathbf{x}_{min} \approx \mathbf{420.97}$.

4.2 Fitness evaluation

Fitness was defined as $\|f - target\|$, where f is the computed value for the function under optimization, and *target* is the global minimum within the range specified in the domain of the function. For most of the experiments, the search for the minimum was interrupted when the

[2]To make harder to find the optimum, Storn and Price [29] suggested that the noise term should be included within the summation. Although this would make things harder, such an implementation of the perturbation term does not qualify as noise, as the noise should be small compared to the signal (in this case, the value of the noise-free part of the function being minimized). Therefore, the noise term was not included within the summation, it was only added to the summation result.

fitness of the best individual of the population was better than 10^{-6}. In the experiments with the Schwefel's function, due to the growing difficulty to achieve a solution as the size n of the function increases, the convergence criterion was relaxed to better than 10^{-2}, in order to collect statistically meaningful data.

The fourth function requires some explanation. In the case of this function, *fitness* was computed as $\|f - \overline{\nu}\|$, and the search was interrupted when $f < \overline{\nu}$. So that, by using a threshold to verify convergence, the search is for a minimum on a statistical sense. Alternatively, the noisy function could have been used to compute fitness and, consequently guide the evolution, whereas the noise-free part of the function would check for convergence. However, in practical applications it is not possible to remove the noise.

4.3 Comparison with other methods

The performance of the *PMA* approach was tested on the functions described in Section 4.1, and the results were compared to those obtained with three other methods: Differential Evolution [29, 32], Particle Swarm Optimization [33] and a real-valued implementation of a genetic algorithm. These methods have been selected because they have been reported to be the state of the start in function optimization.

4.3.1 Differential Evolution

Differential evolution (*DE*) is reported to be a very effective optimization algorithm for real-valued problems. Its standard implementation includes three free parameters: size of the population, crossover probability and a scaling factor. These parameters have to be properly chosen to achieve a good performance. Storn and Price [29] report good results by using a different set of values for each function in the test suite. This is the parameter setting used in the experiments with functions *i*, *ii*, *iii*, *v*, *vi*, and *vii*. As the implementation of the fourth function differs from that by Storn and Price, a different parameter setting was selected to obtain optimal performance on the optimization of this function.

4.3.2 Particle Swarm Optimization

Particle swarm optimization (*PSO*) is another recent development in optimization inspired by biology. In this method, an individual of the population can move to another position of the search space guided by two different motivations: a cognitive one, that leads the individual to move to its previous best position, and a social one, that drives the individual to the best position ever encountered by the population. Both impulses are modulated by two parameters: a cognitive factor and a social factor, respectively. Other parameters include: the size of the population and the maximum speed with which the individual is allowed to move from one position to another. Most implementations of *PSO* set the values of the cognitive factor and the social factor to 2 [33]. As a consequence, only two parameters are left to be specified. These were set to give the optimal performance for each function being optimized.

4.3.3 Genetic Algorithm

Genetic Algorithms (*GAs*) are a very powerful and popular optimization tool [34, 35]. Several *GA* implementations, with different coding schemes and different selection, crossover and mutation operators, have been proposed [36, 37]. As this work focused on continuous parameter optimization, we opted for a real-valued representation, where the parameters to be optimized were encoded as a real-valued vector. A steady-state *GA* was implemented by using tournament selection with a specified tournament size (*TS*). Crossover was carried out with probability *CP*, by using *extended intermediate recombination* [30], where an offspring was generated as $r \times parent_1 + (1-r) \times parent_2$. r is a random number generated anew for each component of the vector by a uniform distribution in the range $(-\alpha, 1+\alpha)$. α is a constant set to a value of 0.2 in all experiments. This operator is also known as blend crossover [31]. Mutation was invoked with a probability $1 - CP$. Mutation was implemented as random noise added to each component of the parent vector with a probability *MP*. Noise was introduced by multiplying the component of the vector by $(1+\beta)$, where β was a random number generated by a uniform distribution in the range $(-NS, NS)$. Whenever the offspring was superior to the parent with the worst performance, the offspring replaced it, otherwise it was discarded. The parameter setting of the *GA* was also selected to achieve optimal performance for each benchmark function.

4.4 Description of the experiments

To evaluate the performance of *PMA*, 100 independent runs were carried out for each function on the test suite. All runs were carried out up to a maximum of 2×10^6 fitness evaluations. If in a particular run, an algorithm failed to converge to a solution within the maximum alloted number of fitness evaluations, the search was interrupted, and the run was labeled as unsuccessful. The average number of fitness evaluations (*NFE*) over the successful runs was computed (the evaluation of those individuals discarded by the *non-destructive headless chicken* crossover were also included), and also the number of successful runs (*HITS*). (By using *NFE* and *HITS* combined with the maximum number of fitness evaluations alloted to each run, it is possible to obtain additional statistical information about performance.)

For the *PMA* runs, the *GP* constants and the raw values of the parameters to be assigned to the *GP* variables were randomly initialized within the range (-1,+1) by using a uniform distribution. One hundred *GP* variables were utilized to encode each parameter (see Section 4.5.3 for a discussion about the parameter setting in *PMA*). For each parameter, the output of the *GP* tree was then transformed to the appropriate search range defined for each benchmark function (see Section 4.1) by using a linear transformation given by:

$ADAPT = LOWER + OUT \times (UPPER - LOWER)$,
where:
ADAPT is the adapted value of the parameter;
LOWER is the lower limit of the search range;
UPPER is the upper limit of the search range;
OUT is the output of the *GP* tree.

This normalization procedure for *PMA* is useful for two reasons:

- The parameters being optimized may have very different ranges, and it would be difficult to encode them with a single function without a normalization procedure. As *DE*, *PSO* and *GAs* deal with the numerical values of the parameters directly and individually, for them it is enough to initialize the parameters within the appropriate range.
- If a complex *GP* tree has to be evolved to represent parameters of very different ranges, numerical instability may arise.

Note that the output of the *GP* tree is not constrained to the range (-1,+1). As a consequence, we do not constrain the adapted values of the parameters to the range (*LOWER*, *UPPER*). Of course, if desired, the range of the search can be constrained by applying a squashing function to the output of the *GP* tree. Indeed, in the experiments with the third benchmark function, in order to make possible to define a global minimum, the search was constrained within the specified search limits by using a logistic function.

With *DE*, *PSO*, and the *GA*, the individuals of the population were randomly initialized in the range defined by the *LOWER* and *UPPER* values, by using a uniform distribution. And, in the case of the third benchmark function, the values generated by these methods were simply truncated if these limits were exceeded.

Four sets of experiments were carried out. In the first set of experiments, seven of the standard functions described in Section 4.1 were used to compare the performance of *PMA* to that of the other methods. However, although the benchmark functions are multi-dimensional, all dimensions

Table 1: Results for *DE*, *PSO*, *GA* and *PMA* in the first set of experiments. Standard deviations are given in brackets. The parameter settings for each algorithm and function are given in Table 2.

Fitness function	DE		PSO		GA		PMA	
	NFE	HITS	NFE	HITS	NFE	HITS	NFE	HITS
f_1	487	98	4,164	100	1,401	100	905	100
	(49)		(851)		(359)		(1,162)	
f_2	1,100	94	4,503	100	7,189	100	3,653	100
	(1,441)		(1,786)		(4,231)		(6,482)	
f_3	156	100	52	100	509	100	30	100
	(48)		(11)		(214)		(35)	
f_4	2,688	100	461	100	2,918	100	92	100
	(534)		(156)		(1,053)		(56)	
f_5	706	100	3,262	100	4,174	100	413	100
	(129)		(1,191)		(1,660)		(401)	
f_6	1,235	88	1,810,042	66	10,203	100	9,639	100
	(103)		(115,044)		(9,933)		(12,235)	
f_7	35,850	100	2,000,000	no hit	102,398	100	3,654	100
	(1,479)		(0)		(12,027)		(7,428)	

Table 2: Parameters used in the first and second sets of experiments (rows 1–7) and in the fourth set of experiments (last row).

Fitness function	DE			PSO		GA					PMA			
	POP	SF	CP	POP	MSP	POP	CP	MP	NS	TS	POP	FC	TS	NV
f_1	10	0.5	0.3	10	0.01	500	0.9	0.1	0.1	20	100	0.9	100	100
f_2	6	0.95	0.5	10	0.01	500	0.9	0.1	0.1	10	50	0.9	50	100
f_3	10	0.8	0.3	20	10	200	0.5	0.1	0.5	100	50	0.9	50	100
f_4	10	0.75	0.5	5	0.1	100	0.3	1	0.1	5	100	0.9	100	100
f_5	15	0.9	0.3	150	1.0	1,000	0.9	0.1	0.1	20	50	0.9	50	100
f_6	10	0.4	0.2	200	0.1	500	0.5	0.1	0.1	20	200	0.9	100	100
f_7	30	1.0	0.3	200	0.01	2,000	0.8	0.2	0.1	2	100	0.9	100	100
f_8	20	0.5	0.5	50	1.0	2,000	0.5	0.1	0.5	100	100	0.9	100	100

have the same minima, and this might be seen as a bias in favor of *PMA*. Therefore, a second set of experiments was carried out, where the benchmark functions were modified to remove this possibility. In a third set of experiments, the influence of the parameter setting on the performance of *PMA* was investigated. Finally, in a fourth set of experiments, the scalability of *PMA*, *DE*, *PSO* and the *GA* was compared, by using the Schwefel's functions of different sizes.

In all cases we evaluate performance using standard measures (namely computational effort and success rates). However, we do not report an analysis of the evolved GP trees, which is often customary in the GP literature. In this work evolved trees are often complex structures with hundreds of nodes, which, even after simplification, are typically equivalent to large and obscure rational functions.

4.5 Experimental results

4.5.1 First set of experiments

The results of this set of experiments are summarized in Table 1. The corresponding parameters are reported in Table 2. (The meaning of the control parameters of *PMA* is explained later in the text.)

In terms of number of fitness evaluations to find a solution, *PMA* outperformed *DE* in 4 out of 7 of the test functions, and was superior to *PSO* and to the *GA* in all of them. In terms of convergence to a solution, *PMA* and the *GA* were able to converge in 100% of the runs. *DE* did not achieve 100% convergence for all functions, whereas *PSO* failed to find any solution to the 7th function. On other hand, in a few experiments *PMA* showed larger standard deviations for *NFE* as compared those of the other methods. This is more apparent for the 6th test function, where the standard deviation was even larger than the mean. We believe this to be a consequence of the initialization procedure. In the case of *DE*, *PSO* and the *GA*, the individuals of the population were always initialized within a specified range appropriate for each test function. As a consequence, different runs may show similar convergence patterns. This is not the case with *PMA*, where the population of *GP* functions can produce values very far from the desired range. So that, in a particular run the population may quickly converge to a solution, while in others it may take much longer, leading to a large standard deviation. But we do not consider this feature as a drawback of *PMA*, since in most applications one does not know the target values of the parameters. Moreover, as mentioned in Section 4.4, one can bypass this problem by applying a squashing function to the output of the *GP* tree, to constrain its output to a desired range.

Although outstanding, the results of *PMA* should not lead one to jump to conclusions. For most of the functions optimized, the optima are either ***0*** or ***1***, and these values are extremely easy to generate by using the *PMA* representation. For example, a *GP* tree encoding the algebraic expression $x - x$, will always produce *0* as a result. Similarly, the algebraic expression $x \div x$ will always produce *1*, provided that x is not zero. Although this is a characteristic of the parameter mapping approach, it might be considered an unfair advantage (for other problems this might be a hindrance). Therefore, to eliminate this advantage, modifications were introduced in the original benchmark functions, and a second set of experiments was carried out.

4.5.2 Second set of experiments

In the second set of experiments, the degree of difficulty was increased by introducing a random shift in each dimension of the functions $f(\mathbf{x})$ being optimized. A different random shift was applied to each dimension. So that the functions are now $f(x_1 - r_1, \ldots, x_i - r_i, \ldots, x_N - r_N)$, where r_i are random constants in the range (-1,+1), generated by a uniform distribution. As a consequence, the optimal values are not *0s* and *1s* anymore, they are shifted by the amount provided by the random shifts. For each dimension, the lower and upper boundaries of the search range were also shifted: by -1 if the corresponding shift was negative, and by +1 if it was positive. The boundaries were shifted to guarantee that a solution could be found, without actually using information about the value of the random shifts. The same parameter settings of the first set of experiments were used in this second set of experiments. (No attempt was made to fine tune the parameter settings again, neither for *PMA* nor for the other algorithms.) The results of this set of experiments are summarized in Table 3.

As expected, things were harder for *PMA* this time. In terms of fitness evaluations it was outperformed by *DE* in the optimization of 5 out of 7 of the test functions. It was also outperformed by *PSO* in two cases, and by the *GA* in three of the test functions. This was particularly remarkable for the 4th test function, where the performance of *PMA* degraded considerably as compared to that observed in the first set of experiments.

In terms of convergence rate, *PMA* and the *GA* still showed a 100% convergence record. In contrast, *DE* had its 100% convergence in the 5th function reduced to 96%, whereas the performance of *PSO* degraded to a 60% convergence rate in the 6th test function.

These results show that, although slower in some cases, *PMA* seems to be more robust, since convergence was still achieved in 100% of the cases, irrespective of the problem, while for *DE* and *PSO*, there were problems where the performance was reduced or they simply failed to converge to any solution at all.

It is important to point it out that the generation of the initial population gives an advantage to *DE*, *PSO*, and the *GA*. In these methods, the population was initialized within the specified range of the search for the global minima. With *PMA* this is not possible, as the output of the *GP* function is unconstrained, unless one uses a function to squash the output of the *GP* tree to the range [-1,+1].

Table 3: Results for *DE*, *PSO*, *GA* and *PMA* in the second set of experiments. Standard deviations are given in brackets. The parameter settings for each algorithm and function are given in Table 2.

Fitness function	DE		PSO		GA		PMA	
	NFE	HITS	NFE	HITS	NFE	HITS	NFE	HITS
f_1	498 (53)	98	4,504 (867)	100	1,382 (308)	100	2817 (1,572)	100
f_2	1,221 (1,683)	95	4,338 (1,721)	100	8,890 (12 747)	100	5250 (4,698)	100
f_3	134 (39)	100	51 (30)	100	337 (117)	100	42 (39)	100
f_4	3,821 (792)	100	761 (181)	100	4,473 (1500)	100	100 283 (50,554)	100
f_5	697 (136)	96	3,278 (1,103)	100	4,959 (4497)	100	885 (831)	100
f_6	1,233 (97)	91	1,832,757 (127,133)	60	9,755 (22 828)	100	26,452 (16,793)	100
f_7	31,485 (1,406)	100	2,000,000 (0)	no hit	101,908 (16 992)	100	16,779 (8,262)	100

4.5.3 Third set of experiments

To investigate the influence of certain control parameters on the performance of *PMA*, a series of experiments were carried out.

In these experiments the effect of each parameter was investigated separately. In turn, each parameter was set to five different values, whereas the other ones were set to the values shown in Table 2. Primarily, *PMA* has three free parameters to be defined by the user: fraction of constants (FC), tournament size (TS), and the number of *GP* variables (NV):

- *Fraction of constants* This defines the fraction (FC) of leaf nodes that are initialized as *GP* constants instead of *GP* variables. It is not possible to assign *GP* constants to all leaves, otherwise the *GP* tree will always deliver the same output, regardless of the parameter being optimized. On the other hand, it is in principle possible to assign *GP* variables to all leaf nodes. However, it is more reasonable to use a mix of *GP* constants and *GP* variables, and let the evolutionary process combine them properly.

  A few experiments were carried out to investigate the mix of *GP* constants and *GP* variables most appropriate to optimize the functions in the test suite. These experiments revealed that $FC = 0.9$ was the best setting for most of the functions (see Fig. 6).

- *Tournament size* The size of the tournament also affects the performance of *PMA* (see Fig. 7). For most of the benchmark functions, a tournament size equal to the size of the population was superior. However, for the 6*th* test function a tournament size equal to half the size of the population was more effective. This suggests that a strong selection pressure represented by a large tournament size boosts the *PMA* performance. However, it may not be a good idea to blindly set the tournament size equal to the size of the population all the time.

- *Number of variables* The number of *GP* variables used to encode each parameter to be optimized also affects the performance of *PMA* (see Fig. 8). Although the best value is not clear cut, the results show that a multivariate encoding is consistently superior to a single variable encoding. Although for the 4*th* function $NV = 200$ has produced the best result, the increase in the number of *GP* variables to encode the parameters, also increases the search space, and this may be counterproductive. So there is a limit to the benefits that a multivariate representation may provide.

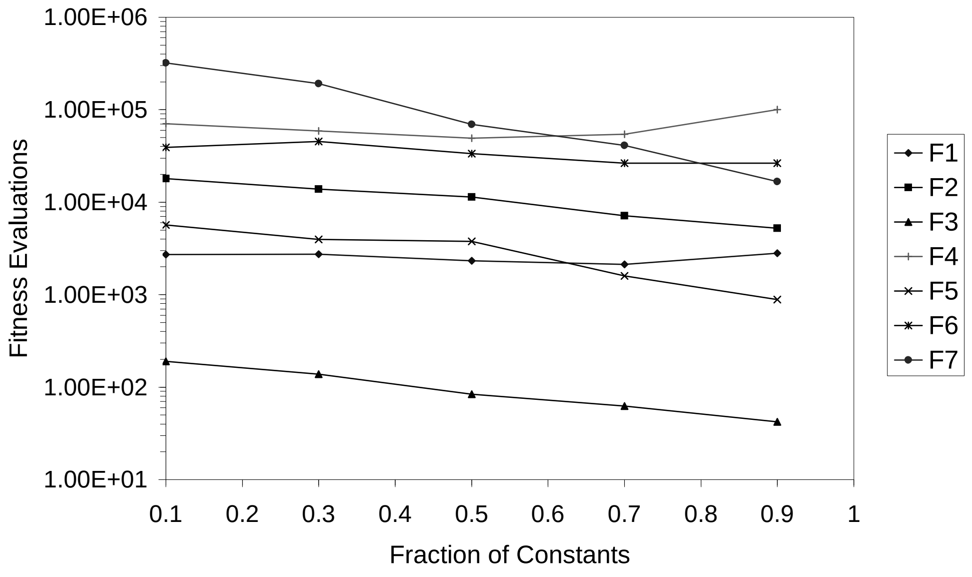


Figure 6: The effect of the value of the fraction of constants (FC) on the number of fitness evaluations.

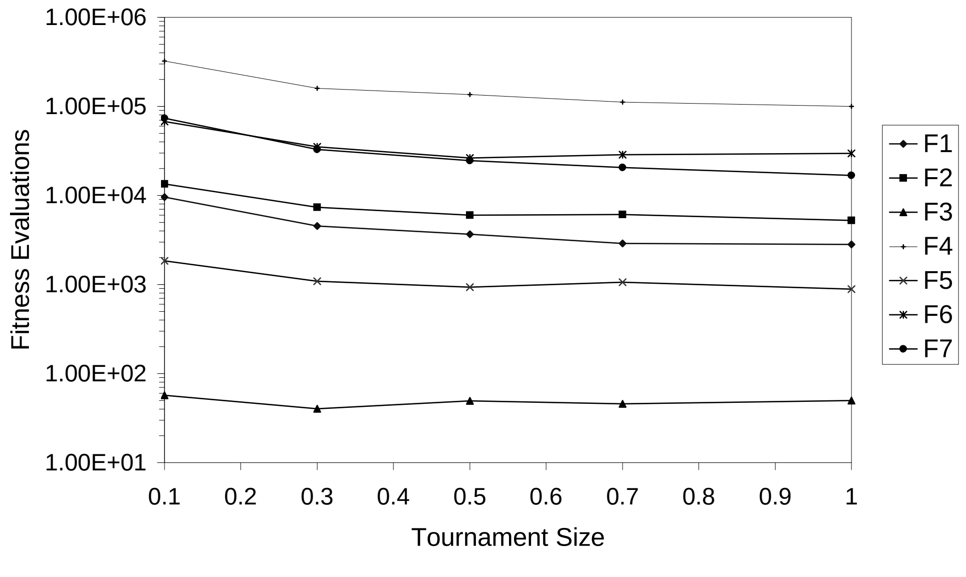


Figure 7: The effect of the value of the tournament size (TS) on the number of fitness evaluations. TS was represented here as a fraction of the size of the population.

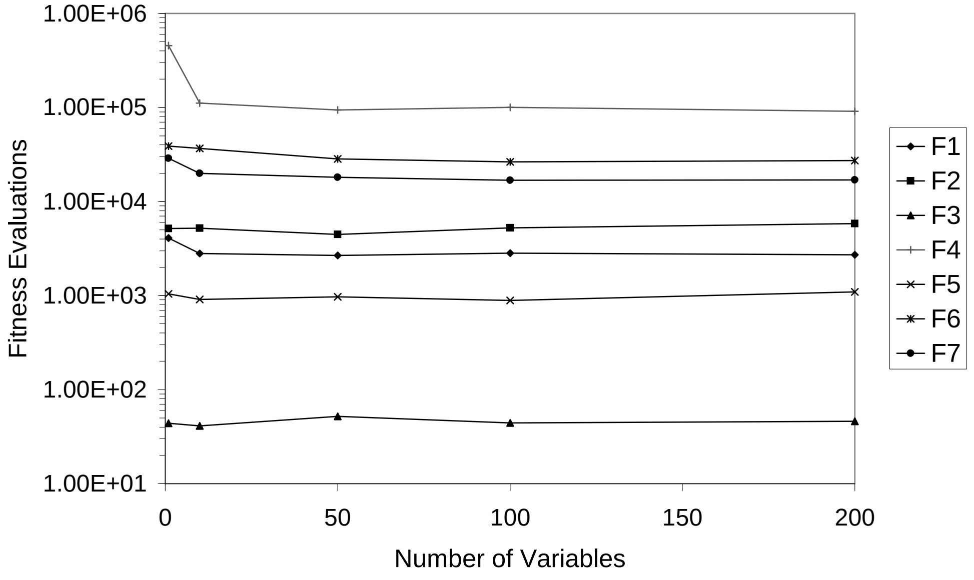


Figure 8: The effect of the number of *GP* variables to encode each parameter on the number of fitness evaluations.

The workings of genetic programming also require the specification of additional parameters: the size of the population, the maximum size of the *GP* trees in the initial population, and also a maximum size to control their excessive growth in the course of the evolution. In all experiments, a value of 10 was set as the maximum depth for the *GP* trees in the initial population and, in the course of the evolutionary process, no tree was allowed to grow larger than 10 000 nodes. The size of the population was chosen to give a good performance for each benchmark function.

4.5.4 Fourth set of experiments

This set of experiments was carried out to compare *PMA*, *DE*, *PSO* and the *GA* with regard to scalability. To do that, experiments with Schwefel's functions of several sizes were performed. The parameter settings of the algorithms are shown in the last row of Table 2. They were selected to achieve optimal performance on the standard Schwefel's function of size $n = 10$.

The results of the experiments are summarized in Table 4. *PMA* clearly outperformed the *DE*, the *GA* and PSO. As a matter of fact, *PSO* failed to converge to any solution at all. Moreover, the performance of these algorithms degraded considerably with the size of the problem. In contrast, *PMA* was hardly affected by the increase of the dimension of the problem.

We also investigated the effect of a rotation of coordinates on the performance of the algorithms. To do that, we used a general rotation matrix $ROTM$ with elements randomly selected within the range [-1, +1]. The function to be optimized is now $f_8(ROTM \times \mathbf{x}^T)$, and fitness was measured as $\left\| f_8(ROTM \times \mathbf{x}^T) - f_8(ROTM \times \mathbf{x}^T_{min}) \right\|$. This procedure couples the dimensions of the function, and should make it more difficult to optimize the function. The results of the experiments are summarized in Table 5.

Indeed, the performance of *DE* and the *GA* on the rotated function degraded considerably as compared to that on the unrotated one. Surprisingly, the performance of *PSO* improved considerably, and convergence to a solution was achieved in many cases. The performance of *PMA* also degraded with the transformation of coordinates, still it was superior to the other algorithms.

These results support the idea that *PMA* may be more robust to the increase of the size of the problem than *DE*, *PSO* and the *GA*.

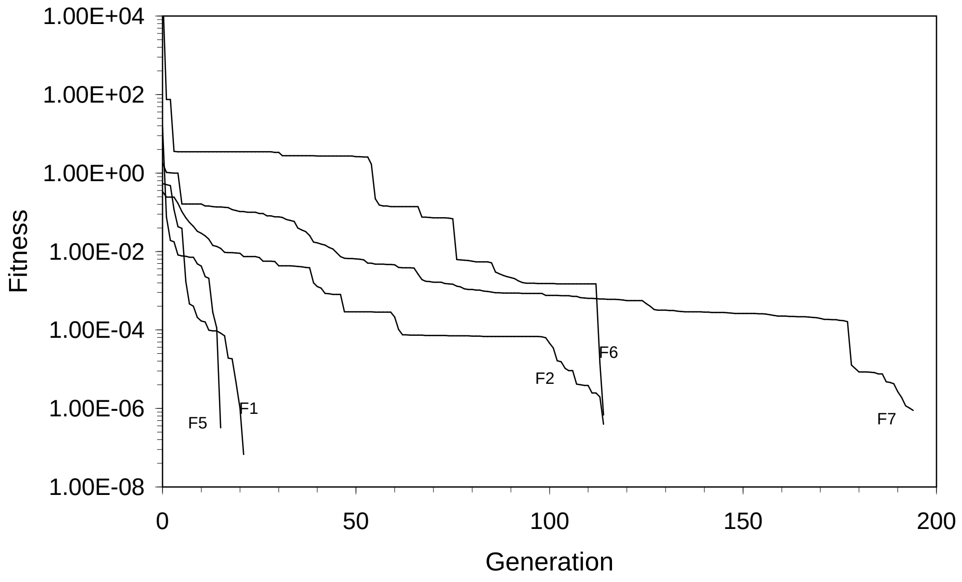


Figure 9: Typical evolution of the fitness of the best individual in the population for functions f_1, f_2, f_5, f_6, and f_7. (Function f_4 is shown in Fig. 10.)

5 Discussion

In the form implemented in this work, *PMA* is basically a random-mutation-hill-climber [38]: an offspring is inserted in the population only if it is superior to the parent selected from the population (the other is randomly created and discarded after crossover). The hill-climbing behaviour of *PMA* is caused by this, coupled with a large tournament to select the parents. As a consequence, most of the time the parent selected from the population is the best individual. In other words, the first population provides an initial survey only, henceforth the search is carried out by using the best individual of the population most of the time. This implies that the fitness of the best individual of the current population may not change for some generations. This is apparent in Figs. 9 and 10, where typical runs for the optimization of functions f_1, f_2, f_4, f_5, f_6, and f_7 were plotted (a similar result for the 3rd function is not shown, as this function required 1 or 2 generations only to converge to a solution most of the time).[3] This effect is particularly clear for functions f_2, f_4, f_6, and f_7, where the fitness of the current best individual of the population is constant over dozens of generations.

However, it must be pointed out that the essence of *PMA* is the representation, not the strategy for selection of the parents or the policy to insert the offspring in the population. There are endless alternatives to those implemented here. Investigating this is the object of future research. What is crucial is that *PMA* explores a new search space, which is different from the parameter space. One may wonder, then, as to why optimization should be more successful on this space. That is, what makes it that easy for GP to have a 100% success rate?

At this stage of the development of *PMA* we are not entirely certain. All our attempts to come up with a strong theoretical explanation have so far failed. This has to be expected since the method and the space it explores are so radically different from other population based search algorithm, that we can hardly use any prior wisdom as a starting point. After all the reasons

[3]A typical run with function f_4 is plotted separately in Fig. 10 due to the different scale needed.

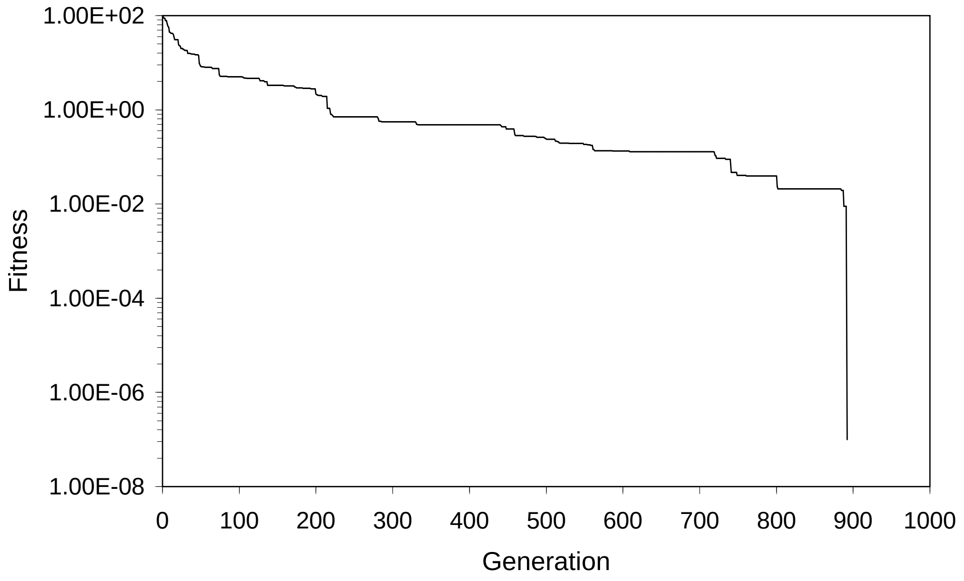


Figure 10: A typical evolution of the fitness of the best individual in the population for function f_4.

for the success and robustness of PSO and DE are still largely not understood, despite these paradigms having being studied for over a decade.

Naturally, we can formulate some conjectures. A different distribution of local optima might be a reason why PMA performed well as compared to the other methods in terms of number of hits. Optimization methods can be trapped in local optima, e.g., if the population quickly converges to a region of the search space that does not include an acceptable solution and the operators are unable to escape from that region. The high success rate obtained by PMA might then be due to a smaller number of local optima. However, other explanations are possible. For example, it is possible that PMA is associated with a different degree of neutrality and/or that the neutral networks it induces have topologies with some beneficial feature. Or it may be the case that PMA has better scaling properties. This property is apparent in the experiments with the Schwefels function, where the performance of PMA was hardly affected by the increase in the size of the task, whereas the performance of the other methods dropped significantly. This might be due to the fact that the size of individuals does not grow proportionally with the size of the task.

All of these are plausible possibilities. More research will be needed to clarify the issue. It is clear, however, that whatever is the reason why we observed an elevated performance of PMA over our test problems, that same reason will depress PMA's performance on some other problems as per No-Free Lunch theorem (see Section 1).

6 Conclusion and further research

In this paper, parameter mapping, a new approach for parameter optimization has been presented. The results of the experiments have shown that the new paradigm can be at least as robust and as efficient as the best function optimization methods.

There are many ideas to be explored in the scope of the new paradigm, which we believe may lead to even further improvements. For example:

- The good results obtained by using a form of macro-mutation represented by the headless chicken crossover may be an indication that evolutionary programming techniques [39, 40,

Table 4: Results for *DE*, *PSO*, *GA* and *PMA* in the fourth set of experiments with the **standard Schwefel's function** f_8. Standard deviations are given in brackets. The parameter settings for each algorithm are given at the bottom of Table 2.

size	DE		PSO		GA		PMA	
n	NFE	HITS	NFE	HITS	NFE	HITS	NFE	HITS
10	481,348	98	2,000,000	no hit	16,766	93	1,657	100
	(390,834)		(0)		(10,323)		(3,170)	
30	695,662	75	2,000,000	no hit	33,910	99	1,827	100
	(481,932)		(0)		(16,364)		(2,958)	
50	893,179	58	2,000,000	no hit	47,955	98	1,764	100
	(560,694)		(0)		(23,189)		(2,327)	
70	875,406	50	2,000,000	no hit	63,618	99	2,378	100
	(597,123)		(0)		(23,985)		(4,964)	
100	890,654	35	2,000,000	no hit	163,244	50	1,404	100
	(564,328)		(0)		(221,783)		(1,113)	

Table 5: Results for *DE*, *PSO*, *GA* and *PMA* in the fourth set of experiments with **Schwefel's function** f_8 **with rotation of coordinates**. Standard deviations are given in brackets. The parameter settings for each algorithm are given at the bottom of Table 2.

size	DE		PSO		GA		PMA	
n	NFE	HITS	NFE	HITS	NFE	HITS	NFE	HITS
10	329,471	98	108,276	99	36,850	100	1,131	100
	(370,529)		(109,828)		(28,528)		(1,202)	
30	778,089	72	377,654	99	119,917	100	3,146	100
	(579,363)		(325,583)		(87,715)		(13,381)	
50	789,146	55	725,645	95	426,756	97	5,320	100
	(581,155)		(507,130)		(355,266)		(23,476)	
70	892,219	39	608,784	81	616,900	79	3,557	99
	(585,324)		(469,010)		(486,473)		(8,927)	
100	861,254	45	766,426	71	787,442	46	2,621	100
	(582,506)		(535,354)		(548,469)		(4,140)	

41], that rely on mutation exclusively, should be investigated. Other more advanced *GP* techniques such as automatic defined functions should be also explored [20].

- The fact that the output of the *GP* tree is the value of the parameter being optimized does not prevent the use of the raw values of other parameters as input, to convey context information. In conventional learning methods a similar strategy is beneficially used. For example, this happens in the training of neural networks, where the updating of a particular weight depends on the value of many other weights.
- Although for the problems addressed in this paper the *GP* function set used has produced good results, for other classes of problems it might be interesting to include a variety of other functions.
- A different policy for the insertion of the offspring in the population might be used. For example, the offspring may have to face a competition with individuals other than its parents.
- The multivariate (sparse) encoding makes the use of binary encodings possible. In this case, each bit would be represented by a single variable. The canonical representation of genetic algorithms has steadily lost ground to real-valued representations in continuous parameter optimization tasks. *PMA* may result in a revival of this form of encoding.

This will be the target of future research. Beyond these and many other ideas we have for extending the parameter mapping approach, *PMA* needs to be better understood and positioned in terms of other *POT* algorithms. This can be achieved through comparative experimentation involving a range of optimization algorithms applied to a variety of interesting test problems, exhibiting various levels of complexity. Furthermore, in the future PMA will need to be tested on real-world function optimisation problems and combinatorial optimisation problems to evaluate its potential “outside the lab”.

Acknowledgment

The authors would like to thank the Leverhulme Trust for the financial support to carry out this research. The authors would also like to thank Professor Ian Parmee for useful comments and suggestions on an earlier grant proposal on *PMA*.

References

[1] D. Wolpert and W. Macready, “No free funch theorems for optimization,” *IEEE Transactions on Evolutionary Computation*,vol. 1, no. 1:67-82, Apr. 1997.

[2] D. Fogel and A. Ghozeil, “A note on representations and variation operators,” *IEEE Trans. on Evolutionary Computation*, vol. 1, no. 2, pages 159-161, 1997.

[3] J. Hall, “A novel, real-valued genetic algorithm for optimizing radar absorbing materials,” *CR-2004-212669*, NASA, Lockheed Martin Space Operations, Hampton, Virginia, 2004.

[4] P. Ballester and J. Carter, “Real-parameter genetic algorithms for finding multiple optimal solutions in multi-modal optimization,” in E. Cantú-Paz *et al.*, editors, *Proc. of the Genetic and Evolutionary Computation Conference*, pages 706-717, Seattle, USA, Jun. 2003.

[5] N. Hansen and A. Ostermeir, “Completely derandomized self-adaptation in evolution strategies,” *Evolutionary Computation*, vol. 9, no. 2, pages 159-195, Summer 2001.

[6] T. Ueda, N. Koga and M. Okamoto, “Efficient numerical optimization technique based on real-coded genetic algorithm,” *Genome Informatics*, vol. 12, pages 451-453, 2001.

[7] D. Corne *et. al*, editors, *New Ideas in Optimization*, McGraw-Hill Publishing Company, 1999.

[8] A. Radi and Riccardo Poli, “Evolutionary discovery of learning rules for feedforward neural networks with step activation function,” in W. Banzhaf *et al.*, editors, *Proc. of the Genetic and Evolutionary Computation Conference*, vol. 2, pages 1178-1183, Orlando, USA, 13-17 Jul. 1999.

[9] M. Dorigo, V. Maniezzo and A. Colorni, “Ant system: optimization by a colony of cooperating agents,” *IEEE Trans. on Systems, Man and Cybernetics, Part B*, vol. 26, no. 1, pages 29-41, 1996.

[10] T. Bäck, “An overview of evolutionary algorithms for parameter optimization,” *Evolutionary Computation*, vol. 1, no. 1, pages 1-23, Spring 1993.

[11] A. Engelbrecht, *Computational Intelligence, an Introduction*, John Wiley & Sons, 2002.

[12] L. Ingber and B. Rosen, “Genetic algorithms and very fast simulated reannealing: a comparison”, *Mathematical and Computer Modelling*, vol. 16, no. 11, pages 87-100, 1992.

[13] H. Cho, S. Oh and D. Choi, “A new evolutionary programming approach based on simulated annealing with local cooling schedule”, in W. Banzhaf, R. Poli, M. Schoenauer and T. Fogarty, editors, *Proceedings of the IEEE International Conference on Evolutionary Computation (ICEC)*, vol. 686, pages 598-602, Springer-Verlag (LNCS), 1998.

[14] M. Riedmiller and H. Braun, “A direct adaptive method for faster backpropagation learning: the RPROP algorithm”, in *Proc. of the IEEE International Conference on Neural Networks (ICNN)*, San Francisco, CA, USA, 1993.

[15] L. Altenberg, “Genome growth and the evolution of the genotype-phenotype map,” in W. Banzhaf and F. H. Eeckman, editors, *Evolution as a Computational Process*, pages 205-259. Springer-Verlag, Berlin, Germany, 1995.

[16] W. Banzhaf, “Genotype-phenotype-mapping and neutral variation - A case study in genetic programming,” in Y. Davidor *et al.*, editors, *Proc. of Parallel Problem Solving from Nature III*, vol. 866 of *LNCS*, pages 322–332, Jerusalem, 9-14 Oct. 1994. Springer-Verlag.

[17] M. Shackleton, R. Shipman and M. Ebner, “An investigation of redundant genotype-phenotype mappings and their role in evolutionary search,” in *Proc. of the 2000 Congress on Evolutionary Computation CEC00*, pages 493–500, La Jolla, California, USA, 6-9 Jul. 2000. IEEE Press.

[18] T. Bäck, M. Schültz and S. Khuri, “Evolution strategies: an alternative evolutionary algorithm”, in J. Alliot *et. al*, editors, *Artificial Evolution*, pages 3-20, Springer-Verlag, 1996.

[19] T. Bäck and H. Schwefel, “Evolution strategies I: variants and their computational implementation”, in G. Winter *et. al*, editors, *Genetic Algorithms in Engineering and Computer Science*, pages 111-126, John Wiley, 1995.

[20] J. Koza, *Genetic Programming: on the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA, USA, 1992.

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

[22] J. Pujol, “Evolution of artificial neural networks using a two-dimensional representation,” PhD thesis, School of Computer Science, University of Birmingham, UK, Apr. 1999.

[23] J. Pujol and R. Poli. “Evolution of neural networks using weight mapping”, in W. Banzhaf *et al.*, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, vol. 2, pages 1170-1177, Orlando, Florida, USA, 13-17 Jul. 1999. Morgan Kaufmann. (Available from `http://cswww.essex.ac.uk/staff/poli/papers/Pujol-GECCO1999.pdf`)

[24] N. Radcliffe, "Genetic set recombination and its application to neural networks topology optimization", *Technical Report EPCC-TR-91-21*, University of Edinburgh, Scotland,1991.

[25] P. Hancock, "Genetic algorithms and permutation problems: a comparison of recombination operators for neural structure specification", *Proceedings of COGANN Workshop, IJCNN*, 1992. IEEE Computer Society Press.

[26] J. Pujol and R. Poli. "A highly efficient function optimization with genetic programming", in W. Banzhaf *et al.*, editors, *Late Breaking Papers of the Genetic and Evolutionary Computation Conference*, Seattle, USA, Jun. 2004.

[27] J. Pujol and R. Poli, "Optimization via parameter mapping with genetic programming", in X. Yao *et al.*, editors, *Proceedings of the 8th International Conference on Parallel Problem Solving from Nature*, pages 382-390, Birmingham, UK, 18-22 Sept. 2004.

[28] P. Angeline, "Subtree crossover: building block engine or macromutation?", in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9-17, Stanford University, CA, USA, 13-16 Jul. 1997, Morgan Kaufmann.

[29] R. Storn and K. Price, "Differential avolution - a simple and efficient adaptive scheme for global optimization over continuous spaces", *TR-95-012*, International Computer Science Institute, Berkeley, USA,1995.

[30] H. Mühlenbein and D. Schlierkamp-Voosen, "Predictive models for the breeder genetic algorithm: I. continuous parameter optimization", *Evolutionary Computation*, vol. 1, no. 1, pages 1-23, Spring 1993.

[31] L. J. Eshelman and J. D. Schaffer, "Real-Coded Genetic Algorithms and Interval-Schemata", in D. Whitley, editor, *Proceedings of the Foundations of Genetic Algorithms Workshop (FOGA-92)*, Vail, Colorado, 1992.

[32] K. Price, "An introduction to differential evolution", in D. Corne *et. al*, editors, *New Ideas in Optimization*, pages 379-387, McGraw-Hill Publishing Company,1999.

[33] J. Kennedy and R. Eberhart, "The particle swarm: social adaptation in information-processing systems", in D. Corne *et. al*, editors, *New Ideas in Optimization*, pages 379-387, McGraw-Hill Publishing Company,1999.

[34] D. Goldberg, *Genetic Algorithm in Search, Optimization and Machine Learning*, Addison-Wesley, Massachusets, USA, 1989.

[35] M. Mitchell, *An Introduction to Genetic Algorithms*, MIT Press, Cambridge, Massachusets, USA, 1996.

[36] Z. Michalewicz, *Genetic Algorithms [Plus] Data Structures = Evolution Programs.* Springer-Verlag, Berlin, Germany, 1994.

[37] T. Bäck, D. Fogel and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, Oxford Press, 1997.

[38] M. Mitchell and S. Forrest, "Royal road functions", in T. Bäck, D. Fogel and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*, pages B2.7:20 - B2.7:25, Oxford University Press, 1997.

[39] V. Porto, "Evolutionary programming", in T. Bäck, D. Fogel and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*, pages B1.4:1 - B1.1:10, Oxford University Press, 1997.

[40] K. Chellapilla, "Evolutionary programming with tree mutations: evolving computer programs without crossover", in *Proceedings of the Second Conference on Genetic Programming*, pages 431-438, 1997.

[41] P. Angeline, G. Saunders and J. Pollack, "An evolutionary algorithm that constructs recurrent neural networks", *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pages 54-65, 1994.