Symbolic regression of discontinuous and multivariate functions by Hyper-Volume Error Separation (HVES)

Cyril Fillon Alberto Bartoli
University of Trieste, Via Valerio, 10, 34127 Trieste, Italy
{cfillon, bartolia}@univ.trieste.it

***Abstract*— Symbolic regression is aimed at discovering mathematical expressions, in symbolic form, that fit a given sample of data points. While Genetic Programming (GP) constitutes a powerful tool for solving this class of problems, its effectiveness is still severely limited when the data sample requires different expressions in different regions of the input space — i.e., when the approximating function should be discontinuous.**

In this paper we present a new GP-based approach for symbolic regression of discontinuous functions in multivariate data-sets. We identify the portions of the input space that require different approximating functions by means of a new algorithm that we call Hyper-Volume Error Separation (HVES). To this end we run a preliminary GP evolution and partition the input space based on the error exhibited by the best individual across the data-set. Then we partition the data-set based on the partition of the input space and use each such partition for driving an independent, preliminary GP evolution. The populations resulting from such preliminary evolutions are finally merged and evolved again.

We compared our approach to the standard GP search and to a GP search for discontinuous functions in univariate data-sets. Our results show that coupling HVES with GP is an effective approach and provides significant accuracy improvements while requiring less computational resources.

I. Introduction

In the last decades, engineers and decision makers expressed a growing interest in the development of effective modeling and simulation methods to understand or predict the behavior of many phenomena in science and engineering. Many such methods are based on regression analysis of a data sample in order to construct mathematical models for convenience and ease of interpretation. It is usually assumed that the data points in the sample are related to a unique function. There are many applications, however, in which this assumption may constitute an oversimplification, such as signal processing, time series prediction, pattern recognition and so on. In such cases the data-set could span across portions of the input space that are to be modeled differently, which means that the symbolic regression should produce a discontinuous function.

Attacking this problem involves facing several challenging issues:

1) Localizing discontinuity boundaries in the data sample without preliminary knowledge about their number and location.
2) Partitioning the data-set according to such boundaries, in order to improve the fit on each partition.
3) Assembling the formulas found in each partition into a consistent hierarchy in order to provide a single discontinuous function.

Obviously, performing the above steps in a multidimensional space adds substantial further complexity.

In this work we propose a novel approach based on Genetic Programming (GP). GP is an automatic method for creating computer programs by means of artificial evolution [1]. GP may be a powerful means for coping with problems in which finding a solution and its representation is difficult but evaluating the performance of a candidate solution is reasonably simple [2]. GP is particularly suitable for symbolic regression problems, especially when the form of the approximating function is not known beforehand, because the process automatically optimizes both the functional form and the coefficient values of the formula.

With GP, a population of computer programs is generated at random. Each program is associated with a fitness value which depends on its ability to solve the problem. Fitter programs are selected for recombination to produce a new population by using genetic operators, such as crossover and mutation. This step is iterated for some number of generations until the termination criterion of the run has been satisfied — e.g. a program exhibiting the maximum possible fitness value has been found. Programs are usually represented as abstract syntax trees, where a branch node is an element from a *function set* which may contain arithmetic, logic operators, elementary functions with at least one argument. A leaf node of the tree is instead an element from a *terminal set*, which usually contains variables, constants and functions with no arguments.

Symbolic regression problems are usually faced with a function set including basic arithmetic operators (e.g., $+, -, \times, /$) and other elementary functions (e.g., $exp, log, cosine, sine$). As these basic elements are used by the GP process to evolve more elaborate programs — i.e., formulas — most of any resulting formula will be continuous and smooth. To improve accuracy when the underlying model is discontinuous, one can introduce in the function set conditional operators and relational operators (*if*, $\leq, \geq, =$). Another approach is to change the representation of the solution. In [3], the authors propose to introduce a new hybrid structure to deal with discontinuity for univariate functions. This structure called point-tree representation contains the discontinuity fields and the associated sub-functions in the same individual. Authors report significant improvements of success rate and better solutions than the classical GP approach. This new repre-

1-4244-1340-0/07$25.00

sentation implies a modification of the genetic operators and preliminary knowledge on the number of discontinuity points. Most importantly, it can be applied only to univariate data-sets.

In this paper we propose a novel approach, suitable for multivariate data-sets and that does not require any prior knowledge about the number of discontinuities. We are not aware of any other proposal with similar features. We generate an initial population from scratch and let this population evolve for a small number of generations. We select the best individual and evaluate the error for each fitness case. This error is used by an algorithm developed by us, that we call *Hyper-Volume Error Separation (HVES)* and implements an heuristic for identifying the portions of the input space requiring different approximating functions. Next we reflect such partition of the input space on the data-set and run several preliminary evolutions, one for each partition. The populations resulting from such independent evolutions are finally merged and evolved again.

We compared our approach to the three approaches mentioned above on 8 distinct benchmarks, including all those considered in [3]. The results show that our approach is very effective and largely outperforms the existing alternatives, because it provides significant improvements of the accuracy of the solutions with either the same or lower computational resources.

This paper is organized as follows. In Section II we give an overview of the Genetic Programming strategy used to discover discontinuous functions. Then, we describe the underlying mechanisms of the Hyper-Volume Error Separation algorithm. Section IV describes the experimental procedure used to benchmark our approach. Section V discusses the results and the behavior of the new algorithm. Section VI concludes and anticipates on further evolutions related to this new methodology.

II. Coupling GP with HVES: an overview

In this section we describe step by step the working principles of our approach. In section III we will enter more in detail in the HVES algorithm. Clearly, if the search finds an individual that solves the problem for a given data sample, then the search stops immediately. For ease of description, we omit this action from the description below.

A. Model description

We generate an initial population P_I from scratch and let this population evolve for a predefined number of generations. We select the best individual and evaluate the error for each fitness case. A fitness case is an input(s)/output(s) pair, which allows measuring how well an evolved individual estimates the output(s) from the input(s). The resulting errors and the entire dataset D are given as parameters to our HVES algorithm. This algorithm partitions D in two subsets D_H and D_R according to an heuristic described later. Then we generate two further populations from scratch, say P_H and P_R, and let them evolve for a small and predefined number of generations on only part of the dataset: P_H is given only D_H whereas P_R is given only P_R (Figure 1).

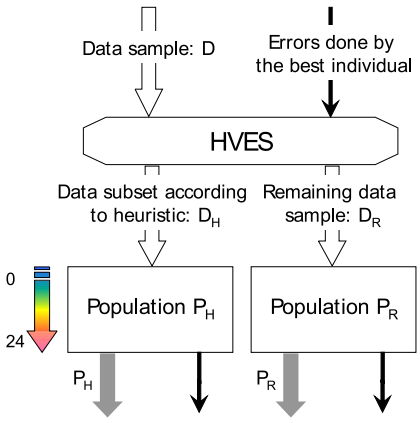


Fig. 1. GP with HVES in the division phase (thick gray arrows represent populations, thick empty arrows represent data-sets)

Finally, we merge the evolved P_I, P_H, P_R and let the resulting final population P_F evolve for a predefined number of generations on the entire dataset D (Figure 2). We discovered in our early experiments that this merging step is very helpful. Each evolution phase consists of the same number of generations and involves a population of the same size.

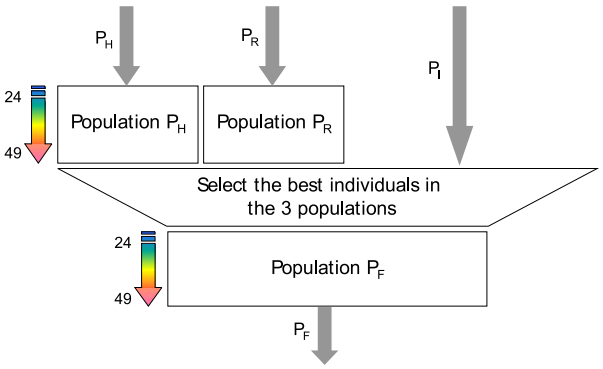


Fig. 2. GP with HVES in the merging phase

Our HVES algorithm works as follows. It partitions the input space in several hyper-volumes whose boundaries are determined by discontinuities in the error function (i.e., the function associating the error of the best individual with each fitness case). Then, it selects the "most difficult" hyper-volume (see below) and partitions the dataset D in two regions: one D_H including all the fitness cases within this hyper-volume, and one D_R including all the remaining fitness cases. Recall that after HVES we focus the evolution of a population on D_H and of *another* population on D_R. The choice of the "most difficult" hyper-volume is made through an index describing a trade-off between number of fitness cases and resulting error

— either few points with a large error, or many points with a small error. The rationale is that such hyper-volume should not contain any discontinuity.

The algorithm is also applied *recursively* in between HVES and the merging phase, as follows (recursion is not shown in the figures, for clarity). The pair (P_H, D_H) produced by HVES plays the role of (P_I, D). The population P_H actually used for the merging phase is the one produced by this recursion. Recursion stops when one of the following conditions is satisfied.

(i) A maximum decomposition depth defined by the user is reached.
(ii) The HVES algorithm does not find any discontinuity boundaries.

The same applies to (P_R, D_R). Recursion turns out to be helpful for finding all discontinuity boundaries and for improving the accuracy on difficult regions of the data sample.

III. HVES

In this section we provide the details of our HVES algorithm, which has already been illustrated above. Recall that HVES attempts to identify discontinuities by analyzing the error rather than the dataset. The reason why this is possible is as follows. The problem consists in finding a function $G(\vec{x})$ that approximates an unknown function $F(\vec{x})$ minimizing the error function $\varepsilon(\vec{x})$.

$$F(\vec{x}) = G(\vec{x}) + \varepsilon(\vec{x}) \tag{1}$$

where $\vec{x} = (x_1, x_2, \ldots, x_n)$ denotes a point in R^n where R is the set of real numbers.

Note that if $F(\vec{x})$ is a discontinuous function and $G(\vec{x})$ is a continuous function then discontinuities in $F(\vec{x})$ are reflected in $\varepsilon(\vec{x})$. We rely on this property to infer the discontinuity boundaries in the data sample from the error vector of the best individual.

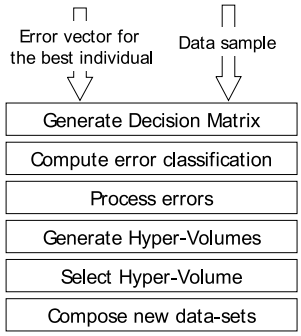


Fig. 3. HVES steps

Rather than describing hyper-volumes explicitly, our HVES algorithm groups fitness cases depending on the hyper-volume they belong to. We represent each such group by means of a tree in which each node corresponds to a fitness case. What makes the algorithm rather complex is that all its steps — e.g., measuring errors, finding discontinuities, grouping points — is done in a multidimensional space. The algorithm can be decomposed in a succession of six steps as illustrated in Figure 3. We will describe these steps one by one in the following subsections.

A. Decision matrix

We define a decision matrix DM as a convenient support for further processing. The decision matrix contains a row for each fitness case, as follows. The first n columns describes the n input variables; column $n+1$ contains the desired output; column $n+2$ contains the errors produced by the best individual. The last column contains the error category according to a classification following the rules defined in the next subsection. m denotes the number of fitness cases.

$$DM = \begin{array}{c} \begin{array}{cccccc} \multicolumn{3}{c}{inputs} & output & error & category \end{array} \\ \begin{pmatrix} x_{11} & \ldots & x_{1n} & f_1 & e_1 & c_1 \\ x_{21} & \ldots & x_{2n} & f_2 & e_2 & c_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ldots \\ x_{m1} & \ldots & x_{mn} & f_m & e_m & c_m \end{pmatrix} \end{array} \tag{2}$$

We also generate a map M for accessing DM conveniently, as follows. For each dimension of the input space R^n, say i, we determine the array L_i containing all distinct values for the i-th input variable, across all fitness cases. Then we sort L_i in ascending order. The map M is composed by the n resulting arrays. We will move on an axis in a discrete way by incrementing or decrementing an index to access to each cell of the array.

B. Error classification

We perform a coarse grain classification of the errors in three categories, as follows. We denote the error on fitness case i as e_i. First, we compute the error standard deviation (or root mean square error) across all fitness cases:

$$\sigma e = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(e_i)^2} \tag{3}$$

where $e_i = f_i - g_i$, g_i is the output computed by the candidate solution, and m is the number of fitness cases. This index captures the dispersion of the errors. Then, we compute the error kurtosis:

$$ke = \sqrt{\frac{1}{m\,(\sigma e)^4}\sum_{i=1}^{m}(e_i)^4} \tag{4}$$

Kurtosis is the degree of peakedness of a distribution, defined as a normalized form of the fourth central moment of a distribution. This index gives insights on the shape of the error distribution. Using these indexes we classify errors according to the rules in Table I.

TABLE I

RULES FOR ERROR CLASSIFICATION

Error category	Rule
No error (denoted $NoError$ in DM)	When error $e_i = 0$
Low error level (denoted $LowError$ in DM)	When $\|e_i\| < \frac{\sigma e}{1+ke}$
High error level (denoted $HighError$ in DM)	When $\|e_i\| \geq \frac{\sigma e}{1+ke}$

C. Error processing

In order to detect discontinuities in the error, we use a simple heuristic based on angles. Each fitness case is considered as a point of a space with $n+1$ dimensions, where the last dimension is the error associated with that point. For deciding whether there is a discontinuity at point v, we determine the two points closest to v, say u and w, and construct two vectors $\vec{uv}, \vec{vw}$. Then we evaluate the angles between these vectors and each axis of the input space R^n. If, for each axis, the absolute value of the difference between these angles is greater than a certain axis-specific threshold, then we assume there is a discontinuity boundary separating $\vec{uv}$ and $\vec{vw}$. An example is given in Figure 4 for a one-dimensional space and for a threshold $\Phi = \frac{\pi}{2}$.

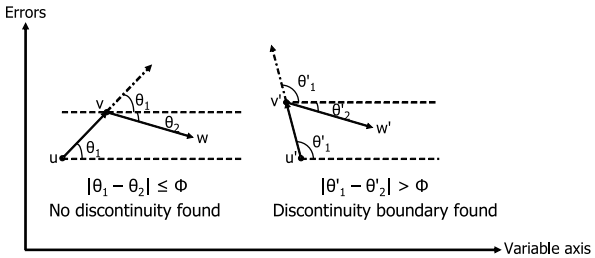


Fig. 4. Discontinuity detection

The evaluation of the axis-specific threshold Φ_i for each axis i is as follows. For each fitness case v, we determine the closest neighbors for each axis, as follows. Let $M(v)_i$ denote the index in map M (section III-A) of v along axis i. For each axis i we collect all fitness cases whose index in M is $M(v)_i + 1$. Then we keep the closest fitness case in terms of Euclidean distance.

Next, we determine the point of R^n, say v_c, whose coordinates are those of the closest points previously found, e.g., the i-th element of v_c is the i-th element of the fitness case closest to v along axis i. We calculate the gradient $\nabla\varepsilon$ of the ε function with respect to vector $\vec{vv_c}$:

$$\nabla\varepsilon = \left(\frac{\partial\varepsilon}{\partial x_1}, \frac{\partial\varepsilon}{\partial x_2}, \ldots, \frac{\partial\varepsilon}{\partial x_n}\right) \quad (5)$$

We convert each component of the gradient to an angle by means of the tan^{-1} function (error gradients may also interpreted like the slopes of the error for each axis), thereby obtaining the following tuple:

$$\Theta = (\theta_1, \theta_2, \ldots, \theta_n) \quad (6)$$

Finally, for each axis i, we calculate the standard deviation of the angles across all the m fitness cases and determine the angle thresholds Φ_i accordingly:

$$\Phi = (\sigma(\theta_1), \sigma(\theta_2), \ldots, \sigma(\theta_n)) \quad (7)$$

D. Hyper-Volume generation

As observed at the beginning of this section, we group fitness cases based on the hyper-volume they belong to, without describing such hyper-volumes explicitly. In order to describe such grouping in a convenient way we developed a tree structure (Figure 5) in which each node corresponds to a fitness case and stores: (i) the corresponding DM row; and (ii) the corresponding indexes in map M. The algorithm for building

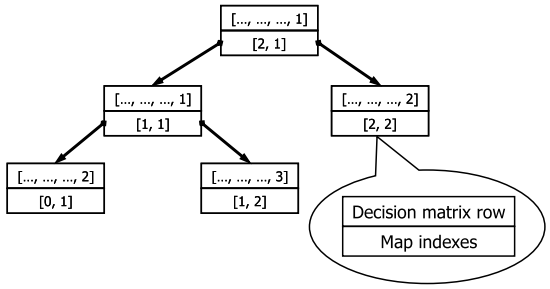


Fig. 5. Hyper-Volume tree representation

such trees is given in Figure 6 and not discussed further due to space constraints. The entry point is *buildHVTreeSet()*. Essentially, a child node N_c is inserted according to a system of rules using information contained in the root node N_r, the parent node N_p and in some cases the grand parent node N_{gp} (at the beginning root node and parent node are the same). A new tree is created when there is no more fitness case which satisfies any of the the rules embedded in procedure *insertNode()* in Figure 6.

E. Hyper-Volume selection

We defined an index capturing the size of an hyper-volume and the amount of errors associated with the fitness cases in it. We called this index Weighted Euclidean Distance (WED):

$$WED = \sum_{k}^{s}\sum_{l}^{c}\left(\delta(node_k, childNode_l)\,(\frac{e_k + e_l}{2})\right) \quad (8)$$

where: $\delta(N_p, N_q)$ denotes the Euclidean distance between the points of the input space associated with nodes N_p, N_q; s is the number of nodes in the Hyper-Volume tree; c is the number of children of node $node_k$.

F. Data-sets composition

We define an hyper-parallelepiped enclosing the hyper-volume found at the previous step, as follows. We determine, for each axis, the minimum and maximum values across all fitness cases stored in the tree. Such values define the boundaries of the hyper-parallelepiped. Finally, we partition the data-set D so that fitness cases within the hyper-parallelepiped belong

Global variables

DM	Decision matrix
M	Generated map
$setNodesAnyTree$	Set of the nodes belonging to one of the trees
$setNodesThisTree$	set of the nodes belonging to the tree under construction

buildHVTreeSet()

1) **Foreach** row in DM,
   a) Generate a root node N_r
      i) **if** $N_r \notin setNodesAnyTree$ **then**
         A) Clear $setNodesThisTree$
         B) **buildHVTree**(N_r)

buildHVTree(N_p)

N_p Parent node

1) **Foreach** $i \in [1, n]$, // all the axis
   a) **While** there is a node to insert in this tree,
      i) N_c := point closest to N_p along current axis and not yet inserted in this tree
      ii) **insertNode**(N_p, N_c)

insertNode(N_p, N_c)

1) N_r := root node of the tree
2) C_r := error category for the root node N_r
3) C_p := error category for the parent node N_p
4) C_c := error category for the node in exam N_c
5) **if** $C_r = NoError \wedge C_p = NoError \wedge C_c = NoError \wedge N_c \notin setNodesAnyTree$ **then**
   Insert N_c as a child of N_p, put in $setNodesAnyTree$
6) **if** $C_r = LowError \wedge (C_p = NoError \vee C_p = LowError)$ **then**
   a) **if** $(C_c = NoError \wedge N_c \notin setNodesThisTree)$ **then**
      Insert N_c as a child of N_p, put in $setNodesThisTree$
   b) **elseif** $(C_c = LowError \wedge N_c \notin setNodesAnyTree)$ **then**
      Insert N_c as a child of N_p, put in $setNodesAnyTree$
7) **if** $C_r = HighError \wedge (C_p = LowError \vee C_p = HighError)$ **then**
   a) **if** $(C_c = NoError \wedge N_c \notin setNodesThisTree)$ **then**
      Insert N_c as a child of N_p, put in $setNodesThisTree$
   b) **else**
      i) N_{gp} := the parent of the parent node N_p
      ii) α := angle vector computed from the nodes N_{gp} and N_p
      iii) β := angle vector computed from the nodes N_p and N_c
      iv) **if** $(C_c = LowError \wedge N_c \notin setNodesThisTree$ **then**
         A) **if** $|\alpha_i - \beta_i| < \Phi_i \ \forall i \in [1, n]$ **then**
            Insert N_c as a child of N_p, put in $setNodesThisTree$
      v) **elseif** $(C_c = HighError \wedge N_c \notin setNodesAnyTree)$ **then**
         A) **if** $|\alpha_i - \beta_i| < \Phi_i \ \forall i \in [1, n]$ **then**
            Insert N_c as a child of N_p, put in $setNodesAnyTree$
8) **if** N_c has been inserted **then**
   a) **buildHVTree**(N_c)

Fig. 6. The Hyper-Volume building algorithm

to set D_H whereas the remaining fitness cases belong to D_R. Using an hyper-parallelepiped simplifies the interpretation of the final formulas, since discontinuity boundaries may be delimited using the conditional operator *if* associated with a predicate based on the intersection of the intervals found.

IV. Experimental setup

We compared the following approaches: (i) *Classical GP*; (ii) *Conditional GP*, i.e., GP with conditional and relational operators in the functions set in order to make capturing of discontinuities easier (as outlined in the introduction); (iii) *Hybrid Structure GP*: the approach proposed in [3]; finally, (iv) *GP-HVES*, our proposal.

We used a set of eight functions, denoted $F_1, \ldots, F_8$ as benchmark. Functions F_1 to F_6 have one input variable, functions F_7, F_8 have two input variables. Functions F_1, F_2, F_3, F_4 are those used in [4]. We defined the other functions so that they indeed include discontinuities. All functions are shown in the Appendix. We remark that we defined the univariate functions F_5, F_6 so that their discontinuities are not very pronounced. Our method is suitable for this kind of problems, unlike the proposal in [4] that has been explicitly designed for non-smooth functions. We executed all the approaches on all benchmarks, except that we did not apply Hybrid Structure GP to F_5, F_6 (because it would have been unfair) nor to F_7, F_8 (because these functions are multivariate). The functions and terminals set used in each case are shown in Table II.

TABLE II
TERMINALS AND FUNCTIONS SET

	GP, GP-HVES	
Functions	**Terminals set**	**Functions set**
F_1, F_2, F_3, F_4	$x, 1$	$+, -, /, \times$
F_5, F_6	$x, 1$	$+, -, /, \times$
F_7, F_8	$x, y, 1$	$+, -, /, \times, cos, sin$
	Extra terminals and functions for GP conditional	
F_1 to F_8	$true$	$if, \leq, \geq, or, and, not$

For $F_1, \ldots, F_4$ we used the same fitness as in [4], i.e., the sum of absolute errors:

$$SAE = \sum_{i=1}^{m} |f_i - g_i| \tag{9}$$

For all other functions we used as fitness the mean of the squared distances between the expected values f_i and the values g_i obtained by the individual:

$$MSE = \frac{1}{m} \sum_{i=1}^{m} (f_i - g_i)^2 \tag{10}$$

For $F_1, \ldots, F_5$ we used a data-set composed of 100 points drawn according to a uniform probability distribution. For F_6 we used 300 points. For F_7 and F_8 we used 441 points corresponding to all combinations of the values in $[-1.5, 1.5]$ by step of 0.15.

All the parameters are summarized in Table III. Whenever GP-HVES runs an evolution, the maximum number of generations is set to 25.

We used our own Genetic Programming API. This API is implemented in Java and is based on a modular architecture where each step of the GP process may be delegated to a plugin. Plugins for all common algorithms for tree generation, fitness evaluation, selection and variation are provided. Moreover this API is based on a strongly typed GP approach similar to [5].

We executed 3 tests for each test case. Each test is the result of 100 independent executions. Each execution starts with a different seed for the random number generator but we used the same seeds for each test. We ran all simulations on a PC based on a processor Intel Xeon 3.20 GHz with 2 GB of RAM.

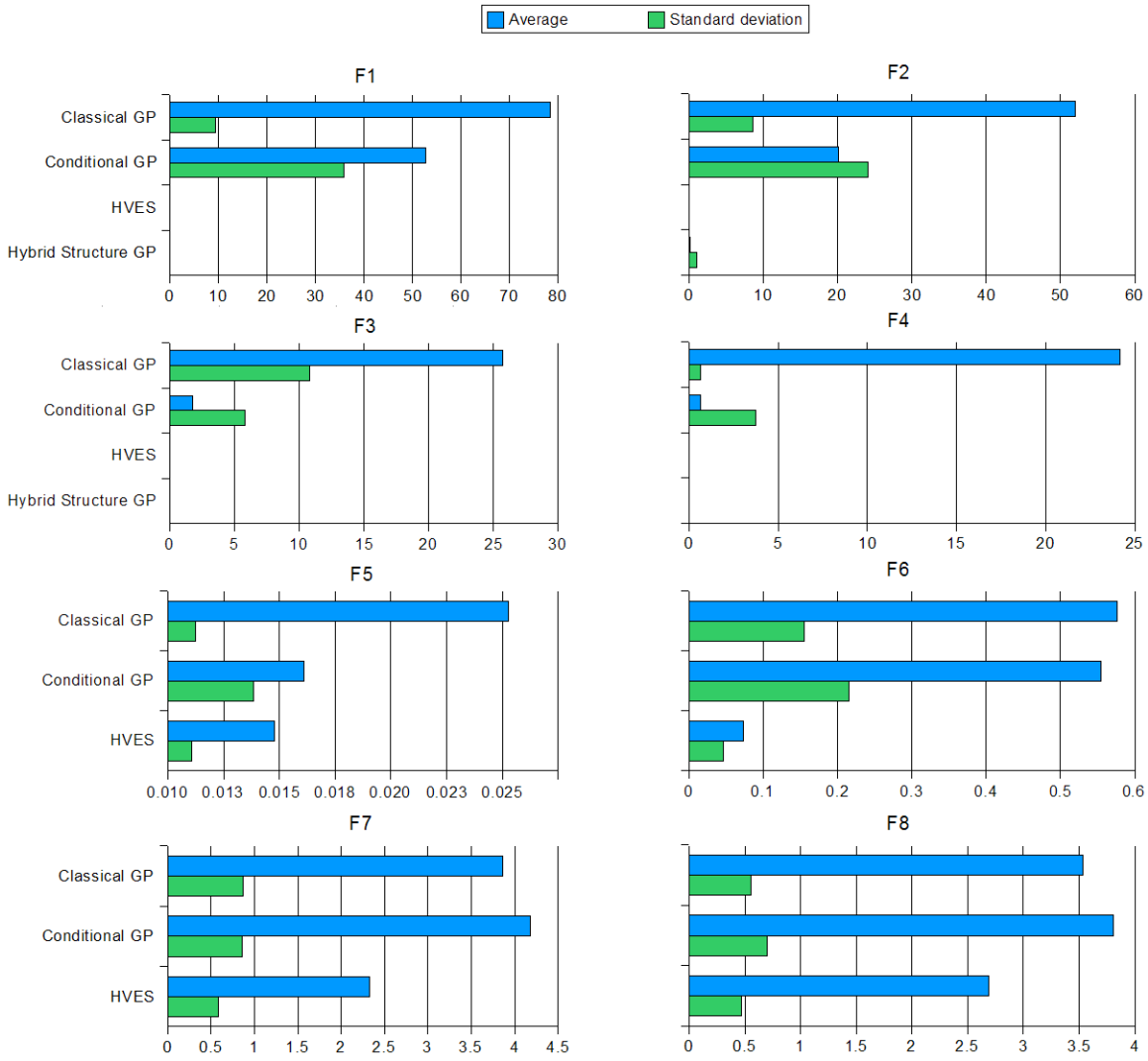


Fig. 7. Average and standard deviation of the fitness values for each tested function F_i

TABLE III
PARAMETER SETTINGS

Parameter	**Setting (F_1 to F_4)**	**(F_5 to F_8)**
Population size	500	1000
Max generation number	600	200
Initialization method	Ramped Half-and-Half	
Initialization depths	2 levels	2-4 levels
Max depth for trees	10	8
Selection	Tournament of size 7	
Elitism	0	1
Node bias for crossover	90% internals, 10% terminals	
Duplication rate	10%	5%
Crossover rate	70%	85%
Mutation rate	20%	15%
Recursion depth (HVES only)	11	3

V. RESULTS

To assess the accuracy of the solutions, we considered the fitness of the best individual found in each run. We computed average and standard deviation of this fitness value across all runs. Figure 7 shows the results for each approach.

Our approach is able to find the exact discontinuous functions with a success rate of 100% for the four first functions. It is worth to note that our proposal does at least as well as the GP based on hybrid structure. For the functions F_1, F_2, F_3, F_4 we never reach the maximum recurrence depth defined in Table III. The algorithm stops after two or three recursions only since it finds the exact discontinuity boundaries and the correct discontinuous functions.

We note that our proposal exhibits the best accuracy for

F_5, F_6 , the improvement with respect to the best result with the GP algorithm using conditional and relational operators is 109% and 765%, respectively. Another important result is that our approach is able to improve the accuracy on the multivariate functions F_7 and F_8. Improvements for these functions respect to the best result with the classical GP algorithm is 166% and 132% respectively. It is interesting to note that in the multivariate test cases, the solutions found by the classical GP algorithm are more accurate than those found with a GP approach using the relational and conditional operators.

Concerning computational cost, we decided not to measure the number of evaluations because this index is not very meaningful in this context: in GP-HVES the data-set size is not the same across all evolutions. Consequently, we decided to measure the computational cost by logging the time spent on all runs. This index has also the advantage of capturing the fact that, in practice, different evaluations do not have the same cost, depending for instance on the number of nodes composing an individual or the complexity of each node.

We report in the Table IV the total time spent in hours for each tested function and each approach. It appears clearly that the HVES outclasses the others strategies whatever the function involved.

TABLE IV
TOTAL TIME SPENT

	Functions							
Total Time	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8
Classical GP	0.7	0.6	4.8	1.8	7.1	14.6	36.5	39.4
Conditional GP	6.1	11.9	3.1	0.7	9.5	17.2	31.8	37.2
HVES	**0.1**	**0.3**	**1.8**	**0.2**	**7.1**	**13.0**	**25.0**	**28.4**

VI. CONCLUSIONS AND FUTURE WORK

In this paper we introduced a new approach based on GP for symbolic regression of multivariate data-sets where the underlying phenomenon is best characterized by a discontinuous function. We are not aware of any other GP-based approach with similar features, in particular, regarding its ability of handling multivariate data-sets.

In our approach we execute a preliminary evolution and use the error exhibited by the best individual in order to infer discontinuity boundaries in the data sample. Then we apply an algorithm designed by us for selecting an Hyper-Volume in the input space whose boundaries approximately follow the discontinuities. This Hyper-Volume partitions the fitness cases in two sets that are used for driving further preliminary evolutions. The best individuals found by these independent evolutions are then merged and evolved again. The process is also applied recursively until either no further discontinuities are found or a predefined recursion depth is reached.

The results confirm the ability of our algorithm to greatly increase the accuracy of the solutions with respect to existing GP-based approaches. In some benchmarks we have also observed a significant improvement in computational cost and we have never observed an increase in computational cost.

In the future, we plan to investigate other heuristics for discontinuity detection in noisy data-sets. We are currently studying the work done in computer vision, signal processing and statistics fields in [6] [7] [8] [9]. The challenge will be to adapt these techniques for multidimensional spaces since these approaches usually work in one or two dimensions only.

We also envisage to apply our approach for time series prediction and classification tasks. However for these applications we should develop an appropriate policy to use cross validation inside our algorithm in order to avoid over fitting problems.

ACKNOWLEDGMENTS

This work was supported by the Marie-Curie RTD network AI4IA, EU contract MEST-CT-2004-514510 (December 14th 2004)

REFERENCES

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

[2] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming˜: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*. Heidelberg and San Francisco CA, resp.: dpunkt – Verlag für digitale Technologie GbmH and Morgan Kaufmann Publishers, Inc., 1998.

[3] X. Shengwu and W. Weiwu, "A new hybrid structure genetic programming in symbolic regression," in *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, Eds. Canberra: IEEE Press, 8-12 Dec. 2003, pp. 1500–1506.

[4] S. Xiong, W. Wang, and F. Li, "A new genetic programming approach in symbolic regression," in *Proceedings 15th IEEE International Conference on Tools with Artificial Intelligence*. IEEE, 3-5 Nov. 2003, pp. 161–165.

[5] D. J. Montana, "Strongly typed genetic programming," *Evolutionary Computation*, vol. 3, no. 2, pp. 199–230, 1995.

[6] D. Lee, "Coping with discontinuities in computer vision: their detection, classification, and measurement," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 4, pp. 321–344, Apr. 1990.

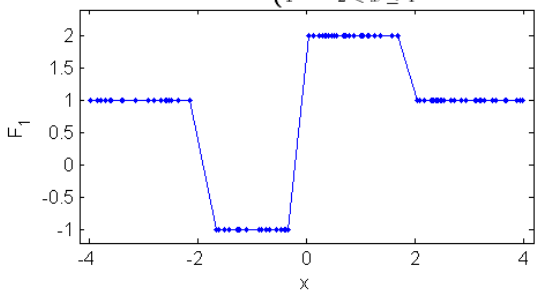
[7] D. L. and W. G.W., "Discontinuity detection and thresholding-a stochastic approach," in *Proceedings of the 1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, Eds. IEEE Press, 3-6 June 1991, pp. 208–214.

[8] A. W. Bowman, A. Pope, and B. Ismail, "Detecting discontinuities in nonparametric regression curves and surfaces," *Statistics and Computing*, vol. 16, no. 4, pp. 377–390, 2006.

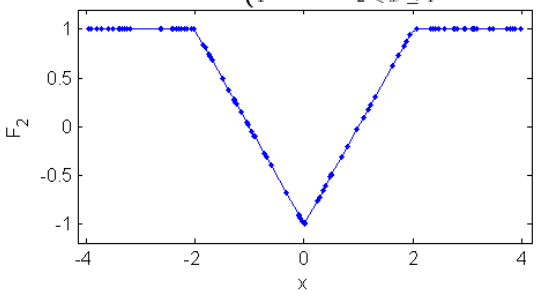
[9] P. Qiu, "Discontinuous regression surfaces fitting," *Annals of Statistics*, vol. 26, no. 6, 1998.

APPENDIX

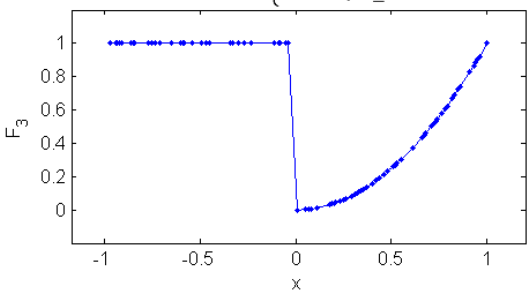
$$F_1(x) = \begin{cases} 1 & -4 \le x \le -2 \\ -1 & -2 < x \le 0 \\ 2 & 0 < x \le 2 \\ 1 & 2 < x \le 4 \end{cases}$$



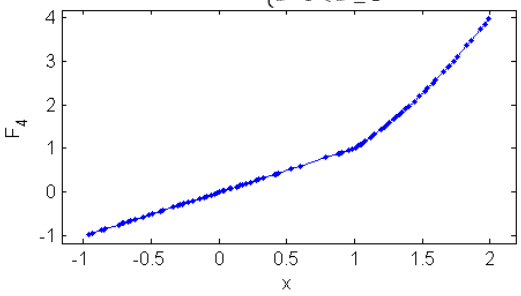
$$F_2(x) = \begin{cases} 1 & -4 \le x \le -2 \\ -(x+1) & -2 < x \le 0 \\ x-1 & 0 < x \le 2 \\ 1 & 2 < x \le 4 \end{cases}$$



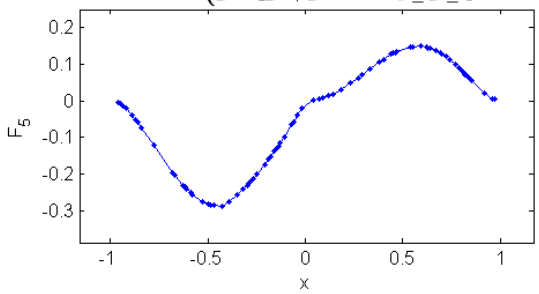
$$F_3(x) = \begin{cases} 1 & -1 \le x \le 0 \\ x^2 & 0 < x \le 1 \end{cases}$$



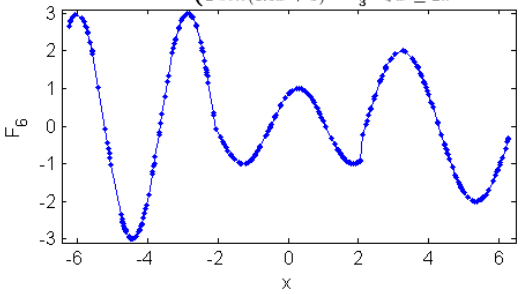
$$F_4(x) = \begin{cases} x & 0 \le x \le 1 \\ x^2 & 1 < x \le 2 \end{cases}$$



$$F_5(x) = \begin{cases} x^5 - 2x^3 + x & -1 \le x < -0.4 \\ x^4 + x^3 + x^2 + x & -0.4 \le x < 0 \\ x^6 - 2x^4 + x^2 & 0 \le x \le 1 \end{cases}$$



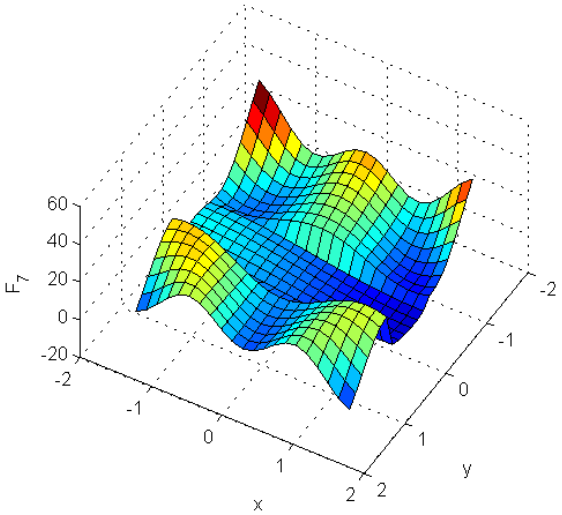
$$F_6(x) = \begin{cases} 3\,sin(2x+1) & -2\pi \le x < \frac{-2\pi}{3} \\ sin(2x+1) & \frac{-2\pi}{3} \le x \le \frac{2\pi}{3} \\ 2\,sin(1.5x+3) & \frac{2\pi}{3} < x \le 2\pi \end{cases}$$



$$f_b(x,y) = 0.1(y-x^2)^2 + (1-x)^2 + 2(2-y)^2 + 17\,sin(1.5x)\,sin(1.7xy)$$

$$f_a(x,y) = x^3 - 3y^2 + 11xy^2 - 7x + y + 3$$

$$F_7(x,y) = \begin{cases} f_a(x,y) & -1.5 \le x \le 1.5, -0.4 \le y \le 0.6 \\ f_b(x,y) & -1.5 \le x \le 1.5, -1.5 \le y \le 1.5 \end{cases}$$



$$F_8(x,y) = \begin{cases} f_b(x,y) & -1.5 \le x \le -0.5, -1.5 \le y \le 0.6 \\ f_b(x,y) & 0.5 \le x \le 1.5, -1.5 \le y \le 1.5 \\ f_a(x,y) & -1.5 \le x \le 1.5, -1.5 \le y \le 1.5 \end{cases}$$

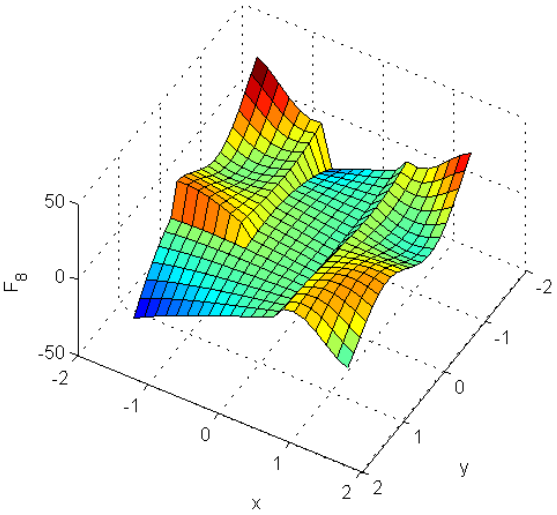


Fig. 8. Test functions