Evolving 2-Dimensional Fuzzy Systems

Víctor M. Rivas[1]

Dpto. de Informática, Universidad de Jaén
E.P.S., Avda. de Madrid 35
E.23071, Jaén (Spain)
vrivas@ujaen.es • http://pagina.de/vrivas

J.J. Merelo[1], **I. Rojas, G. Romero**[1], **P.A. Castillo**[1], **J. Carpio**[1]

Dpto. de Arquitectura y Tecnología de Computadores
Universidad de Granada
[1] *GeNeura Team*
todos@geneura.ugr.es • http://geneura.ugr.es

Abstract

The design of Fuzzy Logic Systems (FLS) generally involves determining the structure of the rules and the parameters of the membership functions. In this paper we present a methodology, based on Evolutionary Computation, for simultaneously designing membership functions and appropriate rule sets. This property makes it different from many techniques that address these goals separately, resulting in suboptimal solutions, because the design elements are mutually dependent. We also apply a new approach in which the evolutionary algorithm is applied directly to a FLS data structure, instead of a binary or other codification. Results on function approximation show improvements over other incremental and analytical methods.

Key words: Fuzzy systems, genetic algorithms, evolutionary algorithms, hybrid methods, function approximation.

1 Introduction

Since the introduction of the basic methods of fuzzy reasoning by Zadeh [23], and the success of their original application to fuzzy control, fuzzy logic and its application to fuzzy control have been widely studied. However, certain

important questions still remain open, including: 1) the selection of the fuzzy rule base; 2) the subjective definitions of the membership functions; and 3) the structure of the fuzzy system (number of rules and membership functions).

The transfer function of a fuzzy system is not based on a mathematical model but is given by the definition of fuzzy rules and fuzzy sets of linguistic variables (for each membership function). The fuzzy rules and fuzzy sets are designed on the basis of the human operator's experience, decisions and control actions. In conventional expert systems, the operator often cannot clearly explain why he/she acts in a certain way. Furthermore, there is no reason to believe that an operator's control is optimal. An automatic design method based on a set of examples for the input/output relationship, termed the referential data set, then becomes important.

In general, creating a Fuzzy Logic System (FLS) involves designing the structure of the rules of the system and the parameters of the membership functions. Most techniques deal with these separately, which may result in a suboptimal solution, because the design elements are mutually dependent. For example, Genetic Algorithms (GAs) can be used firstly to determine the rules of the system and then, in a second stage, to tune the parameters of the linguistic values, as in [10]. We propose optimizing these parts simultaneously, using Evolutionary Computation techniques, with a new method slightly different from those presented in [19], [18] and [20], that will be discussed later.

The rest of the paper is organized as follows: the next section deals with the current state of the art, including an introduction to Evolutionary Algorithms (EA); section 3 describes the EA used in this work, including genetic operators, the algorithm itself and the evolutionary computation library, *Evolutionary Objects* [17]. After this, section 4 describes some experiments and their results; and finally, section 5 presents some conclusions and future lines of work.

2 State of the art

2.1 Evolutionary Algorithms

Evolutionary Algorithms (EAs) represent a set of strategies to efficiently search for near optimal solutions in hard to search spaces, imitating natural genetics. Depending on the problem, the optimal solution may be one that either maximizes or minimizes a given function. Nevertheless, as minimization problems can be easily changed into maximization ones, EA terminology tends to refer only to the latter.

EAs can be characterized by the following features ([13]):

- A genetic representation for potential solutions to the problem. Solutions are called individuals. However, in this work, we apply a new approach in which individuals are not encoded into a chromosome (as they usually are), but in which the EA can deal directly with the solutions as they are, i.e, as FLSs.
- A way to create an initial population of potential solutions. The most common method is by means of a random generator.
- An evaluation function that plays the role of the environment, rating solutions in terms of their fitness. In general, the best individuals are those which have the highest fitness.
- Genetic operators are used to manipulate the genetic composition of the population. New individuals are created by applying these operators to the previously existing ones. The best individuals should generate more offspring than the rest.
- A set of parameters that provide the initial settings for the algorithm: population size, probabilities employed by the genetic operators, termination conditions, and, probably, a set of constraints for individuals.

EAs usually implement three basic operators to manipulate the genetic composition of a population: selection, recombination and mutation.

Selection. Selection is the process by which individuals with higher fitness values have a higher probability than individuals with smaller fitness values of being chosen to reproduce, generating an offspring. Because of this, it is considered a diversity-destruction operator. The most common method used is the weighted roulette selection.

Recombination. Recombination is the process by which one or more new individuals are created using parts from two or more parents. The underlying idea is that the optimal solution is composed of several optimal parts (or *building blocks* [7]), so the massive interchange of information between individuals may lead to better and better new ones. Like selection, recombination operator also decreases diversity.

Mutation. Mutation operators alter, in a random way, the structure or the stored information of the individual to which they are applied. They increase the diversity of population, providing a mechanism to escape from local optima and premature convergence. High rates for these operators allow better exploration of the search space, but make convergence slower and can result in random search.

The properties of EAs make them a powerful technique for selecting high performance parameters for FLSs. Previous work focused basically on optimizing the FLS parameters and on reducing the number of the rules, while in this paper, EAs are used to search for an optimized subset of rules (both the number of rules and the rule values) from a given knowledge base to achieve the goal of minimizing the number of rules used while maintaining the FLS performance. EAs will eliminate all unnecessary rules, i.e., those which have no significant contribution to improving system performance.

Recently, EAs have been combined with fuzzy logic and neural networks in the process of designing fuzzy systems. For a good review see [1]. Ishibuchi et al. [9], for instance, propose a hybrid approach where a set of fuzzy rules is first extracted from a trained neural network, and an EA is then used to select a small subset of rules from the extracted rule set. The fitness function of the EA is designed to minimise the number of selected rules and maximise the number of correctly classified examples. For example, Karr and Gentry [11] control the pH of an acid-base system with the fuzzy system's input membership functions manipulated by an EA.

Some other methods [21] combine EA and FLS in order to tune the parameters of the membership functions and outputs of a Takagi-Sugeno fuzzy rule base, and have been used for the approximation of one-input analytical functions.

On the other hand, codification of rules and membership functions is typically done using vectors (1-D matrices) [3][15][12][8][11], which is not the most natural way to do it, because it keeps "natural" building blocks (for instance, four contiguous cells in the matrix) apart from each other. In this work, FLSs are not coded that way, but implemented by 2-D matrices; this allows rules having close antecedents (thus, interfering with each other), which are generally activated at the same time, to be close in their representation too; that way, they can be more easily transmitted together to the offspring.

It should also be noted that several researchers have concentrated on using real number coding for chromosomal representation of individuals instead of traditional bit string based coding; and it is reported that for some problems, these techniques outperform the conventional bit string based EAs [4][6][22]. This partly supports our work, since we use real numbers for evolution.

A special mention has to be made to some papers in which EAs are used to model and/or tune complete fuzzy systems. By chronological order, the two first are due to Lian, Marzuki and Rubuyah, [18] and [19]. In these works, the authors developed a method to tune a neuro-fuzzy controller using genetic algorithms, applied to a set of problems like coupled-tank liquid-level control,

unstable plants control and automatic car parking. Main difference with the method presented in this paper is that the size of the controller and the number of parameters is fixed, and fitted to the problem being solved at every moment, so that the method needs a previous study of that problem and cannot be applied directly to any kind of problem with two inputs and one output. Second difference is that parameters, that are real values, are represented using bit strings, that are not the natural and logical way it should be done, as was discussed before. A more recent paper is the one by Setnes and Roubos, [20], showing how GA can be used to create fuzzy systems applied to modeling and classification problems. Once more, the problem of how to represent the solutions has been solved in an unnatural way, given that rule antecedents and consequents are stored sequentially in the chromosome. This adds a new problem, given that the genetic operators can produce solutions that violate the different constraints imposed to both the input space and the output space.

3 The evolutionary algorithm

To program this algorithm, we used the **EO** [17] library (*Evolutionary Objects*) because of the facilities it offers to evolve any object (in the sense of object oriented programing) that can be assigned a cost or fitness function. **EO** is a library, currently programed in C++ but easily portable to any other object oriented language, that defines the interfaces of several types of evolutionary algorithms, and includes several examples of their use. It is open-source, and available from http://eodev.sourceforge.net.

The EO library directly evolves classes of objects, so there is no need to code them in a binary chromosome. In this work, evolved objects are fuzzy logic systems (FLS), which are implemented as 2-D arrays; thus, some specific operators are needed to mutate and combine them, in order to create new FLSs.

The following subsections explain the architectures of both the FLS and the EA that optimizes it.

3.1 The 2-D Fuzzy Logic System

In this work, each FLS is implemented as a two-dimensional matrix storing two different things: a) the centres of the triangular partition membership functions of two input variables, X and Y, and b) the values of the output variable, Z. Thus, any 2-D matrix stores both the precedents and the consequents of the FLS rules.

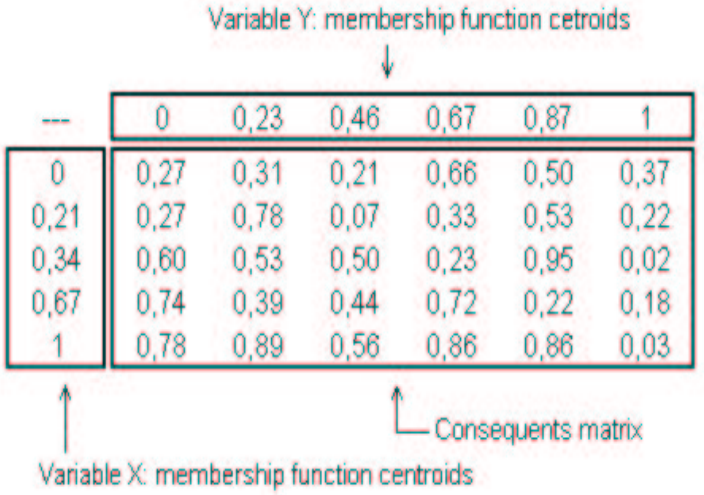

Fig. 1. Example of FLS implementation. In this case it corresponds to a 6x5 FLS.

A little more formally, a m by n FLS is implemented by a two-dimensional matrix:

$$M(m+1, n+1) = \begin{vmatrix} - & Y_1 & Y_2 & \dots & Y_n \\ X_1 & Z_{1,1} & Z_{1,2} & \dots & Z_{1,n} \\ X_2 & Z_{2,1} & Z_{2,2} & \dots & Z_{2,n} \\ \dots & \dots & \dots & \dots & \dots \\ X_m & Z_{m,1} & Z_{m,2} & \dots & Z_{m,n} \end{vmatrix}$$

with $m+1$ rows, and $n+1$ columns, being the minimum number of columns and rows equal to 3 (i.e., $m_{min} = 2$ and $n_{min} = 2$.), while the maximum is one of the parameters the evolutionary algorithm must find. In this matrix, m corresponds to the number of membership functions related to input variable X, and n is the number of membership functions related to input variable Y. It must be taken into account that in any given EA generation, the population will be composed of FLSs having different numbers of rows and columns.

Fig. 1 shows a typical FLS implemented as a 2-D matrix, where the three different parts in which this matrix can be split have been highlighted using boxes.

Each 2-D matrix implementing a FLS is divided into:

- $M[1, 0]$ to $M[m, 0]$ (First column). It stores the centres of the triangular partition membership functions of the first input variable, X, except the first cell, $M[0, 0]$. Variable X takes values in the range $[x_{min}, x_{max}]$, so the following must be true at any time:

$$x_{min} = M[1,0] < M[2,0] < ... < M[m,0] = x_{max}$$
and $m \geq 2$

- $M[0,1]$ to $M[0,n]$ (First row). As the first column but for the second input variable, Y. Newly, the following conditions must be true:
$$y_{min} = M[0,1] < M[0,2] < ... < M[0,n] = y_{max}$$
and $n \geq 2$.
- $M[1,1]$ to $M[m,n]$ (*Consequents matrix*, i.e, the whole matrix except the first row and the first column). Every cell of this submatrix stores a value for Z, the output variable, where:
$$z_{min} \leq M[i,j] \leq z_{max}$$
$$\forall i, 1 \leq i \leq m \text{ and } \forall j, 1 \leq j \leq n.$$
- The cell $M[0,0]$ does not represent anything, thus it is not used.

Using the above implementation, the FLS works using rules with the following form:

$$\text{IF } X \text{ is } M[i,0] \text{ AND } Y \text{ is } M[0,j] \text{ THEN } Z \text{ is } M[i,j]$$

The values x_{max}, x_{min}, y_{max}, y_{min}, z_{max} and z_{min} are parameters that must be provided to the algorithm.

3.2 Genetic operators

Using EO library to develop the EA allows us to directly evolve a FLS, instead of its bitstring representation. For this reason, we are not constrained to use traditional genetic operators. Thus, diversity-generation (mutators) and diversity-destruction (crossover-like or recombination) operators have been designed, making use of specific problem knowledge. Moreover, since EO is implemented using C++, an Object Oriented language, operators act over the evolvable objects via their interfaces, and they never modify the objects directly. This property makes unnecessary the use of penalty functions or repair methods to deal with invalid objects generated by the operators: the FLSs themselves control that changes ordered by the operator are carried out in such a way that the resulting objects are always valid.

3.2.1 Recombination

This operator splices values from one matrix into another. Taking into account that these two FLSs do not necessarily have the same dimensions, the recombination cannot be performed in a trivial way.

The operator works as follows:

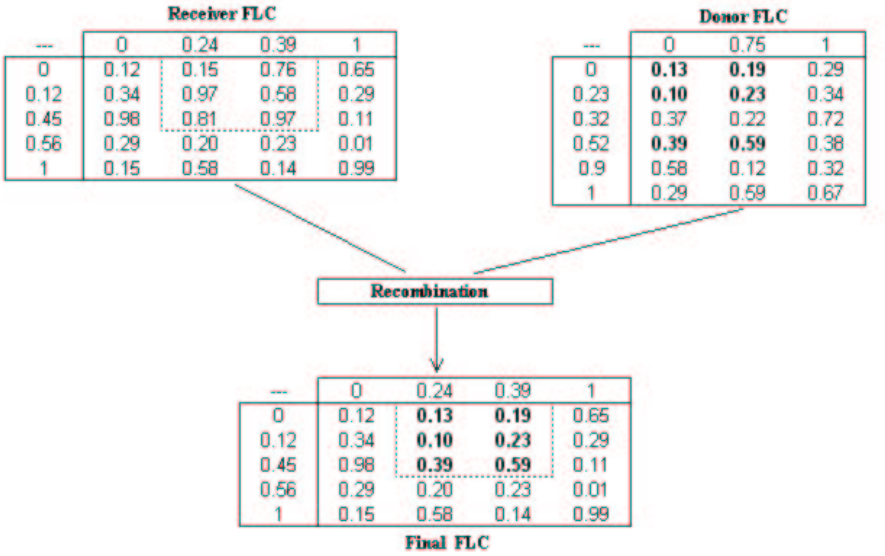

Fig. 2. Recombination operator: the resulting FLS has values in its consequents matrix belonging to both parents.

(1) Randomly choose a FLS to be modified (we will call it M_r, or *receiver FLS*), and another that will provide the genes to be recombined (the M_d, or *donor FLS*). Only the first one, M_r, will be changed.
(2) Choose a random block of cells from the consequent matrix of M_r. To specify this block we only need to randomly select two cells corresponding to the top left corner and bottom right corner of the block, respectively. Call these two cells $M_r[r_1, c_1]$ and $M_r[r_2, c_2]$. The values of the cells included in this block will be changed by values coming from M_d.
(3) For each cell $M_r[r_i, c_j]$, where r_i goes from r_1 to r_2, and c_j goes from c_1 to c_2 do the following:
(4) From the first column of the donor, M_d, select the cell whose value is closest to $M_r[r_i, 0]$. Design that cell as $M_d[r_d, 0]$.
(5) From the first row of the donor, M_d, select the cell whose value is closest to $M_r[0, c_j]$. Design that cell $M_d[0, c_d]$.
(6) Finally, change the value stored in $M_r[r_i, c_j]$ by the one stored in $M_d[r_d, c_d]$. As can be seen, the donor FLS, M_d, remains unchanged.

An example of the action of this operator is shown in Fig. 2.

3.2.2 Addition of membership functions (Size Increment mutators)

There are two independent operators that modify the structure of the FLS they are applied to by adding a membership function to input variables. One of these operators affects to input variable X while the other operates with input variable Y.

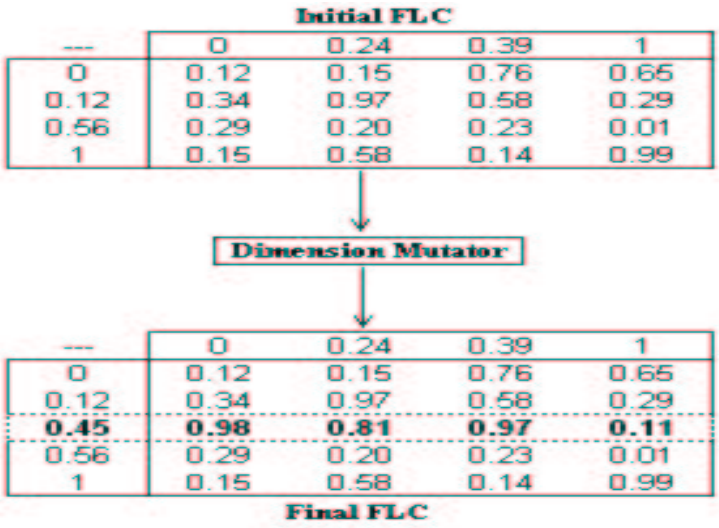

Fig. 3. Size increment mutator: a new row (membership function for X variable) is added to the FLS.

Figure 3 graphically shows the effect of incrementing the number of rows. In order to do that, once the FLS to be changed has been randomly chosen, the operator works as follows:

(1) Inserts an empty row in a random position, different from the first and last ones.
(2) Fills the cells of the new row with random values generated by a Gaussian function centered on the middle point between the corresponding to the preceding row and that corresponding to the following one.

The form these operators work ensures that the resulting FLS is always valid, because the way the values for the centroid and consequents of the new row are set results into an ordered FLS (i.e., the value for the new centroid is greater than that of the preceding row and smaller than that of the following row), and thus is valid.

The algorithm for the operator that adds a new column is the same as the one presented above but dealing with columns instead of rows.

3.2.3 Removal of membership functions (Size Decrement mutators)

There are two operators that, as the previous ones did, also modify the structure of the FLS by decreasing the number of membership functions of input variable X or Y.

These operators work by choosing a row/column to delete (different from the first and last ones) and removing it (see Fig. 4).

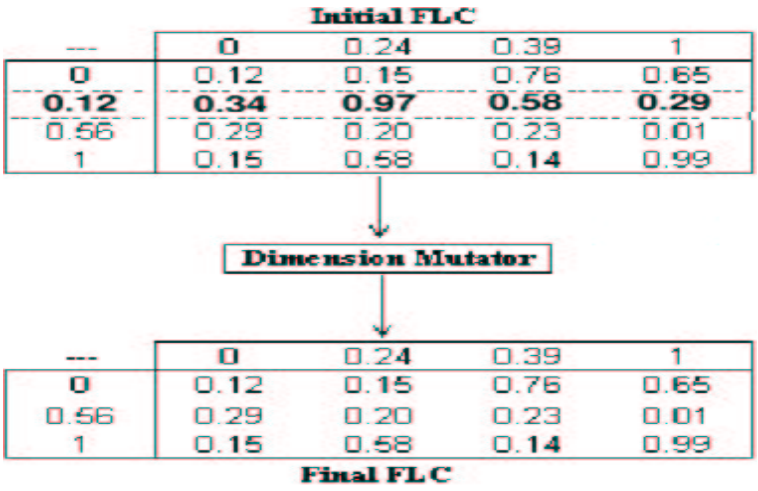

Fig. 4. Size decrement mutator: a row (membership function for the X variable) is deleted from the FLS.

Once again, they ensure that the resulting FLS is valid, given that a) they cannot delete a row/column of a FLS that has the minimum number allowed (i.e., 3 column by 3 rows); b) the first and last rows/columns cannot be chosen to be deleted; and c) obviously, the FLS remains sorted once the row or column has been removed.

3.2.4 Modification of a centroid value (Precedent mutators)

These two operators, one for input X and the other for input Y, modify one of the values stored in the FLS first row or first column, respectively.

For the operator that modifies centroids of variable Y, once the FLS to which the operator will be applied has been selected, the following operations are carried out (see Fig. 5):

(1) Randomly select a cell in the first row (because we are going to change a centroid of Y). This cell can be neither the first nor the last ones.
(2) Set the selected cell to a random value, greater than the one in the precedent column and smaller than that in the following column. Once more, a Gaussian function centred on the current value, and with asymmetric widths (distances from current value to previous and next ones, respectively) is used.

The same algorithm, with rows by columns swapped, is used for the operator that changes the values of variable X.

As in the precedent cases, these operators ensure that the resulting FLS is valid, because the first and last membership function centroids cannot be

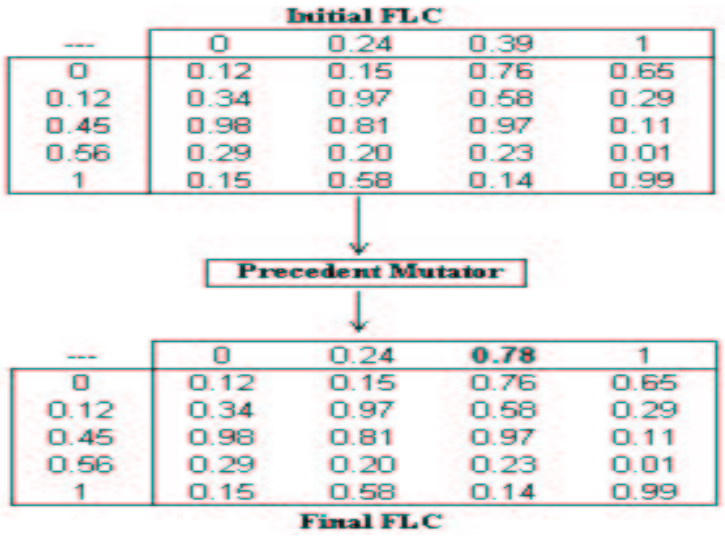

Fig. 5. Precedent mutator: The value for the centroid of the third membership function of input Y has been changed by the operator.

changed. Likewise, the way the new value for the cell is set ensures that the FLS remains sorted.

3.2.5 Modification of a consequent value (Consequent mutator)

This operator works in the following way (see Fig. 6):

(1) Randomly select a cell in the FLS consequent matrix (i.e., select a random $M[i,j]$, where $1 \leq i \leq m$ and $1 \leq j \leq n$).
(2) Put in that cell a new random value determined by a Gaussian function centred on the existing value, and varying from z_{min} to z_{max}.

3.3 Fitness Function

To calculate the fitness of each individual, a set (namely the *training set*) of input-output pairs from a known function is presented to it (known functions are shown in top left graphics of Figures 7 - 10). The fitness assigned to the FLS is the inverse of the Normalized MSE distance from the known correct outputs to the outputs produced by the FLS, calculated using the training set, i.e.:

$$F_j = \frac{N}{\sum_{i=1}^{N} ((z_i - z_{ij})/(z_{max} - z_{min}))^2} \tag{1}$$

where F_j is the fitness for the individual number j, N is the number of samples in the training set, z_i is the correct output, and z_{ij} is the output provided by

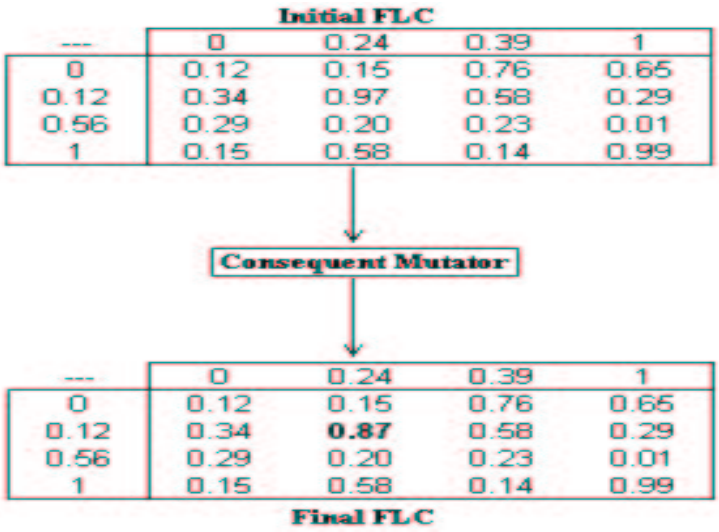

Fig. 6. Consequent mutator: one of the consequents is changed for a new, valid value.

the FLS.

A special case is when $z_i = z_{ij}, \forall i, 1 \leq i \leq m$, because F_j should be equal to $N/0$; so in this case, F_j is assigned a very high value.

It should be taken into account that FLS size does not intervene in fitness computation, since what is measured is the generalization error, which should resolve the matter since smaller networks usually generalize better.

3.4 The algorithm

An evolutionary algorithm with elitist selection [5], fixed population size and the operators described above is used. Here are its main steps:

(1) Create the first population, composed of p randomly generated individuals of random size (with an upper limit in the number of rows and columns, only for this first generation). Set the generation counter to 1.
(2) Compute the fitness value of every individual.
(3) Sort the individuals from highest to lowest fitness values.
(4) Select the q best individuals (population elite subset). Delete the remaining $p - q$ individuals.
(5) Generate $p - q$ new FLSs, increasing population size up to p. Every new individual is created by (1) duplicating one individual in the elite subset, and after that, (2) applying one of the operators to the copy. The probability of one individual in the elite subset being selected for reproduction is related to its fitness value.
(6) Evaluate the new individuals.

(7) Increment the generation counter, and go back to step 3, unless a specified number of generations has been reached.
(8) Finally, the best FLS found is the first individual (because they are sorted form highest to lowest fitness) of the current (last) population.

This algorithm is run with the following free parameters:

- Population size.
- Rate of individuals to be removed in every new generation (the *non-elite rate)*.
- Application rates of every operator (henceforth, these are normalized to become probabilities).
- Maximum number of rows and columns for the first generation.
- Maximum number of generations.

Every operator has an associated probability of being applied that does not change during the application of the algorithm (although **EO** allows adaptive operator rates). Furthermore, every new individual is created by applying one, and only one, operator to the copy or duplicate of its parent, be it the recombination operator or any of the mutators.

As usual, the recombination operator is applied with a higher rate to allow mixing of building blocks, while mutation-like operators are applied with smaller rates in order to escape from random search. On the other hand, all the mutation-like operators share the same rate in order to balance the increase and decrease of FLS size. Additionally, termination condition is fixed at a given number of generations specifically to limit the algorithm running time.

4 Experiments and results

The goal of these experiments was to validate the behavior of the operators created. Parameters of the algorithm were fixed to default values, although further study is necessary in order to establish which values are the best. Table 1 shows the values used for the parameters in the following examples.

In every experiment we attempted to obtain a FLS that estimates a known function. Several functions were used, although only four of them are shown here. For every function, the algorithm was run three times with different random seeds, in order to get average values and standard deviations. Functions have been taken from [14], chosen for being used in other works to compare results.

Parameter	Value
Population size	500
Non-elite rate	0.7
Recombination rate	0.1
Dimension mutator rates	0.01
Precedent mutator rates	0.01
Consequent mutator rate	0.01
Number of generations	300
Max. init. number of rows/columns	40

Table 1
Values for parameters.

To carry out the experiments, one file per function, with 400 points, was generated with equally spaced values for inputs X and Y. The resultant values for output Z were modified adding small quantity of random noise. Every time the algorithm was executed, 320 randomly chosen points (80% of 400) were used as the training set, while the remaining 80 points were used as *validation set*, used to test the generalizing capabilities of the FLS.

Taking into account that the algorithm deals with large matrices storing floating point numbers, it is obviously very time consuming. For this reason, the number of individuals, generations and initial maximum rows/columns was set to small values, allowing us to carry out several experiments in a relatively short time: every experiment lasted about two hours on a computer with twin Pentium II 450 MHz, and using Linux as the operating system.

Figs. 7 to 10 show the original functions to be fitted (top left), the best output produced during the three executions of the algorithm in each problem (top right), fitness evolution along generations (bottom left), and size (rows by columns, less 1) evolution along generations (bottom right). Figure headers show the function formulas that generate the original graphics.

Table 2 shows average and standard deviation of final fitness, generalization error, and size (rows by columns, less 1) of the best FLS found (the best individual in the last generation). To calculate the generalization error, once the EA has finished, a validation set of input-output pairs different from those used in the EA, was presented to the best FLS, and the normalized MSE was computed using the outputs produced by the FLS and the known ones.

Results obtained show the approach presented in this paper is s appropriate to solve the problem. Thus, as usual, the evolutionary algorithm behaves very well in the task of finding good solutions (in these cases, function shapes are

Function ID	*Fitness*	*Generalization error*	*Size*
1	54 ($\pm$8)	0.0288 ($\pm$0.0007)	34 ($\pm$3)
2	197 ($\pm$35)	0.006 ($\pm$0.003)	41 ($\pm$40)
3	114 ($\pm$8)	0.0122 ($\pm$0.0011)	18.6 ($\pm$1.2)
4	34 ($\pm$12)	0.046 ($\pm$0.012)	117 ($\pm$36)

Table 2
Numerical results: fitness, generalization error, and size (rows by columns-1) of the best individual during the different runs of the algorithm for each function.

quickly learned by the genetic algorithm), although fitting more precisely is more difficult, and it is necessary to increase the number of generations in order to get more accuracy. Nevertheless, adding more generations might not be desirable because generalization might be negatively affected, resulting on the well known problem of over-fitting, i.e., points in the training set would be approximated better and better at each generation, but approximation of points in the test set would be worst and worst.

The estimation of centroid values is quite good, as can be seen in the figures by analyzing the contours plotted when output Z is projected over the surface created by inputs X and Y. These contours show how the changes in the shapes of the original functions are reflected in the shape of the results provided by the best FLS. Fitting the values of centroids to the exact values used in the functions make us face the same problem stated above. It should be done carefully, because once reached a given generation the individuals would be fitting the errors associated to the points of the training set so that generalization would be performed very badly.

One of the most interesting results provided by this algorithm is related to the complexity of the solutions it finds. As can be seen in Tables 2 and 3, the final size of the FLSs did not grow up to the greatest possible value, as in principle might be thought. This is specially relevant because the way the fitness is calculated does not include explicit penalization of large individuals. Fitting the 400 points involved in every experiment can be exactly done if FLSs composed of $20 + 1$ rows and $20 + 1$ columns are used. This would be possibly the solution found if random search algorithms were used. But in our method, the search is guided through the use of the different genetic operators as well as the method to select individuals to be reproduced and individuals to be removed.

Table 3 compares the results obtained by the herein proposed algorithm and some others found in literature. Results being compared have been taken from Pomares [14] who developed an automatic method for fuzzy system design. Pomares' algorithm is based on mathematical analysis and uses gradient methods when any value has to be optimized. More precisely, Pomares' algorithm

Function ID	*Proposed Algorithm*		*Pomares [14]*		*Rojas [16]*		*Cherkassky [2]*
	NMSE	Size	NMSE	Size	NMSE	Size	
1	**0.0288**	34	0.047	36	0.089	64	0.033
2	**0.006**	34	0.008	48	–	–	0.016
3	**0.0122**	18	–	–	–	–	–
4	**0.046**	117	0.031	64	–	–	0.106

Table 3
Comparison between the algorithm presented in this paper and some others in literature. Columns show, for each function and algorithm, the Normalized Mean Square Error and number of fuzzy rules. For the column related to Cherkassky no information about size is reported.

is divided into four steps: the first one calculates the optima consequents for the fuzzy rules for a fixed number of membership functions. The second step optimizes the central values of the membership functions and the consequent values at the same time. The third step analyzes the surface provided by the generalization error in order to determine in which variables is necessary to increment the number of membership functions, trying to improve the normalized mean squared error. On the last step, from the various configurations found in the process, the one which represents the best trade-off between accuracy and complexity is chosen as the final fuzzy system. In the other hand, the work by Cherkassky [2] corresponds to the method named *Constrained Topological Mapping*, in which self-organizing maps were used to divide the input domain into different areas not connected with each other. Finally, Rojas [16] developed an algorithm for function approximation that optimized both the number of rules and the rules themselves. To do this, the rule consequents were determined by weighting the output provided by each input data with the degree of activation of every rule. The number of membership functions was optimized by minimizing an index that determined when two of those membership should be joint into only one.

Some conclusions can be extracted when analyzing Table 3. Firstly, the number of rules automatically found by the proposed algorithm is very often less than the number of rules found by the rest of algorithms, and this is achieved without imposing any restriction with respect to the size when the algorithm generates new individuals. Secondly, the FLSs created by the proposed algorithm perform better than the others. This is specially important taking into account that no local method is used to improve the results found by the EA. Finally, the method can be applied to any kind of function, even when its exact shape is unknown, given that it only needs a set of inputs-outputs pairs, without regarding if the function to be approximated is differentiable or continuously defined.

5 Conclusions and future work

The algorithm presented here takes advantage of the fact that the object being evolved is itself the solution to the problem, not a representation. This has allowed us to design new operators that include specific problem knowledge; for this very fact, every time an operator is applied, the new individual it produces is always valid, and penalty functions and repair methods need not to be used.

The evolutionary algorithm performs well in the task of learning the shape of the function to be fitted (this is, approximating the solution to the optimum one). This means that membership function centroids are well estimated, and even trends in the rule consequents are detected, although exact values are more difficult to find. All this is done while keeping a small number of membership functions; thus, the fitness function used is simple to calculate, and good enough to ensure that good results are found without requiring any method that penalizes big FLSs.

The research presented in this paper will be continued along the following lines:

- To use some kind of local searching in order to fine tune the solutions found by the EA, getting more accurate values for consequents.
- To find some other way to compute the fitness in order to significantly reduce the time needed to run the algorithm.
- To test variations on current operators, especially in recombination, to allow the interchange of full rows/columns, and, in general, the interchange of cells from the first row and first column. This is difficult given that every individual has a different size.
- To generalize the method so that any number of input variables can be used. This represents a serious challenge given that a new representation and a different way to handle the individuals, as well as new operators would be needed.

Acknowledgements

This work has been partially supported by the FEDER I+D project 1FD97-0439-TEL1, and by the CICYT projects TIC97-1149 and CICYT, TIC99-0550 (PUFO).

We would also thank the anonymous referees who reviewed this work. Their comments have helped to significantly improve the paper.

$$f(x, y) = sin(xy)$$

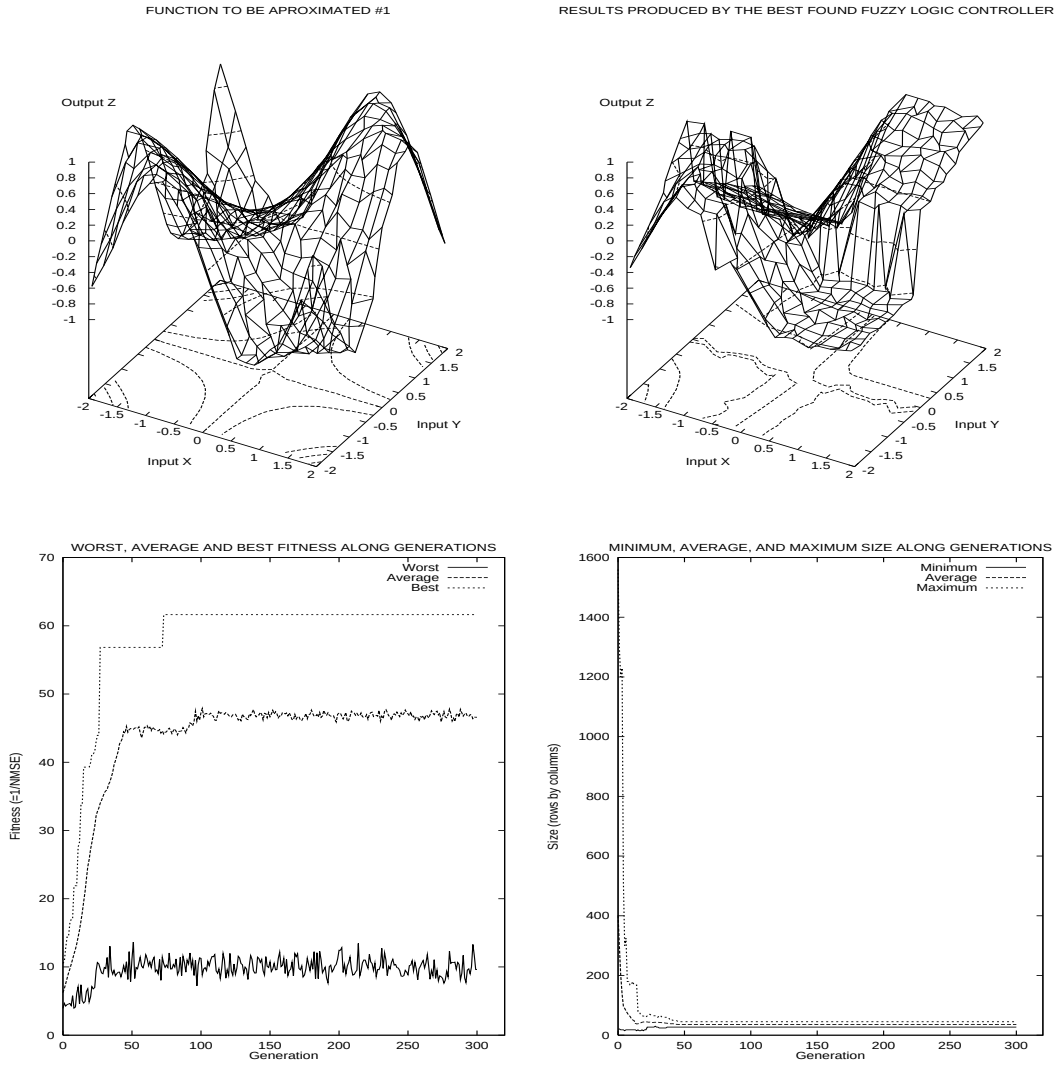

Fig. 7. Result number 1: function to be fitted (top left), solution found by the algorithm (top right), fitness evolution (bottom left), and size evolution (bottom right, logarithmic scale).

$$f(x, y) = e^{x sin(\pi y)}$$

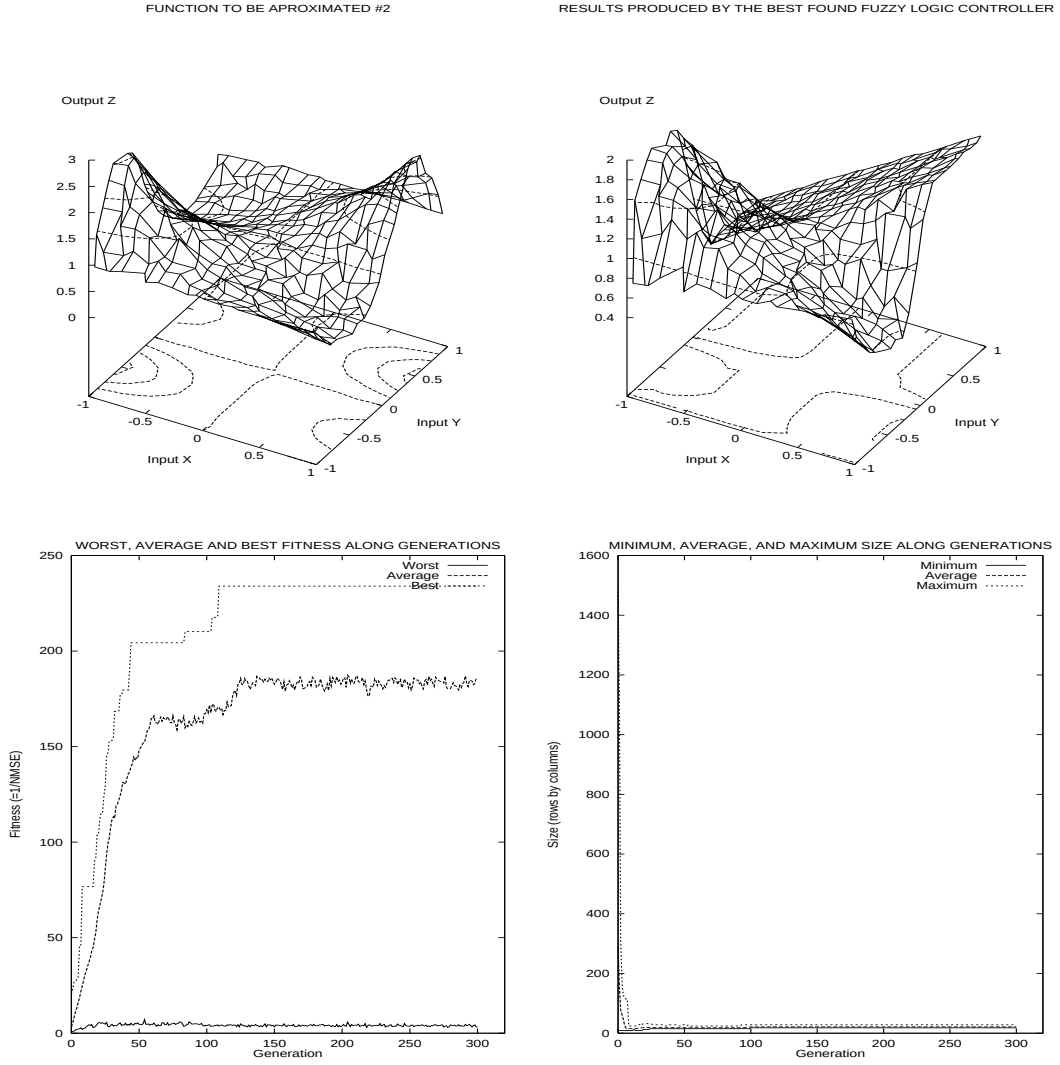

Fig. 8. Result number 2: function to fit (top left), solution found by the algorithm (top right), fitness evolution (bottom left), and size evolution (bottom right, logarithmic scale).

$$f(x,y) = \frac{40e^{8((x-0.5)^2+(y-0.5)^2)}}{e^{8((x-0.2)^2+(y-0.7)^2)}+e^{8((x-0.7)^2+(y-0.2)^2)}}$$

FUNCTION TO BE APROXIMATED #3

RESULTS PRODUCED BY THE BEST FOUND FUZZY LOGIC CONTROLLER

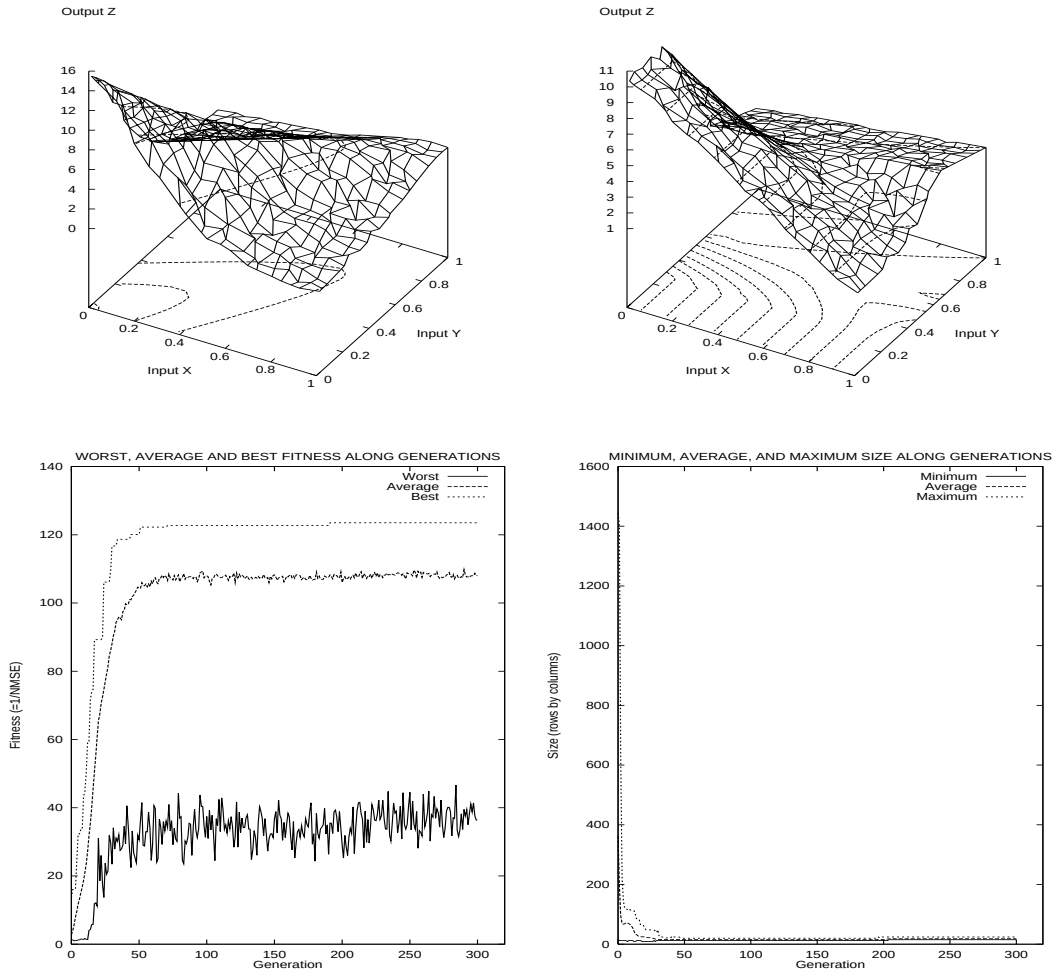

Fig. 9. Result number 3: function to fit (top left), solution found by the algorithm (top right), fitness evolution (bottom left), and size evolution (bottom right, logarithmic scale).

$$f(x,y) = sin(2\pi\sqrt{x^2+y^2})$$

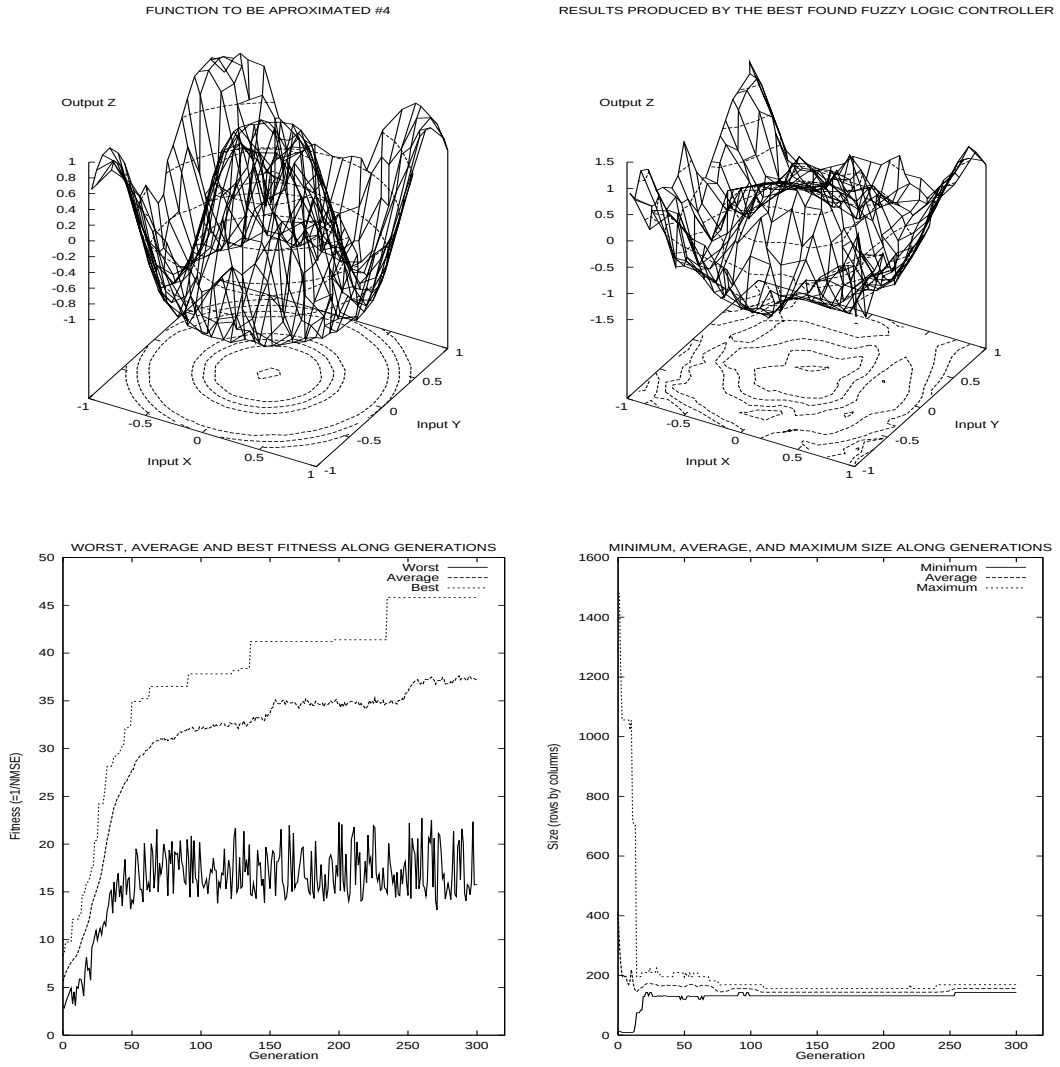

Fig. 10. Result number 4: function to fit (top left), solution found by the algorithm (top right), fitness evolution (bottom left), and size evolution (bottom right, logarithmic scale).

References

[1] F. Herrera amd M. Lozano and J.L. Verdegay. *Genetic Algorithms and Soft Computing.* Physyca-Verlag, Heidelberg, 1996.

[2] V. Cherkassky, D. Gehring, and F. Mulier. Comparison of adaptative methods for non-parametric regression analysys. *IEEE Trans. Neural Networks*, 7(4):969–984, July 1996.

[3] O. Cordón, M.J. del Jesús, F. Herrera, and E.López. Selecting fuzzy rule-based classification systems with specific reasoning methods using genetic algorithms. In *Proceedings of the International Fuzzy Systems Association (IFSA)*, pages 424–429, Prague, Czech Republic, june 1997.

[4] L. Davis. *The Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York, 1991.

[5] K.A. De Jong. An analysis of the behavior of a class of genetic adaptative systems. *Dissertation Abstract International*, 1975.

[6] L.J. Eshelman and J.D. Shaffer. Real-coded genetic algorithms and interval-schema. In *Foundation of Genetic Algorithms 2*, pages 187–202. Morgan Kaufmann Publishers, 1993.

[7] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* AddisonWesley, Reading, MA, USA, 1989.

[8] A. Homaifar and E. McCormick. Simultaneous design of membership functions and rule sets for fuzzy controllers using genetic algorithms. *IEEE Transactions on fuzzy systems*, 3(2):129–139, May 1995.

[9] H. Ishibuchi, T. Murata, and I.B. Trksen. A genetic-algorithm-based approach to the selection of linguistic classification rules. In *Proceedings of EUFIT'95*, pages 1414–1419, Aachen, Germany, 1995.

[10] C. Karr. Applying genetics to fuzzy logic. *AI Expert*, 6:26–33, 1991.

[11] C. Karr and E. Gentry. Fuzzy control of ph using genetic algorithms. *IEEE Trans. Fuzzy Systems*, 1:46–53, 1993.

[12] D. Kim and C. Kim. Forecasting time series with genetic fuzzy predictor ensemble. *IEEE Transactions on Fuzzy Systems*, 5(4):523–535, November 1997.

[13] Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs.* Springer-Verlag, NewYork USA, 3 edition, 1999.

[14] Hector Pomares. *Nueva metodología para el diseño automático de sistemas difusos.* PhD thesis, University of Granada (Spain), 1999.

[15] I. Rojas, J.J. Merelo, J.L. Bernier, and A. Prieto. A new approach to fuzzy controller designing and coding via genetic algorithms. In *Proceedings of the 6th IEEE International Conference on Fuzzy Systems*, volume II, pages 1505–1510, Barcelona, Spain, 1997. IEEE Service Center, NJ, USA.

[16] I. Rojas, H. Pomares, J. Ortega, and A. Prieto. Self-organized fuzzy system generation from training examples. *IEEE Trans. Fuzzy Systems*, 8(1):23–36, February 2000.

[17] J.J. Merelo; J. Carpio; P.A. Castillo; V. Rivas; G. Romero. Evolving Objects. *in Proc. of Int'l Workshop on Evolutionary Computation (IWEC'2000) pp. 202-208. State Key Laboratory of Software Engineering. Wuhan University*, 2000.

[18] Teo Lian Seng, Marzuki Bin Khalid, and Rubiyah Yusof. Tuning of a neuro-fuzzy controller by genetic algorithms with and application to a coupled-tank liquid-level control system. *Engineering Applications of Artificial Intelligence*, 11:517–529, 1998.

[19] Teo Lian Seng, Marzuki Bin Khalid, and Rubiyah Yusof. Tuning of a neuro-fuzzy controller by genetic algorithm. *IEEE Transactions on Systems, Man, and Cybernetics*, 29(2):226–236, April 1999.

[20] Magne Setnes and Hans Roubos. Ga-fuzzy modeling and classification: Complexity and performance. *IEEE Transactions of Fuzzy Systems*, 8:509–522, October 2000.

[21] P. Siarry and F. Guely. A genetic algorithm for optimizing Takagi-Sugeno fuzzy rule bases. *Fuzzy Sets and Systems*, 99:37–47, 1998.

[22] A. Wright. Genetic algorithms for real parameter optimization. In *Foundation of Genetic Algorithms*, pages 205–218. Morgan Kaufmann Publishers, 1991.

[23] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.