FISSION: An Evolutionary Method for Fuzzy Learning

by

Eric A. Morris

(Under the direction of Walter D. Potter)

Abstract

This paper presents an evolutionary method of configuring fuzzy inference systems, entitled FISSION (Fuzzy Inference System Setup by evolutION). The primary goal of FISSION is to provide a robust software tool applicable to a wide range of supervised training scenarios. Another aim is to produce human-interpretable fuzzy inference systems with a limited number of fuzzy sets and inference rules, which nonetheless attain a high degree of accuracy. In order to accomplish these goals, this paper draws on techniques related to genetic programming, real parameter optimization, and user interface design. User-friendly software has been developed which implements many of the ideas discussed herein, and this paper presents an application of the software to predict the rate of biosolids composting.

Index words: Fuzzy logic, Genetic programming, Kalman filter, Biosolids composting

FISSION: An Evolutionary Method for Fuzzy Learning

by

Eric A. Morris

B.S. in Mathematics, The University of Georgia, 2004

B.S. in Computer Science, The University of Georgia, 2004

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Master of Science

Athens, Georgia

2005

FISSION: An Evolutionary Method for Fuzzy Learning

by

Eric A. Morris

Dedicated to the glory of God

Acknowledgments

I would like to thank Dr. Potter and Dr. McClendon for instructing me in the field of computational intelligence, and for their helpful suggestions. Also, I thank Dr. Covington for insight into many topics, and for providing me with employment during my studies. I express my gratitude to my mother for helping proofread this paper, and to my father for always setting an example of excellence. Last but not least, my wife Sarah deserves the greatest appreciation for supporting me in this project and in everything I do.

Table of Contents

Chapter 1

Introduction

A central problem in the field of Artificial Intelligence involves capturing mathematical, natural, and mental processes. Artificial neural networks are very effective at automatically finding functional relations between a set of input values and their resultant outputs. However, the trained network sheds very little light on the process itself; it is merely a mathematical abstraction. Fuzzy inference systems (FIS), on the other hand, provide a clearly understandable mapping from input variables to outputs by using fuzzy sets, also known as "linguistic variables," (Zadeh 1975) along with if-then rules. Unfortunately, the FIS cannot train itself and often the exact calibration information is difficult to elicit from a human expert (Sugeno 1985).

A hybrid machine learning/fuzzy inference system is an attractive proposal with the potential to produce an optimal predictive or classifier model which can be easily interpreted and verified by humans. Effective training processes for such a model are useful in advancing the study of human learning and understanding. Evolutionary computing provides a robust global optimization strategy that is simple and flexible enough to apply to the problem of FIS calibration. This paper presents such a method, entitled FISSION (Fuzzy Inference System Setup by Evolution).

1.1 Project Goals

The goal of this project was to develop a general evolutionary method for fitting an FIS to a given data set. A similar approach was originally proposed by Pham et al. and Thrift (both 1991), and has been developed by several papers in the past decade. This project proposes a unique combination of hybrid chromosome representation, overfitting prevention methods, and flexibility. The latter characteristic means that FISSION should produce reliable models in a wide variety of supervised training scenarios. Model reliability is herein measured by root mean squared (RMS) error. It will also be convenient at times to cite the R^2 value (squared correlation coefficient) for a model. These statistics will be compared to results obtained using a neural network based FIS generator called ANFIS (MathWorks, 2005).

A secondary goal has been to make the fuzzy models as human-understandable as possible. In general, this involves minimizing the number and complexity of the inference rules and fuzzy sets. FISSION incorporates features to promote this “parsimony” criterion, allowing the user to specify the maximum number of rules and to penalize models based on complexity.

1.2 Model Representation

The models developed by FISSION are Takagi-Sugeno style fuzzy inference systems. In short, they consist of three components: nonlinear input membership functions, if-then fuzzy rules, and linear (or constant) output membership functions. All components are trained simultaneously, but are represented differently. The output of an FIS is given by a linear combination of the output member functions, determined by the firing strength of the rules. It represents a prediction of the observed output variable based on the inputs given.

The input membership functions are represented by a string of real numbers, which gives their parameterization. They represent levels of fuzzy set membership for each input. These are configured by the evolutionary algorithm (EA) using standard real-parameter crossover.

Sets of fuzzy inference rules are represented in binary tree format. Each FIS contains a single tree representing a set of rules. The antecedent of a rule contains references to input member functions joined by the logical connectives $\{AND, OR, NOT\}$. The consequent of each rule is its corresponding output membership function. Rules are configured using genetic programming.

Finally, the output functions are represented as a string of real numbers, like the input membership functions. However, they represent constant or linear functions of the original input values. They are also trained differently; after each iteration of the EA, they are configured using the Kalman method of least squares. This approach is similar to the one taken by ANFIS.

1.3 Model Evaluation

Evolutionary algorithms are population-based search strategies. The search begins with the generation of an initial population of individuals. FISSION evaluates the fitness of each individual based on RMS error and (optionally) parsimony. Individuals with higher fitness are more likely to be selected to reproduce and/or survive (a fundamental characteristic of evolutionary algorithms). The best individuals obtained in the course of training are remembered. When the training is finished, these elite individuals are tested on previously unseen data. The resulting RMS error is used to judge the completed model.

FISSION has been tested using supervised training data from the field of biosolids composting (Liang et al. 2003a). The data set consists of inputs to a composting system, (namely moisture, time, and temperature) and the resultant O_2 uptake rate (an indicator of microbial activity). Liang et al. (2003b) presents the results of using an artificial neural network (ANN) to predict O_2 uptake rate based on the provided inputs. Improved (unpublished) results have been obtained by using ANFIS to generate a fuzzy inference system to perform the prediction. However, the high performance models obtained all had some undesirable characteristics, such as an excessive number of rules. FISSION addresses some of these issues while maintaining similarly low error rates.

Chapter 2

Fuzzy Inference Systems

Fuzzy logic is based on the concepts of fuzzy set theory, which was developed by Zadeh (1965). A fuzzy set S differs from a traditional "crisp" set in that an element belongs, or fails to belong, to S *to a certain degree*. More precisely, membership in S is defined by a function $f_S : \mathbf{R} \rightarrow [0, 1]$, where 1 indicates full or certain membership and 0 indicates no membership. In fact, the domain of the membership function need not be all of the real numbers; it may comprise some subset of $\mathbf{R}$, or may even be defined only on a discrete/countable set.

2.1 Fuzzy Set Theory

Operators are defined on fuzzy sets, analogous to the way they are defined on crisp sets. Assuming fuzzy sets S and T and membership functions $f_S : D \rightarrow [0, 1]$ and $f_T : D \rightarrow [0, 1]$, membership in the union, intersection, and complement are defined for all $x \in D$ as follows:

$$f_{S \cup T}(x) = f_S(x) + f_T(x) - f_S(x) \cdot f_T(x) \tag{2.1}$$

$$f_{S \cap T}(x) = f_S(x) \cdot f_T(x) \tag{2.2}$$

$$f_{\sim S}(x) = 1 - f_S(x) \tag{2.3}$$

Alternatively, membership in the union and intersection can can be defined for all $x \in D$ by:

$$f_{S \cup T}(x) = \max(f_S(x), f_T(x)) \tag{2.4}$$

$$f_{S \cap T}(x) = \min(f_S(x), f_T(x)) \tag{2.5}$$

Membership functions can take on a wide variety of shapes, such as triangular, trapezoidal, bell curved, or sigmoidal. Any curve which can be parameterized by a continuous or piecewise function of the form shown above could conceivably be used as a fuzzy set. For example, a bell-shaped

(Gaussian) fuzzy set S can be parameterized by:

$$f_S(x; \mu, \sigma) = e^{\frac{-(x-\mu)^2}{2\sigma^2}} \tag{2.6}$$

where $x \in (R)$, μ is the mean (peak) of the bell curve, and σ is the standard deviation. Note that this is an example of a *normal* fuzzy set; that is, it has at least one element with membership level equal to 1. As a design decision, all membership functions used in FISSION are normal.

While fuzzy set theory is useful in its own right, the primary concern of this project is the application to fuzzy inference systems (also called fuzzy controllers). Engelbrecht (2003) defines a fuzzy controller "as a nonlinear static function that maps controller inputs onto controller outputs." In the case of FISSION, the inputs are variables in the problem domain and the outputs are the values we wish to predict.

2.2 Fuzzy Controllers

Typically, a fuzzy controller defines several fuzzy sets (characterized by their membership functions) for each input. Some of the first work on fuzzy controllers *per se* was done by Mamdani et al. (1975). In Mamdani's design, membership functions are defined on controller outputs as well as the inputs. In this scheme, fuzzy rules have the following form. Note that `InputMF(i, j)` denotes the `j`th membership function defined for input `i`. Similarly, `OutputMF(i, j)` denotes the `j`th membership function defined for output `i`:

```
1.  IF Input1 IS InputMF(1,1) AND Input2 IS InputMF2(2,1)
    THEN Output1 IS OutputMF(1,1)

2.  IF Input1 IS InputMF(1,2) OR Input2 IS InputMF(2,2)
    THEN Output1 IS OutputMF(1,2)

3.  IF NOT Input1 IS InputMF(1,3) THEN Output1 IS OutputMF(1,3)
```

When evaluated on a particular set I of input values, each rule R gains a *firing strength* $F(I, A)$ which is determined by the antecedent A. The firing strength of an individual assertion (e.g. `Input1 IS InputMF(1,1)`) is computed by evaluating the specified membership function on the specified input value. The firing strength of the entire antecedent is found by combining the strengths of the assertions using set operations corresponding to the logical operators. In particular, intersection ($\cap$) corresponds to `AND`, union ($\cup$) corresponds to `OR`, and complement ($\sim$) corresponds to `NOT`. So, taking into account the preceding discussion of fuzzy set theory, the above rule #1 can be evaluated as follows. As regards notation, F represents a function that operates on a set of input values and an antecedent *or part of an antecedent*. Assume that:

$$F(I, \texttt{'Input1 IS InputMF(1,1)'}) = 0.5 \tag{2.7}$$

$$F(I, \texttt{'Input2 IS InputMF(2,1)'}) = 0.2 \tag{2.8}$$

Then it follows that

$$\begin{aligned} F(I, A_{R1}) &= F(I, \texttt{'Input1 IS InputMF(1,1)'}) \times & (2.9) \\ &\quad F(I, \texttt{'Input2 IS InputMF(2,1)'}) & (2.10) \\ &= 0.1 & (2.11) \end{aligned}$$

Once firing levels have been found for all rules, *defuzzification* takes place to determine the value Z for each output. In the Mamdani design, this process involves taking the union of modified versions of the output fuzzy sets, then finding the centroid of this (combined) region. Typically, the output membership functions are truncated or scaled according to the firing levels of the rules which specify them as consequents. A new, combined output membership function C is created as the union of these modified functions. To find the centroid Z, integration is performed as follows. Here D is the domain of the output being computed:

$$Z = \frac{\int_{x \in D} x f_C(x) dx}{\int_{x \in D} f_C(x) dx} \tag{2.12}$$

2.3 The Takagi-Sugeno Controller

In 1985, Takagi and Sugeno introduced an alternative to the Mamdani approach which is more efficient computationally (Ying et al. 1999) and allows the controller outputs to vary directly with the input values. Instead of using fuzzy sets for the output membership functions, each rule gives a crisp output directly. These rule outputs are then combined to find the controller output.

In a zero-order Takagi-Sugeno system, the consequent of each rule R_i is a single constant c_i. The output Z of a system with N_r rules can be computed by

$$Z = \frac{\sum_{k=1}^{N_r} F_k c_k}{\sum_{k=1}^{N_r} F_k} \tag{2.13}$$

On the other hand, in a first-order Takagi-Sugeno system with N_{in} inputs I_k, the consequent of every rule is a linear combination $c_0 + c_1 I_1 + ... + c_{N_{in}} I_{N_{in}}$ of the origninal input values. Accordingly, the controller output is given by

$$Z = \frac{\sum_{k=1}^{N_r} F_k(c_0 + c_1 I_1 + ... + c_{N_{in}} I_{N_{in}})}{\sum_{k=1}^{N_r} F_k} \tag{2.14}$$

FISSION uses the Takagi-Sugeno approach because the evolutionary training procedure needs to evaluate fuzzy models very frequently. Hence, a more efficient FIS will receive more training in a given amount of time. In addition, there is some intuition that this approach will yield a superior model. As Engelbrecht (2003) states, "for Takagi-Sugeno controllers, the fact that the consequent of rules is a mathematical function, provides for a more dynamic control."

Chapter 3

Evolutionary Algorithms

An evolutionary algorithm is an optimization procedure designed to find global extrema within a search space by simulating the process of natural selection. An EA has three major components: an population of individuals coded by some representation, a fitness function for evaluating the desirability of an individual, and finally, a set of genetic operators which control how genetic information is passed from one generation to the next. The EA is a probabilistic optimization technique. It relies upon the Schema Theorem which states that segments of a representation which contribute to high fitness grow exponentially in number as the search proceeds. The schema theorem holds for a particular problem if the chosen genetic operators do not disrupt the representation of such "building blocks." (Holland 1975)

The power of the EA is best demonstrated in nonlinear search spaces with multiple local optima. More traditional search methods (such as gradient descent) tend to converge on suboptimal solutions in such problems. The advantage of the EA is its property of "implicit parallelism", which means that the algorithm can explore multiple areas of the search space simultaneously. In addition, the EA does not rely on derivative information, which may be misleading. In fact, EAs have been used to find the global extrema of several mathematical functions which are considered very deceptive (De Jong 1975).

On the other hand, EAs have a number of disadvantages as well. They rely only upon fitness information to make moves in the search space; there is no guarantee that the search is moving in a "good" direction at any given time. Again, contrast this to gradient based methods which *always* move towards a (local) optimum. Holland's early (1975) experiments dealt with populations of fixed-length bit strings. Also, because of the large population sizes needed to solve many problems, EAs are very computation-intensive. The combination of these two factors can lead to slow performance in some cases; traditional optimization methods are typically more efficient, though often less robust.

3.1 The Simple GA

The study of evolutionary algorithms was initiated by John Holland in 1975. His early experiments involved fixed length binary representation for individuals (also referred to as chromosomes to indicate the genetic analogy of the representation). Selection, crossover, and mutation operators are defined on this "simple genetic algorithm." Of course, the fitness function is problem dependent, but so-called standard fitness functions v (Koza 1992) map $v : \mathbf{R}^N \rightarrow [0, 1]$, where N is the length of the chromosome, and 1 represents maximal fitness while 0 stands for minimal fitness. Of course, it is not always possible to couch the fitness function in these terms, since the maximum or minimum fitness may be unknown or unbounded. The GA proceeds roughly as follows:

```
1.  Generate a random initial population of chromosomes.
2.  Evaluate the fitness of each individual.  If an
    individual is better than any previously seen, remember it.
3.  Select two parents from the population.
4a. With a certain probability, breed the two parents and
    place the offspring in the next generation. With a certain
    probability, mutate the offspring.
4b. Otherwise, place the two parents unchanged in the next
    generation.
5.  If the next generation is not yet full, go to step 3.
6.  Copy the next generation as the current generation.
7.  If the GA is not finished, go to step 2.
8.  Report the best individual seen.
9.  End.
```

Holland used fitness proportional ("roulette wheel") selection to choose individuals for mating. In this scheme, an individual is selected with probability directly proportional to its value as assigned by the fitness function. Crossover was performed by selecting a random point $p \in \mathbf{Z}, 0 < p < N$, and swapping the portions of the parent chromosomes to the right of p. This method is referred to as one-point crossover. Finally, mutation involved choosing a single bit (allele) from the child chromosome and flipping its value (from 0 to 1 or vice-versa).

3.2 EA Enhancements

Many enhancements have been suggested to Holland's basic GA. In particular, a plethora of genetic operators have been proposed which yield superior performance on a variety of problem domains. Several of these will be discussed in a later section. Turning to the issue of representation, there are two topics of particular interest to the FISSION project: real coding and genetic programming.

Real coded genetic algorithms use vectors of floating point values for chromosome representation instead of binary strings. Janikow et al. (1991) showed that real coded GAs can be just as effective as their binary counterparts while using a more intuitive representation. In addition, the designer need not worry about the issue of Hamming peaks which appears in binary coding. While traditional crossover and mutation methods are generally applicable to real-coded GAs, some alternative schemes have been proposed to increase performance. These are motivated by the density of the real numbers; the initial population is unlikely to contain a representative sample of the continuous range of allele values. Specific examples of such operators will be discussed in the chapter on implementation. FISSION uses real coding to represent input and output membership functions.

In genetic programming (GP), chromosomes are represented by trees (in the graph theoretic sense). GP was pioneered by John Koza (1992), who advocates its use for a wide variety of problems. This method has the advantage of not imposing so rigid a structure on the chromosome. The layout and size of the chromosome evolve along with the information contained within. One particular advantage of GP is that variable-size genomes can be handled elegantly. There is no need to resort to what Goldberg et al. (1989b) refers to as a "messy GA," with a varible length chromosome string. Standard crossover and mutation operators are defined for trees. Tree crossover corresponds roughly to the two-point crossover used with standard representations. Mutation can take on a variety of forms which will be discussed later. Since a set of fuzzy if/then rules has a quite intuitive tree representation, it is logical to use GP when training an FIS using evolutionary methods. FISSION uses GP to train rule sets in symbiosis with sets of input membership functions.

Chapter 4

Training Fuzzy Inference Systems: Previous Concepts

Both neural network based and evolutionary methods have been applied to the problem of training fuzzy inference systems. Some of these methods operate mainly on the fuzzy sets of an FIS, while others are designed to find optimal rule sets. FISSION attempts to combine the best features of several previous approaches.

Typically, the problem of fuzzy inference system calibration falls under the category of supervised training. Supervised training can be viewed as a search over the space of possible FIS configurations, directed by a data set. Every pattern (row) in the data set contains input and output values. At each step of the training, the model makes a prediction based on each input pattern. The predictions are compared to the observed output values, producing an error value. The error value, in turn, is used to guide the search. In the case of an evolutionary training procedure, each pattern in the data set is referred to as a "fitness case."

4.1 Training with ANFIS

Perhaps the best known example of automatic FIS configuration is ANFIS, which ships with the Matlab Fuzzy Logic Toolbox (MathWorks 2005). It has been shown that fuzzy inference systems are equivalent to multi-layer artificial neural networks in computational power (Jang 1993a). Based on this result, ANFIS models an FIS as a neural network, which is trained by a mixture of backpropagation gradient descent and least squares. In particular, the input membership functions are configured by backpropagation and the output membership functions by Kalman least squares.

The rule set in ANFIS does not change during training. Rather, the set consists of the k^m possible conjunctions where m is the number of inputs, and k is the number of membership functions per input. (The number of rules can be reduced by using the built-in clustering option.) Cordon et al. (2004) would consequently classify the ANFIS training mechanism as an "adaptation" process rather than a "learning" one. FISSION, on the other hand, is intended to implement a learning process by generating an effective (and compact) set of rules.

ANFIS offers a wide variety of training options, of which many are duplicated in FISSION. These include input membership function type, AND method, OR method, defuzzification method, and output function type, to name a few. FISSION uses a customized version of the Kalman least squares code designed for ANFIS to determine output membership functions. While ANFIS provided the original motivation for FISSION, the resemblance ends here. The latter has much more in common with several EA-based approaches developed in the past fifteen years.

4.2 EA-based Training

The earliest methods for evolutionary configuration of fuzzy inference systems were mainly concerned with finding optimal rules. The problem of learning fuzzy rules falls under the more general category of evolving optimal rule-based systems (RBS). According to Cordon et al. (2004), there are three main categories of EAs designed to configure RBS.

The earliest is the so-called Michigan approach, due to Holland et al. (1978). In this method, each rule is represented by an individual in the population. Some optimal subset of the population constitutes the rule base for the RBS. The main challenges in this approach are fitness assignment to individual rules and choosing the optimal subset of individuals (Carse et al. 1996).

Introduced by Venturini (1993), the second category of RBS evolution is known as "iterative rule learning." In this approach, each rule is represented as a separate individual as in the Michigan scheme. However, the initial population consists of one or only a few individuals. Each generation, more rules are added to the population. In this way, a "core" of high fitness rules are generated early in the search. The core is constantly expanded as the algorithm proceeds.

On the other hand, the "Pittsburgh" approach (Smith 1980) represents an *entire* rule set in a single individual of the population. This solves the problems of fitness assignment and rule set determination, but introduces complexity for the genetic operators. Crossover, in particular, tends to disrupt high-fitness individuals by breaking up good rule combinations. Additionally, Pittsburgh style systems tend to have longer training times because of the complexity of each individual (Carse et al. 1996). However, these difficulties can be overcome by using techniques such as the permutation operator (Goldberg 1989a, Grefenstette 1987). Koza (1992) suggests presenting only a subset of the fitness cases for each evaluation. FISSION uses the Pittsburgh approach, incorporating variations on these suggested improvements.

The first work in the realm of *fuzzy* rule learning was done by Pham et al. (1991) and Thrift (1991). Both of these approaches used the Pittsburgh learning scheme, although they incorporated slightly different rule representations. Thrift used a "relational matrix" chromosome, where a matrix represents a single FIS rule set for the two input cart centering problem. Pham employed a decision table similar to the one used in ANFIS to represent FIS rules. His training scheme had two training stages: the best individuals from the first stage constituted the entire population of the second stage. As Cordon et al. (2004) points out, both of these representations are monolithic in the sense that the rules cannot be divided among multiple individuals.

Recently, there has been growing interest in genetic programming for configuring RBS. One of the first examples can be found in (Alba et al. 1999). In this approach, (fuzzy) rule sets are viewed as abstract computer programs, represented by type constrained syntactic trees (Cordon et al. 2004). Genetic programming is the central theme in FISSION, and a detailed description of the tree representation is given in section 5.2.2.

Of course, fuzzy inference systems incorporate fuzzy sets as well as inference rules. These sets can be parameterized by numerical values, so they can be optimized by real coded GAs of the sort described by (Goldberg 1989a). Although some approaches, e.g. (Karr 1991), optimize the fuzzy sets separately from the rule base, there is some intuition that a combined approach would have greater fitting potential. As Homaifar et al. (1995) demonstrates, optimizing the components together is superior to the approach of first optimizing the membership functions, then generating the rule base. Interdependencies between the two are fully explored using such a combined approach. Drawing inspiration from Liska et al. (1994), FISSION uses a two-chromosome approach to represent global membership functions separately from the inference rules. However, Liska used two standard chromosomes instead of the hybrid approach employed in FISSION.

The alternative to global membership functions of the type used in FISSION is incorporating membership functions within the rules themselves. This method was employed with some success by Cooper et al. (1994). However, this feature resulted in an excessive number of fuzzy sets when incorporated experimentally into FISSION.

Although all of the approaches mentioned above use the Mamdani FIS, there is also precedent for using EAs to configure Takagi-Sugeno fuzzy systems (Lee et al. 1993). Lee also included in his method the feature of penalizing FIS models which were too complex. FISSION implements a similar strategy.

Chapter 5

Software Design and Implementation

The FISSION software is designed with usability, flexibility, and accuracy in mind. It comprises two major components: a DLL (dynamic link library) containing training and inference functions, and a GUI (graphical user interface) for exposing these functions in a user-friendly way. Users load data and set training options using the GUI, and the data are subsequently processed by the DLL functions. Visual feedback and results are then displayed in the GUI.

5.1 Component Interaction

A DLL is a Windows file containing reusable code which is linked to an executable program *at run time* rather than compile time. Hence, the DLL code can be compiled once and used concurrently by multiple programs. Every DLL exports a number of functions which can be called by any program.

In FISSION, a DLL is used to encapsulate the code for training an FIS. This design decision has a major benefit: the GUI code can be kept entirely separate from the computationally intensive training code. This means that the two components can be implemented in different languages. In FISSION, the GUI is written in C#, which provides intuitive and powerful tools for user interface design, as well as features like automatic memory management. C# programs compile to Microsoft CLR (Common Language Runtime) bytecodes which are converted to native code at runtime. This procedure is called Just In Time (JIT) compilation. The DLL, on the other hand, is written in C++, which compiles to entirely native code. Because the code is compiled all at once, C++ compilers can make very significant optimizations which improve speed and reduce memory usage. In addition, programs compiled in C++ do not perform run-time checks such as array bounds checking. This omission also serves to increase speed, although it makes writing bug-free code much more difficult.

Combining an executable GUI written in C# with a DLL written in C++ allows FISSION to take advantage of the strong points of both languages. However, it also introduces an extra layer of complexity. Functions written in C++ cannot be called directly from C# code; parameters and return values must be converted into formats which the older language (C++) can understand. This process is known as marshalling. Other considerations stem from the fact that C# code is managed, meaning that the .NET framework takes care of memory management, while C++ is unmanaged, indicating that the programmer must perform memory management manually. However, the .NET framework takes care of most of these issues automatically. For instance, when a managed object is passed to unmanaged code using a function call, the object is "pinned down" in memory, meaning that the CLR cannot move it around for the duration of the call. This allows the unmanaged code to access the object using a standard pointer.

Once the technicalities of communication between components is resolved, there remains the issue of protocol. In other words, a set of common structures must be defined to allow the GUI to send commands to the DLL, and to allow the DLL to report back to the GUI. FISSION defines two structues which contain control parameters and one which contains report values.

The structures `FISConfig` and `EAConfig` contain attributes which control how fuzzy inference and training are carried out. The effect of each attribute is determined by its type. Boolean (1 or 0) attributes turn certain features on and off. Integer attributes can either select between a number of options for a certain feature or specify a whole number value. Double precision (real) attributes specify real number values. The particular meaning of each attribute will be made clear in the discussion of its relevant feature. The attributes and their data types are listed in table 5.1.

On the other hand, the structure `ReportStruct` encapsulates attributes which describe the state of the training process. These values can be obtained by the GUI at any time by calling functions exported by the DLL. The report attributes are also specified in table 5.1.

Of course, in addition to the three structures just mentioned, simple data types (e.g. integer, boolean) can be (and are) used for communication between the GUI and DLL. These primitives can be viewed as "common structures" as well, since they generally have the same meaning in both languages. Single-dimensional arrays (lists) of simple types can also be passed as parameters without explicit marshalling. Multi-dimensional arrays (matrices), on the other hand, have to be manually marshalled into single dimensional arrays before being sent. FISSION accomplishes this by writing consecutive rows of an N by M matrix into a list of length $N \times M$. The multi-dimensional

Table 5.1: Control and Report Parameters

FIS Control Parameters		
name	*type*	*description*
`globalInputMFs`	boolean	Use global input membership functions or not
`linearOutput`	boolean	Use linear or constant output functions
`maxInitialRuleDepth`	integer (value)	Maximum rule depth for initial population
`maxInputMFs`	integer (value)	Maximum # of membership functions per input
`maxOverallRuleDepth`	integer (value)	Maximum rule depth during training
`maxRules`	integer (value)	Maximum number of rules
`mfType`	integer (select)	Type of input membership function to use
`numInputs`	integer (value)	Number of inputs to the FIS
`numOutputs`	integer (value)	Number of outputs from the FIS
`numSetOps`	integer (value)	Whether to use =, > and < (3) or just = (1)
`parsimonyFactor`	double precision	Impact of rule parsimony on fitness
`probor`	boolean	Use probabilistic-or for union (or maximum)
`prod`	boolean	Use product-and for intersection (or minimum)
`pSelectInternal`	double precision	Probability of selecting internal node
`rampedInit`	boolean	Used ramped half-and-half (or random growth)
`reorderTree`	boolean	Use permutation operator (or not)
`wtaver`	boolean	Use weighted average (or weighted sum)
EA Control Parameters		
name	*type*	*description*
`blendParam`	double precision	Parameter for blend and arithmetic crossover
`crossType`	integer (select)	Type of crossover to use
`maxGens`	integer (value)	Maximum number of generations to train
`mutType`	integer(select)	Type of mutation to use
`numSubsets`	integer (value)	# of training subsets to use for Kalman procedure
`pCross`	double precision	Probability of performing crossover
`pMut`	double precision	Probability of performing mutation
`popSize`	integer (value)	EA population size
`realParamUniform`	boolean	Crossover affects whole chromosome (or not)
`selType`	integer (select)	Type of selection to use
`steadyState`	boolean	Use steady state (or generational)
`tourneySize`	integer (value)	Size of tournament in selection
Report Parameters		
name	*type*	*description*
`fitness`	double precision	Best individual's fitness
`MAE`	double precision	Best individuals's MAE
`MSE`	double precision	Best individuals's MSE
`nEvals`	integer (value)	Number of fitness evaulations performed

array can then be reconstituted on the receiving side. This technique is needed for passing the input matrix from the GUI (which loads that data set) to the DLL (which processes it).

5.1.1 DLL Component

The FISSION DLL implements three major parts (see Appendix B for the C++ code). The first is a set of classes (types) representing a Takagi-Sugeno style fuzzy inference system. The second is another set of classes which implement a hybrid evolutionary algorithm. Finally, the DLL contains a number of exported functions which group the FIS and EA capabilities into use cases. These functions can be called by the GUI (or any other program which may be written to take advantage of them). Information is passed from the GUI to the DLL via structures containing certain control parameters.

5.2 Takagi-Sugeno FIS

The main class in the FIS implementation is entitled `SugenoFIS`. This class is responsible for storing the components of a particular FIS, namely the input and output membership functions, as well as the fuzzy rule set. It also contains functions which perform fuzzy inference on a data set using the components just listed. Finally, `SugenoFIS` is responsible for determining a near-optimal set of output membership functions given a set of input membership functions and a set of rules. This is accomplished by using the Kalman method of least squares.

5.2.1 Fuzzy Set Representation

Input membership functions are represented as strings of double precision values. The entire string is referred to as the input chromosome I. The number m of inputs represented is given by the control parameter `numInputs` and the maximum number q of membership functions per input by `maxInputMFs`. Each membership function is specified by up to 4 parameters, depending on the type of function. Hence, the length of the input chromosome for a given FIS is $4mq$, and the l^{th} parameter of the k^{th} membership function defined on the j^{th} input is located at $i = 4jq + 4k + l$. Conversely, the i^{th} position of the chromosome gives the value of the $l = i \bmod 4$ parameter of the $k = \frac{i-l}{4} \bmod q$ membership function defined on the $j = \left\lfloor \frac{i}{4q} \right\rfloor$ input. Of course, the range of valid values for each position i is determined by the type of membership function being used, which is specified by the control parameter `mfType`.

Four types of input membership functions are supported by the current version of FISSION. They are referred to as “triangular” (f_1), “trapezoidal” (f_2), “Gaussian” (f_3), and “generalized Gaussian” (f_4). The definitions for the first three are similar to functions found in the Matlab Fuzzy Logic Toolbox (FLT) (MathWorks 2005). The last is an exponential modification of the ‘gbell” function from the FLT, which uses a sigmoid function. Each of these functions takes at most four parameters. The first parameter specifies the center of the membership function, and subsequent parameters are all *positive* and define points on the curve *relative to the center*. For example, in the triangular membership function, the second parameter is subtracted from the first (center) to obtain the left “foot” of the triangle, and the third parameter is added to the first to get the right “foot.” This approach has the advantage that the list of parameters need not be sorted before the membership function is applied. Here follow definitions of each type of membership function, applied to an input x, in terms of the parameters $p_1 \cdots p_4$:

$$f_1(x; p_1, \cdots, p_4) = \max\left(\min\left(\frac{x-(p_1-p_2)}{p_2}, \frac{(p_1+p_3)-x}{p_3}\right), 0\right) \tag{5.1}$$

$$f_2(x; p_1, \cdots, p_4) = \max\left(\min\left(\frac{x-(p_1-p_3)}{p_3}, 1, \frac{(p_1+p_2+p_4)-x}{p_4}\right), 0\right) \tag{5.2}$$

$$f_3(x; p_1, \cdots, p_4) = e^{-\frac{(x-p_1)^2}{2p_2^2}} \tag{5.3}$$

$$f_4(x; p_1, \cdots, p_4) = \begin{cases} e^{-\frac{(x-p_1)^2}{2p_3^2}} & \text{if } x < p_1 \\ 1 & \text{if } p_1 \leq x \leq (p_1+p_2) \\ e^{-\frac{(x-(p_1+p_2))^2}{2p_4^2}} & \text{otherwise} \end{cases} \tag{5.4}$$

Input membership functions can be randomly generated, which is useful for creating the random initial EA population. The entire chromosome is generated at once. The first position in each parameterization designates the “center” (one of the maximum values) of the membership function, and subsequent positions are defined relative to the center. The center parameters of the membership functions are initially evenly distributed across the input space, then each center is modified by a normal random variable with standard deviation equal to $\frac{1}{20}$ of the size of the space. The other parameters are chosen uniformly from an interval about their respective centers. Non-center parameters may have different interpretations for different membership function types; for instance, the second parameter of a Gaussian membership function represents a standard deviation, while the second parameter of a triangular function represents the left "base" coordinate of the triangle. However, defining the parameters in this *relative* way makes a single random generation procedure compatible with all types of membership functions.

Output membership functions are represented similarly to the input ones. The output chromosome P consists of a string of double precision values, as before. The current version of FISSION supports only one output, and the number of output membership functions r is given by the control parameter `maxRules`. This is because each rule has its own output function in a Takagi-Sugeno style FIS. If the control parameter `linearOutput` is set, then the length of the chromosome is $r(m+1)$, reflecting the fact that each output membership function is a linear combination of the inputs. In this case, the i^{th} position in the output chromosome contains the coefficient of the $(i \bmod (m+1))-1$ input for the $\left\lfloor \frac{i}{m+1} \right\rfloor$ membership function. When $i \bmod (m+1) = 0$, the position contains the constant value for the membership function. On the other hand, if `linearOutput` is not set, then the chromosome will simply have length r, since each function is a constant. Hence, the output membership function g_v for a rule v, applied to input pattern X, is defined as follows:

$$g_v(X;P) = \begin{cases} P_v & \text{if not using linear output} \\ P_{v(m+1)} + \sum_{j=1}^{m} X_j P_{v(m+1)+j} & \text{otherwise} \end{cases} \tag{5.5}$$

5.2.2 Fuzzy Rule Representation

Fuzzy rules are represented in FISSION as binary trees. That is, each node in the tree has at most two children. Each `SugenoFIS` instance has its own tree which contains up to `maxRules` rules. Nodes within a tree are represented by a data structure containing information about the expression represented. Each node contains pointers to its parent and children, as well as subtree information and values (see table 5.2). Not all fields in the data structure apply to every node. There are three main types of nodes: `if/then`, `antecedent`, and `terminal`.

An `if/then` node represents the root of a fuzzy rule. The left (first) child of an `if/then` node is an antecedent node, and is mandatory. The right (second) child is another `if/then` node, and

Table 5.2: Rule Node Properties

property name	*type*	*description*
`parent`	pointer	Refers to the node's parent
`child1`	pointer	Refers to the node's left child
`child2`	pointer	Refers to the node's right child
`depthBelow`	integer (value)	Depth of the subtree rooted at the node
`nodesBelow`	integer (value)	Number of nodes in the subtree rooted at the node
`role`	integer (select)	`if/then`, `antecedent`, or `terminal`
`subRole`	integer (select)	`AND`, `OR`, `NOT`, `IS`, `INPUT`, or `MF`
`value`	integer (value)	Index of an input or membership function.
`operation`	integer (select)	`=`, `<`, or `>`

is optional. If the right child is omitted, the rule represents the last in the tree. An `if/then` node may have no parent, in which case it is the root of the entire tree. A terminal node must have as its parent the `IS` subcategory of `antecedent` nodes.

A `terminal` node represents a leaf in the tree, specifying either an input or a membership function. This is determined by the `subRole` field, which selects between `INPUT` and `MF`. The property `value` has meaning for terminal nodes, specifying either an input index j, $0 \leq j < m$, or a membership function index k, $0 \leq k < mq$.

The `antecedent` nodes stand for fuzzy logical operators. This type is further categorized by the `subRole` property, based on the operator represented. Possible operators are `AND`, `OR`, `NOT`, and `IS`. These connectives function as described in section 2. An `antecedent` node may be the child of an `if/then` node or another `antecedent` node. The left and right children of `AND` and `OR` nodes are mandatory and must be other `antecedent` nodes. `NOT` is similar, except that (being a unary operator) it cannot have a right child. `IS` takes a terminal node of subtype `INPUT` as its left child, and a terminal node of subtype `MF` as its right child. Both children are mandatory. The meaning of `IS` is determined by the `operator` property, which selects between `=`, `<`, and `>`.

Since the evolutionary algorithm used for training involves a randomly generated initial population, FISSION requires a method for generating random fuzzy rule trees. The obvious recursive procedure is:

```
PROCEDURE grow_tree(parent_node)
  IF can_generate_non_terminal(parent_node) THEN
    parent_node->left_child = CALL generate_left(parent_node)
    CALL grow_tree(left_child)
    parent_node->right_child = CALL generate_right(parent_node)
    CALL grow_tree(right_child)
  ELSE
    parent_node->left_child = CALL generate_terminal_left(parent_node)
    parent_node->right_child = CALL generate_terminal_left(parent_node)
  END IF
  CALL update_fields(parent_node)
END PROCEDURE
```

The procedure `grow_tree` would be called with a randomly generated root node as its argument. A branch of the recursion would end when no legal non-terminal children can be created. Of course, the secondary procedures called by `grow_tree` have to be implemented so that all the constraints discussed above are satisfied.

Koza (1992) suggests two variations of this basic "grow" procedure. The first, which he refers to as the "full" method, forces `generate_left` and `generate_right` to pick non-terminal children unless a tree size constraint would be violated. This approach results in maximal trees, in that every leaf occurs at maximum depth. From a genetic programming perspective, this maximality can be beneficial because it ensures that a large amount of genetic material is available. Koza's second variation, entitled "ramped half-and-half," assumes that there are P trees to be randomly initialized, and that a maximum depth of D has been specified. The method forms D groups of $\frac{P}{D}$ individuals, so that the i^{th} group will be initialized with maximum depth i. In addtion, half of each group will be initialized using the standard "grow" approach, and the other half with the "full" method.

FISSION provides the option of using either grow or ramped half-and-half by means of the `rampedInit` control parameter. The maximum tree depth during initialization is specified by the `maxInitialRuleDepth` parameter.

5.2.3 FIS Evaluation

Once an FIS model has been generated by the evolutionary algorithm and its output functions have been set by the Kalman procedure (see section 5.3), it is ready for evaluation on a data set. A data set for FISSION consists of an $n \times m$ matrix X of inputs and a length n vector Y of output values. Each row i, $0 \leq i < n$ represents an input pattern (vector containing a value for each input). The evaluation procedure applies the configured FIS to a data set and reports the resulting RMS error, R^2, and MAE (mean absolute error) values.

The first step in the FISSION model evaluation is to compute a firing strength matrix F from the input matrix X. F will be an $n \times r$ matrix where $F_{i,v}$ is the firing strength of the v^{th} rule applied to the i^{th} input pattern. F is computed recursively as described in section 2. In particular, the calculation of $F_{i,v}$ requires the values of all input membership functions applied to the input pattern X_i. These values are represented in the length mq vector f_i, where q is the maximum number of membership functions per input. The vector f_i only needs to be computed once per input pattern, and can be used to compute $F_{i,0} \cdots F_{i,r-1}$. The following procedure computes $F_{i,v}$ from f_i using the antecedent nodes of rule v:

```
PROCEDURE F(node, f)
  IF node->sub_role=AND
      RETURN and_op( F(node->left, f), F(node->right, f) )
  ELSE IF node->sub_role=OR
      RETURN or_op( F(node->left, f), F(node->right, f) )
  ELSE IF node->sub_role=NOT
      RETURN not_op( F(node->left, f) )
  ELSE IF node->sub_role=IS
      RETURN is_op( f[node->left->value * q + node->right->value] )
  END IF
END PROCEDURE
```

The application of `and_op`, `or_op`, and `not_op` proceed as described in section 2, and are controlled by the `prob` and `probor` parameters. The function `is_op` applies the fuzzy operator specified in the node's `operator` field. Operator `=` causes `is_op` simply to return the input membership function value f. Operator `>` relies on the center c of the membership function. If the given input is to the left of c, `is_op` returns 0. Otherwise, `is_op` returns $1 - f$. The `<` operator is defined symmetrically. These operators, in effect, turn a closed fuzzy set (having finite size) into an open one (having infinite size). The control parameter `numSetOps` controls whether or not `>` and `<` are used.

FISSION does not include $\leq$ or $\geq$ operators. This design decision was made for two reasons: (1) because this functionality can already be achieved using an `OR` node and (2) the meaning of such operators is not intuitive. One obvious candidate for the $\geq$ operator is $\max(=,>)$. However, the shape of the fuzzy set produced by this transformation is somewhat strange. In particular, it has a local minima when $f = 0.5$.

Once the matrix F has been computed, all of the information needed to perform fuzzy inference is available. If the `wtaver` (weighted average) control parameter is set, each value $F_{i,v}$ is normed in the following way:

$$F_{i,v}^{(new)} = \frac{F_{i,v}^{(old)}}{\sum_{j=0}^{r-1} F_{i,j}^{(old)}} \tag{5.6}$$

Otherwise, no adjustment takes place (weighted sum). The FIS output z_i on an input pattern X_i is given by the formula:

$$z_i = \sum_{v=0}^{r-1} F_{i,v} g(X_i) \tag{5.7}$$

After the entire vector Z has been computed, the RMS error and MAE are computed with respect to the vector Y of observed outputs:

$$E_{\text{RMS}} = \sum_{i=0}^{n-1} (Y_i - Z_i)^2 \tag{5.8}$$

$$E_{\text{MAE}} = \sum_{i=0}^{n-1} |Y_i - Z_i| \tag{5.9}$$

$$R^2 = \frac{\left(\sum_{i=0}^{n-1} y_i z_i - n\hat{y}\hat{z}\right)^2}{\left(\sum_{i=0}^{n-1} y_i^2 - n\hat{y}^2\right)\left(\sum_{i=0}^{n-1} z_i^2 - n\hat{z}^2\right)} \tag{5.10}$$

At this point, the FIS evaluation is complete, and the RMS error value is reported for use in the EA fitness function. FISSION also provides an FIS evaluation procedure which does not require an observed output vector Y. This function is used when the model has already been fully trained and the observed outputs for a data set are not known. In this case, FISSION reports the predicted value vector Z.

5.3 Output Function Optimization

FISSION does not use its EA to find an optimal output chromosome. Function optimization is a difficult problem for an EA in its own right and would result in prohibitively long training time if coupled with the search for effective fuzzy sets and inference rules. In addition, any randomly generated population will contain only a few fuzzy inference systems with acceptable output functions (ones which approximately map the range of input values onto the the range of observed outputs). The fitness proportional selection of such individuals would quickly destroy essential fuzzy rule diversity in the population. Hence, FISSION uses linear least squares to determine the output chromosome of an FIS *after* the input membership functions and rules have been configured by the EA. Once this is accomplished, the fitness of the individual can be computed.

5.3.1 Fuzzy Inference as Matrix Manipulation

A reformulation of the FIS evaluation procedure is necessary in order to see how linear least squares can be applied to this problem. Recall that our objective is to minimize the mean squared prediction error on an input i:

$$E_i = (y_i - z_i)^2 \tag{5.11}$$

Expanding this expression yields

$$\begin{aligned} E_i &= \left(Y_i - \sum_{v=0}^{r-1} F_{i,r} g_{i,v}\right)^2 && (5.12)\\ &= \left(Y_i - \sum_{v=0}^{r-1} F_{i,r}\left(P_{v(m+1)} + \sum_{j=0}^{m-1} X_{i,j} P_{v(m+1)+j}\right)\right)^2 && (5.13)\\ &= \left(Y_i - \sum_{v=0}^{r-1}\left(F_{i,r} P_{v(m+1)} + \sum_{j=0}^{m-1} F_{i,r} X_{i,j} P_{v(m+1)+j}\right)\right)^2 && (5.14)\end{aligned}$$

At this point, we recognize that multiplication (dot product) is occurring between two vectors of length $r(m+1)$:

$$E_i = (Y_i - A_i \cdot P)^2 \tag{5.15}$$

where P is the output chromosome and A_i is defined by

$$A_{i,k} = \begin{cases} F_{i,\lfloor \frac{k}{m+1} \rfloor} & \text{if } k \equiv 0 \pmod{m+1} \\ X_{i,(k \bmod (m+1)-1)} F_{i,\lfloor \frac{k}{m+1} \rfloor} & \text{otherwise} \end{cases} \tag{5.16}$$

The FIS evaluation procedure can now be couched in terms of matrix arithmetic (here $\overline{V}$ denotes the norm of vector V). The primary fitness measure (RMS error) is given by

$$E = \sqrt{Y - PA} \tag{5.17}$$

Having constructed the matrix A, and given the vector Y of observed outputs, it is possible to apply any one of several least squares procedures to find the output chromosome P. The simplest such method involves creating an $r(m+1) \times r(m+1)$ partial derivative matrix M. The k^{th} row in the matrix represents the partial derivative of equation 5.14 with respect to P_k. The partial derivatives are set to zero and the constant terms for each row are placed in the vector V. The resulting system represented by M and V can then be solved by the Gauss-Jordan method. Other similar approaches are possible using different matrix decompositions, such as SVD (singular value decomposition). Unfortunately, all of these methods make some assumption about the rank of the matrix used. If the matrix is rank-deficient, Gaussian elimination will fail. Even if the matrix is only *nearly* rank deficient, inaccurate results are possible. In addition, such regression-type methods are prone to overfitting since they operate on all the data at once. Early versions of FISSION tested linear least squares regression based on both QR decomposition and Gaussian elimination, resulting in good accuracy but typically poor generalization.

5.3.2 The Kalman Filter

An alternative is the Kalman filter (Kalman 1960), a recursive predictor/corrector approach originally designed for digital signal processing. This is the approach used by ANFIS to tune linear parameters (nonlinear parameters are configured using backpropagation). FISSION integrates, with modifications, a similar implementation by Jyh-Shing Jang (1991b), which appears to be a precursor to the one used in ANFIS. This implementation is classified as a "discrete" as opposed to an "extended" Kalman filter, since it performs only linear least squares fitting. It is important to note that while the output functions in a Takagi-Sugeno style FIS are linear, the overall system may be non-linear because of the input membership functions and the fuzzy logical operators which combine them.

The purpose of the Kalman filter is, paraphrasing Welch et al. (2004), to estimate the state $x \in \mathbf{R}^n$ of a discrete time controlled process governed by a linear stochastic difference equation using a measurement $z \in \mathbf{R}^m$. In the case of FISSION, the "state" to be estimated is the optimal output chromosome P, and the "measurement" used is the observed output value. "Time" passes as the procedure moves through the training patterns. The term "linear stochastic difference equation" means that the state at time t is a linear function of the state at time $t-1$, and that the process is subject to normally distributed white noise. The matrix A as described above is used to relate the state estimation (output chromosome) to the measurement (observed output value).

A critical component of the Kalman filter is the error covariance matrix S. S is a square $(r(m+1) \times r(m+1))$ matrix whose entries are initialized to 0 at the beginning of the procedure, except for the main diagonal, whose entries are initialized to ∞ (in practice, simply a large value, e.g. 10^6). P is initialized so that all of its entries are 0. The filter proceeds according to the following recursive equations, until all n rows of A have been presented:

$$S_{i+1} = S_i - \frac{S_i A_i A_i^T S_i}{A_i^T S_i A_i + w} \tag{5.18}$$

$$P_{i+1} = P_i + S_{i+1} A_i (Y_i - A_i^T P_i) \tag{5.19}$$

Note that the fraction in equation 5.18 connotes *scalar* division; $A_i^T S_i A_i$ is a single element and w is the "measurement noise covariance" (Welch et al. 2004). The parameter w is not of great importance when there is only one measurement; it serves only to influence how much new patterns affect the covariance matrix. In Jang's code, this parameter is set to 1 and is referred to as the "forgetting factor." Equation 5.19 can be viewed as adjusting the output chromosome proportional to the error (computed by $Y_i - A_i^T P_i$) in a "direction" determined by the covariance matrix and the latest pattern A.

The output chromosome is finalized after all patterns have been presented. The fitness is computed using equation 5.17 and the procedure ends. At this point, the FIS is fully configured.

5.4 Evolutionary Algorithm

The evolutionary algorithm used in FISSION is implemented by the class `TrainerEA` and utilizes a hybrid representation scheme. Individuals in the population P are represented by a pair of chromosomes: a string of real parameters for the input membership functions and a binary tree for the set of rules (as defined in section 5.2). The fitness function is based on the RMS error which results when an individual is applied to the training data set. FISSION draws on various genetic operators developed for both real parameter GAs and genetic programming.

5.4.1 Population and Evaluation

FISSION evolves fuzzy inference systems to fit a training data set T. At all times, FISSION keeps track of the best individual b_T applied to the training set, as well as the corresponding vector Z of predicted outputs. In supervised training, it is often useful to have a second data set U, referred to as the testing set, to prevent the training algorithm from over-fitting T. Typically, this is accomplished by using the testing set to determine when training should be stopped. In FISSION, each time b_T is updated, the new best FIS is applied to the testing set. The best (b_U) of *these* individuals (applied to the testing set) is remembered by FISSION as well, along with its vector of predictions. This way, if b_T turns out to over-fit the training set, the user can switch to b_U, which may generalize better (since U was not used in the fitness evaluations). Finally, a third data set V, referred to as the validation set, is used to validate the model. V is not used at all in the training process, and should therefore be a good indicator of the model quality. The three data sets (training, testing, and validation) can be randomly sampled from a larger data set, or supplied as separate files.

A random population of individuals is generated at the start of training. The size p of the population is controlled by the parameter `popSize`. Individuals are generated by creating a random pair of chromosomes, as described in section 5.2. Each individual has its own randomly generated fuzzy sets and inference rules. Once the population has been filled, each individual is evaluated by the fitness function (described below). The fitness values are stored in a vector F of length p.

The fitness function f_S maps from the space of possible fuzzy inference systems P_i onto $[0, 1]$, where S is the data set used in the evaluation. It takes into account mean squared error (MSE) on the training set and an optional parsimony score. In order to compute the former (f_e), output functions must be generated for the FIS being evaluated. This is accomplished as described in section 5.3, using the training set *only*. The parsimony score f_p is defined as the depth of the rule tree divided by the maximum overall rule depth (specified by the control parameter `maxOverallRuleDepth`). The term f_p is weighted by a user-specified `parsimonyFactor` q, which can be set to zero. The overall fitness formula is:

$$f_S(P_i) = \frac{1}{1+f_e}\,(1 - qf_p) \tag{5.20}$$

The effect of this formula is to ensure that the fitness value is between 0 and 1, and that the fitness value of an FIS is penalized in proportion to the depth of its rule tree. As mentioned before, if $f_T(P_i) > f_T(b_T)$ then the assignment $b_T := P_i$ is made. If $f_U(b_T) > f_U(b_U)$, then FISSION assigns $b_U := b_T$.

The process of fitness evaluation is expensive in terms of time and memory, especially with a large data set. In particular, the Kalman procedure for determining output functions is costly in both resources and speed. Since the fitness function is used so frequently, it is natural to look for ways to speed it up. One approach is inspired by the fact that, for large data sets, the matrix A is likely to be over-determined. That is, A (the $n \times r(m+1)$input matrix to the Kalman procedure) may well have *many* more linearly independent rows than it does columns. So, the information needed to find a good solution x to the equation $Ax = B$ may be contained in a subset A_S of the rows of A.

FISSION takes advantage of this idea by breaking the training set into k random, non-overlapping pattern subsets before training begins. The user can specify the value of k by means of the control parameter `numSubsets`. During fitness evaluation, the Kalman procedure runs on a $n \times \frac{n}{k}$ matrix instead of the whole matrix A. After the output functions have been determined, the FIS is applied to the *entire* training set to determine fitness. As Bacardit et al. (2004) points out, this type of "windowing" scheme has an added advantage in supervised training: high fitness individuals are likely to generalize well to other data, since they have already been successfully generalized from a subset of the training set.

5.4.2 EA Control

A basic generational EA proceeds as described in section 3. There is an alternative to this control scheme, known as the steady state EA. This method does not make sharp distinctions between generations. Instead, offspring are placed directly into the active population, replacing (possibly less fit) individuals. The modified procedure for a steady state EA is as follows:

```
1.  Generate a random initial population of chromosomes.
2.  Evaluate the fitness of each individual.  If an
    individual is better than any previously seen, remember it.
3.  Select two parents from the population.
4.  Select an individual from the population to die.
5.  Breed the two parents and replace the dead individual with
    their offspring. With a certain probability, mutate the offspring.
6.  If the GA is not finished, got to step 2.
7.  Report the best individual seen.
8.  End.
```

A steady state EA typically converges on an optimum faster than the corresponding generational EA, but risks losing population diversity too quickly. In FISSION, the control parameter `steadyState` provides the option of using either a steady state control scheme or a generational one. If the former is chosen, the EA will run for `maxGens*popSize` fitness evaluations. In the generational scheme, the EA completes `maxGens` generation replacements.

Each iteration of the EA begins with the selection of two parents for mating or survival. The selection paradigm gives the EA direction and distinguishes it from a purely stochastic search. Hence, all selection methods are based on fitness. FISSION offers the user a choice between selection methods via the `selType` control parameter. Available selection schemes are roulette, tournament, and rank-based selection.

In roulette wheel selection, individuals are selected with probability directly proportional to their fitness. In order to facilitate this process, FISSION keeps track of the sum f^+ of all fitness values in the population. If the generational control scheme is being used, f^+ is computed each time a generation is evaluated. On the other hand, if the steady state method is in use, f^+ is computed for the initial population and updated as follows for each replacement of individual i:

$$f^{+(\text{new})} = f^{+(\text{old})} - f_i^{(\text{old})} + f_i^{(\text{new})} \tag{5.21}$$

Roulette selection proceeds by choosing a uniform random "threshold" value $v \in [0, f^+]$. Next, a series is computed where the i^{th} term is the sum of the fitness values of individuals $0 \cdots i$:

$$w_i = \sum_{j=0}^{i} F_i \tag{5.22}$$

The selected individual P_i is determined by the first value of i such that $w_i \geq v$.

Rank-based selection is similar in concept to the roulette wheel method, except that individuals are selected with probability proportional to their fitness *rank*. This scheme has two advantages. First, if the fitness values happen to be distributed, say, exponentially, rank-based selection will prevent the few individuals with comparatively high fitness from dominating the population. Second, if the fitness values are very close together (e.g. they are asymptotic), rank-based selection will help distinguish between them. In this method, it is not necessary to keep track of f^+, since it is analytic:

$$f^+ = \frac{p^2 + p}{2} \tag{5.23}$$

However, it is necessary to rank each individual in the population based on fitness. For this purpose, a length p vector F^R is defined so that $F_i^R = \text{rank}(P_i)$. For the initial population and in the generational control scheme, F^R can be determined by a standard sorting procedure. In the steady state scheme, however, it would be very inefficient to sort the entire population after each replacement. Instead, it is possible to update the ranks of each individual in the population in a single pass:

```
PROCEDURE update_ranks(replaceIndex, replaceNew, replaceOld)
    FOR i FROM 0 TO p-1
        IF NOT i=replaceIndex THEN
            IF F[i] < replaceNew AND F[i] > replaceOld THEN
                rank[i] = rank[i] - 1
                rank[replaceIndex] = rand[replaceIndex] + 1
            ELSE IF F[i] > replaceNew AND F[i] < replaceOld THEN
                rank[i] = rank[i] + 1
                rank[replaceIndex] = rand[replaceIndex] - 1
            END IF
        END IF
    END FOR
END PROCEDURE
```

As before, the procedure chooses a uniform random value $v \in [0, f^+]$ and computes the series w_i given by

$$w_i = \sum_{j=0}^{i} F_i^R \tag{5.24}$$

Again, the selected individual P_i will be determined by the first value of i such that $w_i \geq v$.

Tournament selection is the simplest method as far as implementation is concerned. However, it requires the external control parameter `tourneySize` to specify the size v of the tournament used. Tournament selection chooses v individuals from the population using a uniform random distribution with replacement (meaning that an individual can be selected more than once). Of these v contenders, the individual P_i with the highest fitness is selected.

5.4.3 Genetic Operators

The primary engine of evolution in an EA is the crossover (recombination) operator. Crossover is typically performed frequently, as determined by the `pCross` control parameter. If generational control is used, the selected parents are simply copied into the next generation if crossover does not occur, simulating survival of the fittest. In the steady state control scheme, crossover is always performed since survival of the fittest is intrinsic. FISSION implements both real-parameter and tree crossover, both of which produce two children per mating. During a given mating, crossover will be performed either on the real parameter chromosome (fuzzy sets) or the tree chromosome (inference rules), depending on a random variable.

FISSION includes uniform, two-point, arithmetic, and blend (Eshelman et al. 1993) crossovers for optimizing the real-coded chromosome. The control parameter `crossType` determines which type of real parameter crossover is used. Once the parents P^A and P^B have been selected from the population, mating between the two is simulated by exchanging or combining some or all of the a alleles from each parent.

Uniform and two-point crossover are commonly used in binary and discrete representations. Both begin by making copies C^A and C^B of P^A and P^B. In uniform crossover, for each i, $0 \leq i < a$, C_i^A and C_i^B are swapped with probability $\frac{1}{2}$. Assuming v is the random variable used, each allele in C^A is determined as follows (C^B is defined symmetrically):

$$C_i^A = \begin{cases} P_i^A & \text{if } v < \frac{1}{2} \\ P_i^B & \text{otherwise} \end{cases} \tag{5.25}$$

C^A and C^B are then returned as offspring.

On the other hand, two-point crossover generates two uniform random "crossover points" i and j so that $0 \leq i < j \leq a$. For each k, $i \leq k < j$, C_k^A and C_k^B are swapped. More precisely,

$$C_k^A = \begin{cases} P_k^B & \text{if } i \leq k < j \\ P_k^A & \text{otherwise} \end{cases} \tag{5.26}$$

Again, C^A and C^B are returned as offspring.

Arithmetic and blend crossovers are designed with real parameter representations in mind. Child alleles take on values which are *functions* of the parent alleles. This allows the crossover operator to explore the greater cardinality of the real numbers. The "standard" versions of arithmetic and blend crossover combine *every* allele of P^A with its counterpart in P^B using a real-valued function, rather analogous to uniform crossover. FISSION provides the option of only combining alleles from a *segment* of the parents, as in two-point crossover. This option is controlled by the `realParamUniform` parameter. Arithmetic crossover is deterministic, defined by the following expression:

$$C_k^A = \begin{cases} \phi P_k^A + (1-\phi) P_k^B & \text{if } i \leq k < j \text{ OR using uniform} \\ P_k^A & \text{otherwise} \end{cases} \tag{5.27}$$

Blend crossover, on the other hand, depends on a uniform random variable v_k. Let $a_k = \min\left(P_k^A, P_k^B\right)$ and $b_k = \max\left(P_k^A, P_k^B\right)$. Then $a_k - \phi(b_k - a_k) \leq v < b_k + \phi(b_k - a_k)$, and blend crossover is defined as follows:

$$C_k^A = \begin{cases} v & \text{if } i \leq k < j \text{ OR using uniform} \\ P_k^A & \text{otherwise} \end{cases} \tag{5.28}$$

C_k^B is defined symmetrically for both crossover methods. Notice that both formulas involve a variable ϕ. In arithmetic crossover, ϕ controls the bias towards one parent or the other, and in blend crossover ϕ determines the size of the interval about the mean from which the new allele can be chosen. In both cases, $\frac{1}{2}$ is a good default choice. The value of ϕ can be set using the `blendParam` control parameter.

The secondary evolutionary mechanism in an EA is the mutation operator, which operates on a single chromosome P. Typically, mutation occurs infrequently (as controlled by the `pMut` parameter). FISSION incorporates two types of real parameter mutation: uniform creep mutation and Gaussian creep mutation. Both methods pick a single allele P_k to modify, and add to it a random variable v. The input k which is affected by the allele to be modified is determined as shown in section 5.2. In the case of uniform mutation, the random variable uniformly selects from the interval $(-\frac{\max - \min}{10}, \frac{\max - \min}{10})$, where max and min denote the maximum and minimum values of input k respectively. On the other hand, in the Gaussian approach v is a normally distributed random variable with a mean of 0 and a standard deviation of $\frac{\max - \min}{10}$.

Tree crossover, as described by Koza (1992), is analogous to two-point crossover in that both are based on swapping a single contiguous block of genetic material between parents. The general idea is to randomly pick a crossover point (node) in each parent tree, and swap the subtrees rooted at those nodes. Of course, there are technical considerations stemming from the fact that not all trees constructible in this way are syntactically valid fuzzy rules. In addition, the binary rule tree is essentially a variable-length chromosome, so issues of size come into play as well.

The first step in performing tree crossover is picking the crossover point. This node is selected by a uniform random variable v, $0 \leq v < l$, where l is the number of nodes in the tree. The value of l is given by `nodesBelow+1`, applied to the root node of the tree. Having chosen a node v, it is necessary to define what is meant by the v^{th} node in the tree, and to show how to find it. FISSION defines the v^{th} node in the tree as the v^{th} node encountered in a preorder, left-first traversal of that tree. Of course, it is not necessary to traverse the entire tree to find the $(l-1)^{st}$ node; the tree can be viewed as a BST where an element can be found in (expected) $\log_2 l$ moves. The following recursive procedure efficiently locates the i^{th} node in the tree:

```
PROCEDURE find_node(node, whichNum)
   IF whichNum=0 THEN
      RETURN node
   END IF
   IF whichNum <= node->left->nodesBelow+1 THEN
      RETURN find_node(node->left, whichNum - 1)
   ELSE
      RETURN find_node(node->right, whichNum - (node->left->nodesBelow + 2))
   END IF
END PROCEDURE
```

This procedure can be used "as is" to find the crossover point in the first parent P^A. However, performance may be improved if there is a higher probability of selecting internal nodes as the crossover point. Recall that approximately half the nodes in a binary tree are leaves, so selecting the point from all nodes in the tree with uniform probability will result in swapping leaves half the time. Koza (1992) compares leaf exchange to mere point mutation, and selects internal nodes with probability $\frac{9}{10}$. This option is available in FISSION via the `pSelectInternal` control parameter.

Once a crossover point has been chosen in the first tree, a *compatible* node must be chosen in the second parent P^B. In general, compatibility means that the two nodes must have the same `role`, although terminal nodes must match in `subRole` as well. Since there is no *a priori* way to know whether the i^{th} node in P^A is compatible with the j^{th} node in P^B, FISSION proceeds by trial and error. If an incompatible node is chosen, the procedure picks another random number and tries again. Since there are relatively even proportions of each type of node in the tree, it should not take more than a few tries to find a compatible node. A possible improvement would involve a "lookup table" of nodes indexed by number in the tree containing the node's information. Of course, this addition would introduce technical problems of its own; for instance, the lookup table would have to change (perhaps even in size) whenever the tree was altered. It is not clear that any time savings would be gained.

Having chosen crossover points in both trees, it is possible to generate the children. Child C^A receives a copy of the tree from P^A, with the subtree below the crossover point replaced with the subtree from P^B. There is one important exception: if pasting the subtree from P^B into C^A would result in a rule that exceeds `maxOverallRuleDepth`, then C^A is set as a copy of P^A. Child C^B is created symmetrically. The procedure updates the position information (`depthBelow` and `nodesBelow`) in both children.

There is one troubling issue with this method of tree crossover: it can have a disruptive effect by breaking up rules which work well together. As Carse et al. (1996) point out, this problem is intrinsic to the training of Pittsburgh style classifier systems using EAs. A solution inspired by (Grefenstette 1987) involves re-ordering the rules within the tree so that highly effective ones are close together. It is then less likely that crossover will split them up. FISSION implements this knowledge-based permutation operator by storing the firing strength sum (over all training patterns) of a rule in its root node. Immediately before crossover, the rules are sorted within the tree according to firing strength. This feature can be turned on and off by the control parameter `reorderTree`.

FISSION implements four of the commonly used tree mutation operators suggested by Engelbrecht (2003), namely function node mutation, terminal node mutation, grow mutation, and truncate mutation. Each of these methods entails picking out a single node from a tree chromosome and altering it. The `find_node` procedure defined above proves very useful for this purpose. Function node mutation merely involves selecting an `AND` or `OR` node and switching it to the other type. Terminal node mutation picks a terminal and assigns it a valid randomly generated `value`. Grow mutation picks a node in the tree and replaces it with a randomly generated compatible subtree (rather like the crossover operator, but with no second parent to contribute the subtree). Finally, truncate mutation shortens a subtree rooted at the selected node. If operating on an `IF/THEN` node, truncation simply deletes the right subtree (containing other rules). If operating on an `antecedent` node, the operation repaces the node with a randomly generated `IS` subtree (containing two terminals). Truncate mutation does not apply to terminal nodes.

5.4.4 Training Complete

Once the EA run has finished, the fully trained FIS can be retrieved from b_T. The probable best generalized FIS is stored in b_U. Either of these can be applied to the validation set (if any) to judge the model which has been constructed. All that remains is to present an interface which provides this functionality to the user.

5.5 Graphical User Interface

The graphical user interface is designed to expose all of FISSION's functionality in a user friendly manner. It accepts user input and translates them into calls to exported DLL functions. The GUI also displays feedback from the training process and several other visualizations. The interface consists of a standard Windows menu bar, a progress bar, and a tabbed environment containing most of the interface controls.

5.5.1 Menu and Progress Bars

The FISSION menu bar provides drop down menus entitled "File" and "Help." The File menu allows the user to save a trained fuzzy inference system to disk, or to load a previously saved FIS. The "New" option resets all training options to their defaults and removes any trained models from memory. There is also an option to exit the program. The Help menu has two options: one to display FISSION documentation, and the other to show copyright information. The progress bar provides visual feedback as to the status of various processes (most importantly, the training procedure).

5.5.2 Data Setup Tab

The first tab (figure 5.1) allows the user to load data sets from “comma separated values” (CSV) text files. FISSION allows the first row in the CSV file to contain column names, but all other rows must contain data. Microsoft Excel can export any type of spreadsheet to this format. Once the training set is loaded, the user can select which columns in the data set are to be used as inputs, and which one is the output. The column names may then be changed. The data setup tab also displays useful statistics relating to the distribution of column variables. The entire data set is shown in a built-in spreadsheet.

The testing and validation sets are optional and can be loaded in two ways. First, they may be loaded from CSV files like the training set. Alternatively, the user may choose to generate them as partitions of the training set. A partition can be created either by taking the last n percent of the training set or choosing n percent randomly (without replacement).

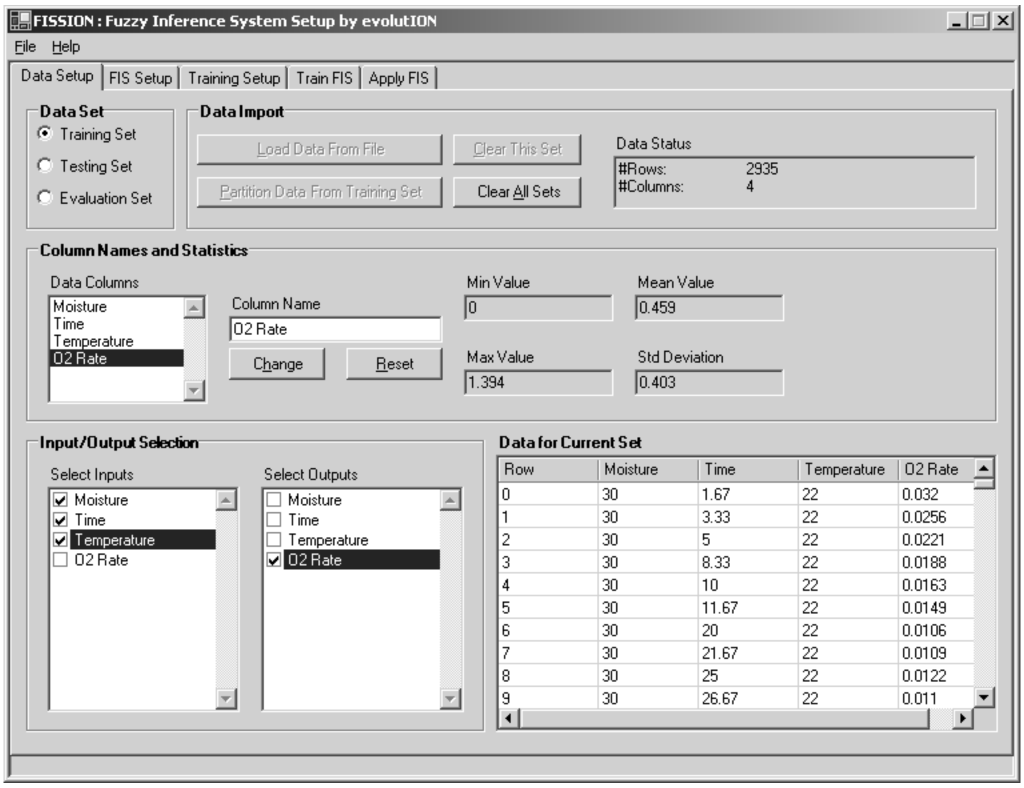
5.5.3 FIS Setup Tab

The FIS setup tab (figure 5.2) is primarily concerned with setting the control parameters relevant to FIS evaluation. These are `prod`, `probor`, `wtaver`, `linearOutput`, and `mfType`. The user can also enter minimum and maximum values for each input and output. These values are not used in the the training procedure; they merely specify bounds for the graphs shown in the GUI. They are set to the observed data minimums and maximums by default. The FIS setup tab also displays a histogram which shows the distribution of the selected input or output. This feature can be very useful for detecting outliers and errors in the data. It also shows clusters in the data which may aid the user in choosing the number of fuzzy sets per input (on the next tab). The histogram can be saved as a bitmap or copied to the clipboard at the user’s discretion.

5.5.4 Training Setup Tab

This tab (figure 5.3) allows the user to set the remaining control parameters, which deal with the training process. There are two sections on the training setup tab.

The first section presents training options related to an FIS. These include the discrete parameters `maxInputMFs`, `maxRules`, `maxInitialRuleDepth`, and `maxOverallRuleDepth`, which are set


FISSION : Fuzzy Inference System Setup by evolutION

File Help

Data Setup | FIS Setup | Training Setup | Train FIS | Apply FIS

Data Set
- [✓ Training Set]
- Testing Set
- Evaluation Set

Data Import

Load Data From File | Clear This Set | Partition Data From Training Set | Clear All Sets

Data Status

#Rows:	2935
#Columns:	4

Column Names and Statistics

Data Columns: Moisture, Time, Temperature, [✓ O2 Rate]

Column Name: O2 Rate

Change | Reset

Min Value: 0

Mean Value: 0.459

Max Value: 1.394

Std Deviation: 0.403

Input/Output Selection

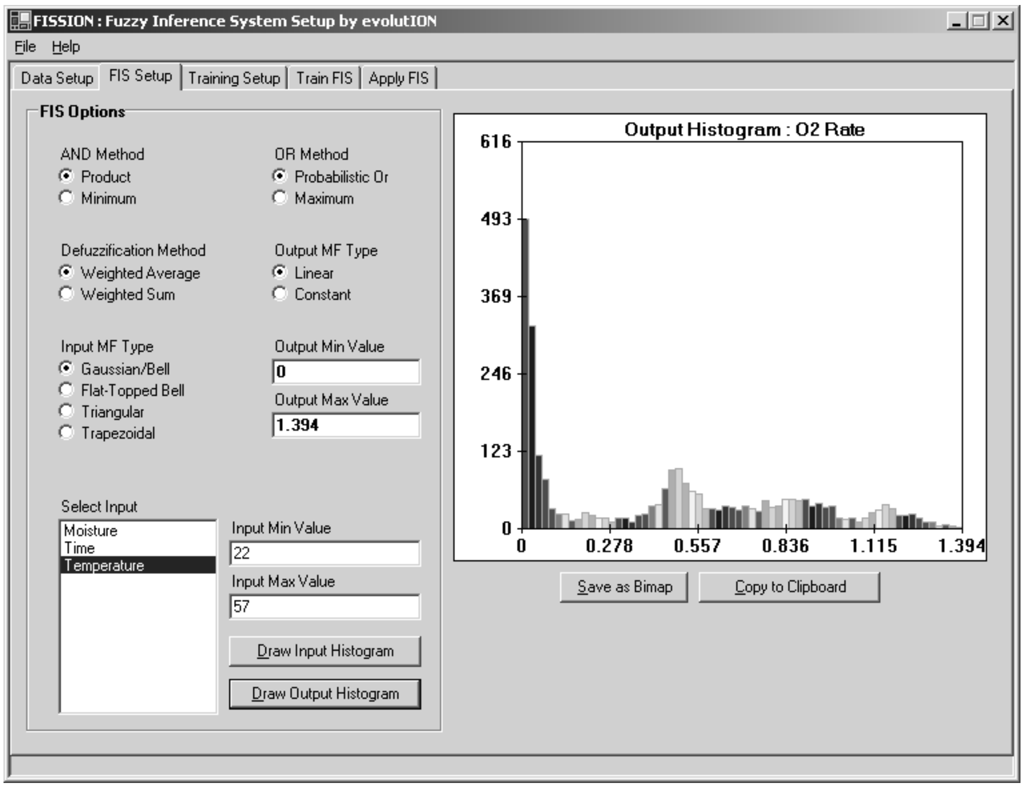
Select Inputs:
- ✓ Moisture
- ✓ Time
- ✓ Temperature
- O2 Rate

Select Outputs:
- Moisture
- Time
- Temperature
- ✓ O2 Rate

Data for Current Set

Row	Moisture	Time	Temperature	O2 Rate
0	30	1.67	22	0.032
1	30	3.33	22	0.0256
2	30	5	22	0.0221
3	30	8.33	22	0.0188
4	30	10	22	0.0163
5	30	11.67	22	0.0149
6	30	20	22	0.0106
7	30	21.67	22	0.0109
8	30	25	22	0.0122
9	30	26.67	22	0.011

Figure 5.1: FISSION data setup tab.


FISSION : Fuzzy Inference System Setup by evolutION

File Help

Data Setup | FIS Setup | Training Setup | Train FIS | Apply FIS

FIS Options

AND Method: [✓ Product] Minimum

OR Method: [✓ Probabilistic Or] Maximum

Defuzzification Method: [✓ Weighted Average] Weighted Sum

Output MF Type: [✓ Linear] Constant

Input MF Type: [✓ Gaussian/Bell] Flat-Topped Bell Triangular Trapezoidal

Output Min Value: 0

Output Max Value: 1.394

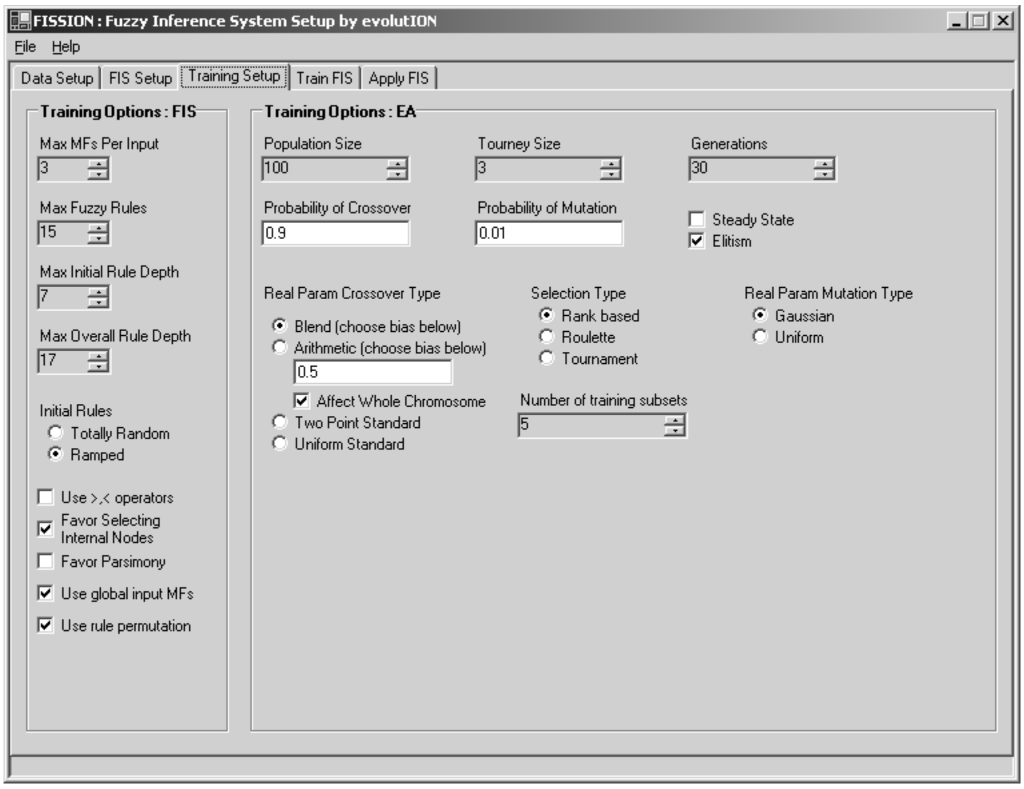
Select Input: Moisture, Time, [✓ Temperature]

Input Min Value: 22

Input Max Value: 57

Draw Input Histogram | Draw Output Histogram

Figure 5.2: FISSION FIS setup tab.


FISSION : Fuzzy Inference System Setup by evolutION

File Help

Data Setup | FIS Setup | Training Setup | Train FIS | Apply FIS

Training Options : FIS

Max MFs Per Input: 3
Max Fuzzy Rules: 15
Max Initial Rule Depth: 7
Max Overall Rule Depth: 17

Initial Rules: Totally Random [✓ Ramped]

Use >,< operators
[✓ Favor Selecting Internal Nodes]
Favor Parsimony
[✓ Use global input MFs]
[✓ Use rule permutation]

Training Options : EA

Population Size: 100
Tourney Size: 3
Generations: 30
Probability of Crossover: 0.9
Probability of Mutation: 0.01

Steady State
[✓ Elitism]

Real Param Crossover Type: [✓ Blend (choose bias below)] Arithmetic (choose bias below) 0.5 [✓ Affect Whole Chromosome] Two Point Standard Uniform Standard

Selection Type: [✓ Rank based] Roulette Tournament

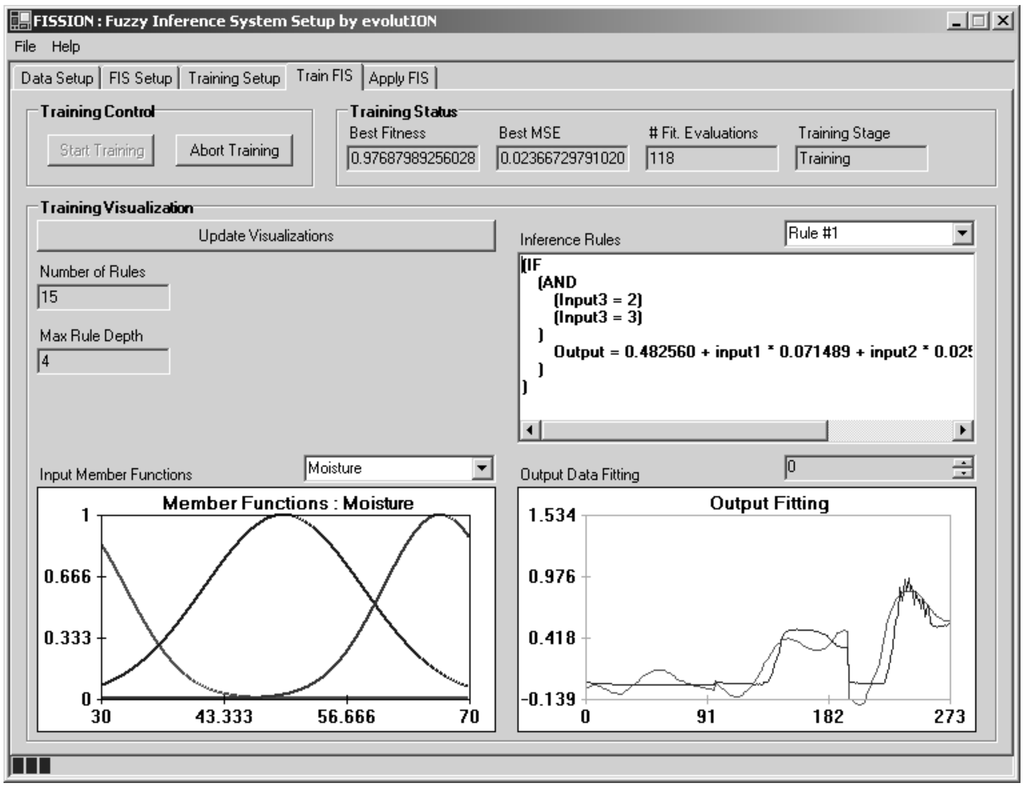
Number of training subsets: 5

Real Param Mutation Type: [✓ Gaussian] Uniform

Figure 5.3: FISSION training setup tab.

using “numeric up/down” controls to ensure validity. The following parameters are set using check boxes (boolean controls): `rampedInit`, `numSetOps`, `pSelectInternal`, `parsimonyFactor`, `globalInputMFs`, and `reorderRules`. Note that while `numSetOps`, `pSelectInternal`, and `parsimonyFactor` are *not* boolean parameters, the interface assigns numerical values to them based on whether or not their boxes are checked. For instance, `pSelectInternal` takes the value 0.9 when the box is checked, and 0.5 otherwise.

The second section deals with training options specific to the EA. These are `popSize`, `tourneySize`, `maxGens`, `pCross`, `pMut`, `steadyState`, `elitism`, `crossType`, `selType`, `mutType`, `numSubsets`, `blendParam`, and `realParamUniform`. Again, discrete parameters are set with numeric up/down controls. “Select” parameters like `crossType` are handled using radio buttons, and real parameters are set using text boxes with validation.



Training Status

Best Fitness: 0.97687989256028
Best MSE: 0.02366729791020
Fit. Evaluations: 118
Training Stage: Training

Number of Rules: 15
Max Rule Depth: 4

Inference Rules: Rule #1

```
(IF
   (AND
      (Input3 = 2)
      (Input3 = 3)
   )
      Output = 0.482560 + input1 * 0.071489 + input2 * 0.02
   )
)
```

Input Member Functions: Moisture
Output Data Fitting: 0

Figure 5.4: FISSION training tab.

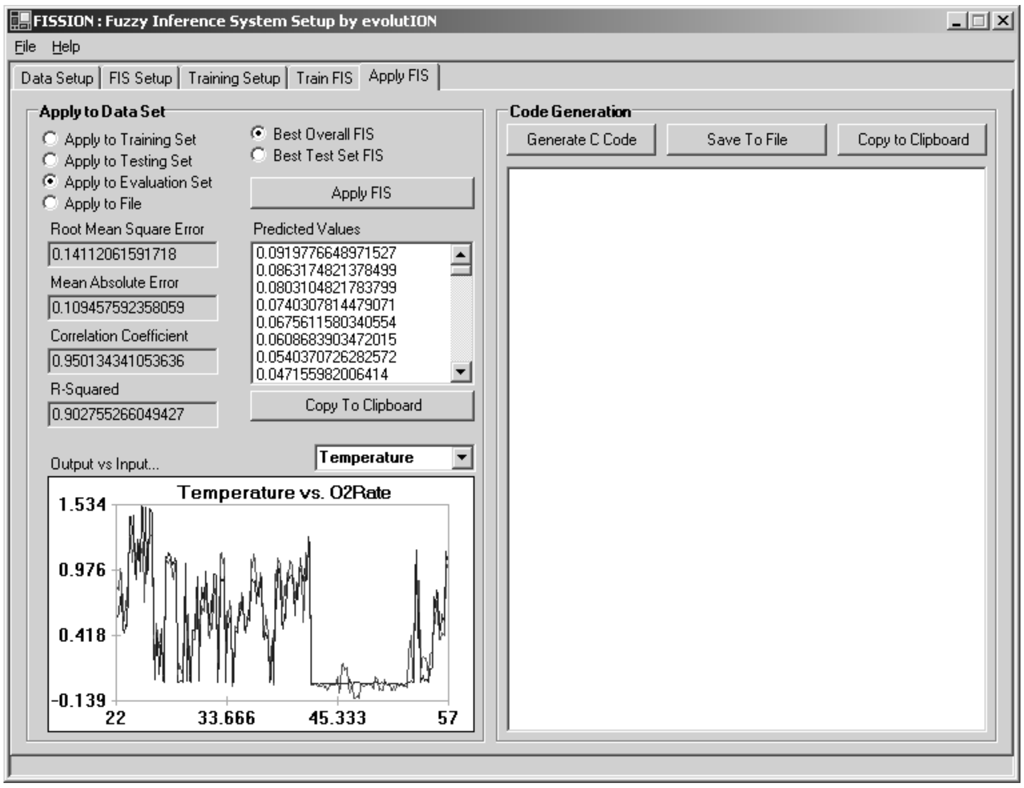
5.5.5 Train FIS Tab

The user begins the training process (figure 5.4) by clicking on the “Start Training” button. There is also an “Abort” button, which causes FISSION to consider the training complete, even if the maximum number of generations has not been reached. During training, the user is provided with text status information which is updated once per second. This information consists of EA state, best RMS Error, number of fitness evaluations, and training time. At the user’s request, visualizations of the current best FIS are displayed. There are visualizations for input membership functions, inference rules, and output data series fitting. Other text information about the FIS is also displayed. Each of these graphs can be saved as a bitmap or copied to the clipboard by means of right-click context menus.

5.5.6 Apply FIS Tab

The apply FIS tab (figure 5.5) allows the user to run a previously trained FIS on a data set. The user can either select one of the previous loaded (training, testing, or evaluation) sets or load a data set *without outputs* and have FISSION make a prediction based on that data set. Note that the data set loaded must have the same column setup as the training set used to train the model, with one fewer column. The user can also specify whether to use the best overall FIS or the best generalized (test set) FIS on the data. When the "Apply FIS" button is clicked, FISSION performs the inference. If the selected data set contains output values, the interface displays RMS error, MAE, correlation coefficient (R), and R^2 values. A list box shows the output values predicted by the model, which may be copied to the clipboard. Finally, another visualization graphs the predicted and observed output values against a selected input. This visualization can be copied to the clipboard or saved as a bitmap via a right-click context menu.

This tab also contains a section for code generation. When the user clicks "Generate C Code", FISSION outputs a C representation of the FIS model, which the user can then save to a file or copy and paste into his/her own project.


FISSION : Fuzzy Inference System Setup by evolutION

File Help

Data Setup | FIS Setup | Training Setup | Train FIS | Apply FIS

Apply to Data Set

- ( ) Apply to Training Set
- ( ) Apply to Testing Set
- (●) Apply to Evaluation Set
- ( ) Apply to File

- (●) Best Overall FIS
- ( ) Best Test Set FIS

Apply FIS

Root Mean Square Error: 0.14112061591718

Mean Absolute Error: 0.109457592358059

Correlation Coefficient: 0.950134341053636

R-Squared: 0.902755266049427

Predicted Values

```
0.0919776648971527
0.0863174821378499
0.0803104821783799
0.0740307814479071
0.0675611580340554
0.0608683903472015
0.0540370726282572
0.047155982006414
```

Copy To Clipboard

Output vs Input... Temperature

Code Generation

Figure 5.5: FISSION apply FIS tab.

Chapter 6

Results and Conclusions

6.1 Experimental Results

In order to test FISSION, two computational experiments were performed. The first is an example run of the program using an actual (observed) data set. The second experiment tests different control parameter configurations to determine good default values.

6.1.1 An Example Run

The first step in running FISSION is loading the data set to be used for training. In this case, the data set measures biosolids composting (Liang et al. 2003a,b). In Liang's experiment, biosolids were taken from a wastewater treatment plant and subjected to a two factor analysis involving moisture and temperature. Observations of O_2 uptake rate were made by a computer-controlled respirometer in order to determine microbial activity.

There are three input (predictor) variables in the data set: moisture (% humidity), time (hours), and temperature (°C). Although four dependent variables are included, the only one of interest here is O_2 uptake rate, measured in mg g^{-1} h^{-1}. For a more complete description, see Liang et al. (2003a), the source of the data set.

Initially, the data consisted of a model development set containing 8,760 patterns, and a model evaluation set containing 1,460 patterns. These patterns consist of 5 moisture regimes (30%, 40%, 50%, 60%, and 70%), and 7 temperature regimes (22°, 29°, 34°, 36°, 43°, 50°, and 57°). The 34° regime is included only in the evaluation set. Each combination of regimes was tested on 1.67 hour intervals over a period of 240 hours, resulting in a total of 146 observations per experiment.

The model development set was divided into two replicates, representing two trials in the original experiment. In the FISSION test, only one of the two replicates was used (to increase training

Table 6.1: Control Parameter Settings

FIS Control Parameters		*EA Control Parameters*	
name	*value*	*name*	*value*
`globalInputMFs`	true	`blendParam`	0.5
`linearOutput`	true	`crossType`	blend
`maxInitialRuleDepth`	7	`elitism`	true
`maxInputMFs`	3	`maxGens`	30
`maxOverallRuleDepth`	17	`mutType`	Gaussian
`maxRules`	15	`numSubsets`	5
`mfType`	Gaussian	`pCross`	0.9
`numInputs`	3	`pMut`	0.01
`numOutputs`	1	`popSize`	300
`numSetOps`	1	`realParamUniform`	true
`parsimonyFactor`	0	`selType`	roulette
`probor`	true	`steadyState`	false
`prod`	true	`tourneySize`	3
`pSelectInternal`	0.9		
`rampedInit`	true		
`reorderTree`	true		
`wtaver`	true		

speed), meaning that 4,380 patterns were available for model development. This reduced set was split randomly (uniformly) into a training set and a testing set, the former taking 67% (2,935) of the training patterns and the latter left with 33% (1,445). The evaluation set was left unchanged (containing both replicates), and each of the three data sets was stored in a separate file.

FISSION was configured as described in section 5.5 with the three data sets and the control parameter settings listed in table 6.1. After the training finished, the best overall FIS was applied to the evaluation data set. This particular validation set is an especially good judge of model quality because it consists of patterns at a temperature value (34°) not seen in the training data. FISSION reported the statistics shown in table 6.2.

These values represent an improvement over the author's best results using ANFIS. In ANFIS, the best RMS error achieved with 3 membership functions per input was 0.1633, compared to 0.1411 in FISSION. In addition, FISSION used fewer rules: 14 as compared to 27. Liang et al. (2003b) does not give RMS error values; however it does provide MAE and R^2 figures. They achieved a MAE of

Table 6.2: Experiment results

statistic	*value*
RMS Error	0.1411
MAE	0.1095
R^2	0.9028
# Rules	14
Max. Rule Depth	5
Training Time	911s (15:11)

0.11 and and R^2 of 0.852 using a two output Ward neural network with 84 hidden nodes, compared to 0.1095 MAE and 0.9028 R^2 for FISSION. While the MAE values are virtually the same (and the difference may be due to rounding,) FISSION appears to have an advantage in correlation (as described by R^2).

The input membership functions for the best FIS are shown in figure 6.1. Note that the input spaces for moisture and time have "gaps" in them; i.e, there are points within the input domain which have low membership in all fuzzy sets. This state of affairs would be a serious problem for systems such as ANFIS which rely soley on the intersection of fuzzy sets. However, it is not an issue for FISSION, which incorporates the *complement* of fuzzy sets as well. If a point has low membership in every fuzzy set defined for an input, then it has *high* membership in every complement.

The inference rules contained in the best FIS after training are represented in Appendix A. Notice that only 14 rules were generated out of a maximum of 15. FISSION can dynamically alter the number of rules during training through crossover and mutation. The number of membership functions can *effectively* change, in that a function may not be used by any rule. In that case, the membership functions has become rather like a "vestigial organ": still present, but not used. In the top-right corner of figure 6.1, there are two membership functions which are very close in position and shape. This could be an indication that only two membership functions are really required for that input (time).

Inspection of rules 13 and 14 show what appears to be a failure to simplify: in 13, it seems that `AND(X X)` should be simplified to `X`, and in 14, `NOT(NOT(NOT(NOT(X))))` is clearly the same

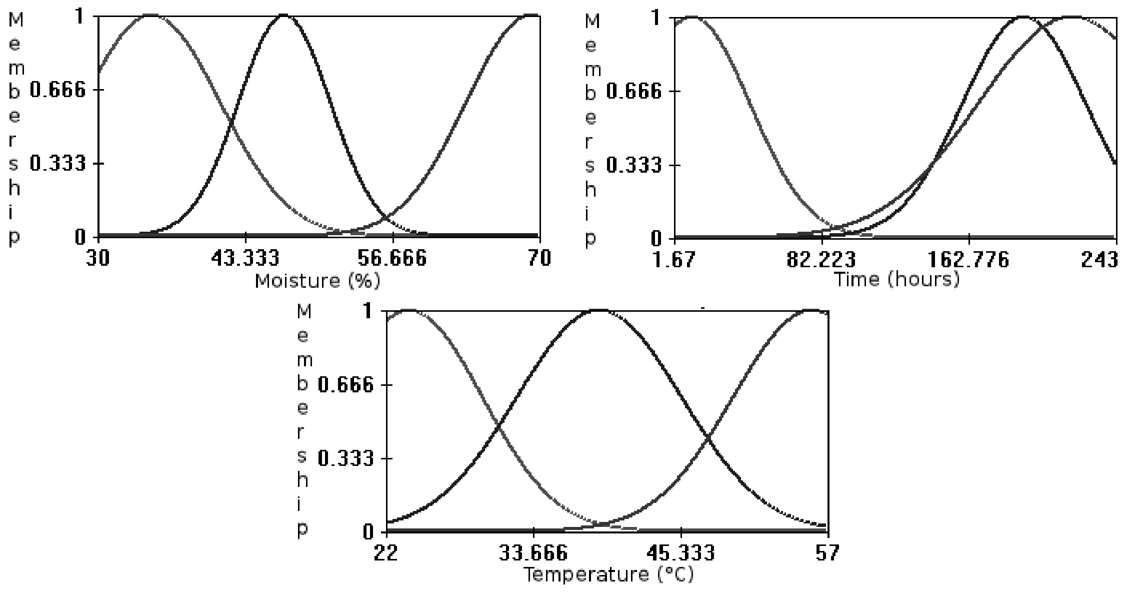


Figure 6.1: Membership functions, clockwise from top left: moisture, time, and temperature.

as `X`. However, it should be noted that in *fuzzy* logic, `AND(X X)` is actually not the same as `X` if the product definition of intersection is being used. If `X` to degree ϕ, then `AND(X X)` to degree ϕ^2. It is for this reason that simplification features have not been included in FISSION.

Figures 6.2 through 6.6 show the trained FIS applied to the five moisture regimes on the validation data. These visualizations demonstrate how the model predictions fit the observed data, with O_2 rate graphed against time. Each observed value is graphed as the mean of the corresponding values from the two replicates. Figure 6.7 shows the progression of best population fitness value and best overall RMS error during training. One might wonder why there are downward spikes in the fitness graph, even though elitism is being used. They are caused by the rotation of the training subsets used to calculate the output membership functions. Elitism still preserves the best individuals from generation to generation, and the best individual is always remembered separate from the population.

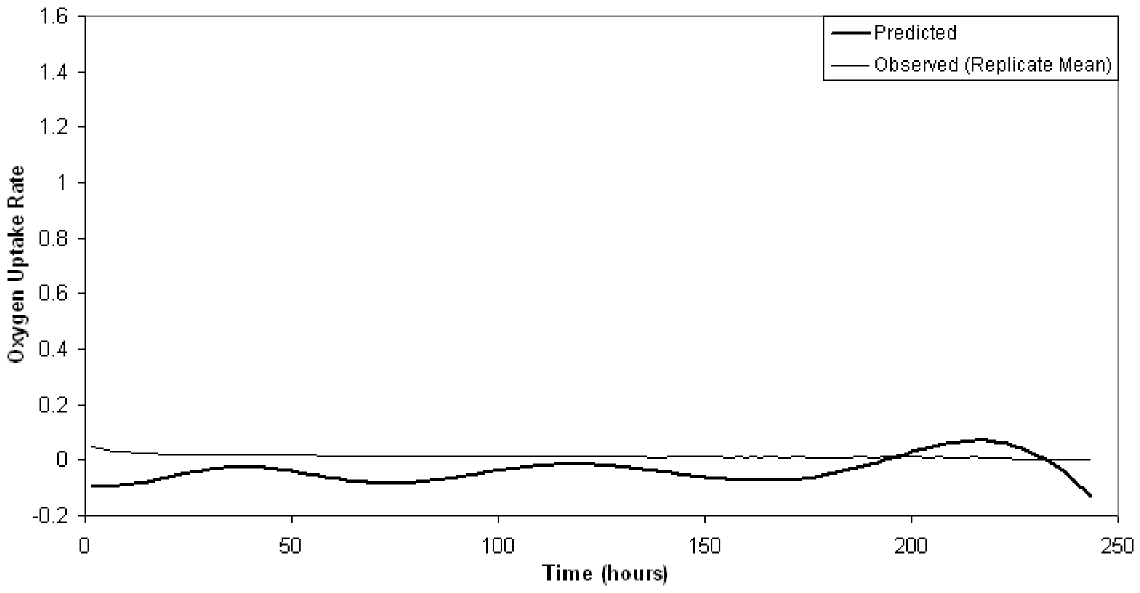


Figure 6.2: Best FIS applied to validation set, moisture 30%

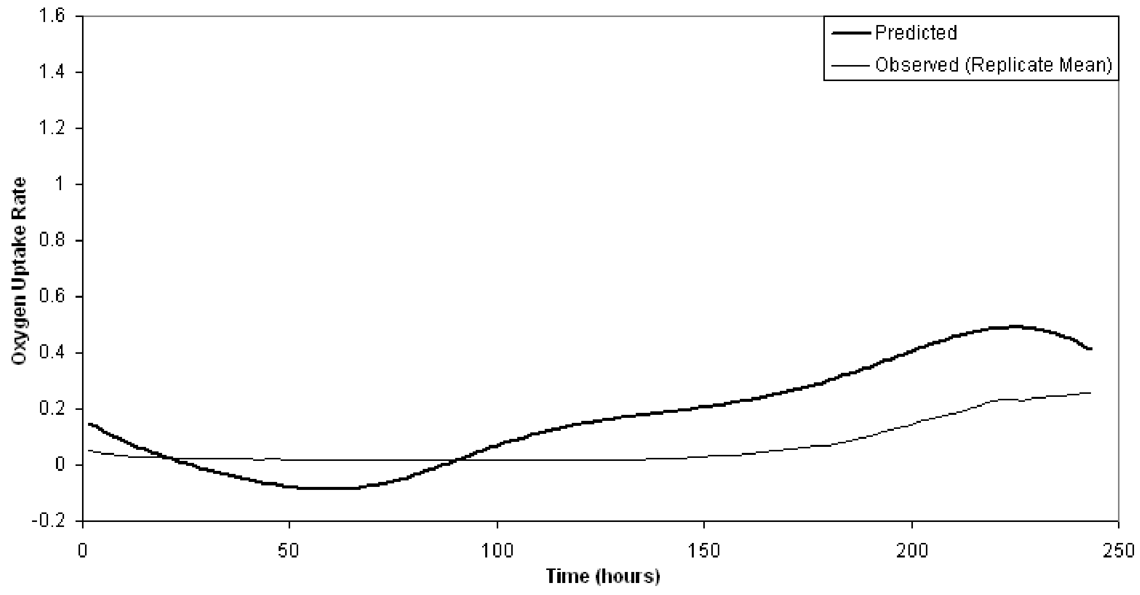


Figure 6.3: Best FIS applied to validation set, moisture 40%

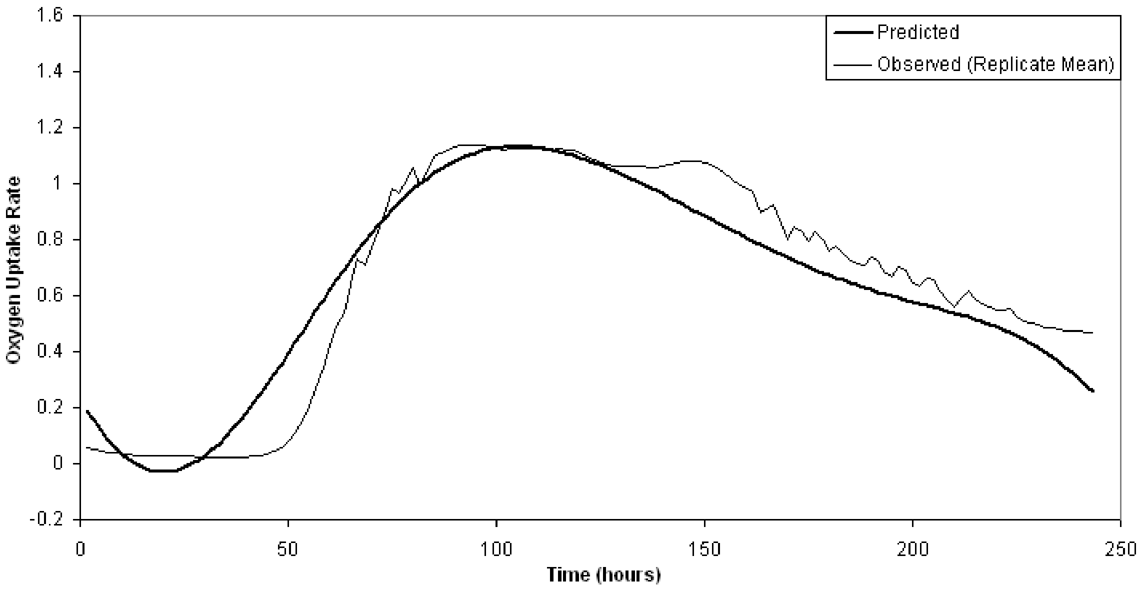


Figure 6.4: Best FIS applied to validation set, moisture 50%

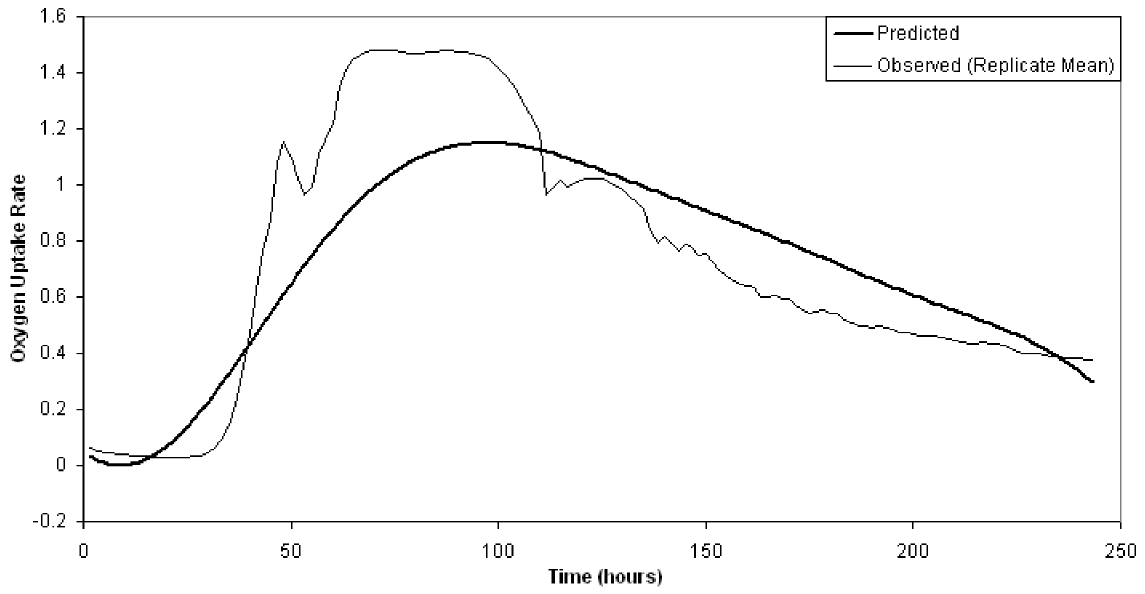


Figure 6.5: Best FIS applied to validation set, moisture 60%

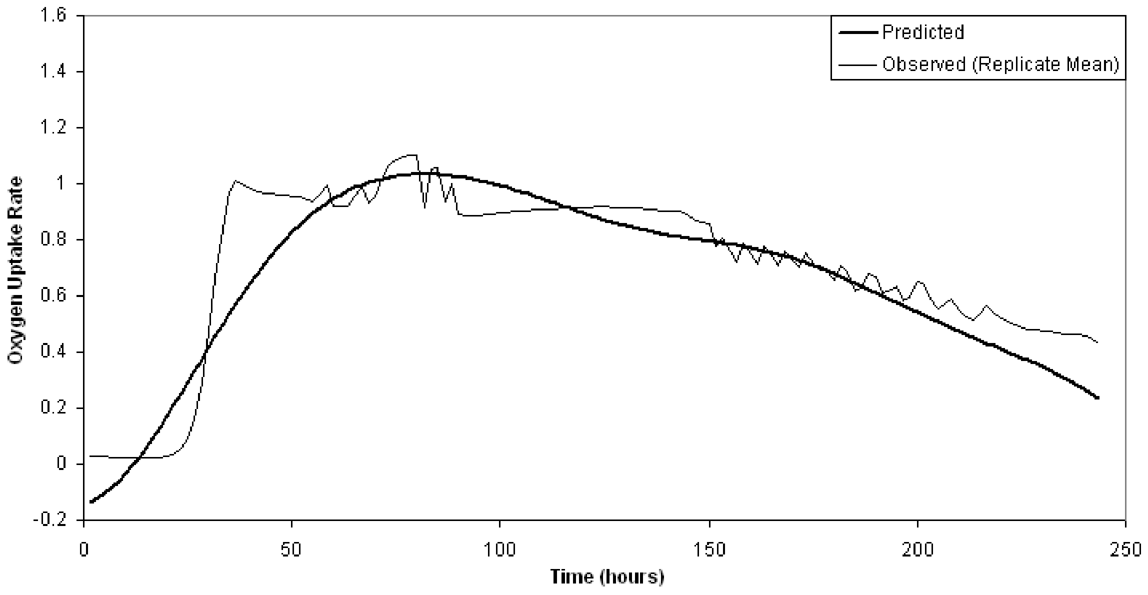


Figure 6.6: Best FIS applied to validation set, moisture 70%

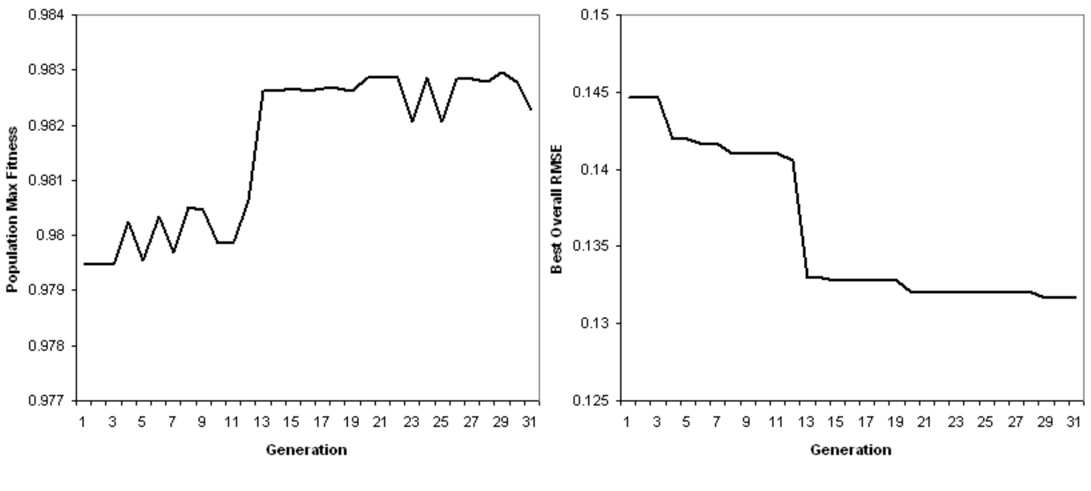


Figure 6.7: Population maximum fitness (left) and overall best RMS error (right).

6.1.2 An Investigation of Control Parameter Settings

FISSION uses 30 control parameters to determine the operation of its training engine. Reasonable default values for these parameters have been established during the development process and based on published literature. However, it is interesting to know which parameters are the most "mission-critical," that is, which have the greatest effect on performance.

An exhaustive study of all parameter values and their interactions would be prohibitively time consuming. In addition, the results of such an experiment might apply only to the particular type of data set upon which it was performed. Hence, this paper will instead present a series of controlled experiments on the biocomposting data sets used in the previous section, relating to a select subset of parameters. In choosing these parameters, those settings which affect the *size* of the model were omitted in favor of those which determine *how* the training proceeds. In general, larger, more complex models (those involving more rules and membership functions) will have greater fitting power. Since many tests were to be run in this experiment, size parameters were chosen to limit the complexity and run-time of the training process. The control for this study consists of the set of parameters listed in table 6.3. Ten trials were run with the control set.

Each experiment involved changing the value of a single parameter and running ten trials using the resulting set. The best FIS from each trial was applied to the *training* set (as described in section 6.1.1), resulting in error and R^2 values. The mean error values from each experiment are compared to the mean error value of the control using a two tailed t-test to determine whether any difference is statistically significant. SAS (SAS Institute 2003) was used to perform the tests. The histogram in figure 6.8 shows the distribution of the error values from all trials. The experiments and their results are listed in table 6.4.

Nine of the experiments yielded apparently significant ($p < 0.05$) results. It should be noted that at least one of the results could have been falsely declared significant with probability up to $1 - (0.95)^9 = 0.37$ (in fact, the probability is less, since all 9 values are less than 0.05). For this reason, the Bonferroni "corrected" values are included in the table as well. The Bonferroni correction is guaranteed to to make a conservative estimate by multipying the p-value by the total number of comparisons made (in this case, 22). This guarantee is based on the theorem that $\mathrm{P}\{\text{a or b or c or ...}\} \leq \mathrm{P}\{\text{a}\} + \mathrm{P}\{\text{b}\} + \mathrm{P}\{\text{c}\} + \cdots$, where P denotes probability (SAS Institute 2003).

Table 6.3: Control Parameter Settings

FIS Control Parameters		*EA Control Parameters*	
name	*value*	*name*	*value*
`globalInputMFs`	true	`blendParam`	0.5
`linearOutput`	true	`crossType`	blend
`maxInitialRuleDepth`	7	`elitism`	true
`maxInputMFs`	3	`maxGens`	30
`maxOverallRuleDepth`	17	`mutType`	Gaussian
`maxRules`	10	`numSubsets`	5
`mfType`	Gaussian	`pCross`	0.9
`numInputs`	3	`pMut`	0.05
`numOutputs`	1	`popSize`	100
`numSetOps`	1	`realParamUniform`	true
`parsimonyFactor`	0	`selType`	roulette
`probor`	true	`steadyState`	false
`prod`	true	`tourneySize`	3
`pSelectInternal`	0.9		
`rampedInit`	true		
`reorderTree`	true		
`wtaver`	true		

Table 6.4: Experimental Results

Experiment	Alteration	Mean RMSE	p-Value	Bonferroni
0	Control	0.1449	N/A	N/A
1	`steadyState`=true	0.1496	0.0028	0.0618
2	`rampedInit`=false	0.1443	0.7083	1
3	`reorderTree`=false	0.147	0.1794	1
4	`prod`=`probor`=false	0.1466	0.2833	1
5	`linearOutput`=false	0.1847	<.0001	<.0001
6	`elitism`=false	0.1511	<.0001	0.0018
7	`wtaver`=false	0.1473	0.1285	1
8	`realParamUniform`=false	0.1472	0.1386	1
9	`selType`=tournament	0.1402	0.0026	0.0566
10	`selType`=rank	0.1405	0.0047	0.1023
11	`crossType`=two point	0.148	0.0478	1
12	`crossType`=uniform	0.1472	0.1459	1
13	`crossType`=arithmetic	0.1476	0.0864	1
14	`mutType`=uniform	0.1456	0.6602	1
15	`parsimonyFactor`=0.1	0.146	0.4984	1
16	`pSelectInternal`=0.5	0.1464	0.3513	1
17	`numSetOps`=3	0.1478	0.0674	1
18	`mfType`=gbell	0.1481	0.0395	0.8687
19	`mfType`=tri	0.1505	0.0004	0.0085
20	`mfType`=trap	0.1499	0.0014	0.0318
21	`pCross`=0.6	0.146	0.4832	1
22	`pMut`=0.15	0.1462	0.4009	1

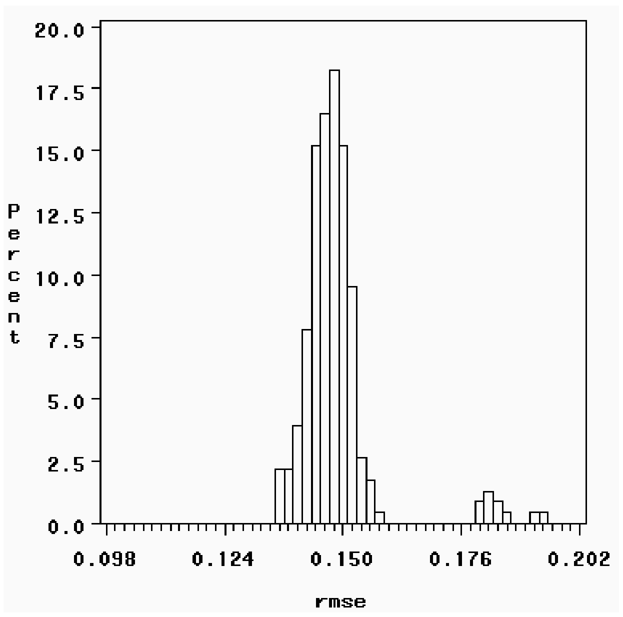


Figure 6.8: Histogram of RMS error values from *all* experiments.

However, this is not a major issue here since the p-values are simply being used to determine which parameters are most interesting for purposes of discussion.

The first significant result is from experiment 1 ($p = 0.0028$), involving the `steadyState` parameter. The mean RMSE using steady state was 0.1496 compared with 0.1449 using generational (in the control). The superior performance of the generational scheme can probably be attributed to its slower convergence, particularly since the population size is only 100. Steady state tends to focus on a single area of the search space much more quickly and thus risks converging on a suboptimal local maximum.

One might complain that the error values are very close together, and hence there is not an interesting performance difference between steady state and generational approaches. However, the small gap is most likely due to the Kalman optimization process, which can mask the effect of the EA. Since there is only $\frac{1}{5}$ of a percent chance that the difference between error values is due to randomness, it can be assumed that the EA is really performing better with the generational scheme, despite the small difference in error values. With another data set (or even in a different experiment with the same set) the discrepancy could well manifest itself to a greater extent.

Experiment 5 ($p < 0.0001$) clearly indicates that linear output functions perform better than constant ones, with RMSEs of 0.1449 and 0.1847 respectively. This should come as no surprise, since the linear method allows the final output of the FIS to be a direct function of the inputs as well as the rule firing strengths, instead of just the latter. Of course, the Kalman procedure runs much faster in the constant method, since only one parameter needs to be adjusted for each rule. The trials using the constant method took, on average, 72 seconds as compared to 175 seconds for the linear method.

The next significant result is found in experiment 6 ($p < 0.0001$). Trials using elitism had a mean RMSE of 0.1449, compared to trials without elitism which averaged 0.1511. Elitism typically boosts the performance of any EA, but the training subset scheme in FISSION makes this feature particularly important. It ensures that the individual Q which performed the best in generation i will survive in generation $i + 1$, regardless of whether Q performs well under the training subset used to create the $(i + 1)^{st}$ generation. Hence, one poorly chosen training subset is less likely to cause the loss of good genetic information.

Experiment 9 ($p = 0.0026$) shows that tournament selection is superior to roulette selection in this application. This experiment had an RMSE of 0.1402 as compared to 0.1449 for the control. Tournament selection (with a reasonably small tourney size) increases the probability that mid-fitness individuals will be selected, relative to high-fitness individuals. That feature helps to preserve diversity and prevent premature convergence.

Similarly, rank-based selection appears to work better than roulette selection (Experiment 10, $p = .0047$). Trials using rank-based selection averaged 0.1405 in RMSE, as opposed to 0.1449 using the roulette method. The advantages of rank-based selection are similar to those of tournament selection, with the caveat that low-fitness individuals are also selected with higher probability (in tournament selection, the lowest-fitness individual will never be selected, unless it is the only individual in the tourney).

All three alternate crossover types performed worse than blend crossover on average, with RMSEs of 0.1480, 0.1472 and 0.1476 for two-point, uniform, and arithmetic crossovers respectively.

Blend crossover was used in the control, and resulted in an RMSE of 0.1449. Only the two-point result from experiment 11 ($p = 0.0478$) is considered significant, although the other two were somewhat close with p-values of 0.0864 and 0.1459. The superiority of blend crossover can be explained by noting that two-point and uniform crossovers are not designed for real-valued chromosomes, and arithmetic crossover results in a loss of diversity by favoring the center of the input space over the periphery.

Finally, experiments 18 ($p = 0.0395$), 19 ($p = 0.0004$), and 20 ($p = .0014$) showed that generalized bell, triangular, and trapeziodal membership function types are less well suited to the biocomposting data than the Gaussian type. The generaliazed bell trials had an average RMSE of 0.1481, the triangular trials had 0.1505, and the trapezoidal method resulted in 0.1499. Gaussian membership functions were used in the control, resulting in an RMSE of 0.1449. These figures agree with the author's results using ANFIS, where the `gauss` membership function type performed the best.

While not strictly ($p < 0.05$) significant, the p-values for experiments 3 ($p = 0.1794$), 7 ($p = 0.1285$), 8 ($p = 0.1386$), and 17 ($p = 0.0674$) are low enough to merit discussion. These experiments deal with the parameters `reorderTree`, `wtaver`, `realParamUniform`, and `numSetOps` respectively.

Reordering the rules by firing strength does appear to help, resulting in a mean RMSE of 0.1449 as opposed 0.1470 without reordering. This policy helps prevent the tree crossover operator from disrupting effective groups of rules.

Weighted average seems to be a better aggregation method than weighted sum, with RMSE values of 0.1449 and 0.1473. This result agrees with the author's results using ANFIS, and may be explained by noting that weighted average can minimize the effect of a single unhelpful rule (since the effect of each individual rule is minimized by division).

Apparently, the feature of applying real parameter crossovers to only *part* of a chromosome was unhelpful. With this feature turned off (`realParamUniform` true,) the control mean RMSE was 0.1449, compared to 0.1472 with the feature turned on (`realParamUniform` false).

Including the $<$ and $>$ operators also seemed to worsen performance, resulting in a mean RMSE of 0.1478 as opposed to 0.1449 for the control. This phenomenon is not easily explained, and merits further research. However, it could be speculated that FISSION *overuses* these operators, when they are only really useful at the boundaries of the input domain.

Setting the *parsimonyFactor* control parameter to 0.1 in experiment 15 led to an increase in mean RMSE (0.1460 as opposed to 0.1449 in the control). However, the average maximum rule depth dropped from 7 to 6.4. Setting the parsimony factor to a larger fraction would likely result in a greater decrease in complexity, albeit at a greater cost in accuracy.

6.2 Future Work

FISSION implements fuzzy inference system calibration using a mixture of evolutionary approaches. Further, the methods described in this paper have been effectively applied to a real-world data set. However, this project exposes a number of areas which are ripe for further investigation. Two such areas are described here.

The particular Pittsburgh style learning algorithm used by FISSION is inherently flawed. Good performance by an individual requires near-optimal interaction between its membership functions and rule set. However, since different crossover operators are used for the two chromosomes, the two evolve separately. Hence, the overall crossover operation is by nature disruptive, since it takes a chromosome pair that works (well) and changes one of the two, but not the other. An interesting alternative would be *coevoluton*. In this scheme, there would be two entirely separate populations, one consisting of membership functions, and the other of inference rules. Individuals in the two populations would bear a *symbiotic* relationship to one another. This approach would mitigate the disruptive effects currently present in FISSION. However, it would also introduce new challenges, such as designing a method for pairing membership functions and assigning fitness values across the two populations.

Recall that one of the goals of FISSION was to produce human-interpretable rule sets. Some progress has been made in this direction by limiting the number of rules and making them more descriptive. However, Takagi-Sugeno style fuzzy inference systems naturally suffer in this respect by eliminating the linguistic output variables and replacing them with mathematical formulas. It also seems desirable to eliminate the costly and overfitting-prone Kalman estimation procedure for output functions. The obvious solution would be to switch to a Mamdani style controller. These models would be easier to interpret, and could be trained entirely by the EA. Challenges to overcome would include reduced accuracy and more complex FIS evaluation.

Bibliography

Alba, E., Cotta, C., & Troya, J. (1999). Evolutionary design of fuzzy logic controllers using strongly-typed GP. *Mathware & Soft Computing*, 6(1), 109-124.

Bacardit, J., Goldberg, D., Butz, M., Llora, X., & Garrell J. (2004). *IlliGAL Report No. 2004022: Speeding-up Pittsburgh Learning Classifier Systems: Modeling Time and Accuracy* Urbana, IL: Illinois Genetic Algorithms Laboratory.

Carse, T., Fogarty, T., & Munro, A. (1996). Evolving fuzzy rule based controllers using genetic algorithms. *Fuzzy Sets and Systems*, 80, 273-293.

Cordon, O., Gomide, F., Herrera, F., Hoffmann, F., & Magdalena, L. (2004). Ten years of genetic fuzzy systems: current framework and new trends. *Fuzzy Sets and Systems*, 141, 5-31.

Cooper, M., & Vidal, J. (1994). Genetic design of fuzzy controllers: The cart and jointed pole problem. *Proceedings of the Third IEEE International Conference on Fuzzy Systems*, 1332-1337.

De Jong, K. (1975). *An analysis of the behavior of a class of genetic adaptive systems.* Ann Arbor, MI: Doctoral dissertation, University of Michigan.

Engelbrecht, A. (2003). *Computational Intelligence: An Introduction.* West Sussex, England: John Wiley & Sons Ltd.

Eshelman, L., & Schaffer, J. (1993). Real-coded genetic algorithms and interval schemata. In D. Whitley (Ed)., *Foundations of Genetic Algorithms II*, 187-202. San Mateo, CA: Morgan Kaufmann Publishers.

Goldberg, D. (1989a) *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading, MA: Addison-Wesley.

Goldberg, D., Korb, B., & Deb, K. (1989b). Messy genetic algorithms: motivation, analysis, and first results. *Complex Systems*, 3, 493-530.

Grefenstette, J. (1987). Multilevel credit assignment in a genetic learning system. *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, 202-209.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems.* Ann Arbor, MI: University of Michigan Press.

Holland, J., & Reitman, J. (1978). Cognitive systems based on adaptive algorithms. In D. Waterman, & F. Hayes-Roth, (Eds)., *Pattern-directed Inference Systems* New York, NY: Academic Press.

Homaifar, A., & McCormick, E. (1995). Simultaneous design of membership functions and rule sets for fuzzy controllers using genetic algorithms. *IEEE Transactions on Fuzzy Systems*, 3(2), 129-139.

Janikow, C., & Michalewicz, Z. (1991). An experimental comparison of binary and floating point representations in genetic algorithms. *Proceedings of the Fourth International Conference on Genetic Algorithms*, 31-36.

Jang, J. (1993a). ANFIS: Adaptive network-based fuzzy inference systems. *IEEE Trans. on Systems, Man and Cybernetics*, 23(3), 665-685.

Jang, J. (1993b). *ANFIS: Adaptive Network-based Fuzzy Inference System (source code)* Retrieved October, 2005, from http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/fuzzy/systems/anfis/.

Kalman, R. (1960). A new approach to linear filtering and prediction problems. *Transactions of the ASME—Journal of Basic Engineering*, March 1960, 35-45.

Karr, C. (1991). Design of an adaptive fuzzy logic controller using a genetic algorithm. *Proceedings of the Fourth Internation Conference on Genetic Algorithms*, 450-457.

Koza, J. (2002). *Genetic Programming.* Cambridge, MA: The MIT Press.

Lee, M., & Takagi, H. (1993). Integrating design stages of fuzzy systems using genetic algorithms. *Proceedings of the Second IEEE International Conference on Fuzzy Systems*, 612-617.

Liang, C., Das, K., McClendon, R. (2003a). The influence of temperature and moisture content regimes on the aerobic micorbial acitivity of biosolids composting blend. *Bioresource Technology*, 86(2), 131-137.

Liang, C., Das, K., & McClendon, R. (2003b). Prediction of microbial activity during biosolids composting using artificial neural networks. *Transactions of the American Society of Agricultural Engineers*, 46(6), 1713-1719.

Liska, J., & Melsheimer, S. (1994). Complete design of fuzzy logic systems using genetic algorithms. *Proceedings of the Third IEEE International Conference on Fuzzy Systems*, 1377-1382.

Mamdani, E., & Assilian, S. (1975). An experiment in linguistic synthesis with a fuzzy controller. *International Journal of Man-Machine Studies*, 7, 1-13.

MathWorks (2005). *Fuzzy Logic Toolbox Documentation* Retrieved October, 2005, from http://www.mathworks.com/access/helpdesk/help/toolbox/fuzzy/index.html.

Microsoft (2005). *MSDN Online Documentation* Retrieved October, 2005, from http://msdn.microsoft.com.

Pham, D., & Karaboga, D. (1991). Optimum design of fuzzy logic controllers using genetic algorithms. *Journal of Systems Engineering*, 1, 114-118.

SAS Institute (2003). *SAS for Windows* Cary, NC: SAS Institute, Inc.

Smith, S. (1980). *A Learning System Based on Genetic Adaptive Algorithms.* Pittsburgh, PA: Doctoral Dissertation, University of Pittsburgh.

Sugeno, M. (1985). *Industrial Applications of Fuzzy Control.* New York, NY: Elsevier Science Inc. Cambridge, MA: The MIT Press.

Takagi, T., & Sugeno, M. (1985). Fuzzy identification of systems and its application to modeling and control. *IEEE Transactions on Systems, Man, and Cybernetics*, 15(1), 116-132.

Thrift, P. (1991). Fuzzy logic synthesis with genetic algorithms. *Proceedings of the Fourth International Conference on Genetic Algorithms*, 509-513.

Venturini, G. (1993). A supervised inductive algorithm with genetic search for learning attribute based concepts. *Proceedings of the European Conference on Machine Learning (Vienna)*, 280-296.

Welch, G., & Bishop, G. (2004). *Technical Report 95-041: An Introduction to the Kalman Filter.* Chapel Hill, NC: University of North Carolina.

Ying, H., Ding, Y., & Shao, S. (1999). Comparision of necessary conditions for typical Takagi-Sugeno and Mamdani fuzzy systems as universal approximators. *IEEE Transactions on Systems, Man, and Cybernetics Part A*, 29(5), 508-514.

Zadeh, L. (1965). Fuzzy sets. *Information and Control*, 8, 338-353.

Zadeh, L. (1975). The concept of a linguistic variable and its application to approximate reasoning, parts 1 and 2. *Information Sciences*, 8(3), 199-249.

Appendix A

Example Inference Rules

```
1.  (IF
       (Input2 = 1)
       Output = 7.337117 + input1 * -0.099313 + input2 * -0.020941
          + input3 * -0.107748
    )

2.  (IF
       (AND
          (OR
             (Input2 = 3)
             (NOT
                (OR
                   (Input2 = 3)
                   (Input2 = 2)
                )
             )
          )
          (NOT
             (Input3 = 2)
          )
       )
       Output = 1.135712 + input1 * 0.099991 + input2 * -0.018968
          + input3 * -0.089579
    )

3.  (IF
       (AND
          (NOT
             (NOT
                (AND
                   (Input2 = 1)
                   (Input1 = 1)
                )
             )
          )
          (NOT
             (OR
                (AND
                   (Input2 = 2)
                   (Input1 = 2)
                )
                (NOT
                   (Input1 = 3)
                )
             )
          )
       )
       Output = -4.152116 + input1 * -205.785390 + input2 * 29.755553
          + input3 * -33.435188
    )

   4. (IF
       (AND
          (AND
             (AND
                (AND
                   (Input3 = 2)
                   (Input1 = 2)
                )
                (OR
                   (Input3 = 3)
                   (Input1 = 1)
                )
             )
             (Input3 = 1)
          )
          (OR
             (Input3 = 1)
             (AND
                (NOT
                   (Input1 = 3)
                )
                (Input1 = 2)
             )
          )
       )
       Output = -136.314802 + input1 * 8.642018 + input2 * 0.527937
          + input3 * -7.864042
    )

   5. (IF
       (OR
          (AND
             (NOT
                (Input3 = 1)
             )
             (OR
                (AND
                   (Input2 = 1)
                   (Input1 = 3)
                )
                (NOT
                   (Input3 = 1)
                )
             )
          )
```

```
          (OR
             (AND
                (NOT
                   (Input1 = 2)
                )
                (OR
                   (Input1 = 2)
                   (Input2 = 3)
                )
             )
             (Input1 = 3)
          )
       )
       Output = -6.540776 + input1 * 0.057056 + input2 * 0.010408
          + input3 * 0.066610
    )

6.  (IF
       (NOT
          (NOT
             (AND
                (NOT
                   (Input2 = 1)
                )
                (OR
                   (Input1 = 3)
                   (Input3 = 3)
                )
             )
          )
       )
          Output = -5.317154 + input1 * 0.100194 + input2 * 0.002217
             + input3 * -0.041050
    )

7.  (IF
       (Input3 = 3)
       Output = -15.814353 + input1 * 0.082474 + input2 * -0.012289
          + input3 * 0.298081
    )

8.  (IF
       (OR
          (AND
             (NOT
                (Input1 = 3)
             )
             (OR
                (AND
                   (Input3 = 2)
                   (Input2 = 3)
                )
                (NOT
                   (Input1 = 2)
                )
             )
          )
          (AND
             (Input1 = 3)
             (OR
                (NOT
                   (Input3 = 3)
                )
                (AND
                   (Input1 = 3)
                   (Input1 = 1)
                )
             )
          )
       )
       Output = -11.718658 + input1 * 0.093831 + input2 * -0.006658
          + input3 * 0.187943
    )

9.  (IF
       (OR
          (Input3 = 2)
          (Input3 = 2)
       )
       Output = 2.127390 + input1 * 0.073895 + input2 * -0.030709
          + input3 * 0.011463
    )

10. (IF
       (AND
          (NOT
             (Input3 = 2)
          )
          (Input1 = 1)
       )
       Output = 9.125582 + input1 * -0.059277 + input2 * 0.001141
          + input3 * -0.124425
    )

11. (IF
       (AND
          (AND
             (Input2 = 1)
             (OR
                (NOT
                   (Input1 = 3)
                )
                (Input2 = 1)
             )
          )
          (Input1 = 3)
       )
       Output = -62.000648 + input1 * 0.800438 + input2 * 0.147539
```

```
         + input3 * 0.006198
     )

12.  (IF
        (OR
           (NOT
              (AND
                 (OR
                    (Input2 = 2)
                    (Input1 = 3)
                 )
                 (OR
                    (Input2 = 1)
                    (Input1 = 2)
                 )
              )
           )
           (NOT
              (NOT
                 (NOT
                    (Input3 = 3)
                 )
              )
           )
        )
        Output = 20.447472 + input1 * -0.239381 + input2 * 0.001690
           + input3 * -0.143466
     )

13.  (IF
        (AND
           (Input1 = 2)
           (Input1 = 2)
        )
        Output = 0.640457 + input1 * -0.296678 + input2 * 0.017733
           + input3 * 0.256518
     )

14.  (IF
        (NOT
           (NOT
              (NOT
                 (NOT
                    (Input1 = 1)
                 )
              )
           )
        )
        Output = -22.734805 + input1 * 0.009520 + input2 * 0.147617
           + input3 * 0.335430
     )
```

Appendix B

Program Listing

```
////////////////////////////////////////////////////////////////////////////
// Eric A. Morris
// Univ. of Georgia AI Center
// FISSION Project
//
// FISSION_Structures.h
// Declaration for the FIS/EA configuration structures
// and various constants.

#pragma once
#include <iostream>

#define E 2.71828183
#define PARAM_SIZE 4
  using namespace std;
static const int CROSS_2PT = 0;
static const int CROSS_UNIFORM = 1;
static const int CROSS_BLEND = 2;
static const int CROSS_ARITHMETIC = 3;
static const int MUT_GAUSS = 0;
static const int MUT_UNIFORM = 1;
static const int SEL_RANK = 0;
static const int SEL_TOURNAMENT = 1;
static const int SEL_ROULETTE = 2;
static const int SEL_UNIV = 3;
static const int MF_GAUSS = 0;
static const int MF_GBELL = 1;
static const int MF_TRAP = 2;
static const int MF_TRI = 3;
struct FISConfig
{
  int numInputs;
   int numOutputs;
   int maxInputMFs;
   int maxRules;
   int maxInitialRuleDepth;
   int maxOverallRuleDepth;
   int mfType;
   int numSetOps;
   bool probor;
   bool prod;
   bool wtaver;
   bool linearOutput;
   bool rampedInit;
   bool globalInputMFs;
   bool reorderTree;
   double parsimonyFactor;
   double pSelectInternal;
 };
struct EAConfig
{
  int maxGens;
   int popSize;
   int tourneySize;
   int crossType;
   int mutType;
   int selType;
   int numSubsets;
   double pCross;
   double pMut;
   double blendParam;
   bool steadyState;
   bool elitism;
   bool realParamUniform;
 };
struct ReportStruct
{
  int nEvals;
   double MAE;
   double MSE;
   double fitness;
   int stage;
 };
inline void
initReportStruct (ReportStruct * rs, int ty, int nev, double ae, double se,
          double fit, int st)
{
  rs->nEvals = nev;
  rs->MAE = ae;
  rs->MSE = se;
  rs->fitness = fit;
  rs->stage = st;
} inline void

initEAConfig (EAConfig * ec)
{
  ec->popSize = 100;
  ec->maxGens = 30;
  ec->tourneySize = 3;
  ec->steadyState = false;
  ec->crossType = CROSS_BLEND;
  ec->selType = SEL_ROULETTE;
  ec->mutType = MUT_GAUSS;
  ec->pCross = .9;
  ec->pMut = .05;
  ec->elitism = true;
  ec->numSubsets = 5;
  ec->blendParam = 0.5;
  ec->realParamUniform = true;
} inline void

printEAConfig (EAConfig * ec, ostream & out)
{
  out << "Pop size: " << ec->popSize << endl
    <<"Max gens: " << ec->maxGens << endl
    <<"Tourney size: " << ec->tourneySize << endl
    <<"Steady state: " << ec->steadyState << endl
    <<"Crossover type: " << ec->crossType << endl
    <<"Mutation type: " << ec->mutType << endl
    <<"Selection type: " << ec->selType << endl
    <<"pCrossover: " << ec->pCross << endl
    <<"pMutation: " << ec->pMut << endl
    <<"elitism: " << ec->elitism << endl
    <<"# subsets: " << ec->numSubsets << endl
    <<"Blend Parameter: " << ec->blendParam << endl
    <<"Uniform real parameter crossovers: " << ec->realParamUniform << endl;
} inline void

initFISConfig (FISConfig * fc)
{
  fc->numInputs = 3;
  fc->numOutputs = 1;
  fc->maxInputMFs = 3;
  fc->maxRules = 10;
  fc->maxInitialRuleDepth = 7;
  fc->maxOverallRuleDepth = 17;
  fc->probor = true;
  fc->prod = true;
  fc->wtaver = true;
  fc->linearOutput = true;
  fc->mfType = MF_GAUSS;
  fc->rampedInit = true;
  fc->parsimonyFactor = 0;
  fc->pSelectInternal = .9;
  fc->globalInputMFs = true;
  fc->numSetOps = 1;
  fc->reorderTree = true;
} inline void

printFISConfig (FISConfig * fc, ostream & out)
{
  out << "# Inputs: " << fc->numInputs << endl
    <<"# Outputs: " << fc->numOutputs << endl
    <<"MFs per Input: " << fc->maxInputMFs << endl
    <<"Max # rules: " << fc->maxRules << endl
    <<"Max initial rule depth: " << fc->maxInitialRuleDepth << endl
    <<"Max final rule depth: " << fc->maxOverallRuleDepth << endl
    <<"Use probor: " << fc->probor << endl
    <<"Use prod: " << fc->prod << endl
    <<"Use wtaver: " << fc->wtaver << endl
    <<"Use linear output: " << fc->linearOutput << endl
    <<"Use global input MFs: " << fc->globalInputMFs << endl
    <<"# of set operations: " << fc->numSetOps << endl
    <<"Reorder tree: " << fc->reorderTree << endl
    <<"pSelectInternal: " << fc->pSelectInternal << endl
    <<"rampedInit: " << fc->rampedInit << endl
    <<"parsimonyFactor: " << fc->parsimonyFactor << endl
    <<"mfType: " << fc->mfType << endl;
} struct VarBounds

{
  double maximum;
   double minimum;
 };




/////////////////////////////////////////////////////////////////////////////
// Eric A. Morris
// Univ. of Georgia AI Center
// FISSION Project
//
// FISRule.h
// Declaration for the FISRuleTree class

#pragma once

#include ".\fisrule.h"
#include ".\rulenode.h"
#include ".\fission_structures.h"
#include <iostream>
#include <fstream>
static bool full = false;
static int antSubRoleSet[] =
  { RuleNode::SUBROLE_AND, RuleNode::SUBROLE_OR, RuleNode::SUBROLE_NOT,
RuleNode::SUBROLE_IS };
using namespace std;
class FISRuleTree
{
private:RuleNode * root;
  FISConfig * theConfig;
  int *antSubRoleSet;
  void growTree (RuleNode * root, int rule, int depth, int maxDepth,
          bool full);
  double firingAux (RuleNode * curNode, double *mfLevels,
             bool * inputsLeftOfCenter);
  double and (double d1, double d2);
  double or (double d1, double d2);
  double not (double d1);
  void doSpacing (char *str, int len);
  void printNode (char *str, RuleNode * cur);
  RuleNode * findKthRule (int k);
  double applySetOp (int op, bool leftOfCenter, double mfLevel);
  void writeRuleAux (ostream & out, RuleNode * node);
  void serializeNode (ostream & out, RuleNode * node);
  RuleNode * deSerializeNode (istream & in, bool & p, bool & l, bool & r);
  void serializeTreeAux (ostream & out, RuleNode * node);
  void deSerializeTreeAux (istream & in, RuleNode * node, bool left);
public:FISRuleTree (FISConfig * conf, double pos);
  FISRuleTree (FISRuleTree * rt);
  FISRuleTree (FISConfig * conf, istream & in);
  ~FISRuleTree (void);
  void createWholeTree (double pos);
  void mutate ();
  void crossover (FISRuleTree * mate, FISRuleTree * &child1,
          FISRuleTree * &child2);
  double getFiringLevel (int rule, double *mfLevels,
             bool * inputsLeftOfCenter);
  int getNumRules ();
  int getDepth ();
  int getMaxRuleDepth ();
  int getNumNodes ();
  bool usedMF (int mf);
  bool usedMFAux (int mf, RuleNode * cur);
  void printTree (char *str, int rule);
  void printTreeAux (char *str, RuleNode * t, int level);
  void removeRules (bool * toRemove);
  void sortRulesByFiringLevel ();
  void writeRule (ostream & out, int rule);
  void serializeTree (ostream & out);
  void deSerializeTree (istream & in);
};
int chooseRandomSubRole (int num);
RuleNode * getRandomSubTree (RuleNode * root, double pSelectInternal);
RuleNode * getCompatibleSubTree (RuleNode * root, int rle, int sbrle,
                  double pSelectInternal);
RuleNode * findNode (RuleNode * root, int which, int minDepthBelow);
RuleNode * copyTree (RuleNode * root, RuleNode * parent);
int whichChild (RuleNode * parent, RuleNode * child);
void fixAncestorInfo (RuleNode * child);
int countRulesInTree (RuleNode * rn);




///////////////////////////////////////////////////////////////////////////////
// Eric A. Morris
// Univ. of Georgia AI Center
// FISSION Project
//
// FISSION_Utility.h
// Declaration for the utility functions (random, matrix, etc.)

#pragma once

#include <stdlib.h>
#include <cmath>
```

```
#include ".\FISSION_Structures.h"
const double RANDOM_SCALE = 1 / ((double) (RAND_MAX + 1));
void seedRand ();
void seedRand (long t);
double genRand ();
double genNormRand ();
int genRandInt (int lb, int ub);
double *copyArray (double *src, int n);
void mult_matrix_vector (double **m, double *v, int r, int c, double *out);
void sub_vector_vector (double *v1, double *v2, int n, double *out);
double norm_vector (double *v, int n);
double rSquared (int n, double *x, double *y);
double min3 (double a, double b, double c);
double max3 (double a, double b, double c);
```

```
////////////////////////////////////////////////////////////////////////////
// Eric A. Morris
// Univ. of Georgia AI Center
// FISSION Project
//
// FISSION-DLL.h
// Declaration for the DLL exported functions

#ifdef FISSIONDLL_EXPORTS
#define FISSIONDLL_API __declspec(dllexport)
#else   /*  */
#define FISSIONDLL_API __declspec(dllimport)
#endif  /*  */

#include "sugenofis.h"
#include "fisrule.h"
#include "fission_structures.h"
#include "fission_utility.h"
FISSIONDLL_API void abortGA ();
FISSIONDLL_API void applyFIS (SugenoFIS * sf, int n, double *id, double *od,
                   double &MSE, double &MAE, double &rSq,
                   double *predVals);
FISSIONDLL_API void applyFISNoOpt (SugenoFIS * sf, int n, double *id,
                    double *predVals);
FISSIONDLL_API void trainFIS (FISConfig * fc, EAConfig * ec, int n,
                   double *id, double *od, int m, double *testin,
                   double *testout);
FISSIONDLL_API bool usedMF (SugenoFIS * sf, int input, int mf);
FISSIONDLL_API SugenoFIS * getBestFIS ();
FISSIONDLL_API SugenoFIS * getGeneralizeFIS ();
FISSIONDLL_API int getInputMFCount (SugenoFIS * sf, int input);
FISSIONDLL_API double getInputMFValue (SugenoFIS * sf, int input, int mf,
                    double x);
FISSIONDLL_API int getMaxRuleDepth (SugenoFIS * sf);
FISSIONDLL_API int getNumRules (SugenoFIS * sf);
FISSIONDLL_API void getPrediction (double *pred);
FISSIONDLL_API void getReport (ReportStruct * rs);
FISSIONDLL_API char *getRule (SugenoFIS * sf, int r);
FISSIONDLL_API char *genCode (SugenoFIS * sf, const char *fn);
FISSIONDLL_API void saveFIS (SugenoFIS * sf, const char *fn);
FISSIONDLL_API SugenoFIS * loadFIS (const char *fn);
```

```
////////////////////////////////////////////////////////////////////////////
// Eric A. Morris
// Univ. of Georgia AI Center
// FISSION Project
//
// FISSION-EA.h
// Declaration for the TrainerEA class

#pragma once

#include "sugenofis.h"
#include "fisrule.h"
#include "fission_structures.h"
#include "fission_utility.h"
#include "fission-ea.h"
#include "windows.h"
static const bool GEN_OUTPUT = true;
static const char *EA_OUT_FILE = "ea-report.csv";
class TrainerEA
{
private:static const int nFitnessParams = 1;
  FISConfig * theFISConfig;
  EAConfig * theEAConfig;
  VarBounds * varBounds;
  SugenoFIS ** population;
  SugenoFIS ** nextGen;
  double *fitness;
  int *fitnessRank;
  double bestFitness;
  double bestMAE;
  double bestMSE;
  double bestGeneralizeFitness;
  double bestGeneralizeMAE;
  double bestGeneralizeMSE;
  double bestGeneralizeR2;
  int bestFitnessIndex;
  double popMaxFitness;
  double popMinFitness;
  double fitnessSum;
  int numFitEvals;
  double **inputData;
  double **auxInputData;
  double *outputData;
  double *auxOutputData;
  double *vals;
  double *vals2;
  double *bestVals1;
  double *bestVals2;
  int trainingSetSize;
  int auxSetSize;
  int inputChrSize;
  int outputChrSize;
  int crStage;
  SugenoFIS * bestFIS1;
  SugenoFIS * bestFIS2;
  bool abortRun;
  int trainingStage;
  int curSubset;
  int subsetSize;
  int **subsets;
  int *fullSet;
  long startTime;
  HANDLE semReportData;
  void initPopulation ();
  double evaluateFitness (int i);
  void copyGens ();
  void breed (SugenoFIS * &child1, SugenoFIS * &child2);
  int doSelect ();
  int rouletteSelect ();
  int selectDeath ();
  int rankSelect ();
  int tournamentSelect ();
  void generateSubsets ();
  void doCrossover (double *parent1, double *parent2, double *&child1,
              double *&child2, int strLen);
  void twoPtCrossover (double *parent1, double *parent2, double *&child1,
            double *&child2, int strLen);
  void uniformCrossover (double *parent1, double *parent2, double *&child1,
              double *&child2, int strLen);
  void blendCrossover (double *parent1, double *parent2, double *&child1,
            double *&child2, int strLen);
  void arithmeticCrossover (double *parent1, double *parent2,
                 double *&child1, double *&child2, int strLen);
  double gaussNoise (double x, double y);
  double blend (double p1, double p2, double a);
  double linCombine (double p1, double p2, double a);
  void treeCrossover (FISRuleTree * parent1, FISRuleTree * parent2,
               FISRuleTree * &child1, FISRuleTree * &child2);
  void doMutate (double *child, int strLen);
  void mutateStringGauss (double *child, int strLen);
  void mutateStringUniform (double *child, int strLen);
  void mutateTree (FISRuleTree * child);
  void updateRanks (int childIndex, double child, double oldValue);
  void steadyStatePostBreed (SugenoFIS * child);
  void generationalPostBreed ();
  void runSteadyState ();
  void runGenerational ();
  void writeStatus ();
public:TrainerEA (FISConfig * fc, EAConfig * ec, VarBounds * vb, int n,
          double **id, double *od, int n1, double **aid,
          double *aod);
  ~TrainerEA ();
  void runGA ();
  void abort ();
  double getBestFitness ();
  double getBestMAE ();
  double getBestMSE ();
  int getNumEvals ();
  void getCurrentPrediction (double *pred);
  SugenoFIS * getBestFIS ();
  int getStage ();
  SugenoFIS * getGeneralizeFIS ();
  void getGeneralizePrediction (double *pred);
  double getBestGeneralizeMAE ();
  double getBestGeneralizeMSE ();
  double getBestGeneralizeR2 ();
  double getBestGeneralizeFitness ();
};
```

```
//////////////////////////////////////////////////////////////////////////////
// Eric A. Morris
// Univ. of Georgia AI Center
// FISSION Project
//
// RuleNode.h
// Declaration and implementation for the RuleNode class (inline)

#pragma once
  class RuleNode
{
public:int role;
  int subRole;
  int value;
  int operation;
  static const int ROLE_IFTHEN = 1;
  static const int ROLE_ANT = 2;
  static const int ROLE_TERM = 3;
  static const int SUBROLE_NOT = 5;
  static const int SUBROLE_AND = 6;
  static const int SUBROLE_OR = 7;
  static const int SUBROLE_IS = 8;
  static const int SUBROLE_IN = 9;
  static const int SUBROLE_MF = 10;
  static const int OP_EQ = 0;
  static const int OP_GT = 1;
  static const int OP_LT = 2;
  static const int OP_GEQ = 3;
  static const int OP_LEQ = 4;
  static const int OP_NEQ = 5;
```

```
int depthBelow;
int nodesBelow;
double firingLevel;
RuleNode * parent;
RuleNode * left;
RuleNode * right;
RuleNode ()
{
  role = ROLE_IFTHEN;
  depthBelow = 0;
  nodesBelow = 0;
  parent = NULL;
  left = NULL;
  right = NULL;
  firingLevel = 0;
  operation = OP_EQ;
} RuleNode (RuleNode * prnt, int rl, int sbrl, int vl, int op)
{
  left = NULL;
  right = NULL;
  parent = prnt;
  depthBelow = 0;
  nodesBelow = 0;
  role = rl;
  subRole = sbrl;
  value = vl;
  firingLevel = 0;
  operation = op;
} RuleNode (RuleNode * r, RuleNode * newParent)
{
  left = NULL;
  right = NULL;
  parent = newParent;
  role = r->role;
  subRole = r->subRole;
  value = r->value;
  depthBelow = r->depthBelow;
  nodesBelow = r->nodesBelow;
  firingLevel = r->firingLevel;
  operation = r->operation;
} ~RuleNode ()
{
  if (left)
    {
  delete left;
  left = NULL;
    }
  if (right)
    {
  delete right;
  right = NULL;
    }
}
int getDepth ()
```

```
  {
    int depth = 0;
    RuleNode * cur = this;
    while (cur != NULL && cur->role != RuleNode::ROLE_IFTHEN)
      {
    cur = cur->parent;
    depth++;
      }
    return depth;
  }
  int getRuleNumber ()
  {
    RuleNode * cur = this;
    int count = 0;
    while (cur && cur->role != RuleNode::ROLE_IFTHEN)
      {
    cur = cur->parent;
      }
    cur = cur->parent;
    while (cur)
      {
    cur = cur->parent;
    count++;
      }
    return count;
  }
};
```

```
//////////////////////////////////////////////////////////////////////////////
// Eric A. Morris
// Univ. of Georgia AI Center
// FISSION Project
//
// SugenoFIS.h
// Declaration for the SugenoFIS class

#pragma once

#include ".\fission_structures.h"
#include ".\FISRule.h"
  class SugenoFIS
{
private:double *inputMFs;
  double *outputMFs;
  FISRuleTree * infRules;
  FISConfig * theConfig;
  VarBounds * varBounds;
  int inputChrSize;
  int outputChrSize;
  void doCommonInit ();
  void generateRandomInputMFs ();
  void generateRandomOutputMFs ();
  double gaussmf (int mf, double x);
  double gbellmf (int mf, double x);
  double trimf (int mf, double x);
  double trapmf (int mf, double x);
  void getFiringLevels (int N, double **x, double **&f);
  double applyOutputMF (int which, bool useLinear, double *x);
  void removeDependencies (int N, double **x, double **&f);
  double getMFCenter (int mf);
  void writeInputMFs (ostream & out);
  void writeOutputMFs (ostream & out);
public:SugenoFIS (FISConfig * conf, VarBounds * vb, double pos);
  SugenoFIS (FISConfig * conf, VarBounds * vb, double *inmf,
          FISRuleTree * ft);
  SugenoFIS (FISConfig * conf, VarBounds * vb, double *inmf,
          FISRuleTree * ft, double *omf);
  SugenoFIS (SugenoFIS * sf);
  SugenoFIS (const char *fn);
  ~SugenoFIS (void);
  double *copyInputMFs ();
  double *copyOutputMFs ();
  void apply (int N, double **x, double *y, double *outputValues,
           double &MSE, double &MAE, double &rSq);
  void applyNoOutputs (int N, double **x, double *outputValues);
  double runLinearKalmanFilter (int N, int *set, double **x, double *y);
  double runConstantKalmanFilter (int N, int *set, double **x, double *y);
  double evaluateEfficiency ();
  double *getInputMFs ();
  double *getOutputMFs ();
  FISRuleTree * getRuleSet ();
  FISRuleTree * copyRules ();
  FISConfig * getConfig ();
  VarBounds * getBounds ();
  void getMFsForInput (int input, double *MFs);
  void printOutputMF (char *str, int mf);
  double getMFValue (int mf, double x);
  double getInputMin (int input);
  double getInputMax (int input);
  char *codeFIS (const char *fn);
  void serializeFIS (const char *fn);
};
void findMaxMins (FISConfig * conf, int N, double **inputs, double *outputs,
            VarBounds * &bounds);
void scaleInputs (FISConfig * conf, int N, double *maxes, double *mins,
           double **inputs, double **&scaledInputs);
bool loadDataFromFile (int N, int M, const char *fn, double **&x,
            double *&y);
void allocFISMem (int N, int M, int S, int numInputMFs);
void deallocFISMem (int N, int M, int S);
void serializeConfig (ostream & out, FISConfig * conf);
void serializeBounds (ostream & out, VarBounds * vb);
FISConfig * deSerializeConfig (istream & in);
VarBounds * deSerializeBounds (istream & in);
```

```
//////////////////////////////////////////////////////////////////////////////
// Eric A. Morris
// Univ. of Georgia AI Center
// FISSION Project
//
// FISRule.cpp
// Implmentation for the FISRuleTree class

#include "StdAfx.h"
#include ".\fisrule.h"
#include ".\fission_structures.h"
#include ".\fission_utility.h"
#include <iostream>
#include <fstream>
  using namespace std;
FISRuleTree::FISRuleTree (FISRuleTree * rt)
{
  theConfig = rt->theConfig;
  root = copyTree (rt->root, NULL);
} FISRuleTree::FISRuleTree (FISConfig * conf, double pos)
{
  theConfig = conf;
  root = NULL;
  createWholeTree (pos);
} FISRuleTree::FISRuleTree (FISConfig * conf, istream & in)
{
  theConfig = conf;
  root = NULL;
  deSerializeTree (in);
} FISRuleTree::~FISRuleTree ()
{
  delete root;
  root = NULL;
} int

FISRuleTree::getNumRules ()
{
  int count = 0;
  RuleNode * curNode;
  if (!root)
    return -1;
  curNode = root->right;
  while (curNode != NULL)
    {
      count++;
      curNode = curNode->right;
    }
  return count + 1;
}
int
FISRuleTree::getMaxRuleDepth ()
{
  int maxDepth = 0;
  RuleNode * curNode = root;
  if (!root)
    return -1;
  while (curNode != NULL)
    {
      if (curNode->left->depthBelow > maxDepth)
    maxDepth = curNode->left->depthBelow;
      curNode = curNode->right;
    }
  return maxDepth + 2;
}
int
FISRuleTree::getDepth ()
{
  return root->depthBelow;
}
int
FISRuleTree::getNumNodes ()
{
  return root->nodesBelow + 1;
}

bool FISRuleTree::usedMF (int mf)
{
  return usedMFAux (mf, root);
}

bool FISRuleTree::usedMFAux (int mf, RuleNode * cur)
{
  if (!cur)
    return false;
  if (cur->role == RuleNode::ROLE_ANT
    && cur->subRole == RuleNode::SUBROLE_IS
    && (cur->left->value * theConfig->maxInputMFs + cur->right->value ==
        mf))
    return true;
  if (usedMFAux (mf, cur->left))
    return true;
  if (usedMFAux (mf, cur->right))
    return true;
  return false;
}

RuleNode * FISRuleTree::findKthRule (int k)
{
  RuleNode * curNode = root;
  int
    curIndex = 0;
  while (curNode != NULL && curIndex < k)
    {
      curNode = curNode->right;
      curIndex++;
    }
  return curNode;
```

```
}
double
FISRuleTree::getFiringLevel (int rule, double *mfLevels,
                 bool * inputsLeftOfCenter)
{
  RuleNode * rn = findKthRule (rule);
  return firingAux (rn->left, mfLevels, inputsLeftOfCenter);
}
double
FISRuleTree::firingAux (RuleNode * curNode, double *mfLevels,
            bool * inputsLeftOfCenter)
{
  int whichMF;
  if (!curNode)
    {
      cout << "ERROR IN TREE!" << endl;
      return 0;
    }
  switch (curNode->subRole)
    {
    case RuleNode::SUBROLE_AND:
      return and (firingAux (curNode->left, mfLevels, inputsLeftOfCenter),
           firingAux (curNode->right, mfLevels, inputsLeftOfCenter));
    case RuleNode::SUBROLE_IS:
      whichMF =
    curNode->left->value * theConfig->maxInputMFs + curNode->right->value;
      return applySetOp (curNode->operation, inputsLeftOfCenter[whichMF],
             mfLevels[whichMF]);
    case RuleNode::SUBROLE_NOT:
      return not (firingAux (curNode->left, mfLevels, inputsLeftOfCenter));
    case RuleNode::SUBROLE_OR:
      return or (firingAux (curNode->left, mfLevels, inputsLeftOfCenter),
          firingAux (curNode->right, mfLevels, inputsLeftOfCenter));
    }
  return curNode->value;
}
double
FISRuleTree::applySetOp (int op, bool leftOfCenter, double mfLevel)
{
  double temp;
  switch (op)
    {
    case RuleNode::OP_EQ:
      return mfLevel;
    case RuleNode::OP_NEQ:
      return 1 - mfLevel;
    case RuleNode::OP_LT:
      if (leftOfCenter)
    return 1 - mfLevel;
      return 0;
    case RuleNode::OP_GT:
      if (!leftOfCenter)
    return 1 - mfLevel;
      return 0;
    case RuleNode::OP_LEQ:
      if (leftOfCenter)
    temp = 1 - mfLevel;

      else
    temp = 0;
      return max (mfLevel, temp);
    case RuleNode::OP_GEQ:
      if (!leftOfCenter)
    temp = 1 - mfLevel;

      else
    temp = 0;
      return max (mfLevel, temp);
    }
  return mfLevel;
}
void
FISRuleTree::createWholeTree (double pos)
{
  int depth;
  root = new RuleNode (NULL, RuleNode::ROLE_IFTHEN, 999, 999, 999);
  if (theConfig->rampedInit)
    {
      full = !full;
      depth = (int) (pos * (theConfig->maxInitialRuleDepth - 2)) + 3;
      growTree (root, 1, 1, depth, full);

    //cout << pos << "   " << depth << "   " << full << endl;
    //cin.get();
    }

  else
    {
      growTree (root, 1, 1, theConfig->maxInitialRuleDepth, false);
    }
}
void
FISRuleTree::growTree (RuleNode * curNode, int rule, int depth, int maxDepth,
               bool full)
{
  int chosenSubRole1;
  int chosenSubRole2;
  int nIMF = theConfig->maxInputMFs * theConfig->numInputs;
  int nOMF = theConfig->maxRules * theConfig->numOutputs;
  if ((maxDepth - depth) <= 2)
    {
      chosenSubRole1 = RuleNode::SUBROLE_IS;
      chosenSubRole2 = RuleNode::SUBROLE_IS;
    }

  else
    {
      if (full)
    {
      chosenSubRole1 = chooseRandomSubRole (3);
      chosenSubRole2 = chooseRandomSubRole (3);
    }

      else
    {
      chosenSubRole1 = chooseRandomSubRole (4);
      chosenSubRole2 = chooseRandomSubRole (4);
    }
    }
  if (curNode->role == RuleNode::ROLE_IFTHEN)
    {
      if (!curNode->left)
    {
      curNode->left =
        new RuleNode (curNode, RuleNode::ROLE_ANT, chosenSubRole1, 999,
              genRandInt (0, theConfig->numSetOps));
      growTree (curNode->left, rule, depth + 1, maxDepth, full);
    }
      if (!curNode->right)
    {
      if (rule < theConfig->maxRules)
        {
          curNode->right =
        new RuleNode (curNode, RuleNode::ROLE_IFTHEN, 999, 999, 999);
          growTree (curNode->right, rule + 1, 1, maxDepth, full);
        }

      else
        {
          curNode->right = NULL;
        }
    }
    }

  else if (curNode->role == RuleNode::ROLE_ANT)
    {
      if (curNode->subRole == RuleNode::SUBROLE_IS)
    {
      curNode->left =
        new RuleNode (curNode, RuleNode::ROLE_TERM, RuleNode::SUBROLE_IN,
              genRandInt (0, theConfig->numInputs), 999);
      curNode->right =
        new RuleNode (curNode, RuleNode::ROLE_TERM, RuleNode::SUBROLE_MF,
              genRandInt (0, theConfig->maxInputMFs), 999);
    }

      else if (curNode->subRole == RuleNode::SUBROLE_NOT)
    {
      if (!curNode->left)
        {
          curNode->left =
        new RuleNode (curNode, RuleNode::ROLE_ANT, chosenSubRole1,
                  999, genRandInt (0, theConfig->numSetOps));
          growTree (curNode->left, rule, depth + 1, maxDepth, full);
        }
      curNode->right = NULL;
    }

      else
    {
      if (!curNode->left)
        {
          curNode->left =
        new RuleNode (curNode, RuleNode::ROLE_ANT, chosenSubRole1,
                  999, genRandInt (0, theConfig->numSetOps));
          growTree (curNode->left, rule, depth + 1, maxDepth, full);
        }
      if (!curNode->right)
        {
          curNode->right =
        new RuleNode (curNode, RuleNode::ROLE_ANT, chosenSubRole2,
                  999, genRandInt (0, theConfig->numSetOps));
          growTree (curNode->right, rule, depth + 1, maxDepth, full);
        }
    }
    }
  if (!curNode->left && !curNode->right)
    {
      curNode->nodesBelow = 0;
      curNode->depthBelow = 0;
    }

  else if (!curNode->left)
    {
      curNode->nodesBelow = 1 + curNode->right->nodesBelow;
      curNode->depthBelow = 1 + curNode->right->depthBelow;
    }

  else if (!curNode->right)
    {
      curNode->nodesBelow = 1 + curNode->left->nodesBelow;
      curNode->depthBelow = 1 + curNode->left->depthBelow;
    }

  else
    {
      curNode->nodesBelow =
    2 + curNode->left->nodesBelow + curNode->right->nodesBelow;
      curNode->depthBelow =
    1 + max (curNode->left->depthBelow, curNode->right->depthBelow);
    }
}
void
FISRuleTree::mutate ()
{
  int wc = 0;
  int whichMutation = genRandInt (0, 4);
  int sub;
  int val;
  RuleNode * mutateNode;
  RuleNode * mutateRoot;
  RuleNode * mutateParent;
  switch (whichMutation)
    {
    case 0:         // terminal mutation
      if (genRand () < .5)
    {
      sub = RuleNode::SUBROLE_IN;
      val = genRandInt (0, theConfig->numInputs);
    }

      else
    {
      sub = RuleNode::SUBROLE_MF;
      val = genRandInt (0, theConfig->maxInputMFs);
    }
      mutateNode = getCompatibleSubTree (root, RuleNode::ROLE_TERM, sub, .5);
      mutateNode->value = val;
      break;
    case 1:         // function mutation
      mutateNode = getCompatibleSubTree (root, RuleNode::ROLE_ANT, -1, .5);
      if (mutateNode->subRole == RuleNode::SUBROLE_AND)
    mutateNode->subRole = RuleNode::SUBROLE_OR;

      else if (mutateNode->subRole == RuleNode::SUBROLE_OR)
    mutateNode->subRole = RuleNode::SUBROLE_AND;
      break;
    case 2:         // truncate mutation
      mutateNode = getRandomSubTree (root, 1);
      if (mutateNode->role == RuleNode::ROLE_IFTHEN)
    {
      if (mutateNode->right)
        {
          delete mutateNode->right;
          mutateNode->right = NULL;
        }
      fixAncestorInfo (mutateNode->left);
    }

      else if (mutateNode->role == RuleNode::ROLE_ANT)
    {
      if (mutateNode->left)
        {
          delete mutateNode->left;
          mutateNode->left = NULL;
        }
      if (mutateNode->right)
        {
          delete mutateNode->right;
          mutateNode->right = NULL;
        }
      mutateNode->subRole = RuleNode::SUBROLE_IS;
      mutateNode->operation = genRandInt (0, theConfig->numSetOps);
      growTree (mutateNode, mutateNode->getRuleNumber () + 1,
             mutateNode->getDepth () + 1,
             theConfig->maxInitialRuleDepth, false);
      fixAncestorInfo (mutateNode->left);
    }
      break;
    case 3:         // grow mutation
      mutateRoot = getRandomSubTree (root, theConfig->pSelectInternal);
      mutateParent = mutateRoot->parent;
      if (mutateParent != NULL)
    {
      wc = whichChild (mutateParent, mutateRoot);
      if (wc == 0)
        {
          delete mutateParent->left;
          mutateParent->left = NULL;
        }

      else
        {
          delete mutateParent->right;
          mutateParent->right = NULL;
        }
      growTree (mutateParent, mutateParent->getRuleNumber () + 1,
              mutateParent->getDepth () + 1,
              theConfig->maxInitialRuleDepth, false);
      fixAncestorInfo (mutateParent->left);
    }
      break;
    }
}
void
FISRuleTree::crossover (FISRuleTree * mate, FISRuleTree * &child1,
            FISRuleTree * &child2)
{
  int d1, d2, wc1, wc2;
  RuleNode * subTree1;
  RuleNode * subTree1copy;
  RuleNode * subTree2;
  RuleNode * subTree2copy;
  RuleNode * parentNode1;
  RuleNode * parentNode2;
  int rulesInSubtree1;
  int rulesInSubtree2;
  child1 = new FISRuleTree (this);
  child2 = new FISRuleTree (mate);

    // try to prevent breaking up building blocks
    // by grouping strong rules together
    if (theConfig->reorderTree)
    {
```

```
    child1->sortRulesByFiringLevel ();
    child2->sortRulesByFiringLevel ();
  }
subTree1 = getRandomSubTree (child1->root, theConfig->pSelectInternal);
subTree2 =
  getCompatibleSubTree (child2->root, subTree1->role, subTree1->subRole,
            theConfig->pSelectInternal);

  //cout << subTree1->role << "   " << subTree2->role << endl;
  parentNode1 = subTree1->parent;
parentNode2 = subTree2->parent;
subTree1copy = copyTree (subTree1, NULL);
subTree2copy = copyTree (subTree2, NULL);
rulesInSubtree1 = countRulesInTree (subTree1copy);
rulesInSubtree2 = countRulesInTree (subTree2copy);
if (parentNode1 == NULL && parentNode2 == NULL)
  {
    return;
  }

  // paste some of child 2 into child 1
  if (parentNode1 == NULL)
  {
    delete child1->root;
    child1->root = NULL;
    child1->root = subTree2copy;
    subTree2copy->parent = NULL;
  }

else
  {
    wc1 = whichChild (parentNode1, subTree1);
    d1 = parentNode1->getDepth ();
    if (d1 + subTree2copy->depthBelow <= theConfig->maxOverallRuleDepth
      &&(rulesInSubtree2 + parentNode1->getRuleNumber () + 1) <
      theConfig->maxRules)
  {
    if (wc1 == 0)
      {
        delete parentNode1->left;
        parentNode1->left = subTree2copy;
        subTree2copy->parent = parentNode1;
        fixAncestorInfo (subTree2copy);
      }

    else
      {
        delete parentNode1->right;
        parentNode1->right = subTree2copy;
        subTree2copy->parent = parentNode1;
        fixAncestorInfo (subTree2copy);
      }
  }
```

```
    else
  {

      //cout << "REJ" << endl;
      delete subTree2copy;
    subTree2copy = NULL;
  }
  }

  // paste some of child1 into child2
  if (parentNode2 == NULL)
  {
    delete child2->root;
    child2->root = NULL;
    child2->root = subTree1copy;
    subTree1copy->parent = NULL;
  }

else
  {
    wc2 = whichChild (parentNode2, subTree2);
    d2 = parentNode2->getDepth ();
    if (d2 + subTree1->depthBelow <= theConfig->maxOverallRuleDepth
      &&(rulesInSubtree1 + parentNode2->getRuleNumber () + 1) <
      theConfig->maxRules)
  {
    if (wc2 == 0)
      {
        delete parentNode2->left;
        parentNode2->left = subTree1copy;
        subTree1copy->parent = parentNode2;
        fixAncestorInfo (subTree1copy);
      }

    else
      {
        delete parentNode2->right;
        parentNode2->right = subTree1copy;
        subTree1copy->parent = parentNode2;
        fixAncestorInfo (subTree1copy);
      }
  }

    else
  {

      //cout << "REJ" << endl;
      delete subTree1copy;
    subTree1copy = NULL;
  }
  }

  //cout << "After Crossover: " << child1->getNumRules()
         //<< "   " << child2->getNumRules() << endl;
```

```
}
double
FISRuleTree::and (double d1, double d2)
{
  if (theConfig->prod)
    {
      return d1 * d2;
    }
  return min (d1, d2);
}
double
FISRuleTree::or (double d1, double d2)
{
  if (theConfig->probor)
    {
      return d1 + d2 - d1 * d2;
    }
  return max (d1, d2);
}
double
FISRuleTree::not (double d1)
{
  return 1 - d1;
}
int
chooseRandomSubRole (int num)
{
  int ret = genRandInt (0, num);
  return antSubRoleSet[ret];
}

RuleNode * getRandomSubTree (RuleNode * root, double pSelectInternal)
{
  int minDepthBelow = 0;
  bool selectInternal = (genRand () < pSelectInternal);
  RuleNode * cur;

  do
    {
      int which = genRandInt (0, root->nodesBelow);
      cur = findNode (root, which, minDepthBelow);
    }
  while ((selectInternal && cur->role == RuleNode::ROLE_TERM)
     || (!selectInternal && cur->role != RuleNode::ROLE_TERM));
  return cur;
}

RuleNode * getCompatibleSubTree (RuleNode * root, int r, int sbrle,
                   double pSelectInternal)
{
  int which, minDepthBelow = 0;
  RuleNode * found;

  do
```

```
    {
      which = genRandInt (0, root->nodesBelow);
      found = findNode (root, which, minDepthBelow);
    }
  while (found->role != r
     || (r == RuleNode::ROLE_TERM && found->subRole != sbrle));
  return found;
}

RuleNode * findNode (RuleNode * root, int which, int minDepthBelow)
{
  if (which == 0)
    return root;
  if (root->depthBelow < minDepthBelow)
    {
      if (root->parent != NULL)
    return root->parent;

      else
    return root;
    }
  if (!root->left && !root->right)
    cout << "This should not happen!!" << endl;
  if (which <= (root->left->nodesBelow + 1))
    {
      return findNode (root->left, which - 1, minDepthBelow);
    }
  return findNode (root->right, which - (root->left->nodesBelow + 2),
           minDepthBelow);
}

RuleNode * copyTree (RuleNode * root, RuleNode * parent)
{
  if (!root)
    return NULL;
  RuleNode * ret = new RuleNode (root, parent);
  ret->left = copyTree (root->left, ret);
  ret->right = copyTree (root->right, ret);
  return ret;
}
int
whichChild (RuleNode * parent, RuleNode * child)
{
  if (parent->left == child)
    return 0;
  if (parent->right == child)
    return 1;
  return -999;
}
void
fixAncestorInfo (RuleNode * child)
{
  RuleNode * par = child->parent;
  while (par != NULL)
```

```
    {
      if (!par->left && !par->right)
    {
      par->nodesBelow = 0;
      par->depthBelow = 0;
    }

      else if (!par->left)
    {
      par->nodesBelow = 1 + par->right->nodesBelow;
      par->depthBelow = 1 + par->right->depthBelow;
    }

      else if (!par->right)
    {
      par->nodesBelow = 1 + par->left->nodesBelow;
      par->depthBelow = 1 + par->left->depthBelow;
    }

      else
    {
      par->nodesBelow =
        2 + par->left->nodesBelow + par->right->nodesBelow;
      par->depthBelow =
        1 + max (par->left->depthBelow, par->right->depthBelow);
    }
      par = par->parent;
    }
}
void
FISRuleTree::printTree (char *str, int rule)
{
  RuleNode * rn = findKthRule (rule);
  printTreeAux (str, rn, 0);
} void
FISRuleTree::printTreeAux (char *str, RuleNode * t, int level)
{
  if (t == NULL || t->subRole == RuleNode::SUBROLE_IN
      || t->subRole == RuleNode::SUBROLE_MF)
    return;
  char *temp = new char[50];
  doSpacing (str, level);
  printNode (str, t);
  printTreeAux (str, t->left, level + 1);
  if (t->role != RuleNode::ROLE_IFTHEN)
    {
      printTreeAux (str, t->right, level + 1);
    }
  if (t->role == RuleNode::ROLE_ANT && t->subRole != RuleNode::SUBROLE_IS)
    {
      doSpacing (str, level);
      strcat (str, ")\r\n");
    }
  delete[]temp;
```

```
}
void
FISRuleTree::doSpacing (char *str, int len)
{
  if (len <= 0)
    return;
  char *temp = new char[3 * len + 1];
  for (int i = 0; i < 3 * len; i++)
    temp[i] = ' ';
  temp[3 * len] = '\0';
  strcat (str, temp);
  delete[]temp;
} void
FISRuleTree::printNode (char *str, RuleNode * cur)
{
  char *temp = new char[50];
  char *temp2 = new char[5];
  temp[0] = '\0';
  temp2[0] = '\0';
  if (cur->role == RuleNode::ROLE_IFTHEN)
    {
      sprintf (temp, "IF");
    }

  else if (cur->role == RuleNode::ROLE_ANT)
    {
      if (cur->subRole == RuleNode::SUBROLE_AND)
    {
      sprintf (temp, "AND");
    }

      else if (cur->subRole == RuleNode::SUBROLE_OR)
    {
      sprintf (temp, "OR");
    }

      else if (cur->subRole == RuleNode::SUBROLE_NOT)
    {
      sprintf (temp, "NOT");
    }

      else if (cur->subRole == RuleNode::SUBROLE_IS)
    {
      switch (cur->operation)
        {
        case RuleNode::OP_EQ:
          sprintf (temp2, "=");
          break;
        case RuleNode::OP_NEQ:
          sprintf (temp2, "!=");
          break;
        case RuleNode::OP_LEQ:
          sprintf (temp2, "<=");
          break;
        case RuleNode::OP_GEQ:
          sprintf (temp2, ">=");
          break;
        case RuleNode::OP_LT:
          sprintf (temp2, "<");
          break;
        case RuleNode::OP_GT:
          sprintf (temp2, ">");
          break;
        }
      sprintf (temp, "Input%d %s %d)", cur->left->value + 1, temp2,
            cur->right->value + 1);
    }
    }
  strcat (str, "(");
  strcat (str, temp);
  strcat (str, "\r\n");
  delete[]temp;
  delete[]temp2;
}
void
FISRuleTree::removeRules (bool * toRemove)
{

    //cout << "Before: " << root->numChildren << endl;
    /*RuleNode* newRoot = new RuleNode(this->root, NULL);
       int nCh=0;
       for(int i=0; i<this->root->numChildren; i++) {
       if(!toRemove[i]) {
       newRoot->children[nCh]=root->children[i];
       newRoot->children[nCh]->parent=newRoot;
       nCh++;
       }
       else {
       delete root->children[i];
       }
       root->children[i]=NULL;
       }
       newRoot->numChildren = nCh;
       fixAncestorInfo(newRoot->children[0]);

       delete root;
       root = newRoot;
       //cout << "After: " << root->numChildren << endl; */
} int

countRulesInTree (RuleNode * rn)
{
  RuleNode * cur = rn;
  int count;
  if (rn == NULL || rn->role != RuleNode::ROLE_IFTHEN)
    {
      return 0;
    }
  count = 1;
  cur = cur->right;
  while (cur != NULL)
    {
      count++;
      cur = cur->right;
    }
  return count;
}
void
FISRuleTree::sortRulesByFiringLevel ()
{
  int nRules = getNumRules ();
  RuleNode ** ruleArray = new RuleNode *[nRules];
  RuleNode * curNode = root;
  RuleNode * temp = NULL;
  int curIndex = 0;
  while (curNode)
    {
      ruleArray[curIndex] = curNode;
      curIndex++;
      curNode = curNode->right;
    }
  for (int i = 0; i < nRules; i++)
    {
      for (int j = i + 1; j < nRules; j++)
    {
      if (ruleArray[i]->firingLevel < ruleArray[j]->firingLevel)
        {
          temp = ruleArray[i];
          ruleArray[i] = ruleArray[j];
          ruleArray[j] = temp;
        }
    }
    }
  root = ruleArray[0];
  root->parent = NULL;
  curNode = root;
  curIndex = 1;
  while (curIndex < nRules)
    {
      curNode->firingLevel = 0;
      curNode->right = ruleArray[curIndex];
      curNode->right->parent = curNode;
      curIndex++;
      curNode = curNode->right;
    }
  ruleArray[nRules - 1]->right = NULL;
  fixAncestorInfo (ruleArray[nRules - 1]->left);
  delete[]ruleArray;
}
void
FISRuleTree::writeRule (ostream & out, int rule)
{
  RuleNode * rn = findKthRule (rule);
  writeRuleAux (out, rn->left);
} void

FISRuleTree::writeRuleAux (ostream & out, RuleNode * node)
{
  int whichMF;
  switch (node->subRole)
    {
    case RuleNode::SUBROLE_AND:
      out << "and(";
      writeRuleAux (out, node->left);
      out << ",";
      writeRuleAux (out, node->right);
      out << ")";
      break;
    case RuleNode::SUBROLE_IS:
      whichMF =
    node->left->value * theConfig->maxInputMFs + node->right->value;
      out << "setOp(";
      out << node->operation;
      out << ",loc[";
      out << whichMF;
      out << "],mf[";
      out << whichMF;
      out << "])";
      break;
    case RuleNode::SUBROLE_NOT:
      out << "not(";
      writeRuleAux (out, node->left);
      out << ")";
      break;
    case RuleNode::SUBROLE_OR:
      out << "or(";
      writeRuleAux (out, node->left);
      out << ",";
      writeRuleAux (out, node->right);
      out << ")";
      break;
    }
}
void
FISRuleTree::serializeNode (ostream & out, RuleNode * node)
{
  bool p = (node->parent != NULL);
  bool l = (node->left != NULL);
  bool r = (node->right != NULL);
  out.write ((char *) &node->depthBelow, sizeof (int));
  out.write ((char *) &node->firingLevel, sizeof (double));
  out.write ((char *) &node->nodesBelow, sizeof (int));
  out.write ((char *) &node->operation, sizeof (int));
  out.write ((char *) &node->role, sizeof (int));
  out.write ((char *) &node->subRole, sizeof (int));
  out.write ((char *) &node->value, sizeof (int));
  out.write ((char *) &p, sizeof (bool));
  out.write ((char *) &l, sizeof (bool));
  out.write ((char *) &r, sizeof (bool));
}

RuleNode * FISRuleTree::deSerializeNode (istream & in, bool & p, bool & l,
                       bool & r)
{
  RuleNode * node = new RuleNode ();
  in.read ((char *) &node->depthBelow, sizeof (int));
  in.read ((char *) &node->firingLevel, sizeof (double));
  in.read ((char *) &node->nodesBelow, sizeof (int));
  in.read ((char *) &node->operation, sizeof (int));
  in.read ((char *) &node->role, sizeof (int));
  in.read ((char *) &node->subRole, sizeof (int));
  in.read ((char *) &node->value, sizeof (int));
  in.read ((char *) &p, sizeof (bool));
  in.read ((char *) &l, sizeof (bool));
  in.read ((char *) &r, sizeof (bool));
  return node;
}
void
FISRuleTree::serializeTree (ostream & out)
{
  serializeTreeAux (out, root);
} void

FISRuleTree::deSerializeTree (istream & in)
{
  bool p, l, r;
  root = deSerializeNode (in, p, l, r);
  root->parent = NULL;
  if (l)
    {
      deSerializeTreeAux (in, root, true);
    }
  if (r)
    {
      deSerializeTreeAux (in, root, false);
    }
}
void
FISRuleTree::serializeTreeAux (ostream & out, RuleNode * node)
{
  serializeNode (out, node);
  if (node->left)
    {
      serializeTreeAux (out, node->left);
    }
  if (node->right)
    {
      serializeTreeAux (out, node->right);
    }
}
```

```
void
FISRuleTree::deSerializeTreeAux (istream & in, RuleNode * node, bool left)
{
  bool p, l, r;
  RuleNode * child = deSerializeNode (in, p, l, r);
  child->parent = node;
  if (left)
    {
      node->left = child;
    }

  else
    {
      node->right = child;
    }
  if (l)
    {
      deSerializeTreeAux (in, child, true);
    }
  if (r)
    {
      deSerializeTreeAux (in, child, false);
    }
}
```

```
////////////////////////////////////////////////////////////////////////////////
// Eric A. Morris
// Univ. of Georgia AI Center
// FISSION Project
//
// FISSION_Utility.cpp
// Implmentation for utility functions (random, matrix, etc)

#include "StdAfx.h"
#include <cmath>
#include <time.h>
#include <iostream>
#include ".\fission_utility.h"
  using namespace std;
double nextNorm = 999;
void
seedRand ()
{
  srand ((unsigned) time (NULL));
} void
seedRand (unsigned int t)
{
  srand (t);
} double
```

```
genRand ()
{
  return rand () * RANDOM_SCALE;
}
int
genRandInt (int lb, int ub)
{
  return (int) (lb + genRand () * (ub - lb));
} double

genNormRand ()
{

    // uses the polar Box-Muller transform (1958)
    if (nextNorm != 999)

    {
      double temp = nextNorm;
      nextNorm = 999;
      return temp / 3.0;
    }
  double x, y, r;

  do

    {
      x = -1 + 2 * genRand ();
      y = -1 + 2 * genRand ();
      r = x * x + y * y;
    }
  while (r == 0 || r > 1);
  r = sqrt (-2 * log (r) / r);
  nextNorm = y * r;
  return (x * r) / 3.0;
}
double *
copyArray (double *src, int n)
{
  double *ret = new double[n];
  for (int i = 0; i < n; i++)
    {
      ret[i] = src[i];
    } return ret;
}
double
rSquared (int n, double *x, double *y)
{
  double xbar = 0;
  double ybar = 0;
  double ssxx = 0;
  double ssyy = 0;
  double ssxy = 0;
  for (int i = 0; i < n; i++)
    {
      xbar += x[i];
      ybar += y[i];
      ssxx += x[i] * x[i];
      ssyy += y[i] * y[i];
      ssxy += x[i] * y[i];
    } xbar /= n;
  ybar /= n;
  ssxx -= n * xbar * xbar;
  ssyy -= n * ybar * ybar;
  ssxy -= n * xbar * ybar;
  return (ssxy * ssxy) / (ssxx * ssyy);
}
void
mult_matrix_vector (double **m, double *v, int r, int c, double *out)
{
  double sum;
  for (int i = 0; i < r; i++)
    {
      sum = 0;
      for (int j = 0; j < c; j++)
    {
      sum += m[i][j] * v[j];
    } out[i] = sum;
} } void
sub_vector_vector (double *v1, double *v2, int n, double *out)
{
  for (int i = 0; i < n; i++)
    {
      out[i] = v1[i] - v2[i];
} } double
norm_vector (double *v, int n)
{
  double sum = 0;
  for (int i = 0; i < n; i++)
    {
      sum += v[i] * v[i];
    } return sqrt (sum);
}
double
min3 (double a, double b, double c)
{
  return min (min (a, b), c);
}
double
max3 (double a, double b, double c)
{
  return max (max (a, b), c);
}




///////////////////////////////////////////////////////////////////////////////
```

```
// Eric A. Morris
// Univ. of Georgia AI Center
// FISSION Project
//
// FISSION-DLL.cpp
// Implmentation of the DLL exported functions

#include "stdafx.h"
#include "FISSION-DLL.h"
#include "sugenofis.h"
#include "fisrule.h"
#include "fission_structures.h"
#include "fission_utility.h"
#include "fission-ea.h"
#include "string.h"
#include <iostream>
#include <fstream>
  using namespace std;
TrainerEA * runningEA = NULL;
FISConfig * theFISConfig = NULL;
EAConfig * theEAConfig = NULL;
VarBounds * varBounds = NULL;
BOOL APIENTRY DllMain (HANDLE hModule, DWORD ul_reason_for_call,
             LPVOID lpReserved )
{
  switch (ul_reason_for_call)

    {
    case DLL_PROCESS_ATTACH:

    //runningEA=NULL;
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:

    /*if(runningEA!=NULL) {
       delete [] runningEA;
       runningEA=NULL;
       } */
    break;
    }
  return TRUE;
}
FISSIONDLL_API void
trainFIS (FISConfig * fc, EAConfig * ec, int n, double *input_marsh,
      double *output_data, int m, double *test_input_marsh,
      double *test_output_data)
{
  seedRand ();
  double **input_data = new double *[n];
  double **test_input_data = new double *[m];
  if (varBounds != NULL)
    delete[]varBounds;
  if (runningEA != NULL)
```

```
    delete runningEA;
  if (theFISConfig != NULL)
    delete theFISConfig;
  if (theEAConfig != NULL)
    delete theEAConfig;
  theFISConfig = new FISConfig (*fc);
  theEAConfig = new EAConfig (*ec);

    // de-marshall the 2d arrays
    for (int i = 0; i < n; i++)
    {
      input_data[i] = new double[theFISConfig->numInputs];
      for (int j = 0; j < theFISConfig->numInputs; j++)
    {
      input_data[i][j] = input_marsh[i * theFISConfig->numInputs + j];
  } } for (int i = 0; i < m; i++)
    {
      test_input_data[i] = new double[theFISConfig->numInputs];
      for (int j = 0; j < theFISConfig->numInputs; j++)
    {
      test_input_data[i][j] =
        test_input_marsh[i * theFISConfig->numInputs + j];
    } }
    // run the GA
    findMaxMins (theFISConfig, n, input_data, output_data, varBounds);
  allocFISMem (n, theFISConfig->numInputs, theFISConfig->maxRules,
         theFISConfig->numInputs * theFISConfig->maxInputMFs);
  runningEA =
    new TrainerEA (theFISConfig, theEAConfig, varBounds, n, input_data,
           output_data, m, test_input_data, test_output_data);
  runningEA->runGA ();
  deallocFISMem (n, theFISConfig->numInputs, theFISConfig->maxRules);

    // clean up
    for (int i = 0; i < n; i++)
    {
      delete[]input_data[i];
  } for (int i = 0; i < m; i++)
    {
      delete[]test_input_data[i];
    } delete[]test_input_data;
  delete[]input_data;
} FISSIONDLL_API void

abortGA ()
{
  runningEA->abort ();
} FISSIONDLL_API int
getInputMFCount (SugenoFIS * sf, int input)
{
  if (sf == NULL)
    return 0;
  return sf->getConfig ()->maxInputMFs;
}
```

```
FISSIONDLL_API double
getInputMFValue (SugenoFIS * sf, int input, int mf, double x)
{
  if (sf == NULL)
    return 0;
  return sf->getMFValue (input * sf->getConfig ()->maxInputMFs + mf, x);
}

FISSIONDLL_API bool usedMF (SugenoFIS * sf, int input, int mf)
{
  if (!sf)
    return false;
  if (sf->getConfig ()->globalInputMFs)
    return sf->getRuleSet ()->usedMF (input * sf->getConfig ()->maxInputMFs +
                       mf);
  return true;
}

FISSIONDLL_API SugenoFIS * getBestFIS ()
{
  if (runningEA)
    return runningEA->getBestFIS ();

  else
    return NULL;
}

FISSIONDLL_API SugenoFIS * getGeneralizeFIS ()
{
  if (runningEA)
    return runningEA->getGeneralizeFIS ();

  else
    return NULL;
}
FISSIONDLL_API int
getNumRules (SugenoFIS * sf)
{
  if (sf == NULL)
    return 0;
  return sf->getRuleSet ()->getNumRules ();
}
FISSIONDLL_API int
getMaxRuleDepth (SugenoFIS * sf)
{
  if (sf == NULL)
    return 0;
  return sf->getRuleSet ()->getMaxRuleDepth ();
}
FISSIONDLL_API char *
getRule (SugenoFIS * sf, int r)
{
  if (sf == NULL)
    return NULL;
  char *ret = new char[1000];
  ret[0] = '\0';
  sf->getRuleSet ()->printTree (ret, r);
  sf->printOutputMF (ret, r);
  return ret;
}
FISSIONDLL_API void
getReport (ReportStruct * rs)
{
  if (runningEA != NULL)
    {
      rs->fitness = runningEA->getBestFitness ();
      rs->MAE = runningEA->getBestMAE ();
      rs->MSE = runningEA->getBestMSE ();
      rs->nEvals = runningEA->getNumEvals ();
      rs->stage = runningEA->getStage ();
    }

  else
    {
      rs->fitness = rs->MAE = rs->MSE = rs->nEvals = rs->stage = 0;
    }
}
FISSIONDLL_API void
getPrediction (double *pred)
{
  if (runningEA)
    runningEA->getCurrentPrediction (pred);
}
FISSIONDLL_API void
applyFIS (SugenoFIS * sf, int n, double *input_marsh, double *output_data,
      double &MSE, double &MAE, double &rSq, double *predVals)
{
  double **input_data = new double *[n];
  if (sf == NULL)
    {
      return;
    }

    // de-marshall the 2d array
    for (int i = 0; i < n; i++)
    {
      input_data[i] = new double[sf->getConfig ()->numInputs];
      for (int j = 0; j < sf->getConfig ()->numInputs; j++)
    {
      input_data[i][j] =
        input_marsh[i * sf->getConfig ()->numInputs + j];
    } } sf->apply (n, input_data, output_data, predVals, MSE, MAE, rSq);
  for (int i = 0; i < n; i++)
    {
      delete[]input_data[i];
    } delete[]input_data;
} FISSIONDLL_API void
applyFISNoOpt (SugenoFIS * sf, int n, double *input_marsh, double *predVals)
```

```
{
  double **input_data = new double *[n];
  if (sf == NULL)
    {
      return;
    }

    // de-marshall the 2d array
    for (int i = 0; i < n; i++)
    {
      input_data[i] = new double[sf->getConfig ()->numInputs];
      for (int j = 0; j < sf->getConfig ()->numInputs; j++)
    {
      input_data[i][j] =
        input_marsh[i * sf->getConfig ()->numInputs + j];
    } } sf->applyNoOutputs (n, input_data, predVals);
  for (int i = 0; i < n; i++)
    {
      delete[]input_data[i];
    } delete[]input_data;
} FISSIONDLL_API char *
genCode (SugenoFIS * sf, const char *fn)
{
  if (sf == NULL)
    return NULL;
  return sf->codeFIS (fn);
}
FISSIONDLL_API void
saveFIS (SugenoFIS * sf, const char *fn)
{
  if (sf == NULL)
    return;
  sf->serializeFIS (fn);
}

FISSIONDLL_API SugenoFIS * loadFIS (const char *fn)
{
  SugenoFIS * sf = new SugenoFIS (fn);
  if (!sf->getRuleSet ())
    return NULL;
  return sf;
}
```

```
//////////////////////////////////////////////////////////////////////////////
// Eric A. Morris
// Univ. of Georgia AI Center
// FISSION Project
//
// FISSION-EA.cpp
// Implmentation for the TrainerEA class
```

```
#include "StdAfx.h"
#include ".\sugenofis.h"
#include ".\fisrule.h"
#include ".\fission_structures.h"
#include ".\fission_utility.h"
#include ".\fission-ea.h"
#include <windows.h>
#include <iostream>
#include <iomanip>
#include <fstream>
  using namespace std;
void
TrainerEA::initPopulation ()
{
  char *str = new char[1000];
  for (int i = 0; i < theEAConfig->popSize; i++)

    {
      if (abortRun)
    return;
      population[i] =
    new SugenoFIS (theFISConfig, varBounds,
               i / ((double) (theEAConfig->popSize)));

    /*int z = population[i]->getRuleSet()->getNumRules();
       for(int k=0; k<z; k++) {
       str[0]='\0';
       population[i]->getRuleSet()->printTree(str, k);
       cout << str << endl << endl;
       cin.get();
       } */
} } double
TrainerEA::evaluateFitness (int i)
{
  double MAE = 0, MSE = 0, R2, MAE2 = 0, MSE2 = 0;
  double fit, efi = 1, var = 0, fit2;
  curSubset = genRandInt (0, theEAConfig->numSubsets);
  if (theFISConfig->linearOutput)
    {
      MSE =
    population[i]->runLinearKalmanFilter (subsetSize, subsets[curSubset],
                          inputData, outputData);
    }

  else
    {
      MSE =
    population[i]->runConstantKalmanFilter (subsetSize,
                        subsets[curSubset], inputData,
                        outputData);
    }
  population[i]->apply (trainingSetSize, inputData, outputData, vals, MSE,
             MAE, R2);
```

```
efi = population[i]->evaluateEfficiency ();

  //cout << MSE << endl;
  fit = (1.0 / (1.0 + MSE));
fit -= (theFISConfig->parsimonyFactor * efi * fit);
if (fit > bestFitness)
  {

  // update the best guy and related info
  WaitForSingleObject (semReportData, INFINITE);
    bestMAE = MAE;
    bestMSE = MSE;
    bestFitness = fit;
    if (bestFIS1 != NULL)
  {
    delete bestFIS1;
    bestFIS1 = NULL;
  }
    bestFIS1 = new SugenoFIS (population[i]);
    for (int q = 0; q < trainingSetSize; q++)
  {
    bestVals1[q] = vals[q];
  } cout << MSE << "   " << numFitEvals << "   " << population[i]->
  getRuleSet ()->getNumRules () << "/" << population[i]->getRuleSet ()->
  getMaxRuleDepth () << endl;
    if (auxSetSize > 0)
  {

      // run it on the test set
      bestFIS1->apply (auxSetSize, auxInputData, auxOutputData, vals2,
                 MSE, MAE, R2);
    fit2 = (1.0 / (1.0 + MSE));
    fit2 -= fit * theFISConfig->parsimonyFactor * efi;
    if (fit2 > bestGeneralizeFitness)
      {
        bestGeneralizeFitness = fit2;
        bestGeneralizeMAE = MAE;
        bestGeneralizeMSE = MSE;
        bestGeneralizeR2 = R2;
        if (bestFIS2 != NULL)
      {
        delete bestFIS2;
        bestFIS2 = NULL;
      }
        bestFIS2 = new SugenoFIS (bestFIS1);
        for (int q = 0; q < auxSetSize; q++)
      {
        bestVals2[q] = vals2[q];
      } cout << " * ";
      }
    cout << MSE << endl;
  }
    ReleaseMutex (semReportData);
  }
  numFitEvals++;
  return fit;
}
void
TrainerEA::copyGens ()
{
  for (int i = 0; i < theEAConfig->popSize; i++)

    {
      delete population[i];
      population[i] = NULL;
      population[i] = nextGen[i];
      nextGen[i] = NULL;
} } void

TrainerEA::breed (SugenoFIS * &child1, SugenoFIS * &child2)
{
  int p1 = doSelect ();
  int p2 = doSelect ();
  child1 = NULL;
  child2 = NULL;
  if (theEAConfig->steadyState || genRand () < theEAConfig->pCross)

    {
      double *c1input = NULL;
      FISRuleTree * c1rules = NULL;
      double *c1output = NULL;
      double *c2input = NULL;
      FISRuleTree * c2rules = NULL;
      double *c2output = NULL;
      if (theFISConfig->globalInputMFs)
    crStage = genRandInt (0, 2);

      else
    crStage = 1;
      if (crStage == 0)

    {
      doCrossover (population[p1]->getInputMFs (),
            population[p2]->getInputMFs (), c1input, c2input,
            inputChrSize);
      c1rules = population[p1]->copyRules ();
      c2rules = population[p2]->copyRules ();
      c1output = population[p1]->copyOutputMFs ();
      c2output = population[p2]->copyOutputMFs ();
      if (genRand () < theEAConfig->pMut)
        doMutate (c1input, inputChrSize);
      if (genRand () < theEAConfig->pMut)
        doMutate (c2input, inputChrSize);
    }

      else if (crStage == 1)

    {
      treeCrossover (population[p1]->getRuleSet (),
              population[p2]->getRuleSet (), c1rules, c2rules);
      c1input = population[p1]->copyInputMFs ();
      c2input = population[p2]->copyInputMFs ();
      c1output = population[p1]->copyOutputMFs ();
      c2output = population[p2]->copyOutputMFs ();
      if (genRand () < theEAConfig->pMut)
        mutateTree (c1rules);
      if (genRand () < theEAConfig->pMut)
        mutateTree (c2rules);
    }

      else
    {
      doCrossover (population[p1]->getOutputMFs (),
            population[p2]->getOutputMFs (), c1output, c2output,
            outputChrSize);
      c1input = population[p1]->copyInputMFs ();
      c2input = population[p2]->copyInputMFs ();
      c1rules = population[p1]->copyRules ();
      c2rules = population[p2]->copyRules ();
      if (genRand () < theEAConfig->pMut)
        doMutate (c1output, outputChrSize);
      if (genRand () < theEAConfig->pMut)
        doMutate (c2output, outputChrSize);
    }
      child1 =
    new SugenoFIS (theFISConfig, varBounds, c1input, c1rules, c1output);
      child2 =
    new SugenoFIS (theFISConfig, varBounds, c2input, c2rules, c2output);

    //cout << population[p1]->getRuleSet()->getDepth() << "   "
           //<< population[p2]->getRuleSet()->getDepth() << endl;
    //cout << child1->getRuleSet()->getDepth() << "   "
           //<< child2->getRuleSet()->getDepth() << endl;
    }

  else

    {
      child1 = new SugenoFIS (population[p1]);
      child2 = new SugenoFIS (population[p2]);
    }
}
int
TrainerEA::doSelect ()
{
  switch (theEAConfig->selType)

    {
    default:
    case SEL_TOURNAMENT:
      return tournamentSelect ();
    case SEL_ROULETTE:
      return rouletteSelect ();
    case SEL_RANK:
      return rankSelect ();
    }
}
int
TrainerEA::rouletteSelect ()
{
  double thresh =
    genRand () * (fitnessSum - theEAConfig->popSize * popMinFitness);
  double sum = 0;
  for (int i = 0; i < theEAConfig->popSize; i++)

    {
      sum += (fitness[i] - popMinFitness);
      if (sum >= thresh)
    return i;
    }
  cout << "bad thresh" << endl;
  return theEAConfig->popSize - 1;
}
int
TrainerEA::selectDeath ()
{
  int ret = 999999;
  if (theEAConfig->elitism)
    {

      do

    {
      ret = genRandInt (0, theEAConfig->popSize);
    }
      while (ret == bestFitnessIndex);
    }

  else
    {
      ret = genRandInt (0, theEAConfig->popSize);
    }
  return ret;
}
int
TrainerEA::rankSelect ()
{
  int ps = theEAConfig->popSize - 1;
  int total = (ps * (ps + 1)) / 2;
  int thresh = genRandInt (0, total);
  int sum = 0;
  for (int i = 0; i < theEAConfig->popSize; i++)

    {
      sum += fitnessRank[i];
      if (sum >= thresh)
    return i;
    }
  return 999999;
}
int
TrainerEA::tournamentSelect ()
{
  double bf = 0;
  int bfi = -1;
  for (int i = 0; i < theEAConfig->tourneySize; i++)

    {
      int cur = (int) (genRand () * theEAConfig->popSize);
      if (bfi < 0 || fitness[cur] > bf)

    {
      bfi = cur;
      bf = fitness[cur];
    }
    }
  return bfi;
}
void
TrainerEA::doCrossover (double *parent1, double *parent2, double *&child1,
            double *&child2, int strLen)
{
  child1 = NULL;
  child2 = NULL;
  switch (theEAConfig->crossType)

    {
    case CROSS_2PT:
      twoPtCrossover (parent1, parent2, child1, child2, strLen);
      break;
    default:
    case CROSS_BLEND:
      blendCrossover (parent1, parent2, child1, child2, strLen);
      break;
    case CROSS_UNIFORM:
      uniformCrossover (parent1, parent2, child1, child2, strLen);
      break;
    case CROSS_ARITHMETIC:
      arithmeticCrossover (parent1, parent2, child1, child2, strLen);
      break;
    }
}
void
TrainerEA::twoPtCrossover (double *parent1, double *parent2, double *&child1,
               double *&child2, int strLen)
{
  child1 = new double[strLen];
  child2 = new double[strLen];
  int point1 = 0, point2 = 0;
  while (point1 == point2)
```

```
    {
      point1 = (int) (genRand () * strLen);
      point2 = (int) (genRand () * strLen);
  } if (point1 > point2)

    {
      int temp = point1;
      point1 = point2;
      point2 = temp;
    }
  for (int i = 0; i < strLen; i++)

    {
      if (i < point1 || i >= point2)

    {
      child1[i] = parent1[i];
      child2[i] = parent2[i];
    }

      else

    {
      child1[i] = parent2[i];
      child2[i] = parent1[i];
    }
    }
}
void
TrainerEA::uniformCrossover (double *parent1, double *parent2,
                 double *&child1, double *&child2, int strLen)
{
  child1 = new double[strLen];
  child2 = new double[strLen];
  for (int i = 0; i < strLen; i++)

    {
      if (genRand () < .5)

    {
      child1[i] = parent1[i];
      child2[i] = parent2[i];
    }

      else

    {
      child1[i] = parent2[i];
      child2[i] = parent1[i];
    }
    }
}
void
```

```
TrainerEA::blendCrossover (double *parent1, double *parent2, double *&child1,
               double *&child2, int strLen)
{
  int start = 0, finish = strLen, temp;
  if (theEAConfig->realParamUniform)
    {
      child1 = new double[strLen];
      child2 = new double[strLen];
      start = 0;
      finish = strLen;
    }

  else
    {
      child1 = copyArray (parent1, strLen);
      child2 = copyArray (parent2, strLen);
      start = genRandInt (0, strLen);
      finish = genRandInt (0, strLen);
      if (start > finish)
    {
      temp = start;
      start = finish;
      finish = temp;
    }
    }
  for (int i = start; i < finish; i++)

    {
      child1[i] = blend (parent1[i], parent2[i], theEAConfig->blendParam);
      child2[i] = blend (parent1[i], parent2[i], theEAConfig->blendParam);
} } void
TrainerEA::arithmeticCrossover (double *parent1, double *parent2,
                double *&child1, double *&child2, int strLen)
{
  int start = 0, finish = strLen, temp;
  if (theEAConfig->realParamUniform)
    {
      child1 = new double[strLen];
      child2 = new double[strLen];
      start = 0;
      finish = strLen;
    }

  else
    {
      child1 = copyArray (parent1, strLen);
      child2 = copyArray (parent2, strLen);
      start = genRandInt (0, strLen);
      finish = genRandInt (0, strLen);
      if (start > finish)
    {
      temp = start;
      start = finish;
      finish = temp;
    }
    }
  for (int i = start; i < finish; i++)

    {
      child1[i] =
    linCombine (parent1[i], parent2[i], theEAConfig->blendParam);
      child2[i] =
    linCombine (parent2[i], parent1[i], theEAConfig->blendParam);
} } double
TrainerEA::gaussNoise (double x, double y)
{
  return x + genNormRand () * abs (y);
}
double
TrainerEA::linCombine (double p1, double p2, double a)
{
  return a * p1 + (1 - a) * p2;
}
double
TrainerEA::blend (double p1, double p2, double a)
{
  if (p1 == p2)

    {
      return p1;
    }
  if (p2 < p1)

    {
      double temp = p1;
      p1 = p2;
      p2 = temp;
    }
  double minimum = p1 - a * (p2 - p1);
  double maximum = p2 + a * (p2 - p1);
  return minimum + genRand () * (maximum - minimum);
}
void
TrainerEA::treeCrossover (FISRuleTree * parent1, FISRuleTree * parent2,
              FISRuleTree * &child1, FISRuleTree * &child2)
{
  parent1->crossover (parent2, child1, child2);
} void
TrainerEA::doMutate (double *child, int strLen)
{
  switch (theEAConfig->mutType)

    {
    default:
    case MUT_GAUSS:
      mutateStringGauss (child, strLen);
      break;
    case MUT_UNIFORM:
      mutateStringUniform (child, strLen);
      break;
    }
}
void
TrainerEA::mutateStringGauss (double *child, int strLen)
{
  int which = (int) (genRand () * strLen);
  int whichInput = which / (PARAM_SIZE * theFISConfig->maxInputMFs);
  double inf = population[0]->getInputMin (whichInput);
  double sup = population[0]->getInputMax (whichInput);
  double range = sup - inf;
  child[which] += genNormRand () * 0.1 * range;
} void
TrainerEA::mutateStringUniform (double *child, int strLen)
{
  int which = (int) (genRand () * strLen);
  int whichInput = which / (PARAM_SIZE * theFISConfig->maxInputMFs);
  double inf = population[0]->getInputMin (whichInput);
  double sup = population[0]->getInputMax (whichInput);
  double range = sup - inf;
  child[which] = -0.1 * range + genRand () * 0.2 * range;
} void

TrainerEA::mutateTree (FISRuleTree * child)
{
  child->mutate ();
} void
TrainerEA::updateRanks (int childIndex, double child, double oldValue)
{
  for (int i = 0; i < theEAConfig->popSize; i++)

    {
      if (i != childIndex)

    {
      if (fitness[i] < child && fitness[i] > oldValue)

        {

        //moving down
        fitnessRank[i]--;
          fitnessRank[childIndex]++;
        }

      else if (fitness[i] > child && fitness[i] < oldValue)

        {

        //moving up
        fitnessRank[i]++;
          fitnessRank[childIndex]--;
        }
    }
    }
}
void
TrainerEA::steadyStatePostBreed (SugenoFIS * child)
{
  int ind = selectDeath ();
  double old = fitness[ind];
  delete population[ind];
  population[ind] = NULL;
  population[ind] = child;
  double fit = evaluateFitness (ind);
  fitness[ind] = fit;
  fitnessSum -= old;
  fitnessSum += fit;
  if (fit > popMaxFitness)
    {
      popMaxFitness = fit;
      bestFitnessIndex = ind;
    }
  if (fit < popMinFitness)
    popMinFitness = fit;
  if (GEN_OUTPUT && (numFitEvals % theEAConfig->popSize == 0))
    {
      this->writeStatus ();
    }
  if (theEAConfig->selType == SEL_RANK)
    updateRanks (ind, fit, old);
}
void
TrainerEA::generationalPostBreed ()
{
  fitnessSum = 0;
  popMaxFitness = -999999;
  popMinFitness = 999999;
  bestFitnessIndex = -1;
  double fit;
  for (int i = 0; i < theEAConfig->popSize; i++)

    {
      if (abortRun)
    return;
      fit = evaluateFitness (i);
      fitness[i] = fit;
      fitnessSum += fit;
      if (fit > popMaxFitness)
    {
      popMaxFitness = fit;
      bestFitnessIndex = i;
    }
      if (fit < popMinFitness)
    popMinFitness = fit;
    }
  if (GEN_OUTPUT)
    {
      this->writeStatus ();
    }
  if (theEAConfig->selType == SEL_RANK)

    {
      int *temp = new int[theEAConfig->popSize];
      for (int i = 0; i < theEAConfig->popSize; i++)

    {
      if (abortRun)
        return;
      fitnessRank[i] = i;
      temp[i] = i;
    }
      for (int i = 0; i < theEAConfig->popSize; i++)

    {
      for (int j = i + 1; j < theEAConfig->popSize; j++)

        {
          if (fitness[temp[i]] > fitness[temp[j]])

        {
          if (abortRun)
            return;
          int t = temp[i];
          temp[i] = temp[j];
          fitnessRank[temp[j]] = i;
          temp[j] = t;
          fitnessRank[t] = j;
        }
    } } delete[]temp;
      temp = NULL;

    }
} void

TrainerEA::runSteadyState ()
{
  SugenoFIS * child1;
  SugenoFIS * child2;
  crStage = 0;
  int nBreed = theEAConfig->popSize * theEAConfig->maxGens;
  int nBreedOver2 = nBreed / 2;
  for (int i = 0; i < nBreedOver2; i++)

    {
      if (abortRun)
    return;
      breed (child1, child2);
      steadyStatePostBreed (child1);
      steadyStatePostBreed (child2);
    }
}
```

```
void
TrainerEA::runGenerational ()
{
  int genStart = 0;
  int nBreed = theEAConfig->popSize * theEAConfig->maxGens;
  for (int k = 0; k < theEAConfig->maxGens; k++)

    {
      cout << "Average Fitness: " << fitnessSum /
    theEAConfig->popSize << endl;
      cout << "Starting Generation" << endl;
      if (theEAConfig->elitism)
    {
      nextGen[0] = new SugenoFIS (population[bestFitnessIndex]);
      nextGen[1] = new SugenoFIS (population[bestFitnessIndex]);
      genStart = 2;
    }

      else
    {
      genStart = 0;
    }
      for (int i = genStart; i < theEAConfig->popSize; i += 2)

    {
      if (abortRun)
        return;
      breed (nextGen[i], nextGen[i + 1]);
    }
      copyGens ();
      cout << "Evaluating generation..." << endl;
      generationalPostBreed ();
    }
}

TrainerEA::TrainerEA (FISConfig * fc, EAConfig * ec, VarBounds * vb, int n,
            double **id, double *od, int n1, double **aid,
            double *aod)
{

    // set the config structs
    theFISConfig = fc;
  theEAConfig = ec;
  varBounds = vb;

    // set the training data
    trainingSetSize = n;
  inputData = id;
  outputData = od;

    // set the aux data
    auxSetSize = n1;
  auxInputData = aid;
  auxOutputData = aod;
```

```
    // set the best-FIS cache
    bestFIS1 = NULL;
  bestFIS2 = NULL;
  trainingStage = 0;
  semReportData = CreateMutex (NULL, FALSE, NULL);
  inputChrSize =
    theFISConfig->numInputs * theFISConfig->maxInputMFs * PARAM_SIZE;
  outputChrSize =
    theFISConfig->numOutputs * theFISConfig->maxRules *
    (theFISConfig->numInputs + 1);
  population = new SugenoFIS *[theEAConfig->popSize];
  nextGen = new SugenoFIS *[theEAConfig->popSize];
  fitness = new double[theEAConfig->popSize];
  fitnessRank = new int[theEAConfig->popSize];
  bestFitnessIndex = -1;
  bestFitness = -999999;
  bestGeneralizeFitness = -9999999;
  bestMAE = 9999999;
  bestMSE = 9999999;
  bestGeneralizeMAE = 9999999;
  bestGeneralizeMSE = 9999999;
  bestGeneralizeR2 = 0;
  numFitEvals = 0;
  abortRun = false;
  vals = new double[trainingSetSize];
  vals2 = new double[auxSetSize];
  bestVals1 = new double[trainingSetSize];
  bestVals2 = new double[auxSetSize];
  subsetSize = trainingSetSize / theEAConfig->numSubsets;
  subsets = new int *[theEAConfig->numSubsets];
  fullSet = new int[trainingSetSize];
  for (int i = 0; i < theEAConfig->numSubsets; i++)
    {
      subsets[i] = new int[subsetSize];
    } generateSubsets ();
} void

TrainerEA::generateSubsets ()
{
  bool * used = new bool[trainingSetSize];
  int rndInd;
  for (int i = 0; i < trainingSetSize; i++)
    {
      used[i] = false;
      fullSet[i] = i;
  } for (int i = 0; i < theEAConfig->numSubsets; i++)
    {
      for (int j = 0; j < subsetSize; j++)
    {
      rndInd = genRandInt (0, trainingSetSize);
      while (used[rndInd])
        {
          rndInd++;
          if (rndInd >= trainingSetSize)
        rndInd = 0;
        }
      subsets[i][j] = rndInd;
      used[rndInd] = true;
    }
    }
  delete[]used;
}

TrainerEA::~TrainerEA ()
{
  delete[]population;
  population = NULL;
  delete[]nextGen;
  nextGen = NULL;
  delete[]fitness;
  fitness = NULL;
  delete[]fitnessRank;
  fitnessRank = NULL;
  delete[]vals;
  delete[]vals2;
  vals = NULL;
  vals2 = NULL;
  delete[]bestVals1;
  bestVals1 = NULL;
  delete[]bestVals2;
  bestVals2 = NULL;
  if (bestFIS1 != NULL)
    {
      delete bestFIS1;
      bestFIS1 = NULL;
    }
  if (bestFIS2 != NULL)
    {
      delete bestFIS2;
      bestFIS2 = NULL;
    }
  for (int i = 0; i < theEAConfig->numSubsets; i++)
    {
      delete[]subsets[i];
    } delete[]subsets;
  delete[]fullSet;
} void

TrainerEA::runGA ()
{
  trainingStage = 0;
  startTime = GetTickCount ();
  bestFIS1 = NULL;
  bestFIS2 = NULL;
  printFISConfig (theFISConfig, cout);
  printEAConfig (theEAConfig, cout);
  abortRun = false;
  initPopulation ();
  generationalPostBreed ();
  if (abortRun)
    {
      trainingStage = 3;
      return;
    }
  cout << "Init Complete" << endl;
  trainingStage = 1;
  if (theEAConfig->steadyState)

    {
      runSteadyState ();
    }

  else

    {
      runGenerational ();
    }
  trainingStage = 3;
}
void
TrainerEA::abort ()
{
  abortRun = true;
} int

TrainerEA::getNumEvals ()
{

    //WaitForSingleObject(semReportData,INFINITE);
  int d = numFitEvals;

    //ReleaseMutex(semReportData);
    return d;
}
double
TrainerEA::getBestMSE ()
{
  return bestMSE;
}
double
TrainerEA::getBestMAE ()
{
  return bestMAE;
}
double
TrainerEA::getBestGeneralizeMAE ()
{
  return bestGeneralizeMAE;
}
double
TrainerEA::getBestGeneralizeMSE ()
```

```
{
  return bestGeneralizeMSE;
}
double
TrainerEA::getBestGeneralizeR2 ()
{
  return bestGeneralizeR2;
}
double
TrainerEA::getBestFitness ()
{
  return bestFitness;
}
double
TrainerEA::getBestGeneralizeFitness ()
{
  return bestGeneralizeFitness;
}
void
TrainerEA::getCurrentPrediction (double *pred)
{
  WaitForSingleObject (semReportData, INFINITE);
  for (int i = 0; i < trainingSetSize; i++)
    {
      pred[i] = bestVals1[i];
    } ReleaseMutex (semReportData);
} void
TrainerEA::getGeneralizePrediction (double *pred)
{
  WaitForSingleObject (semReportData, INFINITE);
  for (int i = 0; i < auxSetSize; i++)
    {
      pred[i] = bestVals2[i];
    } ReleaseMutex (semReportData);
} SugenoFIS * TrainerEA::getBestFIS ()
{
  if (bestFIS1 == NULL)
    return NULL;
  SugenoFIS * ret = NULL;
  WaitForSingleObject (semReportData, INFINITE);
  ret = new SugenoFIS (bestFIS1);
  ReleaseMutex (semReportData);
  return ret;
}

SugenoFIS * TrainerEA::getGeneralizeFIS ()
{
  if (bestFIS2 == NULL)
    return NULL;
  SugenoFIS * ret = NULL;
  WaitForSingleObject (semReportData, INFINITE);
  ret = new SugenoFIS (bestFIS2);
  ReleaseMutex (semReportData);
  return ret;
}
int
TrainerEA::getStage ()
{
  return trainingStage;
}
void
TrainerEA::writeStatus ()
{
  ofstream out (EA_OUT_FILE, ios_base::app | ios_base::ate);
  out << this->numFitEvals << ", " << (GetTickCount () -
                    this->startTime) << ", " << this->
    popMaxFitness << ", " << (this->fitnessSum /
                  theEAConfig->popSize) << ", " << this->
    popMinFitness << ", " << this->bestMAE << ", " << sqrt (this->
                              bestMSE) << ", "
    << this->bestFIS1->getRuleSet ()->getMaxRuleDepth () << ", " << this->
    bestFIS1->getRuleSet ()->getNumRules () << endl;
  out.close ();
}




///////////////////////////////////////////////////////////////////////////////
// Eric A. Morris
// Univ. of Georgia AI Center
// FISSION Project
//
// SugenoFIS.cpp
// Implmentation for the SugenoFIS class

#include "StdAfx.h"
#include ".\sugenofis.h"
#include ".\fisrule.h"
#include ".\fission_structures.h"
#include ".\fission_utility.h"
#include <windows.h>
#include <fstream>
#include <sstream>
extern "C"
{

#include "standard.h"
}
#define LAMBDA 1
#define TOLERANCE .00001
 using namespace std;
double *kdp;
double **firingLevels;
double **matrix;
double **derivMatrix;
double *derivVector;
double *mfVals;
```

```
bool * leftOfCenter;
double *vector;
double *temp1;
double *temp2;
int *indexList;
double *scaleList;
void
allocFISMem (int N, int M, int S, int numInputMFs)
{
  int SM = S * (M + 1);
  kdp = new double[SM + 1];
  firingLevels = new double *[N];
  matrix = new double *[N];
  for (int i = 0; i < N; i++)
    {
      firingLevels[i] = new double[S];
      matrix[i] = new double[SM];
    } derivMatrix = new double *[SM];
  for (int i = 0; i < SM; i++)
    {
      derivMatrix[i] = new double[SM];
    } derivVector = new double[SM];
  mfVals = new double[numInputMFs];
  leftOfCenter = new bool[numInputMFs];
  vector = new double[N];
  temp1 = new double[N];
  temp2 = new double[N];
  indexList = new int[SM];
  scaleList = new double[SM];
  initialize_kalman (SM, 1);
} void
deallocFISMem (int N, int M, int S)
{
  int SM = S * (M + 1);
  for (int i = 0; i < N; i++)
    {
      delete[]firingLevels[i];
      delete[]matrix[i];
  } for (int i = 0; i < SM; i++)
    {
      delete[]derivMatrix[i];
    } delete[]mfVals;
  delete[]leftOfCenter;
  delete[]firingLevels;
  delete[]vector;
  delete[]matrix;
  delete[]derivMatrix;
  delete[]temp1;
  delete[]temp2;
  delete[]derivVector;
  delete[]indexList;
  delete[]kdp;
  delete[]scaleList;
} SugenoFIS::SugenoFIS (FISConfig * conf, VarBounds * vb, double pos)
```

```
{
  theConfig = conf;
  varBounds = vb;
  doCommonInit ();
  inputMFs = new double[inputChrSize];
  outputMFs = new double[outputChrSize];
  infRules = new FISRuleTree (theConfig, pos);
  if (theConfig->globalInputMFs)
    {
      generateRandomInputMFs ();
    }
}
SugenoFIS::SugenoFIS (FISConfig * conf, VarBounds * vb, double *inmf,
            FISRuleTree * ft)
{
  theConfig = conf;
  varBounds = vb;
  doCommonInit ();
  infRules = ft;
  inputMFs = inmf;
  outputMFs = new double[outputChrSize];
} SugenoFIS::SugenoFIS (FISConfig * conf, VarBounds * vb, double *inmf,
               FISRuleTree * ft, double *omf)
{
  theConfig = conf;
  varBounds = vb;
  doCommonInit ();
  infRules = ft;
  inputMFs = inmf;
  outputMFs = omf;
} SugenoFIS::SugenoFIS (const char *fn)
{
  ifstream in (fn, ios::binary | ios::in);
  infRules = NULL;
  if (in.good ())
    {
      theConfig = new FISConfig;
      in.read ((char *) theConfig, sizeof (FISConfig));
      in.read ((char *) &inputChrSize, sizeof (int));
      in.read ((char *) &outputChrSize, sizeof (int));
      varBounds = new VarBounds[theConfig->numInputs + 1];
      inputMFs = new double[inputChrSize];
      outputMFs = new double[outputChrSize];
      for (int i = 0; i < (theConfig->numInputs + 1); i++)
    {
      in.read ((char *) &varBounds[i], sizeof (VarBounds));
    }
      for (int i = 0; i < inputChrSize; i++)
    {
      in.read ((char *) &inputMFs[i], sizeof (double));
      } for (int i = 0; i < outputChrSize; i++)
    {
      in.read ((char *) &outputMFs[i], sizeof (double));
    } infRules = new FISRuleTree (theConfig, in);
```

```
      in.close ();
    }

  else
    {
      cout << "Error: could not open file." << endl;
    }
}

SugenoFIS::SugenoFIS (SugenoFIS * sf)
{
  theConfig = sf->theConfig;
  varBounds = sf->varBounds;
  doCommonInit ();
  inputMFs = sf->copyInputMFs ();
  outputMFs = sf->copyOutputMFs ();
  infRules = new FISRuleTree (sf->infRules);
}
void
SugenoFIS::doCommonInit ()
{
  int numInputMFs = theConfig->numInputs * theConfig->maxInputMFs;
  int numOutputMFs = theConfig->numOutputs * theConfig->maxRules;
  inputChrSize = numInputMFs * PARAM_SIZE;
  outputChrSize = numOutputMFs * (theConfig->numInputs + 1);
} SugenoFIS::~SugenoFIS (void)
{
  delete infRules;
  infRules = NULL;
  delete[]outputMFs;
  outputMFs = NULL;
  delete[]inputMFs;
  inputMFs = NULL;
} double *

SugenoFIS::getOutputMFs ()
{
  return outputMFs;
}
double *
SugenoFIS::copyInputMFs ()
{
  double *ret = new double[inputChrSize];
  for (int i = 0; i < inputChrSize; i++)
    ret[i] = inputMFs[i];
  return ret;
}
double *
SugenoFIS::copyOutputMFs ()
{
  double *ret = new double[outputChrSize];
  for (int i = 0; i < outputChrSize; i++)
    ret[i] = outputMFs[i];
  return ret;
```

```
}
void
SugenoFIS::generateRandomInputMFs ()
{

    //double maximum, minimum;
  int iptNum;
  double range;
  double spacing;
  double width;
  int mfNum;
  for (int i = 0; i < (theConfig->numInputs * theConfig->maxInputMFs); i++)
    {
      iptNum = i / theConfig->maxInputMFs;
      mfNum = i % theConfig->maxInputMFs;
      range = varBounds[iptNum].maximum - varBounds[iptNum].minimum;
      spacing = range / (theConfig->maxInputMFs - 1);
      width = range / (theConfig->maxInputMFs);
      for (int j = 0; j < PARAM_SIZE; j++)
    {
      if (j == 0)
        {
          inputMFs[i * PARAM_SIZE + j] =
        varBounds[iptNum].minimum + mfNum * spacing;
          inputMFs[i * PARAM_SIZE + j] +=
        -0.1 * range + genRand () * 0.2 * range;
        }

      else
        {
          inputMFs[i * PARAM_SIZE + j] = width;
          inputMFs[i * PARAM_SIZE + j] +=
        -0.1 * range + genRand () * 0.2 * range;
        }
    }
    }
}
void
SugenoFIS::generateRandomOutputMFs ()
{

    //double maximum, minimum;
    for (int i = 0; i < outputChrSize; i++)
    {
      outputMFs[i] = -1000 + genRand () * 2000;
} } double
SugenoFIS::getMFValue (int mf, double x)
{
  switch (theConfig->mfType)

    {
    default:
    case MF_GAUSS:
      return gaussmf (mf, x);
    case MF_GBELL:
      return gbellmf (mf, x);
    case MF_TRI:
      return trimf (mf, x);
    case MF_TRAP:
      return trapmf (mf, x);
    }
}
double
SugenoFIS::gaussmf (int mf, double x)
{
  int si = PARAM_SIZE * mf;
  double mean = inputMFs[si];
  double stdev = inputMFs[si + 1] / 2;
  double exp = x - mean;
  exp *= (exp * -1);
  exp /= (2 * stdev * stdev);
  return pow (E, exp);
}
double
SugenoFIS::gbellmf (int mf, double x)
{
  int si = PARAM_SIZE * mf;
  double mean1 = inputMFs[si];
  double mean2 = mean1 + inputMFs[si + 1] / 2;
  double stdev1 = inputMFs[si + 2] / 2;
  double stdev2 = inputMFs[si + 3] / 2;
  double exp;
  if (x >= mean1 && x <= mean2)
    {
      return 1;
    }

  else if (x < mean1)
    {
      exp = x - mean1;
      exp *= (exp * -1);
      exp /= (2 * stdev1 * stdev1);
      return pow (E, exp);
    }

  else
    {
      exp = x - mean2;
      exp *= (exp * -1);
      exp /= (2 * stdev2 * stdev2);
      return pow (E, exp);
    }
}
double
SugenoFIS::trimf (int mf, double x)
{
  int si = PARAM_SIZE * mf;
  double center = inputMFs[si];
  double left = center - inputMFs[si + 1];
  double right = center + inputMFs[si + 2];
  return
    max (min ((x - left) / (center - left), (right - x) / (right - center)),
     0);
}
double
SugenoFIS::trapmf (int mf, double x)
{
  int si = PARAM_SIZE * mf;
  double center1 = inputMFs[si];
  double center2 = center1 + inputMFs[si + 1];
  double left = center1 - inputMFs[si + 2];
  double right = center2 + inputMFs[si + 3];
  return
    max (min3
     ((x - left) / (center1 - left), 1, (right - x) / (right - center2)),
     0);
}
void
SugenoFIS::getFiringLevels (int N, double **x, double **&f)
{
  int M = theConfig->numInputs;
  int S;
  int ind;
  int nIMF = theConfig->numInputs * theConfig->maxInputMFs;
  double fSum;
  double *mfv = new double[nIMF];
  bool * loc = new bool[nIMF];

    //compute the firing levels
    S = infRules->getNumRules ();
  f = new double *[N];
  for (int i = 0; i < N; i++)
    {

    //build MF table for this input
    if (theConfig->globalInputMFs)
    {
      for (int v = 0; v < theConfig->numInputs; v++)

        {
          for (int w = 0; w < theConfig->maxInputMFs; w++)
        {
          ind = v * theConfig->maxInputMFs + w;
          mfv[ind] = getMFValue (ind, x[i][v]);
          loc[ind] = (x[i][v] < getMFCenter (ind));
    } } }

    // find the firing values for this input
    f[i] = new double[S];
      fSum = 0;
      for (int r = 0; r < S; r++)
    {
      f[i][r] = infRules->getFiringLevel (r, mfv, loc);
      fSum += f[i][r];
    } }
    //cout << fSum << endl;
    delete[]mfv;
  delete[]loc;
} void
SugenoFIS::removeDependencies (int N, double **x, double **&f)
{
  double S = infRules->getNumRules ();
  bool dep, anyDependence;
  double factor;
  int rndPick;
  bool * toRemove = new bool[theConfig->maxRules];

    //check for linear dependence...
    anyDependence = false;
  for (int h = 0; h < S; h++)
    {
      toRemove[h] = false;
  } for (int r1 = 0; r1 < S; r1++)
    {
      for (int r2 = r1 + 1; r2 < S; r2++)
    {
      dep = true;
      if (f[0][r2] == 0)
        factor = 0;

      else
        factor = f[0][r1] / f[0][r2];
      for (int i = 0; (i < 20 && dep); i++)
        {
          rndPick = genRandInt (0, N);
          if (abs (f[rndPick][r1] - f[rndPick][r2] * factor) > TOLERANCE)
        dep = false;
        }
      if (dep)
        {

        //cout << "Dep Found: " << r2 << endl;
        toRemove[r2] = true;
          anyDependence = true;
        }
    }
    }
  if (anyDependence)
    {

    // delete the computed firing levels;
    // they will be replaced
    for (int i = 0; i < N; i++)
    {
      delete[]f[i];
      f[i] = NULL;
    } delete[]f;
      f = NULL;

    /*cout << "---------------" << endl;
       for(int q=0; q<S; q++) {
       cout << toRemove[q] << endl;
       } */

    // delete redundant rules
    infRules->removeRules (toRemove);

    // replace the firing levels
    getFiringLevels (N, x, f);
    }
  delete[]toRemove;
} void
SugenoFIS::apply (int N, double **x, double *y, double *outputVals,
          double &MSE, double &MAE, double &rSq)
{
  double **firL;
  double firingSum, productSum, sqErrorSum, abErrorSum, zval, error;
  int S = infRules->getNumRules ();
  getFiringLevels (N, x, firL);
  sqErrorSum = 0;
  abErrorSum = 0;
  for (int i = 0; i < N; i++)
    {
      firingSum = 0;
      productSum = 0;
      for (int r = 0; r < S; r++)
    {
      zval = applyOutputMF (r, theConfig->linearOutput, x[i]);
      firingSum += firL[i][r];
      productSum += firL[i][r] * zval;
      } if (theConfig->wtaver)
    {
      if (firingSum == 0)
        outputVals[i] = 0;

      else
        outputVals[i] = productSum / firingSum;
    }

      else
    {
      outputVals[i] = productSum;
    }
      error = outputVals[i] - y[i];
      sqErrorSum += error * error;
      abErrorSum += abs (error);
    }
  MSE = sqErrorSum / N;
  MAE = abErrorSum / N;
  rSq = rSquared (N, outputVals, y);
```

```
    /*for(int i=0; i<this->outputChrSize; i++) {
       cout << outputMFs[i] << endl;
       }
       cin.get(); */
    for (int i = 0; i < N; i++)
    {
      delete[]firL[i];
      firL[i] = NULL;
    } delete[]firL;
  firL = NULL;
} void
SugenoFIS::applyNoOutputs (int N, double **x, double *outputVals)
{
  double **firL;
  double firingSum, productSum, zval;
  int S = infRules->getNumRules ();
  getFiringLevels (N, x, firL);
  for (int i = 0; i < N; i++)
    {
      firingSum = 0;
      productSum = 0;
      for (int r = 0; r < S; r++)
    {
      zval = applyOutputMF (r, theConfig->linearOutput, x[i]);
      firingSum += firL[i][r];
      productSum += firL[i][r] * zval;
      } if (theConfig->wtaver)
    {
      if (firingSum == 0)
        outputVals[i] = 0;

      else
        outputVals[i] = productSum / firingSum;
    }

      else
    {
      outputVals[i] = productSum;
    }
    }
  for (int i = 0; i < N; i++)
    {
      delete[]firL[i];
      firL[i] = NULL;
    } delete[]firL;
  firL = NULL;
} double

SugenoFIS::evaluateEfficiency ()
{
  int depth = infRules->getMaxRuleDepth ();
  return depth / theConfig->maxOverallRuleDepth;
}
```

```
double
SugenoFIS::applyOutputMF (int which, bool useLinear, double *x)
{
  int st;
  double sum;
  if (useLinear)
    {
      st = which * (theConfig->numInputs + 1);
      sum = outputMFs[st];
      for (int i = 0; i < theConfig->numInputs; i++)
    {
      sum += x[i] * outputMFs[st + i + 1];
    } return sum;
    }

  else
    {
      return outputMFs[which];
    }
}

bool loadDataFromFile (int N, int M, const char *fn, double **&x,
              double *&y)
{
  y = new double[N];
  x = new double *[N];
  char trash;
  ifstream in;
  in.open (fn);
  if (!in)
    {
      return false;
    }
  for (int i = 0; i < N; i++)
    {
      x[i] = new double[M];
      for (int j = 0; j < M; j++)
    {
      in >> x[i][j];
      in >> trash;
    } in >> y[i];
    } in.close ();
  return true;
}
void
findMaxMins (FISConfig * conf, int N, double **inputs, double *outputs,
         VarBounds * &bounds)
{
  bounds = new VarBounds[conf->numInputs + 1];

    //cout << "here" << endl;
    //determine maxes and mins
    for (int j = 0; j < conf->numInputs; j++)
    {
      bounds[j].minimum = inputs[0][j];
      bounds[j].maximum = inputs[0][j];
      for (int i = 0; i < N; i++)
    {
      if (inputs[i][j] > bounds[j].maximum)
        {
          bounds[j].maximum = inputs[i][j];
        }
      if (inputs[i][j] < bounds[j].minimum)
        {
          bounds[j].minimum = inputs[i][j];
        }
    }
    }
  bounds[conf->numInputs].minimum = outputs[0];
  bounds[conf->numInputs].maximum = outputs[0];
  for (int i = 0; i < N; i++)
    {
      if (outputs[i] > bounds[conf->numInputs].maximum)
    {
      bounds[conf->numInputs].maximum = outputs[i];
    }
      if (outputs[i] < bounds[conf->numInputs].minimum)
    {
      bounds[conf->numInputs].minimum = outputs[i];
    }
    }
}
void
scaleInputs (FISConfig * conf, int N, double *maxes, double *mins,
         double **inputs, double **&scaledInputs)
{
  double minimum, maximum;
  scaledInputs = new double *[N];

    //scale the inputs
    for (int i = 0; i < N; i++)
    {
      scaledInputs[i] = new double[conf->numInputs];
      for (int j = 0; j < conf->numInputs; j++)
    {
      minimum = mins[j];
      maximum = maxes[j];
      if (maximum - minimum == 0)
        scaledInputs[i][j] = .5;

      else
        scaledInputs[i][j] =
          (inputs[i][j] - minimum) / (maximum - minimum);
    }
    }
}
double *
SugenoFIS::getInputMFs ()
```

```
{
  return inputMFs;
}

FISRuleTree * SugenoFIS::getRuleSet ()
{
  return infRules;
}

FISRuleTree * SugenoFIS::copyRules ()
{
  FISRuleTree * ret = new FISRuleTree (infRules);
  return ret;
}
void
SugenoFIS::printOutputMF (char *str, int mf)
{
  char *temp = new char[100];
  if (theConfig->linearOutput)
    {
      int st = mf * (theConfig->numInputs + 1);
      sprintf (temp, "      Output = %f", outputMFs[st]);
      strcat (str, temp);
      for (int i = 1; i < theConfig->numInputs + 1; i++)
    {
      sprintf (temp, " + input%d * %f", i, outputMFs[st + i]);
      strcat (str, temp);
    } }

  else
    {
      sprintf (temp, "      Output = %f", outputMFs[mf]);
      strcat (str, temp);
    }
  strcat (str, "\r\n   )\r\n)");
  delete[]temp;
}
double
SugenoFIS::runLinearKalmanFilter (int N, int *set, double **x, double *y)
{
  int M = theConfig->numInputs;
  int S = infRules->getNumRules ();;
  int ind;
  int SM = S * (M + 1);
  int baseInd;
  double fSum;
  double error;
  reset_kalman (SM, 1);
  for (int i = 0; i < N; i++)
    {
      fSum = 0;

    //build MF table for this input
    if (theConfig->globalInputMFs)
    {
      for (int v = 0; v < theConfig->numInputs; v++)

        {
          for (int w = 0; w < theConfig->maxInputMFs; w++)
        {
          ind = v * theConfig->maxInputMFs + w;
          mfVals[ind] = getMFValue (ind, x[set[i]][v]);
          leftOfCenter[ind] = (x[set[i]][v] < getMFCenter (ind));
    } } }

    // find the firing values for this input
    for (int r = 0; r < S; r++)
    {
      firingLevels[i][r] =
        infRules->getFiringLevel (r, mfVals, leftOfCenter);
      fSum += firingLevels[i][r];
    }
    // build the input matrix for regression or kalman
    for (int r = 0; r < S; r++)
    {
      baseInd = r * (M + 1);
      if (theConfig->wtaver && fSum != 0)
        firingLevels[i][r] /= fSum;
      kdp[baseInd] = firingLevels[i][r];
      matrix[i][baseInd] = kdp[baseInd];
      for (int j = 1; j < M + 1; j++)
        {
          kdp[baseInd + j] = firingLevels[i][r] * x[set[i]][j - 1];
          matrix[i][baseInd + j] = kdp[baseInd + j];
    } } kdp[SM] = y[set[i]];
      vector[i] = y[set[i]];
      new_kalman (SM, 1, kdp, outputMFs, LAMBDA);
    }
    //cout << fSum << endl;
    /*for(int i=0; i<SM; i++) {
       cout << outputMFs[i] << endl;
       } */
    //cin.get();
    mult_matrix_vector (matrix, outputMFs, N, SM, temp1);
  sub_vector_vector (temp1, vector, N, temp2);
  error = norm_vector (temp2, N);
  return error / sqrt ((double) N);
} double
SugenoFIS::runConstantKalmanFilter (int N, int *set, double **x, double *y)
{
  int S = infRules->getNumRules ();;
  int ind;
  double fSum;
  double error;
  reset_kalman (S, 1);
  for (int i = 0; i < N; i++)
    {
      fSum = 0;
    //build MF table for this input
    if (theConfig->globalInputMFs)
    {
      for (int v = 0; v < theConfig->numInputs; v++)

        {
          for (int w = 0; w < theConfig->maxInputMFs; w++)
        {
          ind = v * theConfig->maxInputMFs + w;
          mfVals[ind] = getMFValue (ind, x[set[i]][v]);
          leftOfCenter[ind] = (x[set[i]][v] < getMFCenter (ind));
    } } }

    // find the firing values for this input
    for (int r = 0; r < S; r++)
    {
      firingLevels[i][r] =
        infRules->getFiringLevel (r, mfVals, leftOfCenter);
      fSum += firingLevels[i][r];
    }
    // build the input matrix for regression or kalman
    for (int r = 0; r < S; r++)
    {
      if (theConfig->wtaver && fSum != 0)
        firingLevels[i][r] /= fSum;
      kdp[r] = firingLevels[i][r];
      matrix[i][r] = kdp[r];
    }
      kdp[S] = y[set[i]];
      vector[i] = y[set[i]];
      new_kalman (S, 1, kdp, outputMFs, LAMBDA);
    }

    //cout << fSum << endl;
    /*for(int i=0; i<SM; i++) {
       cout << outputMFs[i] << endl;
       } */
    //cin.get();
    mult_matrix_vector (matrix, outputMFs, N, S, temp1);
  sub_vector_vector (temp1, vector, N, temp2);
  error = norm_vector (temp2, N);
  return error / sqrt ((double) N);
} FISConfig * SugenoFIS::getConfig ()
{
  return theConfig;
}
void
SugenoFIS::getMFsForInput (int input, double *MFs)
{
  int start = theConfig->maxInputMFs * PARAM_SIZE * input;
  int end = theConfig->maxInitialRuleDepth * PARAM_SIZE * (input + 1);
  for (int i = start; i < end; i++)
    {
      MFs[i - start] = inputMFs[i];
} } double
SugenoFIS::getInputMin (int input)
{
  return varBounds[input].minimum;
}
double
SugenoFIS::getInputMax (int input)
{
  return varBounds[input].maximum;
}
double
SugenoFIS::getMFCenter (int mf)
{
  return inputMFs[mf * PARAM_SIZE];
}
void
SugenoFIS::writeInputMFs (ostream & out)
{
  out << "{";
  for (int i = 0; i < inputChrSize; i++)
    {
      if (i > 0)
    out << ",";
      out << inputMFs[i];
    }
  out << "}";
}
void
SugenoFIS::writeOutputMFs (ostream & out)
{
  out << "{";
  for (int i = 0; i < outputChrSize; i++)
    {
      if (i > 0)
    out << ",";
      out << outputMFs[i];
    }
  out << "}";
}
char *
SugenoFIS::codeFIS (const char *fn)
{
  char *ret = new char[10000];
  char macro[20];
  char ch;
  int count;
  int nr = infRules->getNumRules ();
  ret[0] = '\0';
  cout << fn << endl;
  ostringstream oss (ostringstream::out);
  ifstream in (fn);
  if (!in.good ())
    cout << "error in file." << endl;
```

```
while (in.good ())
  {
    in.get (ch);
    if (ch == '$')
  {
    count = 0;
    ch = '\0';
    while (in.good () && ch != '$')
      {
        in.get (ch);
        if (ch != '$')
      {
        macro[count] = ch;
        count++;
      }
      }
    macro[count] = '\0';

      //cout << macro << endl;
      if (!strcmp (macro, "N_INPUTS"))
      {
        oss << theConfig->numInputs;
      }

    else if (!strcmp (macro, "WTAVER"))
      {
        oss << theConfig->wtaver;
      }

    else if (!strcmp (macro, "PROD"))
      {
        oss << theConfig->prod;
      }

    else if (!strcmp (macro, "PROBOR"))
      {
        oss << theConfig->probor;
      }

    else if (!strcmp (macro, "LINEAR"))
      {
        oss << theConfig->linearOutput;
      }

    else if (!strcmp (macro, "N_RULES"))
      {
        oss << nr;
      }

    else if (!strcmp (macro, "N_MFPI"))
      {
        oss << theConfig->maxInputMFs;
      }
      else if (!strcmp (macro, "MF_TYPE"))
        {
          oss << theConfig->mfType;
        }

      else if (!strcmp (macro, "INPUT_MFS"))
        {
          writeInputMFs (oss);
        }

      else if (!strcmp (macro, "OUTPUT_MFS"))
        {
          writeOutputMFs (oss);
        }

      else if (!strcmp (macro, "THE_RULES"))
        {
          oss << "switch(rule) {\r\n";
          for (int i = 0; i < nr; i++)
        {
          oss << "\t\tcase " << i << ":\r\n\t\t\treturn ";
          infRules->writeRule (oss, i);
          oss << ";\r\n";
        } oss << "\t}";
        }

      else
        {
          cout << "ERROR : Unrecognized tag." << endl;
        }
    }

      else if (ch == '\n')
    {
      oss << "\r\n";
    }

      else
    {
      oss << ch;
    }
    }
  in.close ();
  strcpy (ret, oss.str ().c_str ());
  return ret;
}
void
SugenoFIS::serializeFIS (const char *fn)
{
  ofstream out (fn, ios::out | ios::trunc | ios::binary);
  if (out.good ())
    {
      out.write ((char *) theConfig, sizeof (FISConfig));
      out.write ((char *) &inputChrSize, sizeof (int));
      out.write ((char *) &outputChrSize, sizeof (int));
      for (int i = 0; i < (theConfig->numInputs + 1); i++)
    {
      out.write ((char *) &varBounds[i], sizeof (VarBounds));
    }
      for (int i = 0; i < inputChrSize; i++)
    {
      out.write ((char *) &inputMFs[i], sizeof (double));
      } for (int i = 0; i < outputChrSize; i++)
    {
      out.write ((char *) &outputMFs[i], sizeof (double));
    } infRules->serializeTree (out);
      out.close ();
    }

  else
    {
      cout << "Error: could not write output file." << endl;
    }
}

VarBounds * SugenoFIS::getBounds ()
{
  return varBounds;
}
```