Study of a neural network-based system for stability augmentation of an airplane

Author: Roger Isanta Navarro

Annex 3

ANFIS Network Development

Supervisors: Oriol Lizandra Dalmases
Fatiha Nejjar Akhi-Elarab
Aeronautical Engineering
September 2013
This page intentionally left blank
## Contents

1. **Introduction** .............................................................................................................. 1
2. **Basic requirements of the ANFIS code** ...................................................................... 2
3. **Structure** .................................................................................................................... 3
   3.1 ANFIS network structure ......................................................................................... 3
   3.2 Code structure .......................................................................................................... 3
4. **Variations to traditional ANFIS structure and execution** ........................................ 5
5. **Main public procedures within the ANFIS network class** ........................................ 8
   5.1 Constructor ................................................................................................................ 8
   5.2 CreateNN .................................................................................................................. 9
   5.3 InitMF ....................................................................................................................... 9
   5.4 AddTrainingPair ..................................................................................................... 9
   5.5 ClearCPs .................................................................................................................. 10
   5.6 ClearTrData ............................................................................................................ 10
   5.7 Release .................................................................................................................... 10
   5.8 DefineOfflineEpsilon and DefineOnlineEpsilon ..................................................... 10
   5.9 Load and Save ........................................................................................................ 11
   5.10 LoadTrData and SaveTrData ................................................................................. 11
   5.11 PrintInfo ............................................................................................................... 11
   5.12 Calc ........................................................................................................................ 11
   5.13 TrainOffline .......................................................................................................... 12
   5.14 TrainOnline .......................................................................................................... 12
   5.15 PauseOnlineTraining and ResumeOnlineTraining ................................................. 12
   5.16 StopOnlineTraining .............................................................................................. 12
   5.17 OnlineTrainingState ............................................................................................ 12
6. **Output calculation** .................................................................................................... 13
7. **Training algorithms** ................................................................................................. 15
   7.1 Offline training algorithm ....................................................................................... 15
   7.2 Online training algorithm ...................................................................................... 16
| Figure 3.1 | 2-input 4-paths network, example of an all rules combinations structure. ...4 |
| Figure 3.2 | 3-input 3-paths network, example of only corresponding function combination. .................................................................4 |
| Figure 4.1 | Example of partial activation of network paths. ..................................................5 |
| Figure 4.2 | Example of bell shaped membership functions. ..................................................6 |
| Figure 4.3 | Example of linear membership functions. .........................................................6 |
| Figure 6.1 | Flowchart of real time calculation algorithm $Calc$. ...........................................14 |
| Figure 7.1 | Flowchart of offline training algorithm $OfflineTraining$. ..................................16 |
| Figure 7.1 | Flowchart of online training algorithm $OnlineTraining$. .....................................17 |
1 Introduction

This annex concerns the ANFIS Network development as well as other codes that enable the proposed Control System. It has been created in C++ language and compiled using the free distributed compiler from DEV-C++.

The code is presented as a Dynamic Link Library (DLL) for it is the author's intention that the final product use is as much platform independent as possible, i.e. that the system can be used both in an airplane as in a simulation program, such as the case project. Obviously the use of this code is constricted to the initial assumptions and restrictions that have been considered and included in the Report, however the intention of platform independence is desired to be preserved.

The Neural Network is programmed as a C++ class, thus allowing the user to create as many Networks as required, and reducing, in a code using this library, the direct dependence with the Neural Network code.

The code execution performs certain tasks in parallel as the training of the network that runs in a separate thread in order to allow a better response from the network to a request from the simulator, trainer or an airplane, and a faster execution in a multicore processor. The pthreads library from POSIX is used for parallelization [14].

Finally, within this annex, the most relevant functions to the system development are commented. Those functions not directly related to the neural network development or the stability augmentation system proposal, such as, for instance, functions allowing and controlling the parallel execution, will not be discussed, although they can be found in the codes attached to this study.
2 Basic requirements of the ANFIS code

The following are the basic requirements associated with the development of the ANFIS code, defined after the general study of ANFIS Neural Networks, the requisites established within this study and the considerations of one unique output requirement achieved within the general structure definition of the Stability Augmentation System:

- The ANFIS code should be as much independent as possible of the platform (trainer code, simulator or airplane).
- The ANFIS code should be compiled into a Dynamic Link Library.
- The ANFIS code should offer few or none restrictions as for the number of input variables and the number of associated membership functions.
- The ANFIS structure will cover all possible fuzzy rules combinations (see Annex 1 - Introduction to Adaptive Neuro-Fuzzy Inference Systems (ANFIS), Subsection 3.3, Layer 2).
- The ANFIS structure will provide a single output.
- The online training procedures within the ANFIS code should run in parallel threads.
- To increase training and computation speed, the ANFIS structure should allow the choice of membership function amongst lineal and bell-shaped, and the possibility of not training the premise parameters.
- The ANFIS code should allow the loading and saving of the neural network parameters data.
3 Structure

3.1 ANFIS network structure

As commented in Annex 1, the network may consider all possible combinations of input rules (see Figure 3.1) or not (see Figure 3.2) depending, basically, on the possible interrelation between inputs. Although considering all possible combinations results in a network with higher demands in time execution and training, it also reaches better results when there is no clear linear relationship between inputs, the improvement achieved is way more significant in comparison with the growth in complexity and execution time required to obtain the partial derivatives with respect to the premise parameters. Moreover, the presented solution includes some variations to the traditional ANFIS operation which enhance its performance. These variations, along with the consideration of all possible combinations of inputs rules provide, in general, better results.

3.2 Code structure

Since the ANFIS Network code will be provided as class within a Dynamic Link Library (.dll file) there is not a relevant structure comprising the whole of the code, for it will consist of a series of mainly independent procedures that the user may call. These procedures might be grouped in:

- Network runtime control, including processes to create, start, pause, restart and stop the threads execution.
- Parameters definition.
- Calculation request.
- Training request.
- Parameters loading and saving.
Figure 3.1  2-input 4-paths network, example of an all rules combinations structure.

Figure 3.2  3-input 3-paths network, example of only corresponding function combination.
4 Variations to traditional ANFIS structure and execution

During the course of this study, it has become necessary to optimize certain aspects of the functioning of the neural network, for which three variations from the usual performance of ANFIS network we have adopted. However, when using the code provided, these variations are presented as an alternative: the user can choose the mode of operation of the network using a simple definition of parameters.

A lot of the time devoted to the training of the network is being wasted trying to train paths that have an irrelevant specific weight. In the early stages, the network determines the weight of each combination of membership functions. Since the value of the inputs will fall at a specific point of the whole covered space, there will be a large number of paths whose weight is very close to zero and which are traditionally trained.

The first variation is the addition of an activation threshold from which the paths are considered. This allows activating only those paths that will have a significant weight in both the calculation and training processes of the network.

As an example, a network of three inputs with five membership functions for input traditionally results in 125 paths to train for each training pair. Using the proposed variations 125 paths still exist, however, for each training pair only eight of them are activated: those having a significant relative weight.

The second suggested variation is closely related to the first. Partial selection of paths presents a problem: if using bell-shaped membership functions (Figure 4.2) and a high activation threshold, some paths with low relative weight but still of importance, will not...
be taken into account. This results in poor performance of the network, because it loses much of the information that constitutes its output.

To avoid this, the option of using linear membership functions (Figure 4.3) is added: they present a value of 0 in all space except between two specific values where the function increases linearly to reach the value of 1 in the middle. With this change, the membership functions on which the input does not fall do not contribute in weight, and thus there are no paths of small but still relevant weight which are not activated.

![Figure 4.2](image1.png)  
**Figure 4.2**  Example of bell shaped membership functions.

![Figure 4.3](image2.png)  
**Figure 4.3**  Example of linear membership functions.

Finally, the last variation introduced is the possibility of not training the premise parameters. These parameters do not contribute to the output as much as the consequence parameters do, but allow greater adaptation of the network to the
geometry of the function to be approximated. When these parameters are not trained the neural network loses approach capability; however, this can be compensated by increasing the number of membership functions assigned to each input, i.e. the discretization of the space.

The combined use of the proposed variations: the consideration of an activation threshold, the use of linear membership functions and the ability to not train the premise parameters; represents a huge increase in training speed and calculation of the neural network.

In turn, it enables to greatly increase the number of membership functions assigned to the inputs, which in turn allows a better approximation capability of the network.

As an example, a traditional ANFIS network with two inputs and five membership functions (25 paths in total) requires about the same training time that an ANFIS network using the proposed variations with up to 100 membership functions assigned per input (10000 paths in total).
5 Main public procedures within the ANFIS network class

Within the following section the procedures constituting the ANFIS Network code are enumerated along with a short explanation of their function. The class containing the neural network is called NN.

In general, within the code, all variables which belong to the class are preceded with the symbol “_” so as not to confuse with other variables defined within the different functions.

The use of parallel threads always entails the danger of simultaneous access to the same variable. Two safety mechanisms have been used to avoid any problem related to this issue. Firstly, whenever the values of the premise or consequence parameters are to be updated or read, a mutex variable is used (mutex variables work as a key required by a function to enter a section of the code). Secondly, a class called CalcData has been developed which includes all the variable required to perform both the calculation and training of the network. This allows to use two sets of variables: one for calculating with the main thread, and one for the online training using the parallel thread.

5.1 Constructor

This is the function called when a NN variable is created. Its definition follows:

\[
\text{NN} \left(\text{int Inputs, string MF\_Type, bool trainPPs}\right)
\]

This function sets the number of inputs of the network to \(\text{Inputs}\), the membership function type to \(\text{MF\_Type}\) and sets whether to premise parameters should be trained or not according to the value of \(\text{trainPPs}\).

The definition of the bell-shaped parameters is different from linear ones. The later does not allow the training of the premise parameters, therefore, in case of choosing linear membership functions, the function forces the network to not train the premise parameters.

Once these three aspects of the network have been set, the CreateNN function is called which created all the network variables.
5.2 CreateNN

This function initializes all required variables for the network to operate. It is a public function, however it is already called by the constructor. Its definition follows:

\[
\text{void CreateNN(int Inputs)}
\]

within this function the two sets of calculation data are initialized as are the number of membership functions and parameters, according to the properties specified by the user.

This function resets the offline and online values of Epsilon used when training the network.

A call to this function also clears all previously trained consequence parameters and the stored training data.

5.3 InitMF

The call of this function allows defining the number of membership functions associated to an input, as well as the minimum and maximum values of its range. By default, the number of membership functions is set to three, and the minimum and maximum values of the range are set to \(-1\) and \(1\), respectively. The function then sets all initial premise parameters according to the user choices and the membership function type.

The function definition is as follows:

\[
\text{void InitMFs(int Index, string nMFs, double min, double max)}
\]

where, \(\text{Index}\) is the input from which the membership functions are to be modified, \(\text{nMFs}\) s is the new number of membership functions, and \(\text{min}\) and \(\text{max}\) are the minimum and maximum range values.

After the creation of the NN variable, this is the most important function to be called, for it defines the structure of the network.

Any call to this function changes the neural network structure; therefore the consequence parameters are reset.

5.4 AddTrainingPair, DeleteTrainingPair and UpdateTrainingPair

The call to these functions adds a new training pair to the stored training pairs vector, deletes all occurrences of a training pair within the training pairs vectors, or updates them with a different desired output value. Their definitions are:

\[
\text{void AddTrainingPair(double* X, double Yd)}
\]
void DeleteTrainingPair(double* X)
void UpdateTrainingPair(double* X, double Yd)

where X is a vector containing the input values and Yd is the desired output value, of the training pair.

5.5 ClearCPs

The call to this function clears the values of the consequence parameters. This function has no arguments; its definition is:

void ClearCPs()

5.6 ClearTrData

The call to this function clears the stored list of training pairs. This function has no arguments; its definition is:

void ClearTrData()

5.7 Release

The call to this function is mandatory at the very end of the code. This is the last function to be called. It releases all dynamic memory created during the execution of the neural network. Its definition is as follows:

void Release()

This function calls a private function Release_CalcData(CalcData *CD), which releases all dynamic memory associated to the CalcData class.

5.8 DefineOfflineEpsilon and DefineOnlineEpsilon

These functions allow the change of the default value of the offline and online training Epsilons: the maximum allowed change of mean squared error in two consecutive training passes. Their definitions are:

void DefineOfflineEpsilon()
void DefineOnlineEpsilon()
5.9 Load and Save

These two functions allow the user to load and save the data corresponding to the structure of the network and its premise and consequence parameters. These are important functions because the system training may take a long time, however, once the trained neural network is saved, it can be used in multiple simulations. The definitions of these functions are:

```c
void Load(string fileName)
void Save(string fileName)
```

where `fileName` is a string containing the full path to the file, plus its name.

5.10 LoadTrData and SaveTrData

These two functions allow the user to load and save the stored training data. Their definition is:

```c
void LoadTrData(string fileName)
void SaveTrData(string fileName)
```

5.11 PrintInfo

The call to this function prints the most general information of the neural network. It is a really good value function when designing training and simulation algorithms, to check whether the neural network main parameters are correct. Its definition is as follows:

```c
void PrintInfo()
```

5.12 Calc

This is one of the most important functions of the code. It performs the calculation of the neural network output, according to some input values. Its definition is:

```c
double Calc(double *X)
```

where `X` is the vector containing the input values. The function returns a real number corresponding to the output.

The implementation of this function will be discussed in more detail in the following sections.
5.13 TrainOffline

This is also one of the most important functions of the code. It performs the offline training of the network using the training pairs previously. Its definition is:

```c
void TrainOffline()
```

The implementation of the training functions will be discussed in more detail in the following sections.

5.14 TrainOnline

This function performs as the offline version: TrainOffline. The main differences between this function and the previous are the fact that this one is run by a parallel thread and does not stop after the change of mean square error is lower than Epsilon. Its definition is:

```c
void TrainOnline()
```

The implementation of the training functions will be discussed in more detail in the following sections.

5.15 PauseOnlineTraining and ResumeOnlineTraining

These two functions allow to pause and resume the thread on which the online training runs. Their definitions are:

```c
void PauseOnlineTraining()
void ResumeOnlineTraining()
```

5.16 StopOnlineTraining

This function stops the online training and forces the secondary thread to join the main one. Its definition is:

```c
void StopOnlineTraining()
```

5.17 OnlineTrainingState

This function returns a string containing the state of the secondary thread. The possible values are: stopped, running or paused. Its definition is:

```c
string OnlineTrainingState ()
```
6 Output calculation

This section summarizes the conceptual implementation of the function Calc, which computes the output of the network, given a set of inputs.

Two functions Calc exist, a public and a private. The public function accepts only one parameter: an array containing the inputs of the network. The private function also accepts a pointer to the CalcData structure to be used to perform the computations. Their definitions are:

```c
void Calc(double *X)
void Calc(double *X, CalcData *CD)
```

This allows the main thread and the secondary thread to ask for the output of the network to two different input sets simultaneously. The public definition takes the input array, and passes to the second, forcing the CalcData structure to be the one corresponding to the main thread.

The two CalcData structures are called CD_Calc and CD_Train, the first one is used by the main thread, the second one is used by the secondary thread during online training.

The diagram in Figure 6.1 represents the flow of computations required to obtain the network output value. Each block of the algorithm follows the calculation steps defined in Annex 1 - Introduction to Adaptive Neuro-Fuzzy Inference Systems (ANFIS).
where $\mu$, are the membership degrees of every input to all rules, $w$, $\bar{w}$, $f$ and $\bar{wf}$ are the vectors resulting from every calculation step within the ANFIS network as defined in Annex 1, and $y$ is the final output.

After $\mu$ is computed the algorithm checks for values exceeding the activation threshold, and creates all their possible combinations to determine the paths that will be active during this calculation. These active membership functions and paths remain stored after the Calc function ends, because they may be required during the training algorithm.

Figure 6.1 Flowchart of real time calculation algorithm Calc.
7 Training algorithms

7.1 Offline training algorithm

Many combinations exist to perform the training of the network \([X]\). As introduced in Annex 1, the premise parameters may only be trained using an optimization algorithm such as the Steepest Descent Method. The consequence parameters, because the output of the network is linear to them, may be trained using an optimization algorithm such as the Steepest Descent Method or a Least Squares Method.

The consequence parameters in this implementation are also trained using the Steepest Descent Method. The variations introduced to the traditional structure of the ANFIS network (see Section 4) result in the possibility of a high number of membership functions. Inverting the \(A^TA\) matrix corresponding to the Least Squares Method (see Annex 1) in such case would require much more time than performing the Steepest Descent Method also for the consequence parameters.

The conceptual flowchart of the proposed algorithm for offline training is shown in Figure 7.1. The algorithm presents two Steepest Descent Methods, the first optimizing the consequence parameters, the second, the premise parameters. The training of the premise parameters is optional, depending of the user input to the constructor function.

Within the flowchart in Figure 7.1, MSE stands for “Mean Squared Error”. Three changes of this value are evaluated, the change after an iteration over the premise parameters training (MSE\(_{pp}\) change), the change after an iteration over the consequence parameters (MSE\(_{cp}\) change), and the global change after the two optimizations (MSE change).

This algorithm, since it is the offline version of training algorithm, uses the CalcData structure called CD_Calc.
7.2 Online training algorithm

The online training algorithm (Figure 7.2) presents a structure based on the previously shown offline training algorithm. The main differences are the addition of the thread state control (which pauses the thread by putting it to sleep when pause is activated and stops the algorithm at when stop is sent), the elimination of global comparison to Epsilon in order to finish the training execution, and addition of a block which updates the premise and consequence parameters used by the main thread with those recently trained online. The update of the parameters is, in reality, the activation of a switch between two databases of premise and consequence parameters. This allows for the mutex variable to be held as shortly as possible, therefore minimizing the time the threads are blocked.
This algorithm, since it is the online version of training algorithm and is run by the secondary thread, uses the CalcData structure called CD_Train.

Figure 7.2 Flowchart of online training algorithm OnlineTraining.