Optimisation of a Fuzzy Logic Controller Using Genetic Algorithms

M.Eng Project Report
Summer 2002

Student Name: Joseph Foran
Student ID: 51161001

Acknowledgements

I would like to thank my supervisor Ms. Jennifer Bruton for her help, guidance and support throughout the duration of the project.

I would also like to thank my fiancée, family and friends for providing support and the occasional welcome distraction while I worked on this project.

Declaration

I hereby declare that, except where otherwise indicated, this document is entirely my own work and has not been submitted in whole or in part to any other university.

Signed: ....Date: ....

Abstract

Many non-linear, inherently unstable systems exist whose control using conventional methods is both difficult to design and unsatisfactory in implementation. Fuzzy Logic Controllers are a class of non-linear controllers that make use of human expert knowledge and an implicit imprecision to apply control to such systems. The construction of these controllers can be quick and effective in the presence of expert knowledge; conversely, in the absence of such knowledge, their design can be slow and based on trial-and-error rather than a guided approach.

Genetic Algorithms provide a way of surmounting this shortcoming. These algorithms use some of the concepts of evolutionary theory, and provide an effective way of searching a large and complex solution space to give close to optimal solutions in much faster times than random trial-and-error. They are also generally more effective at avoiding local minima than differentiation-based approaches.

In this report the application of Genetic Algorithms to the design and optimisation of Fuzzy Logic Controllers is demonstrated. These controllers are characterised by a set of parameters. The inverted pendulum system, being a commonly used example of a non-linear, unstable system is used as a test system for this approach. Control was successfully achieved in various simulations using the outlined methods and the results from these simulations are presented.

Details are also presented of efforts to use an online GA to continually improve the performance of a Fuzzy Logic Controller. Although these efforts were not fully successful, significant progress was made.

Table of Contents

Acknowledgements i

Declaration i

Abstract ii

1 Introduction 1
1.1 Fuzzy Logic 1
1.2 Genetic Algorithms 2
1.3 Overview of Work of Other Researchers 2

2 Inverted Pendulum System 4
2.1 Equations of Motion 4
2.2 Linearised Equations of Motion 6
2.3 SIMULINK model 7
2.4 Open Loop Response 8
2.5 Closed-Loop Control 9

3 Fuzzy Logic And Fuzzy Control 15
3.1 Fuzzy Logic 15
3.2 Fuzzy Control 17
3.3 FLC Design 20

4 Genetic Algorithms 22
4.1 Genetic Algorithm Basics 22
4.2 Selection Algorithms 23
4.3 Crossover Algorithms 24
4.4 Mutation Algorithms 25
4.5 Elitism 26
4.6 Encoding 26
4.7 Investigating GA Parameters 27

5 Designing a Fuzzy Logic Controller Using Matlab 33
5.1 Assumptions and Constraints 33
5.2 Spacing Parameter 34
5.3 Designing the Rule-Base 35
5.4 Matlab Implementation 36

6 Using Genetic Algorithms to Design Fuzzy Logic Controllers 44
6.1 Evaluation Functions 44
6.2 Parameter Encoding 47
6.3 Running the GA 48
6.4 Controlling Both Pole Angle and Cart Position 53

7 Online Genetic Algorithms 55
7.1 SIMULINK S-Functions 55
7.2 Implementing a GA in SIMULINK 56
7.3 Genetic-Based Machine Learning 61

8 Conclusion 65
8.1 Suggestions for Future Work 65
8.2 Conclusions 66

References 67
Appendix A Matlab Script and Function Files 69
Appendix A.1 Invpenstspace.m 69
Appendix A.2 gaMultimax.m 69
Appendix A.3 StocRemSelec.m 71
Appendix A.4 maskXover.m 72
Appendix A.5 make_fis.m 72
Appendix A.6 create_mfs.m 73
Appendix A.7 create_rules.m 74
Appendix A.8 run_fuzzy_pole_only 76
Appendix A.9 ga_poleonly.m 77
Appendix A.10 run_ga.m 78
Appendix A.11 initGa.m 82
Appendix A.12 ga_sfn.m 84
Appendix A.13 evalfis_sfn.m 87
Appendix A.14 fuzz_class.m 89
Appendix A.15 match_message.m 91
Appendix A.16 apportion_credit.m 91
Appendix A.17 fuzzify_online.m 92
Appendix A.18 defuzzify_online.m 94
Appendix A.19 centroid.m 96
Appendix A.20 calcbits2.m 97

Table of Figures

Figure 2-1 Inverted Pendulum System..4
Figure 2-2 Pole in Isolation 5
Figure 2-3 SIMULINK model of inverted pendulum system 7
Figure 2-4 Model used to obtain Open Loop Response..8
Figure 2-5 Open Loop Response 8
Figure 2-6 System Schematic 9
Figure 2-7 Rearranged Schematic..9
Figure 2-8 State-Space Schematic 10
Figure 2-9 State-Space Controller Impulse Response Model 11
Figure 2-10 State-Space Controller Impulse Response (Impulse: 10N for 100ms) 11
Figure 2-11 State-Space Controller Impulse Response (Impulse: 150N for 100ms) ..12
Figure 2-12 State-Space Controller Balances Pole From Initial Angle of 30°...........13
Figure 2-13 State-Space Controller fails to Balance Pole From Initial Angle of 35°..13
Figure 3-1 The Fuzzy Set “Young” 16
Figure 3-2 “Age” Universe of Discourse 17
Figure 3-3 Membership Functions..18
Figure 3-4 Degree of Firing of Mfs 19
Figure 3-5 Area to be defuzzufied 20
Figure 4-1 Single Point Crossover..24
Figure 4-2 Multi-point Crossover 24
Figure 4-3 Uniform Crossover..25
Figure 4-4 Test Function with Multiple Maxima 27
Figure 4-5 Run 1 29
Figure 4-6 Run 2 29
Figure 4-7 Run 10 30
Figure 4-8 Run 5 30
Figure 4-9 Run 7 31
Figure 4-10 Run 9 32
Figure 5-1 Effects of Spacing Parameters on Mfs 34
Figure 5-2 Seed Points and Grid Pts For Rule-Base Construction.............................36
Figure 6-1 Simulink Model used to evaluate FLCs..46
Figure 6-2 Progress of GA that found best fitness..49
Figure 6-3 Membership functions of Optimised FLC..50
Figure 6-4 Surface Plot of Optimised FLC (Scaled)...50
Figure 6-5 FLC and SSC Impulse Response Comparison 51
Figure 6-6 Response To Large Impulse 51

Figure 6-7 Bringing Pole to Upright Position 52
Figure 6-8 Comparison of FLC and SSC from Initial Angle of 5°.............................52
Figure 6-9 Response to 1kN disturbance 54
Figure 6-10 Response to 150N disturbance impulse 54
Figure 7-1 Simulink Block Characteristics 55
Figure 7-2 Simulink Model for GA 60
Figure 7-3 SIMULINK Implementation of Fuzzy Classifier System.........................62
Figure 7-4 GBML Progress (Disturbance Impulse 0.1N 0.01s)................................63
Figure 7-5 GMBL Progress (Impulse Disturbance 100N, 0.01s)...............................63
Figure 7-6 Close up of Figure 7-5..64

Table of Tables

Table 3-1 Simple Rule Base 17
Table 3-2 Typical Rule Base 19
Table 4-1 Parameters for GA Runs 28
Table 5-1 Derived Rule Base 36
Table 6-1Parameters used for encoding 47
Table 6-2 Parameters for GA Runs with FLC 48
Table 6-3 Rule-Base of Optimised FLC 50

Table of Listings

Listing 4-1Code for Test Objective Function 28
Listing 5-1 make_fis.m 38
Listing 5-2 create_mfs.m 40
Listing 5-3 create_rules.m 43
Listing 6-1 Evaluation Function run_fuzy_pole_only.m 46
Listing 7-1 Initialising State Information 56
Listing 7-2 Updating GA s-function 57
Listing 7-3 S_Function GA Operation 58
Listing 7-4 S-Function Output Routine 59

1 Introduction

This report presents details of the work carried out to optimise a Fuzzy Logic Controller using Genetic Algorithms. Detailed explanations of both these concepts are presented as well as a demonstration of how they can be applied to control a non-linear, unstable system. The inverted pendulum is both unstable and non-linear and is used to test out the methods tried for this project.

1.1 Fuzzy Logic

Fuzzy Logic is a logic system that uses imprecision. Just as human beings would never say that a table is 1.456 metres long, but rather that it is about one and a half metres long, fuzzy logic categorises objects into sets which are described by linguistic variables such as “long”, ‘fast”, “cool”, “heavy”, “middle-aged” and so on. Objects can have varying degrees of membership of such fuzzy sets, ranging from a crisp “definitely not a member” (denoted by 0) to a crisp “definitely a member” (denoted by 1). The crucial distinction is that between these crisp extremes, objects can have less certain degrees of membership such as “not really a member” (perhaps denoted by a 0.1) and “pretty much a member” (0.9 possibly).

Fuzzy Logic can be applied to control, and when it is, is known as Fuzzy Control. Fuzzy Control is made up of control rules which mimic those used by humans when they control or operate machinery: “If you need to go a little bit faster, push the accelerator pedal slightly”. Fuzzy Control can be an especially effective way of controlling non-linear systems when expert human knowledge of the system is available.

The details of how Fuzzy Logic and Fuzzy Control are applied are given in Chapter 3. Much more detailed information is available from Passino et al. [2].

1.2 Genetic Algorithms

Genetic Algorithms (GAs) are useful algorithms for optimising solutions to problems; especially those that are analytically intractable. They are inspired, as the name suggests, by the biological concepts of genetics and evolution.

Considering that the most accomplished controller known to man, the human brain, and other numerous marvels found in nature, all arose through the process of evolution through natural selection, the principles behind this highly successful “design” procedure should be able to provide some inspiration to improve the engineering design process. Genetic Algorithms use these principles to refine and optimise designs whose parameters interact in a complex manner. Individuals representing different potential solutions are preferentially selected according to their fitness and pass on their “genes”, i.e. characteristics, to future generations. Mating takes place between these individuals with the hope that by sharing the characteristics of successful individuals, even fitter individuals can be created. Mutation also occurs to inject new genes into a population. As in nature, most mutations are harmful but the occasional beneficial one can help improve the fitness of the individuals it affects, i.e. find better solutions.

1.3 Overview of Work of Other Researchers

A significant number of researchers in the field have applied Genetic Algorithms to the task of optimising Fuzzy Logic Controllers (FLCs). Many different approaches to this task have been taken. For example, Herrera et al. [11] apply GAs to optimise a rule-base of a FLC for which the membership functions have already been created, whereas Lee at al. [12] use a GA that determines the number of membership functions, the number of fuzzy rules and the rule-base. In both of these cases, the GA is implemented using real-valued encodings, whereas Belarbi et al. [13] use binary encoding for their approach, where they implement the FLC as a neural network and use the GA to train the weights.

Park et. al [8] implement a GA by using characteristic parameters to automate FLC design and this method is used for this project also. Using this technique, a FLC can be designed very flexibly, with the numbers and positions of membership functions determined by the GA as well as the rule-base. Rule-bases that are well-formed, as the term is used by Cheong et al.[9], emerge, which is another benefit.

In this report the basic principles of Fuzzy Logic and Genetic Algorithms are explained after which the main body of work performed is presented along with the results obtained. But before all that, the inverted pendulum system is analysed and an effort is made to build a controller based on a linear model of this system.

2 Inverted Pendulum System

In this chapter the Inverted Pendulum System is examined. It is analysed with a view to obtaining its equations of motion and then to linearise these equations in order to find a state-space-based model of the system. This model is used to design a controller, which is applied to a non-linear model of the system. Following this, the inadequacies of this controller, especially in dealing with the non-linearities are highlighted.

2.1 Equations of Motion

The inverted pendulum system is made up of a cart on top of which a pole is pivoted as shown in Figure 2-1. The cart is constrained to move only in the horizontal x-direction, while the pole can only rotate in the x-y plane.

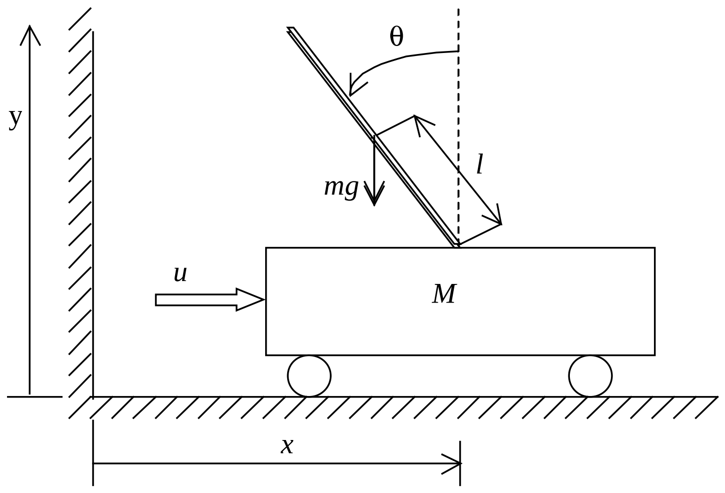


Figure 2-1 Inverted Pendulum System

The equations of motion for the system are derived using the following parameters

- *M* is the mass of the cart,
- *m* the mass of the pole,
- *l* the distance from the pivot to the centre of mass of the pole
- *I* the moment of inertia of the pole about the pivot,
- *x* the cart's position

- θ the angle the pole makes with the vertical and
- *u* the horizontal control force imparted to the cart.

The position of the centre of gravity of the pole is given by

$$x_G = x - \ell \sin\theta \tag{1}$$

Taking the second derivative of this with respect to time we get

$$\ddot{x}_G = \ddot{x} - l\ddot{\theta}\cos\theta + l\dot{\theta}^2 \sin\theta \tag{2}$$

The sum of the external forces on the system equals the mass multiplied by acceleration of each component, i.e.

$$u = M\ddot{x} + m\ddot{x}_G$$

$$\Rightarrow u = M\ddot{x} + m(\ddot{x} - l\ddot{\theta}\cos\theta + l\dot{\theta}^2 \sin\theta)$$

$$\Rightarrow \ddot{x}(M+m) + ml\dot{\theta}^2 \sin\theta - ml\ddot{\theta}\cos\theta = u \tag{3}$$

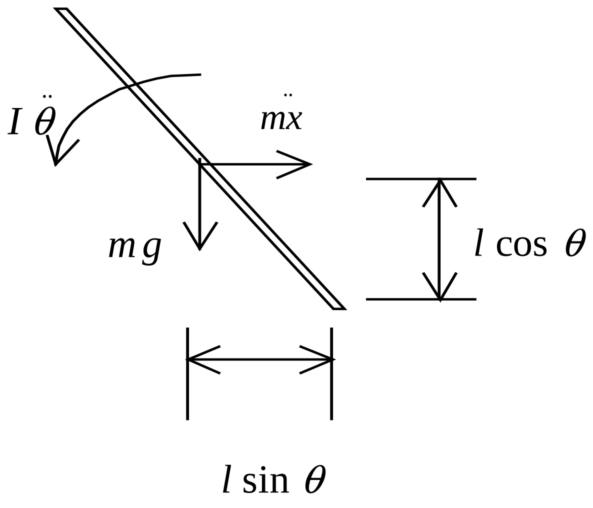


Figure 2-2 Pole in Isolation

Taking the pendulum in isolation (see Figure 2-2 above) and using

$$M_P = I_P\alpha + m\vec{a}d$$

which states that for a rigid body the sum of the moments about a fixed point, P, is equal to the moment of inertia of the body about P multiplied by the angular acceleration plus the product of the mass of the body, its linear acceleration and the perpendicular distance between the point P and the vector representing the acceleration [4]. Taking anticlockwise moments as positive this means that

$$mgl\sin\theta = I\ddot{\theta} - m\ddot{x}l\cos\theta \tag{4}$$

Rearranging (4) gives

$$\ddot{\theta} = \frac{mgl\sin\theta + ml\cos\theta\ddot{x}}{I}$$

Substituting this into (3) and rearranging gives

$$\ddot{x} = \frac{Iu + m^2l^2g\cos\theta\sin\theta - \mathrm{Im}l\dot{\theta}^2\sin\theta}{(M+m)I - m^2l^2\cos^2\theta} \tag{5}$$

Substituting this back into (4) and rearranging gives

$$\ddot{\theta} = \frac{(M+m)mgl\sin\theta + ml\cos\theta u - m^2l^2\dot{\theta}^2\cos\theta\sin\theta}{(M+m)I - m^2l^2\cos^2\theta} \tag{6}$$

For a uniform rod of mass *m* and length *L*=2*l* the moment of inertia about one end is $\frac{4}{3}ml^2$ [4]

Equations (5) and (6) thus become

$$\ddot{x} = \frac{\frac{4}{3}u + mg\cos\theta\sin\theta - \frac{4}{3}l\dot{\theta}^2\sin\theta}{\frac{4}{3}(M+m) - m\cos^2\theta} \tag{7}$$

and

$$\ddot{\theta} = \frac{(M+m)g\sin\theta + \cos\theta u - ml\dot{\theta}^2\cos\theta\sin\theta}{\frac{4}{3}(M+m)l - ml\cos^2\theta} \tag{8}$$

2.2 Linearised Equations of Motion

These equations can be linearised by noting that for small θ, sinθ ≈ θ, cosθ ≈ 1 and $\dot{\theta} \approx 0$. The equations then become

$$\ddot{x} = \frac{\frac{4}{3}u + mg\theta}{\frac{4}{3}(M+m) - m} \tag{9}$$

and

$$\ddot{\theta} = \frac{(M+m)g\theta + u}{\frac{4}{3}(M+m)l - ml} \tag{10}$$

These linearised equations can be represented in state-space form as follows:

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{mg}{\frac{4}{3}(M+m)-m} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{(M+m)g}{\frac{4}{3}(M+m)l-ml} & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{\frac{4}{3}}{\frac{4}{3}(M+m)-m} \\ 0 \\ \frac{1}{\frac{4}{3}(M+m)l-ml} \end{bmatrix} u \tag{11}$$

This is in the usual form $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}u$ used for state-space expressions.

2.3 SIMULINK model

Using SIMULINK[1], a model of the system, (see Figure 2-3 below) was created. In this model, the control action is an input. In combination with the angle and angular velocity from the previous time step, it is used to calculate the cart's acceleration and the pole's angular acceleration using equations (7) and (8) respectively. Integration is then performed to get the speeds and positions for both components. It should also be noted that the value of the pole's angle is limited to remain between ±90°. The pole angle and angular velocity and the cart position and velocity are the outputs of this model.

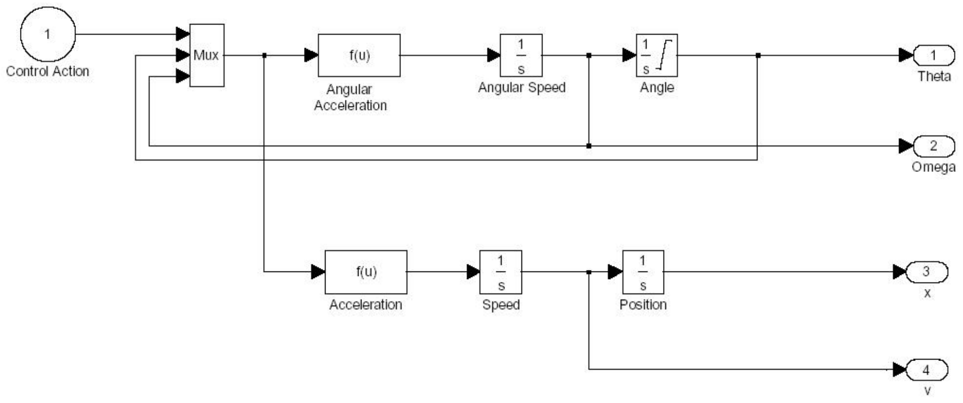


Figure 2-3 SIMULINK model of inverted pendulum system

[1] For the SIMULINK solver, the ode4 (Runge-Kutta) algorithm was used with a fixed time step of 0.01s and Single-Tasking. Except where otherwise noted, these parameters were used for all models mentioned in this report.

2.4 Open Loop Response

The open loop response of this system was obtained using the SIMULINK model shown in Figure 2-4 below.

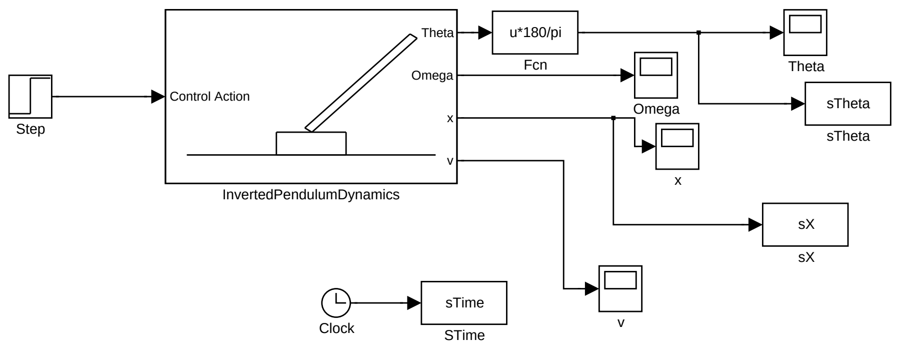


Figure 2-4 Model used to obtain Open Loop Response

The simulation was run for ten seconds and the parameters[1] used were:

• M, the mass of the cart:	1kg
• m, the mass of the pendulum:	0.1kg
• l, the distance from the pivot to the pendulum centre of mass:	0.5m

The open loop response is plotted in Figure 2-5. The system can be seen to be unstable, with the pole becoming horizontal very quickly. To keep the pole vertical, a controller will be needed.

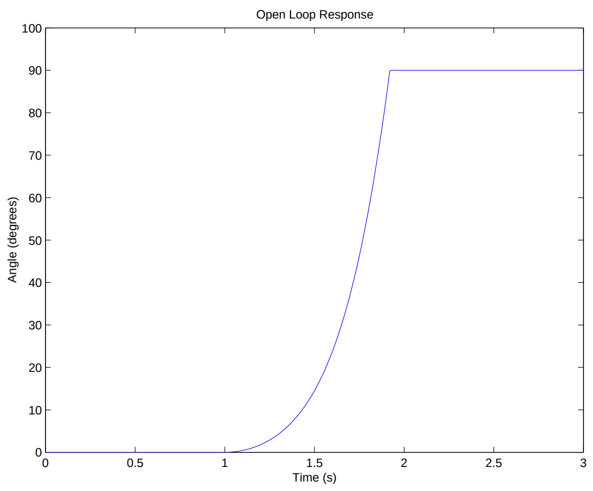


Figure 2-5 Open Loop Response

[1] These parameters were also used for all models detailed in this report

2.5 Closed-Loop Control

An attempt is now made to design a controller for the system. The state-space method is used to do this as this allows the design of a controller of both the pendulum angle and cart position. Much of the work in this section is based on a tutorial on the University of Michigan website [6].

As the main objective for controlling this system is to keep the pendulum upright as much as possible, the reference value for the angle to which the output is compared will be zero throughout all simulations. When analysing the system's response, it is the response to a disturbance force applied to the cart, rather than to a change in the reference that will be examined. This is shown schematically in Figure 2-6 below.

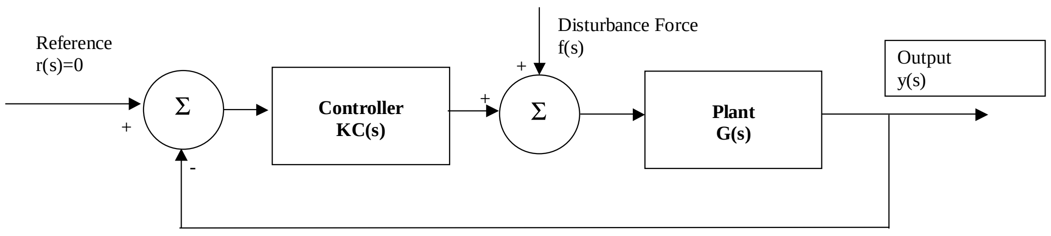


Figure 2-6 System Schematic

The schematic can be rearranged as in Figure 2-7.

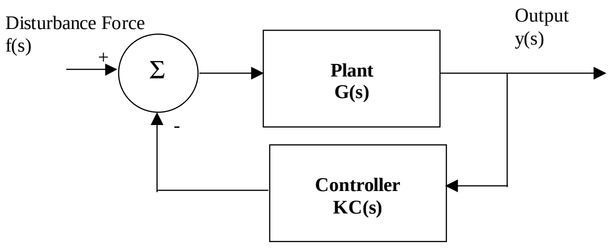


Figure 2-7 Rearranged Schematic

As it is a response to a disturbance force that is being analysed, the reference position for the cart will also be zero. To control both angle and position, a state-space controller will be designed. The schematic for this is shown in Figure 2-8.

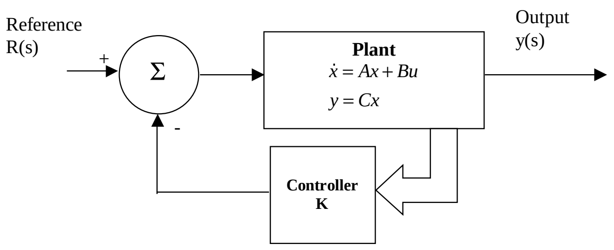


Figure 2-8 State-Space Schematic

To design a controller the Linear-Quadratic Regulator (LQR) method described by Friedland [3] is used. Using the state-space equation (11), this method finds the optimal K based on the state feedback law

$$u = -\mathbf{K}\mathbf{x} \tag{12}$$

such that the cost function

$$J = \int (\mathbf{x}'\mathbf{Q}\mathbf{x} + \mathbf{u}'\mathbf{R}\mathbf{u}) d\tau \tag{13}$$

is minimised. **Q** is a weighting matrix specifying the relative importance of each of the system states and similarly, **R** is a weighting matrix that specifies the relative importance of the inputs. As there is only one input into the system, R is set to one in this specific case. The function *lqr* provided with the Control Systems Toolbox for Matlab is used to calculate the matrix K.

Code obtained from the University of Michigan Control Tutorials website [6] was modified to obtain the controller gain K (See Appendix A.1). The weighting is set so that the position and angle each had a relative importance of 5000 compared to the control action. The modified code was run and the vector K obtained was

$$\mathbf{K} = [-70.711 \quad -52.919 \quad 205.05 \quad 49.97] \tag{14}$$

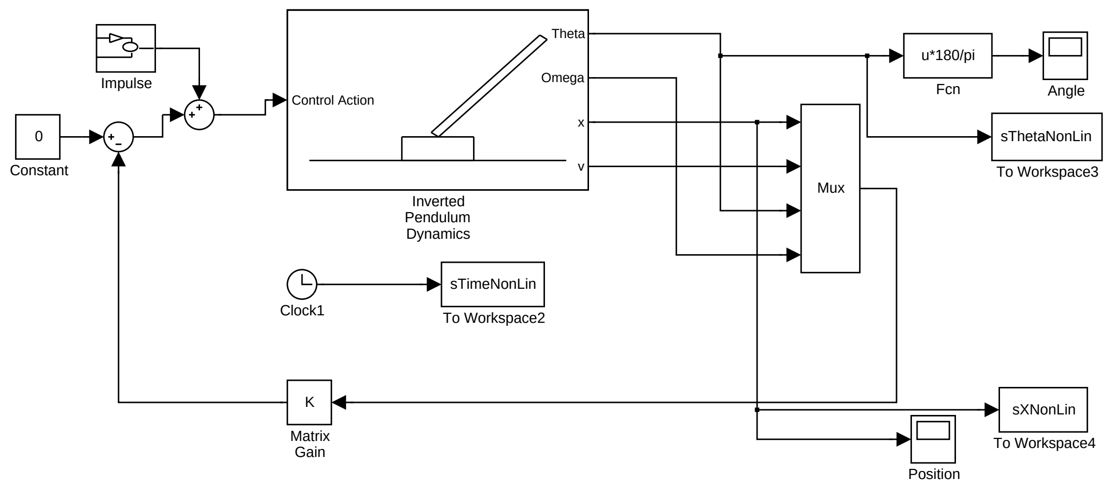


Figure 2-9 State-Space Controller Impulse Response Model

This controller was modelled in SIMULINK (Figure 2-9). The response to an impulse disturbance of 10N applied for 100ms was obtained and the plot of this response appears in Figure 2-10 below. As can be seen, the controller succeeds in restoring both the angle and pendulum to zero after a settling time of approximately 2s.

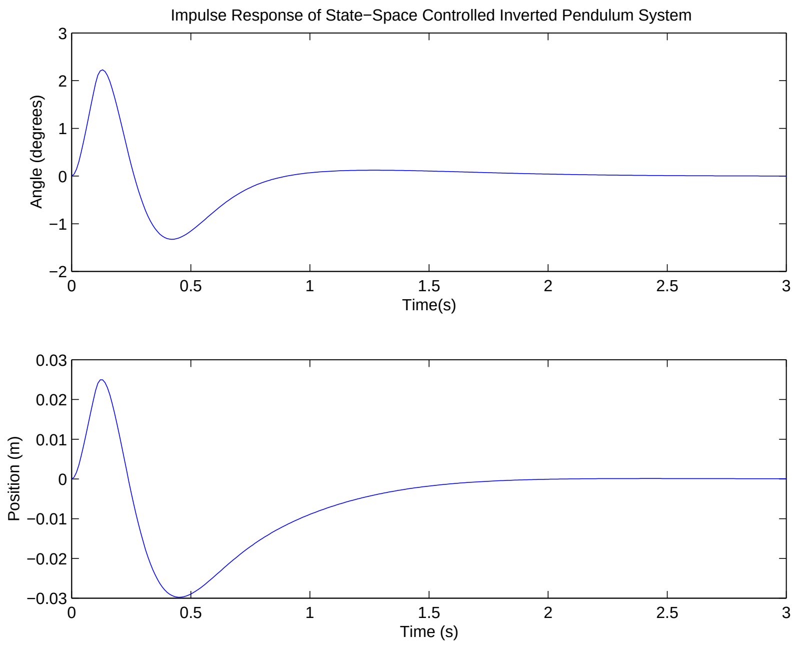


Figure 2-10 State-Space Controller Impulse Response (Impulse: 10N for 100ms)

This controller is based upon a linearised model and hence is expected to work when the assumptions taken to linearise the model are valid, i.e. when the angle, θ, is small.

To see how the controller works when the angle is quite large, the disturbance force needs to be bigger. Hence, the above system was modelled again with an impulse force of 150N again applied for 100ms. The response obtained is plotted in Figure 2-11.

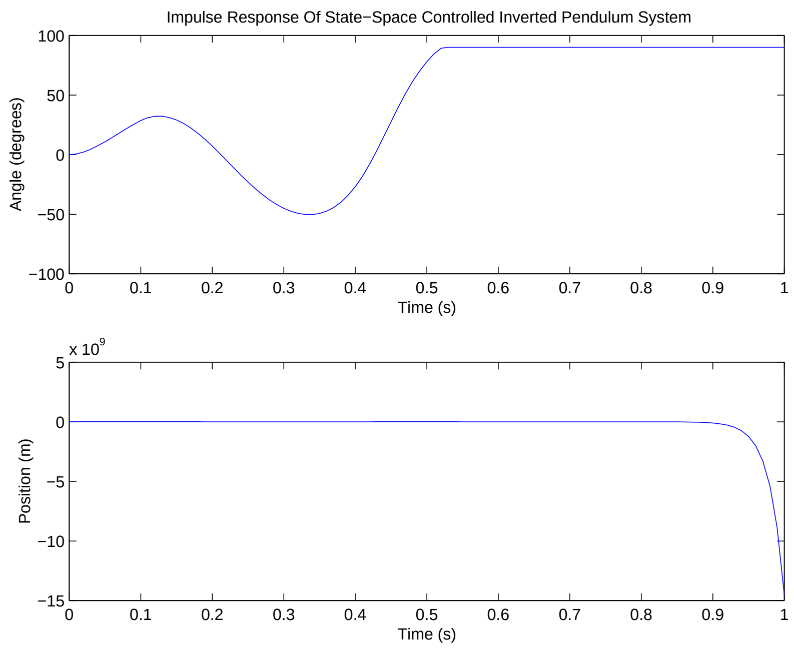


Figure 2-11 State-Space Controller Impulse Response (Impulse: 150N for 100ms)

Here, the perturbation to the system is too large for the controller to handle. The non-linearities, assumed to be negligible when the controller was designed, begin to dominate when the angle gets too large and hence, the pole collapses and falls to 90°.

Following this, the ability of the controller to bring the pendulum to its equilibrium position from an initial non-equilibrium position was investigated. As can be seen from Figure 2-12, when the initial angle is 30°, the controller successfully balances the pole. However, when the initial angle is increased to 35°, the pole collapses. This gives an indication of the region in which the controller can operate.

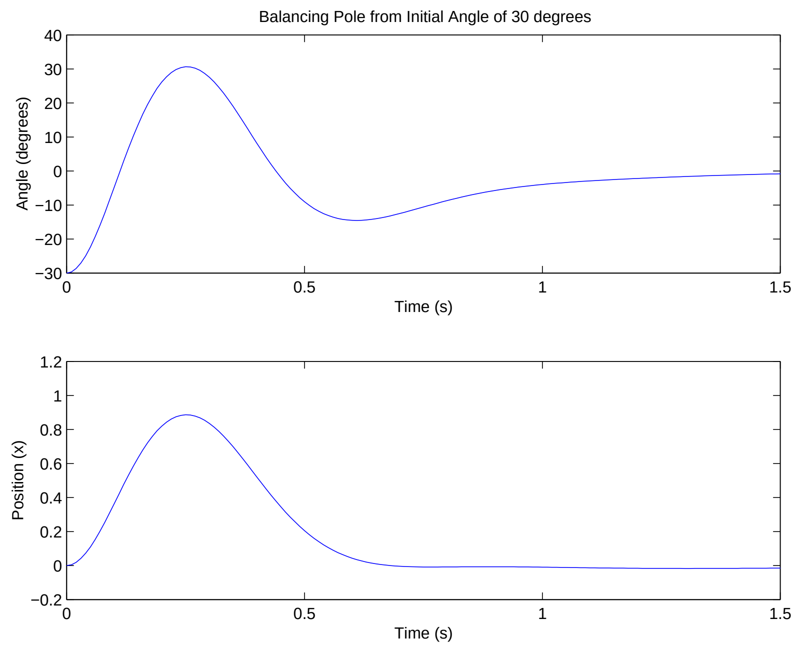


Figure 2-12 State-Space Controller Balances Pole From Initial Angle of 30°

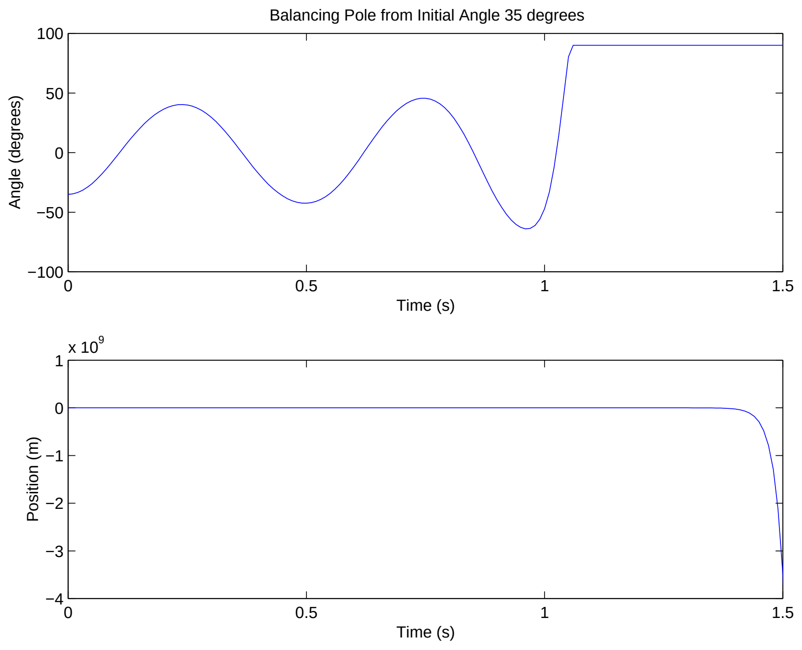


Figure 2-13 State-Space Controller fails to Balance Pole From Initial Angle of 35°

This controller successfully controls the system as long as the pole is not allowed to stray too much from the equilibrium position. When the pole angle gets too large it is unable to restore it to the upright position. A controller that can successfully achieve this needs to be able to handle the non-linearities. For this reason, Fuzzy Logic Controllers are investigated to see if they can help solve the problem.

3 Fuzzy Logic And Fuzzy Control

Fuzzy Control is based on the principles of Fuzzy Logic developed by Zadeh [14] in 1965. It is a non-linear control method, which attempts to apply the expert knowledge of an experienced user to the design of a controller. In this chapter, the basic principles of fuzzy logic are presented as well as a demonstration of how these principles are applied to the control of engineering systems.

3.1 Fuzzy Logic

Fuzzy logic is based on the theory of fuzzy sets where variables can have differing degrees of membership of sets. This is unlike the more familiar crisp set theory where a variable is either a full member of a set or it is not a member of that set at all. The degree to which a variable belongs to a set can vary between 0 and 1. This can allow for the handling of borderline cases and other hard-to-categorise situations in a more intuitively satisfying way.

A universe of discourse is defined as the whole range of fuzzy sets to which a variable can belong. Each set on this universe of discourse is referred to as a membership function and is often described using a 'linguistic variable'. On a universe of discourse, a variable has a degree of membership of each membership function that varies between 0 and 1.

Fuzzy Logic uses rules with antecedents and consequents to produce outputs from inputs. The antecedents are the inputs that are used in the decision-making process or the "IF" parts of the rules. The consequents are the implications of the rules or the "THEN" parts.

3.1.1 Fuzzy Logic Example

An example of a fuzzy set is the set of humans who could be described as young. Most people would agree that anyone aged between zero and twenty could be described as being definitely young, whereas anyone over forty would be described as not at all young. The ages between twenty and forty are more of a grey area, however. The closer a person’s age is to twenty the more readily they could be described as young. This fuzzy set is displayed in Figure 3-1 below. According to this fuzzy set a person aged fifteen is definitely young, i.e. their degree of membership of the fuzzy set *Young* is one. However, it is less clear-cut whether a person aged thirty could be described as young, i.e. their degree of membership is less than one, in this example, 0.5.

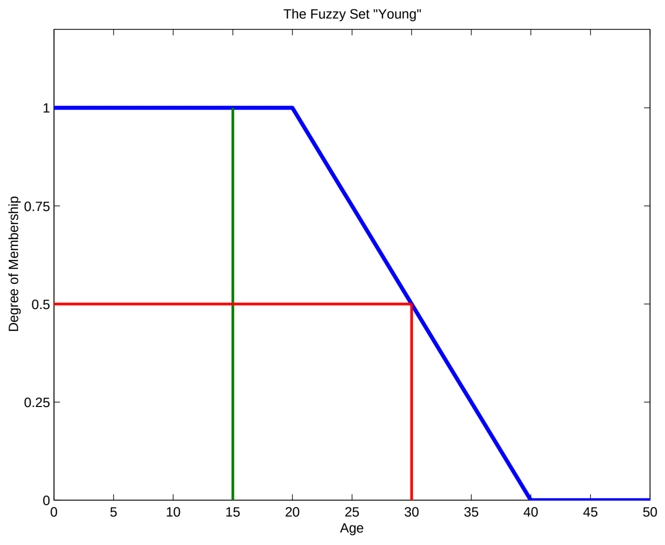


Figure 3-1 The Fuzzy Set “Young”

The Universe of Discourse of a variable is the range of values that that variable can take. Many fuzzy sets can be defined on one Universe of Discourse and a single variable can have membership of more than one fuzzy set. For example, in Figure 3-2 below we return to our “Age” example and examine the Universe of Discourse on which, in addition to *Young*, the fuzzy sets *Middle Aged* and *Old* are added. Here we can see that a person aged 35 has membership of two sets, *Young* and *Middle Aged* with degrees of membership of 0.25 and 0.33 respectively.

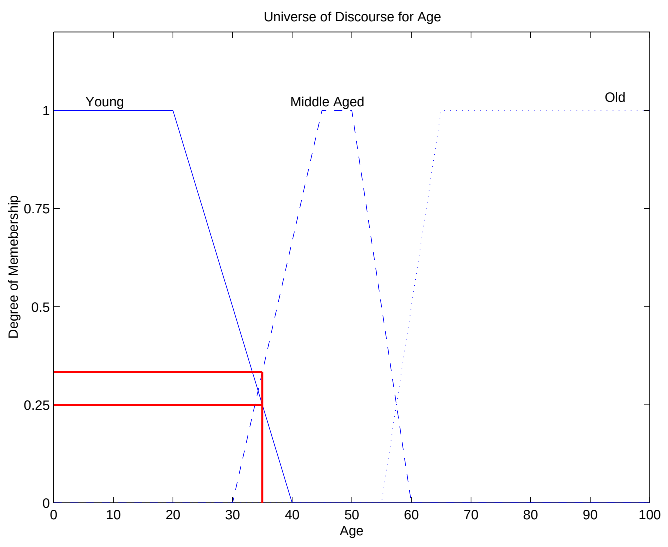


Figure 3-2 "Age" Universe of Discourse

3.2 Fuzzy Control

Fuzzy Control applies fuzzy logic to the control of processes by utilising different categories, usually 'error' and 'change in error', for the process state and applying rules to decide a level of output, i.e. a suitable control action. The linguistic variables used for both input and output variables are often of the form 'negative large', 'positive small', 'zero' etc. A typical rule base for a two-input, single-output system with three membership functions per variable is shown in Table 3-1

DE \ E	N	Z	P
N	N	N	Z
Z	N	Z	P
P	Z	P	P

Table 3-1 Simple Rule Base

For example, using this rule base, if the error, E, was negative, N, and the change in error, DE, was positive, P, then the output would be zero, Z. This particular rule recognises the fact that although there is an error in the system, it is approaching zero so no control action needs to be applied.

3.2.1 Defuzzification

Often input variables are members of more than one fuzzy set defined on their universe of discourse. This means that for each combination of inputs, there will, in general, be more than one rule fired. Also, the output needs to have a precise numeric value. Defuzzification is the process by which the sets of the various fired rules are combined to produce an output.

To illustrate this, take the following example. The FLC has two inputs, *Error* and *Change in Error* and one output, *Action.* The universe of discourse for each of these variables is normalised so that their values always lie between –1 and 1. (Scaling gains are applied to each variable to give the appropriate range). There are five membership functions for each variable spaced as in Figure 3-3. The rule-base is as given in Table 3-2.

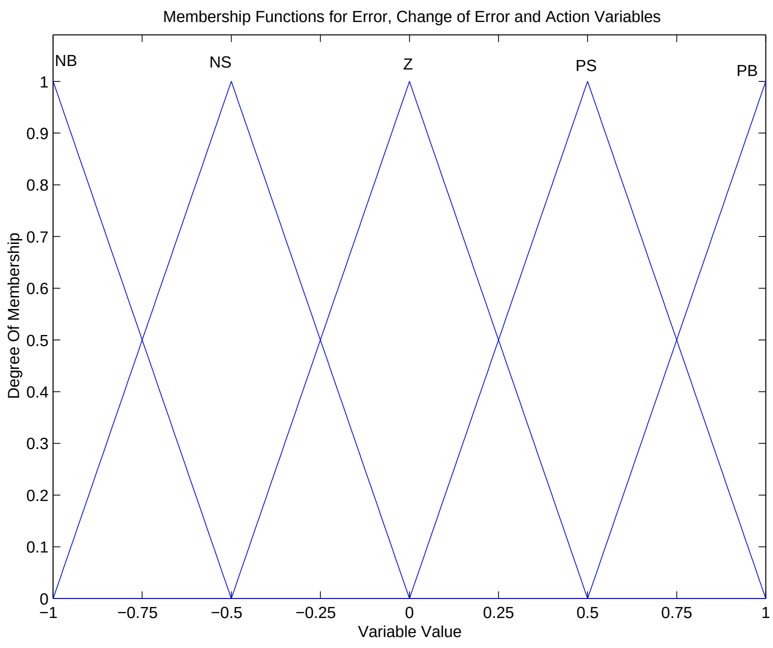


Figure 3-3 Membership Functions

de \ e	N B	N S	Z	PS	P B
N B	N B	N B	N S	NS	Z
N S	N B	N S	N S	Z	PS
Z	N S	N S	Z	PS	PS
P S	N S	Z	P S	PS	PB
P B	Z	P S	P S	PB	PB

Table 3-2 Typical Rule Base

If the input error is 0.4, this means that it is a member of two sets, *Z* to a degree of 0.2, and *PS,* to a degree of 0.8 (see Figure 3-4). If the input change in error is –0.2, this is a member of *Z* to a degree of 0.6 and of *NS* to a degree of 0.4. Four rules altogether are fired in this situation and the degree to which each is fired is given by the minimum degree of the two inputs firing it. (This is how the boolean AND operation is performed in Fuzzy Logic). The rules fired and the degrees to which they are fired are:

- If error is Z and change in error is Z then the control action is Z (degree 0.2)
- If error is Z and change in error is NS, then the control action is NS (degree 0.2)
- If error is PS and change of error is Z then the control action is PS (degree 0.6)
- If error is PS and change of error is NS then the control action is Z (degree 0.4)

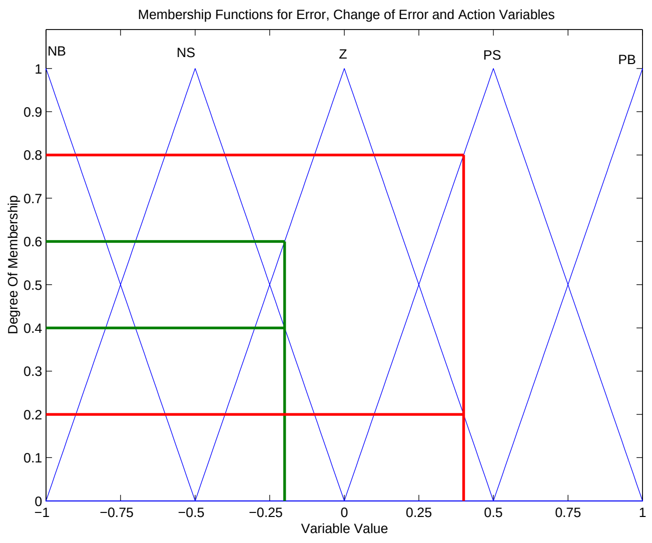


Figure 3-4 Degree of Firing of Mfs

Combining these control actions, with each output fuzzy set cut off at the level corresponding to the degree to which they are fired, gives a shape as shown in Figure 3-5. This needs to be converted into a numerical value for the control action to be applied. The process of conversion is known as defuzzification and there are many methods of performing it. A very popular method is to take the centre of gravity of the resultant shape and apply a control action corresponding to the x-axis value of this point. In our example, the centre of gravity lies at (0.1528,0.2213) which means that the control action has a value of 0.1528. This is then scaled by the appropriate scaling factor to give a suitable output.

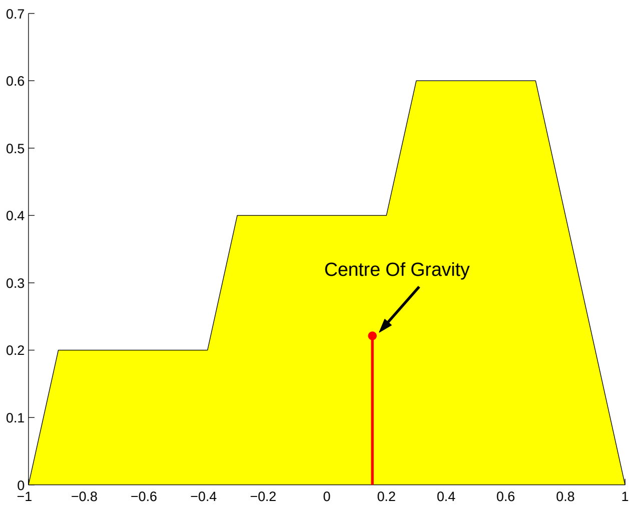


Figure 3-5 Area to be defuzzufied

3.3 FLC Design

When designing a Fuzzy Logic Controller (FLC) expert knowledge of the process to be controlled can be used to design the membership functions and rule base. Unfortunately there is no general procedure for designing a FLC from first principles, and heuristics are usually required as well as much trial and error to achieve a FLC that meets the design objectives and constraints. In this traditional approach, designing an FLC can be a laborious, time-consuming process [16]. These FLCs also tend to be non-portable to other applications.

However, another technique exists for optimising a FLC. Genetic Algorithms (GAs) are a class of algorithms that can be used to search large solution spaces for solutions that are close to optimal. How they work is explained in the next chapter.

4 Genetic Algorithms

Genetic Algorithms are reliable and robust methods for searching solution spaces [1], [5]. They are inspired by the biological theory of evolution through natural selection and much of the terminology is similar.

4.1 Genetic Algorithm Basics

A chromosome is an encoded string of possible values for the parameters to be optimised. These chromosomes can be made up of real-valued or binary strings. Often one of the main challenges in designing a genetic algorithm to find a solution to a problem is finding a suitable way to encode the parameters.

A set of potential solutions, called a population, is created. Each member of this set is referred to as an individual and they are evaluated by decoding the parameter values from the chromosomes and applying them to the problem to see how well they perform the task at hand (the objective that is to be optimised). The score that an individual achieves at performing the required task is called its fitness.

After the fitness of each individual has been calculated, a procedure known as selection is performed. Individuals are selected to contribute towards creating the next generation, the probability of selection being related to the individual's fitness.

Once selection has occurred, crossover takes place between pairs of selected individuals. The strings of two individuals are mixed. In this way, new individuals are created that contain characteristics that come from different hereto relatively successful individuals.

A third operation that occurs is mutation, the random changing of bits in the chromosome. It is generally performed with a relatively low probability. Mutation ensures that the probability of searching a given part of the solution space is never zero.

There are many ways in which these different operations can be applied. Different algorithms can be used for each and they can also be applied with varying degrees of probability. Some of the more popular algorithms for each of these operations are now examined and their effects on the GA's performance is investigated.

4.2 Selection Algorithms

A popular selection algorithm is stochastic sampling with replacement, more commonly known as the "Roulette Wheel" algorithm, so called because this method works in a way that is analogous to a roulette wheel. Each individual in a population is allocated a share of a wheel, the size of the share being in proportion to the individual's fitness. A pointer is spun (a random number generated) and the individual to which it points is selected. This continues until the requisite number of individuals has been selected. An individual's probability of selection is thus related to its fitness ensuring that fitter individuals are more likely to leave offspring. A problem with this approach is that the number of times an individual is actually selected has a high variance so there is no guarantee that fitter individuals will be represented in the next generation.

Stochastic Remainder Selection is another popular algorithm. In this method the expectation of the number of times selected is calculated for each individual. The integer portions of this expectation for each individual are assigned deterministically and the fractional remainders are assigned in the same way as in roulette wheel selection. For example, an individual whose expectation of number of times selected was 2.4 would be certain of being selected twice and the probability of being selected a third time would be 0.4. This approach reduces the variance associated with the roulette wheel algorithm and ensures that all individuals with above-average fitness will be represented in the next generation.

4.3 Crossover Algorithms

Once selection is finished crossover is performed. Individuals are paired for mating and by mixing their strings new individuals are created. The most basic crossover algorithm is known as Single Point Crossover. A single point along the string is chosen and the strings are swapped over at this point.

Parent 1	101|11100
Parent 2	110|10010
Child 1	101 10010
Child 2	110 11100

Figure 4-1 Single Point Crossover

Multipoint crossover algorithms extend simple crossover by selecting multiple crossover points and alternately assigning to the first or second offspring the portions of string between these points.

Parent 1	10|111|100
Parent 2	11|010|010
Child 1	10 010 100
Child 2	11 111 010

Figure 4-2 Multi-point Crossover

Uniform crossover is another crossover algorithm. This time a random mask of 1s and 0s of the same length as the parent strings is generated. If a bit in the mask is 1 then the corresponding bit in the first child will come from the first parent and the second parent will contribute that bit to the second offspring. If the mask bit is 0 the first parent contributes to the second child and the second parent to the first child.

Parent 1	10111100
Parent 2	11010010
Mask	01001101
Child 1	10011110
Child 2	11110000

Figure 4-3 Uniform Crossover

Uniform crossover is the most disruptive of the crossover algorithms [15], i.e. it is the most likely to cause neighbouring bits that contribute in a positive way to the fitness of the individual to be split up. However at the same time, uniform crossover allows for more extensive searching of the solution space as there are significantly more potential offspring using this method.

4.4 Mutation Algorithms

For binary codings, there is really only one way to mutate. For each bit generate a random number and if it is less than the specified mutation probability, flip the bit, i.e., if it is a 1 change it to 0 or vice versa. This mutation probability is generally kept quite low and is constant throughout the lifetime of the GA. However, a variation on this basic algorithm changes the mutation probability throughout the lifetime of the algorithm, starting with a relatively high rate and steadily decreasing it as the GA progresses. This allows the GA to search more for potential solutions at the outset and to settle down more as it approaches convergence.

When real-valued codings are used, the mutation algorithm can be more complex. Many different algorithms are used some of which are as follows:

- Uniform Mutation: A random value within the constraints of the variable is chosen.
- Boundary Mutation: The variable is set to either its lower or upper bound.
- Non-Uniform Mutation: The variable is assigned a value based on a bell curve that becomes progressively narrower as the GA progresses. This ensures that as

convergence is approached, the range within which a variable can be mutated also narrows.

4.5 Elitism

With crossover and mutation taking place, there is a high risk that optimum solutions may be lost as there is no guarantee that these operations will preserve fitness. To combat this elitist models are often used [1]. In these models, the best individual from a population is saved before any of the operations take place. After the new population is formed and evaluated, it is examined to see if this best structure has been preserved. If not, the saved copy is reinserted back into the population, usually at the expense of the weakest member. The GA then proceeds to perform the operations on this population.

4.6 Encoding

Genetic Algorithms can be performed using either binary or real-valued encodings. With binary encoding, each parameter is converted into a binary string. These strings are concatenated and the genetic operations are performed on this concatenated string. With real-valued encoding, the parameters are kept in their real number format. Both forms of encodings are used in practice.

The advantages of using a binary format is that it

> *… maximises the number of hyperplane partitions directly available in the encoding for schema processing.* [15]

In other words, binary alphabets allow for greater sampling of the solution space and for the processing of more combinations of alleles.

However encoding using higher cardinalities can be more efficient. For example, if a certain parameter could take on five possible values then it would need to be encoded using three bits in a binary scheme. However, this leads to eight possible alleles, three of which are superfluous. Using a five-letter alphabet in this case would lead to more efficient coding.

4.7 Investigating GA Parameters

Other factors also have an effect on how the GA performs. These include population size, mutation probability and crossover probability among others. To investigate how some of these factors affected the GA an experiment was performed using a test function with multiple maxima.

The test function is $8+5\sin(7x)-2\cos(5x)$. This function is plotted in Figure 4-4. As can be seen from the plot, this function has many maxima whose peak values are different. It is hoped that the GA searching this solution space will find the peak value in this range which occurs at x = 5.618, y = 14.958 or somewhere close to it. There are also two local maxima whose peaks fall just short of the global peak. Those searches that succeed in avoiding getting stuck at these local maxima are those whose properties should be noted and emulated.

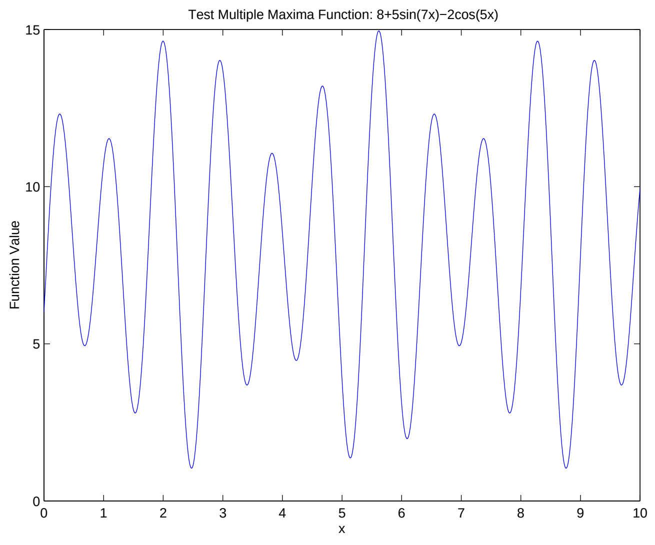


Figure 4-4 Test Function with Multiple Maxima

To perform the GA, Matlab was used in conjunction with the GAOT toolbox, which is open-source code provided by Houck et al. [7]. An objective function has to be provided for this toolbox that evaluates the string passed in. This code is shown below.

```
function [x, multimaxval]  = multimax(x,Ops)
%Evaluates the test multiple maxima function
%for the GAOT toolbox

multimaxval = 8 + (5*sin(7*x) - 2*cos(5*x));
```

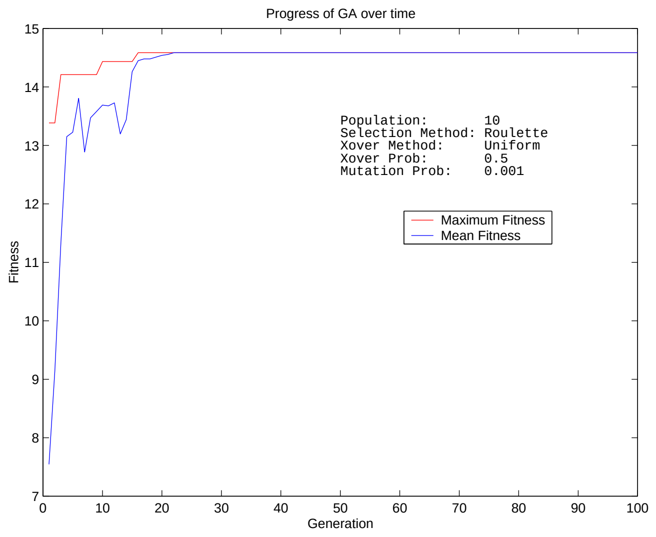
Listing 4-1Code for Test Objective Function

To apply the parameters to the GA a script file was written in which the essential parameters are set. This file is listed in Appendix A.2. Ten different runs of the GA were performed, each for 100 generations. The parameters for these runs are given in Table 4-1. To implement some of the algorithms in these runs new Matlab .m files were written to supplement the GAOT toolbox. These were *stocRemSelec.m,* which performs Stochastic Remainder Selection, and *maskXover.m* which performs uniform crossover. The code for these files is listed in Appendices A.4 and A.5.

Run	Population Size	Selection Algorithm	Crossover Algorithm	Crossover Probability	Mutation Probability	Best Find	Gen
1	10	Roulette	Uniform	0.5	0.001	14.586	16
2	200	Stoc. Rem.	Single Pt.	1.0	0.1	14.958	77
3	200	Stoc. Rem.	Uniform	1.0	0.01	14.958	16
4	10	Stoc. Rem.	Uniform	1.0	0.01	14.632	28
5	50	Stoc. Rem.	Uniform	1.0	0.01	14.958	86
6	100	Stoc. Rem.	Uniform	1.0	0.01	14.958	41
7	50	Stoc. Rem.	Single Pt.	1.0	0.01	14.958	20
8	50	Stoc. Rem.	Uniform	0.5	0.01	14.958	13
9	50	Roulette	Uniform	1.0	0.01	14.958	12
10	50	Stoc. Rem.	Uniform	1.0	0.1	14.958	66

Table 4-1 Parameters for GA Runs

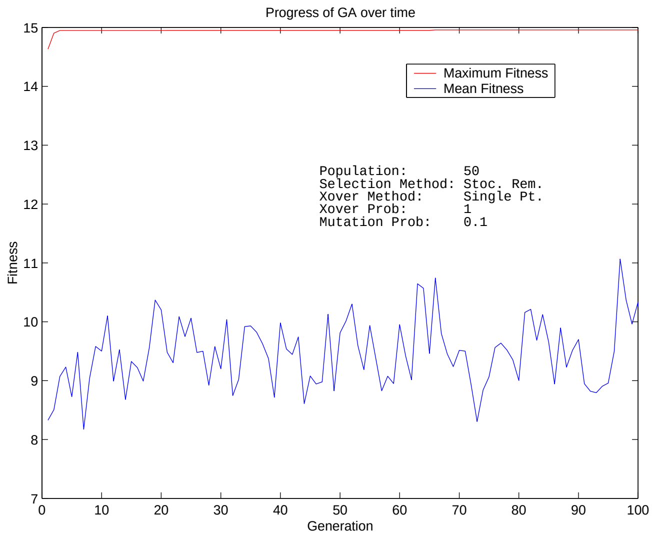
Of these ten runs only runs 1 (Figure 4-5) and 4 failed to come to within 1% of the actual peak value within the first hundred generations. In both of these cases the run converged to the final value before the thirtieth generation suggesting that both of these runs got stuck at a local maximum. This suggests strongly that population size has a strong bearing on the performance of a genetic algorithm.



```
Population:        10
Selection Method:  Roulette
Xover Method:      Uniform
Xover Prob:        0.5
Mutation Prob:     0.001
```

Figure 4-5 Run 1

Runs 2 (Figure 4-6) and 10 (Figure 4-7) both have high mutation rates (10%). As the average value of the test function is 8.02 and the mean fitness of the individuals for these runs does not go much higher than this, it can be seen that high mutation rates render the GA as almost a random search, i.e., the purposefulness of the search is reduced significantly.



```
Population:        50
Selection Method:  Stoc. Rem.
Xover Method:      Single Pt.
Xover Prob:        1
Mutation Prob:     0.1
```

Figure 4-6 Run 2

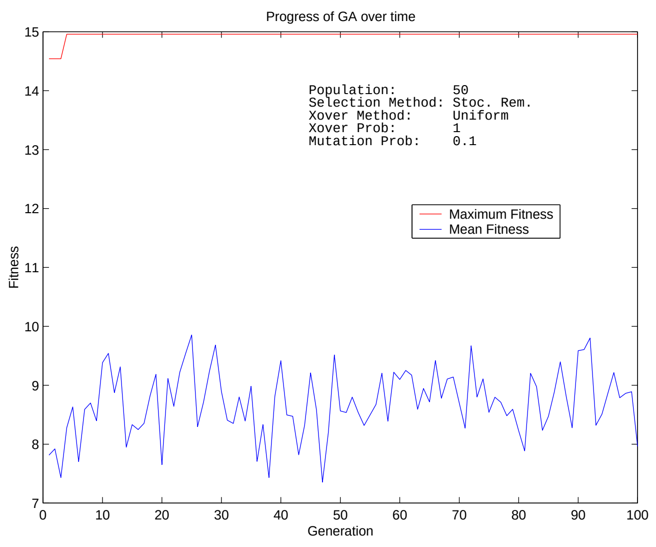


Figure 4-7 Run 10

Runs 3 to 6 differ in only their population size. Run 5 (Figure 4-8) performed well as the mean fitness is significantly larger than that of a random search while it remains low enough to suggest that the GA is doing a significant amount of searching of the solution space. This is desirable as it increases the likelihood of the GA finding a better solution (even though we know there isn't one in this case). If the GA converges too quickly (prematurely), it is more likely to get stuck in a local maximum.

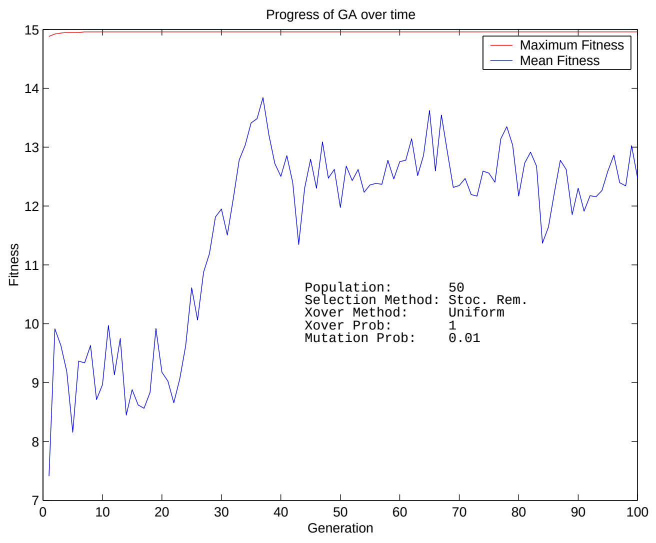


Figure 4-8 Run 5

Using the parameters of Run 5 as a baseline, the effects of changing the other parameters are then investigated. In Run 7 (Figure 4-9), the crossover algorithm changes from uniform to single-point crossover. This has the effect of increasing the rate at which the mean fitness approaches the maximum fitness, showing that single-point crossover is indeed less disruptive than uniform crossover.

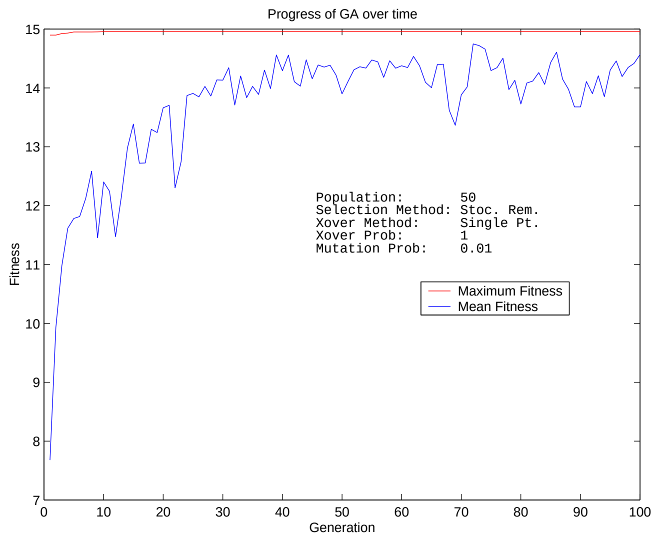


Figure 4-9 Run 7

In Run 8, the crossover rate is changed to 0.5. Again, this leads to less disruption and hence quicker convergence. These runs suggest that instead of large population sizes a medium-sized population used in conjunction with stochastic remainder selection and high crossover rates can be quite effective.

The roulette wheel selection algorithm is tried in Run 9 (Figure 4-10). This also leads to quick convergence. Perhaps because of the reduced variance in selection probabilities, the less fit individuals contribute more consistently in the Stochastic Remainder Selection Algorithm than in the Roulette Wheel one. This however should be confirmed with more investigation.

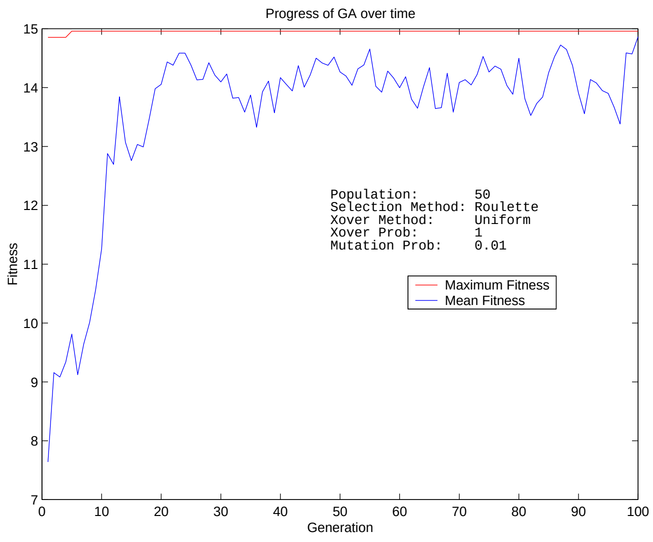


Figure 4-10 Run 9

Having gained a good knowledge of how various parameters affect GA performance, the GA can now be applied to the main task of optimising a FLC. Before this can be done, a way of automating the design of the FLC in Matlab needs to be devised. This problem is tackled in the next chapter.

5 Designing a Fuzzy Logic Controller Using Matlab

In this chapter, a demonstration is given of how to automate the design of a Fuzzy Logic Controller. The assumptions used and the constraints introduced to simplify this process are explained. Reference is made to the Matlab code that is written to implement this process and a summary of how this code works is provided.

5.1 Assumptions and Constraints

To apply the Fuzzy Logic Controller to the Inverted Pendulum System, certain properties of the system are exploited so that the design of the controller can be made easier. As the system is symmetrical, it is assumed that symmetrical membership functions about the y-axis will provide a valid controller. A symmetrical rule-base is also assumed.

- Other constraints are also introduced to the design of the FLC:
- All universes of discourses are normalised to lie between –1 and 1 with scaling factors external to the FLC used to give appropriate values to the variables.
- It is assumed that the first and last membership functions have their apexes at –1 and 1 respectively. This can be justified by the fact that changing the external scaling would have similar effect to changing these positions.
- Only triangular membership functions are to be used.
- The number of fuzzy sets is constrained to be an odd integer greater than unity. In combination with the symmetry requirement, this means that the central membership function for all variables will have its apex at zero.
- The base vertices of membership functions are coincident with the apex of the adjacent membership functions. This ensures that the value of any input variable is a member of at most two fuzzy sets, which is an intuitively sensible situation. It also ensures that when a variable’s membership of any set is certain, i.e. unity, it is a member of no other sets.

Using these constraints the design of the membership functions can be described using two parameters:

- The number of membership functions
- The positioning of the triangle apexes

5.2 Spacing Parameter

The second parameter specifies how the centres are spaced out across the universe of discourse. A value of one indicates even spacing, while a value larger than unity indicates that the membership functions are closer together in the centre of the range and more spaced out at the extremes as shown in Figure 5-1. The position of each centre is calculated by taking the position the centre would be if the spacing were even and by raising this to the power of the spacing parameter. For example, in the case where there are five sets, with even spacing (p=1) the centre of one set would be at 0.5. If p is set to two, the position of this centre moves to 0.25. If the spacing parameter is set to 0.5 then this centre moves to 0.707 in the normalised universe of discourse.

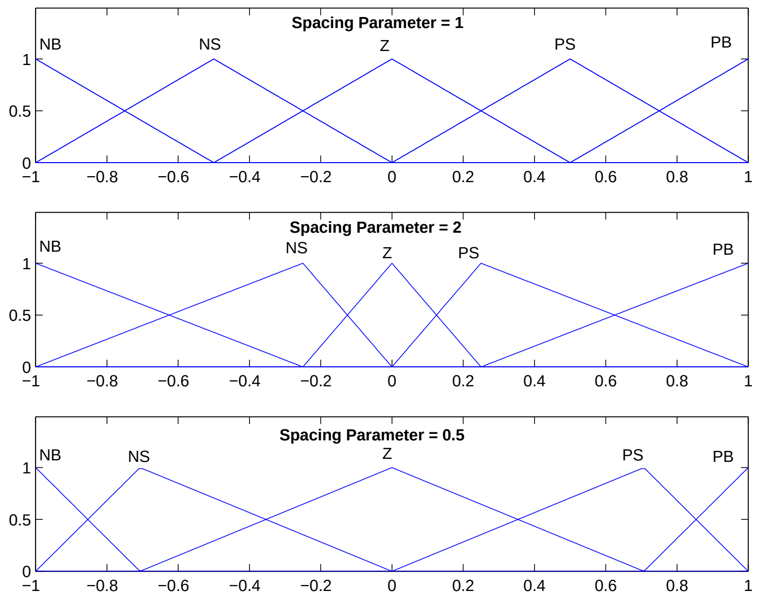


Figure 5-1 Effects of Spacing Parameters on Mfs

This method of designing the membership functions is inspired by the work of Park et al. [8] and Cheong et al [9]. It does mean that there is a reduction in the number of possible FLCs than if the design was fully flexible but the trade-off is that the design process is made much simpler. Also it is felt that even within these constraints there is sufficient flexibility to allow a FLC that meets the design requirements to be built.

5.3 Designing the Rule-Base

As well as specifying the membership functions, the rule-base also needs to be designed. Again ideas presented by Park et al. [8] were used. In specifying a rule base, characteristic spacing parameters for each variable and characteristic angles for each input variable less one are used to construct the rules.

Certain characteristics of the rule-base are assumed in using the proposed construction method:

- Extreme outputs more usually occur when the inputs have extreme values while mid-range outputs generally are generated when the input values are mid-range.
- Similar combinations of input linguistic values lead to similar output values

Using these assumptions the output space is partitioned into different regions corresponding to different output linguistic values. How the space is partitioned is determined by the characteristic spacing parameters and the characteristic angle. The angle determines the slope of a line[1] through the origin on which seed points are placed. The positioning of the seed points is determined by a similar spacing method as was used to determine the centres of the membership functions.

Grid points are also placed in the output space representing each possible combination of input linguistic values. These are spaced in the same way as before. The rule-base is determined by calculating which seed-point is closest to each grid point. The output linguistic value representing the seed-point is set as the consequent of the antecedent represented by the grid point. This is illustrated in Figure 5-2, which is a graph showing seed points (blue circles) and grid-points (red circles).

[1] For two input (2-D) variables the angle of the line with the x-axis needs to be specified. For three inputs (3-D) two angles need to be specified to position the line, and so on. Therefore the number of characteristic angles needed is one less than the number of input variables. To account for the lost degree of freedom note that the line is fixed to pass through the origin.

Table 5-1 shows the derived rule base. The lines on the graph delineate the different regions corresponding to different consequents. The parameters for this example are 0.9 for both input spacings, 1 for the output spacing and 45° for the angle.

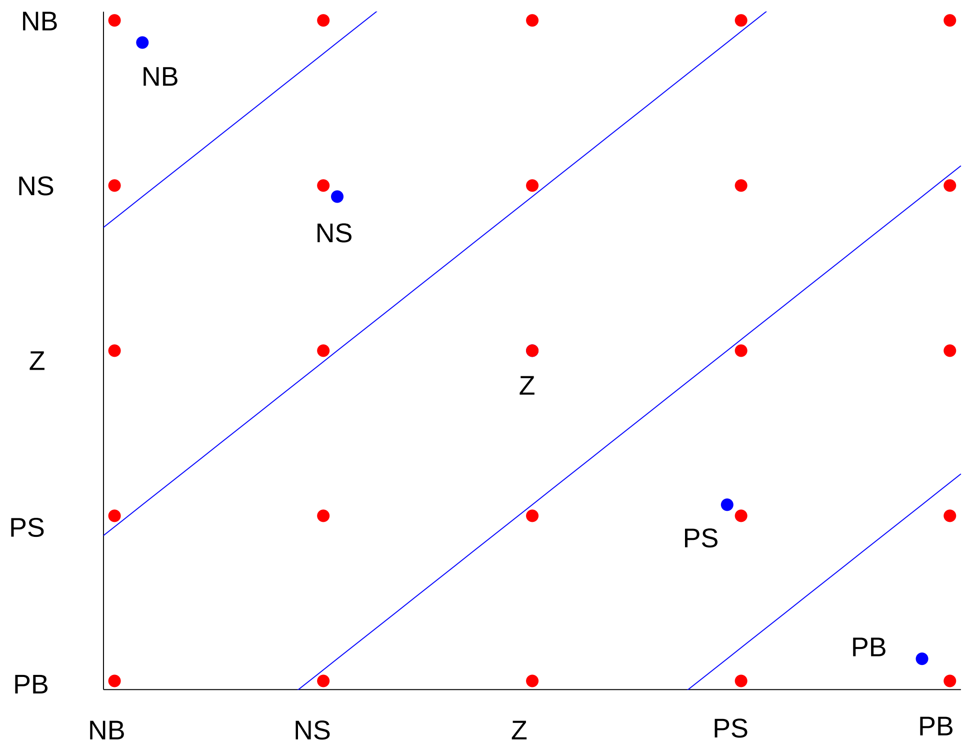


Figure 5-2 Seed Points and Grid Pts For Rule-Base Construction

de \ e	NB	NS	Z	PS	PB
NB	NB	NB	NS	NS	Z
NS	NB	NS	NS	Z	PS
Z	NS	NS	Z	PS	PS
PS	NS	Z	PS	PS	PB
PB	Z	PS	PS	PB	PB

Table 5-1 Derived Rule Base

5.4 Matlab Implementation

To implement these design methodologies in Matlab, .m function files were written. These functions make use of the Fuzzy Logic Toolbox for Matlab in creating the fuzzy logic controller. Two function files *create_mfs.m* and *create_rules.m* respectively create the membership functions and the rule-base. A third function *make_fis.m* puts together the Fuzzy Inference System (FIS) by using these two functions.

The code for *make_fis.m* is listed in Listing 5-1. Firstly, error checking is performed to ensure that the parameters are valid. Once this is done, *create_mfs.m* is called to get the parameters for the membership functions of each of the variables. Then a suitable rule-base is created for each of the output variables by calling *create_rules.m*, which returns a rules matrix in the format required by the Fuzzy Logic Toolbox. This information is then put together in a suitable way to create the FIS. Only triangular membership functions can be created using this function.

```
function the_fis = make_fis(inp,out,n,pm,ps,theta_s)
%MAKE_FIS        Create a FIS based on inputted parameters
%
%the_fis = make_fis(inp, out, n, pm, ps, theta_s)
%inp        Number of input variables
%out        Number of output variables
%n          Column vector of length inp+out containing the number
%           of membership functions per variable
%pm         Column vector of length inp+out indicating how the
%           membership functions are spread for each variable
%ps         Matrix of size inp+1 x out indicating how the rule-
%           base is % formed
%theta_s    Matrix of size inp-1 x out. Each column contains the
%           seed angles for the rule matrices
%
%Created By Joe Foran July 2002

if ~isposint(inp) | ~isposint(out)
   error('inp and out should be positive integer scalars')
end

if size(n) ~= [inp+out 1]
   error('n should be a column vector whose length is equal to the
number of inputs and outputs')
end

if size(pm)~= [inp+out 1]
   error('pm should be a column vector whose length is equal to
the number of inputs and outputs')
end

if size(ps)~= [inp+1 out]
   error('pm should be a column vector whose length is equal to
the number of outputs')
end

if size(theta_s) ~= [inp-1 out]
   error('theta_s should be a matrix of size inp-1 x out')
end
```

```
%Get the membership function parameters
mfpar = zeros(inp+out,3*max(n));

for i = 1:inp+out
   mfpar(i,1:3*n(i)) = create_mfs(n(i),pm(i));
end

%Create the rule matrix
for i = 1:out
   rule_mat = create_rules(inp,[n(1:inp); n(inp+i)], ps(:,i),
theta_s(:,i));
   if i == 1
      rules = rule_mat;
   else
      rules(:,inp+i+1:inp+i+2) = rules(:,inp+i:inp+i+1);
      rules(:,inp+i) = rule_mat(:,end-2);
   end
end

%Create the FIS
the_fis = newfis('fisname');

for i = 1:inp
   varname = ['inp' int2str(i)];
   range = [mfpar(i,2) mfpar(i,n(i)*3-1)];
   the_fis = addvar(the_fis,'input',varname,range);
end
for i = inp+1:inp+out
   varname = ['out' int2str(i-inp)];
   range = [mfpar(i,2) mfpar(i,n(i)*3-1)];
   the_fis = addvar(the_fis,'output',varname,range);
end

%Initially only triangular membership functions are to be allowed.
for i=1:inp
   for j = 1:n(i)
      mfname = int2str(j);
      the_fis = addmf(the_fis,'input',i,mfname,'trimf',
mfpar(i,((j*3)-2):(j*3)));
   end
end
for i = inp+1:inp+out
   for j = 1:n(i)
      mfname = int2str(j);
      the_fis = addmf(the_fis,'output',i-
inp,mfname,'trimf',mfpar(i,((j*3)-2):(j*3)));
   end
end

the_fis = addrule(the_fis, rules);
```

Listing 5-1 make_fis.m

The function that calculates the appropriate parameters for these membership functions, *create_mfs.m* is listed in Listing 5-2. This function takes two parameters, the number of membership functions and the spacing parameter for the centres of these functions. Based on these parameters, the centres of each membership function are calculated. As the base vertices are at the same positions as the centres of adjacent membership functions, calculating the full set of parameters is then a relatively easy task, as only vertex coordinates are required for triangular membership functions. These parameters are returned by the function.

```
function mfparams = create_mfs(no_mfs,p)
%CREATE_MFS Calculate MFparams for a fuzzy variable
%
%Only triangular MFs can be created. The MFs are such that the
%base vertices are coincident with the apexes of adjacent
%triangles.
%
%no_mfs   The number of membership functions
%p        A tuning parameter which indicates how the centres are
%         spaced out. 1 indicates even spacing, <1 indicates
%         compressed at the extremes
%         while >1 indicates compression at the centre

n = (no_mfs-1)/2; %Order of membership functions
if ~isposint(n)
   error('The number of membership functions should be a positive
odd integer greater than 1')
end

if ~(p>0)
   error('p should be a positive number')
end

%Calculate how the centres are spaced
c = zeros(n,1);
for i = 1:n;
   c(i) = (i/n)^p;
end

%Allocate centre positions
centres = zeros(no_mfs,1);

for i = 1:no_mfs
   if i < n+1
      centres(i) = -c(n-i+1);
   elseif i == n+1
      centres(i) = 0;
   else
      centres(i) = c(i-n-1);
   end
end

%Create triangular mfs based on these centres
%The base vertices are coincident with the apexes of
%the adjacent triangle.
pt = zeros(no_mfs,3);

for i = 1:no_mfs
   if i == 1
      pt(i,1) = 2*centres(i)-centres(i+1);
   else
      pt(i,1) = centres(i-1);
   end
   if i ==no_mfs
      pt(i,3) = 2*centres(i)-centres(i-1);
   else
      pt(i,3) = centres(i+1);
   end
   pt(i,2) = centres(i);
end

mfparams = zeros(1,no_mfs*3);
for i = 1:no_mfs
   mfparams(3*i-2:3*i) = pt(i,:);
end
```

Listing 5-2 create_mfs.m

The next piece of code used is *create_rules.m* given in Listing 5-3. This function returns the rule-base based on the parameters passed in. These parameters are the number of membership functions per variable, the spacing parameters for each variable and the characteristic angles for the seed line. Firstly the co-ordinates of the seed points are calculated. The grid-point co-ordinates are then calculated. The consequents for each rule are then generated by, for each grid-point, measuring the distance to each seed point and finding the shortest. The antecedents and consequents are then returned in a matrix in the format required by the Fuzzy Logic Toolbox.

```
   function rulemat = create_rules(inp,no_mfs,p_s,theta_s)
   %CREATE_RULES Create a rule matrix based on the input
   %characteristic parameters
   %
   %rulemat    The rulebase matrix to be used by a FIS
   %
   %inp       Number of input variables
   %no_mfs    Vector of number of membership functions - should
   %          be of length one more than no. of inputs. The last one
   %          is number of output membership functions.
   %p_s       A vector of parameters indicating spacing of seed pts.
   %          and grid pts.
   %theta_s    Specifies slope of seed line.
   %
   %Written by Joe Foran July 2002
   %
   %Last Modified 20-08-02

   if size(inp) ~= [1 1] | ~isposint(inp)
      error('inp should be a positive integer scalar')
   end

   if size(no_mfs) ~= [inp+1 1]
      error('n should be a column vector whose length is one greater
   than the number of input variables')
   end

   if size(theta_s) ~= [inp-1 1]
      error('theta_s should be a column vector whose length is one
   less than the number of input variables')
   end

   if size(p_s) ~= [inp+1 1]
      error('p_s should be a column vector whose length is one
   greater than the number of input variables')
   end

   mfOrder = (no_mfs-1)/2;

   for i = 1:length(mfOrder)
      if ~isposint(mfOrder(i))
         error('The number of output membership fns. should be an odd
   integer of at least value 3')
      end
   end

   outOrder = mfOrder(end);
   % Find positions of seed pts. along seed hyperplane
   c = zeros(outOrder,1);
   for i = 1:outOrder;
      c(i) = (i/outOrder)^p_s(end);
   end

   n_out = no_mfs(end);
   %Get the co-ordinates of the seed points
   %There are n_out seed points and they need to be specified
   %in n-dimensional space, n being equal to inp
   co_ords = zeros(inp,n_out);

   %Specify x-position of co_ords
   for j = 1:n_out
      if j < outOrder+1
         co_ords(1,j) = -c(outOrder-j+1);
      else
         if j == outOrder+1
            co_ords(1,j) = 0;
         else
            co_ords(1,j) = c(j-outOrder-1);
         end
      end
   end

   %To get co-ordinates in other dimensions multiply by tan
   %of appropriate angle
   for k = 2:inp
      co_ords(k,:) = co_ords(1,:)*tan(theta_s(k-1));
   end

   %Normalise the co_ordiantes to be between -1 and 1
   norm_fact = max(max(abs(co_ords)));
   co_ords = co_ords./norm_fact;


   %Get the grid point co-ordinates
   no_rules = prod(no_mfs(1:inp));
   c = zeros(inp,(max(no_mfs(1:inp))-1)/2);
   centres = zeros(inp,max(no_mfs(1:inp)));
   antecedents = zeros(no_rules,inp);
```

```
%Space out grid-points in each dimension
for i = 1:inp
   for j = 1:no_mfs(i);
      c(i,j) = (j/((no_mfs(i)-1)/2))^p_s(i);
   end
end

for i = 1:inp
   for j = 1:no_mfs(i)
      if j < mfOrder(i)+1
         centres(i,j) = -c(i,mfOrder(i)-j+1);
      elseif j == mfOrder(i)+1
         centres(i,j) = 0;
      else
         centres(i,j) = c(i,j-mfOrder(i)-1);
      end

   end
end

%Create each possible combination of antecedents
for i = 1:no_rules
   for j = 1:inp
      if j == inp
         antecedents(i,j) = mod(i,no_mfs(inp));
      else
         antecedents(i,j) =
mod(ceil(i/prod(no_mfs(j+1:inp))),no_mfs(j));
      end

      if antecedents(i,j) ==0
         antecedents(i,j) = no_mfs(j);
      end
   end
end


%Distance from each seed-point of all grid-pts
region = zeros(no_rules,n_out);

point = ones(no_rules,inp); %Co-ordinates of each grid-point

%Calculate distance from each grid-point to each seed-point
for i = 1:no_rules
   for j = 1:inp
      point(i,j)=centres(j,antecedents(i,j));
   end
   for j = 1:n_out
      region(i,j) = sum((point(i,:)-co_ords(:,j)').^2);
   end
end

consequent = zeros(no_rules,1);

%To ensure full rotation of rules adjust the region finding
algorithm according to the regime
temp = mean(theta_s); %Find the average angle
temp = mod(temp,2*pi); %Map back to between 0->360 degrees

%If regime is in left-hand half-plane
if 180*temp/pi>90 & temp*180/pi<=270
```

```
      t2 = -1;
   else
      t2 =1;
   end

   flip =1;
   %Find the region in which each grid_pt lies.
   for i = 1:no_rules
      index = find(region(i,:)==min(region(i,:)));
      if size(index) == [1 1]
         consequent(i) = index;
      else %if grid-point is equidistant from two seed-points
         if flip ==1
            consequent(i) = index(1);
         else
            consequent(i) = index(2);
         end
         flip = -flip;
      end

      if t2 == -1 %Swap over consequents if we are in left-hand half-
   plane
         consequent(i) = n_out+1-consequent(i);
      end
   end


   rulemat = [antecedents consequent ones(no_rules,2)];
```

Listing 5-3 create_rules.m

With all of this code a full FIS can be specified using only a few parameters. This is ideal for using a Genetic Algorithm to find an optimal FLC as the GA can work on these parameters and improve the FLC characteristics. How this is achieved is demonstrated in the next chapter.

6 Using Genetic Algorithms to Design Fuzzy Logic Controllers

In this chapter an attempt is made to apply GAs to achieve satisfactory design of a FLC. It is shown how designs can be evaluated to determine their efficacy. Results are presented from various different experimental runs to show the effectiveness of the approach taken and a resultant FLC is then evaluated more thoroughly.

6.1 Evaluation Functions

To apply a Genetic Algorithm to the design of Fuzzy Logic Controllers, a means of evaluating different designs is required. This evaluation needs to be performed relatively quickly as a GA needs to be able to process large numbers of different combinations of parameters.

Evaluation Functions are functions called by a GA to calculate the fitness of a set of parameters. The parameters are passed to the evaluation function, which processes them and returns a value corresponding to how well the parameters performed the task at hand.

For this project the evaluation function *run_fuzzy_pole_only.m*_was written. This is listed in Listing 6-1. This function firstly extracts the relevant parameters from the chromosome passed in. After performing some error checking, the parameters are used to create a Fuzzy Inference System (FIS) and set the appropriate scaling factors.

A SIMULINK model is then called, from which a record of the error in the pole angle throughout the duration of the simulation is returned. The square of the error is multiplied by a time weight and the sum of this time weighted square error is inverted to give a fitness value. If the pole angle should at any stage of the simulation saturate, i.e. reach ±90°, then the simulation is stopped immediately so that time is not wasted modelling controllers that fail to balance the pole. In this case the fitness is calculated as the time taken before the pole collapsed multiplied by 1x10^9. This gives a reward to designs that manage to keep the pole from collapsing that little bit longer.

```
function [chrom, eff] = run_fuzzysim(chrom,options)
%RUN_FUZZYSIM This is the evaluation function for the fuzzy
simulation used by the GA
%This runs a 2 input one output function
%
%chrom    The real-valued chromosome passed in%
%eff      The calculated fitness value

%Written by:     Joe Foran July 2002
%
%Last Modified: Joe Foran 07/08/2002

n = chrom(1:3)';
pm = (chrom(4:6).^chrom(10:12))';
pr = (chrom(7:9).^chrom(13:15))';
scale = chrom(16:18)';
theta = chrom(19);
inp = 2;
out = 1;

if size(n) ~= [inp+out 1]
   error('n should be a column vector whose length is equal to the
number of inputs and outputs')
end

if size(pm)~= [inp+out 1]
   error('pm should be a column vector whose length is equal to
the number of inputs and outputs')
end

if size(pr)~= [inp+1 out]
   error('pm should be a column vector whose length is equal to
the number of outputs')
end

if size(theta) ~= [inp-1 out]
   error('theta_s should be a matrix of size inp-1 x out')
end

if size(scale) ~= [inp+out 1]
   error('scale should be a column vector whose length equals the
number of inputs and outputs')
end

global FUZZY_SCALING INVPEND_FUZZY

FUZZY_SCALING = scale;

INVPEND_FUZZY = make_fis(inp,out,n,pm,pr,theta);

t = 0:0.01:6;

stable = 1;

eval ('sim(''s_invpen_fuzzy_dist'',t);','stable = 0;');
if stable == 0 | s_Time(end)~= t(end)
```

```
      eff = 1e-9*s_Time(end)*stable;
   else
      eff = 1/((s_Error.^2)'*s_Time);
   end
```

Listing 6-1 Evaluation Function run_fuzzy_pole_only.m

A diagram of the SIMULINK model that is used is shown in Figure 6-1. The desired angle of the pole is 0°, i.e. upright. The error in the angle and the change of error are scaled by the appropriate gains (these parameters are also set by the GA) and the result is clipped so that it lies in the range –1 to 1. These inputs are fed into the FLC and the FLC's output is then scaled by another gain. This force is applied to the system, which in turn, outputs the pendulum angle, angular speed, cart position and cart velocity.

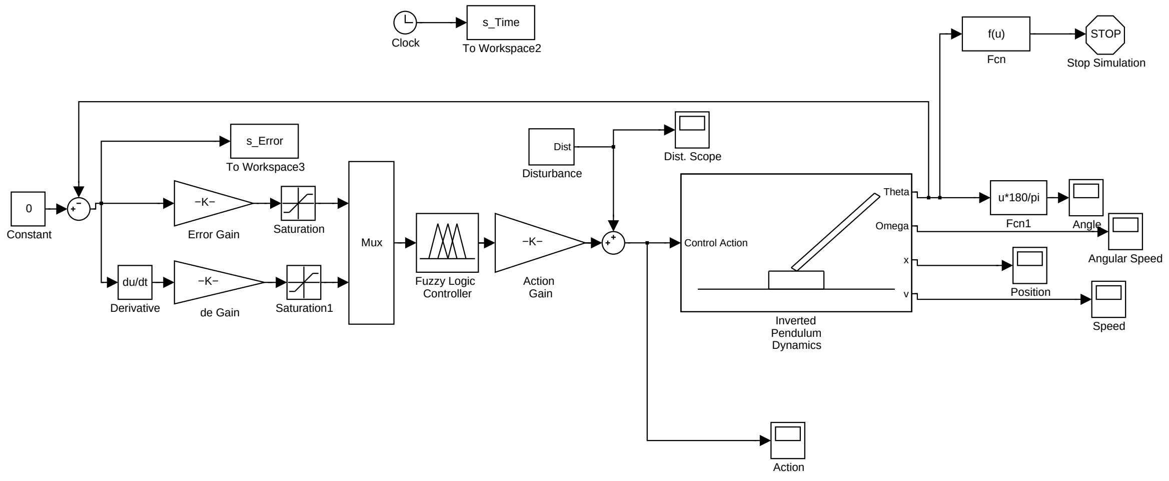


Figure 6-1 Simulink Model used to evaluate FLCs

A periodic disturbance is also applied to the system. This is a force of relatively large magnitude, which ensures that the arena in which the FLC is examined is a testing one. This disturbance is of the magnitude of 8kN and has a duration of 10ms (which is also the step time of the simulation) and is applied every two seconds, in alternately positive and negative directions. This disturbance perturbs the pendulum enough to make controlling the system a sufficiently challenging task. Each FLC is tested over three cycles, i.e. for six seconds. It is felt that this is enough time for an adequately large range of the system's properties to be examined.

6.2 Parameter Encoding

To run a GA, a suitable encoding for each of the parameters and bounds for each of them needs to be decided. For this task the parameters given in Table 6-1 are used with the shown ranges and precisions. Binary encoding is used as it is felt that this allows the GA more to search the solution space more thoroughly.

Parameter	Range	Precision	No. of Bits
Number of Membership Functions	3 –9	2	2
Membership Function Spacing	0.1 – 1.0	0.01	7
MF Spacing (Power to be Raised by)	-1 – 1	2	1
Rule-Base Scaling	0.1 – 1.0	0.01	7
Rule-Base Spacing (Power to be Raised by)	-1 – 1	2	1
Input Scaling	0 – 100	0.1	10
Output Scaling	0 – 10000	0.1	17
Rule-Base Angle	$0 – 2\pi$	$\pi/512$	11

Table 6-1Parameters used for encoding

The numbers of membership functions are limited to the odd integers inclusive between three and nine. As most FLCs reported in the literature are within this range [12], this was felt to be a reasonable constraint to apply. The advantage of doing this is that this information can be captured in just two bits per variable.

For the spacing parameters, two separate parameters are used. The first, with the range [0.1 – 1.0], determines the magnitude and the second, which takes only the values –1 or 1, is the power by which the magnitude is to be raised. This determines whether the membership functions compress in the centre or at the extremes. The precision required for the magnitude is 0.01, meaning that 8 bits are used in total for each spacing parameter.

The scaling for the input variables is allowed to vary in the range [0 – 100], while that of the output variable is given the range [0 – 10,000]. These values were decided after a few trials of the GA using wider ranges as the values returned were found to lie in these ranges.

For this encoding scheme the total number of bits per individual is 102. This means that there are 2^{102} or approximately $5x10^{30}$ potential solutions, an unknown but likely very small fraction of which represents viable controllers. This indicates that the solution space is large despite the constraints that are introduced. If GAs succeed in finding close to optimal solutions in such a large space despite having no prior knowledge this would indicate their power.

6.3 Running the GA

Run	Pop. Size	Selection Routine	Crossover Routine	Crossover Probability	Mutation Probability	Best Fitness
1	150	Stoc. Rem	Uniform	1.0	0.01	1.1597
2	50	Stoc. Rem	Uniform	1.0	0.01	0.9105
3	100	Stoc. Rem	Uniform	1.0	0.01	0.9449
4	100	Stoc. Rem	Single Pt.	1.0	0.01	1.1171
5	100	Stoc. Rem	Uniform	0.5	0.01	1.1851
6	50	Stoc. Rem	Uniform	1.0	0.05	0.8880
7	50	Roulette	Uniform	1.0	0.01	1.0925
8	50	Roulette	Uniform	1.0	0.001	0.0141
9	50	Stoc. Rem	Single Pt.	0.5	0.05	1.2525
10	50	Stoc. Rem	Uniform	1.0	0.01	0.8937

Table 6-2 Parameters for GA Runs with FLC

The file in which the GA parameters are set and which calls the main GA routine is *ga_poleonly.m.* The source code for this file is given in Appendix A.9 ga_poleonly.mThis file calls the Matlab functions *run_ga.m* and *initGa.m*, which are modified versions of the routines provided with the GAOT toolbox. The code for these files is listed in Appendices A.10 and A.11 respectively. The reason modified code is used is

- to eliminate some small bugs discovered in the process of converting from binary to real values and vice-versa (see also Appendix A.20) and
- to change the way the progress of the GA was output to the screen.

An elitist model was used for all runs of the GA also.

Ten runs of the GA were performed for a hundred generations each. The details of these runs are listed in Table 6-2. Again the parameters that were varied were selection routine, crossover routine, crossover probability and mutation probability. The best controller was found during Run 9. This run was continued for a further two hundred generations, but the best fitness did not increase much. A plot of the progress of this GA run is shown in Figure 6-2.

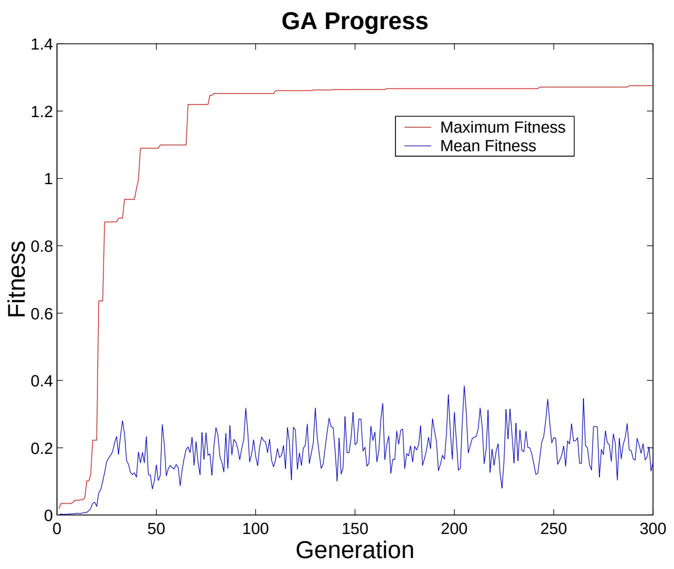


Figure 6-2 Progress of GA that found best fitness

Details of the FLC found by the GA are presented in Figure 6-3, Table 6-3 and Figure 6-4. The Error variable was assigned seven membership functions, whereas both the error derivative and control actions were assigned nine membership functions. For both of the inputs, the membership functions are compressed towards the centre, while they were compressed at the extremes for the output. The control surface shown in Figure 6-4 shows that the mapping between the input and output spaces is highly linear. An advantage of FLCs over linear controllers is that they allow the creation of these kinds of mappings; the control surface of a two-input linear controller, no matter how well designed, will always be a plane in three-dimensional space [2].

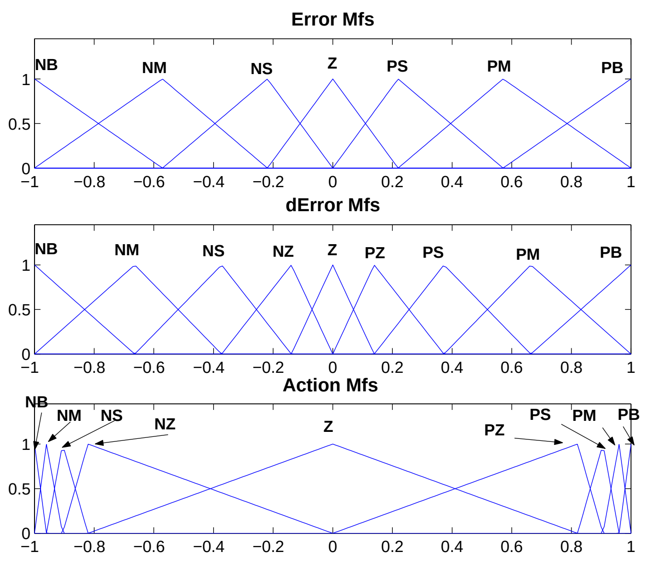


Figure 6-3 Membership functions of Optimised FLC

		Rate of Change of Error								
		NB	NM	NS	NZ	Z	PZ	PS	PM	PB
Error	NB	NB	NB	NB	NB	NM	NM	NS	NZ	NZ
	NM	NB	NM	NS	NS	NZ	NZ	NZ	NZ	NZ
	NS	NS	NZ	NZ	NZ	NZ	NZ	NZ	NZ	Z
	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z
	PS	Z	PZ	PZ	PZ	PZ	PZ	PZ	PZ	PS
	PM	PZ	PZ	PZ	PZ	PZ	PS	PS	PM	PB
	PB	PZ	PZ	PS	PM	PM	PB	PB	PB	PB

Table 6-3 Rule-Base of Optimised FLC

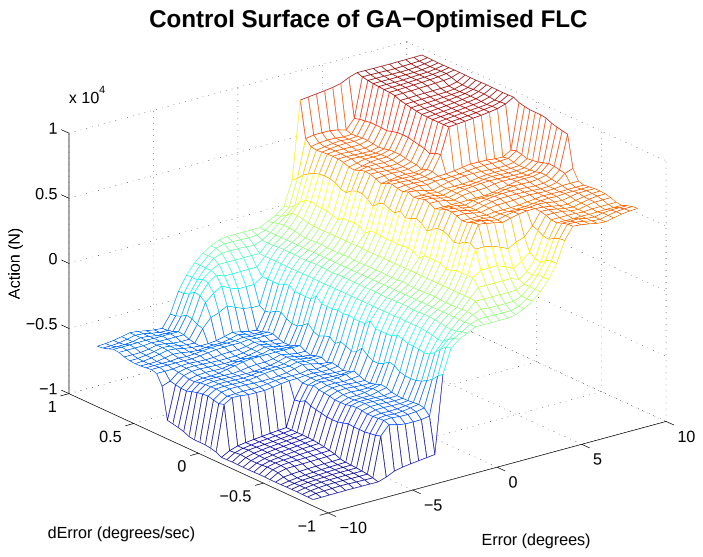


Figure 6-4 Surface Plot of Optimised FLC (Scaled)

In Figure 6-5 the impulse responses of this FLC and that of the linear controller examined in Chapter 2 are compared. It can be seen that while the pole collapses when controlled by the linear controller, it hardly moves when the FLC is used.

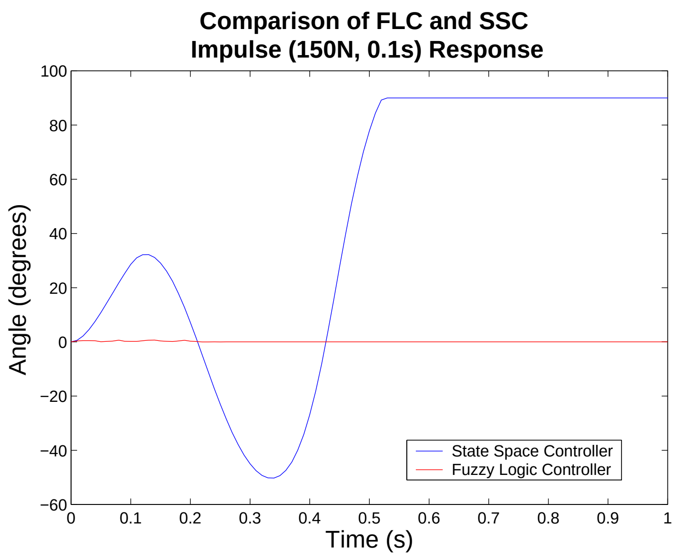


Figure 6-5 FLC and SSC Impulse Response Comparison

The response of the controller to a larger impulse is shown in Figure 6-6. This time the pole does move, but it is quickly brought under control and settles to the equilibrium position in about 0.1s.

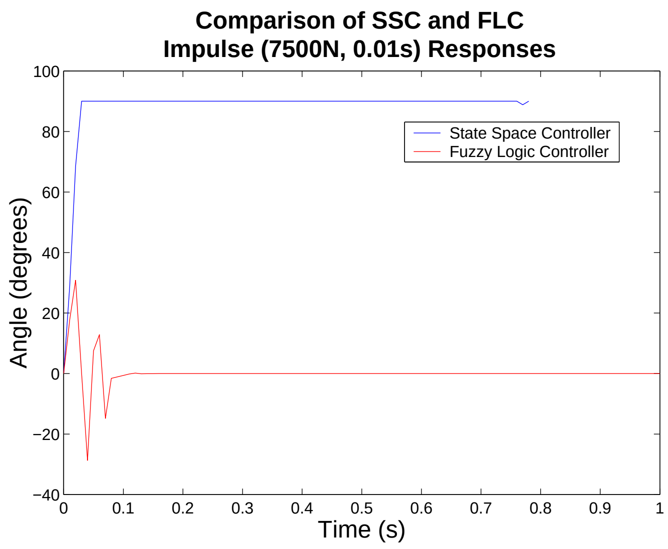


Figure 6-6 Response To Large Impulse

This FLC is also successful at bringing the pole to its equilibrium position from a large range of initial angles as shown in Figure 6-7. Interestingly, the settling time is quicker for the initial angle of -89° than for -30°. This suggests that the FLC is better designed when the deviations are large and is not so well designed at smaller perturbations. This is further emphasised by comparing how it and the State-Space Controller balance a system with an initial angle of 5° as shown in Figure 6-8. To obtain a better controller, the evaluation may have to include smaller impulses as well as the larger ones used here.

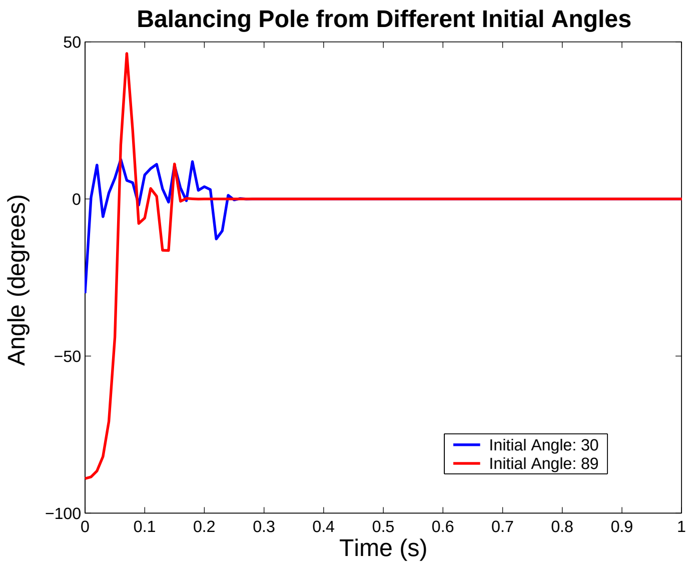


Figure 6-7 Bringing Pole to Upright Position

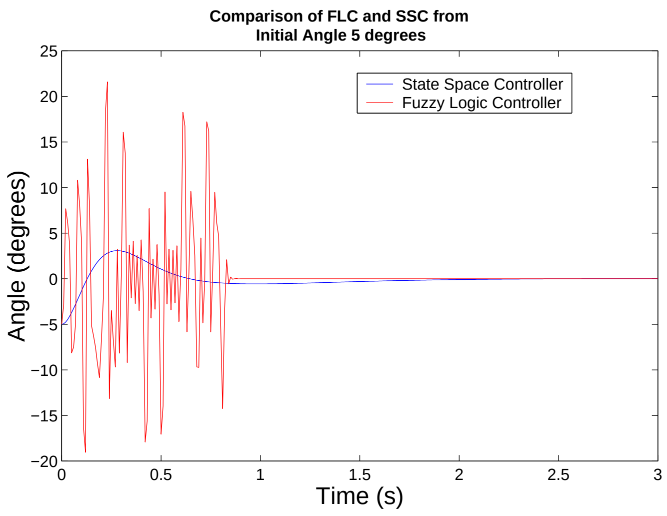


Figure 6-8 Comparison of FLC and SSC from Initial Angle of 5°

6.4 Controlling Both Pole Angle and Cart Position

These algorithms were run with the cart being allowed to move an infinite distance in either direction, i.e., no effort was made to control the cart's position. As this would be impractical if a real controller were to be built, an attempt was made to use the Genetic Algorithm to find a Fuzzy Controller that could control both the angle and cart position.

To this end, the Fuzzy Logic Controller used would have to have four input variables, the error and change in error for both angle and position. It was found that the time taken to evaluate the fitness for one individual was approximately one minute for five seconds of simulation time (This was on a machine whose processor speed was 1.3 GHz). Thus, it took one hour to evaluate a single generation. This length of time was considered impractical and so this approach was abandoned. The reason the time had increased so much was as there was now four input variables, each being generally a member of two fuzzy sets, there was 2^4 rules fired per time-step. The extra time required to infer the control action slowed the simulation significantly.

A new method was then tried. This time two two-input fuzzy controllers were used, one for the angle and one for the position. Their output actions were then added to give the actual force applied to the system. It was found that using the 8kN disturbance as used in the previous runs to test the controllers was too onerous a task and no controllers of significant fitness were found. The requirement was reduced to coping with a disturbance impulse of 1 kN and a controller that met this requirement was discovered by the GA. The response of this controller to this 1kN impulse is plotted in Figure 6-9. The controller in this case does keep the system close to the desired state, though there are still oscillations about this position.

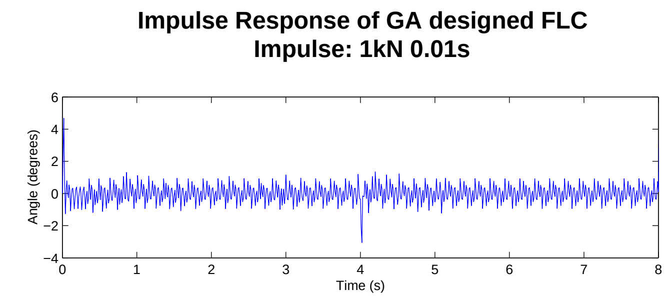


Figure 6-9 Response to 1kN disturbance

The response to a disturbance input of 150N was also plotted and is shown in Figure 6-10. Even though the impulse was reduced, the controller still fails to remove the oscillations from the system.

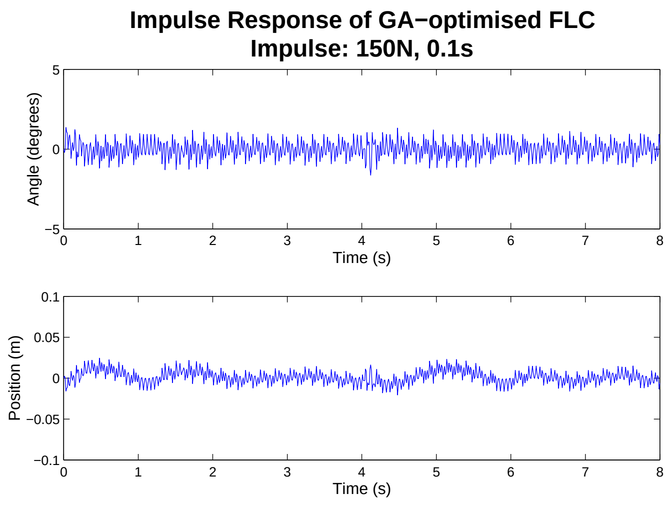


Figure 6-10 Response to 150N disturbance impulse

The combinations of FLC designed here is not an ideal controller. A four input FLC would probably work better but the time taken to find a good one through GAs is significant. Much work may have to be done to achieve this.

7 Online Genetic Algorithms

Online GAs applied to controllers are Genetic Algorithms that attempt to adjust the parameters of a controller during operation. When successfully implemented, they enable continuous improvement in performance. However, as they are operating on a real system, care must be taken to ensure that no action is imparted to the system that could cause damage. For this reason, online GAs are more difficult to design and are less flexible than offline GAs which only work on models of a system and can therefore handle inappropriate actions [17].

7.1 SIMULINK S-Functions

To design an online GA, the first step taken was to try to apply it in SIMULINK. No pre-existing software tools were available so the code for this needed to be written. This code was written in the form of S-functions, which are functions that allow the creation of user-defined blocks in SIMULINK [18].

Each block in SIMULINK has the following general characteristic. A vector of inputs is applied to the system and these inputs interact with the system states, also in vector form, to generate a vector of outputs.

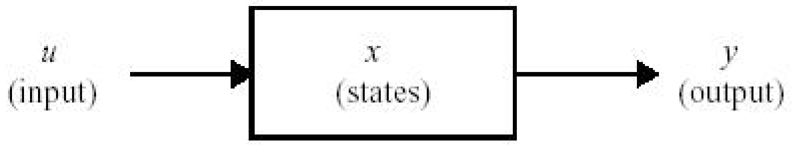


Figure 7-1 Simulink Block Characteristics

To create S-Functions, the user must write code that instructs SIMULINK on how to update the states and how to generate outputs. Therefore, the code written must have a special form. Each S-Function generally consists of sub-functions that initialise states, generate outputs and update state information. The only memory that the S-Function has is contained in the state vector so it is important that any information that needs to be carried over from time-step to time-step is contained here. The size of the state vector is static which also introduces another constraint on how the code can be designed.

7.2 Implementing a GA in SIMULINK

With all this in mind, an s-function was written to create a GA block for SIMULINK. This code is listed in Appendix A.12 The parameters passed into this block by the user are population size, a matrix of bounds for each variable, a vector of bits specifying how many bits are to represent each variable and the time allocated to each individual. The input to the block is the fitness of the set of parameters outputted in the previous time-step, while the block output is the set of chromosome parameters to be evaluated as well as details of the generation number and optimum fitness found so far.

In the code to initialise the model, listed below, each bit of every individual is a system state. States are allocated for a tracker of the optimum individual found up to the present time as well. Another three states are also used, a generation counter, an individual counter and a store for the number of bits per individual. This is stored as a state so that it doesn't need to be recalculated every simulation step.

```
function [sys,x0,str,ts] =
mdlInitializeSizes(popSize,bounds,bits,period)

numVar = size(bounds,1);
initPop = initga(popSize,bits);
numGenes = size(initPop,2);
gen = 1; %Generation counter

sizes = simsizes;
sizes.NumContStates  = 0;
sizes.NumDiscStates  = (popSize+1)*numGenes+3;
sizes.NumOutputs     = numVar+3;
sizes.NumInputs      = 1;
sizes.DirFeedthrough = 0;
sizes.NumSampleTimes = 1;

sys = simsizes(sizes);

initPop = reshape(initPop',popSize*numGenes,1); %Make initPop a vector

x0  = zeros(sizes.NumDiscStates,1);
x0(1:popSize*numGenes) = initPop; %Place initPop into state vector
x0(end-2) = numGenes; %Store this value so it doesn't have to be
constantly recalculated
x0(end-1) = gen; %Place generation counter into state vector;
x0(end) = 1; %Tracks which chromosome is being evaluated


str = [];
ts  = [period 0];
```

Listing 7-1 Initialising State Information

The code for updating the system states is presented in Listing 7-2. Firstly, the input is checked to make sure it is positive and if it isn't, is set to a low fitness value. This is to prevent fitness values of zero, which would cause division by zero in later routines. The input fitness is then assigned to the previous chromosome and compared to the present optimum, which is updated if necessary. If the GA has reached the end of a generation then the population is updated using selection, crossover and mutation. The states of the system are then updated with details of the new population.

```
function [sys] = mdlUpdate(t,x,u,popSize)
input = u;
if u<=0
   input =1e-9;
end
numGenes = x(end-2);
cc = x(end);

%Get current population into matrix form
pop = reshape(x(1:(popSize+1)*numGenes),numGenes,popSize+1)';

current_optimum = pop(end,end);

%The fitness in input refers to the previous chromosome
if cc>1 %This is because output occurs before update
   pop(cc-1,end)=input;%Read in input into the fitness value of the
                       %previous chromosome
   if input>current_optimum %Save new best solution
      pop(end,:) = pop(cc-1,:);
   end
end

x(end) = x(end)+1; %Increment chromosome counter

if x(end) > popSize+1 %End of a generation
   if max(pop(:,end-1))<current_optimum %If the optimum was lost in
this generation
      [wval windx] = min(pop(:,end-1)); %Replace the worst of this
generation
      pop(windx,:) = pop(end,:); %With the optimum
   end
   newPop = create_next_generation(pop(1:end-1,:)); %find next
generation
   pop = [newPop; pop(end,:)];
   x(end) = 1;   %Start again at beginning of next generation
   x(end-1) = x(end-1)+1; %Increment Generation Counter
end

x(1:(popSize+1)*numGenes) = reshape(pop',numGenes*(popSize+1),1);
%Transform matrix back into vector

sys = x;

%end mdlUpdate
```

Listing 7-2 Updating GA s-function

The functions that perform the GA operations are listed below. The selection routine used was Stochastic Remainder Selection and uniform crossover was performed with 100% probability. A mutation rate of 10% was also specified. The code specifies these routines and probabilities at present, but it could be easily modified to have them specified by the user using input parameters to the block.

```
function newPop = create_next_generation(oldPop)
[popSize, nVars] = size(oldPop);

%Select using stochastic Remainder Selection
newPop = stocRemSelec(oldPop);

%Crossover using mask crossover
list = rand(popSize,1); %Generate a list to decide how crossovers are
%paired

for i = 1:floor(popSize/2)
   [tmp,ind] = max(list(:,1));
   a = newPop(ind,:);
   list(ind) = 0;
   [tmp,ind] = max(list(:,1));
   b = newPop(ind,:);
   list(ind) = 0;
   [c d]  = maskXover(a,b);
   newPop(i*2-1,:) = c;
   newPop(i*2,:) = d;
end

%Mutate
for i = 1:popSize
   newPop(i,:) = binMutation(newPop(i,:),0.01);
end

%end of create_next_generation
```

Listing 7-3 S_Function GA Operation

The last major part of the S-Function is the output routine (Listing 7-4). In this part of the code, the binary representation of the chromosome to be evaluated is converted into a real-valued representation. The present optimum fitness, the current generation number and the current individual number are also outputted for the user's information.

```
function sys = mdlOutputs(t,x,u,popSize,bounds,bits)

numGenes = x(end-2);
generation = x(end-1);
cc = x(end);
optimum = x((popSize+1)*numGenes);
pop = reshape(x(1:(popSize+1)*numGenes),numGenes,popSize+1)';
chrom = b2f(pop(cc,1:end-1),bounds,bits);
```

```
%Output current chromosome, best chromosome so far, best fitness so far
and current generation number
sys = [chrom cc optimum generation];
```

Listing 7-4 S-Function Output Routine

The reader may have noted that the s-function as presented outputs the chromosome corresponding to the optimum at the end of each generation. The reason this is done is to synchronise the assignment of the fitness of the last chromosome so that it occurs before the GA operations are performed.

The GA S-function thus created is then applied to the optimisation of a FLC as shown in Figure 7-2. The parameters outputted from the GA are input into another S-Function (listed in Appendix A.13), which creates the FLC described by the parameters and uses this controller to calculate a control action based on information fed back from the plant. The square of the error is calculated and it is multiplied by a time-weight. This is fed into an integration block which sums up the total time-weighted error. After a unit delay this is then fed into another sub-system that calculates the inverse. This sub-system is a triggered one and it outputs the calculated inverse sum of time-weighted error back to the GA as the fitness. The reason for the delay is to ensure that the fitness fed into the GA is the correct value. If this were not done the fitness input to the GA would be zero corresponding to the beginning of a new cycle. A disturbance impulse is used to reset the integrators and to trigger the sub-system so that everything is synchronized with the GA.

When running this model initially it was found that unless the angle and angular velocity were reset to zero at the beginning of each cycle, then the GA wouldn't be able to find a good controller as it found it too hard to lift the pole from its horizontal state. However, if the angle and angular velocity are being reset, then the behaviour of this system is more like an offline GA as the ability to reset the system would not be present in a real-life application. Therefore a different approach to creating an online GA is taken, and this is outlined in the next section. The work done here is not in vain however, as a block for performing offline GAs in SIMULINK has been created which wasn't available beforehand.

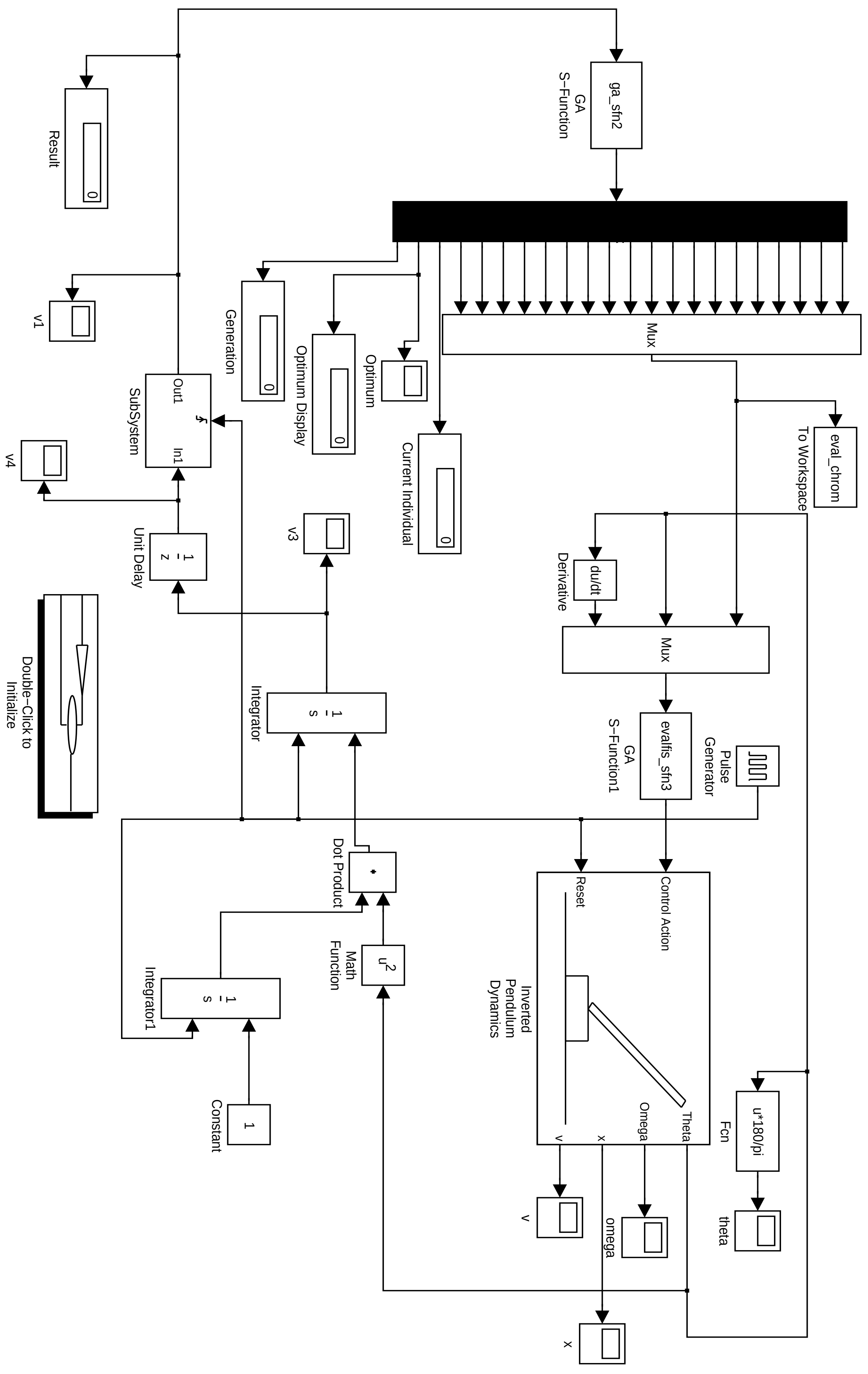


Figure 7-2 Simulink Model for GA

7.3 Genetic-Based Machine Learning

In an effort to achieve a working online genetic algorithm, the use of Genetic-Based Machine Learning (GBML) was studied. More specifically, classifier systems, common GBML architectures, were investigated to see if they could help in the search for a continually self-improving controller.

A major feature of classifier systems[1] is that each individual classifier consists of a message and an action. Each message is compared to a message from the environment (in the case of controllers this could represent the state of the system) and those that match compete in an auction to have their associated action become the output. The amount each classifier bids is proportional to its strength, which increases when it is successful and decreases when it competes in auctions and loses. A winner is chosen and their action is performed. If the action has a beneficial effect then a reward is paid to the winner, who becomes stronger and hence has a better chance of winning subsequent auctions. In this way, those classifiers whose actions are the most suitable responses to the states that their messages represent should become stronger and dominate the population.

To apply this idea to a fuzzy logic controller, the classifier population was initialised to be every possible combination of antecedents and consequents for a fuzzy system. Each classifier was given the same initial strength. As code to implement this was not immediately available, the Pascal code presented by Goldberg [1] was converted to Matlab code and then inserted into an S-Function. Code was also written to fuzzify the inputs and de-fuzzify the outputs of the system. The code written for all this is listed in Appendices A.14 to A.19.

A SIMULINK model was created to implement this classifier system and is shown in Figure 7-3. A reference value of 0 degrees is used and periodic disturbances are applied as before. The error and error derivative are input into a S-Function block *fuzzifyOnline.m* which converts the real-numbered variables to fuzzy variables. The four fuzzy sets are passed to the classifier system for an auction to determine suitable output rules. These output rules are combined with the degrees of firing calculated by the fuzzification block to produce a control action.

[1] For a more detailed discussion of how GBML and classifier systems operate, please refer to Goldberg [1]

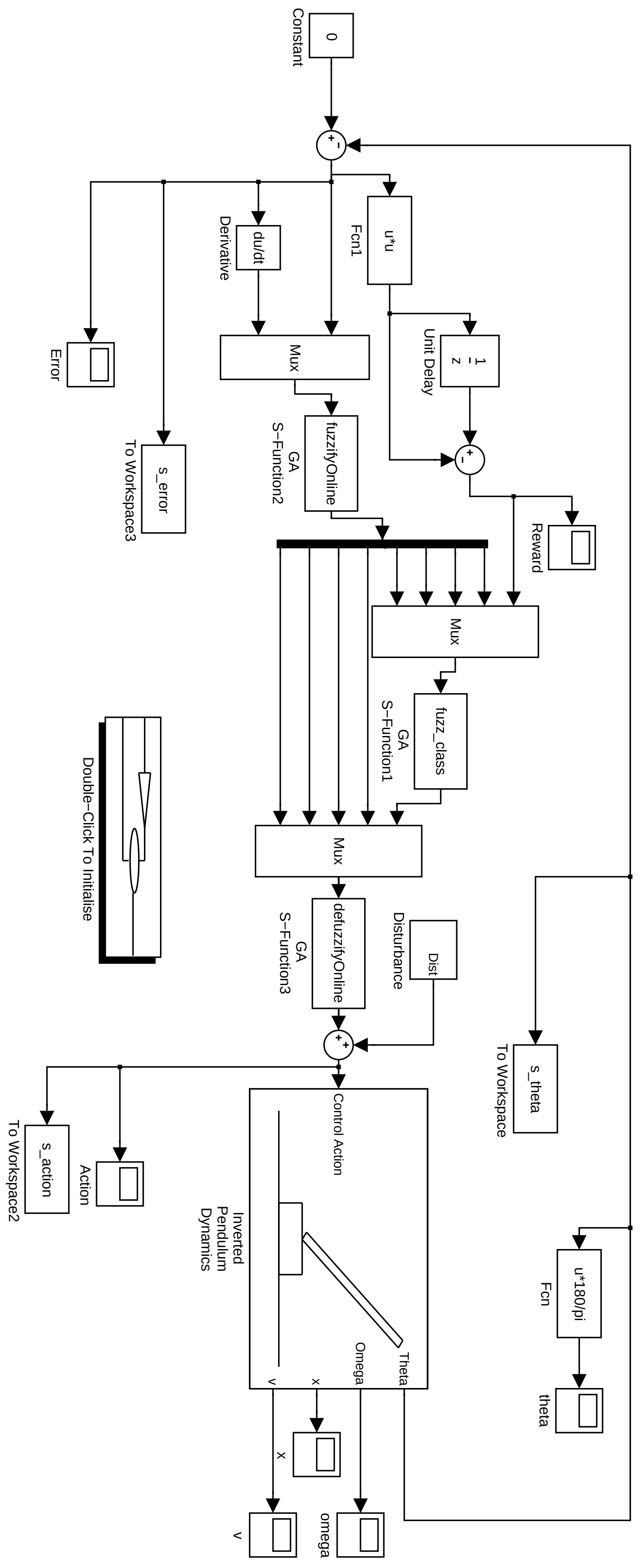


Figure 7-3 SIMULINK Implementation of Fuzzy Classifier System

The error in the system is compared with that of the previous time-step and used to calculate the reward, which is passed to the classifier S-Function Block so that the appropriate classifier can have its strength increased. The membership functions and scaling for all variables were ones found using an offline GA. Also the disturbance impulse applied was a very small one, being 0.1N for 0.01s every second.

When this model was run, it was found to have some beneficial effect but with lots of room for improvement. In the early stages, the pole swung dramatically around (the limits on the poles movement to the upper half of the plane was removed). It then settled to a steady state value as shown in Figure 7-4. When the impulse was increased to 100N the classifier system still failed to remove the steady state error (see Figure 7-5).

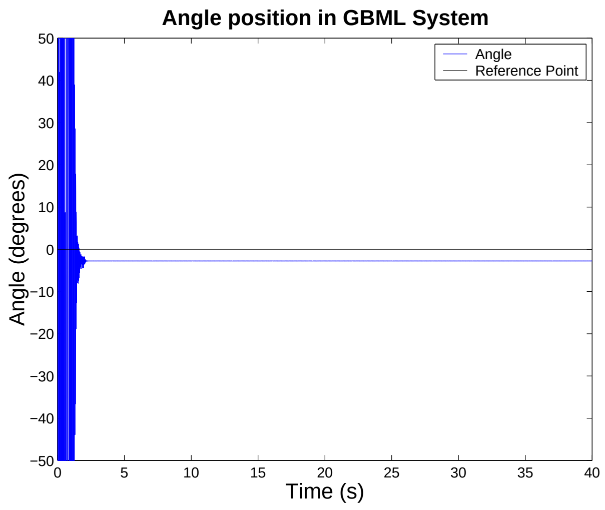


Figure 7-4 GBML Progress (Disturbance Impulse 0.1N 0.01s)

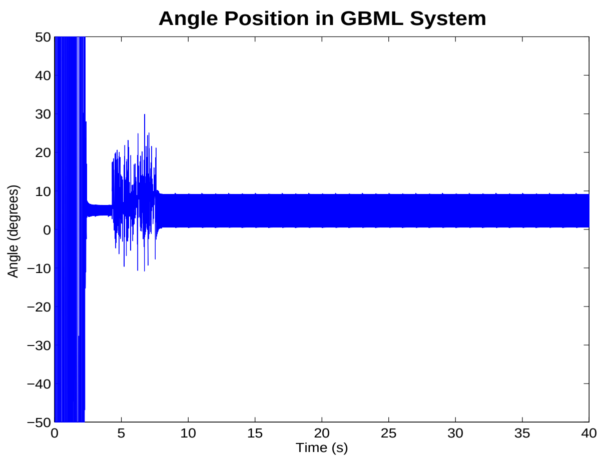


Figure 7-5 GMBL Progress (Impulse Disturbance 100N, 0.01s)

Examining a close-up, Figure 7-6, we see that the angle oscillates between approximately 1 and 9 degrees, showing that there is still some work to be done to improve this system. It is felt that the system of deciding the reward passed to the classifiers is too simplistic and that a more sophisticated algorithm may be required. Also more investigation may be needed into the parameters used for the classifiers such as what proportion of a classifier's strength is bid in the auctions.

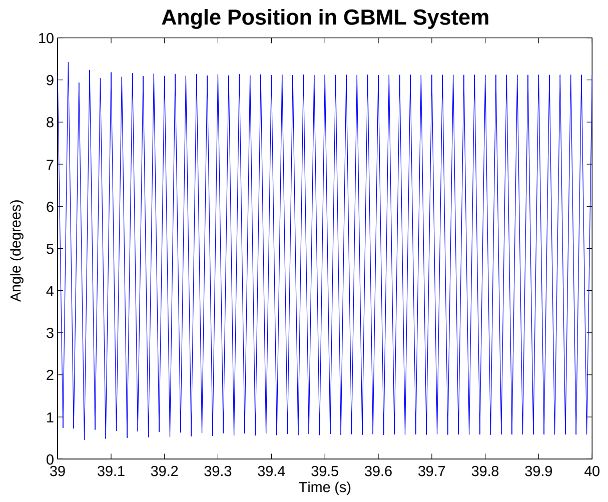


Figure 7-6 Close up of Figure 7-5

Unfortunately due to time constraints, it was not possible to investigate these matters further before the project was due to be completed. However, should there be a continuation of this work, enough may have been done to provide a foundation for further investigation and improvements.

It must also be noted that no action was taken in developing this system to ensure that no damage was inflicted on the plant. For an online GA to be successfully implemented, this defect needs to be remedied.

8 Conclusion

The main objective of this project was to find a way to optimise a Fuzzy Logic Controller using Genetic Algorithms and this was successfully achieved. However, the attempt to do this using online GAs was less successful. This effort was not in vain and some insights arose from it.

8.1 Suggestions for Future Work

During this project, a broad investigation into applying Genetic Algorithms to Fuzzy Logic Controllers was carried out. As well as successfully keeping the pole upright, an attempt to control both pole angle and cart position was made. Though the results from this effort were not entirely satisfactory, progress was made. Progress was also made in the quest to apply an online GA to the system.

Suggestions for follow-up work that may come after this project are:

- Perform a more in-depth study of how the various parameters and encodings affect GA performance in FLC optimisation. This work attempted here was too broad to allow enough time for more investigations of this kind.
- Investigate more thoroughly if GAs can be applied to FLCs or other types of non-linear controllers so that they may successfully control fully the inverted pendulum system. This may then lead eventually to the building of a real-life controller. Perhaps by limiting the range of the parameters using heuristics, GAs with faster performance may be found.
- Search for a more sophisticated algorithm to pass reward to successful classifiers so that a successful online GA may be built.
- Investigate how damaging control actions may be prevented from affecting the plant when building such a system. One possibility may be to process an offline GA in parallel using a model that is updated using System Identification techniques and only trying actions that the offline GA suggests would be successful.
- Investigate whether any of the assumptions made in designing FLCs such as using only triangular membership functions or only allowing an odd number of

these sets have a significant impact on the performance obtainable from the controllers.

- Attempt to optimise some of the code written here by using techniques such as vectorisation so that these algorithms can run quicker.

8.2 Conclusions

Genetic Algorithms have been shown to be powerful search tools that can reduce the time and effort involved in designing systems for which no systematic design procedure exists. They can quickly find close-to-optimal solutions and if set-up well can avoid local optima. They are certainly useful tools when trying to solve analytically difficult problems. However, they are only as good as the model they are presented with. In other words, if the model of the problem to be solved is not realistic, then a good solution should not be expected.

Fuzzy Logic Controllers can provide more effective control of non-linear systems than linear controllers, as there is more flexibility in designing the mapping from the input to the output space. Whereas expert knowledge is usually required to design a fuzzy controller using traditional methods, it has been shown in this report that even without using any knowledge of the system, GAs can build an effective controller relatively quickly. This technique may lead an increase in the use of FLCs as the previously time-consuming design procedure can be reduced dramatically.

It has also been shown that it is possible to use Genetic-Based Machine Learning to build an intelligent controller, intelligent in this sense meaning one that can improve its own performance as it progresses. More work is needed in this area but it is felt that a good foundation has been laid.

References

[1] Goldberg, D.E. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[2] Passino, K.M., Yurkovich, S., *Fuzzy Control* Addison Wesley Longman, 1997

[3] Friedland, B. *Control System Design* 2nd Ed., McGraw Hill 1986.

[4] Meriam, J.L., and Kraige, L.G., *Engineering Mechanics,* 3rd Ed., J. Wiley & Sons, 1993.

[5] Chipperfield, A., Fleming, P., Pohlheim, H., Fonseca, C., *Genetic Algorithm Toolbox User's Guide Version 1.2,* Dept of Automatic Control and Systems Engineering, University of Sheffield, 1995

[6] University of Michigan *Control Tutorials for Matlab* http://www.engin.umich.edu/group/ctm/index.html

[7] C. R. Houck, J. Joines. and M.Kay. A genetic algotithm for function optimisation: A Matlab implementation. ACM Transactions on Mathematical Software, 1996 http://www.eos.ncsu.edu/eos/service/ie/research/kay_res/GAToolBox/gaot

[8] Park, Y. J., Cho, H.S., Cha, D.H., *Genetic Algorithm-Based Optimization of Fuzzy Logic Controller Using Characteristic Parameters,* Proceedings of the 1995 IEEE International Conference on Evolutionary Computation, pp 831-836

[9] Cheong, F., Lai, R., *Constraining the Optimization of a Fuzzy Logic Controller Using an Enhanced Genetic Algorithm,* IEEE Transactions on Systems, Man and Cybernetics-Part B: Cybernetics, Vol 30, No.1, Feb 2000

[10] Pavel, O., Radomil, M. *Automatic Optimal Design of Fuzzy Controllers Based on Genetic Algorithms* IEE Conference Publication 446, Genetic Algorithms in Engineering Systems: Innovations and Applications, 2-4 Sept 1997.

[11] Herrera, F., Lozano, M., Verdegay, J. L., *Tuning Fuzzy Logic Controllers by Genetic Algorithms* International Journal of Approximate Reasoning 1995; 12:299-315

[12] Lee, M.A., Takagi, *Integrating Design Stages of Fuzzy Systems Using Genetic Algortithms,* Proc. 2nd IEEE Int. Conf. Fuzzy Systems, San Francisco, 1993; 612-617

[13] Belarbi K; Titel F, *Genetic algorithm for the design of a class of fuzzy controllers: An alternative approach,* IEEE Transactions On Fuzzy Systems 2000, Vol 8, Iss 4, pp 398-405

[14] Zadeh, L.A., *Fuzzy Sets,* Information and Control, 1965, Vol 8 pp338-353

[15] Whitley, *A Genetic Algorithm Tutorial,* Technical Report CS-93-103, Dept. of Computer Science, Colorado State University, 1993.

[16] Linkens, D.A., Nyongesa, H.O., *Genetic Algorithms for Fuzzy Control. Part 1: Offline System Development and Application* IEEE Proc.- Control Theory Appl., Vol. 142 No.3 May 1995 pp161-176

[17] Linkens, D.A., Nyongesa, H.O., *Genetic Algorithms for Fuzzy Control. Part 1: Online System Development and Application* IEEE Proc.- Control Theory Appl., Vol. 142 No.3 May 1995 pp177-185

[18] MathWorks Inc., The *Using S-Functions*, Version for Simulink 3.0, Oct. 1998

Appendix A Matlab Script and Function Files

Appendix A.1 Invpenstspace.m

```
%This script file creates the state-space matrices and using the lqr function
%creates a controller for the linearised inverted pendulum model.

%This is a modification of work found at
%http://www.engin.umich.edu/group/ctm/examples/pend/invSS.html#lqr
%
%Last Modified: Joe Foran 06-08-02

%First set the parameters of the system.
M = 1; %Mass of cart
m = 0.1; %Mass of pendulum
l = 0.5; %Distance from pivot to centre of mass of pendulum
i = (4/3)*m*l^2; %Moment of Inertia of pendulum
g = 9.81; %Acceleration due to gravity


p = (M+m)*i-(m*l)^2; %denominator

%Create the state-space matrices
A = [ 0       1                     0                      0;
       0      0 ((m*l)^2)*g/p       0;
      0       0                     0                      1;
      0 0 (M+m)*m*g*l/p       0];
B = [0;
    i/p;
     0
   m*l/p];
C = [1 0 0 0;
   0 0 1 0];
D = [0;0];

%Set relative weights of Q matrix
x = 5000;
y = 5000;
Q = [x 0 0 0;
   0 0 0 0;
   0 0 y 0;
   0 0 0 0];
R = 1;

%Design controller
K = lqr(A,B,Q,R)
Ac = [(A-B*K)];
Bc = [B];
Cc = [C];
Dc = [D];

%find Nbar to eliminate the steady state error
Nbar = rscale(A,B,Cn,0,K)
%The function rscale was created by the original authors and is available
%at http://www.engin.umich.edu/group/ctm/extras/rscale.html
```

Appendix A.2 gaMultimax.m

```
%This scriptfile sets the parameters, initialises the population, runs the genetic
%algorithm and collects the results for the test function multimax
echo off
```

```
clear

objFun = 'multimax';
ngens = 100;
lb = [0]';         %Lower Bound on the Population
ub = [10]';         %Upper Bound on the Population
prec = [0.0001]';
bounds = [lb ub];
bits = calcbits2(bounds,prec);

evOpts = [];
popSize = 50;    %Number of individuals in population

epsilon = 1e-6; % change required to consider two solutions different
prob_ops = 0;   % 0 if you want to apply the genetic operators probabilisticly to each
solution, 1 if
% you are supplying a deterministic number of operator applications
display =0;     % 1 to output progress 0 for quiet.
opts = [epsilon prob_ops display];

termFn = 'maxGenTerm'; %Termination Function
termOps = ngens; %No of Generations to be run for in case of maxGenTerm
selectFn = 'stocRemSelec'; %name of the .m selection function (['normGeomSelect'])
selectOps = [0.08]; %options for select Function
xoverFn = ['maskXover']; % a string containing blank seperated names of Xover.m files
% (['arithXover heuristicXover simpleXover'])
xoverOps = [1 0]; % A matrix of options to pass to Xover.m files with the
% first column being the number of that xOver to perform similiarly for mutation
mutFn = 'binaryMutation'; % a string containing blank seperated names of mutation.m
files
%(['boundaryMutation multiNonUnifMutation nonUnifMutation unifMutation'])
mutOps = [0.1 ngens 3]; % A matrix of options to pass to Xover.m files with the
% first column being the number of that xOver to perform
% similiarly for mutation


%Create a random initial population for the GA
initPop=initGa(popSize,bounds,bits,objFun, evOpts,opts(1:2));

%Run the GA
[x endPop bpop trace] =
run_ga(bounds,bits,objFun,evOpts,initPop,opts,termFn,termOps,...
   selectFn,selectOps,xoverFn,xoverOps,mutFn,mutOps);

%Plot the progress of the fitness
hold off
plot(trace(:,1),trace(:,2),'r-');
hold on
plot(trace(:,1),trace(:,3),'b--');
xlabel('Generation');
ylabel('Fitness');
title('Progress of GA over time');
popSizeText = num2str(popSize);
pXoverText = num2str(xoverOps(1),4+log10(xoverOps(1)));
pMutText = num2str(mutOps(1),4+log10(mutOps(1)));
if strcmp(selectFn, 'roulette')
   selectFnText = 'Roulette';
elseif strcmp(selectFn,'stocRemSelec')
   selectFnText = 'Stoc. Rem.';
else
   selectFnText = selectFn;
end

if strcmp(xoverFn, 'maskXover')
   xoverFnText = 'Uniform';
elseif strcmp(xoverFn, 'simpleXover')
   xoverFnText = 'Single Pt.';
else
   xoverFnText = xoverFn;
end

plotText{1} = ['Population:       ' popSizeText];
plotText{2} = ['Selection Method: ' selectFnText];
plotText{3} = ['Xover Method:     ' xoverFnText];

plotText{4} = ['Xover Prob:       ' pXoverText];
plotText{5} = ['Mutation Prob:    ' pMutText];
```

```
text(ngens/2,13,char(plotText),'FontName','FixedWidth');

if 0 %Comment out next section
   load GAresults.mat
   GA(end+1).gen = bpop(end-1,1);
   GA(end).best = bpop(end,end);
   save GAresults.mat GA
end
```

Appendix A.3 StocRemSelec.m

```
function newPop = stocRemSelec(oldPop,options)
%STOCREMSELEC Stochastic Remainder Selection a la Goldberg p.123
%
%This algorithm returns a population the same size as the one passed in.
%
%newPop    The new population selected from the old one
%
%oldPop    The population from which the new Population is to be selected
%options   Options to be passed to selection algorithms.
%          Not used for this algorithm

popSize = size(oldPop,1);
expec = oldPop(:,end)/mean(oldPop(:,end)); %Expectation of selection
intexp = floor(expec); %Integer portion of expectation
fracexp = expec - intexp; %Fractional portion of expectation.

choice = zeros(popSize,1); %Holds record of selected individuals
count = 1; %Record of which position is to be filled.

%allocate Integer portions
for j = 1:popSize
   alloc = intexp(j);
   while alloc >0
      choice(count) = j;
      alloc = alloc -1;
      count = count+1;
   end
end

%Allocate fractional portions
nremain = popSize - count+1; %Number of places remaining

while nremain >0
   randNums = rand(popSize,1); %Generate random numbers
   selec = fracexp > randNums; %Compare to fract. exp. of selection for each ind.
   numSelec = sum(selec); %Number selected this round

   if numSelec>nremain %If more were selected than places are available
      selec(:,2) = cumsum(selec); %This gives a different ticket to each entry
      randNums = rand(numSelec,1);
      [tmp ind] = sort(randNums); %Choose the winning tickets at random
      for i = 1:nremain %Find the winners and gieve them their prize
         choice(count) = find((selec(:,2) == ind(i))& (selec(:,1) == 1));
         fracexp(ind(i)) = 0;
         count = count+1;
      end
      nremain = 0;
   else %Otherwise allocate places to all those selected
      nremain = nremain-numSelec;
      for i = 1:popSize
         if selec(i) == 1
            choice(count) = i;
            count = count+1;
            fracexp(i) = 0;
         end
      end
   end
end

%Shuffle the selected population
[ans, shuf] = sort(rand(popSize,1));
newPop= oldPop(choice(shuf),:);
```

Appendix A.4 maskXover.m

```
function [c1,c2] = maskXover(p1,p2,bounds,Ops)
% Mask crossover takes two parents P1,P2 and performs
% uniform crossover using a random mask.
%
% function [c1,c2] = simpleXover(p1,p2,bounds,Ops)
% p1      - the first parent ( [solution string function value] )
% p2      - the second parent ( [solution string function value] )
% bounds  - the bounds matrix for the solution space
% Ops     - Options matrix for simple crossover [gen #SimpXovers].

%Written by Joe Foran 19 - 07 -2002
%
numVar = size(p1,2)-1;                       % Get the number of variables

mask1 = round(rand(1,numVar)); %Generate a mask
mask2 = ~mask1;                                      %Generate its complement



c1 = zeros(1,numVar+1); %Allocate space for children
c2 = zeros(1,numVar+1);

%Generate children
c1(1:numVar) = (p1(1:end-1).*mask1+p2(1:end-1).*mask2);
c2(1:numVar) = (p1(1:end-1).*mask2+p2(1:end-1).*mask1);
```

Appendix A.5 make_fis.m

```
function the_fis = make_fis(inp,out,n,pm,ps,theta_s)
%MAKE_FIS        Create a FIS based on inputted parameters
%
%the_fis = make_fis(inp, out, n, pm, ps, theta_s)
%inp        Number of input variables
%out        Number of output variables
%n          Column vector of length inp+out containing the number
%           of membership functions per variable
%pm         Column vector of length inp+out indicating how the membership
%           functions are spread for each variable
%ps         Matrix of size inp+1 x out indicating how the rule-base is formed
%theta_s    Matrix of size inp-1 x out. Each column contains the seed angles for the
rule matrices

if ~isposint(inp) | ~isposint(out)
   error('inp and out should be positive integer scalars')
end

if size(n) ~= [inp+out 1]
   error('n should be a column vector whose length is equal to the number of inputs
and outputs')
end

if size(pm)~= [inp+out 1]
   error('pm should be a column vector whose length is equal to the number of inputs
and outputs')
end

if size(ps)~= [inp+1 out]
   error('pm should be a column vector whose length is equal to the number of
outputs')
end

if size(theta_s) ~= [inp-1 out]
   error('theta_s should be a matrix of size inp-1 x out')
end

%Get the membership function parameters
mfpar = zeros(inp+out,3*max(n));

for i = 1:inp+out
```

```
   mfpar(i,1:3*n(i)) = create_mfs(n(i),pm(i));
end

%Create the rule matrix
for i = 1:out
   rule_mat = create_rules(inp,[n(1:inp); n(inp+i)],ps(:,i),theta_s(:,i));
   if i == 1
      rules = rule_mat;
   else
      rules(:,inp+i+1:inp+i+2) = rules(:,inp+i:inp+i+1);
      rules(:,inp+i) = rule_mat(:,end-2);
   end
end

%Create the FIS
the_fis = newfis('fisname');

for i = 1:inp
   varname = ['inp' int2str(i)];
   range = [mfpar(i,2) mfpar(i,n(i)*3-1)];
   the_fis = addvar(the_fis,'input',varname,range);
end
for i = inp+1:inp+out
   varname = ['out' int2str(i-inp)];
   range = [mfpar(i,2) mfpar(i,n(i)*3-1)];
   the_fis = addvar(the_fis,'output',varname,range);
end

%Initially only triangular membership functions are to be allowed.
for i=1:inp
   for j = 1:n(i)
      mfname = int2str(j);
      the_fis = addmf(the_fis,'input',i,mfname,'trimf',mfpar(i,((j*3)-2):(j*3)));
   end
end
for i = inp+1:inp+out
   for j = 1:n(i)
      mfname = int2str(j);
      the_fis = addmf(the_fis,'output',i-inp,mfname,'trimf',mfpar(i,((j*3)-2):(j*3)));
   end
end

the_fis = addrule(the_fis, rules);
```

Appendix A.6 create_mfs.m

```
function mfparams = create_mfs(no_mfs,p)
%CREATE_MFS Calculate MFparams for a fuzzy variable
%
%Only triangular MFs can be created. The MFs are such that the base vertices
%are coincident with the apexes of adjacent triangles.
%
%no_mfs       The number of membership functions
%p               A tuning parameter which indicates how the centres are spaced out
%             1 indicates even spacing, <1 indicates compressed at the extremes
%             while >1 indicates compression at the centre

n = (no_mfs-1)/2; %Order of membership functions
if ~isposint(n)
   error('The number of membership functions should be a positive odd integer greater
than 1')
end

if ~(p>0)
   error('p should be a positive number')
end

%Calculate how the centres are spaced
c = zeros(n,1);
for i = 1:n;
   c(i) = (i/n)^p;
end

%Allocate centre positions
centres = zeros(no_mfs,1);
```

```
for i = 1:no_mfs
   if i < n+1
      centres(i) = -c(n-i+1);
   elseif i == n+1
      centres(i) = 0;
   else
      centres(i) = c(i-n-1);
   end
end

%Create triangular mfs based on these centres
%The base vertices are coincident with the apexes of
%the adjacent triangle.
pt = zeros(no_mfs,3);

for i = 1:no_mfs
   if i == 1
      pt(i,1) = 2*centres(i)-centres(i+1);
   else
      pt(i,1) = centres(i-1);
   end
   if i ==no_mfs
      pt(i,3) = 2*centres(i)-centres(i-1);
   else
      pt(i,3) = centres(i+1);
   end
   pt(i,2) = centres(i);
end

mfparams = zeros(1,no_mfs*3);
for i = 1:no_mfs
   mfparams(3*i-2:3*i) = pt(i,:);
end
```

Appendix A.7 create_rules.m

```
function rulemat = create_rules(inp,no_mfs,p_s,theta_s,plot_on)
%CREATE_RULES Create a rule matrix based on the input characteristic parameters
%
%rulemat    The rulebase matrix to be used by a FIS
%
%inp                  Number of input variables
%no_mfs        Vector of number of membership functions - should be of length one more
than no. of inputs
%                            The last one is number of output membership functions.
%p_s                  A vector of parameters indicating spacing of seed pts. and grid
pts.
%theta_s              Specifies slope of seed line.
%
%Written by Joe Foran July 2002
%
%Last Modified 20-08-02

if size(inp) ~= [1 1] | ~isposint(inp)
   error('inp should be a positive integer scalar')
end

if size(no_mfs) ~= [inp+1 1]
   error('n should be a column vector whose length is one greater than the number of
input variables')
end

if size(theta_s) ~= [inp-1 1]
   error('theta_s should be a column vector whose length is one less than the number
of input variables')
end

if size(p_s) ~= [inp+1 1]
   error('p_s should be a column vector whose length is one greater than the number of
input variables')
end

n_out = no_mfs(end);
```

```
n = (n_out-1)/2;

if ~isposint(n)
   error('The number of output membership fns. should be an odd integer of at least 
value 3')
end

% Find positions of seed pts. along seed hyperplane
c = zeros(n,1);
for i = 1:n;
   c(i) = (i/n)^p_s(end);
end

%Get the co-ordinates of the seed points
%There are n_out seed points and they need to be specified
%in n-dimensional space, n being equal to inp
co_ords = zeros(inp,n_out);

%Specify x-position of co_ords
for j = 1:n_out
   if j < n+1
      co_ords(1,j) = -c(n-j+1);
   else
      if j == n+1
         co_ords(1,j) = 0;
      else
         co_ords(1,j) = c(j-n-1);
      end
   end
end

%To get co-ordinates in other dimensions multiply by tan
%of appropriate angle
for k = 2:inp
   co_ords(k,:) = co_ords(1,:)*tan(theta_s(k-1));
end

%Normalise the co_ordiantes to be between -1 and 1
norm_fact = max(max(abs(co_ords)));
co_ords = co_ords./norm_fact;


%Get the grid point co-ordinates
no_rules = prod(no_mfs(1:inp));
c = zeros(inp,(max(no_mfs(1:inp))-1)/2);
centres = zeros(inp,max(no_mfs(1:inp)));
antecedents = zeros(no_rules,inp);

%Space out grid-points in each dimension
for i = 1:inp
   for j = 1:no_mfs(i);
      c(i,j) = (j/((no_mfs(i)-1)/2))^p_s(i);
   end
end

for i = 1:inp
   for j = 1:no_mfs(i)
      n = (no_mfs(i)-1)/2;
      if j < n+1
         centres(i,j) = -c(i,n-j+1);
      elseif j == n+1
         centres(i,j) = 0;
      else
         centres(i,j) = c(i,j-n-1);
      end

   end
end

%Create each possible combination of antecedents
for i = 1:no_rules
   for j = 1:inp
      if j == inp
         antecedents(i,j) = mod(i,no_mfs(inp));
      else
         antecedents(i,j) = mod(ceil(i/prod(no_mfs(j+1:inp))),no_mfs(j));
      end

      if antecedents(i,j) ==0
         antecedents(i,j) = no_mfs(j);
      end
   end
end


region = zeros(no_rules,n_out); %Distance from each seed-point of all grid-pts
point = ones(no_rules,inp); %Co-ordinates of each grid-point

%Calculate distance from each grid-point to each seed-point
for i = 1:no_rules
   for j = 1:inp
      point(i,j)=centres(j,antecedents(i,j));
   end
   for j = 1:n_out
      region(i,j) = sum((point(i,:)-co_ords(:,j)').^2);
   end
end

consequent = zeros(no_rules,1);

%To ensure full rotation of rules adjust the region finding algorithm according to
%the regime
temp = mean(theta_s); %Find the average angle
temp = mod(temp,2*pi); %Map back to between 0->360 degrees

if 180*temp/pi>90 & temp*180/pi<=270 %If regime is in left-hand half-plane
   t2 = -1;
else
   t2 =1;
end

flip =1;
%Find the region in which each grid_pt lies.
for i = 1:no_rules
   index = find(region(i,:)==min(region(i,:)));
   if size(index) == [1 1]
      consequent(i) = index;
   else %if grid-point is equidistant from two seed-points
      if flip ==1
         consequent(i) = index(1);
      else
         consequent(i) = index(2);
      end
      flip = -flip;
   end

   if t2 == -1 %Swap over consequents if we are in left-hand half-plane
      consequent(i) = n_out+1-consequent(i);
   end
end


rulemat = [antecedents consequent ones(no_rules,2)];
```

Appendix A.8 run_fuzzy_pole_only

```
function [chrom, eff] = run_fuzzysim(chrom,options)
%RUN_FUZZYSIM This is the evaluation function for the fuzzy simulation used by the GA
%                                This runs a 2 input one output function
%
%chrom                The real-valued chromosome passed in
%
%eff                          The calculated fitness value

%Written by:    Joe Foran July 2002
%
%Last Modified: Joe Foran 07/08/2002
n = chrom(1:3)';
pm = (chrom(4:6).^chrom(10:12))';
pr = (chrom(7:9).^chrom(13:15))';
```

```
scale = chrom(16:18)';
theta = chrom(19);
inp = 2;
out = 1;

if size(n) ~= [inp+out 1]
   error('n should be a column vector whose length is equal to the number of inputs
and outputs')
end

if size(pm)~= [inp+out 1]
   error('pm should be a column vector whose length is equal to the number of inputs
and outputs')
end

if size(pr)~= [inp+1 out]
   error('pm should be a column vector whose length is equal to the number of
outputs')
end

if size(theta) ~= [inp-1 out]
   error('theta_s should be a matrix of size inp-1 x out')
end

if size(scale) ~= [inp+out 1]
   error('scale should be a column vector whose length equals the number of inputs and
outputs')
end

global FUZZY_SCALING INVPEND_FUZZY

FUZZY_SCALING = scale;
INVPEND_FUZZY = make_fis(inp,out,n,pm,pr,theta);

t = 0:0.01:3;
stable = 1;
eval ('sim(''s_invpen_fuzzy'',t);','stable = 0;');

if stable == 0 | s_Time(end)~= t(end)
   eff = 1e-9*s_Time(end)*stable;
else
   eff = 1e4/(s_Error.^2)'*s_TimeWeight;
end
```

Appendix A.9 ga_poleonly.m

```
%Script File to set bounds and parameters for offline GA optimisation of
%FLC for Inverted Pendulum System - this only tries to balance the pole
%without regard for the cart position.

global FUZZY_SCALING INVPEND_FUZZY %Global variables needed by SIMULINK model
load runDetails.mat %Structure run has different parameters for different runs

for i = 1:size(run,2)

   popSize = run(i).popSize;
   selectFn = run(i).sRoutine;
   xoverFn = run(i).Xroutine;
   xOverRate = run(i).xRate;
   mutProb = run(i).mRate;

   objFun = 'run_fuzzy_pole_only_2';
   ngens = 100;
   lb = [3 3 3 0.1 0.1 0.1 0.1 0.1 0.1 -1 -1 -1 -1 -1 -1 0 0 0 0]';         %Lower
Bound on the Population
   ub = [9 9 9 1 1 1 1 1 1 1 1 1 1 1 1 100 100 10000 2*pi]';         %Upper Bound on
the Population
   prec = [2 2 2 0.01 0.01 0.01 0.01 0.01 0.01 2 2 2 2 2 2 0.1 0.1 0.1 pi/512]';
   bounds = [lb ub];
   bits = calcbits2(bounds,prec);
   evOpts = 1;
   %popSize = 60;    %Number of individuals in population

   epsilon = 1e-6; % change required to consider two solutions different
   prob_ops = 0;   % 0 if you want to apply the genetic operators probabilisticly to
each solution, 1 if
   % you are supplying a deterministic number of operator applications
   display =1;     % 1 to output progress 0 for quiet.
   details = 0;
   opts = [epsilon prob_ops display details];
   termFn = 'maxGenTerm'; %Termination Function
   termOps = ngens; %No of Generations to be run for in case of maxGenTerm
   %selectFn = 'stocRemSelec'; %name of the .m selection function (['normGeomSelect'])
   selectOps = [0.08]; %options for select Function
   %xoverFn = ['maskXover']; % a string containing blank seperated names of Xover.m
files
   % (['arithXover heuristicXover simpleXover'])
   xoverOps = [xOverRate 0]; % A matrix of options to pass to Xover.m files with the
   % first column being the number of that xOver to perform similiarly for mutation
   mutFn = 'binaryMutation'; % a string containing blank seperated names of mutation.m
files
   %(['boundaryMutation multiNonUnifMutation nonUnifMutation unifMutation'])
   mutOps = [mutProb ngens 3]; % A matrix of options to pass to Xover.m files with the
   % first column being the number of that xOver to perform
   % similiarly for mutation

   %Initialise the GA
   initPop=initGa(popSize,bounds,bits,objFun, evOpts,opts);

   %Now let's run the ga
   [x endPop bpop trace] =
run_ga(bounds,bits,objFun,evOpts,initPop,opts,termFn,termOps,...
      selectFn,selectOps,xoverFn,xoverOps,mutFn,mutOps);

   traceRecord(i).pop = endPop;
   traceRecord(i).bestPop = bpop;
   traceRecord(i).tracedetails = trace;
   save traceRecord.mat traceRecord;
end
```

Appendix A.10 run_ga.m

```
function [x,endPop,bPop,traceInfo] =
run_ga(bounds,bits,evalFN,evalOps,startPop,opts,...
   termFN,termOps,selectFN,selectOps,xOverFNs,xOverOps,mutFNs,mutOps)
% RUN_GA run a genetic algorithm
% function [x,endPop,bPop,traceInfo]=ga(bounds,evalFN,evalOps,startPop,opts,
%                                     termFN,termOps,selectFN,selectOps,
%                                     xOverFNs,xOverOps,mutFNs,mutOps)
%
% Output Arguments:
%   x            - the best solution found during the course of the run
%   endPop       - the final population
%   bPop         - a trace of the best population
%   traceInfo    - a matrix of best and means of the ga for each generation
%
% Input Arguments:
%   bounds       - a matrix of upper and lower bounds on the variables
%   evalFN       - the name of the evaluation .m function
%   evalOps      - options to pass to the evaluation function ([NULL])
%   startPop     - a matrix of solutions that can be initialized
%                  from initialize.m
%   opts         - [epsilon prob_ops display] change required to consider two
%                  solutions different, prob_ops 0 if you want to apply the
%                  genetic operators probabilisticly to each solution, 1 if
%                  you are supplying a deterministic number of operator
%                  applications and display is 1 to output progress 0 for
%                  quiet. ([1e-6 1 0])
%   termFN       - name of the .m termination function (['maxGenTerm'])
%   termOps      - options string to be passed to the termination function
%                  ([100]).
%   selectFN     - name of the .m selection function (['normGeomSelect'])
%   selectOpts   - options string to be passed to select after
%                  select(pop,#,opts) ([0.08])
%   xOverFNS     - a string containing blank seperated names of Xover.m
%                  files (['arithXover heuristicXover simpleXover'])
%   xOverOps     - A matrix of options to pass to Xover.m files with the
%                  first column being the number of that xOver to perform
%                   similiarly for mutation ([2 0;2 3;2 0])
%   mutFNs       - a string containing blank seperated names of mutation.m
%                   files (['boundaryMutation multiNonUnifMutation ...
%                            nonUnifMutation unifMutation'])
%   mutOps       - A matrix of options to pass to Xover.m files with the
%                   first column being the number of that xOver to perform
%                   similiarly for mutation ([4 0 0;6 100 3;4 100 3;4 0 0])

% Binary and Real-Valued Simulation Evolution for Matlab
% Copyright (C) 1996 C.R. Houck, J.A. Joines, M.G. Kay
%
% C.R. Houck, J.Joines, and M.Kay. A genetic algorithm for function
% optimization: A Matlab implementation. ACM Transactions on Mathmatical
% Software, Submitted 1996.
%
% This program is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 1, or (at your option)
% any later version.
%
% This program is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
% GNU General Public License for more details. A copy of the GNU
% General Public License can be obtained from the
% Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

%%$Log: ga.m,v $
%Revision 1.10  1996/02/02  15:03:00  jjoine
% Fixed the ordering of imput arguments in the comments to match
% the actual order in the ga function.
%
%Revision 1.9  1995/08/28  20:01:07  chouck
% Updated initialization parameters, updated mutation parameters to reflect
% b being the third option to the nonuniform mutations
%
%Revision 1.8  1995/08/10  12:59:49  jjoine
%Started Logfile to keep track of revisions
%


n=nargin;
if n<2 | n==6 | n==10 | n==12
   disp('Insufficient arguements')
end
if n<3 %Default evalation opts.
   evalOps=[];
end
if n<5
   opts = [1e-6 1 0 0];
end
if isempty(opts)
   opts = [1e-6 1 0 0];
end

if any(evalFN<48) %Not using a .m file
   if opts(2)==1 %Float ga
      e1str=['x=c1; c1(xZomeLength)=', evalFN ';'];
      e2str=['x=c2; c2(xZomeLength)=', evalFN ';'];
   else %Binary ga
      e1str=['x=b2f(endPop(j,:),bounds,bits); endPop(j,xZomeLength)=',...
            evalFN ';'];
      end
   else %Are using a .m file
      if opts(2)==1 %Float ga
         e1str=['[c1 c1(xZomeLength)]=' evalFN '(c1,[gen evalOps]);'];
         e2str=['[c2 c2(xZomeLength)]=' evalFN '(c2,[gen evalOps]);'];
      else %Binary ga
         %         e1str=['x=b2f(endPop(j,:),bounds,bits);[x v]=' evalFN ...
         %               '(x,[gen evalOps]); endPop(j,:)=[f2b(x,bounds,bits) v];'];
         e1str=['x=b2f(endPop(j,:),bounds,bits);[x v]=' evalFN ...
               '(x,[gen evalOps]); endPop(j,end)= v;'];  %There seems to be a
discrepancy in how f2b works so avoid using it
      end
   end
   if n<6 %Default termination information
      termOps=[100];
      termFN='maxGenTerm';
   end
   if n<12 %Default muatation information
      if opts(2)==1 %Float GA
         mutFNs=['boundaryMutation multiNonUnifMutation nonUnifMutation
unifMutation'];
         mutOps=[4 0 0;6 termOps(1) 3;4 termOps(1) 3;4 0 0];
      else %Binary GA
         mutFNs=['binaryMutation'];
         mutOps=[0.05];
      end
   end
   if n<10 %Default crossover information
      if opts(2)==1 %Float GA
         xOverFNs=['arithXover heuristicXover simpleXover'];
         xOverOps=[2 0;2 3;2 0];
      else %Binary GA
         xOverFNs=['simpleXover'];
         xOverOps=[0.6];
      end
   end
   if n<9 %Default select opts only i.e. roullete wheel.
      selectOps=[];
   end
   if n<8 %Default select info
      selectFN=['normGeomSelect'];
      selectOps=[0.08];
   end
   if n<6 %Default termination information
      termOps=[100];
      termFN='maxGenTerm';
   end
   if n<4 %No starting population passed given
      startPop=[];
   end
   if isempty(startPop) %Generate a population at random
      %startPop=zeros(80,size(bounds,1)+1);
      startPop=initializega(80,bounds,evalFN,evalOps,opts(1:2));
   end

   if opts(2)==0 %binary
      %bits=calcbits(bounds,opts(1));
      %bits = calcbits(bounds(:,1:2),bounds(:,3)');
   end
   bounds = bounds(:,1:2);

   xOverFNs=parse(xOverFNs);
   mutFNs=parse(mutFNs);

   xZomeLength  = size(startPop,2);   %Length of the xzome=numVars+fittness
   numVar       = xZomeLength-1;             %Number of variables
   popSize      = size(startPop,1);   %Number of individuals in the pop
   endPop       = zeros(popSize,xZomeLength); %A secondary population matrix
   c1           = zeros(1,xZomeLength);      %An individual
   c2           = zeros(1,xZomeLength);      %An individual
   numXOvers    = size(xOverFNs,1);   %Number of Crossover operators
   numMuts      = size(mutFNs,1);            %Number of Mutation operators
   epsilon      = opts(1);                  %Threshold for two fittness to differ
   oval         = max(startPop(:,xZomeLength)); %Best value in start pop
   bFoundIn     = 1;                 %Number of times best has changed
   done         = 0;                        %Done with simulated evolution
   gen          = 1;                 %Current Generation Number
   collectTrace = (nargout>3);               %Should we collect info every gen
   floatGA      = opts(2)==1;               %Probabilistic application of ops
   display      = opts(3);                  %Display summary
   details      = opts(4);                   %Display details

   if collectTrace
      figure
   end

   while(~done)
      %Elitist Model
      [bval,bindx] = max(startPop(:,xZomeLength)); %Best of current pop
      best =  startPop(bindx,:);
```

```
if collectTrace
   traceInfo(gen,1)=gen;                         %current generation
   traceInfo(gen,2)=startPop(bindx,xZomeLength);       %Best fittness
   traceInfo(gen,3)=mean(startPop(:,xZomeLength));     %Avg fittness
   traceInfo(gen,4)=std(startPop(:,xZomeLength));
   hold off
   plot(traceInfo(:,1),traceInfo(:,2),'r')
   hold on
   plot(traceInfo(:,1),traceInfo(:,3),'b')
end

if ( (abs(bval - oval)>epsilon) | (gen==1)) %If we have a new best sol
   if display
      %fprintf(1,'\n%d %f\n',gen,bval);          %Update the display
      fprintf(1,'Starting Generation %d. New best is %g\n',gen,bval);
   end
   if floatGA
      bPop(bFoundIn,:)=[gen startPop(bindx,:)]; %Update bPop Matrix
   else
      bPop(bFoundIn,:)=[gen b2f(startPop(bindx,1:numVar),bounds,bits)...
            startPop(bindx,xZomeLength)];
   end
   bFoundIn=bFoundIn+1;                      %Update number of changes
   oval=bval;                                %Update the best val
      end

endPop = feval(selectFN,startPop,[gen selectOps]); %Select

if floatGA %Running with the model where the parameters are numbers of ops
   for i=1:numXOvers,
      for j=1:xOverOps(i,1),
         a = round(rand*(popSize-1)+1);        %Pick a parent
         b = round(rand*(popSize-1)+1);        %Pick another parent
         xN=deblank(xOverFNs(i,:));    %Get the name of crossover function
         [c1 c2] = feval(xN,endPop(a,:),endPop(b,:),bounds,[gen xOverOps(i,:)]);

         if c1(1:numVar)==endPop(a,(1:numVar)) %Make sure we created a new
            c1(xZomeLength)=endPop(a,xZomeLength); %solution before evaluating
         elseif c1(1:numVar)==endPop(b,(1:numVar))
            c1(xZomeLength)=endPop(b,xZomeLength);
         else
            %[c1(xZomeLength) c1] = feval(evalFN,c1,[gen evalOps]);
            eval(e1str);
         end

         if c2(1:numVar)==endPop(a,(1:numVar))
            c2(xZomeLength)=endPop(a,xZomeLength);
         elseif c2(1:numVar)==endPop(b,(1:numVar))
            c2(xZomeLength)=endPop(b,xZomeLength);
         else
            %[c2(xZomeLength) c2] = feval(evalFN,c2,[gen evalOps]);
            eval(e2str);
         end

         endPop(a,:)=c1;
         endPop(b,:)=c2;
      end
   end

   for i=1:numMuts,
      for j=1:mutOps(i,1),
         a = round(rand*(popSize-1)+1);
         c1 = feval(deblank(mutFNs(i,:)),endPop(a,:),bounds,[gen mutOps(i,:)]);

         if c1(1:numVar)==endPop(a,(1:numVar))
            c1(xZomeLength)=endPop(a,xZomeLength);
         else
            %[c1(xZomeLength) c1] = feval(evalFN,c1,[gen evalOps]);
            eval(e1str);
         end
         endPop(a,:)=c1;
      end
   end

else %We are running a probabilistic model of genetic operators
   % for i=1:numXOvers,
        i = 1;
           xN=deblank(xOverFNs(i,:));       %Get the name of crossover function
           cp=find(rand(popSize,1)<xOverOps(i,1)==1);
           if~isempty(cp)
              if rem(size(cp,1),2) cp=cp(1:(size(cp,1)-1)); end

              list = rand(size(cp)); %Generate a random list for choosing which
individuals get paired for xOver

              for j=1:size(cp,1) %NEED TO TEST THAT INDIVIDUALS ARE CHOSEN RANDOMLY
                 [tmp, a] = max(list);
                 list(a) = 0;
                 [tmp, b] = max(list);
                 list(b) = 0;
                 a = cp(a);
                 b = cp(b);
                 [endPop(a,:) endPop(b,:)] = feval(xN,endPop(a,:),endPop(b,:),...
                    bounds,[gen xOverOps(i,:)]);
              end
           end
        %end

        %for i=1:numMuts

           mN=deblank(mutFNs(i,:));
           for j=1:popSize
              endPop(j,:) = feval(mN,endPop(j,:),bounds,[gen mutOps(i,:)]);
              eval(e1str);
              if details
                 fprintf(1,'Gen: %d\t Ind: %d\t Fitness: %g\n',gen,j,v);
              end
           end
        %end
     end

     gen=gen+1;
     done=feval(termFN,[gen termOps],bPop,endPop); %See if the ga is done
     startPop=endPop;                        %Swap the populations
     [bval,bindx] = min(startPop(:,xZomeLength)); %Keep the best solution
     startPop(bindx,:) = best;              %replace it with the worst

     save latest.mat startPop traceInfo bPop

  end

  endPop = startPop;
  [bval,bindx] = max(startPop(:,xZomeLength));
  if display
     fprintf(1,'\n%d %f\n',gen,bval);
  end
  x=startPop(bindx,:);
  if opts(2)==0 %binary
     x=b2f(x,bounds,bits);
     bPop(bFoundIn,:)=[gen b2f(startPop(bindx,1:numVar),bounds,bits)...
           startPop(bindx,xZomeLength)];
  else
     bPop(bFoundIn,:)=[gen startPop(bindx,:)];
  end

  if collectTrace
     traceInfo(gen,1)=gen;           %current generation
     traceInfo(gen,2)=startPop(bindx,xZomeLength); %Best fittness
     traceInfo(gen,3)=mean(startPop(:,xZomeLength)); %Avg fittness
     traceInfo(gen,4)=std(startPop(:,xZomeLength)); %Std. Deviation
     hold off
     plot(traceInfo(:,1),traceInfo(:,2),'r')
     hold on
     plot(traceInfo(:,1),traceInfo(:,3),'b')
  end
```

Appendix A.11 initGa.m

```
function [pop] = initialize_run_ga(num, bounds, bits, evalFN,evalOps,opts)
% function [pop]=initializega(populationSize, variableBounds,evalFN,
%                           evalOps,opts)
%    initializega creates a matrix of random numbers with
%    a number of rows equal to the populationSize and a number
%    columns equal to the number of rows in bounds plus 1 for
%    the f(x) value which is found by applying the evalFN.
%    This is used by the ga to create the population if it
%    is not supplied.
%
% pop            - the initial, evaluated, random population
% populatoinSize - the size of the population, i.e. the number to create
% variableBounds - a matrix which contains the bounds of each variable, i.e.
%                  [var1_high var1_low; var2_high var2_low; ....]
% evalFN         - the evaluation fn, usually the name of the .m file for
%                  evaluation
% evalOps        - any opts to be passed to the eval function defaults []
% opts        - options to the initialize function, ie.
%                  [type prec] where eps is the epsilon value
%                  and the second option is 1 for float and 0 for binary,
%                  prec is the precision of the variables defaults [1e-6 1]

% Binary and Real-Valued Simulation Evolution for Matlab GAOT V2
% Copyright (C) 1998 C.R. Houck, J.A. Joines, M.G. Kay
%
% C.R. Houck, J.Joines, and M.Kay. A genetic algorithm for function
% optimization: A Matlab implementation. ACM Transactions on Mathmatical
% Software, Submitted 1996.
%
% This program is free software; you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation; either version 1, or (at your option)
% any later version.
%
% This program is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
% GNU General Public License for more details. A copy of the GNU
% General Public License can be obtained from the
% Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

if nargin<5
   opts = [1e-6 1 0 0];
end
if isempty(opts)
   opts = [1e-6 1 0 0];
end

if any(evalFN<48) %Not a .m file
   if opts(2)==1 %Float GA
      estr=['x=pop(i,1); pop(i,xZomeLength)=', evalFN ';'];
   else %Binary GA
      estr=['x=b2f(pop(i,:),bounds,bits); pop(i,xZomeLength)=', evalFN ';'];
   end
else %A .m file
   if opts(2)==1 %Float GA
      estr=['[ pop(i,:) pop(i,xZomeLength)]=' evalFN '(pop(i,:),[0 evalOps]);'];
   else %Binary GA
      %    estr=['x=b2f(pop(i,:),bounds,bits);[x v]=' evalFN ...
      %          '(x,[0 evalOps]); pop(i,:)=[f2b(x,bounds,bits) v];'];
      estr=['x=b2f(pop(i,:),bounds,bits);[x v]=' evalFN ...
            '(x,[0 evalOps]); pop(i,end) = v;']; %There seems to be a discrepancy in
how f2b works so avoid using it
   end
end
details = opts(4);



numVars     = size(bounds,1);                %Number of variables
rng         = (bounds(:,2)-bounds(:,1))'; %The variable ranges'

if opts(2)==1 %Float GA
   xZomeLength = numVars+1;           %Length of string is numVar + fit
   pop         = zeros(num,xZomeLength);     %Allocate the new population
   pop(:,1:numVars)=(ones(num,1)*rng).*(rand(num,numVars))+...
      (ones(num,1)*bounds(:,1)');
else %Binary GA
   %bits=calcbits(bounds(:,1:2),bounds(:,3)');
   xZomeLength = sum(bits)+1;                %Length of string is numVar + fit
   pop = round(rand(num,sum(bits)+1));
end
bounds = bounds(:,1:2);
for i=1:num
   eval(estr);
   if details
      fprintf('Initialising: Individual %d \tFitness %g\n',i,v);
   end
end
```

Appendix A.12 ga_sfn.m

```
function [sys,x0,str,ts] = ga_sfn(t,x,u,flag,popSize,bounds,bits,period)
%Run a genetic algorithm as an S-Function
%Assume a binary GA with rouletete selection,maskXover,
%binary mutation with probability .01
%

%Written By Joe Foran August 2002

switch flag,

   %%%%%%%%%%%%%%%%%%
   % Initialization %
   %%%%%%%%%%%%%%%%%%
case 0,
   [sys,x0,str,ts] = mdlInitializeSizes(popSize,bounds,bits,period);

   %%%%%%%%%%
   % Update %
   %%%%%%%%%%
case 2,
   [sys] = mdlUpdate(t,x,u,popSize);

   %%%%%%%%%%
   % Output %
   %%%%%%%%%%
case 3,
   sys  = mdlOutputs(t,x,u,popSize,bounds,bits);

   %%%%%%%%%%%%%
   % Terminate %
   %%%%%%%%%%%%%
case 9,
   mdlTerminate(t,x,u,popSize,bounds,bits); %save results to disk

   %%%%%%%%%%%%%%%%%%%%
   % Unexpected flags %
   %%%%%%%%%%%%%%%%%%%%
otherwise
   error(['unhandled flag = ',num2str(flag)]);
end

%
%=======================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=======================================================================
%
function [sys,x0,str,ts] = mdlInitializeSizes(popSize,bounds,bits,period)

numVar = size(bounds,1);
initPop = initga(popSize,bits);

numGenes = size(initPop,2);
gen = 1; %Generation counter

sizes = simsizes;
sizes.NumContStates  = 0;
sizes.NumDiscStates  = (popSize+1)*numGenes+3;
sizes.NumOutputs     = numVar+3;
sizes.NumInputs      = 1;
sizes.DirFeedthrough = 0;
sizes.NumSampleTimes = 1;

sys = simsizes(sizes);

initPop = reshape(initPop',popSize*numGenes,1); %Make initPop a vector


x0  = zeros(sizes.NumDiscStates,1);
x0(1:popSize*numGenes) = initPop; %Place initPop into state vector
x0(end-2) = numGenes; %Store this value so it doesn't have to be constantly
recalculated
x0(end-1) = gen; %Place generation counter into state vector;
x0(end) = 1; %Tracks which chromosome is being evaluated


str = [];
ts  = [period 0];

% end mdlInitializeSizes
%
%=========================================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time step
% requirements.
%=========================================================================
%
function [sys] = mdlUpdate(t,x,u,popSize)
input = u;
if u<=0
   input =1e-9;
end
numGenes = x(end-2);
cc = x(end);

%Get current population into matrix form
pop = reshape(x(1:(popSize+1)*numGenes),numGenes,popSize+1)';

current_optimum = pop(end,end);

%The fitness in input refers to the previous chromosome
if cc>1 %This is because output occurs before update
   pop(cc-1,end)=input;%Read in input into the fitness value of the previous
chromosome
   if input>current_optimum %Save new best solution
      pop(end,:) = pop(cc-1,:);
   end
end

x(end) = x(end)+1; %Increment chromosome counter

if x(end) > popSize+1 %End of a generation
   if max(pop(:,end-1))<current_optimum %If the optimum was lost in this generation
      [wval windx] = min(pop(:,end-1)); %Replace the worst of this generation
      pop(windx,:) = pop(end,:); %With the optimum
   end
   newPop = create_next_generation(pop(1:end-1,:)); %find next generation
   pop = [newPop; pop(end,:)];
   x(end) = 1;   %Start again at beginning of next generation
   x(end-1) = x(end-1)+1; %Increment Generation Counter
end

x(1:(popSize+1)*numGenes) = reshape(pop',numGenes*(popSize+1),1); %Transform matrix
back into vector

sys = x;

%end mdlUpdate
%
%=========================================================================
% mdlOutputs
% Return Return the output vector for the S-function
%=========================================================================
%
function sys = mdlOutputs(t,x,u,popSize,bounds,bits)

numGenes = x(end-2);
```

```
generation = x(end-1);
cc = x(end);
optimum = x((popSize+1)*numGenes);
pop = reshape(x(1:(popSize+1)*numGenes),numGenes,popSize+1)';
chrom = b2f(pop(cc,1:end-1),bounds,bits);

%Output current chromosome, best chromosome so far, best fitness so far and current
generation number
sys = [chrom cc optimum generation];

%end mdlOutputs

function mdlTerminate(t,x,u,popSize,bounds,bits)

numGenes = x(end-2);
currentChrom = x(end);
currentGen = x(end)-1;

pop = reshape(x(1:(popSize+1)*numGenes),numGenes,popSize+1)';
best_chrom = pop(end,:);

save finalState.mat best_chrom currentChrom currentGen pop


%end MdlTerminate



function [pop] = initga(num,bits)
% function [pop]=initializega(populationSize, variableBounds, bits)
%
%    initializega creates a matrix of random numbers with
%    a number of rows equal to the populationSize
%
% pop            - the initial, evaluated, random population
% populatoinSize - the size of the population, i.e. the number to create
% bits                   - a vector with the number of bits required per variable

pop = round(rand(num,sum(bits)+1));
pop(:,end) = zeros(num,1);

%end initga


function newPop = create_next_generation(oldPop)
[popSize, nVars] = size(oldPop);

%Select using stochastic Remainder Selection
newPop = stocRemSelec(oldPop);

%Crossover using mask crossover

list = rand(popSize,1); %Generate a list to decide how crossovers are paired

for i = 1:floor(popSize/2)
   [tmp,ind] = max(list(:,1));
   a = newPop(ind,:);
   list(ind) = 0;
   [tmp,ind] = max(list(:,1));
   b = newPop(ind,:);
   list(ind) = 0;
   [c d]  = maskXover(a,b);
   newPop(i*2-1,:) = c;
   newPop(i*2,:) = d;
end

%Mutate
for i = 1:popSize
   newPop(i,:) = binMutation(newPop(i,:),0.01);
end

%end of create_next_generation

function parent = binMutation(parent,mut_prob)
% Binary mutation changes each of the bits of the parent
% based on the probability of mutation
```

```
%Based on binaryMutation in GAOT toolbox. Modified by Joe Foran
numVar = size(parent,2)-1;           % Get the number of variables
% Pick a variable to mutate randomly from 1-number of vars
rN=rand(1,numVar)<mut_prob;
parent=[abs(parent(1:numVar) - rN) parent(numVar+1)];
```

Appendix A.13 evalfis_sfn.m

```
function [sys,x0,str,ts] = evalfis_sfn3(t,x,u,flag)
%Evaluates a two-input FIS based on parameters and error
%values input into the block

global the_fuzzy
switch flag,

   %%%%%%%%%%%%%%%%%%
   % Initialization %
   %%%%%%%%%%%%%%%%%%
case 0,
   [sys,x0,str,ts]=mdlInitializeSizes;

   %%%%%%%%%%%%%%%
   % Derivatives %
   %%%%%%%%%%%%%%%
case 1,
   sys=[];

   %%%%%%%%%%
   % Update %
   %%%%%%%%%%
case 2,
   sys=mdlUpdate(t,x,u);

   %%%%%%%%%%%
   % Outputs %
   %%%%%%%%%%%
case 3,
   sys=mdlOutputs(t,x,u);

   %%%%%%%%%%%%%%%%%%%%%%%
   % GetTimeOfNextVarHit %
   %%%%%%%%%%%%%%%%%%%%%%%
case 4,
   sys=[];

   %%%%%%%%%%%%%
   % Terminate %
   %%%%%%%%%%%%%
case 9,
   sys=[];

   %%%%%%%%%%%%%%%%%%%%
   % Unexpected flags %
   %%%%%%%%%%%%%%%%%%%%
otherwise
   error(['Unhandled flag = ',num2str(flag)]);

end

% end sfuntmpl

%
%=============================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts]=mdlInitializeSizes

%
% call simsizes for a sizes structure, fill it in and convert it to a
% sizes array.
%
% Note that in this example, the values are hard coded.  This is not a
% recommended practice as the characteristics of the block are typically
% defined by the S-function parameters.
%
sizes = simsizes;

sizes.NumContStates  = 0;
sizes.NumDiscStates  = 19;
sizes.NumOutputs     = 1;
sizes.NumInputs      = 21;
sizes.DirFeedthrough = 1;
sizes.NumSampleTimes = 1;   % at least one sample time is needed

sys = simsizes(sizes);

%
% initialize the initial conditions
%
x0  = zeros(sizes.NumDiscStates,1);

%
% str is always an empty matrix
%
str = [];

%
% initialize the array of sample times
%
ts  = [-1 0];

% end mdlInitializeSizes

%
%=============================================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time step
% requirements.
%=============================================================================
function sys = mdlUpdate(t,x,u)
sys = u(1:19);



%=============================================================================
% mdlOutputs
% Return the block outputs.
%=============================================================================
%
function sys=mdlOutputs(t,x,u)
global the_fuzzy
input = u(20:21);

%Check if FLC needs to be changed
if any(u(1:15) ~= x(1:15))| u(19)~=x(19)
   n = u(1:3);
   pm = u(4:6).^u(10:12);
        pr = u(7:9).^u(13:15);
   theta = u(19);

   the_fuzzy = make_fis(2,1,n,pm,pr,theta);
end

scale = u(16:18);
input = input.*scale(1:2);

%Saturate the input to between -1 and 1
for i = 1:length(input)
   if abs(input(i))>1
      input(i) = sign(input(i));
   end
end

%Calculate output
sys = scale(end)*evalfis(input,the_fuzzy);

% end mdlOutputs
```

Appendix A.14 fuzz_class.m

```
function [sys,x0,str,ts] = fuzz_class(t,x,u,flag,bidOpts,taxOpts)
%S_Function that runs a genetic based fuzzy classifier

global RULES THE_FUZZY WINNERS

switch flag,

   %%%%%%%%%%%%%%%%%%
   % Initialization %
   %%%%%%%%%%%%%%%%%%
case 0,
   [sys,x0,str,ts]=mdlInitializeSizes;

   %%%%%%%%%%%%%%%
   % Derivatives %
   %%%%%%%%%%%%%%%
case 1,
   sys=[];

   %%%%%%%%%%
   % Update %
   %%%%%%%%%%
case 2,
   sys=mdlUpdate(t,x,u);

   %%%%%%%%%%%
   % Outputs %
   %%%%%%%%%%%
case 3,
   sys=mdlOutputs(t,x,u,bidOpts,taxOpts);

   %%%%%%%%%%%%%%%%%%%%%%%
   % GetTimeOfNextVarHit %
   %%%%%%%%%%%%%%%%%%%%%%%
case 4,
   sys=[];

   %%%%%%%%%%%%%
   % Terminate %
   %%%%%%%%%%%%%
case 9,
   sys=mdlTerminate(t,x,u);

   %%%%%%%%%%%%%%%%%%%%
   % Unexpected flags %
   %%%%%%%%%%%%%%%%%%%%
otherwise
   error(['Unhandled flag = ',num2str(flag)]);

end

% end sfuntmpl

%
%=============================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts]=mdlInitializeSizes

%
% call simsizes for a sizes structure, fill it in and convert it to a
% sizes array.
%
% Note that in this example, the values are hard coded.  This is not a
% recommended practice as the characteristics of the block are typically
% defined by the S-function parameters.
%
global RULES THE_FUZZY WINNERS

numMfs = zeros(1,2);
```

```
for i = 1:2
   numMfs(i) = size(THE_FUZZY.input(i).mf,2);
end

numOutMfs = size(THE_FUZZY.output.mf,2);
RULES = combineVectors([numMfs numOutMfs]);
numRules = size(RULES,1);
RULES(:,4) = 3*ones(numRules,1);
RULES(:,5) = 100*ones(numRules,1); %Make Initial Fitness of all rules 100;
WINNERS = [1 1 1 1]'; %Set the first rule to be the initial winner


sizes = simsizes;

sizes.NumContStates  = 0;
sizes.NumDiscStates  = numRules*5+5;
sizes.NumOutputs     = 4;
sizes.NumInputs      = 5;
sizes.DirFeedthrough = 1;
sizes.NumSampleTimes = 1;   % at least one sample time is needed

sys = simsizes(sizes);

%
% initialize the initial conditions
%
x0  = [reshape(RULES,numRules*5,1); numRules; WINNERS];

%
% str is always an empty matrix
%
str = [];

%
% initialize the array of sample times
%
ts  = [-1 0];

% end mdlInitializeSizes

%
%=============================================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time step
% requirements.
%=============================================================================
%
function sys=mdlUpdate(t,x,u)
global RULES WINNERS

numRules = x(end-4);
sys = [reshape(RULES,numRules*5,1); numRules; WINNERS];

% end mdlUpdate

%
%=============================================================================
% mdlOutputs
% Return the block outputs.
%=============================================================================
%
function sys=mdlOutputs(t,x,u,bidOpts,taxOpts)
global RULES WINNERS

reward = u(1);
mfs = u(2:end)';
message = [mfs(1) mfs(1) mfs(2) mfs(2) mfs(3) mfs(4) mfs(3) mfs(4)]';

message = reshape(message,4,2);
numRules = x(end-4);
RULES = reshape(x(1:end-5),numRules,5);
oldWinner = x(end-3:end);
newWinner = oldWinner;
action = zeros(4,1);
numOldWinners = sum(find(oldWinner>0));
```

```
for i = 1:4
   if~all(message(i))
      newWinner(i) = 0; %Default to rule 1 if not applicable
      action(i) = 0;
   else
      matchList = matchMessage(RULES,message(i));
      if oldWinner(i) == 0
         [winners,RULES] = apportionCredit(RULES,matchList,[],bidOpts,taxOpts);
      else
         [winners,RULES] = apportionCredit(RULES,matchList,oldWinner(i),bidOpts,
taxOpts);
         RULES(oldWinner(i),end) = RULES(oldWinner(i),end)+reward/numOldWinners;
      end
      if isempty(winners)
         newWinner(i) = 0;
         action(i) = 0;
      else
         newWinner(i) = winners(ceil(rand*length(winners)));
         action(i) = RULES(newWinner(i),3);
      end
   end
end

WINNERS = newWinner;


sys = action;

% end mdlOutputs


%
%=============================================================================
% mdlTerminate
% Perform any end of simulation tasks.
%=============================================================================
%
function sys=mdlTerminate(t,x,u)

numRules = x(end-4);
RULES = reshape(x(1:end-5),numRules,5);
save FinalRules.mat RULES
clear RULES;
sys = [];

% end mdlTerminate
```

Appendix A.15 match_message.m

```
function matchList = matchMessage(classifiers,message)
%MATCHMESSAGE Returns a list of which classifiers match the message
%
%Returns a 0 if no match 1 if a succesful match

[numMess lenMess] = size(message);

testSamp = round(classifiers(:,1:lenMess));

for i = 1:size(classifiers,1)
   for j = 1:numMess
      matchList(i,j) = all((testSamp(i,:) == message(j,:))|(testSamp(i,:) == -1));
   end
end
```

Appendix A.16 apportion_credit.m

```
function [winner,classifiers] =
apportionCredit(classifiers,matchList,oldWinner,bidOpts,taxOpts)
%APPORTIONCREDIT apportions credit in a classifier system
%
%winner               The winner of this round
```

```
%
%classifiers   The list of classifiers
%matchList             List of those matching current messages
%oldWinner             The winner from previous rounds
%b1,b2                 Parameters for scaling bid with regard to specificity

if nargin<3
   error('Insufficient input arguments');
end

if nargin<5 | length(taxOpts) < 2
   taxOpts = [0 0];
end
existTax = taxOpts(1);
bidTax = taxOpts(2);


if nargin<4 |length(bidOpts <3)
   bidOpts = [0.1 0.25 0.125];
end
cBid = bidOpts(1);
b1 = bidOpts(2);
b2 = bidOpts(3);

maxBid = 0;
if any(matchList)
   activeInd = find(matchList == 1);
   bid = cBid*classifiers(activeInd,end).*(classifiers(activeInd,end-1)*b2+b1);
   maxBid  = max(bid);
   maxInd = find(bid == maxBid);
   winner = activeInd(maxInd);
else
   winner = oldWinner; %If there is no match, the old winner wins again
end

%Collect Taxes
classifiers(:,end) = classifiers(:,end) - classifiers(:,end)*existTax -
classifiers(:,end).*matchList*bidTax;

%Distribute Payments

classifiers(winner,end) = classifiers(winner,end)-maxBid;
negInd = find(classifiers(winner,end)<0);
classifiers(negInd,end) =0;
if ~isempty(oldWinner)
   classifiers(oldWinner,end) =
classifiers(oldWinner,end)+maxBid*length(winner)/length(oldWinner);
end
```

Appendix A.17 fuzzify_online.m

```
function [sys,x0,str,ts] = fuzzifyOnline(t,x,u,flag,scale)
%S_Function that fuzzifies error for classifier
%Only works for triangular MFs and assumes that the sets are spaced such
%that any value is a member of at most two sets

global THE_FUZZY

switch flag,

   %%%%%%%%%%%%%%%%%%
   % Initialization %
   %%%%%%%%%%%%%%%%%%
case 0,
   [sys,x0,str,ts]=mdlInitializeSizes;

   %%%%%%%%%%%%%%%
   % Derivatives %
   %%%%%%%%%%%%%%%
case 1,
   sys=[];

   %%%%%%%%%%
   % Update %
   %%%%%%%%%%
case 2,
   sys = [];

   %%%%%%%%%%%
   % Outputs %
   %%%%%%%%%%%
case 3,
   sys=mdlOutputs(t,x,u,scale);

   %%%%%%%%%%%%%%%%%%%%%%%
   % GetTimeOfNextVarHit %
   %%%%%%%%%%%%%%%%%%%%%%%
case 4,
   sys=[];

   %%%%%%%%%%%%%
   % Terminate %
   %%%%%%%%%%%%%
case 9,
   sys=[];

   %%%%%%%%%%%%%%%%%%%%
   % Unexpected flags %
   %%%%%%%%%%%%%%%%%%%%
otherwise
   error(['Unhandled flag = ',num2str(flag)]);

end

% end sfuntmpl

%
%=============================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts]=mdlInitializeSizes

%
% call simsizes for a sizes structure, fill it in and convert it to a
% sizes array.
%
% Note that in this example, the values are hard coded.  This is not a
% recommended practice as the characteristics of the block are typically
% defined by the S-function parameters.
%

global THE_FUZZY

numErrorMfs = size(THE_FUZZY.input(1).mf,2);
numDeMfs = size(THE_FUZZY.input(2).mf,2);

errorMfParams = zeros(numErrorMfs,3);
deMfParams = zeros(numDeMfs,3);

for i = 1:numErrorMfs
   errorMfParams(i,:) = THE_FUZZY.input(1).mf(i).params;
end

for i = 1:numDeMfs
   deMfParams(i,:) = THE_FUZZY.input(2).mf(i).params;
end

sizes = simsizes;

sizes.NumContStates  = 0;
sizes.NumDiscStates  = 2+3*numErrorMfs+3*numDeMfs;
sizes.NumOutputs     = 8;
sizes.NumInputs      = 2;
sizes.DirFeedthrough = 1;
sizes.NumSampleTimes = 1;   % at least one sample time is needed

sys = simsizes(sizes);

%
```

```
% initialize the initial conditions
%
x0  = [numErrorMfs; numDeMfs; reshape(errorMfParams,numErrorMfs*3,1);
reshape(deMfParams,3*numDeMfs,1)];
%
% str is always an empty matrix
%
str = [];

%
% initialize the array of sample times
%
ts  = [-1 0];

% end mdlInitializeSizes

% end mdlUpdate

%
%=============================================================================
% mdlOutputs
% Return the block outputs.
%=============================================================================
%
function sys = mdlOutputs(t,x,u,scale)

error = u(1)*scale(1);
dError = u(2)*scale(2);

numErrorMfs = x(1);
numDeMfs= x(2);
errorMfParams = reshape(x(3:2+numErrorMfs*3),numErrorMfs,3);
deMfParams = reshape(x(3+numErrorMfs*3:end),numDeMfs,3);

[errorSets, errorDof] = findSets(error,numErrorMfs,errorMfParams);
[deSets, deDof] = findSets(dError,numDeMfs,deMfParams);

sys = [errorSets; deSets; errorDof; deDof];


% end mdlOutputs
```

Appendix A.18 defuzzify_online.m

```
function [sys,x0,str,ts] = defuzzifyOnline(t,x,u,flag,scale)
%S_Function that fuzzifies error for classifier
%Only works for triangular MFs and assumes that the sets are spaced such
%that any value is a member of at most two sets

global THE_FUZZY

switch flag,

   %%%%%%%%%%%%%%%%%%
   % Initialization %
   %%%%%%%%%%%%%%%%%%
case 0,
   [sys,x0,str,ts]=mdlInitializeSizes;

   %%%%%%%%%%%%%%%
   % Derivatives %
   %%%%%%%%%%%%%%%
case 1,
   sys=[];

   %%%%%%%%%%
   % Update %
   %%%%%%%%%%
case 2,
   sys = [];

   %%%%%%%%%%%
   % Outputs %
   %%%%%%%%%%%
case 3,
   sys=mdlOutputs(t,x,u,scale);

   %%%%%%%%%%%%%%%%%%%%%%%
   % GetTimeOfNextVarHit %
   %%%%%%%%%%%%%%%%%%%%%%%
case 4,
   sys=[];

   %%%%%%%%%%%%%
   % Terminate %
   %%%%%%%%%%%%%
case 9,
   sys=[];

   %%%%%%%%%%%%%%%%%%%%
   % Unexpected flags %
   %%%%%%%%%%%%%%%%%%%%
otherwise
   error(['Unhandled flag = ',num2str(flag)]);

end

% end sfuntmpl

%
%=============================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts]=mdlInitializeSizes

%
% call simsizes for a sizes structure, fill it in and convert it to a
% sizes array.
%
% Note that in this example, the values are hard coded.  This is not a
% recommended practice as the characteristics of the block are typically
% defined by the S-function parameters.
%

global THE_FUZZY

numOutputMfs = size(THE_FUZZY.output.mf,2);

outputMfParams = zeros(numOutputMfs,3);

for i = 1:numOutputMfs
   outputMfParams(i,:) = THE_FUZZY.output.mf(i).params;
end
outputMfParams(1,1) = -1;
outputMfParams(end,end) = 1;


sizes = simsizes;

sizes.NumContStates  = 0;
sizes.NumDiscStates  = 1+3*numOutputMfs;
sizes.NumOutputs     = 1;
sizes.NumInputs      = 8;
sizes.DirFeedthrough = 1;
sizes.NumSampleTimes = 1;   % at least one sample time is needed

sys = simsizes(sizes);

%
% initialize the initial conditions
%
x0  = [numOutputMfs; reshape(outputMfParams,numOutputMfs*3,1)];
%
% str is always an empty matrix
%
str = [];

%
```

```
% initialize the array of sample times
%
ts  = [-1 0];

% end mdlInitializeSizes

%
%=============================================================================
% mdlOutputs
% Return the block outputs.
%=============================================================================
%
function sys = mdlOutputs(t,x,u,scale)

numOutputMfs = x(1);
outputMfParams = reshape(x(2:end),numOutputMfs,3);
mfs = u(1:4);
dof = u(5:8);

mfs = mfs(find(mfs>0));
dof = dof(find(mfs>0));
triangles = outputMfParams(mfs,:);
sys = scale*centroid(triangles,dof);
```

Appendix A.19 centroid.m

```
function centroid = centroid(triangles,cutoffs)
%CENTROID Returns the centroid of the overall group of triangles

numTri = size(triangles,1);

if size(triangles,2) ~=3
   error('Triangles matrix should have three columns')
end

if any([triangles(:,2)-triangles(:,1) triangles(:,3)-triangles(:,2)]<0)
   error('Parameters for all triangles should be in ascending order')
end

if length(cutoffs)~=numTri
   error('cutoffs should be a vector of same length as there are triangles')
end

triangles = sortrows(triangles);

sumIyx = 0;
sumIy = 0;
prevOverlap = 0;

for i = 1:numTri
   if i<numTri&(triangles(i,3)>triangles(i+1,1)) %Overlap between this and next
triangle
      nextOverlap = 1;
   else
      nextOverlap = 0;
   end
   if ~prevOverlap
      a = triangles(i,1);
      b = a+cutoffs(i)*(triangles(i,2)-a);
      [sumIyx sumIy] = addLineInts(a,b,0,cutoffs(i),sumIyx,sumIy);
   end
   if ~nextOverlap
      d = triangles(i,3);
      c = d-cutoffs(i)*(d-triangles(i,2));
      [sumIyx sumIy] = addLineInts(b,c,cutoffs(i),cutoffs(i),sumIyx,sumIy);
      [sumIyx sumIy] = addLineInts(c,d,cutoffs(i),0,sumIyx,sumIy);
   else
      if cutoffs(i)<cutoffs(i+1)
         c = triangles(i+1,1)+cutoffs(i)*(triangles(i+1,2)-triangles(i+1,1));
         d = triangles(i+1,1)+cutoffs(i+1)*(triangles(i+1,2)-triangles(i+1,1));
      else
         c = triangles(i,3)-cutoffs(i)*(triangles(i,3)-triangles(i,2));
         d = triangles(i,3)-cutoffs(i+1)*(triangles(i,3)-triangles(i,2));
      end
      [sumIyx,sumIy] = addLineInts(b,c,cutoffs(i),cutoffs(i),sumIyx,sumIy);
      [sumIyx sumIy] = addLineInts(c,d,cutoffs(i),cutoffs(i+1),sumIyx,sumIy);
      b = d; %Bring forward for next calculations
   end
   prevOverlap = nextOverlap;
end

centroid = sumIyx/sumIy;

%END centroid function

function [Iyx, Iy] = lineInts(x1,x2,y1,y2)
Iyx = ((y2-y1)/(x2-x1))*((x2^3)/3+(x1^3)/6-(x1*(x2^2))/2)+0.5*y1*(x2^2-x1^2);
Iy = (x2-x1)*(y1+0.5*(y2-y1));

function [sIyx, sIy] = addLineInts(x1,x2,y1,y2,sIyx,sIy)
if x1~=x2 %avoid division by zero - in this case integrals are both zero anyway so no
need to add
   [iyx,iy] = lineInts(x1,x2,y1,y2);
   sIyx = sIyx+iyx;
   sIy = sIy+iy;
end
```

Appendix A.20 calcbits2.m

```
function [bits,prec] = calcbits2(bounds,des_prec)
%calcbits2 Calculate the number of bits required to represent the
%desired precision between bounds

%Based on calcbits.m in GAOT toolbox
%This fixes a small bug in the original program

[numVar, ans] = size(bounds);

if(size(des_prec,1) ~= numVar)
   error('des_prec should have the same number of rows as bounds')
end

range = (bounds(:,2) - bounds(:,1))';
%Originally was ceil and 1 wasn't added.
bits=floor(log2(range ./ des_prec'))+1;
prec = range./(2.^bits-1);
```