Fuzzy Logic, Control, and Optimisation

A Thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Engineering (Mechanical)
in the
University of Canterbury
by
W. J. Hoyle B.E.

University of Canterbury
1996

Abstract

This thesis examines the utility of fuzzy logic in the field of control engineering. A tutorial introduction to the field of fuzzy control is presented during the development of an efficient fuzzy controller. Using the controller as a starting point, a set of criteria are developed that ensure a close connection between rule base construction and control surface geometry. The properties of the controller are exploited in the design of a global controller optimiser based on a genetic algorithm, and a tutorial explaining how the optimiser may be used to effect automatic controller design is given. A library of software that implements a fast fuzzy controller, a genetic algorithm, and various utility routines is included.

Acknowledgments

I wish to thank my supervisor G.R. Dunlop and my parents, Steve and Cheryl Hoyle, for their support and encouragement during the writing of this thesis.

Contents

1. **Introduction** 1
1.1 Description 1
1.2 Fuzzy Logic and Control Engineering 1
1.2.1 Some Limitations of Modern Control Methods 1
1.2.2 An Alternative: Rule Based Control 2
1.3 An Outline of this Thesis 2

2. **Basic Fuzzy Logic** 4
2.1 The Fuzzy Rule Base 4
2.2 Fuzzy Sets 6
2.2.1 Membership functions 6
2.2.2 Fuzzy Relations 7
2.2.3 Height 8
2.2.4 Support 8
2.2.5 Core 8
2.2.6 Fuzzy Partitions 8
2.3 Operations on Fuzzy Sets 9
2.3.1 Union and Intersection 9
2.3.2 Degree of Matching 10
2.3.3 Projection 10
2.3.4 Cylindrical Extension 11
2.3.4 Composition 11
2.4 Fuzzy Rules 11
2.4.1 Fuzzy Connectives 11
2.4.2 Fuzzy Implication 12
2.4.3 Reasoning with a Fuzzy Rule 12
2.5 An Example of Inference with a Fuzzy Rule 13
2.6 The Fuzzy Rule Base 16
2.6.1 Reasoning with a Fuzzy Rule Base 16
2.6.2 The Completeness of a Rule Base 17
2.7 Example Continued 17
2.8 Defuzzification 19
2.8.1 Maximum-Membership Defuzzification 19
2.8.2 Centroidal Defuzzification 19
2.8.3 Consequent Set Modelling 21
2.9 Summary 23

3. **Fuzzy Rule Base Design** 24
3.1 Rule Base Legibility 24
3.2 The Well Conditioned Rule Base 25

3.2.1 Rules as Points on a Control Surface 25
3.2.2 Output Set Weighting 27
3.2.4 Writing a Well Conditioned Rule Base 31
3.3 Writing a Rule Base for a Fuzzy Controller 31
3.3.1 Choice of control variables 32
3.3.2 Input Set Design 32
3.3.3 Rule Design 32
3.3.4 Output Set Weighting 32
3.3.5 Performance Tuning 33
3.3.6 Rule Base Tuning 33
3.4 Traditional Design Methods and Fuzzy Logic 34
3.5 Global Optimisation Methods 34
3.6 Combining Methods 34
3.7 Stability 35
3.8 Summary 35

4. The Optimisation of a Fuzzy Controller 37
4.1 Fuzzy logic and Genetic Algorithms 38
4.2 Genetic Algorithms 40
4.2.1 The SGA Chromosome 41
4.2.2 The SGA Population 41
4.2.2 The SGA Evaluation Function 41
4.2.3 The SGA Generation Function 42
4.2.4 The SGA Selection Function 43
4.2.5 The SGA Crossover Function 43
4.2.6 The SGA Mutation Function 43
4.3 Some Properties of the SGA 44
4.4 An Efficient Parameterisation of a Fuzzy Controller . 44
4.5 Incorporating Expert Knowledge 46
4.6 Summary 46

5. The Design of a Fuzzy Inference Engine 48
5.1 Design concerns 48
5.2 Representing Fuzzy Sets in *C* 48
5.3 Representing Fuzzy Rules in *C* 49
5.4 Possible Optimisations 51
5.5 Summary 51

6. Example: Velocity Control of a Variable Pitch Fan 52
6.1 A Simple Model of a Variable Pitch Propeller 52
6.2 The Control Problem 54
6.2.1 Input Sensor 54
6.2.2 Sampling Period 54

6.2.3 Output 54
6.2.4 The Task 55
6.3 A Fuzzy Controller 55
6.3.1 Input Set Design 55
6.3.2 The Rule Base 57
6.4 Optimisation of the Controller 58
6.4.1 Incorporation of Expert Knowledge 59
6.4.2 The Main Loop 59
6.4.3 The Fitness Function 60
6.4.4 The Optimisation Run 64
6.5 Results 65
6.6 Examination of Results 66
6.6.1 Calculating Rule Usage 66
6.6.2 Rule Reduction 67
6.7 Discussion 69

7. Conclusion 71
7.1 Suggested Further Work 72

References 73

A. Software 76

1. Introduction

This thesis examines both fuzzy logic and genetic algorithms, discusses the possibilities inherent in the combination of the two technologies, and describes the development of software to implement them in conjunction with each other.

Fuzzy logic has a wide variety of applications. This thesis examines the use of fuzzy logic methods in control.

1.1 Description

Fuzzy Logic was developed by Zadeh (1965) to provide a set of tools for manipulating imprecise data. Since its introduction, fuzzy logic has been applied in many areas, some of which include: systems analysis, signal processing, pattern recognition, decision analysis, diagnostics, and control.

Fuzzy logic was first introduced to the area of control by Mamdani (1975), with the intent of duplicating the behaviour of human system operators. Imprecise descriptions of a control task that are obtained from operators can be modelled with fuzzy logic, allowing automated control without the need for a formal analysis of the system.

Discussion of fuzzy technology is limited to the calculus of fuzzy if/then rules (CFR) for purposes of concision and clarity; from Zadeh (1992):

> *The calculus of fuzzy if/then rules is simple and close to intuition. Furthermore, it is largely self contained and does not require an extensive familiarity with fuzzy logic.*

Controllers can be designed and built from the tools of CFR.

1.2 Fuzzy Logic and Control Engineering

On the most abstract level, all controllers are simply mapping functions—they map their inputs and their state to their outputs. Standard controllers do this computationally; fuzzy logic controllers use a rule base.

1.2.1 Some Limitations of Modern Control Methods

The majority of the literature on modern control methods concerns linear control. Nonlinear problems are usually dealt with by approximating them with a linear

model. Those solutions that do address nonlinear problems lack generality; they are only applicable to a specific type of problem.

1.2.2 An Alternative: Rule Based Control

Fuzzy logic may be viewed as a language for describing arbitrary control surfaces. As such, it is not restricted, as classical controllers are, to the description of linear control surfaces, and it is not restricted by the complexities of nonlinear math, as some possible solutions to the control of nonlinear systems are.

1.3 An Outline of this Thesis

Since the introduction of fuzzy logic to the field of control engineering, many variations on the basic design of fuzzy controllers have been presented. Proceeding from the point of view that a practical design should lead to a fast, compact representation on a microcontroller, chapter 2 begins with an introduction to the calculus of fuzzy rules. The standard simplifications to the theory are presented and controller design that takes advantage of several less well known optimisations is described.

If fuzzy logic is viewed as a language for describing control surfaces, then the utility of any definition of a fuzzy controller is determined not only by the speed and compactness of the resulting code, but additionally by how easily a control surface can be built using it. Chapter 3 addresses a problem in the traditional design methodology of fuzzy controllers — discussion in the current literature invariably fails to address the connection between fuzzy rule bases and the control surfaces that they describe; given a fuzzy rule base there is no way to discover the geometry of the control surface (other than by extensive calculation), and given a control surface there is no way that a defining rule base can be built easily.

A set of criteria, referred to as "the well conditioned rule base," that enable the construction of rule bases with a strong connection to their control surfaces is defined, and methods of determining control surface geometry are presented. The chapter concludes with a description of the traditional method of fuzzy controller design from the point of view of the well conditioned rule base.

In chapter 4 it is noted that the traditional method of fuzzy controller design can be viewed as a hand optimisation scheme. Design consists of iterated testing and adjustment, which is conducted until a solution is found. A summary of other optimisation design techniques is presented, and it is noted that fuzzy logic has particular promise in the field of computer aided controller design because 1) it allows

an arbitrary control surface and 2) the result of optimisation is presented in an easily understandable form (that is, a set of simple, English-like rules).

A survey of the various global optimisation schemes that have been used with fuzzy logic is presented and it is noted that particular attention has been paid to the use of genetic algorithms. Using the well conditioned rule base as a model, it is shown that previous schemes have used inefficient parameterisations of the fuzzy controllers that they optimised. An introduction to the use of genetic algorithms is presented, and an efficient parameterisation of a fuzzy controller (based on the well conditioned rule base) is developed. Software that implements a simple genetic algorithm was written to test these ideas. Listings appear in appendix A.

Chapter 5 describes the development of efficient fuzzy controller software that implements the controller design developed in chapter 2 and makes use of several optimisations that the well conditioned rule base makes possible.

The concluding chapter contains a tutorial that shows how the software in appendix A may be used to automatically design a controller (specifically, a speed controller for a variable pitch propeller), and demonstrates the utility of the well conditioned rule base.

2. Basic Fuzzy Logic

The following is a brief introduction to the calculus of fuzzy rules. It includes only those parts of fuzzy theory that will be used in this thesis. A more comprehensive introduction can be found in Jager (1995) and Kruse (1994).

2.1 The Fuzzy Rule Base

Traditional rule bases are bivalent. If a controller is implemented with a bivalent rule base, the control surface generated will suffer from heavy quantisation. As an example, consider a set of rules for stopping a car:

In this rule base the speed of the car is represented by the variable s (in km/h), where s is in the set S of applicable car velocities: $S = \{s \mid s \in [0,180]\}$.

The variable d (in metres) represents the position of the car relative to the point where it must be stopped; d is in the set **D** of applicable stopping distances: $D = \{d \mid d \in [0,140]\}$.

Braking effort (a percentage of maximum possible braking effort) is represented by the variable b.

Partition S into the following subsets:
slow = [0,10), **med** = [10, 50), **fast** = [50, 80]

Partition **D** into the following subsets:
short = [0 ,30), **mid** = [30, 100), **long** = [100, 140]

Now, define some rules:

if s is **fast** and d is **long** then b is 50%
if s is **fast** and d is **mid** then b is 100%
if s is **fast** and d is **short** then b is 100%
if s is **med** and d is **long** then b is 50%
if s is **med** and d is **mid** then b is 50%
if s is **med** and d is **short** then b is 100%
if s is **slow** and d is **long** then b is 5%
if s is **slow** and d is **mid** then b is 5%
if s is **slow** and d is **short** then b is 50%

Table 2.1 *Rule Base For Stopping a Car.*

Note that the subsets of S form a *partition*, as do the subsets of **D**. If they did not, the

rule base might be ambiguous; if some of the sets intersected it would be possible for two or more contradictory rules to be true at the same time (for example, if the speed was fast, and the distance fell somewhere in the intersection of **mid** and **long**, the first and second rules would contradict); if some elements of **S** or **D** were not included in any of the subsets, the rule base would not produce any corresponding output for them.

The control surface that this rule base defines appears in figure 2.1. Obviously, the coarse surface pictured will not result in good control — it is unlikely that the car will stop at the desired point.

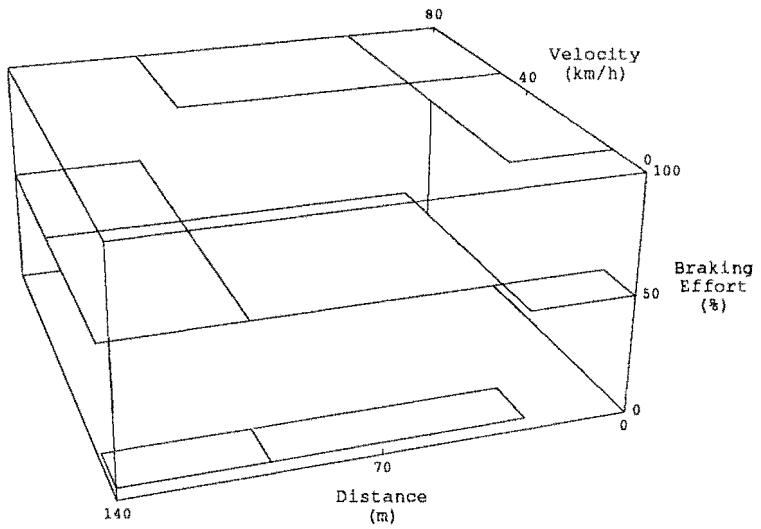


Figure 2.1: *Control surface generated by a bivalent rule base*

The stepped surface is not bad in itself — quantisation is always a factor in digital control — but the size of the bivalent rule base needed to gain adequate resolution for good control would be prohibitive.

One solution to this problem is to use fuzzy logic. The central idea of fuzzy logic is that membership in sets may be partial; the practical effect of this concept is a smoother control surface. A fuzzy logic controller may be viewed as a combination of a bivalent rule base and an interpolative mechanism.

Figure 2.2 shows a non-quantised control surface generated by the fuzzy equivalent of the bivalent rule base in table 1. The process of fuzzy control is explained in the next section.

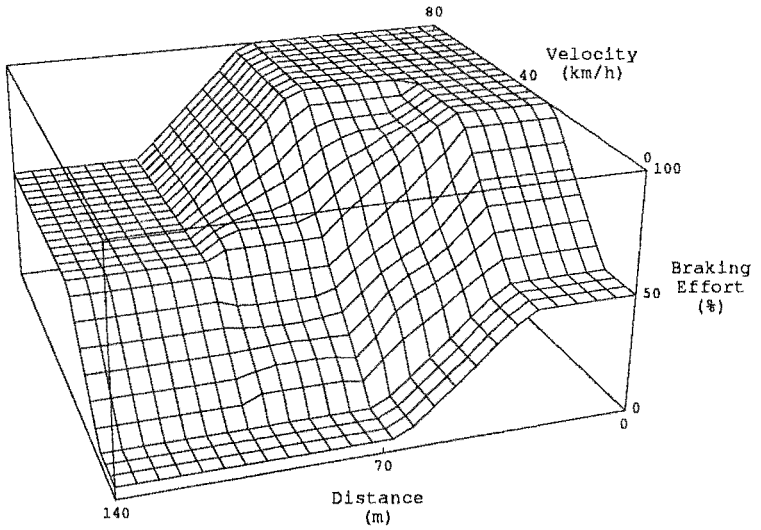


Figure 2.2 *Control surface generated by a fuzzy rule base*

2.2 Fuzzy Sets

A problem arises when classical sets are used to model vague concepts like "a medium velocity." The classification of velocities into the set **med** is difficult because it is not obvious which velocities belong to the set. Is 10 km/h a medium velocity? If it is, is then 9.99 km/h a medium velocity? A distinction between **med** and not **med** must be made somewhere, so at some point a group of very similar velocities must be divided into two differing classifications.

Fuzzy logic addresses this problem by allowing graded set membership. All elements of a fuzzy set have a membership grade in the interval [0,1]. The interval [0,1] corresponds to the truth set {0,1} used in bivalent logic (where 1 means true and 0 means false). So while 30 km/h may be considered an archetypal medium velocity, and assigned a membership of 1.0 in the set **med**, 20 km/h may be considered to be only 0.5 **med**, and it may be decided that 9.99 km/h is not a medium velocity (and so it would be assigned a membership of 0.0 in the set **med**.)

2.2.1 Membership functions

A function that maps a set of elements to their corresponding grades of truth in a fuzzy set is called a *membership function*. The membership of an element x; $x \in \mathbf{X}$, in the fuzzy set A is denoted:

$\mu_A(x)$

and **X** is called the *universe of discourse* of A.

If $\forall\, x_1, x_2, x_3 \in \mathbf{X}$

$x_1, x_2, x_3 \rightarrow \mu_A(x_2) \geq \min(\mu_A(x_1), \mu_A(x_3))$

(that is, the membership function either increases or decreases monotonically either side of its maximum) then the set **A** is called a *convex* fuzzy set.

Figure 2.3 shows the fuzzy set **med** that will be used in this thesis to model the concept of a medium velocity. It is convex.

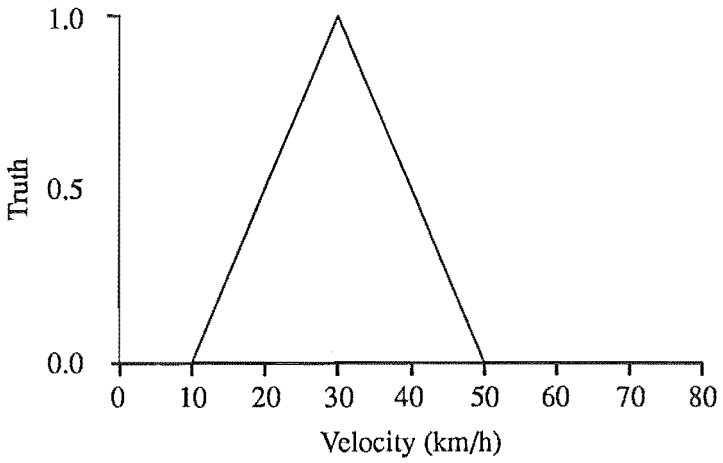


Figure 2.3: *A membership function for the set* **med**.

2.2.2 Fuzzy Relations

A fuzzy set with a membership function in two or more variables is called a *fuzzy relation*. Fuzzy relations can be used to represent the relationship between two or more variables in the same way that an ordinary relation can. For example, the relation $x = y$ defines the set of pairs $\{\{x, y\} \mid x \in \textbf{Real}, y \in \textbf{Real}, x = y\}$. A fuzzy relation may be constructed that defines a similar but less precise relation, for example the linguistic statement "approximately equal".

Jager (1995, p. 34) gives the following example of a membership function expressing the concept $x \approx y$:

$\mu_{\approx}(x, y) = \max(0, 1 - 0.5|x - y|)$

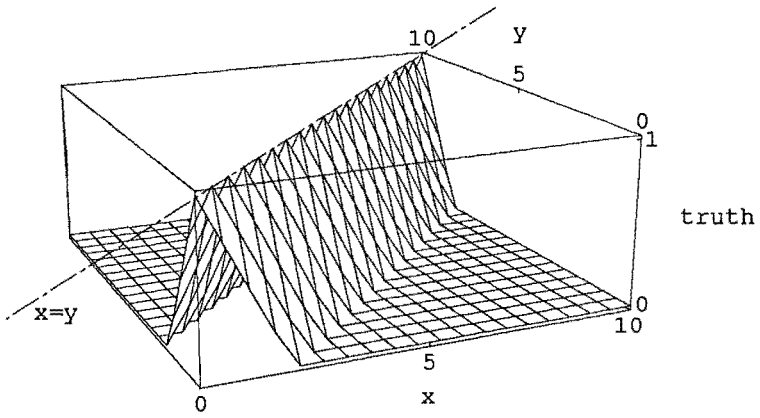


Figure 2.4: *A fuzzy relationship expressing $x \approx y$.*

2.2.3 Height

The *height* of a fuzzy set is defined as the supremum of its membership function over its universe of discourse:

$$\text{hgt}(\mathbf{A}) = \sup_{x \in \mathbf{X}} \mu_{\mathbf{A}}(x)$$

A fuzzy set with a height of 1 is called a *normal* fuzzy set, a fuzzy set with a height less than 1 is called a *subnormal* fuzzy set.

The set in figure 2.3 has a height of 1, and is thus a normal fuzzy set.

2.2.4 Support

The *support* of a fuzzy set **A** is defined as the set of elements that have a non zero membership in **A**:

$$\text{supp}(\mathbf{A}) = \{x \in \mathbf{X} \mid \mu_{\mathbf{A}}(x) > 0)\}$$

The support of the set in figure 2.3 is the interval (10, 50).

2.2.5 Core

The *core* of a fuzzy set **A** is defined as the set of elements that have a membership of 1 in **A**:

$$\text{core}(\mathbf{A}) = \{x \in \mathbf{X} \mid \mu_{\mathbf{A}}(x) = 1)\}$$

The core of the set in figure 2.3 consists of a single element, {30}.

2.2.6 Fuzzy Partitions

A group of sets on the universe **X** form a *fuzzy partition* on **X** if for all $x \in \mathbf{X}$ the sum of the membership grades of x in the sets is 1. That is, a group of n sets ($\mathbf{A}_1$, $\mathbf{A}_2$, ..., $\mathbf{A}_n$) on the universe **X** is called a fuzzy partition when:

$$\forall x \in X, \sum_{i=1}^{n} \mu_{A_i}(x) = 1$$

Provided that there is more than one set and none of the sets are empty. A fuzzy partition that is formed from normal, convex sets does not contain more than two overlapping sets.

2.3 Operations on Fuzzy Sets

2.3.1 Union and Intersection

Fuzzy sets can be combined by union and intersection in a similar manner to classical sets. Zadeh (1965) defined the union of two fuzzy sets by the maximum of their membership functions:

$$\mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x))$$

And the intersection of two fuzzy sets by the minimum of their membership functions:

$$\mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x))$$

Figure 2.5 gives an example of these operators in use:

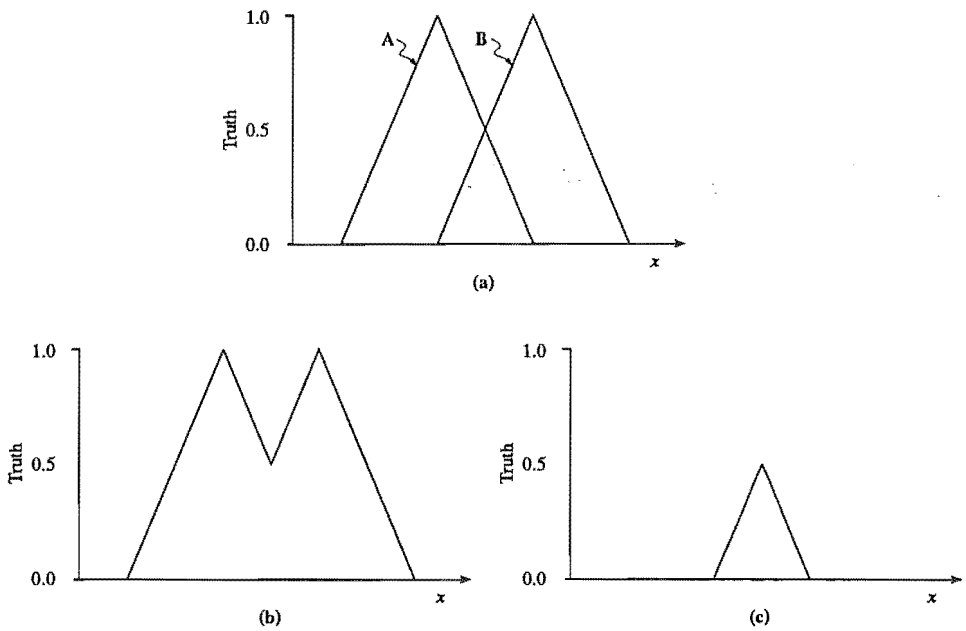


Figure 2.5: *(a) Two fuzzy sets,* **A** *and* **B**, *and (b) their union* $\mu_{A \cup B}(x) = \max(m_A(x), m_B(x))$, *and (c) their intersection* $\mu_{A \cap B}(x) = \min(m_A(x), m_B(x))$.

These are not the only definitions of union and intersection, but they are the most widely used in practice.

Many other operators have been suggested for the union and intersection of fuzzy sets.

To varying degrees all the suggestions are based on three main criteria:

1) The result of their operation is intuitive (that is, it models the human notion of intersection/union between two fuzzy sets).

2) They have properties that may be exploited in other areas of fuzzy logic.

3) The calculation can be easily performed.

Any binary operator that maps $[0,1] \times [0,1]$ to $[0,1]$ may be used to model intersection or union. With 1) and 2) in mind, the functions that are used to model intersection are restricted to *triangular norms* (t-norms), and those that are used to model union are restricted to *triangular conorms* (t-conorms or s-norms).

T-norms are binary operators that map the space $[0,1] \times [0,1]$ to $[0,1]$ and conform to the following criteria:

(1) $T(a, 1) = a$

(2) $T(a, b) \leq T(c, d)$ whenever $a \leq c$ and $b \leq d$

(3) $T(a, b) = T(b, a)$

(4) $T(T(a, b), c) = T(a, T(b, c))$

S-norms conform to the criterion:

(1) $S(a, 0) = a$

and (2), (3), (4) above.

A summary of the various T-norms and S-norms that have been suggested is given by Bellman (1973). For the purposes of this thesis, the norms suggested by Zadeh (1965) suffice.

2.3.2 Degree of Matching

The *degree of matching* between two relations is defined as:

$\text{match}(\mathbf{A}, \mathbf{B}) = \text{hgt}(\mathbf{A} \cap \mathbf{B})$

2.3.3 Projection

If a fuzzy relation $\mathbf{Z}$ is a fuzzy subset of the space $\mathbf{X}^i \times \mathbf{Y}^j$, where $\mathbf{X}^i = \mathbf{X}_1 \times \ldots \times \mathbf{X}_i$ and $\mathbf{Y}^j = \mathbf{Y}_1 \times \ldots \times \mathbf{Y}_j$, then the *projection* of $\mathbf{Z}$ in $\mathbf{X}^i$ is defined:

$$\text{proj}(\mathbf{Z}; \mathbf{X}^i) = \mu_{\mathbf{Z}\text{proj}\mathbf{X}^i}(x_1, \ldots, x_i) = \sup_{\mathbf{Y}^j} \mu_{\mathbf{Z}}(x_1, \ldots, x_i, y_1, \ldots, y_j)$$

The projection of a fuzzy relation is the profile of the relation as it appears from $\mathbf{X}^i$ (the set of supremums over $\mathbf{Y}^j$).

2.3.4 Cylindrical Extension

If a fuzzy relation, **Z**, is a fuzzy subset of the space $\mathbf{X}^i$, where $\mathbf{X}^i = \mathbf{X}_1 \times \ldots \times \mathbf{X}_i$, then its *cylindrical extension* of **Z** into the space $\mathbf{X}^i \times \mathbf{Y}^j$, where $\mathbf{Y}^j = \mathbf{Y}_1 \times \ldots \times \mathbf{Y}_j$, is defined:

$\text{cext}(\mathbf{Z}; \mathbf{X}^i \times \mathbf{Y}^j) = \mu_{\mathbf{Z}\text{cext}\mathbf{X}^i \times \mathbf{Y}^j}(x_1, \ldots, x_i, y_1, \ldots, y_j) = \mu_{\mathbf{Z}}(x_1, \ldots, x_i)$
so cylindrical extension simply expands the product space of a relation.

2.3.4 Composition

Cylindrical extension and projection can be used to infer a set on one universe given a set on another universe and a fuzzy relation between the two universes. The process, called *composition*, is conducted by taking the cylindrical extension of the set to be composed, finding the intersection of the extension and the relation, and projecting the result into the second universe. For example, given a set **A** on a universe **X** and a fuzzy relation **R** in the space $\mathbf{X} \times \mathbf{Y}$, a set **B** can be inferred by composition:

$\mathbf{B} = \text{proj}(\mathbf{R} \cap \text{cext}(\mathbf{A}; \mathbf{X} \times \mathbf{Y}); \mathbf{Y})$

The membership function of **B** is then:

$\mu_{\mathbf{B}}(y) = \sup \min(\mu_{\mathbf{A}}(x), \mu_{\mathbf{R}}(x, y))$

Composition is annotated with the symbol $\circ$, so the above would be written:

$\mathbf{B} = \mathbf{A} \circ \mathbf{R}$

Composition is analog to the process of "lookup" that may be used to find pairs of sets defined by discrete relations. Take, for example, the discrete relation $x = y$, which defines the set of pairs $\{\{x, y\} \mid x \in \mathbf{X}, y \in \mathbf{Y}, x = y\}$. A specific member of **Y**, y_i, can be extended into the space $\mathbf{X} \times \mathbf{Y}$ to create the set of pairs $\{\{x, y_i\} \mid x \in \mathbf{X}\}$. Taking the intersection of the two sets yields a single pair $\{x_i, y_i\}$ that can be projected onto **X** to yield the value of x_i that the relation $x = y$ maps y_i to. A similar process is used when working with relations that are represented with Cartesian graphs.

2.4 Fuzzy Rules

To model a rule of the form "if **A** and **B** then **C**" (where **A**, **B**, and **C** are fuzzy sets) with fuzzy logic, two definitions need to be made: how the connective, "and," is modelled in fuzzy logic; and how implication, "if...then" is modelled.

2.4.1 Fuzzy Connectives

The connective "and" is usually modelled with the fuzzy intersection (from 2.3.1, the min operator), and the connective "or" is usually modelled with fuzzy union (the max operator). Thus, if two propositions on the same universe are linked by a connective, the result is a fuzzy set on that universe, and if two fuzzy propositions on differing

universes are linked by a connective the result is a fuzzy relation on the product space of the two universes.

2.4.2 Fuzzy Implication

There are several models of fuzzy implication. In general, the fuzzy relation mapping the antecedent of a rule to the consequent of a rule is annotated:

$\mathbf{R} = \mathrm{I}(\mathbf{A}, \mathbf{C})$

where **R** is the fuzzy relation that defines the rule, **A** is the fuzzy relation that defines the antecedent of the rule, and **C** is the fuzzy relation that defines the consequent of the rule.

The most popular definitions of $\mathrm{I}(a, c)$ are min inference, where

$\mathrm{I}(a, c) = \min(a, c)$

and product inference, where

$\mathrm{I}(a, c) = ac$

2.4.3 Reasoning with a Fuzzy Rule

Zadeh (1973) introduced the *compositional rule of inference* along with the above definitions of a fuzzy rule. It states that, given a rule that has been modelled with the fuzzy relation **R**, an output set **C'** can be inferred from an input set (or *proposition relation*) **P** by composition:

$\mathbf{C'} = \mathbf{P} \circ \mathbf{R}$

For example, if **R** describes the rule "if **A** and **B** then **C**", the proposition "*a* is **A** and *b* is **B**" (where *a* and *b* are fuzzy variables) can be tested by constructing the input relation "**P** = *a* and *b*" and composing it with **R**. The resulting set, on the universe of **C**, will be equal to **C** if the proposition is true; will be some subset of **C** if the proposition is partly true; and will be empty if the proposition is false. The match between **C** and **C'** gives a fuzzy measure of the truth of the proposition.

It can be shown that, when using the min operator for conjunction (c.f. §2.4.1) and implication (c.f. §2.4.2), a major simplification can be made to the calculations required for compositional inference (Jager, 1995, p. 68-71). In this case composition reduces to clipping the consequent relation to the degree of matching (c.f. §2.3.2) between the input relation and the antecedent relation (called the *input match*). Further, the input match can be determined by finding the minimum degree of matching that occurs between any input/antecedent set pair on the same universe of discourse (for example, "**a** is **A**" defines a pair).

If the min operator is used for conjunction and the product operator is used for implication, it can be shown that composition reduces to scaling the consequent relation by the input match (Jager, 1995, p. 68-71).

By way of example, the rule base in table 2.1 will be recast as a fuzzy rule base, and used to demonstrate the above methods of fuzzy reasoning.

2.5 An Example of Inference with a Fuzzy Rule

The sets from table 2.1 may be modified for use in a fuzzy rule base. Some simple membership functions for the sets on **S** are shown in figure 2.6, and some simple membership functions for the sets on **D** appear in figure 2.7. Note that the sets form a fuzzy partition.

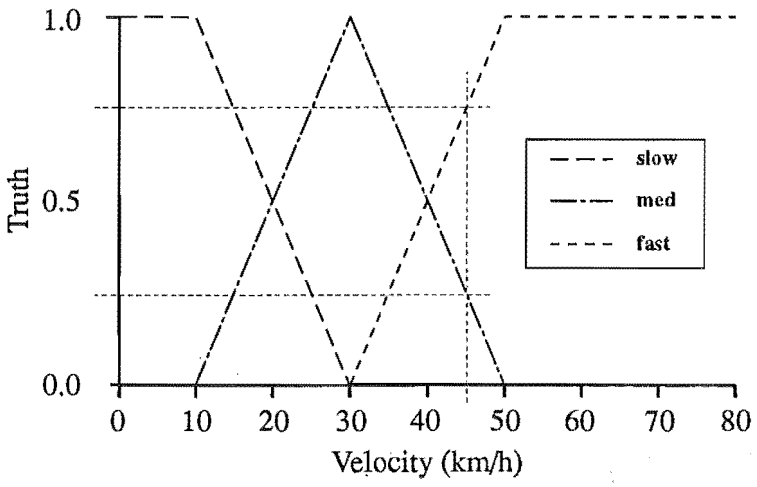


Figure 2.6: *Membership functions on* **S**.

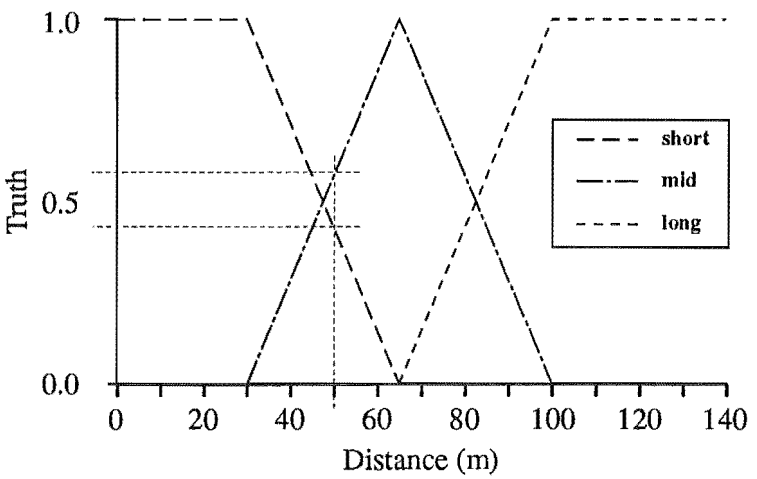


Figure 2.7: *Membership functions on* **D**.

The braking efforts used in the consequent sets of the rules, 5%, 50%, and 100%, are

replaced with the fuzzy sets **light**, **average**, and **hard** (figure 2.8).

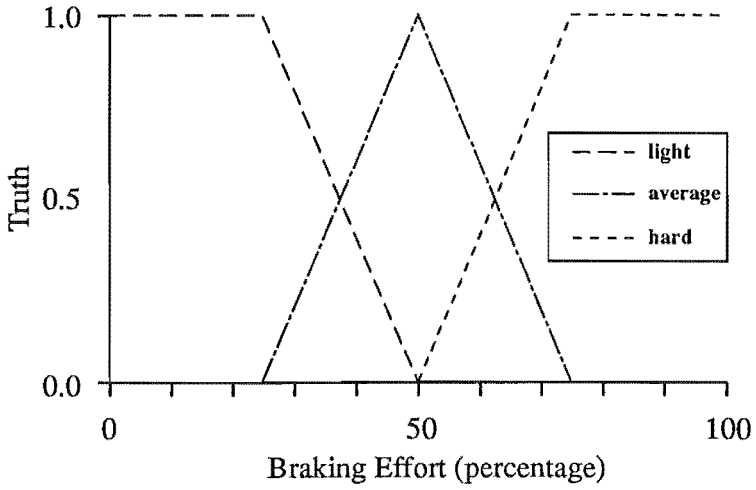


Figure 2.8: *Membership functions on* **B**.

The hairlines on figures 2.6 and 2.7 are guides for the example condition where s = 45 km/h and d = 50 m.

In this case, the rule "if s is **med** and d is **mid** then b is **average**" would be interpreted as follows:

Firstly, the degree of match between the input relation, $s \cap d$, and the antecedent relation, **med**$\cap$**mid**, must be found. That is,

match($s \cap d$, **med**$\cap$**mid**)

must be calculated. From §2.4.3, this can be calculated by finding the minimum match between an input/antecedent set pair. In this case the pairs are "s is **med**" and "d is **mid**", and the minimum degree of match is:

min(match(s, **med**), match(d, **mid**))

or, substituting from §2.3.2,

min(hgt($s \cap$**med**), hgt($d \cap$**mid**))

Where an input set is discrete (as is the case in control engineering) it must be *fuzzified* before it can be intersected with a fuzzy set. The possibility exists that a discrete input be represented with a fuzzy set that models its precision (for example, sensor precision), but this is seldom done in practice. The usual method of fuzzification is to represent the input with a *fuzzy singleton*, which is a singleton set with an associated truth grade of 1.

The intersection of a singleton and non-singleton fuzzy set is:

$s \cap$**med** = min($m_s(s)$, $m_{med}(s)$)

where s is a singleton representing some discrete value of s, say s_{input}:

$\mu_s(s) = \{1, \text{if } s = s_{input}; 0, \text{otherwise}\}$

therefore,

$s \cap \mathbf{med} = \{\min(1, \mu_{med}(s)), \text{if } s = s_{input}; \min(0, \mu_{med}(s)), \text{otherwise}\}$

$= \{\mu_{med}(s), \text{if } s = s_{input}; 0, \text{otherwise}\}$

The height of the resulting singleton is then the membership of the discrete value in the fuzzy set. This further simplifies the calculations required to find the input match:

$$\text{match}(s \cap d, \mathbf{med} \cap \mathbf{mid}) = \min(\text{match}(s, \mathbf{med}), \text{match}(d, \mathbf{mid}))$$

$$= \min(\text{hgt}(s \cap \mathbf{med}), \text{hgt}(d \cap \mathbf{mid}))$$

$$= \min(\mu_{med}(s), \mu_{mid}(d))$$

when s and d are fuzzy singleton models of the discrete values s and d respectively.

The task is now to find the minimum membership of an input in its associated antecedent set. From figure 2.6, the membership $\mu_{med}(s)$ for the example condition $s = 45$ is 0.25. From figure 2.7, the membership $\mu_{mid}(d)$ for the example condition $d = 50$ is 0.57. The minimum of these two values is 0.25, which is the input match for the example conditions.

Having found the input match, composition can be completed by modifying the consequent relation (in this case the set **average**). From§ 2.4.3, when using the min operator for implication the output set is created by clipping the consequent set by the input match (figure 2.9), and when using the product operator for implication the output set is created by scaling the consequent set by the input match (figure 2.10).

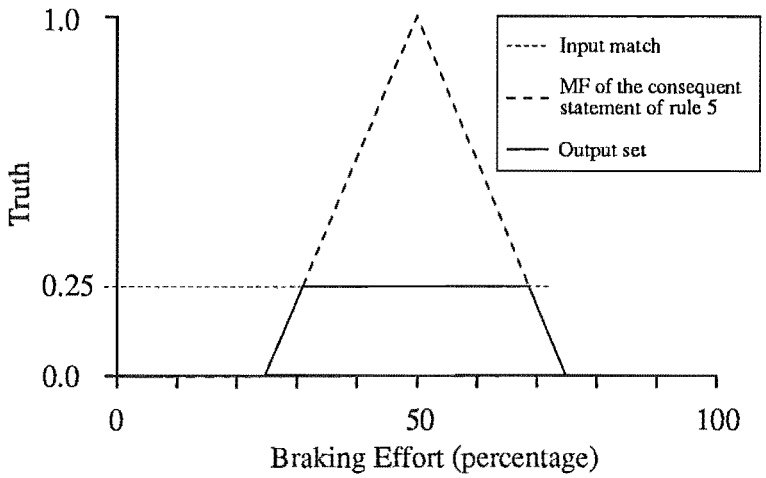


Figure 2.9: *An output set created by composition with min implication*

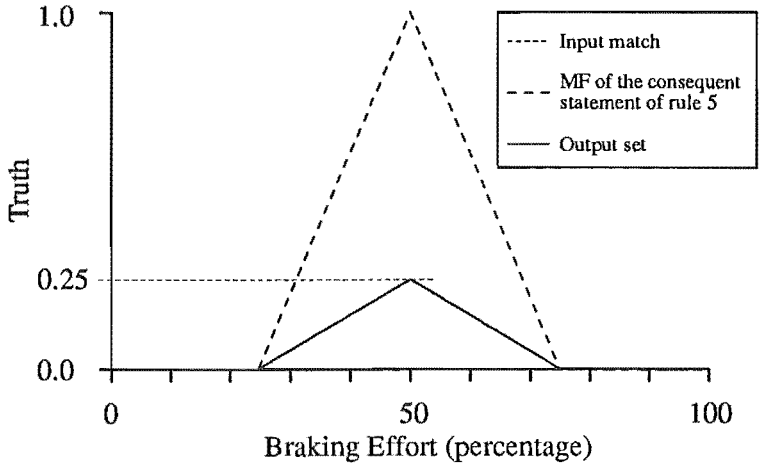


Figure 2.10: *An output set created by composition with product implication*

Kosko (1992, p. 312) gives some motivation to select product implication as the method of choice, in that it "preserves more information" than minimum implication. This point is largely irrelevant when simple membership functions are used, because simple membership functions contain little information to begin with (consider the case of a trapezoidal membership function—composition by minimum implication does not alter its shape). In practice most fuzzy rule bases use simple membership functions (triangles or trapezoids) in their consequent statements, so it would seem that there is little reason to choose one method of implication over the other.

In any case, the software developed in this thesis uses discrete sets in its consequent statements, and minimum and product implication produce identical output sets under this condition.

2.6 The Fuzzy Rule Base

To construct the fuzzy rule base described by a set of fuzzy rules, the relations that describe each individual rule are *aggregated* into a single relation. Aggregation is usually conducted by taking the union of the rule relations.

2.6.1 Reasoning with a Fuzzy Rule Base

Inference with a fuzzy rule base is conducted in the same way as inference with a single rule — the input relation is composed with the rule base relation to infer an output relation:

$$\mathbf{C'} = \mathbf{P} \circ \mathbf{R}$$

However, C' cannot be found as easily as for a single rule. To ease the computational burden a scheme called *local inference* is used. In local inference, composition is conducted with each individual rule relation and the resulting output relations are

aggregated by taking their union. It can be shown that, for methods of composition employing T-norms (such as the min and product operator) for implication, local inference is equivalent to normal, or *global*, inference.

Some confusion exists in the literature over the difference between composition and aggregation — many authors refer to aggregation after local inference as "composition," probably because it produces an equivalent set to composition with global inference.

The previous example will be extended in section 2.7 to illustrate inference using a fuzzy rule base.

2.6.2 The Completeness of a Rule Base

It is useful to have some measure of the quality of the information that is inferred from an input relation for a given rule base. The standard measure is the *completeness* of the rule base, defined:

$$\mathrm{CM}(x) = \sum_{i=1}^{r}\left\{\prod_{j=1}^{n}\mu_{A_{i,j}}(x_i)\right\}$$

where r is the number of rules in the rule base, n is the number of input universes, and x is a vector containing the inputs. A rule base is said to be *incomplete* at a point x if $\mathrm{CM}(x) = 0$. A rule base will produce no output from input relations that lie within its incomplete regions. Points with a completeness between 0 and 1 are called *subcomplete* points; points with a completeness of 1 are called *strict complete*, and points with a completeness greater than 1 are called *overcomplete*.

If the sets on each input universe of a rule base form fuzzy partitions, and the rule base contains one rule for each possible combination of input sets, then the rule base will be strict complete with respect to all points within the product space of its input universes. If some of the rules from such a rule base are omitted, some parts of the rule base will be subcomplete, or possibly incomplete. A rule base will produce no output from a proposition relation that falls within an incomplete region.

2.7 Example Continued

Continuing with the example given in section 2.5, the rule base specified in table 2.1 can be used together with the fuzzy sets specified in figures 2.6, 2.7, and 2.8 to infer an output set from the example conditions $s = 45$ and $d = 50$. Firstly the input match for each rule must be determined. From section 2.5, the input match for rule 5 is 0.57. All of the other rules in the table have an input match of 0, except for rule 3 (if s is **fast** and d is **short** then b is **hard**), which has an input match of 0.43:

$$\text{match}(s \cap d, \textbf{fast} \cap \textbf{short}) = \min(\mu_{\text{fast}}(s), \mu_{\text{short}}(d))$$

$$= \min(0.75, 0.43)$$

$$= 0.43$$

The output set generated by composition using product implication appears in figure 2.11. All of the output sets generated by the rules with zero input match are empty, so the output set of the entire rule base can be found by aggregating the output sets from rules 3 and 5. The output set of the rule base is the union of these sets. It is depicted in figure 2.12.

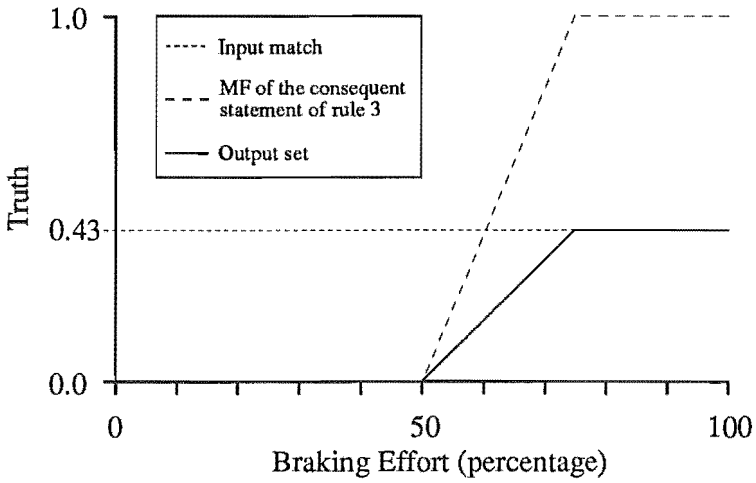


Figure 2.11: *The output set of rule three, inferred from the inputs* s = *45 and* d = *50 by composition using product implication.*

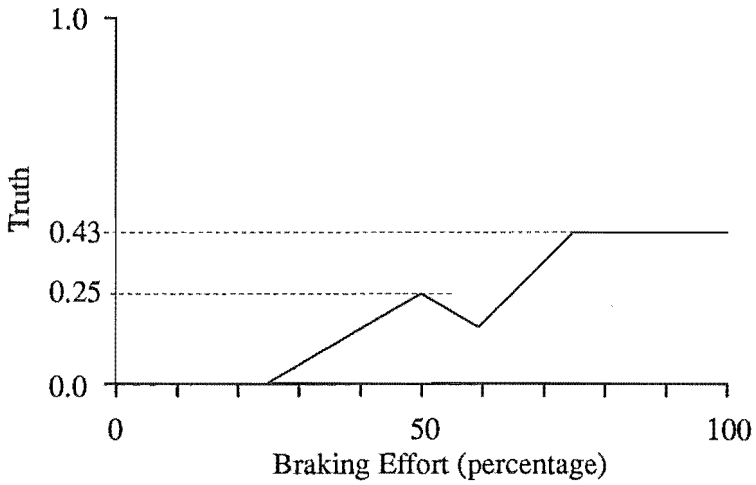


Figure 2.12: *The aggregated outputs of rules three and five.*

2.8 Defuzzification — Discrete Output from Fuzzy Rule Bases

If the output of the rule base is to be used for control it must be transformed from fuzzy set form into a more usable form, usually a discrete value.

The process of transformation is called *Defuzzification.*

2.8.1 Maximum-Membership Defuzzification

There are several popular defuzzification schemes, the simplest is called *maximum-membership defuzzification*, in which the point of maximum truth is chosen as the output value. This method has two fundamental problems, as Kosko (1992, p. 315) points out.

Firstly, the point of maximum truth may not be unique. Most notably, this occurs when an output set has a membership function with a flat top (resulting, for example, from composition with the min implication), and results in a region of maximum truth rather than a point. This problem is usually resolved by averaging multiple points.

Secondly, the output value tends to change in discontinuous steps when the point of maximum truth moves from one output set to another. In practice, maximum-membership defuzzification finds most use as a method of conflict resolution for bivalent rule bases.

2.8.2 Centroidal Defuzzification

A more sophisticated alternative is *centroidal defuzzification*, in which the abscissa of the centre of gravity of the output membership function is chosen as the discrete output value (hereinafter *the abscissa of the centre of gravity of the membership function of a set* will be referred to as the *centre* of the set).

Kosko (1986) has suggested a modification to classical fuzzy logic which simplifies the process of finding the centre of the output set. He notes that aggregation by summation (rather than the classical method of union) preserves the area of the local output membership functions. Thus, the centre of an output set produced by additive aggregation is the same as that found by combining the centres of the local output sets. Therefore, in the situation where additive aggregation and defuzzification by the centroidal method is used, aggregation (and its accompanying calculations) may be omitted.

Additive aggregation does not conform to the classical model of fuzzy logic; it admits the possibility of supernormal output sets, which have no defined meaning. It is a modification of the model that is made in an attempt to justify the omission of the aggregation step. As defuzzification is a heuristic process to begin with, the proof of

the validity of this modification is its utility, rather that its consistency with fuzzy theory. There are further reasons (beyond the simplification of the necessary calculations) to make this modification. They will be presented in the remainder of this chapter and in the next.

The example calculation from section 2.7 may now be concluded.

In the example situation the rule base has two non-zero output sets, generated by rules 3 and 5. They appear in figure 2.11.

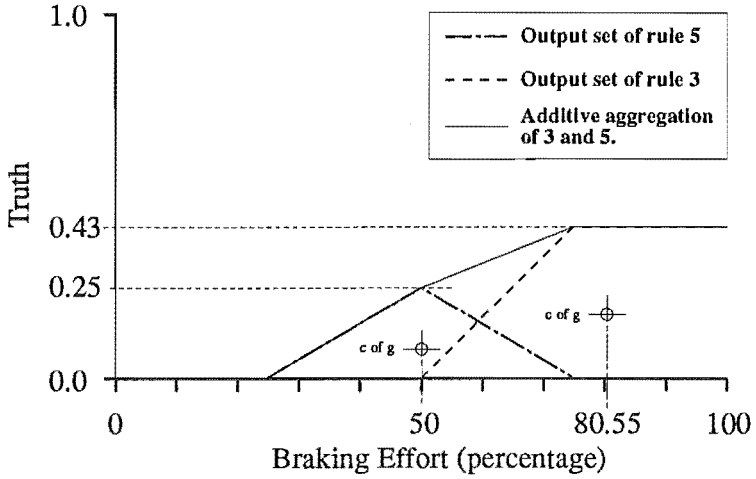


Figure 2.11: *Some example output sets.*

The centres of the output sets of rules 3 and 5 occur at $b = 80.55\%$ and $b = 50\%$ respectively. Their areas are 16.13 and 6.25.

Taking moments about 0, the combined centre of the two sets may be calculated as follows:

$$c_t = \frac{a_3c_3 + a_5c_5}{a_3+a_5}$$

;where a_3 = the area of the output set of rule 3
a_5 = the area of the output set of rule 5
c_3 = the centre of output set 3
c_5 = the centre of output set 5
c_t = the combined centre of the two sets.

Now,

$$c_t = \frac{16.13 \times 80.55 + 6.25 \times 50}{16.13 + 6.25}$$

$$= 72.02$$

Which is the defuzzified output of the rule base for the example conditions $s = 45$ and

$d = 50$.

2.8.3 Consequent Set Modelling

The defuzzification calculations may be simplified by noting that product implication scales the area of the consequent set membership function linearly, and does not move the abscissa of the centre of gravity of the function. Consequent sets may therefore be represented by their area and centre alone, a representation familiar to engineers as the "lumped mass model."

For example, the area of the consequent set **average** is 25, and its centre falls at $b = 50\%$. Product inference can be conducted simply by multiplying the area of the consequent set by the input match. In the case of the example this is $50 \times 0.25 = 6.25$, which is the area of the output set of rule 5. The centre of the output set is the same as that of the consequent set, that is, 50.

This method of modelling consequent sets has obvious parallels with the use of singleton sets in consequent statements. The model is a singleton set (the centre of the set) with an associated weighting.

To make an intelligent choice of weighting, some knowledge of the effects of consequent set area on the control surface is needed. It can be shown that, if linear membership functions are used, a choice of identical weights for all consequent sets results in linear output. A short proof of this and a description of the effects of alternate weighting schemes are given in section 3.2.2.

The choice of a weighting of 1 is desirable because it simplifies the defuzzification calculations.

Under the above conditions the example rule base with the singleton consequent statements $b = 5\%$, $b = 50\%$, and $b = 100\%$ (Table 2.1) would be defuzzified as follows:

$$c_t = \frac{1 \times (50 \times 0.25) + 1 \times (100 \times 0.43)}{0.25 + 0.43}$$

$$= 81.62$$

This output value is different from that obtained when using the fuzzy consequent sets from figure 2.8, simply because the sets are not equivalent—that is, the crisp sets used are not singleton models of the fuzzy sets given in figure 2.8.

The crisp sets result in better output, as can be seen in the example where all output is derived from the maximal braking set, which occurs when only rules 2, 3, or 6 produce output (in which case one would expect maximum braking). In this situation centroidal defuzzification will pick the centre of the set that represents maximal

braking as an output, because no other sets make a contribution. The use of a crisp set with a centre at 100 will produce 100% braking, whereas the use of the fuzzy set **hard**, with its centre at 80.55, will produce only 80.55% braking.

This is not due to any fault in the fuzzy consequent sets — they adequately model the concepts of light, average, and hard braking — but is, rather, an artefact of centroidal defuzzification.

The determining property of consequent sets is the position of the centroids of their membership functions, the abscissa of which falls somewhere around the middle of the set. Accordingly, the centroid of a consequent set should be placed so that correct defuzzification occurs when all output is derived from that set. Furthermore, the centroid should coincide with the point of maximum membership so that the membership function can be easily read and manipulated by designers. These two restrictions lead to counter-intuitive set constructions—for instance, to place the centroid of the set **hard** at 100% would require that it span the range of braking from 50% to 150%; the need for such contortions indicates that there is something wrong with the system as it is currently described.

To understand the problem some knowledge of the origins of fuzzy logic and its use in control is needed. When Lotfi Zadeh developed fuzzy logic in 1964, he thought that it would find use mainly in the fields in which analytic techniques had been ineffectual; for work outside of the hard sciences in fields such as philosophy, psychology, linguistics and biology (Perry 1995). He designed the system to manipulate the vague and qualitative notions of sets that humans use — the use of fuzzy logic in control engineering and other fields requiring quantitative output came later.

When Mamdani made the first use of fuzzy logic as a control method he was seeking a way to incorporate expert knowledge into a controller (Mamdani 1975). The antecedent side of Zadeh's fuzzy rule base was well suited to Mamdani's purpose, but the consequent side needed modification to make its output usable as a control signal. Mamdani used a heuristic to convert the fuzzy output of his rule base into a control signal — defuzzification.

Many defuzzification schemes have been suggested with the aim of providing a better interface between the fuzzy output of a rule base and the discrete output required by controllers (Mizmoto 1989). In practice, however, centroidal defuzzification is used most often because it is simple and fast. The problem of non-intuitive output is circumvented by working directly with discrete consequent sets — an approach that Sugeno (1985) is generally credited with formalising.

2.9 Summary

A tutorial introduction to the use of fuzzy logic in control has been presented, and the various aspects of fuzzy logic that effect the efficiency of fuzzy inference in fuzzy controllers discussed. It has been shown that an efficient controller can be built using the following methods for inference:

1) The min operator for conjunction in rules.

2) The product operator for implication.

3) Fuzzy singleton modelling of input sets.

4) Additive local aggregation.

5) Centroidal defuzzification.

6) Lumped mass modelled (singleton) output sets.

3. Fuzzy Rule Base Design

This chapter explains the various methods used to develop fuzzy logic controllers. It begins by describing a method that may be used to construct an arbitrary control surface, and works through several popular development strategies. The latter part of this chapter describes how the various methods may be used together for greater flexibility.

3.1 Rule Base Legibility

One of the most useful properties of the fuzzy controller arises from its rule based design — it is easy for a designer to predict how a controller will behave because its action is described in simple, English-like statements.

This openness makes tuning the controller easy — an important property when initial design work is done with a simulated plant, because computer models seldom match real systems exactly, and fine tuning is usually required after the controller is installed in the real plant.

Both of the above claims — that fuzzy controllers are open and easy to tune — are only true if the "simple, English-like statements" that describe a controller are not confusing or misleading. Unfortunately, it is quite possible to write illegible fuzzy controllers. It is difficult to predict the output of long, complicated rules constructed with many statements and a variety of operators. Even if the rules are kept simple, multiple rules may produce output from the same inputs, making lengthy calculations necessary before their combined (defuzzified) output can be found. And even if all rules are simple and produce cohesive output, the geometry of the control surface is hard to predict because it depends on so many variables — set overlap, membership function shape, and output set position and weighting.

Most examples of fuzzy controller design reflect the need for simple rules, simple sets, and simple membership functions. However, the literature omits descriptions of methods that may be used to translate between rule base and control surface, and usually authors provide little justification for using the membership functions and rule forms that they do select. Typical is Kosko (1992, p. 382), who presented the following heuristic:

> *Fuzzy membership functions can have different shapes depending on the designer's preference or experience. In practice fuzzy engineers have found triangular and trapezoidal shapes help capture the modeller's sense of fuzzy*

> *numbers and simplify computation. ...The [example] fuzzy controller uses trapezoidal fuzzy-set values.... The lengths of the upper and lower bases provide design parameters that we must calibrate for satisfactory performance. A good rule of thumb is* adjacent fuzzy-set values should overlap approximately 25 percent.

A more formal methodology for designing and reading fuzzy logic controllers follows. It allows the simple prediction of control surface geometry from a rule base. It may also be used to move the other way — from control surface to rule base — a technique which can be used to bring the openness and adjustability of fuzzy controllers to other controller designs.

3.2 The Well Conditioned Rule Base

The concept of a *well conditioned rule base* will now be developed. In this thesis the term will be used to refer to a rule base with sets and rules that meet certain criteria. Conformance to these criteria ensures that the output of a rule base can be easily read and modified; the form of the control surface can be deduced from the rule base without calculation and it can be modified easily with predictabe results.

To be well conditioned, a rule base must conform to 5 criteria.

1)	Each input must be represented exactly once in every rule.
2)	The propositions in every antecedent statement are connected with "and".
3)	Every set must have at least one unique member.
4)	The slopes of the membership functions of intersecting sets are opposite.
5)	The unique members of a set have a membership of 1.

Table 3.1

3.2.1 Rules as Points on a Control Surface

Taken together, the first three criteria provide a simple way to place *characteristic points* on a control surface. Every rule has a characteristic point. A characteristic point is a point on the control surface defined by the combination of input values for which only one rule produces output (the *characteristic input values* of that rule), and that output.

The first three criteria, and their utility, will now be addressed in turn:

1) Each input must be represented exactly once in every rule.

That is, the antecedent statement of each rule includes a proposition about each input (a proposition statement is a statement of the form: x is **A**; where x is the input and **A** is a fuzzy set).

2) The propositions in every antecedent statement are connected with "and".

Here, the "and" conjunction is assumed to be modelled with fuzzy intersection and performed with the min operator (c.f. §2.4).

3) Every set must have at least one unique member.

That is, every set must have at least one member with non-zero membership that has zero membership in all other sets.

The characteristic input values of a rule are simply the input values that are unique to the scope of the rule. By criterion three every set must contain at least one unique member. Therefore, for every proposition there must be some input value for which the input of the proposition matches (see §2.3.2 and §2.5 for an explanation and example of input matching) the antecedent set of that proposition and *matches no other antecedent set* (that is, no other proposition will have a non-zero match). If all the antecedents in a rule base consist only of membership statements combined by "and," then no antecedent will have a non-zero input match unless every one of its propositions have a non-zero match. It follows that, for a rule in a well conditioned rule base, some unique combination of input values exist for which it alone has a non-zero antecedent, and therefore it alone produces an output.

Because only one output set exists for each combination of characteristic inputs, defuzzification is not complicated by the necessity to combine the output sets of several rules, and the defuzzified output corresponding to a combination of characteristic points may be easily found. In the case where the consequent set that the output set is derived from is a singleton, or is represented by a lumped mass model, the defuzzified output can be read directly — it is simply the discrete value.

The procedure for placing a characteristic point is then:

a) For each of the characteristic input values of the point, design a set that uniquely includes the input value.

b) Write a rule that includes a membership statement for each input. It should take the form: x is **A**; where **A** is the set that contains the characteristic value of the input x.

c) Write a consequent statement for the rule of the form: y is **B**; where **B** is the value that output y should take on when the inputs match the characteristic values of the rule.

For example, when the rule base in chapter two was written, it was desired that the rule base should output a median braking signal when the distance and velocity inputs

were at the middle of their ranges. That is, it was desired that the rule base produce the output 50% when the velocity input is 30 km/h and the distance input is 65 m (the characteristic point {50, 30, 65}). To place this point, the set **med** was designed to uniquely include the velocity 30 km/h (Figure 2.6); the set **mid** was designed to uniquely include the distance 65 m (Figure 2.7); and the rule "if *s* is **med** and *d* is **mid** then *b* = 50%" was written (Table 2.1) to set the value of *b* when *s* is uniquely in **mid** and *d* is uniquely in **med**.

Because points on the control surface can be placed independently of each other, any surface that can be represented with a table can be represented with a fuzzy rule base. In practice the number of characteristic points needed to adequately define a particular control surface with a fuzzy rule base is small, because the rule base will interpolate any points not explicitly provided. Understanding the nature of the interpolative mechanism makes it easier to design controllers with a minimum number of rules, which makes them easier to read and faster operating.

3.2.2 Output Set Weighting and Control Surface Geometry.

Firstly, the nature of interpolation in a SISO fuzzy controller will be investigated.

Interpolation occurs where more than one rule produces output. The output is the defuzzified output of the rule base; in the case of centroidal defuzzification this is the weighted average of the centres of the output sets. In section 3.2.1 was shown that, in a rule base that conforms to the first three criteria, a single output set will be produced when all inputs are unique to the scope of a single set. Conversely, it can be deduced that multiple output sets will be produced when some inputs fall in the scope of more than one set; one output set being produced by each rule with a non-zero input match. Interpolation occurs when all or some of the inputs fall in regions where sets overlap.

In any rule base that conforms to criterion 3, all set overlaps involve exactly 2 sets (three or more continuous sets on a line cannot intersect without at least one set being a subset of another, a condition which contradicts the requirement that every set have at least one unique member).

The interpolative equation mapping the inputs to the outputs depends on the shape of the membership functions of the overlapping sets, the position of the centres of the output sets, and the weights associated with the output sets. The following derivation establishes the nature of the interpolative equation for a SISO controller:

Figure 3.1 shows a region of overlap of two membership functions. To make this derivation simpler, both membership functions are linear and continuous in the region. In the region in question, the equation of the membership function of set **a** is:

$$\mu_a(z) = k_a z + c_a \tag{3.1}$$

and that of set **b** is:

$$\mu_b(z) = k_b z + c_b \tag{3.2}$$

where z is the input variable, k_a and k_b are constants determining line slope, and c_a and c_b are constants determining line offset.

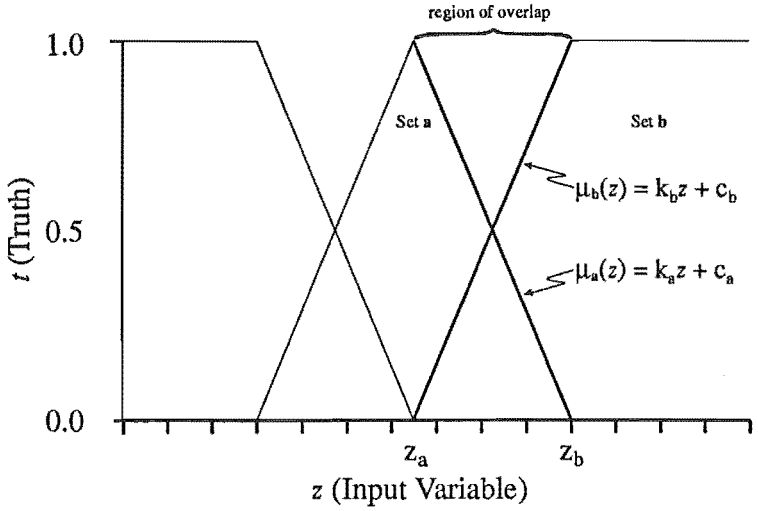


Figure 3.1: *A region of overlap in a fuzzy partition.*

Figure 3.2 shows some output sets. They have been represented using the lumped mass model; the centre of set **c** occurs at x_c, the centre of set **d** occurs at x_d; the centres are x_{cd} apart. The sets **c** and **d** have the associated weights w_c and w_d respectively.

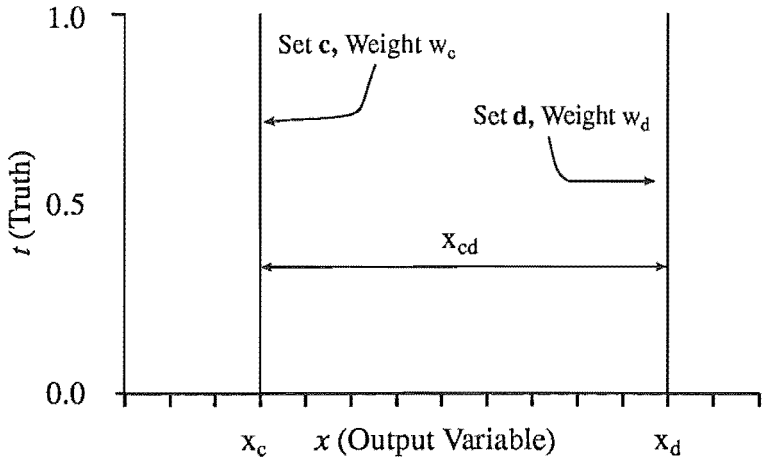


Figure 3.2: *Some examples of lumped mass modelled consequent sets.*

In the case of a single input system, two rules will produce output when the input lies in a region where two input sets overlap. For this derivation, the two rules are

represented arbitrarily:

if z is **a** then x is **c**
if z is **b** then x is **d**

These rules represent the characteristic points $\{z_a, x_c\}$ and $\{z_b, x_d\}$ respectively (see figure 3.1 for the location of the constants z_a and z_b, and figure 3.2 for x_c and x_d).

Inference proceeds as described in section 2.5, with each rule producing an output set. The output set produced by the first rule is a modification of set **c**, achieved by composition using product or minimum implication. As set **c** is discrete, both methods of implication result in the same output set — the result is identical to set **c**, but with a height clipped to match(z, **a**). In section 2.5, it was shown that match(z, **a**) = $\mu_a(z)$ when z is represented by a singleton. That is, the height of the output set is $\mu_a(z)$. In the region in question, $\mu_a(z)$ may be found using (3.1). Similarly, for rule 2, the height of the output set may be found from (3.2).

If centroidal defuzzification is used, the defuzzified output, x, is simply the combined centre of mass of the two output sets, the "mass" of each output set being the truth of the set multiplied by the weight associated with the set (c.f. §3.8.3).

Taking moments about x_c:

$$x = x_c + \frac{x_{cd}\mu_a(z)w_d}{\mu_a(z)w_c + \mu_b(z)w_d}$$

Which is the equation that describes how the output, x, varies with the input, z. Substituting from (3.1) and (3.2):

$$x = x_c + \frac{x_{cd}(k_b z + c_b)w_d}{(k_a z + c_a)w_c + (k_b z + c_b)w_d}$$

Collecting terms in z:

$$x = x_c + \frac{x_{cd}k_b w_d z + x_{cd}w_d c_b}{(k_a w_c + k_b w_d)z + c_a w_c + c_b w_d} \tag{3.3}$$

The term in z in the denominator of (3.3) will be zero when:

$$k_a w_c + k_b w_d = 0 \tag{3.4}$$

...yielding the linear equation:

$$x = k_t z + c_t \tag{3.5}$$

where

$$k_t = \frac{x_{cd} k_b w_d}{c_a w_c + c_b w_d}$$

and

$$c_t = x_c + \frac{x_{cd} w_d c_b}{c_a w_c + c_b w_d}$$

If $k_a = -k_b$, then (3.4) will be true when $w_c = w_d$, in which case equation 3.3 will reduce to the linear equation 3.5. That is, if the slopes of the overlapping input membership functions are opposite, and the weights associated with the output sets are equal, then the interpolative equation will be linear. This suggests the fourth criterion:

4) The slopes of the membership functions of intersecting sets are opposite.

In practice this is most simply achieved by using triangular or trapezoidal membership functions with opposing, overlapping sides (see figures 2.7 and 2.7). More elaborately shaped membership functions behave identically to these simple functions in the case where output set weightings are the same (that is, the output is still linearly interpolated). When the weightings differ, output is biased more towards heavily weighted sets when the input is in a region of high slope, and approaches linear interpolation when the input is in a region of low slope (see equation 3.4).

In a rule base that adheres to the first four criteria and uses linear membership functions, the nature of control surface interpolation is determined solely by the weights associated with the output sets. Figure 3.3 gives some examples of interpolation for differing set weightings.

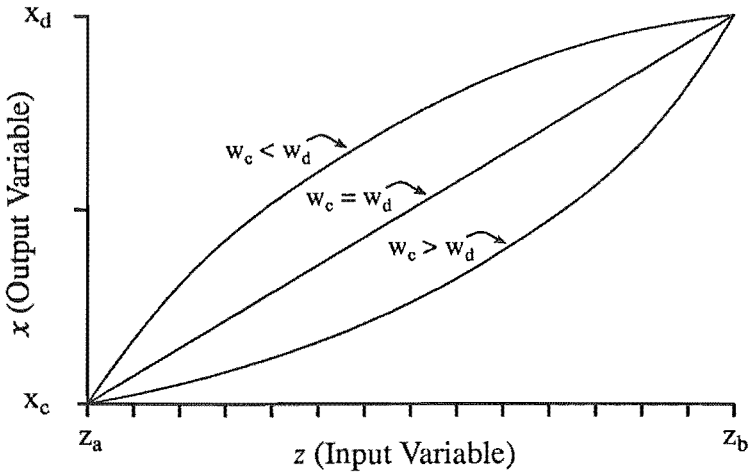


Figure 3.3: *The effect of varying the ratio of consequent set weightings.*

The behaviour illustrated in figure 3.3 is close to intuition — output is biased toward consequent sets with higher weightings, and varies about a neutral, linear midpoint

that occurs when the consequent set weightings are equal.

This result may be extended to higher dimensions (multiple input systems) by noting that multi-input rules behave like single input rules when the input match of one proposition in the antecedent of the rule varies and the others have a match of 1. That is, a statement

if x_1 is s_1 and x_2 is s_2 and x_3 is s_3... then...

reduces to:

if x_1 is s_1 then...

when all match(x_2, s_2) = match(x_3, s_3) =... = 1

because inference is conducted with the minimum proposition match (c.f. §2.3.3) and $\min(x, 1) = x$ when $x \in \{0...1\}$.

If all characteristic input values have a membership of 1 in their associated antecedent sets, then the nature of interpolation between adjacent characteristic points will be as for the single input system. That is, when all input variables are held constant at characteristic values, varying one input between characteristic values will cause the output to change as per equation 3.3. This suggests the fifth and last criterion:

5) The unique members of a set have a membership of 1.

The nature of the control surface of a well conditioned rule base can be easily deduced without calculation; rules represent points on the surface, and output set weights determine the shape of the control surface between those points.

3.2.4 Writing a Well Conditioned Rule Base

Any rule base that uses fuzzy partitions on its input universes (c.f. §2.2.6), which are constructed from normal triangular or trapezoidal sets, will conform to criteria 3, 4 and 5.

It remains only to write a complete set of rules (criteria 1 and 2), and the rule base will be well conditioned.

Further, such a rule base will be strict complete (c.f. §2.6.2) with respect to all points in its input space.

3.3 Writing a Rule Base for a Fuzzy Controller

There are many introductions to fuzzy controller design technique.

The following section contains a general introduction, from the view point of working

with a well conditioned rule base.

3.3.1 Choice of control variables

First, pick the input and output variables for the controller. Some initial assumptions regarding controllability/observability may have to be made if a formal physical model has not been constructed.

3.3.2 Input Set Design

Cover the range of each input with overlapping triangular sets. The sets should form a fuzzy partition. The number of sets is determined by how nonlinear the control surface needs to be — two sets is sufficient for a linear control surface, five or seven sets are usually used as a starting point. An odd number of sets is usually chosen, with one set positioned at the set-point of the controller, and an equal number of sets positioned either side of it. Sets are usually concentrated around the set point to allow more precise control in that area.

3.3.3 Rule Design

Next, write the rules to determine the action of the controller. This procedure is usually done systematically: for every combination of input sets that may reasonably be expected to occur in practice, the necessary output value is estimated by the expert, and a rule is written expressing the relation. This requires that the designer has sufficient knowledge of the plant to be able to predict its reaction to a given control signal (the output) in a given situation (the state of the plant for the given input region). This does not imply that a complete and accurate mathematical model of the plant is needed. However, it does imply that some form of model must exist, even if it is only a linguistic one (that is, a description based on the designers empirical experience of the plant). Knowledge may be incorporated from many sources: from empirical models, from locally approximated mathematical models, from the observations of plant operators, and from trial and error methods.

3.3.4 Output Set Weighting

At this point every rule has its own unique output set. The weighting associated with the set can therefore be viewed as being associated with the rule, or with the characteristic point that the rule defines. Weightings are assigned to output sets to achieve the desired transition between characteristic points, the nature of the transition being determined by the ratio of the weights associated with adjacent characteristic points. A ratio of 1 results in a linear transition. A ratio of 3 produces a strong bias towards the more heavily weighted output. During the initial design phase it is usual to select identical weights for all output sets. This results in a linearly interpolated control surface. Set weightings are then adjusted during the tuning phase

of design.

3.3.5 Performance Tuning

After the controller is installed (in either a real or simulated situation) performance measurements can be made. Tuning is an iterative process. The performance of the controller is measured on a region by region basis, the regions being defined by the input coverage of individual rules. The output of the rules is then adjusted with the aim of improving performance at characteristic points. The performance of the controller in regions that are covered by more than one rule must be adjusted by changing the weights associated with the output sets of the rules. Higher weights will bias the output towards the associated output.

If the desired performance cannot be obtained by adjusting the output sets, the number of input sets in the region of inadequate performance should be increased. This may be achieved by concentrating existing input sets, or by adding more input sets. Design then proceeds from the beginning, with new rules being written for the newly introduced combinations of sets.

If satisfactory control still cannot be achieved, further investigation of the physical characteristics of the plant—with particular reference to any initial assumptions of controllability/observability—will be necessary.

3.3.6 Rule Base Tuning

Next, steps may be taken to simplify the rule base to make it easier to read and faster executing. This is generally referred to as *rule reduction.*

There are two basic methods of rule reduction.

1) Where output behaviour does not vary between adjacent sets, the sets may be collected together under one set.

For example, for two adjacent sets $\mathbf{a}_1$ and $\mathbf{a}_2$, the rules involving them and other set combinations may all result in the same output:

if x is $\mathbf{a}_1$ and y is $\mathbf{b}_j$ then z is $\mathbf{c}_j$
if x is $\mathbf{a}_2$ and y is $\mathbf{b}_j$ then z is $\mathbf{d}_j$

If $\mathbf{c}_j = \mathbf{d}_j$ for all j, then $\mathbf{a}_1$ and $\mathbf{a}_2$ may be replaced by their union. The membership function of the new set should be constructed to conform with the criteria that define a well conditioned rule base. It will be trapezoidal if $\mathbf{a}_1$ and $\mathbf{a}_2$ are triangular.

2) Where output remains constant for all rules that include one particular set, the rules may be represented by one rule.

For example, for some rules involving combinations of the set $\mathbf{a}_1$ and the sets $\mathbf{b}_j$:

if x is $\mathbf{a}_i$ and y is $\mathbf{b}_j$ then z is $\mathbf{k}_j$

If $\mathbf{k}_j$ does not vary with j (that is, it has the same centre and weighting for all j), then the rules may be represented with the single rule:

if x is $\mathbf{a}_i$ then z is $\mathbf{k}$

Finally, a check should be made for redundant rules (rules that contain combinations of inputs that do not occur in practice), and any found should be removed.

3.4 Traditional Design Methods and Fuzzy Logic

It is possible to employ both classical and modern design methods with fuzzy logic. This is accomplished by first designing a traditional controller and then translating it into an equivalent fuzzy logic controller. The translation may be of use in illuminating the workings of the traditional controller and assisting in the tuning process.

Table based controllers (such as fuzzy logic controllers) differ from many traditional controllers in that they have no internal state. In order to translate a controller that has some internal state into a fuzzy controller, the internal state must be factored out. The part of the controller that is translated into its fuzzy equivalent is then the function that transforms the internal state of the controller and the state of the plant into the output signal.

3.5 Global Optimisation Methods

Chapter 4 discusses methods of designing a controller by global optimisation. Although this method does offer the possibility of automatic controller design, it cannot be used if a mathematical description of the system is not available.

3.6 Combining Methods

Fuzzy logic can be used to define control surfaces in a piecewise way. Thus, all of the above design methods can be used in the creation of a controller. For instance, the control surface of a linear design that is known to have good performance in a certain region may be modelled with fuzzy logic. The areas away from the region, corresponding to extremes where the linear controller performs poorly, can then be adjusted with traditional fuzzy controller techniques. This approach can be used to make controllers that have been designed for linearised plants more robust.

3.7 Stability

There have been several papers published that describe methods of determining the stability of fuzzy controllers. However, Jager (1995, p. 8-9) notes that in most cases the stability proofs are trivial due to the simplicity of the controller and process.

Fuzzy control methods have often been criticised on the basis that stability must be verified by extensive testing. However, the traditional design process of fuzzy controllers involves the iterative testing and modification of the controller, so the designer can be reasonably sure of the stability of a controller. This suffices for non-critical systems, but is obviously inadequate for systems where stability must be guaranteed. Mamdani (1993) said the following in response to criticism of the use of fuzzy controllers in critical systems:

> *Industry has never put forward a view that the mathematical stability analysis is a necessary and sufficient requirement for the acceptance of a well designed control system. That is merely the view that control system scientists wished to put forward, but has never gained currency outside academic circles. ...Prototype testing is more important than stability analysis; stability analysis by itself can never be considered a sufficient test. Moreover, in any practically useful methodology, a stability analysis step would need to be made a desirable but optional step; it cannot be a necessary step.*

Stability proofs for controllers designed using traditional, linear control methods often rely on system models that are only approximations of reality, and thus extensive testing is required for almost all controllers that will be used in critical systems. Further, fuzzy methods are often applied in situations where it is not possible to model the system (see Östergaard, 1990, for an example involving the control of a cement kiln), and no stability proof can be made without a system model.

Mamdani (1993) makes the following comment about the current state of research in fuzzy stability:

> *Stability is still an important issue but a different way has to be found to study it. In the final analysis all one may be able to do is to build prototypes for the purpose of approval certification. This is a well tried and tested approach used in industry and there is no reason why it may not suffice with control in fuzzy systems as well.*

3.8 Summary

The concept of a "well conditioned rule base" has been put forward with the aim of providing guidelines which will ensure that there is a strong connection between rule base and control surface geometry. It has been shown that, in a rule base adhering to the following 5 criteria:

1) Each input must be represented exactly once in every rule.

2) The propositions in every antecedent statement are connected with "and".

3) Every set must have at least one unique member.

4) The slopes of the membership functions of intersecting sets are opposite.

5) The unique members of a set have a membership of 1.

...and using the method of inference defined in chapter 2, each rule represents a characteristic point on the control surface, and the weights associated with the output sets determine the nature of inter-rule interpolation.

It was noted that if a controller of the type developed in chapter 2 employs fuzzy partitions on its input universes, and has a full set of rules, it will conform to the above 5 criteria.

An introduction to the design of fuzzy controllers from the point of view of working with a well conditioned rule base has also been presented.

4. The Optimisation of a Fuzzy Controller

In section 3.3 the traditional method of fuzzy controller design was described. The system can be viewed as a hand optimisation scheme: the designer conducts an iterative search for the best control surface. With the aim of automating this process, several machine optimisation schemes specific to fuzzy controllers have been developed. They fall into two main categories:

1) On-line optimisers, which improve the performance of a fuzzy controller while the controller is operating. They seek a performance goal, and use performance statistics for feedback.

2) Off-line optimisers, which create an optimal fuzzy controller for a given system. They seek an optimal controller for a given system model and performance goal.

The first variety of optimiser is useful for finishing the traditional design process of a fuzzy controller (c.f. §4.3), and for maintaining control quality in a changing plant.

The traditional design process is iterative — the rule base is created using a pool of expert knowledge (and as such is an estimated solution), the controller is tested, and the rule base is revised. Testing and revision are repeated until the desired quality of control is achieved. On-line optimisers can automate this iteration process by tuning the first design as it runs. Also, they are inherently robust because they seek to optimise the performance characteristics of a controller over time, and so cope well with any changes in system parameters. However, on-line optimisers must be designed specifically for use with one type of control problem, because their behaviour depends on the characteristics of the plant that they are used with. Mamdani (1979), Lee (1989), Patrikar (1989), Lemke (1992) and Tanscheit (1992) describe different approaches to creating on-line fuzzy optimisers, variously referred to as "self organising process controllers", "fuzzy PID supervisors", "self tuning fuzzy rule based controllers" and so on.

This chapter is concerned with the second approach to the problem — off-line optimisation. The goal of off-line optimisation is the automated design of the whole controller. Off-line optimisers attempt to minimise a cost function for a given system. Usually, the cost function specifies the desired operation of the controller in terms of relevant control parameters, for example: rise time, settling time and tracking error. The value of such a cost function must be determined by simulation. Robustness can be built into the optimisation scheme by running the simulation with varying system parameters and summing the cost function for all the simulations.

On-line optimisers need only track the movement of a known optimum. The situation is different for off-line optimisation schemes, which must search for an unspecified global optimum in a much larger search space that potentially has many local optima. While on-line optimisers may make use of standard search techniques, off-line optimisers require the use of more powerful algorithms, such as simulated annealing (Ingber, 1992), dynamic hill climbing (de la Maza and Yuret, 1994), genetic algorithms (Goldberg, 1989a), and neural network techniques (Kosko, 1992).

4.1 Fuzzy logic and Genetic Algorithms

Most efforts to date have focused on the use of genetic algorithms, probably because it is easy to parameterise the many variables of fuzzy controllers with a genetic algorithm, and because the method is topical.

The papers surveyed by the author all suffer, to differing extents, from the same fault: they describe methods of optimising the data structure that they use to represent their fuzzy controllers, rather than methods of optimising the control surface. The problem lies not so much in the approach — it is obvious that some data structure must be used to represent the control surface during optimisation — but in the implicit assumption that the data structure used efficiently represents the control surface (that is, none of the parameters in the data structure are redundant). An efficient parameterisation of the control surface is desirable because the size of the search space, and thus the difficulty of the search, increases exponentially with the number of parameters to be optimised.

While complex membership functions, complex rule forms, and the use of linguistic hedges may aid in the design stages of a controller (when it is being created from a pool of linguistically specified knowledge), they do not extend the range of possible control surface geometries that may be represented with simple sets and operators. Thus they should not be included in the search for an optimum control surface.

One of the earliest attempts at the optimisation of a fuzzy controller by genetic algorithm was made by Karr (1991). Karr used equilateral triangular sets, parameterised by their left and right bounds. This method of coding admits a group of controllers with incomplete regions into the search space (because the possibility exists that some region of the input space will not be covered by any set).

In section 3.2, a model of fuzzy logic was developed in which the coarse geometry of the control surface is specified by the rules, which define characteristic points on the surface, and in which the fine geometry (that is, the nature of interpolation between the characteristic points on the surface) is specified by the weights associated with the output sets. During the derivation of the interpolation equation for the model

it was noted that the nature of interpolation between characteristic points depends on two things: the ratio of overlapping membership functions, and the weights associated with the output sets. It was recommended that interpolation be adjusted solely by varying just one of these parameters, as it is easier to divine the nature of a control surface from a rule base when there are a minimum number of variables to consider (output set weight was the preferred variable; c.f. §3.2.2). Karr's parameterisation varies both output set weighting (area) and membership function shape, and thus redundantly parameterises control surface interpolation, adding extra dimensions to the search space for no gain in possible controller performance.

Lee and Takagi (1993) attempted to improve upon Karr's method by extending the search to include the minimisation of the number of rules. This approach extends the search space while introducing a variable that is not associated with controller performance. The suggested gain in utility was automatic rule reduction, but it is hard to justify the expense of increased search space when rule reduction can be accomplished easily by hand (c.f. §3.3.6). Additionally, Lee and Takagi encoded each input and output set with three parameters: left bound, centre, and right bound, adding redundancy to Karr's already redundant parameterisation, and not addressing the problem of rule base incompleteness.

Cooper and Vidal (1993) attempted to improve on Lee and Takagi's method by allowing the number of rules, and thus the size of the search space, to vary dynamically during the search. Although they achieved reasonable results with their test case, it is not clear how their novel variation impacts the general efficacy of the genetic algorithm. They encoded the membership functions in their controller redundantly, using two parameters — centre and half length — to specify equilateral triangles. They did not address the problem of incompleteness.

The authors of the final paper surveyed, Homaifar and McCormick (1995), also included incomplete rule bases in the search space. They used five output sets with fixed centres and weights, effectively fixing the resolution of their controller's output at one quarter of the size of the output space. While it is desirable to limit the precision of a search when using a genetic algorithm (c.f. §4.3), this seems excessive. Limiting the number of output sets does not reduce the number of parameters in the search space — one output parameter is still required for each rule output — so this restriction would not make sense if an optimisation scheme other than the genetic algorithm were to be used. The centres of the input sets were fixed as well, but the base lengths were included in the optimisation. Thus Homaifar and McCormick's method searches a limited set of characteristic points (because the core points of the sets are all fixed the number of possible locations for each characteristic point is fixed), and attempts to optimise the interpolation between them (by varying the shape of the input membership functions).

Before developing an efficient parameterisation for a fuzzy controller the basic theory of genetic algorithms will be outlined.

4.2 Genetic Algorithms

Holland (1975) is generally acknowledged as the beginning of genetic algorithm research, although several authors have previously published work in the area: Fraser (1962), Bremmermann (1965), and Reed (1967).

These authors made the observation that the process of natural selection has created many optimal structures in a chaotic environment. They sought to capture the qualities of this natural optimisation in an algorithm. Fogel (1994) outlines the qualities of natural selection that make evolutionary optimisation an attractive proposition:

> *Darwinian evolution is intrinsically a robust search and optimisation mechanism. Evolved biota demonstrate optimised complex behaviour at every level: the cell, the organ, the individual, and the population. The problems that biological species have solved are typified by chaos, chance, temporality, and nonlinear interactivity. These are also characteristics of problems that have proved to be especially intractable to classic methods of optimisation. The evolutionary process can be applied to problems where heuristic solutions are not available or generally lead to unsatisfactory results.*

Holland's success in capturing the most important aspect of natural selection in an optimisation algorithm lead many other researches to investigate the area. There now exist a large number of variations on the algorithm, all of which conform to the same general structure:

```
initialise population, G(0)
evaluate G(0)
t := 0
repeat
    t := t + 1
    generate G(t) using G(t-1)
    evaluate G(t)
until stopping condition
```

The *population* is composed of a group of *chromosomes*. At the beginning of the search, the population is initialised with randomly constructed chromosomes. Each chromosome in the population is then evaluated and assigned a measure of fitness. The next population is generated by constructing new chromosomes from the fittest chromosomes in the population. Each succeeding population (*generation*) is evaluated and used to generate another until some stopping condition is

encountered. The iteration is usually stopped when the average fitness in the population plateaus (*convergence*), or when a predefined level of fitness is achieved, or after a certain number of generations.

Although there are many variations on this algorithm, the most important aspects of the various methods are captured in the *simple genetic algorithm*, or SGA (Goldberg, 1989a):

4.2.1 The SGA Chromosome

A chromosome of length n is a vector of the form: $<g_1, g_2, \ldots, g_n>$, where each g_i is called a *gene*. The set of the possible values that a gene may take on is called the *alphabet* of the problem, and a specific member of the alphabet is called an *allele*. The SGA uses a binary alphabet for all problems.

4.2.2 The SGA Population

A population of size m is a vector of the form: $<c_1, c_2, \ldots, c_m>$, where each c_j is a chromosome of identical length. The optimal choice of population size depends on the computation methods in use (serial or parallel), the length of the chromosomes in the population, and to some extent on the variety of problem. Goldberg (1989b) discusses each of these variables in depth. Population size is generally chosen on a heuristic basis; a general guide is: chromosome lengths of below 20 genes require populations of about 50, lengths of 20-50 require populations from 50-100, and populations of 100-200 are used for larger chromosomes.

4.2.2 The SGA Evaluation Function

Each chromosome encodes a possible solution to the given problem. The *fitness function* decodes the chromosome into solution form and evaluates the fitness of the solution. Consider the problem of minimising the stress in the truss shown in figure 4.1.

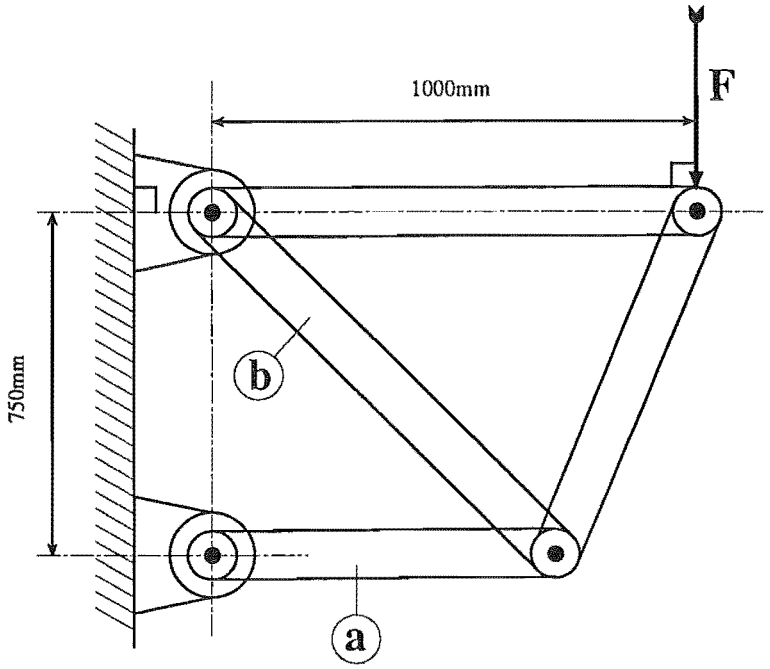


Figure 1: *Truss with applied force F.*

The problem is to find the lengths of the beams **a** and **b** that minimise the stress in the truss. The size of the chromosome required to encode the two lengths depends on the size of the search space and the precision of the search. For example, a search including all beam lengths from 0 to 1 m, with a resolution of 4 mm, would require that each length be represented to a precision of at least 8 bits.

If eight bits of precision were used to encode both **a** and **b**, then the total length of the chromosome required to encode a possible solution would be 16 bits. The first eight bits of the chromosome would represent one of the lengths, and the next eight bits would represent the other length.

The fitness function would then decode the beam lengths by extracting each eight bit number from the chromosome and scaling it into the range 0 to 1 m. The function could then determine the maximum stress in the truss for the encoded geometry, and rate the chromosomes fitness accordingly. A suitable measure of fitness might be the difference between the maximum allowable stress and the maximum stress in the truss.

4.2.3 The SGA Generation Function

Generation of a new chromosome from chromosomes in the current population is accomplished in three steps: *selection* of two parent chromosomes, reproduction by *crossover*, and finally *mutation*. New populations are built by the successive generation of new chromosomes. The transfer of the fittest chromosome from one generation to the next (called *elitist selection*), ensures that the best solution is always available for inspection and improves the rate of convergence slightly.

4.2.4 The SGA Selection Function

Roulette wheel selection is used to pick chromosomes from the population for reproduction. Firstly, the fitness values for the entire population are scaled into the range 0 to 1. The fitness values are then summed, and a random number between 0 and the total is generated. The fitness values are then summed again in order. When the sum exceeds the random number, the chromosome corresponding to the last fitness value to be added to the sum is selected. This scheme ensures that each chromosome has a probability of being selected proportional to its fitness. The term "roulette wheel" comes from an analogy with a biased roulette wheel; each chromosome being associated with a slot sized in proportion to the chromosomes fitness.

4.2.5 The SGA Crossover Function

After two chromosomes are selected, they are combined by crossover to form a new, chromosome. Crossover is the central operator of all genetic algorithms. It is derived from an analogy with sexual reproduction; two "parent" chromosomes are selected from the population with a probability proportional to their fitness; they "reproduce" to form a "child" chromosome; the child receives some of its genes from one parent and some of its genes from the other. It is the crossover function that determines which genes come from which parent.

The SGA uses single point crossover. In a population with chromosomes of length n, single point crossover is accomplished by generating a random integer, r, in the range 1 to n-1. The first parent selected contributes its genes $g_1, \ldots, g_r$ to the child, and the second parent contributes its genes $g_{r+1}, \ldots, g_n$. This process is depicted in figure 4.2.

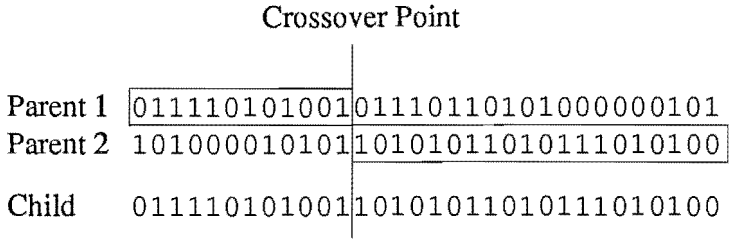


Figure 4.2: *Single point crossover.*

4.2.6 The SGA Mutation Function

Once a new chromosome has been created by crossover, it is mutated by randomly changing its genes. Only a small number of genes in the population should be mutated (typical mutation rates are about 0.005), or the algorithm will become inefficient. Mutation is applied to guard against the situation where one of a gene's alleles does not exist in any chromosome in the population (called *allele loss*). If this occurs the allele can only be returned to the population by mutation.

4.3 Some Properties of the SGA

Holland (1975) demonstrated that the process of selection and crossover exponentially increases the frequency with which "fit" genes occur in the population. This property has been termed "implicit parallelism," because the increase in frequency occurs for all the genes simultaneously. Because fit genes spread at an exponential rate throughout the population, initial convergence is very fast. Later in the search, when the population largely consists of chromosomes with similar fitnesses, the rate of convergence slows because there is little difference in selection probability from one chromosome to the next. This slow down later in the search has lead some authors, like Davis (1991) to suggest that genetic algorithms be used to find approximate solutions to problems, and that more traditional techniques, like hill climbing, be used to finish the search.

Search time is proportional to the size of the search space. For a population of chromosomes of length n, Goldberg (1989a) defined each possible solution as a hyperplane in n-dimensional binary space (assuming a binary alphabet). The number of possible hyperplanes is 2^n, so the number of solutions in the search increases exponentially with the length of the chromosome. Although the algorithm converges exponentially, the rate at which it converges seldom approaches order n, so it is desirable to keep chromosome length to a minimum. Any efficient parameterisation of a fuzzy controller must take this restriction into account.

4.4 An Efficient Parameterisation of a Fuzzy Controller

It was noted in section 2.6.2 that the use of fuzzy partitions to cover the input space of a controller guaranteed that the controller would be complete if a full rule base was written. A variety of fuzzy controller was developed which allowed control surface interpolation to be specified entirely by weights associated with the singleton output sets. It was further noted that if all the weights associated with the output sets were equal, interpolation would be linear.

The controller presented in chapter 2 may be used to efficiently parameterise a control surface. If triangular fuzzy sets are used to form the input partitions, the partitions may be completely specified with one parameter to locate the core of each of the central sets; the locations of the bounding sets are fixed by the bounds of the partition. As the output sets are singleton, only one parameter is needed to specify the output of each rule. If the search is to include control surfaces with nonlinear interpolation, an extra parameter must be added to specify the weight associated with each rule output. Thus, for a controller with $n_1, n_2, \ldots, n_i$ input sets on each of its respective i input universes,

the total number of parameters required to specify a complete controller is:

$$\sum_{i=1}^{x} n_i + \prod_{i=1}^{x} n_i$$

and the total number of parameters required to specify a complete controller with nonlinear interpolation is:

$$\sum_{i=1}^{x} n_i + 2\prod_{i=1}^{x} n_i$$

These parameterisations are adequate for use with algorithms in which the precision of the search varies dynamically. However, the precision to which a parameter is specified affects the size of the search space for a genetic algorithm, so it is desirable to make the most efficient possible use of precision when parametising for a genetic algorithm.

The fact that the sets on each input partition are ordered can be exploited to increase the precision of a genetic search. If the relative distance between set cores is parameterised, rather than the absolute location of each core, a much more efficient encoding can be achieved. Here, relative distance means that each parameter encodes the distance between two sets as a fraction of the sum of all the parameters on the same input universe. For example, the partition in figure 4.3 is encoded with four parameters, r_1, r_2, r_3, r_4, which specify the distances between the five cores on the universe. The location of the centre of the set at x_{core} can be found from:

$$x_{core} = (x_{right} - x_{left}) \frac{r_1}{r_1 + r_2 + r_3 + r_4}$$

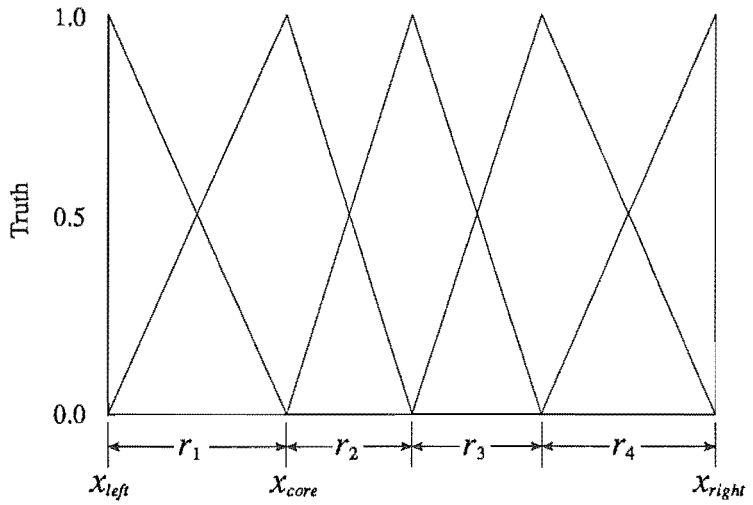


Figure 4.3: *Relative parameterisation of a fuzzy partition.*

This method of encoding a partition requires one more parameter than absolute

encoding, but it greatly increases the precision of a genetic search without increasing the size of the search space. If a the values of n relative parameters are assumed to be randomly distributed, then the average value will be half of the maximum value of the parameters. The input universe will be divided with an average resolution of

$$p = \frac{nr_{max}}{2x}$$

where r_{max} is the maximum value of one of the parameters and x is the size of the input universe. Compare this with the precision achieved with absolute parameters:

$$p = \frac{r_{max}}{x}$$

and it becomes apparent that the cost of the extra parameter is justifiable when more than two sets are used on an input universe.

4.5 Incorporating Expert Knowledge

The size of a search space can be further reduced by predefining rules and sets that must obviously be part of a valid solution. For example, the rules defining the behaviour of a controller at its set-point can often be predefined. In the case where a system is at its set-point, and is not moving away from its set-point, the output of the controller is obviously correct, and should not be changed. In rule form : if *input state* is **set-point** then *output change* = 0.

Fuzzy logic provides an easy way to predefine those parts of the control surface that are already known, so that only the unknown parts of the controller need be optimised.

4.6 Summary

In the introduction of this chapter the traditional fuzzy controller design process was compared to an iterative optimisation algorithm. This analogy was presented to highlight the goal of the various optimisation schemes that exist for fuzzy controllers: automated design of a controller.

The differences between the two main approaches to the optimisation of fuzzy controllers — on-line and off-line optimisation — were explained, and off-line optimisation was selected for further investigation because it offers the most promise for automated controller design.

Of the available off-line optimisation schemes, the genetic algorithm was selected for investigation because several authors have published some preliminary work in the area. A survey of the work revealed that all the previous methods used to parameterise

fuzzy controllers could be improved, and so an effort was made to develop a more efficient parameterisation scheme.

Further to the goal of reducing the search space, it was noted that known parts of the controller can be predefined (and thus removed from the search) because fuzzy logic allows the piecewise construction of a control surface.

Fuzzy logic provides a good interface to control surface optimisation algorithms: it can be used to parameterise an arbitrary surface with reasonable efficiency, and its easily readable format provides a way of investigating the solution that was found and tuning the solution for real world use.

As a final note to this chapter: A library of C++ code that implements a SGA has been written and appears in listings 6 through 11. A tutorial in its use and application to the field of fuzzy control appears in section 6.4.

5. The Design of a Fuzzy Inference Engine

The main features to be considered in the design of a fuzzy inference engine are the speed and compactness of the software. These criteria, and their relation to currently available solutions, are discussed in the first half of this chapter. The development of fuzzy control software with good speed and compactness is discussed in the remaining sections of this chapter.

5.1 Design concerns

There are several public domain software libraries available for the implementation of fuzzy controllers. However, all of these libraries are geared towards the study of fuzzy control, rather than its practical application; they are optimised for flexibility rather than speed and size. Practical solutions to fuzzy controller design are generally proprietary in nature, so there is some motivation to develop a fast, compact library of fuzzy controller software for the public domain.

In chapters 2 and 3 a controller design was developed that provides for flexible definition of a control surface and at the same time requires a minimum amount of calculation to perform inference. As the inference method of this type of controller is predefined, and the construction of rules and sets is restricted to a predefined format, it is possible to make several software optimisations specific to the design.

The *C* programming language was selected for use in the library, because it is compact and widely available in a standard form. A description of the development of software which implements the controller design specified in chapters 2 and 3 follows. The final design appears in listings 1 and 2 in appendix A.

5.2 Representing Fuzzy Sets in *C*

In 3.2.4 the format of fuzzy sets used in the well conditioned rule base (3.2) were restricted to fuzzy partitions constructed from normal triangular or trapezoidal sets. Noting that:

1) Triangular sets are a special case of trapezoidal sets; a triangular set is a trapezoidal set with a zero length top.

and:

2) A normal, trapezoidal set can be described by four points, one delimiting the left

bound of the set, one delimiting the left bound of the core of the set, one delimiting the right bound of the core of the set, and a final point delimiting the right bound of the set, because the bounds of the set must have a membership grade of 0 and the core of the set must have a membership grade of 1.

Both types of set can be described with the same data structure. In listing 2, a set is implicitly an array of integers of length 4. Sets are accessed via a pointer to the array. The type of the integer that is used to hold a set point is defined in listing 1:

```
typedef short set;
```

It is, by default, a short. In the C programming language a short can, at minimum, hold values in the range -32767 to +32767. When running on a small microprocessor, it may be desirable to change this definition to a smaller integer type.

Integers are used for all calculations in the controller software, so membership grades cannot be represented, as they usually are, in the interval [0, 1]. The constant MAX_GRADE is used to represent the maximum membership grade. It is defined in listing 1, and defaults to 255 (the maximum number representable with an unsigned 8 bit integer). The minimum grade of membership is implicitly assumed to be 0.

Making the further observation that:

3) In a fuzzy partition constructed from normal, trapezoidal sets, the bounds of each set correspond with the bounds of the cores of the sets to either side of it (or with one of the bounds of the partition).

...a fuzzy partition can be represented compactly by an array of integers that delimit the core points of the sets. Thus, the sets in figure 2.6 could be represented with the array:

```
set velocitySets = {0, 0, 10, 30, 30, 50, 80, 80};
```

Array members 0 through 3 define the set **slow**, members 2 through 5 define the set **mid**, and members 4 through 7 define the set **fast**.

5.3 Representing Fuzzy Rules in *C*

In 3.2 the conjunctions that may be used to combine the propositions in the rules of a well conditioned rule base were restricted to just one, "and," which was assumed to be modelled with fuzzy union. Thus the conjunctive operator may be implicitly assumed to be "and" and a rule may be defined with only an array of propositions and a consequent set. In listing 1 a proposition is defined:

```
struct proposition {
     set*       antecedent;
     short*     input;
};
```

```
typedef struct proposition proposition;
```

The member `antecedent` is assumed to point to an array of `set` 4 long. The member `input` is a pointer to the input that is to be associated with the antecedent set by the proposition. Thus the input match of the proposition can be determined by finding the height of the trapezoidal set `antecedent` at the point `*input`. The function `short Match(proposition* p)` in listing 2 returns the input match of a proposition that it is passed. It has been optimised for use with trapezoidal membership functions.

A proposition is declared:

```
proposition propl = {&velocitySets[0], &velocity};
```

; where velocity is a short that holds the input value.

In listing 1, a rule is defined:

```
struct rule {
     proposition*    propositions[TOTAL_INPUTS];
     short           consequent;
     short           weight;
};
typedef struct rule rule;
```

Here the consequent set is assumed to be a lumped mass model (c.f. §2.8.3). The array that holds the pointers to the propositions is of a fixed length, `TOTAL_INPUTS`, that is defined in listing 1. When a rule has fewer propositions than `TOTAL_INPUTS`, the remainder of array must be filled with null pointers.

An array of rules may be declared:

```
rule rules[] = {
     {
          {&propl, &prop2, &prop3}, /* propositions */
          127,                      /* consequent position */
          1                         /* weight */
     },
     /* ...more rules here */
};
```

Thus the entire data structure of a rule base may be declared in a compact, intuitive manner. Listing 3 gives an example of a fuzzy rule base that has been declared in the above manner. It describes the example controller from chapter 2, and was used to generate figure 2.2.

One final data structure must be defined to hold the other data structures. The structure `engine` is defined in listing 1:

```
struct engine {
     short      ruleCount;
     rule*      rules;
     short      output;
};
typedef struct engine engine;
```

The member `ruleCount` should hold the number of rules in the rule base, and the member `rules` should point to the array of rules. Inference is conducted by setting up the inputs and passing the engine structure to the procedure `RunEngine(engine* theEngine)`, which is defined in listing 1 and declared in listing 2. When `RunEngine` returns the `output` member of the engine structure will contain the results of the fuzzy inference.

5.4 Possible Optimisations

If maximum performance is required from the code, there are several optimisations that can be made.

1) If triangular sets are used to partition the input spaces, rather than trapezoidal sets, the partition may be defined using only one point per set, and the process of input matching is simplified slightly.

2) If the controller is restricted to linear interpolation, consequent set weights need not be stored, and the process of defuzzification is simplified slightly.

3) The function `short RunRule(rule* theRule)` (see listing 2) may be moved in-line, as may the function `short Match(proposition* p)`.

Listings 4 and 5 show the software with these modifications in place.

4) As a last resort, the software may be hand coded in assembler.

Some effort has been made to create a fast fuzzy inference library, so there is little to be gained from these optimisations.

5.5 Summary

A fast, compact library of software that implements the type of fuzzy controller that was defined in chapters 2 and 3 of this thesis has been developed, and a short introduction to its use has been given. Chapter 6 gives a practical example of the use of the software.

A short list of possible optimisations was presented, but the software, which appears in listings 1 and 2, is close to optimal, and little can gained from the optimisations.

6. Example: Velocity Control of a Variable Pitch Fan

This chapter presents a worked example demonstrating the use of the software package described in the previous sections of this thesis.

The package will be used to solve a simple control problem for a nonlinear plant: the speed control of a variable pitch propeller. A model of the propeller is developed in the first section and the control problem defined in the second. The third section discusses the use of the genetic algorithm library to evolve a fuzzy controller for the plant. The design solution is presented and discussed in the conclusion to this chapter.

6.1 A Simple Model of a Variable Pitch Propeller

The plant that will be used to demonstrate the software is a statically mounted, variable pitch propeller spinning in air. In 1991 the task of designing a test rig for frigate propellers was given to a third year mechanical engineering class at Canterbury University. The rig was to be used in the balancing of propellers and the testing of their control hydraulics. Students were asked to design a rig capable of spinning the large propellers up to a speed of approximately 190 rpm, and keeping the speed stable so that measurements could be made to detect any imbalances. The physical dimensions used in the following model are taken from one of the design solutions found by the class (Forster, 1991).

The following are the estimated specifications of the test rig:

Power required to drive a propeller with a pitch of 45° at 19.9 rad/s: 69.5kW
Rotational inertia: 5350 kgm^2
Maximum output of drive unit: 4000 Nm
Pitch variation: 5° to 45°

The power required to turn a propeller at a constant rate may be approximated:

$$P = kn^3 \tag{6.1}$$

where n is the rotational velocity of the propeller and k is some constant of proportionality.

Furthermore, if the air velocity in front of the propeller is small relative to its velocity behind the propeller, the power required to turn a variable pitch propeller may be approximated:

$$P = kn^3(\sin\alpha + \cos\alpha) \tag{6.2}$$

where α is the blade pitch.

From (6.2) a constant of proportionality (k) may be found that relates required power to rotational velocity for these dimensions:

$$69.5 \times 10^3 = k \times 19.9^3 \times (\sin 45° + \cos 45°)$$

$$\therefore k = 6.236$$

and the equation describing the relationship is:

$$P = 6.236n^3(\sin\alpha + \cos\alpha) \tag{6.3}$$

Dividing by n, the torque required to turn the prop at a constant speed is:

$$T_{prop} = 6.236n^2(\sin\alpha + \cos\alpha) \tag{6.4}$$

for positive n. The equation of motion for the plant is then:

$$T_{out} - T_{prop} = Jn' \tag{6.5}$$

where T_{out} is the torque applied to the propeller shaft, J is the rotational moment of inertia of the propeller, and n' is the acceleration of the propeller. By substituting (6.4) and the specified value of J into (6.5) an explicit equation for the acceleration of the propeller is obtained:

$$n' = \frac{T_{out} - 6.236n^2(\sin\alpha + \cos\alpha)}{5350} \tag{6.6}$$

Equation (6.6) is a very simple model of the plant in question — it is not accurate enough to be used in the design of a good controller for a real test rig. However, in this case on-line tuning of the controller is possible (that is, there is no danger of damage to the plant if the controller is tuned on-line), and the open nature of the fuzzy logic controller will aid in any such effort. Additionally, the solution to (6.6) can be found quickly, which was an important factor during the computationally expensive optimisation phase of the design.

The plant may be simulated by numerically integrating (6.6). The C code which was used to simulate the test rig appears in listing 17 of Appendix A. Equation (6.6) appears towards the end of the listing, in the function `Accn`:

```
double
Accn(double n)
{
      return (gTorque + ((n<0)?1:-1) * (n*n * 6.236 * (sin(gPitch) +
      cos(gPitch)))) / 5350.0;
}
```

As (6.4) is only valid for positive n, the sense of T_{prop} must be calculated explicitly.

This is done by multiplying by the factor `((n<0)?1:-1)`, which ensures that the sense of T_{prop} is always opposite to that of n.

6.2 The Control Problem

The task of the controller is to vary the output torque, T_{out}, in such a way that the rotational velocity of the propeller remains constant at 18.0 rad/s.

6.2.1 Input Sensor

From (6.6) it is apparent that the blade angle, α, is observable from T_{out}, n, and n'. Therefore, it should be possible to design a controller that requires only one sensor, measuring n, because T_{out} is determined by the controller and n' may be calculated from n.

When choosing the input precision of a controller, care must be taken to ensure that quantisation does not affect the quality of control. If a typical 8 bit precision is assumed, the entire range of propeller speeds cannot be represented with great accuracy (the range 0-25 rad/s can only be represented with a resolution of approximately 0.1 rad/s). For this reason a reduced control region spanning the range of speeds from 17.5 to 18.5 rad/s was chosen. The resolution of the sensor must approximately match or be greater than the resolution of the controller inputs for good control. The sensor chosen for the test rig was an 8 bit rotary encoder geared to run at 8 times the speed of the propeller shaft. The output of the encoder is fed into an accumulator that is sampled once every quarter second, giving a measure of the average speed during that time. This achieves a resolution of 0.006 rad/s in the control region. Code for simulating the sensor appears in listing 17 of Appendix C.

6.2.2 Sampling Period

It is usually desirable to have as short a sampling period as the available hardware will allow. In this case, however, the sampling period chosen (0.25 s) was determined not by the proposed hardware, but by the time required to simulate the plant. The optimisation of the controller (c.f. §6.4) required that the plant be simulated many times. As the time that each simulation took was directly proportional to the sampling period, a longer than usual sampling period was chosen to reduce the design time.

6.2.3 Output

In the interests of robustness, the output of the controller was taken to represent the change in the controller output (that is, the controller output was designated to be the integral of the rule base output), rather than an absolute value. Assuming an 8 bit precision, the range of output torques (0 Nm to 4000 Nm) can be represented with a

resolution of 15.7 Nm. As the output of the fuzzy rule base was taken to represent the change in output torque, the actual resolution of the output is half this (that is, 31.4 Nm), because the eight bit rule base output must be scaled into the range -4000 Nm to 4000 Nm. This proved to be an adequate precision in this case.

6.2.4 The Task

The task of the controller is, then, to drive the propeller into the control region (without feedback), and then hold the speed steady at 18.0 rad/s (with speed feedback) regardless of propeller pitch.

6.3 A Fuzzy Controller

This section outlines the design of the input side of the fuzzy controller and the rule base needed to drive the output side.

6.3.1 Input Set Design

The first step taken in designing the fuzzy controller was the selection of suitable input sets.

The inputs to the controller are torque, acceleration, and speed. The most important aspect of fuzzy set design is correct coverage at the set-point of the controller, in this case 18.0 rad/s and 0 acceleration. The torque required at the set-point can be calculated from (6.4), and varies widely with blade angle.

A minimum number of sets were used on each input universe. Six sets were chosen for the speed input; they appear in figure 6.1. One, labelled **off**, represents speeds below the control region. The set just above it, **vlow**, represents speeds in the low end of the control region; **right** represents the set-point, and **vhigh** represents speeds in the high end of the control region. The sets either side of the set-point, **low** and **high**, were included to allow better control near the set-point — they may be used to adjust the control strategy when the propeller nears the set-point. The separation of **low** and **high** from the set-point was determined by the sampling period; they are far enough apart that the propeller cannot overshoot the region they define in one sampling period. That is, at maximum acceleration (~0.75 rad/s^2) the speed of the propeller will change 0.1875 rad/s in one sampling period (0.25 s), and so the speed cannot pass through the central part of the control region (delimited by the core points of **low** and **high**) without at least one sample being taken either side of the set point.

Three sets were used to cover the acceleration input: **neg**, for negative accelerations, **zero** for no acceleration (the set-point), and **pos** for positive accelerations.

At the set-point, the required change in output torque is zero for all input torques. To

reflect this linear variation, only two sets were chosen to represent the range of possible input torques: **min**, to represent zero input torque, and **max** to represent maximum input torque (4000 Nm). See section 4.2.2 for notes on the nature of interpolation in fuzzy rule bases.

In accordance with the theory presented in chapter 3, the sets on each universe form a fuzzy partition.

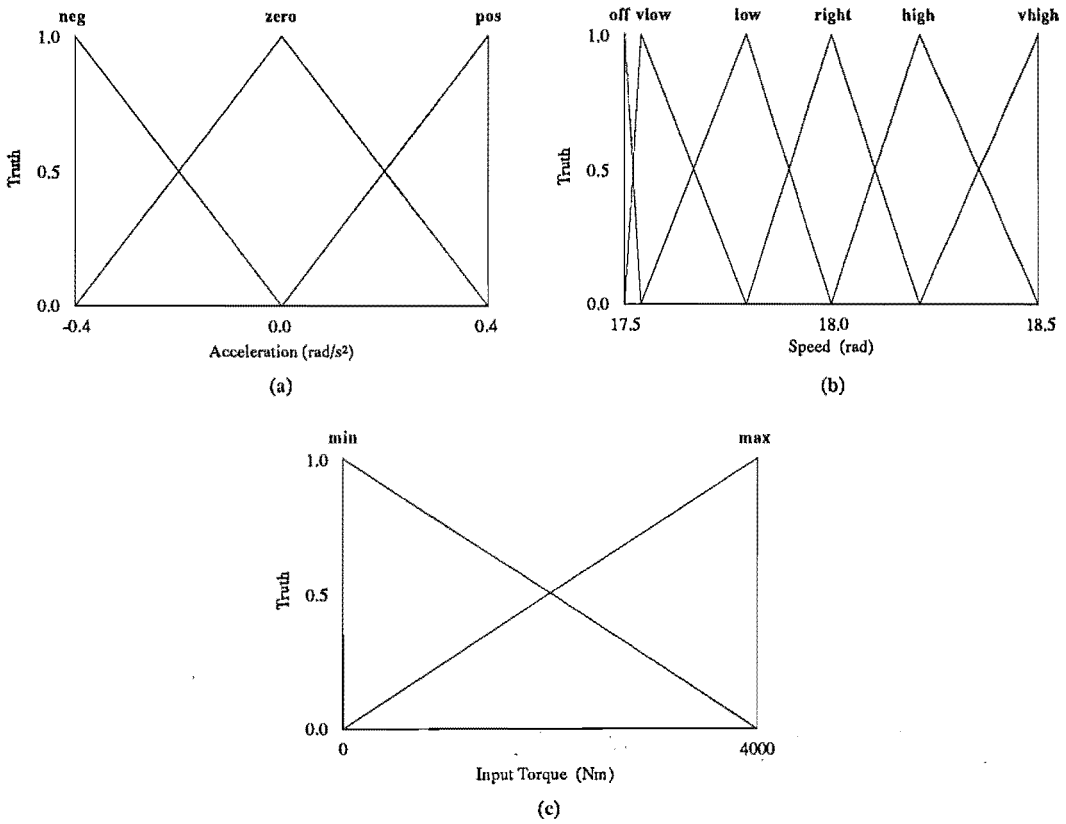


Figure 6.1: *The sets chosen to cover (a) the range of input velocities, (b) the possible range of acceleration, and (c) the range of the applied torque.*

Triangular sets were used (as opposed to trapezoidal sets) because rules that use them define characteristic points (rather than regions); a property that aids in the translation from rule base to control surface. This will be useful during the discussion of the final design. Also, the version of the fuzzy controller software that uses only triangular sets runs faster, a factor that was important given the limited computing resources available and the lengthy simulations that were required to evolve the controller.

The *C* code description of the fuzzy controller appears in Listing 5 of Appendix A. The input sets are represented using the compact triangular set data structure developed in section 5.2.

```
set speedSets[]  = {0, 0, 10, 77, 127, 177, 255, 255};
```

```
set accnSets[]   = {0, 0, 127, 255, 255};

set torqueSets[] = {0, 0, 255, 255};
```

This method of set representation needs some explanation. A normal, triangular set can be defined by three points: its left bound, its core, and its right bound. In a triangular fuzzy partition the core of a set coincides with the bounds of the sets to either side of it. This property can be exploited to reduce the storage requirements of the set data. All of the sets on a fuzzy partition can be defined using only the left and right bounds of the partition, and the core points of the sets. An individual set can then be referred to with a pointer to the number that defines its left bound. This is depicted in figure 6.2:

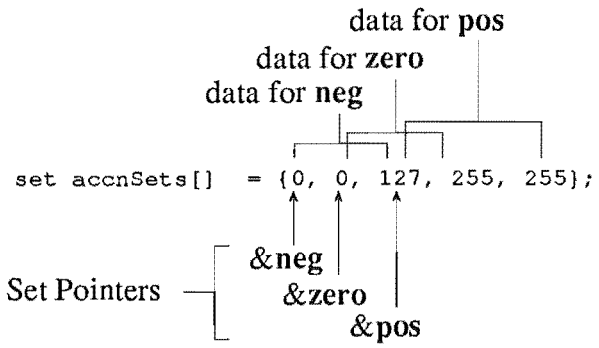


Figure 6.2: *The compact storage of triangular fuzzy sets.*

6.3.2 The Rule Base

From chapter 5, the definition if a proposition is:

```
struct rule {
     proposition*    propositions[TOTAL_INPUTS];
     short           consequent;
};
typedef struct rule rule;
```

where `propositions` is an array that holds all the propositions that make up the rule and `consequent` is the singleton output set of the rule.

The definition if a proposition is:

```
struct proposition {
     set*       antecedent;
     short*     input;
};
typedef struct proposition proposition;
```

where `antecedent` is a pointer to an input set and `input` is a pointer to an input. Assuming that the inputs to the controller are held in an array `inputs[]`, and that the propeller speed is held in `input[0]`, then the proposition "*speed* is **low**" may be written:

```
proposition sIsRight = {&speedSets[4], &inputs[0]};
```

and the rule "*speed* is **right** and acceleration is **zero** and *torque* is **min** then *output* is 0" may be written:

```
rule rule1 = {
     {
          &sIsRight, /* proposition 1: speed is right       */
          &aIsZero,  /* proposition 2: acceleration is zero */
          &tIsMin    /* proposition 3: torque is min        */
     },
     0               /* consequent: output is 0             */
};
```

A rule base is represented by an array of rules:

```
rule rules[] = {
     {{&sIsOff,   &aIsNeg,  &tIsMin}, 0},
     {{&sIsOff,   &aIsNeg,  &tIsMax}, 0},
     ...
     {{&sIsVHigh,  &aIsPos,  &tIsMin}, 0},
     {{&sIsVHigh,  &aIsPos,  &tIsMax}, 0}
};
```

A definition of a complete rule base using the above sets appears in Appendix A, listing 19.

Finally, to complete the controller, a data structure to hold all the above structures must be declared. The structure `engine` is defined:

```
struct engine {
     short      ruleCount;
     rule*      rules;
     short      output;
};
typedef struct engine engine;
```

Only the first two variables need be declared outside the engine, so the controller may be declared simply by:

```
engine controller = {
     TOTAL_RULES,
     rules
};
```

where `TOTAL_RULES` is the total number of rules in the array `rules[]`. In this case it is 36 — the number of rules needed for a complete rule base, as defined in 2.6.2.

The controller may now be compiled and used. However, as it is currently declared, the controller will not be of any practical use, because the output of all the rules defined in the listing is zero. The following section describes how the genetic algorithm software was used to find output values for the rules.

6.4 Optimisation of the Controller

Although an optimisation algorithm may be used to find the optimal position of each of the characteristic points defined in the rule base, and thus optimise the entire rule

base, in this case only the rule output values were optimised. This choice was made partly to increase the clarity of this example through concision, but the main motivation was a lack of computing power. The entire optimisation takes approximately 100 minutes on a personal computer, and so the time required to optimise the entire rule base (a procedure which would have added another 3 dimensions to the search) was prohibitive.

6.4.1 Incorporation of Expert Knowledge

A further way to cut down the time taken to optimise the controller is to predefine the outputs of some rules, and thus cut down the size of the search space.

Obviously, the controller must output a maximum torque signal when the speed of the propeller is below the control region. The output of the first six rules in the listing (of the form "if *speed* is **off** and...") should therefore be maximum (that is, 127). The two rules that define the behaviour of the controller at the set-point can also be predefined: it is obvious that if the propeller is spinning at the target speed, and is not accelerating, then the rule base should specify zero change in torque. Thus, eight of the most important rules can be completely specified before optimisation begins, leaving only 28 outputs to be found in the search. The eight rules with predefined outputs appear first in listing 19.

To run the genetic algorithm two small pieces of code must be written — one, the main loop, to iterate the algorithm, and another, the fitness function, to provide an interface between the genetic algorithm and the fuzzy logic controller.

6.4.2 The Main Loop

The iterative loop that was used to optimise the fuzzy controller appears in listing 15 of appendix A. It is, in outline:

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include "Dawkins.h"

#define    FILE_NAME  "Fan.pop"
#define    OUTPUT_PRECISION     6
#define    SIZE       100
#define    LENGTH     OUTPUT_PRECISION*TOTAL_RULES-PREDEFINED_RULES
#define    MUTATE     0.005
#define    GAP        0.05
#define    GREED      1.0

void
main ()
{
     Population*     pop;
     unsigned short  generations;

     if (InitPopulation(FILE_NAME, pop)) {
```

```
            cout << "How many generations? ";
            cin >> generations;

            for (int i=0; i<g; i++) {
                  pop->TheNextGeneration();
                  pop->Stats(cout);
            }

            fstream popFile(FILE_NAME, ios::in|ios::out|ios::trunc);
            popFile << *pop;
      }
      else {
            cout << "Not enough memory to store the population.";
            cout << endl;
      }
}
```

The listing is fairly self explanatory; the function `InitPopulation` attempts to restore a population from the file `FILE_NAME`, if it fails it creates a new population from scratch, using the constants defined at the beginning of the listing; the central loop then iterates the algorithm by calling `Population::TheNextGeneration` the requested number of times; after each generation some statistics are dumped to the standard output so that the progress of the algorithm may be monitored. An explanation of the use and design of genetic algorithms appears in chapter 4.

6.4.3 The Fitness Function

The genetic algorithm calls the fitness function for each new member (Chromosome) of the population that it generates to determine the fitness of the new member relative to the rest of the population. New members are created by "breeding" from only the fittest members of the population. In this a way fitter population is evolved.

The algorithm passes the fitness function a reference to a `Chromosome` object, and expects the fitness function to pass back a measure of the fitness of the `Chromosome`. A fitness function should be defined like this:

```
#include "Dawkins.h"

fitnessFunction      MyFF;
```

and declared like this:

```
double
MyFF(const Chromosome &it)
{
      double fitness = /* your code here */
      return fitness;
}
```

In general, the task of the fitness function is to translate the `Chromosome` into a useable data structure and evaluate it against the optimisation criteria. In the case of a controller, it is usual to simulate the operation of the controller and rank its performance using measures of control (for example, rise time and tracking error).

The fitness function used in the optimisation of the propeller controller was defined

as:

```
double
MyFF(Chromosome &it)
{
      double fitness;
      BuildEngine(it);
      fitness = -FanFitness(&controller, 5.0*DEG_TO_RAD);
      fitness -= FanFitness(&controller, 22.5*DEG_TO_RAD);
      fitness -= FanFitness(&controller, 45.0*DEG_TO_RAD);

      return fitness;
}
```

The function `BuildEngine` translates the `Chromosome` object (which is just a bit string) into a fuzzy controller data structure that can be passed to the simulation code. BuildEngine looks like this:

```
void
BuildEngine(Chromosome &it)
{
      static Chromosome bits(OUTPUT_PRECISION);

      it.SetMarker(0);

      for (int i=PREDEFINED_RULES; i<TOTAL_RULES; i++) {
            it >> bits;
            bits.UnGray();
            short raw = bits.CastToWord();
            short scaled =
                  raw*(double) MAX_GRADE/
                  (pow(2,OUTPUT_PRECISION)-1) - 127.5;
            controller.rules[i].consequent = scaled;
      }
}
```

It extracts bits from the `Chromosome` it is passed `OUTPUT_PRECISION` bits at a time (in this case, 6), scales the numbers into the range -127 to +127, and inserts them into the consequent parts of the undefined rules. The required length of every `Chromosome` in the population is the product of the number of undefined outputs in the controller and the number of bits of precision chosen for the controller outputs.

A low output precision is desirable from the point of view of search time, whereas a high precision is desirable from the point of view of potential controller performance. The six bit precision chosen was a compromise between the two. It proved to be adequate.

After `BuildEngine` returns to the fitness function the controller data structure is passed to the simulation software for evaluation. The function `FanFitness` takes a pointer to a controller and a propeller pitch angle as arguments, and returns a measure of fitness, namely the error integral over a simulation time of 80 seconds. The fitness function calls `FanFitness` three times, once with a propeller pitch of 5°, once with a pitch of 20°, and once with 45°. The fitness that it returns to the genetic algorithm is the sum of the three error integrals, with its sign reversed. It is necessary

to reverse the sign of the error integral because the genetic algorithm seeks to maximise fitness, and the desired outcome of the controller optimisation is a minimal error integral. `FanFitness` is a fairly simple function:

```
double
FanFitness(engine* controller, double p)
{
      double* simArray = FanSimulation(controller, p);
      double totalErr = 0.0;

      for (int i=0; i<STEPS; i++) {
            totalErr += fabs((18.0-simArray[i])*STEP_SIZE);
      }

      return totalErr;
}
```

It calls the function `FanSimulation` to obtain an array of speed values for the 80 second simulation run. It then integrates the speed error (using a simple, but fast, integration scheme) and returns the error integral to the fitness function.

The function `FanSimulation` demonstrates the use of the fuzzy logic controller in context:

```
#include "rk4.h"      // include the integration package.

double*
FanSimulation(engine* controller, double p)
{
      static      double simArray[STEPS];
      vector      initCond(1);
      short       lastTorque = 0;
      short       nextTorque = 0;
      short       lastRawSpeed = 0;

      gPitch = p;
      gTorque = 0.0;
      initCond[0] = 0;
      rk4 r(initCond, myDFunc, TZERO, STEP_SIZE);

      for (int i=0; i<STEPS; i++) {

/*
Fetch the sensor output:
 */
            short rawSpeed = Sensor(r(0));

/*
Now, scale the raw sensor reading into the range 0-MAX_GRADE.
5867 sensor clicks/quarter second corresponds to a propeller speed
of 18.0 rad/s, which is out set-point. Subtracting this from the raw
reading gives us a zero. In order to scale the zeroed reading into
eight bits (representing the range 17.5 to 18.5 rad/s), we multiply
by 4/5 and add 127.
 */
            short n = (rawSpeed - 5867)*4/5;
            n += MAX_GRADE/2;

            if (n < 0) n = 0;                    // clip n
            if (n > MAX_GRADE) n = MAX_GRADE;

/*
```

```
Now calculate the acceleration and scale it into 0-255, with 127
representing the set-point (that is, 0 acceleration). The maximum
acceleration is about 0.75 rad/s/s. After scaling by 0.5 and adding
127, the range of accelerations represented is about -0.75 to +0.75
rad/s/s.
 */
          short a = (rawSpeed - lastRawSpeed)/2;
          a += MAX_GRADE/2;

          if (a < 0) a = 0;                    // clip a
          if (a > MAX_GRADE) a = MAX_GRADE;

/*
Plug the inputs into the controller:
 */
          controller->inputs[0] = n;
          controller->inputs[1] = a;
          controller->inputs[2] = lastTorque;

/*
Find the output:
 */
          RunEngine(controller);

/*
Calculate the output torque:
 */
          nextTorque = lastTorque + controller->output * 2;


          if (nextTorque < 0) nextTorque = 0;  // clip nextTorque
          if (nextTorque > MAX_GRADE) nextTorque = MAX_GRADE;

          lastTorque = nextTorque;

/*
Calculate the real torque for simulation:
 */
          gTorque = nextTorque * 4000.0/MAX_GRADE;

/*
simulate the fan by integrating the system under this torque:
 */
          r.Step();

/*
save the result:
 */
          lastRawSpeed = rawSpeed;
          simArray[i] = r(0);
     }
     return simArray;
}
```

The object `r` of class `rk4` is a Runge-Kutta (order 4) integrator. The integrator's state vector (in this case the single state *n*) is accessed with `operator()`; *n* is referred to in the above code as `r(0)`. The function `myDFunc` that is passed to the constructor of `r` is used by `r` to obtain a state derivative; it returns a state derivative vector, in this case the single derivative *n'*, calculated from (6.6).

All of the necessary software for the genetic optimisation of a fuzzy logic controller has now been described. It remains only to execute the main loop and collect the

results when the algorithm converges.

6.4.4 The Optimisation Run

The software detailed in the previous sections was run on a Macintosh 660AV. The statistics produced at the end of each generation were directed into a file and monitored to detect convergence of the algorithm. Figure 6.3 shows the rise of fitness with successive generations. After about 50 generations it became apparent that the algorithm had converged. This took approximately 60 minutes.

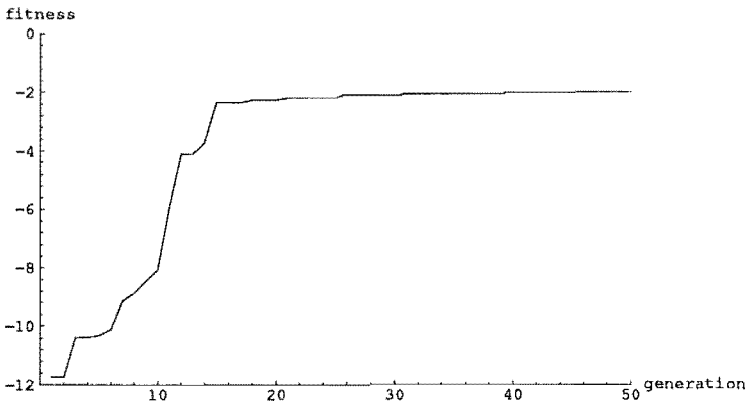


Figure 6.3: *Convergence of a genetic algorithm while optimising a fuzzy propeller controller.*

6.5 Results

Performance graphs for the fuzzy controller appear in figure 6.4. It is apparent from the graphs that the optimiser produced a reasonably efficient controller. The speed settles to 18.0±0.01 rad/s after ~1 s for all propeller pitches in the range 5° to 45°. Best case performance occurs at a pitch of 45°, with control to within ±0.005 rad/s.

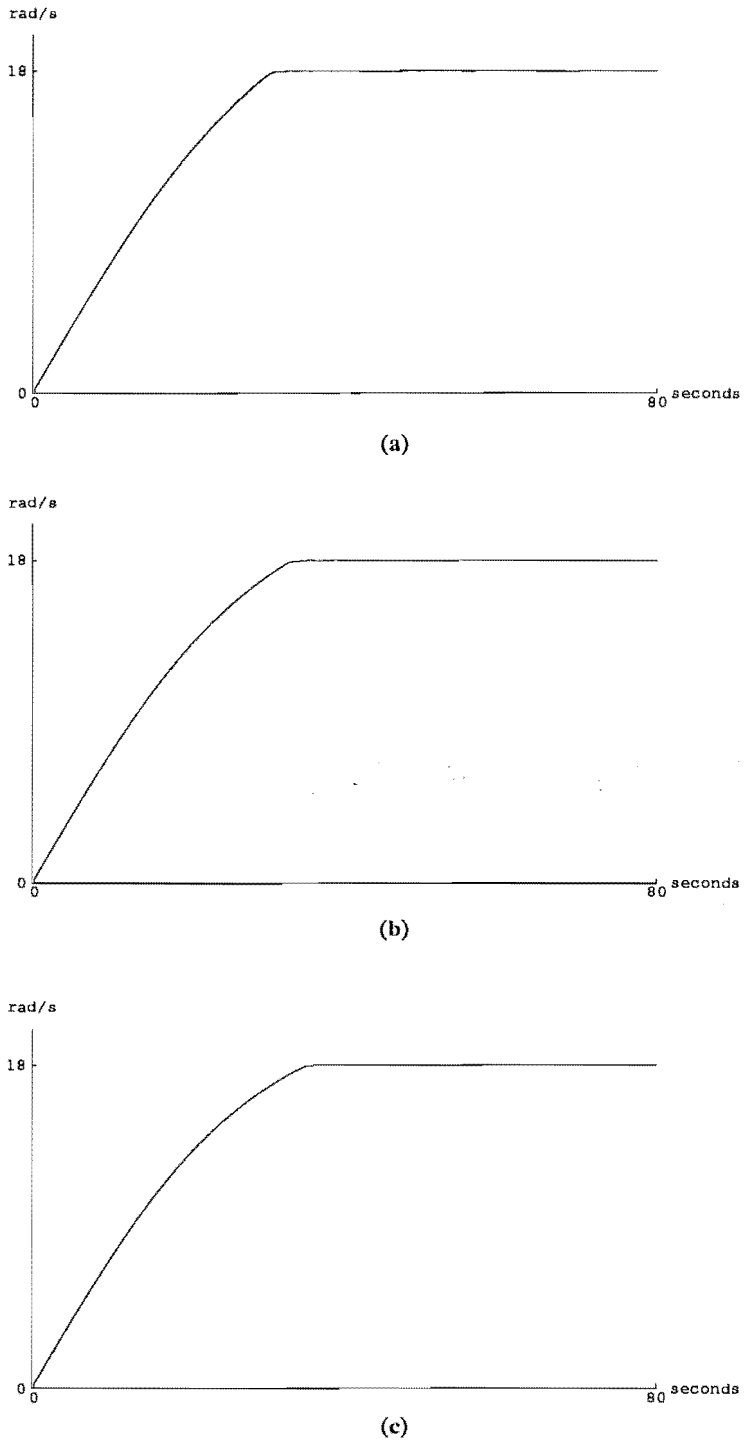


Figure 6.4: *Performance of the optimised controller for propeller pitches (a) 5°, (b) 20°, and (c) 45°.*

6.6 Examination of Results

The control strategy that the genetic algorithm found can now be read from the optimised rule base, which appears in table 6.1. The table lists each rule, its output, and the percentage use it had during the simulations.

#	Rule	Output	% Use
1.	if n is **off** and n' is **neg** and T is **min** then *output* is	127	0.000%
2.	if n is **off** and n' is **neg** and T is **max** then *output* is	127	0.087%
3.	if n is **off** and n' is **zero** and T is **min** then *output* is	127	0.011%
4.	if n is **off** and n' is **zero** and T is **max** then *output* is	127	10.937%
5.	if n is **off** and n' is **pos** and T is **min** then *output* is	127	0.087%
6.	if n is **off** and n' is **pos** and T is **max** then *output* is	127	10.937%
7.	if n is **right** and n' is **zero** and T is **min** then *output* is	0	19.898%
8.	if n is **right** and n' is **zero** and T is **max** then *output* is	0	19.800%
9.	if n is **vlow** and n' is **neg** and T is **min** then *output* is	46	0.000%
10.	if n is **vlow** and n' is **neg** and T is **max** then *output* is	26	0.000%
11.	if n is **vlow** and n' is **zero** and T is **min** then *output* is	99	0.087%
12.	if n is **vlow** and n' is **zero** and T is **max** then *output* is	14	0.293%
13.	if n is **vlow** and n' is **pos** and T is **min** then *output* is	-115	0.087%
14.	if n is **vlow** and n' is **pos** and T is **max** then *output* is	-38	0.293%
15.	if n is **low** and n' is **neg** and T is **min** then *output* is	62	1.227%
16.	if n is **low** and n' is **neg** and T is **max** then *output* is	127	1.227%
17.	if n is **low** and n' is **zero** and T is **min** then *output* is	10	8.244%
18.	if n is **low** and n' is **zero** and T is **max** then *output* is	10	8.407%
19.	if n is **low** and n' is **pos** and T is **min** then *output* is	-50	0.858%
20.	if n is **low** and n' is **pos** and T is **max** then *output* is	-123	1.021%
21.	if n is **right** and n' is **neg** and T is **min** then *output* is	58	2.411%
22.	if n is **right** and n' is **neg** and T is **max** then *output* is	123	2.313%
23.	if n is **right** and n' is **pos** and T is **min** then *output* is	-115	1.640%
24.	if n is **right** and n' is **pos** and T is **max** then *output* is	-107	1.615%
25.	if n is **high** and n' is **neg** and T is **min** then *output* is	-111	0.141%
26.	if n is **high** and n' is **neg** and T is **max** then *output* is	-111	0.065%
27.	if n is **high** and n' is **zero** and T is **min** then *output* is	-22	3.780%
28.	if n is **high** and n' is **zero** and T is **max** then *output* is	-99	3.693%
29.	if n is **high** and n' is **pos** and T is **min** then *output* is	10	0.402%
30.	if n is **high** and n' is **pos** and T is **max** then *output* is	-22	0.402%
31.	if n is **vhigh** and n' is **neg** and T is **min** then *output* is	-115	0.000%
32.	if n is **vhigh** and n' is **neg** and T is **max** then *output* is	18	0.000%
33.	if n is **vhigh** and n' is **zero** and T is **min** then *output* is	-54	0.000%
34.	if n is **vhigh** and n' is **zero** and T is **max** then *output* is	-18	0.000%
35.	if n is **vhigh** and n' is **pos** and T is **min** then *output* is	74	0.000%
36.	if n is **vhigh** and n' is **pos** and T is **max** then *output* is	18	0.000%

Table 6.1: *A rule base generated by genetic optimisation.*

6.6.1 Calculating Rule Usage

The percentage use was found by modifying the fuzzy inference code so that it counted the uses of each rule (see listing 5). The modifications can be automatically compiled in by defining the constant `HIT_LIST` in the header file `fastfuzz.h`.

This causes the structure `rule` to be modified to include the member `hits`, and the routine `RunEngine` to be modified to increment the `hits` member of a rule each time the rule returns a non-zero grade. To count the number of uses of a particular rule during a simulation, set its `hits` member to zero and then run the simulation. After the simulation completes `hits` will contain the number of uses.

6.6.2 Rule Reduction

From table 6.1, it is obvious that some rules have not been used during the simulations. Specifically, rules 1, 9, 10, and 31 through 36 were not used in the simulation. They had no effect on the fitness of any controller, and therefore, because there was no selection pressure associated with them, their outputs are purely random (except for rule 1, which was predefined in section 6.4.1). They can be dropped from the rule base with no effect on the quality of control.

Rules 31-36 concern the behaviour of the controller in the **vhigh** part of the control region. In fact, the set **vhigh** is involved in no other rules, so its declaration can be dropped from the rule base as well.

As per section 3.3.6, some of the rules that have identical outputs can be rolled together. Rules 2 through 5 all have an output of 127 regardless of the acceleration or torque sets that they involve. They can be combined into the single rule:

if *n* is **off** then *output* is 127

which would be written in *C* as:

```
rule r1 = {{&sIsOff, NULL, NULL}, 0};
```

The fuzzy inference code ignores NULL pointers, so only the first proposition in the above rule will be involved in determining the output grade.

Similarly, rules 7 and 8 (the set-point rules) can be combined into:

if *n* is **right** and *n'* is **zero** then *output* is 0

rules 17 and 18 can be combined into:

if *n* is **low** and *n'* is **zero** then *output* is 10

and rules 25 and 26 can be combined into:

if *n* is **high** and *n'* is **neg** then *output* is -111

The final rule base consists of 10 sets which are used in just 20 rules. A full C description of the controller appears in listing 20.

The control surface that the rules describe appears in figure 6.5. The plots were generated by the procedure `WalkEngine` (see listing 15), which creates a three

dimensional *Mathematica* plot showing the control surface of the strict complete parts of the rule base. Subcomplete and incomplete parts of the rule base correspond to regions where unused rules have been removed (c.f. §2.6.2), and thus are not involved in the production of output; only the parts of the control surface that were used in the simulation are shown in the plots.

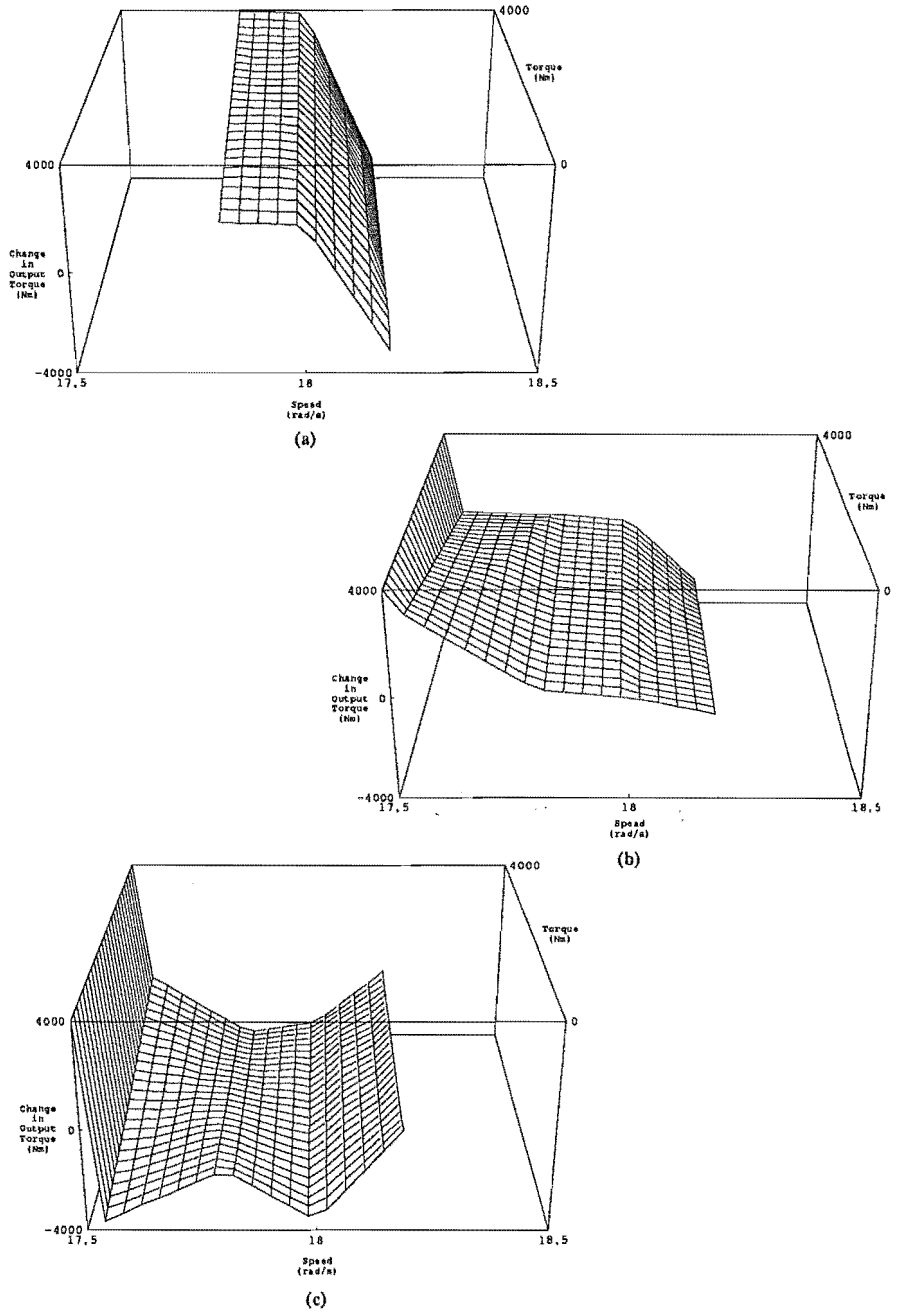


Figure 6.5: *The control surface described by the optimised fuzzy controller for: (a) constant acceleration at -0.75 rad/s², (b) zero acceleration, and (c) constant acceleration at +0.75 rad/s².*

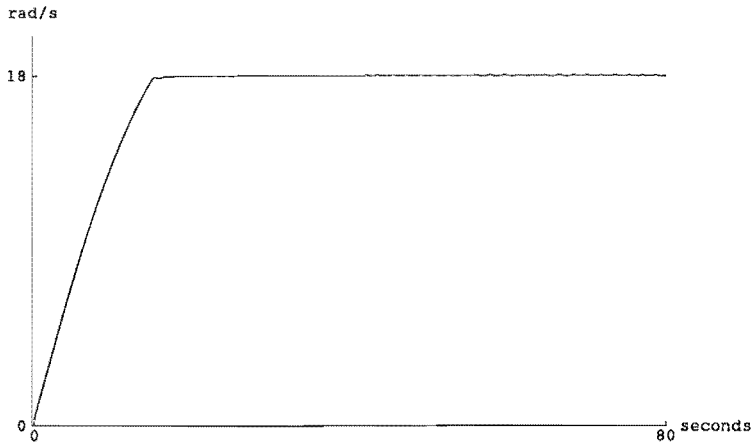
6.7 Discussion

The model that the fuzzy rule base was created to control is fairly simple, but does demonstrate the main physical characteristics of a variable pitch propeller. In reality a better model of the test rig would be required in the design of the controller, but without real test data it is difficult to create one.

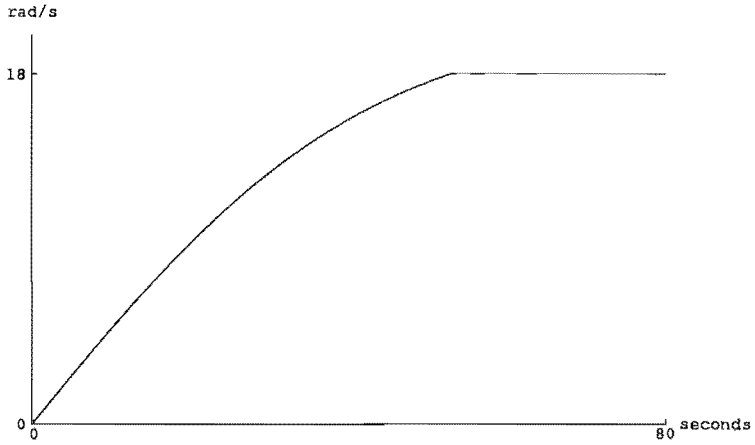
No attempt was made to design robustness into the controller. This could have been accomplished by writing a fitness function that better modelled real world conditions; for instance by simulating the operation of the test rig with varying propeller characteristics, but the increased computational requirements made this undesirable. In any case, this would not have been necessary because the final controller is quite robust, as is demonstrated by the performances shown in figure 6.6.

The graphs show the behaviour of the test rig when fitted with a considerably underweight propeller ($J = 2675$ kgm^2, a 50% reduction in rotational inertia) and a considerably overweight propeller ($J = 8025$ kgm^2, a 50% increase in rotational inertia). In both cases good control is maintained (± 0.01 rad/s).

Finally, the size of the code needed to effect the fuzzy controller and describe the rule base is 336 bytes and 418 bytes respectively. These figures are for a 32 bit machine — the expected size of the controller on an 8 bit microprocessor is under 500 bytes.



(a)



(b)

Figure 6.6: *Performance of the controller at (a) 5° propeller pitch and a 50% reduction in propeller rotational inertia, and (b) 45° propeller pitch and a 50% increase in propeller rotational inertia.*

7. Conclusion

It has been shown that a controller that uses the following methods for inference can be used to conduct efficient fuzzy inference from a rule base:

1) The min operator for conjunction in rules.

2) The product operator for implication.

3) Fuzzy singleton modelling of input sets.

4) Additive local aggregation.

5) Centroidal defuzzification.

6) Lumped mass modelled (singleton) output sets.

Taking the point of view that fuzzy logic is a language that may be used to define a control surface, it was determined to construct a set of criteria that lead to a close connection between rule base and control surface. This lead to the additional restrictions that:

7) Each input must be represented exactly once in every rule.

8) The propositions in every antecedent statement are connected with "and".

9) Every set must have at least one unique member.

10) The slopes of the membership functions of intersecting sets are opposite.

11) The unique members of a set have a membership of 1.

If these restrictions are made, then any rule base constructed will have a strong connection to the control surface; each rule will define a characteristic point on the surface, and the output set weightings will determine the nature of the interpolation between those points.

It was noted that 7 through 10 are true of a controller with fuzzy partitions on its input universes and a fully defined rule base.

The qualities of the controller defined above were exploited in the design of global optimisation software that made use of fast, compact fuzzy inference software and an efficient parameterisation of the control surface to be optimised.

A comprehensive software package for the optimisation and implementation of fuzzy controllers has been written (appendix A), and a tutorial describing its use was presented in chapter 6.

7.1 Suggested Further Work

Further investigation of the many possible alternatives to the genetic algorithm as an optimiser for fuzzy controllers is warranted, as the identification of the most efficient algorithm is desirable. Ingber (1992) has suggested that simulated annealing is superior to the genetic algorithm, and de la Maza (1994) claims that the dynamic hill climbing algorithm is superior to both. However, the types of problem that both authors investigate differ from those found in control, so it would be necessary to test all of these algorithms against a suite of "typical" control problems to establish their relative utility in optimising control surfaces.

The efficiency of the various possible methods of representing an arbitrary control surface (for example, fuzzy rule bases, Kohonen maps, neural nets, and more traditional table based interpolation methods) could be investigated in conjunction with the optimisation algorithms to find an efficient way of producing arbitrary optimal control surfaces.

If a general numerical method of measuring stability were to be developed, it could be used to prove the validity of any control surface produced by the discovered optimiser, without the need for extensive testing.

A combination of the above tools (an optimisation algorithm, an efficient control surface representation, and a general measure of stability) would provide a general method of generating optimal controllers for any given system model. Fuzzy logic would then provide a flexible interface to the method, allowing easy investigation and tuning of the resulting controllers, and the use of other design methods such as linguistic modelling in conjunction with the general optimisation method.

References

BELLMAN, R.E. and GIERTS, M. (1973) "On the analytical formalism of the theory of fuzzy sets," *Information Sciences*: vol 5, 149-156.

BREMERMANN, H.J. and others (1965) "Search by Evolution," *Biophysics and Cybernetic Systems*, Washington, DC, Spartan Books: 157-167.

COOPER, M.G and VIDAL, J.J (1993) "Genetic Design of Fuzzy Controllers," *The Proceedings of the Second International Conference on Fuzzy Theory and Technology*, Durham, NC.

DAVIS, L. (1991) *Handbook of Genetic Algorithms*, New York, Van Nostrand Reinhold: 26-27.

FORSTER, D. et al (1991) "ANZAC Frigate Propeller Test Rig for Mace Engineering," *Department of Mechanical Engineering Design 3 (Mech) Project Two*, Christchurch, University of Canterbury.

FRASER, A.S. (1962) "Simulation of Genetic Systems," *Journal of Theoretical Biology*, vol 2: 329-346.

GAINS, B.R. (1976) "Foundations of Fuzzy Reasoning," *International Journal of Man Machine Studies*, vol. 8: 623-668.

GOLDBERG, D.E. (1989a) *Genetic Algorithms in search, Optimisation, and Machine Learning*: Reading, MA, Addison-Wesley.

GOLDBERG, D.E. (1989b) "Sizing Populations for Serial and Parallel Genetic Algorithms," The *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, California, Morgan Kaufmann.

HOLLAND, J.H. (1975) *Adaption in Natural and Artificial Systems*, Ann Arbour, University of Michigan Press.

INGBER, L. (1992) "Genetic Algorithms and Very Fast Simulated Reannealing; a Comparison," *Mathematical and Computer Modelling*, November: 87-100.

JAGER, R. (1995) *Fuzzy Logic in Control*, Amsterdam, Technische Universiteit Delft: 34p. (Thesis: PhD: Electrical Engineering)

KARR, C. (1991) "Genetic Algorithms for Fuzzy Controllers," *AI Expert*, February: 26-33.

KOSKO, B. (1986) "Fuzzy knowledge combination," *International Journal of Intelligent*

Systems: vol. 1, 293-320.

KOSKO, B. (1992) *Neural Networks and Fuzzy Systems*, New Jersey, Prentice-Hall.

KRUSE, R. and others (1994) *Foundations of Fuzzy Systems*, West Sussex, England, John Wiley & Sons Ltd.

LEE, C.C and BERNJI, H.R. (1989) "An Intelligent Controller Based On Approximate Reasoning and Reinforcement Learning," *Procedures of the IEEE International Symposium on Intelligent Control*, Albany, New York: 200-205.

LEE, M.A. and TAKAGI, H. (1993) "Integrating Design Stages of Fuzzy Systems Using Genetic Algorithms," *Proceedings of the Second IEEE International Conference on Fuzzy Systems*. New York, NY, Institute of Electrical and Electronics Engineers: 612-617.

LEMKE, H.R. and KRIJGSMAN, A.J. (1992) "Design of Fuzzy PID Supervisors for Systems with Different Performance Requirements," *Mathematic of the Analysis and Design of Process Control*, Amsterdam, North-Holland Press: 593-602.

MAMDANI, E.H. (1975) "An Experiment in Linguistic Synthesis with a Fuzzy Controller," *International Journal of Man Machine Studies*, vol 8: 669-678.

MAMDANI, E.H. and PROCYK, T.J. (1979) "A Self-Organising linguistic process controller," *Automatica*, vol 15: 15-30.

DE LA MAZA, M. and YURET, D. (1994) "Dynamic Hill Climbing," *AI Expert*, March: 26-31.

MIZMOTO, M. (1989) "Improvement Methods of Fuzzy Controls," *Proceedings of the 3rd IFSA Congress*: 60-62.

ÖSTERGAARD, J.J (1990) "Fuzzy II — the new generation of high level kiln control," *Zement-Kalk-Gips*, vol 11.

PATRICAR, A. and PROVENCE, J. (1989) "Neural Network Implementation of Linguistic Controllers," *Procedings of the 12th IASTED International Symposium on Robotics and Manufacturing*, Santa Barbara, California.

PERRY, T.S. (1995) "Profile: Lotfi A. Zadeh," *IEEE Spectrum*, June: 32-35.

REED, J. and others (1967) "Simulation of Biological Evolution and Machine Learning," *Journal of Theoretical Biology*, vol 17: 319-342.

SUGENO, M. (1985) "An Introductory Survey of Fuzzy Control," *Information Sciences*, vol. 36: 59-83.

TANSCHEIT, R, and LEMBESSIS, E (1992) "On the Behaviour and Tuning of a Fuzzy

Rule-Based Self-Organising Controller," *Mathematics of the Analysis and Design of Process Control*, Amsterdam, North-Holland Press: 603-612.

ZADEH, L.A. (1965) "Fuzzy Sets," *Information and Control*, vol 8: 338-353.

ZADEH, L.A. (1973) "Outline of a New Approach to the analysis of complex systems and Decision Processes," *IEEE Transactions on Systems, Man, and Cybernetics SMC-3*: 28-44

ZADEH, L.A. (1992) "The calculus of fuzzy if/then rules," *AI Expert*: March, 23-27.

A. Software for the Implementation and Optimisation of Fuzzy Controllers

Listing 1: *FastFuzz.h; Public interfaces for a fast fuzzy controller.*

```
/*
Public interface for a small, fast, fuzzy inference engine. Version 1.
W. J. Hoyle, February 1996.

This version assumes trapezoidal (4 point) set definitions.
 */

#ifndef FASTFUZZ_HEADER
#define FASTFUZZ_HEADER

#ifdef __cplusplus
extern "C" {
#endif

#define    TOTAL_INPUTS    3     /* The number of inputs */
#define    MAX_GRADE       255   /* The maximum truth of a set */
#define    NO_OUTPUT       256   /* signals an inference error */
#undef     HIT_LIST              /* define this to generate a hit count */

typedef short set;               /* should be able to hold MAX_GRADE */

struct proposition {
     set*       antecedent;
     short*     input;
};
typedef struct proposition proposition;

struct rule {
     proposition*    propositions[TOTAL_INPUTS];
     short           consequent;
     short           weight;
     #ifdef HIT_LIST
     unsigned long   hits;
     #endif
};
typedef struct rule rule;

struct engine {
     short ruleCount;
     rule*      rules;
     short      output;
};
typedef struct engine engine;

void RunEngine(engine *theEngine);
short fuzzify(proposition* p);

#ifdef __cplusplus
}
#endif

#endif
```

Listing 2: *FastFuzz.c Routines for implementing a fast fuzzy controller.*

```
/*****************************************************************************
A simple, fast fuzzy logic toolkit. Version 1.
W. J. Hoyle, February 1996.

This version assumes trapezoidal (4 point) set definitions.

Minimal C code for a fuzzy logic controller.

See fastfuzz.h for an explanation of the data structures.

*****************************************************************************/

#include "fastfuzz.h"


short
RunRule(rule *theRule);

/*****************************************************************************/

/*
This routine runs the fuzzy inference engine through one inference.

The engine inputs must be set by the calling routine. The results of the
inference are returned in the engine output.
 */

void
RunEngine(engine *theEngine)
{
      short i;
      long massSum = 0;
      long momentSum = 0;
      short grade;
      rule*theRule;

/*
This loop runs the rules:
 */

      for (i=0; i<theEngine->ruleCount; i++) {
            theRule = &theEngine->rules[i];
            grade = RunRule(theRule);
            if (grade) {
                  massSum += theRule->weight * grade;
                  momentSum += theRule->consequent * theRule->weight * grade;
                  #ifdef HIT_LIST
                  theEngine->rules[i].hits++;
                  #endif
            }
      }

/*
This defuzzes the engine:
 */

      if (massSum) {
            theEngine->output = momentSum/massSum;
      }
      else {
            theEngine->output = NO_OUTPUT;
      }
}
```

```
/*************************************************************************/
/*
This routine finds the input match for the rule that it is passed.
 */

short
RunRule(rule *theRule)
{
     short          i;
     short          minGrade;
     long           grade;
     proposition**  propositions = theRule->propositions;

/*
This loop finds the minimum input match in the rule.
 */

     minGrade = MAX_GRADE;
     for (i=0; i<TOTAL_INPUTS && propositions[i]; i++) {
          grade = Match(propositions[i]);
          if (minGrade > grade) minGrade = grade;
     }

     return minGrade;
}

/
*************************************************************************/

short
Match(proposition* p)
{
     short set0; /* position of left set point (grade = 0)           */
     short set1; /* position of second set point (grade = MAX_GRADE) */
     short set2; /* position of third set point (grade = MAX_GRADE)  */
     short set3; /* position of right set point (grade = 0)          */

     short input;
     long  grade = MAX_GRADE;

     if (p) {

/*
Find the membership grade of the input in the antecedent set. Points 0
and 3 are assumed to have a grade of 0, points 1 and 2 are assumed to
have a grade of MAX_GRADE.
 */

          set0 = p->antecedent[0];
          set1 = p->antecedent[1];
          set2 = p->antecedent[2];
          set3 = p->antecedent[3];
          input = *(p->input);

/*
first check to see if the input is outside the set:
 */

          if (input >= set3 || input <= set0) {
               return (input == set1 || input == set2) ? MAX_GRADE : 0;
          }

/*
do an intersection with the left diagonal of the set if the input is left
of the second set point:
 */
```

```
            else {
                if (input < set1) {
                    grade *= (input - set0);
                    grade /= (set1 - set0);
                }

/*
otherwise do an intersection with the right diagonal of the set:
 */

                else {
                    if (input > set2) {
                        grade *= (input - set2);
                        grade = (long) MAX_GRADE - grade/(set3 - set2);
                    }
                }
            }
        }
        return grade;
}
```

Listing 3: *The example controller from chapter 2, declared in* C.

```
#include "controller.h"

short inputs[TOTAL_INPUTS];

set distSets[] = {
        0,
        0,
        54,
        118,
        118,
        182,
        255,
        255
};

set speedSets[] = {
        0,
        0,
        32,
        96,
        96,
        159,
        255,
        255
};

proposition sIsSlow   = {&speedSets[0],  &inputs[0]};
proposition sIsMed    = {&speedSets[2],  &inputs[0]};
proposition sIsFast   = {&speedSets[4],  &inputs[0]};

proposition dIsShort  = {&distSets[0],   &inputs[1]};
proposition dIsMid    = {&distSets[2],   &inputs[1]};
proposition dIsLong   = {&distSets[4],   &inputs[1]};

rule rules[] = {
        {{&sIsFast,  &dIsLong},  127, 1},
        {{&sIsFast,  &dIsMid},   255, 1},
        {{&sIsFast,  &dIsShort}, 255, 3},

        {{&sIsMed,   &dIsLong},  127, 1},
        {{&sIsMed,   &dIsMid},   127, 1},
        {{&sIsMed,   &dIsShort}, 255, 1},

        {{&sIsSlow,  &dIsLong},  13,  1},
```

```
     {{&sIsSlow,  &dIsMid},   13,  1},
     {{&sIsSlow,  &dIsShort}, 127, 1},
};

engine controller = {
     TOTAL_RULES,
     rules,
     inputs
};
```

Listing 4: *FastFuzz2.h; Public interfaces for optimised controller software.*

```
/*
Public interface for a small, fast, fuzzy inference engine. Version 2.
W. J. Hoyle, February 1996.

This version implements a minimal controller; Sets are assumed to be
triangular, and output set weights are not used, so control surface
interpolation will be linear.
 */

#ifndef FASTFUZZ_HEADER
#define FASTFUZZ_HEADER

#ifdef __cplusplus
extern "C" {
#endif

#define    TOTAL_INPUTS    3     /* The number of inputs */
#define    MAX_GRADE       255   /* The maximum truth of a set */
#define    NO_OUTPUT       256   /* signals an inference error */
#undef     HIT_LIST              /* define this to generate a hit count */

typedef short set;               /* should be able to hold MAX_GRADE */

struct proposition {
     set*       antecedent;
     short*     input;
};
typedef struct proposition proposition;

struct rule {
     proposition*    propositions[TOTAL_INPUTS];
     short           consequent;
     #ifdef HIT_LIST
     unsigned long   hits;
     #endif
};
typedef struct rule rule;

struct engine {
     short ruleCount;
     rule*      rules;
     short      output;
};
typedef struct engine engine;

void RunEngine(engine *theEngine);
short fuzzify(proposition* p);

#ifdef __cplusplus
}
#endif

#endif
```

Listing 5: *FastFuzz2.c; Optimised controller software.*

```
/*************************************************************************
A simple, fast fuzzy logic toolkit. Version 2
W. J. Hoyle, February 1996.

This version implements a minimal controller; Sets are assumed to be
triangular, and output set weights are not used, so control surface
interpolation will be linear.

Minimal C code for a fuzzy logic controller.

See fastfuzz.h for an explanation of the data structures.

*************************************************************************/

#include "fastfuzz.h"

/*************************************************************************/

/*
This routine runs the fuzzy inference engine through one inference.

The engine inputs must be set by the calling routine. The results of the
inference are returned in the engine output.
 */

void
RunEngine(engine *theEngine)
{
      short i;
      short j;
      long massSum = 0;
      long momentSum = 0;
      short grade;
      rule*theRule;

/*
This loop runs the rules:
 */

      for (i=0; i<theEngine->ruleCount; i++) {
            theRule = &theEngine->rules[i];
            minGrade = MAX_GRADE;
            for (j=0; i<TOTAL_INPUTS && theRule->propositions[j]; j++) {
                  grade = Match(theRule->propositions[j]);
                  if (minGrade > grade) minGrade = grade;
            }
            if (grade) {
                  massSum += grade;
                  momentSum += theRule->consequent * grade;
                  #ifdef HIT_LIST
                        theEngine->rules[i].hits++;
                  #endif
            }
      }

/*
This defuzzes the engine:
 */

      if (massSum) {
            theEngine->output = momentSum/massSum;
      }
      else {
            theEngine->output = NO_OUTPUT;
```

```
    }
}

/*************************************************************************/

short
Match(proposition* p)
{
    short set0; /* position of left set point (grade = 0)           */
    short set1; /* position of second set point (grade = MAX_GRADE) */
    short set2; /* position of right set point (grade = 0)          */
    short input;
    long  grade = MAX_GRADE;

    if (p) {

/*
Find the membership grade of the input in the antecedent set. Points 0
and 2 are assumed to have a grade of 0, point 1 is assumed to have a
grade of MAX_GRADE.
 */

        set0 = p->antecedent[0];
        set1 = p->antecedent[1];
        set2 = p->antecedent[2];
        input = *(p->input);

/*
first check to see if the input is outside the set:
 */

        if (input >= set2 || input <= set0) {
            return (input == set1) ? MAX_GRADE : 0;
        }

/*
do an intersection with the left diagonal of the set if the input is left
of the second set point:
 */

        else {
            if (input < set1) {
                grade *= (input - set0);
                grade /= (set1 - set0);
            }

/*
otherwise do an intersection with the right diagonal of the set:
 */

            else {
                grade *= (input - set1);
                grade = (long) MAX_GRADE - grade / (set2 - set1);
            }
        }
    }
    return grade;
}
```

Listing 6: *Chromo.h; Public interfaces for a chromosome class*

```
/*
chromo.h, Chromosome object for Dawkins v1.0:
Genetic algorithm software for optimising stuff.
W. J. Hoyle, Feb 1994.
 */
```

```
#include <fstream.h>
#include <limits.h>

/*************************************************************************
 * A Chromosome's bit string is stored as an array of "word".
 *
 * The following definitions are for portability. "word" should be
 * some native, unsigned int type that gives good speed on your machine.
 * WORD_BIT then needs to be defined as the number of bits in the "word"
 * type.
 *
 * The define "WORD" gives the array index of the word where the bit at
 * position "pos" is stored. Bit numbering in this implementation is
 * done according to the BigEndian convention (ie. MSB is at position 0).
 *
 * The define "BASE_MASK" gives a mask of the base-most n bits in a word;
 * by base-most I mean lowest in memory (that is, the MS n bits on a
 * BigEndian machine, or the LS n bits on a LittleEndian machine).
 */

typedef    unsigned int word;
#define    WORD_BIT   (sizeof (word)*CHAR_BIT)
#define    BIT(pos)   (~(~(word)0>>1) >> (pos)%WORD_BIT)

#ifdef BIG_ENDIAN
     #define WORD(pos)     ((pos)/WORD_BIT)
     #define BASE_MASK(n)  (~(word)0 << WORD_BIT-(n))
#else
     #define WORD(pos)     ((wordCount-1) - (pos)/WORD_BIT)
     #define BASE_MASK(n)  (~(word)0 >> WORD_BIT-(n))
#endif

/*************************************************************************
Chromosome class. This is the basic working object for managing bitstring
storage and manipulation.
 */

class Population;

class Chromosome {

          friend class Population;

          Chromosome() {status = allocationError; string = NULL;}

          void ChromosomeInit(int theLength);
          void Copy (const Chromosome &it);

          inline void Randomise();
          void Mutate(double frequency);
          // crossover:
          const Chromosome operator &(const Chromosome &mate) const;

          unsigned int length;
          unsigned int wordCount;
          word *string;
          unsigned int marker;
          int status;

     public:

          enum {
               good,
               allocationError
          };
          inline Chromosome(unsigned int len) {ChromosomeInit(len);}
          inline Chromosome(const Chromosome &it);
```

```
            inline Chromosome &operator = (const Chromosome &it);

            double fitness;

            virtual ~Chromosome() {delete[] string;}

            int Status() const {return status;}
            int Good() const {return (status == good);}
            inline void SetMarker(unsigned int point);
            inline word GetBit(unsigned int pos) const;
            inline void SetBit(word state, unsigned int pos);
            inline void ToggleBit(unsigned int pos);
            word CastToWord()
                  {return string[WORD(length)] >> WORD_BIT - length;}
            void UnGray();
            Chromosome &operator >>(Chromosome &dest);
            int operator ==(Chromosome& it);

            friend ostream &operator <<(ostream &out, const Chromosome &it);
            friend fstream &operator >>(fstream &in, Chromosome &it);
};


/*****************************************************************************
 * Sets the current bit marker for a chromosome. Legal values are 0 to
 * length-1.
 */

inline void Chromosome::SetMarker (unsigned int point)
{
      marker = (point >= length) ? length-1 : point;
}

/*****************************************************************************
 * To randomise a Chromosome's bits:
 */

inline void Chromosome::Randomise ()
{
      Mutate(0.5);
}

/*****************************************************************************
 * Assignment...
 */

inline Chromosome &Chromosome::operator = (const Chromosome &it)
{
      if (this != &it) {         // beware of *this = *this.
            if (it.length > length) {
                  delete string;   // reallocate if we don't have enough space.
                  string = new word[it.wordCount];
            }
            Copy (it);
      }
      return *this;
}

/*****************************************************************************
 * Initialisation...
 */

inline Chromosome::Chromosome(const Chromosome &it)
{
      string = new word[it.wordCount];
      Copy (it);
}
```

Listing 6: *Chromo.cpp; routines for a C++ chromosome class*

```
/*
chromo.cpp, Chromosome object methods for Dawkins v1.0:
Genetic algorithm software for optimising stuff.
W. J. Hoyle, Feb 1994.
 */

#include <float.h>
#include <iostream.h>
#include "chromo.h"
#include "utilities.h"

#define    CHROMOSOME_HDR  "Bit:"
#define    FITNESS_HDR          "Fit:"

/*****************************************************************************
 * These are the constructors for the Chromosome type. The null constructor
 * and initialiser function are provided to facillitate dynamic allocation.
 *
 * This is the basic working object for managing bitstring storage and
 * manipulation.
 *
 * The methods and operators are explained in the commented code.
 */

void Chromosome::ChromosomeInit (int theLength)
{
     if (theLength > 0) {
          length = theLength;
          wordCount = length/WORD_BIT;
          if (length%WORD_BIT != 0) {
               wordCount++;
          }

          string = new word[wordCount];
          if (string) {
               status = good;
               marker = 0;
               return;                                          // success
          }
     }
     status = allocationError;
}

/*****************************************************************************
 *  Bit fetch. Returns the bit at position pos.
 */

inline word Chromosome::GetBit (unsigned int pos) const
{
     int  theWord = WORD(pos);
     word theBit = BIT(pos);

     return string[theWord] & theBit;
}

/*****************************************************************************
 * Sets the bit at position pos to (theBit) ? 1:0.
 */

inline void Chromosome::SetBit (word state, unsigned int pos)
{
     int  theWord = WORD(pos);
     word theBit = BIT(pos);
```

```
    state ? (string[theWord] |= theBit) : (string[theWord] &= ~theBit);
}

/*****************************************************************************
 * To toggle a bit.
 */

inline void Chromosome::ToggleBit (unsigned int pos)
{
    int  theWord = WORD(pos);
    word theBit = BIT(pos);

    string[theWord] ^= theBit;
}

/*****************************************************************************
 * This operator reads bits from *this, starting from bit this->marker, and
 * writes them to the dest Chromosome, starting from bit 0, until either we
 * run out of bits or fill up dest. See the file test.cpp for an example of
 * its use.
 */

Chromosome &Chromosome::operator >>(Chromosome &dest)
{
    for (dest.marker=0; dest.marker<dest.length && marker<length;) {
        dest.SetBit(GetBit(marker++), dest.marker++);
    }

    return *this;
}

/*****************************************************************************
 * To mutate a Chromosome's bits with probability "frequency":
 */

void Chromosome::Mutate (double frequency)
{
    for (int i=0; i<length; i++) {
        if (Random0to1() < frequency) {
            ToggleBit(i);
        }
    }
}

/*****************************************************************************
 * The crossover operator:
 * This method fills a Chromosome with bits from two "parents" by crossing
 * their bit strings at some random point. It should only be used in
 * populations with constant length Chromosomes.
 */

const Chromosome Chromosome::operator &(const Chromosome &mate) const
{
    Chromosome kid(length);         // assume length == this->length

/*
generate a random cut point in the range 1 to length-1:
 */
    int cut = Random0to1()*(length-1) + 1;

/*
copy some bits from the calling parent:
 */
    for (int i=0; i<cut/WORD_BIT; i++) {
        kid.string[i] = string[i];
    }
```

```
/*
copy some bits from both parents to make up the word where the cut point
occurs:
 */
     word baseBits = BASE_MASK(cut%WORD_BIT);
     kid.string[i] = string[i]&baseBits | mate.string[i]&~baseBits;

/*
copy some bits from the mate:
 */
     while (++i < mate.wordCount) {
          kid.string[i] = mate.string[i];
     }

     return kid;
}

/*****************************************************************************
 * == operator (for debugging).
 */

int
Chromosome::operator ==(Chromosome& it)
{
     if (length != it.length) return 0;

     for (int i=0; i<length; i++) {
          if (GetBit(i) != it.GetBit(i)) return 0;
     }

     return 1;
}

/*****************************************************************************
 * This method dumps a text representation of a Chromosome to the
 * specified ostream.
 */

ostream &operator <<(ostream &out, const Chromosome &it)
{
     out << CHROMOSOME_HDR;
     for (int pos=0; pos<it.length; pos++) {
          out << (it.GetBit(pos) ? '1':'0');
     }
     out << "\n" << FITNESS_HDR;
     int Oldprecision = out.precision(DBL_DIG);
     out << it.fitness;
     out.precision(Oldprecision);

     return out;
}

/*****************************************************************************
 * Restore a Chromosome from a file:
 */

fstream &operator >>(fstream &in, Chromosome &it)
{
     char bit;

     in >> WaitFor(CHROMOSOME_HDR);        // wait for the header...
     if (in) {
          for (int pos=0; pos<it.length; pos++) {
               in >> bit;
               it.SetBit(bit-'0', pos);
          }
          in >> WaitFor(FITNESS_HDR) >> it.fitness;
```

```
    }

    return in;
}

/*************************************************************************
 * Copier... for assignment and initialisation.
 */

void Chromosome::Copy (const Chromosome &it)
{
    length = it.length;
    wordCount = it.wordCount;
    marker = it.marker;
    fitness = it.fitness;

    if (string) {
        status = good;
        for (int i=0; i<wordCount; i++) {
            string[i] = it.string[i];
        }
    }
    else {
        status = allocationError;
    }
}

/
*************************************************************************
 * This method changes a Chromosome from Gray coding to binary coding.
 * It is intended that the large Chromosomes in the population are read,
 * in sections, into smaller ones for decoding by your fitness
 * function. See the test.cpp file for examples. Gray coding can be
 * helpful in genetic algorithms because it spreads the selection pressure
 * on an allele more evenly over its component bits. For example: if
 * the optimum value for a gene (say 8 bits out of the Chromosome) is
 * 128, and our fitness function simply returns the error, there will be
 * a strong selection pressure on the MSB, and minimal pressure on the low
 * order bits -- but the final accuracy relies on the algorithm selecting
 * the low order bits correctly. Gray coding encourages more accurate
 * results, and prevents premature convergence.
 */

void Chromosome::UnGray()
{
    for (int i = 0 ; i < length-1; i++) {
        if (GetBit(i)) ToggleBit(i+1);
    }
}
```

Listing 7: *Pop.h; public interfaces for a C++ population class*

```
/*
pop.h, Population object for Dawkins v1.0:
Genetic algorithm software for optimising stuff.
W. J. Hoyle, Feb 1994.
 */

#include <fstream.h>

/*************************************************************************
 * Your fitness function must be of this type. It should evaluate the fitness
 * of the Chromosome passed to it and save the value in the Chromosome's
 * fitness variable (a double). The fitness variables can have values anywhere
 * in the range that double covers. This code works to maximise "fitness"
 * -- if it is more convenient to work to minimum "fitness" just change sign.
 */
```

```
typedef void fitnessFunction (Chromosome &);
typedef fitnessFunction *fitnessFunctionPtr;

/
*****************************************************************************
 * Population Type. This class holds the population of Chromosomes, and
 * provides functions for selecting and combining them to implement the GA.
 * See the file pop.cpp for a full explanation of the use of this type.
 */

class Population {
          unsigned int size;          //the number of Chromosomes
          unsigned int length;        //the length of Chromosomes
          int generation;             //the number of this generation
          double mutationRate;        //yep.
          unsigned int gap;           //the generation gap.
          double greed;               //simple Population fitness profiler
          double totalFitness;        //sum of scaled fitness values.
          int status;                 //allocation status.
          Chromosome* currentPop;     //pointer to the current array
          Chromosome* nextPop;        //where to build the next one.
          double* rouletteWheel;      //holds scaled fitness values.
          Chromosome** rank;          //holds a ranked list of pointers to
Chromosomes.

          void PopulationInit();
          Population &operator =(const Population&);      // can't assign
          Population(const Population&);                  // or copy
          const Chromosome &Select() const;
          void Sort();
          inline void Swap(int a, int b);
          void CrapSort();
          void Breed();
          void Mutate();
          void Evaluate();
          void Rank();

          fitnessFunctionPtr FitnessFunction;

     public:

          enum {
               good,
               allocationError,
               zeroFitness
          };

          Population(
               unsigned int size,
               unsigned int length,
               fitnessFunctionPtr theFF,
               double mutation = 0.001,
               double theGap = 0.02,
               double greed = 1.0
          );
          Population(fstream &in, fitnessFunctionPtr theFF);
          virtual ~Population();

          int Status() const {return status;}
          int Good() const {return (status == good);}
          void TheNextGeneration();
          void Stats(ostream &out) const;
          double AverageFitness() const;
          double MaxFitness() const {return BestChromosome().fitness;}
          Chromosome &BestChromosome() const {return *rank[0];}
```

```
        friend ostream &operator <<(ostream &out, const Population &it);
};
```

Listing 8: *Pop.cpp; a c++ population class*

```
/*
pop.cpp, Population object methods for Dawkins v1.0:
Genetic algorithm software for optimising stuff.
W. J. Hoyle, Feb 1994.
 */

#include <math.h>
#include <iostream.h>
#include "Dawkins.h"

#define   FILE_HDR        "Dawkins.cpp GA library, v1.0. Population
file:"
#define   SIZE_HDR        "Population size:"
#define   LENGTH_HDR      "Chromosome length:"
#define   MUTATION_HDR    "Probability of bit mutation:"
#define   GAP_HDR         "Generation gap:"
#define   GREED_HDR       "Greedy selection exponent:"
#define   GENERATION_HDR  "Generation:"
#define   FILE_END        "End of file."

#ifdef    HAVE_CONSTANT_FF
     #define   EVALUATE_FROM   gap
#else
     #define   EVALUATE_FROM   0
#endif


/*************************************************************************
The constructor for the Population class: This is the heart of the li-
brary. To work the algorithm,  first create a population object by pass-
ing the following parameters to its constructor:

int theSize: the number of Chromosome objects in the population. The
optimum depends mainly on Chromosome size -- a population of 100 is
plenty for Chromosomes of 30-40 bits in length, but after 40 bits optimal
population size grows rapidly (at least in theory), and time constraints
tend to govern the choice. See Goldberg, D.E. "Sizing Populations for
Serial and Parallel Genetic Algorithms", the Proceedings of the Third
International Conference on Genetic Algorithms, 1989.

int theLength: the number of bits in each Chromosome in the population.
Try not to go overboard on accuarcy, the algorithm will take forever to
reach a solution if the Chromosomes are too long. Consider using the GA
to find the correct global maximum/minimum and then conducting a finer
search in that area.

fitnessFunctionPtr theFF: a pointer to your fitness function. This is the
function that you provide to rank the Chromosomes in the population.
Your function will be passed a reference to a Chromosome, it should
store some index of performance in Chromosome.fitness. The performance
index can have a value anywhere in the range that type "double" covers.
This code in this library works to maximise "fitness" -- if it is more
convenient to work to minimum "fitness" just change sign.

double mutation: the likleyhood of a particular bit in the population
undergoing mutation. This is a secondary search operator; it is included
to guard against the loss of information from the bit pool during the
later stages of the run, when the alleles assosiated with fitter
Chormosomes begin to dominate population. To put it another way, if
one particular bit (say position one) of every Chromosome in the
population holds an identical value (say 1) then the crossover operator
```

cannot reintroduce the lost information (ie no Chromosome can gain a 0 in bit one). The mutation operator has a default value of 0.001. Values above 0.05 tend to do more harm than good, by giving the algorithm too much of a random search characteristic.

double theGap: A coefficient for the generation gap. Legal values are 0-1. This specifies the fraction of the population that is copied to the next generation without alteration. Doing this ensures that the best Chromosomes are never lost. It also gives them a better chance of being chosen for crossover (that is, it implements a "greedy" selection scheme), because they will take part in subsequent selection rounds. The default value for this variable is 0.02. If this code has compiled with the switch HAVE_CONSTANT_FF #defined, it will not pass your fitness function the Chromosomes that have been copied directly, and will therefore run somewhat faster.

double greediness: exponent for a simple greedy selection scheme. This library uses dynamic fitness scaling over a scaling window of 1 -- after one generation of Chromosomes has been evaluated by your fitness function their fitness values are scaled into the range 0 to 1. This approach was taken because it allows your fitness function to return some arbitary index of performance (some schemes require the index of fitness to be positive and in some specified range), and because emperical evidence suggests that a scaling window of 1 generation gives better performance (see Greffenstette, J.J. "Optimisation of Control Parameters for Genetic Algorithms", IEEE Transactions on Systems, Man, and Cybernetics, 16, Number 1, 1986). Greedy selection involves giving fitter organisms a better chance of reproduction than their index of fitness would indicate -- with the objective of faster convergence. This can be useful, but if abused will lead to premature convergence (ie the loss of useful alleles from the population -- see the discussion on mutation, above).
This library implements a simple greedy selection scheme by raising the scaled fitness of every Chromosome to a power (the greed exponent). This depresses the fitness values of the lesser Chromosomes, but leaves the fittest (which always has a fitness of 1 after scaling) untouched. The default greediness of 1.0 has no effect (ie no greedy selection), 2.0 is mild, and 5.0 is heavy. Values from 0.0 to 1.0 have the opposite effect, (sort of egalitarian...) and may have some use in preventing premature convergence in cases where selection pressure is already profound.
*/

```
Population::Population(
     unsigned int theSize,
     unsigned int theLength,
     fitnessFunctionPtr theFF,
     double mutation,
     double theGap,
     double greediness
) :
     size(theSize),
     length(theLength),
     FitnessFunction(theFF),
     mutationRate(mutation),
     gap(theGap),
     greed(greediness),
     generation(0)
{
     PopulationInit();
     if (Good()) {
          gap = theGap*size;
          if (gap >= size) gap = size-1;
          for (int i=0; i<size; i++) {
               currentPop[i].Randomise();
          }
          Evaluate();          // what have we got?
          Rank();              // sort them out.
```

```
    }
}

/*****************************************************************************
 * Construct a population from a file:
 */

Population::Population(fstream &in, fitnessFunctionPtr theFF) :
    FitnessFunction(theFF)
{
    in.seekp(0);
    in >> WaitFor(FILE_HDR);
    if (in) {
        in >> WaitFor(SIZE_HDR) >> size;
        if (in) {
            in >> WaitFor(LENGTH_HDR) >> length;
            if (in) {
                in >> WaitFor(MUTATION_HDR) >> mutationRate;
                if (in) {
                    in >> WaitFor(GAP_HDR) >> gap;
                    if (in) {
                        in >> WaitFor(GREED_HDR) >> greed;
                        if (in) {
                            in >> WaitFor(GENERATION_HDR) >>
                            generation;
                            if (in) {
                                PopulationInit();
                                if (Good()) {
                                    for (int i=0; i<size
                                    &&(in); i++) {
                                        in >> currentPop[i];
                                    }
                                    in >> WaitFor(FILE_END);
                                    if (in) {
                                        Rank();
                                        return;
                                        // success
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    status = allocationError;
}

/*****************************************************************************
 * Memory allocation for the Population class:
 */

void Population::PopulationInit()
{
    status = good;          // optimism...
    nextPop = NULL;         // in case we fail, and have to call the dtor.
    rouletteWheel = NULL;// ditto.
    rank = NULL;            // double ditto.

    currentPop = new Chromosome[size];
    if (currentPop) {
        nextPop = new Chromosome[size];
        if (nextPop) {
            for (int i=0; i<size; i++) {
                currentPop[i].ChromosomeInit(length);
                if (!currentPop[i].Good()) {
```

```
                        status = allocationError;
                        break;
                    }
                    nextPop[i].ChromosomeInit(length);
                    if (!nextPop[i].Good()) {
                        status = allocationError;
                        break;
                    }
                }
                if (Good()) {
                    rouletteWheel = new double[size];
                    if (rouletteWheel) {
                        rank = new Chromosome*[size];
                        if (rank) {
                            return;
    // success at last...
                        }
                    }
                }
            }
        }
        status = allocationError;
}

/*****************************************************************************
 * Memory deallocation for the Population class:
 */

Population::~Population()
{
    delete[] currentPop;
    delete[] nextPop;
    delete[] rouletteWheel;
    delete[] rank;
}

/*****************************************************************************
 * This is where everything comes together... just allocate your population
 * object, and apply this method to it until the answer looks good enough.
 */

void Population::TheNextGeneration()
{
    Breed();
    Mutate();
    Evaluate();
    Rank();
}

/*****************************************************************************
 * This function calls your fitness function for each Chromosome in the
 *   population.
 */

void Population::Evaluate()
{
/*
Call the fitness function for every Chromosome in the Population. Unless_
the switch HAVE_CONSTANT_FF has beed #defined.

If a Chromosome has been copied directly (by the Breed() method, in
accordance with the generation gap coefficient) - _and_ the fitness func-
tion remains constant from generation to generation - then its fitness
will already be correct (that is, it will have been evaluated correctly
during a previous generation). Define the switch HAVE_CONSTANT_FF (to
save time by avioding needless re-evaluation) if this is the case. If
this is the first generation we have to evaluate everything anyway.
```

```
 */

     for (int i=(generation==0) ? 0:EVALUATE_FROM; i<size; i++) {
          FitnessFunction(currentPop[i]);
          cout << "Fitness of Chromosome " << i << ": ";
          cout << currentPop[i].fitness << endl;
     }
}

/*************************************************************************
 * This routine sorts the population into order and calculates the stats
 * that are needed by the selection method: it scales the returned fitness
 * indexes into the range 0.0-1.0, raises them to the greed exponent to
 * implement the greedy selection scheme, and builds the roulette wheel for
 * use by the Select() method.
 */

void
Population::Rank()
{
/*
sort the Population into order:
 */
     Sort();

/*
if all of the Chromosomes have the same fitness, selection is pointless
-- so we set the population's status to zeroFitness, which provides the
user with some clue that convergence has occured, and disables the
Breed() operator (because we can't build the roulette wheel it needs
without dividing by zero...).
 */

   totalFitness = 0.0;
   double minFit = rank[size-1]->fitness;
   double range = rank[0]->fitness-minFit;

     if (range == 0) {
          status = zeroFitness;
     }

/*
otherwise we scale the fitness values to build the roulette wheel:
 */

     else {
          status = good;
          for (int i=0; i<size; i++) {
               double scaled =
                    pow((currentPop[i].fitness-minFit)/range, greed);
               totalFitness += rouletteWheel[i] = scaled;
          }
     }
}

/*************************************************************************
 * This routine builds new population of Chromosomes by selecting pairs of
 * parents from the current population, with probability proportional to
 * their fitness, and crossing over their bit strings to produce children.
 * This routine needs a valid roulette wheel to function, so don't call
 * it if the Population hasn't been evaluated (with the Evaluate() function).
 */

void Population::Breed()
{
     generation++;
```

```
/*
first, check to see that the population has some range of fitness:
 */
     if (status != zeroFitness) {

/*
transfer the best Chromosomes without alteration:
 */
          for (int i=0; i<gap; i++) {
               nextPop[i] = *rank[i];
          }

/*
make some kids:
 */
          while (i<size) {
               nextPop[i++] = Select() & Select(); // crossover.
          }

/*
swap the new population and current population:
 */
          Chromosome *temp = currentPop;
          currentPop = nextPop;
          nextPop = temp;
     }
}

/*************************************************************************
 * This routine applies the mutation operator to every Chromosome in the
 * population except those preserved by the generation gap constant.
 */

void Population::Mutate()
{
     for (int i=gap; i<size; i++) {
          currentPop[i].Mutate(mutationRate);
     }
}

/*************************************************************************
 * This selects a Chromosome from the population with a probability
 * proportional to its scaled fitness.
 */

const Chromosome &Population::Select() const
{
/*
spin the wheel...
 */
     double spot = Random0tol() * totalFitness;
     double currentSpot = 0.0;
     int i=0;

/*
find where it landed...
 */
     while ((currentSpot += rouletteWheel[i]) < spot) i++;

/*
return the lucky punter:
 */
     return currentPop[i];
}

/*************************************************************************
 * A simple reporting function:
```

```
 */

void Population::Stats(ostream &out) const
{
     out << "generation: " << generation << "\n";
     out << "average: " << AverageFitness() << "\n";
     out << "best: " << "\n" << BestChromosome() << endl;
}

/*************************************************************************
 * This function returns the population's average (unscaled) fitness:
 */

double Population::AverageFitness() const
{
     double total = 0;
     for (int i=0; i<size; i++) {
          total += currentPop[i].fitness;
     }
     return total/size;
}

/*************************************************************************
 * Dump a text representation of a Population to an ostream:
 */

ostream &operator <<(ostream &out, const Population &it)
{
     out << FILE_HDR << "\n\n";
     out << SIZE_HDR << it.size << "\n";
     out << LENGTH_HDR << it.length << "\n";
     out << MUTATION_HDR << it.mutationRate << "\n";
     out << GAP_HDR << it.gap << "\n";
     out << GREED_HDR << it.greed << "\n\n";
     out << GENERATION_HDR << it.generation << "\n\n";
     for (int i=0; i<it.size && out; i++) {
          out << *it.rank[i] << "\n";
     }
     out << "\n" << FILE_END << endl;

     return out;
}

/*************************************************************************
 * CrapSort:
 * This sorts the current population into descending order (of fitness) by
 * swapping pointers (held in the array "rank") to the individuals in the
 * Population.
 */

void Population::Sort()
{
     for (int i=0; i<size; i++) {
          rank[i] = currentPop+i;         // initialise pointers.
     }
     CrapSort();
}

inline void
Population::Swap(int a, int b)
{
     Chromosome* temp = rank[a];
     rank[a] = rank[b];
     rank[b] = temp;
}

void
```

```
Population::CrapSort()
{
     for (int i=0; i<size-1; i++) {
          for (int j=i+1; j<size; j++) {
               if (rank[i]->fitness < rank[j]->fitness) {
                    Swap(i,j);
               }
          }
     }
}
```

Listing 9: *utilities.h; public interfaces for some SGA utility functions*

```
/*
utilities.h, miscellaneous declarations and definitions for Dawkins v1.0:
Genetic algorithm software for optimising stuff.
W. J. Hoyle, Feb 1994.
 */

#include <iomanip.h>

/*****************************************************************************
 * This should return a double in the range 0.0 to 1.0.
 * Patch in your favourite random number generator here...
 *
 * I use the #define ULTRA to patch in a random number generator (see
 * below, and DawkinsInit::DawkinsInit(), in "utilities.cpp". Either #undef
 * ULTRA, in which case the standard C RNG will be used, or patch in your own
 * RNG.
 */

#ifdef ULTRA
     #include "Ultra.h"
     #define MY_RANDOM Ultra_duni()
#else
     #include <stdlib.h>
     #define MY_RANDOM ((double) rand()/RAND_MAX)
#endif

inline double Random0to1(){
     return MY_RANDOM;
}

/*****************************************************************************
 * The waitfor... operator. This waits for the specified string in the
 * specified fstream.
 */

IMANIP<char *> WaitFor(char *theString);
```

Listing 10: *utilities.cpp; some utilities functions for a SGA*

```
/*
utilities.cpp, Miscellaneous functions for Dawkins v1.0:
Genetic algotithm software for optimising stuff.
W. J. Hoyle, Feb 1994.
 */

#include <fstream.h>
#include <time.h>
#include "utilities.h"

/*****************************************************************************
 * This manipulator waits for the delimiting string "theString" in an
 * istream and then returns. The stream will be EOF if the operation fails.
 */
```

```
istream & wf(istream &in, char *theString)
{
     char *chPtr = theString;
     char test;

     while (in && *chPtr) {     // exit on bad stream
                                // or end of delimiting string
          in.get(test);
          if (test == *chPtr) {
               chPtr++;
          }
          else chPtr = theString;
     }

     return in;
}

IMANIP<char*> WaitFor(char *theString)
{
     return IMANIP<char*>(&wf, theString);
}

/*****************************************************************************
 * Initialiser object for the random number generator:
 */

class RandInit {
     public:
          RandInit();
};

RandInit InitialiserObject;     // initialiser object.

RandInit::RandInit ()
{
/*
seed the random number generator:
 */
#ifdef ULTRA
     Ultra_seed1 = clock();
     Ultra_seed2 = ~clock();
     Ultra_Init();
#else
     srand(clock());
#endif
}
```

Listing 11: *Dawkins.h; collected headers for the SGA library*

```
/*
Dawkins.h, public interfaces for for Dawkins v1.0:
Genetic algotithm software for optimising stuff.
W. J. Hoyle, Feb 1994.
 */


#include "chromo.h"             // Chromosome object

#include "utilities.h"          // some miscellaneous stuff

#include "pop.h"                // Population object
```

Listing 12: *vector.h; a simple vector class*

```
/*
```

```
A vector class:
 */

#ifndef VECTOR_HEADER
#define VECTOR_HEADER

#include <math.h>

class vector {
     unsigned int size;
     double *numbers;

     inline void copy(const vector &it);

public:
     vector() {numbers = 0;}
     vector(unsigned int size) : size(size) {numbers = new double[size];}
     inline vector(const vector &it);
     ~vector() {delete[] numbers;}

     inline vector *vectorInit(const unsigned int size);
     inline unsigned int Size() {return size;}
     inline void abs();
     inline void pow(double exponent);
     inline double min() const;
     inline double max() const;
     friend inline vector abs(const vector &it);
     friend inline vector pow(const vector &it, double exponent);
     inline vector &operator =(const vector &it);
     inline double &operator [](const unsigned int i) {return numbers[i];}
     inline double operator ()(const unsigned int i) const {return numbers[i];}
     inline vector &operator +=(const vector&it);
     inline vector &operator -=(const vector&it);
     inline vector &operator *=(const vector&it);
     inline vector &operator /=(const vector&it);

     inline vector operator +(const vector&it);
     inline vector operator -(const vector&it);
     inline vector operator *(const vector&it);
     inline vector operator /(const vector&it);

     inline vector &operator +=(const double scalar);
     inline vector &operator -=(const double scalar);
     inline vector &operator *=(const double scalar);
     inline vector &operator /=(const double scalar);

     inline vector operator +(const double scalar);
     inline vector operator -(const double scalar);
     inline vector operator *(const double scalar);
     inline vector operator /(const double scalar);

     friend inline vector operator +(const double scalar, const vector &it);
     friend inline vector operator -(const double scalar, const vector &it);
     friend inline vector operator *(const double scalar, const vector &it);
     friend inline vector operator /(const double scalar, const vector &it);
};


inline void vector::copy(const vector &it)
{
     size = it.size;
     for (unsigned int i=0; i<size; i++) {
          numbers[i] = it.numbers[i];
     }
}

inline vector::vector(const vector &it)
```

```
{
      numbers = new double[it.size];
      copy(it);
}

inline vector *vector::vectorInit(const unsigned int size)
{
      numbers = new double[size];
      vector::size = size;

      return (vector *) numbers;
}

inline vector &vector::operator =(const vector &it)
{
      if (it.size != size) {
            delete numbers;
            numbers = new double[it.size];
      }
      copy(it);
      return *this;
}

inline void vector::abs()
{
      for (unsigned int i=0; i<size; i++) {
            if (numbers[i] < 0) {
                  numbers[i] = 0-numbers[i];
            }
      }
}

inline vector abs(vector &it)
{
      vector v = it; v.abs(); return v;
}

inline void vector::pow(double exponent)
{
      for (unsigned int i=0; i<size; i++) {
            numbers[i] = ::pow(numbers[i], exponent);
      }
}

inline vector pow(vector &it, double exponent)
{
      vector v = it; v.pow(exponent); return v;
}

inline double vector::min() const
{
      double min = numbers[0];
      for (unsigned int i=1; i<size; i++) {
            if (numbers[i] < min) {
                  min = numbers[i];
            }
      }
      return min;
}

inline double vector::max() const
{
      double max = numbers[0];
      for (unsigned int i=1; i<size; i++) {
            if (numbers[i] > max) {
                  max = numbers[i];
            }
```

```
    }
    return max;
}

inline vector &vector::operator +=(const vector &it)
{
    for (unsigned int i=0; i<size; i++) {
        numbers[i] += it.numbers[i];
    }
    return *this;
}

inline vector &vector::operator -=(const vector &it)
{
    for (unsigned int i=0; i<size; i++) {
        numbers[i] -= it.numbers[i];
    }
    return *this;
}

inline vector &vector::operator *=(const vector &it)
{
    for (unsigned int i=0; i<size; i++) {
        numbers[i] *= it.numbers[i];
    }
    return *this;
}

inline vector &vector::operator /=(const vector &it)
{
    for (unsigned int i=0; i<size; i++) {
        numbers[i] /= it.numbers[i];
    }
    return *this;
}

inline vector vector::operator +(const vector &it)
{
    vector v = *this; v += it; return v;
}

inline vector vector::operator -(const vector &it)
{
    vector v = *this; v -= it; return v;
}

inline vector vector::operator *(const vector &it)
{
    vector v = *this; v *= it; return v;
}

inline vector vector::operator /(const vector &it)
{
    vector v = *this; v /= it; return v;
}

inline vector operator +(const double scalar, const vector &it)
{
    vector v = it;
    for (unsigned int i=0; i<v.size; i++) {
        v.numbers[i] = scalar/v.numbers[i];
    }
    return v;
}

inline vector operator -(const double scalar, const vector &it)
{
```

```
    vector v = it;
    for (unsigned int i=0; i<v.size; i++) {
        v.numbers[i] = scalar-v.numbers[i];
    }
    return v;
}

inline vector operator *(const double scalar, const vector &it)
{
    vector v = it;
    for (unsigned int i=0; i<v.size; i++) {
        v.numbers[i] = scalar*v.numbers[i];
    }
    return v;
}

inline vector operator /(const double scalar, const vector &it)
{
    vector v = it;
    for (unsigned int i=0; i<v.size; i++) {
        v.numbers[i] = scalar/v.numbers[i];
    }
    return v;
}

inline vector &vector::operator +=(const double scalar)
{
    for (unsigned int i=0; i<size; i++) {
        numbers[i] += scalar;
    }
    return *this;
}

inline vector &vector::operator -=(const double scalar)
{
    for (unsigned int i=0; i<size; i++) {
        numbers[i] -= scalar;
    }
    return *this;
}

inline vector &vector::operator *=(const double scalar)
{
    for (unsigned int i=0; i<size; i++) {
        numbers[i] *= scalar;
    }
    return *this;
}

inline vector &vector::operator /=(const double scalar)
{
    for (unsigned int i=0; i<size; i++) {
        numbers[i] /= scalar;
    }
    return *this;
}

inline vector vector::operator +(const double scalar)
{
    vector v = *this; v += scalar; return v;
}

inline vector vector::operator -(const double scalar)
{
    vector v = *this; v -= scalar; return v;
}
```

```
inline vector vector::operator *(const double scalar)
{
     vector v = *this; v *= scalar; return v;
}

inline vector vector::operator /(const double scalar)
{
     vector v = *this; v /= scalar; return v;
}

#endif
```

Listing 13: *sint.h; an abstract base class for a simple integrator*

```
/*
Simple integrator, abstract base class:
 */

#ifndef SINT_HEADER
#define SINT_HEADER

#include "vector.h"

typedef vector &dFunc(double t, const vector &state, vector& stateD);
typedef dFunc *dFuncPtr;

class SIntegrator {
protected:
     double h, hmax;            // step size
     double t;                  // time
     dFuncPtr dfn;              // pointer to your derivative function
     vector state;              // the state vector
     vector stateD;             // the state derivative

public:
     SIntegrator(vector &init, dFuncPtr dfn, double t, double h);
     virtual ~SIntegrator() {}

     virtual void Step() = 0;
     virtual void StepTo(const double endT);
     double Time() const {return t;}
     vector State() const {return state;}
     double operator ()(const unsigned int i) const {return state(i);}
};

#endif
```

Listing 13: *sint.cpp; utilities for the abstract base class of a simple integrator*

```
/*
Base class utilities for a simple integrator:
 */

#include "sint.h"

SIntegrator::SIntegrator(vector &init, dFuncPtr dfn, double t, double h):
     state(init),
     stateD(init.Size()),
     dfn(dfn),
     t(t),
     h(h),
     hmax(h)
{}

void SIntegrator::StepTo(const double endT)
{
```

```
        h = hmax;
        while (t<endT) {
                if (t+h > endT) {
                        h = endT-t;
                }
                Step();
        }
}
```

Listing 13: *rk4.h; public interfaces for a Runge-Kutta (order 4) integrator*

```
/*
Public interfaces for a Runge-Kutta (order 4) de solver:
 */

#include "sint.h"

class rk4 : public SIntegrator {
        vector k1, k2, k3, k4;                  // working variables

public:
        rk4(vector &init, dFuncPtr dfn, double t, double h);
        virtual ~rk4() {}

        virtual void Step();
};
```

Listing 14: *rk4.cpp; a Runge-Kutta (order 4) integrator*

```
/*
Runge-Kutta (order 4) de solver:
 */

#include "rk4.h"

rk4::rk4(vector &init, dFuncPtr dfn, double t, double h):
        SIntegrator(init, dfn, t, h),
        k1(init.Size()),
        k2(init.Size()),
        k3(init.Size()),
        k4(init.Size())
{}

void rk4::Step()
{
        /* pretty:
        k1 = h * dfn(t, state, stateD);
        k2 = h * dfn(t + h/2, state + k1/2, stateD);
        k3 = h * dfn(t + h/2, state + k2/2, stateD);
        k4 = h * dfn(t + h, state + k3, stateD);

        state += (k1 + 2*k2 + 2*k3 + k4)/6;
        t += h;
        */

        // ugly but generally faster:

        k1 = dfn(t, state, stateD);
        k1 *= h/2;
        k2 = dfn(t + h/2, state + k1, stateD);
        k2 *= h/2;
        k3 = dfn(t + h/2, state + k2, stateD);
        k3 *= h;
        k4 = dfn(t + h, state + k3, stateD);
        k4 *= h;
        k1 *= 2;
```

```
        k2 *= 4;
        k3 *= 2;
        k1 += k2;
        k1 += k3;
        k1 += k4;
        k1 /= 6;
        state += k1;

        t += h;
}
```

Listing 15: *main.cpp; example code for the SGA*

```
/*
test.cpp, testbed for for Dawkins v1.0:
Genetic algotithm software for optimising stuff.
W. J. Hoyle, Feb 1994.
 */

#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include "Dawkins.h"
#include "SimFan.h"
#include "MatArray"
#include "controller.h"

#define    DEG_TO_RAD           (2.0*3.141592653589793/360.0)
#define    ROUND_SHORT(x)       ((short)((x)+0.5))

#define    FILE_NAME  "Fan.pop"

#define    OUTPUT_PRECISION     6
#define    SIZE            100
#define    LENGTH          OUTPUT_PRECISION*(TOTAL_RULES-PREDEFINED_RULES)
#define    MUTATE          0.005
#define    GAP             0.05
#define    GREED           1.0

fitnessFunction MyFF;
int InitPopulation(fstream& popFile);
void BuildEngine(Chromosome &it);
void WalkEngine(engine* theEngine, int input1, int input2, fstream& out);
double Completeness(engine *e);

Population*                pop;

void
main ()
{
        cout << "Testbed for Dawkins v1.0 SGA...\n" << endl;

        fstream popFile(FILE_NAME);

        if (InitPopulation(popFile)) {

                int i;

                unsigned short g;

                cout << "How many generations? ";
                cin >> g;

                fstream mon("Monitor", ios::out|ios::app|ios::translated);

                for (i=0; i<g; i++) {
                        pop->TheNextGeneration();
```

```
        pop->Stats(cout);
        cout << endl;
        mon << pop->BestChromosome().fitness << ",\n";
}

cout << "Saving population to file " << FILE_NAME << endl;
popFile.close();
popFile.open
(FILE_NAME, ios::in|ios::out|ios::translated|ios::trunc);
popFile << *pop;


BuildEngine(pop->BestChromosome());



#ifdef HIT_LIST
        for (i=0; i<TOTAL_RULES; i++) {
                controller.rules[i].hits = 0;
        }
#endif


fstream graphFile5
("5 deg Graph", ios::out|ios::trunc|ios::translated);
FanGraph(graphFile5, &controller, 5.0*DEG_TO_RAD);
fstream graphFile20
("20 deg Graph", ios::out|ios::trunc|ios::translated);
FanGraph(graphFile20, &controller, 20.0*DEG_TO_RAD);
fstream graphFile45
("45 deg Graph", ios::out|ios::trunc|ios::translated);
FanGraph(graphFile45, &controller, 45.0*DEG_TO_RAD);

#ifdef HIT_LIST
        fstream hitFile
        ("Rule hits", ios::out|ios::trunc|ios::translated);
        hitFile << setprecision(3) << showpoint << fixed;
        int totalHits = 0;
        for (i=0; i<TOTAL_RULES; i++) {
                totalHits += controller.rules[i].hits;
        }
        for (i=0; i<TOTAL_RULES; i++) {
                hitFile << setw(6);
                hitFile << controller.rules[i].hits/totalHits*100;
                hitFile << "%, ";
                hitFile << controller.rules[i].hits;
                hitFile << " hits on rule " << i+1 << "\n";
        }
#endif

fstream outFile
("Rule outputs", ios::out|ios::trunc|ios::translated);
outFile << setprecision(2) << showpoint << fixed;
for (i=0; i<TOTAL_RULES; i++) {
        outFile << setw(10);
        outFile << controller.rules[i].consequent*8000.0/MAX_GRADE;
        outFile << " Nm, ";
        outFile << controller.rules[i].consequent;
        outFile << ", output from rule " << setw(3) << i+1 << endl;
}

controller.inputs[0] = 10;
controller.inputs[1] = 0;
controller.inputs[2] = 10;
RunEngine(&controller);

controller.inputs[1] = 0; // acceleration = 0
```

```
            fstream surface0
            ("surface0", ios::out|ios::trunc|ios::translated);
            WalkEngine(&controller, 0, 2, surface0);
            controller.inputs[1] = 127;
            fstream surface1
            ("surface1", ios::out|ios::trunc|ios::translated);
            WalkEngine
            (&controller, 0, 2, surface1);
            controller.inputs[1] = 255;
            fstream surface2
            ("surface2", ios::out|ios::trunc|ios::translated);
            WalkEngine(&controller, 0, 2, surface2);

      }

      else {
            cout << "Error: couldn't initialise the population." << endl;
      }
}

void
MyFF(Chromosome &it)
{
      BuildEngine(it);
      it.fitness = -FanFitness(&controller, 5.0*DEG_TO_RAD);
      it.fitness -= FanFitness(&controller, 22.5*DEG_TO_RAD);
      it.fitness -= FanFitness(&controller, 45.0*DEG_TO_RAD);
}

void
BuildEngine(Chromosome &it)
{
      static Chromosome bits(OUTPUT_PRECISION);

      it.SetMarker(0);

      for (int i=PREDEFINED_RULES; i<TOTAL_RULES; i++) {
            it >> bits;
            bits.UnGray();
            short raw = bits.CastToWord();
            short scaled = raw*(double) MAX_GRADE/(pow(2, OUTPUT_PRECISION)-1)
            - 127.5;
            controller.rules[i].consequent = scaled;
      }
}

int
InitPopulation(fstream& popFile)
{
      popFile.seekp(0, ios::end);
      if (popFile.tellg()) {
            pop = new Population(popFile, MyFF);
      }
      else {
            pop = new Population(SIZE, LENGTH, MyFF, MUTATE, GAP, GREED);
            if (pop) {
                  if (pop->Good()) {
                        popFile << *pop;
                  }
            }
      }
      if (!pop) {
            return 0;
      }
      if (!pop->Good()) {
            return 0;
      }
```

```
      return 1;
}

/*
This procedure creates a Mathematica Graphics3D file that describes
the control surface of an engine. It will only plot the
complete parts of the rule base.
 */

#define RESOLUTION 25
#define SURFACE_STEP ((double)MAX_GRADE/RESOLUTION)

void
WalkEngine(engine* theEngine, int input1, int input2, fstream& out)
{
      out << "Graphics3D[{\n";
      out << setprecision(3) << fixed;
      for (double x=0; ROUND_SHORT(x)<MAX_GRADE; x+=SURFACE_STEP) {
            double x2 = x+SURFACE_STEP;
            for (double y=0; ROUND_SHORT(y)<MAX_GRADE; y+=SURFACE_STEP) {
                  double y2 = y+SURFACE_STEP;
                  theEngine->inputs[input1] = ROUND_SHORT(x);
                  theEngine->inputs[input2] = ROUND_SHORT(y);
                  if (Completeness(theEngine) > 0.99) { // allow for errors.
                        RunEngine(theEngine);
                        short z1 = theEngine->output;
                        theEngine->inputs[input2] = ROUND_SHORT(y2);

                        if (Completeness(theEngine) > 0.99) {
                              RunEngine(theEngine);
                              short z2 = theEngine->output;
                              theEngine->inputs[input1] = ROUND_SHORT(x2);
                              if (Completeness(theEngine) > 0.99) {
                                    RunEngine(theEngine);
                                    short z3 = theEngine->output;
                                    theEngine->inputs[input2] =
                                    ROUND_SHORT(y);
                                    if (Completeness(theEngine) > 0.99) {
                        RunEngine(theEngine);
                        short z4 = theEngine->output;
                        out << "Polygon[{";
                        out << "{" << x  << ", " << y  << ", " << z1 << "}, ";
                        out << "{" << x  << ", " << y2 << ", " << z2 << "}, ";
                        out << "{" << x2 << ", " << y2 << ", " << z3 << "}, ";
                        out << "{" << x2 << ", " << y  << ", " << z4 << "}}]";
                        out << ",\n";
                                    }
                              }
                        }
                  }
            }
      }
      out.seekp(-2, ios::cur); // get rid of out last comma and newline
      out << "\n}]" << endl;
}

/*
This function calculates the completeness of the rule base for the cur-
rent
engine inputs.

0 is incomplete.
(0, 1) is subcomplete.
1 is strict complete.
> 1 is overcomplete.

There may be some rounding error.
```

```
 */

double
Completeness(engine *e)
{
     double completeness = 0;
     for (int r=0; r<e->ruleCount; r++) {
          double ruleProduct =
(double) Match(e->rules[r].propositions[0])/MAX_GRADE;
          for (int p=1; p<TOTAL_INPUTS; p++) {
               ruleProduct *=
(double) Match(e->rules[r].propositions[p])/MAX_GRADE;
          }
          completeness += ruleProduct;
     }
     return completeness;
}
```

Listing 16: *simfan.h; public interfaces for a variable pitch propeller simulation*

```
/*
SimFan.h
Public interface for a propellor simulation.
 */

#ifndef SIMFAN_HEADER
#define SIMFAN_HEADER

#include <iostream.h>
#include "fastfuzz.h"

double FanFitness(engine* controller, double p);

void FanGraph(ostream& out, engine* controller, double p);

double* FanSimulation(engine* controller, double p);

#endif
```

Listing 17: *simfan.cpp; a variable pitch propeller simulation*

```
/*
Fan Simulator:
 */

#include "rk4.h"
#include "SimFan.h"
#include <math.h>
#include <iostream.h>

#define    STEP_SIZE  0.25
#define    STEPS      320
#define    TZERO      0.0
#define    RAD_TO_REV (1.0/(2.0*3.141592653589793))

dFunc myDFunc;
double Accn(double n);
short Sensor(double currentN);

double gTorque;
double gPitch;

double
FanFitness(engine* controller, double p)
{
     double* simArray = FanSimulation(controller, p);
```

```
/*
Our fitness is the error integral relative to a set point of 18 rad/s:
 */

      double totalErr = 0.0;

      int i=0;
      while(simArray[i]<17.5 && i<STEPS) i++;
      while (i<STEPS) {
            totalErr += fabs((18.0-simArray[i])*STEP_SIZE);
            i++;
      }

      return totalErr;
}

void
FanGraph(ostream& out, engine* controller, double p)
{
      double* simArray = FanSimulation(controller, p);

      out << "ListPlot[{\n";
      for (int i=0; i<STEPS; i++) {
            out << simArray[i] << ",\n";
      }
      out.seekp(-2, ios::cur);
      out << "\n},\n";
      out << "{\n";
      out << "PlotJoined->True,\n";
      out << "AxesLabel->{\"seconds\", \"rad/s\"},\n";
      out << "PlotRange->{{0," << STEPS << "},{0,20}},\n";
      out << "Ticks->{{{1, 0}, {" << STEPS << ", 80}}, {0, 18}}\n";
      out << "}\n]\n";
}

double*
FanSimulation(engine* controller, double p)
{
      static     double simArray[STEPS];
      vector     initCond(1);
      short lastTorque = 0;
      short     nextTorque = 0;
      short lastRawSpeed = 0;

      gPitch = p;
      gTorque = 0.0;
      initCond[0] = 0;
      rk4 r(initCond, myDFunc, TZERO, STEP_SIZE);

      for (int i=0; i<STEPS; i++) {

/*
Set up controller inputs:
 */

            short rawSpeed = Sensor(r(0));
            double speed = r(0);

/*
First, scale the raw sensor reading into the range 0-MAX_GRADE.
5867 sensor clicks/quarter second corresponds to a propellor speed of
18.0 rad/s, which is out set-point. Subtracting this from the raw reading
gives us a zero. In order to scale the zeroed reading so that the raw
readings that represent the range 17.5-18.5 rad/s, we multiply by 4/5 and
add 127.
 */
```

```
            short n = (rawSpeed - 5867)*4/5;  // scale n into 0-MAX_GRADE
            n += MAX_GRADE/2;

            if (n < 0) n = 0;                 // clip
            if (n > MAX_GRADE) n = MAX_GRADE;

/*
Now calculate the acceleration and scale it into 0-255, with 127 repre-
senting the set-point (that is, 0 acceleration). The maximum acceleration
is about 0.75 rad/s/s. After scaling by 0.5 and adding 127, the range of
accelerations represented is about -0.75 to +0.75 rad/s/s.
 */

            short a = (rawSpeed - lastRawSpeed)/2;
            a += MAX_GRADE/2;

            if (a < 0) a = 0;                 // clip
            if (a > MAX_GRADE) a = MAX_GRADE;

/*
Put the inputs in the controller:
 */

            controller->inputs[0] = n;
            controller->inputs[1] = a;
            controller->inputs[2] = lastTorque;

/*
Find the output:
 */

            RunEngine(controller);

            nextTorque = lastTorque + controller->output * 2 ;
            if (nextTorque < 0) nextTorque = 0;
            if (nextTorque > MAX_GRADE) nextTorque = MAX_GRADE;

            lastTorque = nextTorque;

            gTorque = nextTorque * 4000.0/MAX_GRADE;
            // scale torque into 0 to 4000 Nm

/*
simulate the fan by integrating the system under this torque:
 */

            r.Step();

/*
save the result:
 */
            lastRawSpeed = rawSpeed;
            simArray[i] = r(0);
      }

      return simArray;
}


vector &myDFunc(double t, const vector &state, vector& stateD)
{
      stateD[0] = Accn(state(0));

      return stateD;
}
```

```
/*
Sensor simulator. This is supposed to approximate an 8 bit encoder
(that is, 256 clicks per revolution) which is driven at 8 times the
shaft speed. The encoder feeds into an accumulator that is sampled
and reset every  quarter second. The accumulated total of clicks over the
interval between samples represents the average speed during that time.
 */

short
Sensor(double currentN)
{
      static double lastN = 0;

      /*
      We have to approximate the average speed by interpolation:
       */

      double average = (currentN + lastN)/2.0;

      /*
      We can accumulate 32767 clicks in 15 bits without overflowing. This
      gives us a max speed of about 100 rad/s, which is plenty.
       */

      short clicks = RAD_TO_REV * average * 256 * 8;

      /*
      Remember N for next time:
       */

      lastN = currentN;

      return clicks;
}

/*
A simple dynamic model of a variable pitch propeller in air:
 */

double
Accn(double n)
{
      return (gTorque + ((n<0)?1:-1)*(n*n*12.47*sin(gPitch)))/5350.0;
}
```

Listing 18: *controller.h; public intefaces to a fan controller data structure*

```
#ifndef CONTROLLER_HEADER
#define CONTROLLER_HEADER

#include "fastfuzz.h"

#ifdef __cplusplus
extern "C" {
#endif

#define    TOTAL_RULES     TORQUE_SETS*SPEED_SETS*ACCN_SETS-DEAD_RULES
#define    DEAD_RULES           0
#define    SPEED_SETS           6
#define    ACCN_SETS            3
#define    TORQUE_SETS          2
#define    PREDEFINED_RULES     8

extern engine controller;

#ifdef __cplusplus
}
```

```
#endif

#endif
```

Listing 19: *controller.c; a fan controller data structure*

```
#include "controller.h"

short inputs[TOTAL_INPUTS];

set speedSets[] = {
      0,
      0,
      10,
      77,
      127,
      177,
      255,
      255
};

set accnSets[] = {
      0,
      0,
      127,                  /* set point */
      255,
      255
};

set torqueSets[] = {
      0,
      0,
      255,
      255
};

#define sIsOff   {&speedSets[0],  &inputs[0]}
#define sIsVLow  {&speedSets[1],  &inputs[0]}
#define sIsLow   {&speedSets[2],  &inputs[0]}
#define sIsRight {&speedSets[3],  &inputs[0]}
#define sIsHigh  {&speedSets[4],  &inputs[0]}
#define sIsVHigh {&speedSets[5],  &inputs[0]}

#define aIsNeg   {&accnSets[0],   &inputs[1]}
#define aIsZero  {&accnSets[1],   &inputs[1]}
#define aIsPos   {&accnSets[2],   &inputs[1]}

#define tIsZ     {&torqueSets[0], &inputs[2]}
#define tIsMax   {&torqueSets[1], &inputs[2]}

rule rules[] = {
      {{sIsOff,  aIsNeg,  tIsZ }, 127},
      {{sIsOff,  aIsNeg,  tIsMax}, 127},

      {{sIsOff,  aIsZero, tIsZ }, 127},
      {{sIsOff,  aIsZero, tIsMax}, 127},

      {{sIsOff,  aIsPos,  tIsZ }, 127},
      {{sIsOff,  aIsPos,  tIsMax}, 127},

      {{sIsRight, aIsZero, tIsZ }, 0},
      {{sIsRight, aIsZero, tIsMax}, 0},

      {{sIsVLow,   aIsNeg,  tIsZ }, 0},
      {{sIsVLow,   aIsNeg,  tIsMax}, 0},

      {{sIsVLow,   aIsZero, tIsZ }, 0},
```

```
      {{sIsVLow,   aIsZero, tIsMax}, 0},

      {{sIsVLow,   aIsPos,  tIsZ }, 0},
      {{sIsVLow,   aIsPos,  tIsMax}, 0},

      {{sIsLow,   aIsNeg,  tIsZ }, 0},
      {{sIsLow,   aIsNeg,  tIsMax}, 0},

      {{sIsLow,   aIsZero, tIsZ }, 0},
      {{sIsLow,   aIsZero, tIsMax}, 0},

      {{sIsLow,   aIsPos,  tIsZ }, 0},
      {{sIsLow,   aIsPos,  tIsMax}, 0},

      {{sIsRight, aIsNeg,  tIsZ }, 0},
      {{sIsRight, aIsNeg,  tIsMax}, 0},

      {{sIsRight, aIsPos,  tIsZ }, 0},
      {{sIsRight, aIsPos,  tIsMax}, 0},

      {{sIsHigh,  aIsNeg,  tIsZ }, 0},
      {{sIsHigh,  aIsNeg,  tIsMax}, 0},

      {{sIsHigh,  aIsZero, tIsZ }, 0},
      {{sIsHigh,  aIsZero, tIsMax}, 0},

      {{sIsHigh,  aIsPos,  tIsZ }, 0},
      {{sIsHigh,  aIsPos,  tIsMax}, 0},

      {{sIsVHigh,  aIsNeg,  tIsZ }, 0},
      {{sIsVHigh,  aIsNeg,  tIsMax}, 0},

      {{sIsVHigh,  aIsZero, tIsZ }, 0},
      {{sIsVHigh,  aIsZero, tIsMax}, 0},

      {{sIsVHigh,  aIsPos,  tIsZ }, 0},
      {{sIsVHigh,  aIsPos,  tIsMax}, 0}
};

engine controller = {
      TOTAL_RULES,
      rules
};
```

Listing 20: *controller.c; an optimised fan controller data structure*

```
#include "controller.h"

short inputs[TOTAL_INPUTS];

set speedSets[] = {
      0,
      0,
      10,
      77,
      127,
      177,
      177
};

set accnSets[] = {
      0,
      0,
      127,                  /* set point */
      255,
      255
};
```

```
set torqueSets[] = {
     0,
     0,
     255,
     255
};

proposition sIsOff   = {&speedSets[0],  &inputs[0]};
proposition sIsVLow  = {&speedSets[1],  &inputs[0]};
proposition sIsLow   = {&speedSets[2],  &inputs[0]};
proposition sIsRight = {&speedSets[3],  &inputs[0]};
proposition sIsHigh  = {&speedSets[4],  &inputs[0]};

proposition aIsNeg  = {&accnSets[0],   &inputs[1]};
proposition aIsZero = {&accnSets[1],   &inputs[1]};
proposition aIsPos  = {&accnSets[2],   &inputs[1]};

proposition tIsZ    = {&torqueSets[0], &inputs[2]};
proposition tIsMax  = {&torqueSets[1], &inputs[2]};

rule rules[] = {
     {{&sIsOff,  NULL,  NULL}, 128},

     {{&sIsRight, &aIsZero, NULL}, 0},

     {{&sIsVLow,   &aIsZero, &tIsZ}, 99},
     {{&sIsVLow,   &aIsZero, &tIsMax}, 14},

     {{&sIsVLow,   &aIsPos,  &tIsZ}, -115},
     {{&sIsVLow,   &aIsPos,  &tIsMax}, -38},

     {{&sIsLow,   &aIsNeg,  &tIsZ}, 62},
     {{&sIsLow,   &aIsNeg,  &tIsMax}, 127},

     {{&sIsLow,   &aIsZero, NULL}, 10},

     {{&sIsLow,   &aIsPos,  &tIsZ }, -50},
     {{&sIsLow,   &aIsPos,  &tIsMax}, -123},

     {{&sIsRight, &aIsNeg,  &tIsZ }, 58},
     {{&sIsRight, &aIsNeg,  &tIsMax}, 123},

     {{&sIsRight, &aIsPos,  &tIsZ }, -115},
     {{&sIsRight, &aIsPos,  &tIsMax}, -107},

     {{&sIsHigh,  &aIsNeg,  NULL}, -111},

     {{&sIsHigh,  &aIsZero, &tIsZ }, -22},
     {{&sIsHigh,  &aIsZero, &tIsMax}, -99},

     {{&sIsHigh,  &aIsPos,  &tIsZ }, 10},
     {{&sIsHigh,  &aIsPos,  &tIsMax}, -22}
};

engine controller = {
     TOTAL_RULES,
     rules
};
```