4.800

Flint Reference

by Rebecca Shalfield

Flint

The contents of this manual describe the product, FLINT toolkit, and are believed correct at the time of going to press. They do not embody a commitment on the part of Logic Programming Associates (LPA), who may from time to time make changes to the specification of the product in line with their policy of continual improvement. No part of this manual may be reproduced or transmitted in any form, electronic or mechanical, for any purpose other than the licensee's personal use without the prior written agreement of LPA.

Logic Programming Associates Ltd.
Studio 30
The Royal Victoria Patriotic Building
Trinity Road
London
SW18 3SX
England

phone: **+44 (0) 20 8871 2016**

fax: **+44 (0) 20 8874 0449**

web site: **http://www.lpa.co.uk/**

20 July, 2005

Contents

Chapter 1 - Flint – Reasoning Under Uncertainty

Given a rule:

rule1: if A & B then C

there are 3 potential areas for uncertainty.

- Uncertainty in data (how true are A and B)

- Uncertainty in the rule (how often does A and B imply C)

- Impreciseness in general

The first two can be handled using techniques based on probability theory and the third using fuzzy logic.

Flint supports various modes of uncertainty handling, namely:

- Fuzzy Logic
- Bayesian updating
- Certainty factors

These are presented in a uniform and consistent way, using a common set of access routines.

These are made available in the context of a powerful AI programming environment supported by text editing facilities, compilers, debuggers and a variety of GUI tools.

This means you can integrate uncertainty with databases, spreadsheets, expert systems, graphics, etc.

Rules with uncertainty can be represented in both a native Prolog syntax, or where flex is present, in the KSL syntax that flex uses.

Chapter 2 - Fuzzy Logic

Fuzzy logic is a technology that allows realistic complex models of the real world to be defined with some simple and understandable fuzzy variables and fuzzy rules. Flint provides you with the ability to use fuzzy logic within the context of the Prolog or flex KSL languages.

Why Use Fuzzy Logic?

Expert systems have to deal with items that come from the real world. Typically, our understanding of the workings of items in the real world is imprecise. Fuzzy logic gives a mathematically sound methodology for dealing with imprecision.

Traditional expert systems rely on exact thresholds in the conditions of their rules, e.g. "if the temperature is greater than 120". This often leads to a exaggerated jump in the output as the input value approaches, and then exceeds, the threshold.

Fuzzy logic smoothes out these thresholds by allowing rules to use qualitative descriptions in their conditions, e.g. "if the temperature is hot" where 'hot' does not refer to a precise value, but rather a range of values that could possibly be described as hot. This leads to rules having a greater influence as their conditions become more and more true.

Dr. Earl Cox clarifies this when he says "Conventional logic and computers run on a logic that is Boolean - that is, a thing is, or it isn't. For instance, you're tall or you're not, and you have to define tall as a number, say 6 feet. But, using common sense, if 6 feet is tall, you wouldn't say someone who is 5 feet 11 ¾ inches is short, would you?".

The following example contrasts the use of Boolean and fuzzy logic. The rules refer to the scores of a medical examination as being either high or low:

```
if the score is high then patient is fit

if the score is low then patient is unfit
```

In a crisp expert system, the qualifier 'high' might refer to all the values of 50 and above and 'low' to all the values below 50. In this system, if someone scores 100 they are definitely 'fit' and a score of 0 means they are definitely 'unfit'. All fine and good, these results are what we would expect, the real difficulty comes around the threshold values. If someone scores 49 they are definitely 'unfit', whereas if someone scores 50 they are definitely 'fit'. A large jump in the output over a change of just one point in the score.

In contrast, in a fuzzy expert system, the qualifier 'high' could be defined so that the value 0 is considered as definitely not 'high', the value 100 as definitely 'high' and values in between are considered gradually more and more 'high' as they approach 100. The other qualifier, 'low', similarly could be defined so that the value 0 is considered as definitely 'low', the value 100 as definitely not 'low' and the values in between considered gradually less and less 'low' as they approach 100. In this system, if someone scores 100 this means they are definitely 'fit' and a score of 0 means they are definitely 'unfit'. These results are what we would expect and conform to the traditional expert system results. The real advantage of fuzzy logic can be seen by looking at the mid-range values where the traditional expert system's thresholds lay. If someone scores 49 they are slightly more 'unfit' than 'fit', if they score 50 they are half 'unfit' and half 'fit' and if they score 51 they are slightly more 'fit' than 'unfit'.

This feature of fuzzy logic is of particular use in expert system controllers, where sudden changes in behaviour may be inappropriate. An example of this is the fuzzy logic system used to control the Tokyo underground, where trains are brought to a smooth stand-still at the stations in all types of conditions.

Another main feature of fuzzy logic is its robustness and tolerance for faults. In traditional expert systems, only one rule is used at a time, e.g. the temperature is either hot or cold. This means that each rule applied has a strong influence on the outcome. Consequently, it is important to ensure that each rule is correct.

In fuzzy logic expert systems, all rules are used but to differing amounts. Each rule influences the outcome according to the extent its conditions are true, which leads to a more consensus-based approach. There is less dependency on the correctness of any individual rule, which makes fuzzy logic expert systems less "brittle" than traditional expert systems.

Prof. Dr. Lotfi A. Zadeh says "Exploit the tolerance for imprecision, uncertainty and partial truth to achieve tractability, robustness and low solution cost." Fuzzy logic is a means for achieving this end.

Chapter 3 - Fuzzy Logic Programs

This section gives an overview of the terminology surrounding fuzzy logic and the structure of a fuzzy logic program. If you are familiar with the concepts and terminology involved in fuzzy logic you may want to skip this section and go to the section titled *Chapter 4* - Fuzzy Components on page *12*.

Fuzzy Logic Terminology

There are two main components to a fuzzy logic program: fuzzy variables and fuzzy rules. The keywords in the following text are shown in bold.

Fuzzy variables take a range of numeric values. The value of a fuzzy variable is referred to as a 'crisp' value. The range can be divided into several sub-ranges by defining qualifiers for the fuzzy variable. Fuzzy variable qualifiers consist of a name, known as a **linguistic qualifier**, and a **membership function** which shows for each possible 'crisp' value of the fuzzy variable its **degree of membership** of the **fuzzy set**.

When a **'crisp' value** is assigned to a fuzzy variable this is converted into a degree of membership for each linguistic qualifier using its membership function. When a 'crisp' value is required back from the fuzzy variable, so that it can be used outside the fuzzy logic system, a **de-fuzzifying expression** is used. This expression is based on the degrees of membership of the fuzzy variable's qualifiers.

The degrees of membership of a fuzzy variable qualifier can be affected by a **linguistic** '**hedge**'. This has the effect of either concentrating or diluting the fuzziness of the qualifier.

A **Fuzzy Rule** is a rule that refers to one or more fuzzy variable qualifiers in its conditions and a one or more fuzzy variable qualifiers in its conclusion. Rules are applied to fuzzy variables by a process called **propagation**. When a rule is applied it looks at the degrees of membership for the fuzzy variable qualifiers mentioned in its conditions and calculates the new degree of membership for the fuzzy variable qualifier mentioned in its conclusion. The calculation depends upon whether the conditions are formed by conjunctions (and), disjunctions (or), negations (not) or a mixture of these. Fuzzy rules can also be joined together into a **fuzzy matrix**, also known as a **fuzzy associative memory**.

The Structure of a Fuzzy Logic Program

A fuzzy logic program can be viewed as a three stage process:

- Stage 1 - Fuzzification; the 'crisp' input values are assigned to the appropriate input fuzzy variables. The 'crisp' input value is converted into a degree of membership for each of the qualifiers for the fuzzy variable it is assigned to.

- Stage 2 - Propagation; fuzzy rules are applied to the fuzzy variables and their qualifiers. When a fuzzy rule is applied to some fuzzy variables the degrees of membership for the qualifiers mentioned in the conditions of the rule are propagated to the qualifiers mentioned in the conclusions of the rule.

- Stage 3 - De-fuzzification; the resultant degrees of membership for the qualifiers of the output fuzzy variables are converted back into 'crisp' values.

The following diagram shows this three stage process for an example fuzzy steam turbine program. Before entering the fuzzy logic program the 'crisp' values for the 'temperature' and 'pressure' of the turbine are found to be 200°C and 15Kpa respectively.

In the first stage of the fuzzy logic program these values are fuzzified into degrees of membership for the qualifiers of the 'temperature' and 'pressure' fuzzy variables. The 'temperature' qualifier with the highest degree of membership is 'normal' followed by 'cool'. The 'pressure' qualifier with the highest degree of membership is 'weak' followed by 'low'.

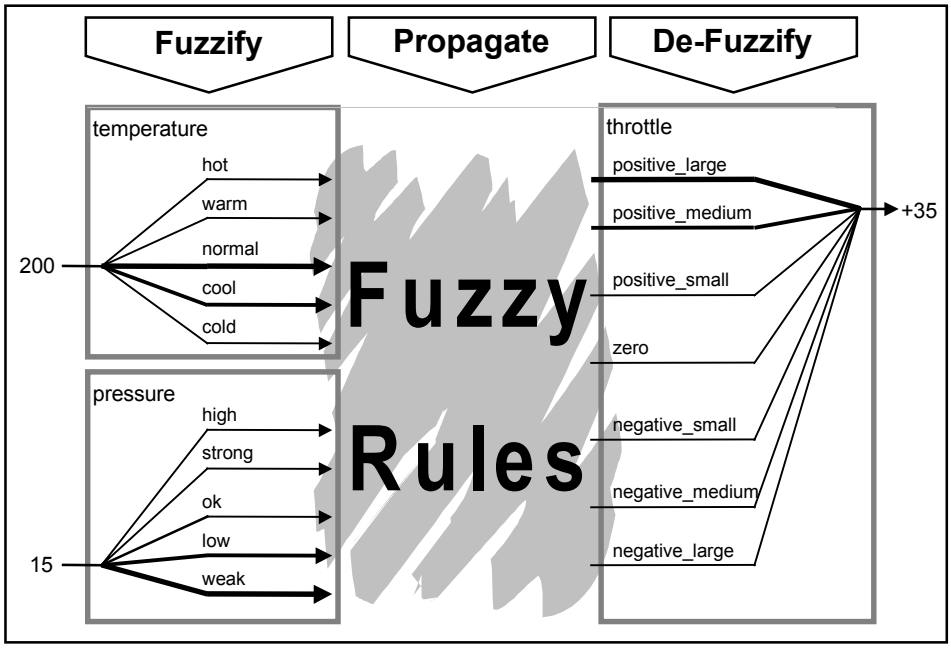


Figure 1 - The Stages of a Fuzzy Logic Program

The second stage is the fuzzy rule propagation; this stage generates, using the fuzzy rules and the degrees of membership of the input fuzzy variable qualifiers, a resultant degree of membership for each of the qualifiers of the 'throttle' fuzzy variable. The 'throttle' qualifier with the highest degree of membership is 'positive_large' followed by 'positive_medium'.

The final stage is to take all the degrees of membership for the qualifiers of the 'throttle' fuzzy variable and calculate a 'crisp' output value. This value turns out to be 35. After exiting the fuzzy logic program with the 'crisp' value this can then be applied back in some way to the throttle for the steam turbine.

Chapter 4 - Fuzzy Components

This section tells you how to define the components that make up a fuzzy logic program. The components fall into four groups: fuzzy variables, linguistic 'hedges', fuzzy rules and fuzzy matrices.

Defining Fuzzy Variables

Fuzzy variables and their qualifiers are the basis for fuzzy logic programs. You provide input values for fuzzy variables prior to applying the fuzzy rules and retrieve output values at the end of the process. To define a fuzzy variable, define its name, range, its qualifiers and a de-fuzzifying expression.

The Anatomy Of A Fuzzy Variable

The following diagrams show Prolog and KSL syntax fuzzy variable definitions. The parts in bold indicate keywords and syntax recognised by the fuzzy interpreter. The plain text indicates the names and parameters chosen by the programmer. The qualifiers are highlighted by a dotted border.

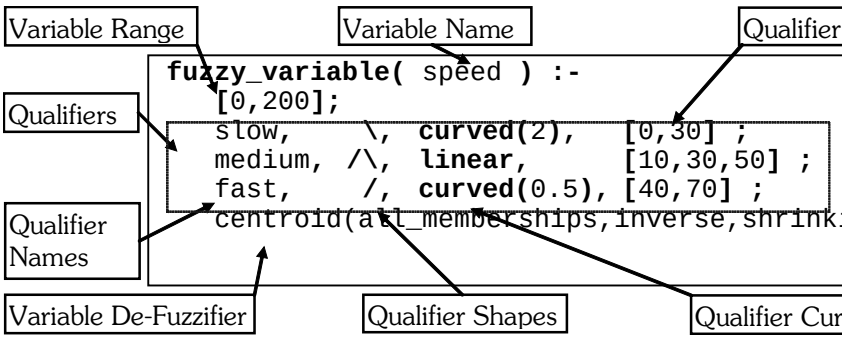


Figure 2 - The parts of a Prolog fuzzy variable definition

The next diagram shows the KSL definition of a fuzzy variable.

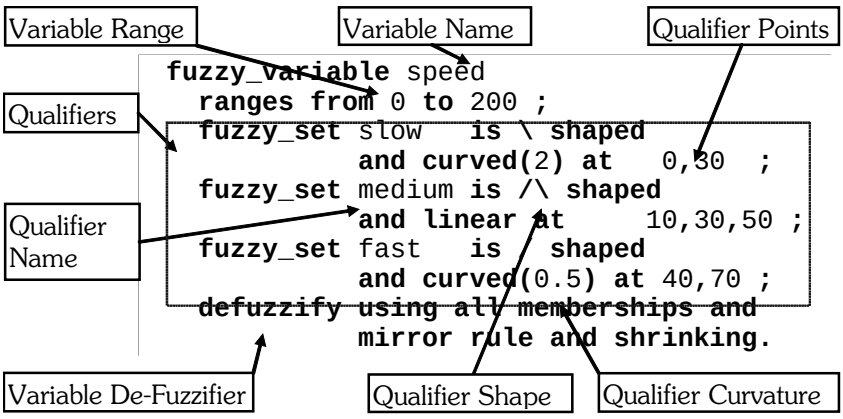


Figure 3 - The parts of a flex KSL fuzzy variable definition

The Fuzzy Variable Name

The name of a fuzzy variable is an atom, usually descriptive of the quantity being defined, to be used whenever referring to the variable. An example of a fuzzy variable name could be 'temperature'. This would be defined by a fuzzy variable program that started:

```
fuzzy_variable( temperature ) :-
```

Setting a Fuzzy Variable's Lower and Upper Bounds

The lower and upper bounds of a fuzzy variable define the range of possible values that the variable can take. For example the fuzzy 'temperature' variable could have a lower bound of -100 and an upper bound of 150. This would be defined by a fuzzy variable program that started as follows:

```
fuzzy_variable( temperature ) :-
    [-100,150] ;
```

The explicit setting of lower and upper bounds for a fuzzy variable is optional. If this section is left out, the fuzzy logic system works out the lower and upper bounds for the fuzzy variable from the shapes of its qualifier membership functions. For example our fuzzy 'temperature' variable could have two qualifiers, 'hot' and 'cold'. The 'cold' qualifier could specify that everything below -20 is definitely 'cold' and that everything above 10 is definitely not 'cold'. The 'hot' qualifier could specify that everything above 80 is definitely 'hot' and that everything below 10 is definitely not 'hot'. In this case, if the upper and lower bounds for the 'temperature' variable were not specified, the fuzzy logic system would assign the value -20 to the lower bound and 80 to the upper bound.

Defining Fuzzy Variable Qualifiers

Fuzzy variable qualifiers consist of a name, known as a 'linguistic qualifier', and a membership function which shows for each possible 'crisp' value of the fuzzy variable its degree of membership of the set referred to by the qualifier.

The membership function is defined by its overall shape, its curvature and some relevant points. This could be represented by a graph where the *Y* axis shows the degree of membership of the set and the *X* axis the 'crisp' value of the fuzzy variable. The degree of membership is a value somewhere between 0 and 1. A degree of membership of 1 specifies that a value is definitely a member of the set. A degree of membership of 0 specifies that a value is definitely not a member of the set. Numbers in between define in fuzzy terms varying degrees of membership of a set.

There are seven different overall shapes that can be used: an upwards slope, a downwards slope, an upwards triangle, a downwards triangle, an upwards trapezoid, a downwards trapezoid and a freehand shape. The following table shows the symbols and points needed to define each shape.

Symbol	Points	Description
\	[A,B]	A downward slope
/	[A,B]	An upward slope
/\	[A,B,C]	An upward pointing triangle
\/	[A,B,C]	A downward pointing triangle
/-\	[A,B,C,D]	An upward pointing trapezoid
\-/	[A,B,C,D]	A downward pointing trapezoid
?	[V_1/M_1,V_2/$M_{2,...}$ V_k/M_k]	A freehand shape

Table 1 - The Qualifier Membership Function Shapes

The following sections look at each basic membership function shape in turn. The symbol column shows the symbol used to define the shape. The points column shows the number of points required to define the shape. The shape column contains a diagram of the function shape, showing where the defined points occur. The points LB and UB in the diagram refer to the lower and upper bounds for the fuzzy variable.

The Downward Slope Shape

The downward slope shape is normally used for qualifiers that cover the lower ranges of a fuzzy variable.

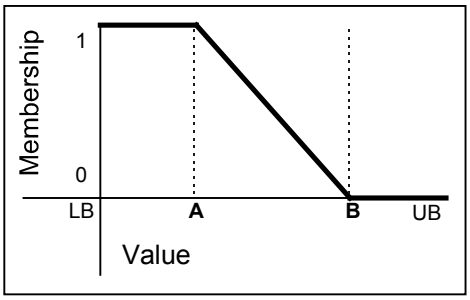
Symbol	Points	Shape
\	[A,B]	

Table 2 - The downward slope shape membership function

To illustrate the downward slope value, let us suppose we have a 'direction' fuzzy variable with possible values between 0 and 180. The following example defines a 'left' qualifier for the 'direction' variable where all the values below 60 are definitely 'left', all the values above 90 are definitely not 'left', the values in between 60 and 90 become less and less definitely 'left' as they approach 90:

```
left, \, linear, [60,90] ;
```

Note that 'linear', in the above example, means that the specified points are joined by straight lines only.

The Upward Slope Shape

The upward slope shape is normally used for qualifiers that cover the upper ranges of a fuzzy variable.

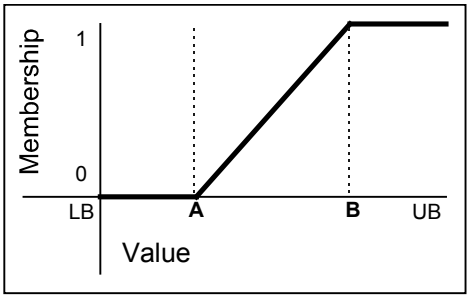
Symbol	Points	Shape
/	[A,B]	

Table 3 - The upward slope shape membership function

Using our 'direction' fuzzy variable, the following example shows a 'right' qualifier where all the values below 90 are definitely not 'right' and all the values above 120 are definitely 'right'. The values in between 90 and 120 become more and more definitely 'right' as they approach 120:

```
right, /, linear, [90,120] ;
```

The Upward Triangle Shape

The upward triangle shape is normally used for qualifiers that cover a range centred around a single definite value in the middle of a fuzzy variable.

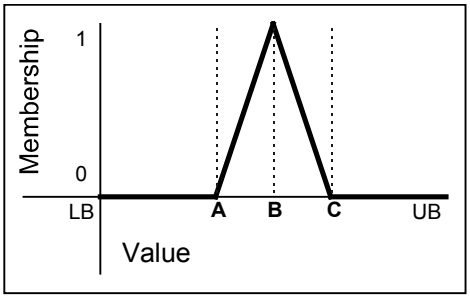
Symbol	Points	Shape
/\	[A,B,C]	

Table 4 - The upward triangle shape membership function

Using our 'direction' fuzzy variable, the following example shows a 'central' qualifier where all the values below 45 are definitely not 'central', 90 is definitely 'central' and all the values above 135 are

definitely not 'central'. The values in between 45 and 135 become more and more definitely 'central' as they approach 90:

central, /\, **linear, [**45,90,135**] ;**

The Downward Triangle Shape

The downward triangle shape is normally used for qualifiers that indicate the negation of a quality, e.g. 'not_warm', that cover everything *except* a range centred around a single definite value somewhere in the middle of a fuzzy variable.

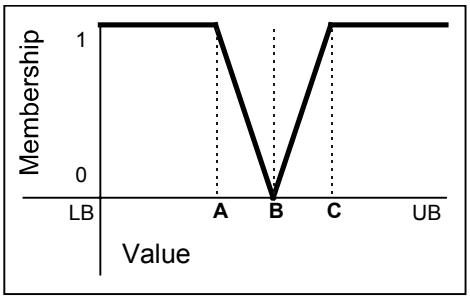
Symbol	Points	Shape
\/	[A,B,C]	

Table 5 - The downward triangle shape membership function

Using our 'direction' fuzzy variable, the following example shows an 'extreme' qualifier where all the values below 30 are definitely 'extreme', 90 is definitely not 'extreme' and all the values above 150 are definitely 'extreme'. The values in between 30 and 150 become less and less definitely 'extreme' as they approach 90:

extreme, \/, **linear, [**30,90,150**] ;**

The Upwards Trapezoid Shape

The upwards trapezoid shape is normally used for qualifiers that cover a range of values in the middle of a fuzzy variable.

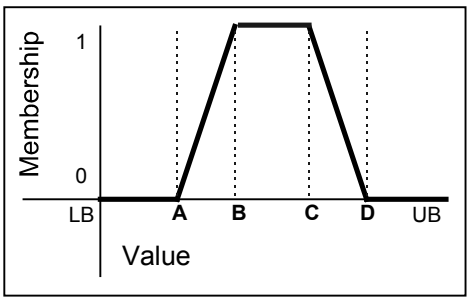
Symbol	Points	Shape
/-\	[A,B,C,D]	

Table 6 - The upwards trapezoid shape membership function

Using our 'direction' fuzzy variable, the following example shows the 'forward' qualifier where a range of values between 75 and 105 are considered to be definitely 'forward'. All the values below 60 are

definitely not 'forward' and all the values above 120 are definitely not 'forward'. The values in between 60 and 75 become more and more definitely 'forward' as they approach 75 and the values in between 105 and 120 become less and less definitely 'forward' as they approach 120:

forward**, /-\, linear, [**60,75,105,120**] ;**

The Downwards Trapezoid Shape

The downwards trapezoid shape is normally used for qualifiers that indicate the negation of a quality, e.g. 'not_mild', that cover everything *except* a range of values somewhere in the middle of a fuzzy variable.

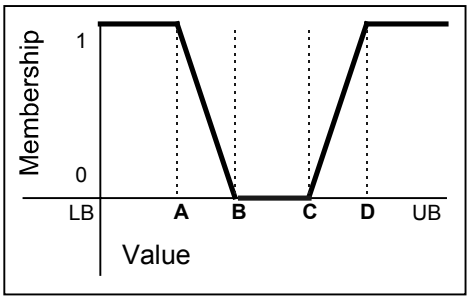
Symbol	Points	Shape
\-/	[A,B,C,D]	

Table 7 - The downwards trapezoid shape membership function

Using our 'direction' fuzzy variable, the following example shows the 'peripheral' qualifier where a range of values between 60 and 120 are definitely not 'peripheral'. All the values below 30 are definitely 'peripheral' and all the values above 150 are definitely 'peripheral'. The values in between 30 and 60 become less and less definitely 'peripheral' as they approach 60 and the values in between 120 and 150 become more and more definitely 'peripheral' as they approach 150:

peripheral**, _/, linear, [**30,60,120,150**] ;**

The 'Freehand' Shape

The 'freehand' shape covers all possible uses. Each point is described in the form *V/M*, where *V* is a fuzzy variable value and *M* is a membership value. The respective start and end values for *M* can be either 0 or 1. The values for *V* must be given in strictly ascending order.

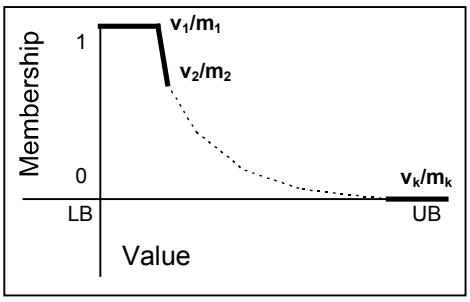
Symbol	Points	Shape
?	[v_1/m_1,v_2/m_2... v_k/m_k]	

Table 8 - The freehand shape membership function

The following example shows how the 'left' qualifier, shown previously, could be defined as a more rounded membership graph using the freehand shape and a set of points:

extreme**, ?, linear, [** 60/1, 65/.75, 70/.4, 85/.25, 90/0 **] ;**

Defining the Curvature of a Qualifier

The examples in the previous section all used straight lines to join the points of their membership graphs. It is also possible to add a curvature parameter to the overall shape. The purpose of this is to round off the straight lines drawn between the points.

The curvature parameter can be a value anywhere between 0.1 and 9.9. The shape of the curve varies according to the relative positions of the start and end points, i.e. whether the slope between the points is increasing, flat or decreasing and whether the curvature parameter is less than 1, equal to 1 or greater than 1. These nine cases are shown in the following table:

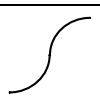
Curvature Parameter	**Shape**		
	Slope Between Points		
	Increasing	Decreasing	Flat
Less than 1			
Equal to 1			
Greater than 1			

Table 9 - Curvature Parameter Curve Shapes

The curvature parameter also determines how extreme the curve is. The closer the curvature parameter is to 1, the closer to a straight line the resultant curve is. The further away from 1, the more exaggerated the curve.

The following graph shows the upward slope membership function (the dotted line shows the 'linear' version of the function). The graph is the function with a curvature parameter of 0.5.

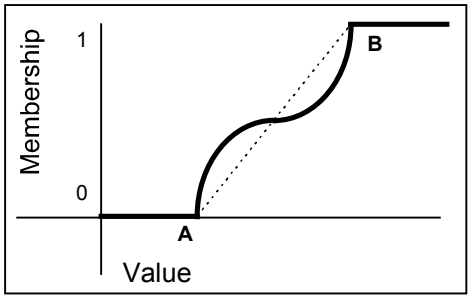


Figure 4 - an upward slope shape with a curvature of 0.5

Notice that this shape, typical of curvature parameters less than 1, tends to decrease the slope of the function around the mid-values between points A and B, making the function less like the vertical 'step' function of traditional Boolean logic and thus increasing the fuzziness of the membership function.

The following graph shows the upward pointing triangle function with a curvature parameter of 2.

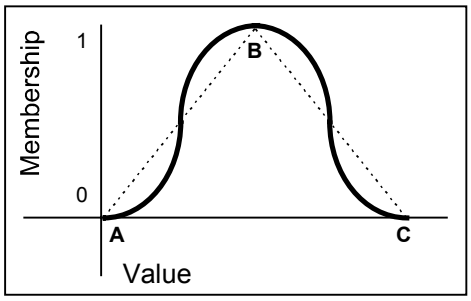


Figure 5 - an upward slope shape with a curvature of 2

Notice that the shape produced this time, typical of curvature parameters greater than 1, tends to increase the slope of the curve around the mid-values between points A and B and B and C, making the curve more like the vertical 'step' function of traditional Boolean logic and thus decreasing the fuzziness of the membership function.

Defining Fuzzy Variable De-Fuzzifiers

A de-fuzzifier is used to convert a fuzzy variable's current qualifier membership values into a single 'crisp' value for the variable as a whole. This is done when you need to get a value back from the variable. There are two built-in defuzzifiers, centroid, and peak. User-defined expressions may also be used.

The Centroid Method

The default de-fuzzifier is the **centroid** method. This works by finding the 'centre of gravity' of the collection of all the degrees of membership for all the qualifiers of the fuzzy variable. To illustrate the centroid method, let's take a fuzzy variable called 'rainfall' that has three qualifiers: 'light', 'medium' and 'heavy'. At the end of our fuzzy logic session these qualifiers have the following values: 'light' 0.75, 'medium' 0.25 and 'heavy' 0. The three qualifiers - light, heavy and medium - are shown in the following diagram:

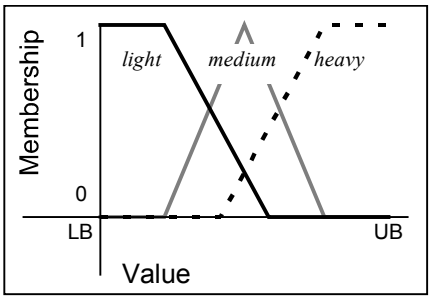


Figure 6 - the qualifiers of the fuzzy 'rainfall' variable

The centroid method then takes the membership function for each qualifier and cuts off any portion of the function which contains values above the current value for the qualifier. So the 'light' qualifier in our example would have a cut-off point at 0.75. This is shown in the following diagram:

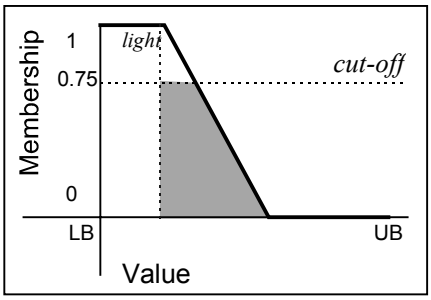


Figure 7 - the cut-off point for the 'light' qualifier

The 'medium' qualifier in our example has a cut-off point at 0.25. The portion of the membership function which is going to be used is shown as the shaded part of the following diagram:

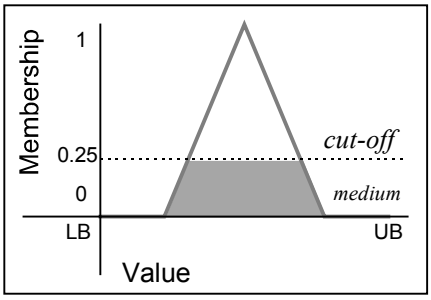


Figure 8 - the cut-off point for the 'medium' qualifier

The 'heavy' qualifier has a current value of 0 so its cut-off point is along the x axis and no portion of its membership function will be used. The adjusted 'light' and 'medium' qualifiers are now merged and their combined centre of gravity is calculated. The following diagram shows the combination:

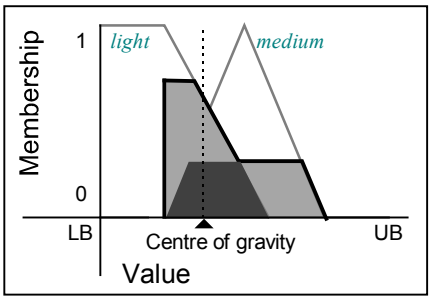


Figure 9 - combining the cut-off membership functions

The shaded portion of the diagram is the shape that will be considered when calculating the centre of gravity. Note that part of the diagram has been shaded twice by both the 'light' and the 'medium' qualifiers. This portion of the shape is considered to be twice as dense as the rest and this is also taken into account when calculating the centre of gravity. The centre of gravity indicates a single value on the x axis which is deemed to be the resultant 'crisp' value for the fuzzy variable.

The Peak Method

Another built-in de-fuzzier is the **peak** method. This involves finding the qualifier or qualifiers with the highest degree of membership and then calculating the mid-point of all the plateaux occurring at that cut-off point.

To illustrate the peak method, let's take our fuzzy 'rainfall' variable again. This time we look for the qualifier which currently has the highest membership value, which in this case is the 'light' qualifier. The membership function for this qualifier is then cropped at its current membership value and the mid-point of the resultant plateau is calculated. This is shown in the following diagram:

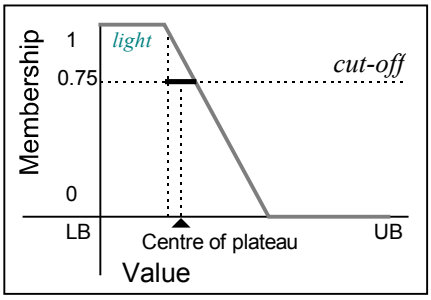


Figure 10 - The peak de-fuzzifying method

Defining Your Own Method

The final method for de-fuzzifying a fuzzy variable allows you to define your own expression for calculating the resultant value. Usually this expression takes the current values for the fuzzy variables qualifiers into account.

The following shows a de-fuzzifying expression that calculates the 'crisp' value by taking the sum of 10 times the 'light' qualifier, 20 times the 'medium' qualifier and 30 times the 'heavy' qualifier divided by the sum of the 'light', 'medium' and 'heavy' qualifiers.

```
30 * light + 20 * medium + 30 *  heavy
/ light + medium + heavy .
```

A Fuzzy Variable Example

As an example of creating a fuzzy variable we could specify a fuzzy variable called 'temperature' and assign it the range -50 to 50. Notionally we will refer to the temperature values in degrees Celsius. The program that represented this fuzzy variable would start as follows:

```
fuzzy_variable(temperature):-
    [-50,50];
```

Then we could define four qualifiers for the 'temperature' variable: 'freezing', 'cold', 'warm' and 'hot'. We want to say that 'freezing' definitely refers to all the temperature values below zero and possibly the values between 0 and 6. This would be done by adding the following line to the definition:

```
freezing, \, linear, [0,6];
```

This defines the following 'freezing' membership function for the fuzzy variable 'temperature':

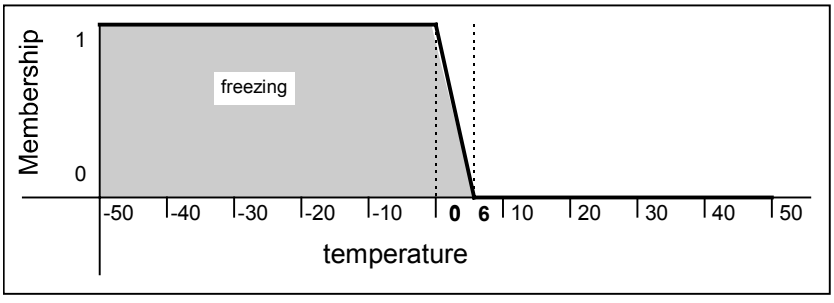


Figure 11 - the 'freezing' qualifier for the 'temperature' variable

We want the 'cold' qualifier to possibly refer to the values between -10 and 0, definitely refer to all the temperature values between 0

and 5 and possibly refer to the values between 5 and 12. This would be done by adding the following line to the definition:

```
cold, /-\, linear, [-10,5,10,15];
```

This defines the following 'cold' membership function for the fuzzy variable 'temperature':

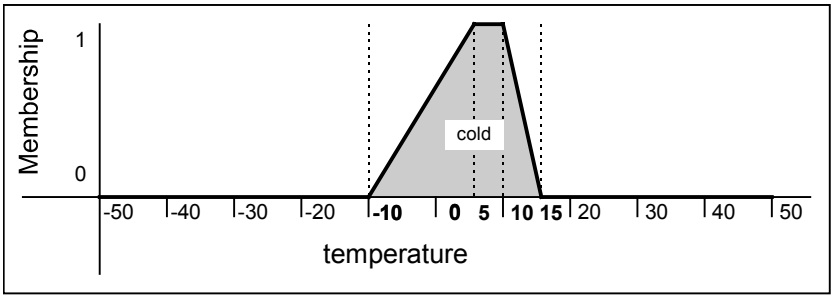


Figure 12 - the 'cold' qualifier for the 'temperature' variable

The 'warm' qualifier should possibly refer to the values between 5 and 15, definitely refer to all the temperature values between 15 and 25 and possibly to the values between 25 and 35. This would be done by adding the following line to the definition:

```
warm, /-\, linear, [5,15,25,35];
```

resulting in the following 'warm' membership function for the fuzzy variable 'temperature':

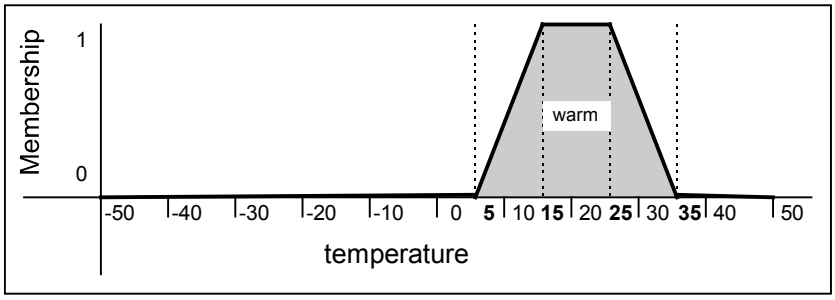


Figure 13 - the 'warm' qualifier for the 'temperature' variable

Finally we want the 'hot' qualifier to possibly refer to the values between 20 and 30 and definitely refer to all the temperature values above 30. This would be done by adding the following line to the definition:

```
hot, /, linear, [20,30];
```

This defines the following 'hot' membership function for the fuzzy variable 'temperature':

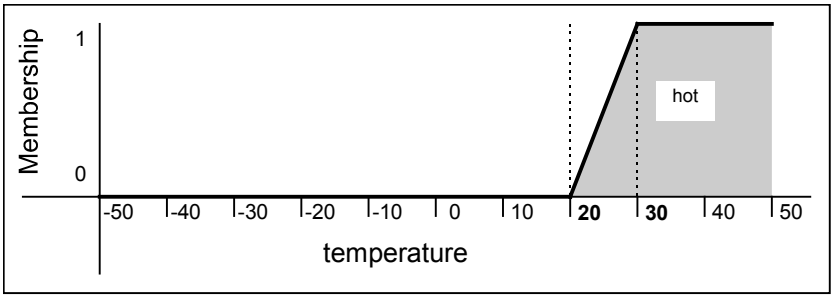


Figure 14 - the 'hot' qualifier for the 'temperature' variable

The fuzzy 'temperature' variable now has four qualifiers, 'freezing', 'cold', 'warm' and 'hot'. The domains for these qualifiers overlap as is shown in the following diagram which combines all the qualifiers in a single graph:

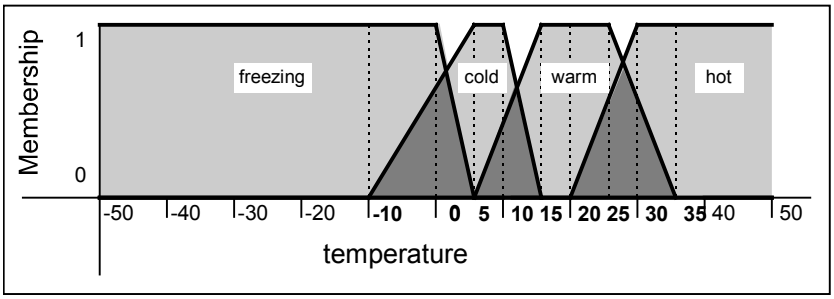


Figure 15 - the qualifiers for the 'temperature' variable

Having defined the qualifiers for the fuzzy variable, we then need to decide upon the method for de-fuzzifying the variable. In this example we will use the default method for de-fuzzification - the centroid method. The complete code follows:

```
fuzzy_variable(temperature):-
    [-50,50];
    freezing,       \,      linear, [0,6];
    cold,           /-\,    linear, [-10,5,10,15];
    warm,           /-\,    linear, [5,15,25,35];
    hot,            /,      linear, [20,30];
    centroid.
```

Defining Linguistic Hedges

A linguistic 'hedge' is used to either concentrate or dilute the characteristic of the membership function for a fuzzy variable qualifier. Hedges like 'extremely' or 'very' normally intensify the effect of the qualifier whilst their counterparts like 'slightly' or 'mildly' dilute the effect of the qualifier.

The Anatomy of a Linguistic Hedge

To define a linguistic 'hedge' you define its name and the formula to be used when the 'hedge' is applied. The Prolog anatomy of a linguistic 'hedge' definition is shown in the following diagram. The parts in bold indicate keywords and syntax recognised by the fuzzy interpreter. The plain text indicates the names and parameters that are chosen by the programmer.

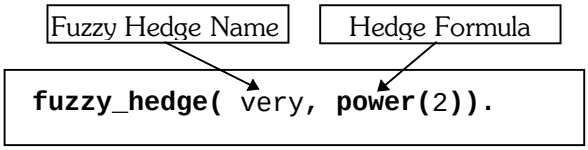


Figure 16 - The parts of a Prolog fuzzy 'hedge' definition

The following diagram shows the equivalent flex KSL code:

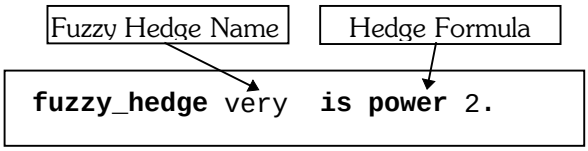


Figure 17 - The parts of a flex KSL fuzzy hedge definition

The Linguistic Hedge Name

The name of a linguistic 'hedge' is an atom to be used whenever the 'hedge' is applied. An example of a fuzzy 'hedge' could be 'slightly'. This 'hedge' could be defined by the following program:

```
fuzzy_hedge( slightly , power(0.5)).
```

Note that the 'slightly' hedge would then be 'universal' in that it could be applied to any qualifier.

The Linguistic Hedge Formula

This part of a linguistic 'hedge' program defines the formula to be applied whenever the 'hedge' is used. Hedges apply to the qualifiers of fuzzy variables and affect their membership functions.

Currently the only formula supported is that of 'power'. The value of the power must be between 0.1 and 9.9.

A Linguistic Hedge Example

The following example shows the 'very' hedge with a power value of 2.

```
fuzzy_hedge(very, power(2)).
```

In a fuzzy rule we could then apply this hedge to a fuzzy variable qualifier. For example, imagine a 'happy' qualifier for a 'mood' fuzzy variable where the higher the value the happier the 'mood' until you get to the value 80, above which the mood is definitely 'happy':

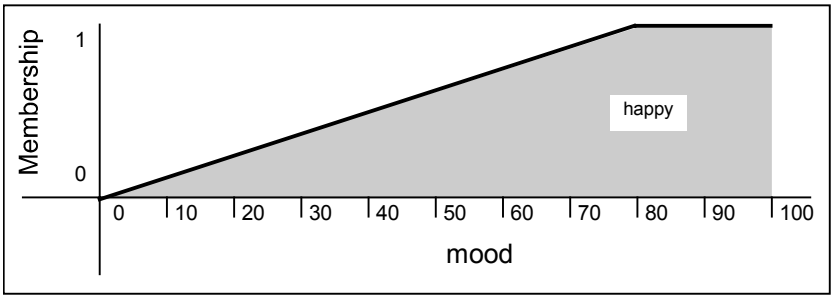


Figure 18 - the 'happy' qualifier for the mood variable

We could refer to this in a rule as follows:

```
if mood is very happy
```

This would then have the effect of exaggerating the 'happy' membership function. The resultant shape is shown in the following diagram. The original qualifier membership function is shown by a dotted line. To show how the membership function is affected let's take a degree of membership of 0.5. On the original function this would correspond to a 'crisp' value of 40. In the 'hedged' function this now corresponds to a 'crisp' value of around 70, biasing the function towards the definitely (or very) 'happy' end.

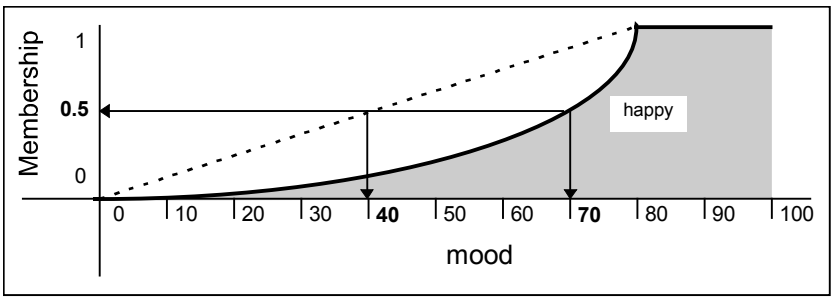


Figure 19 - the 'very happy' qualifier for the mood variable

The converse example shows the 'slightly' hedge with a power value of 0.2. This would be defined by the following fuzzy hedge program:

```
fuzzy_hedge(slightly, power(0.2)).
```

In a fuzzy rule we could then apply this hedge to the 'happy' qualifier for the 'mood' fuzzy variable. This would then have the effect of moderating the 'happy' membership function. The resultant shape is shown in the following diagram. Again the original qualifier membership function is shown by a dotted line. On the original function a degree of membership of 0.5 corresponds to a 'crisp' value of 40. In the 'hedged' function, this now corresponds to a 'crisp' value of around 10, biasing the function towards the definitely not (or slightly) 'happy' end.

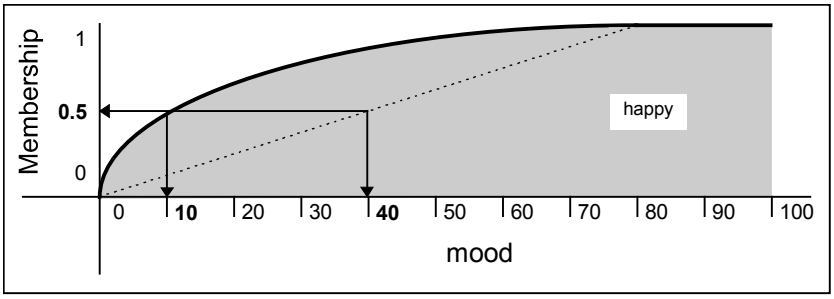


Figure 20 - the 'slightly happy' qualifier for the mood variable

Defining Fuzzy Rules

Another major component of a fuzzy logic system is the set of rules that reason about the qualifiers of fuzzy variables. Rules consist of a set of 'if' conditions and one 'then' conclusion and an optional 'else' conclusion.

The Anatomy of a Fuzzy Rule

The anatomy of a Prolog fuzzy rule definition is shown in the following diagram. The parts in bold indicate keywords and syntax recognised by the fuzzy logic toolkit. The plain text indicates the names and parameters that are chosen by the programmer. The conditions for the rule are highlighted with a dotted border.

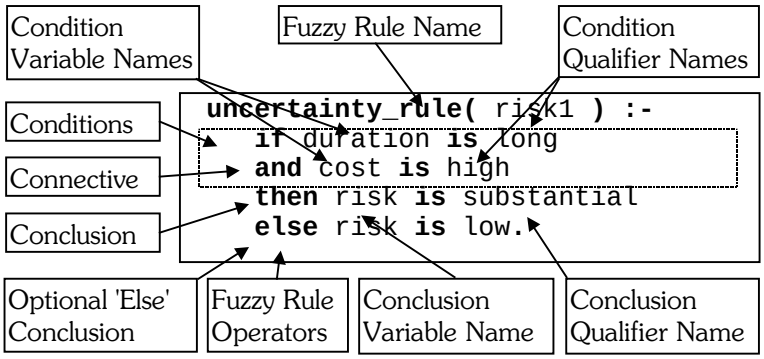


Figure 21 - The parts of a Prolog fuzzy rule definition

The following diagram shows the equivalent flex KSL code:

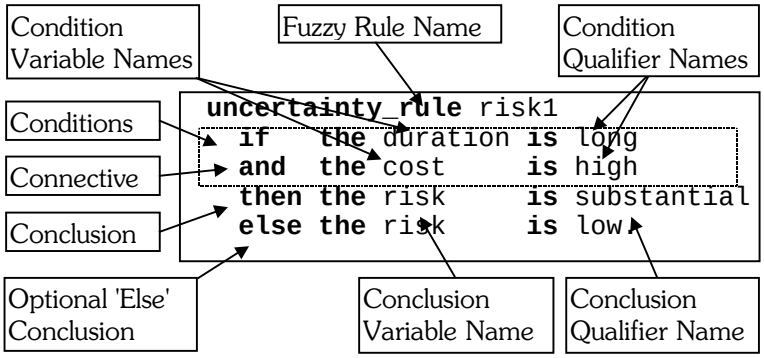


Figure 22 - The parts of a flex KSL fuzzy rule definition

The Fuzzy Rule Operators

The above diagram, and all of the fuzzy rule examples that follow, make use of certain operators. Any fuzzy logic program that implements rules according to the examples should include the following operator declarations:

```
:- op( 1150,  fy, ( if   ) ),
   op( 1150, xfy, ( then ) ),
   op( 1150, xfx, ( else ) ),
   op( 1100, xfy, ( or   ) ),
   op( 1000, xfy, ( and  ) ),
   op(  700, xfx, ( is   ) ),
   op(  600,  fy, ( not  ) ).
```

The Fuzzy Rule Name

The name of a fuzzy rule is an atom to be used whenever referring to the rule. An example of a fuzzy variable rule could be 'pressure_rule1'. This would be defined by a fuzzy rule program that started:

```
uncertainty_rule( pressure_rule1 ) :-
```

The Fuzzy Rule Conditions

The conditions of a fuzzy rule refer to fuzzy variables and their qualifiers. Multiple conditions are joined together using the connectives 'and', 'or' and 'not'. Depending on the type of connective applied to the conditions, the conclusion of the rule is calculated in different ways. The various methods of handling conjunctions, disjunctions and negations are dealt with in more detail in the description of the *fuzzy_propagate/4* predicate.

The following example conditions could be used in a fuzzy rule for assessing the risk of a project. The conditions refer to the fuzzy variables 'duration' and 'staffing' and their respective qualifiers 'long' and 'large'. The qualifier for the 'staffing' fuzzy variable is negated using 'not'. The two conditions are joined using 'and'.

```
if duration is long
and staffing is not large
```

The Fuzzy Rule 'Then' Conclusion

The value of the qualifier for the fuzzy variable mentioned in the 'then' part of a fuzzy rule is calculated using the current degrees of membership values for the fuzzy variable qualifiers mentioned in the conditions.

The following example concludes the degree of membership for the 'high' qualifier of the 'risk' fuzzy variable.

```
then risk is high
```

The Fuzzy Rule 'Else' Conclusion

The value of the qualifier for the fuzzy variable mentioned in the 'else' part of a fuzzy rule is found by taking 1 minus the value calculated for

the fuzzy variable qualifier mentioned in the 'then' part of the fuzzy rule. This section of a fuzzy rule is optional. The following example concludes the degree of membership for the 'low' qualifier of the 'risk' fuzzy variable.

```
else risk is low
```

Fuzzy Rule Examples

To illustrate fuzzy rules, the following examples are taken from a fuzzy program that shows how to increase or decrease the throttle on a steam turbine according to the current temperature and pressure inside the turbine.

The following rule is used to calculate a membership value for the 'positive_large' qualifier for the 'throttle' variable using the current membership values for the 'cold' qualifier of the 'temperature' variable and the 'weak' qualifier of the 'pressure' variable.

```
uncertainty_rule( throttle1 ) :-
    if temperature is cold
    and pressure is weak
    then throttle is positive_large .
```

The following rule calculates a membership value for the 'positive_medium' qualifier for the 'throttle' variable using the current membership values for the 'cold' qualifier of the 'temperature' variable and the 'low' qualifier of the 'pressure' variable.

```
uncertainty_rule( throttle2 ) :-
    if temperature is cold
    and pressure is low
    then throttle is positive_medium .
```

The third rule calculates a membership value for the 'positive_small' qualifier for the 'throttle' variable using the current membership values for the 'cold' qualifier of the 'temperature' variable and the 'ok' qualifier of the 'pressure' variable.

```
uncertainty_rule( throttle3 ) :-
    if temperature is cold
    and pressure is ok
    then throttle is positive_small .
```

Defining Fuzzy Matrices

The three rules shown in the previous section have the same structure (i.e. they all have conditions that refer only to qualifiers of the 'temperature' and 'pressure' fuzzy variables and draw a conclusion for a qualifier of the 'throttle' fuzzy variable). Also the conditions are all joined by conjunctions. When this is the case they, and all other rules of the same form in the program, can be combined together for convenience using a fuzzy rule matrix. This creates what is also known as a fuzzy associative memory.

The Anatomy of a Fuzzy Rule Matrix

To define a fuzzy rule matrix you specify the fuzzy variables to be used in all the lines of the matrix. Then you specify the qualifiers you are going to use with those variables. The anatomy of a Prolog fuzzy matrix definition is shown in the following diagram. The parts in bold indicate keywords and syntax recognised by the fuzzy logic interpreter. The plain text indicates the names and parameters that are chosen by the programmer. The variables of the matrix are highlighted with a dotted border.

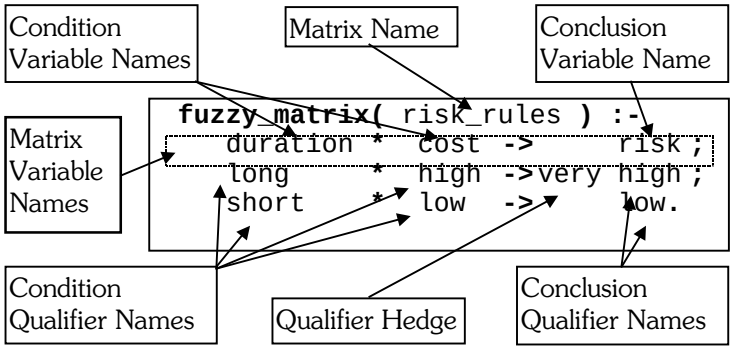


Figure 23 - The parts of a Prolog fuzzy matrix definition

The following diagram shows the equivalent flex KSL code:

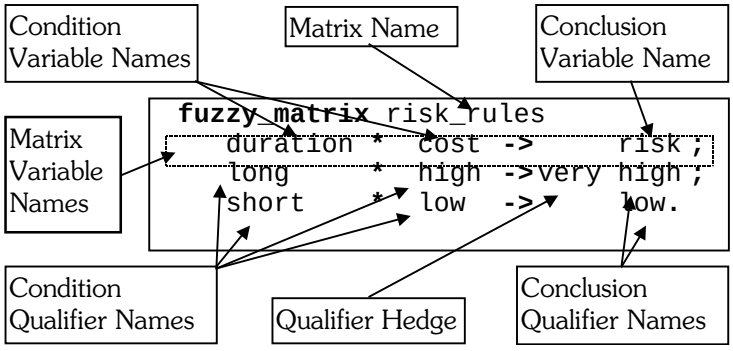


Figure 24 - The parts of a flex KSL fuzzy matrix definition

The Fuzzy Rule Matrix Name

The name of a fuzzy rule matrix is an atom to be used whenever referring to the matrix. An example of a fuzzy matrix could be 'throttle_rules'. This would be defined by a fuzzy matrix program that started:

```
fuzzy_matrix( throttle_rules ) :-
```

The Fuzzy Rule Matrix Variable Names

The first line in the body of the definition defines the fuzzy variables that will be used in all the rules of the matrix. The following example shows a fuzzy rule matrix that will apply to the fuzzy variables 'temperature', 'pressure' and ' throttle':

```
temperature     *     pressure     ->     throttle ;
```

This represents rules of the form:

```
if temperature is _
and pressure is _
then throttle is _
```

The underscores show that the qualifiers are not specified at this stage.

The Fuzzy Rule Matrix Qualifiers

All the remaining lines, following the first fuzzy variable declaration line in the body of the matrix definition, should contain qualifiers for each of the fuzzy variables in the same relative positions.

Continuing on from the previous example, the following example shows a rule that applies to the fuzzy variables 'temperature', 'pressure' and ' throttle':

```
    cold    *     weak   ->     positive_large;
```

This represents the rule:

```
if temperature is cold
and pressure is weak
then throttle is positive_large
```

Fuzzy Matrix Example

The following example completes the 'throttle_rules' fuzzy matrix program. Note that this is equivalent to the three 'throttle' rules defined in the *Fuzzy Rule Examples* section:

```
fuzzy_matrix( throttle_rules ):-
 temperature      *      pressure      ->      throttle ;

    cold          *      weak          ->      positive_large;
    cold          *      low           ->      positive_medium;
    cold          *      ok            ->      positive_small.
```

Chapter 5 - Fuzzy Variable Editor

The fuzzy variable editor is used to edit existing fuzzy variable definitions and to generate new definitions. The dialog is initiated from a fuzzy editor menu option.

The Fuzzy Variable Editor Menu Item

"Run/Fuzzy Editor"

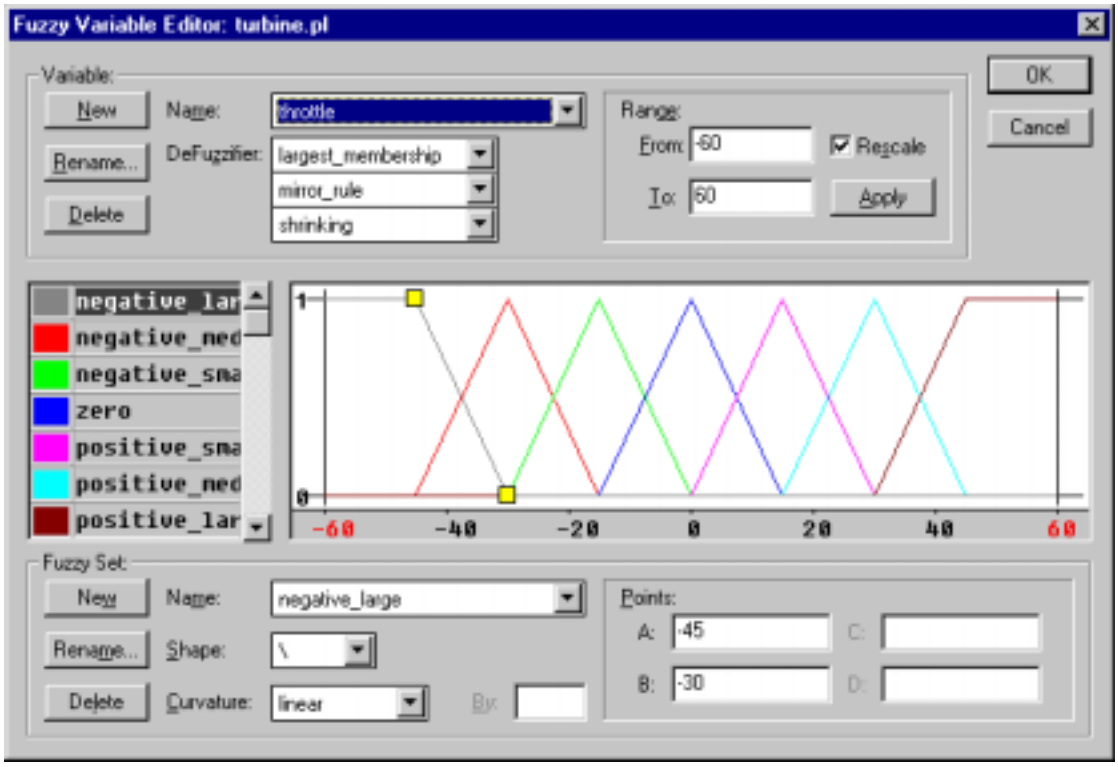
Clicking on the "Run/Fuzzy Editor" menu option invokes the fuzzy variable editor.

The Fuzzy Variable Editor Dialog

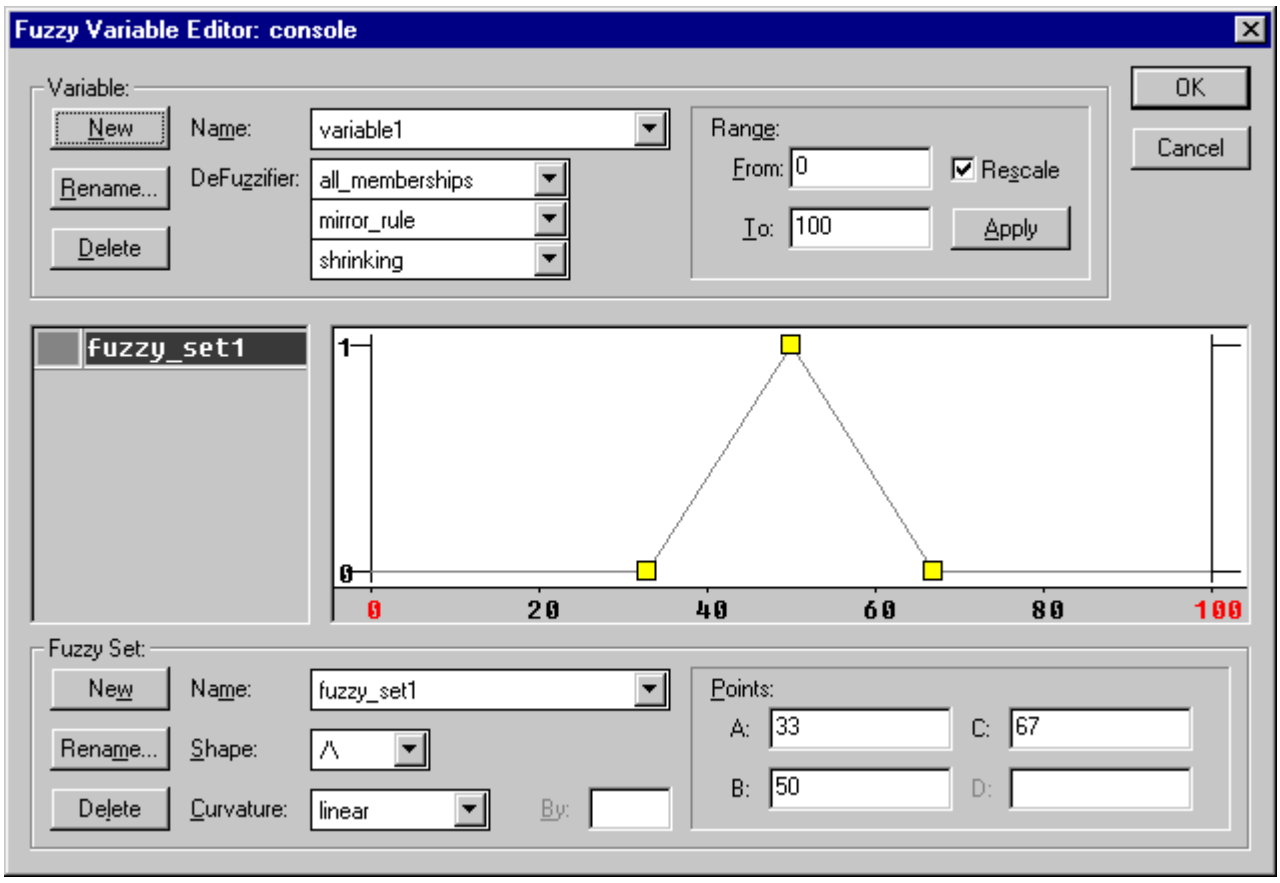
The Fuzzy Variable Editor dialog is modal.

Input

The editor works on the edit window that was in focus when the "Fuzzy Editor" menu option was selected. This window is the origin for the fuzzy variable definitions. The text of the fuzzy variables is interpreted by the fuzzy variable editor and the start and end points of the definitions stored. The fuzzy variable editor is pre-filled with the first definition found in the edit window. If the edit window does not contain any fuzzy variable definitions a new fuzzy variable is created in the fuzzy variable editor. If TURBINE.PL was in focus when the fuzzy editor was invoked, the Fuzzy Variable Editor dialog would look like:



If the console is in focus when the fuzzy editor is invoked, a new fuzzy variable is created in the fuzzy variable editor:



When the dialog is accepted by clicking on the 'OK' button, the user is given the option of creating a new file to place the definition for the new fuzzy variable(s). If this option is rejected, the dialog behaves as if the dialog was cancelled.

Output

When the "OK" button is selected, the text for all the fuzzy variables from the edit window that have been changed in the editor will be replaced with their changed definitions. Any new fuzzy variable definitions are appended to the end of the edit window.

The dialog contains three sections:

The Variable Section

This section contains the details of the fuzzy variable(s).

The "New" Button

This "New" button is used to create a new fuzzy variable.

An assumption is made about its initial name, range and de-fuzzifier. Selecting this option will side-affect the entire dialog.

The "Rename" Button

This "Rename" button allows you to rename the currently selected fuzzy variable.

The "Delete" Button

This "Delete" button allows you to delete, without warning, the currently selected fuzzy variable.

The Name: Field

This Name: field is a combobox showing the name of the fuzzy variable currently being edited. The list box component contains the fuzzy variables defined in the edit window that was in focus when the Fuzzy Editor menu option was selected and any new fuzzy variable definitions created using the "New" button.

Selecting a new name side-affects the entire dialog.

DeFuzzifier:

The DeFuzzifer: section consists of three comboboxes indicating the de-fuzzifier to be used for the fuzzy variable currently being edited.

Users can also specify their own de-fuzzifying expression.

The Range:, From: and To: Fields

The Range: subsection consists of two edit fields specifying the range of the fuzzy variable currently being edited. Changing the range side-affects the range in the Graphic Editor Window.

Graphic Editor Section

This section graphically displays the parameters and qualifiers for the fuzzy variable, allowing the shape and position of the qualifiers to be edited. It consists of two components:

The Key Panel

This shows the names and colours of the qualifiers in the Graphic Editor window. Chosing a name makes that qualifier the selection in the Graphic Editor window and propagates its values in the Fuzzy Set section.

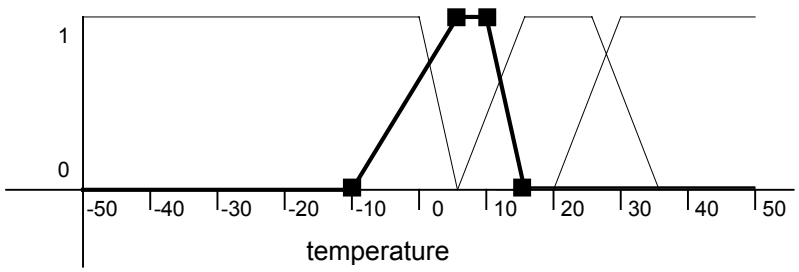
The Graphic Editor Window

This window contains a graph showing the currently selected fuzzy variable's range along the X axis, and truth values between 0 and 1 along the Y axis. Plotted on this graph are the membership functions for all the qualifiers for the fuzzy variable.

The selected membership function has "handles" at its significant points.

A qualifier can be selected by left-clicking on the lines of the membership function. This side-affects the Key section and the Fuzzy Set section.

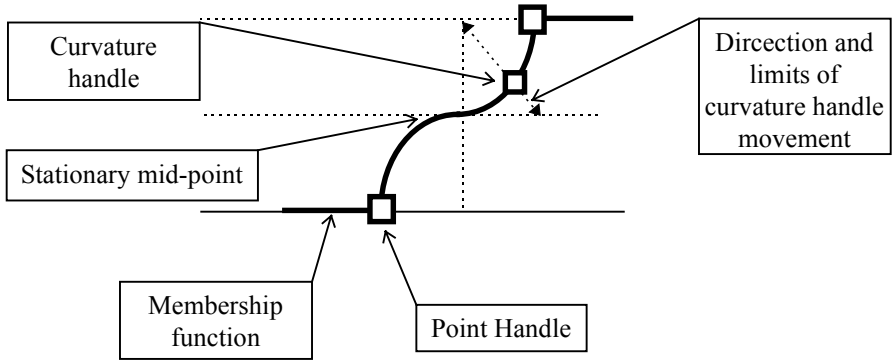
The membership function for the selected qualifier is highlighted using thicker lines and handles. When a qualifier is selected all other qualifiers are de-selected. The following diagram shows the representation for the fuzzy variable, where the "cold" qualifier is selected:



A selected qualifier's points can be moved by left-clicking and dragging the appropriate handle. This side-affects the appropriate point in the Fuzzy Set section's points sub-section.

When changing the location of a point for one of the six basic shapes, the Y co-ordinate remains fixed and the X co-ordinate is constrained between the selected membership function points immediately before and after its original location and the lower and upper bounds for the fuzzy variable.

When a qualifier's curvature is set to "curved", additional handles appear on the membership function, occurring at the upper quarter point of the slopes of the function. These handles move perpendicular to the slope of the curve at these points and affect the degree of curvature, as shown in the following diagram:



Fuzzy Set Section

This section allows new fuzzy sets to be created for the current fuzzy variable. This section contains a "points" sub-section.

The "New" Button

This "New" button is used to create a new qualifier for the current fuzzy variable. It is assigned an initial name (i.e. "fuzzy_set#"), shape (i.e. /\), curvature (i.e. linear), point locations (i.e. 33,50,67) and colour.

The "Rename" Button

This "Rename" button allows you to rename the currently selected fuzzy variable qualifier.

The "Delete" Button

This "Delete" button allows you to delete, without warning, the currently selected fuzzy variable qualifier.

The Name: Field

This combobox shows the name of the currently selected fuzzy variable qualifier. The listbox component contains all the qualifiers for the current fuzzy variable. Selecting a new name changes the currently selected qualifier and side effect all the fields in the fuzzy

variable qualifier section as well as showing the change of qualifier selection in the Graphic Editor Window.

Editing the name changes the name of the currently selected qualifier.

The Shape: Field

A combo-box showing the shape of the currently selected fuzzy variable qualifier. The listbox component contains all the available built-in shapes: "\", "/", "\-/", "/-\", "\/", "/\".

Changing the shape of a qualifier adds or removes points to the qualifier membership function as appropriate, so some assumption is made as to the co-ordinates for the new points.

The freehand shape, "f", is not supported by the Fuzzy Variable Editor.

The Curvature: Field

A combo-box showing the curvature of the currently selected fuzzy variable qualifier. The listbox component contains the available curvatures: "linear" and "curved". Changing the curvature of a qualifier side-affects the qualifier's membership function in the Graphic Editor Window. An assumption of an initial curvature of "2" is made to give the often-used bell curve shape.

The By: Field

When the "curved" curvature is selected this edit field specifies the degree of curvature. If the "linear" curvature is selected this edit field is disabled.

Changing this value side-affects the display of the qualifier's membership function in the Graphic Editor Window.

Points Section:

This section shows the values for the points on the membership function for the currently selected fuzzy variable qualifier.

For the six basic shapes, there are four points shown, "A", "B", "C" and "D", each of which is portrayed by two edit fields indicating the X and Y co-ordinates. The built-in shapes may have two, three or four points; when the points are not applicable for a particular shape they will be disabled.

The Y component of the points is fixed for the six built-in shapes, being either 1 or 0. The X component is editable and side effects the graphical representation of the currently selected qualifier membership function.

When changing the location of a point the Y co-ordinate remains fixed and the X co-ordinate is constrained between the points immediately before and after its original location (using the lower and upper bounds as start and end points for the fuzzy variable).

Dialog Buttons

The Fuzzy Variable Editor dialog has two main buttons - OK and Cancel.

OK

Clicking the OK button accepts any editing done and updates the fuzzy variable code in the edit window that was in focus when the editor was invoked.

If the Console window was in focus, the option of creating a new source window is given with "Yes", "No" and "Cancel" buttons. If "Yes" is selected a new window is created and the edited fuzzy variable definitions are appended to this new window. If "No" is selected the dialog behaves as if the "Cancel" button had been pressed. If "Cancel" is selected the Fuzzy Variable Editor dialog will not terminate.

Cancel

Clicking the Cancel button terminates any editing. If changes have been made, a warning is given: "Some fuzzy variables have been changed. Do you want to save changes?" with "Yes", "No" and "Cancel" buttons. If "Yes" is selected the fuzzy variables in the window currently in focus will be updated. If "No" is selected the Fuzzy Variable Editor dialog will terminate and no output will be produced. Clicking "Cancel" will return to the Fuzzy Variable Editor dialog.

Chapter 6 - Flint Predicates

---

defuzzify/2

de-fuzzify the value of a fuzzy variable by aggregating the membership values of its qualifiers using the de-fuzzification method defined for that variable.

Syntax defuzzify(*Variable*, *Value*)

+*Variable*	The name of a fuzzy variable	<atom>
?*Value*	Number in range of fuzzy variable	<number>

Comments

Examples

?- **defuzzify(brake_force, V).** *<return>*
V = 100

defuzzify/3

de-fuzzify the value of a fuzzy variable by aggregating the membership values of its qualifiers using the specified de-fuzzification method.

Syntax defuzzify(*Variable*, *Value, Method*)

+*Variable*	The name of a fuzzy variable <atom>
?*Value*	Number in range of fuzzy variable <number>
+*Method*	De-fuzzification method <term> one of { E (explicit arithmetic expression over the qualifiers) or Centroid(M1,M2,M3) (centroid of membership values) } where…
	M1 is one of:
	all_memberships (consider only non-zero membership values (aggregation)), *largest_membership* (consider only the largest_membership membership values (winner(s) takes all))
	M2 is one of:
	mirror_rule (de-fuzzify individual membership values at the extremeties as if they are isosceles triangles. *bounded_range* (de-fuzzify individual membership values at the bounded_range point which balances the modified shape post truncation/shrinking), *inverse* (de-fuzzify individual membership values by their inverse function) } and
	M3 is one of:
	shrinking (shrinking down to membership value whilst retaining the shape of the membership function) *truncation* (chop off membership shapes at the point of membership value thus creating a plateau) }.

Comments

Examples

?- **defuzzify(brake_force, V, ((low + 2*medium + high) / 4)).** *<return>*
V = 0

?- **defuzzify(brake_force, V, centroid(all_memberships, mirror_rule, shrinking)).** *<return>*
V = 100

?- **defuzzify(brake_force, V, centroid(all_memberships, bounded_range, shrinking)).**
<return>
V = 92.6776695296637

---

fuzzify/2

Fuzzify the value of a fuzzy variable by instantiating the membership values of its qualifiers.

Syntax fuzzify(*Variable*, *Value*)

+Variable	The name of a fuzzy variable	<atom>
+Value	Number in range of fuzzy variable	<number>

Comments

Examples

?- **fuzzify(buffer_distance, 1000).** *<return>*
yes

?- **fuzzify(train_speed, 150).** *<return>*
yes

fuzzy_propagate/1

propagate the degrees of membership values of fuzzy variable qualifiers using the specified rule agenda

Syntax *fuzzy_propagate(Agenda)*

+Agenda	list of <atom>

Comments The predicate *fuzzy_propagate/1* is used to apply a list of fuzzy rules, whose names are given in the *Agenda* argument, to the current fuzzy variable state by propagating changes to the membership values for the fuzzy variable qualifiers. This is done using the default methods for handling conjunctions, disjunctions and negations in the fuzzy rules. The *Agenda* argument may also be the name of a fuzzy rule matrix.

This predicate is defined by the program:

```
fuzzy_propagate( Agenda ) :-
  fuzzy_propagate(minimum, maximum, complement, Agenda).
```

Examples The following call to *fuzzy_propagate/1* takes the fuzzy rules 't1', 't2' and 't3' and applies them to the current fuzzy variable state. The propagation is done using the 'minimum' method for handling conjunctions, the 'maximum' method for handling disjunctions and the 'complement' method for handling negations (see *fuzzy_propagate/4*):

```
?- fuzzy_propagate([t1,t2,t3]). <return>
yes
```

fuzzy_propagate**/4**

propagate the degrees of membership values of fuzzy variable qualifiers using the specified combinators and rule agenda

Syntax *fuzzy_propagate(Conjunction, Disjunction, Negation, Agenda)*

+Conjunction	<atom> in the domain {minimum,product,truncate}
+Disjunction	<atom> in the domain {maximum,addition,strengthen}
+Negation	<atom> in the domain {complement}
+Agenda	list of <atom>

Comments The predicate *fuzzy_propagate/4* is used to apply a list of fuzzy rules, whose names are given in the *Agenda* argument, to the current fuzzy variable state by propagating changes to the membership values for the fuzzy variable qualifiers. This is done using the specified methods for handling conjunctions, disjunctions and negations in the fuzzy rules. The *Agenda* argument may also be the name of a fuzzy rule matrix.

The following table shows all the different methods available for calculating membership values, where P and Q are expressions and X_P is the degree of membership of P and X_Q is the degree of membership of Q.

Example

```
?- fuzzy_propagate( minimum, maximum, complement, [t1,t2,t3] ). <return>
yes
```

Expression	Method	Description
P and Q	*minimum*	$\min(X_P,X_Q)$
	product	$X_P * X_Q$
	truncate	$\max((X_P + X_Q - 1), 0)$
P or Q	*maximum*	$\max(X_P,X_Q)$
	strengthen	$X_P + X_Q * (1-X_P)$
	addition	$\min(X_P+X_Q,1)$
not P	*complement*	$1-X_P$

Table 10 - Methods for calculating membership values

For example, given the fuzzy rule:

```
uncertainty_rule( throttle1 ) :-
    if temperature is cold
    and pressure is weak
    then throttle is positive_large .
```

let's assume the degree of membership for the 'cold' qualifier of the 'temperature' fuzzy variable is 0.5 and the degree of membership for the 'weak' qualifier of the 'pressure' fuzzy variable is 0.25.

The resultant degree of membership for 'positive_large' of 'throttle' using the 'minimum' method is:

```
min(0.5,0.25)
```

which gives the value 0.25. In contrast, the resultant degree of membership for 'positive_large' of 'throttle' using the 'product' method is:

```
0.5 * 0.25
```

which this time gives the value 0.125. Finally the resultant degree of membership for 'positive_large' of 'throttle' using the 'truncate' method is:

```
max((0.5 + 0.25 - 1),0)
```

which gives the value 0.

Examples The following call to *fuzzy_propagate/4* takes the fuzzy rules 't1', 't2' and 't3' and applies them to the current fuzzy variable state. The propagation is done using the 'minimum' method for handling conjunctions, the 'strengthen' method for handling disjunctions and the 'complement' method for handling negations:

```
?- fuzzy_propagate( minimum, strengthen, complement, [t1,t2,t3] ). <return>
yes
```

The following call to *fuzzy_propagate/4* uses the fuzzy rule matrix 'throttle_rules' and applies them to the current fuzzy variable state. The propagation is done using the 'product' method for handling conjunctions, the 'sum' method for handling disjunctions and the 'complement' method for handling negations:

```
?- fuzzy_propagate( product, sum, complement, [throttle_rules] ). <return>
yes
```

fuzzy_reset_membership**/0**

reset the degrees of membership of all fuzzy variable qualifiers to zero

Comments The *fuzzy_reset_membership/0* predicate is used to reset an initial state by setting the degrees of membership for all the currently defined qualifiers attached to all the currently defined fuzzy variables to zero.

Examples Regardless of the number of fuzzy variables currently defined, the following call will reset all of the degrees of membership for all of their qualifiers to zero:

```
?- fuzzy_reset_membership. <return>
yes
```

fuzzy_reset_membership**/1**

set the degrees of membership for a fuzzy variable's qualifiers to zero

Syntax fuzzy_reset_membership(*Variable*)

+*Variable*	<atom>

Comments The *fuzzy_reset_membership/1* predicate is used to reset the initial state for a named fuzzy variable by setting the degrees of membership for all its qualifiers to zero.

Examples The following call resets all of the degrees of membership for all of the qualifiers of the 'temperature' fuzzy variable to zero:

```
?- fuzzy_reset_membership( temperature). <return>
yes
```

fuzzy_variable_value**/2**

get or set the value for a fuzzy variable

Syntax fuzzy_variable_value(*Variable*, *Value*)

+*Variable*	<atom>
?*Value*	<number>

Comments The *fuzzy_variable_value/2* predicate is used to assign or retrieve a value for a named fuzzy variable. If the *Value* argument is 'crisp' then it is converted into degrees of membership for all the fuzzy variable's qualifiers using each qualifier's membership function, specified when the qualifier was defined (see *Defining Fuzzy Variable Qualifiers* on page *13*).

If the *Value* argument is an unbound variable, the de-fuzzification expression for the fuzzy variable (see *Defining Fuzzy Variable De-Fuzzifiers* on page *19*) is used to produce a single 'crisp' value which is returned in the *Value* argument. Note that because of the difference between the fuzzification and de-fuzzification processes, you are not guaranteed to get back the same 'crisp' value you put in.

Examples The following example sets the value for the fuzzy 'temperature' variable to 15.5. This results in various degrees of membership being assigned to its qualifiers:

```
?- fuzzy_variable_value( temperature, 15.5 ). <return>
yes
```

The following example returns a value for the fuzzy 'temperature' variable. This is done using the 'temperature' variable's de-fuzzification expression:

```
?- fuzzy_variable_value( temperature, TempValue ).
```

---

uncertainty_dynamics/0

initialise the fuzzy sub-system by removing all fuzzy variables and rules

Comments The *uncertainty_dynamics/0* predicate is used to reset an initial state by removing all the fuzzy variables and rules that have been created. The *uncertainty_dynamics/0* predicate declares the following dynamic predicates: *uncertainty_rule/1*, *uncertainty_rule/3*, *uncertainty_data/4*, *fuzzy_variable/1*, *fuzzy_variable_qualifier/5*, *fuzzy_variable_range/3* and *fuzzy_variable_defuzzifier/2*.

Examples The following goal removes all the fuzzy rules and variables:

```
?- uncertainty_dynamics. <return>
yes
```

uncertainty_listing**/0**

lists all dynamic predicates representing fuzzy variables and rules

Comments The *uncertainty_listing/0* predicate is used to show the code for all the currently defined fuzzy variables and rules.

Examples This example assumes that the following fuzzy variable definition has been compiled:

```
fuzzy_variable( temperature ) :-
        [ 0 , 500 ] ;
        cold,           \ ,     linear, [ 110 , 165     ] ;
        cool,           /\,     linear, [ 110 , 165 , 220 ] ;
        normal,         /\,     linear, [ 165 , 220 , 275 ] ;
        warm,           /\,     linear, [ 220 , 275 , 330 ] ;
        hot,            /,      linear, [     275 , 330 ] .
```

A call to *uncertainty_listing/0* shows the current state of the fuzzy sub-system, showing the fuzzy facts that support the definition:

```
?- uncertainty_listing. <return>

/* fuzzy_variable/1 */

fuzzy_variable(temperature).

/* fuzzy_variable_range/3 */

fuzzy_variable_range(temperature,0,500).

/* fuzzy_variable_defuzzifier/2 */

fuzzy_variable_defuzzifier(temperature,A):-
 fuzzy_variable_centroid(temperature,A).

/* fuzzy_variable_qualifier/5 */

fuzzy_variable_qualifier(temperature,cold,\,linear,[110,165]).
fuzzy_variable_qualifier(temperature,cool,/\,linear,[110,165,220]).
fuzzy_variable_qualifier(temperature,normal,/\,linear,[165,220,275]).
fuzzy_variable_qualifier(temperature,warm,/\,linear,[220,275,330]).
fuzzy_variable_qualifier(temperature,hot,/,linear,[275,330]).

?- uncertainty_listing. <return>

% uncertainty_rule/1

uncertainty_rule( r3_1 ).

uncertainty_rule( r3_2 ).
```

```
uncertainty_rule( r3_3 ).

uncertainty_rule( r3_4 ).

% uncertainty_rule/2

uncertainty_rule( r3_1, A ) :-
  uncertainty_value_get( A, release_valve, stuck, B ),
  uncertainty_norm( A, then(1), [B], C ),
  uncertainty_value_set( A, release_valve, need_cleaning, C ),
  true.

uncertainty_rule( r3_2, A ) :-
  uncertainty_value_get( A, warning_light, on, B ),
  uncertainty_norm( A, affirms_denies(positive,2.2,0.2,_), [B], C ),
  uncertainty_norm( A, then(0.2), [C], D ),
  uncertainty_value_set( A, release_valve, stuck, D ),
  true.

uncertainty_rule( r3_3, A ) :-
  uncertainty_value_get( A, pressure, high, B ),
  uncertainty_norm( A, affirms_denies(positive,85,0.15,_), [B], C ),
  uncertainty_norm( A, then(0.9), [C], D ),
  uncertainty_value_set( A, release_valve, stuck, D ),
  true.

uncertainty_rule( r3_4, A ) :-
  uncertainty_value_get( A, temperature, high, B ),
  uncertainty_norm( A, affirms_denies(positive,18,0.11,_), [B], C ),
  uncertainty_value_get( A, water_level, low, D ),
  uncertainty_norm( A, affirms_denies(negative,1.9,0.1,_), [D], E ),
  uncertainty_norm( A, (and), [C,E], F ),
  uncertainty_norm( A, then(0.5), [F], G ),
  uncertainty_value_set( A, pressure, high, G ),
  true.

yes
```

---

uncertainty_notrace/0

Stop tracing the propagation of uncertainty values.

Example

```
?- uncertainty_notrace. <return>
yes
```

uncertainty_operators**/0**

Comments The *uncertainty_operators/0* predicate declares the following operators: if, then, else, with, or, and, is, are, not, affirms, denies, @, the and certainty_factor.

Example

?- **uncertainty_operators.** *<return>*
yes

uncertainty_propagate**/2**

Propagate uncertainty values via the specified rules.

Syntax *uncertainty_propagate(Measure, Rules)*

+Measure	An uncertainty measure <atom> being one of {probability, odds, certainty_factor, fuzzy}
+Rules	<list> of named rules or rule matrix

Examples

?- **uncertainty_propagate(fuzzy, [train1, train2, train3, train4, train5]).** *<return>*
yes

?- **uncertainty_propagate(odds, boiler_matrix).** *<return>*
yes

uncertainty_trace**/0**

Trace the propagation of uncertainty values.

Examples

```
boiler :-
  uncertainty_trace,
  boiler( probability,      0,  1,     1,     0.099, 0.020, _P6  ), nl, nl,
  boiler( odds,             0,  65535, 65535, 0.110, 0.020, _06  ), nl, nl,
  boiler( certainty_factor, -1, 1,     1,     0,     0,     _CF6 ), nl, nl.
```

```
?- boiler. <return>
Prob. : UPDATE    : (water_level is low) = 0
Prob. : UPDATE    : (warning_light is on) = 1
Prob. : UPDATE    : (temperature is high) = 1
Prob. : UPDATE    : (pressure is high) = 0.099
Prob. : UPDATE    : (release_valve is stuck) = 0.02
Prob. : TRY       : r3_4
Prob. : LOOKUP    : (temperature is high) = 1
Prob. : AFFIRMS   : weight(18) @ 1 -> 0.947368421052632
Prob. : LOOKUP    : (water_level is low) = 0
Prob. : DENIES    : weight(1.9) @ 0 -> 0.655172413793103
Prob. : AND       : 0.947368421052632 + 0.655172413793103 -> 0.971590909090909
Prob. : LOOKUP    : (pressure is high) = 0.099
Prob. : CONFIRMS  : 0.099 + 0.971590909090909 -> 0.789819912288887
Prob. : UPDATE    : (pressure is high) = 0.789819912288887
Prob. : FIRED     : r3_4
yes
```

---

uncertainty_value_get/4

Get the uncertainty value for a qualifier of a variable.

Syntax *uncertainty_value_get(Measure, Variable, Qualifier, Value)*

+Measure	An uncertainty measure <atom> being one of {probability, odds, certainty_factor, fuzzy}
+Variable	The name <atom> of an uncertainty variable
+Qualifier	The name <atom> of a qualifier for that variable
?Value	The numeric value <number> for the qualifier

Examples

```
?- uncertainty_value_get( fuzzy, brake_force, high, V ). <return>
V = 0

?- uncertainty_value_get( probability, pressure, low, V ). <return>
V = 0
```

---

uncertainty_value_reset/1

Reset the uncertainty value for all variables

Syntax *uncertainty_value_reset(Measure)*

+Measure	An uncertainty measure <atom> being one of {probability, odds, certainty_factor or fuzzy}

Examples

?- **uncertainty_value_reset(fuzzy).** *<return>*
yes

?- **uncertainty_value_reset(certainty_factor).** *<return>*
yes

---

uncertainty_value_reset/2

Reset the uncertainty values for all qualifiers of a variable

Syntax *uncertainty_value_reset(Measure, Variable)*

+Measure	An uncertainty measure <atom> being one of {probability, odds, certainty_factor or fuzzy}
+Variable	*The name <atom> of an uncertainty variable*

Examples

?- **uncertainty_value_reset(fuzzy, brake_force).** *<return>*
yes

?- **uncertainty_value_reset(certainty_factor, pressure).** *<return>*
yes

uncertainty_value_set/4

Set the uncertainty value for a qualifier of a variable

Syntax *uncertainty_value_set(Measure, Variable, Qualifier, Value)*

+*Measure*	An uncertainty measure <atom> being one of {probability, odds, certainty_factor, fuzzy}
+Variable	The name <atom> of an uncertainty variable
+Qualifier	The name <atom> of a qualifier for that variable
+Value	The numeric value <number> for the qualifier

Examples

?- **uncertainty_value_set(fuzzy, brake_force, high, 1).** *<return>*
yes

?- **uncertainty_value_set(certainty_factor,pressure,low,-1).** *<return>*
yes

Chapter 7 - Fuzzy Logic - A Worked Prolog Example

In this section we will implement a fuzzy logic controller. In the general model the device being controlled has a set of activators that take input and use this to affect the settings of the device and a set of sensors that get information from the device. The fuzzy logic controller takes the 'crisp' information from the sensors and fuzzifies the input into some fuzzy variables, propagates membership values using the fuzzy rules and finally de-fuzzifies the output and returns these 'crisp' values to the activators. The structure of the fuzzy logic controller is shown in *Figure 25*.

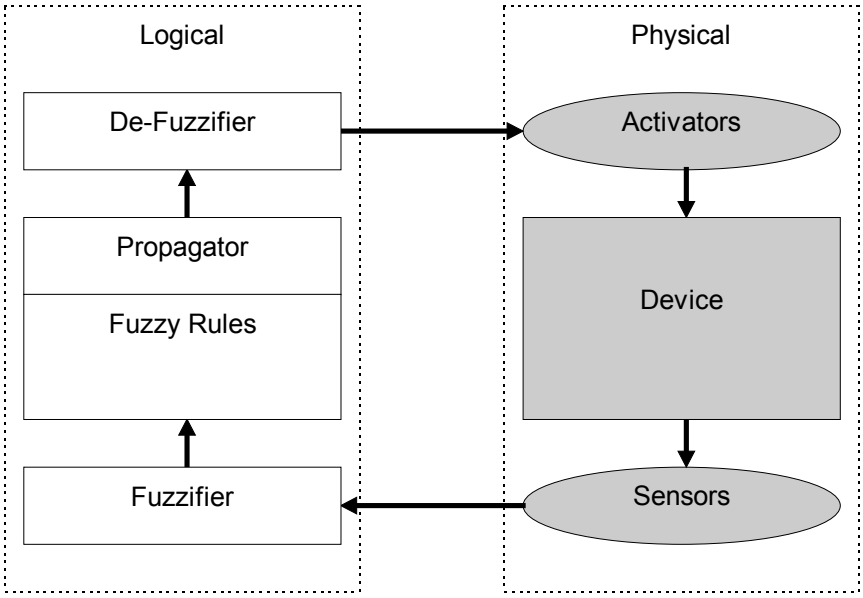


Figure 25 - A Fuzzy Logic Controller

A Fuzzy Turbine Controller

The controller we will implement will use a fuzzy set of rules to adjust the throttle of a steam turbine according to its current temperature and pressure to keep it running smoothly. The structure of the turbine controller is shown in *Figure 26*.

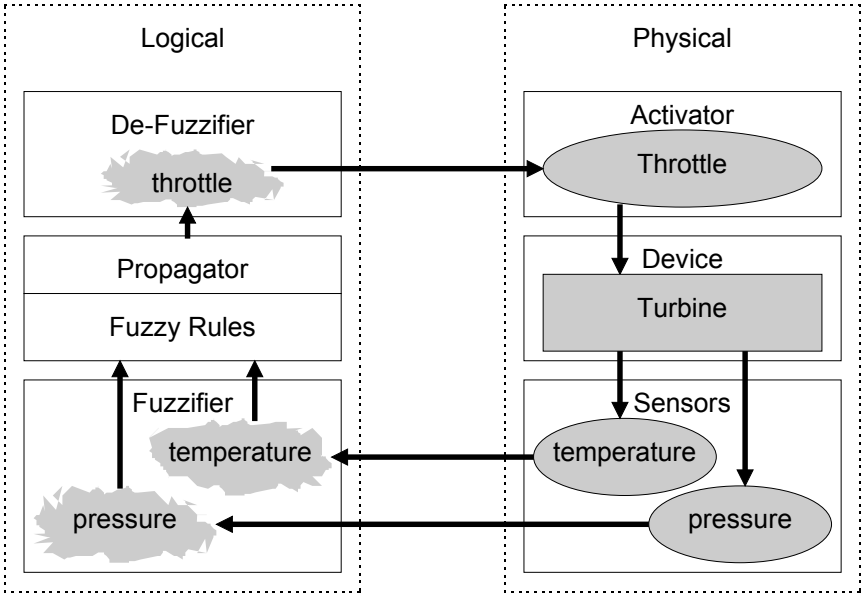


Figure 26 - A Fuzzy Turbine Controller

Defining Fuzzy Operators

Before coding the example we need to define the operators that are going to be used in defining the fuzzy rules. These are the words that are used to define conditions, conclusions, conjunctions, disjunctions and negations in fuzzy rules.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Linguistic operators %
%%%%%%%%%%%%%%%%%%%%%%%%%%%

:- op( 1150,  fy, ( if   ) ),
   op( 1150, xfy, ( then ) ),
   op( 1150, xfx, ( else ) ),
   op( 1100, xfy, ( or   ) ),
   op( 1000, xfy, ( and  ) ),
   op(  700, xfx, ( is   ) ),
   op(  600,  fy, ( not  ) ).
```

Defining Fuzzy Variables

First we need to decide upon the fuzzy variables and qualifiers that are going to be used in the fuzzy program. We do this first by looking at the quantities that will be used by the fuzzy rules, in this case 'temperature', 'pressure' and 'throttle'. We also know that we want to work out values for the 'throttle' using the values for the 'temperature' and 'pressure' so we can guess that 'temperature' and 'pressure' will be input variables and 'throttle' will be an output variable. We will define a fuzzy variable for each of these.

Then we need to know what sorts of words the fuzzy rules are going to use to describe each fuzzy variable. These words may exist in 'rules of thumb' already used by turbine operators or we may need to decide upon them ourselves. These words become the linguistic qualifiers of the fuzzy variable.

Let's look at the 'temperature' fuzzy variable first. The definition of a fuzzy variable may also include a definition of the range of 'crisp' values it may take. The range of temperatures possible for a turbine may broadly be defined as being between 0 and 500 degrees Celsius, though the significant values lie between 100 and 350. The start of our fuzzy 'temperature' variable will be:

```
fuzzy_variable( temperature ) :-
    [ 0 , 500 ] ;
```

The temperature of the turbine is normally described using the adjectives: cold, cool, normal, warm and hot. We need to define a qualifier for the 'temperature' fuzzy variable for each of these.

The easiest way to implement a rough guide to any qualifier is to define it using a linear shape so that the lines between the points are always straight and not curved. Refinements to the membership function can be introduced, if needed, at a later stage.

We can guess that the 'cold' qualifier in our example should refer to temperatures at the lower end of the fuzzy variable; this probably means it will use the downward slope shape. By talking to the steam turbine experts or making a rough estimate, we want the 'cold' qualifier to definitely refer to all the values below 110 and possibly to the values below 165 but not to the values above 165. The values between 110 and 165 should become steadily less 'cold' in a linear fashion. This could be done by adding the following definition to the 'temperature' fuzzy variable:

```
    cold,  \, linear, [ 110 , 165     ] ;
```

Notice that this 'cold' qualifier is notionally cold for temperatures in a steam turbine; other situations may require a completely different definition for 'cold'.

Next we want a 'cool' qualifier. We can guess that this qualifier is not going to apply to the extremes of the temperature range and that it is a positive qualifier (as opposed to 'uncool'), so it will probably be an upwards pointing triangle or an upwards pointing trapezoid. Again by examining the real situation, talking to the experts or estimating, we decide that 'cool' refers to values between 110 and 220, reaching a definite peak at 165. The values between 110 and 165 should become more 'cool' and the values between 165 and 220 should

become less 'cool' in a linear fashion. This can be added to our 'temperature' fuzzy variable using the following 'cool' definition:

```
cool, /\, linear, [ 110 , 165 , 220 ] ;
```

In this fashion we can continue to add the other qualifiers for the 'temperature' fuzzy variable ending up with the following definition:

```
fuzzy_variable( temperature ) :-
    [ 0 , 500 ] ;
    cold,   \ , linear, [ 110 , 165     ] ;
    cool,   /\, linear, [ 110 , 165 , 220 ] ;
    normal, /\, linear, [ 165 , 220 , 275 ] ;
    warm,   /\, linear, [ 220 , 275 , 330 ] ;
    hot,    /, linear, [     275 , 330 ] .
```

The following diagram shows the membership functions for all the 'temperature' qualifiers combined on a single graph:

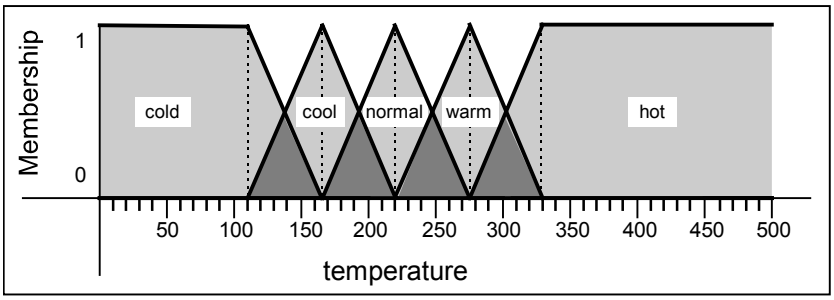


Figure 27 - the qualifiers for the 'temperature' variable

The other input fuzzy variable, 'pressure', can be defined in a similar way. The pressure of the turbine is normally described using the adjectives: weak, low, ok, strong and high, a qualifier for the 'pressure' fuzzy variable should be added for each of these. The fuzzy 'pressure' variable should have a range of 0 to 300 (measured in 100 Kpa units of pressure) and each qualifier needs reasonable values to define its membership function in the context of this example. The result is the following definition for the 'pressure' fuzzy variable:

```
fuzzy_variable( pressure ) :-
    [ 0 , 300 ] ;
    weak,   \ , linear, [  10 ,  70     ] ;
    low,    /\, linear, [  10 ,  70 , 130 ] ;
    ok,     /\, linear, [  70 , 130 , 190 ] ;
    strong, /\, linear, [ 130 , 190 , 250 ] ;
    high,   /, linear, [     190 , 250 ] .
```

The following diagram shows the membership functions for all the 'pressure' qualifiers combined on a single graph:

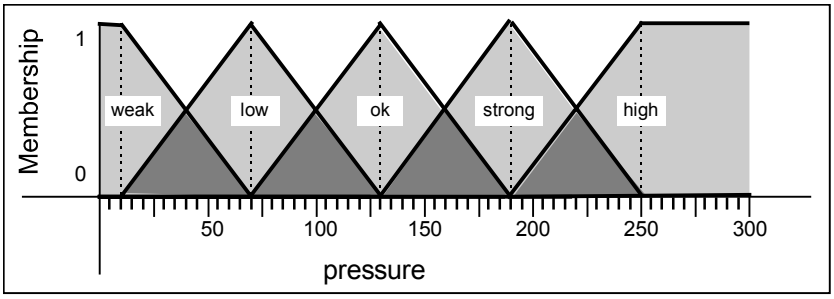


Figure 28 - the qualifiers for the 'pressure' variable

The final fuzzy variable in the example is the 'throttle' output variable.

For convenience we will define the range for the 'throttle' as being between -60 and 60 (a negative number indicating that the throttle should be moved back and a positive value that it should be moved forward). The problem results in moving the throttle by large, medium or small amounts in the negative and positive directions. So we'll assign seven qualifiers that will cover these resultant actions and give them values suitable to their range. This time we'll also need to consider how to return a 'crisp' value from the variable at the end of the fuzzy program. As a reasonable starting point we'll use the default 'centroid' method. This results in the following definition for the 'throttle' fuzzy variable:

```
fuzzy_variable( throttle ) :-
    [ -60 , 60 ] ;
    negative_large,  \ , linear, [ -45 , -30      ] ;
    negative_medium, /\, linear, [ -45 , -30 , -15 ] ;
    negative_small,  /\, linear, [ -30 , -15 ,   0 ] ;
    zero,            /\, linear, [ -15 ,   0 ,  15 ] ;
    positive_small,  /\, linear, [   0 ,  15 ,  30 ] ;
    positive_medium, /\, linear, [  15 ,  30 ,  45 ] ;
    positive_large,   /, linear, [       30 ,  45 ] ;
    centroid(largest_membership,mirror_rule,shrinking) .
```

The following diagram shows the membership functions for all the 'throttle' qualifiers combined on a single graph:

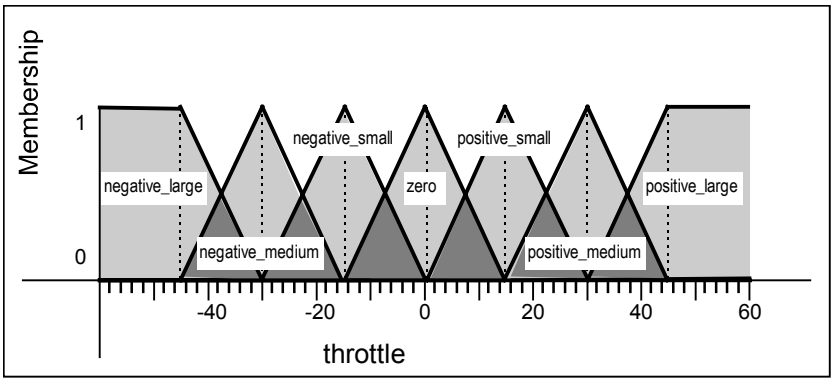


Figure 29 - the qualifiers for the 'throttle' variable

Defining Fuzzy Rules

After defining the fuzzy variables for our problem we can then proceed to the fuzzy rules. Let's suppose that there are many rules relating to temperature, pressure and throttle, that have been obtained by talking to the steam turbine experts. In this example the rules will cover all the possible combinations of temperature and pressure and give a resultant throttle change for each.

Three example rules, from the experts, are written below:

'*if the temperature is cold and the pressure is weak then increase the throttle by a large amount*'

'if the temperature is hot and the pressure is high then decrease the throttle by a large amount'

'*if the temperature is normal and the pressure is ok then don't change the throttle*'

Each of these rules could be defined separately using the following fuzzy rule definitions:

```
uncertainty_rule(throttle1) :-
 if temperature is cold
 and pressure is weak
 then throttle is positive_large .

uncertainty_rule(throttle2) :-
 if temperature is hot
 and pressure is high
 then throttle is negative_large .

uncertainty_rule(throttle3) :-
 if temperature is normal
 and pressure is ok
 then throttle is zero .
```

We could then proceed to define the remaining rules in this form. Alternatively, because the rules always apply to temperature, pressure and throttle, we can combine them all into a single fuzzy matrix. The complete set of rules for the problem would then be as follows (the three rules individually defined above are shown in the matrix in italics):

```
fuzzy_matrix( t ) :-

    temperature * pressure -> throttle      ;

    cold      * weak    -> positive_large ;
    cold      * low     -> positive_medium ;
    cold      * ok      -> positive_small  ;
    cold      * strong  -> negative_small  ;
    cold      * high    -> negative_medium ;

    cool      * weak    -> positive_large  ;
    cool      * low     -> positive_medium ;
    cool      * ok      -> zero            ;
    cool      * strong  -> negative_medium ;
    cool      * high    -> negative_medium ;

    normal    * weak    -> positive_medium ;
    normal    * low     -> positive_small  ;
    normal    * ok      -> zero            ;
    normal    * strong  -> negative_small  ;
    normal    * high    -> negative_medium ;

    warm      * weak    -> positive_medium ;
    warm      * low     -> positive_small  ;
    warm      * ok      -> negative_small  ;
    warm      * strong  -> negative_medium ;
    warm      * high    -> negative_large  ;

    hot       * weak    -> positive_small  ;
    hot       * low     -> positive_small  ;
    hot       * ok      -> negative_medium ;
    hot       * strong  -> negative_large  ;
    hot       * high    -> negative_large  .
```

Wrapping Up Fuzzy Programs

The concluding process in implementing the turbine program is to wrap up the fuzzy variables and rules in a program that can be run. This program will set the inputs for the 'temperature' and 'pressure' fuzzy variables, decide how to propagate the degrees of membership using the fuzzy rule matrix and finally, after propagation, get the result from the 'throttle' fuzzy variable.

The predicate we will define will have three arguments. The first two will be the input values for the 'temperature' and 'pressure' fuzzy variables and the third argument will return the resultant value of the 'throttle' fuzzy variable. Setting the inputs for and getting the results from fuzzy variables can be done using the predicate

fuzzy_variable_value/2. There will be three calls to this predicate in our program, one for each fuzzy variable.

The fuzzy rule propagation can be arranged using the *fuzzy_propagate/4* predicate. We need to decide how the membership values will be propagated from the condition qualifiers to the conclusion qualifier according to the conjunctions, disjunctions and negations contained within the fuzzy rules. As a starting point we will use the defaults: minimum for conjunctions (the minimum of the membership values in the conjunction will be propagated), maximum for disjunction (the maximum of the membership values in the conjunction will be propagated) and complement for negations (1 minus the negated membership value will be propagated).

Combining the setting of inputs, fuzzy propagation and the getting of outputs into a single program, we end up with the following:

```
find_throttle( Temperature, Pressure, Throttle ) :-
    fuzzy_variable_value( temperature, Temperature ),
    fuzzy_variable_value( pressure,    Pressure    ),
    fuzzy_propagate( minimum, maximum, complement, [t] ),
    fuzzy_variable_value( throttle,    Throttle    ).
```

This program will work fine the first time it is run, but it is not quite complete. We also need to ensure that the 'throttle' fuzzy variable does not contain any 'old' values that might affect the fuzzy propagation. The following definition shows the complete *find_throttle/3* predicate:

```
find_throttle( Temperature, Pressure, Throttle ) :-
    fuzzy_reset_membership( throttle ),
    fuzzy_variable_value( temperature, Temperature ),
    fuzzy_variable_value( pressure,    Pressure    ),
    fuzzy_propagate( minimum, maximum, complement, [t] ),
    fuzzy_variable_value( throttle,    Throttle    ).
```

Compiling The Fuzzy Program

Fuzzy logic programs consist of extensions to the normal Prolog syntax. Prior to compiling any fuzzy logic program you will need to load the fuzzy logic interpreter. This is in the file FLINT.PC which will have been placed in the SYSTEM directory when you installed the *FLINT* toolkit from the ***WIN-PROLOG*** CD-ROM. To load this file, enter the following command at the Prolog command line:

```
?- ensure_loaded( system(flint) ).
```

A fuzzy logic example, TURBINE.PL, containing the complete 'turbine' program will also have been installed when you installed the *FLINT* toolkit from the ***WIN-PROLOG*** CD-ROM, this time in the

EXAMPLES\FLINT directory. Having loaded the fuzzy logic interpreter this example may then be compiled using the command:

```
?- ensure_loaded( examples('flint\turbine') ).
```

Running The Fuzzy Program

The fuzzy turbine program has the predicate *find_throttle/3* as its top-level goal. This program can be queried at the Prolog command-line using the following goal:

```
?- find_throttle( 300, 150, Throttle ) .
Throttle = -15
```

Chapter 8 - Fuzzy Logic - A KSL Example

This section investigates how to define the steam turbine example from the previous section this time using flex and its Knowledge Specification Language (KSL). The basic idea of the example was to define a set of fuzzy rules to adjust the throttle of a steam turbine according to its current temperature and pressure to keep it running smoothly.

The fuzzy logic components of this KSL example are similar to their equivalents in the Prolog example shown in the previous section. The fuzzy variables and fuzzy rule matrix are shown in their entirety below:

```
% Fuzzy variables

fuzzy_variable temperature ;
 ranges from 0 to 500 ;
 fuzzy_set cold   is \  shaped and linear at 110 , 165      ;
 fuzzy_set cool   is /\ shaped and linear at 110 , 165 , 220 ;
 fuzzy_set normal is /\ shaped and linear at 165 , 220 , 275 ;
 fuzzy_set warm   is /\ shaped and linear at 220 , 275 , 330 ;
 fuzzy_set hot    is /  shaped and linear at 275 , 330      .

fuzzy_variable pressure ;
 ranges from 0 to 300 ;
 fuzzy_set weak   is \  shaped and linear at 10 ,  70       ;
 fuzzy_set low    is /\ shaped and linear at 10 ,  70 , 130  ;
 fuzzy_set ok     is /\ shaped and linear at 70 , 130 , 190  ;
 fuzzy_set strong is /\ shaped and linear at 130, 190 , 250  ;
 fuzzy_set high   is /  shaped and linear at 190, 250  .

fuzzy_variable throttle ;
 ranges from -60 to 60  ;
 fuzzy_set negative_large  is \  shaped and linear at -45 , -30      ;
 fuzzy_set negative_medium is /\ shaped and linear at -45 , -30 , -15 ;
 fuzzy_set negative_small  is /\ shaped and linear at -30 , -15 ,   0 ;
 fuzzy_set zero            is /\ shaped and linear at -15 ,   0 ,  15 ;
 fuzzy_set positive_small  is /\ shaped and linear at   0 ,  15 ,  30 ;
 fuzzy_set positive_medium is /\ shaped and linear at  15 ,  30 ,  45 ;
 fuzzy_set positive_large  is /  shaped and linear at  30 ,  45      .

% FAM - Fuzzy Associative Memory
```

```
fuzzy_matrix throttle_value
  temperature  *  pressure -> throttle       ;
  cold       *  weak     -> positive_large  ;
  cold       *  low      -> positive_medium ;
  cold       *  ok       -> positive_small  ;
  cold       *  strong   -> negative_small  ;
  cold       *  high     -> negative_medium ;
  cool       *  weak     -> positive_large  ;
  cool       *  low      -> positive_medium ;
  cool       *  ok       -> zero            ;
  cool       *  strong   -> negative_medium ;
  cool       *  high     -> negative_medium ;
  normal     *  weak     -> positive_medium ;
  normal     *  low      -> positive_small  ;
  normal     *  ok       -> zero            ;
  normal     *  strong   -> negative_small  ;
  normal     *  high     -> negative_medium ;
  warm       *  weak     -> positive_medium ;
  warm       *  low      -> positive_small  ;
  warm       *  ok       -> negative_small  ;
  warm       *  strong   -> negative_medium ;
  warm       *  high     -> negative_large  ;
  hot        *  weak     -> positive_small  ;
  hot        *  low      -> positive_small  ;
  hot        *  ok       -> negative_medium ;
  hot        *  strong   -> negative_large  ;
  hot        *  high     -> negative_large  .
```

Wrapping Up Fuzzy KSL Programs

As in the Prolog example the KSL *find_throttle/3* program will set the inputs for the 'temperature' and 'pressure' fuzzy variables, decide how to propagate the degrees of membership using the fuzzy rule matrix and finally, after propagation, get the result from the 'throttle' fuzzy variable.

```
% Propagation of fuzzy values

relation get_throttle_value(Temperature,Pressure,Throttle)
  if reset all fuzzy values
  and fuzzify the temperature from Temperature
  and fuzzify the pressure    from Pressure
  and propagate throttle_value fuzzy rules
  and defuzzify the throttle   to Throttle .
```

So far we have simply provided a syntactic re-write of the Prolog example. The flex KSL language provides access to many features aimed at the expert system builder. The code that follows ties the example closer to the flex methodology using frames, attributes and

data-driven programming. As a start we will create a 'turbine' frame that will model the real turbine that we are attempting to control:

```
% Turbine Frame
frame turbine
 default  temperature is 0
 and default pressure is 0
 and default throttle is 0 .
```

When a temperature or pressure measurement comes from the physical turbine we will update the appropriate attributes in the 'turbine' frame; this can be done with the following KSL actions:

```
% Set Turbine Values

action set_turbine_temperature(T)
 do the temperature of turbine becomes T .

action set_turbine_pressure(P)
 do the pressure of turbine becomes P .
```

We then need an action to get the 'temperature' and 'pressure' values from the 'turbine' frame pass them through the fuzzy logic program and finally assign the resultant 'throttle' value to the 'turbine' frame's 'throttle' attribute. This is done in the following action called *set_turbine_throttle/0*:

```
action set_turbine_throttle
 do  check the temperature of turbine is Temperature
 and check the pressure of turbine is Pressure
 and get_throttle_value(Temperature,Pressure,Throttle)
 and the throttle of turbine becomes Throttle .
```

We will now provide a way of automatically ensuring that the 'throttle' value is calculated whenever the 'temperature' or 'pressure' attributes are set. This is done by implementing the following two demons:

```
% Data-Driven Programs

demon react_to_temperature_update
 when the temperature of turbine changes to T
 then set_turbine_throttle .

demon react_to_pressure_update
 when the pressure of turbine changes to P
 then set_turbine_throttle .
```

The following action will print out the current state of the 'turbine' frame:

```
% Display Turbine Values
```

```
action display_turbine_values;
  do write('The current temperature is: ')
  and write(the temperature of turbine)
  and nl
  and write('The current pressure is: ')
  and write(the pressure of turbine)
  and nl
  and write('The current throttle is: ')
  and write(the throttle of turbine)
  and nl .
```

The following action will test setting the parameters of the turbine:

```
action test_turbine_values
  do  set_turbine_temperature(300)
  and set_turbine_pressure(150)
  and display_turbine_values .
```

Compiling The Fuzzy KSL Program

Prior to compiling any fuzzy logic KSL program you will need to load the fuzzy logic interpreter. This is in the file FLINT.PC which will have been placed in the SYSTEM directory when you installed the *FLINT* toolkit from the ***WIN-PROLOG*** CD-ROM. To load this file enter the following command at the Prolog command line:

```
?- ensure_loaded( system(flint) ).
```

A fuzzy logic example, TURBINE.KSL, containing the complete 'turbine' program will also have been installed when you installed the *FLINT* toolkit disk, this time in the EXAMPLES directory. Having loaded the fuzzy logic interpreter this example may then be compiled using the command:

```
?- reconsult_rules( 'c:\program files\win-prolog 4300\examples\flint\turbine' ).
```

Running The Fuzzy KSL Program

After the KSL fuzzy turbine program has been compiled. You can set values for the temperature and pressure attributes of the turbine frame and the throttle value will be assigned automatically. Our test program shows this:

```
?- test_turbine_values.
The current temperature is: 300
The current pressure is: 150
The current throttle is: -27.6136363636364
yes
```

If we wanted to, we could also set the temperature and pressure separately. The following goals individually set the temperature to 300 and the pressure to 150 (as in our test program):

```
?- set_turbine_temperature(300).
Yes

?- set_turbine_pressure(150).
yes
```

The next goal reports the current state of the turbine frame showing that the throttle has been set:

```
?- display_turbine_values.
The current temperature is: 300
The current pressure is: 150
The current throttle is: -27.6136363636364
yes
```

Chapter 9 - Dealing with Uncertainty

This section aims to serve as a basic introduction to the various techniques that Flint now offers to support uncertainty. For a more detailed explanation, you are referred to:

> "Intelligent Systems for Engineers and Scientists"
>
> (Hopgood, CRC Press, ISBN: 0-8493-0456-3)

Traditional expert systems work on the basis that everything is either true or false, and that any rule whose conditions are satisfiable is useable, i.e. its conclusion(s) are true. This is rather simplistic and can lead to quite brittle expert systems. Flint offers support for where the domain knowledge is not so clearcut.

Given a rule:

> rule1: if A & B then C

there are 3 potential areas for uncertainty.

- Uncertainty in data (how true are A and B)

- Uncertainty in the rule (how often does A and B imply C)

- Impreciseness in general

The first two can be handled using techniques based on probability theory and the third using fuzzy logic. The latter has been extensively explained in previous chapters. Now, we will look at the first two.

Bayesian updating

If I tell you that the average height of people in your country is 1.75m, then you might reasonably expect that the probability of the next person you meet being over or under that height is 50%.

If I now tell you that there is a Giants Convention in town, then you might revise you expectations in light of this; such that you increase the likelihood of meeting someone over 1.75m from 50% to, say, 60%.

This revision of probabilities is fundamental to Bayesian updating.

Using Flint, we can combine any number of Bayesian rules into a Bayesian network, and then propagate values through the network. Typically, this is done in the light or absence of evidence.

Bayesian networks get their name from the Rev. Thomas Bayes, who wrote an essay, posthumously published in 1763, that offered a mathematical formula for calculating probabilities among several variables that are causally related but for which – unlike calculating the probability of a coin landing on heads or tails – the relationships can't easily be derived by experimentation.

But it was the rapid progress in computer power and the development of key mathematical equations that made it possible for the first time, in the late 1980s, to compute Bayesian networks with enough variables that they were useful in practical applications.

Bayesian networks are complex diagrams that organise the body of knowledge in any given area by mapping out cause-and-effect relationships among key variables and encoding them with numbers that represent the extent to which one variable is likely to affect another.

Programmed into computers, these systems can automatically generate optimal predictions or decisions even when key pieces of information are missing.

Bayesian networks provide an overarching graphical framework that brings together diverse elements of AI and increasing the range of its likely application to the real world.

Bayes' theorem

Bayes' theorem states:

$P(H/E) = P(H) * P(E/H) / P(E)$

This states the probability of a hypothesis given some evidence in terms of the probablity of the evidence given the hypothesis. This is useful as it is generally easier to estimate the probablity of evidence given a hypothesis than the reverse.

Combining Probabilities

The general syntax for a probabilistic rule in Flint is:

uncertainty_rule(*rule_name*) :-

if the *variable* is [*not*] [*hedge*] *qualifier* [*(affirms number; denies number)*]
[or/and the *variable* is [*not*] [*hedge*] *qualifier* [*(affirms number; denies number)*]]
then the *variable* is *qualifier*
[and the *variable* is *qualifier*]
[with certainty_factor *number*] .

A very simple rule could be:

```
uncertainty_rule( r33 ) :-
 if the temperature is high
 and the water_level is not low
 then the pressure is high .
```

In flex, using an extension of the KSL, this would be:

```
uncertainty_rule r33
 if the temperature is high
 and the water_level is not low
 then the pressure is high .
```

Later on, we shall see that this is equivalent to having affirms and denies values of 1; i.e:

```
uncertainty_rule r33
 if the temperature is high ( affirms 1.00 ; denies 1.00 )
 and the water_level is not low ( affirms 1.00 ; denies 1.00 )
 then the pressure is high .
```

and with a prior probablity for 'pressure is high' of 0.5

Probabilistic rules can be invoked via:

```
boiler( Measure, M1, M2, M3, M4, M5, M6 ) :-
 uncertainty_value_reset( Measure ),
 uncertainty_value_set(   Measure, water_level,   low,  M1 ),
 uncertainty_value_set(   Measure, warning_light, on, M2 ),
 uncertainty_value_set(   Measure, temperature, high, M3 ),
 uncertainty_value_set(   Measure, pressure, high, M4 ),
 uncertainty_value_set(   Measure, release_valve, stuck, M5 ),
 uncertainty_propagate(   Measure, [r3_4,r3_3,r3_2,r3_1]          ),
 uncertainty_value_get(   Measure, release_valve, need_cleaning, M6 ).
```

Or in flex:

```
relation simple_boiler_probability( P )
 if  trace propagation
 and reset all probability values
 and the probability that the water_level is low  = 0.03
 and the probability that the temperature is high  = 0.98
 and propagate simple_boiler_control probability rules
 and the probability that the pressure is high  = P .
```

Affirms and denies

We can attach weights to update our confidence in a hypothesis given new evidence. The larger the affirmed weight, the more confident we can be in an hypothesis.

A Bayesian rule in flex looks like:

```
uncertainty_rule r44
 if the temperature is high ( affirms 18.00 ; denies 0.11 )
 and the water_level is not low ( affirms  1.90 ; denies 0.10 )
 then the pressure is high .
```

Advantages of Bayesian updating are:

1) technique is based on a proven statistical theorem

2) likelihood is expressed as a probability (or odds)

3) weightings are based upon the probability of evidence

Disadvantages of Bayesian updating are:

1) the prior probability of an assertion must be known or estimated

2) dependant probabilities must be measured or estimated

3) the probability value tells us nothing about its accuracy

4) adding new rules often requires alterations to prior probablities and weightings in other rules

Odds and Probability

For the purpose of updating probabilities in a rule-based system, it is often more convenient to deal with the odds of an event arising rather than the probability. The odds of an hypothesis, O(H), are related to its probability, P(H), by the following relations.

O(H) = P(H) / P(~H) = P(H) / (1-P(H))

and

P(H) = O(H) / (O(H)+1)

Thus a hypothesis with a probability of 0.2 (1 in 5 chance) has odds of 0.25 (or "4 to 1" against).

Similarly, a hypothesis with a probability of 0.8 (4 in 5 chance) has odds of 4 (or "4 to 1" on).

If you would rather use odds than probabilities for the above example, then you can have the following:

```
relation boiler_odds( 0 )
 if  trace propagation
 and reset all odds values
 and the odds that the water_level is low  = 0
 and the odds that the temperature is high  = 65535
 and the odds that the pressure is high  = 0.110
 and propagate boiler_control odds rules
 and the odds that the release_valve is need_cleaning = 0 .
```

An answer, such as, O = 2.18, indicates that the odds that the valve needs cleaning are 2.18.

Evidence based probabilities

The standard formula for updating the odds of a hypothesis, H, given that evidence, E, is observed is:

O(H/E) = A * O(H)

where O(H/E) is the odds of H given E, and A is the affirms weigth of E. The definition of A is:

A = P(E/H) / P(E/~H)

This is also sometimes referred to as the 'Likelihood Ratio' and is the ratio of probabilities that the evidence is there when the hypothesis is true to when the hypothesis is false; i.e. given the evidence exists, how likely is it that the hypothesis is true.

Absence of Evidence

The absence of evidence is different from not knowing if the evidence is present or not, and can be used to reduce the probability of a hypothesis. The standard formula for updating the odds of a hypothesis, H, given that evidence, E, is absent is:

$O(H/\sim E) = D * O(H)$

where $O(H/\sim E)$ is the odds of H given the absence of E, and D is the denies weigth of E. The definition of D is:

$D = P(\sim E/H) / P(\sim E/\sim H)$

or

$D = (1 - P(E/H)) / (1 - P(E/\sim H))$

This is the ratio of probabilities that the evidence is not there when the hypothesis is true to when the hypothesis is false; i.e. given the evidence is absent, how likely is it that the hypothesis is true.

Uncertain Evidence

To refelect uncertainty in E, we scale both A and D to A' and D' respectively using linear interpolation. The expressions used to calculate interpolated values are:

$A' = [2(A-1) * P(E)] + 2 - A$

$D' = [2(1-D) * P(E)] + D$

While P(E) is greater than 0.5 we use the *affirms* weight, and when P(E) is less than 0.5, we use the *denies* weight.

Certainty Theory

Certainty theory, as used in MYCIN, represents an attempt to overcome some of the shortcomings of Bayesian updating. Instead of using probablities, each assertion has a certainty value between 1 and -1 associated with it, as do rules.

The updating procedure for certainty values consists of adding a +ve or -ve value to the current certainty of a hypothesis. This contrasts with Bayesian updating where the odds of a hypothesis are always multiplied by the appropriate weighting. The basic formulae are:

CF' = CF' x C(E)

a) if C(H) >= 0 and CF' >= 0

C(H/E) = C(H) + [CF' x (1 - C(H))]

b) if C(H) <= 0 and CF' <= 0

C(H/E) = C(H) + [CF' x (1 + C(H))]

c) if C(H) and CF' have opposite signs

C(H/E) = C(H) + CF'/ (1 - min(|C(H)|, |CF|))

where:

C(H/E) is the certainty of H updated in the light of E

C(H) is the initial certainty of H

```
uncertainty_rule r41
 if the release_valve is stuck
 then the release_valve is need_cleaning
 with certainty factor 1.0 .

relation boiler_cf( CF1, CF2, CF3 )
 if trace propagation
 and reset all certainty_factor values
 and the certainty_factor that the water_level is low  = -1
 and the certainty_factor that the warning_light is on  = 1
 and propagate boiler_control certainty_factor rules
 and the certainty_factor that the release_valve is stuck = CF2
 and the certainty_factor that the release_valve is need_cleaning = CF3
```

Tracing calculations

You can see what is going on inside the propagation engine by switching on the tracing mechanism. You can do this by typing into the Console:

```
?- uncertainty_trace .
```

and you can turn it off using:

```
?- uncertainty_notrace .
```

Alternatively, you can include calls to these directives within your program code.

If using flex, you can use the KSL equivalents within your code, namely the keywords: **trace propagation** and **notrace propagation**. However, you cannot directly enter these into the Console window.

You will then get something like:

```
| ?- boiler.

Prob. : UPDATE    : (water_level is low) = 0
Prob. : UPDATE    : (warning_light is on) = 1
Prob. : UPDATE    : (temperature is high) = 1
Prob. : UPDATE    : (pressure is high) = 0.099
Prob. : UPDATE    : (release_valve is stuck) = 0.02

Prob. : TRY       : r3_4
Prob. : LOOKUP    : (temperature is high) = 1
Prob. : AFFIRMS   : weight(18) @ 1 -> 0.947368421052632
Prob. : LOOKUP    : (water_level is low) = 0
Prob. : DENIES    : weight(1.9) @ 0 -> 0.655172413793103
Prob. : AND       : 0.947368421052632 + 0.655172413793103 -> 0.971590909090909
Prob. : LOOKUP    : (pressure is high) = 0.099
Prob. : CONFIRMS  : 0.099 + 0.971590909090909 -> 0.789819912288887
Prob. : UPDATE    : (pressure is high) = 0.789819912288887
Prob. : FIRED     : r3_4

Prob. : TRY       : r3_3
Prob. : LOOKUP    : (pressure is high) = 0.789819912288887
Prob. : AFFIRMS   : weight(85) @ 0.789819912288887 -> 0.980272143906399
Prob. : LOOKUP    : (release_valve is stuck) = 0.02
Prob. : CONFIRMS  : 0.02 + 0.980272143906399 -> 0.503494513349305
Prob. : UPDATE    : (release_valve is stuck) = 0.503494513349305
Prob. : FIRED     : r3_3
```

This should help you track the updating of the values as the computation happens.

Chapter 10 - Uncertainty Syntax

Notational Conventions

Alternatives

```
[ alternative_1 | alternative_2 | ... | alternative_K ]
```

Repeat Zero or More Times

```
*** repeated ***
```

```
Optional
```

```
{ option }
```

Terminal (Emboldened)

```
Terminal
```

Non-Terminal

```
<non_terminal>
```

Example

reset all [**fuzzy** | **probability** | **odds** | **certainty**] **values**

can be any of the following :-

```
reset all fuzzy values
```

```
reset all odds values
```

```
reset all probability values
```

```
reset all certainty values
```

Example

{ **the** } **probability that**
{ **the** } <name> [**is** | **are**] <name>
= [<variable> | <number>]

can be any of, but not exclusively, the following :-

```
the probability that the pressure is low = P1
```

```
probability that clouds are high = 0.333
```

Example

```
*** ; qualifier <name> is [ \ | / | /\ ] shaped ***
```

can be, but not exclusively, the following :-

```
; qualifier low is \ shaped
; qualifier medium is /\ shaped
; qualifier high is / shaped
```

Declaring Uncertainty Variables

Fuzzy Variable

Example

```
fuzzy_variable brake_force ;
  ranges from 0 to 100 ;
  qualifier very_low  is \  shaped and linear     at 0, 25              ;
  qualifier lowish    is /\ shaped and curved 0.5 at 0, 25, 50          ;
  qualifier average   is /\ shaped and linear     at    25, 50, 75      ;
  qualifier highish   is /\ shaped and curved 2.0 at        50, 75, 100 ;
  qualifier very_high is /  shaped and linear     at            75, 100 ;
  defuzzifier = centroid .
```

Syntax

```
fuzzy_variable <variable_name>
  { ; ranges { from <number> } { to <number> } }
  ***
      ;
      qualifier <qualifier_name>
      is [ / | \ | /\ | \/ | /-\ | \-/ ] shaped
      { and [ linear | curved <number> ] }
      at <number> , <number> { *** , <number> *** }
  ***
  { ; defuzzifier = [ centroid | peak ] }
  .
```

Accessing / Updating Uncertainty Variables

Reset Values

Examples

```
reset all fuzzy values

reset all probability values

reset all odds values

reset all certainty values
```

Syntax

```
reset all [ fuzzy | probability | odds | certainty ] values
```

Fuzzy Values

Examples

```
the absolute value of buffer_distance is 500

the absolute value of brake_force is BufferDistance
```

Syntax

```
the absolute value of { the } <variable_name>
[ is | are ] [ <variable> | <number> ]
```

Probability Values

Example

```
the probability that the clouds are dark = 0.5

the odds that the clouds are dark = 1.0
```

Syntax

```
the [ probability | odds ] that { the } <variable_name>
[ is | are ] <qualifier_name>
= [ <variable> | <number> ]
```

Certainty Factor Values

Example

```
the certainty that the release_valve is need_cleaning = CF6
```

Syntax

```
the certainty that { the } <variable_name>
[ is | are ] <qualifier_name>
= [ <variable> | <number> ]
```

Defining Uncertainty Rules

Fuzzy Hedge

Examples

```
fuzzy_hedge very  is power 3 .
fuzzy_hedge quite is power 0.5 .
```

Syntax

```
fuzzy_hedge <hedge_name> is power <number> .
```

Uncertainty Rule

Examples

```
uncertainty_rule braking_very_high
  if   the buffer_distance is not quite low
  or   the train_speed     is     very  high
  then the brake_force     is     very  high .

uncertainty_rule r44b
  if    the temperature   is high      ( affirms 18.00 ; denies 0.11 )
  and   the water_level   is low       ( affirms  0.10 ; denies 1.90 )
  then  the pressure      is high .

uncertainty_rule r43c
  if   the pressure      is high       ( affirms 1.00 ; denies 0.1 )
  then the release_valve is stuck
  with certainty factor 0.9 .
```

Syntax

```
uncertainty_rule <rule_name>
  if { not }
    { the } <variable_name> [ is | are ] { not }
    { <hedge_name> } <qualifier_name>
    { ( affirms <number> ; denies <number> ) }
    ***
        [ or | and ] { not }
        { the } <variable_name> [ is | are ] { not }
        { <hedge_name> } <qualifier_name>
        { ( affirms <number> ; denies <number> ) }
    ***
```

```
  then
    { the } <variable_name> [ is | are ] <qualifier_name>
    ***
        and { not }
        { the } <variable_name> [ is | are ] { not }
        { <hedge_name> } <qualifier_name>
        { ( affirms <number> ; denies <number> ) }
    ***
  { with certainty factor <number> }
  .
```

Propagating Uncertainty Rules

Examples

```
group braking
  braking_very_high,
  braking_highish,
  braking_lowish,
  braking_very_low .

group weather_forecast
  r1 , r2 , r3 , r4 , r5 , r6 , r7 , r8 , r9 , r10 , r11 , r12 .

group boiler_control_c
  r44c , r43c , r42c , r41c .

…
and propagate braking fuzzy rules
and propagate weather_forecast probability rules
and propagate weather_forecast odds rules
and propagate boiler_control_c certainty factor rules
…
```

Syntax

```
propagate <group_name>
[ fuzzy | probability | odds | certainty factor ]
rules
```

The Shape of Inference: Using LPA's AI Toolkits

By Clive Spenser & Charles Langley

Inference is at the core of Knowledge-Based Artificial Intelligence. Different inference techniques require slightly different rules and give somewhat different behaviour.

In this document we contrast brittle production rule inference with more sophisticated methods by setting up four separate KBSs constructed using snippets of code from LPA's AI toolkits. To keep it simple, the examples all have two inputs and one output. This has the advantage of allowing us to show the results of such inference by means of simple graphs.

Production Rule Inference

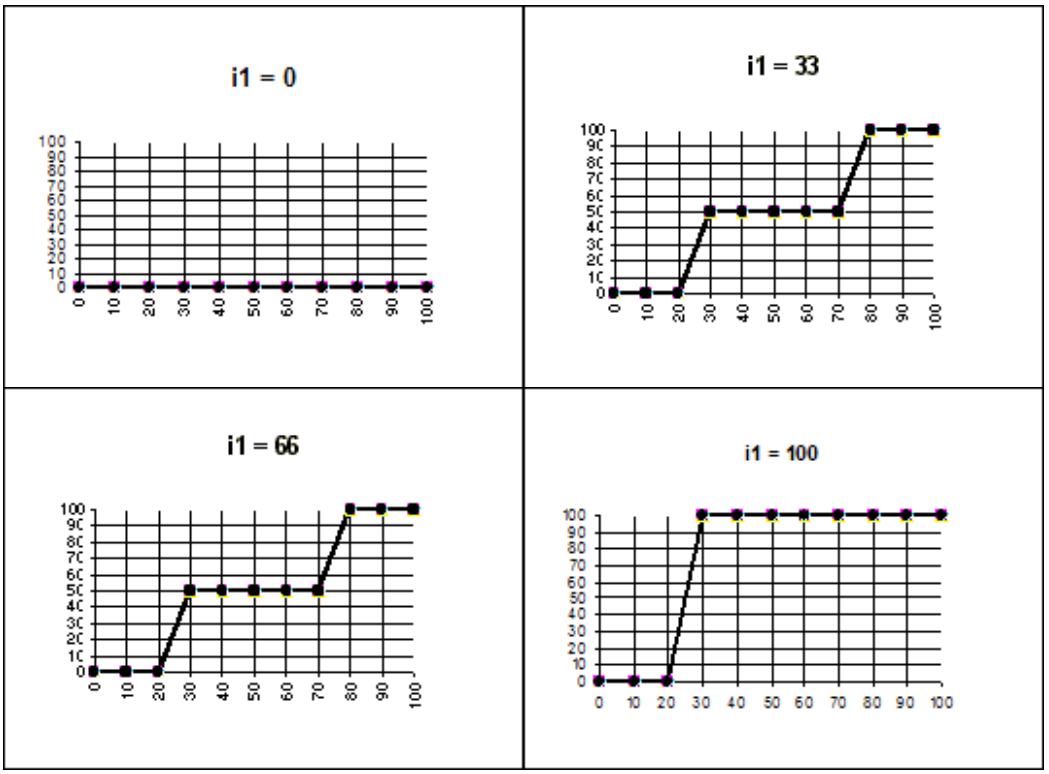
In the early days of Knowledge Based systems inference methods were brittle. That is to say, a rule representing an item of knowledge either fired or did not fire. Yes or No.

Example One Production Rule Inference

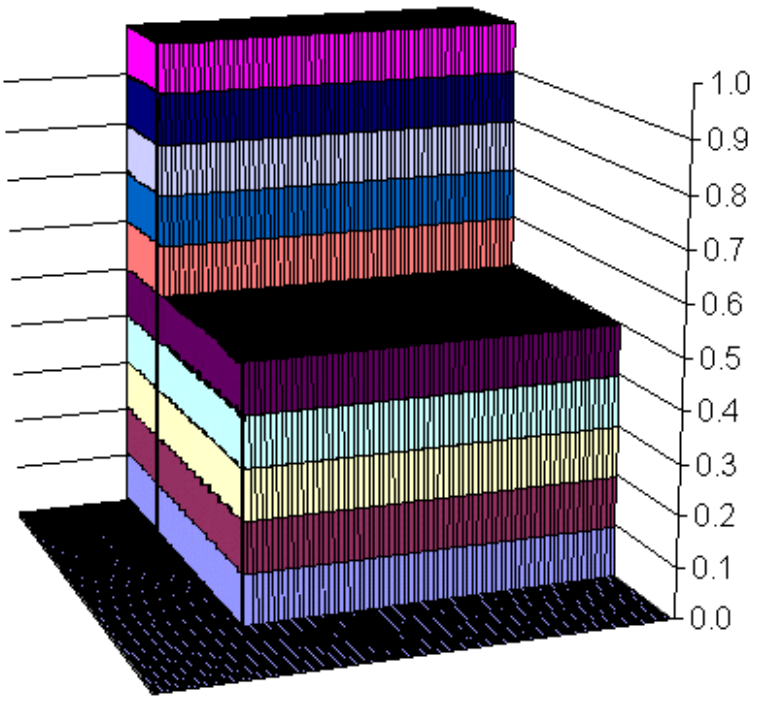
```
rule d_1 if i1 <   25             and i2 <   25             then output becomes   0.
rule d_2 if i1 <   25             and i2 >= 25 and i2 =< 75 then output becomes   0.
rule d_3 if i1 <   25             and i2 >   75             then output becomes   0.
rule d_4 if i1 >= 25  and i1 =< 75 and i2 <   25             then output becomes   0.
rule d_5 if i1 >= 25  and i1 =< 75 and i2 >= 25 and i2 =< 75 then output becomes  50.
rule d_6 if i1 >= 25  and i1 =< 75 and i2 >   75             then output becomes 100.
rule d_7 if i1 >   75             and i2 <   25             then output becomes   0.
rule d_8 if i1 >   75             and i2 >= 25 and i2 =< 75 then output becomes  50.
rule d_9 if i1 >   75             and i2 >   75             then output becomes 100.
```

To show the characteristic shapes of the various inference techniques we hold one of the two inputs, I1, constant at 0, 33, 66 and 100 and vary the other from 0 to 100.

Given the magnitude of the two inputs, the magnitude of the output is shown on the vertical axis.



So far we have only seen two-dimensional slices of the output data. Here is the full 3D graph:



In the 3D graph 100 datapoints are plotted on each axis rather than 10 in order to counteract the sloping shown on the 2D graphs. This sloping is an artefact of the graphing itself. The 3D graph shows that we really have steps rather than slopes. The number of such steps can be increased simpy by increasing the number of production rules used.

Reasoning under Uncertainty

In the next three examples we shall be dealing with inference under uncertainty. There are two main sources of uncertainty, evidential uncertainty and semantic ambiguity (imprecision of language).

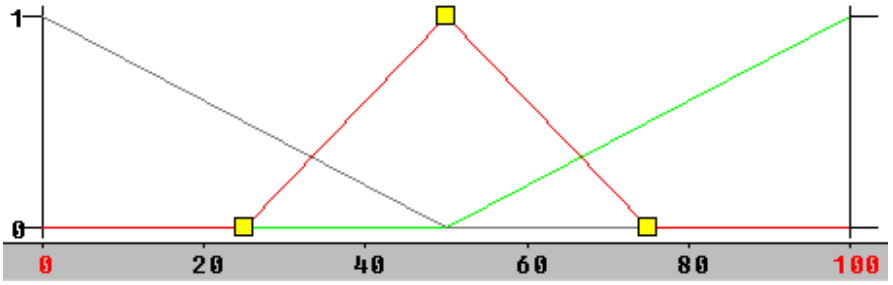
Fuzzy Logic

In the example below we use fuzzy logic, which deals with the ambiguity of terms such as low, medium and large.

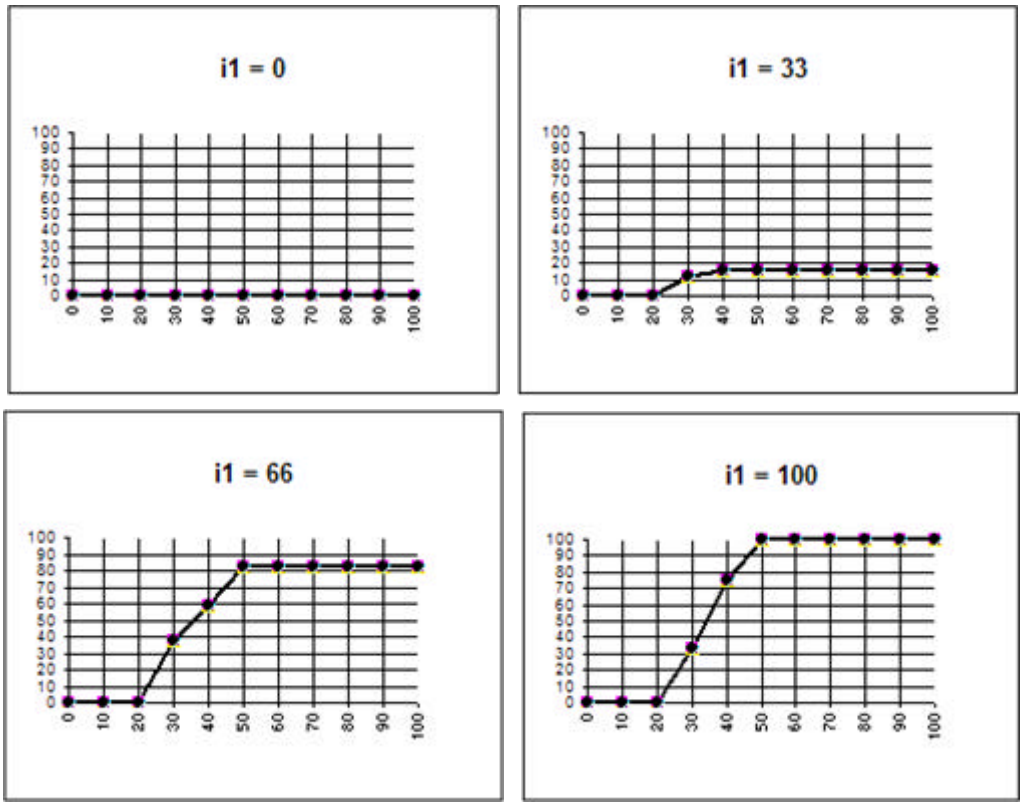
Example Two - Fuzzy Logic Inference

```
fuzzy_variable input1 ;
   ranges from 0 to  100 ;
   fuzzy_set  small is  \ shaped and linear at      0,  50 ;
   fuzzy_set medium is /\ shaped and linear at 25, 50,  75 ;
   fuzzy_set  large is /  shaped and linear at     50, 100 .
fuzzy_variable input2 ;
   ranges from 0 to  100 ;
   fuzzy_set  small is  \ shaped and linear at      0,  50 ;
   fuzzy_set medium is /\ shaped and linear at 25, 50,  75 ;
   fuzzy_set  large is /  shaped and linear at     50, 100 .
fuzzy_variable output ;
   ranges from 0 to  100 ;
   fuzzy_set  small is  \ shaped and linear at      0,  50 ;
   fuzzy_set medium is /\ shaped and linear at 25, 50,  75 ;
   fuzzy_set  large is /  shaped and linear at     50, 100 ;
   defuzzify using all memberships and mirror rule and shrinking .

fuzzy_rule d_1 if input1 is small   and input2 is small   then output is small .
fuzzy_rule d_2 if input1 is small   and input2 is medium  then output is small .
fuzzy_rule d_3 if input1 is small   and input2 is large   then output is small .
fuzzy_rule d_4 if input1 is medium  and input2 is small   then output is small .
fuzzy_rule d_5 if input1 is medium  and input2 is medium  then output is medium .
fuzzy_rule d_6 if input1 is medium  and input2 is large   then output is large .
fuzzy_rule d_7 if input1 is large   and input2 is small   then output is small .
fuzzy_rule d_8 if input1 is large   and input2 is medium  then output is medium .
fuzzy_rule d_9 if input1 is large   and input2 is large   then output is large .
```

Note that the fuzzy logic program contrasts with the expert system in that rather than using fixed values for the input and output: **0** for small, **50** for medium, **100** for large, the fuzzy logic program defines three fuzzy sets, **small**, **medium** and **large** using value ranges **0-50, 25-75** and **50-100**. Fuzzy inference then determines the degree of membership of each input and corresponding output in these three sets. As can be seen from the graphs, this results in a smoother distribution of output values.



An example fuzzy set

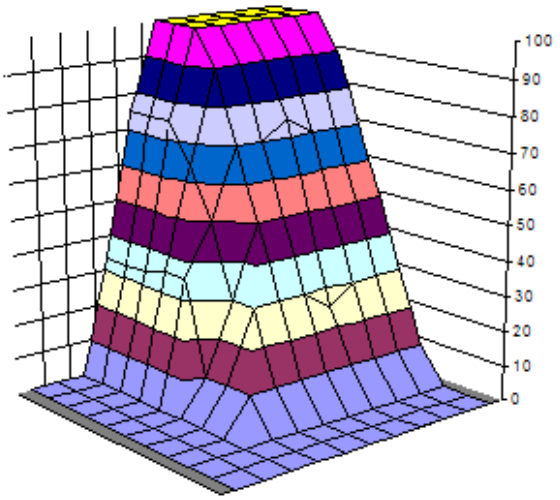


What is interesting here is the contrast with the graphs from the Production Rule expert system.

With Production Rules the graphs for I1=33 and I1=66 are the same, since each of these values is greater than 25 yet less than 75, hence medium in both cases.

The reason that we get variations on the shape of the graphs from the fuzzy system for values i1=33 and i1=66, is that the fuzzy system interprets these values as ***different degrees of being medium*** whereas for the production rule expert system, medium is just plain medium.

Here is the 3-D graph showing all slices fron the fuzzy system:



Bayesian inference

Our next example, Bayesian inference, deals with evidential uncertainty. Thus we interpret the two inputs as being ***evidence*** for the output. To implement a Bayesian knowledge base all we need to do is to attach an affirms and a denies weight to each of the conditions of the rules.

The affirms weight is calculated as:

$$A = \frac{P(E \mid H)}{P(E \mid \sim H)}$$

and the denies weight is calculated as:

$$D = \frac{P(\sim E \mid H)}{P(\sim E \mid \sim H)}$$

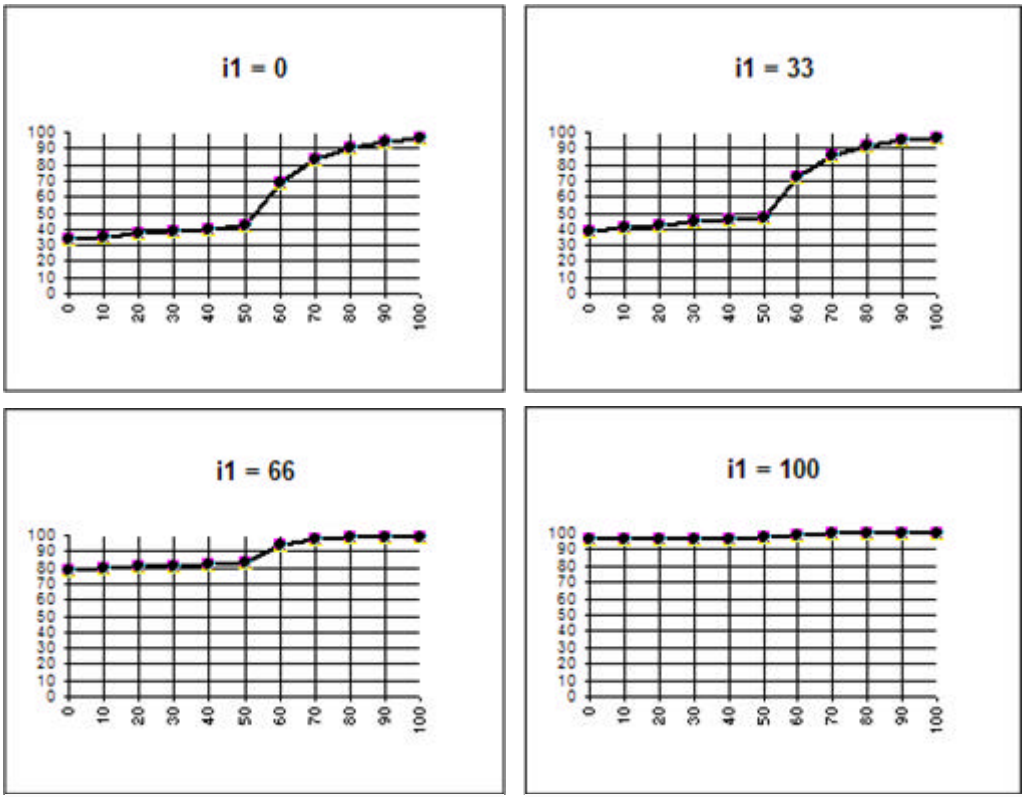
where P(E | H) is the conditional probability that H is true given that E is true, P(E | ~H) is the conditional probability that H is false given that E is true, P(~E | H) is the conditional probability that H is true given that E is false and P(~E | ~H) is the conditional probability that H is false given that H is false.

The advantage of using conditional probabilities to represent uncertainty is that their values can all be obtained from databases of previous cases. Note that when the affirms weight is less than the denies weight, this means that the evidence disconfirms the hypothesis.

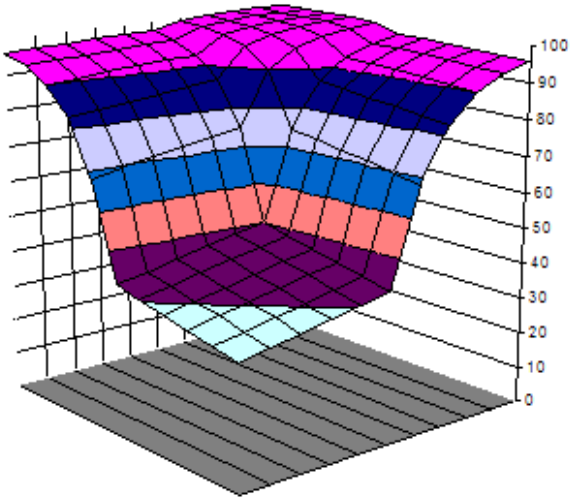
Example Three - Bayesian Inference

```
uncertainty_rule d_1
  if      i1 is not high ( affirms 0.895; denies 3.20 )
     and  i2 is not high ( affirms 0.895; denies 9.00 ) then output is high .
```

```
uncertainty_rule d_2
  if      i1 is not high ( affirms 0.895; denies 3.20 )
     and  i2 is     high ( affirms 9.00; denies 0.895 ) then output is high .
uncertainty_rule d_3
  if      i1 is     high ( affirms 3.20; denies 0.895 )
     and  i2 is not high ( affirms 0.895; denies 9.00 ) then output is high .
uncertainty_rule d_4
  if      i1 is     high ( affirms 3.20; denies 0.895 )
     and  i2 is     high ( affirms 9.00; denies 0.895 ) then output is high .
```



And here is the full 3D graph:



Notice that Bayesian inference is highly non-linear. This is useful in modelling dynamic systems with sudden transitions, such as water temperature being raised to boiing point.

Certainty Theory

The final inference method we shall look at, Certainty Theory, is a simplified adaptation of Bayesian inference which was incorporated into the famous early expert system shell EMYCIN.

In Certainty Theory rules have the following structure:

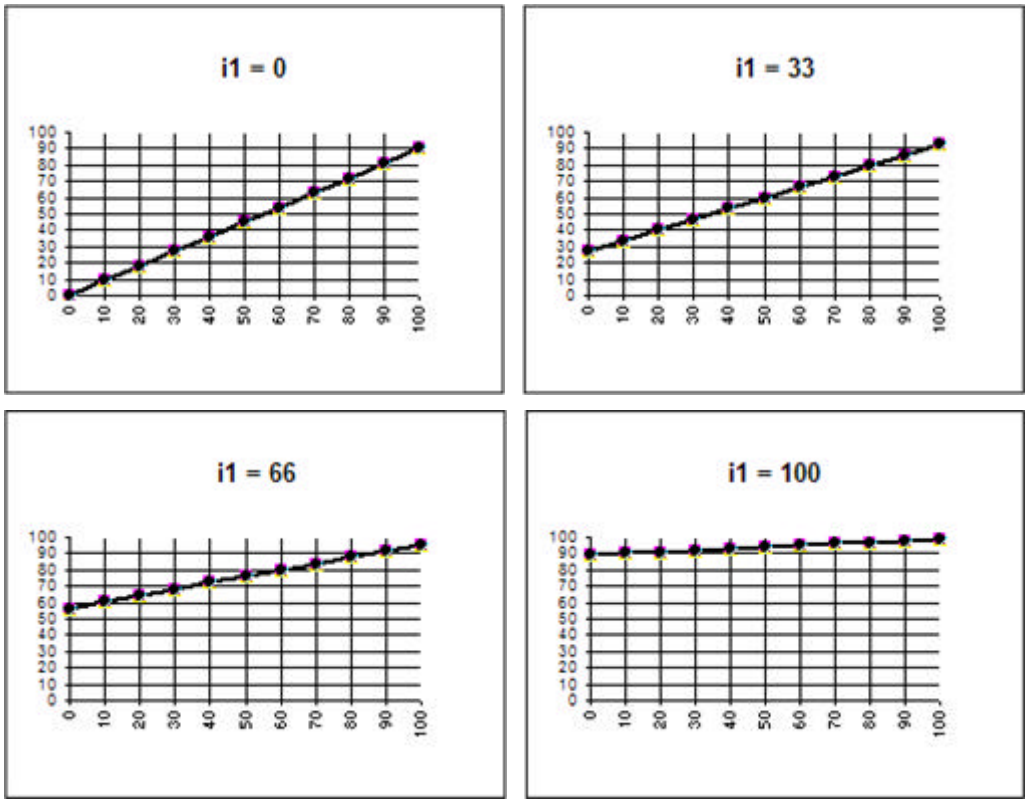
If then with certainty_factor CF

where CF ranges from -1 to +1 such that:

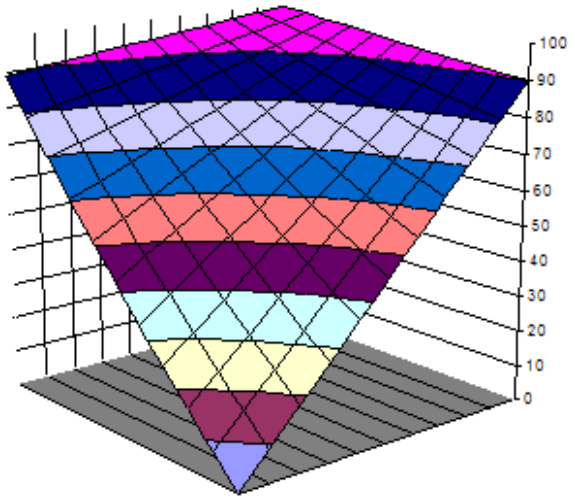
C(H)=1 corresponds to P(H)=1 (true)
C(H)=0 corresponds to P(H) is at its a prior value (unknown)
C(H)=-1 corresponds to P(H)=0 (false)

Example Four: Certainty Theory Inference

```
uncertainty_rule d_1
  if input1 is not high and input2 is not high then
    output is high with certainty factor -0.9 .
uncertainty_rule d_2
  if input1 is not high and input2 is high then
    output is high with certainty factor  0.1 .
uncertainty_rule d_3
  if input1 is high and input2 is not high then
    output is high with certainty factor  0.1 .
uncertainty_rule d_4
  if input1 is high and input2 is high then
    output is high with certainty factor  0.9 .
```



Here is the full 3D graph:



Conclusion

We have seen how rule-based systems dealing with uncertainty result in a different inferential topology to standard production rule systems. Instead of step-wise inference we get either smooth slopes or curves.

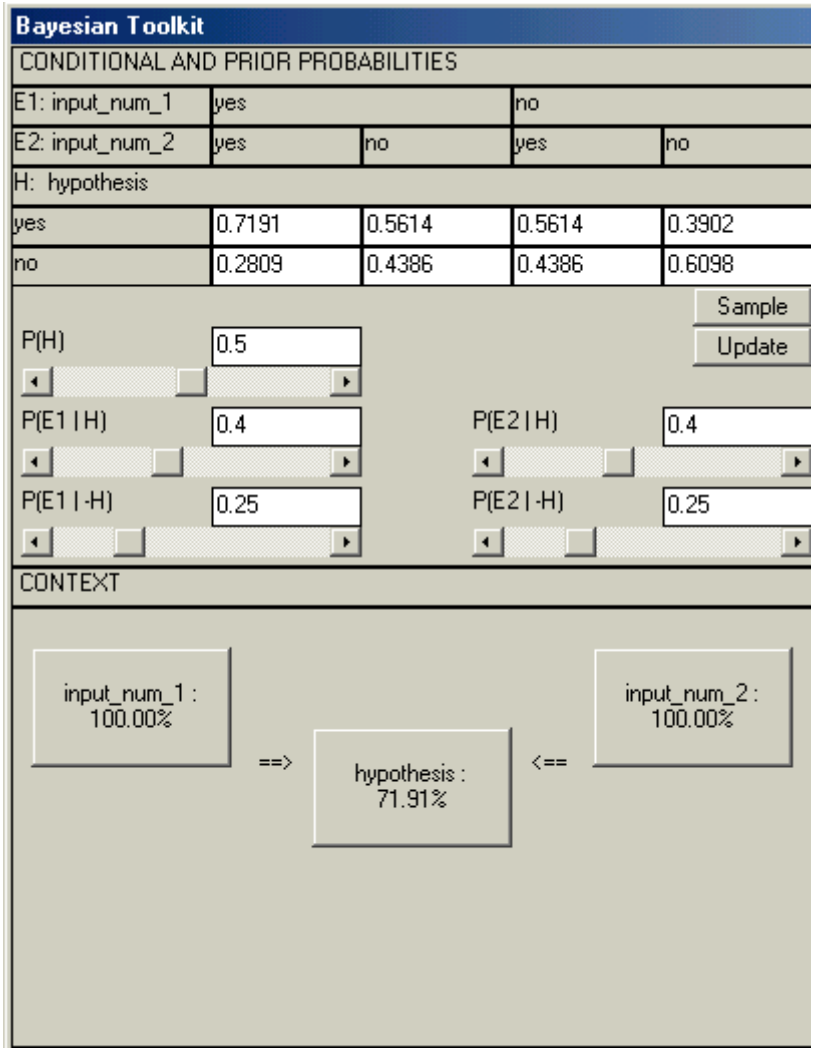
There are also contrasts between the differing uncertainty handling paradigms. Both Fuzzy and Bayesian systems result in curves, whereas systems based on Certainty theory result in non-curved slopes.

As for which technique is appropriate in which cases, if uncertainty is involved then either Fuzzy, Bayesian or Certainty Theory will be more accurate. Also Fuzzy techniques are useful for cases where the uncertainty is semantic and either Bayesian or Certainty Theory approaches are appropriate for cases involving evidential uncertainty.

In the latter cases, Bayesian networks have the advantage of mathematical rigour and the corresponing Affirms/Denies weights can be derived from databases of previous cases if these are available. Bayesian techniques are also more appropriate in dynamic domains which contain sudden transitions. On the other hand, Certainty Theory has the advantage that the Certainty Factors are more easily estimated. In the end, pragmatic considerations such as these are the only guide to what works best in a particular domain.

Chapter 12 - Probability Modulation and Linearity

To understand how to increase or decrease the linearity of a Bayesian network we need first to look at the five factors involved in a simple binary network like the one below:

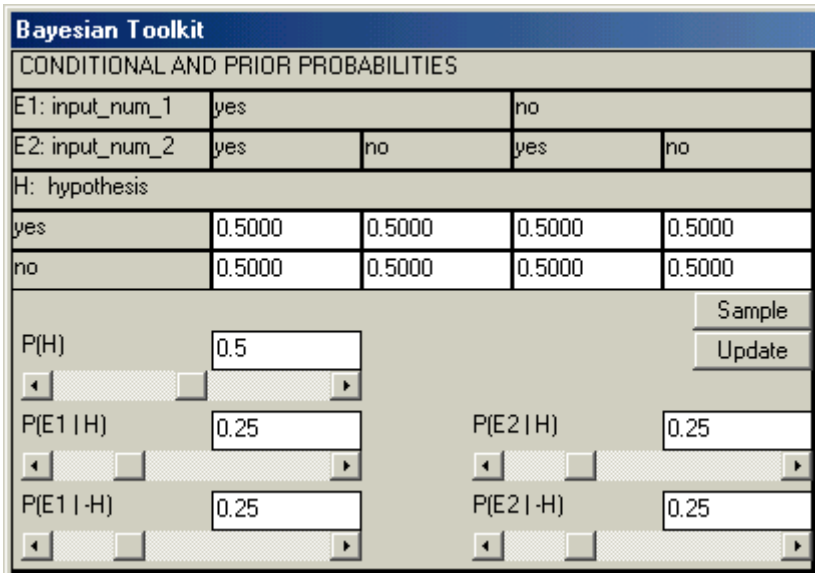


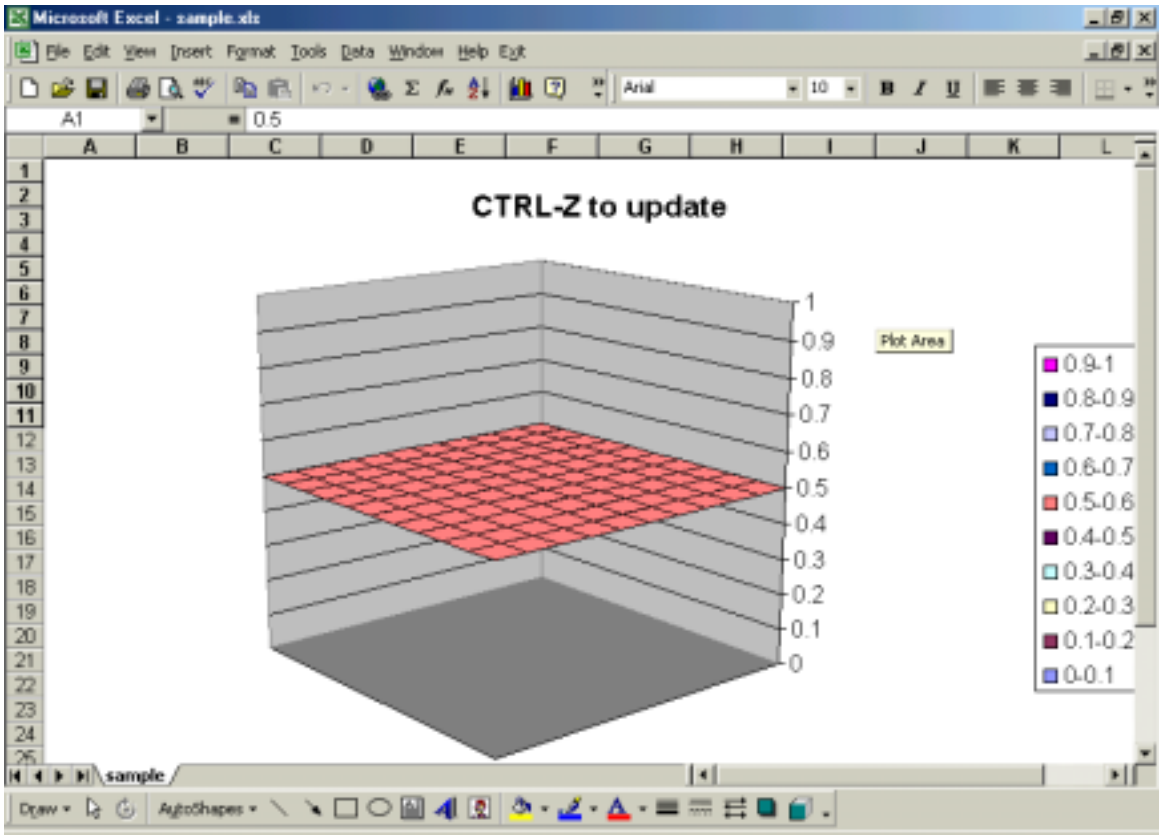
The five factors are P(H), P(E1|H), P(E2|H), P(E1|~H) and P(E2|~H).

To make matters simpler, however, we can look at symetrical networks where P(E1|H)= P(E2|H) and P(E1|~H) = P(E2|~H) thus reducing the relevant factors to three:

P(H), P(E|H) and P(E|~H).

Let us start with the simplest case of all, where the ratio between P(E|H) and P(E|~H) is equal to 1. This results in a flat plane with a vertical value equal to P(H):





Now we will increase the ratio of P(E|H) and P(E|~H) to 1.6:

Bayesian Toolkit

CONDITIONAL AND PRIOR PROBABILITIES

E1: input_num_1	yes		no	
E2: input_num_2	yes	no	yes	no
H: hypothesis				
yes	0.7191	0.5614	0.5614	0.3902
no	0.2809	0.4386	0.4386	0.6098

Sample

Update

P(H) 0.5

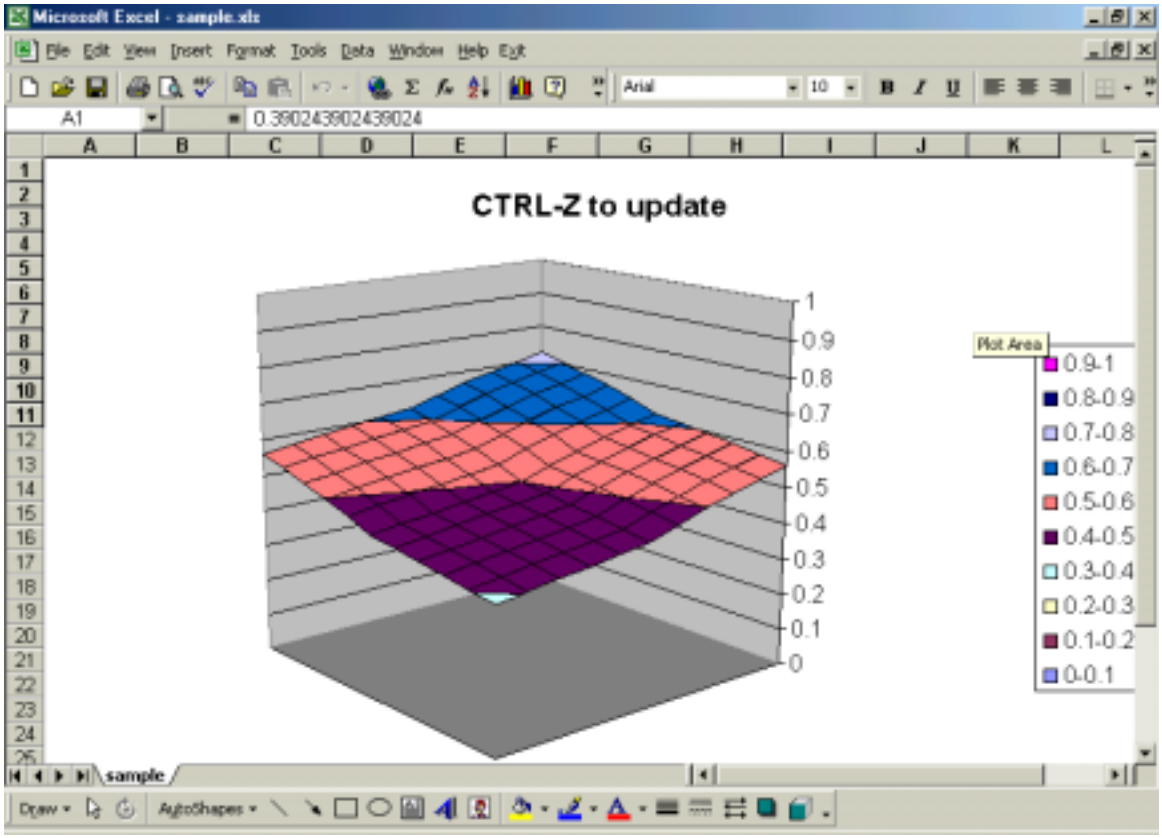
P(E1|H) 0.4

P(E2|H) 0.4

P(E1|~H) 0.25

P(E2|~H) 0.25

The corresponding graph is what we might call a flying carpet:



Note that it is the ratio of P(E|H) to P(E|~H) that is relevant here as can be seen by reducing both by a factor of ten:

Bayesian Toolkit

CONDITIONAL AND PRIOR PROBABILITIES

E1: input_num_1	yes		no	
E2: input_num_2	yes	no	yes	no
H: hypothesis				
yes	0.7191	0.6117	0.6117	0.4922
no	0.2809	0.3883	0.3883	0.5078

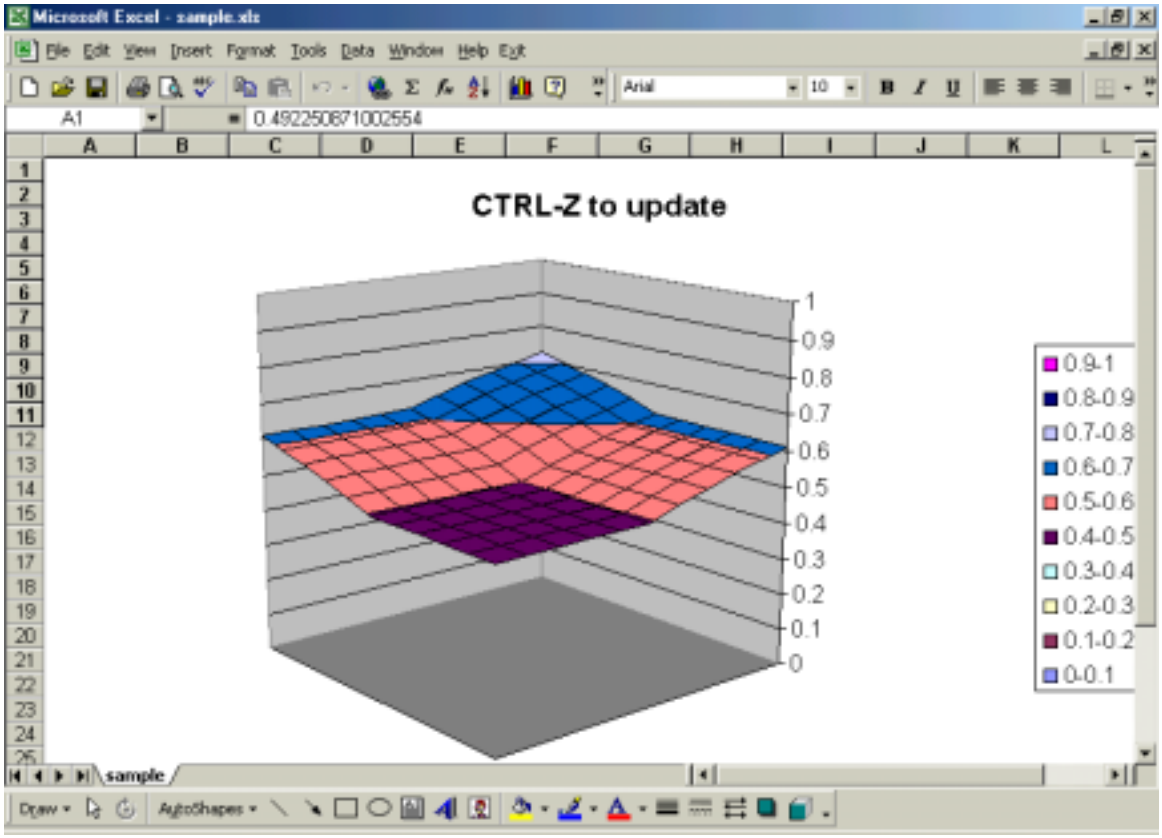
Sample

Update

P(H) 0.5

P(E1 | H) 0.04 P(E2 | H) 0.04

P(E1 | ~H) 0.025 P(E2 | ~H) 0.025



However, this is only true when the ratio between P(E|H) and P(E|~H) is small (1.6 to 1 here).

We shall see later what effect this has when the ratio is higher.

The effect of changing P(H) in this network is merely to position the flying carpet vertically:

Bayesian Toolkit

CONDITIONAL AND PRIOR PROBABILITIES

E1: input_num_1	yes		no	
E2: input_num_2	yes	no	yes	no
H: hypothesis				
yes	0.2215	0.1490	0.1490	0.0972
no	0.7785	0.8510	0.8510	0.9028

Sample

Update

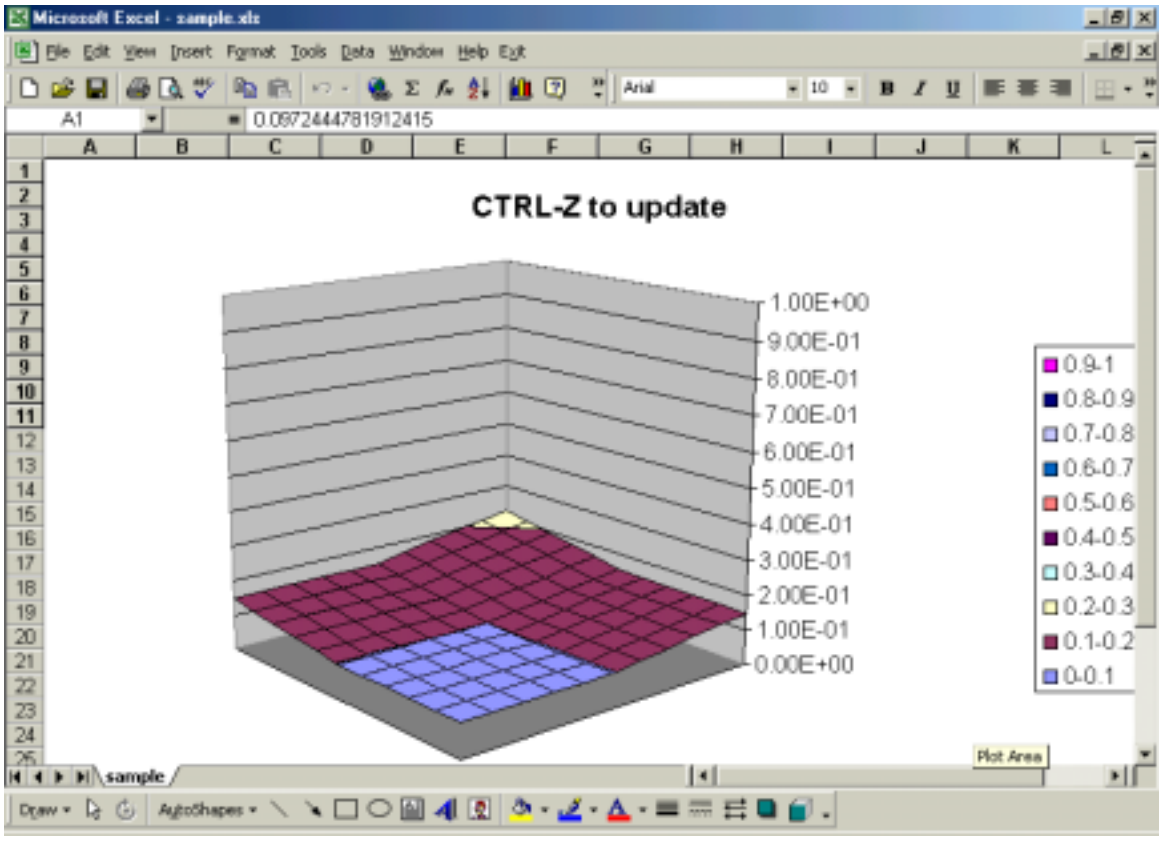
P(H) 0.1

P(E1 | H) 0.04

P(E2 | H) 0.04

P(E1 | -H) 0.025

P(E2 | -H) 0.025



If we want to decrease the linearity in the model, all we need to do is to increase the ratio between P(E|H) and P(E|~H). Firstly a small increase:

Bayesian Toolkit

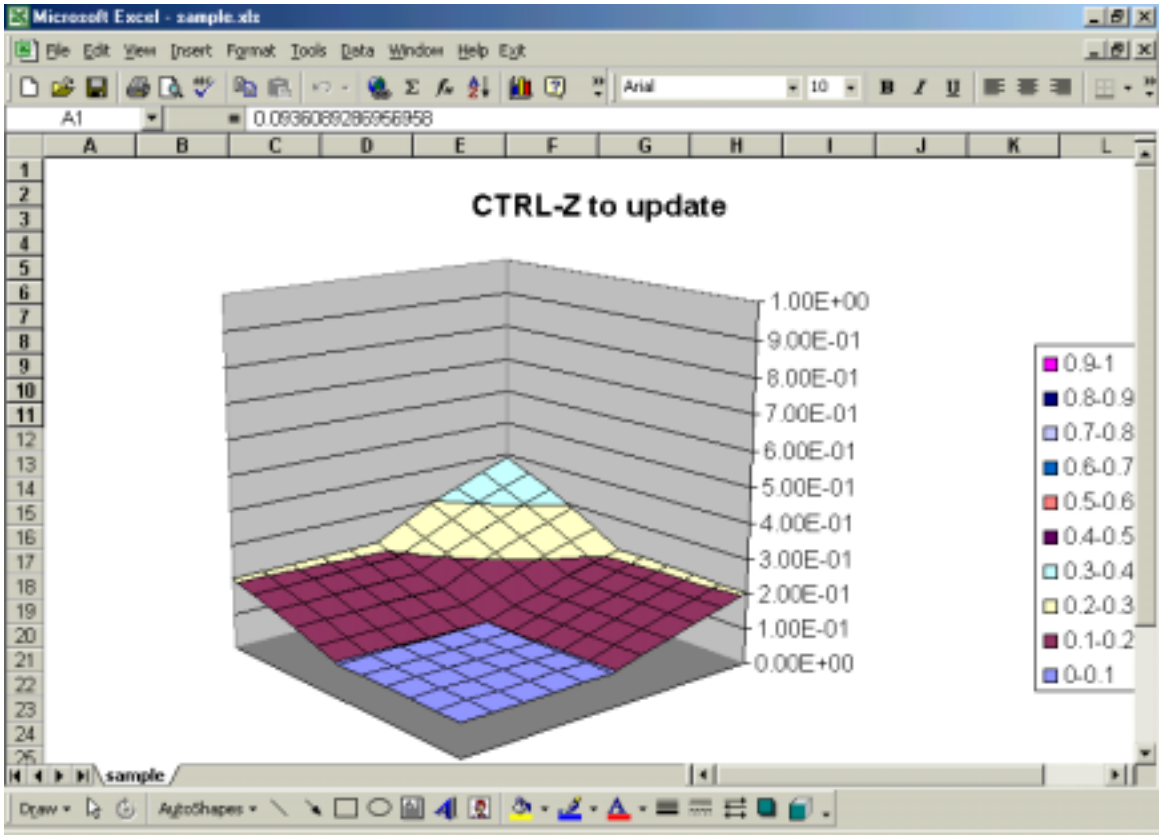
CONDITIONAL AND PRIOR PROBABILITIES				
E1: input_num_1	yes		no	
E2: input_num_2	yes	no	yes	no
H: hypothesis				
yes	0.3902	0.2045	0.2045	0.0936
no	0.6098	0.7955	0.7955	0.9064

Sample
Update

P(H) 0.1

P(E1 | H) 0.06 P(E2 | H) 0.06

P(E1 | -H) 0.025 P(E2 | -H) 0.025



Next a more significant increase:

Bayesian Toolkit

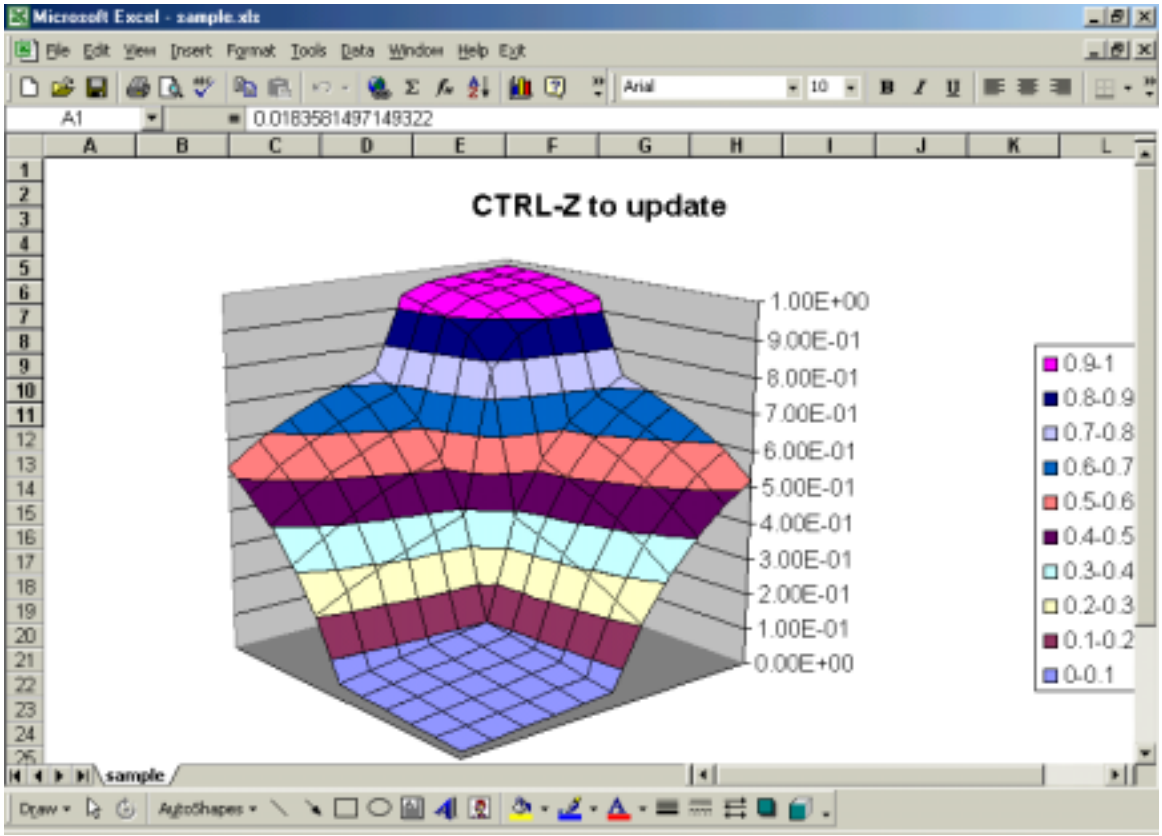
CONDITIONAL AND PRIOR PROBABILITIES				
E1: input_num_1	yes		no	
E2: input_num_2	yes	no	yes	no
H: hypothesis				
yes	0.9846	0.5224	0.5224	0.0184
no	0.0154	0.4776	0.4776	0.9816

Sample
Update

P(H) 0.1

P(E1 | H) 0.6 P(E2 | H) 0.6

P(E1 | -H) 0.025 P(E2 | -H) 0.025



Finally a very significant increase:

Bayesian Toolkit

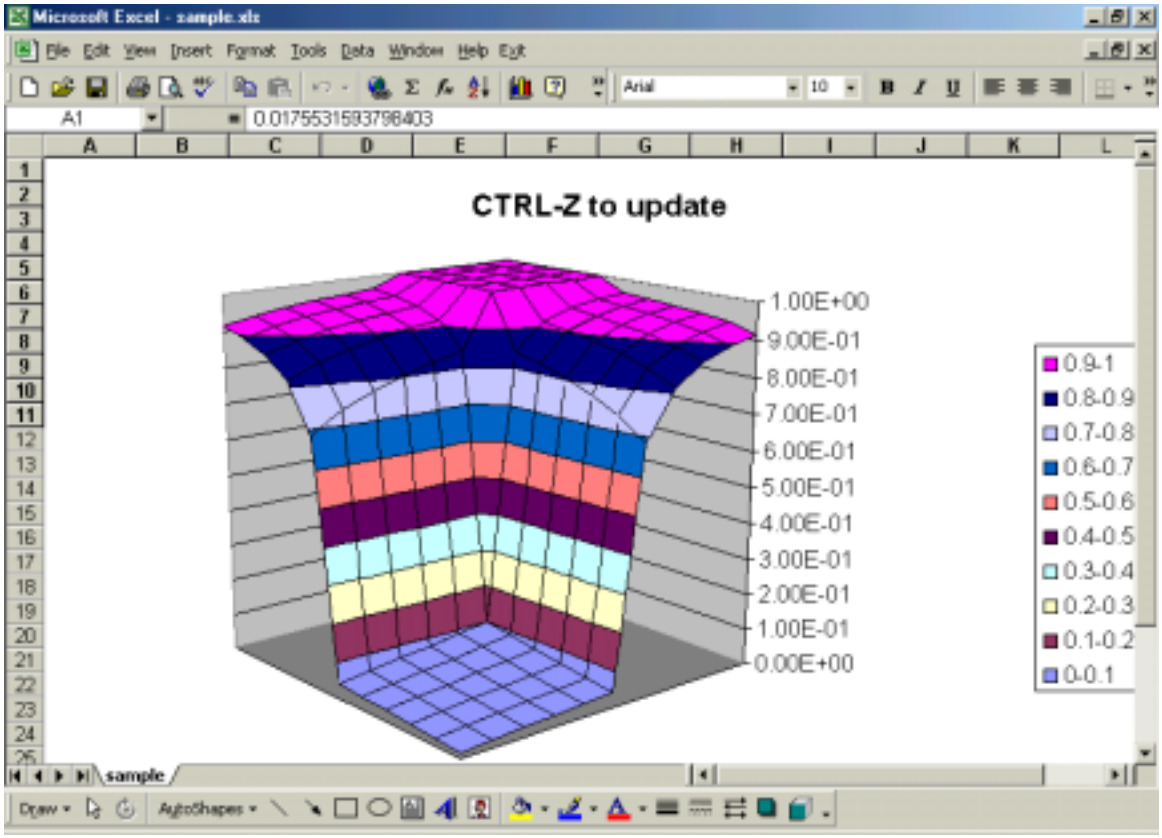
CONDITIONAL AND PRIOR PROBABILITIES				
E1: input_num_1	yes		no	
E2: input_num_2	yes	no	yes	no
H: hypothesis				
yes	0.9998	0.9145	0.9145	0.0176
no	0.0002	0.0855	0.0855	0.9824

Sample
Update

P(H) 0.1

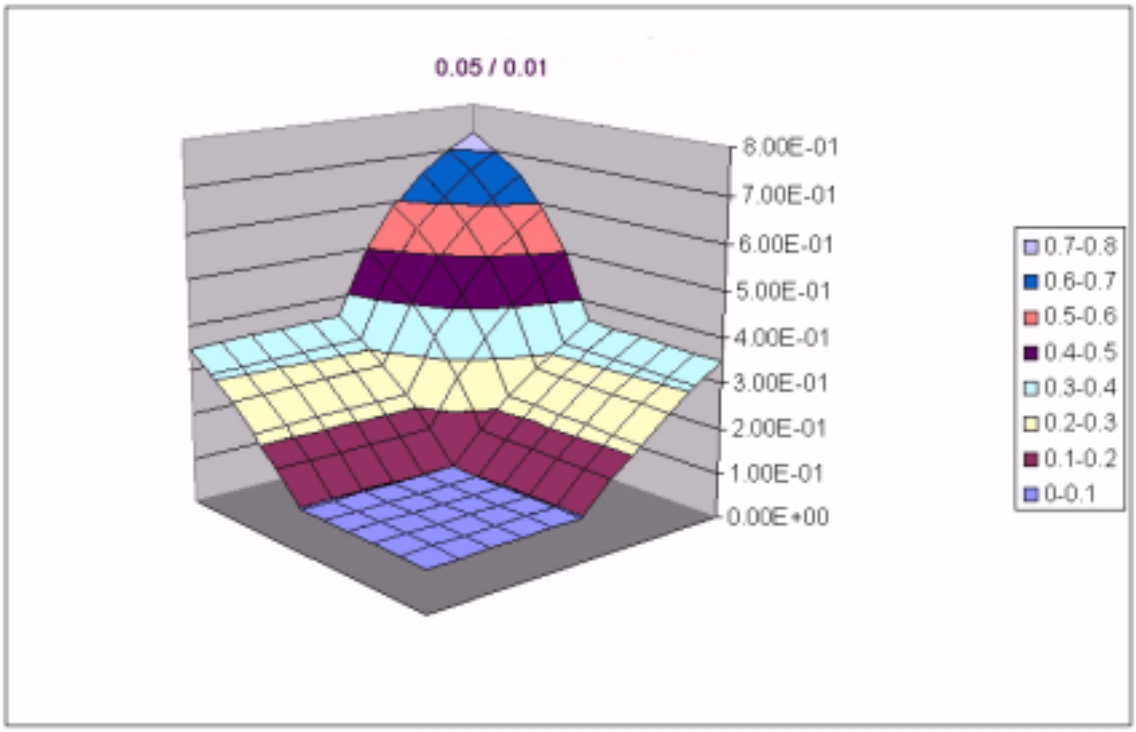
P(E1 | H) 0.6 P(E2 | H) 0.6

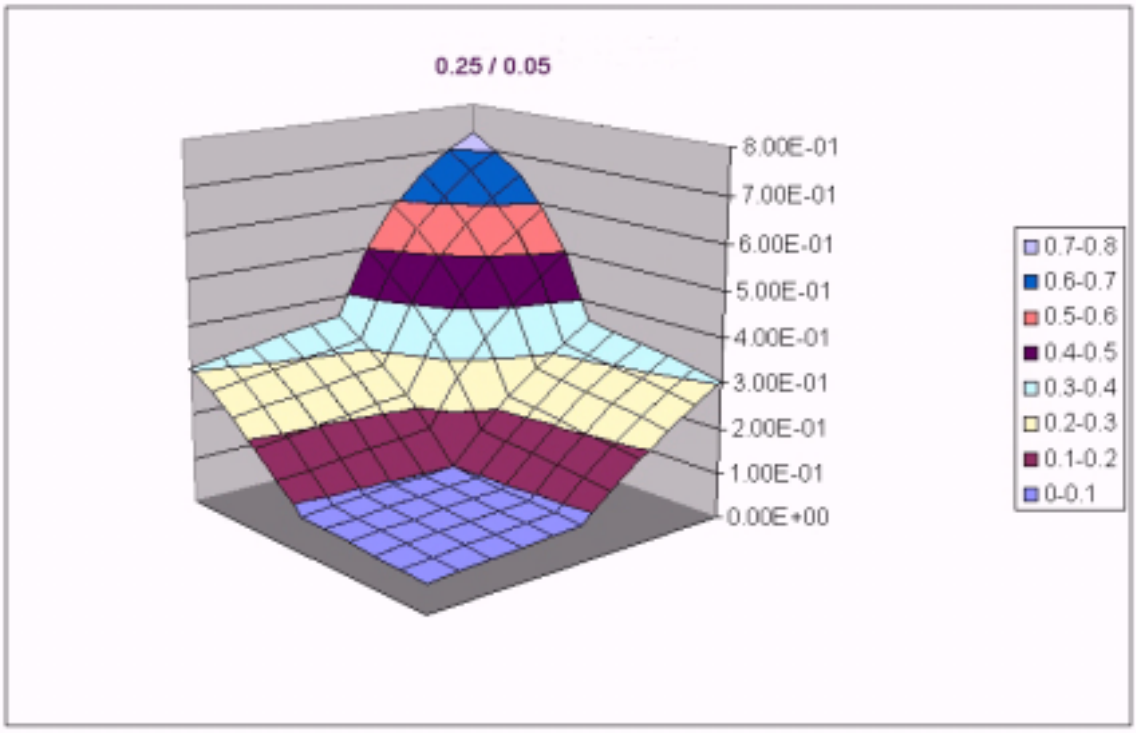
P(E1 | -H) 0.0025 P(E2 | -H) 0.0025

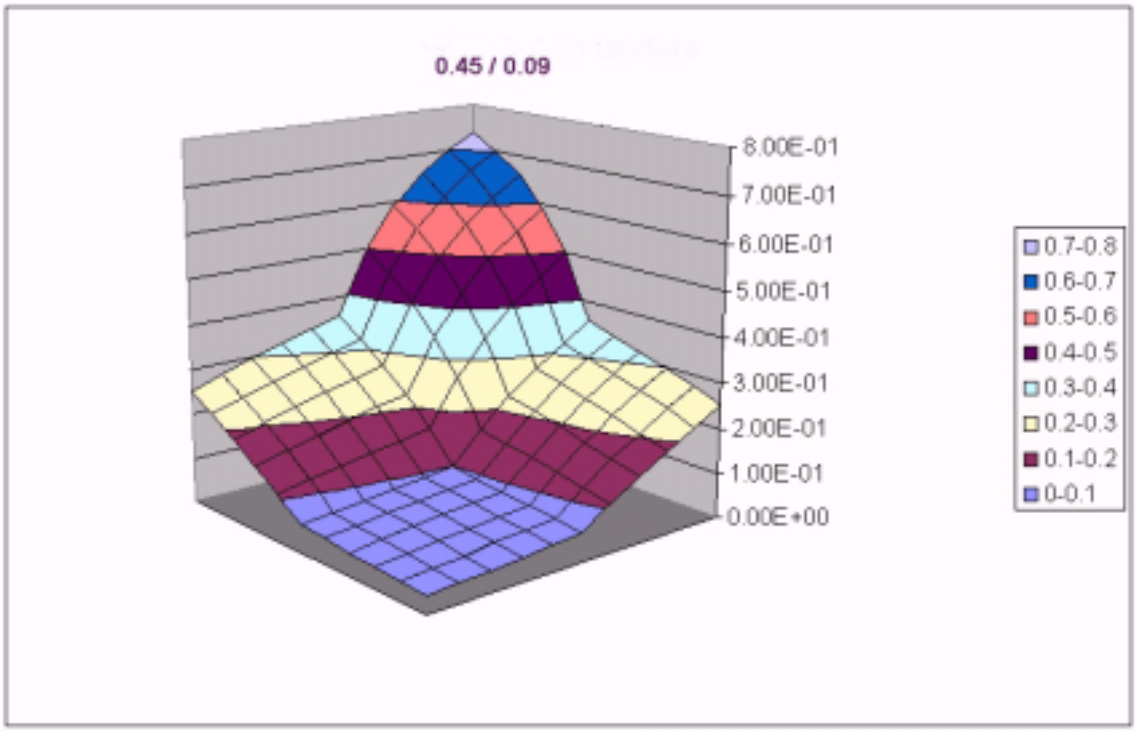


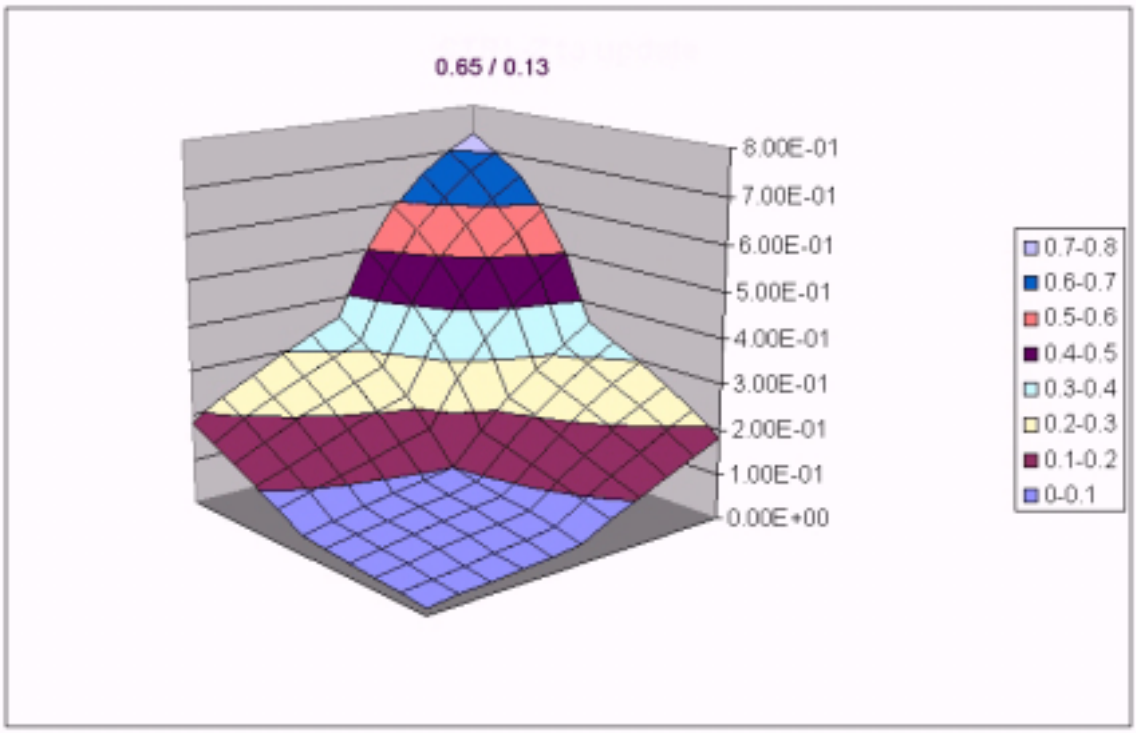
Earlier we saw that with a ratio of 1.6 to 1, the magnitude of P(E|H) and P(E|~H) is insignificant. As the ratio increases however, the impact of such a change in magnitude becomes greater.

The following graphs are all of a network with a ratio of 5 to 1 at differing levels of magnitude:


0.05 / 0.01
8.00E-01
7.00E-01
6.00E-01
5.00E-01
4.00E-01
3.00E-01
2.00E-01
1.00E-01
0.00E+00
0.7-0.8
0.6-0.7
0.5-0.6
0.4-0.5
0.3-0.4
0.2-0.3
0.1-0.2
0-0.1


0.25 / 0.05
8.00E-01
7.00E-01
6.00E-01
5.00E-01
4.00E-01
3.00E-01
2.00E-01
1.00E-01
0.00E+00
0.7-0.8
0.6-0.7
0.5-0.6
0.4-0.5
0.3-0.4
0.2-0.3
0.1-0.2
0-0.1


0.45 / 0.09
8.00E-01
7.00E-01
6.00E-01
5.00E-01
4.00E-01
3.00E-01
2.00E-01
1.00E-01
0.00E+00
0.7-0.8
0.6-0.7
0.5-0.6
0.4-0.5
0.3-0.4
0.2-0.3
0.1-0.2
0-0.1


0.65 / 0.13
8.00E-01
7.00E-01
6.00E-01
5.00E-01
4.00E-01
3.00E-01
2.00E-01
1.00E-01
0.00E+00
0.7-0.8
0.6-0.7
0.5-0.6
0.4-0.5
0.3-0.4
0.2-0.3
0.1-0.2
0-0.1

Conclusions

We have seen here that the most significant factor in determining the linearity of a Bayesian network is the ratio between P(E|H) and P(E|~H).

In the networks we are building for the FRISC project, we will be constrained by the available data as to which ratios we use at all except the intermediate nodes. These however are crucial for the linearity we require, and as is shown in the latest demonstration model, a ratio of around 1.6 to 1 seems quite appropriate.

Since we want reasonably high results at the conclusion level, a high P(H) must also be used for the intermediate nodes, and we have found that 0.5 works reasonably well for this purpose.

The values we have used are as follows:

P(H) = 0.5

P(E|H) = 0.4

P(E|~H) = 0.25

Probability Modulation and Non-linearity in Bayesian Networks

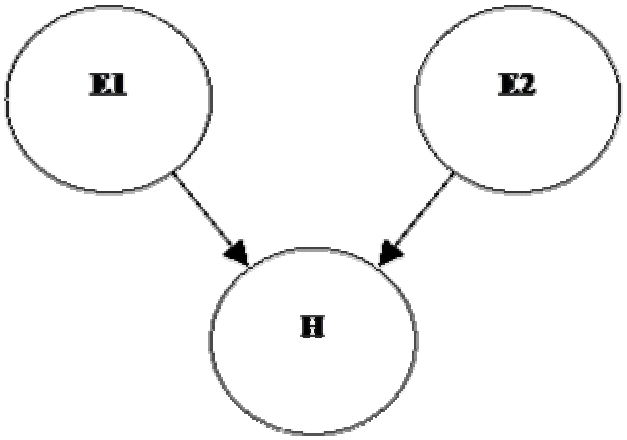
By Clive Spenser & Charles Langley

Introduction

This document is the second of a series on artificial inference. The first document in the series looked at the differences between production rules, fuzzy logic, Bayesian networks and certainty theory. Here we concentrate on Bayesian networks, and in particular at the non-linearity of their inference. We start by looking at how probabilities are represented in Bayesian network tools, *contrasting conditional probability tables* and *affirms/denies weights*. Following this we examine seven different levels of non-linearity using both three dimensional graphs and two dimensional slices of these graphs.

Conditional Probabilities, Affirms/Denies Weights and Conditional Probability Tables

To understand how to increase or decrease the linearity of a Bayesian network we need first to look at the five factors involved in a simple binary network like the one below:



The five factors are:

P(H)	The prior probability of the hypothesis H
P(E1|H)	The probability of evidence E1 when H is true
P(E2|H)	The probability of evidence E2 when H is true
P(E1|~H)	The probability of evidence E1 when H is false
P(E2|~H)	The probability of evidence E2 when H is false

To make matters simpler, however, we can look at symmetrical networks where

P(E1|H) = P(E2|H) and
P(E1|~H) = P(E2|~H)
thus reducing the relevant factors to three:
P(H), P(E|H) and P(E|~H).
To implement a Bayesian knowledge base using LPA software, all we need to do is to convert these conditional probabilities to affirms and denies weights according to the equations below.

The affirms weight is calculated as:

$$A = \frac{P(E \mid H)}{P(E \mid \sim H)}$$

and the denies weight is calculated as:

$$D = \frac{1 - (P(E \mid H))}{1 - P(E \mid \sim H)}$$

These weights can then be used in rules such as:

uncertainty_rule r1
if e1 is high (affirms 3.20; denies 0.895) and e2 is high (affirms 9.00; denies 0.895) then h is high .

Some Bayesian tools produced by other companies, such as Hugin and Nettica express conditional probabilities by means of what are called Conditional Probability Tables (CPTs). Here is a typical CPT.

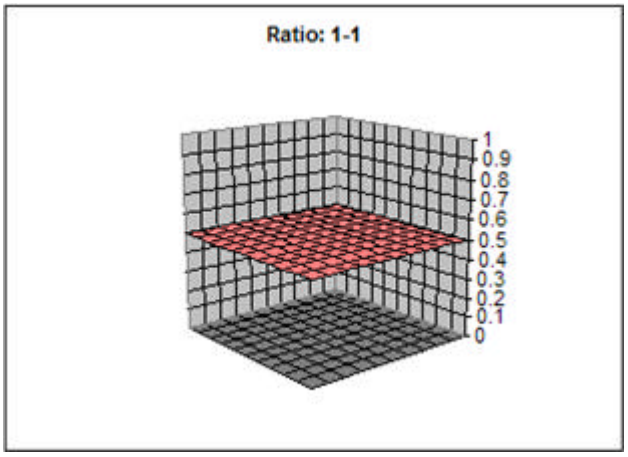
E1	yes	yes	no	no
E2	yes	no	yes	no
H				
yes	0.71	0.56	0.56	0.29
no	0.28	0.44	0.44	0.61

CPTs such as this are designed to express P(H|E1) etc. rather than P(E1|H). Logic Programming Associates uses the latter rather than the former, but one can be calculated from the other by means of Bayes' theorem. The advantage of LPA's approach is that it enables these values to be obtained directly from databases of previous cases.

Example One

Let us start with the simplest case of all, where the ratio between P(E|H) and P(E|~H) is equal to 1. This results in a flat plane with a vertical value equal to P(H):

E1	yes	yes	no	no
E2	yes	no	yes	no
H				
yes	0.5	0.5	0.5	0.5
no	0.5	0.5	0.5	0.5
P(H)	0.5			
P(E1 | H)	0.25			
P(E1 | ~H)	0.25			
P(E2 | H)	0.25			
P(E2 | ~H)	0.25			

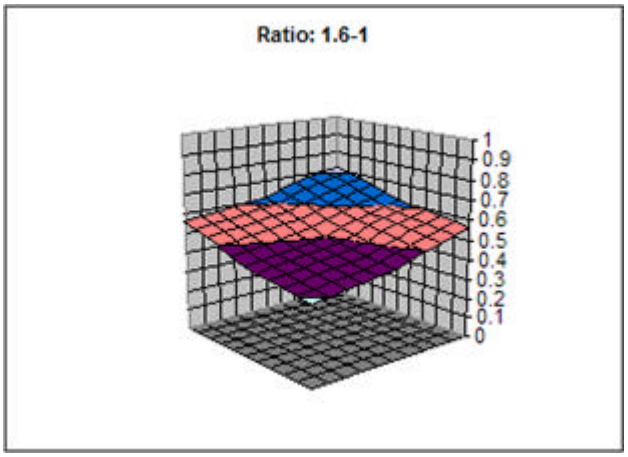


Example Two

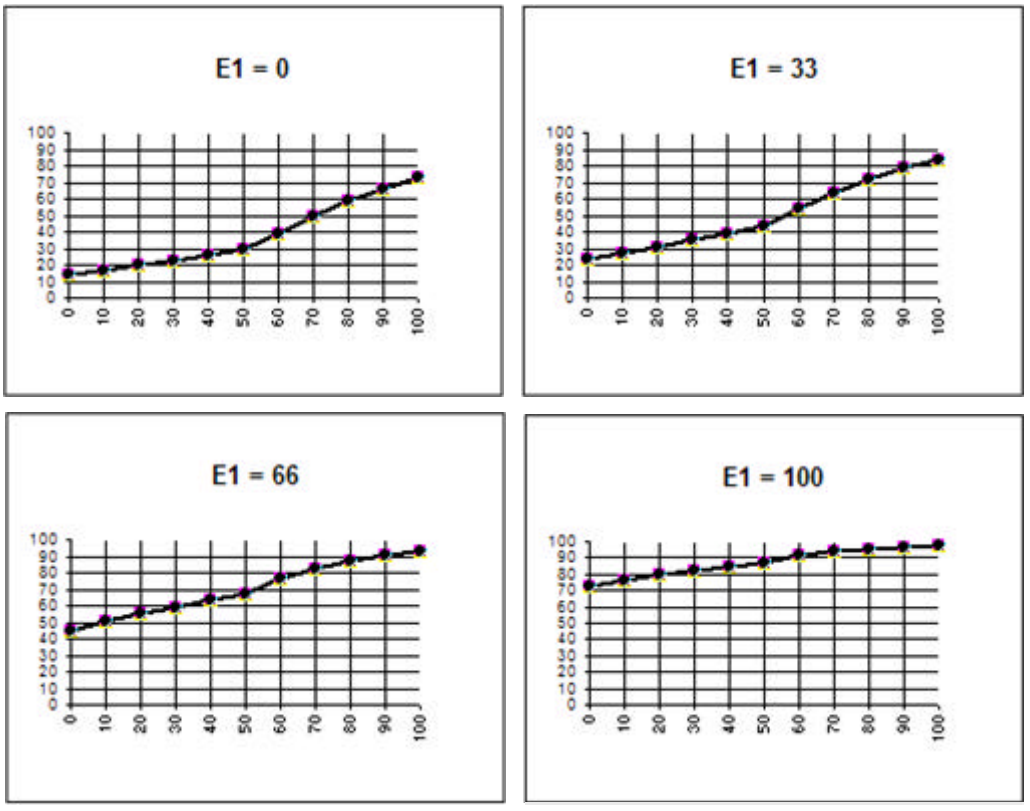
Now we will increase the ratio of P(E|H) and P(E|~H) to 1.6:

The corresponding graph is what we might call a flying carpet:

E1	yes	yes	no	no
E2	yes	no	yes	no
H				
yes	0.72	0.56	0.56	0.29
no	0.28	0.44	0.44	0.61
P(H)	0.5			
P(E1 | H)	0.4			
P(E1 | ~H)	0.25			
P(E2 | H)	0.4			
P(E2 | ~H)	0.25			



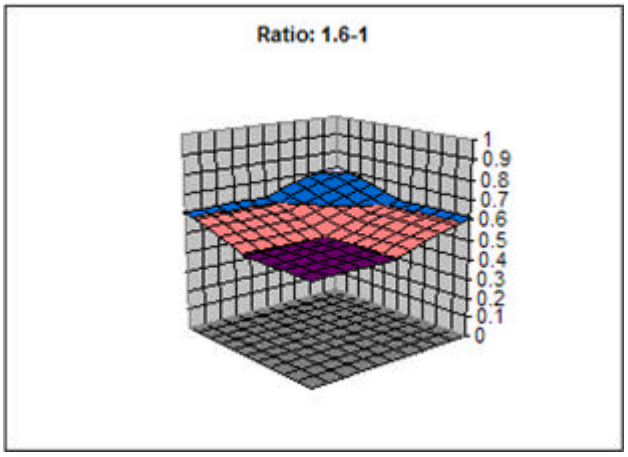
We will examine this shape more closely by looking at some of the two-dimensional slices of this three-dimensional graph.



These are curves with a fairly low non-linearity compared with what we shall see below.

Example Three

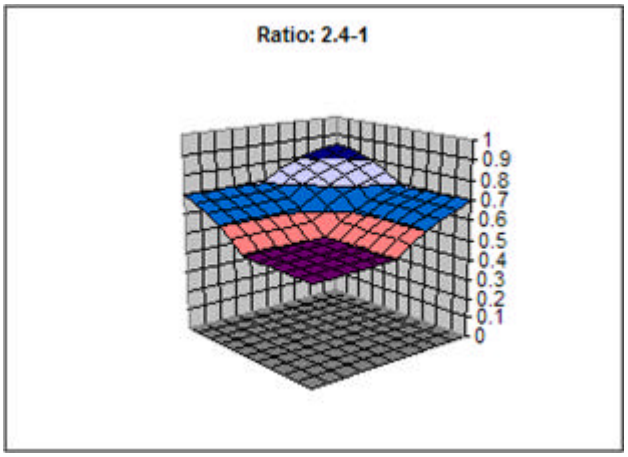
E1	yes	yes	no	no
E2	yes	no	yes	no
H				
yes	0.72	0.61	0.61	0.49
no	0.28	0.39	0.39	0.51
P(H)	0.5			
P(E1 | H)	0.04			
P(E1 | ~H)	0.025			
P(E2 | H)	0.04			
P(E2 | ~H)	0.025			



Note that it is the ratio of P(E|H) to P(E|~H) that is relevant here as can be seen by reducing both by a factor of ten:

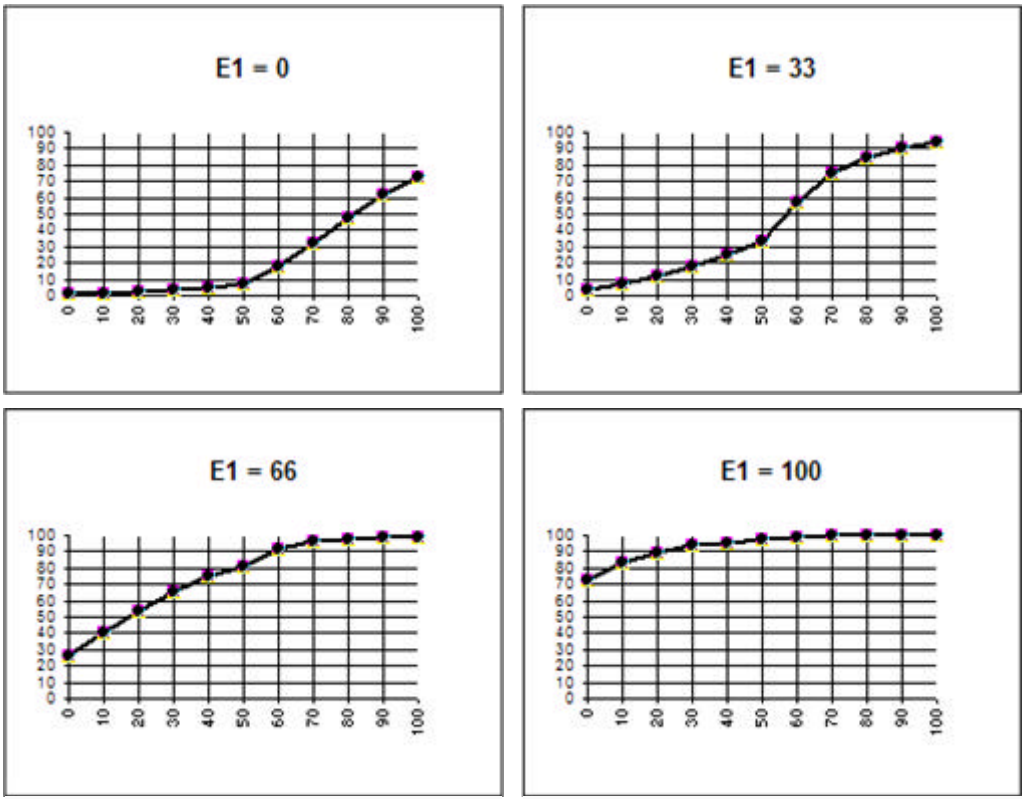
Example Four

E1	yes	yes	no	no
E2	yes	no	yes	no
H				
yes	0.85	0.56	0.56	0.22
no	0.15	0.44	0.44	0.78
P(H)	0.5			
P(E1 | H)	0.6			
P(E1 | ~H)	0.25			
P(E2 | H)	0.6			
P(E2 | ~H)	0.25			



If we want to decrease the linearity in the model, all we need to do is to increase the ratio between P(E|H) and P(E|~H). Firstly a small increase:

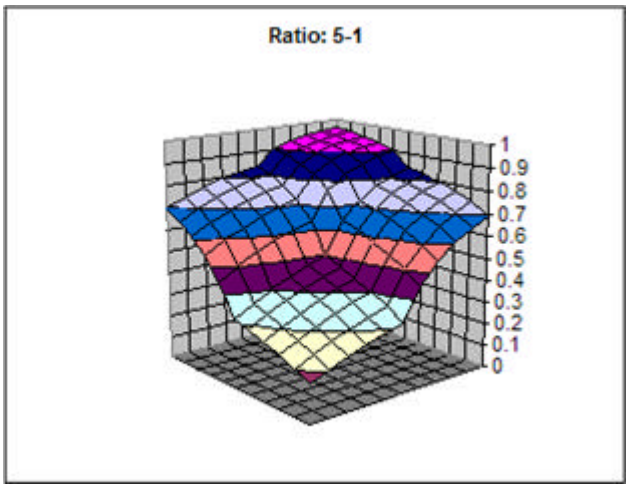
Here are the two-dimensional slices:



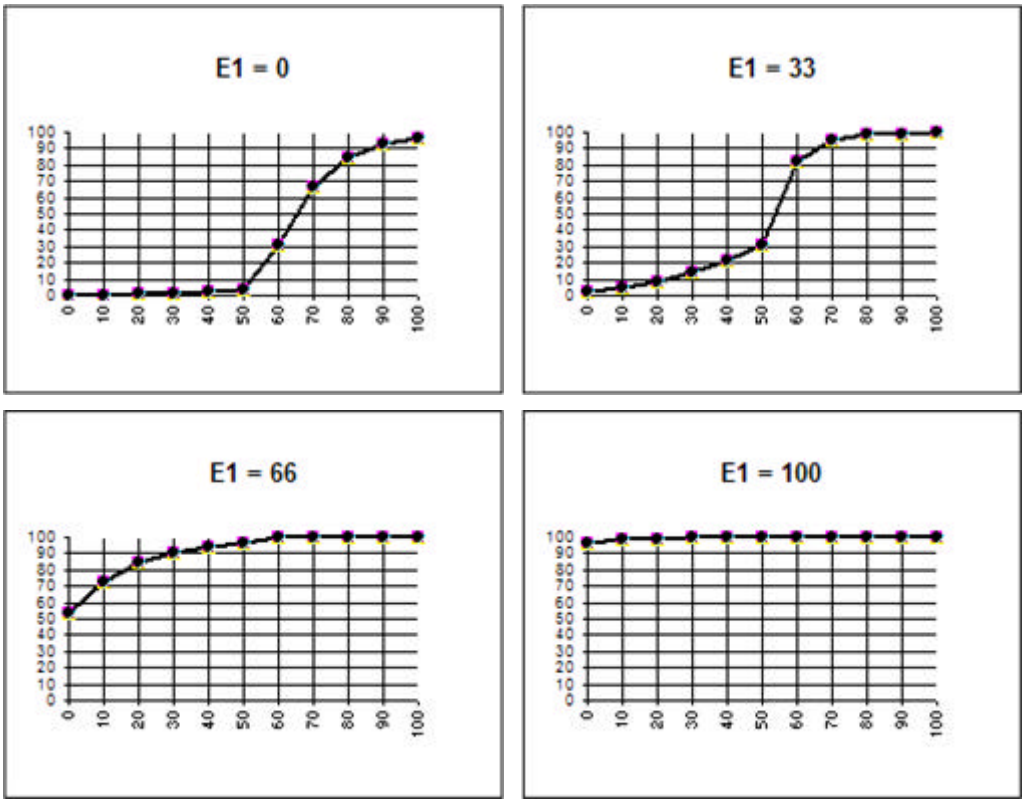
In the graph above which represents E1 held at 33% we now see the classical ‘S’ curve of Bayesian non-linearity.

Example Five

E1	yes	yes	no	no
E2	yes	no	yes	no
H				
yes	0.96	0.69	0.69	0.17
no	0.04	0.31	0.31	0.83
P(H)	0.5			
P(E1 | H)	0.6			
P(E1 | ~H)	0.12			
P(E2 | H)	0.6			
P(E2 | ~H)	0.12			

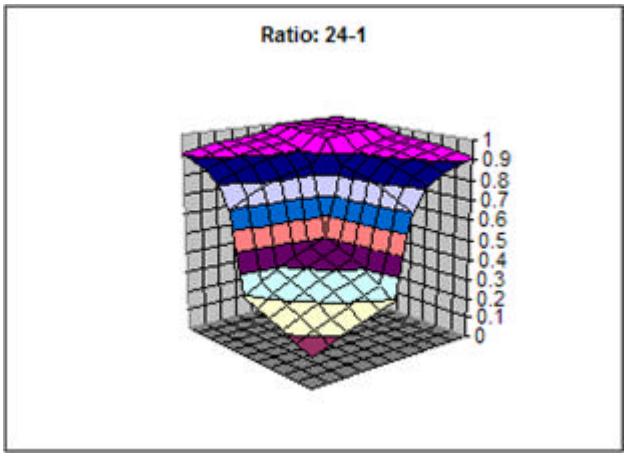


Another increase:

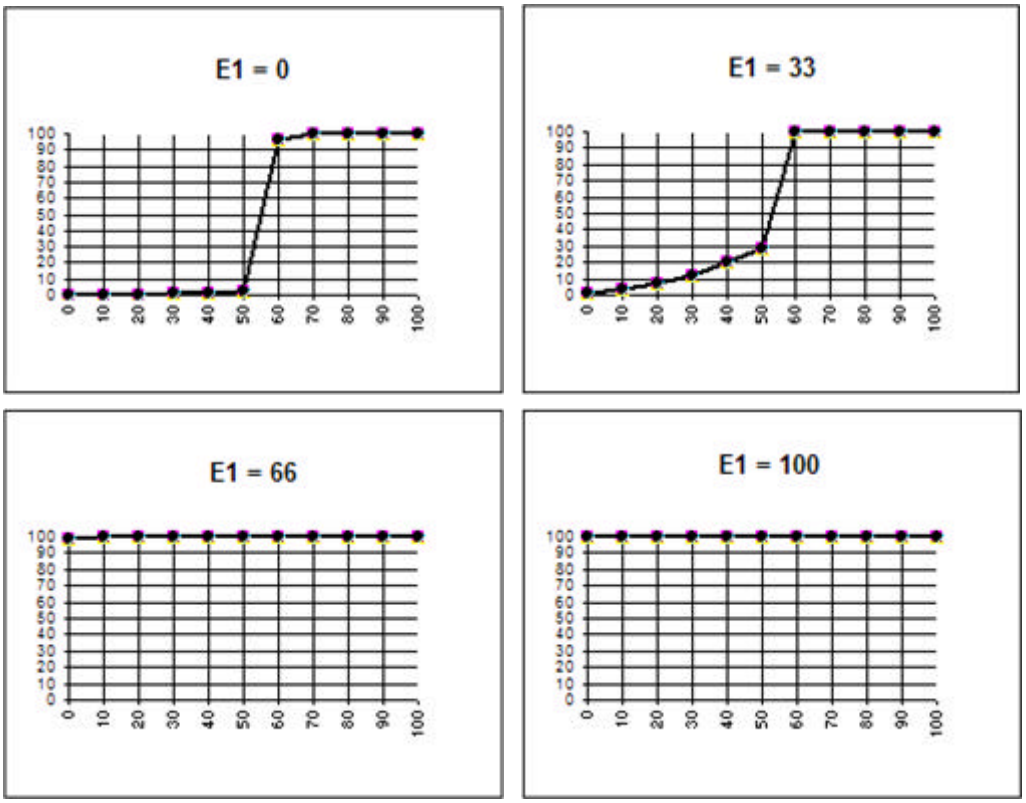


Example Six

E1	yes	yes	no	no
E2	yes	no	yes	no
H				
yes	1.00	0.90	0.90	0.14
no	0.00	0.10	0.10	0.86
P(H)	0.5			
P(E1 | H)	0.6			
P(E1 | ~H)	0.025			
P(E2 | H)	0.6			
P(E2 | ~H)	0.025			



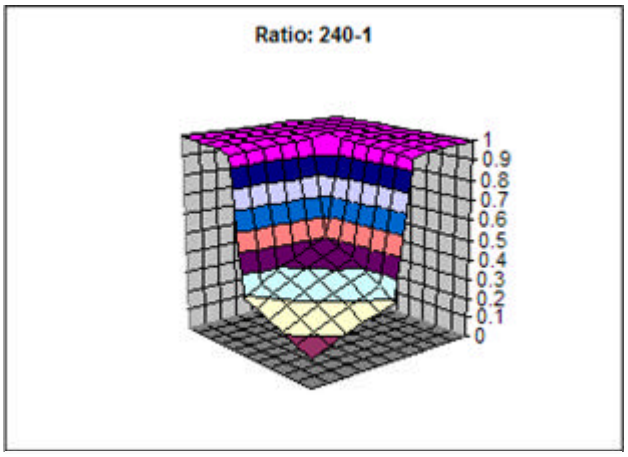
Next a more significant increase to the ratio of P(E | H) to P(E | ~H):



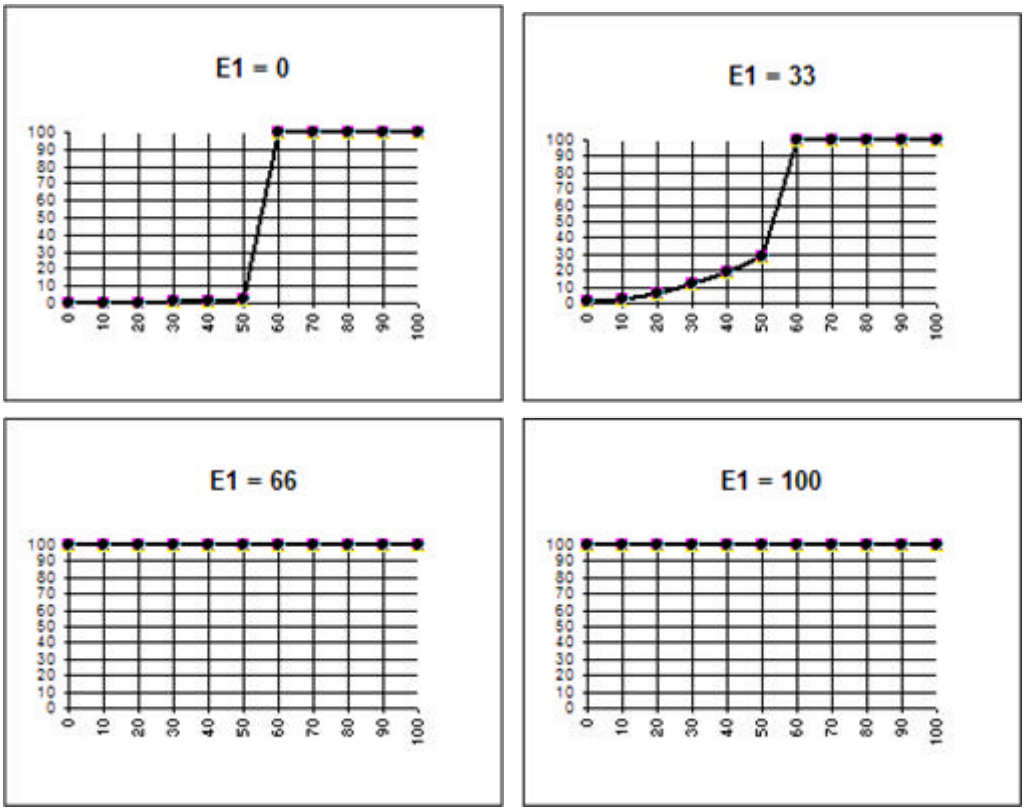
Example Seven

E1	yes	yes	no	no
E2	yes	no	yes	no
H				

yes	1.00	0.99	0.99	0.14
no	0.00	0.01	0.01	0.86
P(H)	0.5			
P(E1 | H)	0.6			
P(E1 | ~H)	0.0025			
P(E2 | H)	0.6			
P(E2 | ~H)	0.0025			



Finally a very significant increase:



Conclusions

We have seen here that the most significant factor in determining the linearity of a Bayesian network is the ratio between P(E|H) and P(E|~H). It is this ratio which defines the **Affirms** weight (and indirectly the **Denies** weight) used to represent Bayesian rules using LPA software.

Defuzzification Options in Flex

By Clive Spenser & Charles Langley

Consider the following (partial) fuzzy logic program in Flex:

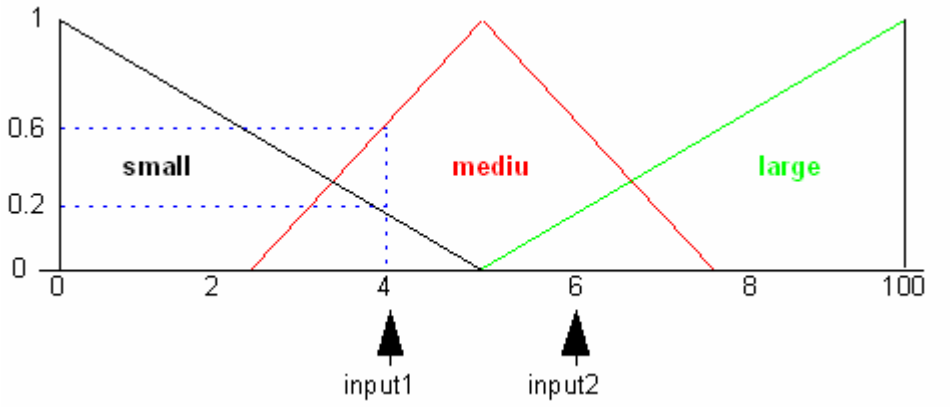
```
% fuzztest.ksl

% (1) Fuzzy set definitions
fuzzy_variable input1 ;
  ranges from 0 to 100 ;
  fuzzy_set small is \ shaped and linear at 0, 50 ;
  fuzzy_set medium is /\ shaped and linear at 25, 50, 75 ;
  fuzzy_set large is / shaped and linear at 50, 100 ;

fuzzy_variable input2 ;
  ranges from 0 to 100 ;
  fuzzy_set small is \ shaped and linear at 0, 50 ;
  fuzzy_set medium is /\ shaped and linear at 25, 50, 75 ;
  fuzzy_set large is / shaped and linear at 50, 100 ;

fuzzy_variable output ;
  ranges from 0 to 100 ;
  fuzzy_set small is \ shaped and linear at 0, 50 ;
  fuzzy_set medium is /\ shaped and linear at 25, 50, 75 ;
  fuzzy_set large is / shaped and linear at 50, 100 ;

  defuzzify using all memberships
    and mirror rule
    and shrinking .

% (2) Fuzzy Rule definitions
fuzzy_rule d_1 if input1 is small and input2 is small then
  output is small.
fuzzy_rule d_2 if input1 is small and input2 is medium then
  output is small.
fuzzy_rule d_3 if input1 is small and input2 is large then
  output is small.
fuzzy_rule d_4 if input1 is medium and input2 is small then
  output is medium.
fuzzy_rule d_5 if input1 is medium and input2 is medium then
  output is medium.
fuzzy_rule d_6 if input1 is medium and input2 is large then
  output is large.
fuzzy_rule d_7 if input1 is large and input2 is small then
  output is small.
fuzzy_rule d_8 if input1 is large and input2 is medium then
  output is medium.
fuzzy_rule d_9 if large is small and input2 is large then
  output is large.
```

Fuzzy Inference

To determine the value of the output in the above fuzzy program there are three distinct steps:

1. fuzzify the values of the two input variables
2. propagate the rules to determine the fuzzy value of the output variable
3. defuzzify the output variable.

Fuzzifying the value of a variable is straightforward, simply determining the degree of membership that the value has (between 0 and 1) for each of the variable's component fuzzy sets according to their shapes (membership functions).



For example, let us suppose that the value of input1 is 40 and the value of input 2 is 60. We might intuitively say that input1 is somewhat small and somewhat medium but not at all large whilst input2 is somewhat medium and somewhat large and not at all small.

If we set the fuzzy_trace mechanism on, this is what we see at the fuzzification stage:

```
Mem. : UPDATE     : (input1 is small) = 0.2
Mem. : UPDATE     : (input1 is medium) = 0.6
Mem. : UPDATE     : (input1 is large) = 0
Mem. : FUZZIFY    : input2 = 60
Mem. : UPDATE     : (input2 is small) = 0
Mem. : UPDATE     : (input2 is medium) = 0.6
Mem. : UPDATE     : (input2 is large) = 0.2
```

Here we see that **input1** has been assigned the possibility values of **0.2** for small, **0.6** for medium and 0 for large. This is its degree of membership of these three sets.

Rule Propagation

After fuzzification the rules are propagated. First d_1. Since input1 is partially small but input2 is not at all small we get:

```
fuzzy_rule d_1 if input1 is small and input2 is small then
  output is small.

Mem. : TRY        : d_1
Mem. : LOOKUP     : (input1 is small) = 0.2
Mem. : LOOKUP     : (input2 is small) = 0
Mem. : AND        : 0.2 + 0 -> 0
Mem. : UPDATE     : (output is small) = 0
Mem. : FIRED      : d_1
```

At this point the possibility value for output is small is 0, but since input2 is partially medium, rule d_2 will update this as follows:

```
fuzzy_rule d_2 if input1 is small and input2 is medium then
  output is small.

Mem. : TRY        : d_2
Mem. : LOOKUP     : (input1 is small) = 0.2
Mem. : LOOKUP     : (input2 is medium) = 0.6
Mem. : AND        : 0.2 + 0.6 -> 0.2
Mem. : LOOKUP     : (output is small) = 0
Mem. : CONFIRMS   : 0 + 0.2 -> 0.2
Mem. : UPDATE     : (output is small) = 0.2
```

```
Mem. : FIRED       : d_2
```
Since input1 is partially small and input2 is partially large, the rule d_3 will give the following result:
```
fuzzy_rule d_3 if input1 is small and input2 is large then
  output is small.

Mem. : TRY         : d_3
Mem. : LOOKUP      : (input1 is small) = 0.2
Mem. : LOOKUP      : (input2 is large) = 0.2
Mem. : AND         : 0.2 + 0.2 -> 0.2
Mem. : LOOKUP      : (output is small) = 0.2
Mem. : CONFIRMS    : 0.2 + 0.2 -> 0.2
Mem. : UPDATE      : (output is small) = 0.2
Mem. : FIRED       : d_3
```
The last two rules have both updated the value of output is small to 0.2. This process of updating continues throughout the ruleset until the final update for membership of each of output's three fuzzy sets.

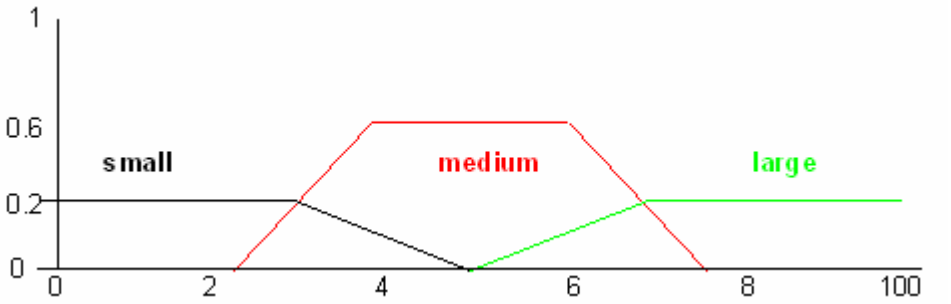
The Defuzzification process

After rule propagation comes deffuzzification to provide a crisp value for Output.
```
Mem. : DE-FUZZIFY : centroid(all_memberships,mirror_rule,shrinking) @
output
Mem. : LOOKUP      : (output is small) = 0.2
Mem. : LOOKUP      : (output is medium) = 0.6
Mem. : LOOKUP      : (output is large) = 0.2
Mem. : DE-FUZZIFI : output = 50
Output = 50
```

Two steps of defuzzification

Defuzzification takes place in two distinct steps. First the membership functions are scaled according to their possibility values, secondly the scaled membership functions are used to obtain the ***centroid*** of the joint fuzzy sets.

Scaling



Since the possibility values for the fuzzy variable output are small=0.2, medium=0.6, large=0.2, the above diagram shows how the membership functions are scaled.

Finding the Centroid

Once scaling has taken place, the centroid can be found by calculating the **balance point** of the conjoined scaled functions. A useful analogy given by Hopgood (Intelligent Systems for Engineers and scientists, p.81, ISBN 0-8493-0456-3, CRC Press) invites us to see the scaled functions as overlapping pieces of stiff cardboard. Imagine them glued together and then find the point on this surface where the card

could be balanced on a pin. The position of this point on the horizontal axis is the resulting crisp defuzzified value for the variable.

Options in defuzzification

The program we are considering uses three fuzzy variables, **input1**, **input2** and **output**. The definition of the **output** variable contains three options for how the variable is defuzzified:

```
defuzzify using all memberships          <--- Option 1
  and mirror rule                        <--- Option 2
  and shrinking .                        <--- Option 3
```

Option 1 can be **all memberships** or **largest membership**
Option 2 can be **mirror rule** or **bounded** or **inverse**
Option 3 can be **shrinking** or **truncation**

Let us look at these options in turn.

All Memberships versus Largest Membership

When the value of a fuzzy variable is defuzzified, a choice exists as to which of its memberships of its component fuzzy sets should be taken into account. Taking into account membership of all fuzzy sets is the normal behaviour, but it is possible to only use the fuzzy set of which it has the largest membership. This results in an output which is much like the step-wise function typical of non-fuzzy production rule inference as in the second of the two diagrams below:

All Memberships + Mirror Rule + Shrinking

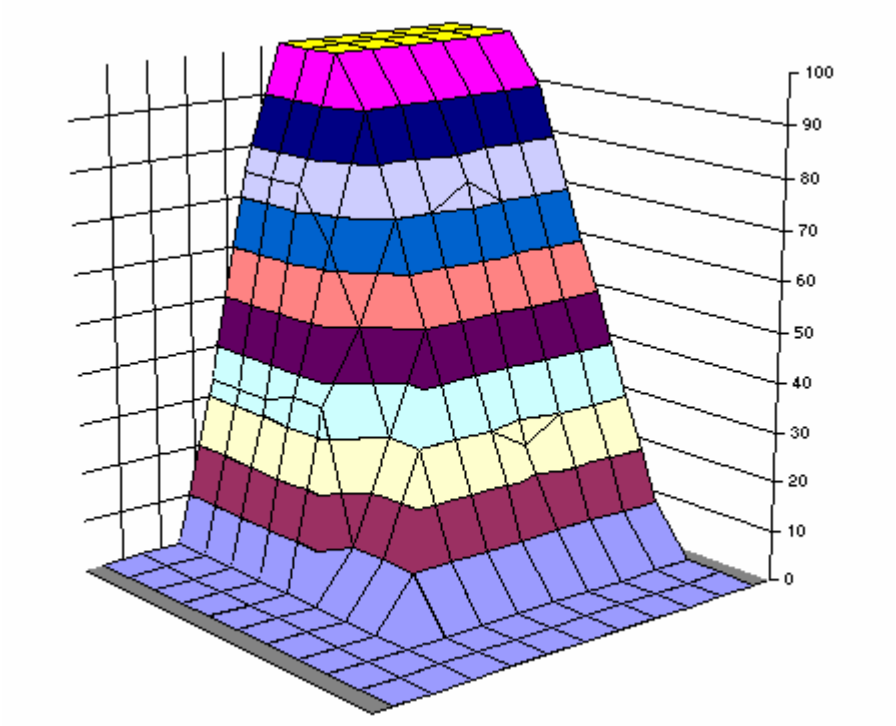


figure 1

Largest Membership + Mirror Rule + Shrinking

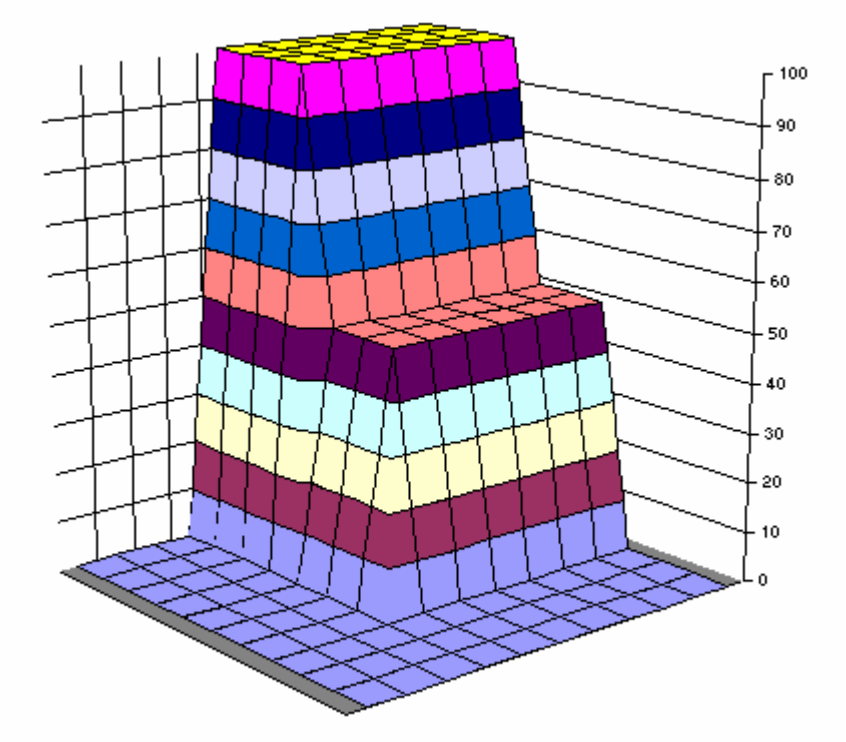


figure 2

Mirror Rule Versus Bounded Versus Inverse

As can be seen in the definition of the fuzzy variable **output**, the two sets at the extremes are **small** and **large** which (before scaling) have the shapes \ and / respectively.

Whenever we have such shapes at the extremes of the fuzzy graph we have a problem finding the balance point since the extreme functions continue indefinitely to the left and to the right respectively.

There are various ways of overcoming this problem. One is by considering these extreme functions to be ***mirrored*** symmetrically at the boundaries of their ranges, another is to simply treat all values beyond the boundaries as being the same value as at the boundaries. The former method is known as the ***mirror rule*** and the latter as the ***bounded range*** method.

Both previous graphs show defuzzified outputs using the mirror rule. Figures 3 and 4 below show what they would look like if we had used the bounded parameter instead. Note that this means that the defuzzified values of the variable can never reach the extremes of their ranges (0 and 100 in our case).

All Memberships + Bounded Range + Shrinking

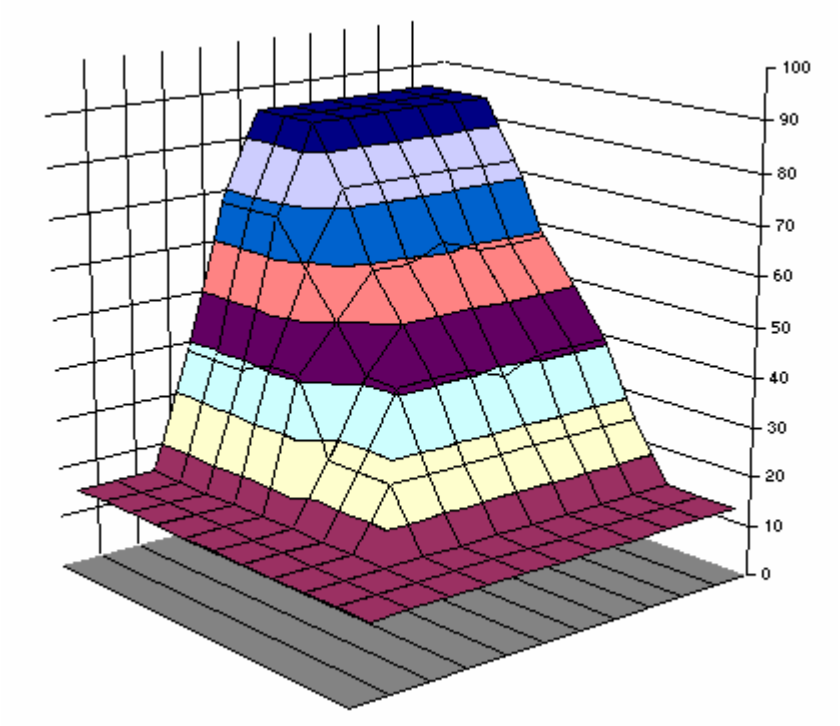


figure 3

Largest Membership + Bounded + Shrinking

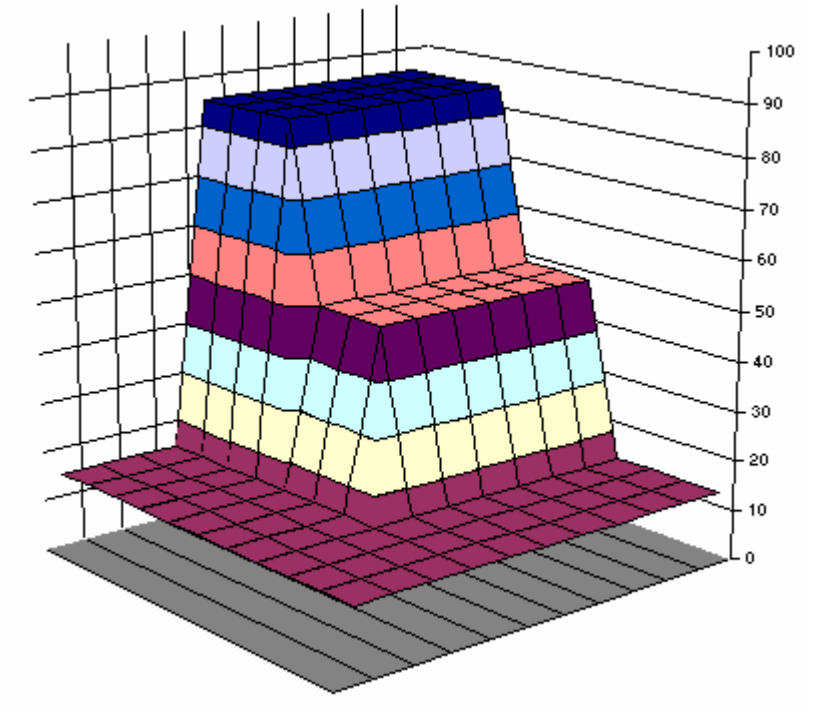


figure 4

The final method for dealing with defuzzification at the extremes is **inverse**. The range of outputs acheived using it is shown in figure 5.

All Memberships + Inverse + Shrinking

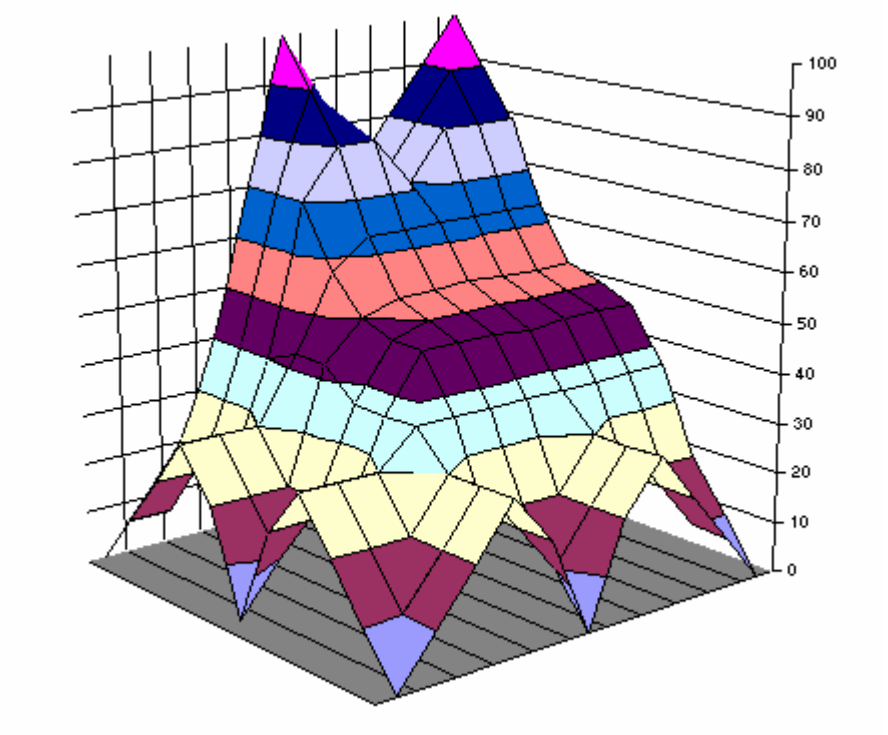


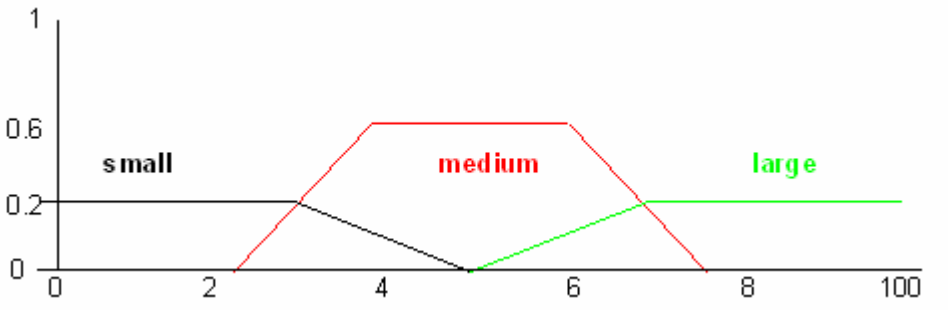
figure 5

This somewhat bizarre shape shows the effect of the following rule: for the extreme sets, look up the value that would fuzzify to the membership and use this value instead of the individual centroids of the sets in finding the overall centroid.

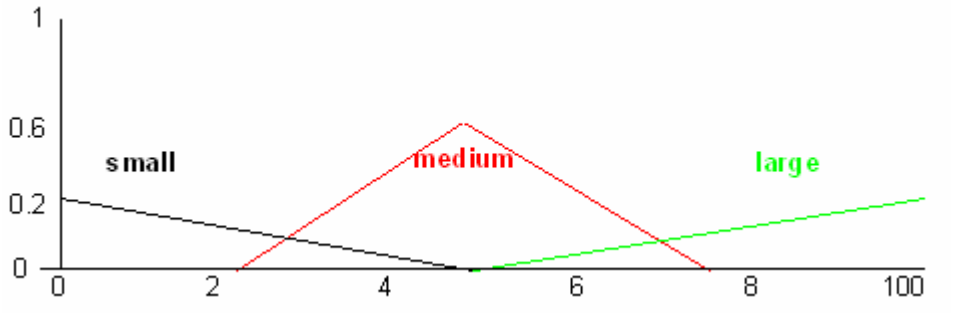
Shrinking Versus Truncation

The final defuzzification option we have is shrinking versus truncation. The distinction here is one of how we scale the membership functions as opposed to how we obtain the centroid. The truncation method of scaling is the first of the two diagrams below, the shrinking method is the second.

Truncation



Shrinking



Graphs of the output derived using the **truncation** parameter are virtually indistinguishable from those we showed above for this example using **shrinking**.

Defuzzification Calculated

For completeness we will now show how the defuzzified value of the output variable is calculated using Larsen's product operation rule. The example uses All Memberships + Mirror Rule + Shrinking as the defuzzification options.

We need to know two things, the final possibility values for output after all fuzzy rules have fired and the centre points of the base of each of the triangular fuzzy sets which make up the fuzzy variable output.

Running the trace with inputs at 40 and 60 we get the following:

```
Mem. : DE-FUZZIFY : centroid(all_memberships,mirror_rule,shrinking) @
output
Mem. : LOOKUP     : (output is small) = 0.2
Mem. : LOOKUP     : (output is medium) = 0.6
Mem. : LOOKUP     : (output is large) = 0.2
Mem. : DE-FUZZIFI : output = 50
C = 50
```

Thus the possibility values are 0.2, 0.6, 0.2. Let us call these P1, P2, P3.

The three sets have the midpoints along their bases at 0, 50 100 (after the mirror rule is applied). Call these C1, C2, C3.

The calculation is:

$$\frac{\sum(P_i * C_i)}{\sum P_i}$$

$$= \frac{(0.2 * 0) + (0.6 * 50) + (0.2 * 100)}{(0.2 + 0.6 + 0.2)}$$

$$= \frac{0 + 30 + 20}{1} = 50$$

Note that we are able to use the midpoints of the triangle's bases as Ci here because the triangles are all equilateral. Had they not been then we would have to use their respective **areas** instead.

Index