Title: Fuzzy Service

Date: June 2018 Author: Clive Spenser, LPA

This document explains the basic of Fuzzy Logic as found in the FLINT toolkit using a classical problem, which combines 2 inputs and produces a single output.

Much of the underlying algorithms implemented in FLINT are drawn from
Intelligent Systems for Engineers and Scientists by Adrian Hopgood

https://www.crcpress.com/Intelligent-Systems-for-Engineers-and-Scientists-Third-Edition/Hopgood/p/book/9781439821206

Book Summary

The third edition of this bestseller examines the principles of artificial intelligence and their application to engineering and science, as well as techniques for developing intelligent systems to solve practical problems. Covering the full spectrum of intelligent systems techniques, it incorporates knowledge-based systems, computational intelligence, and their hybrids.

Using clear and concise language, **Intelligent Systems for Engineers and Scientists, Third Edition** features updates and improvements throughout all chapters. It includes expanded and separated chapters on genetic algorithms and single-candidate optimization techniques, while the chapter on neural networks now covers spiking networks and a range of recurrent networks. The book also provides extended coverage of fuzzy logic, including type-2 and fuzzy control systems. Example programs using rules and uncertainty are presented in an industry-standard format, so that you can run them yourself.

Check out the significantly expanded set of free resources that support the book at:

http://www.adrianhopgood.com/aitoolkit/

Introduction

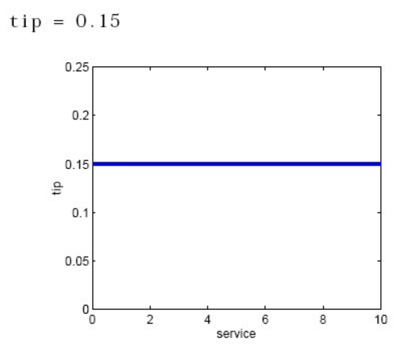
Let’s consider the tipping problem: what is the "right" amount to tip your waitperson?

The Basic Tipping Problem

Given a number between 0 and 10 that represents the quality of service at a restaurant (where 10 is excellent), what should the tip be?
An average tip for a meal is 15%, though the actual amount may vary depending on the quality of the service provided.

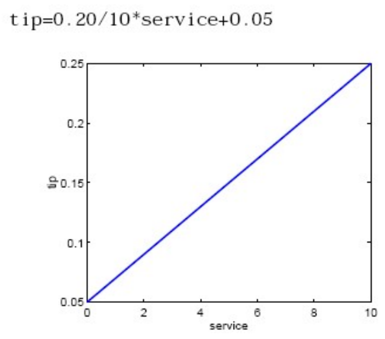
The Non-Fuzzy Approach

A flat tip of say 15% would be a simple starting point.



But this doesn't reward good service, so we need to add a new term to the equation. Since service is rated on a scale of zero to ten, then we could have the tip go from 5% if the service is bad to 25% if the service is excellent. Now our relation looks like this:

```
tip% = 5 + 2 * service
```



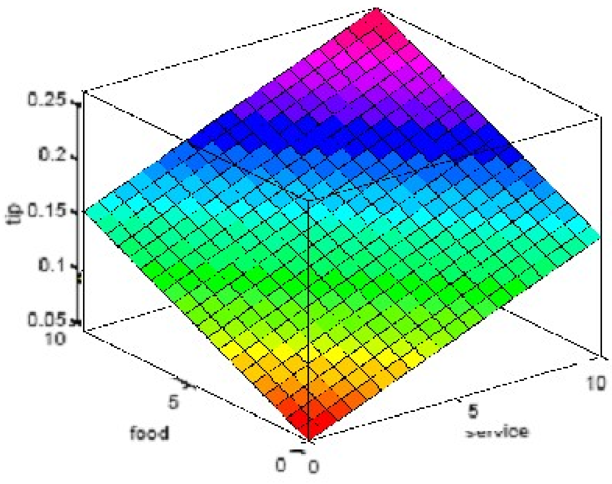
So, when service is 0, the tip is 5% and when service is 10, the tip is 25%.

So far so good. The formula does what we want it to do, and it's pretty simple. What about if we want the tip to reflect the quality of the food as well as the service.

The Extended Tipping Problem

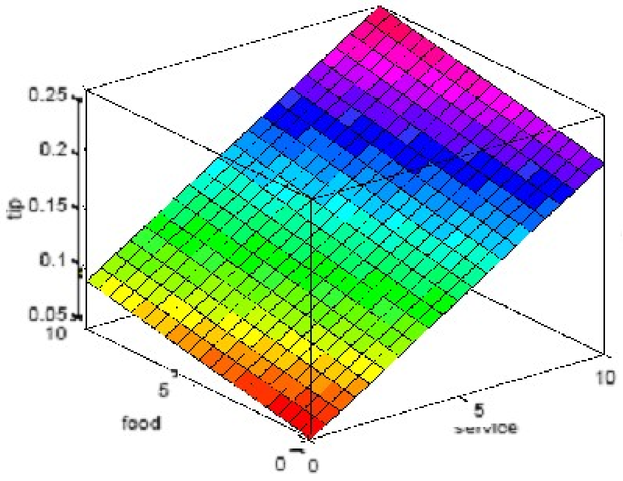
Given numbers between 0 and 10 (where 10 is excellent) that represent the quality of the service and the quality of the food, respectively, at a restaurant, what should the tip be? How will our formula be affected now that we've added another variable? Here's one attempt:

```
tip% = 5 + service + food
```



So when service is 0 and food is 0, the tip is 5% and when both are 10, the tip is 25%; and if either one is zero and the other is 10, then the service is 15%. Well, OK, but what if we want service to be more important than the food quality. Let's say that we want the service to account for 80% of the overall tipping "grade" and the food to make up the other 20% (i.e. a ratio of 4:1). This gives:

```
tip% = 5 + (4*service+food)*0.4
```



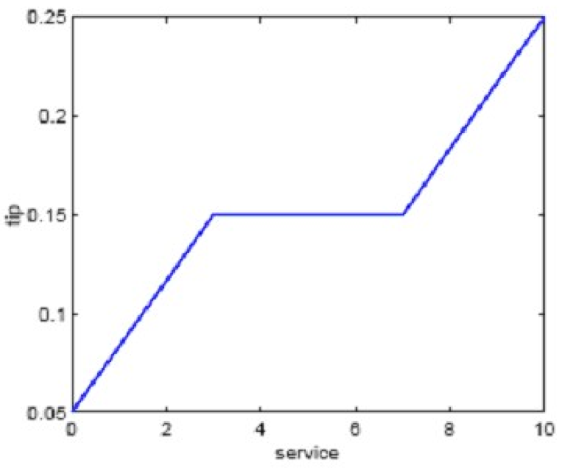
So, when service is 0 and food is 0, the tip is 5% and when both are 10, the tip is 25%; and if food is zero and the service is 10, then the service is 21%, and when the food is 10 and the service is zero, then the tip is 9%.

The response is still a bit too uniformly linear. Suppose we want more of a flat response in the middle, i.e., to give a 15% tip in general, and to depart from this only if the service is exceptionally good or bad.

This, in turn, means that those nice linear mappings no longer apply. We can still salvage things by using a piecewise linear construction. Let's return to the one-dimensional problem of just considering the service. We can use a simple conditional statement like:

```
if  service<3, tip% = 5 + (10/3)*service;
      else if  service<  7, tip% = 15;
      else if  service<=10, tip% = 15 + (10/3)*(service-7);
```

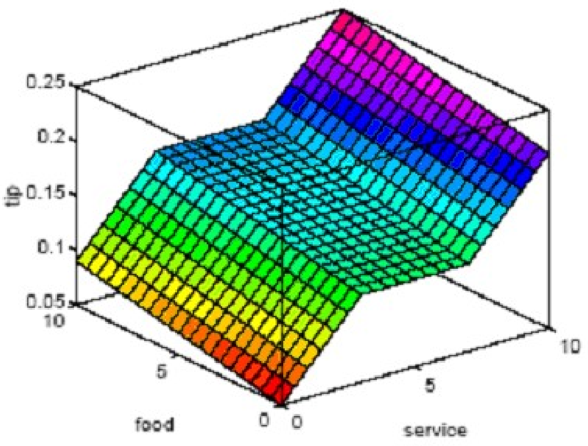
This gives something like the following graph.



So, when service is 0, the tip is 5% and when service is 10, the tip is 25%, but stays at 15% for any value of service between 3 and 7 inclusive.

Now, if we extend this to two dimensions, we can get something like:

```
if service<3,
       tip% = 5+(10/3)*(4*service*0.2) + food*0.4;
 else if service<7,
       tip% = 5+(10/3)*(4*3*0.2) + food*0.4;
 else if service<=10,
       tip% = 5+(10/3)*(4*(service-4)*0.2) + food*0.4
```



So, when service is 0 and food is 0, the tip is 5% and when both are 10, the tip is 25%; and if food is zero and the service is 10, then the service is 21%, and when the food is 10 and the service is zero, then the tip is 9%.

The plot looks good, but the function is surprisingly complicated considering its humble start. How did we end up here? It was a little tricky to code this correctly, and it's definitely not easy to modify in the future. It works, but it's not easy to troubleshoot. It has hard-coded numbers going through the whole thing. It's even less apparent how the algorithm works to someone who didn't witness the original design process.

The Fuzzy Approach

It would be nice if we could just capture the essentials of this problem, leaving aside all the factors that could be arbitrary. If we make a list of what really matters in this problem, we might end up with this:

1. if service is poor then tip is cheap

2. if service is good then tip is average

3. if service is excellent then tip is generous

The order in which the rules are presented here is arbitrary. It doesn't matter which rules come first. If we wanted to include the food's effect on the tip, we might add the following two rules:

4. if food is rancid then tip is cheap

5. if food is delicious then tip is generous

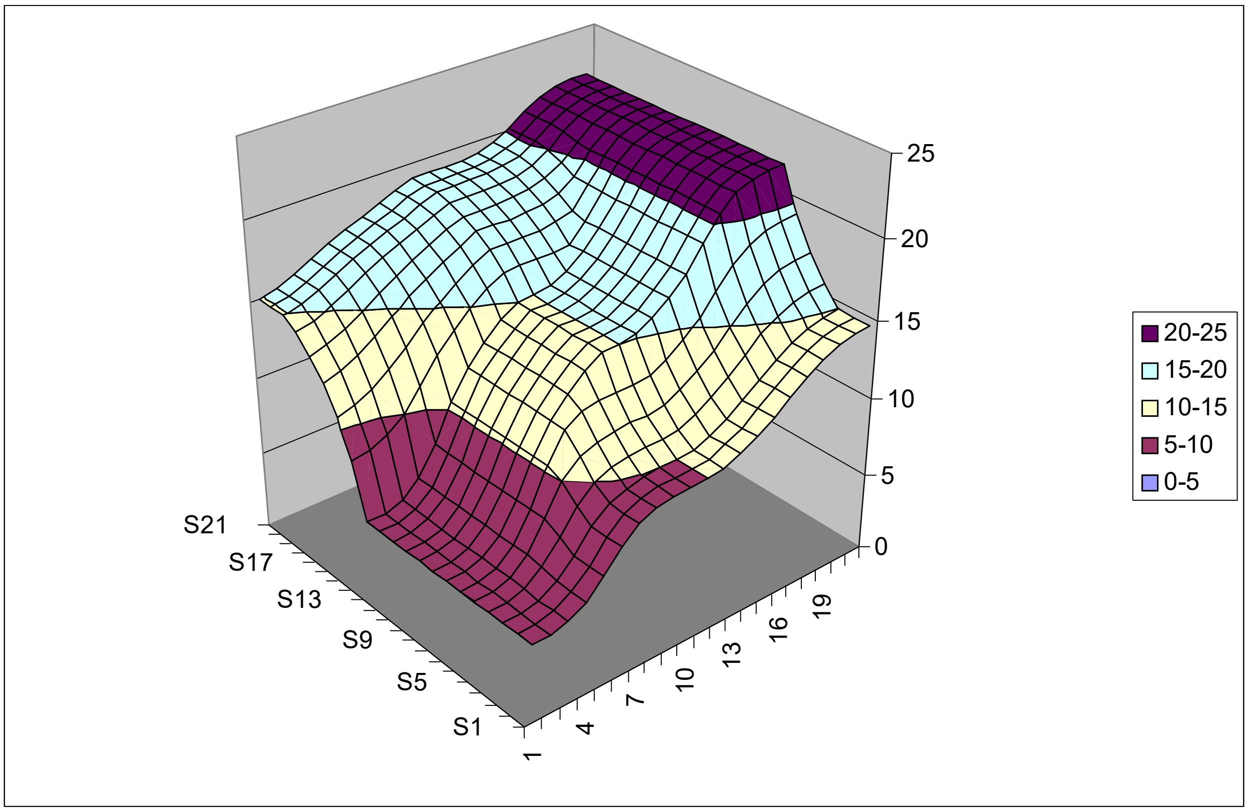
In fact, we can combine the two different lists of rules into one list of three rules like so:

1. if service is poor or the food is rancid then tip is cheap

2. if service is good then tip is average

3. if service is excellent or food is delicious then tip is generous

These three rules are the core of our solution. And coincidentally, we've just defined the rules for a fuzzy logic system. Now if we give mathematical meaning to the linguistic variables (what is an "average" tip, for example?) we would have a complete fuzzy inference system. And we've shown that by using linguistic qualifiers, we can describe our rules in an intuitive and flexible manner.



Above is the picture associated with the fuzzy system that solves this problem. The picture above was generated by the three rules above. The mechanics of how fuzzy inference works will be explained in the next section.

Some Observations

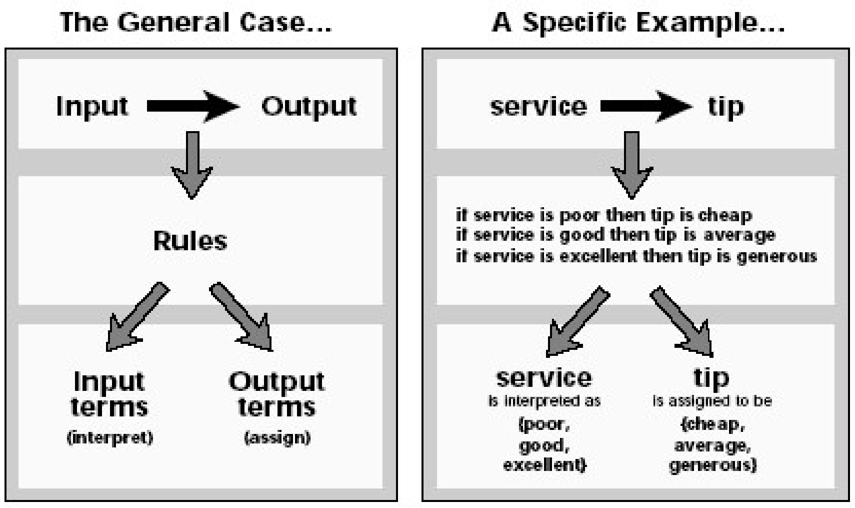
Some observations about the example so far. We found a piecewise linear relation that solved the problem. It worked, but it was something of a nuisance to derive, and once we wrote it down as code it wasn't very easy to interpret. On the other hand, the fuzzy system is based on some "common sense" statements. Also, we were able to add two more rules to the bottom of the list that massaged the shape of the overall output without needing to hack into what had already been done. In other words, the subsequent modification was pretty easy.

Moreover, by using fuzzy logic rules, the maintenance of the algorithm decouples along fairly clean lines. My notion of an average tip might change from day to day, city to city, country to country. But the underlying logic is the same: if the service is good, the tip should be average. I don't have to change that part, no matter where in the world I travel. I can recalibrate the method quickly by simply shifting the fuzzy set that defines average without rewriting my rule.

You can do this sort of thing with lists of piecewise linear functions (in a non-fuzzy approach), but the medium is working against you. You're more likely to get tangled up in wires than you are to recalibrate the problem quickly. You can also buttress the piecewise linear solution by including many helpful comments. However, even if we lightly pass over the fact that the vast majority of code is woefully uncommented, it's still true that as code gets revised and updated, the comments can quickly slip into uselessness, or worse, they can actually provide misinformation.

The Big Picture

We'll start with a little motivation for where we are headed in this chapter. The point of fuzzy logic is to map an input space to an output space, and the primary mechanism for doing this is a list of if-then statements called rules. All rules are evaluated in parallel, and the order of the rules is unimportant. The rules themselves are useful because they refer to variables and the adjectives that describe those variables. Before we can build a system that interprets rules, we have to define all the terms we plan on using and the adjectives that describe them. If we want to talk about how hot the water is, we need to define the range that the water's temperature can be expected to vary over as well as what we mean by the word *hot*. These are all things we'll be discussing in the next several sections of the manual. The diagram below is something like a roadmap for the fuzzy inference process. It shows the general description of a fuzzy system on the left and a specific fuzzy system (the tipping example from the Introduction) on the right. The whole idea behind fuzzy inference is to interpret the values in the input vector and, based on some set of rules, assign values to the output vector. And that's really all there is to it.



If-Then Rules

Fuzzy sets and fuzzy operators are the subjects and verbs of fuzzy logic. But in order to say anything useful we need to make complete sentences. Conditional statements, if-then rules, are the things that make fuzzy logic useful.

A single fuzzy if-then rule assumes the form:

if x is A then y is B

where A and B are linguistic values defined by fuzzy sets on the ranges X and Y, respectively.

The if-part of the rule "x is A" is called the *antecedent* or premise, while the then-part of the rule "y is B" is called the *consequent* or conclusion. An example might be:

if service is good then tip is average

Note that *good* is represented as a sequence of numbers between 0 and 1 (membership function), so the antecedent is an interpretation that returns a single number between 0 and 1. *Average*, the consequent, is represented as a fuzzy set, and so is an assignment that assigns the entire fuzzy set B to the output variable *y*. The word "is" gets used in 2 different ways depending on whether it appears in the antecedent or the consequent.

In general, the input to an if-then rule is the current value for the input variable (*service*) and the output is an entire fuzzy set (*average*). This will later get defuzzified back to a crisp value.

Interpreting an if-then rule involves distinct parts: first evaluating the antecedent (which involves fuzzifying the input and applying any necessary *fuzzy operators*) and second applying that result to the consequent (known as *implication*). In the case of two-valued or binary logic, if-then rules don't present much difficulty. If the premise is true, then the conclusion is true. But if we relax the restrictions of two-valued logic and let the antecedent be a fuzzy statement, how does this reflect on the conclusion? The answer is a simple one: if the antecedent is true to some degree of membership, then the consequent is also true to that same degree. In other words

in binary logic: p -> q *(p and q are either true or false)*
in fuzzy logic: 0.5 p -> 0.5 q (*partial antecedents imply partial implication*)

The antecedent of a rule can have multiple parts:

if sky is gray and wind is strong and barometer is falling, then ...

in which case all parts of the antecedent are calculated simultaneously and resolved to a single number using the logical operators.

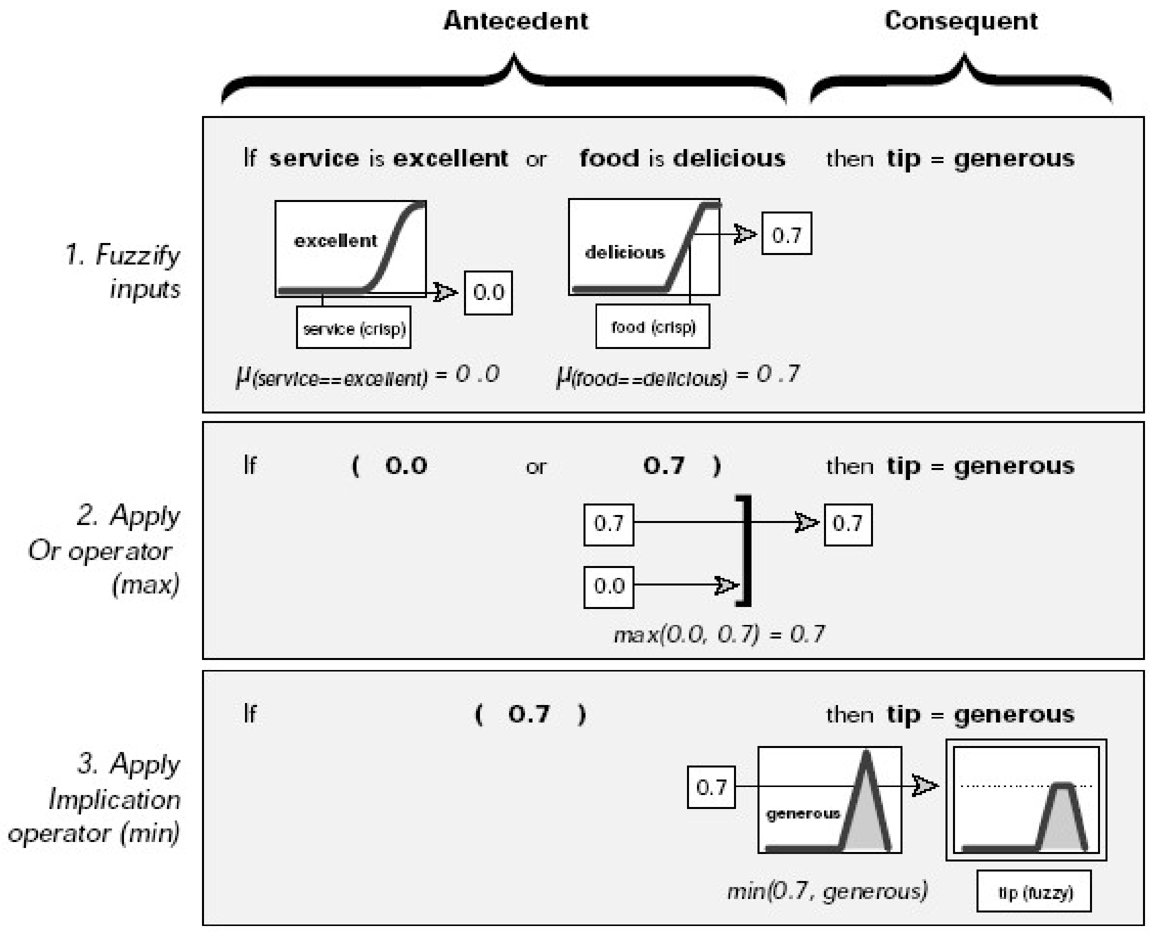
In Boolean logic A AND B, A OR B, NOT A can be defined using simple truth tables.

In Fuzzy Logic, as counterparts to A AND B, A OR B, NOT A, we use functions such as min(A,B), max(A,B), 1-A . The consequent of a rule can also have multiple parts (i.e. multiple outputs):

if temperature is cold then hot water valve is open and cold water valve is shut

in which case all consequents are affected equally by the result of the antecedent.

How is the consequent affected by the antecedent? The consequent specifies a fuzzy set be assigned to the output. The implication function then modifies that fuzzy set to the degree specified by the antecedent. The most common ways to modify the output fuzzy set are truncation using the min function (where the fuzzy set is "chopped off" as shown below) or scaling using the prod function (where the output fuzzy set is "squashed"). Both are supported by Flint, but we will mainly use truncation.


Antecedent
Consequent
1. Fuzzify inputs
If service is excellent or food is delicious then tip = generous
excellent
delicious
0.0
0.7
service (crisp)
food (crisp)
μ(service==excellent) = 0 .0
μ(food==delicious) = 0 .7
2. Apply Or operator (max)
If (0.0 or 0.7) then tip = generous
0.7
0.0
0.7
max(0.0, 0.7) = 0.7
3. Apply Implication operator (min)
If (0.7) then tip = generous
0.7
generous
min(0.7, generous)
tip (fuzzy)

Summary of If-Then Rules

Interpreting if-then rules is a three part process.

1 Fuzzify inputs

Resolve all fuzzy statements in the antecedent to a degree of membership between 0 and 1. If there is only one part to the antecedent, this is the degree of support for the rule.

2 Apply fuzzy operator

If there are multiple parts to the antecedent, apply fuzzy logic operators and resolve the antecedent to a single number between 0 and 1. This is the degree of support for the rule.

3 Apply implication method

Use the degree of support for the entire rule to shape the output fuzzy set. The consequent of a fuzzy rule assigns an entire fuzzy set to the output. If the antecedent is only partially true, then the output fuzzy set is trimmed according to the implication method.

In general, one rule by itself doesn't do much good. What's needed are two or more rules that can play off one another. The output of each rule is a fuzzy set, but in general we want the output for an entire collection of rules to be a single number. How are all these fuzzy sets distilled into a single crisp result for the output variable? First the output fuzzy sets for each rule are aggregated into a single output fuzzy set. Then the resulting set is defuzzified, or resolved to a single number. The next section shows how the whole process works from beginning to end.

Fuzzy Inference Systems

Fuzzy inference is the actual process of mapping from a given input to an output using fuzzy logic. The process involves all the pieces that we have discussed in the previous sections: membership functions, fuzzy logic operators, and if-then rules.

Fuzzy inference systems have been successfully applied in fields such as automatic control, data classification, decision analysis, expert systems, and computer vision. Because of its multi-disciplinary nature, the fuzzy inference system is known by a number of names, such as fuzzy-rule-based system, fuzzy expert system, fuzzy model, fuzzy associative memory, fuzzy logic controller, and simply (and ambiguously) fuzzy system.

We will be using FLINT to implement the various fuzzy logic examples which are described herein, and use the tracing mechanism that FLINT provides to give insight into the internal workings.

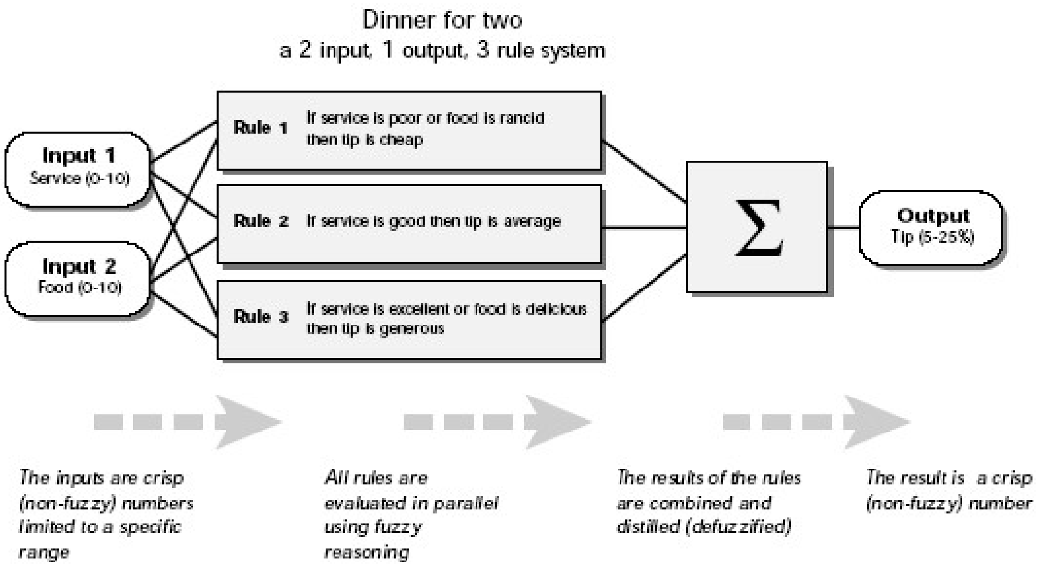
FLINT is implemented in LPA Prolog and can be used using a Prolog rules syntax, and embedded within a Prolog program.

FLINT is also integrated in to Flex, LPA's expert system toolkit, which means that you can write your fuzzy rules in using the KSL that Flex provides/ In this way, you can incorporate fuzzy reasoning within other Flex programs, be they forward-chaining or backward chaining.

You can also use FLINT in VisiRule, as VisiRule supports user defined programs in the form of code boxes and statement boxes. This means you can have a visual outline for your fuzzy program.

Dinner for Two, Reprise

In this section, we'll see how everything fits together using the same two-input one-output three-rule tipping problem that we saw in the introduction. Only this time we won't skip over any details. The basic structure of this example is shown in the diagram below.


Dinner for two
a 2 input, 1 output, 3 rule system
Input 1
Service (0-10)
Input 2
Food (0-10)
Rule 1
If service is poor or food is rancid then tip is cheap
Rule 2
If service is good then tip is average
Rule 3
If service is excellent or food is delicious then tip is generous
Σ
Output
Tip (5-25%)
The inputs are crisp (non-fuzzy) numbers limited to a specific range
All rules are evaluated in parallel using fuzzy reasoning
The results of the rules are combined and distilled (defuzzified)
The result is a crisp (non-fuzzy) number

Information flows from left to right, from two inputs to a single output. The parallel nature of the rules is one of the more important aspects of fuzzy logic systems. Instead of sharp switching between modes based on breakpoints, we will glide smoothly from regions where the system's behavior is dominated now by this rule, now by that one.

In FLINT, there are various parts of the fuzzy inference process: fuzzification of the input variables, application of the fuzzy operator (AND or OR) in the antecedent, implication from the antecedent to the consequent, aggregation or propagation of the consequents across the rules, and defuzzification. These sometimes cryptic and unusual names have very specific meaning and we'll try to define them clearly as we visit them.

In Flex KSL syntax, the fuzzy service program will look something like:

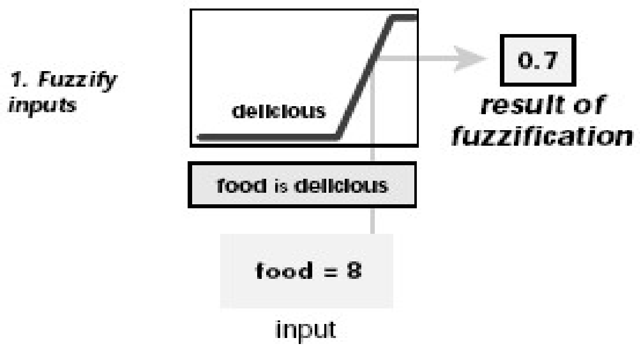
```
% Fuzzy Rules
fuzzy_rule r1f
  if   the service is poor
  or   the food is rancid
  then the tip is cheap .

fuzzy_rule r2f
  if   the service is good
  then the tip is average .

fuzzy_rule r3f
  if   the service is excellent
  or   the food is delicious
  then the tip is generous .

% Fuzzy memberships
fuzzy_variable food ;
   ranges from 0 to  10 ;
   fuzzy_set rancid    is \ shaped and linear at 1, 3 ;
   fuzzy_set delicious is / shaped and linear at    6, 8.857 .

fuzzy_variable service ;
   ranges from 0 to  10 ;
   fuzzy_set poor is \ shaped and curved 1.25 at 0, 4.5 ;
   fuzzy_set good is /\ shaped and curved 3 at 0, 5, 10 ;
   fuzzy_set excellent is / shaped and curved 2 at 5, 10  .

fuzzy_variable tip ;
   ranges from 0 to  30 ;
   fuzzy_set cheap    is /\ shaped and linear at  0,  5, 10 ;
   fuzzy_set average  is /\ shaped and linear at 10, 15, 20 ;
   fuzzy_set generous is /\ shaped and linear at 15, 22, 29 ;
   defuzzify using
      all memberships
       and bounded range
       and shrinking .
```

In FLINT, you can edit the fuzzy variables using the Fuzzy Editor.

Step 1. Fuzzify Inputs

The first step is to take the inputs and determine the degree to which they belong to each of the appropriate fuzzy sets via membership functions. The input to a fuzzification process is always a crisp numerical value limited to the universe of discourse of the input variable (in this case the interval between 0 and 10) and the output is a fuzzy degree of membership (always the interval between 0 and 1). So fuzzification really doesn't amount to anything more than table lookup or function evaluation.

The example we're using in this section is built on three rules, and each of the rules depends on resolving the inputs into a number of different fuzzy linguistic sets: service is poor, service is good, food is rancid, food is delicious and so on. Before the rules can be evaluated, the inputs must be fuzzified against these linguistic sets. For example, to what extent is the food really delicious? The figure below shows how well the food at our hypothetical restaurant (rated on a scale of 0 to 10) fits the linguistic variable "delicious". In this case, we rated the food as an 8, which, given our graphical definition of delicious, corresponds to ? = 0.7.



(The compliment to the chef would be "your food is delicious to the degree 0.7.") In this manner, each input is fuzzified over all the membership functions required by the rules.

Note that in general a crisp value will map on to more than one fuzzy qualifier, as a fuzzy variable often has multiple overlapping fuzzy qualifiers. This is like saying, the service was ‘a bit’ excellent, ‘very’ good and ‘not at all’ poor.

We can use the trace mechanism in FLINT to see these calculations:

```
Mem.  : FUZZIFY    : service = 3
Mem.  : UPDATE     : (service is poor) = 0.301200667869948
Mem.  : UPDATE     : (service is good) = 0.744
Mem.  : UPDATE     : (service is excellent) = 0
Mem.  : FUZZIFY    : food = 8
Mem.  : UPDATE     : (food is rancid) = 0
Mem.  : UPDATE     : (food is delicious) = 0.700035001750088
```

Step 2. Apply Fuzzy Operator

Once the inputs have been fuzzified, we know the degree to which each part of the antecedent has been satisfied for each rule. If the antecedent of a given rule has more than one part, the fuzzy operator is applied to obtain one number that represents the result of the antecedent for that rule. This number will then be applied to the output function. The input to the fuzzy operator is two or more membership values from fuzzified input variables. The output is a single value which reflects the degree of strength that that fired rule has generated.

There are three built-in AND methods supported: minimum, product and truncate.
There are three built-in OR methods supported: maximum, strengthen and addition.

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

Table 10 - Methods for calculating membership values

```
Mem.  : TRY       : r1f
Mem.  : LOOKUP    : (service is poor) = 0.301200667869948
Mem.  : LOOKUP    : (food is rancid) = 0
Mem.  : OR        : 0.301200667869948 + 0 -> 0.301200667869948
Mem.  : UPDATE    : (tip is cheap) = 0.301200667869948
Mem.  : FIRED     : r1f

Mem.  : TRY       : r2f
Mem.  : LOOKUP    : (service is good) = 0.744
Mem.  : UPDATE    : (tip is average) = 0.744
Mem.  : FIRED     : r2f

Mem.  : TRY       : r3f
Mem.  : LOOKUP    : (service is excellent) = 0
Mem.  : LOOKUP    : (food is delicious) = 0.700035001750088
Mem.  : OR        : 0 + 0.700035001750088 -> 0.700035001750088
Mem.  : UPDATE    : (tip is generous) = 0.700035001750088
Mem.  : FIRED     : r3f
```

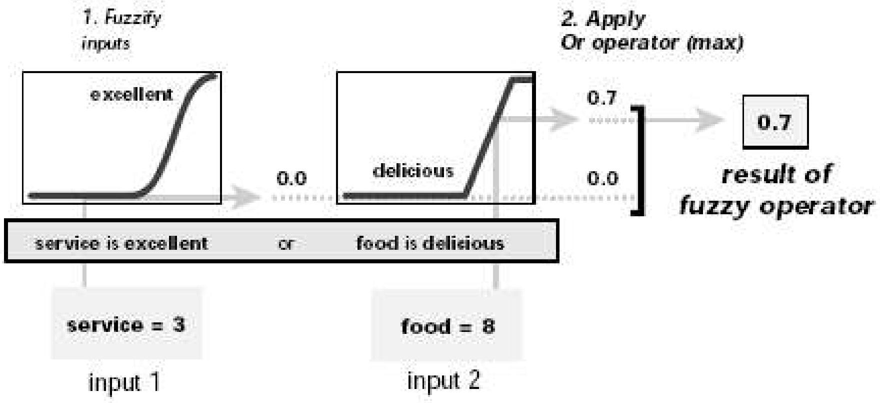
Example usage: Maximum being used to implement OR

Note, these traces are from real Flint sessions. The + symbol on lines labeled OR is used to reflect the OR operation in general and is always used irregardless of the method being applied.

Any number of methods can fill in for the AND or the OR operation, you can create your own methods by writing any function and setting that to be your method of choice.

```
logical_operator( (dor),  binary, 1100 ).
logical_operator_definition( (dor),  fuzzy, (X,Y), min( X+Y, 1 )   ).
```

Below is an example of the OR operator max at work. We're evaluating the antecedent of the rule 3 for the tipping calculation. The two different pieces of the antecedent (service is excellent and food is delicious) yielded the fuzzy membership values 0.0 and 0.7 respectively. The fuzzy OR operator simply selects the maximum of the two values.



Step 3. Fuzzy Propagation

Once we have established truth levels for all the antecedents of all the rules, then we can fire them. This will update the output variables. Where the same fuzzy set of the output variable is affected, we have the same built-in methods available as the OR operator, namely:

`maximum`, `addition` and `strengthen`.

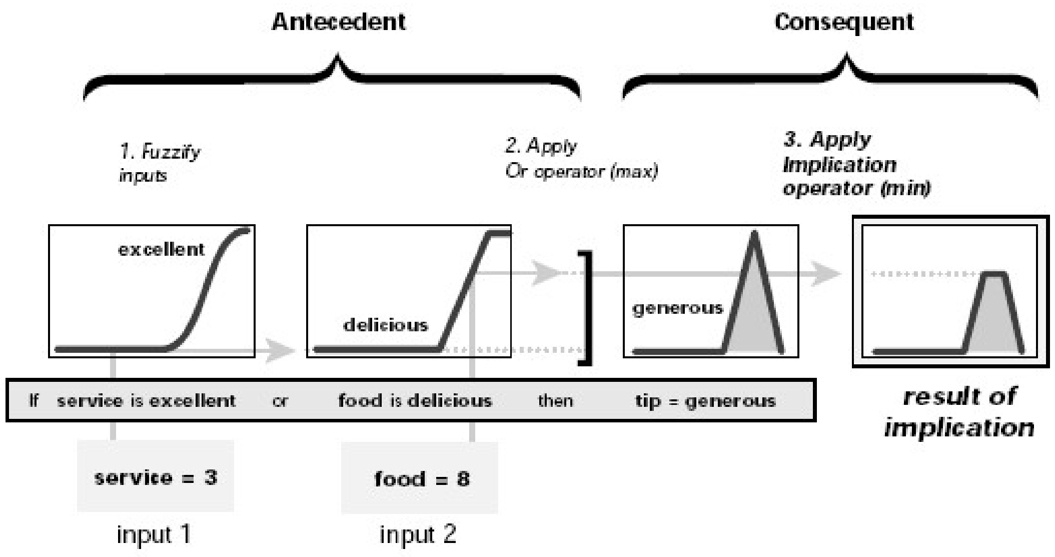
These are defined as `max(A,B), min(A+B,1)` and `(A+B/1-A)` respectively.

Notice that as long as these are commutative, then the order of the rules is unimportant.

The propagation method is defined as the shaping of the consequent (a fuzzy set) based on the antecedent (a single number). The input is a single number given by the antecedent, and the output is a fuzzy set. Implication occurs for each rule.

Three built-in methods are supported, and they are the same functions that are used by the AND method:

- minimum: truncates output set
- product: scales output set
- truncate.


Antecedent
Consequent
1. Fuzzify inputs
2. Apply Or operator (max)
3. Apply Implication operator (min)
excellent
delicious
generous
If service is excellent or food is delicious then tip = generous
result of implication
service = 3
food = 8
input 1
input 2

```
Mem.   : TRY         : t,2
Mem.   : LOOKUP      : (temperature is cold) = 0.8
Mem.   : LOOKUP      : (pressure is ok) = 0.3
Mem.   : AND         : 0.8 + 0.3 -> 0.3
Mem.   : LOOKUP      : (throttle is positive) = 0
Mem.   : CONFIRMS    : 0 + 0.3 -> 0.3
Mem.   : UPDATE      : (throttle is positive) = 0.3
Mem.  : FIRED      : t,2
Mem.   : TRY          : t,6
Mem.   : LOOKUP       : (temperature is normal) = 0.2
Mem.   : LOOKUP       : (pressure is high) = 0.7
Mem.   : AND          : 0.2 + 0.7 -> 0.14
Mem.   : LOOKUP       : (throttle is negative) = 0.56
Mem.   : CONFIRMS     : 0.56 + 0.14 -> 0.7
Mem.   : UPDATE       : (throttle is negative) = 0.7
Mem.   : FIRED        : t,6
```

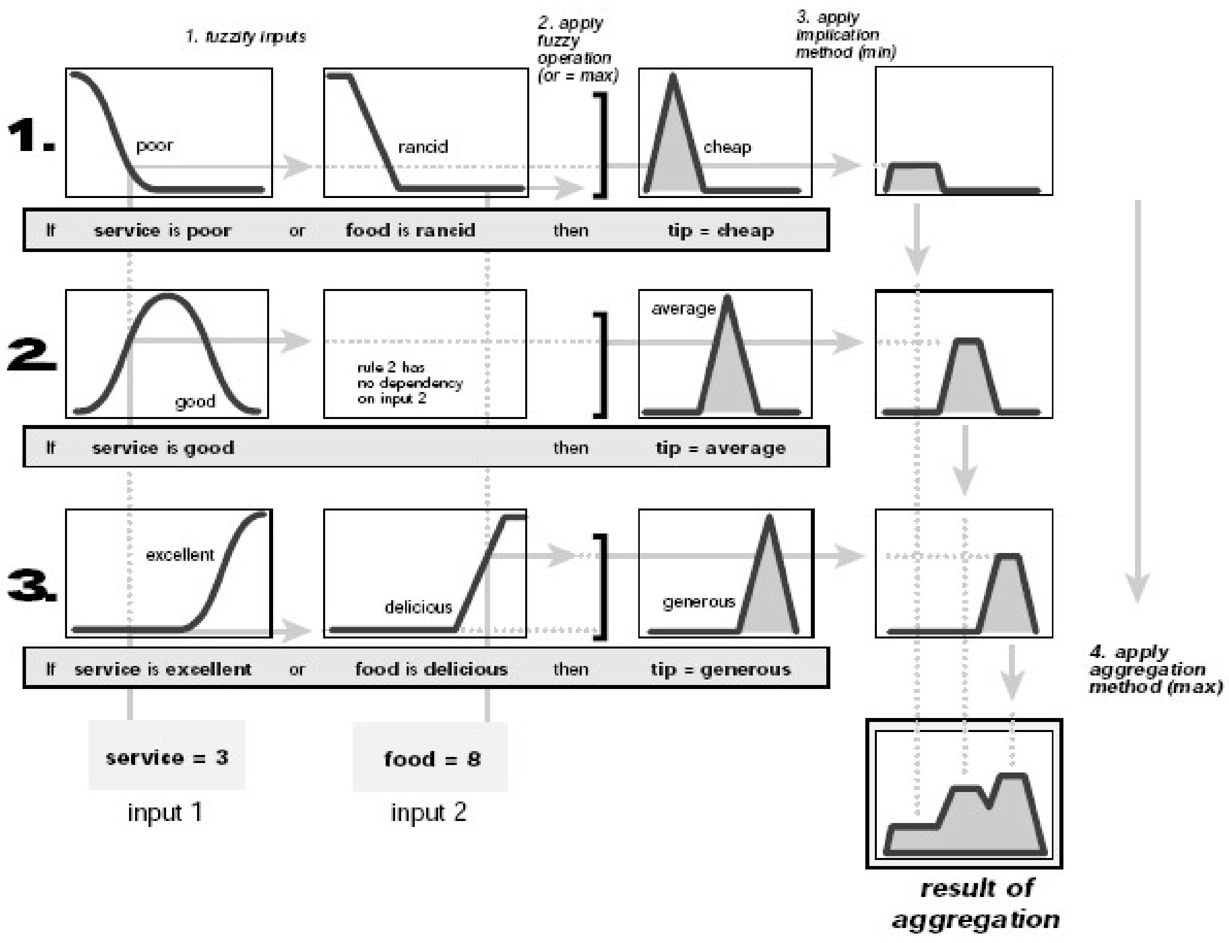
Propagation using `fuzzy_propagate(product,addition,complement,[t])`

Note, these traces are from real Flint sessions. The + symbol on lines labeled AND is used to reflect the AND operation in general and is always used irregardless of the method being applied. Similarly, the + symbol on lines labeled CONFIRMS is used to reflect the propagation process in general and is always used irregardless of the method being applied.

Aggregation is when we unify the outputs of each rule by joining the parallel threads. It's just a matter of taking all the fuzzy sets that represent the output of each rule and combining them into a single fuzzy set in preparation for the final step, defuzzification.

Aggregation only occurs once for each output variable. The input of the aggregation process is the list of truncated output functions returned by the implication process for each rule. The output of the aggregation process is one fuzzy set for each output variable.

In the diagram below, all three rules have been placed together to show how the output of each rule is combined, or aggregated, into a single fuzzy set for the overall output.


1. fuzzify inputs
2. apply fuzzy operation (or = max)
3. apply implication method (min)
1.
poor
rancid
cheap
If service is poor or food is rancid then tip = cheap
2.
good
rule 2 has no dependency on input 2
average
If service is good then tip = average
3.
excellent
delicious
generous
If service is excellent or food is delicious then tip = generous
4. apply aggregation method (max)
service = 3
input 1
food = 8
input 2
result of aggregation

Step 4. Defuzzify

The input for the defuzzification process is a fuzzy set (the aggregate output fuzzy set) and the output is a single brittle value. As much as fuzziness helps the rule evaluation during the intermediate steps, the final output for each variable is generally a single crisp number. So, given a fuzzy set that encompasses a range of output values, we need to return one number, thereby moving from a fuzzy set to a crisp output.

Perhaps the most popular defuzzification method is the centroid calculation, which returns the center of gravity under the curve.

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.

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, 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

There are three options for how the definition of the **output** 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`**

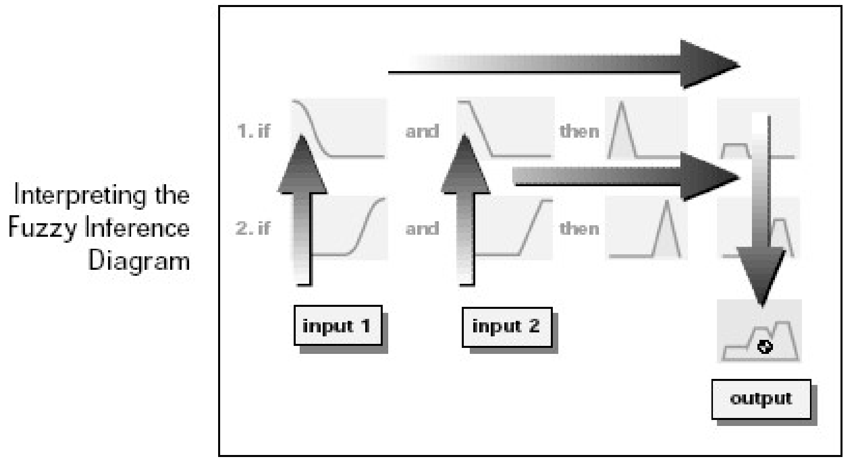
Option 2 can be **`mirror rule`** or **`bounded range`** or **`inverse`**

Option 3 can be **`shrinking`** or **`truncation`** (Shrinking uses Larsen's product)

```
Mem.  : DE-FUZZIFY : centroid(all_memberships,bounded_range,shrinking)
@ tip
Mem.  : LOOKUP     : (tip is cheap) = 0.301200667869948
Mem.  : LOOKUP     : (tip is average) = 0.744
Mem.  : LOOKUP     : (tip is generous) = 0.700035001750088
Mem.  : DE-FUZZIFI : tip = 16.9001787260349
Suggested Tip is: 16.90%
```

The Fuzzy Inference Diagram – part 1

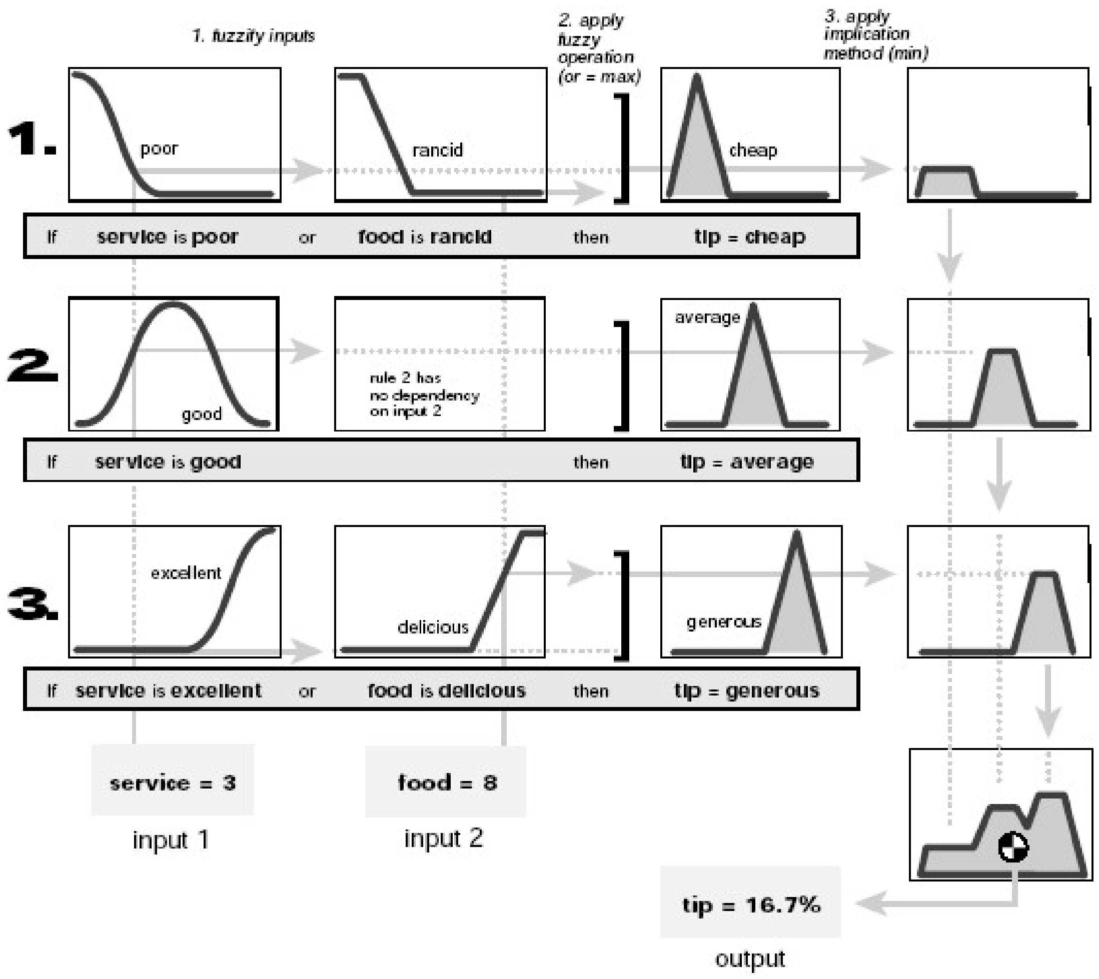
The fuzzy inference diagram is the composite of all the smaller diagrams we've been looking at so far in this section. It simultaneously displays all parts of the fuzzy inference process we've examined. Information flows through the fuzzy inference diagram as below.



Notice how the flow proceeds up from the inputs in the lower left, then across each row, or rule, and then down the rule outputs to finish in the lower right. This is a very compact way of showing everything at once, from linguistic variable fuzzification all the way through defuzzification of the aggregate output.

The Fuzzy Inference Diagram – part 2

Shown below is the real full-size fuzzy inference diagram. There's a lot to see in a fuzzy inference diagram, but once you become accustomed to it, you can learn a lot about a system very quickly. For instance, from this diagram with these particular inputs, we can easily tell that the implication method is truncation with the min function. The max function is being used for the fuzzy OR operation. Rule 3 (the bottom-most row in the diagram shown opposite) is having the strongest influence on the output.



1. fuzzify inputs
2. apply fuzzy operation (or = max)
3. apply implication method (min)
1.
poor
rancid
cheap
If service is poor or food is rancid then tip = cheap
2.
good
rule 2 has no dependency on input 2
average
If service is good then tip = average
3.
excellent
delicious
generous
If service is excellent or food is delicious then tip = generous
service = 3
input 1
food = 8
input 2
tip = 16.7%
output