Linguistic Variables:
Clear Thinking with Fuzzy Logic

Walter Banks
Byte Craft Limited
A2-490 Dutton Drive
Waterloo, Ontario • N2L 6H7

Voice: (519) 888 6911
Fax: (519) 746 6751
Email: walter@bytecraft.com

Abstract:

Linguistic variables represent crisp information in a form and precision appropriate for the problem. For example, to answer the question "What is it like outside?", one might observe "It is warm outside." Experience has shown that if it is "warm" and the time is mid-day, a jacket is unnecessary, but if it is warm and early evening, it would be wise to take a jacket along (the day will change from warm to cool). The linguistic variables like "warm", so common in everyday speech, convey information about our environment or an object under observation.

We will show how linguistic variables can be defined and used in a variety of common applications, including home environment control, product pricing, and process control. The use of linguistic variables in many applications reduces the overall computation complexity of the application. Linguistic variables have been shown to be particularly useful in complex non-linear applications.

Linguistic variables are central to fuzzy logic manipulations, but are often ignored in the debates on the merits of fuzzy logic.

Linguistic Variables

What is a linguistic variable? Linguistic variables are used every day to express what is important and its context. 'This room is hot" is specific: it represents an opinion independent of measuring system, and it has information that most listeners will understand. Linguistic variables are used in ordinary daily activities, including preparation instructions for instant soup mixtures. These instructions are filled with linguistic references. *Bring to a boil*, *stirring constantly*, *reduce heat*, *partially cover*, *simmer*, *stirring occasionally* are all linguistic variables within the context of soup preparation. The manufacturers of instant soup apparently believe these seemingly vague instructions clearly tell the consumer how to make their product successfully.

The following examples show some of the ways linguistic variables can be formally defined and used in application software.
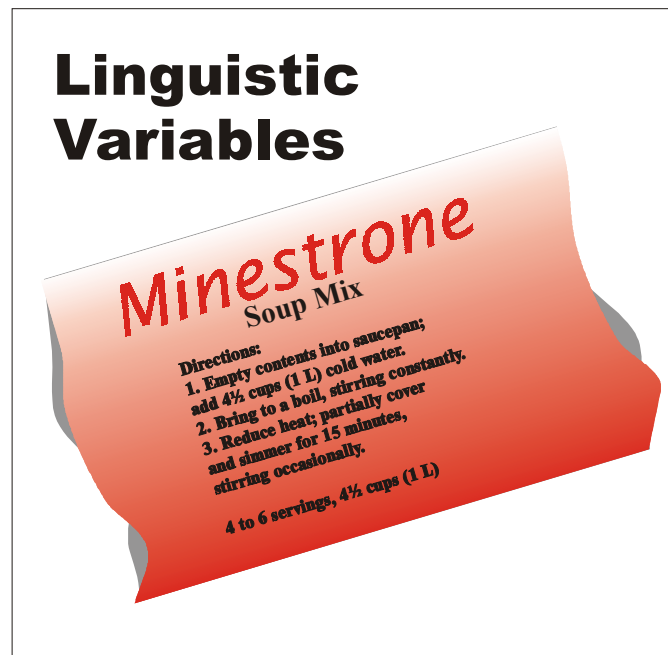

Figure 1: Preparation instructions for Minestrone Soup Mix

Directions:
1. Empty contents into saucepan; add 4½ cups (1 L) **_cold_** water.
2. **_Bring to a boil_**, **_stirring constantly_**.
3. **_Reduce heat_**; **_partially cover_** and **_simmer_** for 15 minutes, **_stirring occasionally_**.
4 to 6 servings, 4½ cups (1 L)

Example 1: Linguistic variables in soup instructions

An old friend comes into your shop asking to buy a few widgets, and wants your best price. The onus is on you to come up with a price given many parameters. Taking this hypothetical case we need to account for:

- Cost of the widgets

- Normal markup

- Shelf time of the product

- Shelf life of the product

- Length of the relationship

- Customer payment history

- Quantity of the sale

- Repeat business potential

Computerized record keeping will provide hard (*crisp*) numbers for many of the parameters. This still leaves the need to combine the numbers in some way to compute a discount from the normal list price, to quote to the customer.

The first surprise is that of all the parameters in the sale, all but "repeat business potential", are *crisp* numbers they can be precisely defined with information from a well-organized database. How should each of these parameters affect the final quotation discount?

Sale Parameter	What is important	Why it's important
Crisp measurement		
`Shelf_time`	`Long, Short`	Cost of keeping a product in inventory
`Shelf_life`	`Short, Long, Forever`	When shelf life is reached the product loses value
`Payment`	`Prompt, Normal, Eventually, Nagging`	Payment history is part of cost of doing business with him.
`Quantity`	`Small, Normal, Large, Huge`	There are shop savings and increased profit to selling larger quantities
`Customer`	`New, Recent, LongTerm`	Looks at potential that a new customer might get special treatment just as long term customers are rewarded for loyalty
Vague, subjective measurement		
Repeat business potential	`Yes, Maybe, No`	Both product and customer dependent.

Table 1: Parameters to consider pricing a widget

Table 1 identifies what is important in the decision making process. Each of the important items is context dependent. We can say "if the quantity is huge then profit is higher". *huge* is meaningful in the context of quantity, and *higher* is a consequence of *profit*. This is an example of a rule: an entire set of rules as follows will define the logic behind establishing a discounted price for the potential customer.

```
IF shelf_time IS long     THEN discount IS large
IF shelf_time IS short    THEN discount IS low
IF shelf_life IS short    THEN discount IS high
IF shelf_life IS Long     THEN discount IS normal
IF quantity   IS small    THEN discount IS none
IF quantity   IS large    THEN discount IS large
IF quantity   IS huge     THEN discount IS high
IF Customer   IS new      THEN discount IS special
IF customer   IS recent   THEN discount IS normal
IF customer   IS longterm THEN discount IS large
IF shelf_life IS short AND
   shelf_time IS long     THEN discount IS deep
```

This group of eleven rules (and perhaps a half-dozen more) can be used to establish computable pricing for many different products. This collection of rules individually describes the relationship between the sale parameters and the discount offered. These rules, when they are all evaluated, will provide a weighted value for the discount.

Appendix A has a copy of the pricing rules, as Fuzz-C™ source. The source (`pricing.fuz`), and its translation into standard C, are part of the code distributed with this paper. Fuzz-C™ is a preprocessor that effectively adds linguistic variable support to most C compilers. Fuzz-C translates linguistic variable declarations, consequence and fuzzy functions into standard C. You can write fuzzy logic directly into a C application program.

There is a book on the Byte Craft Limited website that has examples and an overview of the implementation details of linguistic variables in C. (http://www.bytecraft.com/fuzzybookform.html).

Using Linguistic Variables

```
IF room IS cold THEN heat IS on;
IF room IS hot THEN heat IS off;
```

Simple thermostats have been doing this for a hundred years or more. Why would we need linguistic variables and fuzzy logic to operate a simple switch? How do we evaluate a crisp temperature under such vague terms as hot and cold? What are *hot* and *cold* anyway?

The heating control problem sounds quite simple: we measure a temperature for the room, and use two fuzzy logic rules to control a furnace switch (heat). When the meanings of cold and hot are not precise opposites, the outcome becomes more complex and useful.

Linguistic variables associate a *linguistic condition* with a crisp variable. A crisp variable is the kind of variable that is used in most computer programs: an absolute value. A linguistic variable, on the other hand, has a proportional nature: in all of the software implementations of linguistic variables, they are represented by fractional values in the range of 0 to 1.

In the above example, *room* and *heat* are crisp variables, and *hot, cold, on* and *off* are linguistic variables. The linguistic variables *on* and *off* in the above example are represented in the crisp variable heat as a 1 and a 0 respectively. The *hot* and *cold* linguistic variables represent a range of values corresponding to the crisp variable *room*. This relationship can be represented as shown in the following graph.
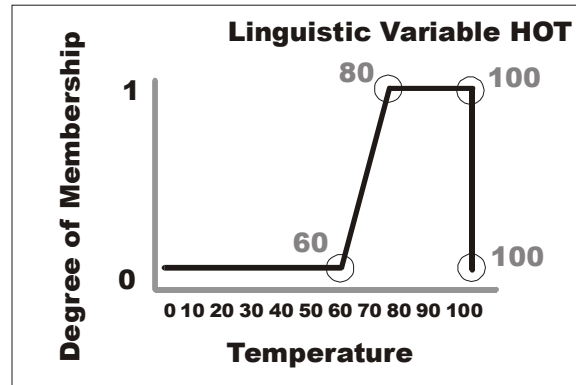
Figure 2: Linguistic variable HOT

Most linguistic variables can be represented in software with co-ordinates of 4 points. A crisp variable *room* is associated with a linguistic variable *hot*, defined using four break points from the graph.

```
LINGUISTIC room TYPE unsigned int MIN 0 MAX 100
  {
     MEMBER HOT { 60, 80, 100, 100 }
  }
```

A lot of literature has been written on representation of linguistic variables, but implementations for most applications utilize four points as above. There are arguments for smooth curves to represent linguistic variables for accuracy, and against smooth curves because of computational intensity. The worst-case error in 4-point presentation is in the corners. The robust nature of fuzzy logic rules in applications compensates for the simplistic representation of linguistic variables.

The Anatomy of a Fuzzy Rule

```
IF room IS cold THEN heat IS on;
```

Each fuzzy rule consists of two parts: a predicate and consequence part. The predicate determines the rule weight or truth. The result of the **room IS cold** is a *Degree of Membership* (DOM) value between the values of fuzzy zero and fuzzy one.

The DOM of the predicate weighs upon the consequence part of a fuzzy rule. In plain language, the urgency to turn the heat on with the above rule is determined by how cold the room is. A single fuzzy rule offers nothing over a crisp comparison and action; multiple competing rules do, however.

```
IF room IS cold THEN heat IS on;
IF outside_temperature IS hot THEN heat IS off;
IF day IS morning THEN heat IS off;
IF day IS afternnon OR day IS evening THEN heat IS on;
IF room IS hot THEN heat IS off;
```

Multiple independent rules are evaluated in parallel. Each rule contributes in a control system that smoothly goes from one dominant rule to the next.

All fuzzy rule calculations are done between fuzzy zero and fuzzy one. If more resolution is needed to make sure the calculations are accurate, the range remains constant and the number of bits in the DOM are increased. This effectively normalizes the problem space to the resolution of the DOM. As applications are developed, the resolution of the DOM is essentially determined by the resolution needed by the Consequence functions.

Logically combining Linguistic variables

Just as Boolean expressions can be combined to yield a Boolean result that represents the combined result of the expressions, so can linguistic variables. Linguistic variables can be combined with *or*, **and** and **not** operators. The following C defines can be used in application code

```
#define F_OR(a,b)  ((a) > (b) ? (a) : (b))
#define F_AND(a,b) ((a) < (b) ? (a) : (b))
#define F_NOT(a)   (F_ONE+F_ZERO-a)
```

Fuzzy *or* is the largest DOM of its arguments. Fuzzy **and** is the smallest of its arguments. Fuzzy **not** is the space between the argument and fuzzy 1. If the resolution of linguistic variables is reduced to have only the values of 0 and 1, the logical definitions for manipulating linguistic variables are the same as conventional Boolean logic.
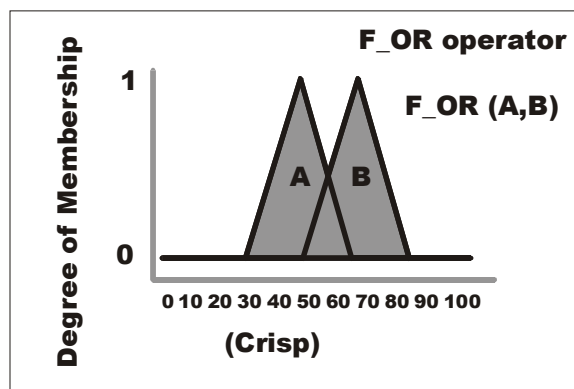

Figure 3: F_OR operator (Fuzzy OR)

PID Controller

The classical PID controller creates a manipulated variable signal as the sum of three terms: the first is the absolute error multiplied by a constant; the second is the rate of change of error multiplied by a second constant; the third is the accumulated error multiplied by a constant.

$$mv = (pe \times K1) + \frac{d\,pe}{d\,t} \times K2 + (\Sigma\,pe \times K3)$$

Figure 4: General equation for PID control system

Each of the three parts of the PID control system is (loosely) intended to: correct for errors; anticipate potential error conditions; overcome small accumulated errors (sticky bits). This layman's description comes close to what is needed to implement a PID controller using linguistic variables.

The approach we've taken to implement a linguistic variable-based PID controller uses three different control strategies keyed to the size of system error. If the error is large then the manipulated variable is driven primarily by the error value alone. Smaller errors are dominated by rate-of-error change rules. Finally, with very small errors, control is dominated by the integration of the error.

Linguistic variables can create a control system that is more tolerant of changes in system constants. Most real-world control applications are nonlinear. Some examples include airplane control systems, motor controllers, food and chemical processing; all these have system parameters of which vary widely in normal use.

Linguistic Time of Day

Many applications can refer to the time of day in linguistic terms. Implementation details for an example are shown in Appendix C. This example comes from a home environment application that divided the day into 0.1-hour segments, conveniently storing the crisp time of day into one byte. The following definitions show some creative usage of the definition of linguistic variables.

The crisp **hours** is a wrap-around number system that resets at midnight. The linguistic

variable *day* is conventional: it starts being "day" at 5:30 am and is truly "day" at 6:30 am; "day" continues to 5:30pm, where it declines until 6:30pm (this is a four point graph described above). The linguistic variable *night* avoids this complication by being defined, as a fuzzy function **hours** IS NOT *day*. Just as we do in our daily lives, we can define a new linguistic variable in terms of other defined linguistic variables. The night setback time (*nightsb*) is *night* but not *evening*.

```
//   hours   0.1 resolution hour 0 .. 240 for a day
 LINGUISTIC hours TYPE char MIN 0 MAX 240
   {
    MEMBER day      { 55 ,   65 ,   175 , 185 }
    MEMBER night   {FUZZY { hours IS NOT day }}
    MEMBER morning {50,   60 , 80 , 90 }
    MEMBER evening { 160 ,170 ,190 , 200 }
    MEMBER nightsb {FUZZY { hours IS night AND hours IS NOT evening }}
   }
```

Example 2: Environment control linguistic variable

Linguistic comparisons

Many crisp comparisons are not intended to match precisely a single value: fuzzy comparisons are available to solve the dilemma. We might want to say that someone is *about* 45. This means that 44 and 46 are significant and perhaps 40 and 50 are limits that are not relevant.

The crisp equality comparison is replaced with a fuzzy comparison that accounts for a range of data. We base a comparison on three data values: the comparison point, range until the comparison has failed (delta), and the current variable value. Delta is the distance to a value where the current comparison ceases to be important. Consider for a moment the following definitions; in each case the delta value returns a fuzzy zero or fuzzy one and any further deviation from the center point will not change the result. The following definition of Fuzzy equal shows a definition that is easily implemented.
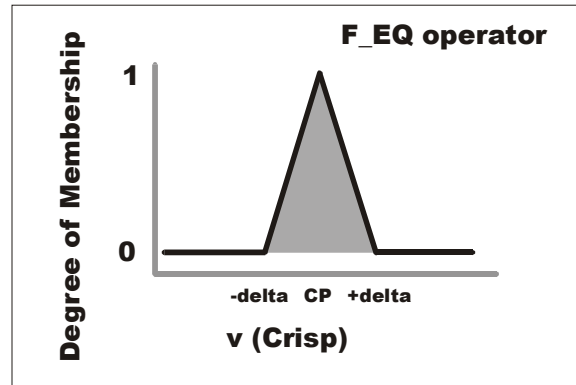
Figure 5: F_EQ (Fuzzy equal)

The arguments for **F_EQ** are: **v** for the crisp variable under test**, cp** for the center point, and **delta** which is the significant distance from the center point in the fuzzy comparisons. The **F_EQ** function can be used in any expression that accepts linguistic variables.

```
DOMtype F_EQ(v,cp,delta)
 {
  long m = ABS(cp-v);
  if (m > delta) return(F_ZERO);
    return( (m/delta) *
            (F_ONE-F_ZERO));
  }
```

In essence, **cp** and **delta** help declare an anonymous linguistic variable. This technique can be extended to all of the normal arithmetic comparisons.

Summary

Linguistic variables provide a normalized number system whose resolution is dependent on the consequence requirements of the application. Linguistic variables provide a natural smooth transition between competing rules describing different strategies. Linguistic rules focus on problem solution, not problem analysis.

The implementation of linguistic variables and their use work well on conventional embedded microprocessors, and are generally not as computationally intensive as alternative application implementations. The reduction of computation requirements is almost entirely due to the normalization of the data of interest to the application. Linguistic variables can easily be combined with conventional application software.

Linguistic variable types are taking their place alongside such other data types as *character*, *string*, *real* and *float*. They are an extension to the already familiar enumerated data types common in many high-level languages. The linguistic domain is simply another tool that application developers have at their disposal to communicate clearly. When applied appropriately, linguistic variable-based solutions are competitive with conventional algorithmic solutions, with considerably less implementation effort.

References:

An internet search will reveal a huge amount of fuzzy logic material. The following are a few references that you might find useful.

http://www.bytecraft.com/fuzzylogictools.html is our resources page for fuzzy logic.

http://www.bytecraft.com/fuzzybookform.html offers an online PDF file of a fuzzy logic book mentioned earlier. It is a collection of papers by Gordon Hayward and Walter Banks. It shows implementation details and examples of fuzzy logic using Fuzz-C™.

Earl Cox has written a number of good books on fuzzy logic, with many good implementation examples. "The Fuzzy Systems Handbook", originally published in the mid 1990s, is a very good read; it provides C++ source code for the examples and tools. Much of Dr. Cox's work has been looking for patterns in databases.

Appendix A

The `pricing.fuz` example.

```
LINGUISTIC Shelf_time TYPE int  MIN 0  MAX 1500
 {
   MEMBER long     { 60, 100,  1500, 1500 }
   MEMBER short    { 0,  0,    40,   60 }
 }

LINGUISTIC Shelf_life  TYPE int  MIN 0  MAX 1500
 {
  MEMBER Short     { 0, 0,  30,  60 }
  MEMBER Long      { 30,60, 300, 360 }
  MEMBER Forever   { 0, 0,1500,1500 }
 }

LINGUISTIC Payment  TYPE int  MIN 0  MAX 300
 {
   MEMBER Prompt      { 0,   0,  30,  45 }
   MEMBER Normal      { 30,  45,  60,  75 }
   MEMBER Eventually  { 60,  80, 180, 300 }
   MEMBER Nagging     { 75, 120, 300, 300 }
 }

LINGUISTIC Quantity  TYPE int  MIN 0  MAX 500
 {
  MEMBER Small   { 1, 3,  3,  5 }
  MEMBER Normal  { 2, 5,  7, 10 }
  MEMBER Large   { 7,10, 25, 30 }
  MEMBER huge    { 25,50,500,500 }
 }

LINGUISTIC Customer  TYPE int  MIN 0  MAX 150
 {
  MEMBER New      { 1, 1,  2,  2 }
  MEMBER Recent   { 2, 5, 10, 20 }
  MEMBER LongTerm { 5,10,150,150 }
 }

CONSEQUENCE discount TYPE float DEFUZZ cg
 {
  MEMBER deep    { 120 }
  MEMBER large   {  65 }
  MEMBER high    {  50 }
  MEMBER special {  35 }
  MEMBER normal  {  20 }
  MEMBER low     {   5 }
  MEMBER none    {   0 }
 }
```

```
FUZZY CalculateDiscount
   {
     IF shelf_time IS long     THEN discount IS large
     IF shelf_time IS short    THEN discount IS low
     IF shelf_life IS short    THEN discount IS high
     IF shelf_life IS Long     THEN discount IS normal
     IF quantity   IS small    THEN discount IS none
     IF quantity   IS large    THEN discount IS large
     IF quantity   IS huge     THEN discount IS high
     IF Customer   IS new      THEN discount IS special
     IF customer   IS recent   THEN discount IS normal
     IF customer   IS longterm THEN discount IS large
     IF shelf_life IS short AND
        shelf_time IS long     THEN discount IS deep
   }

void main (void)
   {
     //   . . . Application code to get crisp numbers for
     //         Cost Markup Shelf_time Shelf_life  Payment
     //         Quantity  Customer

    CalculateDiscount ();
    Quote = (cost + (cost * markup)) * ( 1.0 –
                          (discount / 100.0));
   }
```

Appendix B

PID control system implemented with lingustic variables.

```
int OldError,SumError;
int process(void);

LINGUISTIC Error  TYPE int  MIN -90  MAX 90
  {
   MEMBER LNegative    { -90, -90, -20,  0  }
   MEMBER normal       { -20,   0,  20      }
   MEMBER close        {  -3,   0,   3      }
   MEMBER LPositive    {   0,  20,  90,  90 }
  }
LINGUISTIC DeltaError  TYPE int  MIN -90  MAX 90
  {
   MEMBER Negative     { -90, -90, -10,  0  }
   MEMBER Positive     {   0,  10,  90,  90 }
  }
LINGUISTIC SumError  TYPE int  MIN -90  MAX 90
  {
   MEMBER LNeg    { -90, -90,  -5,  0  }
   MEMBER LPos    {   0,   5,  90,  90 }
  }

CONSEQUENCE  ManVar TYPE int   MIN -20  MAX 20 DEFUZZ cg
  {
   MEMBER  LNegative  {  -18 }
   MEMBER  SNegative  {   -6 }
   MEMBER  SPositive  {    6 }
   MEMBER  LPositive  {   18 }
  }

FUZZY pid
  {
   IF Error IS LNegative THEN ManVar IS LPositive
   IF Error IS LPositive THEN ManVar IS LNegative
   IF Error IS normal AND DeltaError IS Positive
     THEN ManVar IS SNegative
   IF Error IS normal AND DeltaError IS Negative
     THEN ManVar IS SPositive
   IF Error IS close AND SumError IS LPos
     THEN ManVar IS SNegative
   IF Error IS close AND SumError IS LNeg
     THEN ManVar IS SPositive
  }
```

```
void main (void)
 { while(1)
     {
      OldError = Error;
      Error = Setpoint - Process();
      DeltaError = Error - OldError;
      SumError = SumError + Error;
      pid();
     }
}
```

Appendix C

Lingustic variables for time of day.

```
#define fifty // powerline frequency
#ifdef fifty
#define rollover 100
#else
#define rollover 120
#endif

char ticks;    // 1/60 sec tics roll over at 120
               // 1/50 sec tics roll over at 100
char seconds; // actually two seconds 180 = 6 min

//  hours;     .1 resolution hour 0 .. 240 for a day
 LINGUISTIC hours TYPE char MIN 0 MAX 240
    {
     MEMBER day     { 55 ,  65 ,  175 , 185 }
     MEMBER night   {FUZZY { hours IS NOT day }}
     MEMBER morning {50,  60 , 80 , 90 }
     MEMBER evening { 160 ,170 ,190 , 200 }
     MEMBER nightsb {FUZZY { hours IS night AND hours IS NOT
                                            evening }}

    }


void time (void)
/* called each 1/60 of a second */
 {
   if (++ticks >= rollover)
     {
      ticks = 0;
      if (++seconds >= 180)
       {
         seconds = 0;
         if (++hours >= 240)
         { // new day
          hours = 0;
         }
       }
     }
 }
```

Walter Banks

Byte Craft Limited

A2-490 Dutton Drive

Waterloo, Ontario • N2L 6H7

Voice: 1 (519) 888 6911

Email: walter@bytecraft.com

Walter Banks is the president of Byte Craft Limited, a company specializing in software development tools for embedded microprocessors. His interests include highly reliable system design, code generation technology, and programming language development and standards. Walter Banks is a member of the Canadian delegation to ISO WG-14, where he co-authored WDTR 18037 (a technical report on C language extensions to support embedded processors). He has co-authored one book, and numerous journal and conference papers.

Byte Craft Limited is a software development company specializing in embedded systems software development tools for single-chip microcomputers. We provide innovative solutions for developers, consultants and manufacturers around the world. Our main products are C cross-compilers targeted to a variety of microcontroller families.

Edited by Kirk Zurell.