C++ Classes for Fuzzy Logic

Thomas E. Janzen

Thomas E. Janzen has written about the use of computers in music, multimedia performance, and measurement. He has written software to test electronic hardware for engineering and manufacturing. He is currently seeking new opportunities from Hudson, Massachusetts, and can be contacted at (508)562-1295 or tej@world. std.com.

Everything in life is not black and white, except perhaps that a software engineer with a future keeps abreast of new technology. Sometimes a half-truth is all you have. Your data is noisy, or easily misinterpreted; conditions are ambiguous; the system to be emulated is too complex to model. In these kinds of situations, fuzzy systems can be very useful. Fuzzy systems can make decisions with poor and missing data, and control a system for which there is no model.

Fuzzy systems are frequently developed by hardware engineers in C, but C++ may be more appropriate. Fuzzy values, like complex numbers, have their own operations and representation. Fuzzy sets and matrices are naturals for representation as container classes. With operator overloading, developers can express fuzzy operations as simple arithmetic and logic, simplifying their coding task. From a more philosopical perspective, fuzzy systems resemble object-oriented systems in the sense that they can operate without a model. In OO systems, abstraction presents a top-level repertoire of behavior without burdening the client with implementation details. In a similar fashion, the internal behavior of systems that are controlled with fuzzy logic can be hidden. In this article, I provide examples of fuzzy logic using classes in C++. I provide basic operations for ordinary truth values, classes for fuzzy sets, and classes for Fuzzy Associative Memories, as well as membership functions for all classes.

Implementing Fuzzy Values

A fuzzy truth value can be anything from 0.0 to + 1.0, inclusive:

```
0.0 <= x <= 1.0
```

In Listing 1, I use the built-in *double* type for truth values in the fuzzy class. A value of 0.0 represents falsehood or absence of a condition, with no ambiguity. 1.0 represents truth or presence of a condition. If I am half in a room, then the statement "I am in the room" is true to 0.5. My member and friend functions include the following: a constructor and destructor; a *get* function for retrieving the *double* value for truth, ! (a unary complement); functions *very, somewhat,* and associated logic (|, &, implies, iff); assignment; comparisons (<, >, <=, >=, ==, !=) and combination operators (|=, &=); output (<<); and input (>>).

These functions have a variety of definitions in the literature (see Klir). I have implemented what seem to be the most popular versions here. For example:

```
x|y is x OR y, the maximum of 2 values.


x&y is x AND y the minimum of 2 values


!x is NOT(x) == 1.0 - x
```

The following partial truth tables result from the preceding relationships:

```
x   y   x | y
----------------
0.0    0.0     0.0
1.0    0.0     1.0
0.0    1.0     1.0
1.0    1.0     1.0
0.5    0.4     0.5
0.6    0.0     0.6
```

```
x   y   x & y
------------------
0.0    0.0     0.0
0.0    1.0     0.0
1.0    0.0     0.0
1.0    1.0     1.0
1.0    0.5     0.5
0.5    0.0     0.0


x   !x
--------------
0.0     1.0
0.0     1.0
0.5     0.5
0.4     0.6
```

Fuzzy logic also offers "very" and "somewhat" operations. Suppose that I'm half in the kitchen. It may be true to 0.5 that I'm in the kitchen, but it's less true that I'm "very" in the kitchen. The *very* operation is the

square of the truth value. Since fuzzy truth is always between 0.0 and 1.0, when *very* operates on a truth value, it always produces a smaller value, corresponding to a less true condition. It is 0.25 (one quarter) true that I am "very" in the kitchen. The *somewhat* operation is the opposite. If it's haft true that I'm in the kitchen, it is more true to say that I'm somewhat in the kitchen. The *somewhat* operation is a square root; after being processed by *somewhat,* a truth value always gets bigger, or *more* true. So, it is 0.707... true that I am "somewhat" in the kitchen.

There is also an "implies" relationship, that is: x implies y. This relationship is calculated as follows:

```
{(1.0 - x + y) <= 1.0 : (1.0 - x + y)
{(1.0 - x + y) > 1.0 : 1.0
```

This notation is shorthand for a conditional expression; in English it would state "If (1.0 - x + y) is less than 1.0, return fuzzy value (1.0 - x + y). If (1.0 - x + y) is greater than or equal to 1.0, return fuzzy value 1.0." Another operation of interest is the *IFF* (if and only if) function. In fuzzy logic, this is calculated as:

```
1 - |x - y| = !|x - y|
```

that is, the complement of the absolute value of the difference of x and y. A sample truth table follows:

```
x   y   IFF   x, y
0.0   0.0    1.0
0.0   1.0    0.0
1.0   0.0    0.0
1.0   1.0.   1.0
0.6   0.4    0.8
```

For discrete values, this function is identical to an Exclusive NOR

Fuzzy Sets

A fuzzy set is a collection of fuzzy truth values. It is implemented in Listing 2 as an array of fuzzy values (I am imitating a vector example provided by Hansen in *The C++ Answer Book* — see Bibliography). Typically, the array corresponds to an array of membership functions for the spectrum or range involved. The private data areas of *fzy_set* are *dimension* and **fzy_data*. The public functions include the following: contructors (one that accepts an array length and a copy contractor); a destructor, assignment; binary comparison operators (<, <=, ==, >=, >, !=); subscripting ([]); output <<; *max* (find the maximum value in the set); and a multiplier for fuzzy sets and fuzzy associative memories (described later in the article). Some of the binary comparisons can be interpreted as set membership, as in the following examples:

```
==   the two sets are improper subsets of one another;
<    the first set is a proper subset of the second;
>  the second set is a proper subset of the first;
<=   the first set is a subset of the second;
>=   the second set is a subset of the first.
```

The subscript operator ([]) returns a fuzz value object.

Fuzzy Associative Memories

Kosko calls a matrix of fuzzy values a "fuzzy associate memory" (FAM). In an operation that is similar to transforming a vector with a matrix, a FAM operates on a fuzzy set to produce a new fuzzy set from which some answer is extracted. That is, "multiplication" of a fuzzy set by a FAM produces a decision dictating the next action for the fuzzy system to take. This operation transforms the fuzzy set input into a fuzzy set output. FAMs, like matrices from linear algebra, may be of any dimension. In Listing 3, I have implemented only a two-dimensional FAM. Called *fam_2*, it is a container class that holds a one-dimensional array of fuzzy sets, all of the same length. (Because the sets must all have the same length for the * operator, it might have been better to create a two-dimensional array of fuzzy objects.) I throw no exceptions for mismatched lengths of the contained fuzzy sets, since my compiler lacks exception handling. The private data area includes the two dimensions (*set_qty* and *set_len*) and an array of pointers to *fzy_set* objects (***fam_data*). The fam-2 public functions include constructors and a destructor, an assignment operator, a subscripting operator ([], returning a *fzy_set*), output (< <, borrowing from Hansen's matrix example), and the * operator, which is a *friend* of both *fam_2* and *fzy_set*.

I declare the * operator to work this way:

```
fzy_set seta(3), setb(3);
fam_2   fam(3,3);
...//fill in the data in the sets and fam


setb = seta * fam;
```

This operation compares each corresponding member of *seta* and *fzy_set* in *fam*; it saves the minimum value of either set in each position in a temporary *fzy_set*. After the comparison of each whole *fzy_set*, the operator saves the maximum in the temporary *fzy_set* in *setb*. This operation is called a "max-min" and is a common decision-making operation in fuzzy systems.

Fuzzy Membership Functions

Fuzzy membership functions, implemented in Listing 4, input any value and return the truth value of a statement asserting that the input value belongs to a particular condition. (Finding the membership value for a condition is also called fuzzyfying or fuzzification.)

I have assumed that the input value is a *double*, as it typically is; but you could create functions that operate over a complex plane, or in multiple real dimensions. A more abstract class would be altogether empty, which is certainly acceptable in a thoroughly-designed inheritance tree.

I provide an abstract class called *membership_function*, with two public virtual functions (*membership* and *weight*) and no private data area. The *membership* function accepts a *double* and returns a fuzzy value representing the membership of the *double* value in the membership function of the object. The *weight* function finds the area of the trapezoid under a horizontal line that intersects the trapezoid at the same place as a particular vertical line. This vertical line intersects the trapezoid at a location determined by the input value (see Figure 1) . You can inherit from this class while creating almost any membership function accepting a type *double* input. The *trapezoidal-membership* class inherits from the abstract class.

A situation modeled by trapezoidal membership function is a train in a tunnel, when the tunnel is longer than the train. Before the train enters the tunnel, it is false to say that its position is in the membership function. As the engine enters the tunnel, part of the train is in the tunnel and part is still out, so the membership function is between zero and one, and increases as the train continues. When the whole train is in the tunnel, its position is fully a member of the function, i.e., to 1.0.

Sometimes a membership function needs to be open on one side so that all values to one side are in the function. The class I provided does not really permit this, but certainly the values for one side or the other could be set well outside expected input values, for example, using *HUGE_VAL* or *-HUGE_VAL*, a macro from the C header *math.h*. Alternatively, you could implement a special class for such a function by inheriting from the *membership_function* class just as class *trapezoid_membership* does.

If the values for the corners of the membership functions are out of order or incorrect, I force them to valid values. An alternative to this would have been sorting the values. Also, If my compiler had supported exceptions, I could have thrown an exception for invalid values.

Relating Classes and Operations

I now give an example to illustrate the relationships between the classes and operations. Suppose I decide how much to open a window in the bedroom depending on the temperature outside. I will create four trapezoidal membership functions for outside temperature. These functions are defined in the fuzzy sets *cold, cool, warm* and *hot* in Listing 5. These sets are designed to overlap at a value of *0.5*. There are also three fuzzy sets for the output, the window opening in inches. The output sets are called *closed,*

part_open, and *open*. Between the input sets and the output sets is a FAM called *memory*.

First, I ask for an outside temperature and use it to find the fuzzy set *temperature_set*. Then I "multiply" *temperature_set* by the FAM *memory* to return the output fuzzy set, *set_result*. I find and store weights for the fuzzy values from *set_result* in *weights*, an array of *double*. I then calculate a centroid for these weights (per Kosko), and print the resulting opening in the window.

Some results from the program are:

```
<I>Temperature Opening</I>


0 degrees: 5.67 inches
20 degrees: 9 inches
40 degrees: 9 inches
60 degrees: 10.37
80 degrees: 9 inches
```

set_result is found from:

```
set_result = temperature_set * memory
```

Conclusion

Fuzzy logic is being applied widely today, both as a standalone solution to difficult real-world problems and as a component of hybrid systems. It is typically combined with such tools as neural-networks, symbolic AI, and databases. At the least, fuzzy logic is being applied to medical diagnosis, tracking and system control, information filtering, and decision support. As fuzzy systems grow in complexity, they may need to draw upon the values and methodologies developed in software engineering. C++ and object-oriented methodology will help bring solid discipline to a fuzzy domain.

Bibliography

Ellis, Margaret A., and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Reading, MA: Addison-Wesley, 1990.

Hansen, Tony L. *The C++ Answer Book*. Reading, MA: Addison-Wesley, 1990.

Klir, George J., and Tina A. Folger. *Fuzzy Sets, Uncertainty, and Information*. Englewood Cliffs, NJ: Prentice Hall, 1988.

Kosko, Bart. *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine*

Intelligence. Englewood Cliffs, NJ: Prentice Hall, 1992.

Stewart, Ian. "A Partly True Story." *Scientific American*, February 1993, Volume 268 Number 2, p110-ff.

Stroustrup, Bjarne. *The C++ Programming Language. 2 ed. Reading, MA: Addison-Wesley*, 1991, 1992.

Sidebar: "What is Fuzzy Logic?"

Page 1

Thomas E. Janzen

Thomas E. Janzen has written about the use of computers in music, multimedia performance, and measurement. He has written software to test electronic hardware for engineering and manufacturing. He is currently seeking new opportunities from Hudson, Massachusetts, and can be contacted at (508)562-1295 or tej@world. std.com.

Everything in life is not black and white, except perhaps that a software engineer with a future keeps abreast of new technology. Sometimes a half-truth is all you have. Your data is noisy, or easily misinterpreted; conditions are ambiguous; the system to be emulated is too complex to model. In these kinds of situations, fuzzy systems can be very useful. Fuzzy systems can make decisions with poor and missing data, and control a system for which there is no model.

Fuzzy systems are frequently developed by hardware engineers in C, but C++ may be more appropriate. Fuzzy values, like complex numbers, have their own operations and representation. Fuzzy sets and matrices are naturals for representation as container classes. With operator overloading, developers can express fuzzy operations as simple arithmetic and logic, simplifying their coding task. From a more philosopical perspective, fuzzy systems resemble object-oriented systems in the sense that they can operate without a model. In OO systems, abstraction presents a top-level repertoire of behavior without burdening the client with implementation details. In a similar fashion, the internal behavior of systems that are controlled with fuzzy logic can be hidden. In this article, I provide examples of fuzzy logic using classes in C++. I provide basic operations for ordinary truth values, classes for fuzzy sets, and classes for Fuzzy Associative Memories, as well as membership functions for all classes.

Implementing Fuzzy Values

A fuzzy truth value can be anything from 0.0 to + 1.0, inclusive:

```
0.0 <= x <= 1.0
```

In Listing 1, I use the built-in *double* type for truth values in the fuzzy class. A value of 0.0 represents falsehood or absence of a condition, with no ambiguity. 1.0 represents truth or presence of a condition. If I am half in a room, then the statement "I am in the room" is true to 0.5. My member and friend functions include the following: a constructor and destructor; a *get* function for retrieving the *double* value for truth, ! (a unary complement); functions *very, somewhat,* and associated logic (|, &, implies, iff); assignment; comparisons (<, >, <=, >=, ==, !=) and combination operators (|=, &=); output (<<); and input (>>). These functions have a variety of definitions in the literature (see Klir). I have implemented what seem to be the most popular versions here. For example:

```
x|y is x OR y, the maximum of 2 values.


x&y is x AND y the minimum of 2 values


!x is NOT(x) == 1.0 - x
```

The following partial truth tables result from the preceding relationships:

```
x   y   x | y
----------------
0.0    0.0     0.0
1.0    0.0     1.0
0.0    1.0     1.0
1.0    1.0     1.0
0.5    0.4     0.5
0.6    0.0     0.6


x   y   x & y
------------------
0.0    0.0     0.0
0.0    1.0     0.0
1.0    0.0     0.0
1.0    1.0     1.0
1.0    0.5     0.5
0.5    0.0     0.0


x   !x
--------------
```

```
0.0     1.0
0.0     1.0
0.5     0.5
0.4     0.6
```

Fuzzy logic also offers "very" and "somewhat" operations. Suppose that I'm half in the kitchen. It may be true to 0.5 that I'm in the kitchen, but it's less true that I'm "very" in the kitchen. The *very* operation is the square of the truth value. Since fuzzy truth is always between 0.0 and 1.0, when *very* operates on a truth value, it always produces a smaller value, corresponding to a less true condition. It is 0.25 (one quarter) true that I am "very" in the kitchen. The *somewhat* operation is the opposite. If it's haft true that I'm in the kitchen, it is more true to say that I'm somewhat in the kitchen. The *somewhat* operation is a square root; after being processed by *somewhat,* a truth value always gets bigger, or *more* true. So, it is 0.707... true that I am "somewhat" in the kitchen.

There is also an "implies" relationship, that is: x implies y. This relationship is calculated as follows:

$$\begin{cases} (1.0 - x + y) <= 1.0 : (1.0 - x + y) \\ (1.0 - x + y) > 1.0 : 1.0 \end{cases}$$

This notation is shorthand for a conditional expression; in English it would state "If (1.0 - x + y) is less than 1.0, return fuzzy value (1.0 - x + y). If (1.0 - x + y) is greater than or equal to 1.0, return fuzzy value 1.0." Another operation of interest is the *IFF* (if and only if) function. In fuzzy logic, this is calculated as:

```
1 - |x - y| = !|x - y|
```

that is, the complement of the absolute value of the difference of x and y. A sample truth table follows:

```
x   y   IFF   x, y
0.0   0.0     1.0
0.0   1.0     0.0
1.0   0.0     0.0
1.0   1.0.    1.0
0.6   0.4     0.8
```

For discrete values, this function is identical to an Exclusive NOR

Fuzzy Sets

A fuzzy set is a collection of fuzzy truth values. It is implemented in Listing 2 as an array of fuzzy values (I am imitating a vector example provided by Hansen in *The C++ Answer Book* — see Bibliography).

Typically, the array corresponds to an array of membership functions for the spectrum or range involved. The private data areas of *fzy_set* are *dimension* and **fzy_data*. The public functions include the following: contructors (one that accepts an array length and a copy contractor); a destructor, assignment; binary comparison operators (<, <=, ==, >=, >, !=); subscripting ([]); output <<; *max* (find the maximum value in the set); and a multiplier for fuzzy sets and fuzzy associative memories (described later in the article). Some of the binary comparisons can be interpreted as set membership, as in the following examples:

```
==   the two sets are improper subsets of one another;
<    the first set is a proper subset of the second;
>  the second set is a proper subset of the first;
<=   the first set is a subset of the second;
>=   the second set is a subset of the first.
```

The subscript operator ([]) returns a fuzz value object.

Fuzzy Associative Memories

Kosko calls a matrix of fuzzy values a "fuzzy associate memory" (FAM). In an operation that is similar to transforming a vector with a matrix, a FAM operates on a fuzzy set to produce a new fuzzy set from which some answer is extracted. That is, "multiplication" of a fuzzy set by a FAM produces a decision dictating the next action for the fuzzy system to take. This operation transforms the fuzzy set input into a fuzzy set output. FAMs, like matrices from linear algebra, may be of any dimension. In Listing 3, I have implemented only a two-dimensional FAM. Called *fam_2*, it is a container class that holds a one-dimensional array of fuzzy sets, all of the same length. (Because the sets must all have the same length for the * operator, it might have been better to create a two-dimensional array of fuzzy objects.) I throw no exceptions for mismatched lengths of the contained fuzzy sets, since my compiler lacks exception handling. The private data area includes the two dimensions (*set_qty* and *set_len*) and an array of pointers to *fzy_set* objects (***fam_data*). The fam-2 public functions include constructors and a destructor, an assignment operator, a subscripting operator ([], returning a *fzy_set*), output (< <, borrowing from Hansen's matrix example), and the * operator, which is a *friend* of both *fam_2* and *fzy_set*.

I declare the * operator to work this way:

```
fzy_set seta(3), setb(3);
fam_2   fam(3,3);
...//fill in the data in the sets and fam


setb = seta * fam;
```

This operation compares each corresponding member of *seta* and *fzy_set* in *fam*; it saves the minimum value of either set in each position in a temporary *fzy_set*. After the comparison of each whole *fzy_set*, the operator saves the maximum in the temporary *fzy_set* in *setb*. This operation is called a "max-min" and is a common decision-making operation in fuzzy systems.

Fuzzy Membership Functions

Fuzzy membership functions, implemented in Listing 4, input any value and return the truth value of a statement asserting that the input value belongs to a particular condition. (Finding the membership value for a condition is also called fuzzyfying or fuzzification.)

I have assumed that the input value is a *double*, as it typically is; but you could create functions that operate over a complex plane, or in multiple real dimensions. A more abstract class would be altogether empty, which is certainly acceptable in a thoroughly-designed inheritance tree.

I provide an abstract class called *membership_function*, with two public virtual functions (*membership* and *weight*) and no private data area. The *membership* function accepts a *double* and returns a fuzzy value representing the membership of the *double* value in the membership function of the object. The *weight* function finds the area of the trapezoid under a horizontal line that intersects the trapezoid at the same place as a particular vertical line. This vertical line intersects the trapezoid at a location determined by the input value (see Figure 1) . You can inherit from this class while creating almost any membership function accepting a type *double* input. The *trapezoidal-membership* class inherits from the abstract class.

A situation modeled by trapezoidal membership function is a train in a tunnel, when the tunnel is longer than the train. Before the train enters the tunnel, it is false to say that its position is in the membership function. As the engine enters the tunnel, part of the train is in the tunnel and part is still out, so the membership function is between zero and one, and increases as the train continues. When the whole train is in the tunnel, its position is fully a member of the function, i.e., to 1.0.

Sometimes a membership function needs to be open on one side so that all values to one side are in the function. The class I provided does not really permit this, but certainly the values for one side or the other could be set well outside expected input values, for example, using *HUGE_VAL* or *-HUGE_VAL*, a macro from the C header *math.h*. Alternatively, you could implement a special class for such a function by inheriting from the *membership_function* class just as class *trapezoid_membership* does.

If the values for the corners of the membership functions are out of order or incorrect, I force them to valid values. An alternative to this would have been sorting the values. Also, If my compiler had supported exceptions, I could have thrown an exception for invalid values.

Relating Classes and Operations

I now give an example to illustrate the relationships between the classes and operations. Suppose I decide how much to open a window in the bedroom depending on the temperature outside. I will create four trapezoidal membership functions for outside temperature. These functions are defined in the fuzzy sets *cold, cool, warm* and *hot* in Listing 5. These sets are designed to overlap at a value of *0.5*. There are also three fuzzy sets for the output, the window opening in inches. The output sets are called *closed, part_open*, and *open*. Between the input sets and the output sets is a FAM called *memory*.

First, I ask for an outside temperature and use it to find the fuzzy set *temperature_set*. Then I "multiply" *temperature_set* by the FAM *memory* to return the output fuzzy set, *set_result*. I find and store weights for the fuzzy values from *set_result* in *weights*, an array of *double*. I then calculate a centroid for these weights (per Kosko), and print the resulting opening in the window.

Some results from the program are:

```
<I>Temperature Opening</I>


0 degrees: 5.67 inches
20 degrees: 9 inches
40 degrees: 9 inches
60 degrees: 10.37
80 degrees: 9 inches
```

set_result is found from:

```
set_result = temperature_set * memory
```

Conclusion

Fuzzy logic is being applied widely today, both as a standalone solution to difficult real-world problems and as a component of hybrid systems. It is typically combined with such tools as neural-networks, symbolic AI, and databases. At the least, fuzzy logic is being applied to medical diagnosis, tracking and system control, information filtering, and decision support. As fuzzy systems grow in complexity, they may need to draw upon the values and methodologies developed in software engineering. C++ and object-oriented methodology will help bring solid discipline to a fuzzy domain.

Bibliography

Ellis, Margaret A., and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Reading, MA: Addison-

Wesley, 1990.

Hansen, Tony L. *The C++ Answer Book*. Reading, MA: Addison-Wesley, 1990.

Klir, George J., and Tina A. Folger. *Fuzzy Sets, Uncertainty, and Information*. Englewood Cliffs, NJ: Prentice Hall, 1988.

Kosko, Bart. *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence*. Englewood Cliffs, NJ: Prentice Hall, 1992.

Stewart, Ian. "A Partly True Story." *Scientific American*, February 1993, Volume 268 Number 2, p110-ff.

Stroustrup, Bjarne. *The C++ Programming Language. 2 ed. Reading, MA: Addison-Wesley*, 1991, 1992.

Sidebar: "What is Fuzzy Logic?"