Fuzzy Systems for Control Applications:
The Truck Backer-U pper

In this column, we introduce another tool in the advanced automation toolbox: fuzzy logic. The basis of fuzzy logic lies in the ambiguity in our thinking about concepts such as big, tall, slow, or bright, whose meanings are not only context dependent, but also ambiguous within a particular context. Using fuzzy sets named by these ambiguous linguistic variables, we can build applications that can outperform many of their traditional counterparts. As an example of the application of this technology, we will develop a fuzzy control system that automatically backs up a truck to a specified point on a loading dock.

James A. Freeman

Crisp Sets and Fuzzy Sets

The topic of this column is fuzziness. Fuzziness can be defined as the ambiguity that can be found in the definition of a concept or the meaning of a word [Terano et al. 1992]. Let me illustrate the idea with a short parable.

We are on the beach where I have made a mound with the sand. I ask you to describe the mound.

"It is a large mound of sand," you say.

"Very good," I reply, as I remove a grain of sand from the pile. "Now describe the mound."

"It is still a large mound of sand," you respond, hoping that I do not intend to remove one grain at a time for the next several years as you see me reach for another grain. Mercifully, I pick up a shovel and begin to remove sand in larger quantities.

"Tell me when the mound is no longer large," I say. You let me proceed for a minute, then stop me, saying, "Now. Now it is a small mound of sand."

"How many grains of sand would I have to add back for it to become large again?" I ask you.

"That's not a fair question," you say. "There is no exact boundary between large and medium piles of sand."

"Aha!" I exclaim. "My point exactly!" Since it is very hot on the beach, we march off in search of shade and a cold beer — you’re buying, as I have just imparted valuable knowledge to you for which you are in my debt.

Think about all the ambiguous words we use regularly to describe things: big, small, tall, short, long, fast, slow, and so on. Each of those words can be used to describe sets of objects: small dogs, tall buildings. In the world of Boolean logic, boundaries between sets defined by a particular attribute are sharp. The set of small sand piles is distinct from the sets of medium and large sand piles. If a certain sand pile belongs to one of these sets, it cannot belong to the others. Such sets are called crisp. They have well defined boundaries, and there is no ambiguity regarding an object's membership in a crisp set.

A convenient way to represent a crisp set is by its characteristic function, that is, the function whose value is 1 for elements of the set and 0 outside the set. As an example, here is the characteristic function of the interval [2, 5]:

\[
\begin{align*}
\text{In[1]:=} & \ f[x_\_] := \text{If}[x \geq 2 \&\& x \leq 5, 1, 0]; \\
\text{In[2]:=} & \ \text{Plot}[f[x], \{x, 0, 10\}]
\end{align*}
\]

If we measure the size of sand piles in some units (say, kilograms or gigagrains), we could define the sets of small, medium, and large sand piles by the crisp membership functions shown in Figure 1.

![Characteristics functions for the sets of small, medium, and large sand piles.](Image)

FIGURE 1. Characteristic functions for the sets of small, medium, and large sand piles. The independent variable \(x\)-value) is the size of the sand pile in some chosen units. The values of the functions are always either 0 or 1.
First, however, we need to define some functions. We will apply these ideas to an actual problem.

We say that an element “belongs” to a fuzzy set if the value of the set’s membership function at that element is nonzero. The degree to which an element belongs to a particular set can be any number between zero and one inclusive. Notice that in Figure 2, certain sizes of sand pile (such as 1.5 and 2.5) belong to more than one set.

It is important to realize that we are not discussing uncertainty here; rather, class membership is equivocal in some cases. Class boundaries are often ambiguous in our minds, as in our speech; and this phenomenon is not simply a matter of our inability to be precise. None of us would likely accept $50.00 less in our paycheck even though our paycheck was close to what we normally receive; on the other hand, no one could reasonably argue that a pile of sand changed from one class to another.

Yet we often impose unrealistically precise class boundaries and control setpoints in software that we write. Consider the following rules from a hypothetical control system:

\[
\text{If( flow rate is greater than 3.4 and flow rate is less than 4.5) Then( set valve position to 12)}
\]

\[
\text{If( flow rate is greater than 4.5 and flow rate is less than 5.7) Then( set valve position to 30)}
\]

A crisp set is described by a characteristic function whose value is always either 0 or 1. A fuzzy set is defined by a membership function that takes values anywhere between 0 and 1. In a fuzzy system, we might represent the sets of small, medium, and large sand piles by the fuzzy membership functions shown in Figure 2.

A crisp set is described by a characteristic function whose value is always either 0 or 1. A fuzzy set is defined by a membership function that takes values anywhere between 0 and 1. For example, if we have a sand pile and we want to determine if it is a small pile, we might apply a characteristic function that outputs 1 if the sand pile is small and 0 otherwise. This function would be a crisp set.

We also need operations corresponding to intersection and union of sets. These operations are given by \( \text{Min} \) and \( \text{Max} \) of the membership functions:

\[
\text{If( flow rate is very low) Then( set valve position to open slightly)}
\]

\[
\text{If( flow rate is medium low) Then( set valve position to open somewhat)}
\]

where terms such as very low and open slightly are the names of fuzzy sets. We establish ambiguous class boundaries between very low and medium low, and between open slightly and open somewhat. Then we use fuzzy inferencing techniques to determine what action the system should take based on the measurement of the flow rate. The result is often a smoother transition between states, and a more effective control system, especially in circumstances where an accurate mathematical model of the system is difficult to determine. We will apply these ideas to an actual problem. First, however, we need to define some functions.

We also need operations corresponding to intersection and union of sets. These operations are given by \( \text{Min} \) and \( \text{Max} \) of the membership functions:

\[
\text{If( flow rate is medium low) Then( set valve position to open slightly)}
\]

where terms such as very low and open slightly are the names of fuzzy sets. We establish ambiguous class boundaries between very low and medium low, and between open slightly and open somewhat. Then we use fuzzy inferencing techniques to determine what action the system should take based on the measurement of the flow rate. The result is often a smoother transition between states, and a more effective control system, especially in circumstances where an accurate mathematical model of the system is difficult to determine. We will apply these ideas to an actual problem. First, however, we need to define some functions.
In[18] := union[functions__] := Max[functions]
intersection[functions__] := Min[functions]

Here is an example of the union of two fuzzy sets:

In[20] := Plot[union[fuzzySmall[x], fuzzyMedium[x]], {x, 0, 4}];

The Fuzzy Truck-Driving Academy

Let's apply this technology to a control problem, namely, that of backing up a truck to a specified point on a loading dock. The truck backer-upper problem has become a standard control problem. There are several varieties; the one we will work on here is a simple version taken from the work of Kong and Kosko [Kosko 1992]. Figure 3 shows a simple model. The object of the control system is to back up the truck so that it arrives perpendicular to the dock at position $(x_f, y_f)$. The point $(x, y)$ is the center of the rear of the truck, $\phi$ is the angle of the truck axis to the horizontal, and $\theta$ is the steering angle measured from the truck axis. The controller takes as input the position of the truck, specified by the pair $(x, \phi)$, and outputs the steering angle $\theta$.

The linguistic variables associated with the fuzzy sets for the $x$ position are: LE (left), LC (left center), CE (center), RC (right center), and RI (right). The following five functions define those sets:

In[21] := defineSet[LE, {{0, 1}, {10, 1}, {10, 1}, {35, 0}}];
defineSet[LC, {{30, 0}, {40, 1}, {40, 1}, {50, 0}}];
defineSet[CE, {{45, 0}, {50, 1}, {50, 1}, {55, 0}}];
defineSet[RC, {{50, 0}, {60, 1}, {60, 1}, {70, 0}}];
defineSet[RI, {{65, 0}, {90, 1}, {90, 1}, {100, 1}}];

We can assemble the sets in a list and plot them together.

In[22] := xSets[x_] := {LE[x], LC[x], CE[x], RC[x], RI[x]};
In[23] := Plot[Evaluate[xSets[x]], {x, 0, 100},
PlotStyle -> Dashing /@ {{.01}, {.02}, {.03}, {.04}, {.05}}]

For the output variable, the sets are: NB (negative big), NM (negative medium), NS (negative small), ZE (zero), PS (positive small), PM (positive medium), and PB (positive big).
In[37] := defineSet[NB, {{-30, 1}, {-15, 0}, {-15, 0}, {-15, 0}}];
defineSet[NM, {{-25, 0}, {-15, 1}, {-15, 1}, {-5, 0}}];
defineSet[NS, {{-12, 0}, {-6, 1}, {-6, 1}, {0, 0}}];
defineSet[ZE, {{-5, 0}, {0, 1}, {0, 1}, {5, 0}}];
defineSet[PS, {{0, 0}, {6, 1}, {6, 1}, {12, 0}}];
defineSet[PM, {{5, 0}, {15, 1}, {15, 1}, {25, 0}}];
defineSet[PB, {{18, 0}, {30, 1}, {30, 1}, {30, 1}}];

In[44] := steerSets[theta_] = 
{NB[theta], NM[theta], NS[theta], ZE[theta],
PS[theta], PM[theta], PB[theta]};

In[45] := Plot[Evaluate[steerSets[theta]], {theta, -30, 30},
PlotStyle -> Dashing /@ 
{{.01}, {.02}, {.03}, {.04}, {.03}, {.02}, {.01}}]

With these sets we define rules which we call fuzzy associations. For example, if the angle \( \phi \) is vertical, and the \( x \) position is right center, then we want to steer positive medium. Symbolically,

\[
\text{IF } \phi \text{ is VE AND } x \text{ is RC, THEN } \theta \text{ is PM.}
\]

Rather than write out each of these rules, we can assemble them in a matrix format called a fuzzy associative memory (FAM). The FAM for this problem appears in Table 1.

<table>
<thead>
<tr>
<th>LE</th>
<th>LC</th>
<th>CE</th>
<th>RC</th>
<th>RI</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB</td>
<td>PS</td>
<td>PM</td>
<td>PM</td>
<td>PB</td>
</tr>
<tr>
<td>RU</td>
<td>NS</td>
<td>PS</td>
<td>PB</td>
<td>PB</td>
</tr>
<tr>
<td>RV</td>
<td>NM</td>
<td>NS</td>
<td>PS</td>
<td>PM</td>
</tr>
<tr>
<td>VE</td>
<td>NM</td>
<td>NM</td>
<td>ZE</td>
<td>PM</td>
</tr>
<tr>
<td>LV</td>
<td>NB</td>
<td>N M</td>
<td>NS</td>
<td>PS</td>
</tr>
<tr>
<td>LU</td>
<td>NB</td>
<td>NB</td>
<td>NM</td>
<td>NS</td>
</tr>
<tr>
<td>LB</td>
<td>NB</td>
<td>NB</td>
<td>NM</td>
<td>NM</td>
</tr>
</tbody>
</table>

**TABLE 1.** Fuzzy associative memory for the truck backer-upper system. Each position in the matrix corresponds to a rule. For example, row 3, column 4 corresponds to the rule IF \( \phi \) is RV AND \( x \) is RC, THEN \( \theta \) is PM.

Given a specific input pair \((\phi, x)\), several rules may be applicable since the fuzzy sets overlap. The membership functions for the sets involved in each rule (such as LV, CE, and N S) are combined using AND (Minimum) and the result is evaluated at \((\phi, x)\) to get a new \( \phi \)-set. The \( \phi \)-sets from the different rules are then combined with OR (Maximum) to get the output \( \phi \)-set. Figure 4 illustrates the construction of an output set from two rules, given an input \((\phi_1, x_1)\).

\[
\theta_c = \frac{\int_{-\infty}^{\infty} \phi f(\phi) d\phi}{\int_{-\infty}^{\infty} f(\phi) d\phi}
\]

Let’s walk through a specific example. First, we must construct the FAM matrix:

\[
\text{FAM} = \{(\phi S, \phi M, \phi P, \phi B), (\phi N, \phi S, \phi P, \phi B), (\phi N, \phi N, \phi Z, \phi P, \phi M), (\phi N, \phi N, \phi N, \phi N, \phi N, \phi N, \phi N, \phi N)\}
\]

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{FAM} )</td>
</tr>
</tbody>
</table>

**TABLE 1.** Fuzzy associative memory for the truck backer-upper system. Each position in the matrix corresponds to a rule. For example, row 3, column 4 corresponds to the rule IF \( \phi \) is RV AND \( x \) is RC, THEN \( \theta \) is PM.

Suppose the configuration of the truck is given by \( \phi = 50.0 \) and \( x = 49.0 \). The values of the membership functions for the \( \phi \) and \( x \) attributes are:

In[48] := phiSets[50.] 
Out[48] := \{0., 0.4, 0.222222, 0., 0., 0., 0.\}

In[49] := xSets[49.] 
Out[49] := \{0., 0.1, 0.8, 0., 0., 0.\}
Notice that the \( \phi \) and \( x \) values are each members of two fuzzy sets, so this pair of inputs will result in four rule firings.

To construct the output fuzzy set, take the intersection (Min) of the fuzzy sets for each pair of \( \phi \cdot x \) attributes (such as \{LV, CE\}) and evaluate its membership function at the input \( (\phi, x) = (50., 49.) \). This gives the so-called antecedants matrix:

\[
\text{In}[50] = \text{Outer}[\text{Min}, \phi\text{Sets}[50.], x\text{Sets}[49.]] // \text{MatrixForm}
\]

\[
\text{Out}[50]//\text{MatrixForm} = \\
\begin{bmatrix}
0. & 0. & 0. & 0. & 0. \\
0. & 0.1 & 0.4 & 0. & 0. \\
0. & 0.1 & 0.222222 & 0. & 0. \\
0. & 0. & 0. & 0. & 0. \\
0. & 0. & 0. & 0. & 0. \\
0. & 0. & 0. & 0. & 0. \\
0. & 0. & 0. & 0. & 0.
\end{bmatrix}
\]

The nonzero locations in the antecedants matrix correspond to the four applicable rules in the FAM matrix. We now take the intersections of the entries in the FAM matrix with the corresponding values in the antecedants matrix. The result is a matrix of \( \theta \) membership functions:

\[
\text{In}[51] = \text{MapThread}[\text{Min}, \{\text{%, truckFAM}[\text{theta}]\}, 2] // \text{Short}[#, 3] &
\]

\[
\text{Out}[51]//\text{Short} = \\
\begin{bmatrix}
\text{Min}[0., \text{PS}[\text{theta}]], \text{Min}[0., \text{PM}[\text{theta}]], \\
\text{Min}[0., \text{PM}[\text{theta}]], \text{Min}[0., \text{PM}[\text{theta}]], \\
\text{Min}[0., \text{PB}[\text{theta}]], 0.
\end{bmatrix}
\]

Most of the entries in this matrix have the form \( \text{Min}[0., _] \) and so are equivalent to zero. Only those entries that correspond to applicable rules will be nonzero. The union (Max) of all the entries is the output fuzzy \( \theta \)-set:

\[
\text{In}[52] = \text{Max}[\text{Flatten}[\%]]
\]

\[
\text{Out}[52] = \text{Max}[\{0., \text{NB}[\text{theta}], \text{Min}[0., \text{NM}[\text{theta}]], \\
\text{Min}[0., \text{NS}[\text{theta}]], \text{Min}[0., \text{PB}[\text{theta}]], \\
\text{Min}[0., \text{PM}[\text{theta}]], \text{Min}[0., \text{PM}[\text{theta}]], \\
\text{Min}[0., \text{PM}[\text{theta}]], \text{Min}[0., \text{PB}[\text{theta}]], \\
\text{Min}[0., \text{PB}[\text{theta}]], \text{Min}[0., \text{PS}[\text{theta}]], \\
\text{Min}[0.222222, \text{PS}[\text{theta}]], \text{Min}[0.4, \text{PM}[\text{theta}]]\}]
\]

We can simplify this expression by removing the entries that are equivalent to zero:

\[
\text{In}[53] = \text{fuzzyTheta}[\text{theta}] = \% / . \text{Min}[0., _] \to 0
\]

\[
\text{Out}[53] = \text{Max}[\{0., \text{Min}[0.1, \text{NS}[\text{theta}]], \text{Min}[0.1, \text{PS}[\text{theta}]], \\
\text{Min}[0.222222, \text{PS}[\text{theta}]], \text{Min}[0.4, \text{PM}[\text{theta}]]\}]
\]

Here is the output fuzzy set:

\[
\text{In}[54] = \text{Plot}[\text{fuzzyTheta}[\text{theta}], \{\text{theta}, -30, 30\}]
\]

Now, we must defuzzify this set to get a numerical value. To find the centroid of the output fuzzy set, we should integrate the product of \( \theta \) and \( \text{fuzzyTheta} \) and divide by the integral of \( \text{fuzzyTheta} \), both integrations taken over the range of \( \theta \). In the function \text{defuzzify}, I use an approximation that will speed the calculation while resulting in a small error:

\[
\text{In}[55] = \text{defuzzify}[\text{fuzzyTheta}, \{\text{t}, \text{min}, \text{max}, \text{dt}\}] := \\
\frac{\text{Sum}[\text{t} \text{fuzzyTheta}, \{\text{t}, \text{min}, \text{max}, \text{dt}\}]}{\text{Sum}[\text{fuzzyTheta}, \{\text{t}, \text{min}, \text{max}, \text{dt}\}]}
\]

\[
\text{Out}[55] = 10.6783
\]

The output \( \theta \) value for our example is:

\[
\text{In}[56] = \text{defuzzify}[\text{fuzzyTheta}[\text{theta}], \{\text{theta}, -30, 30, .5\}]
\]

\[
\text{Out}[56] = 10.6783
\]

The function \( \text{steer} \) combines the steps above to compute the steering angle \( \theta \) for a given configuration \( (x, \phi) \):

\[
\text{In}[57] = \text{steer}[x_, \phi_] := \\
\text{defuzzify}[\text{Max}[\text{Flatten}[ \\
\text{MapThread}[\text{Min}, \{\text{Outer}[\text{Min}, \phi\text{Sets}[\phi], x\text{Sets}[x]], \\
\text{truckFAM}[\text{theta}], 2\}] / . \text{Min}[0., _] \to 0, \\
\{\text{theta}, -30, 30, .5\}\]]
\]

The function \( \text{simulateTruck} \) takes the initial values of \( x, y, \) and \( \phi, \) and computes a list of configurations \( \{(x, y, \phi)\} \) giving a trajectory for the truck until it reaches a \( y \) value of at least 95:

\[
\text{In}[58] = \text{simulateTruck}[x0_, y0_, phi0_] := \\
\text{Module}[\{x = x0, y = y0, phi = phi0, newPhi, result = \{\}\}, \\
\text{While}[y < 95, \\
\text{newPhi} = \phi + \text{steer}[x, \phi]; \\
\text{AppendTo}[\text{result}, \\
\{x, y, \phi\} = \\
\{x + 5 \text{Cos[newPhi Pi/180]}, \\
y + 5 \text{Sin[newPhi Pi/180]}, \text{newPhi}\} // \text{N}]; \}; \\
\text{result}
\]

Note that the stepsize (5) used in this function can easily be changed.
The position of the truck can be displayed with the function `showTruck`. It takes as inputs the \( (x, y, \phi) \) configuration and the length and width of the truck:

```mathematica
In[59]:= showTruck[{x_, y_, phi_}, {l_, w_}] := Module[{s = Sin[phi Pi/180]/N, c = Cos[phi Pi/180]/N}, Show[Graphics[
   Polygon[Transpose[
     {{-s w/2, s w/2, s w/2 - c l, -1.2 c l, -s w/2 - c l, -s w/2},
      {c w/2, -c w/2, -c w/2 - s l, -1.2 s l, c w/2 - s l, c w/2}} ]],
   Point[{0,0}],
   Line[{{0,100},{100,100}}],
   Line[{{50,100},{50,95}}],
   Axes -> True, AspectRatio -> Automatic,
   AxesOrigin -> {0,0}]];
```

For example:

```mathematica
In[60]:= showTruck[{20, 40, 45}, {10, 5}]
```

Let's compute the trajectory of a truck starting at the position shown above:

```mathematica
In[61]:= simulateTruck[20, 40, 45]
Out[61]= {{23.8857, 43.1466, 39.}, {28.0791, 45.8698, 33.},
{32.5341, 48.1397, 27.}, {36.8884, 50.5974, 29.441},
{40.962, 53.4967, 35.441}, {44.7102, 56.806, 41.441},
{48.0919, 60.4889, 47.441}, {50.698, 64.756, 58.5859},
{52.6265, 69.3691, 67.3134}, {53.6452, 74.2642, 78.2441},
{53.5617, 79.2635, 90.9564}, {52.6199, 84.174, 100.858},
{51.7959, 89.1057, 99.4856}, {50.9721, 94.0373, 99.4829},
{50.2237, 98.981, 98.6083}}
```

We can produce an animation of the moving truck by mapping the function `showTruck[#1, {10, 5}]&` onto this list of configurations. The resulting graphics objects can also be displayed in one frame by using `Show`. The individual positions of the truck are easier to see if we replace `Polygon` by `Line` in the function `showTruck`:

I have not explored all of the various starting positions, so I cannot guarantee that every one will result in an acceptable trajectory. You can experiment by changing the various parameters, such as the boundaries of the fuzzy sets, the step-size in `simulateTruck`, and so on.

A fuzzy logic implementation of a control system is often a good choice when a mathematical model of the system is either unavailable or too complex to simulate efficiently. Our example shows that it is fairly easy to construct a fuzzy controller for a system that might otherwise require quite an effort at mathematical modeling. Other interesting examples, both hypothetical and real-life, appear in [Terano et al. 1992, Kosko 1992, White and Sofge 1992].

References


James A. Freeman
Loral Space Information Systems, Artificial Intelligence Laboratory, P.O. Box 58487, Mailing Code F8H4A, Houston, TX 77258, (713) 335-6626
71340.3016@compuserve.com

The electronic supplement contains the notebook Fuzzy Control Systems.