Fuzzy Logic System Applications
CMP_SC 4770

Jonas Ferguson
JFR48

University of Missouri

1 Project Description

For this project, I have created a Mamdani fuzzy logic system by utilizing my own library of fuzzy logic functions. I will go over each of these functions, then show how to implement them for the Iris classification problem and for the inverted pendulum problem.

1.1 Fuzzy Logic System Functions

1.1.1 Membership Function Generations

The first type of function that I programmed for the fuzzy logic system are the membership function generators. I programmed these for triangular, trapezoidal, and Gaussian membership functions.

First, let's look at the triangular membership function. It takes 2 inputs: the domain, and the parameters. The parameters are composed of an a value, a b value, and a c value, which represent the left end of the triangle, the mid-point of the triangle, and the right end of the triangle respectively. As it is very helpful when on the ends of the domain, my implementation does allow for a and b or b and c to be the same, but notably does not work for all three values being the same. Additionally $a <= b <= c$ in general, given above specifications about equivalence.

This function is implemented as follows:

```
def trigMem(x, params):
    mem = []
    a,b,c = np.r_[params]
    for val in x:
        if (a != b):
            term1 = (val - a) / (b - a)
        else:
            term1 = 1
        if (b != c):
            term2 = (c - val) / (c - b)
        else:
            term2 = 1
        if(val < a - 0.00001 or val > c + 0.00001):
            value = 0
        else:
            value = max(min(term1,term2), 0)
        mem.append(value)
    return mem
```

We can visualize this function by running it over a domain from 0 to 10, with parameters of $a = 3$, $b = 5$, $c = 9$.

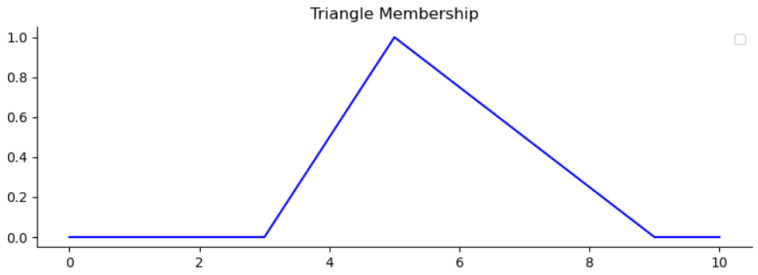

Figure 1: Triangle Membership Function

The next membership function is the trapezoidal membership function. It again takes two inputs: the domain and the parameters. The parameters are composed of an a value, a b value, a c value, and a d value, which represent the left edge of the shape, the left edge of the top side, the right edge of the top side, and the right edge of the shape. Like with the triangle, a and b or c and d can be the same to accommodate membership at the edge of the domain. In general $a <= b <= c <= d$. The function is implemented as follows:

```
def trapMem(x, params):
    mem = []
    a,b,c,d = np.r_[params]
    for val in x:
        if (a != b):
            term1 = (val - a) / (b - a)
        else:
            term1 = 1
        if (c != d):
            term2 = (d - val) / (d - c)
        else:
            term2 = 1
        if (a - 0.00001 <= val <= d + 0.00001):
            value = max(min(term1, term2, 1), 0)
        else:
            value = 0
        mem.append(value)
    return mem
```

We can visualize this function by running it over a domain from 0 to 10, with parameters of $a = 1$, $b = 4$, $c = 7$, and $d = 8$.

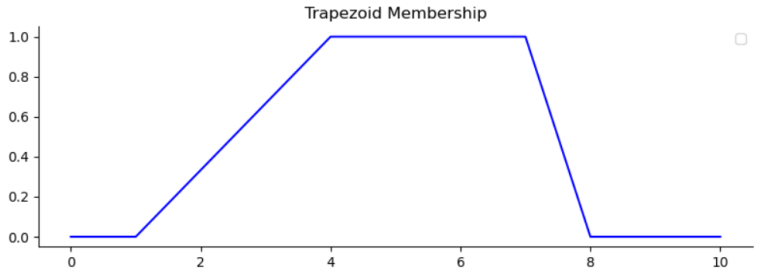

Figure 2: Trapezoid Membership Function

The final membership function is the Gaussian membership function. It takes two inputs: the domain and the parameters. The parameters includes a value for σ and one for c. c represents the center of the function and σ represents the standard deviation. We implement this function as follows:

```
def gaussMem(x, params):
    mem = []
    sigma, c = np.r_[params]
    for val in x:
        term1 = -1 * ((val - c) ** 2)
        term2 = 2 * (sigma ** 2)
        value = np.e ** (term1 / term2)
        mem.append(value)
    return mem
```

We can also visualize this function by running it over a domain from 0 to 10, with parameters of $\sigma = 1$ and $c = 5$

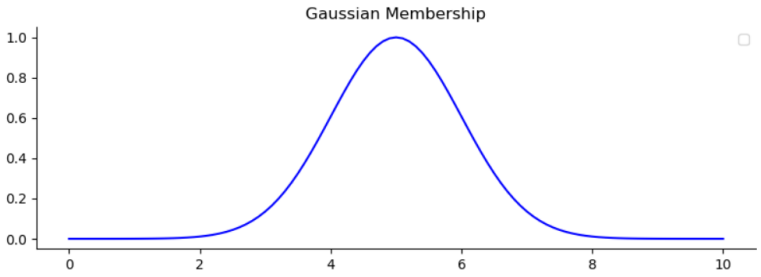

Figure 3: Gaussian Membership Function

1.1.2 Membership Interpretation

My fuzzy logic system implementation only stores membership at specific intervals across the domain, so we have to have a way to find membership when the values are not at that specific interval. For example, if only values for 6.4 and 6.5 are stored, but we need the membership at 6.43, we need a way to estimate this based on the values nearby. The interpMem function does this: it takes three arguments, the domain, the membership values, and the specific value we are looking for. It then finds the nearest stored membership values on either side, weighs these based on how close the target value is to each side, and then in calculates an estimate for the membership value. This function is interpreted as follows.

```
def interpMem(values, memFunc, x):
    lIndex = None

    # find index to the left
    for i in range(len(values)):
        if values[i] > x:
            lIndex = i - 1
            break

    if lIndex is None: #if the index is at the end of the array
        lIndex = len(values) - 2 # set to the second to last index
```

```
    left = memFunc[lIndex]
    right = memFunc[lIndex + 1]

    leftWeight = (values[lIndex + 1] - x) / (values[lIndex + 1] - values[lIndex])
    rightWeight = 1 - leftWeight

    return (leftWeight * left) + (rightWeight * right)
```

1.1.3 Implication Operators

The implementation of implication operators is quite straightforward. For this project, I implemented the Zadeh/Lukasiewicz operator, the correlation-min operator, and the correlation-product operator. I included all of these in a implication function which takes three arguments: the two fuzzy sets being compared, and then either a 0, 1, or 2 signifying the implication type, with 0 being Zadeh, 1 being correlation-min, and 2 being correlation-product. The implementation of this function is as follows:

```
def implication(x,y,type):
    if (type == 0):
        return np.fmin(1.0, 1.0 - x + y)
    elif (type == 1):
        return np.fmin(x, y)
    elif (type == 2):
        return np.multiply(x, y)
```

I will explore in later sections how the choice of implication operator affects the outcome of the fuzzy logic system.

1.1.4 Aggregation Operators

When running the fuzzy logic system, we need a way to combine multiple rules together with an aggregation operator. I have implemented the maximum and sum aggregation operators. Much like the implication operators, the aggregation operators are in an aggregation function which takes two arguments: an array containing all the rule results, and the aggregation type, with 0 representing maximum and 1 representing sum. This is implemented as follows:

```
def aggregate(rules, type):
    if (type == 0):
        return np.fmax.reduce(rules)
    if (type == 1):
        return sum(rules)
```

1.1.5 Defuzzification

Finally, we need a function that gets a single value following the application of all the rules on the fuzzy sets. For this project, I implemented centroid defuzzification. The implementation is very simple and consists of a centroidDF function which takes two arguments: the membership function of the output and the aggregated rules. This is implemented as follows:

```
def centroidDF(x, mf):
    sum = np.sum(mf)
    mfSum = np.sum(np.multiply(mf, x))
    return mfSum / sum
```

1.2 Iris Classification

The Iris classification problem consists of a data set of 150 iris individuals which belong to three species. Each iris includes measurements for sepal length, sepal width, petal length, and petal width, which we must use to classify each individual into its species. We can use a fuzzy logic system to solve this problem. We will create membership functions for each of the 4 inputs, a membership function for the species output, and define fuzzy rules in between.

1.2.1 Iris Membership Functions

For each of the four inputs, we define 3 membership functions for small, medium, and large. This is done with triangular functions measured at .1 intervals. The domains are as follows: sepal length (4-8), sepal width (2-5), petal length (1-7), petal width (0-3). The small, medium, and large functions attempt to divide these domains up roughly equally. This can be seen in the following figure.

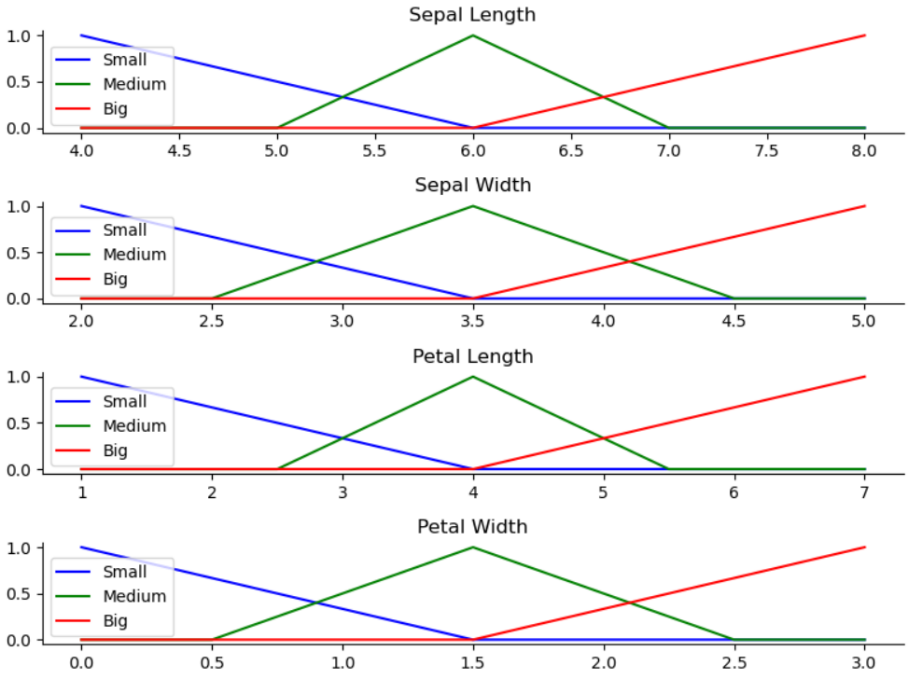

Figure 4: Iris Input Membership Functions

We will represent the output with three triangular functions corresponding to the three species.

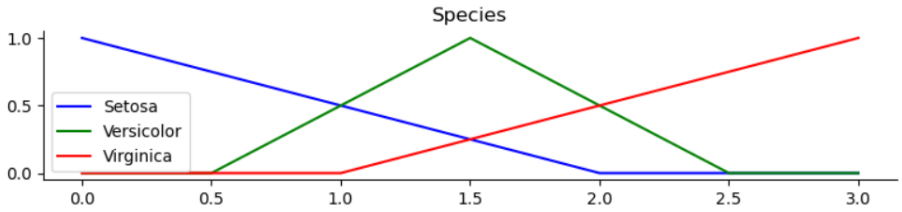

Figure 5: Iris Output Membership Function

1.2.2 Iris Classification Rules

For the base level implementation we use 3 rules: IF petal length and petal width are small THEN setosa, IF petal length is medium THEN versicolor, and IF petal length is big THEN virginica. Further discussion of the rules will occur later on in this paper.

1.2.3 Iris Aggregation and Defuzzification

After applying all the rules, we aggregate the results. From these aggregated results, we use centroid defuzzification to find an output value. We round down to the nearest integer and take species as follows: $0 \rightarrow$ I. setosa, $1 \rightarrow$ I. versicolor, $2 \rightarrow$ I. virginica. Here is an example of an individual which has been evaluated to be I. versicolor. We can see graphically the rule results (in orange) compared to the output membership functions (in blue, green, and red), and the result of centroid defuzzification (black line).

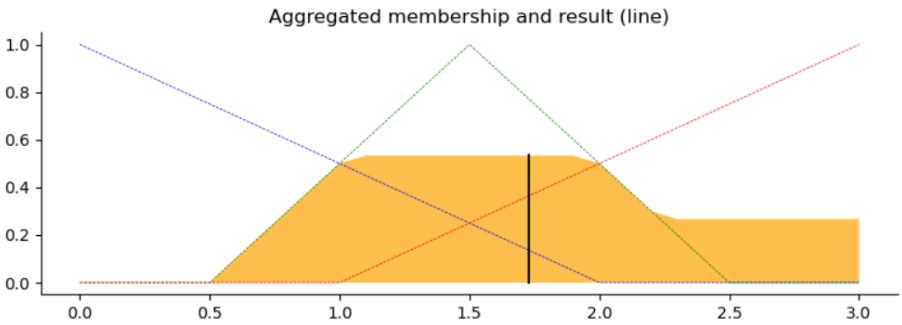

Figure 6: Iris Defuzzification Example

1.3 Inverted Pendulum Problem

The other application that we demonstrate for the fuzzy logic system is the inverted pendulum. This is a situation in two dimensions where a cart has a pole balanced on it. The pole tries to fall over, but the cart can move in a line to stop it from doing so. I have programmed my fuzzy logic system to take inputs of the pole's angle and angular velocity, and provide an output for if the cart should move to the left or to the right. I will break down the setup for this system.

1.3.1 Membership Functions

We have two inputs, angle and angular velocity, and one output, acceleration. For angle, we have two membership functions, left and right. They are each defined as Gaussian functions centered on the edge of the angle range (around -0.22 to 0.22 in radians). The angular velocity inputs are on a domain of -100 to 100 in radians per second. They are trapezoidal, start on each edge, remain high to 50, then shrink down to 0. This gives them the following values: counter-clockwise (-100, -100, -50, 0), clockwise (0, 50, 100, 100). Finally, the output has a domain on 0 to 1. We define left and right acceleration as triangular functions of (0, 0, 1), and (0, 1, 1) respectively.

We can visualize all of these membership functions in the figure below:

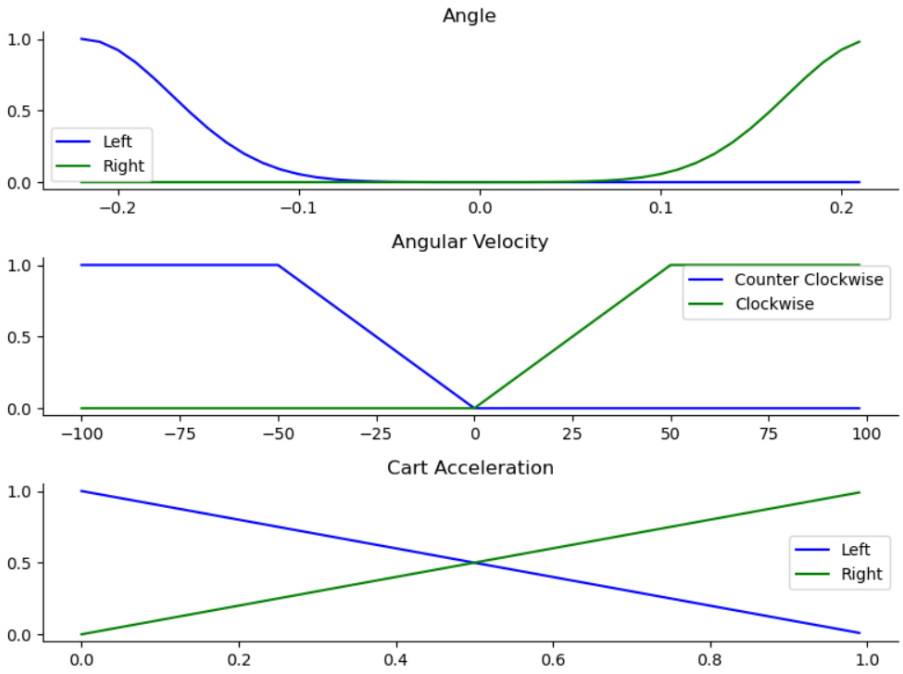

Figure 7: Pendulum Membership Functions

1.3.2 Rules and Aggregation

The inverted pendulum requires only simple and fairly intuitive rules that come in two types. The first type of rule is for observing a leaning angle. IF angle is left THEN acceleration is left. Similarly we get, IF angle is right THEN acceleration is right. The second type of rule is for observing angular velocity. IF velocity is counter-clockwise THEN acceleration is left. IF velocity is clockwise THEN acceleration is right.

We aggregate all these rules together and then use centroid defuzzification to get an acceleration value from 0 to 1. The simulator takes only a 0 or a 1, so we round the number and use that as our acceleration value.

1.3.3 Simulation

To simulate the inverted pendulum, I used the Gymnastics library which contains a premade inverted pendulum simulator, which it calls cart pole. The simulator is very easy to interact with, it returns an observation, which is an array containing cart position, cart velocity, pole angle, and pole angular velocity. To run each step of the simulator, we pass in a 0 to accelerate left or a 1 to accelerate right. The simulator continues to run until one of two conditions are met. Either we tip over past around 12 degrees in either direction, or we reach 500 time steps. Reaching 500 time steps while staying balanced is the win condition of the simulation. I run the simulation for 1000 time steps, which will be 2 full runs of 500 time steps if everything remains balanced. While I cannot show the simulation itself in this paper (as this cannot contain video), here is an image of what the simulator looks like, further behavior will be described to the best of my ability.

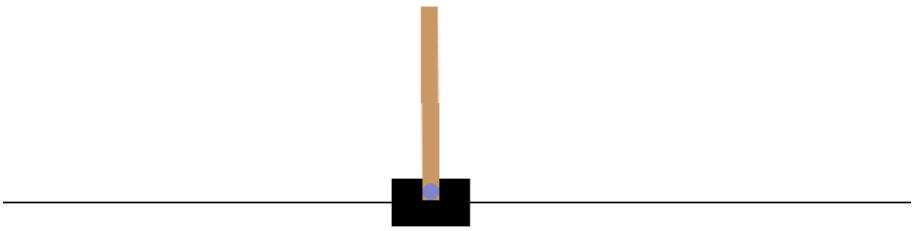

Figure 8: Cart Pole Simulator

2 Experiments and Results

2.1 Validating Fuzzy Logic Functions

Before we can begin to run tests on application problems using this fuzzy logic system, we must first mathematically verify that it works as intended, function by function.

2.1.1 Membership Function Generators

We start by verifying the triangular, trapezoidal, and Gaussian membership function generators. We will compare the results of the function to hand calculations. To verify the triangular membership function generator, I ran it twice over a domain of (0-10) with intervals of 1. The first run had parameters of $a = 2$, $b = 5$, and $c = 6$. The second run had parameters of $a = 1$, $b = 1$, $c = 8$. We compare these computed results to the results calculated with the triangular function defined as $f(x; a, b, c) = max(min(\frac{x-a}{b-a}, \frac{c-x}{c-b}), 0)$. The results are listed below, rounded to three decimal places.

Triangular Membership Function				
Domain	Test 1 Computed	Test 1 Manual	Test 2 Computed	Test 2 Manual
0	0	0	0	0
1	0	0	1	1
2	0.0	0	0.857	0.857
3	0.333	0.333	0.714	0.714
4	0.667	0.667	0.571	0.571
5	1.0	1	0.429	0.429
6	0.0	0	0.286	0.286
7	0	0	0.143	0.143
8	0	0	0.0	0
9	0	0	0	0
10	0	0	0	0

Ignoring divide by zero errors in the manual calculations and some minor floating point errors not seen here due to rounding, we find that the triangular membership function implementation stays true to the intended equation.

We now work to verify the trapezoidal membership function. We will again run two tests over a domain of (0-10) with intervals of 1. The first run has parameters of $a = 1$, $b = 2$, $c = 5$, and $d = 9$. The second run has parameters $a = 1$, $b = 1$, $c = 3$, and $d = 9$. We will compare these computed results to manual calculations of the expected trapezoidal membership function which is $f(x; a, b, c, d) = max(min(\frac{x-a}{b-a}, 1, \frac{d-x}{d-c}), 0)$. The results are summarized below, rounded to three decimal places.

Trapezoidal Membership Function				
Domain	Test 1 Computed	Test 1 Manual	Test 2 Computed	Test 2 Manual
0	0	0	0	0
1	0.0	0	1	1
2	1.0	1	1	1
3	1	1	1	1
4	1	1	0.833	0.833
5	1.0	1	0.667	0.667
6	0.75	0.75	0.5	0.5
7	0.5	0.5	0.333	0.33
8	0.25	0.25	0.167	0.167
9	0.0	0	0	0
10	0	0	0	0

Once again, if we ignore divide by zero errors in the manual calculations and floating point errors in the computation that get rounded out anyway, we get the same answer for both test functions.

We now look to verify the Gaussian membership function. This will be run over two tests, again with a domain of (0-10) with intervals of 1. The first run has parameters of $c = 5$, $\sigma = 1$, and the second has parameters of $c = 0$, $\sigma = 5$. We will compare these computed results with results from the Gaussian membership function, which is $f(x; \sigma, c) = e^{\frac{-(x-c)^2}{2\sigma^2}}$ The results are below, rounded to three significant figures.

Gaussian Membership Function				
Domain	Test 1 Computed	Test 1 Manual	Test 2 Computed	Test 2 Manual
0	3.72×10^{-6}	3.73×10^{-6}	1.0	1
1	3.35×10^{-4}	3.35×10^{-4}	0.980	0.980
2	0.0111	0.111	0.923	0.923
3	0.135	0.135	0.835	0.835
4	0.607	0.607	0.726	0.726
5	1.0	1	0.607	0.607
6	0.607	0.607	0.487	0.487
7	0.135	0.135	0.375	0.375
8	0.111	0.111	0.278	0.278
9	3.35×10^{-4}	3.35×10^{-4}	0.198	0.198
10	3.72×10^{-6}	3.73×10^{-6}	0.135	0.135

Looking at these results, aside from a floating point error visible in the 0 value for test 1, we see identical results between the computed values and manually calculated values.

Overall, we can see that all the membership functions are working as intended and can safely be used in the fuzzy logic system.

2.1.2 Membership Interpretation

To test the membership interpretation function, we will evaluate, both using the computer and by hand, some values. We will use the domain from earlier (0-10) with intervals of 1, as well as the test 1 trapezoidal function from earlier ($a = 1$, $b = 2$, $c = 5$, $d = 9$). We will interpret membership

at values 0.6, 3.4, and 6.7. The function returns results of 0.6 → 0.0, 3.4 → 1.0, and 6.7 → 0.575. Calculating by hand yields 0.6 → 0, 3.4 → 1, and 6.7 → 0.575. As we have the same results for both the computed values and manually calculated values, it should be safe to use this membership interpretation function implementation in the fuzzy logic system.

2.1.3 Implication and Aggregation Operators

The implication function has three types and we can verify each one individually. We will run the function as well as calculate the answer by hand for all three types: Zadeh, correlation-min, and correlation-product. We will run this function on several different sets to test behavior.

Implication Operators						
Test Pair	Zadeh Computed	Zadeh Manual	Corr-min Computed	Corr-min Manual	Corr-prod Computed	Corr-prod Manual
(0,0)	1.0	1	0	0	0	0
(1,2)	1.0	1	1	1	2	2
(7,.5)	-5.5	-5.5	0.5	0.5	3.5	3.5
(-2,0)	1.0	1	-2	-2	0	0
(-4,-4)	1.0	1	-4	-4	16	16
(-4,-6)	-1.0	-1	-6	-6	24	24

Now that we have shown that the implication operators perform as expected, we can run trials to test the aggregation operators. I have set up three membership functions. Two serve as inputs: function 1 which is the triangular function from test 1 earlier with $a = 2$, $b = 5$, and $c = 6$, and function 2 which is the trapezoidal function from test 1 earlier with $a = 1$, $b = 2$, $c = 5$, and $d = 9$. We also have one output membership function defined by the Gaussian function from test 1 earlier with $c = 5$ and $\sigma = 1$. To these, we apply two rules. IF function 1 THEN output function, and IF function 2 THEN output function. We run this test sample for input values of (5.6, 3.4) and (2.7, 9.2), using the maximum aggregation operator and the sum aggregation operator. The results are summarized below, rounded to 3 decimal places.

For maximum aggregation over (5.6, 3.4):

Computed Value → [0.600, 0.600, 0.611, 0.735, 1, 1, 1, 0.735, 0.611, 0.600, 0.600]

Manually Calculated Value → [0.600, 0.600, 0.611, 0.735, 1, 1, 1, 0.735, 0.611, 0.600, 0.600]

For maximum aggregation over (2.7, 9.2):

Computed Value → [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Manually Calculated Value → [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

For sum aggregation over (5.6, 3.4):

Computed Value → [0.600, 0.601, 0.622, 0.871, 1.607, 2.1607, 0.871, 0.622, 0.601, 0.600]

Manually Calculated Value → [0.600, 0.601, 0.622, 0.871, 1.607, 2.1607, 0.871, 0.622, 0.601, 0.600]

For sum aggregation over (2.7, 9.2):

Computed Value → [1.767, 1.767, 1.778, 1.902, 2, 2, 2, 1.902, 1.778, 1.767, 1.767]

Manually Calculated Value → [1.767, 1.767, 1.778, 1.902, 2, 2, 2, 1.902, 1.778, 1.767, 1.767]

We can see that the aggregation functions also perform exactly as expected. We can safely use these in our fuzzy logic system.

2.1.4 Defuzzification

Finally, we must confirm if the implementation of centroid defuzzification performs as expected. In order to do this, I will compare the computed value and the manually calculated value of centroid defuzzification from the maximum aggregation value with inputs (5.6, 3.4) and the sum aggregation with the same inputs.

When we run centroid defuzzification on the maximum aggregation data, we get a computed value of 5.0 compared to a manually calculated value of 5. Similarly, when we run centroid defuzzification on the sum aggregation data, we get a computed value of 5 and a manually calculated value of 5. These results are due to the symmetric nature of the Gaussian function, but reflect a working centroid defuzzification operator.

With all the fuzzy logic functions shown to be working, we can now move on to applications.

2.2 Iris Classification

Now we move to actually using the fuzzy logic system to solve problems. As earlier described, we will use the fuzzy logic system to classify 150 iris individuals into three species, I. setosa, I. versicolor, and I. virginica. To do this, we are given values for sepal width, sepal length, petal width, and petal length.

2.2.1 Individuals

For this experiment, I ran the classifier for all the individuals with a consistent rule set. The rule set is defined as: IF petal length and petal width are small THEN setosa, IF petal length is medium THEN versicolor, and IF petal length is big THEN virginica. I did this for each combination of implication and aggregation operators using for implication, Lukasiewicz, correlation-min, and correlation product, then for aggregation, sum and maximum. Before looking at the results in total, let's examine a few examples of classification to get a better understanding of the behavior.

We will first take a look at individual #4 which is known to belong to I. setosa. It has a sepal length of 4.6, sepal width of 3.1, petal length of 1.5, and petal width of 0.2. The rule set is able to easily determine that this individual is I. setosa as shown by the following figure which shows the membership of this individual in the species output set (orange) alongside the centroid value (black line). We end with a defuzzified species value of 0.633, which places this individual well in the middle of the range for I. setosa (0, 1).

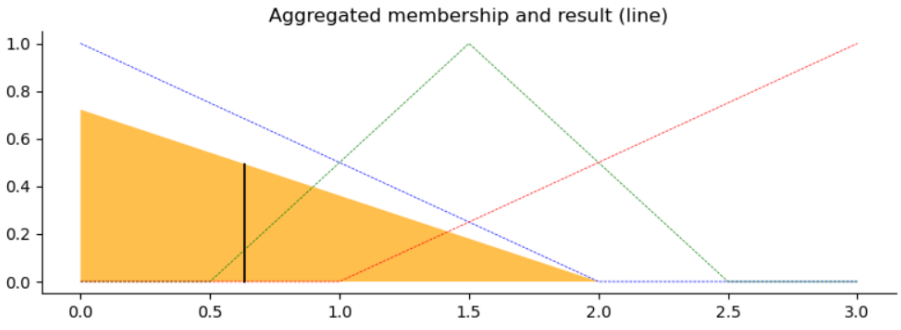

Figure 9: Classification of Individual #4

We now take a look at individual #88 which is known to belong to I. versicolor. This individual has a sepal length of 6.3, a sepal width of 2.3, a petal length of 4.4, and a petal width of 1.3. The rule set struggles slightly more with this individual, classifying it mostly as I. versicolor, but also partially as I. virginica. This is a reoccurring issue due to I. versicolor and I. virginica having very similar values for the properties we have data for. Despite this, the individual gets a defuzzified species value of 1.612, which places it in the I. versicolor range (1-2).

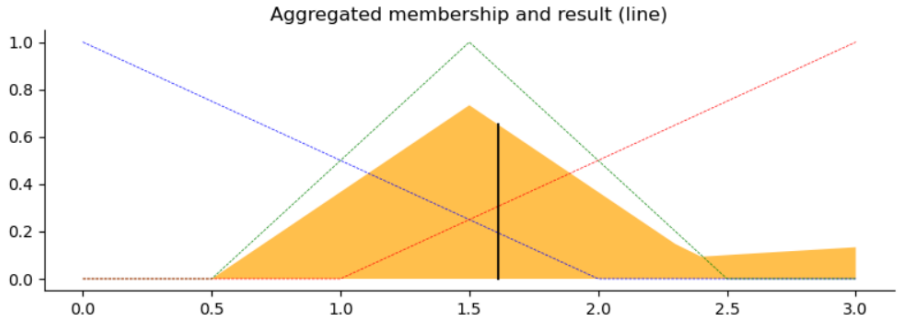

Figure 10: Classification of Individual #88

Next, we look at individual #124 which is known to belong to I. virginica. This individual has a sepal length of 6.3, a sepal width of 2.7, a petal length of 4.9, and a petal width of 1.8. Here we actually see the rule set fail. Almost all the failures of this rule set are incorrectly identifying individuals belong to I. virginica as belonging to I. versicolor. Here we get a defuzzified species value of 1.889, which although close to the I. virginica range of (1,2), leaves this individual incorrectly identified as I. versicolor.

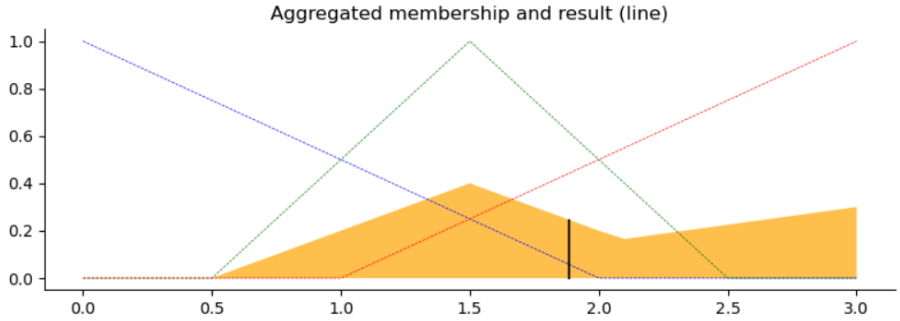

Figure 11: Classification of Individual #124

2.2.2 Implication and Aggregation Operators

Now that we have seen some examples of individual classification, we can move on to looking at the data set more broadly. The experiment ran these rules over all individuals with each combination of implication and aggregation operators. The classification accuracy for each test is shown below.

Classification Accuracy			
Aggregation	Zadeh/Lukasiewicz	Correlation-min	Correlation-product
Maximum	0.333	0.893	0.933
Sum	0.333	0.88	0.933

We can see here that the biggest difference in performance is between the Lukasiewicz implication operator and the other two. This makes sense as correlation-min and correlation-product tend to perform similarly as they are similar formats versus the Lukasiewicz operator which functions quite differently. We also see very minor improvements between the maximum and sum aggregation operators which is somewhat to be expected. Overall we get the best performance using correlation-product implication as well as maximum aggregation.

2.2.3 Improving Classification

Looking at these results, what is preventing us from getting a higher classification percentage? With the best performance from the experiment we have only 10 individuals that are not being correctly classified out of the data set of 150 individuals. 9 of these are I. virginica being incorrectly classified as I. versicolor. The other one is I. versicolor being incorrectly classified as I. virginica. This underlying cause of this is that I. virginica and I. versicolor are very similar, while I. setosa remains quite different from those two. This structure of data is what makes the Iris data set so useful as a test database as we can see both what happens with widely separated clusters as well as those that are very close together.

If we want to improve the classification, it would probably be required to adjust the membership functions and likely add more functions for each input. Alternatively we could try changing the cutoffs in the output to see if maybe this would help with species identification. The more obvious method, changing the rules, proved unhelpful during my attempts but it is possible there are better rules out there that I did not identify.

However, there is the concern of over-fitting if we attempt to make the rules much more specific. We can continue adding membership functions and rules to improve the classification rate, but at that point we are optimizing around just 10 individuals, who may not be well representative of the population. For a database of this size, 150 individuals, it may be best to instead rely on general trends than to try to get extremely specific for each individual.

2.3 Inverted Pendulum Problem

We now move to look at the implementation of the inverted pendulum problem using the fuzzy logic system. The membership functions and rules that I described in the earlier section are what I found to work the best. We can explore the behaviors created by adjusting each parameter to see why these values work as they do.

2.3.1 Input Membership Functions

From my testing, the angle membership functions have a very drastic effect on performance. I eventually settled on a Gaussian function, centered at the edge of the domain $c = \pm 0.22$, and with a standard deviation of $\sigma = .05$. This creates a cart that balances the pole nearly perfectly, making it through the 500 time steps every time with the only downside being some slight drift. If we try

changing σ this does not remain. If we move up to $\sigma = 0.1$ the cart almost immediately gets into a loop of over-correcting back and forth until the simulation ends, surviving only 50 time-steps. If we move down to $\sigma = 0.01$, we instead don't correct enough, the pole starts to lean and the cart slowly chases it off-screen. Neither of these will do, thus we leave the standard deviation at $\sigma = 0.05$.

The angular velocity membership functions are trapezoidal functions with values of (-100, -100, -50, 0) and (0, 50, 100, 100). We can investigate the behavior with changing the middle 50 value. If we move this value to ± 100, making the membership functions (-100, -100, -100, 0) and (0, 100, 100, 100), the pole does stay balanced, but is very jittery and occasionally makes large sudden movements across the screen. If we move the other way and change this value to 25, making membership functions of (-100, -100, -25, 0) and (0, 25, 100, 100), we do keep the pole balanced for all 500 time-steps, but the cart struggles a bit more and has to move around a lot. Overall, it seems that the angular velocity membership makes much less of a difference then that of the angle but it does still have some impact on the performance if we change it a substantial amount.

2.3.2 Rules

We can also try investigating what happens when we change the rules. We currently have a rule for each left and right for angle and for angular velocity. What happens when we remove some of these? When we remove the angle rules and run just off of angular velocity, we notice that the pole starts tipping, the car can never catch up, and we end up running off screen within 200 time-steps. When we instead remove the angular rules and use just the angle, the car pretty much immediately fails. It tries to over-adjust and falls out of range by around 40 time steps. It seems that both sets of rules are important for everything to work.

3 Conclusion

Now that we have looked at the application of the fuzzy logic system to both problems, we can evaluate performance and reflect on what worked and what did not.

3.1 Conclusions - Iris Classification

The Iris classification worked pretty well, overall. The problem set was a natural fit for fuzzy logic. These flowers would likely have been identified and classified originally using natural language rules which the fuzzy logic system can emulate. Using rudimentary membership functions without much connection to the data set alongside rules that attempt to match trends, I observed 140 out of 150 individuals being identified correctly. Especially given how similar I. versicolor and I. virginica individuals can be, this is overall a good performance.

If we want to try to improve this, there are a few possible methods. We can change membership functions and rules to try to optimize, but we run the risk of over-fitting and making a fuzzy logic system that does not work for the general population and instead just for this data-set. Alternatively, we could adjust the output rounding in the chance that changing the cutoffs between I. setosa and I. virginica can fix things.

3.2 Conclusions - Inverted Pendulum

The inverted pendulum is a very different problem to apply the fuzzy logic system to. Instead of a consistent data set that we go through and process once, the inverted pendulum is initialized with randomized starting values and needs to be updated at many time steps. However, it once again makes a natural fit for the fuzzy logic system as balancing a pole is very easy to describe in terms of linguistic rules.

The fuzzy logic system ended up being quite good at balancing the pole, I would be very interested to see how the system would perform if the simulator allowed the pole a full range of motion and we had to worry about getting the pole back to balanced from the down position. This would likely require many more rules than the four that this required.

3.3 Conclusions - Overall

Implementing a fuzzy logic system was a very insightful exercise. Through breaking down each function step-by-step, I feel that I got a much better understanding of everything involved in this process. Getting to use this to solve two problems was also a good way to understand the application of fuzzy logic to the real world, and how being able to do things with rules matching human language can prove useful in a variety of applications.

I think that my fuzzy logic system implementation ended up working pretty well and is fairly easy to follow, even if it did take me a long time to figure out how to make it work. I attempted to make it as modular and adaptable as possible so it is easy to connect into different applications but this does come at a cost of making a lot of the work be specifically written for applications instead of fully premade functions that can do large amounts of work on their own.

A Attributions

All graphs are my own. Some code ideas were taken from course Jupyter pages, all other code is my own. The Gymnastics cart pole library was used for simulation but the control of the simulation and the code to do so is my own. All writing is my own work: no LLM or other AI tools were used in the writing of this paper.