



You have 2 free stories left this month. [Sign up](#) and get an extra one for free.

Fuzzy Inference System implementation in Python

"A story always sounds clear enough at a distance, but the nearer you get to the scene of events the vaguer it becomes."— George Orwell



Carmel Gafa

[Follow](#)

May 13 · 9 min read ★



photo by Carmel Gafa

Introduction

In a [previous article](#), we discussed the basics of fuzzy sets and fuzzy inferencing.

The report also illustrated the construction of a possible control application using a fuzzy inferencing method. In this article, we will build a multi-input/multi-output fuzzy inference system using the Python programming language. It is assumed that the reader has a clear understanding of fuzzy inferencing and has read the article mentioned previously.

All the code listed in this article is available on [Github](#).

System Architecture

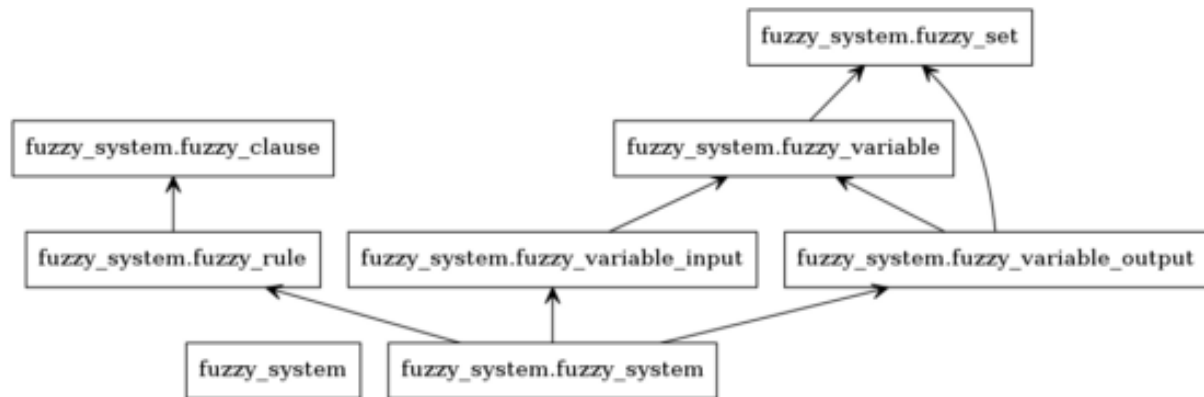
The diagram below illustrates the structure of the application. The design is based on several considerations on Fuzzy Inference Systems, some being:

- A Fuzzy Inference System will require input and output variables and a collection of fuzzy rules.
- Both input and output variables will contain a collection of fuzzy sets if the Fuzzy Inference System is of Mamdani type.
- Input and output variables are very similar, but they are used differently by fuzzy rules. During execution, input variables use the input values to the system to fuzzify their sets, that is they determine the degree of belonging of that input value to all of the fuzzy sets of the variable. Each rule contributes to some extent to the output variables; the totality of this contribution will determine the output of the system.
- Fuzzy rules have the structure of the form;

if {antecedent clauses} **then** {consequent clauses}

Therefore a rule will contain several clauses of antecedent type and some clauses of consequent type. Clauses will be of the form:

{variable name} **is** {set name}



System Diagram

We will discuss some implementation details of the classes developed for this system in the following sections:

FuzzySet class

A *FuzzySet* requires the following parameters so that it can be initiated:

- **name** — the name of the set
- **minimum value** — the minimum value of the set
- **maximum value** — the maximum value of the set
- **resolution** — the number of steps between the minimum and maximum value

It is, therefore, possible to represent a fuzzy set by using two **numpy** arrays; one that will hold the domain values and one that will hold the degree-of-membership values. Initially, all degree-of-membership values will be all set to zero. It can be argued that if the minimum and maximum values are available together with the resolution of the set, the domain numpy array is not required as the respective values can be calculated. While this is perfectly true, a domain array was preferred in this example project so that the code is more readable and simple.

```
1 def create_triangular(cls, name, domain_min, domain_max, res, a, b, c):
2     t1fs = cls(name, domain_min, domain_max, res)
3
4     a = t1fs._adjust_domain_val(a)
5     b = t1fs._adjust_domain_val(b)
6     c = t1fs._adjust_domain_val(c)
7
8     t1fs._dom = np.round(np.maximum(np.minimum((t1fs._domain-a)/(b-a), (c-t1fs._
```

create_triangular_set.py hosted with ❤ by [GitHub](#)

[view raw](#)

FuzzySet init

In the context of a fuzzy variable, all the sets will have the same minimum, maximum and resolution values.

As we are dealing with a discretized domain, it will be necessary to adjust any value used to set or retrieve the degree-of-membership to the closest value in the domain array.

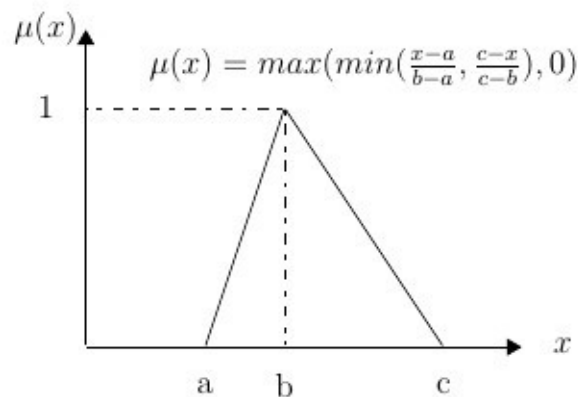
```
1 def _adjust_domain_val(self, x_val):
2     return self._domain[np.abs(self._domain-x_val).argmin()]
```

adjust_domain.py hosted with ❤ by [GitHub](#)

[view raw](#)

FuzzySet domain value adjustment

The class contains methods whereby a set of a given shape can be constructed given a corresponding number of parameters. In the case of a triangular set, for example, three parameters are provided, two that define the extents of the sets and one for the apex. It is possible to construct a triangular set by using these three parameters as can be seen in the figure below.



Triangular Fuzzy Set equation

Since the sets are based on numpy arrays, the equation above can be translated directly to code, as can be seen below. Sets having different shapes can be constructed using a similar method.

```

1  def create_triangular(cls, name, domain_min, domain_max, res, a, b, c):
2      t1fs = cls(name, domain_min, domain_max, res)
3
4      a = t1fs._adjust_domain_val(a)
5      b = t1fs._adjust_domain_val(b)
6      c = t1fs._adjust_domain_val(c)
7
8      t1fs._dom = np.round(np.maximum(np.minimum((t1fs._domain-a)/(b-a), (c-t1fs._

```

`create_triangular_set.py` hosted with ♥ by [GitHub](#)

[view raw](#)

Triangular set creation

The **FuzzySet** class also contains union, intersection and negation operators that are necessary so that inferencing can take place. All operator methods return a new fuzzy set with the result of the operation that took place.

```
1  def union(self, f_set):
2
3      result = FuzzySet(f'({self._name}) union ({f_set._name})', sel
4      result._dom = np.maximum(self._dom, f_set._dom)
5
6      return result
```

fuzzy_union.py hosted with ❤ by GitHub

[view raw](#)

Union of two FuzzySet objects

Finally, we implemented the ability to obtain a crisp result from a fuzzy set using the centre-of-gravity method that is referred to in some detail in the previous article. It is important to mention that there is a large number of defuzzification methods available in the literature. Still, as the centre-of-gravity method is overwhelmingly popular, it is used in this implementation.

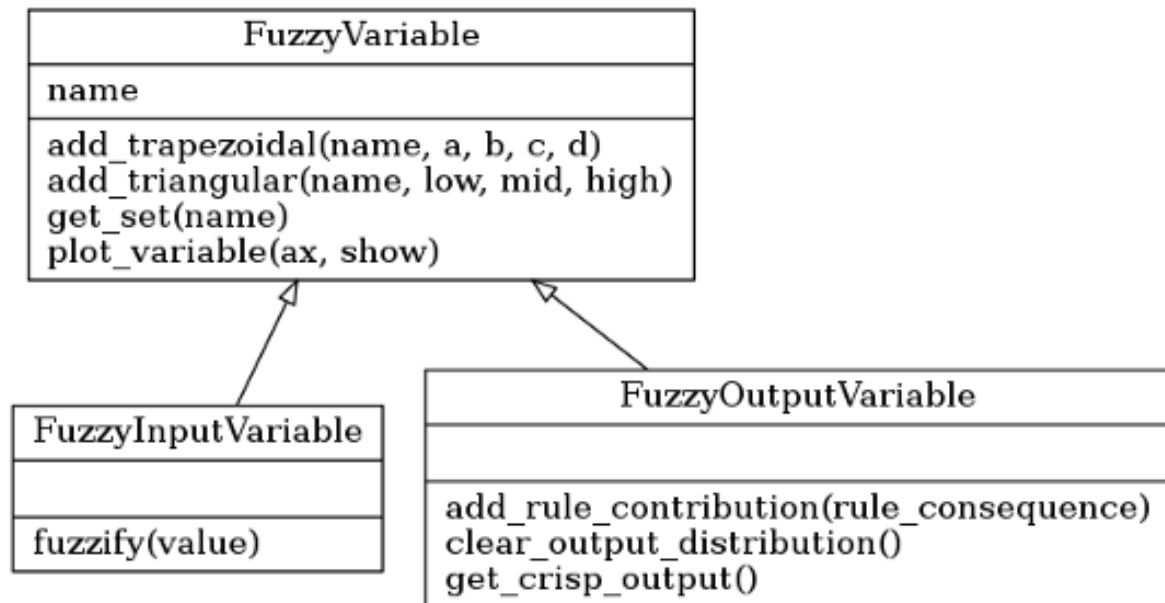
```
1  def cog_defuzzify(self):
2
3      num = np.sum(np.multiply(self._dom, self._domain))
4      den = np.sum(self._dom)
5
6      return num/den
```

cog_defuzzify.py hosted with ❤ by GitHub

[view raw](#)

Centre of gravity defuzzification

Fuzzy Variable classes



FuzzyVariable Classes

As discussed previously, variables can be of input or output in type, with the difference affecting the fuzzy inference calculation. A **FuzzyVariable** is a collection of sets that are held in a python dictionary having the set name as the key. Methods are available to add **FuzzySets** to the variable, where such sets will take the variable's limits and resolution.

For input variables, fuzzification is carried out by retrieving the degree-of-membership of all the sets in the variable for a given domain value. The degree-of-membership is stored in the set as it will be required by the rules when they are evaluated.

```

1  def fuzzify(self, val):
2
3      # get dom for each set and store it -
4      # it will be required for each rule
5      for set_name, f_set in self._sets.items():
6          f_set.last_dom_value = f_set[val]
  
```

fuzzify.py hosted with ❤ by GitHub

[view raw](#)

Fuzzification of an input variable

Output variables will ultimately produce the result of a fuzzy inference iteration.

This means that for Mamdani-type systems, as we are building here, output variables will hold the union of the fuzzy contributions from all the rules, and will subsequently defuzzify this result to obtain a crisp value that can be used in real-life applications.

Therefore, output variables will require an additional **FuzzySet** attribute that will hold the output distribution for that variable, where the contribution that was resulting from each rule and added using the set union operator. The defuzzification result can then be obtained by calling the centre-of-gravity method for output distribution set.

```
1 class FuzzyOutputVariable(FuzzyVariable):
2
3     def __init__(self, name, min_val, max_val, res):
4         super().__init__(name, min_val, max_val, res)
5         self._output_distribution = FuzzySet(name, min_val, max_val, res)
6
7     def add_rule_contribution(self, rule_consequence):
8         self._output_distribution = self._output_distribution.union(rule_cons
9
10    def get_crisp_output(self):
11        return self._output_distribution.cog_defuzzify()
```

Fuzzy output variable output distribution and defuzzification methods

Fuzzy Rules classes

The **FuzzyClause** class requires two attributes; a fuzzy variable and a fuzzy set so that the statement

variable **is** set

can be created. Clauses are used to implement statements that can be chained together to form the antecedent and consequent parts of the rule.

When used as an antecedent clause, the **FuzzyClause** returns the last degree-of-membership value of the set, that is calculated during the fuzzification stage as

we have seen previously.

The rule will combine the degree-of-membership values from the various antecedent clauses using the **min** operator, obtaining the *rule activation* that is then used in conjunction with the consequent clauses to obtain the contribution of the rule to the output variables. This operation is a two-step process:

- The activation value is combined with the consequent **FuzzySet** using the **min** operator, that will act as a threshold to the degree-of-membership values of the **FuzzySet**.
- The resultant **FuzzySet** is combined with the **FuzzySets** obtained from the other rules using the **union** operator, obtaining the output distribution for that variable.

```
1 # execution methods for a FuzzyClause
2 # that contains a FuzzyVariable; _variable
3 # and a FuzzySet; _set
4
5 def evaluate_antecedent(self):
6     return self._set.last_dom_value
7
8 def evaluate_consequent(self, activation):
9     self._variable.add_rule_contribution(self._set.min_scalar(activation))
```

Execution methods of a fuzzy clause as either antecedent or consequent

The **FuzzyRule** class will, therefore, require two attributes:

- a list containing the antecedent clauses and
- a list containing the consequent clauses

During the execution of the **FuzzyRule**, the procedure explained above is carried out. The **FuzzyRule** coordinates all the tasks by utilizing all the various **FuzzyClauses** as appropriate.

```
1 def evaluate(self):
2     # rule activation initialize to 1 as min operator will be performed
3     rule_activation = 1
4     # execute all antecedent clauses, keeping the minimum of the returned
5     for ante_clause in self._antecedent:
6         rule_activation = min(ante_clause.evaluate_antecedent(), rule_activation)
7
8     # execute consequent clauses, each output variable will update its output
9     for consequent_clause in self._consequent:
10        consequent_clause.evaluate_consequent(rule_activation)
```

rule execution by hosted with ❤ by GitHub

[view raw](#)

\Rule execution

Fuzzy System Class — Bringing it all together.

At the topmost level of this architecture, we have the **FuzzySystem** that coordinates all activities between the FuzzyVariables and FuzzyRules. Hence the system contains the input and output variables, that are stored in python dictionaries using variable-names as keys and a list of the rules.

One of the challenges presented at this stage is the method that the end-user will use to add rules, that should ideally abstract the implementation detail of the **FuzzyClause** classes. The method that was implemented consists of providing two python dictionaries that will contain the antecedent and consequent clauses of the rule in the following format;

variable name : set name

A more user-friendly method is to provide the rule as a string and then parse that string to create the rule, but this seemed an unnecessary overhead for a demonstration application.

```
1         def add_rule(self, antecedent_clauses, consequent_clauses):
2             '''
3             adds a new rule to the system.
4             TODO: add checks
5             Arguments:
6             -----
7             antecedent_clauses -- dict, having the form {variable_name:set_name}
8             consequent_clauses -- dict, having the form {variable_name:set_name}
9             '''
10            # create a new rule
11            # new_rule = FuzzyRule(antecedent_clauses, consequent_clauses)
12            new_rule = FuzzyRule()
13
14            for var_name, set_name in antecedent_clauses.items():
15                # get variable by name
16                var = self.get_input_variable(var_name)
17                # get set by name
18                f_set = var.get_set(set_name)
19                # add clause
20                new_rule.add_antecedent_clause(var, f_set)
21
22            for var_name, set_name in consequent_clauses.items():
23                var = self.get_output_variable(var_name)
24                f_set = var.get_set(set_name)
25                new_rule.add_consequent_clause(var, f_set)
26
27            # add the new rule
```

Addition of a new rule to the FuzzySystem

The execution of the inference process can be achieved with a few lines of code given this structure, where the following steps are carried out;

1. The output distribution sets of all the output variables are cleared.
2. The input values to the system are passed to the corresponding input variables so that each set in the variable can determine its degree-of-membership for that input value.
3. Execution of the Fuzzy Rules takes place, meaning that the output

distribution sets of all the output variables will now contain the union of the contributions from each rule.

```
1  # clear the fuzzy consequences as we are evaluating a new set of inputs.
2  # can be optimized by comparing if the inputs have changes from the previous
3  # iteration.
4  self._clear_output_distributions()
5
6  # Fuzzify the inputs. The degree of membership will be stored in
7  # each set
8  for input_name, input_value in input_values.items():
9      self._input_variables[input_name].fuzzify(input_value)
10
11 # evaluate rules
12 for rule in self._rules:
13     rule.evaluate()
14
15 # finally, defuzzify all output distributions to get the crisp outputs
16 output = {}
17 for output_var_name, output_var in self._output_variables.items():
18     output[output_var_name] = output_var.get_crisp_output()
19
```

As a final note, the Fuzzy Inferencing System implemented here contains additional functions to plot fuzzy sets and variables and to obtain information about an inference step execution.

Library Use Example

In this section, we will discuss the use of the fuzzy inference system. In particular, we will implement the fan speed case study that was designed in the previous [article](#) in this series.

A fuzzy system begins with the consideration of the input and output variables, and the design of the fuzzy sets to explain that variable.

The variables will require a lower and upper limit and, as we will be dealing with discrete fuzzy sets, the resolution of the system. Therefore a variable definition will look as follows

```
temp = FuzzyInputVariable('Temperature', 10, 40, 100)
```

where the variable '**Temperature**' ranges between 10 and 40 degrees and is discretized in 100 bins.

The fuzzy sets define for the variable will require different parameters depending on their shape. In the case of triangular sets, for example, three parameters are needed, two for the lower and upper extremes having a degree of membership of 0 and one for the apex which has a degree-of-membership of 1. A triangular set definition for variable '**Temperature**' can, therefore, look as follows;

```
temp.add_triangular('Cold', 10, 10, 25)
```

where the set called '**Cold**' has extremes at 10 and 25 and apex at 10 degrees. In our system, we considered two input variables, '**Temperature**' and '**Humidity**' and a single output variable '**Speed**'. Each variable is described by three fuzzy sets. The definition of the output variable '**Speed**' looks as follows:

```
motor_speed = FuzzyOutputVariable('Speed', 0, 100, 100)
motor_speed.add_triangular('Slow', 0, 0, 50)
motor_speed.add_triangular('Moderate', 10, 50, 90)
motor_speed.add_triangular('Fast', 50, 100, 100)
```

As we have seen before, the fuzzy system is the entity that will contain these variables and fuzzy rules. Hence the variables will have to be added to a system as follows:

```
system = FuzzySystem()
system.add_input_variable(temp)
system.add_input_variable(humidity)
system.add_output_variable(motor_speed)
```

Fuzzy Rules

A fuzzy system executes fuzzy rules to operate of the form

If x_1 is S and x_2 is M then y is S


where the If part of the rule contains several antecedent clauses and the then

Sign up for The Daily Pick

By Towards Data Science

Hands-on real-world examples, research, tutorials, and cutting-edge techniques delivered Monday to Thursday. Make learning your daily ritual. [Take a look](#)

Your email

 Get this newsletter

By signing up, you will create a Medium account if you don't already have one. Review our [Privacy Policy](#) for more information about our privacy practices.

entries for the clause sets. Hence the above rule can be implemented as follows:

```
system.add_rule(
    { 'Temperature': 'Cold',
      'Humidity': 'Wet' },
    'Cold and Wet')
```

[About](#) [Help](#) [Legal](#)



Execution of the system involves inputting values for all the input variables and getting the values for the output values in return. Again this is achieved through the use of dictionaries that use the name of the variables as keys.

```
output = system.evaluate_output({  
    'Temperature':18,  
    'Humidity':60  })
```

The system will return a dictionary containing the name of the output variables as keys and the defuzzified result as values.

Conclusion

In this article, we have looked at the practical implementation of a fuzzy inference system. Whilst the library presented here will require some further work so that it can be used in real projects, including validation and exception handling, it can serve as a basis for projects that require Fuzzy Inferencing. It is also recommended to look at some open-source projects that are available, in particular [skfuzzy](#), a fuzzy logic toolbox for SciPy.

In the next article, we will examine ways whereby a fuzzy system can be created from a dataset so that that fuzzy logic can be used in machine learning scenarios. Similarly to this introduction to Fuzzy Logic concepts, a practical article will follow.