***Practical 10***

**Aim:** *Implement Fuzzy Inference System.*

***Theory:***

*Fuzzy Inference System is the key unit of a fuzzy logic system having decision making as its primary work. It uses the “IF…THEN” rules along with connectors “OR” or “AND” for drawing essential decision rules.*

***Characteristics of Fuzzy Inference System***

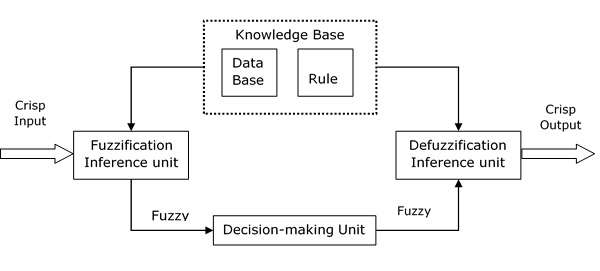
***Following are some characteristics of FIS −***

* *The output from FIS is always a fuzzy set irrespective of its input which can be fuzzy or crisp.*
* *It is necessary to have fuzzy output when it is used as a controller.*
* *A defuzzification unit would be there with FIS to convert fuzzy variables into crisp variables.*

***Functional Blocks of FIS***

*The following five functional blocks will help you understand the construction of FIS −*

* *Rule Base − It contains fuzzy IF-THEN rules.*
* *Database − It defines the membership functions of fuzzy sets used in fuzzy rules.*
* *Decision-making Unit − It performs operation on rules.*
* *Fuzzification Interface Unit − It converts the crisp quantities into fuzzy quantities.*
* *Defuzzification Interface Unit − It converts the fuzzy quantities into crisp quantities. Following is a block diagram of fuzzy interference system.*



*Working of FIS*

*The working of the FIS consists of the following steps −*

* *A fuzzification unit supports the application of numerous fuzzification methods, and converts the crisp input into fuzzy input.*
* *A knowledge base - collection of rule base and database is formed upon the conversion of crisp input into fuzzy input.*
* *The defuzzification unit fuzzy input is finally converted into crisp output.*

*Methods of FIS*

*Let us now discuss the different methods of FIS. Following are the two important methods of FIS, having different consequent of fuzzy rules −*

* *Mamdani Fuzzy Inference System*
* *Takagi-Sugeno Fuzzy Model (TS Method)*

*Mamdani Fuzzy Inference System*

*This system was proposed in 1975 by Ebhasim Mamdani. Basically, it was anticipated to control a steam engine and boiler combination by synthesizing a set of fuzzy rules obtained from people working on the system.*

*Steps for Computing the Output*

*Following steps need to be followed to compute the output from this FIS −*

***Step 1****− Set of fuzzy rules need to be determined in this step.*

***Step 2****− In this step, by using input membership function, the input would be made fuzzy.*

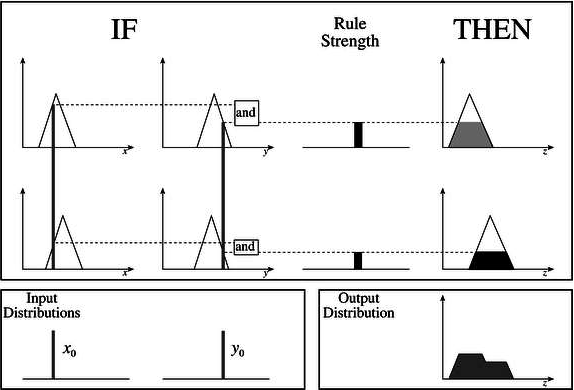
***Step 3****− Now establish the rule strength by combining the fuzzified inputs according to fuzzy rules.*

***Step 4****− In this step, determine the consequent of rule by combining the rule strength and the output membership function.*

***Step 5****− For getting output distribution combine all the consequents.*

***Step 6****− Finally, a defuzzified output distribution is obtained.*

*Following is a block diagram of Mamdani Fuzzy Interface System.*



***Code:***

fuzzy\_clause.py

'''

Fuzzy Clause class. Used in Fuzzy rule

'''

class FuzzyClause():

    '''

    A fuzzy clause of the type 'variable is set'

    used in fuzzy IF ... THEN ... rules

    clauses can be antecedent (if part) or consequent

    (then part)

    '''

    def \_\_init\_\_(self, variable, f\_set, degree=1):

        '''

        initialization of the fuzzy clause

        Arguments:

        ----------

        variable -- the clause variable in 'variable is set'

        set -- the clause set in 'variable is set'

        '''

        if f\_set is None:

            raise Exception('set none')

        if f\_set.name == '':

            raise Exception(str(f\_set), 'no set name')

        self.\_variable = variable

        self.\_set = f\_set

    def \_\_str\_\_(self):

        '''

        string representation of the clause.

        Returns:

        --------

        str: str, string representation of the clause in the form

                    A is x

        '''

        return f'{self.\_variable.name} is {self.\_set.name}'

    @property

    def variable\_name(self):

        '''

        returns the name of the clause variable

        Returns:

        --------

        variable\_name: str, name of variable

        '''

        return self.\_variable.name

    @property

    def set\_name(self):

        '''

        returns the name of the clause variable

        Returns:

        --------

        variable\_name: str, name of variable

        '''

        return self.\_set.name

    def evaluate\_antecedent(self):

        '''

        Used when set is antecedent.

        returns the set degree of membership.

        Returns:

        --------

        dom -- number, the set degree of membership given a value for

                that variable. This value is determined at an earlier stage

                and stored in the set

        '''

        return self.\_set.last\_dom\_value

    def evaluate\_consequent(self, dom):

        '''

        Used when clause is consequent.

        Arguments:

        -----------

        dom -- number, scalar value from the antecedent clauses

        Returns:

        --------

        set -- Type1FuzzySet, a set resulting from min operation with

                the scalar value

        '''

        self.\_variable.add\_rule\_contribution(self.\_set.min\_scalar(dom))

fuzzy\_rule.py

from fuzzy\_logic.fuzzy\_clause import FuzzyClause

class FuzzyRule():

    '''

    A fuzzy rule of the type

    IF [antecedent clauses] THEN [consequent clauses]

    '''

    def \_\_init\_\_(self):

        '''

        initializes the rule. Two data structures are necessary:

            Antecedent clauses list

            consequent clauses list

        '''

        self.\_antecedent = []

        self.\_consequent = []

    def \_\_str\_\_(self):

        '''

        string representation of the rule.

        Returns:

        --------

        str: str, string representation of the rule in the form

                    IF [antecedent clauses] THEN [consequent clauses]

        '''

        ante = ' and '.join(map(str, self.\_antecedent))

        cons = ' and '.join(map(str, self.\_consequent))

        return f'If {ante} then {cons}'

    def add\_antecedent\_clause(self, var, f\_set):

        '''

        adds an antecedent clause to the rule

        Arguments:

        -----------

            clause -- FuzzyClause, the antecedent clause

        '''

        self.\_antecedent.append(FuzzyClause(var, f\_set))

    def add\_consequent\_clause(self, var, f\_set):

        '''

        adds an consequent clause to the rule

        Arguments:

        -----------

            clause -- FuzzyClause, the consequent clause

        '''

        self.\_consequent.append(FuzzyClause(var, f\_set))

    def evaluate(self):

        '''

        evaluation of the rule.

        the antecedent clauses are executed and the minimum degree of

        membership is retained.

        This is used in teh consequent clauses to min with the consequent

        set

        The values are returned in a dict of the form {variable\_name: scalar min set, ...}

        Returns:

        --------

        rule\_consequence -- dict, the resulting sets in the form

                            {variable\_name: scalar min set, ...}

        '''

        # rule dom initialize to 1 as min operator will be performed

        rule\_strength = 1

        # execute all antecedent clauses, keeping the minimum of the

        # returned doms to determine the rule strength

        for ante\_clause in self.\_antecedent:

            rule\_strength = min(ante\_clause.evaluate\_antecedent(), rule\_strength)

        # execute consequent clauses, each output variable will update its output\_distribution set

        for consequent\_clause in self.\_consequent:

            consequent\_clause.evaluate\_consequent(rule\_strength)

    def evaluate\_info(self):

        '''

        evaluation of the rule.

        the antecedent clauses are executed and the minimum degree of

        membership is retained.

        This is used in teh consequent clauses to min with the consequent

        set

        The values are returned in a dict of the form {variable\_name: scalar min set, ...}

        Returns:

        --------

        rule\_consequence -- dict, the resulting sets in the form

                            {variable\_name: scalar min set, ...}

        '''

        # rule dom initialize to 1 as min operator will be performed

        rule\_strength = 1

        # execute all antecedent clauses, keeping the minimum of the

        # returned doms to determine the rule strength

        for ante\_clause in self.\_antecedent:

            rule\_strength = min(ante\_clause.evaluate\_antecedent(), rule\_strength)

        # execute consequent clauses, each output variable will update its output\_distribution set

        for consequent\_clause in self.\_consequent:

            consequent\_clause.evaluate\_consequent(rule\_strength)

        return f'{rule\_strength} : {self}'

fuzzy\_set.py

import numpy as np

import copy

import matplotlib.pyplot as plt

class FuzzySet:

    \_precision: int = 3

    def \_\_init\_\_(self, name, domain\_min, domain\_max, res):

        self.\_domain\_min = domain\_min

        self.\_domain\_max = domain\_max

        self.\_res = res

        self.\_domain = np.linspace(domain\_min, domain\_max, res)

        self.\_dom = np.zeros(self.\_domain.shape)

        self.\_name = name

        self.\_last\_dom\_value = 0

    def \_\_getitem\_\_(self, x\_val):

        return self.\_dom[np.abs(self.\_domain-x\_val).argmin()]

    def \_\_setitem\_\_(self, x\_val, dom):

        self.\_dom[np.abs(self.\_domain-x\_val).argmin()] = round(dom, self.\_precision)

    def \_\_str\_\_(self):

        return ' + '.join([str(a) + '/' + str(b) for a,b in zip(self.\_dom, self.\_domain)])

    def \_\_get\_last\_dom\_value(self):

        return self.\_last\_dom\_value

    def \_\_set\_last\_dom\_value(self, d):

        self.\_last\_dom\_value = d

    last\_dom\_value = property(\_\_get\_last\_dom\_value, \_\_set\_last\_dom\_value)

    @property

    def name(self):

        return self.\_name

    @property

    def empty(self):

        return np.all(self.\_dom == 0)

    @property

    def name(self):

        return self.\_name

    @classmethod

    def create\_trapezoidal(cls, name, domain\_min, domain\_max, res, a, b, c, d):

        t1fs = cls(name, domain\_min, domain\_max, res)

        a = t1fs.\_adjust\_domain\_val(a)

        b = t1fs.\_adjust\_domain\_val(b)

        c = t1fs.\_adjust\_domain\_val(c)

        d = t1fs.\_adjust\_domain\_val(d)

        t1fs.\_dom = np.round(np.minimum(np.maximum(np.minimum((t1fs.\_domain-a)/(b-a), (d-t1fs.\_domain)/(d-c)), 0), 1), t1fs.\_precision)

        return t1fs

    @classmethod

    def create\_triangular(cls, name, domain\_min, domain\_max, res, a, b, c):

        t1fs = cls(name, domain\_min, domain\_max, res)

        a = t1fs.\_adjust\_domain\_val(a)

        b = t1fs.\_adjust\_domain\_val(b)

        c = t1fs.\_adjust\_domain\_val(c)

        if b == a:

            t1fs.\_dom = np.round(np.maximum((c-t1fs.\_domain)/(c-b), 0), t1fs.\_precision)

        elif b == c:

            t1fs.\_dom = np.round(np.maximum((t1fs.\_domain-a)/(b-a), 0), t1fs.\_precision)

        else:

            t1fs.\_dom = np.round(np.maximum(np.minimum((t1fs.\_domain-a)/(b-a), (c-t1fs.\_domain)/(c-b)), 0), t1fs.\_precision)

        return t1fs

    def \_adjust\_domain\_val(self, x\_val):

        return self.\_domain[np.abs(self.\_domain-x\_val).argmin()]

    def clear\_set(self):

        self.\_dom.fill(0)

    def min\_scalar(self, val):

        result = FuzzySet(f'({self.\_name}) min ({val})', self.\_domain\_min, self.\_domain\_max, self.\_res)

        result.\_dom = np.minimum(self.\_dom, val)

        return result

    def union(self, f\_set):

        result = FuzzySet(f'({self.\_name}) union ({f\_set.\_name})', self.\_domain\_min, self.\_domain\_max, self.\_res)

        result.\_dom = np.maximum(self.\_dom, f\_set.\_dom)

        return result

    def intersection(self, f\_set):

        result = FuzzySet(f'({self.\_name}) intersection ({f\_set.\_name})', self.\_domain\_min, self.\_domain\_max, self.\_res)

        result.\_dom = np.minimum(self.\_dom, f\_set.\_dom)

        return result

    def complement(self):

        result = FuzzySet(f'not ({self.\_name})', self.\_domain\_min, self.\_domain\_max, self.\_res)

        result.\_dom = 1 - self.\_dom

        return result

    def cog\_defuzzify(self):

        num = np.sum(np.multiply(self.\_dom, self.\_domain))

        den = np.sum(self.\_dom)

        return num/den

    def domain\_elements(self):

        return self.\_domain

    def dom\_elements(self):

        return self.\_dom

    def plot\_set(self, ax, col=''):

        ax.plot(self.\_domain, self.\_dom, col)

        ax.set\_ylim([-0.1,1.1])

        ax.set\_title(self.\_name)

        ax.grid(True, which='both', alpha=0.4)

        ax.set(xlabel='x', ylabel='$\mu(x)$')

if \_\_name\_\_ == "\_\_main\_\_":

    s = FuzzySet.create\_trapezoidal('test', 1, 100, 100, 20, 30, 50, 80)

    print(s.empty)

    u = FuzzySet('u', 1, 100, 100)

    print(u.empty)

    t = FuzzySet.create\_trapezoidal('test', 1, 100, 100, 30, 50, 90, 100)

    fig, axs = plt.subplots(1, 1)

    s.union(t).complement().intersection(s).min\_scalar(0.2).plot\_set(axs)

    plt.show()

    print(s.cog\_defuzzify())

#fuzzy\_system.py

from fuzzy\_logic.fuzzy\_rule import FuzzyRule

from fuzzy\_logic.fuzzy\_variable\_output import FuzzyOutputVariable

from fuzzy\_logic.fuzzy\_variable\_input import FuzzyInputVariable

import matplotlib.pyplot as plt

from matplotlib import rc

import numpy as np

class FuzzySystem:

    '''

    A type-1 fuzzy system based on Mamdani inference system

    Reference:

    ----------

    Mamdani, Ebrahim H., and Sedrak Assilian.

    "An experiment in linguistic synthesis with a

    fuzzy logic controller." Readings in Fuzzy Sets

    for Intelligent Systems. Morgan Kaufmann, 1993. 283-289.

    '''

    def \_\_init\_\_(self):

        '''

        initializes fuzzy system.

        data structures required:

            input variables -- dict, having format {variable\_name: FuzzyVariable, ...}

            output variables -- dict, having format {variable\_name: FuzzyVariable, ...}

            rules -- list of FuzzyRule

            output\_distribution -- dict holding fuzzy output for each variable having format

                                {variable\_name: FuzzySet, ...}

        '''

        self.\_input\_variables = {}

        self.\_output\_variables = {}

        self.\_rules = []

    def \_\_str\_\_(self):

        '''

        string representation of the system.

        Returns:

        --------

        str: str, string representation of the system in the form

                Input:

                input\_variable\_name(set\_names)...

                Output:

                output\_variable\_name(set\_names)...

                Rules:

                IF [antecedent clauses] THEN [consequent clauses]

        '''

        ret\_str = 'Input: \n'

        for n, s in self.\_input\_variables.items():

            ret\_str = ret\_str + f'{n}: ({s})\n'

        ret\_str = ret\_str + 'Output: \n'

        for n, s in self.\_output\_variables.items():

            ret\_str = ret\_str + f'{n}: ({s})\n'

        ret\_str = ret\_str + 'Rules: \n'

        for rule in self.\_rules:

            ret\_str = ret\_str + f'{rule}\n'

        return ret\_str

    def add\_input\_variable(self, variable):

        '''

        adds an input variable to the system

        Arguments:

        ----------

        variable -- FuzzyVariable, the input variable

        '''

        self.\_input\_variables[variable.name] = variable

    def add\_output\_variable(self, variable):

        self.\_output\_variables[variable.name] = variable

    def get\_input\_variable(self, name):

        '''

        get an input variable given the name

        Arguments:

        -----------

        name -- str, name of variable

        Returns:

        --------

        variable -- FuzzyVariable, the input variable

        '''

        return self.\_input\_variables[name]

    def get\_output\_variable(self, name):

        '''

        get an output variable given the name

        Arguments:

        -----------

        name -- str, name of variable

        Returns:

        --------

        variable -- FuzzyVariable, the output variable

        '''

        return self.\_output\_variables[name]

    def \_clear\_output\_distributions(self):

        '''

        used for each iteration. The fuzzy result is cleared

        '''

        map(lambda output\_var: output\_var.clear\_output\_distribution(), self.\_output\_variables.values())

    def add\_rule(self, antecedent\_clauses, consequent\_clauses):

        '''

        adds a new rule to the system.

        TODO: add checks

        Arguments:

        -----------

        antecedent\_clauses -- dict, having the form {variable\_name:set\_name, ...}

        consequent\_clauses -- dict, having the form {variable\_name:set\_name, ...}

        '''

        # create a new rule

        # new\_rule = FuzzyRule(antecedent\_clauses, consequent\_clauses)

        new\_rule = FuzzyRule()

        for var\_name, set\_name in antecedent\_clauses.items():

            # get variable by name

            var = self.get\_input\_variable(var\_name)

            # get set by name

            f\_set = var.get\_set(set\_name)

            # add clause

            new\_rule.add\_antecedent\_clause(var, f\_set)

        for var\_name, set\_name in consequent\_clauses.items():

            var = self.get\_output\_variable(var\_name)

            f\_set = var.get\_set(set\_name)

            new\_rule.add\_consequent\_clause(var, f\_set)

        # add the new rule

        self.\_rules.append(new\_rule)

    def evaluate\_output(self, input\_values):

        '''

        Executes the fuzzy inference system for a set of inputs

        Arguments:

        -----------

        input\_values -- dict, containing the inputs to the systems in the form

                            {input\_variable\_name: value, ...}

        Returns:

        --------

        output -- dict, containing the outputs from the systems in the form

                    {output\_variable\_name: value, ...}

        '''

        # clear the fuzzy consequences as we are evaluating a new set of inputs.

        # can be optimized by comparing if the inputs have changes from the previous

        # iteration.

        self.\_clear\_output\_distributions()

        # Fuzzify the inputs. The degree of membership will be stored in

        # each set

        for input\_name, input\_value in input\_values.items():

            self.\_input\_variables[input\_name].fuzzify(input\_value)

        # evaluate rules

        for rule in self.\_rules:

            rule.evaluate()

        # finally, defuzzify all output distributions to get the crisp outputs

        output = {}

        for output\_var\_name, output\_var in self.\_output\_variables.items():

            output[output\_var\_name] = output\_var.get\_crisp\_output()

        return output

    def evaluate\_output\_info(self, input\_values):

        '''

        Executes the fuzzy inference system for a set of inputs

        Arguments:

        -----------

        input\_values -- dict, containing the inputs to the systems in the form

                            {input\_variable\_name: value, ...}

        Returns:

        --------

        output -- dict, containing the outputs from the systems in the form

                    {output\_variable\_name: value, ...}

        '''

        info = {}

        # clear the fuzzy consequences as we are evaluating a new set of inputs.

        # can be optimized by comparing if the inputs have changes from the previous

        # iteration.

        self.\_clear\_output\_distributions()

        # Fuzzify the inputs. The degree of membership will be stored in

        # each set

        fuzzification\_info = []

        for input\_name, input\_value in input\_values.items():

            fuzzification\_info.append(self.\_input\_variables[input\_name].fuzzify\_info(input\_value))

        info['fuzzification'] = '\n'.join(fuzzification\_info)

        # evaluate rules

        rule\_info = []

        for rule in self.\_rules:

            rule\_info.append(rule.evaluate\_info())

        info['rules'] = '\n'.join(rule\_info)

        # finally, defuzzify all output distributions to get the crisp outputs

        output = {}

        for output\_var\_name, output\_var in self.\_output\_variables.items():

            output[output\_var\_name], info = output\_var.get\_crisp\_output\_info()

            # info[output\_var\_name] = info

        return output, info

    def plot\_system(self):

        total\_var\_count = len(self.\_input\_variables) + len(self.\_output\_variables)

        if total\_var\_count <2:

            total\_var\_count = 2

        fig, axs = plt.subplots(total\_var\_count, 1)

        fig.tight\_layout(pad=1.0)

        for idx, var\_name in enumerate(self.\_input\_variables):

            self.\_input\_variables[var\_name].plot\_variable(ax=axs[idx], show=False)

        for idx, var\_name in enumerate(self.\_output\_variables):

            self.\_output\_variables[var\_name].plot\_variable(ax=axs[len(self.\_input\_variables)+idx], show=False)

        plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

    pass

fuzzy\_variable\_input.py

from fuzzy\_logic.fuzzy\_variable import FuzzyVariable

class FuzzyInputVariable(FuzzyVariable):

    def \_\_init\_\_(self, name, min\_val, max\_val, res):

        super().\_\_init\_\_(name, min\_val, max\_val, res)

    def fuzzify(self, value):

        '''

        performs fuzzification of the variable. used when the

        variable is an input one

        Arguments:

        -----------

        value -- number, input value for the variable

        '''

        # get dom for each set and store it - it will be required for each rule

        for set\_name, f\_set in self.\_sets.items():

            f\_set.last\_dom\_value = f\_set[value]

    def fuzzify\_info(self, value):

        '''

        performs fuzzification of the variable. used when the

        variable is an input one

        Arguments:

        -----------

        value -- number, input value for the variable

        '''

        # get dom for each set and store it - it will be required for each rule

        for set\_name, f\_set in self.\_sets.items():

            f\_set.last\_dom\_value = f\_set[value]

        res = []

        res.append(self.\_name)

        res.append('\n')

        for \_, f\_set in self.\_sets.items():

            res.append(f\_set.name)

            res.append(str(f\_set.last\_dom\_value))

            res.append('\n')

        return ' '.join(res)

if \_\_name\_\_ == "\_\_main\_\_":

    pass

#fuzzy\_variable\_output.py

from fuzzy\_logic.fuzzy\_variable import FuzzyVariable

from fuzzy\_logic.fuzzy\_set import FuzzySet

class FuzzyOutputVariable(FuzzyVariable):

    def \_\_init\_\_(self, name, min\_val, max\_val, res):

        super().\_\_init\_\_(name, min\_val, max\_val, res)

        self.\_output\_distribution = FuzzySet(name, min\_val, max\_val, res)

    def clear\_output\_distribution(self):

        self.\_output\_distribution.clear\_set()

    def add\_rule\_contribution(self, rule\_consequence):

        self.\_output\_distribution = self.\_output\_distribution.union(rule\_consequence)

    def get\_crisp\_output(self):

        return self.\_output\_distribution.cog\_defuzzify()

    def get\_crisp\_output\_info(self):

        return self.\_output\_distribution.cog\_defuzzify(), self.\_output\_distribution

if \_\_name\_\_ == "\_\_main\_\_":

    pass

#fuzzy\_variable.py

from fuzzy\_logic.fuzzy\_set import FuzzySet

import matplotlib.pyplot as plt

import numpy as np

class FuzzyVariable():

    '''

    A type-1 fuzzy variable that is mage up of a number of type-1 fuzzy sets

    '''

    def \_\_init\_\_(self, name, min\_val, max\_val, res):

        '''

        creates a new type-1 fuzzy variable (universe)

        Arguments:

        ----------

            min\_val -- number, minimum value of variable

            max\_val -- number, maximum value of variable

            res -- int, resolution of variable

        '''

        self.\_sets={}

        self.\_max\_val = max\_val

        self.\_min\_val = min\_val

        self.\_res = res

        self.\_name = name

    def \_\_str\_\_(self):

        return ', '.join(self.\_sets.keys())

    @property

    def name(self):

        return self.\_name

    def \_add\_set(self, name, f\_set):

        '''

        adds a fuzzy set to the variable

        Arguments:

        ----------

            name -- string, name of the set

            f\_set -- FuzzySet, The set

        '''

        self.\_sets[name] = f\_set

    def get\_set(self, name):

        '''

        returns a set given the name

        Arguments:

        ----------

        name -- str, set name

        Returns:

        --------

        set -- FuzzySet, the set

        '''

        return self.\_sets[name]

    def add\_triangular(self, name, low, mid, high):

        new\_set = FuzzySet.create\_triangular(name, self.\_min\_val, self.\_max\_val, self.\_res, low, mid, high)

        self.\_add\_set(name, new\_set)

        return new\_set

    def add\_trapezoidal(self, name, a, b, c, d):

        new\_set = FuzzySet. create\_trapezoidal(name, self.\_min\_val, self.\_max\_val, self.\_res, a, b, c, d)

        self.\_add\_set(name, new\_set)

        return new\_set

    def plot\_variable(self, ax=None, show=True):

        '''

        plots a graphical representation of the fuzzy variable

        Reference:

        ----------

            https://stackoverflow.com/questions/4700614/how-to-put-the-legend-out-of-the-plot

        '''

        if ax == None:

            ax = plt.subplot(111)

        for n ,s in self.\_sets.items():

            ax.plot(s.domain\_elements(), s.dom\_elements(), label=n)

        # Shrink current axis by 20%

        pos = ax.get\_position()

        ax.set\_position([pos.x0, pos.y0, pos.width \* 0.8, pos.height])

        ax.grid(True, which='both', alpha=0.4)

        ax.set\_title(self.\_name)

        ax.set(xlabel='x', ylabel='$\mu (x)$')

        # Put a legend to the right of the current axis

        ax.legend(loc='center left', bbox\_to\_anchor=(1, 0.5))

        if show:

            plt.show()

***Driver Code:***

from fuzzy\_logic.fuzzy\_variable\_output import FuzzyOutputVariable

from fuzzy\_logic.fuzzy\_variable\_input import FuzzyInputVariable

# from fuzzy\_system.fuzzy\_variable import FuzzyVariable

from fuzzy\_logic.fuzzy\_system import FuzzySystem

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

temp.add\_triangular('Cold', 10, 10, 25)

temp.add\_triangular('Medium', 15, 25, 35)

temp.add\_triangular('Hot', 25, 40, 40)

humidity = FuzzyInputVariable('Humidity', 20, 100, 100)

humidity.add\_triangular('Wet', 20, 20, 60)

humidity.add\_trapezoidal('Normal', 30, 50, 70, 90)

humidity.add\_triangular('Dry', 60, 100, 100)

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)

system = FuzzySystem()

system.add\_input\_variable(temp)

system.add\_input\_variable(humidity)

system.add\_output\_variable(motor\_speed)

system.add\_rule(

        { 'Temperature':'Cold',

            'Humidity':'Wet' },

        { 'Speed':'Slow'})

system.add\_rule(

        { 'Temperature':'Cold',

            'Humidity':'Normal' },

        { 'Speed':'Slow'})

system.add\_rule(

        { 'Temperature':'Medium',

            'Humidity':'Wet' },

        { 'Speed':'Slow'})

system.add\_rule(

        { 'Temperature':'Medium',

            'Humidity':'Normal' },

        { 'Speed':'Moderate'})

system.add\_rule(

        { 'Temperature':'Cold',

            'Humidity':'Dry' },

        { 'Speed':'Moderate'})

system.add\_rule(

        { 'Temperature':'Hot',

            'Humidity':'Wet' },

        { 'Speed':'Moderate'})

system.add\_rule(

        { 'Temperature':'Hot',

            'Humidity':'Normal' },

        { 'Speed':'Fast'})

system.add\_rule(

        { 'Temperature':'Hot',

            'Humidity':'Dry' },

        { 'Speed':'Fast'})

system.add\_rule(

        { 'Temperature':'Medium',

            'Humidity':'Dry' },

        { 'Speed':'Fast'})

output = system.evaluate\_output({

                'Temperature':18,

                'Humidity':60

        })

print(output)

# print('fuzzification\n-------------\n', info['fuzzification'])

# print('rules\n-----\n', info['rules'])

system.plot\_system()

***Driver code- 2in 2out***

from fuzzy\_logic.fuzzy\_variable\_output import FuzzyOutputVariable

from fuzzy\_logic.fuzzy\_variable\_input import FuzzyInputVariable

from fuzzy\_logic.fuzzy\_system import FuzzySystem

x1 = FuzzyInputVariable('x1', 0, 100, 100)

x1.add\_triangular('S', 0, 25, 50)

x1.add\_triangular('M', 25, 50, 75)

x1.add\_triangular('L', 50, 75, 100)

x2 = FuzzyInputVariable('x2', 0, 100, 100)

x2.add\_triangular('S', 0, 25, 50)

x2.add\_triangular('M', 25, 50, 75)

x2.add\_triangular('L', 50, 75, 100)

y = FuzzyOutputVariable('y', 0, 100, 100)

y.add\_triangular('S', 0, 25, 50)

y.add\_triangular('M', 25, 50, 75)

y.add\_triangular('L', 50, 75, 100)

z = FuzzyOutputVariable('z', 0, 100, 100)

z.add\_triangular('S', 0, 25, 50)

z.add\_triangular('M', 25, 50, 75)

z.add\_triangular('L', 50, 75, 100)

system = FuzzySystem()

system.add\_input\_variable(x1)

system.add\_input\_variable(x2)

system.add\_output\_variable(y)

system.add\_output\_variable(z)

system.add\_rule(

        { 'x1':'S',

            'x2':'S' },

        { 'y':'S',

            'z':'L' })

system.add\_rule(

        { 'x1':'M',

            'x2':'M' },

        { 'y':'M',

            'z':'M' })

system.add\_rule(

        { 'x1':'L',

            'x2':'L' },

        { 'y':'L',

            'z':'S' })

system.add\_rule(

        { 'x1':'S',

            'x2':'M' },

        { 'y':'S',

            'z':'L' })

system.add\_rule(

        { 'x1':'M',

            'x2':'S' },

        { 'y':'S',

            'z':'L' })

system.add\_rule(

        { 'x1':'L',

            'x2':'M' },

        { 'y':'L',

            'z':'S' })

system.add\_rule(

        { 'x1':'M',

            'x2':'L' },

        { 'y':'L',

            'z':'S' })

system.add\_rule(

        { 'x1':'L',

            'x2':'S' },

        { 'y':'M',

            'z':'M' })

system.add\_rule(

        { 'x1':'S',

            'x2':'L' },

        { 'y':'M',

            'z':'M' })

output = system.evaluate\_output({

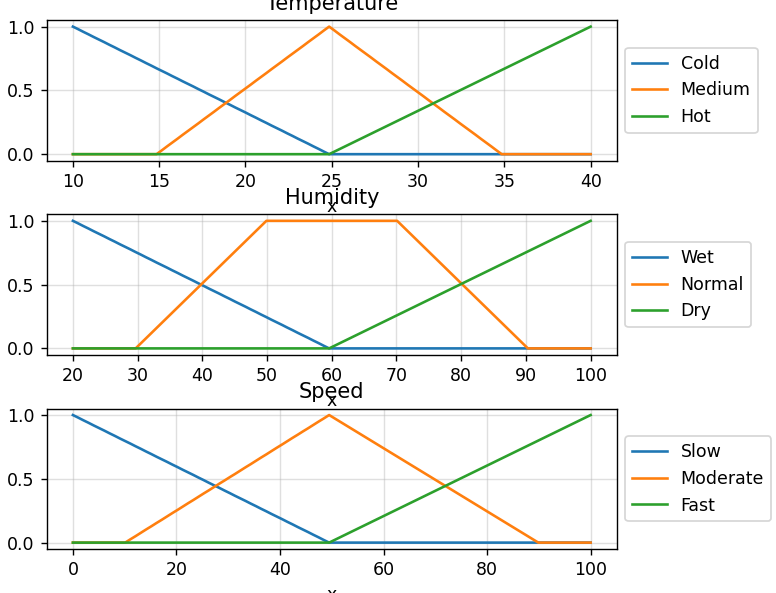
            'x1':35,

            'x2':75

        })

print(output)

***Output:***

******

***2in\_2out***

******

***Conclusion:***

*Implemented Fuzzy Inference System.*