# QFIE_Package

*Release 1.1.0*

**Roberto Schiattarella**

**Mar 13, 2023**

# API:

CHAPTER

ONE

# HOW TO OBTAIN THE CODE

## 1.1 Open Source

The `QFIE` package is open source and available at https://github.com/Quasar-UniNA/QFIE

## 1.2 Installation

The package can be installed manually:

```
git clone https://github.com/Quasar-UniNA/QFIE.git QFIE
cd QFIE
pip install .
```

or if you are planning to extend or develop code replace last command with:

```
pip install -e .
```

CHAPTER

TWO

# MOTIVATION

- QFIE Python package gives the opportunity of easily implementing quantum fuzzy inference engines as those proposed in: 10.1109/TFUZZ.2022.3202348
- **QFIE docs is equipped with two notebook jupyter examples:**
    1. QFIE for controlling a Fan Speed System
    2. QFIE for Lighting Control.
- **The main changes of QFIE v.1.1.0 can be seen in the notebook jupyter:**
    3. QFIE_v_1_1_0.ipynb

CHAPTER

THREE

# HOW TO CITE QFIE?

When using this software in your research, please cite the following publication:

Bibtex:

```
@ARTICLE{9869303,
author={Acampora, Giovanni and Schiattarella, Roberto and Vitiello, Autilia},
journal={IEEE Transactions on Fuzzy Systems},
title={On the Implementation of Fuzzy Inference Engines on Quantum Computers},
year={2022},
volume={},
number={},
pages={1-15},
doi={10.1109/TFUZZ.2022.3202348}}
```

## 3.1 API Documentation

### 3.1.1 QFIE package

#### QFIE.FuzzyEngines module

This module implements the base class for setting up the quantum fuzzy inference engine proposed in doi: 10.1109/TFUZZ.2022.3202348.

**class** QFIE.FuzzyEngines.**QuantumFuzzyEngine**(*verbose=True*)

Bases: object

Class implementing the Quantum Fuzzy Inference Engine proposed in:

G. Acampora, R. Schiattarella and A. Vitiello, "On the Implementation of Fuzzy Inference Engines on Quantum Computers," in IEEE Transactions on Fuzzy Systems, 2022, doi: 10.1109/TFUZZ.2022.3202348.

**add_input_fuzzysets**(*var_name*, *set_names*, *sets*)

Set the partition for the input fuzzy variable 'var_name'.

**Parameters**

- **var_name** (*str*) – name of the fuzzy variable defined with input_variable method previously.
- **set_names** (*list*) – list of fuzzy sets' name as str.
- **sets** (*list*) – list of scikit-fuzzy membership function objects.

**Returns**
None

`add_output_fuzzysets`(*var_name*, *set_names*, *sets*)

Set the partition for the output fuzzy variable 'var_name'.

**Parameters**

- **var_name** (*str*) – name of the fuzzy variable defined with output_variable method previously.
- **set_names** (*list*) – list of fuzzy sets' name as str.
- **sets** (*list*) – list of scikit-fuzzy membership function objects.

**Returns**
None

`build_inference_qc`(*input_values*, *distributed=False*, *draw_qc=False*, ***kwargs*)

This function builds the quantum circuit implementing the QFIE, initializing the input quantum registers according to the 'input_value' argument.

**Parameters**

- **input_values** (*dict*) – dictionary containing the crisp input values of the system. E.g. {'var_name_1' (str): x_1 (float), , 'var_name_n' (str): x_n (float)}
- **default** (*draw_qc (Bool -)* – False): True for drawing the quantum circuit built. False otherwise.
- **distributed** – True to implement the distributed version of the quantum oracle. False otherwise.

**Returns**
None

`counts_evaluator`(*n_qubits*, *counts*)

Function returning the alpha values for alpha-cutting the output fuzzy sets according to the probability of measuring the related basis states on the output quantum register.

**Parameters**

- **n_qubits** (*int*) – number of qubits in the output quantum register.
- **counts** (*dict*) – counting dictionary of the output quantum register measurement.

**Returns**
alpha values for alpha-cutting the output fuzzy sets as 'dict'.

`execute`(*n_shots: int*, *plot_histo=False*, *GPU=False*, ***kwargs*)

Run the inference engine.

**Parameters**

- **n_shots** (*int*) – Number of shots.
- **plot_histo** (*Bool- default False*) – True for plotting the counts histogram.
- **GPU** – True for using GPU for simulation. Use False if backend is a real device.

**Returns**
Crisp output of the system.

**filter_rules**(*rules*, *output_term*)

Searches the rule list and picks only the rules corresponding to the same output value (y_k at fixed k).

Rules must be formatted as follows: 'if var_1 is x_i and var_2 is x_k and and var_n is x_l then out_1 is y_k'

**Parameters**

- **rules** (*list*) – list of rules as strings.
- **output_term** (*str*) – single output term y_k at fixed k as string.

**Returns**
Filtered rules as a new list.

**input_variable**(*name*, *range*)

Define the input variable "name" of the system.

**Parameters**

- **name** (*str*) – Name of the variable as string.
- **range** (*np array*) – Universe of the discourse for the input variable.

**Returns**
None

**output_variable**(*name*, *range*)

Define the output variable "name" of the system.

**Parameters**

- **name** (*str*) – Name of the variable as string.
- **range** (*np array*) – Universe of the discourse for the output variable.

**Returns**
None

**set_rules**(*rules*)

Set the rule-base of the system.

Rules must be formatted as follows: 'if var_1 is x_i and var_2 is x_k and and var_n is x_l then out_1 is y_k'

**Parameters**
**rules** (*list*) – list of rules as strings.

**Returns**
None

**truncate**(*n*, *decimals=0*)

### QFIE.QFS module

QFIE.QFS.**compute_qc**(*backend*, *qc*, *qc_label*, *n_shots*, *verbose=True*, *transpilation_info=False*)

QFIE.QFS.**convert_rule**(*qc*, *fuzzy_rule*, *partitions*, *output_partition*)

Function which convert a fuzzy rule in the equivalent quantum circuit. You can use multiple times convert_rule to concatenate the quantum circuits related to different rules.

QFIE.QFS.**generate_circuit**(*fuzzy_partitions*)

Function generating a quantum circuit with width required by QFS

QFIE.QFS.**merge_subcounts**(*subcounts*, *output_partition*)

QFIE.QFS.**negation_0**(*qc*, *qr*, *bit_string*)

Function which insert a NOT gate if the bit in the rule is 0

QFIE.QFS.**output_register**(*qc*, *output_partition*)

QFIE.QFS.**output_single_qubit_register**(*qc*, *name*)

QFIE.QFS.**select_qreg_by_name**(*qc*, *name*)

Function returning the quantum register in QC selected by name

### QFIE.fuzzy_partitions module

**class** QFIE.fuzzy_partitions.**fuzzy_partition**(*name*, *sets*)

Bases: object

**associate_quantum_states**()

**len_partition**()

**class** QFIE.fuzzy_partitions.**fuzzy_rules**

Bases: object

**add_rules**(*rule*, *partitions*)

NB: specify in partitions the list of partitions which appears in the rule, in the order in which they appear

### Module contents

## 3.2 QFIE For Fan Speed Control

This notebook aims to illustrate how QFIE can control efficiently the speed of a Fan according to the input variables 'Core Temperature' and 'Clock Speed'. Finally, the control surface of the system will be obtained both using QFIE and the canonical Mamdani fuzzy system, which configuration is reported in the paper.

Firstly, let us import the required libraries.

[1]:
```
import numpy as np
import skfuzzy as fuzz
import matplotlib.pyplot as plt
import QFIE.FuzzyEngines as FE
```

The system is a MISO composed of two input variables and one output variable. For each of them three fuzzy sets are considered.

- For core temperature the fuzzy sets used are releted to the linguistic terms: 'cold', 'warm' and 'hot'.
- For clock speed the fuzzy sets are related to the linguistic terms: 'low', 'normal' and 'turbo'
- For the output variable, named 'Fan Speed' two fuzzy sets are used related to the linguistic terms: 'slow' and 'fast

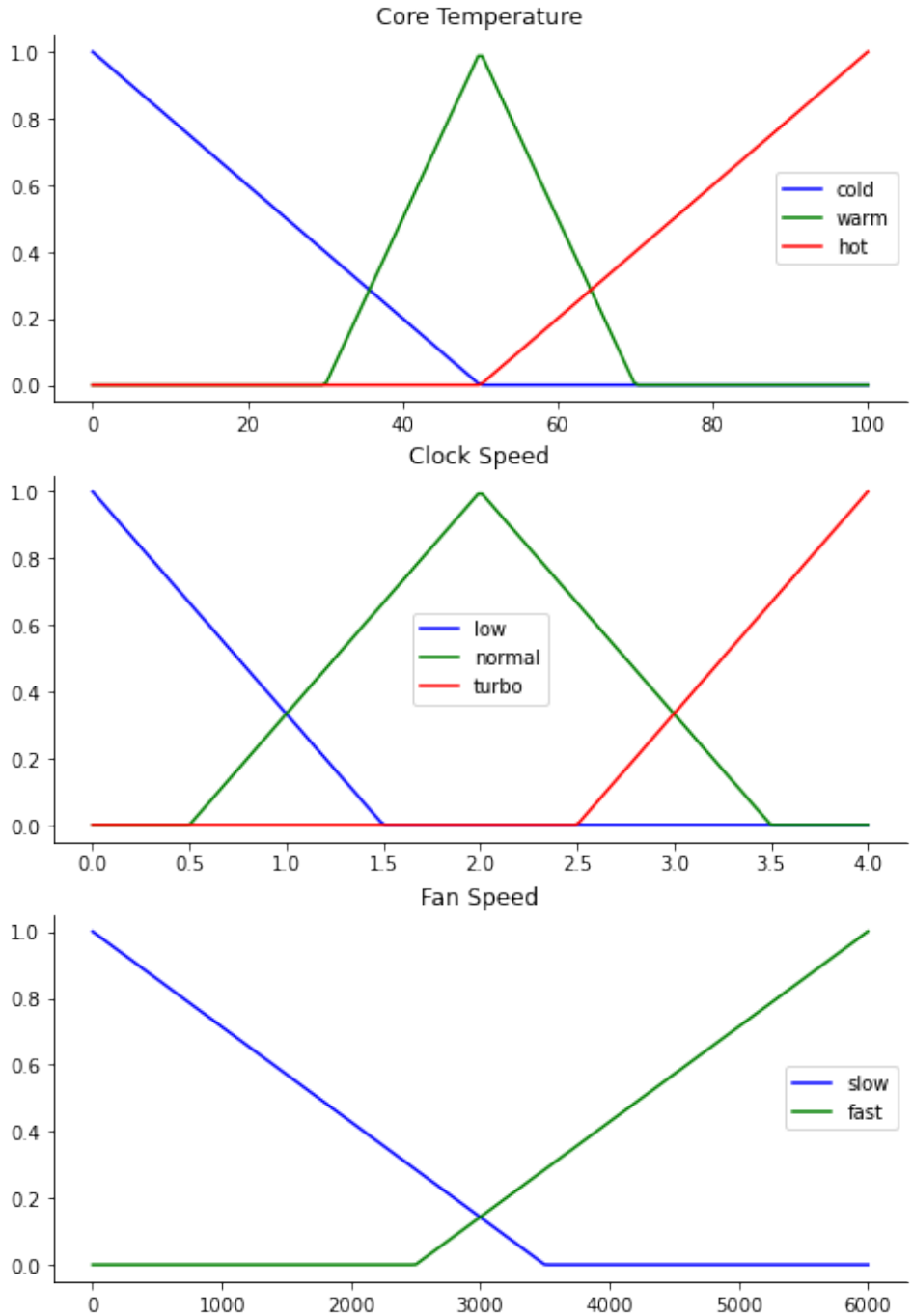
By running the following cell, the fuzzy sets used are graphically shown.

[2]:
```
core_temperature = np.linspace(0, 100, 200)
clock_speed = np.linspace(0, 4, 200)
fan_speed = np.linspace(0, 6000, 200)



t_cold = fuzz.trimf(core_temperature, [0,0,50])
t_warm = fuzz.trimf(core_temperature, [30,50,70])
t_hot = fuzz.trimf(core_temperature, [50, 100, 100])

cs_low = fuzz.trimf(clock_speed, [0,0,1.5])
cs_normal = fuzz.trimf(clock_speed, [0.5,2,3.5])
cs_turbo = fuzz.trimf(clock_speed, [2.5,4,4])

fs_slow = fuzz.trimf(fan_speed, [0, 0, 3500])
fs_fast = fuzz.trimf(fan_speed, [2500, 6000, 6000])

# Visualize these universes and membership functions
fig, (ax0, ax1, ax2) = plt.subplots(nrows=3, figsize=(8, 12))


ax0.plot(core_temperature, t_cold, 'b', linewidth=1.5, label='cold')
ax0.plot(core_temperature, t_warm, 'g', linewidth=1.5, label='warm')
ax0.plot(core_temperature, t_hot, 'r', linewidth=1.5, label='hot')
ax0.set_title('Core Temperature')
ax0.legend()

ax1.plot(clock_speed, cs_low, 'b', linewidth=1.5, label='low')
ax1.plot(clock_speed, cs_normal, 'g', linewidth=1.5, label='normal')
ax1.plot(clock_speed, cs_turbo, 'r', linewidth=1.5, label='turbo')
ax1.set_title('Clock Speed')
ax1.legend()


ax2.plot(fan_speed, fs_slow, 'b', linewidth=1.5, label='slow')
ax2.plot(fan_speed, fs_fast, 'g', linewidth=1.5, label='fast')
ax2.set_title('Fan Speed')
ax2.legend()

# Turn off top/right axes
for ax in (ax0, ax1, ax2):
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()
```

(continues on next page)

(continued from previous page)

```
plt.show()
```




nbsphinx-code-borderwhite

The rules base of the system is composed of the following rules:

- if temp is cold and clock is low then fan is slow
- if temp is cold and clock is normal then fan is slow'
- if temp is cold and clock is turbo then fan is fast'

- if temp is warm and clock is low then fan is slow'
- if temp is warm and clock is normal then fan is slow'
- if temp is warm and clock is turbo then fan is fast'
- if temp is hot and clock is low then fan is fast'
- if temp is hot and clock is normal then fan is fast'
- if temp is hot and clock is turbo then fan is fast'

By running the following two cells QFIE is properly initialized

[3]:
```
rules = ['if temp is cold and clock is low then fan is slow',
         'if temp is cold and clock is normal then fan is slow',
         'if temp is cold and clock is turbo then fan is fast',
         'if temp is warm and clock is low then fan is slow',
         'if temp is warm and clock is normal then fan is slow',
         'if temp is warm and clock is turbo then fan is fast',
         'if temp is hot and clock is low then fan is fast',
         'if temp is hot and clock is normal then fan is fast',
         'if temp is hot and clock is turbo then fan is fast']
```

[4]:
```
qfie = FE.QuantumFuzzyEngine(verbose=False)

qfie.input_variable(name='temp', range=core_temperature)
qfie.input_variable(name='clock', range=clock_speed)
qfie.output_variable(name='fan', range=fan_speed)

qfie.add_input_fuzzysets(var_name='temp', set_names=['cold', 'warm', 'hot'], sets=[t_
↪cold, t_warm, t_hot])
qfie.add_input_fuzzysets(var_name='clock', set_names=['low', 'normal', 'turbo'],␣
↪sets=[cs_low, cs_normal, cs_turbo])
qfie.add_output_fuzzysets(var_name='fan', set_names=['slow', 'fast'],sets=[fs_slow,fs_
↪fast])
qfie.set_rules(rules)
```

To obtain the control surface of QFIE, it is used to fire the rules in 100 different input combinations. In detail, for 10 values of temperature, 10 different values of clock speed are considered. The 100 outputs obtained are collected in a list which will be used at the end of the notebook to plot the control surface.

[6]:
```
temp_list, speed_list = [t for t in range(0, 100, 10)], [s/2.5 for s in range(0, 10, 1)]
f_quantum = []
for t in temp_list:
    for s in speed_list:
        qfie.build_inference_qc({'temp':t, 'clock':s}, draw_qc=False)
        f_quantum.append(qfie.execute(1000, plot_histo=False)[0])
```

---

Then, the classical Mamdani system is implemented by using the python library Sci-kit Fuzzy. The following cells initializes the system as required from the library.

[7]:
```
from skfuzzy import control as ctrl
```

[8]:
```
temp = ctrl.Antecedent(core_temperature, 'temp')
speed = ctrl.Antecedent(clock_speed , 'speed')
fan = ctrl.Consequent(fan_speed, 'fan')

temp['cold'] = t_cold
temp['warm'] = t_warm
temp['hot'] = t_hot

speed['low'] = cs_low
speed['normal'] = cs_normal
speed['turbo'] = cs_turbo

fan['slow'] = fs_slow
fan['fast'] = fs_fast

rule1 = ctrl.Rule(temp['cold'] & speed['low'], fan['slow'])
rule2 = ctrl.Rule(temp['cold'] & speed['normal'], fan['slow'])
rule3 = ctrl.Rule(temp['cold'] & speed['turbo'], fan['fast'])
rule4 = ctrl.Rule(temp['warm'] & speed['low'], fan['slow'])
rule5 = ctrl.Rule(temp['warm'] & speed['normal'], fan['slow'])
rule6 = ctrl.Rule(temp['warm'] & speed['turbo'], fan['fast'])
rule7 = ctrl.Rule(temp['hot'] & speed['low'], fan['fast'])
rule8 = ctrl.Rule(temp['hot'] & speed['normal'], fan['fast'])
rule9 = ctrl.Rule(temp['hot'] & speed['turbo'], fan['fast'])

fan_ctrl = ctrl.ControlSystem([rule1, rule2, rule3, rule4, rule5, rule6, rule7, rule8,␣
↪rule9])
fan_sys = ctrl.ControlSystemSimulation(fan_ctrl)
```

By running the following cell, the same combinations of inputs are passed to the classical control which computes the 100 output. All the output values are stored in a list.

[9]:
```
f_classical = []
for t in temp_list:
    for s in speed_list:
        fan_sys.input['temp']=t
        fan_sys.input['speed']=s

        fan_sys.compute()
        f_classical.append(fan_sys.output['fan'])
```

At this point, both the QFIE and Classical system' control surfaces can be plotted. Firstly, the following cell shows the former surface.

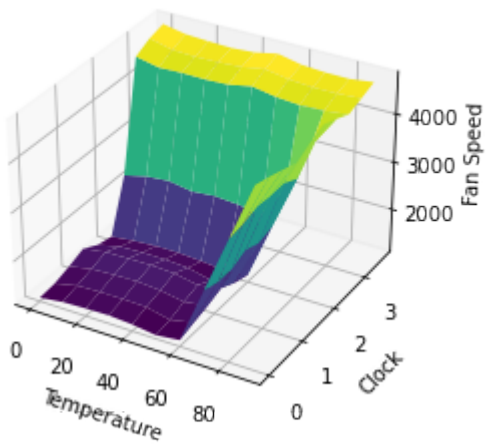
[10]:
```
from mpl_toolkits.mplot3d import Axes3D

X, Y, = temp_list, speed_list
```

(continues on next page)

(continued from previous page)

```
plotx,ploty, = np.meshgrid(np.linspace(np.min(X),np.max(X),10), np.linspace(np.min(Y),np.
↪max(Y),10))
Z = np.zeros_like(plotx)
counter = 0
for i in range(len(temp_list)):
    for j in range(len(speed_list)):
        Z[i,j]=f_quantum[counter]
        counter+=1

zfig = plt.figure()
ax = plt.axes(projection='3d')

ax.plot_surface(plotx, ploty, Z, rstride=1, cstride=1, cmap='viridis',
                        linewidth=0.4, antialiased=True)
ax.set_xlabel('Temperature')
ax.set_ylabel('Clock')
ax.set_zlabel('Fan Speed')
```

[10]:
```
Text(0.5, 0, 'Fan Speed')
```




nbsphinx-code-borderwhite

Then, the classical surface is displayed by running the following cell.

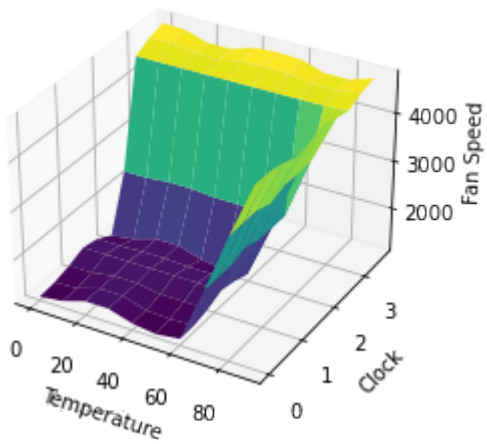
[11]:
```
X, Y, = temp_list, speed_list
plotx,ploty, = np.meshgrid(np.linspace(np.min(X),np.max(X),10), np.linspace(np.min(Y),np.
↪max(Y),10))
Z = np.zeros_like(plotx)
counter = 0
for i in range(len(temp_list)):
    for j in range(len(speed_list)):
        Z[i,j]=f_classical[counter]
        counter+=1
zfig = plt.figure()
ax = plt.axes(projection='3d')

ax.plot_surface(plotx, ploty, Z, rstride=1, cstride=1, cmap='viridis',
                        linewidth=0.4, antialiased=True)
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('Temperature')
ax.set_ylabel('Clock')
ax.set_zlabel('Fan Speed')
```

[11]:
```
Text(0.5, 0, 'Fan Speed')
```




nbsphinx-code-borderwhite

The two surfaces show how QFIE is able to control in a very similar way to a classical Mamdani fuzzy control the Fan Speed system, despite the computational advantage in firing the fuzzy rules in a parallel way.

## 3.3 QFIE For Lighting Controller System

This notebook aims to illustrate how QFIE can control efficiently a lighting system, in which the control system dim the bulb light automatically according to the environmental Light. Two input variables are considered, namely the environment light and the changing rate of the environmental light. Finally, the control surface of the system will be obtained both using QFIE and the canonical Mamdani fuzzy system, which configuration is reported in the paper.

Firstly, let us import the required libraries.

[1]:
```
import numpy as np
import skfuzzy as fuzz
import matplotlib.pyplot as plt
import QFIE.FuzzyEngines as FE
```

The system is a MISO composed of two input variables and one output variable. In detail:

Inputs:

- Environment Light (env_light): three linguistic terms are considered: 'Dark', 'Medium', 'Light'
- Changing Rate (change_rate): three lingiuistic terms are considered: 'Negative Small', 'Zero', 'Positive Small'

Output:

- Dimmer Control (dimmer_ctrl): four linguistic terms are considered: 'Very Small', 'Small', 'Big', 'Very Big'

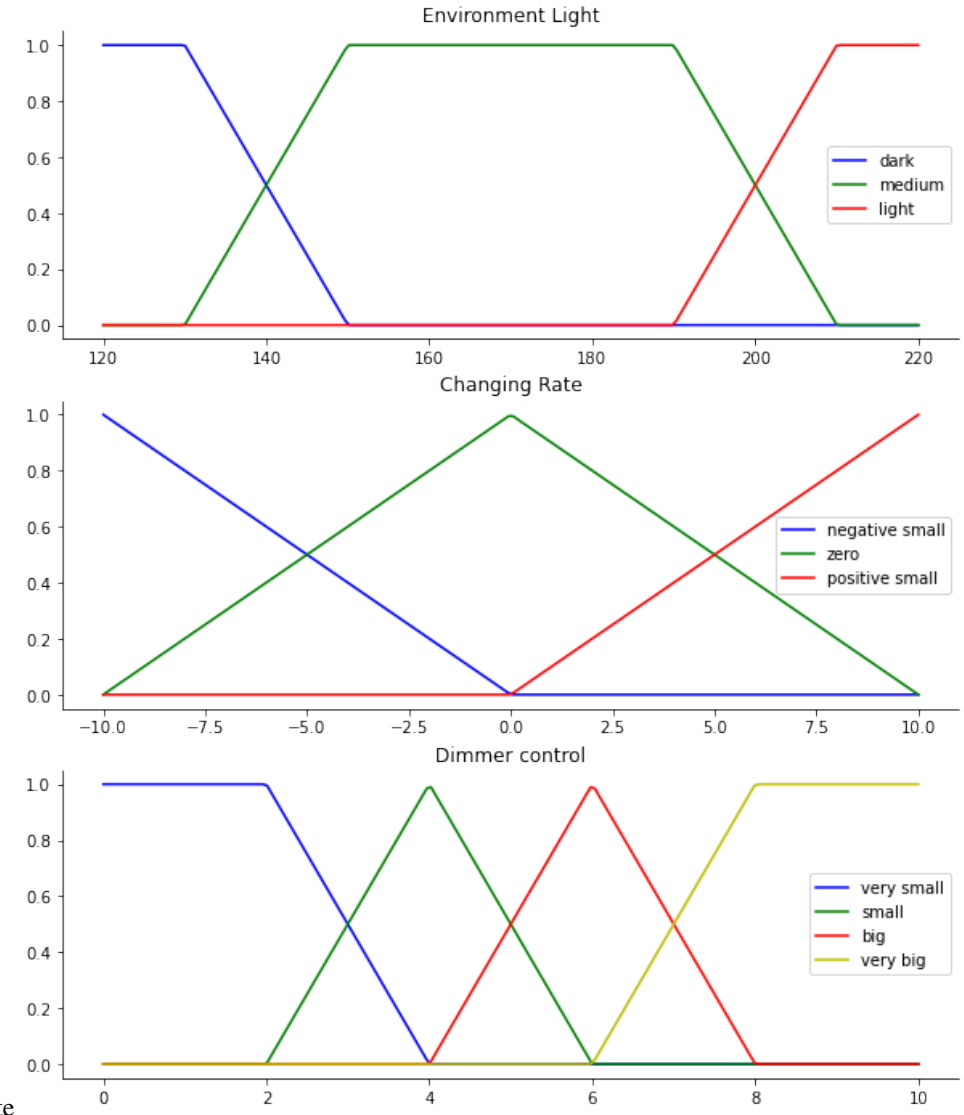
By running the following cell, the fuzzy sets used are graphically shown.

[2]:
```
env_light = np.linspace(120, 220, 200)
changing_rate = np.linspace(-10, 10, 200)
dimmer_control = np.linspace(0, 10, 200)



l_dark = fuzz.trapmf(env_light, [120,120,130,150])
l_medium = fuzz.trapmf(env_light, [130,  150, 190,210])
l_light = fuzz.trapmf(env_light, [190,  210, 220, 220])

r_ns = fuzz.trimf(changing_rate, [-10,-10,0])
r_zero = fuzz.trimf(changing_rate, [-10,0,10])
r_ps = fuzz.trimf(changing_rate, [0,10,10])

dm_vs = fuzz.trapmf(dimmer_control, [0,0,2,4])
dm_s = fuzz.trimf(dimmer_control, [2,4,6])
dm_b = fuzz.trimf(dimmer_control, [4,6,8])
dm_vb = fuzz.trapmf(dimmer_control, [6,8,10,10])




# Visualize these universes and membership functions
fig, (ax0, ax1, ax2) = plt.subplots(nrows=3, figsize=(10, 12))


ax0.plot(env_light, l_dark, 'b', linewidth=1.5, label='dark')
ax0.plot(env_light, l_medium, 'g', linewidth=1.5, label='medium')
ax0.plot(env_light, l_light, 'r', linewidth=1.5, label='light')
ax0.set_title('Environment Light')
ax0.legend()

ax1.plot(changing_rate, r_ns, 'b', linewidth=1.5, label='negative small')
ax1.plot(changing_rate, r_zero, 'g', linewidth=1.5, label='zero')
ax1.plot(changing_rate, r_ps, 'r', linewidth=1.5, label='positive small')
ax1.set_title('Changing Rate')
ax1.legend()


ax2.plot(dimmer_control, dm_vs, 'b', linewidth=1.5, label='very small')
ax2.plot(dimmer_control, dm_s, 'g', linewidth=1.5, label='small')
ax2.plot(dimmer_control, dm_b, 'r', linewidth=1.5, label='big')
ax2.plot(dimmer_control, dm_vb, 'y', linewidth=1.5, label='very big')
ax2.set_title('Dimmer control')
ax2.legend()

# Turn off top/right axes
for ax in (ax0, ax1, ax2):
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()
```

(continues on next page)

(continued from previous page)

```
plt.show()
```




nbsphinx-code-borderwhite

The rules base of the system is composed of the following rules:

- if env_light is dark and change_rate is pos_small then dimmer_ctrl is big
- if env_light is dark and change_rate is zero then dimmer_ctrl is big
- if env_light is dark and change_rate is neg_small then dimmer_ctrl is very_big
- if env_light is medium and change_rate is pos_small then dimmer_ctrl is small
- if env_light is medium and change_rate is zero then dimmer_ctrl is big
- if env_light is medium and change_rate is neg_small then dimmer_ctrl is big
- if env_light is light and change_rate is pos_small then dimmer_ctrl is very_small
- if env_light is light and change_rate is zero then dimmer_ctrl is small
- if env_light is light and change_rate is neg_small then dimmer_ctrl is big

By running the following two cells QFIE is properly initialized

[3]:
```
rules = ['if env_light is dark and change_rate is pos_small then dimmer_ctrl is big',
         'if env_light is dark and change_rate is zero then dimmer_ctrl is big',
         'if env_light is dark and change_rate is neg_small then dimmer_ctrl is very_big
↪',
         'if env_light is medium and change_rate is pos_small then dimmer_ctrl is small',
         'if env_light is medium and change_rate is zero then dimmer_ctrl is big',
         'if env_light is medium and change_rate is neg_small then dimmer_ctrl is big',
         'if env_light is light and change_rate is pos_small then dimmer_ctrl is very_
↪small',
         'if env_light is light and change_rate is zero then dimmer_ctrl is small',
         'if env_light is light and change_rate is neg_small then dimmer_ctrl is big']
```

[4]:
```
qfie = FE.QuantumFuzzyEngine(verbose=False)
qfie.input_variable(name='env_light', range=env_light)
qfie.input_variable(name='change_rate', range=changing_rate)
qfie.output_variable(name='dimmer_ctrl', range=dimmer_control)

qfie.add_input_fuzzysets(var_name='env_light', set_names=['dark', 'medium', 'light'],␣
↪sets=[l_dark, l_medium, l_light])
qfie.add_input_fuzzysets(var_name='change_rate', set_names=['neg_small', 'zero', 'pos_
↪small'], sets=[r_ns, r_zero, r_ps])
qfie.add_output_fuzzysets(var_name='dimmer_ctrl', set_names=['very_small', 'small', 'big
↪', 'very_big'],sets=[dm_vs, dm_s, dm_b, dm_vb])
qfie.set_rules(rules)
```

To obtain the control surface of QFIE, it is used to fire the rules in 121 different input combinations. In detail, for 11 values of env_light 11 different values of change_rate are considered. The 121 outputs obtained are collected in a list which will be used at the end of the notebook to plot the control surface.

[5]:
```
el_list, cr_list = [t for t in range(120, 230, 10)], [s for s in range(-10, 12, 2)]
f_quantum = []
for el in el_list:
    for cr in cr_list:
        qfie.build_inference_qc({'env_light':el, 'change_rate':cr}, draw_qc=False)
        f_quantum.append(qfie.execute(n_shots=1000, plot_histo=False)[0])
```

---

Then, the classical Mamdani system is implemented by using the python library Sci-kit Fuzzy. The following cells initializes the system as required from the library

[6]:
```
from skfuzzy import control as ctrl
```

[7]:
```
env_l = ctrl.Antecedent(env_light, 'env_light')
cng_r = ctrl.Antecedent(changing_rate , 'change_rate')
dim = ctrl.Consequent(dimmer_control, 'dimmer_ctrl')

env_l['dark'] = l_dark
env_l['medium'] = l_medium
env_l['light'] = l_light
```

(continues on next page)

(continued from previous page)

```
cng_r['neg_small'] = r_ns
cng_r['zero'] = r_zero
cng_r['pos_small'] = r_ps

dim['very_small'] = dm_vs
dim['small'] = dm_s
dim['big'] = dm_b
dim['very_big'] = dm_vb


rule1 = ctrl.Rule(env_l['dark'] & cng_r['pos_small'], dim['big'])
rule2 = ctrl.Rule(env_l['dark'] & cng_r['zero'], dim['big'])
rule3 = ctrl.Rule(env_l['dark'] & cng_r['neg_small'], dim['very_big'])
rule4 = ctrl.Rule(env_l['medium'] & cng_r['pos_small'], dim['small'])
rule5 = ctrl.Rule(env_l['medium'] & cng_r['zero'], dim['big'])
rule6 = ctrl.Rule(env_l['medium'] & cng_r['neg_small'], dim['big'])
rule7 = ctrl.Rule(env_l['light'] & cng_r['pos_small'], dim['very_small'])
rule8 = ctrl.Rule(env_l['light'] & cng_r['zero'], dim['small'])
rule9 = ctrl.Rule(env_l['light'] & cng_r['neg_small'], dim['big'])

dim_ctrl = ctrl.ControlSystem([rule1, rule2, rule3, rule4, rule5, rule6, rule7, rule8,␣
↪rule9])
dim_sys = ctrl.ControlSystemSimulation(dim_ctrl)
```

By running the following cell, the same combinations of inputs are passed to the classical control which computes the 121 output. All the output values are stored in a list.

[8]:
```
f_classical = []
for el in el_list:
    for cr in cr_list:
        dim_sys.input['env_light']=el
        dim_sys.input['change_rate']=cr

        dim_sys.compute()
        f_classical.append(dim_sys.output['dimmer_ctrl'])
```

At this point, both the QFIE and Classical system' control surfaces can be plotted. Firstly, the following cell shows the former surface.

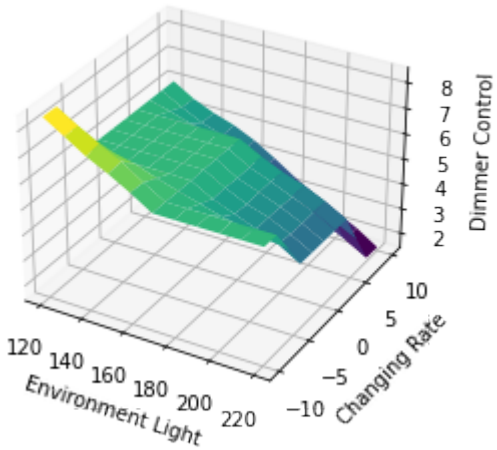
[9]:
```
from mpl_toolkits.mplot3d import Axes3D

X, Y, = el_list, cr_list
plotx,ploty, = np.meshgrid(np.linspace(np.min(X),np.max(X),11), np.linspace(np.min(Y),np.
↪max(Y),11))
Z = np.zeros_like(plotx)
counter = 0
for i in range(len(el_list)):
    for j in range(len(cr_list)):
        Z[i,j]=f_quantum[counter]
        counter+=1
```

(continues on next page)

(continued from previous page)

```
zfig = plt.figure()
ax = plt.axes(projection='3d')

ax.plot_surface(plotx, ploty, Z, rstride=1, cstride=1, cmap='viridis',
                        linewidth=0.4, antialiased=True)
ax.set_xlabel('Environment Light')
ax.set_ylabel('Changing Rate')
ax.set_zlabel('Dimmer Control')
```

[9]:
```
Text(0.5, 0, 'Dimmer Control')
```

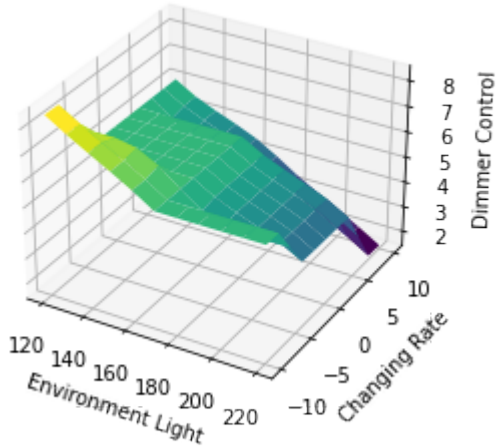



nbsphinx-code-borderwhite

Then, the classical surface is displayed by running the following cell.

[10]:
```
X, Y, = el_list, cr_list
plotx,ploty, = np.meshgrid(np.linspace(np.min(X),np.max(X),11), np.linspace(np.min(Y),np.
↪max(Y),11))
Z = np.zeros_like(plotx)
counter = 0
for i in range(len(el_list)):
    for j in range(len(cr_list)):
        Z[i,j]=f_classical[counter]
        counter+=1

zfig = plt.figure()
ax = plt.axes(projection='3d')

ax.plot_surface(plotx, ploty, Z, rstride=1, cstride=1, cmap='viridis',
                        linewidth=0.4, antialiased=True)
ax.set_xlabel('Environment Light')
ax.set_ylabel('Changing Rate')
ax.set_zlabel('Dimmer Control')
```

[10]:
```
Text(0.5, 0, 'Dimmer Control')
```




nbsphinx-code-borderwhite

The two surfaces show how QFIE is able to control in a very similar way to a classical Mamdani fuzzy control the lighting control system, despite the computational advantage in firing the fuzzy rules in a parallel way.

## 3.4 QFIE 1.1.0 Release

This notebook aims to illustrate the main changes applied to the QFIE package. As an example the lighting system control introduced in the previous notebook will be considered. Firstly, let us import the required libraries.

[1]:
```
import numpy as np
import skfuzzy as fuzz
import matplotlib.pyplot as plt
import QFIE.FuzzyEngines as FE
```

The system is a MISO composed of two input variables and one output variable. In detail:

Inputs:

- Environment Light (env_light): three linguistic terms are considered: 'Dark', 'Medium', 'Light'
- Changing Rate (change_rate): three lingiuistic terms are considered: 'Negative Small', 'Zero', 'Positive Small'

Output:

- Dimmer Control (dimmer_ctrl): four linguistic terms are considered: 'Very Small', 'Small', 'Big', 'Very Big'

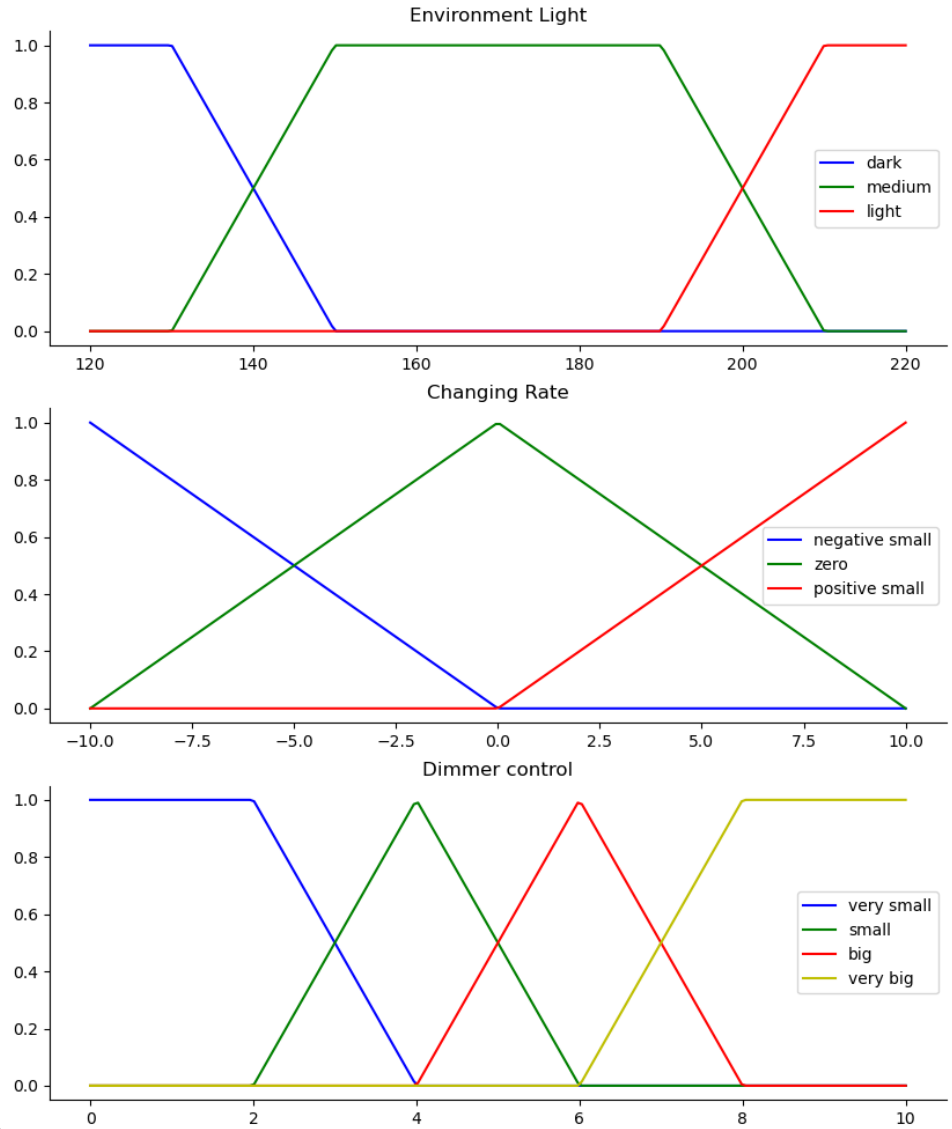
By running the following cell, the fuzzy sets used are graphically shown.

[2]:
```
env_light = np.linspace(120, 220, 200)
changing_rate = np.linspace(-10, 10, 200)
dimmer_control = np.linspace(0, 10, 200)



l_dark = fuzz.trapmf(env_light, [120,120,130,150])
l_medium = fuzz.trapmf(env_light, [130,  150, 190,210])
l_light = fuzz.trapmf(env_light, [190,  210, 220, 220])

r_ns = fuzz.trimf(changing_rate, [-10,-10,0])
```

(continues on next page)

(continued from previous page)

```
r_zero = fuzz.trimf(changing_rate, [-10,0,10])
r_ps = fuzz.trimf(changing_rate, [0,10,10])

dm_vs = fuzz.trapmf(dimmer_control, [0,0,2,4])
dm_s = fuzz.trimf(dimmer_control, [2,4,6])
dm_b = fuzz.trimf(dimmer_control, [4,6,8])
dm_vb = fuzz.trapmf(dimmer_control, [6,8,10,10])




# Visualize these universes and membership functions
fig, (ax0, ax1, ax2) = plt.subplots(nrows=3, figsize=(10, 12))


ax0.plot(env_light, l_dark, 'b', linewidth=1.5, label='dark')
ax0.plot(env_light, l_medium, 'g', linewidth=1.5, label='medium')
ax0.plot(env_light, l_light, 'r', linewidth=1.5, label='light')
ax0.set_title('Environment Light')
ax0.legend()

ax1.plot(changing_rate, r_ns, 'b', linewidth=1.5, label='negative small')
ax1.plot(changing_rate, r_zero, 'g', linewidth=1.5, label='zero')
ax1.plot(changing_rate, r_ps, 'r', linewidth=1.5, label='positive small')
ax1.set_title('Changing Rate')
ax1.legend()


ax2.plot(dimmer_control, dm_vs, 'b', linewidth=1.5, label='very small')
ax2.plot(dimmer_control, dm_s, 'g', linewidth=1.5, label='small')
ax2.plot(dimmer_control, dm_b, 'r', linewidth=1.5, label='big')
ax2.plot(dimmer_control, dm_vb, 'y', linewidth=1.5, label='very big')
ax2.set_title('Dimmer control')
ax2.legend()

# Turn off top/right axes
for ax in (ax0, ax1, ax2):
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()


plt.show()
```




nbsphinx-code-borderwhite

The rules base of the system is composed of the following rules:

- if env_light is dark and change_rate is pos_small then dimmer_ctrl is big
- if env_light is dark and change_rate is zero then dimmer_ctrl is big
- if env_light is dark and change_rate is neg_small then dimmer_ctrl is very_big
- if env_light is medium and change_rate is pos_small then dimmer_ctrl is small
- if env_light is medium and change_rate is zero then dimmer_ctrl is big
- if env_light is medium and change_rate is neg_small then dimmer_ctrl is big
- if env_light is light and change_rate is pos_small then dimmer_ctrl is very_small
- if env_light is light and change_rate is zero then dimmer_ctrl is small
- if env_light is light and change_rate is neg_small then dimmer_ctrl is big

By running the following two cells QFIE is properly initialized

[3]:
```
rules = ['if env_light is dark and change_rate is pos_small then dimmer_ctrl is big',
         'if env_light is dark and change_rate is zero then dimmer_ctrl is big',
         'if env_light is dark and change_rate is neg_small then dimmer_ctrl is very_big
↪',
         'if env_light is medium and change_rate is pos_small then dimmer_ctrl is small',
         'if env_light is medium and change_rate is zero then dimmer_ctrl is big',
         'if env_light is medium and change_rate is neg_small then dimmer_ctrl is big',
         'if env_light is light and change_rate is pos_small then dimmer_ctrl is very_
↪small',
         'if env_light is light and change_rate is zero then dimmer_ctrl is small',
         'if env_light is light and change_rate is neg_small then dimmer_ctrl is big']
```

Firstly, let us initialize the `QuantumFuzzyEngine` class defined in FE. By setting `verbose=True` we will get information about the inference during the computation. Set `verbose=False` if this information is unuseful in your application.

[4]:
```
qfie = FE.QuantumFuzzyEngine(verbose=True)
```

The way in which input and output variables are passed to QFIE remains unchanged:

[5]:
```
qfie.input_variable(name='env_light', range=env_light)
qfie.input_variable(name='change_rate', range=changing_rate)
qfie.output_variable(name='dimmer_ctrl', range=dimmer_control)

qfie.add_input_fuzzysets(var_name='env_light', set_names=['dark', 'medium', 'light'],␣
↪sets=[l_dark, l_medium, l_light])
qfie.add_input_fuzzysets(var_name='change_rate', set_names=['neg_small', 'zero', 'pos_
↪small'], sets=[r_ns, r_zero, r_ps])
qfie.add_output_fuzzysets(var_name='dimmer_ctrl', set_names=['very_small', 'small', 'big
↪', 'very_big'],sets=[dm_vs, dm_s, dm_b, dm_vb])
qfie.set_rules(rules)
```

The main difference in the new version of `QuantumFuzzyEngine` is the fact that any backend can be used to compute the quantum circuit. Now the backend is passed as argument of the execute method as Qiskit Backend object, instead that as string.

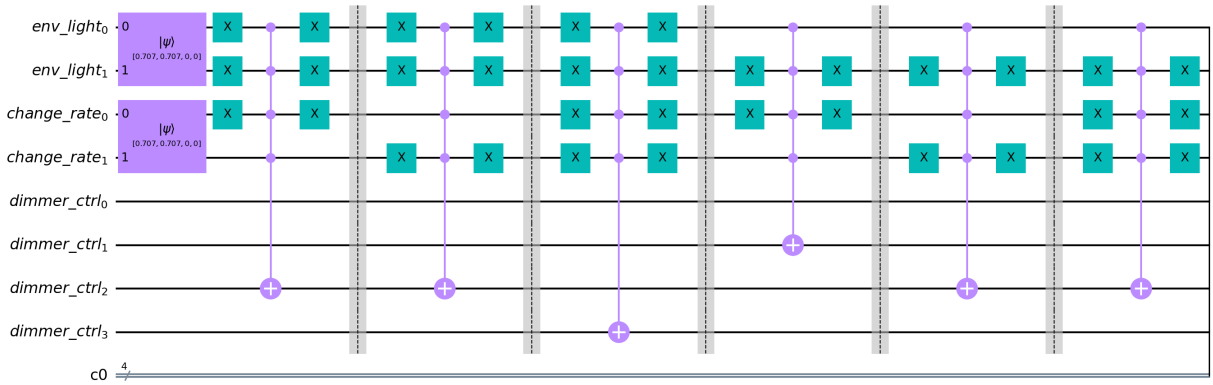
Let see an example by running an inference process.

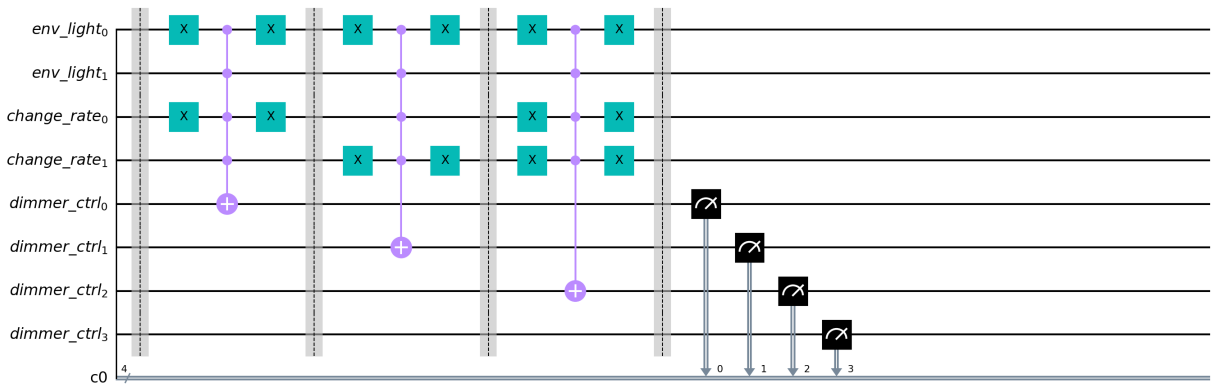
[6]:
```
el, cr = 140, -5
qfie.build_inference_qc({'env_light':el, 'change_rate':cr}, draw_qc=True, filename='qc_
↪plot.png')
```

```
{'env_light': 140, 'change_rate': -5}
Input values  {'env_light': [0.5, 0.5, 0.0], 'change_rate': [0.5, 0.5, 0.0]}
```

```
/Users/rschiattarella/Projects/QFIE_Package/src/QFIE/FuzzyEngines.py:277: UserWarning:␣
↪Matplotlib is currently using module://matplotlib_inline.backend_inline, which is a␣
↪non-GUI backend, so cannot show the figure.
  self.qc["full_circuit"].draw("mpl", filename=kwargs["filename"]).show()
```

By specifying the filename in the build_inference_qc method a png file containing the quantum circuit plot is created. This is particularly useful when the size of the quantum circuit is big and visualization problems during the execution arise. If filename isn't specified the quantum circuit is shown but not saved.




At this point the generated QC can be executed with the `execute` method. If not backend is specified then the qc is running by using the Qiskit local QasmSimulator. Any backend can be specified and used. As an example the following execution is carried out by using the noisy FakeMumbai() simulator.

[7]:
```
from qiskit.providers.fake_provider import FakeMumbai
qfie.execute(n_shots=1000,  backend=FakeMumbai())[0]
```

```
Running qc full_circuit on fake_mumbai
Output Counts {'0001': 0.09833333333333333, '0010': 0.176, '0100': 0.32766666666666666,
↪'1000': 0.191}
```

[7]:
```
5.613499478403967
```

### 3.4.1 Distributed-QFIE (D-QFIE)

Another important change in the new release of QFIE package is the possibility of using the Distributed QFIE algorithm introduced in the paper submitted as proceeding for FUZZ-IEEE 2023' Distributing Fuzzy Inference Engines on Quantum Computers - Acampora Giovanni, Massa Alfredo, Schiattarella Roberto, Vitiello Autilia'.

Here the inference process is distributed over more quantum circuits, one for each possible consequent of the system.

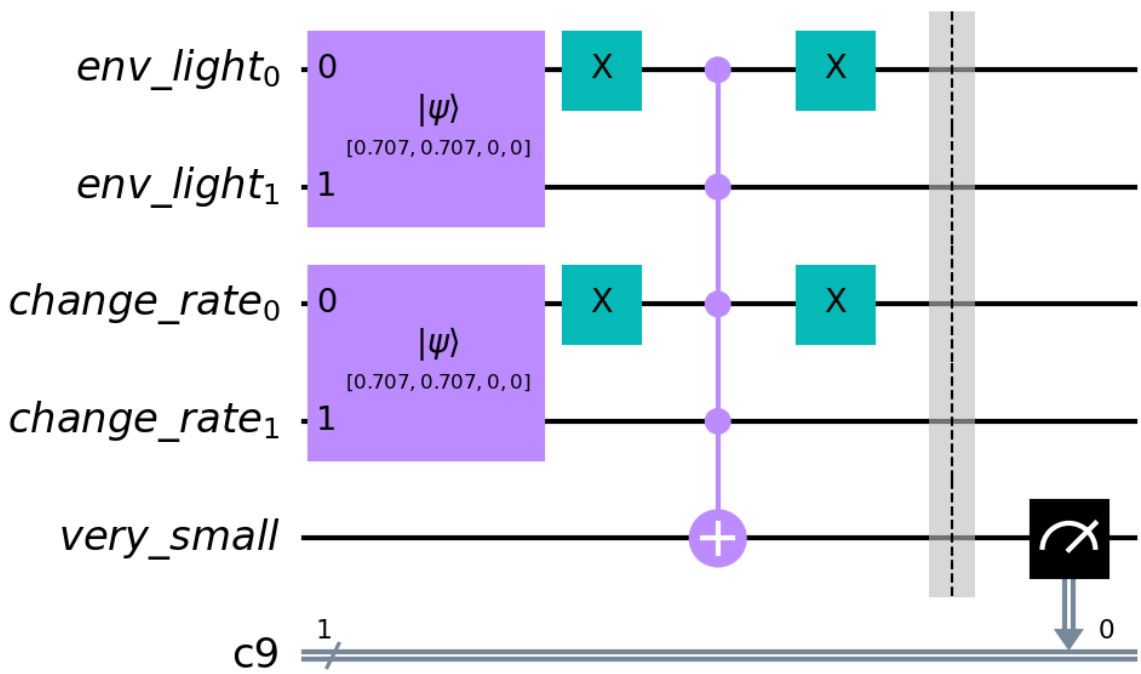
This part of the notebook is dedicated to see how D-QFIE can be used as part of the QFIE package.

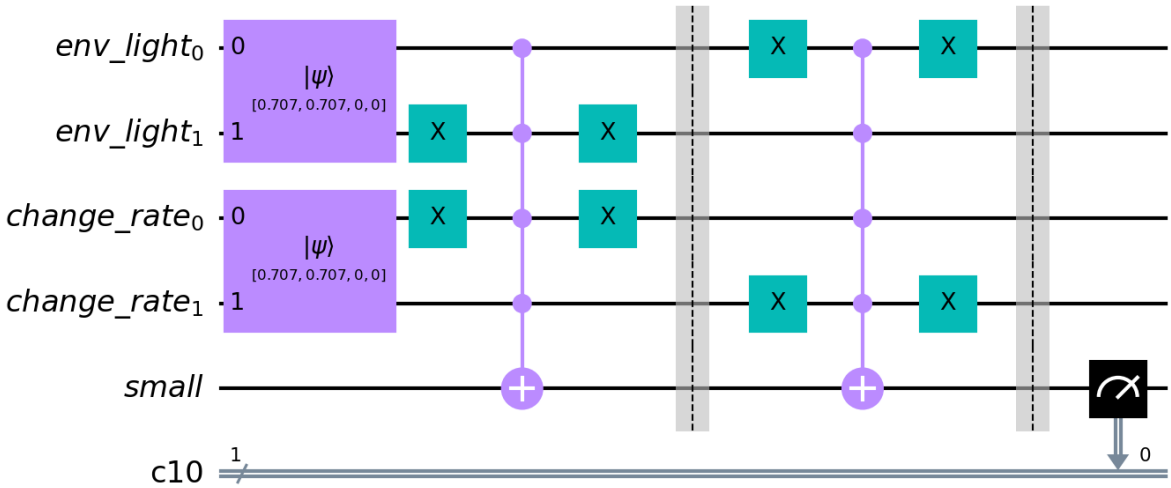
In this particular control system the output variable is composed of four linguistic terms, therefore four quantum circuits will be created. To activate the D-QFIE inference process use the related flag in build_inference_qc method as follows:

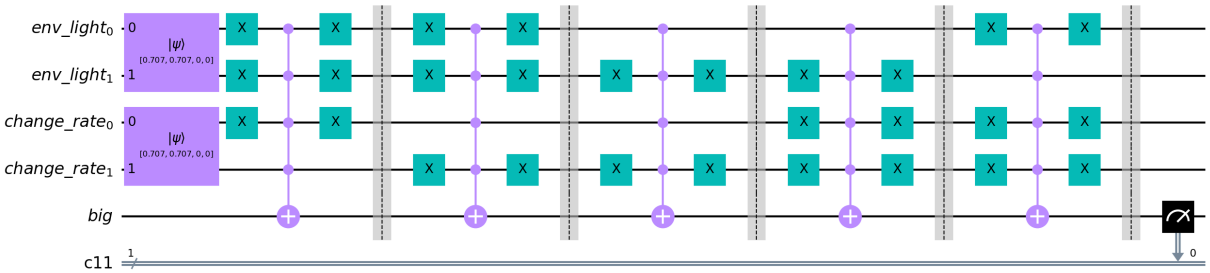
[11]:
```
qfie = FE.QuantumFuzzyEngine(verbose=True)
qfie.input_variable(name='env_light', range=env_light)
qfie.input_variable(name='change_rate', range=changing_rate)
qfie.output_variable(name='dimmer_ctrl', range=dimmer_control)

qfie.add_input_fuzzysets(var_name='env_light', set_names=['dark', 'medium', 'light'],␣
↪sets=[l_dark, l_medium, l_light])
qfie.add_input_fuzzysets(var_name='change_rate', set_names=['neg_small', 'zero', 'pos_
↪small'], sets=[r_ns, r_zero, r_ps])
qfie.add_output_fuzzysets(var_name='dimmer_ctrl', set_names=['very_small', 'small', 'big
↪', 'very_big'],sets=[dm_vs, dm_s, dm_b, dm_vb])
qfie.set_rules(rules)


qfie.build_inference_qc({'env_light':el, 'change_rate':cr}, distributed=True, draw_
↪qc=True, filename='d_qc.png')
```

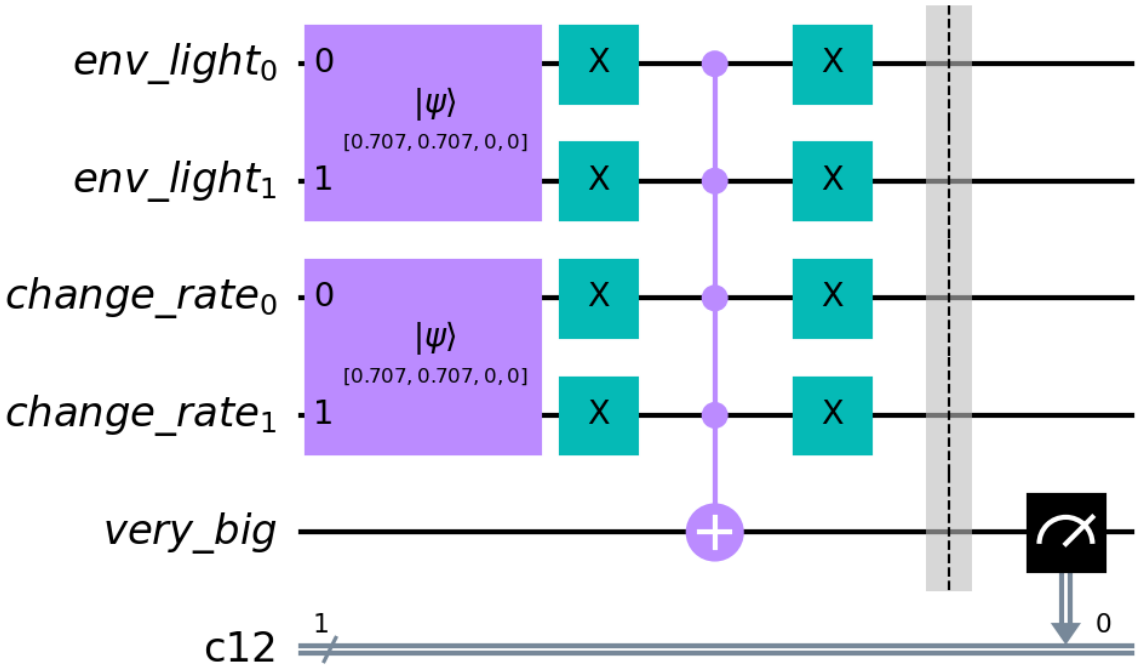
```
{'env_light': 140, 'change_rate': -5}
Input values  {'env_light': [0.5, 0.5, 0.0], 'change_rate': [0.5, 0.5, 0.0]}
```

By specifying the filename four quantum circuit figures will be saved at the path indicated.






By running the execute method the quantum circuits will be executed and the outputs aggregated in order to obtain the

crisp output value. If not backend is specified, then all the quantum circuits will be simulated by means of the noiseless QasmSimulator.

[12]:
```
qfie.execute(n_shots=1000)[0]
```

```
Running qc very_small on qasm_simulator
Running qc small on qasm_simulator
Running qc big on qasm_simulator
Running qc very_big on qasm_simulator
Output Counts {'0001': 0.0, '0010': 0.0, '0100': 0.7519762845849802, '1000': 0.
↪24802371541501977}
```

[12]:
```
6.657587913604732
```

Alternatively, a list of backend can be specified and the quantum circuits will be computed on them.

[13]:
```
from qiskit.providers.fake_provider import FakeMumbai, FakeMontreal
qfie.execute(n_shots=1000, backend=[FakeMontreal(), FakeMumbai()])[0]
```

```
Running qc very_small on fake_montreal
Running qc small on fake_mumbai
Running qc big on fake_montreal
Running qc very_big on fake_mumbai
Output Counts {'0001': 0.09771309771309772, '0010': 0.3277893277893278, '0100': 0.
↪35135135135135137, '1000': 0.22314622314622315}
```

[13]:
```
5.468974659317126
```

The differences in output computation are related to the noise affecting the circuits simulations by using IBM Fake devices.

The usage of D-QFIE is strongly suggested when noisy devices are used to compute QFIE.

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## q

# INDEX