NN EX6

January 12, 2023

S.M. Erfan Moosavi Monazzah

TABLE OF CONTENTS

1) Fuzzy Control

Cell 1: Fuzzy Control Design Steps

i) Defining input and output variables and define fuzzy linguistic variables.

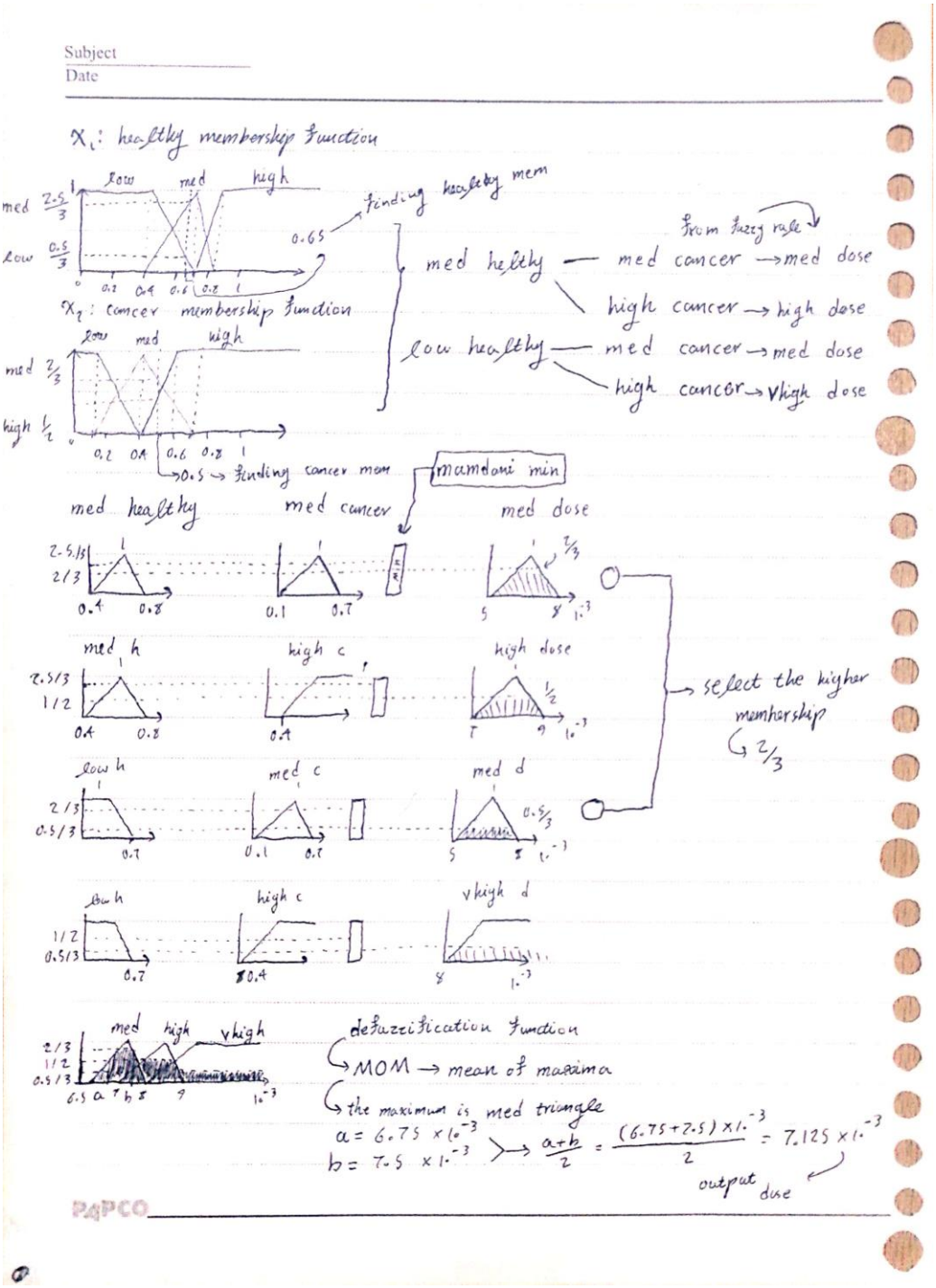
ii) Define fuzzification methods for each input and output variable. This method converts numerical inputs to fuzzy terms.

iii) Extract the rules by asking them from experts or learning them.

iv) Designing the inference engine

v) Define defuzzification methods: these methods convert fuzzy terms back to numerical values.

Cell 2: X1 and x2 membership functions:



Cell 3: Solution in image above.

2) Fuzzy Controller + Python <q2.ipynb>

Cell 1: Here we imported some libraries.

Cell 2: Defining input and output fuzzy terms along with their universes.

```
(1)  x1 = ctrl.Antecedent(np.linspace(0, 1, 1000),
   'healthy') # 1000 points between 0 to 1 inclusive
(2)  x2 = ctrl.Antecedent(np.linspace(0, 1, 1000), 'cancer')
   # 1000 points between 0 to 1 inclusive
(3)  y = ctrl.Consequent(np.linspace(0.004, 0.01, 1000),
   'dose') # 1000 points between 0.004 to 0.01 inclusive
```

Cell 3: Defining membership functions of healthy terms

Cell 4: Defining membership functions of cancer terms

Cell 5: Defining membership functions of dose terms

```
(4)  x1['low'] = fuzz.trapmf(x1.universe, [0, 0, 0.4, 0.7])
   # trapezoidal membership
(5)  x1['med'] = fuzz.trimf(x1.universe, [0.4, 0.7, 0.85]) #
   triangular membership
(6)  x1['high'] = fuzz.trapmf(x1.universe, [0.7, 0.85, 1,
   1]) # trapezoidal membership
(7)
(8)  x1.view()
(9)
(10) x2['low'] = fuzz.trapmf(x2.universe, [0, 0, 0.1, 0.4])
   # trapezoidal membership
(11) x2['med'] = fuzz.trimf(x2.universe, [0.1, 0.4, 0.7]) #
   triangular membership
(12) x2['high'] = fuzz.trapmf(x2.universe, [0.4, 0.6, 1, 1])
   # trapezoidal membership
(13)
(14) x2.view()
(15)
(16) y['vlow'] = fuzz.trapmf(y.universe, [0, 0, 5.5e-3,
   6.5e-3]) # trapezoidal membership
(17) y['low'] = fuzz.trimf(y.universe, [5.5e-3, 6e-3, 7e-3])
   # trapezoidal membership
(18) y['med'] = fuzz.trimf(y.universe, [6.5e-3, 7e-3, 8e-3])
   # triangular membership
```

```
(19) y['high'] = fuzz.trimf(y.universe, [7e-3, 8e-3, 9e-3])
   # trapezoidal membership
(20) y['vhigh'] = fuzz.trapmf(y.universe, [8e-3, 9e-3, 10e-
   3, 10e-3]) # trapezoidal membership
(21)
(22) y.view()
```

Cell 6: In this cell we created our list of rules based on the rule matrix given by the question.

```
(23) list_rules = []
(24) list_rules.append(ctrl.Rule(x1['low'] & x2['low'],
   y['low']))
(25) list_rules.append(ctrl.Rule(x1['low'] & x2['med'],
   y['med']))
(26) list_rules.append(ctrl.Rule(x1['low'] & x2['high'],
   y['vhigh']))
(27) list_rules.append(ctrl.Rule(x1['med']& x2['low'],
   y['low']))
(28) list_rules.append(ctrl.Rule(x1['med']& x2['med'],
   y['med']))
(29) list_rules.append(ctrl.Rule(x1['med']& x2['high'],
   y['high']))
(30) list_rules.append(ctrl.Rule(x1['high'] & x2['low'],
   y['vlow']))
(31) list_rules.append(ctrl.Rule(x1['high'] & x2['med'],
   y['low']))
(32) list_rules.append(ctrl.Rule(x1['high'] & x2['high'],
   y['med']))
```

Cell 7: Now we create the fuzzy controller based on rule list we created out of input and output terms.

```
(33) dosage_ctrl = ctrl.ControlSystem(list_rules)
```

Cell 8: Now based on the above controller we create a system simulation to pass testing inputs and see the output

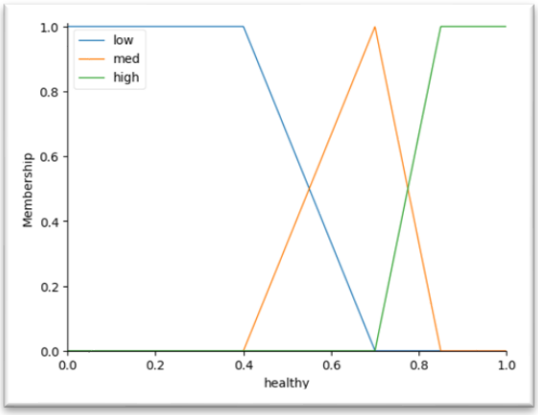
```
(34) dosage = ctrl.ControlSystemSimulation(dosage_ctrl)
```

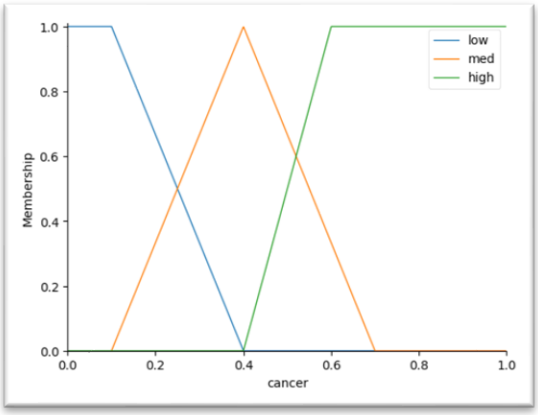
Cell 9: With having simulalor at hand, we can pass in inputs (in our case healthy and cancer cells density) and see the computation graphs and result

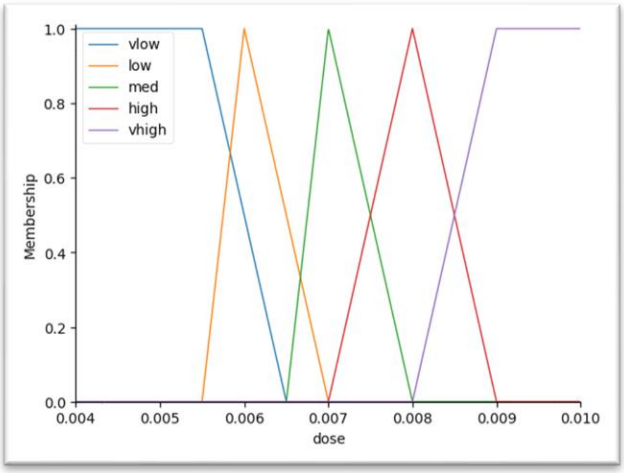
```
(35) dosage.input['healthy'] = 0.65
(36) dosage.input['cancer'] = 0.5
(37) dosage.compute()
(38)
(39) print (f'Appropriate dosage: {dosage.output["dose"]}')
(40) y.view(sim=dosage)
```

Cell 10: Now the second input pair, since the density of cancer cells got bigger and healthy cells reduced, we expect the drug dosage to increase

```
(41) dosage.input['healthy'] = 0.5
(42) dosage.input['cancer'] = 0.65
(43) dosage.compute()
(44)
(45) print (f'Appropriate dosage: {dosage.output["dose"]}')
(46) y.view(sim=dosage)
```







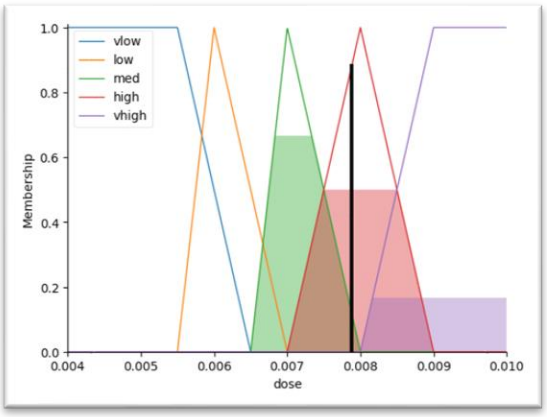


Figure 1: Healthy 0.65, Cancer 0.5, Dose 0.0079

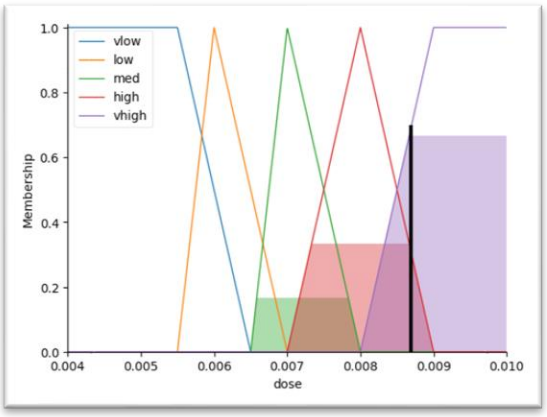
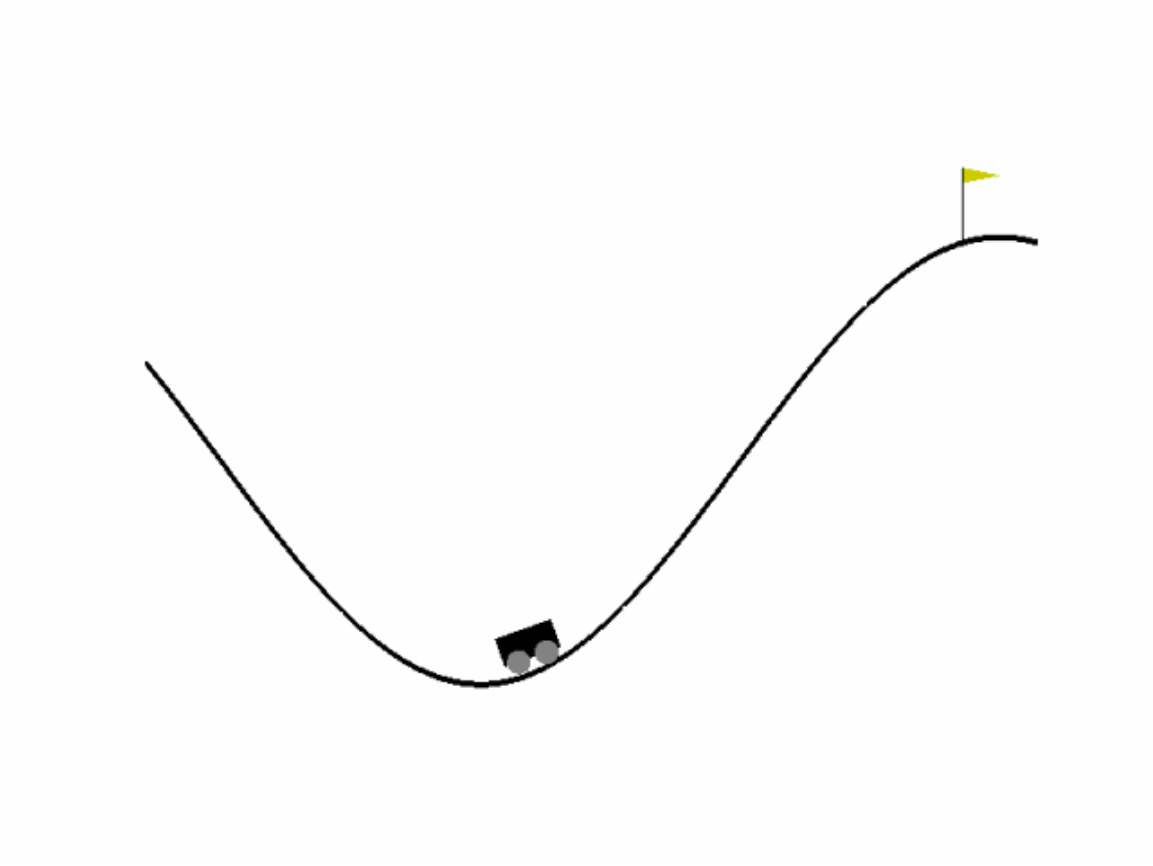


Figure 2: Healthy 0.5, Cancer 0.65, Dose 0.0087

3) Car + Fuzzy <q3.ipynb> (Animations are working in Word Doc)



First of all I installed the required libs, then I used the original GYM documentations to set up and run the simulation env.

Basic Usage - Gym Documentation (gymlibrary.dev)

Also I read the following article too:

Driving Up A Mountain - A Random Walk (jfking50.github.io)

For this problem we have two input variables and one output:

Inputs: Car position and velocity

Output: Car acceleration value

I have created two fuzzy controller for this problem, based on the following strategy:

If you are on the left side of the curve and going left, keep accelerating left otherwise accelerate in right direction.

If you are on the right side of the curve and going right, keep accelerating right, otherwise accelerate in left direction.

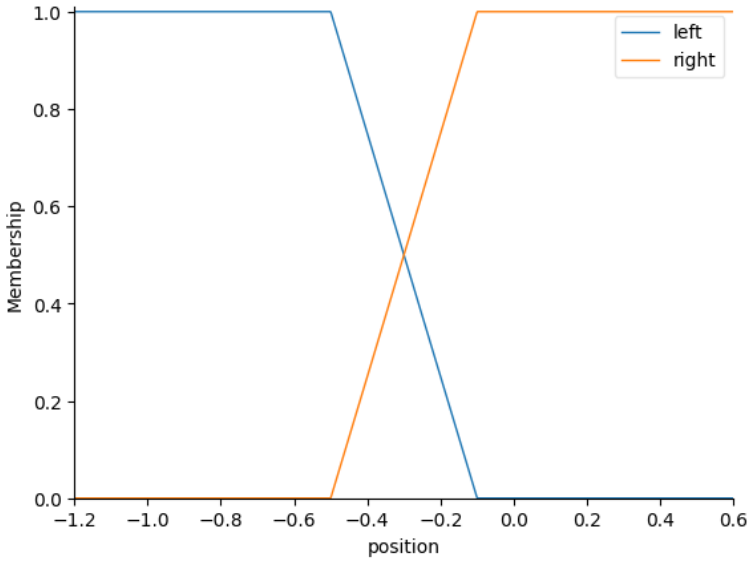
As you can see, the output only depends on the second input (velocity) and this strategy works. So for sake of compeleteness first Ive created a controller with two inputs (it does not both inputs though). Then a simpler controller with only one input.

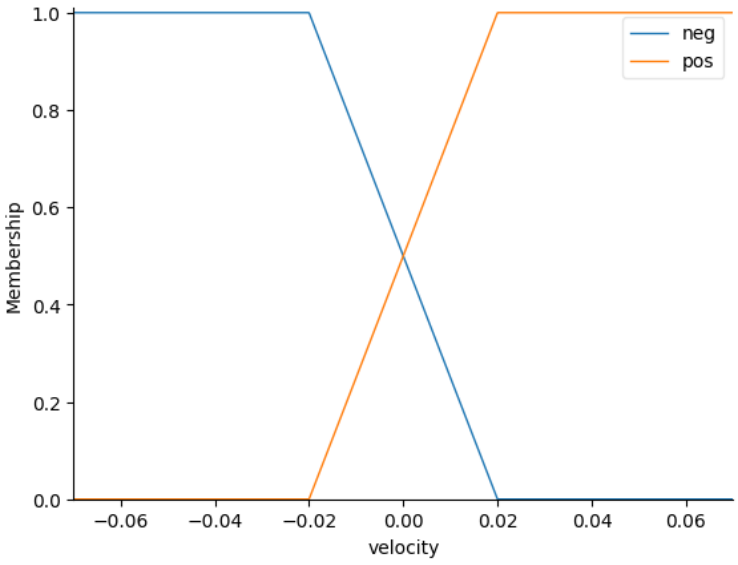
Ok, lets start with the first two-input controller:

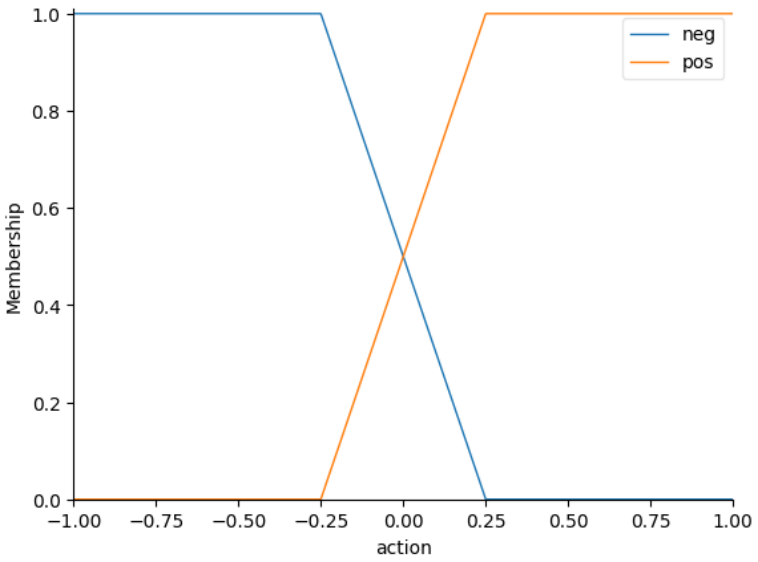
The rule base for this controller is the same as above and fuzzy terms are defined as follows:

```
(47) x1 = ctrl.Antecedent(np.linspace(-1.2, 0.6, 1000),
   'position') # 1000 points between 0 to 1 inclusive
(48) x2 = ctrl.Antecedent(np.linspace(-0.07, 0.07, 1000),
   'velocity') # 1000 points between 0 to 1 inclusive
(49) y = ctrl.Consequent(np.linspace(-1, 1, 1000), 'action')
   # 1000 points between 0.004 to 0.01 inclusive
(50) x1['left'] = fuzz.trapmf(x1.universe, [-1.2, -1.2, -
   0.5, -0.1]) # trapezoidal membership
(51) x1['right'] = fuzz.trapmf(x1.universe, [-0.5, -0.1,
   0.6, 0.6]) # trapezoidal membership
(52)
(53) x1.view()
(54) x2['neg'] = fuzz.trapmf(x2.universe, [-0.07, -0.07, -
   0.02, 0.02]) # trapezoidal membership
(55) x2['pos'] = fuzz.trapmf(x2.universe, [-0.02, 0.02,
   0.07, 0.07]) # trapezoidal membership
(56)
(57) x2.view()
(58) y['neg'] = fuzz.trapmf(y.universe, [-1, -1, -0.25,
   0.25]) # trapezoidal membership
(59) y['pos'] = fuzz.trapmf(y.universe, [-0.25, 0.25, 1, 1])
   # trapezoidal membership
```

```
(60)
(61) y.view()
(62) list_rules = []
(63) list_rules.append(ctrl.Rule(x1['left'] & x2['neg'],
   y['neg']))
(64) list_rules.append(ctrl.Rule(x1['left'] & x2['pos'],
   y['pos']))
(65) list_rules.append(ctrl.Rule(x1['right'] & x2['neg'],
   y['neg']))
(66) list_rules.append(ctrl.Rule(x1['right']& x2['pos'],
   y['pos']))
```







Now with our fuzzy controller at hand, ive created a function to run the game simulation, this function consists of a main loop which controls the maximum number of steps that can be done by agent and in each iteration first we observe the env, then based on our observation we pass two inputs to our fuzzy controller: position and velocity, the fuzzy controller based on these two inputs compute the suitable acceleration which were going to use as our action. Also we have added a control condition to end the loop we reached top of the mountain.

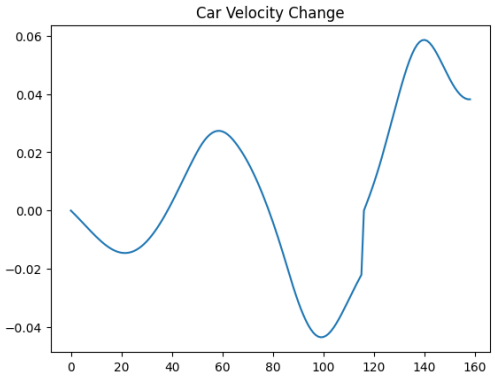
```
(67) def print_obs(obs, precision = 2):
(68)     print(f'Position({round(obs[0], precision)}) |
   Velocity({round(obs[1], precision)})')
(69)
(70)
(71) def run_simulation(max_steps = 500):
(72)     ls_frames = []
(73)     ls_obs = []
(74)     ls_actions = []
(75)     ls_rewards = []
(76)
(77)     env = gym.make("MountainCarContinuous-v0")
(78)
(79)     env.seed(RAND_SEED)
(80)     np.random.seed(RAND_SEED)
(81)
```

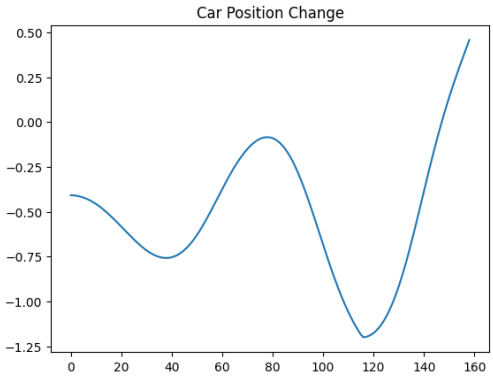
```
(82)     obs = env.reset()
(83)     ls_obs.append(obs)
(84)
(85)     ls_frames.append(env.render('rgb_array'))
(86)
(87)     for i in range(max_steps):
(88)         # action = 1 if i%2==0 else -1
(89)         # action = -1
(90)         # action = 1 if obs[0] < -0.21 else -1
(91)
(92)         car.input['position'] = obs[0]
(93)         car.input['velocity'] = obs[1]
(94)         car.compute()
(95)         a = car.output["action"]
(96)
(97)
(98)         action = np.array([a])
(99)         ls_actions.append(action)
(100)
(101)        obs, reward, done, info = env.step(action)
(102)        ls_obs.append(obs)
(103)        ls_rewards.append(reward)
(104)
(105)        ls_frames.append(env.render('rgb_array'))
(106)
(107)        if done:
(108)            break
(109)
(110)    env.close()
(111)    return ls_frames, ls_obs, ls_actions, ls_rewards, i
```

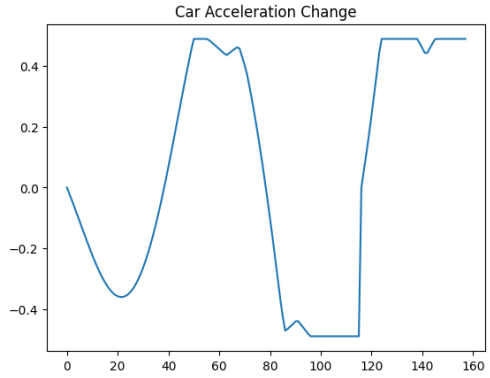
lets see the simulation results:

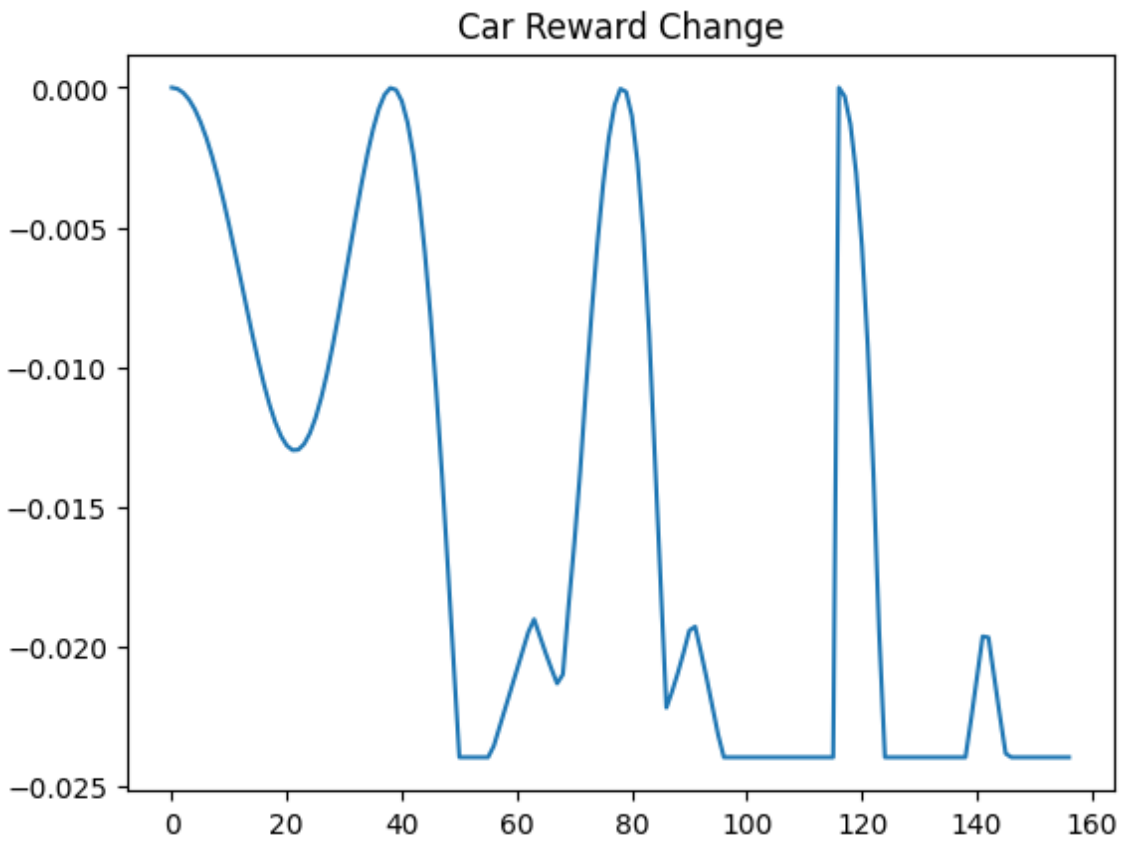
```
Simulation Compeleted: Took 157 steps

Car Started from Position = -0.40813976049094425

Car Goal Position >= 0.45

Car Reached Position = 0.4573492319314634
```

After the simulation we plotted some env variables:









In the last plot, we omitted the final reward (100) because it was too big which would have ruined our plot.

Reward Plot Analysis: Since our car is going back and forth we see a similar pattern in reward plot, our car acceleration is toward the left side at the beginning, so the plot shows a drop in reward then after about 20 iterations our car acceleration is positive toward right side and our car goes to right, so the reward begins to increase. After 40 itererations we reached maximum acceleration but still not on the top of the mountains, so the reward dropped again. After 80 iters we reached a new high, we got a big reward and then we managed to go down and the reward got decreased until at the bottom since our car got so much energy and the acceleration is right ward, we managed to get to the top of the mountain.

We can also solve this problem with a simpler controller consisting of only one input and one output and two rules. The implementation is in the file < CarFuzzyController.ipynb>

All the code is similar to this question except its simpler now, so I wont explain it again.

4) *This algorithm consider each cluster as a fuzzy set and then the clustering task is to assign a membership value to each point in input dataset per each fuzzy set (cluster). There is one more condition, the sum of the memberships of a point across all fuzzy sets (cluster) must be unit. The way this algorithm assigns membersihps to points is by considering the distance between that point and cluster center as a measure of how much membership it has to that cluster.*

Paper

Fuzzy C-Means Clustering (FCM) Algorithm | by Aman Gupta | Geek Culture | Medium

Cell 1: *data = pd.read_csv('credit_card.csv', header=1)*

Cell 2: *extracting BILL_AMT1 to BILL_AMT6 and then summing columns, create a new dataset from LIMIT_BAL and BILL_TOTAL*

```
(112)data['BILL_TOTAL'] = data.loc[:,
   'BILL_AMT1':'BILL_AMT6'].sum(axis=1)
(113)df = data[['LIMIT_BAL', 'BILL_TOTAL']]
```

Cell 3: *3- applying z-score (standardization) on new dataset*

$z = (x - \mu) / \sigma$

μ = *mean*

σ = *standard deviation*

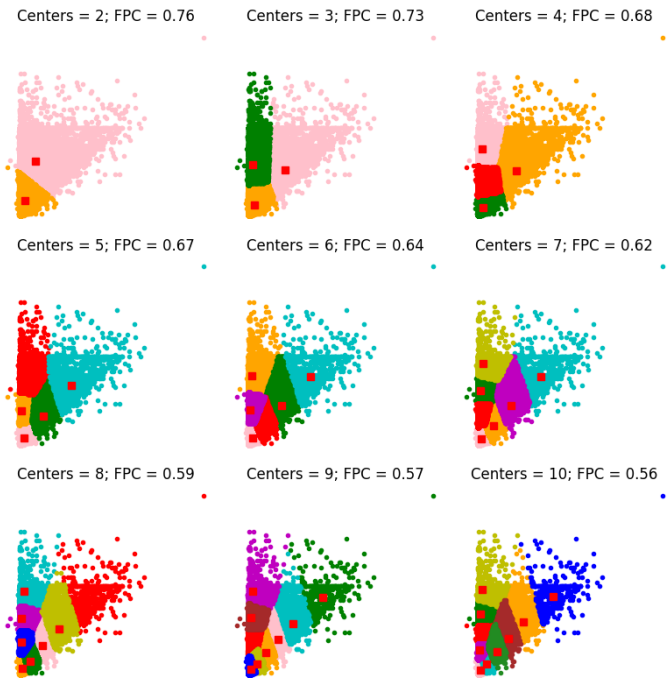
Cell 4: *extracting first and second columns*

```
(114)x1, x2 = df.loc[:, 'BILL_TOTAL'], df.loc[:,
   'LIMIT_BAL']
```

Using FCM to cluster data (2 to 10 clusters)

```
(115)# Set up the loop and plot
(116)fig1, axes1 = plt.subplots(3, 3, figsize=(8, 8))
(117)alldata = np.vstack((x1, x2))
(118)fpcs = []
(119)
(120)for ncenters, ax in enumerate(axes1.reshape(-1), 2):
(121)    cntr, u, u0, d, jm, p, fpc = fuzz.cluster.cmeans(
```

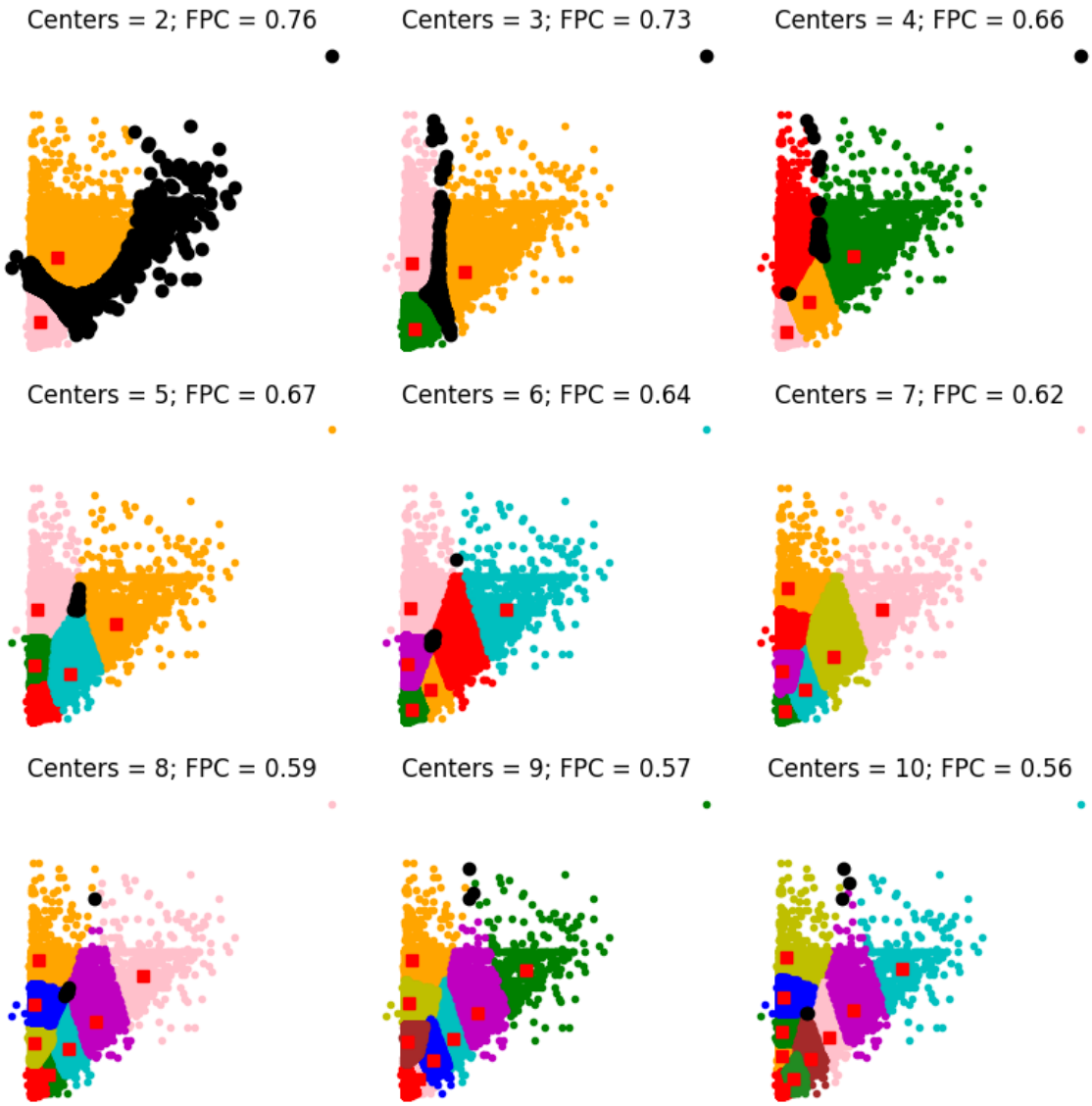
```
(122)        alldata, ncenters, 2, error=0.005,
  maxiter=1000, init=None)
(123)
(124)    # Store fpc values for later
(125)    fpcs.append(fpc)
(126)
(127)    # Plot assigned clusters, for each data point in
  training set
(128)    cluster_membership = np.argmax(u, axis=0)
(129)    for j in range(ncenters):
(130)        ax.plot(x1[cluster_membership == j],
(131)                x2[cluster_membership == j], '.',
  color=colors[j])
(132)
(133)    # Mark the center of each fuzzy cluster
(134)    for pt in cntr:
(135)        ax.plot(pt[0], pt[1], 'rs')
(136)
(137)    ax.set_title('Centers = {0}; FPC =
  {1:.2f}'.format(ncenters, fpc))
(138)    ax.axis('off')
(139)
(140)fig1.tight_layout()
```

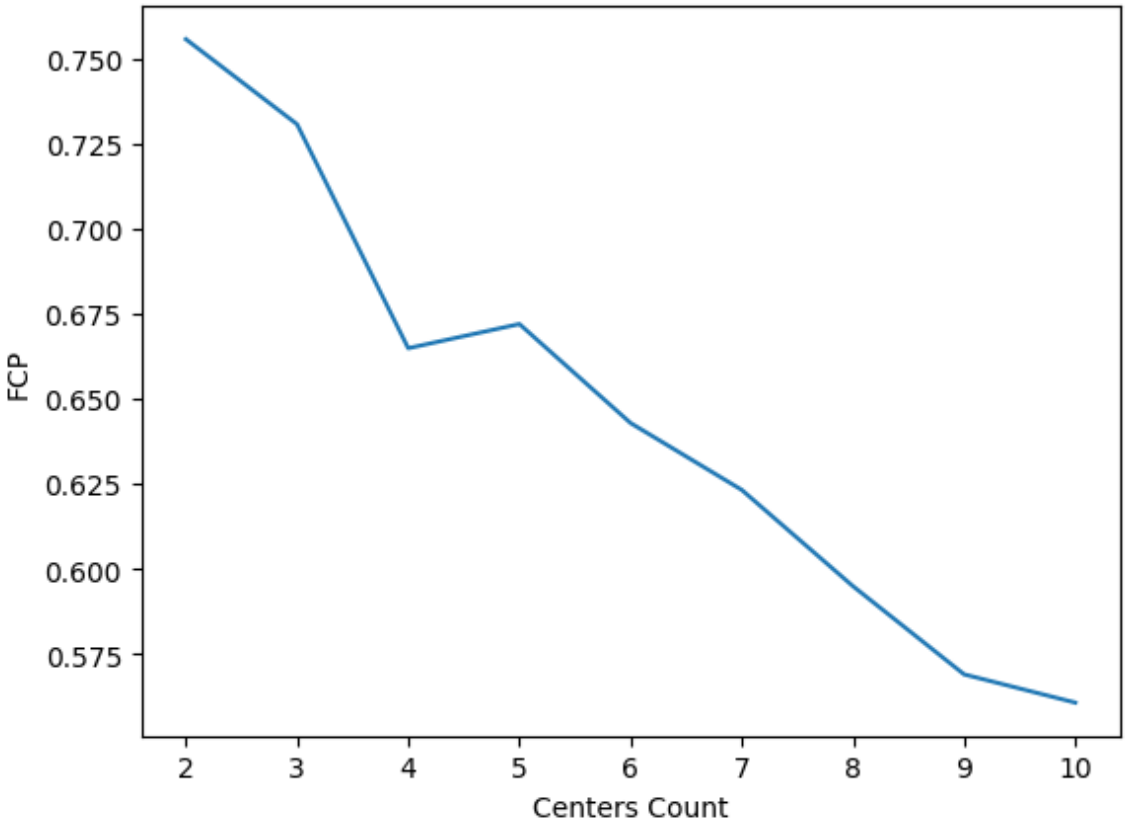


Consider low membership values as suspisous points:
We define low as follows:
*Values below 1 / ncenters * 1.2 + 0.05*
This means the threshold depends on the number of clusters, the we calculated the random clustering algorithm membersihips then add 20% on it and add another 0.05 to it. 20% is usefull since with increasing the number of clusters its reasonable to assume that suspisous point number get a decrease, and the 0.05 is just a small number to ensure a minimum threshold value.
The sus points are in black:

```
(141)
(142)# Set up the loop and plot
(143)fig1, axes1 = plt.subplots(3, 3, figsize=(8, 8))
(144)fpcs = []
(145)alldata = np.vstack((x1, x2))
(146)
(147)for ncenters, ax in enumerate(axes1.reshape(-1), 2):
(148)    cntr, u, u0, d, jm, p, fpc = fuzz.cluster.cmeans(
(149)       alldata , ncenters, 2, error=0.005,
  maxiter=10000, init=None)
(150)
(151)    # Store fpc values for later
(152)    fpcs.append(fpc)
(153)
(154)    tt = 1 / ncenters * 1.2 + 0.05
(155)    cluster_membership = np.argmax(u, axis=0)
(156)    for j in range(ncenters):
(157)        ax.plot(alldata[0, cluster_membership == j],
(158)                alldata[1, cluster_membership == j],
  '.', color=colors[j])
(159)
(160)    list_of_border_patterns = []
(161)    index = 0
(162)    for item in u.T:
(163)        if max(item) < tt:
(164)            ax.plot(alldata[0, index],alldata[1,
  index],'o',color='black')
(165)
```

```
(166)        index += 1
(167)
(168)    # Mark the center of each fuzzy cluster
(169)    for pt in cntr:
(170)        ax.plot(pt[0], pt[1], 'rs')
(171)
(172)    ax.set_title('Centers = {0}; FPC =
   {1:.2f}'.format(ncenters, fpc))
(173)    ax.axis('off')
(174)
(175)fig1.tight_layout()
```





By increasing the number of clusters, the FCP got smaller, so the best number of clusters is 2.

The default payment next month and assigned class are the same for near 48 percent of the times.