Lab exercise 3 - Fuzzy set

Sigve Skaugvoll & Eirik Baug

Task 1 Manual Mandani

Step 1: Fuzzification!

The **value for distance** = 3.7 and intersects `Small` at `0.6` and `Perfect` at `0.1`. The others have value of 0

The **value for delta** = 1.2 and intersects `Stable` at `0.3` and `Growing` at `0.4`. The others have value of 0

Fuzzy logic:
- AND = min
- OR = max
- NOT x = 1-x

Step 2 : Rule evaluation

Values 0.6 0.4 ==>min(0.6, 0.4) = 0.4
IF distance is Small AND delta is Growing THEN action is None

values 0.6 0.3 ==> min(0.6, 0.3) = 0.3
IF distance is Small AND delta is Stable THEN action is SlowDown

values (0.1) (0.4) ==> min(0.1,0.4) = 0.1
IF distance is Perfect AND delta is Growing THEN action is SpeedUp

values 0 (1 - 0.4) (1 - 0) ==> min(0, max(0.6, 1)) = 0
IF distance is VeryBig AND (delta is NOT Growing OR delta is NOT GrowingFast) THEN action is FloorIt

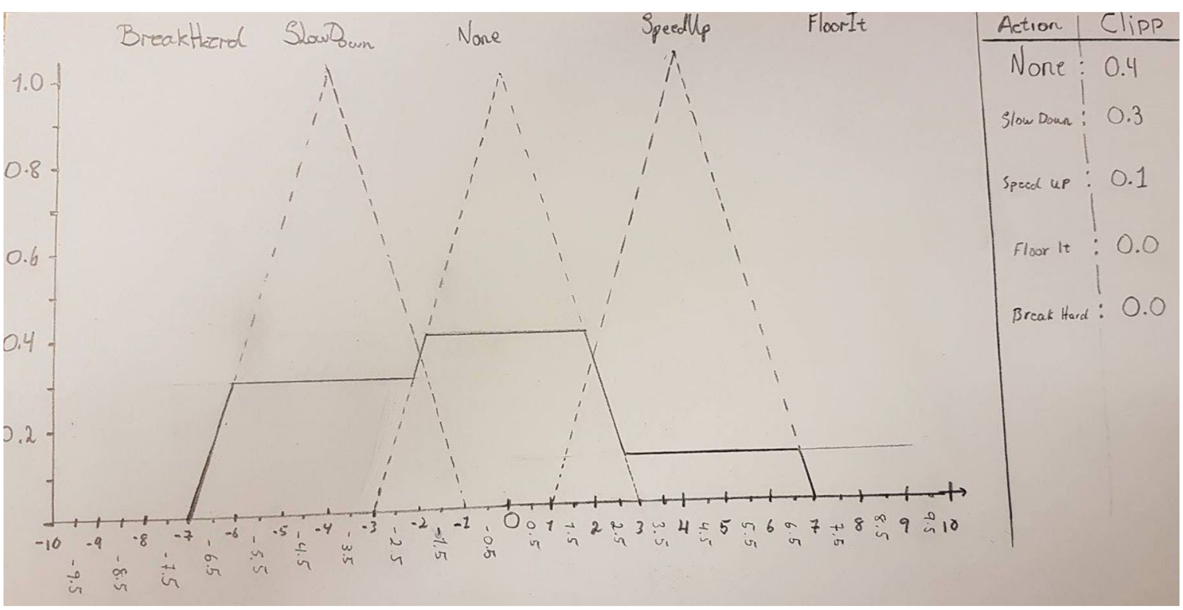
Values 0 ==> 0
IF distance is VerySmall THEN action is BrakeHard

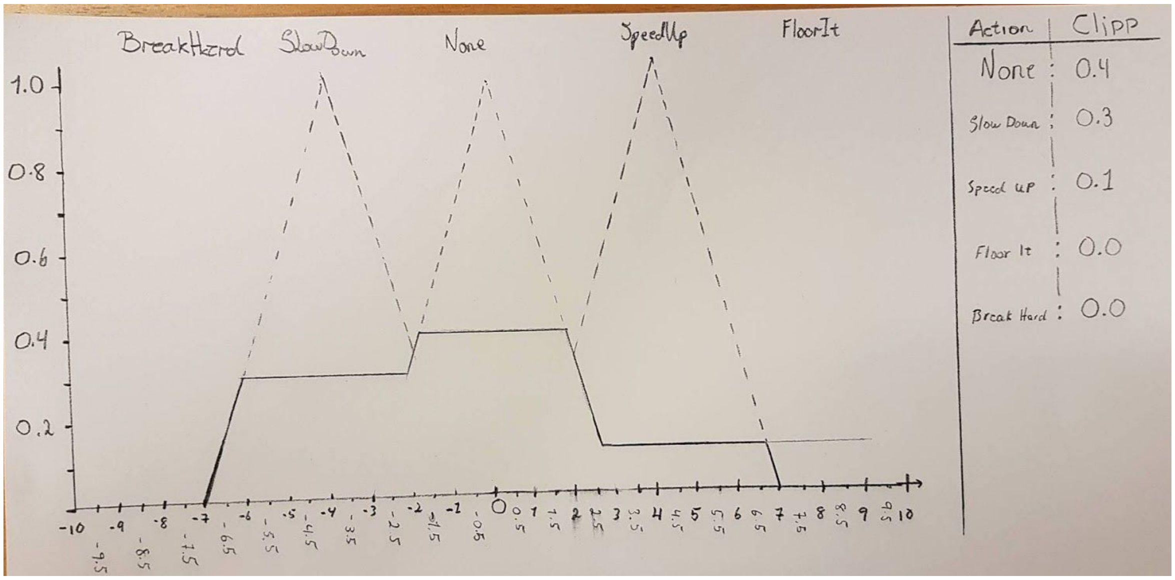
Step 3 : Aggregation!

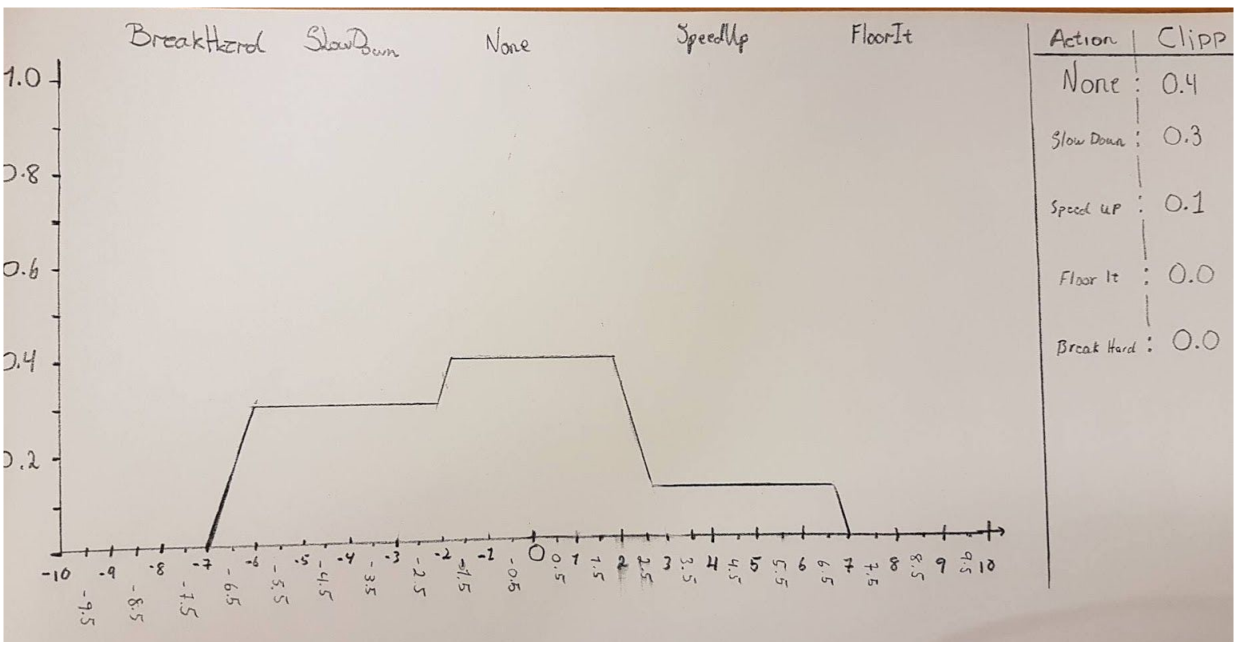
If we now clip the fuzzy sets based on these rule evaluations we get:

The clipped action fuzzy set are combined into just ONE new fuzzy set. (Clipped on right hand side values from step 2 (rule evaluation).

This gives us a NEW fuzzy set that looks like this: (3 images, all of same set, just shows where to cut, cut, and then another cleaner cut).







Now that we have created our new cut action fuzzy set. It's time to find the goddamn solution!

Step 4. Defuzzification

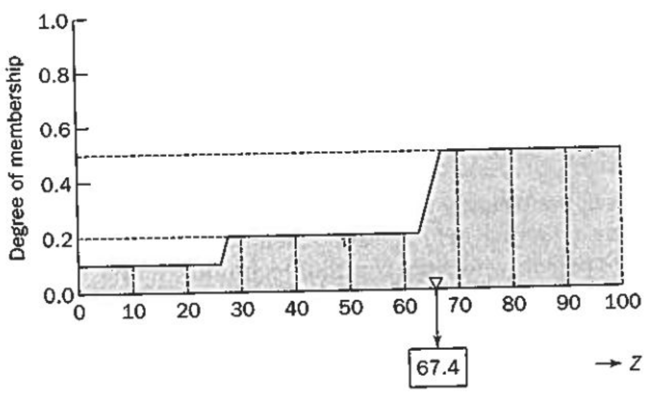
Defuzzification is just to calculate the area under / inside the new set / graph and is the actual fuzzy inference process. We do this by **centroid technique,** integral (∫) calculation. It <u>finds the point where a vertical line would slice the aggregate set into two equal masses</u>. Mathematically this **centre of gravity (COG)** can be expressed as

$$\mathrm{COG} = \frac{\int_a^b \mu_A(x)x\,dx}{\int_a^b \mu_A(x)\,dx}$$

In theory, the COG is calcualted over a continuum of points in the aggregate output membership functio, but in practice, a reasonable estimate can be obtained by calcualting it over a sample of points. Thus COG can be writtesn as

$$\mathrm{COG} = \frac{\sum_{x=a}^{b} \mu_A(x)x}{\sum_{x=a}^{b} \mu_A(x)}$$

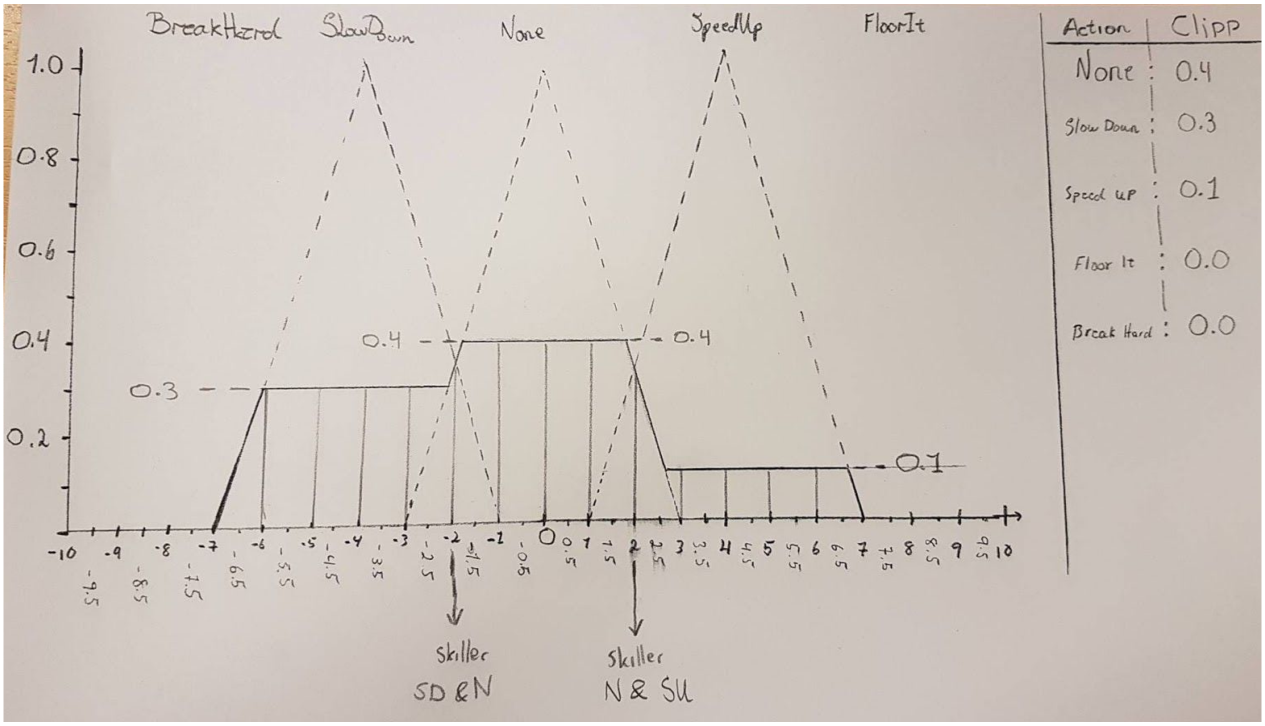
Here is an **example** of how to use the COG on a aggregated fuzzy set!



Clipped values:
0.1
0.2
0.3

Figure 4.14 Defuzzifying the solution variable's fuzzy set

$$\mathrm{COG} = \frac{(0+10+20)\times 0.1+(30+40+50+60)\times 0.2+(70+80+90+100)\times 0.5}{0.1+0.1+0.1+0.2+0.2+0.2+0.2+0.5+0.5+0.5+0.5}$$
$$= 67.4$$



-2 skiller SlowDown & None
2 skiller None & SpeedUp

$$COG = \frac{((-7 + -6 + -5 + -4 + -3 + -2) * 0.3) + ((-2 + -1 + 0 + 1 + 2) * 0.4) + ((3 + 4 + 5 + 6 + 7) * 0.1)}{0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.3 + 0.4 + 0.4 + 0.4 + 0.4 + 0.4 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1}$$

$$COG = \frac{((-27) * 0.3) + ((0) * 0.4) + (25) * 0.1)}{1.8 + 2 + 0.5}$$

$$COG = \frac{(-8.1) + (0) + (2.5)}{4.3}$$

$$COG = \frac{-5.5}{4.3}$$

$$COG = -1.279$$

Task 2 Programming Mandani

```python
import sys
import random as r


class Mamdani:
   def __init__(self, disPos, delPos):
       self.distancePos = disPos
       self.deltaPos = delPos

   def __repr__(self):
       return "I'm Mamdani!\nWho are you?"

   ########
   # Rules
   #######

   def AND_rule(self, *args):
       return min(*args)

   def OR_rule(self, *args):
       return max(*args)

   def NOT_rule(self, value):
       return 1 - value

   ########
   # Sets
   #######

   def distance_set(self):
       setValues = {
           "VerySmall": ['RG', [1.5, 2.75]],
           "Small": ['T', [1.75, 3, 4.75]],
           "Perfect": ['T', [3.75, 5, 6.75]],
           "Big": ['T', [5.75, 7, 8.75]],
           "VeryBig": ['G', [7.75, 9]]
       }
       return setValues
```

```python
def delta_time_set(self):
    setValues = {
        "ShrinkingFast": ['RG', [-3.5, -2.25]],
        "Shrinking": ['T', [-3.25, -1.75, -0.25]],
        "Stable": ['T', [-1.25, -0.25, 1.75]],
        "Growing": ['T', [0.75, 2.25, 3.75]],
        "GrowingFast": ['G', [2.75, 4.25]]
    }
    return setValues

def action_set(self):
    setValues = {
        "BrakeHard": ['RG', [-7.5, -5]],
        "SlowDown": ['T', [-7, -4, -1]],
        "None": ['T', [-3, 0, 3]],
        "SpeedUp": ['T', [1, 4, 7]],
        "FloorIt": ['G', [5, 7.5]]
    }
    return setValues

########
# Membership functions
#######

def triangle(self, position, x0, x1, x2, clip):
    value = 0.0

    if position >= x0 and position <= x1:
        value = (position - x0) / (x1 - x0)

    elif position >= x1 and position <= x2:
        value = (x2 - position) / (x1 - x0)

    if value > clip:
        value = clip

    return value

def grade(self, position, x, y, clip):
    value = 0.0

    if position >= y:
```

```
            value = 1.0
        elif position <= x:
            value = 0.0
        else:
            value = (position - x) / (y - x)

        if value > clip:
            value = clip

        return value

    def reverse_grade(self, position, x, y, clip):
        value = 0.0

        if position <= x:
            value = 1.0
        elif position >= y:
            value = 0.0
        else:
            value = (y - position) / (y - x)

        if value > clip:
            value = clip

        return value

    ########
    # ACTIONS
    #######

    def AND_TRIANGLE(self, distKey, deltKey, clip):
        dix0, dix1, dix2 = self.distance_set()[distKey][1]
        dex0, dex1, dex2 = self.delta_time_set()[deltKey][1]

        value = self.AND_rule(self.triangle(self.distancePos, dix0, dix1,
dix2, clip),
                             self.triangle(self.deltaPos, dex0, dex1, dex2,
clip))
        return value

    def RuleEvaluation(self, distance, delta, clip=100):

        return_set = {}
```

```
        if "Small" in distance and "Growing" in delta:
            t = self.AND_TRIANGLE("Small", "Growing", clip)
            return_set["None"] = t

        if "Small" in distance and "Stable" in delta:
            t = self.AND_TRIANGLE("Small", "Stable", clip)
            return_set["SlowDown"] = t

        if "Perfect" in distance and "Growing" in delta:
            t = self.AND_TRIANGLE("Perfect", "Growing", clip)
            return_set["SpeedUp"] = t

        if "VeryBig" in distance and "Growing" in delta and "GrowingFast" in
delta:
            dix0, dix1 = self.distance_set()["VeryBig"][1]
            dex0, dex1, dex2 = self.delta_time_set()["Growing"][1]
            dex0, dex1 = self.delta_time_set()["GrowingFast"][1]

            a = self.grade(self.distancePos, dix0, dix1, clip)
            print("*******A: " + str(a))
            b = self.NOT_rule(self.triangle(self.deltaPos, dex0, dex1, dex2,
clip))  # not growing
            print("*******B: " + str(b))
            c = self.NOT_rule(self.grade(self.deltaPos, dex0, dex1, clip))  #
not growingFast
            print("*******C: " + str(c))

            t = self.AND_rule(a, self.OR_rule(b, c))
            print("*******T: " + str(t))
            return_set["FoorIt"] = t

        if "VerySmall" in distance:
            dix0, dix1 = self.distance_set()["VerySmall"][1]
            t = self.reverse_grade(self.distancePos, dix0, dix1, clip)
            return_set["BrakeHard"] = t

        # return clip values set
        return return_set

    def getIntersection(self, set, value):
        dict = {}
        for key in set:
            if set[key][1][0] <= value and value <= set[key][1][-1]:
```

```
            if set[key] == 'RG':
                dict[key] = self.reverse_grade(value, set[key][1][0],
set[key][1][1], 100)
            elif set[key] == 'T':
                dict[key] = self.triangle(value, set[key][1][0],
set[key][1][1], set[key][1][2], 100)
            else:
                dict[key] = self.grade(value, set[key][1][0],
set[key][1][1], 100)
        return dict

    def reasoning(self):
        # Step 1: Fuzzication (Find values for each set in fuzzyset)
        intersection_set_dist = self.getIntersection(self.distance_set(),
self.distancePos)
        intersection_set_delta = self.getIntersection(self.delta_time_set(),
self.deltaPos)
        print("distance intersection ", intersection_set_dist, "\ndelta
intersection ", intersection_set_delta)

        # step 2: Rule evaluation
        action_set = self.RuleEvaluation(intersection_set_dist,
intersection_set_delta, 100)
        print("action set: ", action_set)

        # Step 3: Aggregation
        aggregated_set = self.aggregate(action_set)
        print("Aggregated set ", aggregated_set)

        # step 4: Defuzzification
        defuzz = self.Defuzzification(aggregated_set)

        return defuzz

    def Defuzzification(self, aggregated_set):

        top_level = 0.0
        bottom_level = 0.0

        for key in aggregated_set:
            values = aggregated_set[key][1]
            val = 0
            count = 0
```

```
            for x in range(values[0], values[2] + 1):
                val += x
                count += 1

            top_level += val * values[1]
            bottom_level += count * values[1]

        COG = top_level / bottom_level
        return COG

    def aggregate(self, action_set):

        aggregated_set = {}
        action_s = self.action_set()

        for key in action_set:
            aggregated_set[key] = action_s[key]
            if (aggregated_set[key][0] == "RG"):
                aggregated_set[key][1][0] = action_set[key]
            else:
                aggregated_set[key][1][1] = action_set[key]

        return aggregated_set


########
# Running this badboy
#######



def main(distancePos, deltaPos):
    '''
     Assignment values: $ python Mamdani_reasoner.py 3.7 1.7
    '''

    distancePos = float(distancePos)  # convert sys args to float, not string
    deltaPos = float(deltaPos)  # convert sys args to float, not string
    m = Mamdani(distancePos, deltaPos)

    action = m.reasoning()
    print("Action crisp value is: " + str(action))
```

```
main(sys.argv[1], sys.argv[2])
```