VISA
TP 9

Logique floue

Quentin Brault

1. Fonctions d'appartenance

a) Ensembles flous

Chaque fonction d'appartenance est décrite par une fonction qui prend une valeur en entrée et retourne un résultat entre 0 et 1 correspondant à cette valeur.
Dans notre cas, la valeur en entrée peut être n'importe quel nombre entre 0 et 40, ce qui correspond à la température en °C, et il y a 3 fonctions d'appartenance (température basse, moyenne et élevée).
Ces trois fonctions d'appartenance prennent la forme d'un trapézoïde, donc j'ai d'abord créé une fonction trapézoïde générique qui sera réutilisée par chacune des fonctions d'appartenance.
a, b, c et d représentent la position sur l'axe des abscisses des 4 coins du trapézoïde. Sur l'axe des ordonnées, les coins associés à a et d ont une valeur de 0 et les points associés à b et c ont une valeur de 1.

```
def fuzzy_trapezoid(a, b, c, d, x):
  if a < x < b:
    return (x - a) / (b - a)
  if b <= x <= c:
    return 1
  if c < x < d:
    return (d - x) / (d - c)
  return 0
```

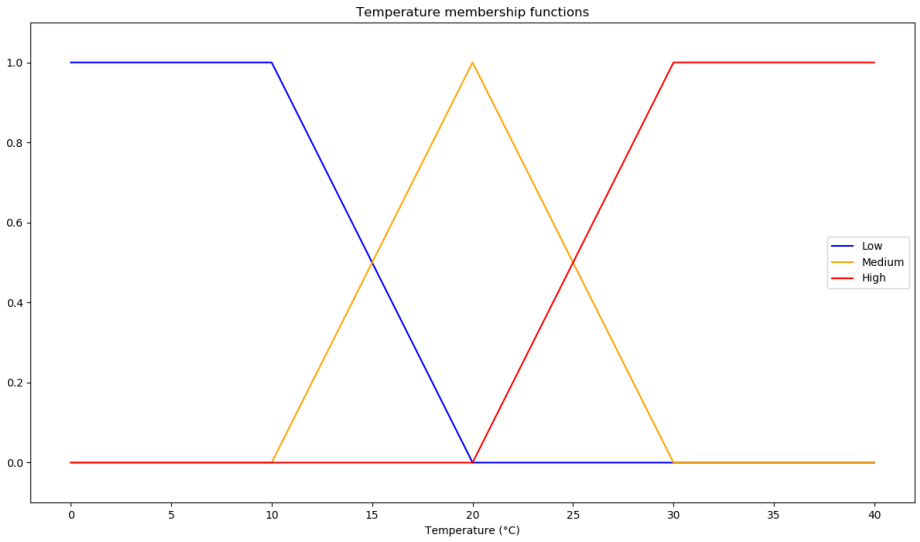
On peut donc maintenant définir nos fonctions d'appartenance en se basant sur le graphique donné..

```
def low_temp(temperature):
  return fuzzy_trapezoid(0, 0, 10, 20, temperature)

def medium_temp(temperature):
  return fuzzy_trapezoid(10, 20, 20, 30, temperature)

def high_temp(temperature):
  return fuzzy_trapezoid(20, 30, 40, 40, temperature)
```

Pour vérifier notre ensemble, on peut appeler ces fonctions en itérant sur les abscisses et dessiner le graphe correspondant.



b) Degrés d'appartenance

J'ai créé une fonction our récupérer les degrés de chaque fonction d'appartenance en une valeur donnée.

```
def fuzzy_grades(membership_functions, value):
  return [membership_function(value) for membership_function in
  ↪  membership_functions]
```

On peut donc appeler cette fonction avec nos fonctions d'appartenances et la valeur 16 pour trouver les degrés d'appartenance de chaque fonction à 16°C.

```
membership_functions = [low_temp, medium_temp, high_temp]
print(fuzzy_grades(membership_functions, 16))
```

Grades of each membership function for 16°C

Low	Medium	High
0.4	0.6	0

c) Température basse ou moyenne

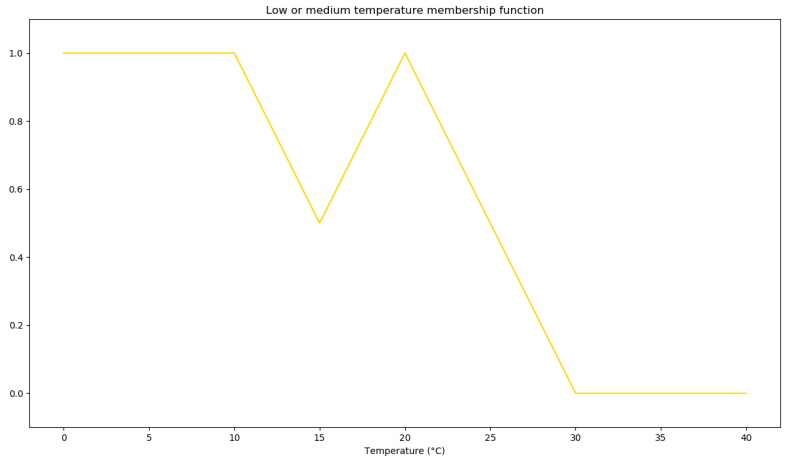
J'ai commencé par créer une fonction qui retourne le OU de deux valeurs provenant d'une fonction floue. J'ai également ajouté les fonctions logiques NOT et AND.

```
def fuzzy_not(grade):
  return 1 - grade

def fuzzy_or(grade1, grade2):
  return max(grade1, grade2)

def fuzzy_and(grade1, grade2):
  return min(grade1, grade2)
```

Le NOT correspond à 1 moins la valeur, car nos valeurs sont entre 0 et 1. Le OR correspond au max des deux valeurs car la valeur maximale correspond à la valeur la plus vraie. Le AND correspond au min des deux valeurs car la valeur minimale correspond à la valeur la moins vraie. Il suffit alors d'appeler, pour chaque valeur souhaitée, les fonctions d'appartenance basse et moyenne et d'appeler *fuzzy_or* sur chaque résultat. J'ai cependant été plus loin et j'ai créé une fonction générique qui fera tout cela pour nous. Je détaille cette fonction générique dans la partie 2.
Si on appelle cette fonction générique pour tracer un graphique on obtient le résultat souhaité :



2. Opérateurs de la logique floue

J'ai souhaité créer une fonction générique qui calculerait pour n'importe quel ensemble d'opérations logiques les résultats de leurs combinaisons. Cette fonction prend en entrée deux choses :

- Un arbre avec pour chaque feuille une fonction d'appartenance et à chaque autre nœud une d'opération logique à appliquer sur les enfants de ce nœud.
- Des valeurs qui seront données à chaque fonction d'appartenance pour calculer toutes les opérations sur l'ensemble des valeurs.

Reprenons le cas de température basse ou moyenne de l'exercie 1.a. On souhaite calculer les degrés d'appartenance pour la fonction basse et la fonction moyenne, puis appeler *fuzzy_or* dessus et cela pour chaque valeur. Les valeurs ici sont les valeurs de l'axe des abscisses. On les appelle *x_data*. On écrit l'arbre d'opération logique dans une liste *low_or_medium_temp*. Cela donne :

```
x_data = np.arange(xMin, xMax, xStep)
low_or_medium_temp = [
  fuzzy_or,
  [low_temp],
  [medium_temp]
]
```

Ces paramètres sont alors donnés à ma fonction qui retourne l'ensemble des résultats calculés pour chaque valeur d'entrée.

```
y_data = fuzzy_combine(low_or_medium_temp, x_data)
```

Regardons maintenant cette fonction. Il s'agit d'une fonction récursive, qui à chaque récursion calcule le résultat de l'opération logique au nœud de l'arbe à partir du ou des enfants de ce nœud. Lorsqu'on est sur une feuille de l'arbre, il faut simplement retourner le degré d'appartenance pour chaque valeur. Lorsqu'on est sur un autre nœud, on applique la fonction logique sur les 1 ou 2 enfants en appelant récursivement *fuzzy_combine*.

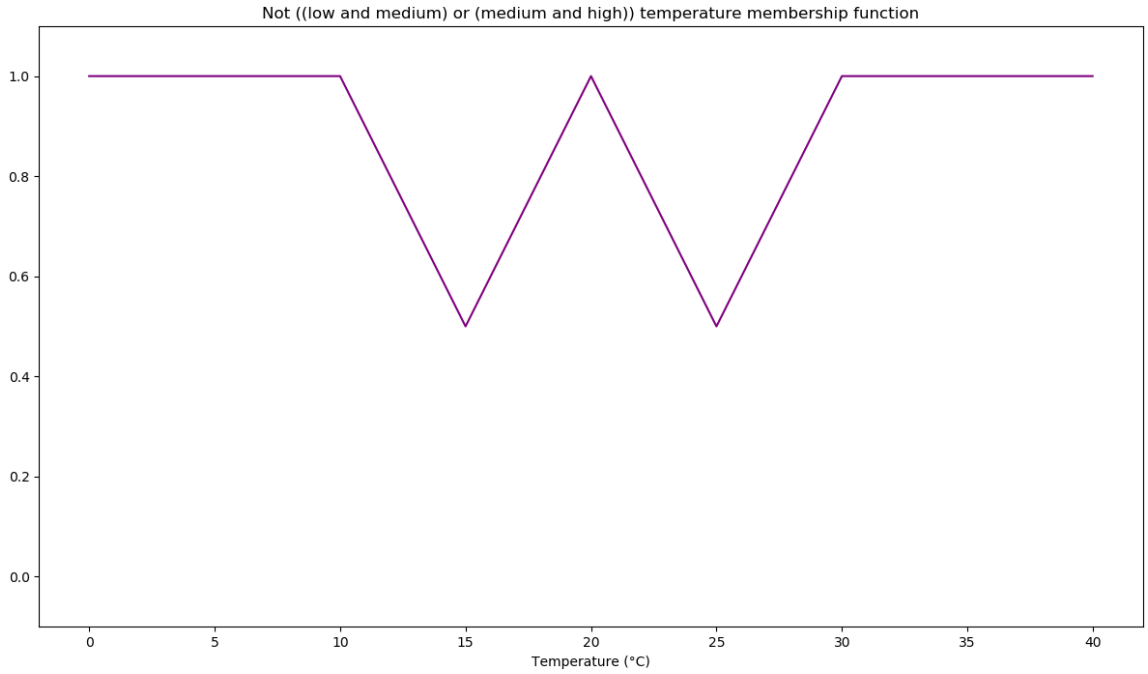
```
def fuzzy_combine(functions, values):
  if len(functions) == 1:
    return list(map(functions[0], values))

  if len(functions) == 2:
    return list(map(functions[0], fuzzy_combine(functions[1], values)))

  if len(functions) == 3:
    return list(map(functions[0], fuzzy_combine(functions[1], values),
    ↪  fuzzy_combine(functions[2], values)))
```

On peut vérifier que cela marche sur des arbres logiques plus complexes. Par exemple, NOT((basse AND moyenne) OR (moyenne AND haute)) :

```
x_data = np.arange(xMin, xMax, xStep)
not_low_and_medium_or_medium_and_high_temp = [
  fuzzy_not,
  [
    fuzzy_or,
    [
      fuzzy_and,
      [low_temp],
      [medium_temp]
    ],
    [
      fuzzy_and,
      [medium_temp],
      [high_temp]
    ]
  ]
]
y_data = fuzzy_combine(not_low_and_medium_or_medium_and_high_temp, x_data)
```



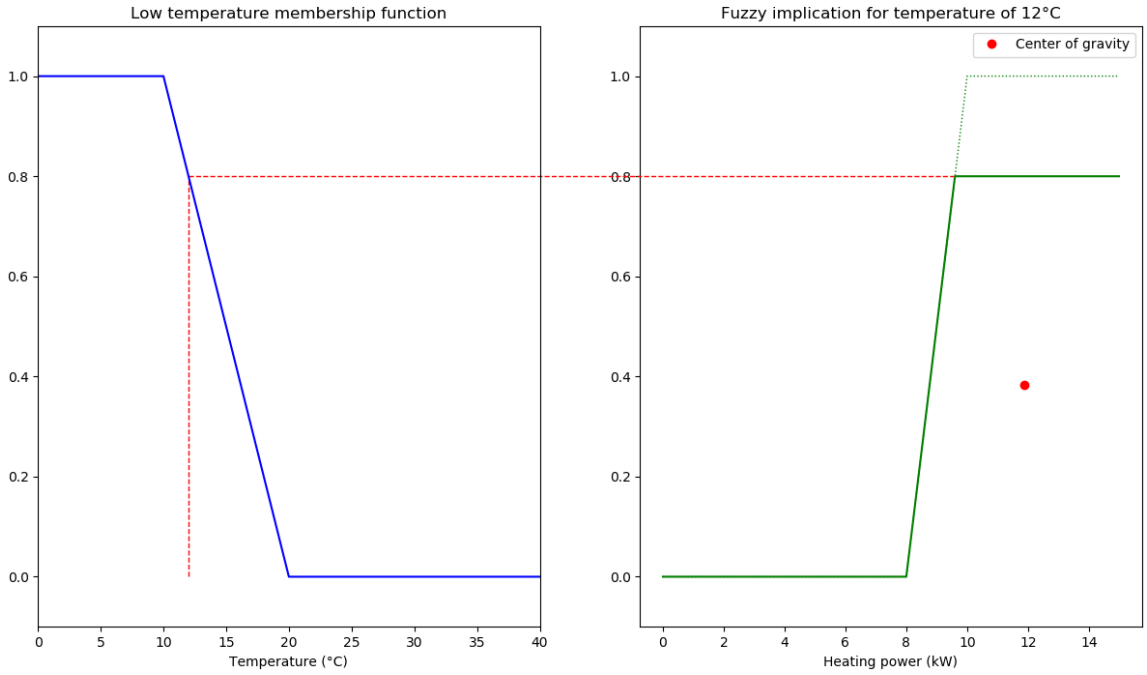
Note : Les résultats ne sont pas normalisés. On pourrait souhaiter implémenter une fonction de normalisation si cela nous était utile.

3. Implication floue

J'ai créé une fonction qui calcule à partir d'un degré *grade* issu d'une fonction d'appartenance, d'une fonction d'appartenance *rule* et d'un ensemble de valeurs *values*, calcule les degrés du résultat de l'implication de Mamdani pour chaque valeur. L'implication de Mamdani revient à faire le minimum.

```
def fuzzy_imply_mamdani(grade, rule, values):
  return [min(grade, rule(value)) for value in values]
```

L'ensemble flou issue de ce calcul d'implication est donc :



Le centre de gravité du résultat de l'implication est calculé avec une fonction de défuzzification COG. Cette fonction calcule d'abord une approximation de l'aire sous la courbe.
Soit f la fonction d'appartenance et val le domaine de définition de la fonction d'appartenance discrétisé avec un pas de $dVal$:
$area = \sum_{i=1}^{|val|} f(val_i).dVal$
Puis la fonction calcule la coordonnée x du centre de gravité avec :
$x = \frac{\sum_{i=1}^{|val|} val_i.f(val_i).dVal}{area}$
Ce calcul revient à appliquer un poids pour chaque val_i correspondant à son ratio par rapport à l'aire sous la courbe. Enfin elle calcule la coordonnée y du centre de gravité :
$y = \frac{1}{2}.\frac{\sum_{i=1}^{|val|} f(val_i).f(val_i).dVal}{area}$
Ce calcul revient à appliquer un poids pour chaque $f(val_i)$ correspondant à son ratio par rapport à l'aire sous la courbe. On multiplie par $\frac{1}{2}$ pour récupérer le milieu de la valeur y trouvée.

```
def fuzzy_defuzzify_cog(values, grades):
  dx = values[1] - values[0]
  area = sum([grades[i] * dx for i in range(len(values))])
  x = sum([values[i] * grades[i] * dx for i in range(len(values))])
  y = sum([grades[i] * grades[i] * dx for i in range(len(values))])
  return [x / area, 0.5 * (y / area)]
```