TP2Sujet

December 3, 2018

1 TP2 : quantification et codage

Ce TP est consacré à l'implémentation de l'algorithme de Lloyd-Max pour quantifier un signal, et au codage et décodage de Huffman.

On chargera les librairies classiques suivantes :

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np
        import scipy
        import random
```

1.1 Quantification par l'algorithme de Lloyd-Max

Dans un premier temps, on chargera le fichier `musique.wav` disponible ici par la librairie

```
from scipy.io import wavfile
```

1.1.1 Lecture du fichiers wav

Chargez le fichier grâce à la fonction `wavfile.read`. On représentera le signal en fonction du temps. Combien de temps dure la musique chargée ? Quelle est la fréquence d'échantillonnage du fichier ?

1.1.2 Extraction d'une sous partie

Extraire la partie du signal entre $t = 1.25s$ et $t = 2s$. Représenter ce signal. Normalisez le signal entre les valeurs $[-1, 1]$.

1.1.3 Fabrication de la densité de probabilité des valeurs du signal

1. À l'aide de la fonction `np.histogram` (ou par une toute autre méthode), définissez la densité de probabilité pour le signal. On pourra la tracer et vérifier numériquement que son integrale vaut bien 1.
2. À l'aide de `scipy.interpolate.interp1d` on définira une fonction f_S à partir des valeurs de sont histogramme.

1.1.4 Intégrale et moment

On définira la fonction $x \to x f_S(x)$ et les fonctions qui retournent $\int_a^b x f_S(x) \mathrm{d}x$ et $\int_a^b f_S(x) \mathrm{d}x$. On pourra utiliser la fonction `scipy.integrate.quad`.

1.1.5 Implémentation de l'algorithme

Implémentez l'algorithme de Lloyd-Max qui calculera les suites (b_i) et (y_i) du cours. Cet algorithme pourra prendre la forme d'une fonction python :

```
def LloydMax(b0,y0,maxiter=300,erreur=1e-03):
```

L'algorithme peut converger lentement. On veillera à ne pas prendre une précision trop importante pour le critère d'arrêt.

1.1.6 Bruit de quantification

1. Définissez une fonction qui calcule numériquement le bruit de quantification. Elle pourra prendre la forme suivante :

   ```
   def calculBruit(signal,ylist,blist):
   ```

   On pourra définir une fonction `erreur` du réel x et d'une valeur y_i qui retournera $(x - y_i)^2 f_S(x)$.

2. On comparera le bruit de quantification entre une quantification uniforme et la quantification issue de l'algorithme de Lloyd-Max. On pourra regarder l'influence de la precision de convergence dans l'algorithme de Lloyd-Max (attention, la convergence peut être longue à obtenir).

1.2 Codage source

Dans cette partie du TP, on s'intéressera au codage source avec notamment l'algorithme de Huffman.

On considère la chaine de caractère suivante que l'on cherchera à coder en binaire :

```
In [2]: phrase = "le traitement du signal est une discipline large qui est à la croisée de plu
```

1.2.1 Dictionnaire et fréquence

Dictionnaire en python En python, un dictionnaire est défini (initialisé) par:

```
In [3]: mondict = {}
```

Par exemple :

```
In [4]: a = {}
        a["nom"] = "engel"
        a["prenom"] = "olivier"
        a["age"] = 23
        print(a)
        print(a["age"])
```

```
{'nom': 'engel', 'prenom': 'olivier', 'age': 23}
23
```

Calcul des fréquences d'apparition En se définissant un dictionnaire (au sens de n-uplet ou d'une liste pour python) pour la langue française (on codera toutes les lettres minuscules ainsi que les espaces), définissez une fonction qui calcule la fréquence d'apparition. Elle retournera un dictionnaire (lettre/fréquence) python et pourra avoir la forme :

```
def frequences(maphrase,ledictionnaire):
```

On pourra utiliser la fonction `count` pour compter dans la chaîne de caractère.

1.2.2 Fabrication de l'arbre de codage de Huffman

On se propose ici d'ulitiser la librairie `heapq` de python (mais vous pouvez faire tout autrement !!). Celle-ci permet d'organiser une liste sous forme d'arbre binaire. Cette librairie nous permettra facilement de repérer les plus petits éléments.

On la chargera par :

```
In [5]: import heapq
```

L'exemple suivant permet de comprendre que le *push* et le *pop* permettent respectivement d'ajouter au bon endroit l'élément, et de récupérer le plus petit élément avec le *pop*.

```
In [6]: l = []
        heapq.heappush(l, 69)
        heapq.heappush(l, 42)
        heapq.heappush(l, 2048)
        heapq.heappush(l, -273.15)
        print(l) # la liste est ordonnée en arbre binaire...
        for x in np.arange(len(l)): # et on depop pour itérer dans l'odre
            print(heapq.heappop(l))
```

```
[-273.15, 42, 2048, 69]
-273.15
42
69
2048
```

Évidemment, les éléments dans l'arbre binaire peuvent ne pas être uniquement des réels, mais par exemple, un nuplet de liste.

Pour fabriquer un arbre à partir d'une liste on fera :

```
In [7]: a = [(45,"albert"),(26,"olivier"),(30,"sophie")]
        arbre = heapq.heapify(a)
        print(a)
        print(heapq.heappop(a))
```

```
        print(a)
        print(heapq.heappop(a))
        print(a)
```

```
[(26, 'olivier'), (45, 'albert'), (30, 'sophie')]
(26, 'olivier')
[(30, 'sophie'), (45, 'albert')]
(30, 'sophie')
[(45, 'albert')]
```

Sur cette exemple, par rapport à quelle valeur des n-uplet, l'arbre est-il construit ?

Quelques petites astuces qui pourront être utiles

```
In [8]: print(['a']+['b'])
```

```
['a', 'b']
```

```
In [9]: import collections

        s = [('yellow', "1"), ('blue', "2"), ('yellow', "3"), ('blue', "4"), ('red', "1")]
        d = collections.defaultdict(list)
        for k, v in s:
            d[k].append(v)

        print(d.items())

        for k,v in d.items():
            d[k] = ''.join(d[k])
        print(d.items())
```

```
dict_items([('yellow', ['1', '3']), ('blue', ['2', '4']), ('red', ['1'])])
dict_items([('yellow', '13'), ('blue', '24'), ('red', '1')])
```

Codez maintenant une fonction `huffman` qui prend en argument le dictionnaire de fréquence et qui retourne le code de Huffman de chaque élément du dictionnaire en base 2 sous forme d'un dictionnaire.

On pourra, en utilisant des librairies graphiques, tracer cet arbre de codage.

1.2.3 Encodage et decodage de la phrase

1. Maintenant que nous avons notre arbre de codage, on peut encoder la phrase. Définissez une fonction d'encodage qui à une chaîne de caractère et un arbre de codage, retourne une chaîne de caractère composée de 0 et de 1.
2. Définissez une fonction de décodage.

1.3 Codage canal

1. Appliquer le codage de Hamming (4,7) au code source précédent.
2. Générez des erreurs aléatoires sur les bits du code et étudiez la détectabilité et la correction suivant la probabilité de bit erroné.