TP2 – Compression d'un signal

Initiation à Matlab

Opérations sur les matrices (suite)

- `length(V)` calcule la taille d'un vecteur (ligne ou colonne) ou la plus grande dimension d'une matrice.
- `indices = find(V==0)` : le vecteur colonne `indices` contient les indices `i` de la *matrice vectorisée* `W = V(:)` tels que `W(i)==0` (n'importe quelle expression booléenne peut être passée en paramètre à la fonction `find`).
- `[V_trie,indices] = sort(V,'ascend')` : si `V` a plus d'une ligne, alors `V_trie` est une version de `V` triée par ordre croissant, colonne par colonne ; sinon, le tri est effectué sur la ligne unique de `V` ; la matrice `indices` contient les indices correspondants ; pour `'descend'` au lieu de `'ascend'`, idem par ordre décroissant.

Exercice d'initiation à Matlab

Le script `exercice_Matlab.m` lit un texte en français, le stocke dans le vecteur `texte` et crée un alphabet ASCII de 128 caractères.

Complétez ce script de manière à calculer les fréquences des différents caractères de l'alphabet dans le texte, en utilisant la fonction `find`, puis leurs fréquences relatives, qui peuvent être assimilées à des probabilités. En triant ces valeurs, affichez l'histogramme des caractères du texte par ordre de fréquence relative décroissante.

Codage de Huffman

Le niveau de gris d'une image numérique est un entier compris entre 0 et 255. Il est généralement codé par une séquence de 8 bits appelée *octet*. La correspondance entre un entier et une séquence de 8 bits peut être a priori quelconque, du moment qu'il existe une bijection entre les deux ensembles. La convention usuelle est celle de la représentation en base 2, mais d'autres conventions que la représentation en base 2 seraient possibles. Avec un tel *codage à longueur fixe*, la taille d'une image est proportionnelle au nombre de pixels. La **compression sans perte** consiste à coder les entiers autrement, de manière à utiliser moins de bits qu'avec un codage à longueur fixe. Son principe est celui du *codage à longueur variable* : les entiers les plus fréquents sont codés sur un plus petit nombre de bits que les entiers les moins fréquents. L'*algorithme de Huffman* permet d'obtenir un codage optimal, qui est fonction des fréquences des différents entiers à coder. Il doit donc disposer de la fréquence f_n de chaque entier n (c'est-à-dire de son nombre d'occurrences). Il construit un arbre binaire de manière récursive, à partir d'un ensemble initial d'arbres binaires. Chaque arbre binaire initial est constitué d'un seul élément, qui est un des entiers n à coder, de fréquence f_n non nulle. La suite de l'algorithme consiste en l'itération suivante :

1. Classer les arbres binaires selon leurs fréquences.
2. Remplacer les deux arbres binaires de fréquences les plus faibles, par un nouvel arbre binaire dont la racine pointe sur les racines des deux arbres binaires supprimés. Affecter comme fréquence à ce nouvel arbre binaire la somme des fréquences des deux arbres binaires supprimés.
3. Retourner en 1 tant que le nombre d'arbres binaires est strictement supérieur à 1.

L'arbre binaire ainsi construit a pour nœuds terminaux l'ensemble des entiers à coder. Pour connaître le code de Huffman associé à un entier, il suffit de descendre l'arbre, en partant de la racine, pour rejoindre cet entier. À chaque embranchement, on ajoute 0 si on passe à gauche et 1 si on passe à droite. Toute l'astuce du codage de Huffman réside dans le fait qu'un code ne peut pas être le préfixe d'un autre code : par exemple, les codes 001 et 00100 ne peuvent pas coexister. C'est grâce à cette propriété qu'un fichier codé pourra être décodé.

Un générateur visuel d'arbre de Huffman est disponible à l'adresse `http://huffman.ooz.ie/`.

Exercice 1 : codage de Huffman d'un texte

Le script `exercice_1.m` code le texte du script `exercice_Matlab.m` par le codage de Huffman. Sachant qu'en ASCII, les caractères sont codés sur 8 bits, calculez le nombre de bits nécessaires pour coder ce texte dans sa version d'origine, puis le coefficient de compression atteint par le codage de Huffman. Vérifiez également qu'il s'agit d'un **codage sans perte** en décodant le texte codé : vous devez retrouver le texte original.

Exercice 2 : codage de Huffman d'une image

Dans le TP1, vous avez effectué la décorrélation des niveaux de gris dans une image. La décorrélation est très utile pour la compression sans perte, puisqu'elle permet de restreindre le nombre de valeurs à coder.

Complétez le script `exercice_2.m` afin de calculer le codage de Huffman de l'image du cameraman, et calculez le coefficient de compression obtenu. Faites de même pour la version décorrélée de cette image. Calculez enfin le gain en compression procuré par la décorrélation (dont on espère qu'il sera supérieur à 1).

Exercice 3 : codage arithmétique d'un texte

Le codage de Huffman est théoriquement optimal, en ce sens qu'un message est codé avec un nombre de bits minimal. Toutefois, pour certaines distributions de caractères, ce codage est limité à cause du fait que chaque caractère est codé par un nombre **entier** de bits. Par exemple pour l'alphabet (s, u, v) avec des probabilités $(0.9, 0.05, 0.05)$, chaque caractère est codé au minimum sur un bit, malgré l'écrasante majorité de s.

Le *codage arithmétique* permet de traiter ce genre de cas, en n'imposant pas un nombre entier de bits. Ce type de codage est généralement plus performant que le codage de Huffman. Dans le codage arithmétique, étant donné une loi de probabilité, chaque message est codé de façon unique par un *intervalle* `[borne_inf,borne_sup[`. Tout nombre dans cet intervalle code alors le message original.

Le dictionnaire est remplacé par un tableau, appelé par exemple `bornes`. Chaque lettre de l'alphabet est codée par un intervalle inclus dans $[0, 1[$, de longueur égale à la probabilité de la lettre (ces intervalles constituent donc une partition de $[0, 1[$). Pour l'exemple précédent, s est codée par $[0, 0.9[$, u par $[0.9, 0.95[$ et v par $[0.95, 1[$. Par conséquent : `bornes = [0 0.9 0.95 ; 0.9,0.95,1]`.

Une fois ce tableau calculé, on encode un message en rétrécissant progressivement la taille de l'intervalle, de la manière suivante :

1. Initialisation : `borne_inf = 0`, `borne_sup = 1`
2. Pour chaque caractère du message (dans l'ordre d'apparition), d'indice `c` dans l'alphabet, faire :

```
largeur = borne_sup-borne_inf
borne_inf_sauv = borne_inf
borne_inf = borne_inf+largeur*bornes(1,c)
borne_sup = borne_inf_sauv+largeur*bornes(2,c)
```

3. Choisir un nombre quelconque dans l'intervalle `[borne_inf,borne_sup[` : ce nombre code le message !

Le décodage s'effectue ensuite de façon similaire.

Complétez le script `exercice_3.m` de façon à implémenter le codage arithmétique. Le choix d'un nombre dans l'intervalle `[borne_inf,borne_sup[` est fourni, ainsi que le décodage. Comparez le nombre de bits nécessaires pour coder un même texte avec l'algorithme de Huffman ou avec le codage arithmétique, pour différents messages courts. Dans quels cas l'algorithme de Huffman est-il vraiment inadapté ?

Essayez enfin de compresser et de décompresser par codage arithmétique des messages longs. Vous constatez qu'un problème apparaît. Comment pourrait-on le résoudre ?