TP1 – Décorrélation et compression

Exercice 1 : corrélations entre pixels voisins et méthode de décorrélation

Cet exercice constitue une illustration du cours de probabilités consacré à un couple de variables aléatoires. Ici, chaque pixel d'une image numérique est considéré comme une variable aléatoire. On s'intéresse alors à la corrélation entre pixels voisins.

FIGURE 1 – Exemple d'image en niveaux de gris

Le script Matlab de nom `exercice_1` affiche une image `I` interne à Matlab (cf. FIGURE 1). Il doit également afficher les paires de niveaux de gris d'un pixel (de gauche) et de son voisin de droite sous la forme d'un nuage de points afin d'évaluer la corrélation entre eux. Écrivez tout d'abord la fonction `vectorisation_par_colonnes` d'en-tête :

```
function [Vg,Vd] = vectorisation_par_colonnes(I)
```

dont les deux paramètres de sortie `Vd` et `Vg` doivent être deux sous-matrices de `I` vectorisées, c'est-à-dire deux vecteurs colonnes, comprenant respectivement les voisins de droite et de gauche associés. Écrivez ensuite la fonction `parametres_correlation` prenant la forme :

```
function [r,a,b] = parametres_correlation(Vd,Vg)
```

qui calcule le coefficient de corrélation linéaire r des données, ainsi que les deux paramètres (a, b) de la droite de régression d'équation $y = ax + b$ (voir ci-dessous pour les rappels des différentes formules).
Remarque : Cet exercice doit être résolu sans boucle `for`, `while` ...

La décorrélation des niveaux de gris consiste, par exemple, à soustraire au niveau de gris d'un pixel le niveau de gris de son voisin de gauche. Dans le script `exercice_1bis`, `I` est remplacée par sa version décorrélée `I_decorrelee`, à l'aide de la fonction `decorrelation_colonnes` que vous devez compléter, et d'en-tête :

```
function I_decorrelee = decorrelation_colonnes(I)
```

Pour cela, initialisez `I_decorrelee` par duplication de `I`, conservez la première colonne et modifiez les autres colonnes de cette matrice.
Remarque : Il est nécessaire de conserver la première colonne de l'image d'origine dans `I_decorrelee`, sans quoi l'opération de décorrélation ne serait pas réversible (il serait impossible de recalculer l'image d'origine).

Rappels

- Moyenne : $\bar{x} = \dfrac{1}{n}\sum_{i=1}^{n} x_i$
- Variance : $\sigma_x^2 = \dfrac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 = \dfrac{1}{n}\sum_{i=1}^{n} x_i^2 - \bar{x}^2$
- Écart-type : $\sigma_x = \sqrt{\sigma_x^2}$
- Covariance : $\sigma_{xy} = \dfrac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) = \dfrac{1}{n}\sum_{i=1}^{n} x_i\, y_i - \bar{x}\,\bar{y}$
- Coefficient de corrélation linéaire : $r = \dfrac{\sigma_{xy}}{\sigma_x\,\sigma_y}$
- Équation de la droite de régression : $y = \dfrac{\sigma_{xy}}{\sigma_x^2}\,(x - \bar{x}) + \bar{y}$

Principe du codage de Huffman

Le niveau de gris d'une image numérique est un entier compris entre 0 et 255. Il est généralement encodé avec 8 bits appelés *octet*. La correspondance entre un entier et ces 8 bits peut être a priori quelconque, du moment qu'il existe une bijection entre les deux ensembles. Avec un tel *codage à longueur fixe*, la taille d'une image est proportionnelle au nombre de pixels. La *compression sans perte* consiste à encoder 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 encodés 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 encoder. Il doit donc disposer de la fréquence f_n de chaque entier n. 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 encoder, 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 encoder. 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 encodé pourra être décodé.

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

Exercice 2 : histogramme et codage de Huffman pour des images

Le script `exercice_2` permet de comparer l'efficacité du codage de Huffman pour la compression entre l'image avant et après décorrélation. Dans un premier temps, on a besoin de connaître la répartition des différents niveaux de gris présents dans l'image `I` en affichant son histogramme. Pour ce faire, complétez la fonction `histogramme_normalise` appelée par ce script, d'en-tête :

```
[vecteurs_frequences,vecteur_Imin_a_Imax] = histogramme_normalise(I)
```

où `vecteurs_frequences` contient les fréquences de chaque niveau de gris et `vecteur_Imin_a_Imax` est un vecteur de pas 1 allant de `Imin`, valeur minimale du niveau de gris, à `Imax`, valeur maximale. Vous pouvez pour cela vous aider de la fonction `histcounts` (attention aux bornes !) afin d'éviter l'utilisation de boucles `for`.

Par la suite, cette fonction va pouvoir être réutilisée pour effectuer le codage de Huffman d'une image à l'aide de la fonction :

```
[I_encodee,dictionnaire,hauteur,largeur] = encodage_image(I)
```

Les différentes étapes de cette fonction consistent en la création d'un dictionnaire associant un code binaire à chaque niveau de gris, puis en l'encodage de l'image à proprement parler, avec les fonctions suivantes :

- `dictionnaire = huffmandict(vecteur_Imin_a_Imax,vecteurs_frequences)`
- `I_encodee = huffmanenco(I(:),dictionnaire)`

Remarque : Il est nécessaire de vectoriser l'image avant de pouvoir effectuer l'encodage.

Une fois le codage effectué, on souhaite connaître le *coefficient de compression* associé. Pour cela, écrivez la fonction `coefficient_compression` d'en-tête :

```
function coeff_comp = coefficient_compression(signal_non_encode,signal_encode)
```

qui doit calculer ce dernier, qui est défini comme le rapport entre le nombre de bits nécessaires pour encoder une image dans sa version d'origine (les pixels sont encodés sur 8 bits) et le nombre de bits de la même image encodée par le codage de Huffman. En décorrélant l'image initiale avant de la compresser, vous devez constater que le coefficient de compression augmente.

Exercice 3 : vérification du décodage sans perte

Afin de vérifier que le codage de Huffman est bien un codage sans perte, écrivez la fonction `reconstruction_image` qui prend la forme :

```
function I_reconstruite = reconstruction_image(I_encodee,dictionnaire,hauteur,largeur)
```

qui doit décoder l'image avec la fonction `I_decodee = huffmandeco(I_encodee,dictionnaire)`, puis la redimensionner avec la bonne hauteur et largeur, et enfin effectuer l'opération inverse de la décorrélation afin de retomber sur l'image d'origine en sortie. Si tout s'est bien passé, vous devriez avoir une norme d'erreur nulle lors de la reconstruction (valeur affichée dans la figure).