Partie 1

Rappels de cours - Images numériques

1.1 Image numérique

Une image numérique est un tableau de pixels. Le fichier de données contient un en-tête précisant le format et les dimensions, et des données pour la couleur de chaque pixel.

Pour manipuler ces données, on doit d'abord les importer dans une matrice (une liste de listes).

Pour une image couleur dont la profondeur est de 24bits (3 octets), chaque élément de la matrice est alors constitué d'un tuple (r,v,b), avec r, v et b codés sur un octet.

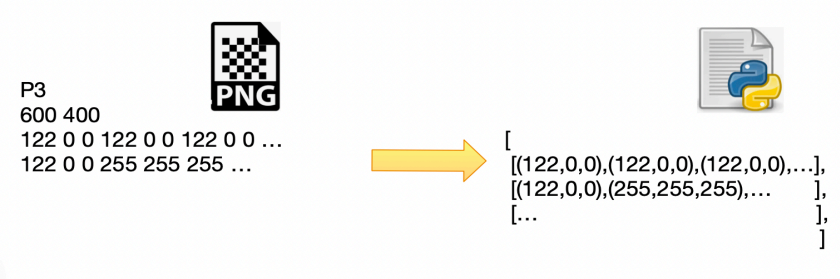

FIGURE 1 – données du fichier image, données importées dans un programme python

1.2 Traitement d'une image sans modifier sa dimension

On utilise la matrice d'une *image source*. On créé une matrice d'*image cible* :

```
f1 = open('image_source.ppm','r')
M_source = []
for line in f1.readlines()[3:]:
    L = line.split(" ")
    M_source.append(L)
M_cible = [[0] * len(M_source[0]) for i in range(len(M_source))]
```

On parcourt ensuite chaque valeur de pixel de l'image source, on effectue un traitement sur ces valeurs, et on place chacune dans la matrice de l'image cible.

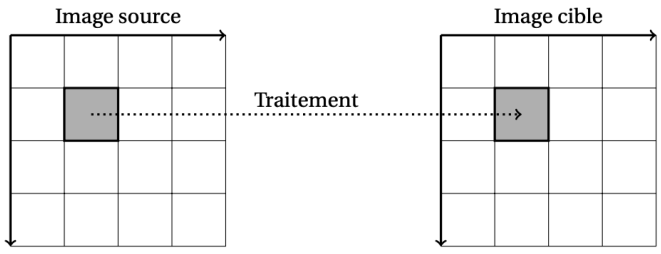

FIGURE 2 – image source => image cible

Puis on construit l'image cible à partir de sa matrice.

1.3 Modification de la dimension

L'image cible (et sa matrice cible) peuvent avoir des dimensions différentes de celle d'origine. Supposons que l'image d'origine fait (L, H).

Avec un facteur d'agrandissement egal à k, la dimension de l'image cible passe alors à $(k \times L, k \times H)$:

- k > 1 : agrandissement
- k < 1 : reduction

Il faut alors établir des règles de transformation pour remplir la matrice cible, à partir de certaines des valeurs de l'image source.

Partie 2

Exercices

2.1 Recherche du min/max

Pour parcourir un tableau `T` à 2 dimensions, il faut 2 boucles imbriquées, la 1ere pour parcourir les lignes, et la 2e pour les colonnes.

Cela donne souvent le script suivant :

script A :

```
for i in range(len(T)):
  # i est le numero de ligne
  for j in range(len(T[0])):
    # toutes les lignes ont la meme longueur que T[0]
    # j est le numero de colonne
    ... traitement sur T[i][j]
```

script B :

```
for line in T:
  for elem in line:
    ... traitement sur elem
```

1. Lequel de ces scripts realise un parcours par indice ? Lequel realise un parcours par élément ?
2. Utiliser l'un de ces scripts pour écrire une fonction `min_max`, qui détermine la valeur minimale et la valeur maximale parmi les données de la matrice.
3. Ecrire une fonction `moyenne` qui retourne la moyenne des valeurs d'une matrice.

2.2 Compréhension de liste

Le script suivant construit une matrice `M` de zeros :

```
n = 4
m = 6
M = # a completer (1)
for i in range(n):
    line = []
```

```
6     for j in range(m):
7       line.append(0)
8     # a completer (2)
```

1. Compléter le script
2. Quelle est la dimension de cette matrice ? Et quelles seraient les dimension de l'image correspondante (largeur/hauteur) ?
3. Simplifier le script après la ligne 4, en utilisant l'instruction `[0] * m`, qui est équivalent, ici, à `[0,0,0,0]` . (Ré-ecrire le script)
4. Ré-ecrire ce même script, en utilisant la compréhension de liste.

2.3 Créer une image à partir d'instructions

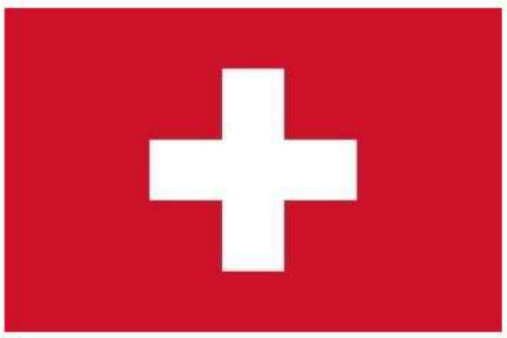

FIGURE 3 – drapeau suisse, croix blanche sur fond rouge

1. Ecrire le script qui permet de construire la matrice de ce drapeau suisse, en noir et blanc. La dimension sera de `700*500` pixels
2. Ecrire le script qui construit la matrice du drapeau, mais cette fois, avec la couleur de fond rouge. Chaque élément de la matrice sera un tuple (R,V,B)

2.4 Traitement d'une image sans modifier sa dimension

On souhaite transformer une image source en couleur, en une image cible en niveaux de gris. On utilise pour cela la formule de calcul de la Luminence : *(r,g,b sont les intensités des couleurs primaires 0..255)*

$$L = 0.299 \times r + 0.587 \times g + 0.114 \times b$$

La valeur calculée pour L sera placée comme niveau de gris dans la matrice cible.

1. Ecrire le script de la fonction luminence, qui calcule la valeur de L à partir d'un tuple `(r,g,b)`
2. Placer dans la fonction un test d'assertion sur les valeurs permises pour r,g,b. Ce test, basé sur les données d'entrée, est appelé test de *précondition*

3. Ecrire une fonction `matrice_rgb_to_gris` qui créé une matrice de niveaux de gris (valeurs de luminence) à partir d'une matrice couleur.

2.5 Agrandissement, reduction

On considère une image source de largeur L et de hauteur H et un entier $k > 0$:

Un *agrandissement* de coefficient `k` de l'image source est une image cible de dimensions $(k \times L, k \times H)$ dont le pixel de coordonnées (x, y) prend la valeur du pixel en $(x//k, y//k)$ dans l'image source.

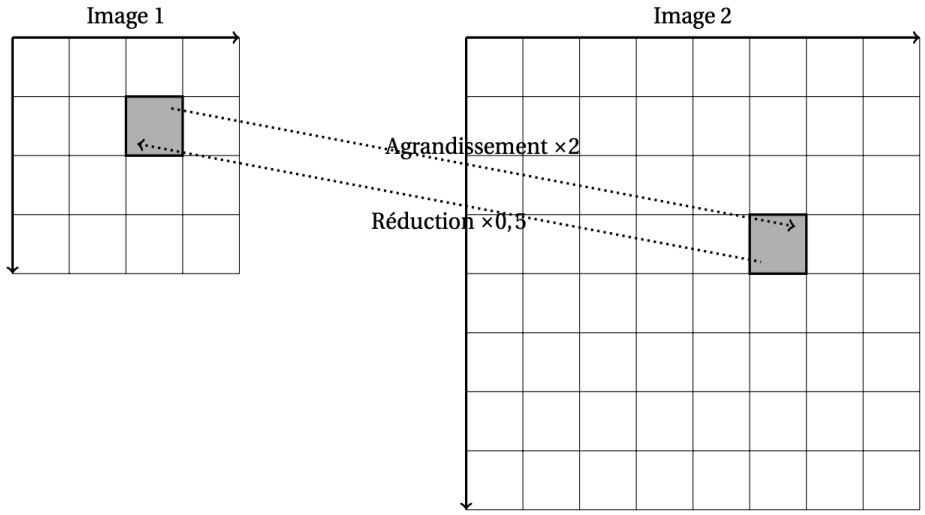

FIGURE 4 – agrandissement - reduction

1. On considère l'image `3*3` de matrice `M = [[1,0,1],[0,1,0],[1,1,1]]`. Quelle sera la dimension de l'image cible si le coefficient `k` est égal à `2` ?
2. Représenter la matrice `M_cible` avec ses valeurs calculées à partir de celles de `M`
3. La fonction `changement_echelle` prend `2` paramètres, `M`, la matrice (L, H) de l'image source, et `k` , le coefficient. Cette fonction retourne une image calculée sur les valeurs de `M`, avec une dimension $(k \times L, k \times H)$. Ecrire le script de cette fonction.
4. Cette fonction, sert-elle aussi pour réaliser une *reduction* de la taille de l'image source ? Donner un exemple.
5. Quel défaut aura une image agrandie selon cette méthode ?
6. Quel sera le poids de l'image reduite d'un facteur k ? Exprimer en fonction du poids P de l'image source.

2.6 Compression, codage de Huffman

Le codage de Huffman est un algorithme de compression de données sans perte. Le codage de Huffman utilise un code à longueur variable pour représenter un symbole de la source (par exemple un caractère dans un fichier).

Le code est déterminé à partir de la frequence d'apparition des symboles de source, un code court étant associé aux symboles de source les plus fréquents.

2.6.1 Fréquence des caractères

1. Compléter la fonction `calculate_frequencies` qui permet de compter le nombre d'occurences des caractères dans un texte. La fonction retourne un dictionnaire.

```
def calculate_frequencies(text):
    frequencies = {}
    for symbol in text:
        if symbol in frequencies:
            ...
        else:
            ...
    return frequencies
```

Exemple d'utilisation

```
text = 'ABACADABRA'
frequencies = calculate_frequencies(text)
frequencies
# affiche
{'A': 5, 'B': 2, 'C': 1, 'D': 1, 'R': 1}
```

2.6.2 Arbre de Huffman

Le code à affecter à chaque lettre est déterminé par sa position dans l'arbre. Précisément, le code d'un symbole de l'alphabet décrit le chemin de la racine à la feuille qui le contient : un 0 indique qu'on descend par le fils gauche et un 1 indique qu'on descend par le fils droit.

Dans le cas de l'arbre ci-contre, le code de X est 00 (deux fois à gauche), le code de Y est 01, et celui de Z est 1.

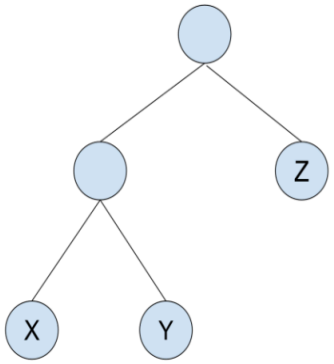

FIGURE 5 – exemple d'arbre de huffman

L'arbre de Huffman est construit à partir de la classe `Node` ci-dessous :

```
class Node:
    def __init__(self, symbol, frequency):
        self.symbol = symbol
        self.frequency = frequency
        self.left = None
        self.right = None

```

```
    def print_tree(self, level=0, prefix="Root: "):

        data = self.symbol

        print(" " * (level * 4) + prefix + str(data))
        if self.left or self.right:
            self.left.print_tree(level + 1, "L--- ")
            self.right.print_tree(level + 1, "R--- ")
```

On fabrique l'arbre de Huffman, appelé `huffman_tree` à partir des caractères et fréquences de ces caractères :

```
symbols = ['X', 'Y', 'Z']
frequencies = [15, 16,40]
```

Puis on affiche une représentation en console de cet arbre avec la méthode `print_tree` :

```
huffman_tree.print_tree()
# affiche
Root: None
    L--- None
        L--- X
        R--- Y
    R--- Z
```

2. Représenter l'arbre de Huffman donné par les listes suivantes. On peut s'aider de sa représentation en console donnée ci-dessous.

```
symbols = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
frequencies = [20, 15, 12, 10, 8, 5, 2]
```

```
huffman_tree.print_tree()
# affiche
Root: None
    L--- None
        L--- B
        R--- None
            L--- None
                L--- G
                R--- F
            R--- E
    R--- None
        L--- A
        R--- None
            L--- D
            R--- C
```

2.6.3 Parcours de l'arbre

3. Donner les codes binaires pour chacun des caractères de l'arbre précédent.

On peut concevoir une fonction de parcours de l'arbre, qui permet de placer les codes binaires de Huffman dans un dictionnaire, où chaque item est du type : `'B':'00` par exemple. Ce sera la fonction `assign_codes` (voir ci-dessous).

Si le noeud contient un symbole, placer ce symbole comme nouvelle clé du dictionnaire `codes`, et la valeur de l'argument `code` comme valeur associée à cette clé.

Sinon, poursuivre le parcours vers le fils gauche, avec `code = code + '0'` puis vers le fils droit avec `code = code + '1'`

4. Compléter le script de la fonction `assign_codes`

```
def assign_codes(node, code, codes):
    # placement des symboles et chemins dans le dict codes
    if node.symbol is not None:
        ...
    else:
        assign_codes(node.left, ..., ...)
        assign_codes(node.right, ..., ...)

codes = {}
assign_codes(huffman_tree, '', codes)

# Affichage des codes attribués à chaque symbole
for symbol, code in codes.items():
    print(symbol + ':', code)
# affiche
B: 00
G: 0100
F: ...
E:
A:
D:
C:
```

2.6.4 Correction

```
def assign_codes(node, code, codes):
    # placement des symboles et chemins dans le dict codes
    if node.symbol is not None:
        codes[node.symbol] = code
    else:
        assign_codes(node.left, code + '0', codes)
        assign_codes(node.right, code + '1', codes)

codes = {}
assign_codes(huffman_tree, '', codes)

# Affichage des codes attribués à chaque symbole
for symbol, code in codes.items():
    print(symbol + ':', code)
# affiche
B: 00
G: 0100
F: 0101
E: 011
A: 10
D: 110
C: 111
```