Images : manipulation, compression, stéganographie

1 Images

On travaillera sur des images bitmap, c'est-à-dire sur des matrices de pixels.

La librairie `Python Imaging Library` (PIL) permet d'importer des images pour les manipuler comme des tableaux de pixels, et ensuite les réimporter.

On pourra commencer le code par

```
import PIL
from PIL import Image

image = Image.open('monImage.jpg')
```

On peut alors récupérer la dimension de l'image par

```
(longueur,largeur) = image.size
```

et la valeur du pixel en position (x,y) par

```
p = image.getpixel(x,y)
```

p est alors un triplet de trois nombres entre 0 et 255 : la valeur du pixel dans ses composantes RGB (si c'est une image couleur RGB, voir `image.mode`).

Un exemple de code complet pour importer une image, et "assombrir" chaque pixel (en divisant par 2 chaque composante).

```
import PIL
from PIL import Image # Pour lire et écrire des images dans un fichier

def noircissement(image) :
    #recuperation des dimensions
    (xmax,ymax) = image.size
    #parcours de l'image
    for x in range(xmax):
        for y in range(ymax):
            #recuperation du pixel
            px = image.getpixel((x,y))
            (r,g,b) = px
            #noircissement du pixel
            px2 = (r//2, g//2, b//2)
            image.putpixel((x,y), px2)


image = Image.open("test.jpg")
noircissement(image)
image.show()
#image.save("testFonce.jpg")
```

Attention, les pixels des images png sont des quadruplets : R,G,B et transparence. On peut les convertir en bitmap ou jpg avant de les ouvrir dans python !

Exercice 1 : Manipulations d'images

1. Ecrire une fonction qui éclaircit une image donnée, sur le même principe que la fonction de noircissement : on rapprochera chaque composante des pixels de 255 en divisant la distance par 2.
2. Calcule le négatif de l'image : la valeur de chaque composante RGB est inversée.
3. Ecrire des fonctions de flip horizontal, de flip vertical, de rotation de 90° à droite (en supposant l'image donnée carrée pour pouvoir le faire en place).
4. Ecrire une fonction `superposition` prenant deux images de mêmes dimensions, et créant l'image moyenne : chaque pixel sera la moyenne des pixels des deux images en même position (pour créer une nouvelle image de même dimension si on ne veut pas supprimer l'une des deux, on pourra utiliser `image.copy()`).

2 Compression

Pour faciliter la compression, on applique une transformation réversible à l'image. On se limite à des images en niveaux de gris, avec kes deux dimensions paires. Chaque couple (x, y) de pixels voisins sera remplacé par $((x+y)//2, y-x)$, La première valeur est donc la moyenne, la deuxième la différence des deux pixels. Si les pixels voisins ont des niveaux de gris proches, la différence sera petite (exploitation de la corrélation entre pixels). Cette transformation est d'abord appliquée colonne par colonne, ensuite ligne par ligne (ou l'inverse). On voit que chaque petit bloc de 2x2 pixels est transformée indépendamment des autres blocs. Le code ci-dessous exploite cela en opérant par bloc de 2x2 pixels. En plus, les 2x2=4 pixels transformées (appelés coefficients) sont regroupés en 4 images de moitié résolution, permettant de visualiser qu'on obtient une image similaire à l'original, ainsi que 3 images de différences. Pour permettre l'affichage, les différences sont augmentées de 128 (zéro sera un gris moyen).

Dans un vrai codeur à ondelettes (comme JPEG 2000), l'image à moitié résolution similaire à l'original est à nouveau transformée, et ceci récursivement sur un petit nombre de niveaux. La transformée dans cet exercice correspond à une ondelette de Haar (moyenne = filtre passe-bas, différence = filtre passe-haut) ; en parique, on utilisera des filtres plus performants en termes de compression et qualité de reconstruction.

```
def s_transform (image) :
    #recuperation des dimensions
    (xmax,ymax) = image.size #parcours de l'image
    res = Image.new('L',(xmax,ymax))
    for x in range(0,xmax,2):
        for y in range(0,ymax,2):
        #recuperation du pixel
            p00 = image.getpixel((x,y))
            p01 = image.getpixel((x,y+1))
            p10 = image.getpixel((x+1,y))
            p11 = image.getpixel((x+1,y+1))
            # vert xform
            s0 = (p00+p01)//2
            d0 = p01-p00
            s1 = (p10+p11)//2
            d1 = p11-p10
            # hor xform
            ss = (s0+s1)//2
            ds = s1-s0
            sd = (d0+d1)//2
```

```
            dd = d1-d0

            xd = x//2
            yd = y//2
            res.putpixel((xd,yd), ss)
            res.putpixel((xd,yd+ymax//2), sd+128)
            res.putpixel((xd+xmax//2,yd), ds+128)
            res.putpixel((xd+xmax//2,yd+ymax//2), dd+128)
    return res
```

La quantification se fait par une division entière par le *pas de quantification*. La fonction suivante comprend aussi la reconstruction après quantification.

```
def quantize (image, q):
    #recuperation des dimensions
    (xmax,ymax) = image.size #parcours de l'image
    for x in range(xmax):
        for y in range(ymax):
        #recuperation du pixel
            px = image.getpixel((x,y))
            image.putpixel((x,y), (px//q)*q)
```

Pour quantifier l'image transformée, il faut tenir compte des valeurs augmentées de 128.

```
def quantize_st (image, q):
    #recuperation des dimensions
    (xmax,ymax) = image.size #parcours de l'image
    for x in range(xmax):
        for y in range(ymax):
        #recuperation du pixel
            px = image.getpixel((x,y))
            if x<xmax//2 and y<ymax//2:
                image.putpixel((x,y), (px//q)*q)
            else:
                image.putpixel((x,y), ((px-128)//q)*q+128)
```

Exercice 2 :

1. Programmez une fonction qui reconstruit une image à partir de sa transformée.
2. Pour estimer le taux de compression possible, on utilisera l'entropie empirique par pixel, calculée à partir d'un histogramme de l'image. Programmez une fonction `entropie(freq)` qui calcule l'entropie à partir de la liste de fréquence. Celle-ci sera obtenue avec `image.histogram()`.
3. Calculez l'entropie de l'image d'origine et de sa transformée, quantifiées et non (4 valeurs en tout).
4. Proposez une formule de reconstruction de la quantification qui donne des meilleurs résultats en terme d'erreur de reconstruction $|x - Q(x)|$. La formule actuelle est $Q(x) = (x//q) \cdot q$.

Remarque On s'est limité à des images en niveaux de gris par simplicité. En pratique, une image couleur RGB ne sera pas compressée bande (couleur) par bande, mais d'abord transformée dans un autre espace couleur, souvent YCbCr. [1] Outre l'avantage d'avoir généralement moins d'entropie,

1. Y est la composante luminance et Cb, Cr sont les composantes chrominance. Historiquement, YCbCr est dérivé du système YUV utilisé dans certaines normes de télévision analogique.

on peut réduire à moitié la résolution des composantes Cb, Cr sans grande perte perceptive, car l'œil est moins sensible aux variations spatiales de la couleur. On divise ainsi le nombre de pixels par deux : une image de taille n^2 comportera $n^2 + 2 \cdot n^2/4 = 1.5n^2$ pixels, à la place de $3n^2$.

Pour vérifier la réduction d'entropie (NB les 3 histogrammes pour les 3 bandes couleur sont regroupés en une liste de longueur 768) :

```
>>> a=Image.open("peppers.png")
>>> a.show()
>>> a.mode
'RGB'
>>> h=a.histogram()
>>> entropy(h[0:256])
7.33882696104637
>>> entropy(h[256:512])
7.496253344995859
>>> entropy(h[512:768])
7.058305579393303
>>> b=a.convert('YCbCr')
>>> h=b.histogram()
>>> entropy(h[0:256])
7.593786990169889
>>> entropy(h[256:512])
5.824147389715153
>>> entropy(h[512:768])
6.66795646853679
```

3 Steganographie

Chaque teinte de chaque pixel est codé par un octet, soit un nombre entre 0 et 255.

Les premiers bits de cet octet ont beaucoup plus de poids que les derniers ; il est possible de changer les derniers bits sans que le changement soit significativement perceptible ; et utiliser ces bits pour dissimuler une autre information.

Exercice 3 : Dissimulation d'un QR code

1. Créer un QRcode en ligne, par exemple ici : `https://www.unitag.io/fr/qrcode` (on peut obtenir un qrcode 300*300 noir et blanc png, à convertir en bmp). Choisir une autre image `img` de même dimension (quitte à la redimensionner).

   Exemple de redimensionnement :

   ```
   (xmax,ymax) = image.size
   image = image.resize((xmax*2,ymax*2))
   ```

2. Ecrire une fonction qui prend les deux images, et pour chaque pixel, remplace le dernier bit du rouge de chaque pixel par 0 ou 1, suivant que le pixel du qrcode est noir ou blanc. Sauvegarder l'image (em bmp !).
3. Ecrire la fonction de décodage, prenant l'image dissimulant le QR code, et qui retourne le QR code.

Exercice 4 : Superposition de deux images

On reprend la même idée que précédemment, mais on cherche à dissimuler une image sous une autre image - en diminuant la résolution.

L'idée vient d'ici : `http://www.bibmath.net/crypto/index.php?action=affiche&quoi=stegano/cacheimage`

1. Ecrire une fonction qui prend deux images de même taille, et retourne une image superposée. Pour chaque teinte de chaque pixel, l'image superposée aura les 4 bits de poids fort de la première image, suivi des quatre bits de poids fort de la deuxième image (on oublie les bits de poids faible de chaque image).
2. Ecrire une fonction de décodage prenant en entrée l'image superposée, et retournant les deux images séparées. On pourra mettre des 0 comme bits de poids faible.

Exercice 5 : Dissimulation d'un texte dans une image

On peut aussi dissimuler une information de toute autre nature : par exemple un texte, que l'on supposera encodé en caractères ascii.

Pour ouvrir un fichier texte au format ascii :

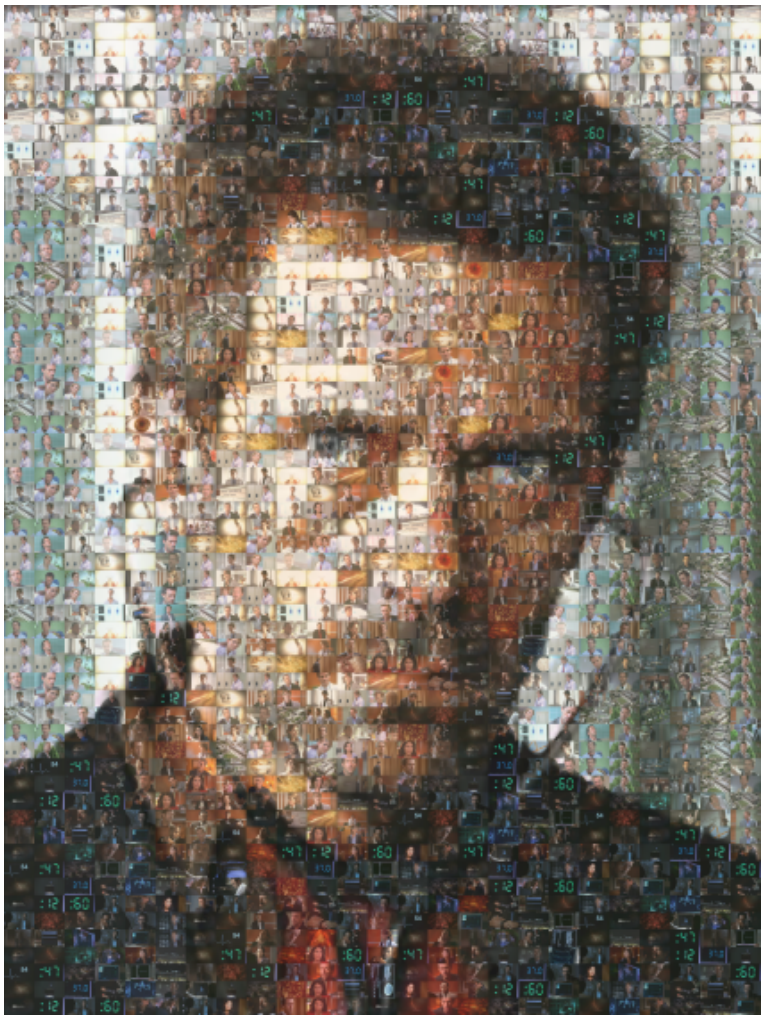
```
fichier=open("monFichier.txt",'r')
s=fichier.read()
fichier.close()
```

Pour convertir chaque caractère en son code ascii, et réciproquement :

```
>>> ord("a")
  97
>>> chr(97)
  'a'
```

1. Ecrire une fonction prenant en entrée une image et une chaîne de caractères, et créant une nouvelle image où les 4 bits de poids faible de chaque teinte de l'image sont utilisés pour dissimuler le texte.
   — Pour faciliter les manipulations, on pourra ajouter la fin du texte des espaces en nombre suffisant pour que "ça tombe juste" : que la quantité de texte à coder occupe toute la place disponible dans l'image.
   — On pourra par exemple, parcourir les pixels dans l'ordre de balayage habituel, et cacher successivement dans le rouge, le vert et le bleu.
2. Ecrire une fonction de décodage prenant l'image dissimulant le texte en entrée, et retournant la chaîne de caractères du texte original.

4 Idée de projet : mosaïque d'images



Il s'agit d'une idée n'ayant pas été implémentée par moi-même, ni testée sur étudiants, et dont le descriptif ci-desssous n'est qu'une esquisse.

La beauté du résultat n'est pas garantie !

4.1 Objectif du projet :

Créer un logiciel prenant une grande image et une grande banque de petites images, et réalisant de matière automatique une "mosaïque" dans laquelle on reconnaît la grande image, rien qu'à l'aide du choix de la disposition des petites images.

4.2 Les contraintes sont :

— on ne doit pas retoucher les petites images dans la mosaïque,
— on doit utiliser chaque petite image au plus une fois.

4.3 Principal problème algorithmique :

Le problème principal à résoudre est de trouver l'agencement des petites images optimal : trouver, pour chaque zone de la grande image, la petite image qui lui correspond le plus.

Pour cela, on peut :

— calculer une "distance" séparant chaque petite image de chaque petite zone de la grande image : mesurer les écarts en rouge/bleu/vert (RGB), ou en d'autres représentations des pixels, tels que la représentation Teinte, Saturation, Luminosité (TSL). C'est ici que l'on pourra sans doute le plus jouer pour obtenir un résultat le plus joli : on pourra tester différences pondérations de ces paramètres, tester différentes représentations des pixels...
— on est alors ramené à un problème d'appariement : trouver l'appariement petites images-zones de la grande image qui minimise la somme des distances. Ce problème est connu et a un algorithme optimal en temps raisonnable, l'agorithme hongrois :
`https://fr.wikipedia.org/wiki/Algorithme_hongrois`

4.4 Variantes :

On peut simplifier beaucoup le problème algorithmique en travaillant par exemple avec des images noir et blanc, et en remplaçant chaque pixel de la "grande" image par une image de la petite variante.

Pour trouver la petite image "optimale" à chaque emplacement, il suffit alors de calculer la "teinte" moyenne de chaque image, de les classer de la plus sombre à la plus claire, et d'apparier ainsi chaque pixel de la grande image avec la petite image de teinte la plus proche.

Esthétique du résultat non garantie !