Manipulations d'images en Python

Raymond Namyst et Marc Zeitoun

9 mai 2014

1 Les images en couleur.

Nous allons maintenant manipuler des images en couleur. À nouveau, voyons comment l'ordinateur « voit » et manipule de telles images. Récupérez d'abord le fichier `image.png` dont le lien est donné ici. Enregistrez l'image et regardez-la en double-cliquant dessus dans l'explorateur de fichiers. Vous devriez voir qu'elle est constituée d'une multitude de *pixels* organisés en lignes (`y`) et colonnes (`x`), tout comme le sont les images en niveaux de gris.

Pour représenter assez finement pour l'œil humain un pixel en niveaux de gris, on n'a besoin que d'une seule valeur entre 0 et 255 (0 : noir, 255 : blanc). Dans une image couleur, on indique pour chaque pixel un *mélange de 3 couleurs* : rouge, vert, bleu. Chaque couleur est donnée par un nombre entre 0 et 255, on a donc besoin de **3 nombres** (qu'on représente chacun sur un octet). Ce codage s'appelle R G B pour Red Green Blue et où pour chacune de ces couleurs

— la valeur 0 indique qu'on met 0% du « pot de cette couleur ».
— la valeur 255 indique qu'on met 100% du « pot de cette couleur ».

Pour illustration, voici un zoom sur l'image `image.png` qui montre quelques valeurs associées aux pixels :

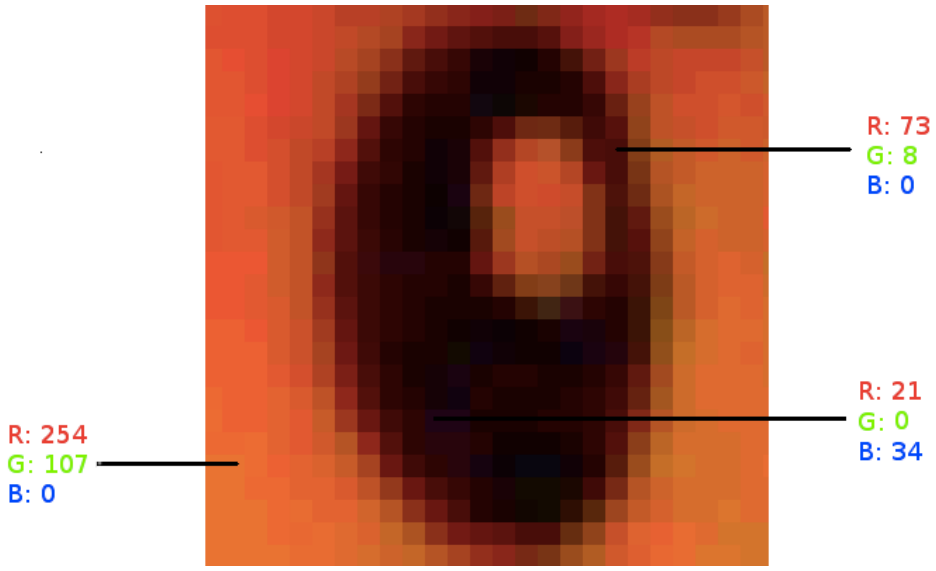


FIGURE 1 – Zoom sur une image en couleur

Voici quelques exemples pour bien comprendre le codage des couleurs :

couleur	R	G	B	Octet 1	Octet 2	Octet 3
	0	0	0	00000000	00000000	00000000
	255	255	255	11111111	11111111	11111111
	255	0	0	11111111	00000000	00000000
	0	255	0	00000000	11111111	00000000
	0	0	255	00000000	00000000	11111111
	132	122	191	10000100	01111010	10111111

2 Les images en couleur et Python

Dans l'environnement Python, on charge en mémoire l'image (fonction `open`) puis on l'affiche (fonction `show`) de la façon suivante :

```
from PIL.Image import *

i = open("image.png")
Image.show(i)
```

On peut ensuite
- récupérer la valeur des 3 couleurs R,G,B de n'importe quel pixel de l'image, grâce à la fonction `Image.getpixel`.
- modifier ces valeurs, et donc modifier la couleur du pixel.

Pour récupérer la valeur des 3 couleurs d'un pixel, on peut par exemple écrire ceci :

```
# Récupère dans les variables rouge, vert et bleu les 3 composantes
#       de couleur du pixel en ligne 10 et colonne 42
# NB: les indices des lignes et colonnes commencent à partir de 0.
# Le point (0,0) est en haut à gauche.
x = 10
y = 42
(rouge,vert,bleu) = i.getpixel((x,y))
print (rouge, vert, bleu)
```

Pour modifier la couleur d'un pixel, on peut utiliser la fonction `Image.putpixel`, comme ceci :

```
# Modifie la couleur du pixel en ligne 10 et colonne 42.
# NB: les indices des lignes et colonnes commencent à partir de 0.
x = 10
y = 42
(rouge,vert,bleu) = (132,122,191)
i.putpixel((x,y),(rouge,vert,bleu))
```

1. Pour vous échauffer, écrivez une fonction qui travaille sur une image `i`, et supprime toute la couleur rouge de chaque pixel. Pour parcourir tous les pixels, on peut effectuer la double boucle `for` suivante :

```
(largeur, hauteur)= i.size
 for x in range(largeur):
     for y in range(hauteur):
        # Ici on traite le pixel (x,y) de l'image i.
```

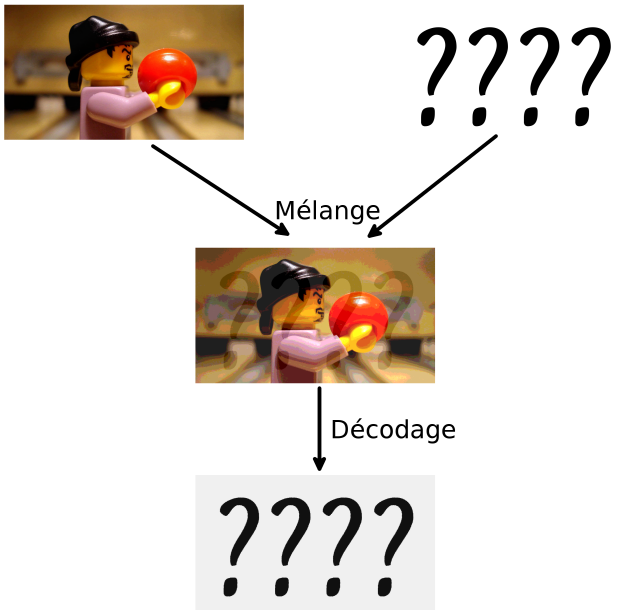
Dans la double boucle ci-dessous, pensez-vous que les pixels sont parcourus ligne par ligne ou par colonne par colonne ?

3 Cacher une image dans une autre

L'image que vous avez chargée contient une autre image cachée que nous allons essayer de découvrir.

Des œufs en chocolat à gagner pour celles/ceux qui trouveront l'image cachée !

L'image a été cachée par nos soins en changeant légèrement la couleur de certains pixels de l'image originale. Tout se passe comme si l'image cachée était « sous » l'image visible :



En fait, on mélange les pixels des deux images originales, en privilégiant l'image 1 pour qu'on ne remarque pas que l'image 2 est cachée dedans. On utilise le fait que modifier légèrement la couleur des pixels d'une image est difficile à discerner pour l'œil humain. Par exemple, le mot

a b r a c a d a b r a

ne contient pas deux lettres de même couleur, ce qui est difficile à voir. Les couleurs employées sont trop proches les unes des autres pour pouvoir être distinguées. En résumé, modifier légèrement les pixels d'une image ne se voit pas facilement.

3.1 Principe pour cacher une image dans une autre

On utilise le fait que l'œil ne distingue pas les faibles différences de couleur pour cacher une image dans une autre. Pour simplifier, voyons ce qui se passe sur une seule composante de couleur, disons le rouge (même si en fait, on appliquera le même traitement aux trois couleurs).

Prenons 2 images de même dimension. Pour chacune des coordonnées `(x,y)`, on va mélanger une partie du pixel `(x,y)` de l'image 1 avec le pixel `(x,y)` de l'image 2. L'image obtenue sera très proche de l'image 1, mais l'image 2 sera « cachée » à l'intérieur. Imaginons qu'on veut mélanger un pixel dans l'image 1 dont l'octet rouge vaut **190** c'est-à-dire

10111110

avec l'octet de l'image 2 qui vaut **121**, c'est-à-dire

01111001

L'idée est de recombiner les 5 chiffres les plus significatifs du premier pixel avec les 3 chiffres les plus significatifs de l'image 2. Les chiffres significatifs sont les plus importants, ceux qui se trouvent à gauche. Au contraire, les unités ne sont pas très importantes pour nous. De nos 2 octets d'origine

10111 110 et 011 11001

on ne garde que les chiffres les plus significatifs, en mettant ceux de la première image en tête. Ce mélange des deux octets d'origine produit l'octet suivant :

10111 011

On remarque qu'on a **peu modifié la valeur par rapport à celle de l'image 1**, parce qu'on a gardé les 5 bits les plus significatifs. Sur notre exemple, la différence est seulement de 3 : la nouvelle valeur est 187, ce qui donne un rouge **comme ce texte** au lieu de 190 dans l'image originale, ce qui donne un rouge **comme celui-ci** qui est très proche à l'œil nu. Au pire, on transforme les 3 derniers bits de **000** à **111**, ou vice-versa, ce qui fait une différence de 7, au plus. En résumé, la fusion des 2 images ressemble beaucoup à la première image.

Qu'a-t-on gagné ? Ce qu'on a gagné est qu'à partir de l'octet qu'on a créé dans l'image transformée :

10111 011

1. on n'a pas beaucoup modifié la 1ère image, et
2. on peut retrouver une couleur proche de celle du pixel **de la 2ème image** grâce aux 3 derniers bits, 011. On sait donc que la valeur du pixel original de la 2ème image se situe

entre 011 00000 et 011 11111.

L'erreur est au maximum de 32. Pour reconstruire approximativement le pixel original, on peut compléter la valeur de façon intermédiaire, en ajoutant 16. À partir d'un pixel de l'image modifiée, on reconstruit donc l'approximation du pixel de l'image cachée :

01110000

Ce nombre vaut **112** au lieu de **121** dans l'image 2 qu'on voulait cacher. On a donc fait une erreur de 9 qui reste acceptable. Au pire, on fera une erreur de 16. Ça reste suffisant pour retrouver l'image cachée, même si la perte de précision la dégradera.

3.2 Comment récupérer l'image cachée ?

Comme on a caché l'image en utilisant chaque pixel, vous devez faire un traitement sur chaque pixel de l'image. Il faut donc récupérer les dimensions de l'image, exprimées en nombre de pixels, et décoder chaque pixel. Pour cela, après avoir ouvert l'image originale, on peut faire une double boucle `for` comme indiqué dans la 1ère question.

1. Écrivez d'abord une fonction `valeur_derniers_bits(n)` qui, à partir d'une valeur `n` codée sur un octet, retourne la valeur correspondant aux 3 derniers bits. Par exemple, si on transmet 187 à cette fonction, dont le code en binaire est `10111011`, elle devra retourner le code des 3 derniers bits, c'est-à-dire `011` qui vaut 3.
2. Écrivez une fonction `décaler(m)` qui, à partir d'un entier `m` dont le code se représente sur 3 bits, retourne la valeur correspondant à ces 3 bits complétés par `10000`. Par exemple, partant de 3 qui est codé sur 3 bits par `110`, la fonction retournera 112, codé par `11010000`.
3. Utilisez les fonctions précédentes pour décoder l'image de départ.
4. Inversement, écrire une fonction qui cache une image dans une autre.