Introduction au traitement d'image

1 Manipulation d'une image en couleurs

1.1 Chargement de l'image

En informatique, une image en couleurs est constituée d'un ensemble de pixels colorés, arrangés en lignes et en colonnes. Chaque pixel est, généralement, constitué d'un mélange de trois couleurs : rouge, vert et bleu.

Nous allons utiliser le module `imageio` pour charger en mémoire une image.

```
In []: import imageio
```

On peut procéder de la sorte pour charger une image `ballons.jpg` qui se trouverait dans le répertoire `images` sur le disque `D:` :

```
In []: image = imageio.imread('D:images/ballons.jpg')
```

Python utilise un tableau à trois dimensions pour représenter l'image en couleur. Comme pour une matrice, le premier indice désigne la ligne, la seconde la colonne. Le troisième indice désigne le canal de couleur (dans l'ordre, rouge, vert et bleu). Ainsi,

```
In []: image[4, 11, 2]
Out[]: 37
```

désigne la couleur bleue du pixel situé à la cinquième ligne et à la douzième colonne (on rappelle que les indices débutent à 0). Chacune des valeurs est un entier sur 8 bits, entre 0 et 255. 0 représente l'absence de la couleur concernée, 255 le maximum. Si les trois valeurs de couleur sont à 0, on obtient du noir. Si elles sont toutes les trois à 255, du blanc. Pour obtenir du rouge, il faut une valeur 255 pour la valeur rouge, et 0 pour les deux autres.

Pour éviter les problèmes de débordement, plutôt que d'utiliser des entiers entre 0 et 255, nous travaillerons plutôt avec des flottants entre 0.0 et 1.0 (pour que l'image s'affiche correctement, il faudra que ces valeurs soient bien dans l'intervale $[0.0, 1.0]$, mais au moins, si un résultat intermédiaire n'est pas dans cet intervale, cela n'aura pas de conséquences fâcheuses). Pour convertir l'image en une représentation flottante, on utilisera :

```
In []: image = image / 255.0
```

On pourra vérifier que les valeurs sont à présent des flottants entre 0.0 et 1.0 :

```
In []: image[4, 11, 2]
Out[]: 0.1450980392156863
```

1.2 Modification d'une image

On peut modifier la valeur correspondant à l'intensité d'une couleur d'un pixel par mutation de la case correspondante du tableau, par exemple de la sorte :

```
In []: image[4, 11, 2] = 0.2
```

Cette commande attribue, pour le pixel situé sur la ligne d'index 4 (la cinquième ligne), dans la colonne d'index 11, une intensité égale à 0.2 pour le canal d'index 2 (le troisième canal de couleur, donc, correspondant au bleu).

On peut également directement attribuer les trois composantes de couleur :

```
In []: image[4, 11] = 0.5, 0.9, 0.2
```

Si, dans un algorithme, il est besoin de connaître la taille de l'image, on rappelle que `image.shape` fournit un tuple de trois éléments, conprenant la hauteur de l'image, sa largeur, et le nombre de canaux de couleur (3, en principe).

1.3 Affichage d'une image

Pour afficher une image ayant trois canaux de couleur, il suffit de faire appel à `matplotlib` :

```
In []: import matplotlib.pyplot as plt

In []: plt.imshow(image)

In []: plt.show()
```

Sur certaines machines, la dernière instruction n'est pas indispensable (elle force simplement l'ouverture de la fenêtre).

2 Traitements basiques de l'image

Dans un premier temps, nous allons écrire quelques fonctions qui *modifient* une image, passée en argument de la fonction. Comme les listes, les `numpy.array` – et donc les images – sont des objets mutables, que l'on peut directement modifier dans les fonctions. Les fonctions n'ont donc, en principe, pas besoin de retourner de résultat.

2.1 Premières manipulations

1.a Écrire une fonction `AjouteRectangle(img)` qui prend en argument une image, et modifie cette image de façon à ajouter sur celle-ci un rectangle vert de hauteur 80 et de largeur 160 dont le coin en haut à gauche se trouve à la ligne 50 et à la colonne 100.

1.b Charger une image, utiliser la fonction précédente, et afficher le résultat à l'écran.

2. Écrire une fonction `AjouteDisque(img)` qui prend en argument une image, et modifie cette image de façon à ajouter sur celle-ci un disque bleu de rayon 75 dont le centre se trouve à la ligne 100 et à la colonne 200, puis tester son bon fonctionnement.

2.2 Inversion

L'inversion d'une image consiste à remplacer, pour chaque pixel et pour chaque canal de couleur, une valeur v par $1.0 - v$ (ou $255 - v$ si l'on travaillait avec des valeurs entières). Cela a pour effet de transformer le blanc en noir (et inversement), le rouge en cyan, le vert en magenta, etc.

3. Écrire une fonction `Inverse(img)` qui prend en argument une image et l'inverse, et tester son bon fonctionnement.

2.3 Retournement

Le retournement d'une image consiste à effectuer une symétrie autour de la ligne médiane de l'image. Les pixels de la dernière ligne se retrouvent sur la première ligne (et inversement), et ainsi de suite.

4. Écrire une fonction `Retourne(img)` qui prend en argument une image et la retourne, et tester son bon fonctionnement.

2.4 Passage en niveaux de gris

Pour obtenir une image en niveaux de gris, c'est-à-dire dépourvue de couleur, il suffit que les valeurs des trois canaux de couleur de chaque pixel soient égales. Attention, cela reste une image en couleur, même si tous les pixels sont gris (on n'aura donc pas besoin de l'argument `cmap="gray"` ici).

Pour convertir une image couleur en niveaux de gris, il faut donc remplacer les valeurs des trois canaux par une unique valeur représentant la luminosité du pixel.

Il existe plusieurs façons de calculer cette valeur, selon l'image que l'on souhaite obtenir à l'arrivée. Lorsque l'on souhaite se rapprocher le plus possible de la vision humaine, on peut utiliser la formule suivante (qui privilégie le vert car c'est la couleur à laquelle notre œil est le plus sensible) :

$$0.2126 \times \text{Rouge} + 0.7152 \times \text{Vert} + 0.0722 \times \text{Bleu}$$

On remarquera que la somme des trois coefficients est égale à 1.0, ce qui permet de s'assurer que le résultat ne dépassera pas 1.0, et atteindra la valeur de 1.0 pour un pixel où les trois composantes de couleur égales à 1.0.

5. Écrire une fonction `Monochrome(img)` qui prend en argument une image en couleurs et la transforme en une image en niveaux de gris selon la formule précédente, et vérifier son bon fonctionnement.

On pourra ensuite choisir d'autres coefficients pour varier le résultat obtenu (pour un bon fonctionnement, la somme des trois coefficients devrait être égale à 1.0).

2.5 Modification du contraste

Certaines images peuvent être surexposées (trop claire, les valeurs des différents canaux de couleur étant toutes grandes), sous-exposée (même chose avec de petites valeurs), ou peu contrastée (toutes les valeurs sont proches).

Corriger le contraste d'une image consiste à modifier la répartition des valeurs entre 0.0 et 1.0 dans chacun des canaux. Cela consiste à appliquer, pour chaque canal de chaque pixel, une fonction définie de la sorte :

- si la valeur v est inférieure à `vmin`, on la remplace par 0.0 ;
- si la valeur v est supérieure à `vmax`, on la remplace par 1.0 ;
- sinon, on remplace la valeur v par (v – vmin) / (vmax – vmin).

On supposera, par simplicité, que l'on utilise la même formule pour chacun des trois canaux de couleurs et pour chacun des pixels. Il est fréquent d'utiliser des fonctions différentes pour chaque couleur, ce qui permet de corriger la teinte d'une image (par exemple une image rougie car prise sous un éclairage artificiel). Il arrive parfois que l'on utilise des fonctions différentes selon la zone de l'image, par exemple lorsque l'on veut obtenir un effet « HDR » (haute dynamique). Cela permet de ne pas avoir, sur une même image, des zones sous-exposées et des zones sur-exposées, ce qui est fréquemment le cas pour les images prises à l'extérieur.

6. Écrire une fonction `AugmenteContraste(img, vmin, vmax)` qui modifie le contraste d'une image, et vérifier son bon fonctionnement. On pourra choisir par exemple comme paramètres vmin = 0.2 et vmax = 0.8.

3 Application de filtres

Nous avons vu beaucoup de traitements qui s'appliquaient séparément sur chaques pixels. Mais pour obtenir une gamme plus large d'effets, on peut envisager que les nouvelles couleurs d'un pixel dépendent non seulement des anciennes couleurs de ce pixel, mais également des pixels voisins.

Dans cette situation, il n'est pas possible de modifier directement l'image (pourquoi ?) Il faut donc créer une nouvelle image vide, dont on pourra ensuite fixer la couleur de chaque pixel sans toucher à l'image originale.

Les fonctions suivantes vont donc *retourner* une *nouvelle* image qui sera le résultat du calcul souhaité, au contraire des fonctions précédentes qui modifiaient leur argument.

On rappelle que pour créer une image couleur (initialement noire) de hauteur `H` et de largeur `L`, on peut utiliser la commande :

```
In []: image = numpy.zeros((H, L, 3))
```

Attention, les fonctions suivantes risquent de demander beaucoup de temps de calcul pour des images de grande taille. Il est possible de grandement accélérer les calculs en écrivant les choses différemment, mais le principal objectif aujourd'hui est de parvenir à écrire des fonctions qui effectuent correctement les tâches recherchées.

3.1 Flou

Pour appliquer un flou à l'image, on remplace chaque couleur de chaque pixel par une moyenne des couleurs des neufs pixels entourant le pixel considéré.

1. Écrire une fonction `Floute(img)` qui prend en entrée une image `img` et retourne une *nouvelle* image correspondant à un flou de l'image originale, et vérifier son bon fonctionnement. On réfléchira à ce qu'il est possible de faire sur les bords de l'image.

Pour un flou plus prononcé, on peut appliquer plusieurs fois la fonction `Floute`, ou bien utiliser une zone de plus de 9 pixels autour de chaque pixel considéré.

3.2 Augmentation de netteté

Pour essayer d'augmenter la netteté d'une image, on peut procéder de la même façon, en remplaçant par exemple la couleur d'un pixel par le double du pixel auquel on a soustrait la moyenne des huit pixels avoisinants.

2. Écrire une fonction `AugmenteNettete(img)` qui prend en entrée une image `img` et retourne une *nouvelle* image correspondant à une augmentation de la netteté de l'image originale, et vérifier son bon fonctionnement.

3. Pourquoi apparaît-il quelques pixels étranges, et comment y remédier ?

Évidemment, il est difficile de rendre nette une image initialement floue, et il y a des limites à ce que l'on peut faire en pratique (ne prenez pas pour argent comptant les téléfilms américains où on rend nette une plaque minéralogique à partir d'une image satellite floue !). Il est en effet impossible de retrouver de l'information qui a été perdue. On pourra le vérifier en essayant de rendre plus nette une image en appelant plusieurs fois de suite la fonction précédente.

3.3 Détection de contours

Il est possible de détecter les contours d'une image en déterminant, pour chaque pixel, la norme du gradient en ce point. En effet, un gradient élevé correspond à une brusque variation de la luminosité ou de la couleur, ce que l'on trouve en général au bord des objets.

4. Écrire une fonction `Contours(img)` qui prend en entrée une image `img` et retourne une *nouvelle* image pour lequel la luminosité de chaque pixel est, en chaque point, proportionnelle à la norme du gradient en ce point (on pourra travailler couleur par couleur), et vérifier son bon fonctionnement.

Le gradient est un opérateur vectoriel qui sera étudié et utilisé notamment en physique, et est une généralisation de la notion de dérivée dans un espace à plusieurs dimensions.

Dans le cas qui nous occupe, on pourra estimer sa norme en prenant la racine carré de la somme

- du carré de la différence entre l'intensité du point (i+1, j) et de celle du point (i, j) ;
- du carré de la différence entre l'intensité du point (i, j+1) et de celle du point (i, j).

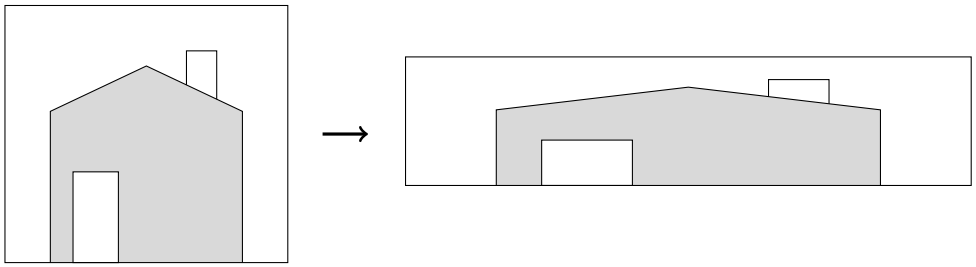
On multipliera le résultat par $\sqrt{2.0}$ (soit `2.0**(-0.5)`) pour éviter de dépasser la valeur 1.0. On fera cette opération sur chacun des canaux de couleur. On pourra retourner une image contenant une ligne et une colonne de moins que l'image originale pour éviter les problèmes sur les bords droit et inférieur.

Une fois cette image calculée, on peut rendre les bords plus nettement visibles en augmentant le contraste, en utilisant par exemple la fonction `AugmenteContraste`. On prendra par exemple `vmin=0.4` et `vmax` très légèrement supérieur (`vmin+0.01` par exemple). Le choix de la valeur de `vmin` dépend de l'image, donc essayez différentes valeurs si le résultat n'est pas satisfaisant.

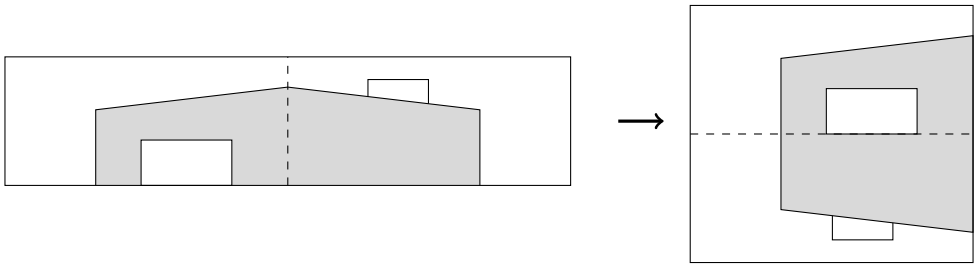
4 Transformation du boulanger

On s'intéresse à une transformation inversible déplaçant les pixels à l'intérieur d'une image. Cette transformation, inspirée par la façon dont un boulanger travaille sa pâte, peut être décrite en deux étapes successives.

Dans un premier temps, on crée une image deux fois plus large et deux fois moins haute, en étirant l'image initiale :

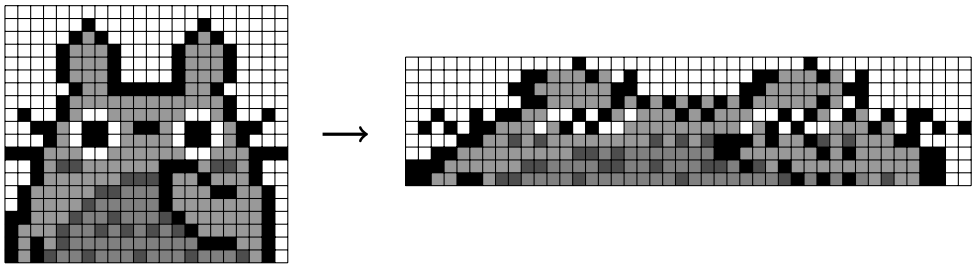


Dans un second temps, on coupe la moitié droite de la nouvelle image, et on la recolle, retournée à 180 °, sous la moitié gauche, de façon à obtenir une image de mêmes dimensions que l'image initiale :

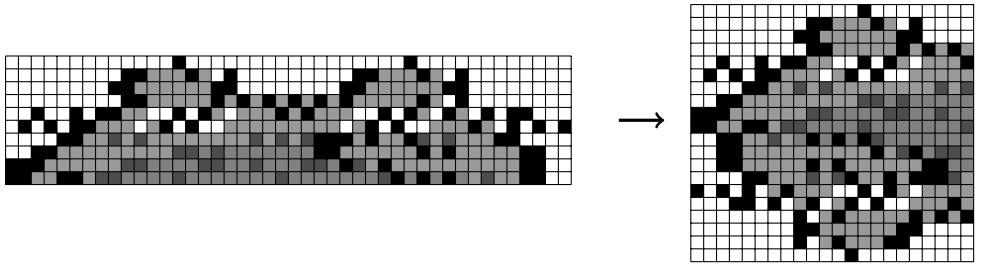


Pour simplifier, on supposera que l'image initiale contient un nombre pair de lignes. Pour construire la première ligne de l'image « aplatie », on utilise les deux premières lignes de l'image originale : on prend le premier pixel de la première ligne, puis le premier pixel de la seconde ligne, puis le second pixel de la première ligne, le second pixel de la seconde ligne, et ainsi de suite.

Pour la seconde ligne de l'image « aplatie », on utilise les troisième et quatrième lignes de l'image originale, et on continue de la sorte pour le reste de l'image :



La seconde partie de la transformation ne pose pas de problème particulier :



1. Écrire une fonction `Boulange(image)` qui prend en argument une image et retourne une nouvelle image, résultat de l'application de la transformation du boulanger à l'image passée en argument.

Note : si la transformation a été décrite en deux étapes, il n'est pas nécessaire de programmer explicitement les deux étapes si vous voyez une façon d'effectuer la transformation plus « directement ».

2. Appliquer la fonction à une image plusieurs fois de suite et vérifier que l'on finit par ne plus identifier l'image.

3. Justifier que si l'on applique suffisamment de fois l'opération, on doit retrouver l'image initiale.

4. Construire une imagette de taille 128×128 en extrayant un morceau d'une des images fournies, et vérifier qu'après un certain nombre d'applications de la transformation, on retrouve l'image initiale.

On souhaite déterminer le nombre de transformations nécessaires pour retrouver l'image initiale pour une image de taille $h \times w$. On fournit une fonction construisant une image de hauteur `h` et de largeur `l` telle que la valeur dans le canal rouge correspond au numéro de ligne et celle dans le canal vert le numéro de colonne. Chaque pixel est donc différent.

```
def GenImage(h, l) :
    return np.fromfunction(np.vectorize(lambda i, j, k : [ i, j, 0 ][k]),
                                (h, l, 3), dtype=int)
```

5. Construire un tableau de taille $h \times l$ où chaque case (i, j) contient le nombre de transformations nécessaire pour que le pixel initialement à la position (i, j) retourne à cette position.

6. Comment déduire de ce tableau le nombre de transformations nécessaires pour retrouver l'image initiale ? Le PGCD de deux entiers peut être obtenu grâce à la fonction `gcd` du module `math`.

7. Retrouver le nombre de transformations nécessaire pour une image de taille 128×128. Qu'en est-il pour l'image utilisée depuis le début de cette séance ?