Thème info 3 – Traitement d'images matricielles

I - Images vectorielles vs images matricielles

Certains logiciels de dessin assisté par ordinateur permettent de créer des images *vectorielles*, c'est-à-dire que ce sont des informations permettant de redessiner l'image qui sont mémorisées. Cela permet de conserver la qualité du tracé quel que soit le redimensionnement de l'image. Les fichiers créés sont peu volumineux, mais l'affichage et le recalcul des images demandent une certaine puissance de calcul.

Les formats d'images vectorielles les plus connus sont PDF, PostScript, SVG.

Parmi les utilisations les plus répandues, les polices de caractères et les cartes routières ou géographiques, deux types d'images sur lesquelles on est amené à zoomer sans cesse...

A contrario, les images *matricielles* ("*bitmap*" en anglais, mais le mot est dans l'Officiel du Scrabble francophone !) sont sauvegardées — comme leur nom l'indique — sous forme d'un tableau de *pixels*. Chaque pixel correspond à un point de l'image affichée (ou imprimée). La taille finale d'un pixel dépend alors du facteur d'agrandissement, d'où un effet de *pixélisation* si l'image est trop agrandie (image formée de "pavés" visibles, effet d'escalier le long des courbes...), défauts invisibles à l'œil nu si le nombre de pixels par unité de longueur est suffisant.

En contrepartie, le principal avantage des images matricielles est un affichage très rapide, puisque les informations stockées dans le fichier pour chaque pixel sont directement interprétables, typiquement le triplet des *composantes de couleurs* RVB (rouge, vert, bleu, RGB en anglais) pour les images en couleurs, ou encore une simple valeur pour les images en *niveaux de gris*. Mais ce principe de stockage induit des fichiers de grande taille pour des images en haute définition, notamment pour imprimer des images de bonne qualité. C'est pourquoi diverses astuces de réduction de la taille du fichier ont été élaborées : utilisation de palettes de couleurs (la couleur d'un pixel étant alors caractérisée par un seul entier, son rang dans la palette, mais cela limite le nombre de couleurs utilisées dans l'image), compression des données, etc.

Le nombre de couleurs standard actuellement correspond au nombre de triplets d'entiers de $[\![0, 255]\!]$, soit $256^3 = 2^{24} = 16\,777\,216$ (d'où la dénomination commerciale "16 millions de couleurs"). De même le standard en noir et blanc est de 256 niveaux de gris.

Les formats d'images matricielles les plus connus sont BMP, GIF, JPEG, PNG, TIFF. Chacun de ces types de fichiers a ses propres spécifications : en général, les informations sur le mode de stockage se trouvent en en-tête du fichier, suivies des données proprement dites. D'où l'intérêt des fonctions de lecture et d'enregistrement de fichier qui se chargent de "décrypter" l'en-tête à la lecture et de le coder à l'enregistrement, selon les normes du type de fichier choisi. Ces opérations sont plus techniques que la lecture ou l'écriture d'un simple fichier texte !

Les images matricielles se sont beaucoup répandues avec l'avènement des capteurs présents dans les scanners, les appareils photos numériques !

Les activités proposées par la suite montrent quelques exemples de traitement d'images numériques matricielles, tout d'abord pixel par pixel, puis par des méthodes de convolution.

Pour alléger les programmes, dans la mesure où le traitement d'une image en couleurs consiste à appliquer trois fois un même traitement aux différentes composantes de couleurs, les exemples traiteront une image en niveaux de gris.

II - Utilisation du module image de matplotlib

Pour des manipulations avancées d'images, il existe un module historique, développé à l'origine par le MIT, nommé `PIL` pour "Python Imaging Library". Le projet a été repris par une autre équipe sous le nom de `Pillow`.

Mais comme nous allons manipuler nous-mêmes les données, nous pourrons nous contenter des fonctions basiques du module `image` de `matplotlib`. Comme l'affichage sera effectué par `pyplot` et comme nous utiliserons les tableaux `numpy`, nous commencerons le script par :

```
import matplotlib.image as img
import matplotlib.pyplot as plt
import numpy as np
```

Les commandes suivantes seront utiles :

- `im=img.imread('Image_gray.png')` place dans `im` un tableau de type $(n, p, 3)$, où n (resp. p) est le nombre de lignes (resp. colonnes) de pixels dans l'image. La troisième dimension fournit pour chaque pixel les trois composantes RVB. Noter que, pour une image en niveaux de gris, les 3 composantes existent bien mais sont égales. On peut donc se ramener à un simple tableau de type (n, p) en "sliçant" : `im0=im[:,:,0]`. Pour remplir un tableau de niveaux de gris à partir d'une image en couleurs, il suffit d'affecter à chaque pixel la moyenne de ses trois composantes couleurs.

  N.B. : nativement, `matplotlib` ne sait lire et convertir que le format PNG ; toutefois, si le module `Pillow` est installé, `imread` pourra ouvrir d'autres types de fichiers.

- `img.imsave('Image.jpg',im)` enregistre l'image au format JPEG, qui est automatiquement sélectionné au vu de l'extension du nom de fichier. Cela suppose que le tableau `im` contient bien les trois composantes de couleurs. Pour enregistrer une image directement à partir d'un simple tableau de niveaux de gris (tel `im0` ci-dessus), deux solutions :
  - on peut copier trois fois les valeurs de niveau de gris en tant que composantes couleurs !
  - on peut aussi indiquer à `matplotlib` que l'on travaille avec la palette ("colormap") des niveaux de gris : `img.imsave('Image_gray.jpg',im0,cmap=plt.cm.gray)` ; cette option sera aussi utile pour l'affichage... La palette inversée (pour afficher le "négatif") existe aussi : `plt.cm.gray_r`.

- `plt.imshow(im)` prépare l'affichage par `pyplot` de l'image matricielle dont les composantes couleurs sont stockées dans le tableau `im`. Par défaut `imshow` attend un tableau tridimensionnel avec trois composantes couleurs par pixel. En cas de simple tableau de niveaux de gris, ajouter l'option "colormap" : `plt.imshow(im,cmap=plt.cm.gray)`.

  L'affichage effectif (qui bloque l'exécution du script...) est obtenu par `plt.show()`.

- Il peut être intéressant d'afficher plusieurs images dans une même fenêtre, par exemple pour comparer l'image originale à l'image traitée. Pour cela, utiliser la commande `plt.subplot(nl,nc,n)`, qui prévoit dans la fenêtre à afficher `nl` lignes de `nc` figures, lesdites figures étant numérotées de gauche à droite et de haut en bas par l'entier `n`, qui peut donc prendre les valeurs de 1 à `nl*nc`. En pratique, pour préparer une fenêtre comportant plusieurs figures, on utilise `subplot` avec les mêmes valeurs de `nl` et `nc`, en faisant varier `n` : les instructions de tracé suivant l'instruction `plt.subplot(nl,nc,n)` affecteront la figure numéro `n`, jusqu'à l'instruction `subplot` suivante (ou jusqu'au tracé par `plt.show()`). Parmi les instructions de tracé, on peut utiliser `plt.xlabel('Titre')` pour écrire un titre sous la figure.

Remarques importantes

Parfois, la lecture d'un fichier image remplit les composantes couleurs avec des entiers de $[\![0, 255]\!]$; mais certaines commandes de `matplotlib` (enregistrement et affichage) requièrent des flottants de $[0, 1]$.
On n'hésitera pas, si besoin, à utiliser les fonctions de conversion $n \mapsto n/255$ ou $x \mapsto int\,(255 * x)$.

Dans `matplotlib`, il semble que la palette des niveaux de gris accepte n'importe quelles valeurs et s'étale du min au max. Par contre, l'affichage en couleurs requiert des flottants de $[0, 1]$.
Dans certains cas (par exemple la détection des contours) on est amené à remplir un tableau `t` avec des nombres dont on ne connaît pas à l'avance les valeurs extrêmes ; on pensera alors à utiliser `M=np.max(t)` et `m=np.min(t)` pour tout ramener dans $[0, 1]$ à l'aide de la fonction $f : x \mapsto (x - m) / (M - m)$.

De façon générale, pour appliquer une fonction à tous les éléments d'un tableau `numpy`, on peut bien sûr programmer les boucles nécessaires ; on peut aussi utiliser la *vectorisation* d'une fonction : `np.vectorize(f)(t)` renvoie le tableau `numpy` de mêmes dimensions que `t` obtenu en appliquant `f` à tous les éléments de `t`.

Pour les exemples qui suivent, la *chaîne de traitement* est la suivante :
1) lecture des données dans le fichier de l'image originale
2) modification des données (les programmes à écrire)
3) enregistrement ou affichage de l'image associée aux nouvelles données.

III - Traitements pixel par pixel

On suppose ici que les niveaux de gris ou les composantes couleurs sont des flottants de $[0,1]$, ce qui est le cas lorsque `matplotlib` lit une image PNG.

1) Effet miroir

Écrire une fonction `miroirH(t)` qui construit l'image miroir symétrique de l'image représentée par `t` par rapport à l'axe vertical qui la coupe en deux. On pensera à créer un nouveau tableau pour conserver l'original. On rappelle que `np.shape(t)` renvoie le type du tableau `t`, en l'occurrence le couple (n,p), si l'on n'a gardé qu'une "tranche" de niveaux de gris, ou le triplet $(n,p,3)$ si l'on a conservé les trois composantes de couleurs.

2) Négatif

Écrire une fonction `neg(t)` qui construit le *négatif* de l'image représentée par `t`.
On peut combiner à `np.vectorize` l'instruction `lambda x: 1-x`, qui renvoie de façon *anonyme* la fonction $x \mapsto 1-x$.
Dans ce cas, on peut aussi utiliser les opérations sur les tableaux `numpy` : `1-t` renvoie le tableau résultant de l'application de la fonction $x \mapsto 1-x$ à tous les éléments de `t`.

3) Ajustement de la luminosité ou du contraste

Écrire une fonction `corr_lum(t,delta)` qui ajoute `delta` à tous les éléments de `t`. On prendra garde à maintenir les valeurs dans $[0,1]$... Tester l'effet produit.
Pour corriger le contraste de l'image, on modifie la vitesse de variation d'intensité en appliquant une fonction "bien choisie" aux éléments de `t`. La *correction* γ consiste à appliquer la fonction $x \mapsto x^\gamma$, γ étant une constante réelle. Tester diverses valeurs de γ.
Tester aussi la *correction linéaire* $x \mapsto \begin{cases} y = x + k.\,(x-0.5) \text{ si } y \in [0,1] \\ 1 \text{ si } y > 1 \\ 0 \text{ si } y < 0 \end{cases}$.

4) Réduction du nombre de couleurs

Pour diminuer la taille de stockage de l'image, on peut diminuer le nombre de couleurs (ou de niveaux de gris) utilisables, le plus brutal étant le "noir et blanc", avec deux valeurs admises, 0 ou 1 (codage sur 1 bit).
Soit $b \in [\![2,7]\!]$. Montrer que la fonction $x \mapsto \left\lfloor 2^b \dfrac{\lfloor 255x \rfloor}{256} \right\rfloor / \left(2^b-1\right)$ de $[0,1]$ dans lui-même prend exactement 2^b valeurs équiréparties (codage sur b bits). Tester l'effet produit par la transformation du tableau `t` associée à cette fonction. Voir aussi selon la valeur de b la taille du fichier obtenu en enregistrant l'image...

5) Réduction du nombre de pixels

Autre piste pour réduire la taille du fichier : réduire le nombre de pixels !
Pour $k > 1$ fixé, mettons que l'on souhaite diviser par k^2 le nombre de pixels. Il suffit de conserver un pixel sur k dans chaque ligne et chaque colonne.
Programmer et tester cette réduction, qui va bien sûr accentuer la pixélisation...
On peut améliorer le rendu en remplaçant chaque bloc de $k \times k$ pixels, non plus par la couleur d'un seul des pixels du bloc, mais par la moyenne des couleurs desdits pixels. La commande `np.mean(t[i1:i2,j1:j2])` renvoie la moyenne de l'extrait de tableau indiqué. Programmer cette méthode et tester.

6) Agrandissement

Difficile d'inventer de l'information pour ajouter des pixels ! Mettons que l'on souhaite multiplier par $k > 1$ fixé le nombre de pixels par ligne et par colonne (afin de conserver les proportions). La solution triviale consiste à remplacer chaque pixel par un bloc de $k \times k$ pixels identiques. Programmer cette méthode et tester.
Le résultat est *a priori* sans intérêt (rendu identique pour une même taille d'image !), mais un *lissage* (cf. § **IV - 2)**) peut permettre d'améliorer la qualité de l'image agrandie.

IV - Filtrage par convolution

1) Convolution discrète

De nombreuses transformations d'image se font, non pas pixel par pixel, mais en prenant en compte les pixels voisins du pixel à modifier. La *convolution discrète* (ainsi nommée en référence au *produit de convolution* des fonctions, définie par une intégrale) est une opération classique dans ce contexte : soient un tableau t de type (n,p), en général assez grand (ici contenant les niveaux de gris de l'image) et un tableau m (on parle parfois de *masque de convolution*), en général assez petit, carré d'ordre impair $2q+1$; le résultat de la convolution est un tableau de même taille que t, où l'élément $t[i,j]$ a été remplacé par la somme

$$\sum_{di=-q}^{q}\sum_{dj=-q}^{q} m[q+di,q+dj]\cdot t[i+di,j+dj].$$

Si la somme des éléments de m vaut 1, il s'agit d'une moyenne pondérée des éléments de t avoisinant $t[i,j]$.

Bien entendu, il faut gérer les "bords" de t, là où le masque "déborde". En général on souhaite conserver la taille de t pour le résultat, il faut donc adapter le calcul, soit en définissant des masques partiels, soit en bordant t en lui ajoutant des valeurs tout autour. Dans les applications pratiques, cela n'a pas une grande importance car m est beaucoup plus petit que t et donc les anomalies sur les bords sont peu visibles.

Les acharnés peuvent programmer eux-mêmes la convolution, mais elle est prévue dans le module `signal` de `scipy` :

```
from scipy.signal import convolve2d
...
tc=convolve2d(t,m,mode='same')
...
```

L'option `mode='same'` permet d'obtenir un tableau de même taille que t. Voir l'aide de `scipy` pour les autres options, notamment celles relatives à la gestion des bords ; par défaut, le tableau est complété par des 0 là où c'est nécessaire.

Comme la convolution nécessite un grand nombre d'opérations, cette version précompilée est bénéfique !

2) Lissage – Floutage

Ces deux opérations sont en fait voisines, puisqu'il s'agit de réduire les "aspérités".
On utilise principalement les *filtres uniformes* (ou *moyens*) et les *filtres gaussiens*, tous associés à un masque de convolution dont les éléments sont positifs et de somme 1. La valeur de $t[i,j]$ est ainsi remplacée par une moyenne des valeurs voisines.
Le filtre uniforme d'ordre $2q+1$ correspond au masque de convolution dont tous les éléments sont égaux à $\dfrac{1}{(2q+1)^2}$.
Les filtres gaussiens sont associés à une "distribution normale" de la forme

$$G_\sigma : (x,y) \mapsto \frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) \quad (\text{où } \sigma>0 \text{ est fixé}).$$

Programmer une fonction `m_gauss(q,s)` qui renvoie le masque m d'ordre $2q+1$ associé à G_σ, en centrant en (q,q), c'est-à-dire que $m[i,j]$ sera calculé à partir de $G_\sigma(i-q,j-q)$, en "normalisant" pour que la somme des éléments de m vaille 1. Plus l'écart type σ est petit, plus la valeur centrale est prépondérante ! Lorsque σ tend vers $+\infty$, on s'approche d'un filtre uniforme...

Un grand filtre uniforme donne un *floutage* ; un filtre gaussien "raisonnable", par exemple avec $q=3$ et $\sigma=1,5$ permet un certain *lissage*.

Tester quelques-uns de ces filtrages, notamment sur l'image `Oeil.png`, obtenue par agrandissement.

3) Réduction du bruit

Des poussières sur un capteur ou des éléments de capteur endommagés peuvent occasionner un *bruitage* de l'image. Un filtrage adapté peut améliorer notablement la situation.

La première idée est d'appliquer un filtre uniforme, qui va "gommer" le bruitage ; hélas il va aussi induire un floutage...

Il est plus efficace d'appliquer un *filtre médian*, qui consiste à remplacer $t\,[i,j]$, non plus par une moyenne, mais par la médiane des valeurs avoisinantes, toujours situées dans un masque d'ordre $2q+1$. Mais ici il n'est plus question de convolution, puisque la détermination de la médiane ne peut pas se faire par une formule "algébrique"...

Programmer l'application dudit filtre médian. On pourra s'abstenir (ou pas) de modifier le "cadre" de $q-1$ pixels de large en bordure de l'image. Pour les couples (i,j) à traiter, on pourra utiliser l'instruction `np.median(t[i-q:i+q,j-q:j+q])`.

Tester les deux méthodes sur l'image `Image+bruit.png`, dont 20% des pixels ont été altérés aléatoirement.

Question subsidiaire : programmer le bruitage aléatoire d'une image donnée. Dans Python `random.random()` renvoie un flottant pseudo-aléatoire de $[0,1[$...

4) Détection des contours

a) Généralités

Ce sujet donne lieu à des recherches intensives, vu la variété de ses applications, notamment à la détection de formes, au suivi du déplacement d'objets ou de visages... Ce qui suit n'est qu'une initiation modeste !

L'idée naturelle est de détecter les points au voisinage desquels le niveau de gris varie brutalement. Pour cela on calcule une sorte de "gradient discret" en chaque point. La version la plus simple consiste à calculer la *dérivée discrète* selon i, $D_1t\,[i,j] = t\,[i+1,j] - t\,[i,j]$ et de même la dérivée discrète selon j, $D_2t\,[i,j] = t\,[i,j+1] - t\,[i,j]$. On en déduit par exemple la norme N_1 de ce "gradient", $|D_1t\,[i,j]| + |D_2t\,[i,j]|$ (plus facile à calculer que la norme euclidienne ; rappelons que toutes les normes sur $\mathbb{R}^2$ sont équivalentes !).

On remarque que le tableau D_1t s'obtient par convolution selon le masque $d1 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$; de même pour D_2t avec le masque $d2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 0 \end{pmatrix} = {}^t d1$.

On trouve dans la littérature divers autres filtres permettant de détecter les pixels à "fort gradient". Voici les masques les plus couramment utilisés. Donnons le masque pour la "dérivée" selon i (de haut en bas), sachant que celui selon j n'est autre que sa transposée (cf. `np.transpose(m)`).

$$\textit{Prewitt :}\ \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}\ ;\quad \textit{Sobel :}\ \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}\ ;\quad \textit{Kirsch :}\ \begin{pmatrix} -3 & -3 & -3 \\ -3 & 0 & -3 \\ 5 & 5 & 5 \end{pmatrix}.$$

À l'aide de `convolve2d`, programmer une fonction `contours(t,m)` recevant le tableau t des niveaux de gris et le masque m de dérivation selon i et renvoyant le tableau des "normes des gradients", lui-même normalisé par division par le maximum des normes obtenues.

Afficher les contours ainsi détectés pour quelques images tests. Le passage au négatif donne souvent une image plus "lisible".

b) Pré-traitement

Une réduction préalable du bruit peut améliorer la détection des contours. On utilise souvent un filtre gaussien.

c) Post-traitement

Une augmentation du contraste de la "carte des contours" permet en général de l'améliorer.
On peut même passer à une image en noir et blanc en définissant un *seuil* d'acceptation d'un point comme appartenant à un contour.

Hélas la détection de contours n'est pas une science exacte, il y a de nombreuses décisions empiriques à prendre, avec des résultats plus ou moins convaincants selon l'origine de l'image initiale. Mais justement, pour certaines sources bien "calibrées" (vidéosurveillance, imagerie médicale, etc.), on peut obtenir des algorithmes très efficaces.

Image originale

Filtre de Sobel brut

Filtre de Sobel traité

Avec seuil de 0,9