Réductions colorimétriques

1 Introduction

1.1 Objectifs poursuivis

Pour de nombreuses raisons, il peut être intéressant de limiter le nombre de couleurs d'une image : parce que l'on ne peut disposer que d'un nombre limité de couleurs (par exemple dans la réalisation d'une mosaïque) ou, dans le cas d'images numériques, pour l'espace nécessaire à son stockage (un pixel occupe généralement trois octets, mais avec une palette de 16 couleurs, cela peut par exemple descendre à un demi-octet).

Dans cette séance, nous allons voir comment réduire le nombre de couleurs d'une image, de quelle façon choisir au mieux les couleurs en question, et enfin exploiter des techniques de diffusion d'erreur (dithering) pour que le rendu final soit le plus proche possible de l'image originale.

1.2 Récupération des données

Pour télécharger et installer les données que l'on va utiliser durant cette séance de TP, lancer l'application « cygwin terminal » dont le raccourci se trouve sur le bureau, et entrer soigneusement la commande suivante [1] (que l'on validera avec la touche entrée) :

```
cd /cygdrive/d; curl cdn.sci-phy.org/psi/tp5-dithering.tgz | tar xvz
```

La commande devrait avoir créé un dossier `TP5-Dithering` dans la racine du disque `D:` (ou `F:`) qui contiendra des images. Vous pouvez ensuite refermer le terminal et lancer l'environnement Pyzo.

En cas de difficultés, ou si vous voulez travailler sur votre propre machine, vous pouvez télécharger le fichier `http://cdn.sci-phy.org/psi/tp5-dithering.tgz` par la méthode de votre choix, de décompresser le fichier ainsi obtenu (par exemple avec l'utilitaire 7zip) et de placer le répertoire à un endroit aisément accessible.

Précisons enfin que chaque image est fournie dans une taille normale et une taille réduite (le nom se termine par `_s`) car bon nombre de fonctions de ce TP prennent du temps, aussi est-il parfois intéressant d'effectuer les tests intermédiaires sur une image de petite taille.

Vous pouvez également récupérer une ébauche de fichier Python pour le TP, à l'adresse usuelle (`llg.sci-phy.org` , rubrique PSI et TP).

1. Si le disque de « données » sur la machine est `F:`, on écrira « `/cygdrive/f` ».

1.3 Rappels sur les images

Dans ce TP, on s'intéresse à des images en couleur, encodée au format RGB. Dans ce format, on décrira une *couleur* par un élément de $[0,1]^3$, que l'on représentera en Python par exemple par une liste contenant trois réels. Le premier élément correspond à la quantité de rouge (0 : absence, 1 : maximum), le second de vert, le troisième de bleu.

Ainsi, `[1.0, 0.0, 0.0]` correspond par exemple à la couleur rouge, `[0.8, 0.8, 0.0]` à un mélange de rouge et de vert (donnant une couleur jaune), `[0.0, 0.3, 0.9]` à un bleu tirant sur le vert, `[1.0, 1.0, 1.0]` à du blanc, `[0.0, 0.0, 0.0]` à du noir, et `[0.5, 0.5, 0.5]` à un gris neutre.

Une image RGB peut être vue comme une matrice de pixels de hauteur h et de largeur l, où chaque pixel correspond à une couleur, autrement dit une liste de trois flottants. Ainsi, si `img` désigne une image, `img[i][j]` désigne le pixel sur la ligne `i`, colonne `j` (une liste de trois réels), et `img[i][j][0]` désigne par exemple la quantité de rouge émis par le pixel.

1.4 Manipulation en TP

Pour travailler sur des images et afficher le résultat, on propose d'utiliser le canevas ci-dessous. Les deux premières lignes importent les bibliothèques utiles. On place ensuite les fonctions demandées, puis les lignes `img = ...` chargent l'image sous la forme d'une liste de listes de listes à trois réels. Une fois l'image chargée, on appelle la ou les fonctions que l'on souhaite tester (et qui modifient `img`), et les deux dernières lignes affichent le résultat.

```python
import imageio as iio
import matplotlib.pyplot as plt
import random as rd

// Définitions diverses (palettes)

// Fonctions à créer

img = [[list(pix) for pix in line] for line in
          iio.imread("C:/hemin/vers/image.jpg")/255.0]

// Effectuer des traitements sur l'image

plt.imshow(img)
plt.show()
```

2 Réduction des couleurs

2.1 Distances dans l'espace RGB

1. Proposer une fonction `dist2(col1, col2)` prenant en argument deux couleurs `col1` et `col2` (des listes de réels de longueur 3) et retournant le carré de la distance euclidienne entre les deux couleurs.

2.2 Palettes de couleurs

Une « palette » de couleurs est un ensemble de plusieurs couleurs, autrement dit plusieurs éléments de $[0,1]^3$. On représente une palette sous la forme d'une liste de ses couleurs. Par exemple la palette EGA par défaut est définie de la sorte :

```
palette_EGA = [[0.0, 0.0, 0.0], [0.0, 0.0, 0.67], [0.0, 0.67, 0.0],
               [0.0, 0.67, 0.67], [0.67, 0.0, 0.0], [0.67, 0.0, 0.67],
               [0.67, 0.33, 0.0], [0.67, 0.67, 0.67],
               [0.33, 0.33, 0.33], [0.33, 0.33, 1.0],
               [0.33, 1.0, 0.33], [0.33, 1.0, 1.0], [1.0, 0.33, 0.33],
               [1.0, 0.33, 1.0], [1.0, 1.0, 0.33], [1.0, 1.0, 1.0]]
```

On trouvera aussi fournie une palette de couleurs LEGO, un peu plus complète.

2. Proposer une fonction `plus_proche(col, palette)` prenant en argument une couleur `col` (une liste de trois réels) et une palette `pal` (une liste de liste de trois couleurs), et retournant l'*index* de la couleur, dans la palette, la plus proche de `col` (un entier entre `0` et `len(pal)-1` donc).

3. En déduire une fonction `applique_palette(img, pal)` qui prend en argument une image et un palette de couleurs, et remplace chaque couleur par la couleur la plus proche dans la palette de couleurs. On traitera l'image ligne par ligne, et les pixels de gauche à droire sur chaque ligne.

4. Tester la fonction sur l'une des images à disposition.

2.3 Palettes aléatoires

La fonction `rd.random()` (`rd` étant un alias pour le module `random`) retourne un flottant choisi aléatoirement dans $[0,1]$.

5. Proposer une fonction `palette_aleatoire(N)` prenant en argument un entier `N`>0 et retournant une palette contenant N couleurs choisies aléatoirement.

6. Afficher l'image réduite à quelques palettes choisies aléatoirement, en faisant varier N entre 4 et 256. À partir de combien de couleurs obtient-on des résultats généralement décents ?

3 Sélection d'une palette

3.1 Motivation et démarche

Comme on l'a vu, choisir une palette prédéfinie ne donne pas des résultats remarquables si le but est de préserver la qualité de l'image. L'idée est donc de choisir une palette spécifiquement adapté à l'image choisie. Pour ce faire, nous allons utiliser l'algorithme des $k-$moyennes. La méthode que l'on va mettre en œuvre est la suivante :

- on part d'une palette de N couleurs choisie aléatoirement
- puis, p fois :
  - on détermine la couleur de la palette la plus proche de chaque pixel de l'image
  - puis pour chaque couleur de la palette, on détermine la moyenne des couleurs des pixels qui lui ont été associé
  - on remplace dans la palette la couleur par la moyenne précédemment calculée

3.2 Implémentation

7. Proposer une fonction `classe(img, pal)` prenant en argument une image et une palette, et retournant une matrice `idx` d'entiers entre `0` et `len(pal)-1`, de mêmes dimensions que l'image, et telle que `idx[i][j]` corresponde à l'index, dans `pal`, de la couleur de la palette la plus proche du pixel `img[i][j]`.

8. Proposer une fonction `moyennes(img, idx, N)` retournant une palette de `N` couleurs, où la couleur `k` est :

- le barycendre de toutes les couleurs des pixels `img[i][j]` dans l'image telles que `idx[i][j] == k` s'il y a au moins un tel pixel dans l'image ;
- une couleur choisie aléatoirement sinon.

Pour des raisons d'efficacité, surtout s'il y a de nombreuses couleurs, il est recommandé d'effectuer le calcul des N moyennes en une seule passe sur l'image. Il pourra être utile de créer pour cette question une fonction `ajoute(col1, col2)` qui prend en argument deux listes à trois éléments et ajoutant à chacun des termes de la seconde liste les termes de la première.

9. En déduire une fonction `opt_palette(img, N, p)` qui, avec l'algorithme des $k-$moyennes effectuant p itérations, détermine une palette de N couleurs adaptée à l'image.

Il serait possible d'utiliser un autre critère d'arrêt qu'un nombre fixé d'itérations pour arrêter l'algorithme, par exemple lorsque les couleurs de la palette, entre deux itérations, n'ont pratiquement pas changé (distance entre les anciennes et les nouvelles couleurs inférieures à un ε choisi préalablement). On ne demande pas d'implémenter cette condition ici, mais c'est un bon entraînement si vous trouvez le temps de le faire.

10. Affcher l'image obtenue une fois convertie à la palette en question en faisant varier p (entre 1 et 20 itérations) et N (entre 4 et 32 couleurs). Que penser des résultats ?

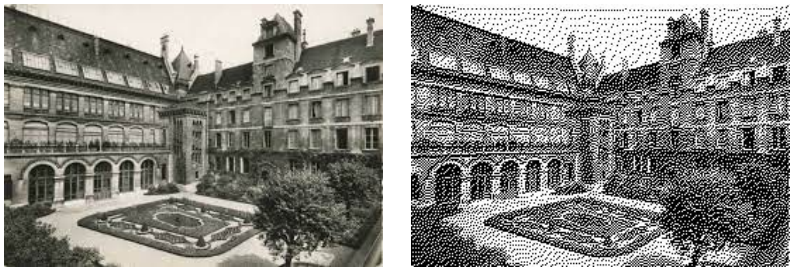
4 Diffusion des erreurs

4.1 Motivation

Prendre systématiquement la couleur la plus proche n'est pas nécessairement la meilleure solution, ce que l'on constate notamment sur les dégradés dans l'image.

Si une partie de l'image est orange, et que l'on ne dispose dans la palette que de jaune et de rouge mais pas d'orange, une solution peut être de mettre côte à côte des pixels jaunes et rouge, et leur mélange pourra, si les pixels sont suffisamment serrés, donner une impression de couleur orange. C'est une technique permettant de donner l'impression d'avoir davantage de couleurs utilisée notamment dans les premiers temps de l'informatique, appelée *diffusion de l'erreur* (*dithering* en anglais).

Dans l'exemple ci-dessous, la photographie de gauche a par exemple été convertie avec cette technique en une image qui ne contient que des pixels blancs et des pixels noirs :



4.2 Mise en œuvre

Pour y parvenir, le principe est simple : si l'on note x la couleur originale du pixel `img[i][j]` et y la couleur de la palette que l'on utilisera pour ce même pixel, on commet une erreur $e = y - x$ sur la couleur de ce pixel (x et y étant des vecteurs de $[0,1]^3$, il est possible de définir une différence, vectorielle, dont le résultat sera dans $\mathbb{R}^3$).

Pour compenser cette erreur, on va modifier la couleur des pixels voisins avant de les associer à une couleur de la palette. Comme on va traiter les pixels ligne par ligne, et de gauche à droite, il est trop tard pour les pixels immédiatement à gauche et ceux sur la ligne du dessus. Pour les autres, on va ajouter :

- $7e/16$ au pixel à droite du pixel `img[i][j]` (s'il existe) ;
- $3e/16$ au pixel en-dessous et à gauche du pixel `img[i][j]` (s'il existe) ;
- $5e/16$ au pixel en-dessous du pixel `img[i][j]` (s'il existe) ;
- $e/16$ au pixel en-dessous et à droite du pixel `img[i][j]` (s'il existe).

Les coefficients n'ont pas été choisis au hasard, mais ajustés pour donner des résultats visuellement satisfaisants (il existe d'autres formules plus ou moins complexes, il s'agit ici de la méthode dite de Floyd-Steinberg). On remarquera au passage que la somme donne bien e, de sorte que l'on tente de corriger complètement l'erreur commise sur le pixel `img[i][j]`.

11. Proposer une fonction `applique_palette_diff(img, pal)` qui prend en argument une image et un palette de couleurs, et remplace chaque couleur par la couleur la plus proche dans la palette de couleurs, en appliquant la méthode de diffusion de l'erreur de Floyd-Steinberg.

12. Afficher le résultat d'une image convertie à la palette EGA grâce à cette méthode.

13. Faire de même avec une palette obtenue par optimisation avec la méthode des $k-$moyennes (on pourra par exemple prendre N = 16 et $p = 15$ itérations. Que penser du résultat obtenu ?

5 Espace CIELAB

Revenons un instant sur l'optimisation de la palette. Il y a un souci avec le choix de l'espace colormétrique RGB : il n'est pas perceptuellement uniforme, c'est-à-dire que la distance euclidienne, dans cet espace, ne correspond pas bien à une perception des différences entre deux couleurs.

Pour améliorer les choses, on peut changer d'espace colorimétrique, au profit par exemple[2] de l'espace CIELAB, où chaque couleur est définie par trois composantes, L (clarté, dérivé de la luminance), a* (écart à la neutralité sur un axe turquoise/magenta) et b* (écart à la neutralité sur un axe bleu/jaune). On fournit deux fonctions `RGB_to_CIELAB` et `CIELAB_to_RGB` permettant de convertir[3] une couleur RGB en une couleur La*b* et inversement.

Pour améliorer le calcul de la palette optimale, on peut donc travailler dans l'espace CIELAB lorsque l'on calcule les distances entre couleurs. L'ennui, c'est que les conversions prennent du temps, aussi procédera-t-on de la façon suivante :

- on construit une image où l'on a converti chacun des pixels de l'image originale dans l'espace colorimétrique CIELAB
- on détermine une palette optimale (au format CIELAB) pour cette image (aucune modification des fonctions écrites n'est nécessaire)
- on convertit les couleurs de cette palette obtenue au format RGB
- on convertit les couleurs des pixels de l'image originale en couleurs appartenant à la palette avec diffusion de l'erreur.

La répartition des couleurs dans l'espace CIELAB n'étant pas homogène, on préférera

```
RGB_to_CIELAB([rd.random() for k in range(3)])
```

2. L'espace CIELAB est encore imparfait, il existe actuellement des solutions un peu plus précises.
3. C'est une conversion approchée, la question des couleurs est un problème infiniment complexe, mais cela suffira pour ce TP.

de préférence à `[rd.random() for k in range(3)]` pour générer une couleur aléatoire (dans `palette_aleatoire` ainsi que dans `moyennes`).

14. Écrire une fonction `optimise(img, N, p)` qui prend en argument une image et retourne une image ne contenant que `N` couleurs différentes, choisies avec la méthode précédente, en ayant effectué `p` itérations de l'algorithme des $k-$moyennes.