

---

# MANIPULATION D'IMAGES EN C

## *Épreuve orale d'informatique du CCMP*

---

### Bibliothèque fournie

Ce sujet va faire l'utilisation de la bibliothèque SDL qui gère divers aspects : ouverture de fenêtre et dessins sur la fenêtre, lecture d'images au format JPG, etc. Vous n'avez pas besoin de connaître le fonctionnement de cette bibliothèque, car le concours fournit un fichier `graphique.h` qui simplifie grandement l'utilisation de la SDL.

**Entête `graphique.h`.** Le fichier d'entête `graphique.h` définit plusieurs types, constantes et fonctions. Voici ce qu'il faut en connaître :

- les constantes entières `HAUTEUR_FENETRE` et `LARGEUR_FENETRE` définissent les hauteur et largeur de la fenêtre ouverte ;
- le type `couleur` correspond à la couleur d'un pixel<sup>1</sup> ;
- le type `image` correspond à une image dans un format interne qu'on peut créer ou manipuler avec certaines des fonctions présentées ci-dessous<sup>2</sup> ;
- `demarre_graphique()` est une fonction qui n'attend aucun argument et qui ouvre une fenêtre graphique ;
- `arrete_graphique()` termine le graphisme<sup>3</sup> ;
- `image charge_image(const char * chemin)` charge une image stockée au chemin fourni en paramètre ;
- `largeur_image(image img)` et `hauteur_image(image img)` renvoient la largeur et la hauteur de l'image passée en paramètre ;
- `couleur couleur_du_pixel(const image img, const int X, const int Y)` renvoie la couleur du pixel (X, Y) de l'image passée en paramètre<sup>4</sup> ;
- `libere_image(image img)` libère la mémoire de l'image passée en paramètre ;
- `affiche_image(image img)` affiche l'image passée en paramètre dans la fenêtre (qui doit avoir été ouverte avec `demarre_graphique()`) et attend que la touche Droite soit pressée avant de terminer, ou bien la touche Échappe, ce qui termine le programme ;
- `image couleurs_vers_image(couleur * c, int largeur, int hauteur)` renvoie une image construite à partir d'un tableau de couleurs et des dimensions largeur et hauteur de l'image. Le tableau qui décrit les couleurs doit être un tableau linéarisé, c'est-à-dire que la couleur du pixel (X, Y) de l'image renvoyée est celui de `c[Y*largeur + X]` ;
- enfin, `int nombre_aleatoire(int maxi)` renvoie un nombre aléatoire entre 0 (inclus) et maxi (exclus), pour  $0 < \text{maxi} \leq 2^{31}$ .

**Compilation.** Pour compiler vos programmes, il est recommandé d'utiliser le fichier `Makefile` fourni et donc d'utiliser la commande `make`. Si vous voulez compiler par vous-même, la ligne de compilation à utiliser est la suivante (en supposant que votre fichier est `image.c`) :

```
gcc image.c -g $(pkg-config --cflags --libs sdl2) -lSDL2_image -lm
```

1. Une couleur est définie par sa composante de rouge, de vert et de bleu, chacune étant un entier positif entre 0 et 255.
2. Ce type est, en interne, un pointeur, donc « copier » une valeur image ne duplique pas l'image.
3. `demarre_graphique` et `arrete_graphique` ne sont prévues que pour être appelées une fois et dans cet ordre.
4. X doit être dans l'intervalle `[0, largeur_image(img)-1]` et Y dans `[0, hauteur_image(img)-1]`.

**Images fournies.** Pour cette partie, on vous fournit des fichiers images au format JPG. Ils se situent dans le dossier `images`.

## Première manipulations avec les images

**Question 1.** Éditer le fichier `image.c` pour que le programme ouvre une fenêtre, affiche une image et attende qu'une touche soit pressée.

**Niveaux de gris.** Étant donné une couleur  $(r, g, b)$  on définit sa *composante grise* en prenant les trois composantes  $r$ ,  $g$  et  $b$  et en calculant la moyenne pondérée  $\frac{3r + 6g + b}{10}$ . Étant donné une image, son passage en niveaux de gris correspond à remplacer la couleur de chaque pixel par un triplet correspondant à sa composante grise.

**Question 2.** Écrire une fonction `image_gris(image img)` qui prend en paramètre une image et renvoie la même image en niveaux de gris.

**Question 3.** Écrire une fonction `image_zoom_naif(image img, double f)` qui prend un `double f` et une image `img` de dimension  $L \times H$  et renvoie l'image de taille  $\lfloor L \times f \rfloor \times \lfloor H \times f \rfloor$  où le pixel  $(x, y)$  a la couleur du pixel  $\left(\left\lfloor \frac{x}{f} \right\rfloor, \left\lfloor \frac{y}{f} \right\rfloor\right)$  dans l'image `img`. Bien que la fonction s'appelle « zoom », on autorisera l'argument  $f$  à être inférieur à 1, auquel cas, la fonction renverra une image plus petite que l'image originale.

*Pour tester, on utilisera un facteur important (par exemple 5), mais pour éviter d'avoir une image trop large, on pourra commencer par faire zoom avec un facteur de 0.2 avant de faire un zoom de facteur 5.*

**Valeur de couleur.** Étant donné une image de dimension  $L \times H$ , la valeur de couleur au point  $(x, y) \in \mathbb{R}^2$  est la moyenne des couleurs des 4 pixels autour  $(\lfloor x \rfloor, \lfloor y \rfloor)$ ,  $(\lfloor x + 1 \rfloor, \lfloor y \rfloor)$ ,  $(\lfloor x \rfloor, \lfloor y + 1 \rfloor)$ ,  $(\lfloor x + 1 \rfloor, \lfloor y + 1 \rfloor)$ , pondérée par les coefficients suivants :  $(1 + \lfloor x \rfloor - x) \times (1 + \lfloor y \rfloor - y)$ ,  $(x - \lfloor x \rfloor) \times (1 + \lfloor y \rfloor - y)$ ,  $(1 + \lfloor x \rfloor - x) \times (y - \lfloor y \rfloor)$ ,  $(x - \lfloor x \rfloor) \times (y - \lfloor y \rfloor)$ . Remarquez que cette définition peut faire intervenir des pixels qui sont hors de l'image ; on supposera que la couleur de tout pixel hors de l'image est  $(0, 0, 0)$ .

**Zoom bilinéaire** Étant donné une image de dimension  $L \times H$ , un zoom bilinéaire de facteur  $\alpha > 1$  est une image de dimension  $\lfloor \alpha L \rfloor \times \lfloor \alpha H \rfloor$  où le pixel  $(x, y)$  de la nouvelle image est la valeur de couleur au point  $(x/\alpha, y/\alpha)$ .

**Question 4.** Implémenter une fonction `image_zoom_bilineaire(image img, double f)` qui renvoie le zoom bilinéaire de l'image fournie par un facteur  $f$ . Comparer visuellement avec `zoom_naif` sur des zooms avec un facteur  $k$  important.

## Limiter le nombre de couleurs

Actuellement, nos images sont en « 16 millions de couleurs », c'est-à-dire que chaque pixel a trois composantes (rouge, verte, bleue) et chacune a une valeur entre 0 et 255. Un pixel a donc une couleur parmi  $256^3 = 16\,777\,216$  possibles.

**Question 5.** De combien de bits a-t-on besoin pour stocker une image dont les dimensions sont  $1024 \times 1024$  en « 16 millions de couleurs » ?

Une manière de réduire la place que prend l'image à stocker est de n'autoriser que  $k$  couleurs différentes (avec  $k$  petit) et de décrire l'image par la donnée des  $k$  couleurs (chacune de ces  $k$  couleurs est décrite par ses 3 composantes qui sont des nombres entre 0 et 255) suivie de la description des pixels.

**Question 6.** De combien de bits a-t-on besoin pour décrire 16 couleurs ? De combien de bits a-t-on besoin pour stocker une image dont les dimensions sont aussi  $1024 \times 1024$  en 16 couleurs ?

**Image avec un nombre de couleurs limité.** Étant donné un ensemble de couleurs  $S$ , une image est  $S$ -coloriée si elle n'utilise que les couleurs de  $S$ . Étant donné deux couleurs de pixels  $c_1 = (r_1, g_1, b_1)$  et  $c_2 = (r_2, g_2, b_2)$ , on définit la distance entre couleurs comme  $d(c_1, c_2) = (r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2$ .

Étant donné deux images  $I_1$  et  $I_2$  de mêmes dimensions, on définit leur distance comme la somme, pour chaque pixel  $p$ , de la distance entre les couleurs que  $I_1$  et  $I_2$  donnent à  $p$ . Formellement, si on note  $I[x, y]$  la couleur du pixel  $(x, y)$  dans  $I$ , pour  $I_1, I_2$  de dimensions  $L \times H$ , on a :

$$d(I_1, I_2) = \sum_{0 \leq x < L} \sum_{0 \leq y < H} d(I_1[x, y], I_2[x, y])$$

Étant donné un ensemble  $S$  et une image  $I$ , on peut définir l'erreur  $err(S, I)$  comme étant la distance minimale entre  $I$  et une image  $S$ -coloriée de même dimension. Intuitivement, cela représente la quantité de défauts visuels que l'on va obtenir en « compressant »  $I$  par l'utilisation de la palette de couleur  $S$  au lieu des  $256^3$  couleurs possibles.

**Question 7.** Proposer un type palette qui permet de représenter un ensemble de couleurs. Écrire ensuite une fonction `palette * palette_aleatoire(int nb_couleurs)` qui renvoie une palette de taille `nb_couleurs`, chaque couleur ayant des composantes rouge, verte et bleue aléatoires.

**Question 8.** Écrire une fonction `couleur choisit_couleur(palette p, couleur c)` qui renvoie la couleur de la palette `p` qui est la plus proche de la couleur `c` fournie. En cas d'égalité, on pourra renvoyer n'importe quelle couleur de distance minimale.

**Question 9.** Écrire une fonction qui prend une image  $I$  ainsi qu'un ensemble de couleurs  $S$  et renvoie l'image  $S$ -coloriée la plus proche de  $I$ . Tester votre fonction avec diverses palettes en incluant la palette qui ne contient que les couleurs blanche et noire.

## Diffusion d'erreur.

**Effet de seuil.** Comme on peut le remarquer avec la palette blanc-noir, le coloriage est très sensible aux effets de seuil. Si on considère, par exemple, un drapeau blanc/gris/noir, la fonction précédente fera apparaître un drapeau blanc et noir alors qu'il aurait été plus judicieux de colorier la zone grise avec un pixel sur deux en blanc et l'autre en noir pour donner une impression visuelle de gris.

**Algorithme de diffusion d'erreur de Floyd-Steinberg.** Une manière de réduire le problème de l'effet de seuil est d'utiliser l'algorithme de Floyd-Steinberg. Dans cet algorithme, on traite les pixels dans un ordre  $y$  croissant puis  $x$  croissant. Pour chaque pixel, on va choisir la couleur la plus proche (comme dans la question) mais, une fois que la couleur est choisie pour le pixel  $(x, y)$ , on va calculer l'erreur que l'on fait sur chacune des composantes rouge, vert et bleu et on va propager cette erreur aux pixels voisins qui ne sont pas encore traités, c'est-à-dire les pixels  $(x + 1, y)$ ,  $(x - 1, y + 1)$ ,  $(x, y + 1)$  et  $(x + 1, y + 1)$ . De cette manière, la couleur moyenne de chaque région sera à peu près bonne.

Par exemple, si on colorie un pixel dont la composante rouge est de 150 mais que la couleur choisie dans la palette est une couleur dont la composante rouge est 70, on fait une erreur de  $150 - 70 = 80$ . On va distribuer cette erreur 80 de rouge aux voisins en augmentant leur composante. Il faut aussi faire une distribution d'erreur similaire pour les composantes verte et bleue.

Dans l'algorithme original de Floyd-Steinberg, la distribution de l'erreur n'est pas équitable, puisqu'on envoie :

- 7/16 de l'erreur au pixel de droite  $(x + 1, y)$ ;
- 5/16 au pixel en dessous  $(x, y + 1)$ ;
- 3/16 au pixel en dessous à gauche  $(x - 1, y + 1)$ ;
- 1/16 au pixel en dessous à droite  $(x + 1, y + 1)$ .

**Question 10.** Implémenter l'algorithme de diffusion d'erreur de Floyd-Steinberg. Tester votre solution sur diverses palettes, dont la palette noir-blanc.

### Sélection d'une palette de couleur optimale

**Question 11.** Si l'on se fixe un nombre de couleurs, par exemple  $k = 16$ , proposer une heuristique qui calcule un ensemble  $S$  avec  $|S| = k$  et  $err(S, I)$  aussi petit que possible.

**Question 12.** Implémenter une fonction `palette(I, k)` qui prend une image  $I$  et un nombre de couleurs  $k$ , et qui renvoie un ensemble  $S$  avec  $|S| = k$  essayant de minimiser  $err(S, I)$ .

**Question 13.** Écrire un algorithme qui prend une image  $I$  et affiche l'image (`palette I k`)-coloriée associée. Afficher le résultat sur les images fournies avec  $k \in \{4, 16, 64\}$ . Que pensez-vous de la qualité visuelle des images obtenues ?

### Segmentation d'image

Un problème classique en traitement d'image est celui de la segmentation de l'image, c'est-à-dire le regroupement des pixels de façon à ce qu'il ne reste qu'un petit nombre de groupes dans l'image.

Par exemple, voici une segmentation de l'image 7 en 10 groupes, où chaque groupe est représenté par une couleur aléatoire :

On propose maintenant un algorithme pour segmenter une image  $I$  en niveaux de gris en  $K$  groupes, avec l'objectif de minimiser l'erreur des groupes. L'erreur totale des groupes est définie comme la somme des erreurs de chaque groupe, où l'erreur d'un groupe  $G$  est définie comme :

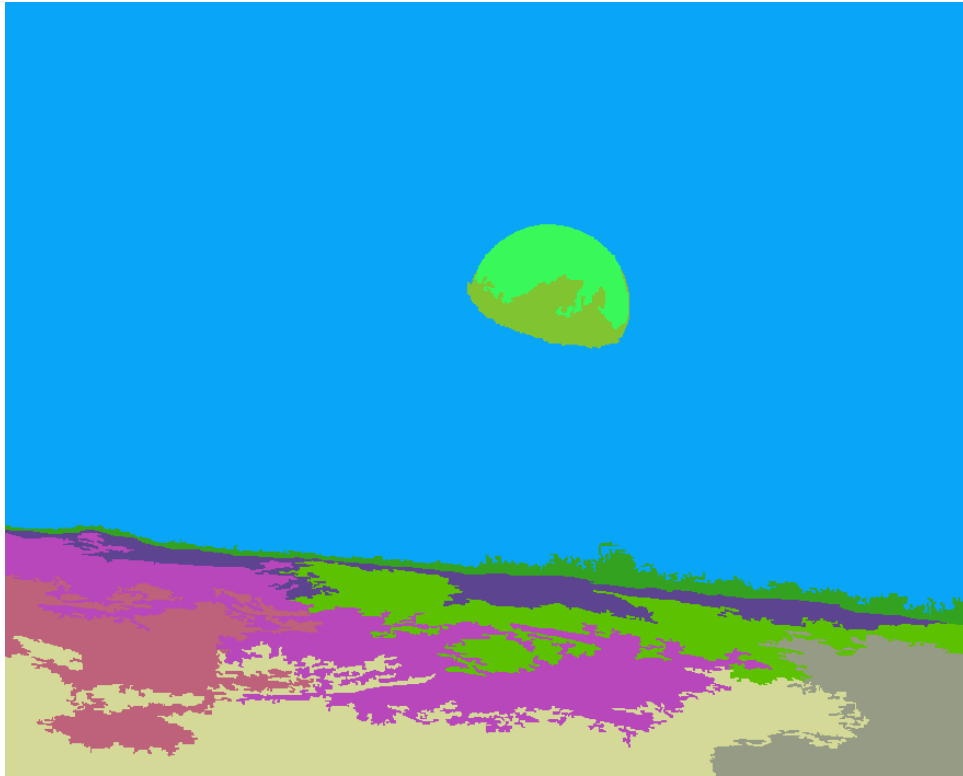
$$err(G) = \sum_{x,y \in G} \sum_{x',y' \in G} d(I[x,y], I[x',y'])$$

L'algorithme que l'on veut implémenter est le suivant. Initialement, chaque pixel est dans son propre groupe. Tant qu'il reste plus de  $K$  groupes, on trouve deux groupes  $G_1, G_2$  voisins (c'est-à-dire des groupes tels qu'il existe  $(x, y) \in G_1$  avec  $(x + 1, y) \in G_2$  ou  $(x, y + 1) \in G_2$ ) tels que l'erreur de la fusion  $G_1 \cup G_2$  est minimale parmi toutes les paires de groupes voisins et on les fusionne.

**Question 14.** Proposer puis implémenter une structure capable de représenter les groupes. Il faudra que cette structure soit capable de fusionner rapidement deux groupes et de trouver le groupe d'un pixel rapidement.

**Question 15.** Comment modifier cet algorithme pour être capable de connaître rapidement l'erreur commise en fusionnant deux groupes ?

*Indication : il est recommandé de stocker (et maintenir après fusion) la somme des couleurs au carré.*



**Tas paresseux.** Pour implémenter l'algorithme efficacement, on voudrait stocker la liste des fusions possibles dans un tas, avec comme priorité l'erreur de la fusion obtenue. Cependant, cela pose un problème : pour fusionner  $G_1$  et  $G_2$ , il faut d'abord enlever du tas toutes les fusions impliquant  $G_1$  et  $G_2$ , pour ensuite remettre toutes les fusions possibles avec la réunion de  $G_1$  et  $G_2$ . C'est très inefficace.

Pour éviter ce problème, on peut utiliser les deux propriétés suivantes :

- les groupes voisins de la fusion de  $G_1$  et  $G_2$  sont des groupes qui sont voisins de  $G_1$  ou  $G_2$  (et donc les fusions possibles avec  $G_1 \cup G_2$  correspondent aux fusions avec  $G_1$  ou  $G_2$ ) ;
- l'erreur de fusionner  $G_1 \cup G_2$  avec un groupe  $G_3$  est supérieure à l'erreur de fusionner  $G_3$  avec  $G_1$  ou  $G_2$ .

Ces propriétés permettent d'utiliser une structure de tas de façon *paresseuse*. Dans ce tas paresseux, on stocke des paires  $(a, b)$  de pixels avec comme priorité  $p$  une valeur inférieure ou égale à l'erreur de fusion des groupes contenant les pixels  $a$  et  $b$ .

Quand on pioche une paire  $(a, b)$  de pixels avec priorité  $p$ , on vérifie d'abord que  $a$  et  $b$  ne sont pas dans le même groupe. On vérifie ensuite que  $p$  est bien la priorité de fusionner les groupes contenant les pixels  $a$  et  $b$ , puis on fusionne ces deux groupes. Si  $p$  n'était pas la bonne priorité, on réinsère la fusion avec la bonne priorité  $p$ .

**Question 16.** Implémenter une structure de tas qui stocke des quintuplets  $(p, x_1, y_1, x_2, y_2)$  d'entiers (de type `int`). Les couples  $(x_1, y_1)$  et  $(x_2, y_2)$  représentent des pixels et  $p$  la priorité. Le tas devra être un tas min (les priorités les plus basses devront être en haut du tas).

**Question 17.** Implémenter l'algorithme pour segmenter des images.

Dans l'algorithme ci-haut, le tas peut contenir de nombreuses fois la même fusion, car deux groupes peuvent avoir beaucoup de pixels voisins.

**Question 18.** Proposer une méthode optimisée qui gère ces duplicata.