Cours de programmation séquentielle

Traitement d'images

1 Traitement d'images

1.1 Buts

- Manipulation de tableaux à deux dimensions en `C`.
- Utilisation de `git`.
- Utilisation de pointeurs.
- Lecture et écriture de fichiers binaires.
- Utilisation de types énumérés.
- Affichage graphique avec la librairie `SDL2`.

1.2 Énoncé

Créer une librairie de manipulation d'images en *niveaux de gris*. Cette librairie devra permettre d'effectuer les opérations suivantes:

1. Lecture d'une image au format PGM depuis un format binaire.
2. Sauvegarde d'une image dans le format PGM binaire.
3. Effectuer des opérations sur les images:
    1. Symétrie *horizontale*, *verticale*, et *centrale* d'une image.
    2. Un effet *photomaton*.
    3. *Filtre* d'une image.
    4. *Rogner* une image.
    5. Calculer le *négatif* d'une image.
4. Visualiser une image PGM obtenue à l'aide de la librairie SDL.

1.3 Cahier des charges

Pour ce travail, en plus de la réalisation de la librairie de traitement d'images, vous devez utiliser le logiciel de gestion de version `git` et évidemment compiler le projet à l'aide d'un `Makefile`.

1.3.1 Gestion d'erreurs

Un grand nombre des fonctions de ce travail peuvent échouer. Pour simplifier, nous nous limiterons à deux retour possibles du type énuméré `pgm_result`:

```
typedef enum _pgm_result {
    success, failure
} pgm_result;
```

1.3.2 Représentation informatique des images

Dans ce travail pratique, une image n'est rien d'autre qu'une matrice de nombres entiers, au sens de celles que vous avez implémentées au précédent travail pratique. Pour simplifier les images seront uniquement représentées en *niveaux de gris*. Ainsi chaque élément de la matrice représente un pixel. Les valeurs des pixels devront être limitées entre `0` et une valeur maximale `max`.

Pour simplifier encore, les images seront supposées être données au format PGM. Le format PGM (portable graymap file format) est un format de fichier très simple permettant de stocker des images en niveau de gris. Nous utiliserons le format PGM **binaire**. Vous trouverez une définition du format sur la page https://fr.wikipedia.org/wiki/Portable_pixmap.

Une image PGM peut être représentée à l'aide de la structure de données suivante:

```
typedef struct _pgm {
    int32_t max;
    matrix pixels;
} pgm;
```

Le format PGM binaire implique la lecture dans un premier temps d'une entête contenant le texte `P5` sur la première ligne, puis les dimensions de l'image sur la deuxième ligne. Sur la troisième ligne se trouve le niveau de gris maximal. Ces trois lignes sont en format `ASCII`. Finalement, les pixels de l'image en format binaire sont stockés dans les lignes restantes. A nouveau pour simplifier, nous supposerons que chaque pixel est un entier non signé d'un octet.

Nous supposerons dans ce travail, pour simplifier, qu'il n'existe pas de commentaires dans les fichier (pas de lignes commençant par `#`).

1.3.3 Lecture et écriture d'une image PGM en format binaire

Afin de lire et écrire un fichier binaire, il faut utiliser le type pointeur de fichier:

```
FILE *f;
```

et les fonctions permettant de manipuler les fichiers:

- `fopen()` avec les option `r` et `w` pour ouvrir un fichier en mode lecture ou écriture.
- `fclose()` pour fermer un fichier.
- `fprintf()` pour écrire le nombre de lignes et de colonnes de l'image, ainsi que `max` (voir la définition du format PGM) qui sont au format ASCII.
- `fgets()` pour lire une ligne d'un fichier au format ASCII (utile pour récupérer le nombre de lignes, colonnes, ainsi que `max`).
- `fwrite()` et `fread()` pour lire et écrire des données contiguës en mémoire dans un fichier.

A l'aide de ces fonctions vous devriez être capables de lire et écrire des fichiers PGM.

Afin de tester vos fonctions vous pouvez utiliser l'image du mandrill (voir fig. 1) en cliquant sur ce lien

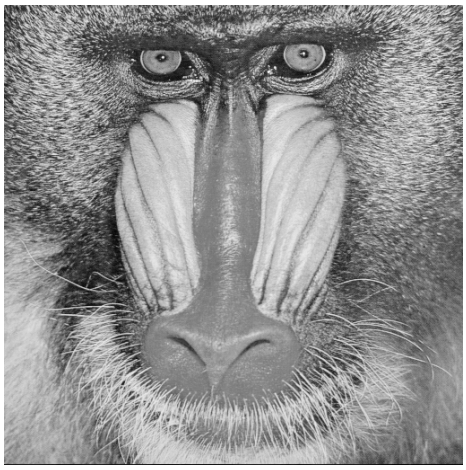

Figure 1: Image d'un mandrill.

1.3.4 Fonctions à implémenter

Il faut implémenter **au minimum** les fonctions suivantes:

- la fonction

  ```
  pgm_result pgm_read_from_file(pgm *p, char *filename);
  ```

  lisant le fichier `filename` et écrivant son contenu dans la variable `p`. Cette fonction retourne un `pgm_result`.

- la fonction

  ```
  pgm_result pgm_write_to_file(pgm *p, char *filename);
  ```

  écrivant le contenu de l'image `p` dans le le fichier `filename`. Cette fonction retourne un `pgm_result`.

2 Les transformations d'images

2.1 Le négatif

Le négatif d'une image consiste à *inverser* la valeur des pixels de l'image par rapport à la valeur maximale permise. Ainsi si on représente `max` niveaux de gris, le négatif d'un pixel de niveau de gris, `p`, est donné par `max-p`. Un exemple de négatif de l'image du mandrill se trouve sur la fig. 2

2.1.1 Fonctions à implémenter

Il faut implémenter **au minimum** la fonction suivante:

- la fonction

  ```
  pgm_result pgm_negative(pgm *neg, pgm *orig);
  ```

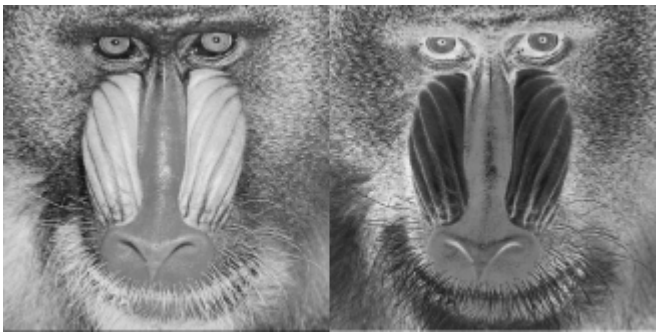

Figure 2: L'original (image de gauche) et le négatif (image de droite) de la photo du mandrill.

calculant le négatif de l'image `orig` et la stockant dans `neg` (qui est également allouée dans cette fonction). L'image `orig` n'est pas modifiée. Cette fonction retourne un `pgm_result`.

2.2 Les symétries

Une symétrie verticale ou horizontale consiste à inverser l'ordre des pixels verticalement ou horizontalement respectivement. La symétrie centrale consiste à échanger les lignes et les colonnes d'une image. Vous pouvez voir un exemple de ces trois symétries aux fig. 3, fig. 4, fig. 5

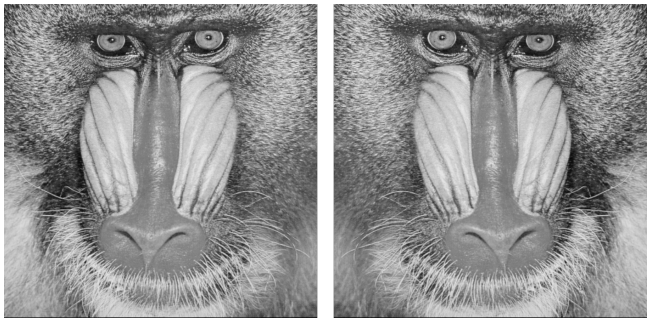

Figure 3: Exemple de symétrie horizontale.

2.2.1 Fonctions à implémenter

Il faut implémenter **au minimum** les fonctions suivantes:

- la fonction

  ```
  pgm_result pgm_symmetry_hori(pgm *sym, pgm *orig);
  ```

  calculant la symétrie horizontale de l'image `orig` et la stockant dans `sym` (qui est également allouée dans cette fonction). L'image `orig` n'est pas modifiée. Cette fonction retourne un `pgm_result`.

- la fonction

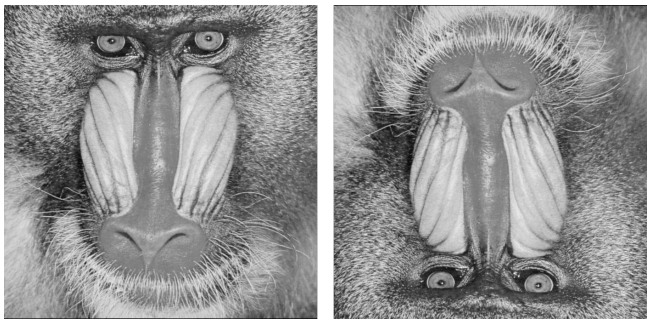

Figure 4: Exemple de symétrie verticale.

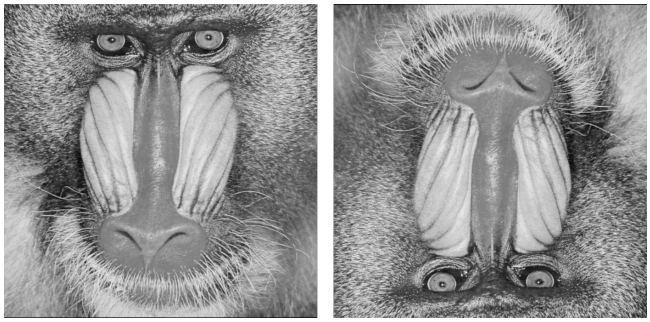

Figure 5: Exemple de symétrie centrale.

```
pgm_result pgm_symmetry_vert(pgm *sym, pgm *orig);
```

calculant la symétrie verticale de l'image `orig` et la stockant dans `sym` (qui est également allouée dans cette fonction). L'image `orig` n'est pas modifiée. Cette fonction retourne un `pgm_result`.

- la fonction

  ```
  pgm_result pgm_symmetry_cent(pgm *sym, pgm *orig);
  ```

  calculant la symétrie centrale de l'image `orig` et la stockant dans `sym` (qui est également allouée dans cette fonction). L'image `orig` n'est pas modifiée. Cette fonction retourne un `pgm_result`.

2.3 Le photomaton

Une étape du photomaton consiste à reproduire l'image donnée 4 fois en 4 fois plus petite. Cette opération sera effectuée plusieurs fois. A chaque étape il n'y a pas de perte d'information: chaque pixel se retrouve simplement déplacé (voir fig. 6).

Pour une image de dimension 8×8, la transformation peut s'observer sur la fig. 7

A partir de cet exemple généraliser cette transformation pour chaque groupe de 4 pixels. Pour bien comprendre la transformation, il faut examiner ce qui se

Figure 6: Exemple d'application de l'algorithme du photomaton où à la fin on a 4 fois l'image originale en 4 fois plus petit.

1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8
2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8
3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8
4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8
5,1	5,2	**5,3**	**5,4**	5,5	5,6	5,7	5,8
6,1	6,2	**6,3**	**6,4**	6,5	6,6	6,7	6,8
7,1	7,2	7,3	7,4	7,5	7,6	7,7	7,8
8,1	8,2	8,3	8,4	8,5	8,6	8,7	8,8

1,1	1,3	1,5	1,7	1,2	1,4	1,6	1,8
3,1	3,3	3,5	3,7	3,2	3,4	3,6	3,8
5,1	**5,3**	5,5	5,7	5,2	**5,4**	5,6	5,8
7,1	7,3	7,5	7,7	7,2	7,4	7,6	7,8
2,1	2,3	2,5	2,7	2,2	2,4	2,6	2,8
4,1	4,3	4,5	4,7	4,2	4,4	4,6	4,8
6,1	**6,3**	6,5	6,7	6,2	**6,4**	6,6	6,8
8,1	8,3	8,5	8,7	8,2	8,4	8,6	8,8

Figure 7: La position avant (gauche) et après (droite) la transformation du photomaton.

passe sur la première ligne.

2.3.1 Fonctions à implémenter

Il faut implémenter **au minimum** la fonction suivante:

- la fonction

  ```
  pgm_result pgm_photomaton(pgm *photomaton,
                            pgm *orig);
  ```

  calculant l'effet photomaton de l'image `orig` et la stockant dans `photomaton` (qui est également allouée dans cette fonction). L'image `orig` n'est pas modifiée. Cette fonction retourne un `pgm_result`.

2.4 Rogner

Le rognage d'une image est une opération assez simple. Elle consiste à extraire une sous partie rectangulaire des pixels de l'image d'origine. Un exemple peut se trouver sur la fig. 8.

2.4.1 Fonctions à implémenter

Il faut implémenter **au minimum** la fonction suivante:

- la fonction

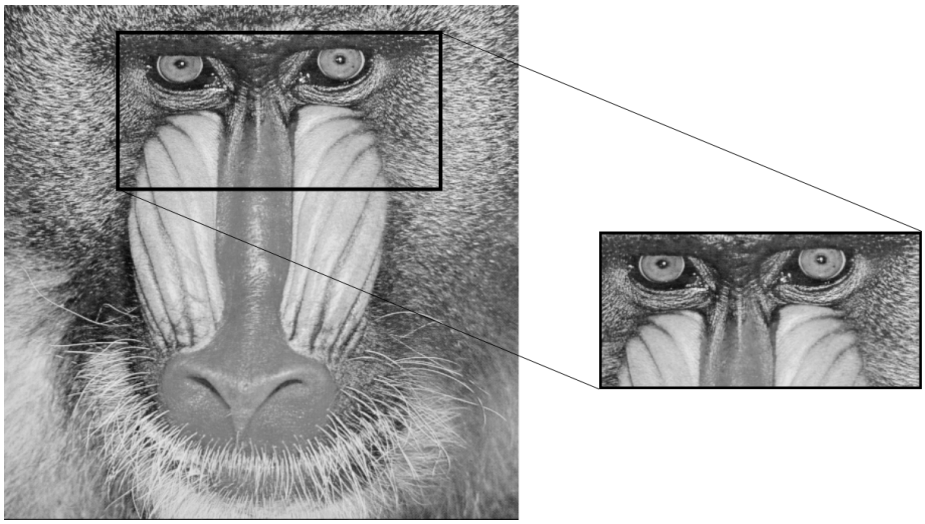

Figure 8: Rognage de la photo de mandrill.

```
pgm_result pgm_crop(pgm *crop, pgm *orig,
                    int32_t x0, int32_t x1,
                    int32_t y0, int32_t y1);
```

calculant la sous-image de `orig` aux coordonnées `x0` à `x1` (non-inclu), et `y0` à `y1` (non-inclu). Le résultat est stocké dans `crop` (qui est également allouée dans cette fonction). L'image `orig` n'est pas modifiée. Cette fonction retourne un `pgm_result`.

2.5 Convolution et filtres [1]

Explication dans la documentation de gimp: https://docs.gimp.org/2.8/fr/plug-in-convmatrix.html

Les matrices de convolutions sont particulièrement utiles dans le traitement d'images. On les appelle également *noyaux* ou *masques*. L'image traitée est obtenue en faisant la *convolution* entre la matrice et l'image. Ce genre d'opération est effectuée tout le temps dans vos téléphones portables pour appliquer des filtres sur vos photos (floutage, vieillissement, ...).

Il existe une grande quantité de noyaux (vous pouvez en trouver des exemple sur la page https://en.wikipedia.org/wiki/Kernel_(image_processing)), mais l'opération pour effectuer le traitement de l'image reste toujours la même. Une

[1] Repris du cours de math de N. Eggenberg et O. Malaspinas

image peut se représenter sous la forme d'une matrice $m \times n$

$$\underline{\underline{A}} = \begin{pmatrix} a_{11} & \dots & a_{1,j-1} & a_{1,j} & a_{1,j+1} & \dots & a_{1n} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i-1,1} & \dots & a_{i-1,j-1} & a_{i-1,j} & a_{i-1,j+1} & \dots & a_{i-1,n} \\ a_{i,1} & \dots & a_{i,j-1} & a_{i,j} & a_{i,j+1} & \dots & a_{i-1,n} \\ a_{i+1,1} & \dots & a_{i+1,j-1} & a_{i+1,j} & a_{i+1,j+1} & \dots & a_{i+1,n} \\ \vdots & \ddots & \vdots & \ddots & \ddots & \ddots & \vdots \\ a_{m,1} & \dots & a_{m,j-1} & a_{m,j+1} & \dots & \dots & a_{mn} \end{pmatrix}. \tag{1}$$

Si nous choisissons une matrice de convolution, $\underline{\underline{C}}$, 3×3 (cela se généralise pour toutes les tailles), de la forme

$$\underline{\underline{C}} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}, \tag{2}$$

nous pouvons écrire la transformation de tous les éléments $a_{i,j}$ de la matrice $\underline{\underline{A}}$, que nous noterons $b_{i,j}$, comme

$$\begin{aligned} b_{i,j} &= \left[\begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix} * \begin{pmatrix} a_{i-1,j-1} & a_{i-1,j} & a_{i-1,j+1} \\ a_{i,j-1} & a_{i,j} & a_{i,j+1} \\ a_{i+1,j-1} & a_{i+1,j} & a_{i+1,j+1} \end{pmatrix} \right] \\ &= c_{11}a_{i-1,j-1} + c_{12}a_{i-1,j} + c_{13}a_{i-1,j+1} \\ &\quad + c_{21}a_{i,j-1} + c_{22}a_{i,j} + c_{23}a_{i,j+1} \\ &\quad + c_{31}a_{i+1,j-1} + c_{32}a_{i+1,j} + c_{33}a_{i+1,j+1}. \end{aligned} \tag{3}$$

On voit donc que la convolution est une combinaison linéaire de tous les éléments d'une sous matrice de $\underline{\underline{A}}$ dont les poids sont donnés par la matrice de convolution. La somme des éléments de la matrice de convolution est en général de 1 (on dit qu'elle est normalisée à 1) pour éviter de modifier la luminosité des pixels.

Illustration 2.1 *(Floutage)*

Si la matrice de convolution est donnée par

$$\underline{\underline{C}} = \frac{1}{9} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}, \tag{4}$$

on voit que son effet est de moyenner la valeur de chaque pixel avec tous ses voisins

$$\begin{aligned} b_{ij} = \frac{1}{9}(&a_{i-1,j-1} + a_{i-1,j} + a_{i-1,j+1} + a_{i,j-1} + a_{i,j} \\ &+ a_{i,j+1} + a_{i+1,j-1} + a_{i+1,j} + a_{i+1,j+1}). \end{aligned} \tag{5}$$

Pour l'implémentation que nous vous proposons ici, les poids sont tous entiers (la `struct matrix` ne contient que des entiers). Afin de pouvoir implémenter un filtre quelconque malgré tout, nous avons rajouté une normalisation dans la signature de la fonction. Dans le cas de la matrice de convolution

$$\underline{\underline{C}} = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix},$$

la variable `kernel` de la fonction

```
pgm_result pgm_conv(pgm *conv, pgm *orig,
                    matrix *kernel, double norm);
```

contiendrait

$$\text{kernel} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix},$$

et la normalisation, `norm`, serait

$$\text{norm} = \frac{1}{9}.$$

On peut aisément constater que ces opérations sont mal définies pour les pixels se trouvant sur les bords de l'image (aux endroits où les pixels n'ont pas suffisamment de voisins pour effectuer l'opération de convolution). Il existe différentes solutions possibles:

1. Diminuer la taille de l'image traitée. L'image sera plus petite que l'image de départ de n pixels, où $n \times n$ est la taille de la matrice de convolution.
2. La taille de l'image est étendue en copiant le dernier pixel sur tous les pixels manquants dans une direction donnée.
3. On effectue une réflexion miroir de l'image dans toutes les directions.

Dans notre cas, on considérera simplement que les pixels absents valent `0`.

2.5.1 Remarques

Il se peut que suite à l'application d'un filtre, vous dépassiez la valeur maximale autorisée pour un pixel, ou obteniez une valeur plus petite que zéro. Dans ces cas il faudra ramener les valeurs dans l'intervalle `[0, max]`:

1. Si la valeur du pixel est inférieure à `0`, alors le pixel vaudra `0`.
2. Si la valeur du pixel est supérieure `max`, alors le pixel vaudra `max`.

Par ailleurs, il se peut que la valeur des pixels ne soit plus entière. Il faudra donc tronquer le nombre obtenu pour en faire un entier.

2.5.2 Fonctions à implémenter

Il faut implémenter **au minimum** la fonction suivante:

- la fonction

```
pgm_result pgm_conv(pgm *conv, pgm *orig,
                    matrix *kernel, double norm);
```

calculant la convolution entre `orig` et le noyau `kernel` ainsi que sa normalisation `norm`. Le résultat est stocké dans `conv` (qui est également allouée dans cette fonction). L'image `orig` n'est pas modifiée. Cette fonction retourne un `pgm_result`.

2.6 Afficher l'image avec la librairie SDL

Uniquement après avoir réalisé **intégralement** toutes les parties qui précèdent, afficher les images en niveau de gris à l'aide de la librairie `SDL2`.

Vous trouverez toutes les fonctions nécessaires dans l'exemple se trouvant sur ce lien. Cet exemple affiche du bruit (des valeurs de gris aléatoires sur un grand nombre de pixels). Vous **devez** réutiliser les fonctions se trouvant dans cet exemple qui sont là pour vous faciliter la vie (et non tenter de réinventer la roue).

Le code dont vous devez vous inspirer est dans le fichier `gfx_example.c` (qui utilise `gfx.h/c`). Outre les fonctions de création/destruction du contexte, la fonction importante est `render()` qui affiche un pixel à la position `x`, `y` à un niveau de gris `color`, à l'aide de la fonction `put_pixel()`.