PROGRAMMATION C
TD/TP 8 - TRAITEMENT D'IMAGE (2)

LICENCE MATHS-INFO
19-20 MARS 2012

Introduction

Le premier TP de la série sur le traitement des images a permis d'avoir des fonctions de lecture/écriture de fichiers pgm. Ces fonctions de lecture/écriture sont disponibles sur le site du module [1].

Nous avons également vu comment faire un traitement simple sur chaque pixel d'une image (effets de binarisation et de négatif) : une double boucle sur les lignes et les colonnes permet de calculer les nouveaux pixels un par un. Ce principe sera conservé par la suite, pour faire des effets plus élaborés.

Ce second TP a pour objectif de mettre en œuvre des effets plus élaborés. Il nécessite une lecture attentive de l'énoncé et un effort de rigueur de programmation. Vous serez guidés, en particulier sur le découpage du code en plusieurs fonctions. N'hésitez pas à recourir à un brouillon avant de coder ou pendant le codage, que ce soit pour faire des schémas ou écrire des morceaux de code. Les fonctions seront testées dans `test_image_processing_TP8.c`. La documentation sera générée via Doxygen.

Code de départ

Un petit changement s'est inséré par rapport au TP 7 : dans le type `Image`, les pixels d'une image ne sont plus stockés dans un tableau de `double` directement accessible par le champ `data` (de type `double *`) ; le champ `data` est désormais de type `double **` afin de stocker un tableau de pointeurs sur les lignes de l'image (cf. figure 1). Ce changement permet d'accéder aux pixels de l'image avec la double indexation, plutôt que de faire la conversion indices 2D ligne/colonne → indice 1D. On pourra ainsi utiliser `p_image->data[i][j]` au lieu de `p_image->data[sub2ind(i,j,p_image->width)]`.

Vous avez donc le choix entre :
- utiliser le code fourni sur le site du module ;
- ou modifier votre propre code pour intégrer ce changement, notamment :
  - modifier le type du champ `data` dans le type `Image` ;
  - mettre à jour la fonction `create_image` ;
  - mettre à jour la fonction `free_image` ;
  - remplacer tous les accès 1D aux pixels (avec conversion) par un accès 2D (sans conversion) ;
  - supprimer les fonctions de conversion (`sub2ind, ind2row, ind2col`) qui ne servent plus à rien ;

1. `http://www.lif.univ-mrs.fr/~vemiya/?page=L2C_2011_2012`

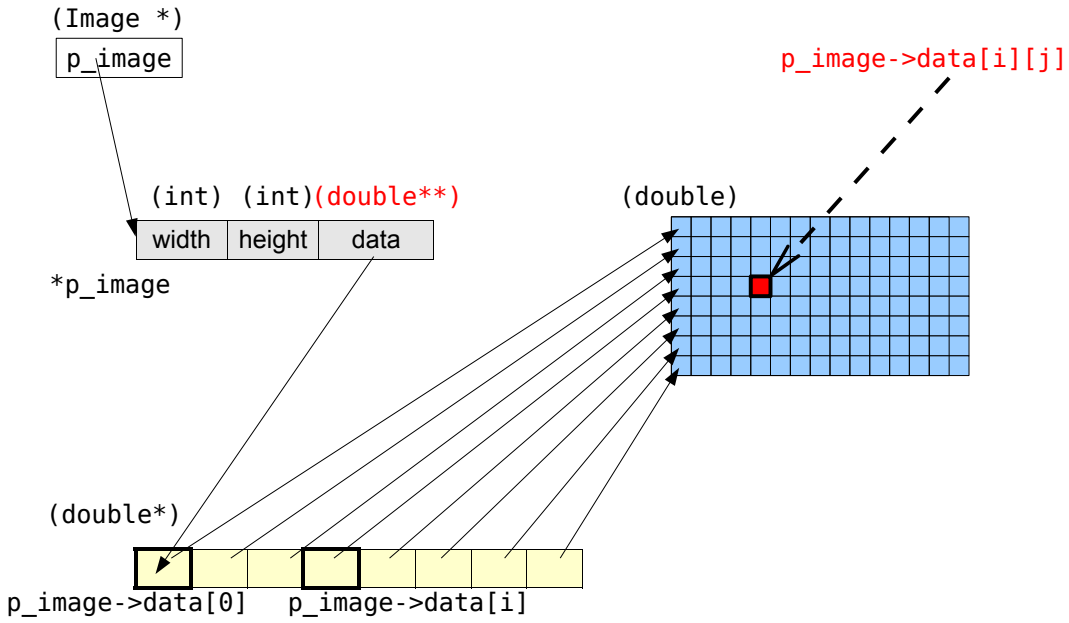

FIGURE 1. Structure pour manipuler les images avec accès 2D.

Rappel synthétique du code existant :

```
#typedef struct {
  int width;
  int height;
  double **data;
} Image;

Image* pgm_read(char *filename);
void pgm_write(char *filename, Image *p_image, int max);

Image *create_image(int width, int height);
void free_image(Image *p_image);
Image *copy_image(Image *p_image);
void affiche_image(Image *p_image);
```

Partie 1 : effet de miroir et de rotation

En utilisant les prototypes ci-dessous, écrivez cinq fonctions (dans `image_effects.c`) qui prennent en argument un pointeur sur une image et créent une nouvelle image qui est, respectivement :

- le miroir horizontal de l'image de départ ;
- le miroir vertical de l'image de départ ;
- une rotation de 90 degrés à droite de l'image de départ (sens des aiguilles) ;
- une rotation de 180 degrés de l'image de départ ;
- une rotation de 90 degrés à gauche de l'image de départ (sens trigonométrique).

Prototypes à utiliser :

```
Image *miroir_horizontal(Image *p_image);
Image *miroir_vertical(Image *p_image);
Image *rotation90_droite(Image *p_image);
Image *rotation180(Image *p_image);
Image *rotation90_gauche(Image *p_image);
```

Traitement par patches

Nous allons mettre en œuvre plusieurs traitements par patches, comme expliqué en cours (cf. figure 2). Chaque pixel de l'image de sortie sera le résultat d'une fonction appliquée sur le patch autour du pixel correspondant dans l'image d'origine.

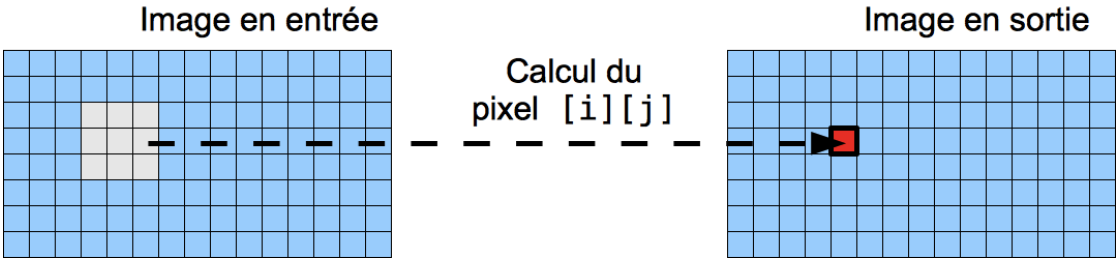

Figure 2. Traitement par patches.

Un patch sera une zone 3×3 autour du pixel courant (les bords de l'image seront ignorés). On définira les contantes suivantes dans `image_types.h`

```
#define PATCH_DIM 3
#define PATCH_RADIUS (PATCH_DIM-1)/2
```

Comme illustré sur la figure 3, un patch est un tableau de `PATCH_DIM` pointeurs sur `double` : le i_0-ième pointeur pointe sur l'élément de l'image correspondant au premier élément de la ligne i_0 du patch.

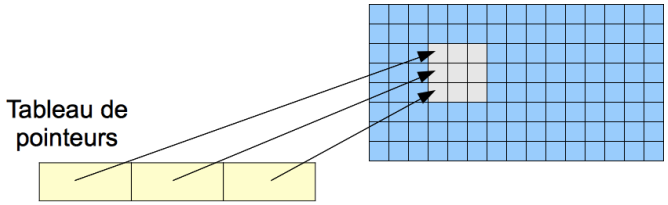

FIGURE 3. Mise en œuvre d'un patch avec un tableau de pointeurs.

PARTIE 2 : FILTRAGE MOYEN

Le premier traitement par patch est le filtrage moyen : chaque pixel y_{ij} de la nouvelle image y est obtenu en faisant la moyenne des 9 pixels voisins dans le patch d'origine x :

$$y_{ij} = \frac{1}{9} \sum_{i_0=-1}^{1} \sum_{j_0=-1}^{1} x_{i+i_0,j+j_0}$$

C'est une fonction typiquement utilisée pour faire du débruitage d'image. Vous l'appliquerez aux images bruitées.

Vous procèderez via deux fonctions décrites ci-dessous.

Fonction de traitement d'un patch. Ecrivez une fonction qui prend un patch en argument, calcule et renvoie la valeur moyenne de ses pixels. Son prototype est :

```
double mean_filter_pix(double **patch);
```

Fonction de traitement de l'image entière. Ecrivez une fonction qui prend une image en argument, applique la fonction `mean_filter_pix` à tous les pixels (sauf sur les bords) pour construire une nouvelle image et la renvoyer. Son prototype est :

```
Image *mean_filter(Image *p_image);
```

PARTIE 3 : FILTRAGE MÉDIAN

Le second traitement par patch est le filtrage médian : chaque pixel y_{ij} de la nouvelle image y est obtenu en triant les 9 pixels voisins dans le patch d'origine et en prenant la valeur médiane.

C'est une autre fonction typiquement utilisée pour faire du débruitage d'image. Vous l'appliquerez aux images bruitées.

Vous procèderez via deux fonctions comme précédemment.

Fonction de traitement d'un patch. Ecrivez une fonction qui prend un patch en argument, calcule et renvoie la valeur médiane de ses pixels. Les pixels pourront être triés avec la fonction `qsort` de la bibliothèque standard. Le prototype à utiliser est :

```
double median_filter_pix(double **patch);
```

Fonction de traitement de l'image entière. Ecrivez une fonction qui prend une image en argument, applique la fonction `median_filter_pix` à tous les pixels (sauf sur les bords) pour construire une nouvelle image et la renvoyer. Son prototype est :

```
Image *median_filter(Image *p_image);
```

Résultats de débruitage. Les images bruitées fournies comportent deux types de bruit : du bruit impulsionnel (seuls quelques pixels sont bruités) et du bruit blanc (tous les pixels sont bruités, mais moins violemment). Que pensez-vous de l'efficacité des deux techniques de filtrage pour débruiter ces images ? L'une est-elle plus adaptée pour un type de bruit ? Avez-vous une explication ?

Partie 4 : vers un code plus modulaire

Vous avez peut-être constaté que la mise en œuvre des deux traitements par patch précédents est très similaire. En particulier, la fonction globale est exactement la même, il suffit juste de changer le nom de la fonction de traitement d'un patch. Dans cette partie, vous allez faire en sorte de passer la fonction de traitement en argument d'une fonction générique unique.

Remplacez les deux fonctions globales `mean_filter` et `median_filter` par une seule fonction qui prend en argument une image ainsi qu'une fonction de traitement d'un patch, applique cette fonction à tous les pixels (sauf sur les bords) et renvoie la nouvelle image. Son prototype est :

```
Image *patch_based_process(Image *p_image, double (*f)(double **) );
```

Partie 5 : filtre de Sobel

Voici un autre filtre qui n'est pas utilisé pour le débruitage mais pour la détection de contours. Un contour est une variation locale importante de la valeur des pixels : la détection de contour s'appuie donc sur le calcul de dérivées. La technique consiste à calculer deux images intermédiaires correspondant aux gradients horizontal et vertical, puis à les combiner pour calculer l'image finale.

Gradient horizontal. Il s'agit de calculer une dérivée discrète selon la dimension horizontale, afin de faire ressortir les grandes variations dans cette direction. Chaque pixel y_{ij} de la nouvelle image y est obtenu en faisant la somme pondérée des 9 pixels voisins dans le patch d'origine x :

$$y_{ij} = \sum_{i_0=-1}^{1} \sum_{j_0=-1}^{1} x_{i+i_0,j+j_0} H_{i_0 j_0} \qquad \text{avec } H = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

En appliquant à la main cette formule sur les patches $\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$, $\begin{pmatrix} 0 & 0.5 & 1 \\ 0 & 0.5 & 1 \\ 0 & 0.5 & 1 \end{pmatrix}$ et $\begin{pmatrix} 0 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \\ 1 & 1 & 1 \end{pmatrix}$, essayez de comprendre comment cette opération calcule une dérivée et met en valeur un contour.

Créez une variable globale pour stocker H une fois pour toutes, puis écrivez une fonction pour traiter un patch, dont le prototype est

```
double gradientH_pix(double **patch);
```

Gradient vertical. On procède de même, selon l'autre dimension. Chaque pixel z_{ij} de la nouvelle image z est défini par

$$z_{ij} = \sum_{i_0=-1}^{1} \sum_{j_0=-1}^{1} x_{i+i_0,j+j_0} V_{i_0 j_0} \qquad \text{avec } V = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

Appliquez à la main ce filtre comme précédemment et commentez.

Créez une variable globale pour stocker V une fois pour toutes, puis écrivez une fonction pour traiter un patch, dont le prototype est

```
double gradientV_pix(double **patch);
```

Norme du gradient. L'image finale w est construite à partir des images y et z obtenues précédemment via la formule suivante :

$$w_{ij} = \sqrt{y_{ij}^2 + z_{ij}^2}$$

Ecrivez une fonction qui calcule les deux gradients (utilisez la fonction générique de traitement par patch) puis calcule l'image finale (la fonction `sqrt` est dans `math.h`). Vous veillerez à libérer la mémoire de façon adéquate. Son prototype est :

```
Image *sobel_filter(Image *p_image);
```

Partie 6 : finalisation du logiciel

Vous disposez maintenant de nombreux effets : binarisation, négatif, débruitage pour bruit impulsionnel, débruitage pour bruit blanc, détection de contour. Faites-en un exécutable `amushop` qui puissent être lancé en ligne de commande de la façon suivante :

```
$ amushop negatif image_entree.pgm image_sortie.pgm
$ amushop binarisation image_entree.pgm image_sortie.pgm
$ amushop attenue_bruit_impuls image_entree.pgm image_sortie.pgm
$ amushop attenue_bruit_blanc image_entree.pgm image_sortie.pgm
$ amushop detection_contour image_entree.pgm image_sortie.pgm
```