# [Math-L312] TP 7 : Dessinez, c'est gagné!

Adrien Semin
adrien.semin@math.u-psud.fr

Dans ce TP, nous allons apprendre à manipuler des images `PGM` (portable graymap file format) et `PPM` (portable pixmap file format). Ce sont des formats d'images multi-plateformes (tout comme le bitmap).

## 1 Structure des images

### 1.1 Image PGM

Une image `PGM` à $N_H$ lignes de pixels et $N_W$ colonnes de pixels peut être vue comme un tableau $T$ de taille $N_H \times N_W$ tel que le pixel de la $i^{\text{ème}}$ ligne et de la $j^{\text{ème}}$ colonne (où $i$ est compris entre 0 et $N_H - 1$, et $j$ est compris entre 0 et $N_W - 1$) est stocké dans la $(N_W * i + j + 1)^{\text{ème}}$ case du tableau $T$ (qui, rappelons-le, correspond à la case `T[NW * i + j]` lorsque nous codons). Par exemple, la numérotation du tableau pour $N_H = N_W = 10$ donne

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 |
| 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 |
| 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 |
| 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 |
| 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 |
| 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 |

Chaque case du tableau est un nombre positif entier codé sur un ou deux octects (pour les images sur lesquelles nous travaillons, le codage est sur un octet). En `C`, nous représentertons l'octet par le type `OCTET` (qui correspond en fait à un caractère non signé).

```
typedef unsigned char OCTET;
```

La couleur noire est associée au caractère de code `ASCII` 0, et la couleur blanche est associée au caractère de code `ASCII` 255. Entre les deux caractères, nous avons

### 1.2 Image PPM

Une image `PPM` à $N_H$ lignes de pixels et $N_W$ colonnes de pixels peut être vue comme un tableau $T$ de taille $N_H \times N_W$, où chaque pixel est stocké sur trois octets (un octet pour chaque couleur primaire : le rouge, le vert et le

bleu). Nous considérerons plutôt que le tableau $T$ est de taille $3 \times N_H \times N_W$ tel que pour le pixel de la $i^{\text{ème}}$ ligne et de la $j^{\text{ème}}$ colonne, `T[3*(NW*i+j)]` contienne la proportion de rouge, `T[3*(NW*i+j)+1]` contienne la proportion de vert et `T[3*(NW*i+j)+2]` contienne la proportion de bleu. Par exemple, la numérotation du tableau pour $N_H 8$, $N_W = 4$ donne

| | | | |
|---|---|---|---|
| 0 1 2 | 3 4 5 | 6 7 8 | 9 10 11 |
| 12 13 14 | 15 16 17 | 18 19 20 | 21 22 23 |
| 24 25 26 | 27 28 29 | 30 31 32 | 33 34 35 |
| 36 37 38 | 39 40 41 | 42 43 44 | 45 46 47 |
| 48 49 50 | 51 52 53 | 54 55 56 | 57 58 59 |
| 60 61 62 | 63 64 65 | 66 67 68 | 69 70 71 |
| 72 73 74 | 75 76 77 | 78 79 80 | 81 82 83 |
| 84 85 86 | 87 88 89 | 90 91 92 | 93 94 95 |

## 2 Exploitation des images :

Il vous faut tout d'abord récupérer

- la bibliothèque `allocation_tableaux.h` téléchargeable à l'adresse `/home/doc/semin/TP/h/allocation_tableaux.h` (je parle ici de la nouvelle version de la bibliothèque)
- la bibliothèque `image_ppm.h` téléchargeable à l'adresse `/home/doc/semin/TP/h/image_ppm.h`

Un programme-type de manipulation d'image est la copie d'une image dans une autre image,

```
#include <stdio.h>
#include <image_ppm.h>
#include <allocation_tableaux.h>
#include <assert.h>
int main(int argc, char* argv[])
{

  char cNomImgLue[250], cNomImgEcrite[250];
  int nH, nW, nTaille, i, j;

  if (argc != 3)
    {
      printf("Usage: %s ImageIn.pgm ImageOut.pgm\n",argv[0]);
      exit(1) ;
    }

  assert(sscanf(argv[1],"%s",cNomImgLue)==1);
  assert(sscanf(argv[2],"%s",cNomImgEcrite)==1);

  lire_nb_lignes_colonnes_image_pgm(cNomImgLue, &nH, &nW);

  nTaille = nH * nW;
```

```
  allocation_tableau(ImgIn, OCTET, nTaille);

  lire_image_pgm(cNomImgLue, ImgIn, nH * nW);

  allocation_tableau(ImgOut, OCTET, nTaille);

  for (i=0; i < nH; i++)
    for (j=0; j < nW; j++)
      ImgOut[i*nW+j] = ImgIn[i*nW+j];

  ecrire_image_pgm(cNomImgEcrite, ImgOut, nH, nW);

  desallocation_tableau(ImgIn);
  desallocation_tableau(ImgOut);
  return 0;
}
```

Ce programme prend deux arguments lors de son exécution (que l'on stocke dans des chaînes de caractères avec l'instruction `sscanf` - cette instruction admet une syntaxe similaire à la syntaxe de l'instruction `scanf`)

```
>   gcc image_copy.c -I. -o image_copy
>   image_copy lena.pgm output.pgm
```

où le fichier `lena.pgm` est téléchargeable à l'adresse `/home/doc/semin/TP/img/lena.pgm`. On visualisera le fichier de sortie avec l'instruction

```
>   xv output.pgm
```

# 3 Travail à rendre

Les images à tester seront téléchargeables à l'adress `/home/doc/semin/TP/img/`

1. `zoom.c` ***∗ À Rendre! ∗*** Ecrire un programme `zoom.c` qui prendra un nom d'image `pgm` d'entrée et un nom d'image `pgm` de sortie à l'exécution et qui construira une image de sortie deux fois plus grande en largeur et en hauteur que l'image d'entrée. Les pixels de la $(2i+l)^{\text{ème}}$ ligne et $(2j+m)^{\text{ème}}$ colonne de l'image de sortie, avec $k, m \in \{0, 1\}$, devront être égaux au pixel de la $i^{\text{ème}}$ ligne et $j^{\text{ème}}$ colonne de l'image d'entrée.
2. `seuil.c` ***∗ À Rendre! ∗*** Ecrire un programme `seuil.c` qui prendra un nom d'image `pgm` d'entrée et un nom d'image `pgm` de sortie à l'exécution et qui construira l'image de sortie de la manière suivante : le pixel de la $i^{\text{ème}}$ ligne et de la $j^{\text{ème}}$ colonne de l'image de sortie sera blanc si le pixel correspondant de l'image d'entrée a une valeur au moins égale à 128, et sera noir sinon.
3. `codage_nb.c` ***∗ À Rendre! ∗*** Ecrire un programme `codage_nb.c` qui prendra un nom d'image `pgm` d'entrée, un nom d'image `pgm` de sortie et un nombre entier entre 0 et 255 à l'exécution, et qui construira une image de sortie telle que le pixel de la $i^{\text{ème}}$ ligne et de la $j^{\text{ème}}$ colonne soit le résultat de l'opération du `xor` entre le pixel de la $i^{\text{ème}}$ ligne et de la $j^{\text{ème}}$ colonne de l'image d'entrée d'une part, et le nombre rentré à l'exécution d'autre part.
4. `conversion_img.c` Ecrire un programme `conversion_img.c` qui prendra un nom d'image `ppm` (couleur donc) d'entrée et un nom d'image `pgm` de sortie à l'exécution et qui construira une image de sortie en noir et blanc tel que son pixel de la $i^{\text{ème}}$ ligne et de la $j^{\text{ème}}$ colonne soit calculé par

   $$pixel_{sortie} = 0.25\ r\ + 0.5\ g\ + 0.25\ b$$

   où $r$, $g$ et $b$ sont respectivement les composantes rouge, verte et bleue du pixel de la $i^{\text{ème}}$ ligne et de la $j^{\text{ème}}$ colonne de l'image d'entrée.
5. `seuil_couleur.c` ***∗ À Rendre! ∗*** Ecrire un programme `seuil_couleur.c` qui prendra un nom d'image `ppm` d'entrée et un nom d'image `ppm` de sortie à l'exécution et qui construira l'image de sortie de la manière suivante : chaque composante du pixel de la $i^{\text{ème}}$ ligne et de la $j^{\text{ème}}$ colonne de l'image de sortie vaudra 255 si la composante du pixel correspondant de l'image d'entrée a une valeur au moins égale à 128, et vaudra 0 sinon.