

# DM3 MP2I : fichiers image

## Thèmes

1. Lecture/écriture dans un fichier en C.
2. Structures, allocations.
3. Compilation séparée.
4. Fichiers d'images en noir et blanc ou en niveau de gris.

**Avant de commencer ce devoir** , il faut se (re)documenter sur la lecture/écriture dans des fichiers en C (par exemple ici).

De plus, nous utilisons des images aux formats **PBM** et **PGM**. On peut avec profit consulter la page Wikipedia concernant ces formats.

Pour afficher une image **pbm**, il faut installer le visionneur **eog** :

```
$ sudo apt install eog
```

Cela dit, **emacs** sait très bien afficher ce format. Par défaut, il l'affiche comme une image. Si on veut considérer l'image comme du texte, cliquer sur **Image**→**Show as text**, ou mieux encore faire **CTRL C** deux fois.

Dans l'archive du devoir, on trouve un fichier **j.pbm** représentant le caractère **J**. Il est reproduit figure 1.

```
P1
# Un exemple bitmap de la lettre "J"
7 10
#pas de ligne pour le maximum
0 0 0 0 0 0 0
0 0 0 0 0 1 0
0 0 0 0 0 0 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 1 0 0 0 1 0
0 0 1 1 1 0 0
0 0 0 0 0 0 0
```

FIGURE 1 – Un J

Un fichier **flamantsNB\_ascii.pgm** est également présent. La figure 2 en donne les premières lignes :

FIGURE 2 – Premières lignes d’une image de flamants

```

P2
690 460
255
184 184 185 186 186 187 188 188 187 187 188 189
189 188 188 187 188 187 187 187 187 188 188 188
187 187 188 188 189 189 189 189 186 186 186 187
188 189 190 191 192 192 192 192 192 192 193 193
192 192 192 192 192 192 193 193 194 194 194 195
196 196 195 194 193 193 193 193 193 193 193 192

```

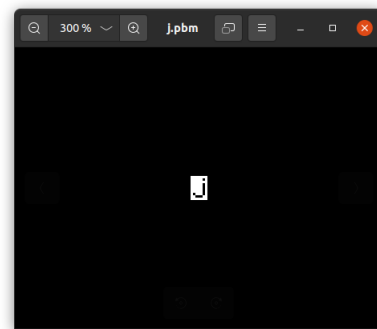
Les passages à la ligne sont facultatifs dans les fichiers PBM comme PGM. Le plus sage est d’éviter des lignes de plus de 80 colonnes comme c’est encore l’usage chez les développeurs.

Dans les fichiers PBM, il n’y a pas de ligne pour indiquer le maximum des valeurs possibles (c’est toujours 1). En revanche, dans un fichier PGM, une ligne indique le maximum des valeurs possibles pour les niveaux de gris (pour nous, ce sera toujours 255). De plus, un chiffre 0 dans un fichier PBM indique la couleur blanche alors que c’est la couleur noire dans un fichier PGM !

Listing 1 – Commande pour afficher **j.ppm**

```
$ eog j.pbm
```

Entrons la commande ci-dessus.



Une fenêtre contenant l’image s’ouvre alors. Comme notre image est toute petite (70 pixels carrés), il peut être utile de zoomer (avec la combinaison de touche **CTRL +**).

Le repère que nous utilisons pour les images est représenté figure 3. Les pixels ont des coordonnées entières. Les pixels visibles sont situés dans le 1/4 de plan  $\begin{cases} x \geq 0 \\ y \geq 0 \end{cases}$ . Par exemple, le point du  $j$  de la figure 1 a pour coordonnées (1, 5).

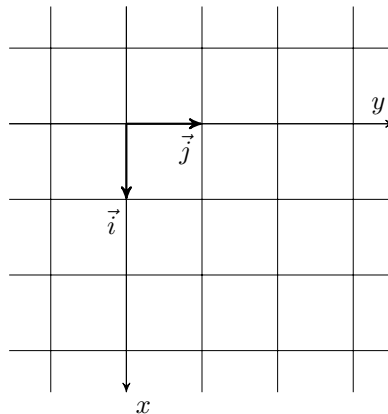
Par commodité, on ne tolère pas les commentaires dans les fichiers images. Comme les niveaux de gris apparaissant dans nos fichiers sont inférieurs à 255, on privilégie le type `uint8_t` pour les exprimer. Le spécifieur de format correspondant est `%hhu`.

## 1 Description du projet

Le projet est organisé avec les fichiers suivants :

**Fichier d’en-têtes** Le fichier **bibli.h** contient toutes les inclusions de fichiers d’en-têtes des

FIGURE 3 – Repère de l'image



fonctions de bibliothèques, les rappels de définition des constantes, les prototypes des fonctions du devoir. Aucune inclusion autre que celles données dans **bibli.h** n'est autorisée. Ce fichier NE DOIT PAS être modifié.

**Fichiers outils** Les fichiers **alloc.c** et **segments.c** sont donnés et ne doivent pas être modifiés :

**alloc.c** Les allocations et libérations de ce devoir se font exclusivement avec les fonctions

`myalloc`, `mycalloc` et `myfree`

```

1 void * mymalloc(size_t size);
2 void myfree(void *ptr);
3 void * mycalloc( size_t elementCount, size_t elementSize );
4 extern int nballoc;
5 extern int nbfree;
6
```

Chaque allocation (resp. libération) incrémente la variable globale `nballoc` (resp. `nbfree`). À la fin de l'exécution de votre code, les deux variables doivent contenir les mêmes valeurs.

**segments.c** Contient la fonction

```

1 void trace_segment(img *im, pt p0, pt p1);
2
```

qui permet de limiter le phénomène d'*aliasing* lors du tracé du segment reliant le premier point au second. La compréhension du code de ce fichier n'est pas un objectif de ce devoir.

**Makefile** Un **Makefile** est fourni pour son côté pratique. Vous pouvez l'utiliser ou non, le modifier à votre guise mais il ne doit pas figurer dans l'archive rendue.

Les tests et les `main` sont supposés écrits dans deux fichiers `divers_main.c` et `divers_polygon.c`.

**Fichiers de code divers.c** Ce fichier contient toutes les fonctions à écrire concernant les images en niveau de gris. Une seule fonction (`seuil`) retourne une image en noir et blanc.

**polygon.c** Ce fichier est destiné au tracé des *flocons de Koch*. Il s'agit d'une courbe fractale très célèbre.

Ce projet peut être réalisé seul ou en binôme. Chaque étudiant dépose son devoir sur Cahier de Prépa même si le devoir a été réalisé à deux. L'archive déposée contient exclusivement les fichiers **divers.c** et **polygon.c**. Aucun test ni fonction `main` ne doit figurer dans le travail rendu.

## 2 Images en niveau de gris

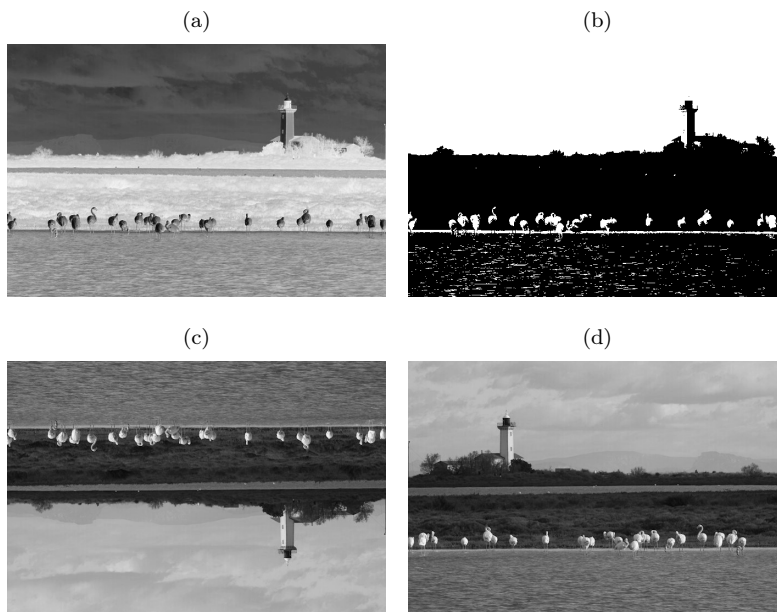
Les images sont représentées par la structure `img` qui possède des champs pour la largeur de l'image, sa hauteur, la valeur maximale des couleurs. Les couleurs de pixels sont stockées dans un tableau unidimensionnel dynamique (ce n'est pas une matrice) : il s'agit d'un simple tableau de nombres. Le champ `max` vaut 255 ou 1 selon que l'image suit le format P2 ou P1.

- Q.1** Écrire la fonction `img* uni(int w, int h, bool gray, COLORS maxi);` qui renvoie un pointeur sur une image à  $w$  colonnes et  $h$  lignes au format PGM si le booléen `gray` vaut `true` et PBM sinon. Tous les pixels de l'image valent 0. Le rendu graphique est donc une image noire si le format est PGM et blanche pour le format PBM. La couleur `maxi` vaut à priori 255 ou 1.
- Q.2** Écrire la procédure `img * fromfile(char *)`; qui ouvre en lecture un fichier PBM ou PGM dont le nom est passé en paramètre. Elle renvoie un pointeur sur une image dont les champs sont remplis en fonction du contenu de ce fichier. Le champ `max` vaut 1 ou 255 selon que le nombre magique est P1 ou P2.
- Q.3** Écrire la procédure `void tofile(char*, img *)`; qui remplit un fichier au format P1 ou P2 selon le champ `magic` du pointeur sur image passé en paramètre.
- Q.4** On effectue des changements simples sur l'image :
- (a) Écrire la fonction `img * neg(img *)`; qui renvoie le négatif de l'image dont l'adresse est passée en paramètre. L'image renvoyée a le même format que l'image initiale.
  - (b) Écrire la fonction `img * seuil (img * im, COLORS blanc)` qui renvoie une image au format PBM à partir d'une image au format PGM. Tous les pixels dont la couleur est supérieure au seuil `blanc` sont considérés comme blancs et les autres sont vus comme noirs.
  - (c) Écrire la fonction `img * symh(img *im)`; qui effectue une symétrie horizontale de l'image (voir figure 4c). Il s'agit juste d'inverser l'ordre des lignes.
  - (d) Écrire la fonction `symv : img -> img` qui effectue une symétrie verticale de l'image (voir figure 4d). Il s'agit juste d'inverser l'ordre des colonnes.

D'autres fonctions bien pratiques peuvent être implémentées mais elles ne sont pas évaluées :

1. `void display_img(img *)`; qui affiche à l'écran le contenu d'une image sous la même forme qu'un fichier PBM ou PGM.
2. `void free_img(img *)`; Cette procédure libère un pointeur sur image et son champ `colors`.

FIGURE 4 – La photo des flamants en négatif, seuillée à 128, symétrisée horizontalement puis verticalement



## 2.1 Tests

Les tests produisent deux fichiers images. Voici l'affichage des images par `display_img` :

```

2  img * img = uni(15,10,true,255);
3  display_img(img);
4  tofile ("uni_p2.pgm",img);
5  free_img(img);
6  img = uni(15,10,false,1);
7  display_img(img);
8  tofile ("uni_p1.pbm",img);
9  free_img(img);

```

---

Listing 2 – Tests Uni

Listing 3 – Rendu sur console

[illegible]

Le fichier **feep.pgm** a été récupéré sur Wikipedia (PGM).

```
37 | img * im = fromfile("feep.pgm");
38 | display_img(im);
39 | free_img(im);
```

Listing 4 – Tests FROMFILE

Listing 5 – Rendu sur console

```

P2
  24 7
  15
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 3 3 3 3 0 0 7 7 7 7 0 0 1 1 1 1 1 1 0 0 1 5 1 5 1 5 0
0 3 0 0 0 0 0 0 7 0 0 0 0 0 1 1 0 0 0 0 0 1 5 0 0 1 5 0
0 3 3 3 0 0 0 7 7 7 0 0 0 1 1 1 1 1 0 0 0 1 5 1 5 1 5 0
0 3 0 0 0 0 0 7 0 0 0 0 0 1 1 0 0 0 0 0 1 5 0 0 0 0
0 3 0 0 0 0 0 7 7 7 7 0 0 1 1 1 1 1 1 0 0 1 5 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

```

```
50 img * im = fromfile("feep.pgm");
51 img * im2 = neg(im);
52 tofile("feep_neg.pgm",im2);
53 display_img(im2);
54 free_img(im);
55 free_img(im2);
```

Listing 6 – Tests NEG

Listing 7 – Rendu sur console

[illegible]

```

66 img * im = fromfile("flamantsNB_ascii.pgm");
67 img * im2 = seuil(im,128);
68 tofile("flamantsNB_ascii_seuil.pbm",im2);
69 free_img(im);
70 free_img(im2);

```

Listing 8 – Tests SEUIL

Début du gros fichier **flamantsNB\_ascii\_seuil.pbm** :

Listing 9 – Rendu sur console

P1  
690 460  
0 0 0 0 0

La première ligne est remplie de 0 (blanc).

Début du gros fichier **flamantsNB\_ascii\_symh.pgm** :

Listing 11 – Rendu sur console

P2
690 460
255
71 73 75 74 73 73 77 80 86 87 91 92 90 95 104 108 100

Début du gros fichier **flamantsNB\_ascii\_symv.pgm** :

Listing 13 – Rendu sur console

```
P2
690 460
255
  180 180 177 177 179 181 184 185 186 186 185 186 187 188 1
```

Enfin, voici des tests pour les symétries :

```

74 img * im = fromfile("flamantsNB_ascii.pgm");
75 img * im2 = symh(im);
76 tofile ("flamantsNB_ascii_symh.pgm",im2);
77 free_img(im);
78 free_img(im2);

```

Listing 10 – Tests SYMH

```
83 img * im = fromfile("flamantsNB_ascii.pgm");
84 img * im2 = symv(im);
85 tofile ("flamantsNB_ascii_symv.pgm",im2);
86 free_img(im);
87 free_img(im2);
```

Listing 12 – Tests SYMH

### 3 Images en noir et blanc

Dans cette section, les images sont au format PBM. Donc leur nombre magique est P1, le maximum des valeurs est 1 (pixel noir). Un *point* est représenté par la structure `pt`. Il possède une couleur flottante et deux coordonnées flottantes. Une ligne brisée est représentée par la

structure `line`. Par défaut, le fond d'une image est blanc et les couleurs des points d'une ligne brisée sont noires. On ne demande pas de vérifier cela par la suite.

### 3.1 Outils

**Q.1** Écrire la fonction `pt proche(pt p, int absmax, int ordmax);`. A partir d'un point dont les coordonnées sont flottantes, elle renvoie le point de coordonnées entières le plus proche de ce point. La couleur reste la même que celle du point d'origine (donc 0 ou 1). L'objectif étant de colorer le pixel proche de `p` dans une image, il faut que `p` soit dans les limites de l'image : l'ordonnée et l'abscisse maximum de l'image sont données. Une erreur d'assertion est soulevée si le point sort de l'image.

Avec le point `pt p = {1.,50.7,30.25}`, `proche(p,80,60)` renvoie `{1.,51.,30.}`.

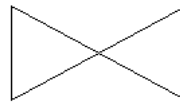
Mais `proche(p,50,60)` soulève une erreur d'assertion.

**Q.2** Écrire la procédure `void draw_line(img *im, line *l);` qui modifie une image de façon à colorer les pixels traversés par une ligne brisée.

Comme les points d'une ligne brisée ont des coordonnées flottantes, l'usage de la fonction `proche` est nécessaire (on ne trace pas les sommets de la ligne mais uniquement des points de coordonnées entières qui leur sont proches).

Avec

```
1 pt points[5]={1,10,20},{1,80,150},{1,10,150},{1,80,20},{1,10,20}};
2 line l;
3 l.card=5;
4 l.sommets = points;
5 img * im = uni(160,90,false,1);
6 draw_line(im,&l);
7 tofile("butterfly.pbm",im);
8 free_img(im);
9
```



le fichier **butterfly.pbm** correspond à l'image :

**Q.3** Écrire la fonction `pt third(pt A, pt B);` qui renvoie l'image de  $B$  par la rotation d'angle  $\frac{\pi}{3}$  et de centre  $A$ .

Avec

```
1 pt p0 = {1,50,30};
2 pt p2 = {1,90,100};
3 pt p1 = third(p0,p2);
4 printf("p1={%f;%f;%f}\n",p1.color,p1.x,p1.y);
5
```

on obtient l'affichage

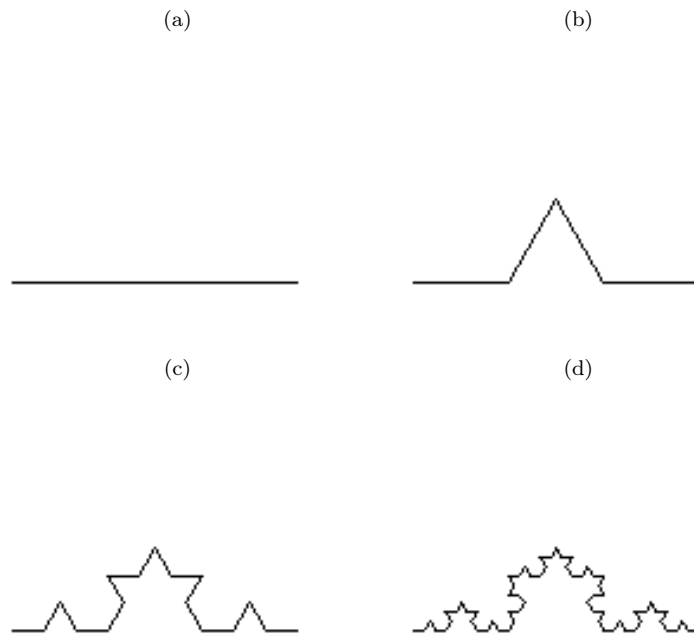
```
p1={1.000000;9.378220,99.641014}
```

### 3.2 Construction des flocons de Koch

Considérons une ligne brisée  $P_0, \dots, P_n$  dont tous les segments sont de longueur égales. Une étape de construction d'un *flocon de Koch* consiste à décomposer chaque segment  $[P_i, P_{i+1}]$  (pour  $0 \leq i < n$ ) en 4 sous-segments  $[Q_j^i, Q_{j+1}^i]$  (pour  $0 \leq j \leq 3$ ) tels que

- $Q_0^i = P_i$  et  $Q_4^i = P_{i+1}$  ;
- $Q_0^i Q_1^i = Q_1^i Q_2^i = Q_2^i Q_3^i = Q_3^i Q_4^i = \frac{1}{3} Q_0^i Q_4^i$  ;
- $Q_2^i$  est l'image de  $Q_3^i$  par la rotation de centre  $Q_1^i$  et d'angle  $+\frac{\pi}{3}$  ;
- $\overrightarrow{Q_0^i Q_1^i} = \frac{1}{3} \overrightarrow{Q_0^i Q_4^i}$  et  $\overrightarrow{Q_0^i Q_3^i} = \frac{2}{3} \overrightarrow{Q_0^i Q_4^i}$ .

FIGURE 5 – 4 étapes de la construction d'un fractale à partir d'un segment initial



**Q.1** Écrire la fonction `void maj(line *l)` qui prend en paramètre un pointeur sur une ligne brisée. La fonction met à jour le champ `sommet` de la ligne pointée par `l` de façon à ce qu'il contienne les nouveaux points décrits plus haut. Il faut donc mettre à jour également le champ `card` et libérer ce qui doit l'être.

A ce stade, aucun arrondi de flottant n'est à réaliser. Au contraire, les coordonnées des points de la ligne brisée doivent être aussi précises que possible.

Par exemple, si `*l` est constituée des points

```
1 pt p0 = {1.,100.,10.};
2 pt p2 = {1.,100.,130.};
3
```

alors, après appel de `maj(&l)`, le tableau des sommets de `l` est constitué des points

```
1 {1.0000, 100.0000, 10.0000},{1.0000, 100.0000, 50.0000},
2 {1.0000,65.3590, 70.0000},{1.0000, 100.0000, 90.0000},
3 {1.0000, 100.0000, 130.0000}
4
```



**Q.2** Écrire la fonction `line * koch(int n, pt A, pt C)` qui prend en paramètres un nombre d'étapes de construction de flocon et deux points (formant le segment initial). La fonction constitue d'abord une ligne brisée représentant un triangle équilatéral  $A, B, C$  tel que  $B$  est l'image de  $C$  par la rotation d'angle  $+\frac{\pi}{3}$  et de centre  $A$ . Elle lui applique ensuite  $n$  fois la fonction de mise à jour.

A ce stade, aucun arrondi de flottant n'est à réaliser. Au contraire, les coordonnées des points de la ligne brisée doivent être aussi précises que possible. Les arrondis de flottants seront produits par l'appel à la fonction `draw_line`.

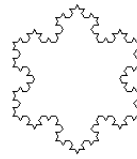
Affichage de l'image **koch3.pbm** qui représente un flocon de Koch construit en 3 étapes.

```

92 pt p0 = {1.,150.,50.};
93 pt p1 = {1.,150.,170.};
94 line * l = koch(3,p0,p1);
95 img * im = uni(250,240,false,1);
96 draw_line(im,l);
97 tofile("koch3.pbm",im);
98 free_img(im);free_line(l);

```

Listing 14 – Tests KOCH



C'est l'appel à la fonction `draw_line` qui réalise les arrondis.