3I022-2018fev
Introduction à l'Imagerie Numérique

Licence Informatique
Sorbonne Université

Dominique.Bereziat@lip6.fr
Arnaud.Dapogny@gmail.com
EmanueldeBezenac@gmail.com

Année universitaire 2017–2018

Énoncé des Travaux Dirigés et Travaux Encadrés sur Machines

Premiers pas en traitement d'images

Travaux Dirigés 1

Exercice 1 — Temps de transmission d'images

En admettant que les données sont transmises sans adjonction de bits de contrôle, combien de temps T_t faudra-t-il pour transmettre les images suivantes (1 Bd = 1 Baud = 1 bit/seconde) ($T_t = \frac{T}{D}$, où T est la taille en bits de l'image, et D le débit en Bd) :

1. Image binaire 256×256, ligne à 300 Bd ?
2. Image 1024×1024 en 256 niveaux de gris, ligne à 2400 Bd ?
3. Image couleur RGB (8 bits/plan) 512×512, ligne à 9600 Bd (réseau GSM) ?
4. Image couleur RGB (8 bits/plan) 512×512, ligne à 56 KBd (modem V.90) ?
5. Image couleur RGB (8 bits/plan) 512×512, ligne à 10 MBd (réseau Ethernet) ?

Exercice 2 — Représentation informatique d'une image

Il y a deux manières de coder une image $N \times M$ (soit N lignes et M colonnes) pour une représentation informatique : une matrice A ou un vecteur V.

1. Pour chacune de ces représentations, dire comment vous accédez (en `C`) :
   (a) au premier pixel de l'image,
   (b) au pixel de coordonnées (x, y) de l'image,
   (c) au dernier pixel de l'image.
2. Sachant que l'on examine le pixel (x, y), quelle est la position du pixel :
   (a) à droite (ou Est),
   (b) à gauche (ou Ouest),
   (c) juste en dessous (ou Sud),
   (d) juste au dessus (ou Nord).

Exercice 3 — Algorithmes

1. Par quelles valeurs sont représentés le blanc et le noir dans les images en niveaux de gris ?
2. Écrire un algorithme de seuillage binaire d'une image de taille $N \times M$. Dans ce seuillage, tous les pixels ayant un niveau de gris inférieur au seuil S sont mis en noir, les autres en blanc.
3. Écrire un algorithme de calcul d'histogramme.
4. Le principe de l'étirement d'histogramme est d'augmenter la dynamique des niveaux de gris d'une image en augmentant l'intervalle de variation de ceux-ci. Écrire un algorithme qui étire un histogramme d'un intervalle $[A, B]$ vers l'intervale $[0, 255]$, avec $A < B$, $A \geq 0$ et $B \leq 255$.

5. Dans une image, on souhaite isoler 10% des pixels les plus sombres (i.e. les mettre en noir), comment procéder ? Écrire un algorithme permettant de faire ceci.

Premiers pas en traitement d'image

Travaux encadrés sur machine 1

Pour ce premier TME, nous aurons besoin de lire et d'écrire des images. Nous choisissons un format très simple, le *Portable Grey Map* (`PGM`), qui permet de stocker des images à niveaux de gris. Une image `PGM` peut être lue avec le code suivant :

```
#include <stdio.h>
#include <stdlib.h>

unsigned char* readpgm( char *nom, int *w, int *h) {
  char line[5];
  unsigned char *buf = NULL;
  FILE *fp = fopen( nom, "r");

  if( fp == NULL) return NULL;
  fgets( line, 4, fp);
  if( *line == 'P' && *(line+1) == '5') {
    int c;
    if( (c=fgetc(fp)) == '#')
      while( (c=fgetc(fp)) != '\n');
    else
      ungetc(c,fp);
    fscanf( fp, "%d␣%d\n%d\n", w, h, &c);
    buf = malloc(sizeof(char)**w**h);
    fread( buf, sizeof(char), *w**h, fp);
  } else
    fprintf( stderr, "readpgm:␣%s␣has␣an␣unsupported␣format\n", nom);
  fclose( fp);
  return buf;
}
```

Créer un répertoire `3I022-2018fev` à la racine de votre compte à l'ARI, puis un répertoire `tme1`. On y déposera tous les fichiers créés pendant ce TME.
Récupérer le fichier `/Infos/lmd/2017/licence/ue/3I022-2018fev/src/readpgm.c` qui contient le code de la fonction `readpgm()`. Il faut aussi récupérer et utiliser le fichier :
`/Infos/lmd/2017/licence/ue/3I022-2018fev/src/pgm.h`
qui contient les signatures des fonctions utiles au chargement (et sauvegarde) des image `PGM`.

Exercice 1 — Histogramme

1. Dans un fichier `histogram.c`, écrire une fonction `C` de prototype

   ```
   void histogramme( int his[256], unsigned char *buf, long size);
   ```

   qui calcule l'histogramme du tampon `buf`. Ce tampon possède `size` octets.

2. Compléter le fichier `histogram.c` avec la fonction `main()` qui :
   — lit le fichier `PGM` dont le nom est passé en paramètre (on utilisera le protocole `argv` [1]).
   — calcule son histogramme,
   — affiche l'histogramme dans la sortie standard sous la forme :

```
0   1001
1   234
2   456
3   0
...
255 123
```

   La première colonne indique un niveau de gris, la seconde le nombre d'occurrences de ce niveau de gris.

   Rappel pour compiler ce programme, on tapera dans le terminal texte :

```
gcc histogram.c readpgm.c -o histogram
```

3. Expérimenter la commande avec les images PGM (elles ont l'extension `.pgm`) disponibles dans le répertoire `/Infos/lmd/2017/licence/ue/3I022-2018fev/images`.
   Pour visualiser graphiquement l'histogramme, on peut rediriger la sortie de votre commande vers la commande `xgraph`.

   Remarque 1 (La commande xgraph) *Cette commande affiche dans une fenêtre X-window un ensemble de points lus dans l'entrée standard. Chaque ligne lue dans l'entrée standard doit comporter deux valeurs : une abscisse et une ordonnée.*
   Par défaut, `xgraph` dessine ces points dans le plan et les relies par un segment de droite dans l'ordre de lecture. Dans cet exemple simple :

```
printf "1␣2\n2␣5" | xgraph
```

   `xgraph` affichera deux points : un premier de coordonnée $(1, 2)$*, un second de coordonnée* $(2, 5)$ *et les relie par un segment de droite.*
   On peut changer le comportement d'affichage de `xgraph` (lire son manuel – `man xgraph`, ou taper `xgraph -help`). Par exemple les options `-nl` et `-bar` permettent d'afficher les points comme des barres verticales. On peut donc utiliser ces deux options pour afficher des histogrammes.

L'écriture d'une image `PGM` se fait très simplement à l'aide de la fonction suivante :

```
#include <stdio.h>
#include <stdlib.h>

int writepgm( char *nom, unsigned char *buf, int w, int h) {
  FILE *fp = fopen( nom, "w");

  if( fp != NULL) {
    fprintf( fp, "P5\n%d␣%d\n255\n", w, h);
    fwrite( buf, sizeof(char), w*h, fp);
    fclose( fp);
```

1. La fonction main est déclarée ainsi : `int main( int argc, char **argv)` où `argc` est le nombre d'arguments plus 1 passé au programme, et `argv[i]` contient le `i`-ième argument, `i` commence à 1.

```
    return 1;
  }
  fprintf(stderr,"writepgm: write error with %s\n", nom);
  return 0;
}
```

Récupérer le fichier `/Infos/lmd/2017/licence/ue/3I022-2018fev/src/writepgm.c` qui contient le code de la fonction `writepgm()`.

Exercice 2 — Etirement d'histogramme.

1. Dans un fichier `etire.c`, écrire la fonction de prototype

   ```
   void etire( int table[256], int his[256]);
   ```

   qui étire l'histogramme décrit par le tableau `his` et place les correspondances entre niveaux de gris avant et après étirement de l'histogramme dans le tableau `table`. Pourquoi est-il plus utile de connaître les correspondances entre niveaux de gris plutôt que l'histogramme étiré ?
2. Ajouter dans `etire.c` la fonction de prototype

   ```
   void applique( unsigned char *buf, long size, int table[256]);
   ```

   qui modifie le tampon `buf` de longueur `size` en appliquant la table `table` de conversion des niveaux de gris donné par la fonction `etire()`.
3. Créer la commande `etire` qui lit une image `PGM` donnée en premier paramètre, étire son histogramme, applique le nouvel histogramme et écrit l'image résultante sous le nom donné en second paramètre.
4. Expérimenter la commande sur les images données dans le répertoire précédent. On peut visualiser une image `PGM` avec les commandes `eog` ou `display` par exemple.
   Comparer visuellement les images avant et après étirement de l'histogramme. Comparer également leur histogramme.

Exercice 3 — Seuillage d'image

1. Écrire une fonction de prototype

   ```
   void seuillage( unsigned char *buf, long size, int bas, int haut);
   ```

   qui réalise un seuillage : tous les pixels dont le niveau de gris est inférieur ou égal à `bas` sont mis à zéro et tous les pixels dont le niveau de gris est supérieur ou égal à `haut` sont mis à 255.
2. Mettre en œuvre cette fonction dans un programme `C` qui lit une image `PGM` donnée comme premier argument, effectue le seuillage et écrit le résultat dans une image `PGM` donnée en second argument. Les paramètres du seuillage sont donnés par les troisième et quatrième arguments. Indication : on peut utiliser la fonction `atoi()`, de la librairie standard, qui convertit une chaîne de caractères en un entier.
3. Tester cette commande sur les images données dans le répertoire précédent en faisant varier les seuils.

Exercice 4 — Chaînage de commandes (facultatif)

1. En C, ils existent deux variables `stdin`, et `stdout`, de type `FILE*`, qui correspondent respectivement à l'entrée standard et la sortie standard. À l'aide de ses variables, modifier les fonctions `writepgm()` et `readpgm()` de façon à ce qu'elles écrivent dans l'entrée standard ou lisent dans la sortie standard lorsque le nom de fichier `'-'` est précisé.
2. Tester le chaînage (tube) des commandes précédemment écrites.

Introduction au logiciel `Inrimage` – la librairie de développement

Travaux dirigés 2 et 3

1 Codage des images et gestion des entrées/sorties

1.1 Codage des images en mémoire

Une image est représentée par :

- un tampon en mémoire, c'est-à-dire une zone contiguë dans la mémoire, dont chaque élément représente la valeur d'un pixel. La convention `Inrimage` est de ranger les pixels de l'image ligne par ligne, la première ligne étant la plus haute sur l'écran. Chaque ligne est organisée en pixels, les premiers étant les plus à gauche sur l'écran.
- un tableau de 9 éléments, appelé *tableau de format*. Ce tableau contient les dimensions de l'image et des informations sur son codage. Ces informations sont utiles aux fonctions de lecture/écriture d'`Inrimage`.

En `C`, nous déclarons :

```
type *buffer;
Fort_int lfmt[9];
```

Exercice 1 — Codages courants

1. En `C`, avec quel type `type` représenteriez-vous en mémoire des images :
   a) en virgule fixe ?
   b) en virgule flottante ?

   En virgule fixe les codages couramment utilisés sont de 1, 2 ou 4 octets et en virgule flottante les codages utilisés sont 4 octets (simple précision) ou 8 octets (double précision).
2. Comment représenter en mémoire un codage sur n bits ? Le codage 1 bit est courant pour gérer des images de masque pour des fichiers de contours. Comment accéder à un bit en particulier ?

Le tableau `lfmt` contient les 9 éléments suivants :

Indice	Nom symbolique	Signification
0	`I_DIMX`	Nombre total de valeurs par ligne
1	`I_DIMY`	Nombre total de lignes dans l'image
2	`I_BSIZE`	Taille du codage du type Si positif indique un nombre d'octets Si négatif indique un nombre de bits
3	`I_TYPE`	Type de codage : `FIXE` pour un codage en virgule fixe `PACKEE` pour un codage en virgule fixe compact `REELLE` pour un codage en virgule flottante
4	`I_NDIMX`	Nombre de pixels par ligne
5	`I_NDIMY`	Nombre de lignes dans un plan
6	`I_NDIMV`	Nombre de composantes du pixel
7	`I_NDIMZ`	Nombre de plans dans l'image
8	`I_EXP`	Type de flottant ou exposant de l'image en virgule fixe

Remarque : le type `Fort_int` représente un "entier Fortran", héritage de la partie Fortran de la librairie. L'usage de l'élément `I_EXP` est complexe et dépasse largement le cadre de ce cours. Sachez simplement que :

- lorsque le codage est en virgule fixe, `lfmt[I_EXP]` doit valoir 200 si l'image n'est pas signée (ce sera toujours notre cas).
- lorsque le codage est en virgule flottante, la valeur dans `lfmt[I_EXP]` n'est pas signifiante. On peut mettre 0.

Exercice 2 — Tableau de format

1. Écrire les liens qui unissent `lfmt[I_DIMY]` à `lfmt[I_NDIMY]`, et `lfmt[I_DIMX]` à `lfmt[I_NDIMX]`.
2. Pour chaque codage suivant on donnera les valeurs des éléments `I_BIZE` et `I_TYPE` du tableau de format :
   - 1 octet en virgule fixe,
   - 2 octets en virgule fixe,
   - 4 octets en virgule flottante,
   - 4 bits en virgule fixe.

1.2 Lecture et écriture des images

Elles sont très similaires aux fonctions de lecture et d'écriture d'`Unix` avec une différence : les accès aux images se font ligne par ligne. Par ligne, nous entendons ligne d'image, soit un ensemble de `lfmt[I_DIMX]` valeurs. Comme les accès aux fichiers sont séquentiels, on commence par lire ou écrire la première ligne. Puis le pointeur de fichier se positionne au début de la ligne suivante et ainsi de suite.

Ouverture d'une image

```
struct image *image_( char *nom, char *mode, char *verif, Fort_int lfmt[]) ;
  nom   : nom de l'image
  mode  : mode d'ouverture :
    - "e" : l'image doit exister
    - "c" : l'image est créée
  verif : mode de vérification (à ignorer)
```

```
  lfmt : si l'image existe, lfmt contiendra le format de l'image
         si l'image est créée, lfmt doit contenir son format.
retour : un descripteur du fichier.
```

Remarque : `image_()` ouvre une image en lecture et en écriture (avec les droits d'accès `Unix` adéquats sur le fichier). Le paramètre `mode` est important : avec la valeur `"c"` l'image, si elle existe, sera écrasée.

Fermeture d'une image

```
void fermnf_( struct image **nf);
  nf : pointeur de fichier (ouvert par image_())
```

Lecture d'une ou plusieurs lignes d'une image

```
void c_lect( struct image *nf, int nligne, void* buf);
  nf     : pointeur d'image,
  nligne : nombre de lignes à lire,
  buf    : tampon où sont écrites les données.
```

Écriture d'une ou plusieurs lignes d'une image

```
void c_ecr( struct image *nf, int nligne, void* buf);
  nf     : pointeur d'image,
  nligne : nombre de lignes à écrire,
  buf    : tampon où sont lues les données.
```

Positionnement du pointeur de ligne dans une image

```
void c_lptset( struct image *nf, int nligne);
  nf     : pointeur d'image,
  nligne : numéro de la ligne à pointer. Par convention Inrimage, les
           numéros de ligne commencent à 1.
```

Cette fonction est l'equivalent de `fseek()` pour les images.

Exercice 3 — Lecture d'image en virgule fixe

On suppose que l'on travaille sur des images scalaires codées en virgule fixe sur un octet. On ne connaît pas explicitement la taille de ces images.

1. Écrire une fonction qui lit le premier plan d'une image dont le nom est passé en paramètre, puis le transfère dans un tampon. Le tampon doit avoir la bonne taille.
2. Écrire une fonction qui lit tout les plans d'une image.
3. Écrire une fonction qui lit un plan de numéro z passé en paramètre.
4. Cette fonction peut-elle lire des codages sur n bits ? Lesquels ?

Exercice 4 — Lecture d'image en virgule flottante

Comment modifier le code de la première fonction de l'exercice précédent de façon à lire des images codées en virgule flottante et simple précision ?

Exercice 5 — Ecriture d'image

Écrire une fonction qui :
- lit le premier plan d'une image codée en virgule fixe sur 1 octet,
- transfère le tampon de ce plan dans un tampon de type flottant simple précision,
- écrit le tampon flottant dans une nouvelle image et dans un format adéquat.

2 Accès aux pixels

Nous savons que les pixels dans une image sont **conventionnellement** ordonnés de la façon suivante : `DIMY` lignes positionnées de manière contiguë en mémoire, chaque ligne se composant de `DIMX` valeurs. Une image se compose de :
- `NDIMZ` plans positionnés de manière contiguë en mémoire. Chaque plan se compose de :
  - `NDIMY` lignes positionnées de manière contiguë en mémoire. Chaque ligne de compose de :
    - `NDIMX` pixels positionnés de manière contiguë en mémoire. Chaque pixel se compose de :
      - `NDIMV` composantes.

Exercice 6 — Tampon monoplan

Soit une image codée sur 1 octet dont on a chargé le premier plan. On suppose que chaque pixel est scalaire (donc que `NDIMV=1`).
- En `C`, comment accéder à la $i^{\text{ième}}$ ligne et à la $j^{\text{ième}}$ colonne ?
- Écrire le code `C` qui transpose ce tampon dans un autre tampon.

Exercice 7 — Tampon multiplan

Soit une image codée sur 1 octet dont on a chargé tous les plans dans un unique tampon. On suppose que chaque pixel est scalaire (donc que `NDIMV=1`).
- En `C`, comment accéder à la $i^{\text{ième}}$ ligne et à la $j^{\text{ième}}$ colonne, situées dans le $k^{\text{ième}}$ plan ?
- Écrire le code `C` qui inverse l'ordre des plans dans le même tampon : le plan `k` est échangé avec le plan `NDIMZ-k+1`.

3 Les images couleurs et vectorielles

`Inrimage` permet de travailler avec les images couleurs. Il existe deux façons de représenter des images couleurs :
- Chaque pixel code une couleur. Cette valeur est un indice dans une table des couleurs associée à cette image. Pour information, la table des couleurs se trouve dans l'entête de l'image. Ces images s'appellent des images indexées. La taille de codage du pixel est généralement de 1 octet (ce qui autorise 256 couleurs).

— Chaque pixel représente directement la couleur c'est-à-dire un triplet RVB. Ainsi `Inrimage` peut représenter des images couleurs en utilisant 3 composantes par pixel. Pour ces images, `NDIMV = 3`.

Exercice 8

Les images `JPEG` sont codées sur 3 octets. Ces trois octets représentent respectivement la composante rouge, la composante verte et la composante bleue. Combien de couleurs peut-on représenter avec le format `JPEG` ?

Exercice 9

Soit un tampon mémoire contenant une image au format `RVB`, chaque composante étant codée sur 1 octet dans l'ordre Rouge, Vert, Bleu.

— Écrire la formule qui permet d'accéder à la composante $v = 0, 1, 2$ du pixel (i, j).

— Soit une fonction qui extrait la composante rouge de ce tampon et la copie dans un nouveau tampon. Quel est le format et la taille du tampon de sortie ? Écrire cette fonction.

Exercice 10 — Table de couleurs

Pour coder les couleurs, on peut aussi utiliser une table de couleur : un pixel de l'image vaut un indice dans la table des couleurs et chaque élément de la table contient un triplet rouge/vert/bleu décrivant la couleur.

1. Soit une image dont les pixels, codés sur 1 octet, décrivent un indice dans une table des couleurs. Quelle est la taille maximal de cette table ?
2. Quels sont les intérêts à utiliser un tel format ?
3. Écrire une fonction de prototype

```
void indexee_rvb( unsigned char *rvb, unsigned char *index,
                  unsigned char *tcoul, long size);
```

qui convertit une image indexée index de taille size en une image `RGB` rvb en utilisant la table des couleurs tcoul.

4 Lecture et écriture des images par conversion en virgule flottante

Les deux fonctions suivantes permettent de lire ou d'écrire une image sans faire d'hypothèse sur leur codage :

```
void c_lecflt( struct image *nf, int nligne, float* buf);
  nf     : pointeur d'image,
  nligne : nombre de lignes à lire,
  buf    : tampon où sont  écrites les données.

void c_ecrflt( struct image *nf, int nligne, float* buf);
  nf     : pointeur d'image,
  nligne : nombre de lignes à écrire,
  buf    : tampon où sont lues les données.
```

Dans ce mode d'accès aux images, les données lues sont converties en virgule flottante et les données écrites sont converties au codage de l'image à écrire. Il y a alors trois cas de figure selon le codage de l'image :

i) si le codage est à virgule flottante : il n'y a pas de conversion et la lecture et l'écriture sont directes.

ii) si le codage est à virgule fixe et non signé (i.e. lfmt[I_EXP]=200) :
   — en lecture, chaque valeur lue est comprise entre 0 et (taille du codage - 1) : la fonction c_lect() divise chaque valeur lue par (taille du codage - 1) pour se ramener à une valeur entre 0 et 1.
   — en écriture, chaque valeur sera multipliée par la taille de codage -1.

iii) nous ne traiterons pas le cas des images à virgule fixe et signées dans ce cours.

Donc, dans ce mode d'accès, il n'est pas absolument utile de faire des hypothèses sur le codage des images pour les charger en mémoire. De même il n'est pas utile de faire des choix spécifiques sur le type du tampon pour écrire des images dans un format quelconque.

Exercice 11 — Lecture

1. Soit la fonction suivante :

```
float *lit_image( char *nom, Fort_int *lfmt) {
  struct image *nf;
  float *buf;
  nf = image_(nom, "e", "", lfmt);
  buf = i_malloc(sizeof(float)*DIMX*DIMY);
  c_lecflt( nf, DIMY, buf);
  fermnf_(&nf);
  return buf;
}
```

   Considérons l'image 3×3 codée sur 1 octet :

$$\begin{pmatrix} 0 & 128 & 255 \\ 32 & 64 & 96 \\ 160 & 192 & 224 \end{pmatrix}$$

   Que contient le tampon buf après lecture de l'image par la fonction lit_image() ?

2. Même question pour une même image mais codée cette fois sur 2 octets.
3. Dans le cas d'un codage de type FIXE comment retrouver les "bonnes valeurs" avec une lecture flottante ?
4. Soit une image de masque codée sur 1 bit en virgule fixe. Après appel de la fonction lit_image(), quelles valeurs contient le tampon buf ?
5. Même question si l'image est codée sur 1 bit en mode PACKEE.

Remarque : il existe une fonction `Inrimage` qui permet de faire ces conversions (il s'agit de c_cnvtbg()) car la situation est plus complexe que celle décrite ici, les images pouvant également être signées et avoir un exposant non nul. Mais nous ne rencontrerons jamais de telles images dans cette UE.

Exercice 12 — Ecriture

Soit la fonction suivante :

```
#include <stdlib.h>
void randimage(float *buf, long size) {
  while(size--)
      *buff++=drand48();
}
```

1. Que fait cette fonction ?
2. Écrire un programme qui initialise une image 256×256 avec la fonction randimage() puis écrit sur cette image sur le disque avec un format à virgule fixe sur 1 octet. Quelle sera la dynamique de l'image écrite ?
3. Modifier le programme pour écrire maintenant une image avec un format à virgule flottante.

Introduction au logiciel `Inrimage`

Travaux sur machines encadrés 2 et 3

1 Les commandes `Inrimage` (séance 2)

Les sous-sections 1.2 à 1.4 font quelques rappels du cours (et aussi de l'interprète de commandes) qui sont utiles pour répondre aux questions du TME. Ces dernières commencent formellement à partir de la sous-section 1.5

1.1 Préparation du TME

Le logiciel `Inrimage` est disponible à http://inrimage.gforge.inria.fr, prêt à l'emploi pour `Linux` (32 bits et 64 bits) et `Cygwin`. Les sources sont également disponibles si l'on souhaite l'installer sur une autre architecture. À l'ARI, `Inrimage` est installé dans la hiérarchie `/usr/local`.
Ouvrir un interpréteur de commande :

- taper la commande `inrinfo` pour vérifier que l'installation d'`Inrimage` est correcte :

```
% inrinfo
Inrimage Version 4.6.3
INR_HOME /usr/local/share/inrimage
INR_EXTFILE /usr/local/share/inrimage/etc/inr_extfile
```

- dans votre répertoire `3I022-2018fev` à la racine de votre compte, créez un répertoire `tme2-3` dans lequel vous placerez les fichiers créés pendant ce TME.
- On trouvera des images dans `/Infos/lmd/2017/licence/ue/3I022-2018fev/images`. Les images au format Inrimage ont l'extension `.inr` ou `.inr.gz`. Les autres images (`.jpg`, `.pgm`, `.tif`) sont également lisibles par Inrimage. Vous pouvez les copier ou bien faire des liens symboliques (commande `ln -s`) ce qui permet d'économiser de la place disque.

1.2 Usage des commandes `Inrimage`

Une commande `Inrimage` est un programme que l'on lance depuis l'interpréteur de commandes et qui agit sur des images. La syntaxe générale est la suivante :

```
commande img_entree [img_entree ...]  img_sortie [img_sortie ...] options
```

On spécifie toujours les images de sortie après les images d'entrée. Certaines commandes nécessitent plusieurs images en entrée et/ou plusieurs images en sortie. Lorsque les entrées (ou les sorties) ne sont pas spécifiées, `Inrimage` lit dans l'entrée standard (ou dans la sortie standard). Lorsque cela est ambiguë, on peut utiliser le nom '-' qui désigne l'entrée ou la sortie standard.
Les options sont de la forme :

```
-nom_option [valeur [valeur]]
```

Les options sont généralement facultatives car elles possèdent des valeurs par défaut. Leur ordre d'apparition n'est pas signifiant. Les options possèdent 0 ou 1 ou 2 valeurs. Ces valeurs peuvent être de type numérique

ou des noms de fichiers. Les options paramètrent le comportement de la commande. Il existe des options qui sont communes à toutes les commandes. En particulier, l'option `'-help'` donne toujours de l'information sur l'usage d'une commande `Inrimage`. De plus, les commandes `Inrimage` sont documentées dans les manuels `Unix` : tapez `man commande` dans votre shell pour avoir de l'information exhaustive sur la commande.

```
commande -help
man commande
```

Attention : les noms de fichier ne doivent pas commencer par un chiffre (sinon, `Inrimage` considère que ce sont des arguments numériques). Toutefois, on peut contourner cette difficulté en préfixant les noms de fichier par "./"

1.3 Chaînage des commandes

Comme on l'a dit, les commandes `Inrimage` peuvent lire dans l'entrée standard et écrire dans la sortie standard. Les commandes `Inrimage` sont très basiques et pour faire des opérations plus complexes, il faut enchaîner plusieurs commandes. Par exemple :

```
commande1 image.inr inter2
commande2 inter2 inter3
commande3 inter3 resultat.inr
```

peut être remplacé par :

```
commande1 image.inr | commande2 | commande3 - resultat.inr
```

Le caractère `|` s'appelle un tube car il connecte la sortie standard d'une commande vers l'entrée standard d'une autre commande. Un tel mécanisme possède plusieurs avantages :

- il permet d'éviter l'écriture d'images intermédiaires,
- il est très rapide car les données restent et transitent en mémoire.

On remarquera l'utilisation du caractère `-` dans la commande `commande3`. Sans ce paramètre (qui signifie entrée standard ou sortie standard selon le contexte), `Inrimage` considérerait le paramètre `resultat.inr` comme image d'entrée ce qui est incorrect.

1.4 Informations supplémentaires

`Inrimage` est très riche en fonctionnalités. Pour approfondir le sujet, la documentation au format HTML est disponible à l'addresse suivante : http://inrimage.gforge.inria.fr/WWW/index.html
La documentation est aussi installée en local à l'ARI dans `/usr/local/share/doc/inrimage/index.html`. On peut aussi obtenir un bon point de départ depuis l'interpréteur de commandes avec :

```
# introduction aux commandes
man Inrimage
# liste thematique des commandes
man LISTE
# introduction a la programmation
man Intro
# liste thematique des routines
man 3 LISTE
```

Enfin, le cours sur Inrimage synthétise toutes ces informations.

1.5 Quelques commandes `Inrimage`

`par,cpar`	Indique le codage et les dimensions d'une ou plusieurs images
`xvis`	Affiche dans une fenêtre X l'image
`gvis`	Alternative à `xvis` lorsque la carte graphique n'est pas supportée par `xvis`
`tpr`	Imprime les valeurs des pixels d'une image
`ical`	Affiche les valeurs minimale, moyenne et maximale d'une image
`norma`	Normalise une image

Exercice 1

1. Lire le manuel `Unix` de chacune de ces commandes. Les essayer sur le jeu d'images fourni en TME. Pour chaque image, indiquez leurs dimensions et leur codage.
2. Comparer visuellement une image et sa version normalisée ; pour visualiser une image normalisée, utiliser la commande `norma` et la rediriger sur la commande `xvis`.

1.6 Création d'images

`raz`	Initialise une image
`cim`	Définit les valeurs d'une image
`create`	Créée un entête d'image
`inrcat`	Extrait le corps d'une image

`xvis` possède un mode de dessin dans les images très rudimentaires : on peut utiliser `Gimp` et exporter dans un format lisible par `Inrimage`.

Exercice 2

1. Quelle commande doit-on appliquer pour générer une image noire 128×128 ?
2. À l'aide de la commande `cim`, créer une image 3×3 dont les éléments de la diagonale sont blancs et les autres sont noirs (utiliser l'option `-rd` de `cim`, ou encore la commande `echo` en redirection sur `cim` sans l'option `-rd`). Vérifier que les valeurs de l'image sont correctes à l'aide de la commande `tpr`.
3. Avec la fonction `izoom` (lire son manuel), fabriquer une image de taille 60×60. Ajouter l'option `-f` à la commande de zoom et comparer les résultats.
4. Comment paramétrer l'appel à `xvis` pour obtenir le même résultat (on lira dans le manuel de la commande le paramétrage de l'option `-p`) ?

1.7 Manipulation des codages et des dimensions

`cco`	Converti le codage des pixels d'une image
`cnvcol`	Conversion d'image couleur en image niveau de gris (et inversement)
`extg`	Extrait une zone d'une image
`melg`	Mélange deux images

Exercice 3

1. À l'aide de la commande `extg` extraire les canaux rouge, vert et bleu de l'image `lena.jpg`. On rappelle que, par convention, dans une image à trois composantes, les canaux rouge, vert et bleu sont

représentés respectivement par les composantes 1, 2 et 3 d'un pixel. Affichez ces composantes avec le programme `xvis`.

2. À l'aide de la commandes `melg` construire une nouvelle image dans laquelle les composantes rouge et verte de `lena.jpg` ont été échangées (lire le manuel de `melg`).
3. Essayer de répondre à la question précédente en utilisant les commandes `inrcat` et `create` au lieu de `melg` et en essayant de construire une image à trois plans : le premier plan représente les composantes rouges, le second les composantes vertes et le troisième les composantes bleues. La commande `xvis` est capable d'afficher ce type d'image.

Exercice 4

Beaucoup de dispositifs d'acquisition fournissent des images codées sur 2 octets ou davantage.

1. Appliquer la commande `par` sur l'image `CT-scan.inr.gz`. Quel est son codage ?
2. Après avoir lu le manuel de la commande `ical`, l'appliquer sur l'image précédente. Pourquoi la valeurs maximale est plus petite que 1 ? Indice : `ical` ramène les valeurs des pixels des images à virgule fixe entre 0 et 1 en divisant par la taille du codage (soit $2^{8\times BSIZE} - 1$).
3. Visualiser l'image avec `xvis`. Qu'observez-vous ? Réessayer en appliquant la commande `norma` en tube sur `xvis`. Qu'observez-vous ?
4. Utiliser la commande `cco` pour convertir l'image en un codage 1 octet par pixel (lire son manuel puis `man Inrimage`, paragraphe OPTIONS DE FORMAT). Examiner l'image produite avec `xvis`. Commenter.
5. En amont de `cco`, quelle commande doit-on utiliser pour convertir correctement notre image en un format de 1 octet par pixel ?

1.8 Opérations entre les images

`ad`	Ajoute deux images
`so`	Soustrait deux images
`sc`	Multiplie une image par un scalaire
`bi`	Ajoute un biais à une image
`min`	Calcule le minimum de deux images
`max`	Calcule le maximum de deux images
`logic`	Réalise une opération logique entre deux images

Remarque : la plupart des manuels des commandes réalisant des opérations arithmétiques entre les images sont regroupées dans les pages `arit1` et `arit2`. Dans bash, essayez :

```
man arit1
man arit2
```

Exercice 5

1. Comment afficher les différences entre deux images ?
2. Comment tester si deux images sont égales ?

Exercice 6

1. En utilisant les images obtenues avec l'exercice 3, calculer la moyenne des trois composantes dans une nouvelle image qui l'on affichera. Indication : utiliser les commandes `ad` et `sc`.
2. Calculer la moyenne pondérée des trois composantes avec les pondérations suivantes : 0.299 pour la composante rouge, 0.587 pour la composante verte et 0.114 pour la composante bleue.
3. Comparer avec le résultat produit par la commande `cnvcol -bw`.

2 Écriture de nouvelles commandes (séance 3)

La bibliothèque de développement d'`Inrimage` fournit les fonctions nécessaires pour la création de nouvelles commandes vérifiant les spécificités imposées par `Inrimage`.
Un programme `Inrimage` comporte généralement 5 étapes :

1. initialisation ;
2. lecture des options ;
3. lecture des images ;
4. traitement ;
5. écriture des images résultats.

L'exemple qui suit est un exemple typique de programmation `Inrimage`.

```
/*
 * LI316 - Introduction a l'imagerie numerique
 * Fichier template Inrimage
 */

#include <inrimage/image.h>

static char version[]="1.00";
static char usage[]="[input][output][-sb_seuil]";
static char detail[]="";

int main( int argc, char **argv) {
  char fname[256];
  struct image *nf;
  Fort_int lfmt[9];
  unsigned char *buf;
  int sb = 0, i, j;

  /* Initialisation */
  inr_init( argc, argv, version, usage, detail);

  /* Lecture des options */
  igetopt1( "-sb", "%d", &sb);
  infileopt( fname);

  /* Ouverture et lecture des images */
  nf = image_(fname, "e", "", lfmt);

  /* verification du format */
```

```
  if( !(TYPE == FIXE && BSIZE==1))
    imerror( 6, "codage non conforme\n");

  /* allocation memoire adequat */
  buf = (unsigned char*)i_malloc( NDIMX * NDIMY*sizeof(unsigned char));

  /* lecture image */
  c_lect( nf, NDIMY, buf);

  /* fermeture image */
  fermnf_( &nf);

  /* Traitement */
  for( j=0; j<NDIMY; j++)
    for( i=0; i<NDIMX; i++)
      if( buf[i+j*NDIMX] < sb)
        buf[i+j*NDIMX] = 0;

  /* Ecriture de l'image */
  outfileopt( fname);
  nf = image_(fname, "c", "", lfmt);
  c_ecr( nf, NDIMY, buf);
  fermnf_( &nf);

  i_Free( (void*)&buf);
  return 0;
}
```

Pour le compiler, on peut utiliser le `Makefile` générique suivant :

```
LDLIBS  = -linrimage -lm

all:
        @echo 'Compilation de module individuel INRIMAGE'
        @echo 'Usage : make module'
```

Ce fichier doit s'appeler `Makefile` et doit être placé à dans le répertoire de vos fichiers sources. Pour compiler le programme `toto.c`, il faut taper la commande `make toto` dans votre interpréteur de commandes.

Exercice 7

1. Récupérer ce programme (dans `/Infos/lmd/2017/licence/ue/3I022-2018fev/src/inrimage.c`) et le compiler.
2. Sur quel type d'image peut-il s'appliquer ?
3. Modifier le code pour exclure les formats non traitable.
4. Modifier le code en remplaçant les trois occurrences du tableau `lfmt` par `gfmt`. Recompilez le programme que constatez-vous ?
5. Rétablir les occurrences `lfmt`. Que fait ce programme ? Compléter la variable `details`.
6. Modifier ce programme pour qu'il puisse faire un seuillage vers le haut (on utilisera une option supplémentaire dans la ligne de commande, lire `man igetopt`).

Exercice 8

1. Réécrire les commandes du TME1 (`histogramme`, `seuillage` et `etire`) en remplaçant les lectures/écritures d'image `PGM` par des lectures/écritures `Inrimage`.
2. Ces nouvelles commandes peuvent-elles traiter les images `PGM` ?

Premiers filtres

Travaux dirigés 4

Exercice 1 — Modifications d'histogramme

On considère l'image de taille 8×8, numérisée selon 8 niveaux de gris (de 0 à 7) suivante :

7	7	7	6	6	6	7	6
6	2	5	5	3	5	7	6
5	2	3	4	2	0	6	5
5	3	6	3	1	5	6	6
6	2	6	2	0	1	2	4
6	5	7	6	0	2	5	6
4	5	6	6	2	4	5	6
6	7	7	3	6	7	6	7

1. Calculer l'histogramme et la fonction de répartition de cette image.
2. Réaliser une égalisation d'histogramme de l'image : on fournira l'histogramme, la fonction de répartition ainsi que les valeurs des pixels de l'image générée.
3. Si on suppose que l'image a été numérisée selon 16 niveaux de gris, réaliser un étirement d'histogramme : on fournira l'histogramme, la fonction de répartition ainsi que les valeurs des pixels de l'image égalisée.

Exercice 2 — Filtrage d'image

On considère l'image de taille 8×8, numérisée selon 8 niveaux de gris (de 0 à 7) suivante :

0	0	0	0	0	0	0	0
0	5	5	5	5	5	5	0
0	5	7	7	7	7	5	0
0	5	7	7	7	7	5	0
0	5	7	7	7	7	5	0
0	5	7	7	7	7	5	0
0	5	5	5	5	5	5	0
0	0	0	0	0	0	0	0

1. Dessiner le profil des niveaux de gris de la ligne 3 de cette image (rappel : les lignes sont numérotées à partir de 0)
2. Réaliser le filtrage par convolution de cette image en utilisant le filtre $\frac{1}{9} \times \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$, en précisant les valeurs des niveaux de gris de l'image filtrée. Dessiner le profil des niveaux de gris de la ligne 3 de cette image, que remarque-t-on ?

3. Mêmes questions que précédemment en utilisant maintenant le filtre

0	-1	0
-1	4	-1
0	-1	0

4. On souhaite réaliser un filtrage en utilisant un filtre gaussien d'écart type $\sigma = 0.5$. Calculer les coefficients de ce filtre. Tracer le profil de niveau de gris selon la ligne centrale du filtre. Mêmes questions avec $\sigma = 1$.

Premiers filtres

Travaux Sur Machines 4

— Déposez les codes dans le répertoire `3I022-2018fev/tme-4`.
— Utilisez les images disponibles dans `/Infos/lmd/2017/licence/ue/3I022-2018fev/images`.

Utilisation des commandes `Inrimage`

Exercice 1 — Réduction de bruit par filtrage linéaire : le filtre moyenneur

1. Utiliser la commande `Inrimage fmoy`, qui effectue la convolution d'une image par un filtre moyenneur, et observer le résultat de la convolution de l'image `camgaussien.inr`. Vous testerez des filtres de taille 3×3, 5×5 et 9×9. Que remarquez-vous pour chacun des filtres testés ?
2. Mêmes questions avec l'image `camimpuls.inr`.
3. Mêmes questions avec l'image `rectangle.inr`. Tracer le profil de niveaux de gris de la ligne du milieu de cette image avant et après filtrage en utilisant les commandes `extg` pour extraire une ligne, `tpr` pour imprimer les valeurs des pixels. Pour produire une sortie graphique du profil, on peut soit :
   — re-diriger la sortie de `tpr -l 1 -c` [1] vers la commande `nl | xgraph` [2].
   — utiliser le programme GNUPLOT. La sortie de `tpr -c -l 1` doit être re-dirigée dans un fichier, nommons-le `toto`. On lance la commande `gnuplot` dans le terminal texte. Dans l'invite de GNUPLOT taper :
   — `plot 'toto' with lines` pour un affichage similaire à `xgraph`, ou bien
   — `plot 'toto'` pour n'afficher que des points.
   Puis taper `quit` dans l'invite de GNUPLOT pour terminer le programme.

Exercice 2 — Réduction de bruit par filtrage non linéaire : le filtre médian

1. Utiliser la commande `Inrimage med` qui applique un filtre médian sur une image. Observer les résultats obtenus sur l'image `camimpuls.inr` pour des fenêtres de taille différentes (par exemple 3×3, 9×9 puis 21×21). Que remarque-t-on ?
2. Comparer les résultats obtenus avec ceux du filtre moyenneur, obtenus dans la dernière question de l'exercice précédent. Que conclure ?

Programmation `Inrimage`

Exercice 3 — Génération d'un filtre moyenneur

1. Écrire un programme `filtreMoy` qui crée un filtre moyenneur de taille $d \times d$, et dont chaque coefficient a pour valeur $\frac{1}{d^2}$, fonctionnant de la sorte : `filtreMoy -d` *nn* `filtre.inr`

1. Lire le manuel de tpr pour l'explication de ces options.
2. Rappel : la commande `xgraph` demande un couple de valeurs (abscisse, ordonnée) pour chaque ligne lue, la commande `nl` numérote les lignes. Ainsi les données sont correctement formatées pour `xgraph`.

où `-d` *nn* désigne la taille du filtre et `filtre.inr` est l'image dans laquelle le filtre est sauvegardé.

2. Tester le filtre moyenneur créé par la commande `filtreMoy` en utilisant la commande Inrimage `conv`. Comparer les résultats obtenus avec ceux obtenus avec la commande `fmoy` (exercice 1).

Exercice 4 — Filtrage médian

Dans cet exercice, on souhaite implémenter un filtre médian de taille 3×3. On rappelle que le filtre médian remplace chaque pixel par la valeur médiane dans son voisinage direct (8 voisins + lui-même). Le principe consiste donc, pour chaque pixel $I(x, y)$ de l'image, de :

1. déterminer la valeur médiane I_m dans son voisinage : pour cela, on utilisera la fonction `C` qsort() [3] de la bibliothèque standard pour trier les 9 valeurs par ordre croissant, la valeur médiane sera alors la cinquième valeur après triage ;
2. remplacer la valeur courante du pixel $I(x, y)$ par cette valeur médiane.

On testera ce programme sur quelques images au choix.

3. lire son manuel : `man qsort`

Filtrage fréquentiel

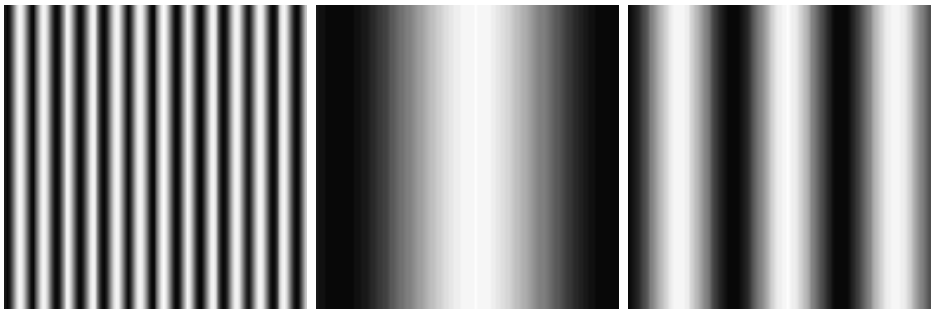
Travaux dirigés 5

Exercice 1 — Rappels sur la transformée discrète de Fourier

1. Rappeler la définition de la transformée de Fourier discrète 1D et ce qu'elle permet de représenter dans les signaux.
2. Même question pour la transformée 2D.

Exercice 2 — Transformée de Fourier sur des images simples

Pour les images `sinus2.inr`, `sinus1.inr` et `sinus3.inr`, décrire précisément l'image de module que l'on obtient après application d'une transformée discrète de Fourier :



Exercice 3 — Filtres fréquentiels

1. Expliquez quelles sont les caractéristiques de filtrages effectués à l'aide :
   — d'un filtre passe-bas
   — d'un filtre pase-haut
   — d'un filtre passe-bande
2. Donnez des exemples de réponses fréquentielles pour chacun de ces filtres et dessinez leur profil de niveaux de gris le long d'une ligne judicieusement choisie.
3. Pourquoi dit-on (c.f. le cours) que le filtre moyenneur est un filtre passe-bas et qu'un détecteur de contours est un filtre passe-haut ?

Exercice 4 — Détection de contours

Écrire un algorithme qui génére les contours horizontaux et verticaux, le module du gradient ainsi que sa phase, en utilisant un filtre de Sobel. On rappelle que les filtres horizontaux et verticaux contiennent respectivement les valeurs :

-1	0	1
-2	0	2
-1	0	1

et

-1	-2	-1
0	0	0
1	2	1

.

La transformée de Fourier discrète (DFT)

Travaux encadrés sur machines 5

— Déposer les codes dans le répertoire `3I022-2018fev/tme-5`.
— Utiliser les images disponibles dans `/Infos/lmd/2017/licence/ue/3I022-2018fev/images`.

Exercice 1 — DFT d'une sinusoïde 2D

Cet exercice est le prolongement de l'exercice 2 du TD 5. On demande de visualiser le résultat de la transformée de Fourier discrète (rapide) des images `sinus1.inr`, `sinus2.inr` et `sinus3.inr`. Les questions 1 et 2 concernent d'abord l'image `sinus1.inr`.

1. Calcul de la DFT d'une image sous la forme module/phase.
   a) Visualiser la partie réelle et la partie imaginaire de la DFT non centrées en utilisant la commande `rdf`.
   b) Visualiser le module et la phase de la DFT non centré en utilisant la commande `ri`. L'image de module peut être mieux visualisée en utilisant la séquence :

      ```
      norma module.inr | xvis -nu
      ```

   c) Afin de mieux étudier les résultats, on centrera le module et la phase en utilisant la commande `ce`.
   d) Pour répéter facilement la séquence `rdf`/`ri`/`ce` pour les exercices suivants, on écrira le script `module.sh` qui prend en argument une image, calcule le module de sa transformée de Fourier dont le nom est également donnée en argument.
2. On remarquera que les coefficients non nuls de l'image du module sont alignés sur une droite. Visualiser l'amplitude (i.e. le module) de ces coefficients, en extrayant la colonne sur laquelle ils apparaissent puis en la traçant. Pour cela, on utilisera la séquence de commandes :

   ```
   tpr -c -l 1 ... | nl | xgraph
   ```

   Tout comme `extg`, `tpr` peut extraire une zone d'intérêt dans l'image (lire son manuel), en paramétrant correctement `tpr`, on affichera donc les valeurs des pixels de la bonne colonne. Enfin, la commande `nl` numérote les lignes imprimées par `tpr` et le résultat est affiché graphiquement par la commande `xgraph`. Il est conseillé de faire un script de cette suite de commandes afin de répéter facilement cette opération sur les autres images.
3. Calculer et visualiser le module des images `sinus2.inr` et `sinus3.inr`. Tracer la ligne des coefficients du module non nuls.
4. Refaire le même test sur l'image `sinus3.inr` binarisée (*i.e.* l'image est ramenée à deux niveaux de gris). Pour ceci, il faut seuiller par rapport au niveau de gris 128 en utilisant les commandes `mb` ou `mh` en utilisant la séquence de commandes [1] :

1. L'expression `$(carflo 128)` sera remplacée par le résultat retourné par la commande `carflo 128`.

```
mh -n $(carflo 128) sinus3.inr >sinus3binarisee.inr}.
```

Visualiser le spectre de cette image et comparer avec celui de `sinus3.inr`.

Exercice 2 — Quelques DFT d'images simples

1. DFT d'une sinusoïde non horizontale : à l'aide du script `module.sh`, observer et interpréter le module de la DFT centrée de l'image `sinrot.inr`.
2. DFT d'une gaussienne : observer et interpréter le module de la DFT centrée de l'image `gaussienne.inr`. Pour cela observer le profil des niveaux de gris sur une ligne ou une colonne de l'image `gaussienne.inr` et du module de sa DFT en les traçant sur une même figure.

Exercice 3 — Propriétés de la DFT

Cet exercice se propose de mettre en avant certaines propriétés de la DFT.

1. Observer et interpréter le module de la DFT centrée de l'image `rectangle.inr`. On tracera le profil de niveaux de gris sur la ligne 65.
2. Observer et interpréter le module de la DFT centrée de l'image `rotate.inr`. Cette image correspond à la précédente, après rotation de 30 degrés.
3. Effectuer la somme des deux images précédentes en utilisant l'opérateur `ad`. Observer et interpréter le module de la DFT centrée de l'image résultat.

Exercice 4 — Compréhension de spectres

Observer les images `texture1.inr`, `texture2.inr`, `h.inr` et interpréter leur DFT.

Exercice 5 — Filtrage dans le domaine fréquentiel

1. Récupérer et observer l'image `pulse.inr`. À quel signal (bidimensionnel) correspond-elle ?
   Observer le spectre de sa DFT centrée. Interpréter le résultat.
2. Récupérer et observer l'image `passe_bas.inr`. Cette image correspond en fait à la réponse fréquentielle d'un filtre passe-bas idéal PB. À quoi le voit-on ?
   Tracer ce profil et celui de la DFT (même colonne) précédente sur un même graphique.
3. Calculer $DFT_{re}(A) \times PB$ et $DFT_{im}(A) \times PB$, où $DFT_{re}(A)$ et $DFT_{im}(A)$ sont respectivement la partie réelle centrée et la partie imaginaire centrée de la DFT de A. Calculer le spectre $DFT(A) \times PB$ et interpréter le résultat obtenu.
4. Calculer la DFT inverse en utilisant les images $DFT_{re}(A) \times PB$ et $DFT_{im}(A) \times PB$ afin d'obtenir l'image reconstruite A'. Visualiser et interprétez la partie réelle de cette image.
5. Effectuer les étapes 1, 2 et 3 avec les filtres suivants : `passe_haut.inr` et `passe_bande.inr`, respectivement réponses fréquentielle d'un filtre passe-haut PH et d'un filtre passe-bande Pb.

Exercice 6 — Implémentation d'un détecteur de contours à partir d'un filtre de Sobel

1. Générer les images `sobelx.inr` et `sobely.inr`, contenant respectivement les valeurs de pixels :

-1	0	1
-2	0	2
-1	0	1

et

-1	-2	-1
0	0	0
1	2	1

.

2. À l'aide de la commande `conv`, générer la carte des contours horizontaux et celle des contours verticaux de l'image `lena.inr.gz` et les observer.
3. À partir de ces images, générer les images d'amplitude et de phase du gradient en utilisant les commandes `Inrimage`. On les nommera `amplitude.inr` et `phase.inr`. Visualiser ces deux images.

Détection d'objets simples (I) : détection des coins

Travaux Dirigés 6

Première partie : exercices préliminaires

Exercice 1 — Détecteur de Moravec

Le détecteur de Moravec consiste en une mesure de similarité de l'image au voisinage d'un point dans une direction particulière. Soit :

$$E(W, t_x, t_y) = \sum_{(x,y)\in W} (I(x,y) - I(x+t_x, y+t_y))^2 \tag{1}$$

où W est une fenêtre centrée sur le point à étudier.

1. En étudiant trois configurations type – région uniforme, région avec un bord rectiligne, région avec un coin – montrer que le détecteur de Moravec a la réponse la plus forte sur les configurations de type "coin" quelle que soit la direction (i.e. le couple $(t_x, t_y) \neq (0,0)$) considérée.
2. Comment se comporte le détecteur en présence de contours isolés ? Comment discriminer ces points isolés des coins ?
3. En pratique, comment calcule-t-on $I(x+t_x, y+t_y)$? Comment analyse-t-on E ? Est-ce coûteux ?

Exercice 2 — Détecteur de Harris

1. En supposant que la fonction $(x,y) \mapsto I(x,y)$ est continue et dérivable, écrire le développement limité d'ordre 1 au voisinage de (x_0, y_0). Quelles hypothèses doit-on faire pour obtenir une expression linéaire en (x,y) ?
2. Remplacer le terme $I(x+t_x, y+t_y)$ par son développement limité dans l'équation (1) et en déduire une nouvelle expression pour E.
3. Montrer que E est maintenant quadratique. Quel est l'impact sur la complexité du calcul de E pour différentes directions ? Quelles hypothèses doit-on faire pour supposer correct ce calcul ?
4. Soit une fenêtre W, centrée au point (x_W, y_W), et telle que :

$$w(x,y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(x-x_W)^2 + (y-y_W)^2}{2\sigma^2}\right) \tag{2}$$

   Montrer que :

$$E(W, t_x, t_y) = E(x_W, y_W, t_x, t_y) = w \star DI_{t_x,t_y}(x_w, y_w) \tag{3}$$

   où $DI_{t_x,t_y}(x,y) = (\frac{\partial I}{\partial x}(x,y)t_x + \frac{\partial I}{\partial y}(x,y)t_y)^2$ et $\star$ l'opérateur de convolution discrète.
5. Même question en supposant un moyennage sur une fenêtre carrée centrée en (x_0, y_0) de côté n, c'est-à-dire que $w(x,y) = \frac{1}{n^2}$ si $|x - x_0| \leq n$ et $|y - y_0| \leq n$.

6. En utilisant une notation vectorielle pour DI, réécrire l'équation (3) sous la forme $(t_x, t_y)A\begin{pmatrix} t_x \\ t_y \end{pmatrix}$ où A est une matrice 2×2 qui dépend de la fenêtre W.
7. On note $H = \lambda_1\lambda_2 - \kappa(\lambda_1 + \lambda_2)^2$ où λ_1 et λ_2 sont les deux valeurs propres de A et $\kappa > 0$. Montrer que $H = \det(A) - \kappa\,\mathrm{trace}(A)^2$.
   Avec $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{pmatrix}$ donner une formule pour H à partir des éléments a_{ij} de la matrice A.
   Expliquer comment on peut caractériser un coin présent dans la fenêtre W à partir de la valeur de H.

   Remarque : H est le détecteur de Harris.
8. Que se passe-t-il lorsque la fenêtre W est réduite à un point (c'est-à-dire lorsque la fenêtre centrée en (x_W, y_W) est réduite au point (x_W, y_W)) ? Que peut-on en déduire sur la valeur de σ pour une fenêtre gaussienne ?

Seconde partie : implémentation

Exercice 3

1. Écrire un algorithme qui, à partir d'une image I, produit une image H qui est la réponse du détecteur de Harris en chaque pixel de I. On pourra considérer une fenêtre gaussienne ou une fenêtre constante carrée.
2. En pratique l'image H est souvent bruitée car elle contient de nombreuses fausses alarmes (de faux "coins"). Pour limiter le nombre de fausses alarmes, on peut :
   — seuiller l'image H
   — ne retenir que les candidats qui sont des maxima locaux.
   Expliquer le bien-fondé de cette démarche.
3. Écrire un code `C` qui seuille l'image H en ne considérant que les maxima locaux en utilisant qu'un seul parcours de l'image. On choisira un système de voisinage aux 4 plus proches voisins pour simplifier l'écriture du code.
4. Modifier le code `C` de façon à utiliser 5 parcours d'image :
   — le premier parcours correspond au seuillage de H et remplit une image Hmax,
   — le second parcours calcule les maxima locaux dans la direction droite et met à jour Hmax,
   — les trois derniers parcours calculent les maxima locaux dans les trois autres directions et mettent à jour Hmax.
5. Considérons un des parcours qui calcule le maxima local dans une direction donnée. Reformuler cet algorithme en terme de convolution discrète. Pour les quatre directions, donner le noyau de convolution associé.
6. Écrire la séquence de commandes Inrimage qui calcule les maxima locaux d'une image aux points dont la valeur est supérieure ou égale à un seuil s.
7. Deux choix de fenêtre sont possibles (gaussienne, ou fenêtre carrée). Comment implémenter l'étape de convolution de DI (voir exercice 2.4) en C et par le biais de commandes Inrimage ?

Détection d'objets simples (I) : détection de coins

Travaux Sur Machines 6

Objectifs du TME

La première partie de ce TME concerne l'implantation des algorithmes vus en TD sur la détection de coins. On peut procéder de plusieurs manières :

- Écrire une commande Inrimage, `harris`, qui lit une image, applique l'ensemble de la chaîne de traitement vue en TD et en cours pour détecter les coins de l'image. La commande prend en option des paramètres de l'algorithme (κ, σ pour une fenêtre gaussienne ou n pour un moyennage sur une fenêtre carrée, le seuil t) puis liste dans la sortie standard les coordonnées des coins.
- On peut aussi décomposer le traitement en différentes étapes. C'est souvent une façon intelligente de procéder car certaines de ces étapes peuvent être des filtres utiles dans d'autres situations.
- Enfin, pour chaque cas, on peut soit écrire un code `C`, soit écrire un script de commandes Inrimage puisque dans le cas du détecteur de Harris tout peut être fait sans avoir à écrire une seule ligne de `C`.

Le TME propose différents exercices menant à l'une ou l'autre de ces solutions :

- en `C` : exercices 1, 2 et 3 ;
- en commandes Inrimage et scripts `bash` : exercices 4, 5 et 6.

Il est recommandé d'opter pour la solution pour laquelle on est le plus à l'aise. On peut également implanter toutes les solutions pour parfaire ses connaissances !
La seconde partie du TME (exercice 7) concerne l'expérimentation de ces commandes.

Implantation du filtre de Harris (en C)

Le problème est découpé en petits programmes. Libre à l'étudiant de les assembler dans un programme `C` unique. Les séquences de code `C` de chargement et sauvegarde d'images sont toujours les mêmes, pensez à copier-coller, ou mieux, à en faire des fonctions uniques.

Exercice 1 — Calcul de l'image de Harris

Récupérer le fichier `/Infos/lmd/2017/licence/ue/3I022-2018fev/src/harris.c`, on va le compléter au fur et à mesure dans cet exercice.

1. Ajouter dans le fichier `harris.c` une ou des fonctions (à votre convenance) qui calcule la dérivée en x et la dérivée en y d'une image passée en paramètre. On conseille d'utiliser un filtre de **Sobel** (et la fonction utilitaire `convol()` donnée dans le fichier).
   On testera le programme sur des images simples.
2. Écrire la fonction

   ```
   void gaussian(float sigma, float *G, int n);
   ```

   qui crée un noyau gaussien, c'est-à-dire qu'elle remplit le tableau G qui est de taille n par n de façon telle que :

```
G[i+n*j] = (1.0/(2*M_PI*sigma*sigma))*
exp(-((i-n/2)*(i-n/2)+(j-n/2)*(j-n/2))/(2*sigma*sigma));
```

avec i=0...n−1 et j=0...n−1. M_PI est une constante représentant le nombre π et définie dans le fichier en-tête math.h.

3. En utilisant une convolution gaussienne et les fonctions précédemment écrites, écrire la fonction

```
void harris( float *out, float *in, int dimx, int dimy,
             float kappa, float sigma) ;
```

qui calcule les trois éléments de la matrice A et forme l'image H. Remarque : on fixe généralement la taille du noyau gaussien à 2∗ceil(3∗sigma)+1.

4. Tester le programme final sur des images susceptibles de posséder des coins !

Exercice 2 — Seuillage et calcul des extrema locaux

1. Récupérer le fichier `/Infos/lmd/2017/licence/ue/3I022-2018fev/src/sloc.c` et compléter la fonction `seuilloc()` pour qu'elle calcule les maxima locaux du tampon passé en paramètre. Un maximum local est un pixel dont la valeur est supérieur au seuil `seuil` et aux valeurs des 8 plus proches voisins. Tester le programme sur une image calculée par la commande `harris` de l'exercice précédent.
2. Compléter la commande en listant dans la sortie standard les coordonnées de ces points au format `ix iy` par ligne. Les coordonnées doivent commencer à 1.

Exercice 3 — Visualisation du résultat

1. Pour visualiser le résultat, nous proposons d'utiliser les capacités d'`xvis` pour dessiner. En redirigeant la sortie de la commande précédente dans la commande `xvisdraw` avec les options `-p` et `-b1`, on obtient des lignes ayant le format suivant :

   ##!draw(p ix iy)

   En redirigant ce résultat vers un fichier temporaire :

```
... | xvisdraw -p -b1 > tempo
```

   on appelle alors `xvis` ainsi :

```
xvis image -xsh tempo
```

   On peut aussi forcer l'affichage en rouge des points :

```
... | xvisdraw -p -b1 -c 255/0/0 > tempo
xvis image -xsh tempo
```

2. Représenter les coins par un point blanc ou même rouge n'est pas très visible. On peut alors générer des instructions `xvis` pour dessiner de simples croix, c'est-à-dire deux segments de droites se croisant. Par exemple la séquence :

   ##!draw(L 8 10 12 10, L 10 8 10 12)

   dessinera une croix centrée en (10, 10). Modifier la commande de l'exercice précédent pour qu'au lieu de générer en sortie des lignes `ix iy`, on génère des lignes du type :

##!draw(L ix0 iy ix1 iy , L ix iy0 ix iy1)

avec `ix0=ix-2`, `ix1=ix+2`, `iy0=iy-2`, `iy1=iy+2`. On peut évidemment modifier les tailles et l'orientation des croix. Pour choisir une couleur de remplissage rouge, le script `xvis` doit contenir une commande :

##!draw(C 255 0 0)

avant les commandes de dessin.

Implantation du filtre de Harris (en commandes Inrimage)

Exercice 4 — Interface des scripts

Cet exercice donne le strict minimum requis pour écrire de simples scripts shell qui permettent de lire un nom d'image et les options et d'appliquer les commandes Inrimage en utilisant ces informations. Les étudiants familiers de la programmation shell peuvent allez directement à l'exercice suivant.

1. Un script shell est un fichier texte dont la première ligne commence par `#! /bin/sh` et dont les droits d'exécution sont effectifs. Écrire le script `premier` :

   ```
   #! /bin/sh
   echo "Bonjour !"
   ```

   Puis lui mettre les droits d'exécution :

   ```
   chmod +x premier
   ```

   Enfin, l'exécuter (sans oublier les droits !) :

   ```
   ./premier
   Bonjour !
   ```

2. Les paramètres donnés au script sont lisibles dans celui-ci : on écrit `$i` avec $i = 1, 2, ...$ pour représenter le i-ième paramètre. Exemple avec le script `second` :

   ```
   #! /bin/sh
   echo "Parametre 1: $1"
   echo "Parametre 2: $2"
   echo "Parametre 3: $3"
   ```

   Exécution :

   ```
   ./second toto.inr 3 1.3
   Parametre 1: toto.inr
   Parametre 2: 3
   Parametre 3: 1.3
   ```

3. Par la suite on écrira de simples scripts en fixant à l'avance le rôle de chaque paramètre. Voici un exemple où l'on binarise une image. Écrire le script `binarise` :

   ```
   #! /bin/sh
   mh -n 0.5 $1 $2
   ```

   Exemple d'utilisation :

   ```
   ./binarise cameraman.inr cam.bin
   xvis cam.bin
   ```

ou même :

```
./binarise cameraman.inr | xvis
```

Exercice 5 — Calcul de l'image de Harris

L'exercice 1 se traduit très simplement par une combinaison bien choisie de commandes Inrimage :

- `conv -dir` pour la convolution
- `cim` combinée avec `echo` pour créer des noyaux de convolution
- `mu`, `ad`, `so`, pour multiplier, ajouter, soustraire deux images, `car`, `sc`, élever au carré, multiplier par un facteur d'échelle une image.

Pour générer un noyau gaussien, on utilisera la commande `gauss`, disponible dans le répertoire `/Infos/lmd/2017/licence/ue/3I022-2018fev/src` qui écrit dans la sortie standard un noyau gaussien (au format inrimage) d'écart-type la valeur passée en paramètre :

```
./gauss 1.5 > gauss15.inr
par  gauss15.inr
gauss15.inr      -F=Inrimage          -hdr=1 -x 10   -y 10   -rdecm  -o 4
```

N'oubliez pas que la commande `gauss` doit être dans votre `PATH`!

1. Écrire le script `harris` qui prend un nom d'image en premier paramètre, le nom de l'image de Harris en second paramètre, la valeur de κ en troisième paramètre et enfin la valeur du paramètre σ.
   Exemple :

   ```
   ./harris cameraman.inr cam.har 0.1 1
   ```

Exercice 6 — Seuillage des maxima locaux

1. En utilisant l'exercice 3.6 vu au TD6 écrire le script `maxloc` qui prend un nom d'image en premier paramètre, le nom de l'image seuillée en second paramètre et la valeur du seuil en dernier paramètre.
   Exemple :

   ```
   ./maxloc cam.har cam.max 0.01
   ```

2. Pour visualiser le résultat sous la même forme que dans l'exercice 3, on pourra utiliser la commande script `croix`, disponible dans le répertoire `/Infos/lmd/2017/licence/ue/3I022-2018fev/src`, qui s'utilise ainsi :

   ```
   ./croix image-fond image-point image-croix
   ```

   qui incruste des croix rouges dans `image-fond` correspondant aux pixels blancs dans `image-point` et écrit le résultat dans `image-croix`.

Expérimentations

Exercice 7

1. En utilisant le jeu d'images donné dans le répertoire de l'UE tester le détecteur de coins en faisant varier :
   — la taille de la fenêtre c'est-à-dire le paramètre σ,

— le paramètre κ,
— le paramètre de seuil t.

2. À l'aide de la commande `cim`, créer le filtre détecteur de points vu en cours, il a pour matrice :

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

Appliquer une convolution par ce noyau sur les images de tests puis un seuillage et comparer les résultats obtenus avec le détecteur de Harris.

Détection d'objets simples (II)

Travaux Dirigés 7

Exercice 1 — Transformée de Hough : détection de droites

La transformée de Hough permet de détecter des formes spécifiques dans les images. La contrainte imposée par cette méthode est que la forme recherchée soit paramétrisable, ce qui rend la transformée de Hough intéressante pour la détection des formes simples telles que des droites, cercles ou ellipses. L'avantage majeur de la transformée de Hough est qu'elle tolère bien les discontinuités dans les contours des formes recherchées, et qu'elle est robuste au bruit.
Par la suite, on s'intéresse à la détection de droites en utilisant la transformée de Hough. Pour la paramétrisation des droites, on adopte la solution suivante :

$$x \cos\theta + y \sin\theta = r$$

où r est la distance entre l'origine et la droite, et θ donne l'orientation de r par raport à l'axe Ox (voir figure 1). Pour un point donné, x et y sont les constantes, alors que (r, θ) sont les variables associées à une des droites qui passe par (x, y).

1. Dans une image, quels sont les points intéressants pour la détection de droites ?
2. Dans l'espace des variables (r, θ), qu'on appelle accumulateur, quelle est la forme générée par toutes les droites qui passent par un point (x, y) donné ? Donnez un exemple du domaine de définition pour ces variables.
3. Considérons deux points (x_1, y_1) et (x_2, y_2). Dans l'espace (r, θ) quelle est la représentation de la droite qui les relie ? Même question s'il s'agit de n points colinéaires.
4. À partir d'une paire (r, θ), comment peut-on dessiner dans l'image initiale la droite qui lui correspond ?
5. Interprétez l'image de la figure 2 et son accumulateur correspondant, figure 3, (θ se trouve sur l'axe horizontal).
6. Écrire un algorithme qui prend en compte les remarques antérieures afin de détecter les droites dans une image donnée.

Exercice 2 — Transformée de Hough généralisée

1. Quels changements sont nécessaires pour que l'algorithme détecte cette fois-ci des cercles ? Des ellipses ?
2. Si on considère que chaque paramètre de l'accumulateur peut prendre 100 valeurs différentes, quelle sera la taille de la mémoire demandée pour une détection :
    1. de droites ?
    2. de cercles ?
    3. d'ellipses ?

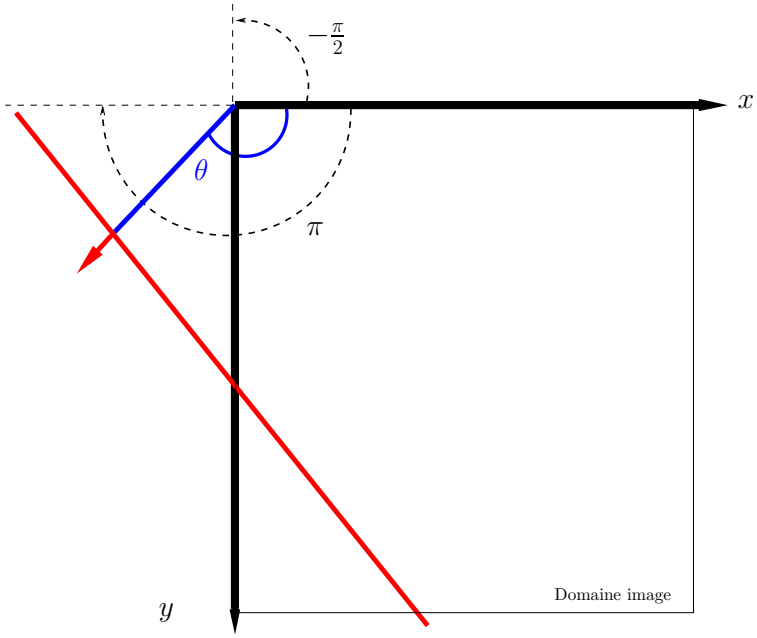


FIGURE 1 – Droite en coordonnées polaire (r, θ)

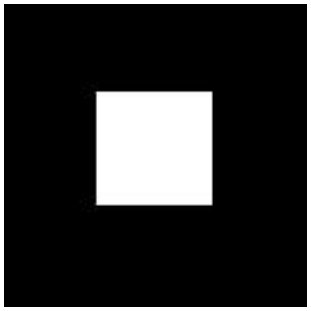


FIGURE 2 – Image initiale

FIGURE 3 – Accumulateur

Détection d'objets simples (II)

Travaux encadrés sur machines 7

Exercice 1 — Transformée de Hough

On souhaite implémenter un détecteur de droites à l'aide de la transformée de Hough. À cette fin, on récupérera le fichier `/Infos/lmd/2017/licence/ue/3I022-2018fevsrc/hough.c` que l'on complétera au gré des questions posées dans cet exercice.

1. Ajouter dans `hough.c` la fonction de prototype :

```
/* retourne 1 si (ix,iy) appartient a la droite
 * d'equation x*cos(theta) + y*sin(theta) = r
 */
int pixelinline( int ix, int iy, float theta, float r, float tolerance);
```

   qui teste si un pixel de coordonnées `(ix,iy)` appartient à la droite `(theta,r)`. Le paramètre `tolerance` indique que l'on tolère que le pixel soit à une distance de la droite d'au plus `tolerance`.

2. Ajouter dans `hough.c` la fonction de prototype :

```
/* *ntheta - calcul de nombre de pas pour theta
   *nr - calcul de nombre de pas pour rho
   dtheta - pas pour theta
   dr - pas pour rho
   tolerance - seuil d'appartenance a la droite
   buf, dimx, dimy - image contours, dimensions.
     le tampon peut etre flottant selon les gouts.
*/
int *hough( int *ntheta, int *nr, float dtheta, float dr,
            float tolerance, unsigned char *bufin, int dimx, int dimy);
```

   qui calcule l'accumulateur de Hough pour un tampon de points de contour (une valeur non nulle indique un contour) donné en entrée.

3. **Vérification du bon fonctionnement de la fonction hough().**
   - Dans `hough.c`, compléter la fonction **int** main(**int** argc, **char** **argv) qui charge une image de contours, calcule l'accumulateur de Hough (par appel à la fonction hough()) et écrit sur le disque l'image de l'accumulateur.
   - Avec l'image `rectangle.inr.gz`, vérifier que l'accumulateur de Hough correspond bien à ce qui est attendu (voir figure 3 du TD). Attention, il faut une image de contours. Pour extraire les contours de `rectangle.inr.gz`, utilisez la séquence de commandes :

```
% detc -sob rectangle.inr.gz | mh -n 0 | cco -f > rectangle.cont
```

   - Une question subsidiaire : que font les commandes `detc`, `mh` et `cco` ? Pourquoi a-t-on utilisé `cco` ?

4. Compléter dans `hough.c` la fonction de prototype :

```
void houghlines( float theta, float r, int dimx, int dimy)
```

qui calcule les deux intersections de la droite d'équation $x\cos\theta + y\sin\theta = r$ avec le rectangle de coordonnées $(1, 1, dimx, dimy)$. Les deux intersections trouvées sont écrites dans la sortie standard au format `xvisdraw` c'est-à-dire :

```
##!draw(p x1 y1)
##!draw(l x2 y2)
```

où $(x1, y1)$ et $(x2, y2)$ sont les coordonnées des deux intersections.

5. Compléter dans `hough.c` la fonction de prototype

```
void findmax(int *acc, int sizex, int sizey,
             double dr, double dt, int n, int dimx, int dimy);
```

qui
— cherche les n premiers maxima locaux d'une image `acc`, de dimensions `sizex` et `sizey`,
— appelle pour chacun de ces n maxima locaux la fonction houghlines(). Pour cela, il faut spécifier les pas en ρ et θ (paramètres dr et dt) qui ont été utilisé pour construire l'accumulateur), ainsi que les dimensions (paramètres dimx et dimy) de l'image.

6. Modifier/compléter la fonction `main()` de `hough.c` pour qu'elle lise une image de contours, calcule l'accumulateur de Hough, affiche les intersections des droites correspondant aux n premiers maxima locaux de l'accumulateur de Hough au format `xvisdraw`.
7. (voir cours Inrimage) Rediriger la sortie de la commande `hough` vers un fichier. Comment paramétrer `xvis` pour afficher simultanément l'image (sur laquelle on a appliqué le détecteur de contours puis la commande `hough`) et les droites que la commande `hough` a détecté ?
8. Testez sur les images `rectangle.inr.gz` et sur `fenetre.inr.gz` disponibles dans `/Infos/lmd/2017/licence/ue/3I022-2018fev/images`.

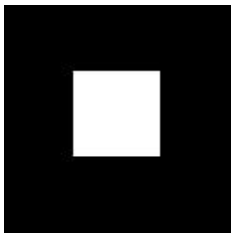


FIGURE 1 – Image initiale

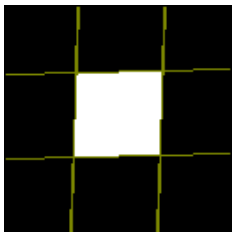


FIGURE 2 – Détection droites

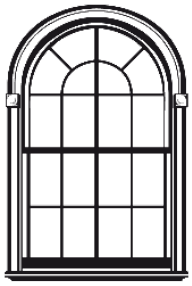


FIGURE 3 – Image initiale

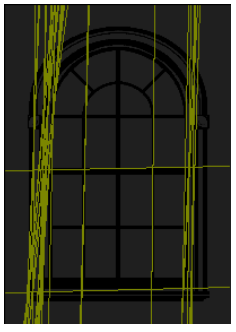


FIGURE 4 – Détection droites

Exercice 2 — Détection de cercle

Implémentez de la même façon l'algorithme de détection de cercles par transformée de Hough. La commande `xvisdraw` qui permet de dessiner le cercle de centre (x, y) et de rayon r est :

```
##!draw(a x-r y-r 2*r)
```

Testez-le sur l'image `cerc.inr.gz` disponible dans `/Infos/lmd/2017/licence/ue/3I022-2018fev/images`.

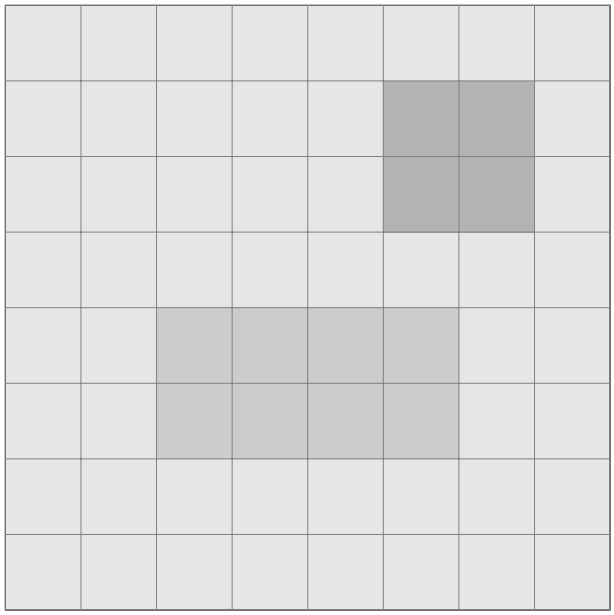
Segmentation

Travaux Dirigés 8 et 9

Première partie : segmentation par découpage (*split*)

Exercice 1 — Principe du découpage

1. Rappeler l'algorithme de segmentation par découpage.
2. Illustrer le fonctionnement de l'algorithme sur l'image 64×64 suivante (chaque élément de la grille représente un carré 8×8) :



   On construira au fur et à mesure l'arbre 4-aire.
3. Quel prédicat de découpage peut-on utiliser pour segmenter correctement cette image ?
4. Calculer les moyennes et variances de chaque feuille de l'arbre (les niveaux de gris du fond, du carré et du rectangle sont respectivement à 150, 50 et 100).

Exercice 2 — Implantation

Pour implanter l'algorithme de découpage, il faut donc construire un arbre 4-aire. Le parcours des feuilles de l'arbre donne alors la liste des régions.
Pour ne pas réinventer la roue, on utilise la librairie glib qui permet de manipuler des arbres n-aires et des listes chaînées. Deux fonctions nous seront utiles :

```
/* creer un noeud */
```

```
GNode* g_node_new( void *data);
/* ajout de 'node' comme fils de 'parent'. Retourne le noeud insere */
GNode *g_node_append( GNode *parent, GNode *node);
```

La structure GNode est abstraite, son contenu nous importe peu, mis à part le champ data qui pointe vers la donnée du nœud.
Exemple : le code suivant crée un arbre avec 3 feuilles, chaque nœud possède une valeur entière.

```
void exemple_arbre( void) {
  int r=1, a=2, b=3, c=4;
  GNode *racine;
  racine = g_node_new( &r);
  g_node_append( racine, g_node_new(&a));
  g_node_append( racine, g_node_new(&b));
  g_node_append( racine, g_node_new(&c));
}
```

1. Que représente un nœud, une feuille dans l'arbre 4-aire construit par l'algorithme de découpage ? Quelles données doit-il contenir ? Définir en `C` cette structure de données (on la nommera region).
2. Proposer un critère simple de découpage et écrire une fonction **int** predsplit(region *r, ...); qui retourne 1 lorsque le critère de découpage est vérifié sur la région r. La signature de la fonction changera en fonction des paramètres utiles au calcul du critère (typiquement un ou deux seuils).
3. Écrire la fonction de prototype GNode *split(**unsigned char** *buf, **int** dimx, **int** dimy, ...); qui implémente l'algorithme de découpage. En pratique, cette fonction appellera une fonction récursive split_rec ().
4. Écrire la fonction **void** print_regions(GNode *qt); qui imprime dans la sortie standard la liste des coordonnées des régions segmentées ainsi que leurs statistiques (moyenne et variance des niveaux de gris). Pour cela, on fera un appel à la fonction de la librairie glib :

   ```
   g_node_traverse( tree, G_PRE_ORDER, G_TRAVERSE_LEAVES, -1,
                    (GNodeTraverseFunc) mafonction, data);
   ```

   Cet appel réalise un parcours en ordre préfixé de l'arbre tree et applique la fonction mafonction() à chaque feuille visitée (paramètre node). Cette fonction doit avoir le prototype suivant :

   ```
   gboolean mafonction(GNode *node, void *data);
   ```

   et doit retourner 0 (sinon le parcours de l'arbre s'arrête). Le paramètre data est passé en paramètre à mafonction().

Seconde étape : segmentation par fusion (*merge*)

Exercice 3 — Principe de la fusion de régions

1. Rappeler l'algorithme de fusion de régions vu en cours.
2. Faire fonctionner l'algorithme sur l'exemple de l'exercice 1 (après découpage) avec un critère de similarité.
3. Soient deux régions R_1 et R_2 de moyennes μ_1 et μ_2, de variances σ_1^2 et σ_2^2. En supposant que les deux régions sont disjointes, donner une formule de la moyenne et de la variance de $R_1 \cup R_2$ à partir de $|R_1|$,$|R_2|$, μ_1, μ_2, σ_1^2 et σ_2^2.

4. En déduire le critère de fusion basé sur la variance.

Exercice 4 — Implantation de l'algorithme

Une fois le découpage de l'image effectué, la structure QuadTree n'est plus utile et seule la liste des feuilles de l'arbre 4-aire est nécessaire.
Pour implanter l'algorithme de fusion en `C`, on va utiliser intensivement les listes chaînées, et chaque élément de ces listes pointera vers la structure décrivant la région (le bloc).

— Une liste chainée est décrite par la structure GSList qui contient un champ data qui pointe vers la donnée et un champ next qui pointe vers l'élément suivant (ou NULL).

— Ajout d'un élément en début de liste :

```
GSList *g_slist_prepend( GSList *first, void *data);
```

— Ajout d'un élément en fin de liste :

```
GSList *g_slist_append( GSList *first, void *data);
```

— Supprimer un élément :

```
GSList *g_slist_remove( GSList *first, void *data);
```

— Ces trois fonctions sont susceptibles de modifier le premier élément de la liste : c'est pourquoi elles retournent l'adresse du premier élément.

— Chercher un élément :

```
GSList *g_slist_find( GSList *first, void *data);
```

— Le parcours d'une liste se fait classiquement :

```
GSList *first, *l;
for( l=first; l; l=l->next) {
  // l->data pour acceder aux donnees
  ...
}
```

1. Test du voisinage : étant donné deux régions de taille quelconque, proposer un critère pour déterminer si les blocs sont voisins. On utilisera un système de voisinage aux quatre plus proches voisins. Écrire la fonction **int** sontvoisins(region *r, region *s); qui retourne 1 lorsque r et s sont voisins, 0 sinon.
2. Écrire la fonction GSList * listevoisins (region *r, GSList *regions); qui crée la liste chaînée des régions voisines de r parmi les régions disponibles (dans la liste regions).
3. Proposer un prédicat de fusion et écrire la fonction **int** predmerge(region *a, region *b, ...); . On complétera les paramètres de la fonction avec ceux utiles au calcul du prédicat.
4. On complète la structure region avec le champs merged qui contiendra la liste des blocs qui constituent la région fusionnée.
   Écrire le code de la fonction GSList *merge(GSList *regions, ...) qui applique l'algorithme de fusion vue dans l'exercice précédent. La fonction merge() réduit donc la liste regions à une liste de régions fusionnées, chaque élément de la liste pointe sur la liste de bloc qui constitue la région fusionnée (voir Fig. 1).

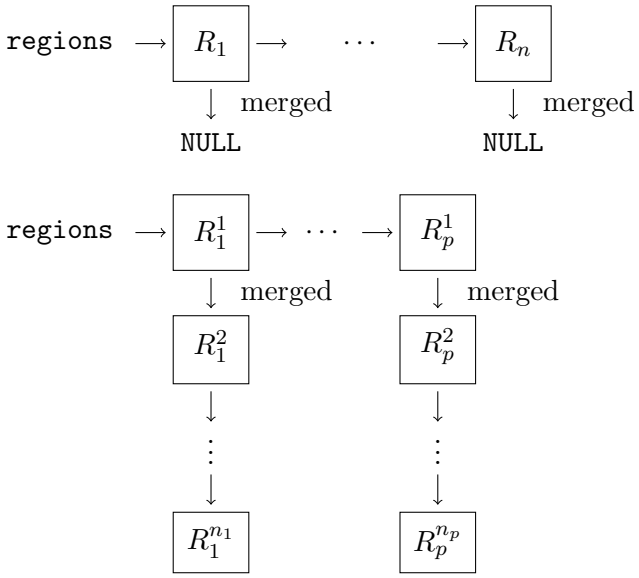


FIGURE 1 – Structure des données avant et après fusion ($p < n$).

Segmentation par découpage/fusion

Travaux encadrés sur machines 8 et 9

Le but du TME est d'utiliser les algorithmes de découpage et de fusion vue en cours et en TD. On travaillera dans le répertoire `3I022-2018fev/tme8-9`.
Récupérez l'archive `/Infos/lmd/2017/licence/ue/3I022-2018fev/src/splitmerge.tar` et décompressez-là dans votre répertoire `tme8-9` [1]. L'archive contient tous les fichiers utiles à la réalisation du TME.

Exercice 1 — Algorithme de découpage (séance 8)

1. Critère de découpage : compléter la fonction predsplit () du fichier `predicats.c` ainsi que la fonction creer_region () du fichier `split.c`.
2. Tester si le critère de découpage fonctionne correctement. Pour ce test, on tapera la commande :

   ```
   make split
   ```

   pour compiler et créer la commande `split` et on la testera sur une des images de la base (par exemple `muscle.inr`).
3. En s'inspirant de la fonction split_print_regions () compléter la fonction du fichier `segment.c` ;

   ```
   void split_segment( GNode *qt, unsigned char *buf, int dimx);
   ```

   qui, en parcourant les feuilles de l'arbre qt, remplit les blocs correspondant dans le tampon image buf ; avec la valeur moyenne de niveau de gris calculé sur le bloc.
   Modifier la fonction main() pour que maintenant elle appelle la fonction split_segment () au lieu d'afficher la liste des régions. L'image segmentée doit ensuite être écrite sur le disque en utilisant comme nom le second nom de fichier trouvé dans la ligne de commande.
4. Modifier la fonction main() (fichier `mainsplit.c`) pour qu'elle imprime dans la sortie d'erreur (c'est-à-dire le canal stderr) des informations utiles à la bonne utilisation de la commande :
   — la moyenne et la variance de l'image (ces informations se déduisent directement du quadtree),
   — le nombre de régions détectées (on pourra modifier la fonction split_segment () de façon à ce qu'elle retourne ce chiffre).
   Pourquoi est-il utile de connaître la variance de l'image ?
5. (question facultative) Proposer un critère de découpage alternatif. Modifier la commande `split` en conséquence et faites des expériences.

Exercice 2 — Algorithme de fusion (séance 9)

1. Critère de fusion : compléter la fonction **int** predmerge(region *a, region *b) du fichier `predicats.c`.

1. Commande tar cvf

Cette fonction retourne 1 si les régions a et b doivent fusionner ou 0 sinon. Si les régions doivent fusionner, alors la région a est modifiée ainsi : les champs `a->n`, `a->mu` et `a->var` doivent valoir respectivement la taille, la moyenne et la variance de la région fusionnée.

2. **Dupliquer** le fichier `mainsplit.c` en `mainmerge.c` et compléter/modifier la fonction main() en ajoutant la phase de fusion après la phase de découpage des régions.
   On pourra utiliser la fonction merge_list_regions () (définie dans `utils.c`) pour transformer l'arbre 4-aire en une liste chaînée de ses feuilles.
   Le paramètre de seuil utilisé par le prédicat de fusion doit être distinct de celui du prédicat de découpage et correspondre à une option dans la ligne de commande.
3. La fonction split_segment () utilisée par la commande `split` est maintenant inadaptée pour deux raisons :
   1. elle traite en entrée un arbre 4-aire et non une liste chaînée.
   2. il faut non seulement parcourir la liste des régions décrite par la liste chaînée regions retournée par merge() mais également la liste des régions pointée par le champ merged de chaque élément de regions. Pour ceux-ci, comment choisir la couleur de remplissage ?

   Dans `segment.c`, compléter la fonction

   ```
   void merge_segment( GSList *regions, unsigned char *buf, int dimx);
   ```

   qui segmente correctement l'image à partir de la liste chaînée retournée par merge().
4. Compléter la fonction main() de façon à ce qu'elle affiche le nombre de régions avant et après la phase de fusion (on pourra utiliser la fonction **int** g_slist_length (GSList *); qui calcule la longueur d'une liste chaînée).
5. La commande `merge` se compile par l'appel suivant : `make merge`.
   Expérimenter la commande sur les images du dépôt de l'UE (`cameraman.inr.gz`, `muscle.pgm`, `lena.inr.gz`) en faisant varier les seuils de découpage et de fusion.

Compression des images

Travaux dirigés 10

Exercice 1 — Codage de Huffman d'une image

Soit l'image `cameraman.inr` 256×256 réduite sur 8 niveaux de gris. On peut déterminer assez facilement les valeurs de niveaux de gris et leur répartition ainsi :

```
cco -b 3 cameraman.inr | tpr -c -l 1 | sort -n | uniq -c
  12046 0
   3325 36
   2390 73
   8716 109
  19544 146
  18748 182
    594 219
    173 255
```

1. Calculer la probabilité d'apparition de chaque niveau de gris.
2. Calculer l'entropie de cette image.
3. Quel taux de compression (non destructive) peut-on espérer ?
4. Établir l'arbre de Huffman.
5. Donner la table de correspondance entre les niveaux de gris et les codes de Huffman.
6. On extrait une petite partie de l'image (autour de l'objectif de la caméra) :

109	182	109	109	219	109	146	182
146	219	182	182	255	219	219	182
219	219	219	219	219	219	219	182
36	182	73	73	219	73	109	182
36	146	73	73	182	73	109	182
36	109	73	73	109	73	146	182
73	36	36	36	182	146	182	182
182	146	146	146	182	182	182	182

   Quelle est la séquence des codes de Huffman associée à cette partie d'image ?
7. Soit la séquence de codes de Huffman suivante :

   0100101010101010110111010101101101100000100010 0

   Décoder cette séquence.

Exercice 2 — Algorithme de Huffman

1. Rappeler l'algorithme de Huffman qui donne l'arbre de décision, celui qui donne la table de Huffman.

2. Définir une structure de données adéquate pour construire l'arbre de Huffman.
3. Écrire la fonction de prototype suivant :

```
struct node *creer_arbre( float *his, int L);
```

qui construit l'arbre de décision de Huffman à partir du tableau de fréquence d'apparition des symboles (c'est-à-dire de l'histogramme normalisé).
On pourra s'aider de fonctions intermédiaires, comme celle, par exemple, qui cherche l'élément de fréquence minimale dans la liste des fréquences.

4. Écrire la fonction de prototype suivant :

```
void litcode( struct node *n, char **table, char *code);
```

qui parcourt l'arbre de décision en profondeur (depuis la racine jusque vers les feuilles).

5. Écrire la fonction de prototype suivant :

```
void hufftable ( float *his, char **table, int L);
```

qui retourne la table de Huffman de l'histogramme normalisé `freq`. La table est décrite par le tableau `table`, chaque entrée de ce tableau décrit la séquence de bits (une suite de caractères '0' ou '1') correspondant aux entrées de `his`.

6. Quel type d'image peut-on traiter ?

Exercice 3 — Encodage et décodage des images

1. Écrire la fonction de prototype

```
int huffenc( unsigned char *out, unsigned char *in, long size, char **table);
```

qui encode l'image `in` codée sur 1 octet de taille size octets et place le résultat dans le tampon `out` en utilisant la table de Huffman `table`. La fonction retourne le nombre de bits écrits dans `out`. On pourra s'aider de la fonction

```
void ecritbit( unsigned char *s, long b, char val) {
  int byte = b / 8;
  int bit = 7 - b % 8;

  s += byte;
  if( val == '1')
    *s |= (1<<bit);
  else
    *s &= ~(1<<bit);
}
```

qui met le bit `b` dans le tampon `s` a 0 si `val` vaut '0' ou 1 si `val` vaut '1'.

2. Écrire la fonction de prototype

```
void huffdec( unsigned char *out, unsigned char *in, long sin, char **table, int n
```

qui décode le tampon `in` qui contient `sin` bits en utilisant la table de Huffman `table` contenant `ntable` codes et place le résultat dans le tampon `out`. On pourra s'aider de la fonction

```
char litbit( unsigned char *s, long b) {
  int byte = b / 8;
  int bit = 7 - b % 8;
  return (*(s+byte) & (1<<bit)) ? '1':'0';
}
```

qui lit le bit b du tampon **s** et retourne sa valeur sous la forme '0' ou '1'.

3. Expliquer le fonctionnement des fonctions ecritbit () et litbit ().

Compression des images

Travaux encadrés sur machines 10

Le but du TME est d'utiliser l'algorithme de Huffman. On travaillera dans le répertoire `3I022-2018fev/tme10`. Récupérez l'archive `/Infos/lmd/2017/licence/ue/3I022-2018fev/src/huffman.tar` et décompressez-là dans votre répertoire `tme10`. Cette archive contient tous les fichiers utiles à la réalisation du TME.

Exercice 1 — Test de l'implémentation

1. Dans le fichier `hisn.c`, compléter la fonction **void** hisn(**float** ∗his , **unsigned char** ∗buf, **long** size); qui calcule l'histogramme **normalisé** du tampon buf.
2. Dans le fichier `essai.c` compléter la fonction `main()` qui :
   — calcule son histogramme normalisé de l'image lue, cette dernière est supposé être de codage 1 octet en virgule fixe,
   — calcule la table de Huffman,
   — encode l'image lue,
   — décode l'image dans un autre tampon et écrit sur le disque ce tampon.
   On utilisera les fonctions définies dans le fichier `huffman.c` pour calculer la table de Huffman, encoder et décoder un tampon.
3. Lecture de l'image : pourquoi se restreint-on à seul un codage, en virgule fixe ?
4. Compiler le programme `essai` à l'aide du `Makefile` :

   ```
   make essai
   ```

   Tester sur différentes images. Comment s'assurer du bon fonctionnement du programme ?
5. Compléter le programme en lui faisant afficher les statistiques suivantes :
   — entropie de l'image d'entrée,
   — taux effectif de la compression.

Exercice 2 — Un encodeur et un décodeur

On souhaite maintenant faire deux commandes distinctes :
— un encodeur de Huffman, que l'on appellera `inr2huf`, qui lit une image et écrit l'image encodée dans un fichier sous une forme que l'on va préciser,
— un décodeur, que l'on appellera `huf2inr`, qui réalise l'opération inverse.

L'image encodée sera stockée dans un fichier binaire. Ce fichier contient une entête (qui décrit la taille de l'image et autres informations utiles). Les fonctions qui permettent d'écrire et de lire de tels fichiers sont définies dans le fichier `huffio.c`.

1. En s'inspirant du fichier `essai.c` de l'exercice précédent, compléter le fichier `inr2huff.c` qui lit une image Inrimage, l'encode avec l'algorithme de Huffman et écrit l'image encodée sur le disque. L'appel à :

```
make inr2huff
```

doit créer la commande `inr2huf` qui encode une image.

2. Même question pour le fichier `huff2inr.c` qui lit une image encodée, la décode et l'écrit sur le disque au format Inrimage. L'appel à :

```
make huff2inr
```

doit créer la commande `huf2inr` qui décode une image.

3. Tester les deux commandes et vérifier leur bon fonctionnement. Vérifier les tailles des images encodées. Remarque : les images du dépôt au format `.inr.gz` sont compressées par la commande `gzip`, un encodeur de type Huffman très efficace. Lequel des deux encodeurs (`gzip` et `inr2huff`) est le plus efficace ?
Pour évaluer l'efficacité de l'encodeur, encoder les images PGM (qui ne sont pas compressées) et comparer les tailles. On peut aussi décompresser les images `.inr.gz` : il faut les copier dans son répertoire puis leur appliquer la commande `gunzip`.

Exercice 3 — Prédicteur spatial

Pour améliorer les performances de l'encodeur de Huffman, on peut utiliser un prédicteur spatial. Le rôle du prédicteur est de prédire la valeur d'un pixel à partir de la valeurs des pixels précédents ou voisins. On encode alors la différence entre l'image et l'image prédite. Si le prédicteur est efficace, on aura peu d'erreurs, d'où une dynamique réduite et l'encodeur de Huffman sera alors plus efficace.
Le but de cet exercice est de le vérifier expérimentalement. Pour cela on utilisera le prédicteur le plus simple : la valeur d'un pixel est prédite par celle du pixel précédent sur la ligne.
Ainsi, pour j variant entre `0` et `NDIMY-1` on a :

$$\hat{I}(0,j) = I(0,j) \quad (1)$$

$$\hat{I}(i,j) = I(i-1,i) \quad i = 1, \cdots, \texttt{NDIMX-1} \quad (2)$$

et $\hat{I}$ est l'image prédite. On encode donc $I - \hat{I}$.

1. Dans le fichier `lpc.c`, compléter la fonction `mat2lpc()` qui calcule l'image prédite et la soustrait à l'image originale.
2. Même question avec la fonction `lpc2mat()` qui réalise l'opération réciproque à celle faite par `mat2lpc()`.
3. Compiler les commandes `mat2lpc` et `lpc2mat` en utilisant le Makefile. Ces deux commandes font appel aux deux fonctions précédentes.
   a) Vérifier leur bon fonctionnement :

```
$ ./mat2lpc image | ./lpc2mat > image2
```

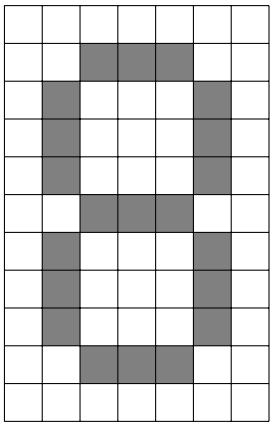
   doit donner la même image.
   b) Visualiser l'image transformée par la commande `mat2lpc`.
   c) Appliquer l'encodeur de Huffman à une image transformée et vérifier que la taille diminue.
   d) Décoder l'image puis appliquer la commande `lpc2mat` et vérifier qu'on retrouve l'image d'origine.

Représentation des objets

Travaux Dirigés 11

Exercice 1 — Représentation des bords

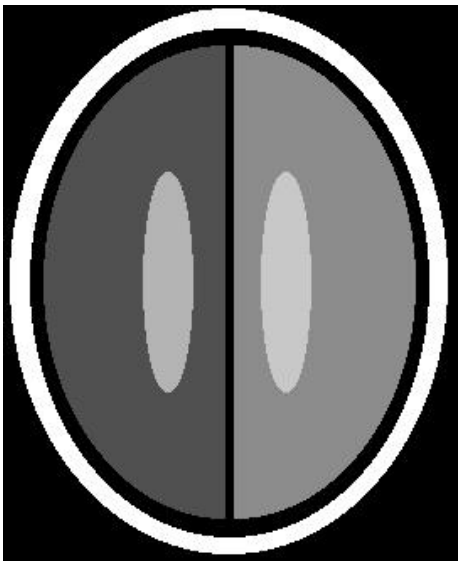
Soit l'image binaire suivante :



1. Appliquer l'algorithme de traçage de bord (algorithme de Moore) pour un système de voisinage aux huit plus proches voisins.
2. Établir le code de Freeman en partant du point le plus haut et le plus à gauche.
3. Établir la première différence du code précédent ainsi que son numéro de forme (*shape number*).
4. Montrer que redéfinir le point de départ pour le calcul de la première différence du code de Freeman ne change pas son numéro de forme.

Exercice 2 — Régions hiérarchiques

Soit l'image synthétique suivante :



Cette image représente un "modèle" d'une coupe IRM d'un cerveau humain. La région blanche représente le crâne. Les deux régions grises foncées représentent la matière grise des deux hémisphères du cerveau. Les deux régions grises claires représentent la matière blanche du cerveau. Enfin la région noire située à l'intérieur du crâne représente le liquide céphalo-rachidien (LCR). Une telle image s'appelle un *atlas* en imagerie médicale.

1. Décomposer cette image en une hiérarchie de régions. Quelles formes géométriques représentent ces régions ?
2. Proposer une série de règles pour décider à quelle région appartient un pixel donné.

Exercice 3 — Utilisation des commandes Inrimage

1. Soit une image binaire représentant une courbe fermée (disons un cercle).
   a) Décrire cette image en terme de régions.
   b) Comment obtenir une description de ces régions ?
   c) Comment créer l'image d'un disque à partir de la description précédente ?
   d) Même question pour obtenir une image de la région extérieure au disque ?
   e) Comment reconstruire l'image originale à partir du fichier des contours ?
2. Soit une image binaire contenant une unique région pleine (disons un disque). Comment créer une image du contour de cette région ?
3. Proposer une suite de commandes `Inrimage` permettant de construire l'image de l'exercice précédent.

Représentation d'objets

Travaux encadrés sur machines 11

Objectif du TME

Dans ce TME, nous allons essayer de segmenter une image médicale (imagerie par résonance magnétique – IRM), de décomposer cette image en différentes régions. Enfin, nous essayerons d'identifier chaque région grâce au modèle (atlas) étudié en TD. On utilisera l'image `IRM.inr.gz` disponible dans le répertoire de l'UE (`/Infos/lmd/2017/licence/ue/3I022-2018fev`). L'atlas est l'image `atlas.inr.gz`.

Exercice 1 — Segmentation du crâne et de la matière blanche

1. Identifier visuellement chaque région de l'image en se référant à l'atlas donné en TD. À l'aide d'`xvis`, inspecter à la souris chaque région et relever les niveaux de gris moyens :
   — En maintenant le bouton droit enfoncé, `xvis` indique en haut de sa boîte le niveau de gris et les coordonnées du pixel pointé par la souris.
   — En sélectionnant l'option '`print pixel value`' dans le menu '`Options->`, un clic avec le bouton du milieu imprime dans la sortie standard le niveau de gris et les coordonnées du pixel pointé par la souris.
2. Essayer d'isoler la région crânienne en utilisant un seuillage adapté (voir `man arit1` qui récapitule toutes les commandes de seuillage).
   Conseils :
   — les paramètres de seuil de ces commandes acceptent des valeurs de niveau de gris normalisées. On peut utiliser la commande `carflo` pour convertir une valeur de niveau de gris vers une valeur normalisée.
   — On peut, de façon facultative, prétraiter l'image avec la commande `fmoy` et ceci pour permettre d'obtenir une région plus régulière après le seuillage.
3. Un seuillage haut permet d'isoler le crâne de façon satisfaisante mais on obtient également des régions non désirées (la matière blanche précisément). Avec la commande `anac`, extraire les contours des régions. Puis avec un appel bien paramétré à la commande `fillc` reconstruire une image du crâne sans les régions parasites.
4. Écrire un script `crane.sh` qui lit l'image `IRM.inr.gz` et génère l'image du crâne qu'on appellera `region1.inr`.
5. Avec une stratégie similaire, écrire un script `mb.sh` qui génère deux images de la matière blanche, chaque image correspondant à l'un des deux hémisphères du cerveau. On les appellera `region2.inr` et `region3.inr`.

Exercice 2 — Segmentation de la matière grise

La segmentation de la matière grise nécessite également la combinaison d'un seuillage bas et l'élimination

de régions parasites.

1. Commencer par trouver une valeur de seuil bas qui soit visuellement correct avec la commande `mh`. Proposer deux valeurs possibles de seuil bas :
   — une première valeur qui segmente la matière grise en une seule région.
   — une seconde valeur qui segmente la matière grise en deux régions, chaque région correspondant à un hémisphère.
   — commenter la régularité des régions obtenues pour chacune de ses deux segmentations.
2. Compléter la chaîne de traitement de façon à éliminer le crâne de l'image ainsi que les autres régions parasites. Faire un script `mg.sh` qui génère :
   — une image binaire appelée `mg.inr` qui contient une unique région de matière grise.
   — une image binaire appelée `region4.inr` qui décrit la matière grise de l'hémisphère droit.
   — une image binaire appelée `region5.inr` qui décrit la matière grise de l'hémisphère gauche.

Exercice 3 — Segmentation du liquide céphaloraphidien

Pour obtenir la région LCR, on procède indirectement, en considérant que c'est la région comprise entre la paroi intracrânienne et la matière grise.

1. À partir du fichier de contour obtenu dans l'exercice 1, générer une image binaire de la cavité intracrânienne.
2. Générer une image binaire de la matière grise la plus régulière possible.
3. Enfin, écrire le script qui génère l'image binaire `lcr.inr` obtenue par soustraction des deux images précédentes.

Exercice 4 — Reconnaissance des régions

Les exercices précédents ont permit de décomposer l'image IRM en plusieurs images (`region1.inr`, ..., `region5.inr`) chacune contenant une région particulière. Nous essayons maintenant de reconnaître ces régions, c'est-à-dire de les mettre en correspondance avec celles de l'atlas.

1. La première étape consiste à trouver une caractérisation suffisante de chaque région. On se propose de calculer la taille et le centre de gravité d'une région, une région étant l'ensemble des pixels d'une certaine couleur dans l'image.
   Écrire un programme `statsreg` qui lit une image, prend une valeur c de niveau de gris comme option dans la ligne de commande et écrit dans la sortie standard la taille et le centre de gravité de la région ayant la couleur c.
   Conseil : il est recommandé d'écrire une commande qui impose un format simple en virgule fixe (disons 1 octet). Pour traiter les images ayant un codage différent, il suffit de convertir l'image à traiter au format 1 octet virgule fixe (`cco -f`) puis de rediriger la sortie de `cco` vers la commande `statsreg`.
2. Utiliser la commande `statsreg` pour établir les statistiques des cinq régions de l'atlas.
3. Utiliser la commande `statsreg` pour établir les statistiques des cinq régions extraites de l'image `IRM.inr.gz`.
4. Proposer une méthode pour renommer les cinq images `region[1-5].inr` avec les noms suivants : `crane.inr`, `mgg.inr`, `mgd.inr`, `mbg.inr`, `mbd.inr`.

5. Comment construire la région `lcr.inr` ?