Projet TIN : Traitement d'Images Numériques

Table des matières

1. Introduction 2
1.1. Évaluation du projet 2
1.2. Niveaux de difficultés du projet 2
1.3. Les différentes parties du projet 2
1.4. Comment tester vos fonctions et programmes 3
2. Commençons avec les images noir et blanc 3
2.1. Le format PBM non compressé (*) 3
2.2. Description du format PBM 3
2.3. Affichage d'une image PBM (*) 3
2.4. Inverser le blanc et le noir (*) 4
2.5. Lecture d'une image PBM (*) 4
2.6. Écriture d'une image PBM (*) 4
2.7. Tests de lecture et d'écriture 5
2.8. Affichage et Inversion 5
3. Attaquons les images en niveau de gris 5
3.1. Description du format PGM non compressé 5
3.2. Le type `ImageGris` 5
3.3. Lecture d'une image PGM (*) 6
3.4. Écriture d'une image PGM (*) 6
3.5. Tests des fonctions de lecture et d'écriture (*) 6
3.6. Inversion des niveaux de gris pour créer des images cliché (*) 6
4. Extraction de contours 7
4.1. Implantez le filtre de Sobel pour estimer le gradient (*) 7
4.2. Seuillage des gradients (*) 8
4.3. Aller plus loin (**) 9
4.4. Seuillage de l'intensité (**) 9
4.5. Double seuillage de l'intensité du gradient (***) 9
4.6. Croissance du contour de 1 pas (***) 9
4.7. Croissance du contour de plusieurs pas et visualisation (**) 10
4.8. Aller plus loin (***) 10
4.9. Aller beaucoup plus loin 10
5. Vers une bibliothèque de traitement d'images 10
5.1. Types d'image (*) 10
5.2. Module `pgm` (*) 10
5.3. Modules `sobel` et `seuillage` (*) 11
5.4. Automatisation avec `make` (*) 11
5.5. Utilisation de la bilbiothèque (*) 11
6. Abordons maintenant les images en couleurs 11
6.1. Description du format PPM 11
6.2. La structure `Couleur` et le type `Image` 12
6.3. Lecture et écricture d'image PGM (**) 12
6.4. De la couleur au niveau de gris (**) 12

6.5. Conversion image couleur vers gris avec des logiciels 13
7. Segmentation en régions par la méthode du super pixel 13
7.1. L'algorithme des K-moyennes (***) 14
7.2. Application au regroupement de pixels (**) 16

1. Introduction

Le projet TIN permet de mettre en pratique la majorité des notions vues dans le module Info 111. Il est prévu que vous y consacriez chacun une vingtaine d'heures de travail intense, dont quatre en séances de TP et le reste en autonomie.

1.1. **Évaluation du projet.** Votre travail sera évalué lors de la dernière séance de TP (semaine du 6 décembre) sous la forme d'une présentation composée d'un petit rapport et d'une soutenance orale individuelle (4 minutes) de votre réalisation, suivie de quelques minutes de questions. La présentation devra inclure :

— Une description précise des fonctionnalités implantées ;
— Quelques éléments pour étayer la robustesse de l'implantation (jeux de tests utilisés) ;
— Les difficultés rencontrées ;
— Une discussion sur quelques extraits de code bien choisis.

Elle pourra utilement être complétée par une mini démonstration.

Il est fortement recommandé de travailler en binôme. Cependant vous devrez démontrer, durant la présentation orale, votre maîtrise de l'ensemble du projet.

1.2. **Niveaux de difficultés du projet.** Les sections contenant des questions à traiter sont annotées comme suit en fonction de leur difficulté :

— Question facile : « (*) »
— Question moyenne : « (**) »
— Question difficile : « (***) »

Les sections *Aller plus loin* sont facultatives, mais vous donnent des bonus.

Ce projet peut paraître impressionnant. Ne vous fiez pas à votre première impression. D'une part, il est volontairement long afin que chacun puisse s'exprimer en fonction de ses compétences, de son éventuelle expérience préalable et de ses goûts. À vous de choisir un sous-ensemble adapté des questions. Il a été conçu pour qu'un étudiant sans expérience de programmation préalable mais ayant suivi le module avec assiduité puisse facilement avoir au minimum 12. D'autre part, l'expérience que vous accumulerez au cours des premières questions vous fera paraître les questions ultérieures plus simples. Et vous aurez un vrai sentiment d'accomplissement en progressant dans le sujet.

Toutes les parties portent sur des techniques basiques de traitement d'images. Mais bien évidemment, ce n'est qu'un prétexte pour vous faire travailler les notions centrales du cours : tests, tableaux, boucles, fonctions, compilation séparée. Aucune notion de traitement d'images n'est prérequise.

1.3. **Les différentes parties du projet.** Le projet comporte six parties principales :

— 2 : Traitements basiques d'images binaires
— 3 : Traitements basiques d'images en niveaux de gris
— 4 : Extraction de contours
— 5 : Mise en place d'une bibliothèque de traitement d'images
— 6 : Traitements d'images en couleurs
— 7 : Segmentation en régions homogènes

Remarque : les premiers exercices sont faits dans des fichiers indépendants. Puis, à la fin du projet, on passe à une organisation multi-fichiers dans le but de créer une bibliothèque de traitement d'images.

Les exercices de la partie 2 sont pensés de façon progressive pour introduire les notions de base dont nous aurons besoin dans le projet.

Les fonctions de la partie 3 : sont indispensables pour la suite du projet. L'objectif est d'arriver à implanter la partie 4 (qui dépend de 3) et la partie 7 (qui dépend de 3 et 6).

À l'intérieur d'une partie, les questions se suivent (sauf entre les sections 7.1 et 7.2 où une fonction est fournie pour continuer). En dehors de ces dépendances, vous pouvez travailler sur ce projet dans l'ordre que vous souhaiterez.

Une archive est fournie sur le web, contenant :

— Une proposition de squelette des fichiers, avec de la documentation et des tests ;
— Des exemples d'images et des résultats de traitements par les différents filtres pour comparer avec les vôtres.

1.4. **Comment tester vos fonctions et programmes.** Lors des premiers exercices, vous trouverez des tests directement dans les fichiers à remplir. Pour le dernier exercice, une batterie de tests est fournie dans le fichier `tests.cpp`. À noter qu'une partie seulement des tests est automatique : le nom des images que vous devrez comparer visuellement s'affichera à l'écran.

2. Commençons avec les images noir et blanc

2.1. **Le format PBM non compressé (*).** Dans la première partie de ce projet, on se contentera de lire et d'écrire des images en noir et blanc au format `PBM` non compressé.

2.2. **Description du format PBM.** La page `http://fr.wikipedia.org/wiki/Portable_pixmap` détaille le format général d'un fichier PBM. L'entête est de la forme :
`P1`
nbColonne nbLigne (par exemple `512 512`)

suivi d'une liste de 0 ou de 1, séparés par des espaces ou des sauts de lignes, décrivant les pixels de haut en bas et de gauche à droite (**Attention :** les lignes de l'image ne sont pas forcément les mêmes que celles du fichier !).

Prenez le temps d'ouvrir les fichiers PBM qui sont fournis dans l'archive du projet, avec un éditeur de texte comme *notepad* ou *gedit* pour observer le format.

En principe, les spécifications du format de fichier PBM indiquent de plus qu'aucune ligne du fichier ne doit dépasser 70 caractères ; cependant la plupart des logiciels s'en sortent même si cette contrainte n'est pas respectée. Les spécifications indiquent aussi qu'un fichier PBM peut contenir des commentaires, sous la forme de ligne commençant par un '#'.

2.3. **Affichage d'une image PBM (*).** Le premier exercice de ce projet consiste à afficher une image PBM dans le terminal : les pixels blancs seront des espaces et les pixels noirs des symboles "@".

Par exemple :

```
0 1 1 0                          @ @
1 0 0 1    s'affichera         @     @
1 0 0 1                        @     @
0 1 1 0                          @ @
```

Dans le fichier `pbm-affiche.cpp`, implantez la fonction suivante. Puis compilez et vérifiez que votre programme affiche correctement l'image smiley.

```
/** Affiche une image binaire PBM à l'écran avec ' ' pour 0 et '@' pour 1
 * @param source le nom d'un fichier PBM
 **/
void affichePBM(string source)  {
```

2.4. **Inverser le blanc et le noir (*).** Complétez la fonction `inversePBM` du fichier `pbm-affiche.cpp`. Cette fonction crée une nouvelle image à partir d'une image existante en inversant le noir et le blanc.

Compilez et vérifiez que l'exécution donne le résultat attendu.

2.5. **Lecture d'une image PBM (*).** Dans ce projet, on va effectuer de nombreux traitements d'images. Parfois, il sera utile de charger l'image dans un tableau pour effectuer ces traitements. En particulier, cela permet de séparer et d'implanter une fois pour toutes tout ce qui concerne la lecture et l'écriture des fichiers.

Pour cela, on définit le type suivant :

```
/** Structure de données pour représenter une image binaire **/
typedef vector<vector<int> > ImageNB;
```

Une image numérique en noir et blanc est donc représentée par un double tableau contenant des entiers (0 ou 1). Par convention, le pixel (0,0) (auquel on accède dans l'image `img` par `img[0][0]`) est le coin en haut à gauche de l'image. Le premier indice désigne les lignes et le deuxième les colonnes. Par exemple, le pixel (2,10) correspondant à `img[2][10]` est le pixel de la troisième ligne et de la onzième colonne (si l'image est suffisamment grande pour contenir ce pixel bien entendu).

Implantez la fonction `lirePBM` du fichier `pbm-tout-en-un.cpp` :

```
/** Construire une image binaire depuis un fichier PBM
 * @param source le nom d'un fichier PBM
 * @return une image binaire (0/1)
 **/
ImageNB lirePBM(string source) {
```

Indication : utilisez les instructions suivantes pour ouvrir un fichier et émettre un message d'erreur si celui-ci n'existe pas :

```
    ifstream PBM;
    PBM.open(source);
    if (not PBM)
        throw runtime_error("Fichier non trouve: "+source);
```

Optionnel : gérer les fichiers contenant des commentaires ; si vous ne le faites pas, il vous faudra vérifier que les fichiers PBM que vous utilisez sont effectivement sans commentaires et les supprimer le cas échéant.

2.6. **Écriture d'une image PBM (*).** Implantez la fonction :

```
/** Ecrit une image binaire dans un fichier PBM
 * @param img une image binaire (0/1)
 * @param cible le nom d'un fichier PBM
 **/
void ecrirePBM(ImageNB img, string cible) {
```

Indications :
— Pour l'écriture, il est acceptable de ne mettre qu'un pixel par ligne (c'est moins lisible pour l'humain mais plus facile à programmer et équivalent pour l'ordinateur).

2.7. **Tests de lecture et d'écriture.** Des tests sont fournis dans le fichier `pbm-tout-en-un.cpp`. Par défaut, le programme lance ces tests. Compilez et exécutez le programme `pbm-tout-en-un.cpp` et vérifiez que vos codes sont corrects.

2.8. **Affichage et Inversion.** À présent, implantez dans `pbm-tout-en-un.cpp` une nouvelle version de vos fonctions `affichePBM` et `inversePBM` en utilisant le type `imageNB`.

```
/** Affiche une image binaire PBM à l'écran avec ' ' pour 0 et '@' pour 1
 * @param img une image binaire (0/1)
 **/
void affichePBM(ImageNB img) {
```

```
/** Echange le noir et le blanc dans une image PBM
 * @param img une image binaire (0/1)
 * @return l'image où le blanc et le noir ont été inversés
 **/
ImageNB inversePBM(ImageNB img) {
```

Modifiez la fonction `main` pour qu'elle affiche et inverse l'image du smiley en utilisant ces nouvelles fonctions.

3. Attaquons les images en niveau de gris

3.1. **Description du format PGM non compressé.** La page `http://fr.wikipedia.org/wiki/Portable_pixmap` détaille le format général d'un fichier PGM non compressé. L'entête est de la forme : `P2`
nbColonne nbLigne (par exemple `512 512`)
`255`
suivi d'une liste d'entiers, séparés par des espaces ou des sauts de lignes, décrivant les pixels de haut en bas et de gauche à droite (**Attention :** les lignes de l'image ne sont pas forcément les mêmes que celles du fichier !).

Prenez le temps d'ouvrir les fichiers PGM fournis dans l'archive du projet avec un éditeur de texte comme *notepad* ou *jedit* pour observer le format.

En principe, les spécifications du format de fichier PGM indiquent de plus qu'aucune ligne du fichier ne doit dépasser 70 caractères ; cependant la plupart des logiciels s'en sortent même si cette contrainte n'est pas respectée. Les spécifications indiquent aussi qu'un fichier PGM peut contenir des commentaires, sous la forme de lignes commençant par un '#'.

3.2. **Le type `ImageGris`.** Tout comme les images en noir et blanc, on va stocker les images en niveau de gris dans un double tableau. On définit le type suivant dans `image.h`.

```
/** Structure de donnees pour representer une image en teintes de gris **/
typedef vector<vector<double> > ImageGris;
```

Lors de la lecture ou de l'écriture d'un fichier image, toutes les valeurs sont des entiers `int`. Cependant il sera commode d'utiliser un type `double` pour ne pas rencontrer de problème de conversion lors des opérations que l'on effectuera sur les images.

3.3. **Lecture d'une image PGM (*).** Implantez la fonction `lirePGM` du fichier `pgm-tout-en-un.cpp` :

```
/** Construire une image en teintes de gris depuis un fichier PGM
 * @param source le nom d'un fichier PGM
 * @return une image en teintes de gris
 **/
ImageGris lirePGM(string source) {
```

Rappel : utilisez les instructions suivantes pour ouvrir un fichier, et émettre un message d'erreur si celui-ci n'existe pas :

```
    ifstream pgm;
    pgm.open(source);
    if (not pgm)
        throw runtime_error("Fichier non trouve: "+source);
```

Optionnel : gérer les fichiers contenant des commentaires ; si vous ne le faites pas, il vous faudra vérifier que les fichiers PGM que vous utilisez sont effectivement sans commentaires et les supprimer le cas échéant.

3.4. **Écriture d'une image PGM (*).** Implantez la fonction `ecrirePGM` du fichier `pgm-tout-en-un.cpp` :

```
/** Ecrit une image en teintes de gris dans un fichier PGM
 * @param img une image en teintes de gris
 * @param cible le nom d'un fichier PGM
 **/
void ecrirePGM(ImageGris img, string cible) {
```

Indications :

— Pour tronquer une valeur `x` de type `double` en entier, il suffit d'utiliser l'opération `((int)x)`. Par exemple `((int)2.5)==2`. Ainsi, `((int)img[2][10])` transforme en entier le pixel (2,10) de l'image `img`.

— Pour l'écriture, il est acceptable de ne mettre qu'un pixel par ligne (c'est moins lisible pour l'humain mais plus facile à programmer et équivalent pour l'ordinateur).

3.5. **Tests des fonctions de lecture et d'écriture (*).** Des tests sont fournis dans le fichier `pgm-tout-en-un.cpp`. Par défaut, le programme lance ces tests. Compilez et exécutez le programme `pgm-tout-en-un.cpp` et vérifiez que vos codes sont corrects.

3.6. **Inversion des niveaux de gris pour créer des images cliché (*).** Une image cliché s'obtient en inversant les valeurs des niveaux de gris dans une image PGM.

— la valeur 0 devient 255
— la valeur 1 devient 254
— la valeur 2 devient 253
— ...
— la valeur 255 devient 0

Implantez la fonction `inversePGM` du fichier `pgm-tout-en-un.cpp` et testez le résultat. Remarque : cette question est indépendante de la suite du projet.

4. Extraction de contours

Les formes des objets présents dans une image constituent une information privilégiée en traitement d'image. Ces formes peuvent être obtenues en détectant les contours (frontières) des objets. Dans beaucoup d'algorithmes, la recherche d'un objet dans une image se ramène souvent à la recherche d'un contour ayant une certaine forme.

Pour voir le contour, il faut une différence (gradient) de couleurs.

Si, dans une image, un objet A touche un objet B, c'est que des pixels de A touchent des pixels de B. Or il y a alors de fortes chances pour que la couleur des pixels de A soit différente de la couleur des pixels de B. Ainsi, les contours semblent intuitivement inclus dans les zones qui présentent de fortes disparités (gradients) de couleur.

On voit donc l'intérêt de rechercher dans les images les zones qui présentent de fortes disparités de couleurs, puisqu'elles contiennent les contours. Ces zones peuvent être obtenues par des opérations simples et efficaces sur l'image (voir figure 1).

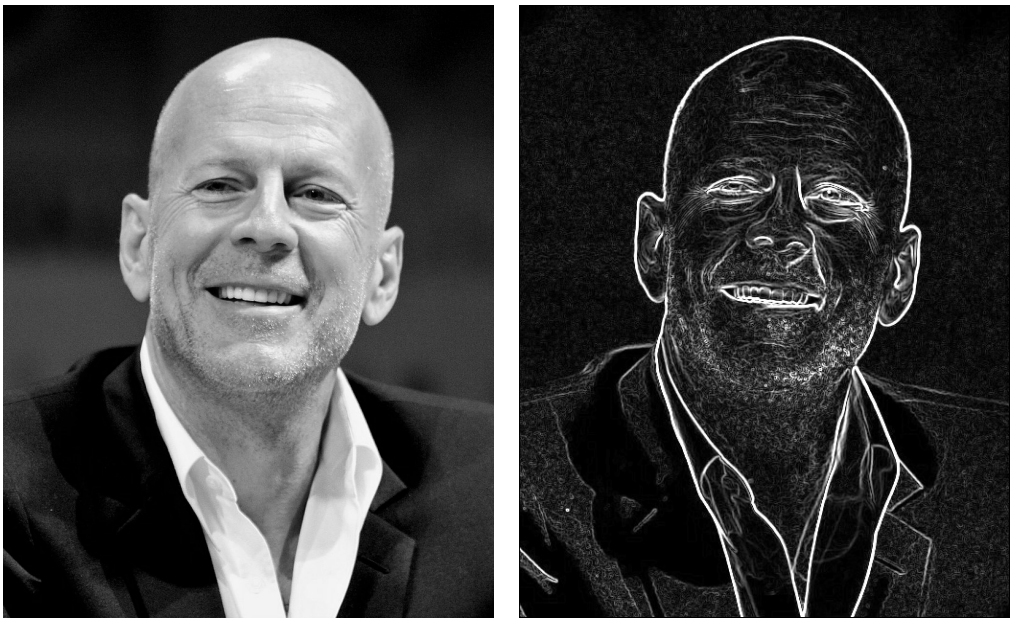

image originale zones à fortes disparités

Figure 1. Illustration de l'intérêt de rechercher les zones à forte disparité. On constate que les contours corrèlent relativement bien aux zones à fortes disparités.

4.1. **Implantez le filtre de Sobel pour estimer le gradient (*).** Le filtre de Sobel est classiquement utilisé pour estimer la valeur de la disparité des intensités. En assimilant l'image à une fonction à deux variables, ce filtre peut s'interpréter comme des calculs d'une version discrète de la dérivée. À vous de réfléchir pourquoi la dérivée est forte lors d'un brusque saut de valeur, comme lorsque l'on traverse un contour.

Pour ce qui nous concerne, ce filtre consiste juste à calculer des différences au niveau du voisinage (horizontal et vertical) de chaque pixel.

L'intensité des différences d'intensité horizontales au niveau du pixel `(i,j)` dans l'image `img` est donnée par :

```
   img[i-1][j-1] + 2*img[i][j-1] + img[i+1][j-1]
 - img[i-1][j+1] - 2*img[i][j+1] - img[i+1][j+1]
```

L'intensité des différences d'intensité verticales est de même donnée par :

```
          img[i-1][j-1] + 2*img[i-1][j] + img[i-1][j+1]
        - img[i+1][j-1] - 2*img[i+1][j] - img[i+1][j+1]
```

L'intensité totale est la norme des intensités horizontale et verticale (voir le rappel plus loin).

4.1.1. *Implantation (*).* Dans `sobel-tout-en-un.cpp`, copier les fonctions `lirePGM` et `ecrirePGM` qui seront utilisées pour les tests.

Ensuite, implanter dans `sobel-tout-en-un.cpp` les trois fonctions suivantes :

```
/** filtre de Sobel horizontal
 * @param img une image en teintes de gris
 * @return une image en teintes de gris de l'intensite horizontale de img
 **/
ImageGris intensiteH(ImageGris img) {
```

```
/** filtre de Sobel vertical
 * @param img une image en teintes de gris
 * @return une image en teintes de gris de l'intensite verticale de img
 **/
ImageGris intensiteV(ImageGris img) {
```

```
/** filtre de Sobel horizontal
 * @param img une image en teintes de gris
 * @return une image en teintes de gris de l'intensite horizontale de img
 **/
ImageGris intensiteH(ImageGris img) {
```

qui prennent une image en entrée et renvoient une nouvelle image de même taille, dans laquelle chaque pixel a pour valeur respectivement l'intensité horizontale, verticale et totale au niveau du pixel dans l'image d'origine (une pseudo image en fait, car ces intensités peuvent être négatives).

Attention : proposez et programmez une façon de traiter spécifiquement les bords mais n'oubliez pas qu'il n'est pas possible de lire une valeur en dehors de l'image !

Indication : `intensiteH` et `intensiteV` sont là pour vous aider, il n'est pas obligatoire de s'en servir pour faire `intensite`. On rappelle que la norme d'un vecteur (h, v) est $\sqrt{h^2 + v^2}$. La fonction `sqrt` permet de calculer la racine.

N'oubliez pas de compiler et d'exécuter le fichier `sobel-tout-en-un.cpp` pour vérifier que vos fonctions sont bien implantées. Pensez à vérifier que vos images sont semblables à celles du dossier `sobel/correction`.

4.2. **Seuillage des gradients (*).** Copier dans `seuillage-tout-en-un.cpp` les fonctions `lirePGM` et `ecrirePGM` qui seront utilisées pour les tests et la fonction `intensite` qui sera utilisée pour le seuillage des gradients.

Implantez dans `seuillage-tout-en-un.cpp` le filtre :

```
/** Renormalize une image en teinte de gris, les ramenant dans l'intervalle [0,255]
 * @param img un image en teintes de gris
 * @return une image en teintes de gris
 **/
ImageGris renormalise(ImageGris img) {
```

L'idée est d'utiliser toutes les teintes possibles dans le cas où toutes les valeurs seraient comprises dans un petit intervalle $[0, v]$ avec $v \leq 255$.

Indication : Il peut être pertinent de chercher le max des valeurs.

4.3. **Aller plus loin (**).** Afin de diminuer l'influence du bruit dans l'intensité, il peut être intéressant de lisser l'image avant de calculer le filtre de Sobel. Une façon de faire est de remplacer la valeur de chaque pixel par la moyenne des quatre pixels voisins, ou des 8 voisins.

4.4. **Seuillage de l'intensité (**).** Vous allez remarquer que l'on détecte beaucoup de contours. Il serait bon de ne garder que ceux qui semblent correspondre aux contours des objets. Les pixels avec une forte réponse au filtre de Sobel corrèlent bien avec les pixels des contours. Implanter le filtre :

```
/** Filtre de seuillage
 * @param img
 * @param seuil un entier dans l'intervalle [0,255]
 * @return image en noir et blanc obtenue en remplacant la teinte de
 *  chaque pixel par
 *  - du blanc si teinte < seuil
 *  - du noir  si teinte > seuil
 **/
ImageGris seuillage(ImageGris img, int seuil) {
```

Utilisez `seuillageTest` pour tester vos résultats. Notamment, proposez plusieurs valeurs de seuil pour les images indiquées.

4.5. **Double seuillage de l'intensité du gradient (***).** Même en réglant manuellement le seuil, des pixels peuvent être indûment considérés comme des contours. Une façon de diminuer ce nombre d'erreurs consiste à appliquer un seuil très élevé pour extraire des graines puis d'appliquer un seuil plus faible mais de ne garder que les pixels connectés aux graines.

4.6. **Croissance du contour de 1 pas (***).** Implanter le filtre :

```
/** Filtre de double seuillage
 * @param imgIntensite image d'intensite
 * @param imgContour image codant un ensemble de pixels selectionnes
 * @param seuil un entier de l'intervalle [0,255]
 * @return une copie d'imgIntensite modifiee de sorte que:
 *  -si teinte > seuil et voisin d'un pixel de imgContour, alors pixel noir
 *  -sinon pixel blanc
 **/
ImageGris doubleSeuillage(ImageGris imgIntensite, ImageGris imgContour, int seuil)
```

qui prend en argument une image d'intensité `imgIntensite` et une image `imgContour`, qui code un ensemble de pixels sélectionnés (0 si sélectionné, 255 sinon) et qui renvoie une image dans laquelle les pixels sont à 0 si et seulement si d'une part ils ont une intensité supérieure à `seuil` et d'autre part ils sont voisins d'un pixel sélectionné dans `imgContour` (255 sinon).

4.7. **Croissance du contour de plusieurs pas et visualisation (**).** Implanter le filtre :

```
/** Filtre de double seuillage iteratif
 * @param imgIntensite image d'intensité
 * @param seuilFort un entier de l'intervalle [0,255]
 * @param seuilFaible un entier de l'intervalle [0,255]
 * @param nbAmeliorations un entier non negatif: le nombre d'itérations
 * @return le double seuillage de img
 **/
ImageGris doubleSeuillage(ImageGris imgIntensite, int seuilFort, int seuilFaible, int nbA
```

qui prend en argument une image `img` et qui renvoie une image dans laquelle un pixel est à 0 si et seulement si il a dans l'image une intensité supérieure à `seuilFort`, ou bien une intensité supérieure à `seuilFaible` tout en étant connecté à un pixel d'intensité supérieure à `seuilFort` par un chemin de taille inférieure à `nbAmelioration` (255 sinon).

Indication : Il suffit d'appliquer `nbAmelioration` fois la fonction de la section précédente.

Utiliser la fonction `doubleSeuillageTest` pour vérifier vos résultats et proposer des paramètres pour les exemples indiqués.

4.8. **Aller plus loin (***).** Dans `doubleSeuillage`, on parcourt l'image en entier pour chercher des pixels connectés aux pixels déjà sélectionnés. Il serait plus efficace de n'explorer que le voisinage des pixels déjà sélectionnés. De plus, il serait intéressant de continuer ce processus tant que de nouveaux pixels sont sélectionnés. Implémenter ces deux améliorations.

4.9. **Aller beaucoup plus loin.** Quelques références sur les techniques d'extraction de contours et son utilisation :

- `http://fr.wikipedia.org/wiki/Filtre_de_Sobel`
- `http://en.wikipedia.org/wiki/Sobel_operator`
- [**?**], un article référence sur le sujet l'extraction de contour
- ce domaine de recherche est encore très actif, on peut citer par exemple la très récente publication [**?**].

5. Vers une bibliothèque de traitement d'images

Jusqu'ici, nous avons implémenté plusieurs fonctionalités de traitement d'images. Néanmoins, nous avons dû dupliquer beaucoup de code d'un fichier à l'autre, comme par exemple les fonctions `lirePPM`. De plus les fichiers commencent à être longs, avec des fonctions main ayant plusieurs rôles simultanés (lancer les tests et faire une autre action).

Nous allons maintenant restructurer le code afin d'éviter ces inconvénients grâce à la compilation séparée. Il ne s'agit pas ici de de créer de nouvelles fonctions ; simplement d'organiser le code différemment.

5.1. **Types d'image (*).** Durant le projet plusieurs types pour stocker et manipuler des images ont été définis. Regrouper tous ces types dans le fichier `image.h` Par la suite, tous les modules qui auront besoin de ces types devront inclure `image.h`.

5.2. **Module `pgm` (*).** Plusieurs fonctions de base sur images PGM ont été implémentées et réutilisées dans plusieurs programmes, notamment pour les tests. Afin d'éviter cette duplication de code, nous allons regrouper ces fonctions dans le module pgm.

(1) Consulter le fichier `pgm.h` ; il a été prérempli avec les entêtes des fonctions.

(2) Compléter le fichier `pgm.cpp` avec vos fonctions `lirePGM`, `ecrirePGM`, `inversePGM`.

(3) Compléter le fichier `pgm-test.cpp` avec les tests de ces fonctions.

(4) Utiliser la compilation séparée pour produire `pgm-test` à partir de `pgm.cpp` et `pgm-test.cpp`.

(5) Exécuter `pgm-test`, et vérifier que les tests passent.

5.3. **Modules `sobel` et `seuillage` (*).** Procéder de même que dans la section précédente pour compléter les modules `sobel.cpp` et `seuillage.cpp`, avec leur fichier d'entête et leurs tests. Vérifier que ces tests passent.

5.4. **Automatisation avec `make` (*).** Compiler à la main devient vite fastidieux avec la compilation séparée, puisque à chaque fois on doit se souvenir de toutes les dépendantes : quels sont tous les fichiers qui doivent être compilés en même temps.

`make` est un des outils permettant d'automatiser le processus. Ouvrez le fichier `Makefile` avec un éditeur type `gedit` ; vous verrez que les dépendances sont indiquées. (Remarque : vous n'avez pas besoin d'éditer ce fichier).

Avec ce `Makefile`, pour compiler `pgm-test` (par exemple), vous n'avez plus qu'à taper la commande suivante dans le terminal :

```
make pgm-test
```

Pour compiler tous les programmes du projet, vous pouvez faire :

```
make all
```

Pour lancer tous les tests du projet, vous pouvez faire :

```
make tests
```

5.5. **Utilisation de la bilbiothèque (*).** Le programme `TIN.cpp` implante un petit « couteau suisse » de traitement d'image permettant d'appliquer tous les filtres que vous avez implanté dans la bibliothèque à des images. Par exemple :

```
./TIN -e entree.pgm sortie.pgm
```

applique le filtre de Sobel à l'image dans `entree.pgm`. Pour voir toutes les options :

```
./TIN -h
```

Pour utiliser la bibliotheque, vous devez au préalable la compiler en tapant la commande suivante dans le terminal :

```
make TIN
```

6. Abordons maintenant les images en couleurs

6.1. **Description du format PPM.** La page `http://fr.wikipedia.org/wiki/Portable_pixmap` détaille le format général d'un fichier PPM non compressé. L'entête est de la forme :
`P3`
nbColonne nbLigne (par exemple `512 512`)
`255`
suivi d'un certain nombre de lignes décrivant les pixels de haut en bas et de gauche à droite (**Attention :** les lignes de l'image ne sont pas forcément les mêmes que celles du fichier !) ayant le format suivant :
rouge vert bleu rouge vert bleu ...

Ce format est très similaire au format PGM ; les commentaires faits précédemment s'appliquent toujours.

6.2. **La structure `Couleur` et le type `Image`.** Tout comme pour les images en noir et blanc et pour les images en niveau de gris, les images en couleur seront stockées dans un double tableau.

Tout d'abord, on définit la structure suivante :

```
/** Structure de donnees pour representer un pixel en couleur **/
struct Couleur {
    /** Intensite de rouge **/
    double r;
    /** Intensite de vert **/
    double g;
    /** Intensite de bleu **/
    double b;
};
```

Une fois la structure ainsi définie, les lignes suivantes permettront par exemple de définir une variable de type `Couleur` et de l'initialiser à un bleu cyan :

```
Couleur cyan;
cyan.r = 0;
cyan.g = 255;
cyan.b = 255;
```

Chaque pixel sera représenté par une variable de type `Couleur`. Une `Image` est donc un double tableau de `Couleur` :

```
/** Structure de donnees pour representer une image **/
typedef vector<vector<Couleur> > Image;
```

Pour obtenir le niveau de bleu du pixel le plus en haut à gauche d'une image convenablement lue et stockée dans une variable `Image1` de type `Image`, il suffira donc de taper `Image1[0][0].b`.

6.3. **Lecture et écricture d'image PGM (**).** Compléter le fichier `ppm.cpp` avec les fonctions suivantes :

```
/** Lit une image au format PPM, retourne un tableau de Couleur
 * @param source vers une image .ppm
 * @return une image
 **/
Image lirePPM(string source);

/**  Écrit une image dans un fichier PPM
 * @param img une image
 * @param cible le nom d'un fichier PPM
 **/
void  ecrirePPM(Image img, string cible);
```

Consulter le programme `ppm-test.cpp`, puis lancez le pour vérifier vos fonctions. Pour la compilation, procéder comme en Section 5.

6.4. **De la couleur au niveau de gris (**).** La méthode qui semble faire consensus pour transformer une image couleur en image en niveaux de gris consiste à remplacer la couleur de chaque pixel, codée généralement par 3 couleurs r, g, b, par le niveau de gris $0.2126 * r + 0.7152 * g + 0.0722 * b$. Il s'agit donc d'une simple moyenne pondérée des trois couleurs pour

obtenir un niveau de gris. Dans l'autre sens, étant donné un niveau de gris `c`, on l'associe souvent à la couleur `(c,c,c)` c'est à dire `r==c,g==c,b==c`.

Compléter `gris-couleurs.cpp` avec les fonctions suivantes :

```
/** Transforme une image couleur en une image en teintes de gris
 * @param img une image
 * @return une image en teintes de gris
 **/
ImageGris CouleurAuGris(Image img);

/** Transforme une image en teintes de gris en une image en couleurs (mais grise)
 * @param img une image en teintes de gris
 * @return une image
 **/
Image GrisACouleur(ImageGris img);
```

Lancer les tests comme précédemment.

6.5. **Conversion image couleur vers gris avec des logiciels.** Il est possible de transformer toute image au format `PNG`, `BMP` ou `JPG` en `PGM` avec les logiciels appropriés.

L'archive fournie contient les images de test suivantes dans le sous-répertoire `images/` :

```
Baboon.512.jpg   Billes.256.jpg  Embryos.512.jpg  House.256.jpg    Lena.512.jpg W
```

Convertissez toutes ces images au format PGM *non compressé*. Pour cela, vous pouvez par exemple utiliser un logiciel comme GIMP (linux/windows/macos) ou irfanview (Windows). Avec GIMP, chercher dans les menus :

```
Fichier -> Exporter vers -> mon_fond_ecran.pgm -> Format ASCII
```

Comme il y a un certain nombre d'images, cette tâche est un peu répétitive. Pour ceux qui sont à l'aise avec l'utilisation du terminal, il est plus efficace d'utiliser le logiciel *imageMagick* (téléchargeable gratuitement sous *linux* et *windows*). Par exemple, sous *linux*, pour transformer le fichier `Info111/Projet/images/mon_fond_ecran.jpg`, il suffit d'ouvrir un terminal, d'aller dans le répertoire contenant l'image :

```
cd Info111/Projet/images
```

et de taper la commande suivante (sur une seule ligne) :

```
convert -compress None mon_fond_ecran.jpg  mon_fond_ecran.pgm
```

ce qui produit le fichier `mon_fond_ecran.pgm`.

7. Segmentation en régions par la méthode du super pixel

Une des difficulté dans le traitement d'images est que le nombre de pixels est important : même pour une petite image de 512x512, il y a déjà 262144 pixels. Un autre problème est que chaque pixel porte individuellement très peu d'information. Une technique très utilisée pour réduire les effets de ces deux problèmes est la technique de segmentation en régions par le principe des *super pixels*, qui permet de regrouper les pixels en zones homogènes à la fois d'un point de vue spatial et colorimétrique. Cela aboutit ainsi à diminuer le nombre de régions tout en regroupant l'information au sein d'une région. Pour cela, il est primordial que les régions soient homogènes, ce qui fait par la même occasion ressortir les contours (voir figure 2). L'image peut alors être traitée avec des méthodes classiques provenant par exemple de traitements de texte qui ne pourraient pas marcher sur l'ensemble des pixels.

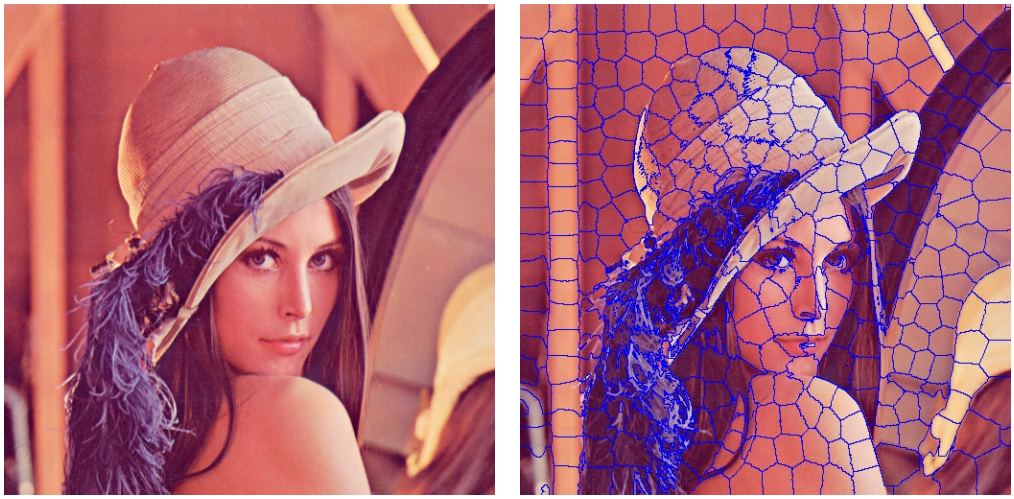

image de départ
262144 pixels

super pixels obtenus
400 régions

FIGURE 2. Illustration de la technique des super pixels

7.1. **L'algorithme des K-moyennes (***).** Le cœur de la méthode des super pixels consiste à former des sous-ensembles *compacts* dans un ensemble de points. Il existe différents algorithmes pour effectuer de tels regroupements. Le plus utilisé est celui des K-moyennes car il est rapide et généralement suffisamment performant.

L'entrée de cet algorithme est un ensemble de points `P` de $\mathbb{R}^D$ qu'on veut décomposer ainsi qu'un ensemble de points `C` (de $\mathbb{R}^D$ aussi) qui serviront de pilote à la décomposition. Pour notre application au traitement d'image, $\mathbb{R}^D$ sera l'espace spacio-colorimétrique (voir plus loin) avec $D = 5$.

L'algorithme consiste à effectuer *plusieurs fois* une amélioration. Chaque amélioration est en deux phases :

(1) Chaque point est associé au point pilote le plus proche ;
(2) Chaque point pilote est déplacé au barycentre de l'ensemble des points qui lui sont associés.

L'algorithme renvoie les positions finales des points pilotes. Dans cet algorithme, chaque répétition des phases 1 et 2 permet de compacter les sous ensembles.

Pour écrire cet algorithme, deux types sont définis dans `superpixel.h` :

```
/** Structure de donnee representant un point dans l'espace
    spacio colorimetrique **/
typedef vector<double> Point;

/** Structure de donnee representant un ensemble de points dans l'espace
    spacio colorimetrique **/
typedef vector<Point> EnsemblePoints;
```

Comme précédemment, des tests sont fournis dans `superpixel-test.cpp`. Lancez-les au fur et à mesure.

7.1.1. *Distance et plus proche voisin entre pixels (*).* Implanter dans `superpixel.cpp` la fonction :

```
/** Renvoie la distance Euclidienne entre deux points
```

```
 * @param p un point
 * @param c un point
 * @return la distance entre p et c
 **/
double distancePoints(Point p, Point c);
```

Implanter une fonction qui calcule la distance entre un point p et un ensemble de points C, c'est à dire le minimum des distances entre p et un point c de C :

```
/** Renvoie la distance Euclidienne d'un point a un ensemble de points
 * @param p un point
 * @param C un ensemble de points
 * @return la distance
 **/
double distanceAEnsemble(Point p, EnsemblePoints C);
```

Implanter la fonction :

```
/** Renvoie le plus proche voisin d'un point p dans un ensemble C
 * @param p un point
 * @param C un ensemble de points
 * @return l'index du plus proche voisin
 **/
int plusProcheVoisin(Point p, EnsemblePoints C);
```

Indication : On pourra utiliser la fonction `distanceAEnsemble`.

7.1.2. *Sous ensemble par regroupement de pixels (*).* Implanter la fonction :

```
/** Renvoie les points p de P tels que C[k] est le plus proche voisin de p dans C
 * @param P un ensemble de points
 * @param C un ensemble de points
 * @param k un entier
 * @return un sous ensemble des points de P
 **/
EnsemblePoints sousEnsemble(EnsemblePoints P, EnsemblePoints C,int k);
```

Indication : il suffit de parcourir les points `p` de `P` avec une boucle `for` et d'utiliser l'opération `push_back` dans le cas où `k==plusProcheVoisin(p,C)` pour mettre `p` dans l'ensemble qui sera renvoyé.

7.1.3. *Barycentre d'un ensemble de pixels (*).* Implanter la fonction :

```
/** Renvoie le barycentre d'unn ensemble de points
 * @param Q un ensemble de points
 * @return c le barycentre de Q
 **/
Point barycentre(EnsemblePoints Q);
```

Indication : Chaque coordonnée du barycentre est la moyenne des coordonnées correspondantes des points de `C`.

7.1.4. *Combinons toutes ces fonctions (***).* En utilisant les fonctions définies précédemment, on peut écrire l'algorithme K-moyenne. Pour simplifier, l'utilisateur de la fonction devra spécifier le nombre de fois que l'opération d'amélioration doit être effectuée. De même, dans le cas où un point de `C` ne serait associé à aucun point de `P` à la fin d'une étape de regroupement, alors ce point est laissé à sa position au lieu d'être déplacé. Implanter la fonction :

```
/** Renvoie la K-moyenne de deux ensembles de points
 * @param P un ensemble de points
 * @param C un ensemble de points
 * @param nbAmeliorations:entier le nombre de fois ou l'amelioration va etre effectuee
 * @return C un ensemble de points les positions finales de points pilotes
 **/
EnsemblePoints KMoyenne(EnsemblePoints P, EnsemblePoints C, int nbAmeliorations);
```

La fonction que vous allez écrire est trop lente pour la suite, principalement à cause des copies des vecteurs entre les différents appels de fonction. Aussi on fournit une fonction

```
/** Implantation optimisee de K-moyenne
 * @param P un ensemble de points
 * @param C un ensemble de points
 * @param nbAmeliorations:entier le nombre de fois ou l'amelioration va etre effectuee
 * @return C un ensemble de points les positions finales de points pilotes
 **/
EnsemblePoints FAST_KMoyenne(EnsemblePoints P, EnsemblePoints C, int nbAmeliorations);
```

qui fait la même chose en un seul tenant et dont vous ne devez pas vous inspirer.

Utilisez la fonction `KMoyenneTest` pour vérifier que les deux fonctions fournissent quand même bien un unique résultat (à quelques centièmes près).

7.1.5. *Aller plus loin (**).* Utilisez la bibliothèque SFML vue en TP pour afficher les différents points ainsi que les différents sous-ensembles au cours de l'algorithme. Par exemple, sur l'exemple de test, coloriez en cyan, magenta ou jaune les points selon qu'ils appartiennent au premier sous ensemble, au deuxième ou au troisième. De plus, changez `KMoyenne` pour continuer les améliorations tant que les ensembles évoluent.

7.2. **Application au regroupement de pixels (**).** Chaque pixel d'une image a une couleur ; ainsi, une image `img` est un tableau 2D, et le pixel `(i,j)` a la couleur `(r,g,b)` car `img[i][j]==(r,g,b)`. On peut voir les choses différemment en considérant le **point** en dimension 5 `(i,j,r,g,b)` ! De cette façon, une image n'est plus qu'un ensemble de points dans $\mathbb{R}^5$! Bien sûr cette façon de voir oublie la structure de l'image. Mais cela invite à utiliser la technique des K-moyennes pour regrouper des pixels en sous ensembles !

Dans le point (i, j, r, g, b), on mélange naïvement couleur et espace. Or, la distance euclidienne standard de $\mathbb{R}^5$ va être utilisée et elle agit de la même façon sur les cinq dimensions, ce qui semble inadapté pour des données n'ayant rien à voir les unes avec les autres.

Aussi, on formera plutôt $(i, j, \lambda r, \lambda g, \lambda b)$, ce qui permet de privilégier soit l'aspect spatial (λ petit) soit l'aspect colorimétrique (λ grand). Le cas extrême $\lambda = 0$ correspond à oublier la couleur : appliquer les K-moyennes créera des cercles dans l'image. Le cas extrême $\lambda = \infty$ correspond à oublier les positions : appliquer les K-moyennes regroupera les couleurs indépendamment de leur position.

7.2.1. *Quels points pivots ? (**).* Avant d'appliquer les K-moyennes, il faut définir les points pivots. L'objectif étant de décomposer l'image en régions compactes en couleur et en espace, il est classique de prendre comme points pivots les points de l'image correspondant à une grille spatiale. Implanter la fonction :

```
/** Renvoie un ensemble de points (espace spatio colorimetrique)
 *  regulierement espaces dans une image
 * @param img une image
 * @param lambda un double
 * @param mu un entier
 * @return un ensemble de points dans l'espace spatio colorimetrique
 **/
EnsemblePoints pivotSuperPixel (Image img, double lambda, int mu);
```

qui renvoie l'ensemble des points `mu`×`a`,`mu`×`b`,`lambda`×`img[mu`×`a][mu`×`b]` où `a` et `b` sont des entiers valant 0,1,2,... jusqu'à ce que l'on sorte de l'image.

Indications :

— La commande `push_back` peut être utile.

— Cette fonction est simple : il s'agit simplement d'une boucle `for` dans une boucle `for` comme pour parcourir l'image, sauf qu'au lieu de se déplacer de 1 on se déplace de `mu`.

7.2.2. *Les super pixels (**).* Implanter la fonction :

```
/** Renvoie les superpixels d'une image dans l'espace spatio colorimetrique
 * @param img une image en teintes de gris
 * @param lambda un double
 * @param mu un entier
 * @param nbAmeliorations un entier
 * @return un ensemble de points, les superpixels
 **/
EnsemblePoints superPixels(Image img, double lambda, int mu, int nbAmeliorations);
```

qui applique l'algorithme des K-moyennes (en faisant `nbAmeliorations` améliorations) sur les points correspondant à l'image (avec le coefficient λ) et avec les points pivots correspondant à la grille μ.

7.2.3. *Visualisation de super pixels (***).* Cette fonction construit les super pixels ; cependant à ce stade, on peut difficilement contrôler le résultat. Afin de visualiser le résultat, on va produire une nouvelle image dans laquelle tous les pixels associés à un pixel pivot prendront sa couleur. Cela permettra d'observer les super pixels. Écrivez une fonction

```
/** Renvoie les superpixels d'une image dans l'espace spatio colorimetrique
 * @param img une image en teintes de gris
 * @param lambda un double
 * @param mu un entier
 * @param nbAmeliorations un entier
 * @return un ensemble de points, les superpixels
 **/
EnsemblePoints superPixels(Image img, double lambda, int mu, int nbAmeliorations);
```

qui renvoie une image dans laquelle la couleur de chaque pixel est remplacée par celle de son pixel pivot.

Utilisez la fonction `superPixelTest` pour vérifier vos résultats et proposez des paramètres pour les exemples indiqués.

7.2.4. *Visualisation des bords des sous ensembles (**).* Rajoutez un bord de 1 pixel bleu à la frontière de chaque super pixel, comme dans la figure 2, afin d'améliorer la visualisation dans la fonction précédente.

7.2.5. *Pour aller beaucoup plus loin.* Quelques références sur les techniques des K-moyennes, des super pixels et leur utilisation :

- `http://fr.wikipedia.org/wiki/Algorithme_des_k-moyennes`
- `http://en.wikipedia.org/wiki/K-means`
- `http://fr.wikipedia.org/wiki/Segmentation_d'image`
- `http://en.wikipedia.org/wiki/Image_segmentation`
- un article qui fait référence sur l'utilisation de l'algorithme des K-moyennes [**?**]
- ce domaine de recherche est encore très actif, on peut citer par exemple la récente publication [**?**].