Projet : traitement d'image

1. Prélude

Ce projet conclue le module Info 111. Il est prévu que vous y consacriez chacun une vingtaine d'heures de travail intense, dont quatre en TP et le reste en autonomie. Votre travail sera évalué lors de la dernière séance de TP (semaine du 8 décembre) sous la forme d'une soutenance orale individuelle composée de quatre minutes de présentation 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, de votre maîtrise de l'ensemble du projet.

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 des questions adapté. 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 fur des premières questions vous fera paraître les questions ultérieures plus simples. Et vous aurez un vrai sentiment d'accomplissement en progressant dans le sujet.

Accrochez vous, et demandez de l'aide aux autres étudiants (sans copier leur code bien évidemment, cela n'aurait aucun intérêt) ainsi qu'à vos chargés de TD.

Table des matières

2. Introduction

Ce projet comporte quatre parties :
- 3 : Prérequis : les notions basiques d'images
- 4 : Extraction de contours
- 5 : Images en couleurs
- 6 : Super pixel

Les deux parties principales sont 4 et 6. Elles dépendent toutes deux de 3 ; la partie 6 dépend en plus de 5. À l'intérieur d'une partie les questions se suivent (sauf entre les sections 6.1 et 6.2 où une fonction est fournie pour continuer). En dehors de ces dépendances, vous pouvez travailler sur ce projet dans l'ordre que vous souhaiterez.

Chaque partie porte sur une technique basique 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.

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.

Une archive est fournie sur le web, contenant :
- Une proposition de squelette des fichiers, avec de la documentation et des tests ;
- La documentation compilée (voir le fichier `html/files.html`) ;
- Des exemples d'images et des résultats de traitements par les différents filtres pour comparer avec les vôtres.

3. Prérequis : les notions basiques d'images

3.1. **Définition.** Une image (en teinte de gris) est classiquement représentée par un tableau à deux dimensions dans lequel chaque case (appelée aussi pixel) contient une une teinte de gris. Généralement, cette teinte de gris est représentée par un entier, typiquement entre 0 et 255, où 0 représente un pixel noir et 255 un pixel blanc. Cependant il sera commode d'utiliser un `double` pour ne pas rencontrer de problème de conversion lors des opérations que l'on effectuera sur les images. Aussi, pour représenter des images en teintes de gris, le type suivant est défini dans image.h :

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

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).

Dans la deuxième partie du projet nous considérerons, c'est plus joli, des images en couleurs. Classiquement, une couleur se code par trois nombres correspondant respectivement à une intensité de rouge, de vert et de bleu. Aussi, l'entête image.h défini les types suivants pour représenter respectivement les couleurs et les images en couleurs :

```
/** Structure de donnée pour représenter un pixel en couleur **/
struct Couleur {
    /** Intensité de rouge **/
    double r;
    /** Intensité de vert **/
```

```
    double g;
    /** Intensité de bleu **/
    double b;
};
```

```
/** Structure de donnée pour représenter une image **/
typedef vector<vector<Couleur> > Image;
```

3.2. **Le format PGM non compressé (-).**

3.2.1. *Conversion depuis d'autres formats.* Dans la première partie de ce projet, on se contentera de lire et d'écrire des images en teintes de gris au format `PGM` non compressé. Cependant il est possible de transformer toute image au format `PNG`, `BMP` ou `JPG` en `PGM` à l'aide du 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.

Remarque : Paint ne permet que de créer des images au format PGM compressé. En revanche, avec une version récente de GIMP, on peut utiliser :

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

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

```
Aerial.512.jpg   Baboon.512.jpg   Billes.256.jpg   Boat.512.jpg
Embryos.512.jpg  F16.512.jpg      House.256.jpg    Lena.512.jpg
Monarch.512.jpg  Phoenix.512.jpg  Poivron.512.jpg  Sailboat.512.jpg
TeaPot.512.jpg   Tulips.512.jpg
```

Convertissez toutes ces images au format PGM non compressé.

3.2.2. *Description du format PGM.* 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 fichier PGM que vous avez produits précédemment avec un éditeur de texte comme *notepad* ou *gedit* 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 ligne commençant par un '#'.

3.2.3. *Lecture (-).* Implantez la fonction lirePGM du fichier pgm.cpp (voir aussi pgm.h) :

```
/** 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);
```

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

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

3.2.4. *Écriture (-).* Implantez la fonction :

```
/** 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 pareil pour l'ordinateur).

3.3. **Tests.** Une batterie de tests est fournie dans le fichier tests.cpp. Par défaut, le programme lance−tests.cpp lance tous les tests. Pour l'instant on ne veut lancer que la fonction testLireEcrirePGM pour tester la lecture et l'écriture de fichier PGM. Mettez tous les autres tests en commentaire dans la fonction main. Puis chargez dans Code : :Blocks le projet lance−test.cbp, compilez-le et le lancez-le pour vérifier le comportement de vos fonctions.

Par la suite, vous activerez au fur et à mesure les tests relevants à la partie sur laquelle vous travaillerez. À noter qu'une partie seulement des tests est automatique : le programme lance−test affichera à l'écran les images que vous devrez comparer visuellement.

3.4. **Autres programmes du projet.** Le projet lance−test.cbp contient deux autres programmes. Pour les compiler, chargez les projets Code : :Blocks correspondant.

- renormalise.cpp : ce programme illustre comment on peut, à partir d'une bibliothèque de fonctions, construire une multitude de petits programmes dédiés à des fonctionnalités spécifiques. Ici, le programme s'utilise comme suit :

  ```
      renormalise entree.pgm sortie.pgm
  ```

  qui charge l'image entree.pgm, applique le filtre de normalisation et sauvegarde le résultat sous le nom sortie.pgm.
- main.cpp : ce programme implante un petit « couteau suisse » de traitement d'image permettant d'appliquer les différents filtres de la bibliothèque à des fichiers. Par exemple :

  ```
      main -e entree.pgm sortie.pgm
  ```

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

```
main -h
```

4. Extraction de contour

Les contours sont une information privilégiée en traitement d'image. 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. Cela est justifié par la stabilité relative du contour vis à vis de la couleur. Intuitivement, un système basé sur le contour reconnaîtra un piéton qu'il soit en noir sur un fond gris ou qu'il ait un gilet jaune sur ce même fond gris alors qu'un système basé sur la couleur n'arrivera probablement qu'à reconnaître le piéton avec le gilet jaune. Autrement dit, pour voir le contour, il faut une différence de couleur mais pas la présence d'une couleur en particulier. Bien entendu, le contour a aussi ses défauts notamment d'être très variable pour les objets articulés, mais expérimentalement il véhicule plus d'informations que la couleur.

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 forte chance pour que la couleur des pixels de A soit différentes de la couleur des pixels de B. Ainsi les contours semblent intuitivement inclus dans les zones qui présentent de forte disparité de couleur.

On voit donc l'intérêt de rechercher dans les images les zones qui présentent de forte disparité de couleur 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. **Filtre de Sobel.** Le filtre de Sobel est classiquement utilisé pour estimer l'intensité de la disparité des intensités. En assimilant l'image à une fonction à deux variables, ce filtre peut s'interpréter comme des calculs de version discrètes de la dérivée seconde. À vous de réfléchir pourquoi la dérivée seconde 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 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-1][j] + img[i-1][j+1]
  - img[i+1][j-1] - 2*img[i+1][j] - 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][j-1] + img[i+1][j-1]
  - img[i-1][j+1] - 2*img[i][j+1] - img[i+1][j+1]
```

L'intensité totale est la norme des intensités horizontale et verticale.

4.1.1. *Implantation (-).* Implanter dans sobel.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'intensité 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'intensité verticale de img
 **/
ImageGris intensiteV(ImageGris img);

/** filtre de Sobel
 * @param img une image en teintes de gris
 * @return une image en teintes de gris de l'intensité de img
 **/
ImageGris intensite(ImageGris img);
```

qui prennent une image en entrée et retourne 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 (précisément la valeur absolue pour les intensités horizontales et verticales car elles peuvent être négatives).

Attention : proposer et programmer 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. Utiliser fabs pour obtenir la valeur absolue d'une valeur.

4.1.2. *Visualisation (-).* Implantez dans seuillage.cpp le filtre :

```
/** Renormalize une image en teintes de gris en 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.

4.1.3. *Tests.* Utilisez la fonction testSobel pour tester vos résultats. Pensez aussi à regarder les images en H et V si les résultats ne correspondent pas.

4.1.4. *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.

4.2. **Seuillage de l'intensité (=).** 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 une image en noir et blanc obtenue en remplacant la teinte de chaque pixel pa
 *  - du blanc si teinte < seuil
 *  - du noir  si teinte > seuil
 **/
ImageGris seuillage(ImageGris img, int seuil);
```

Utiliser testSeuillage pour tester vos résultats. Notamment proposer des seuils pour les images indiquées.

4.3. **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.

4.3.1. *Croissance du contour de 1 (+).* Implanter le filtre :

```
/** Filtre de double seuillage
 * @param imgIntensite image d'intensité
 * @param imgContour image codant un ensemble de pixels selectionnés
 * @param seuil un entier de l'intervalle [0,255]
 * @return une copie d'imgIntensite modifiée 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 si d'autre part ils sont voisins d'un pixel sélectionné dans imgContour (255 sinon).

4.3.2. *Croissance du contour et visualisation (=).* Implanter le filtre :

```
/** Filtre de double seuillage itératif
 * @param img une image *.ppm en teintes de gris
 * @param seuilFort un entier de l'intervalle [0,255]
 * @param seuilFaible un entier de l'intervalle [0,255]
 * @param nbAmeliorations un entier non négatif : le nombre d'itérations
 * @return le double seuillage de img
 **/
ImageGris doubleSeuillage(ImageGris img, int seuilFort, int seuilFaible, int nbAmeliorat
```

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 supérieur à seuilFort ou une intensité supérieur à seuilFaible tout en étant connecté à un pixel d'intensité supérieur à seuilFort par un chemin de taille inférieur à nbAmelioration (255 sinon).

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

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

4.3.3. *Aller plus loin +(+).* Ici, 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 nouveau pixel sont sélectionnés. Implémenter ces deux améliorations.

4.3.4. *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`
- [4] un article qui fait référence en extraction de contour
- ce domaine de recherche est encore très actif, on peut citer par exemple la très récente publication [3]

5. Images en couleurs

5.0.5. *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, et les commentaires fait précédemment s'appliquent toujours.

5.1. **De la couleur au gris (=).** La méthode qui semble faire consensus pour transformer une image couleur en image en teintes de gris consiste à remplacer la couleur de chaque pixel codée généralement par 3 couleurs r, g, b par la teintes 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 teintes de gris. Dans l'autre sens, étant donnée une teintes de gris c, on l'associe souvent à la couleur (c,c,c) c'est à dire r==c,g==c,b==c.

Implanter dans gris.cpp 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);
```

Utilisez la fonction testCouleurAuGris pour tester vos résultats.

6. 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 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 classique provenant par exemple de traitement 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

6.1. **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 est associé.

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 donnée pour représenter un point dans l'espace spacio colorimétrique **
typedef vector<double> Point;

/** Structure de donnée pour représenter un ensemble de points dans l'espace spacio colo
typedef vector<Point> EnsPoint;
```

6.1.1. *Distance et plus proche voisin (-).* Commencer par implanter dans superpixel.cpp la fonction :

```
/** Renvoie la distance entre deux points
 * @param p un point
 * @param c un point
 * @return la distance entre p et c
 **/
double distance(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 à un ensemble de points
 * @param p un point
 * @param C un ensemble de points
 * @return la distance
 **/
double distanceAEnsemble(Point p, EnsPoint 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, EnsPoint C);
```

Indication : On pourra utiliser la fonction distanceAEnsemble.

6.1.2. *Sous ensemble (-).* 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
 **/
EnsPoint sousEnsemble(EnsPoint P, EnsPoint 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.

6.1.3. *Barycentre (-).* 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(EnsPoint Q);
```

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

6.1.4. *Combinons ces fonctions (+).* En utilisant les fonctions définies en 3.1.2 et 3.1.3, 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 un entier le nombre de fois où l'amélioration va être effectué
 * @return C un ensemble de points les positions finales de points pilotes
 **/
EnsPoint Kmoyenne(EnsPoint P, EnsPoint 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 optimisée de K-moyenne
 * @param P un ensemble de points
 * @param C un ensemble de points
 * @param nbAmeliorations un entier le nombre de fois où l'amélioration va être effectué
 * @return C un ensemble de points les positions finales de points pilotes
 **/
EnsPoint FAST_Kmoyenne(EnsPoint P, EnsPoint C, int nbAmeliorations);
```

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

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

6.1.5. *Aller plus loin (=).* Utiliser la bibliothèque SDL 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, colorier en cyan, magenta ou jaune les points selon qu'ils appartiennent au premier sous ensemble, au deuxième ou au troisième. De plus, changer Kmoyenne pour continuer les améliorations tant que les ensembles évoluent.

6.2. **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 sur 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 $\mathbb{R}^3$ va être utilisée et elle agit de la même façon sur les cinq dimensions.

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.

6.2.1. *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 dans l'espace spatio colorimétrique regulièrement espa
 * @param img une image
 * @param lambda un double
 * @param mu un entier
 * @return un ensemble de points dans l'espace spatio colorimétrique
 **/
EnsPoint 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.

6.2.2. *super pixel (=).* Implanter la fonction :

```
/** Renvoie les superpixels d'une image dans l'espace spatio colorimétrique
 * @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
 **/
EnsPoint 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.

6.2.3. *Visualisation (+).* 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

```
/** Filtre SuperPixel
 * @param img une image
 * @param lambda un double
 * @param mu un entier
 * @param nbAmeliorations
 * @return l'image associée aux superpixels d'une image
 **/
Image superPixel(Image img, double lambda, int mu, int nbAmeliorations);
```

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

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

6.2.4. *Aller plus loin (=).* 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.

6.2.5. *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 [2]
- ce domaine de recherche est encore très actif, on peut citer par exemple la récente publication [1]

Références

[1] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk. Slic superpixels. *École Polytechnique Fédéral de Lausssanne (EPFL), Tech. Rep*, 149300, 2010.

[2] Anil K Jain. Data clustering : 50 years beyond k-means. *Pattern Recognition Letters*, 31(8) :651–666, 2010.

[3] Jun Ma and Le Lu. Hierarchical segmentation and identification of thoracic vertebra using learning-based edge detection and coarse-to-fine deformable model. *Computer Vision and Image Understanding*, 2013.

[4] Pietro Perona and Jitendra Malik. Scale-space and edge detection using anisotropic diffusion. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 12(7) :629–639, 1990.