

Traitement d'images, C++, Master 2 SISEA

BOUSSOT Valentin

2 décembre 2020

Table des matières

1 Traitement d'images en C++	2
1.1 Objectifs	2
1.2 Modélisation du programme	3
1.3 Lire/Ecrire une image au format BMP	4
1.3.1 Lecture d'une image BMP	4
1.3.2 Ecriture d'une image BMP	5
1.4 Conversion de codage	6
1.5 Traitement sur des pixels :	7
1.5.1 Résultats	8
1.6 Traitement sur les positions des pixels de la matrice :	9
1.6.1 Résultats	10
1.7 Génération de bruits et filtrage d'image	11
1.8 Filtrage	12
1.8.1 Filtrage linéaire	12
1.8.2 Filtre Median	13
1.9 Surcharge des opérateurs de la classe Image	14
1.10 Mesure de similarité/dissimilitude	15
1.11 Segmentation	16
1.11.1 Méthode Otsu(Segmentation bimodale)	17
1.11.2 K-mean(Classification multi-classe)	18
1.11.3 Résultats	20
1.11.4 EM(Classification multi-classe)	21
1.11.5 Résultats	23
2 Critique Générale	24

1 Traitement d'images en C++

1.1 Objectifs

- TP1 : Lire/Ecrire
 - Une image au format BMP
- TP2 : Appliquer sur l'image :
 - Des traitements sur les pixels (Treshold, AjustContrast, Exchange, Mask, Invert, &)
 - Appliquer une matrice de transformation
 - Générer du bruit salt and pepper et gaussien
- TP3 :
 - Surcharger les opérateurs + - += -= ==
 - Appliquer des filtres linéaires
 - Appliquer un filtre médian
 - Mesure de similarité
 - Segmenter une image
 - Segmentation par classification(Bi et Multi-Classe)
 - Otsu -K Mean, -EM

1.2 Modélisation du programme

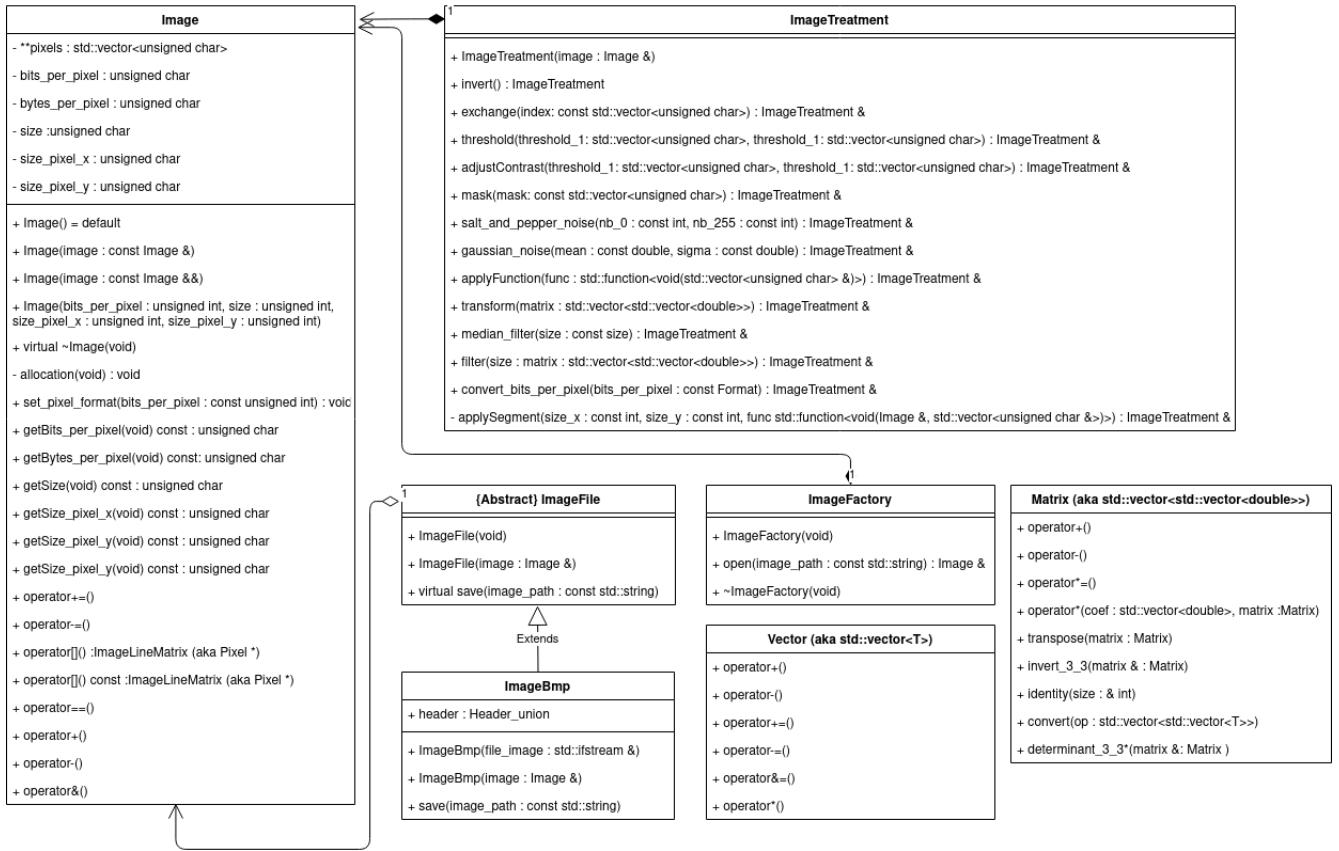


FIGURE 1 – Modélisation UML du programme

Commentaire :

La classe **ImageFactory** est chargée de détecter le format du fichier et de choisir la bonne classe fille de la classe abstraite **ImageFile** pour ouvrir l'image. Elle a aussi la responsabilité de gérer le cycle de vie de l'image pour éviter les recopies inutiles dans le programme.

La classe **ImageBmp** permet d'ouvrir une image au format BMP, elle permet également d'écrire une image sous ce format.

La classe **Image** est essentiellement un adaptateur, elle encapsule une matrice de pixels et donne des informations sur celle-ci, elle redéfinit également quelques opérateurs.

La classe **ImageTreatment** est un builder, elle possède une copie d'une image et applique en cascade des transformations sur celle-ci, elle renvoie la copie par la méthode **get()**.

Les fonctions pour la vérification des dimensions ne sont pas indiquées.

1.3 Lire/Ecrire une image au format BMP

Listing 1 – Structure de données de l'entête

```
1 #define LONGUEUR_ENTETE 52
2
3 union Value_2{
4     unsigned char data[2];
5     unsigned short value;
6 };
7
8 union Value_4{
9     unsigned char data[4];
10    unsigned int value;
11 };
12
13
14 struct Header_struct{
15     Value_4 size_file;
16     Value_2 reserved_1;
17     Value_2 reserved_2;
18     Value_4 offset;
19     Value_4 size_header;
20     Value_4 width;
21     Value_4 height;
22     Value_2 color_planes;
23     Value_2 bits_per_pixel;
24     Value_4 compression;
25     Value_4 size_image;
26     Value_4 pixel_per_meter_x;
27     Value_4 pixel_per_meter_y;
28     Value_4 number_color;
29     Value_4 important_color;
30 };
31
32 union Header_union{
33     unsigned char header[LONGUEUR_ENTETE];
34     Header_struct header_struct;
35 };
```

Une entête de fichier BMP est composé de champs de 2 tailles différentes, il suffit d'écrire l'entête du fichier dans le champ **header** de cette union.

Cette façon de faire n'est pas très portable, il faut que les données de type **int** soient de taille 4 et les données de type **short** de taille 2, en effet les unions prennent la taille maximale des deux types en mémoire.

Également les membres de la structure doivent être alignées en mémoire pour éviter des octets de padding (Le nombre magic n'est pas représenté dans la struct pour cette raison).

1.3.1 Lecture d'une image BMP

Listing 2 – Fonction pour la lecture d'une image BMP

```
1 ImageBmp::ImageBmp(std::ifstream &file_image) : ImageFile(){
2     file_image.read((char *)this->header.header, sizeof(this->header)); //Lecture de l'entête
3
4     int size_pixel_x = this->header.header_struct.width.value;
5     int size_pixel_y = this->header.header_struct.height.value;
6     // Instanciation d'une image avec les informations de l'header
7     this->image = new Image(this->header.header_struct.bits_per_pixel.value,
8                             size_pixel_x * size_pixel_y, size_pixel_x, size_pixel_y);
9     // Lecture des pixels
10    for(int x = 0; x < this->image->getSize_pixel_x(); x++)
11        for(int y = 0; y < this->image->getSize_pixel_y(); y++)
12            file_image.read((char *) (*this->image)[x][y].data(), this->image->getBytes_per_pixel());
13 }
```

Le type `std :: vector<unsigned char>` si initialisé nous garantie d'avoir une zone mémoire contigue accessible par la méthode `data()`.

Elle réserve également un certain nombre d'octets supplémentaire après cette zone dans le cas où l'on souhaite augmenter sa taille, ce nombre d'octets disponibles avant que le vecteur réalloue une autre zone mémoire est accessible par la méthode `capacity()`.

Le choix du conteneur `std :: vector` est motivé par le fait qu'on se réserve la possibilité de modifier le nombre d'octets par pixel pendant l'exécution. D'ailleurs on aurait pu modéliser entièrement notre matrice par ce type ce qui est d'usage en C++. Ce qui nous aurait fait l'économie de quelques lignes dans notre classe `Image` pour la gestion de la partie dynamique.

1.3.2 Ecriture d'une image BMP

En trois parties :

Listing 3 – Modification et écriture du header

```

1  /*-----EDIT HEADER-----*/
2  unsigned char id[2] = {0x42, 0x4d};
3  file_image.write((char *)id, 2); // Écriture du nombre magic correspondant au format BMP
4
5
6  this->header.header_struct.width.value = this->image->getSize_pixel_x();
7  this->header.header_struct.height.value = this->image->getSize_pixel_y();
8  this->header.header_struct.bits_per_pixel.value = this->image->getBits_per_pixel();
9  // Set la taille de l'image
10 this->header.header_struct.size_image.value = this->image->getSize()*this->image->getBytes_per_pixel();
11 this->header.header_struct.offset.value = LONGUEUR_ENTETE+2;
12
13 switch(this->image->getBits_per_pixel()){
14     case 1:
15         this->header.header_struct.number_color.value = 2;
16         break;
17     case 8:
18         this->header.header_struct.number_color.value = 256;
19         break;
20     default:
21         this->header.header_struct.number_color.value = 0;
22         break;
23 }
24 // Ajout de la taille de la color table
25 this->header.header_struct.offset.value += this->header.header_struct.number_color.value*4;
26 // Set la taille totale du fichier
27 this->header.header_struct.size_file.value = this->header.header_struct.offset.value
28 +this->header.header_struct.size_image.value;
29
30 file_image.write((char *)this->header.header, LONGUEUR_ENTETE);

```

Dans le cas où les pixels sont codés avec 1 bit(**BINARY**) ou avec 1 octet(**GRAY_SCALE**) il faut définir une table de couleur. Le format de cette table est une succession de 4 octets (**B, G, R, A**) qui est indexée par les pixels. Il faut prévoir cette taille supplémentaire dans la taille du fichier et dans le champs offset.

Listing 4 – Définition d'une color table

```

1  /*-----COLOR TABLE-----*/
2  std::vector<unsigned char> color_table;
3  switch(this->image->getBits_per_pixel()){
4      case 1: //Binary
5          color_table = {0,0,0,0,255,255,255,0};
6          file_image.write((char *) color_table.data(), color_table.size());
7          break;
8      case 8: //Gray Scale
9          for(unsigned int i = 0; i < 256; i++)
10              color_table.insert(color_table.end(), {(unsigned char)i, (unsigned char)i, (unsigned char)i, 0});
11
12          file_image.write((char *) color_table.data(), color_table.size());
13          break;
14 }

```

Pour 1 bit on prévoit 2 couleurs{0, 255} pour 8 bits une table en nuances de gris.

Listing 5 – Ecriture des pixels

```

1  /*-----WRITE PIXEL-----*/
2  std::vector<unsigned char> pixels_data;
3  switch(this->image->getBits_per_pixel()){
4      case 1:
5          pixels_data.push_back(0);
6          int u;
7          u = 0;
8          for(int x = 0; x < this->image->getSize_pixel_x(); x++){
9              for(int y = 0; y < this->image->getSize_pixel_y(); y++){
10                 if(++u == 8){
11                     pixels_data.push_back(0);
12                     u = 0;
13                 }
14                 pixels_data.back() = (pixels_data.back() << 1) | ((*this->image)[x][y][0] & 1);
15             }
16         }
17         file_image.write((char *) pixels_data.data(), pixels_data.size());
18         break;
19     default:
20         for(int x = 0; x < this->image->getSize_pixel_x(); x++)
21             for(int y = 0; y < this->image->getSize_pixel_y(); y++)
22                 file_image.write((char *) (*this->image)[x][y].data(), (*this->image).getBytes_per_pixel());
23         break;
24 }
```

En règle générale en écrit simplement à la suite le tableau de chaque pixel. Pour le cas du 1 bit il faut agréger 8 octets de pixel dans 1 seul.

Le programme fait abstraction de l'arrangement des canaux en mémoire.

1.4 Conversion de codage

Modification du codage des pixels :

Listing 6 – Conversion de codage des pixels

```

1 ImageTreatment & ImageTreatment::convert_bits_per_pixel(const Format bits_per_pixel){
2     if(this->image.getBits_per_pixel() != bits_per_pixel){
3         switch(bits_per_pixel){
4             case BINARY:
5                 this->convert_bits_per_pixel(GRAY_SCALE); //On s'assure que l'on est bien en gray_scale
6                 for(int x = 0; x < this->image.getSize_pixel_x(); x++)
7                     for(int y = 0; y < this->image.getSize_pixel_y(); y++)
8                         this->image[x][y][0] = this->image[x][y][0] > 128 ? 1 : 0;
9                     break;
10            case GRAY_SCALE:
11                int mean;
12                for(int x = 0; x < this->image.getSize_pixel_x(); x++)
13                    for(int y = 0; y < this->image.getSize_pixel_y(); y++){
14                        mean = std::accumulate(this->image[x][y].begin(), this->image.getBits_per_pixel() < 4 ? this->image[x][y].begin() : this->image[x][y].begin(), 0);
15                        this->image[x][y].clear();
16                        this->image[x][y].push_back(mean);
17                    }
18                break;
19            default:
20                break;
21        }
22        this->image.set_pixel_format(bits_per_pixel);
23    }
24    return (*this);
25 }
```

Dans le cas où l'on souhaite passer en **GRAY_SCALE** on place la moyenne de N canaux dans un seul octet, le canal 4 est réservé à la transparence il ne doit pas faire partie du calcul.

Pour passer en binaire, on seuil l'image à 128 et on place le résultat dans un octet. Normalement cette opération est faite après un seuillage en **GRAY_SCALE** qui à placé les pixels à 255 ou 0, dans le cas contraire l'opération n'a peu de sens.

Par la suite le type **Matrix** et **Pixel** sont des typedef de `std ::vector<std ::vector<double>` et `std ::vector<unsigned char>`.

On peut noter plusieurs types de traitement :

1.5 Traitement sur des pixels :

Exemple de fonctionnement :

Listing 7 – Traitement sur des pixels

```

1 ImageTreatment & ImageTreatment::invert(){
2     applyFunction([](Pixel & pixel) {pixel = Pixel(pixel.size(), 255)-pixel;});
3     return (*this);
4 }
5
6 ImageTreatment & ImageTreatment::applyFunction(std::function<void(Pixel &)> func){
7     for(int x = 0; x < this->image.getSize_pixel_x(); x++)
8         for(int y = 0; y < this->image.getSize_pixel_y(); y++)
9             func(this->image[x][y]);
10    return (*this);
11 }
```

`applyFunction(std :: function<void(Pixel &)>)` permet d'appeler une fonction lambda en argument sur chaque pixel.

Ces méthodes appliquent ce premier type de traitement :

- mask : Supprime un ou plusieurs canaux.
- invert : Inverse les couleurs de l'image.
- exchange : Permute les canaux.
- threshold : Binarise l'image en fonction de deux seuils (Ne change pas implicitement le codage des pixels).
- & : Redéfinition de l'opérateur pour produire une image masquée.
- adjustContrast : Modifie la dynamique de l'image de manière linéaire entre deux valeurs de seuil.

Cette dernière méthode montre l'avantage que l'on a de procéder ainsi, on ne ferme pas notre classe à l'ajout de fonctionnalité, si l'on souhaite modifier la dynamique de l'image de manière non linéaire ou avec d'autres paramètres comme dans ImageJ on pourra le faire à l'extérieur de notre classe.

Listing 8 – Application d'exemple de traitements sur l'image

```

1 int main(int argc, char **argv){
2
3     ImageFactory imageFactory;
4     try {
5         Image &image = imageFactory.open("Images/lena24.bmp");
6         std::string name;
7         ImageTreatment imageTreatment(image);
8
9         name = "lena_invert";
10        imageTreatment.invert();
11
12        //name = "Lena_RGB_to_GB";
13        //imageTreatment.mask({1,1,0});
14
15        //name = "Lena_RGB_to_BGR";
16        //imageTreatment.exchange({2,1,0});
17
18        //name = "lena_binary";
19        //imageTreatment.convert_bits_per_pixel(GRAY_SCALE).threshold({128}, {255}).convert_bits_per_pixel(BINARY);
20
21        //name = "lena_adjust_contrast";
22        //imageTreatment.convert_bits_per_pixel(GRAY_SCALE).adjustContrast({60}, {200});
23
24        //Image image_mask = image & imageFactory.open("Images/mask.bmp");
25        //ImageBmp(image_mask).save("Results/lena_mask.bmp")
26
27        ImageBmp(imageTreatment.get()).save("Results/" + name + ".bmp");
28    } catch(std::string const& error) {
29        std::cout << error << std::endl;
30    }
31    return EXIT_SUCCESS;
32 }
```

De manière générale les arguments des fonctions du programme sont vérifiés et renvoient une exception, ils sont capturés dans le main.

La classe **ImageFactory** gère une référence à une image elle doit dans tous les cas la libérer.

1.5.1 Résultats



FIGURE 2 – Exemple de traitement des valeurs des pixels

1.6 Traitement sur les positions des pixels de la matrice :

Listing 9 – Fonction Produit Scalaire

```

1 ImageTreatment & ImageTreatment::transform(Matrix matrix){
2     if(matrix.size() != 3 || matrix[0].size() !=3) throw std::string("Erreur : Matrice de transformation incorrecte !!!");
3     Image image = this->image;
4     Matrix init = {{this->image.getSize_pixel_x()/2.0, this->image.getSize_pixel_y()/2.0, 0}};
5     Matrix result;
6     Matrix X;
7     std::vector<double> min(3,0);
8     std::vector<double> max = std::vector<double>({this->image.getSize_pixel_x()-1.0, this->image.getSize_pixel_y()-1.0, 0});
9     for(int x = 0; x < this->image.getSize_pixel_x(); x++){
10         for(int y = 0; y < this->image.getSize_pixel_y(); y++){
11             X = {{(double)x, (double)y, 1}};
12             result = (X-init)*matrix+init;
13             this->image[x][y] = result[0] >= min[0] && result[0] <= max[0] ? image[(int)result[0][0]]((int)result[0][1]) : Pixel(this->image.getBytes_per_pixel(), 0);
14         }
15     }
16 }
17 return (*this);
18 }
```

La méthode **transform(Matrix)** prend en argument une matrice de transformation $M_{2,3}$ et l'applique aux positions des pixels de l'image. Pour que cela fonctionne, il faut bien évidemment redéfinir les opérateurs en jeu.

On souhaite pour la rotation conservée l'image centrée. Pour ce faire il faut changer l'origine. Le calcul n'est pas très compliqué mais demande de poser le problème.

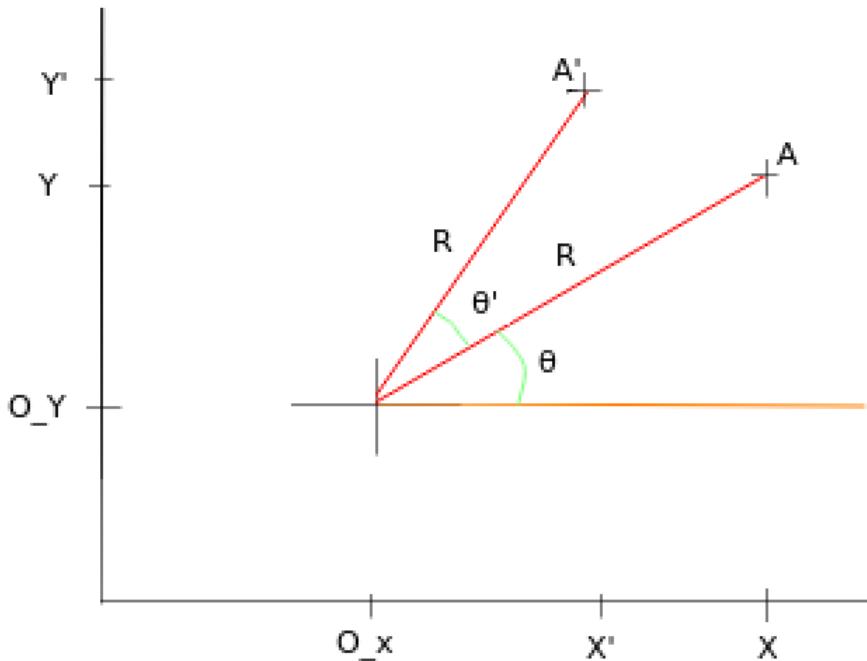


FIGURE 3 – Contexte/Notation du problème

On cherche à trouver x' , y' en fonction de θ' , x et y .

$$\begin{aligned}
 x &= R.\cos(\theta) + O_x & y &= R.\sin(\theta) + O_y \\
 x' &= R.\cos(\theta + \theta') + O_x & y' &= R.\sin(\theta + \theta') + O_y \\
 x' &= R.(\cos(\theta).\cos(\theta') - \sin(\theta).\sin(\theta')) + O_x & y' &= R.(\sin(\theta').\sin(\theta) + \sin(\theta).\cos(\theta')) + O_y
 \end{aligned}$$

En remplaçant :

$$\begin{aligned}
 R.\cos(\theta) &= x - O_x & R.\sin(\theta) &= y - O_y \\
 x' &= (x - O_x).\cos(\theta') - (y - O_y).\sin(\theta') + O_x & x' &= (x - O_x)\sin(\theta') + (y - O_y).\cos(\theta') + O_y
 \end{aligned}$$

D'où le calcul des positions dans le code :

$$O = \begin{bmatrix} O_x \\ O_y \\ 0 \end{bmatrix} \text{ Le point centrale de l'image} \quad P = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \text{ La position à modifier}$$

$$\begin{bmatrix} x' \\ y' \\ ... \end{bmatrix} = (P - O) \cdot \begin{bmatrix} \cos(\theta') & \sin(\theta') & 0 \\ -\sin(\theta') & \cos(\theta') & 0 \\ 0 & 0 & 0 \end{bmatrix} + O$$

Ces fonction lambda permet de prédéfinir 3 modèle de matrice en fonction de l'effet souhaité.

Listing 10 – Modèle de matrice

```

1 static auto translate = [](double x, double y = 0) {
2     return Matrix{{1,0,0},{0,1,0}, {x,-y,0}};
3 };
4
5 static auto rotate = [](double alpha_deg) {
6     double alpha_radian = M_PI/180.0*alpha_deg;
7     return Matrix{{cos(alpha_radian),sin(alpha_radian),0}, {-sin(alpha_radian),cos(alpha_radian),0}, {0,0,0}};
8 };
9
10 static auto zoom = [](double x, double y = 1) {
11     return Matrix{{1/x,0, 0}, {0, 1/y,0}, {0,0,0}};
12 };

```

On peut ainsi s'aider de ces générateurs de matrice pour des transformations classiques, on laisse la possibilité d'injecter une matrice quelconque pour d'autres transformations.

1.6.1 Résultats

Listing 11 – Application d'exemple de transformation géométrique sur l'image

```

1 name = "lena_translate_100_100";
2 imageTreatment.transform(translate(100,100));
3
4 name = "lena_rotate_45";
5 imageTreatment.transform(rotate(45));
6
7 name = "lena_zoom_2";
8 imageTreatment.transform(zoom(2,2));

```



FIGURE 4 – Exemple de transformation géométrique de l'image

1.7 Génération de bruits et filtrage d'image

Le bruit d'une image est de nature aléatoire. On va tenter de simuler ces bruits afin de restituer les images par filtrage.

Comme en C le PRNG doit être initialisé avec une seed différente à chaque appel de la fonction (Ce n'est pas indispensable ici).

Listing 12 – Générateur de bruit de type salt and pepper

```

1 ImageTreatment & ImageTreatment::salt_and_pepper_noise(const int nb_0, const int nb_255){
2     std::default_random_engine generator;
3     generator.seed(time(nullptr));
4     std::uniform_int_distribution<int> distributionX(0, this->image.getSize_pixel_x() - 1);
5     std::uniform_int_distribution<int> distributionY(0, this->image.getSize_pixel_y() - 1);
6     for(int i = 0; i < nb_0; i++)
7         this->image[distributionX(generator)][distributionY(generator)] = Pixel(this->image.getBytes_per_pixel(), 0);
8     for(int i = 0; i < nb_255; i++)
9         this->image[distributionX(generator)][distributionY(generator)] = Pixel(this->image.getBytes_per_pixel(), 255);
10
11    return (*this);
12 }
```

On tire aléatoirement de manière uniforme des pixels de l'image pour les remplacer en noir ou blanc. Le nombre de pixels modifié de cette manière est donné en argument.

Listing 13 – Générateur de bruit additif gaussien

```

1 ImageTreatment & ImageTreatment::gaussian_noise(const double mean, double sigma){
2     std::default_random_engine generator;
3     generator.seed(time(nullptr));
4     std::normal_distribution<double> distribution(mean, sigma);
5     applyFunction([&distribution, &generator](Pixel & pixel) {
6         std::vector<unsigned char> noise(pixel.size(), 0);
7         std::generate(noise.begin(), noise.end(), [&distribution, &generator](){ return distribution(generator); });
8         pixel+=noise;
9     });
10    return (*this);
11 }
```

Pour simuler ce bruit on ajoute à chaque pixel et pour chaque canal (une réalisation) d'une variable aléatoire suivant une loi normale paramétré par mean et sigma.

L'implémentation est simplifiée par le fait que la bibliothèque standard de c++ dispose nativement un générateur de variable aléatoire gaussienne. Il faut prendre le soin de vérifier que les valeurs après addition ne dépassent pas la capacité du type **unsigned char**.

Listing 14 – Génération d'images avec les 2 types de bruit

```

1 name = "lena_salt_and_pepper_noise";
2 imageTreatment.salt_and_pepper_noise(10000,10000);
3
4 name = "lena_gaussian_noise";
5 imageTreatment.gaussian_noise(0,2);
```



FIGURE 5 – Exemple de génération d'images avec les 2 types de bruit

1.8 Filtrage

1.8.1 Filtrage linéaire

Deux types de filtre vont être implémenter : Linéaire et Median

Ces deux filtre sont dit local au sens où le filtrage d'un pixel est fonction du voisinage de celui ci.

En somme il faut associer un pixel à une zone de pixel voisin.

C'est le rôle de cette fonction.

Listing 15 – Découpage de l'image et application de la fonction associé

```
1 void ImageTreatment::applySegments(const int size_x, const int size_y, std::function<void(Image &, Pixel &)> func){
2     int x_tmp, y_tmp;
3     int middle_x = size_x / 2.0;
4     int middle_y = size_y / 2.0;
5     Image imageSrc(image);
6     Image image_segment(this->image.getBits_per_pixel(), size_x * size_y, size_x, size_y);
7     Pixel tmp;
8     for(int x = 0; x < this->image.getSize_pixel_x(); x++){
9         for(int y = 0; y < this->image.getSize_pixel_y(); y++){
10            for(int kx = 0; kx < size_x; kx++){
11                for(int ky = 0; ky < size_y; ky++){
12                    x_tmp = x + kx - middle_x;
13                    y_tmp = y + ky - middle_y;
14                    image_segment[kx][ky] = x_tmp >= 0 && x_tmp < this->image.getSize_pixel_x() && y_tmp >= 0 && y_tmp < this->image.getSize_pixel_y();
15                }
16            }
17        }
18    }
19 }
20 }
```

Soit **n** le nombre de pixels de l'image et **size_x**, **size_y** la taille de la zone considéré voisine d'un pixel. La fonction parcours l'image et découpe les **n** zones de pixels voisin, chaque zone est centrée par rapport au pixel, si la zone calculée dépasse les dimensions de l'image, les pixels manquants sont mis à 0.

L'association du pixel et de sa zone de voisinage est transmise en argument à la fonction lambda qui sera paramétrée en dehors de la fonction.

La fonction lambda va modifier le pixel courant il est donc nécessaire de travailler avec une copie locale de l'image pour remplir nos zones.

Listing 16 – Implémentation du produit de convolution

```
1 ImageTreatment & ImageTreatment::filter(Matrix matrix){
2     int size_matrix_x = matrix.size();
3     int size_matrix_y = 0;
4     if(size_matrix_x) size_matrix_y = matrix[0].size();
5     this->applySegments(size_matrix_x, size_matrix_y, [& matrix](Image & image_segment, Pixel & pixel) {
6         std::vector<double> result(pixel.size(), 0);
7         for(int i = 0; i < matrix.size(); i++)
8             for(int j = 0; j < matrix[i].size(); j++)
9                 result = result + (matrix[i][j] * image_segment[i][j]);
10
11        for(int y = 0; y < pixel.size(); y++){
12            if(result[y] < 0) result[y] = 0;
13            if(result[y] > 255) result[y] = 255;
14            pixel[y] = result[y];
15        }
16    });
17    return (*this);
18 }
```

Cette fonction appelle la fonction **applySegments** et paramètre la fonction lambda, elle se contente d'affecter le pixel avec la somme du produit terme à terme entre le noyau et la zone de pixels voisins. Nous avons ici une multiplication d'un scalaire avec un pixel, l'opérateur est redéfini. À l'utilisateur de faire attention que son noyau de convolution ne fasse pas dépasser la capacité du type **unsigned char**

1.8.2 Filtre Median

Listing 17 – Implémentation du filtre median

```

1 ImageTreatment & ImageTreatment::median_filter(const int size){
2     this->applySegments(size, size, [size](Image & image_segment, Pixel & pixel) {
3         //Split canals
4         std::vector<std::vector<unsigned char>> canaux = std::vector<std::vector<unsigned char>>(pixel.size(), std::vector<unsigned char>(pixel.size()));
5         for(int i = 0; i < size; i++)
6             for(int j = 0; j < size; j++)
7                 for(int u = 0; u < pixel.size(); u++)
8                     canaux[u].push_back(image_segment[i][j][u]);
9
10        for(int u = 0; u < pixel.size(); u++){
11            std::sort(canaux[u].begin(), canaux[u].end());
12            pixel[u] = canaux[u][(size*size)/2];
13        }
14    });
15    return (*this);
16 }
```

La valeur du pixel est modifiée avec la valeur médiane des pixels de la zone. Il est nécessaire de séparer les canaux sur des vecteurs différents pour les trier efficacement, la valeur médiane se trouve alors au milieu du tableau.

On vient de factoriser notre code ce qui le rend plus facile à tester, à modifier et à comprendre.

Listing 18 – Code pour générer des résultats sur les filtres

```

1 imageTreatment.gaussian_noise(0,2);
2 imageTreatment.salt_and_pepper_noise(10000,10000);
3 name = "lena_gaussian_noise_mean_filter";
4 name = "lena_salt_and_pepper_noise_mean_filter";
5 Matrix mean = {{1/9.0,1/9.0,1/9.0},{1/9.0,1/9.0,1/9.0},{1/9.0,1/9.0,1/9.0}};
6 imageTreatment.filter(mean);
7 name = "lena_gaussian_noise_median_filter";
8 name = "lena_salt_and_pepper_noise_median_filter";
9 imageTreatment.median_filter(3);
10 name = "lena_high_pass_filter";
11 Matrix high = {{-1/9.0,-1/9.0,-1/9.0},{-1/9.0,8/9.0,-1/9.0},{-1/9.0,-1/9.0,-1/9.0}};
12 imageTreatment.convert_bits_per_pixel(GRAY_SCALE).filter(high).threshold({10},{255}).convert_bits_per_pixel(BINARY);
```

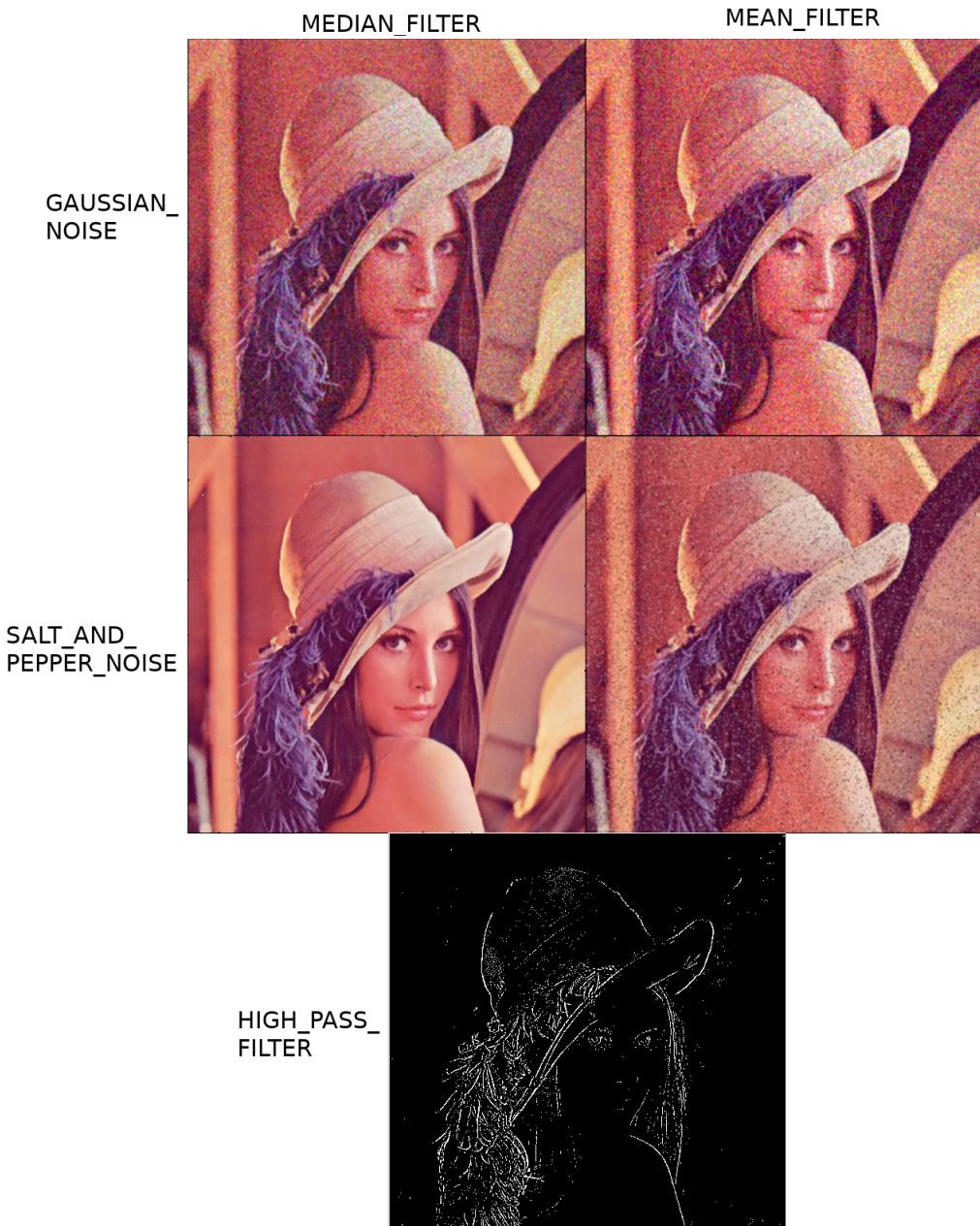


FIGURE 6 – Filtrage des image

Le résultat correspond à ce qu'on s'attend. Si on souhaite augmenter la confiance dans la validité du résultat on pourrait comparer l'image avec le résultat d'ImageJ.

1.9 Surcharge des opérateurs de la classe Image

L'implémentation de ces opérateurs correspond aux opérations classiques sur les matrices.

Les opérateurs sur les pixels sont testés dans le code, les seuls opérateurs non testé sont les suivants : $+= -= + - ==$ de la classe image.

Listing 19 – Test de quelques opérateurs surchargés

```

1 Image &image = imageFactory.open("Images/mask.bmp");
2 Image blackImage = std::move(image); //Copy temporaire => Image 0
3 Image whiteImage = ImageTreatment(blackImage).invert().get(); // Invert => Image 255
4
5 assert(whiteImage == image+ImageTreatment(image).invert().get()); // Mask + !mask = 255
6 assert(blackImage == image-image); //Mask-Mask = 0

```

Par construction les opérateurs membres $+=$ $-=$ sont validés si ce code s'exécute.

1.10 Mesure de similarité/dissimilitude

ImageAnalyse.h
+ histogram(image & : const Image) : std::vector<std::vector<int>>
+ densityHistogram(image & : const Image) : std::vector<std::vector<double>>
+ entropie(image & : const Image) : std::vector<double>
+ mean(image & : const Image) : std::vector<double>
+ var(image & : const Image) : std::vector<double>

ImageMeasure.h
+ jointHistogram(image_1 & : const Image, image_2 & : const Image) : std::vector< std::vector< std::vector<int>>>
+ densityJointHistogram(image_1 & : const Image, image_2 & : const Image) : std::vector< std::vector< std::vector< double>>>
+ difference(image_1 & : const Image, image_2 & : const Image) : std::vector<unsigned int>
+ SSD(image_1 & : const Image, image_2 & : const Image) : std::vector<double>
+ entropieConjointe(image_1 & : const Image, image_2 & : const Image) : std::vector<double>
+ informationMutuelle(image_1 & : const Image, image_2 & : const Image) : std::vector<double>
+ crossCorrelation(image_1 & : const Image, image_2 & : const Image) : std::vector<double>

FIGURE 7 – Prototype des fonctions implémentées

À l'aide des fonctions implémentées dans **ImageAnalyse**, plusieurs mesures de similarité/dissimilitude entre images sont implémentées, leurs utilités et leurs propriétés ont déjà été traité en Master 1.

Pour valider leur implémentation je vais simplement tester leur propriété, à savoir :

$$\forall x, y \in I, S(x, y) = S(y, x) \text{ Symétrie}$$

$$\forall x, y \in I, S(x, y) \geq 0 \text{ Positivité}$$

$$\text{Pour les mesures de dissimilarité : } \forall x, y \in I, S(x, x) \leq S(y, x) \text{ Minimalité}$$

$$\text{Pour les mesures de similarité : } \forall x, y \in I, S(x, x) \geq S(y, x) \text{ Maximalité}$$

Listing 20 – Exemple de tests de propriétés sur la mesure Différence

```

1 Image &image = imageFactory.open("Images/lena24.bmp");
2 std::default_random_engine generator;
3 generator.seed(time(nullptr));
4 std::uniform_int_distribution<int> distributionX(0, image.getSize_pixel_x() - 1);
5 std::uniform_int_distribution<int> distributionY(0, image.getSize_pixel_y() - 1);
6 Image imageTranslate = ImageTreatment(image);
7 transform(::translate(distributionX(generator), distributionY(generator))).get();
8
9 auto differenceMinimal = ::difference(image, image);
10 auto difference1 = ::difference(imageTranslate, image);
11 auto difference2 = ::difference(image, imageTranslate);
12 //Tests de propriétés Mesure de dissimilarité
13 for(int i = 0; i < image.getBytes_per_pixel(); i++)
14     assert(difference1[i] == difference2[i] && difference2[i] >= differenceMinimal[i] && differenceMinimal[i] >= 0);
15 //Pour les tests de propriétés Mesure de similarité(Information Mutuel, CCm) :
16 // assert(difference1[i] == difference2[i] && difference2[i] <= differenceMinimal[i] && difference1[i] >= 0);

```

La mesure **CrossCorrélation** ne possède pas ses propriétés mais on peut la modifier pour quelle passe les tests :

$$CC_m = (1 - CC)^2$$

1.11 Segmentation

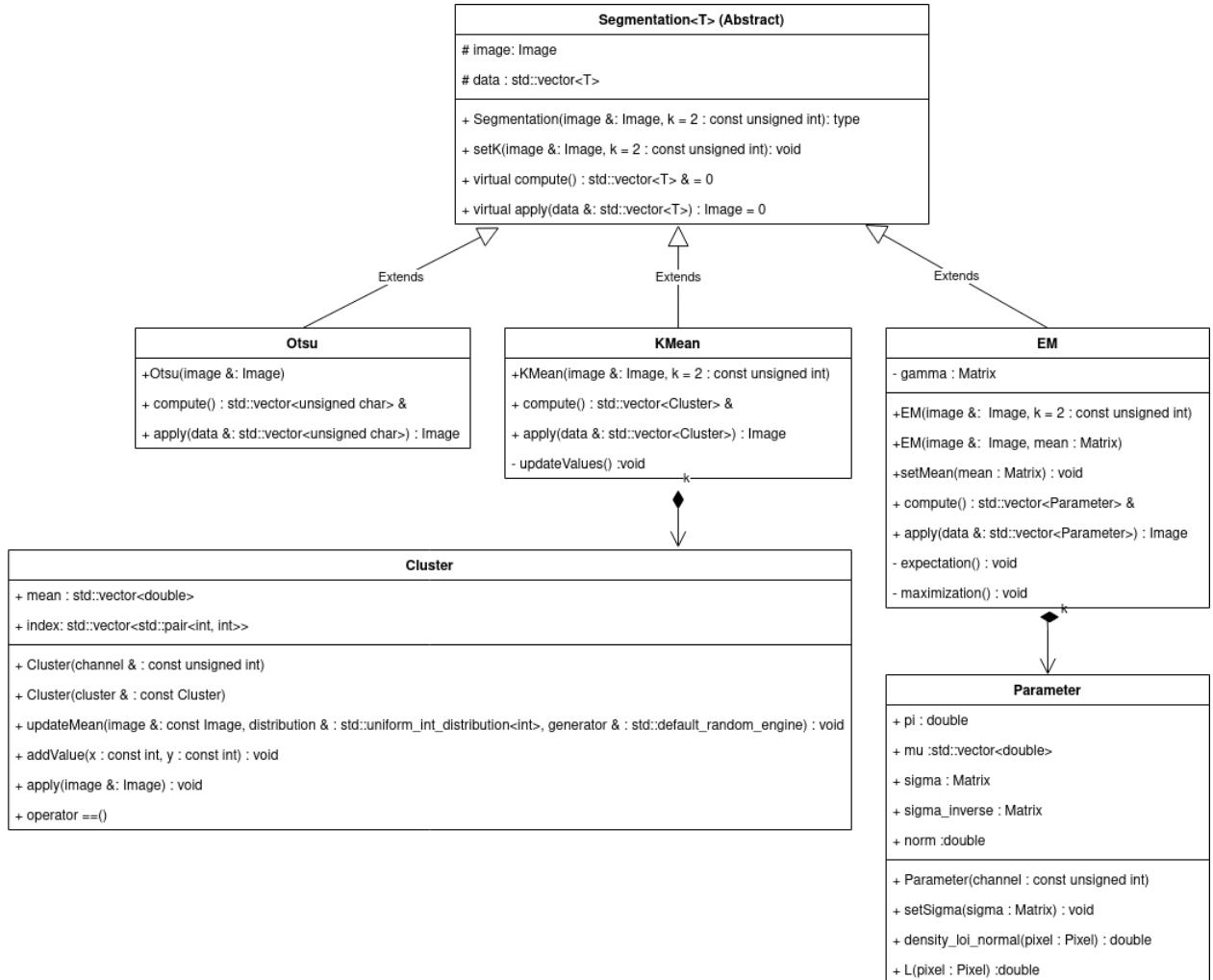


FIGURE 8 – Modélisation UML des 3 algorithmes de segmentation implémentée

1.11.1 Méthode Otsu(Segmentation bimodale)

Listing 21 – Implémentation de la méthode Otsu

```

1 std::vector<unsigned char> & Otsu::compute(){
2     std::vector<std::vector<double>> densityHistogram = ::densityHistogram(image);
3     std::vector<double> var_intra_classe;
4     auto w = [densityHistogram](const int & i, const int & min, const int & max){
5         double w = 0;
6         for(int x = min; x < max; x++) w+=densityHistogram[i][x];
7         return w;
8     };
9     auto mean = [densityHistogram, w](const int & i, const int & min, const int & max){
10        double mean = 0;
11        for(int x = min; x < max; x++, mean+=x*densityHistogram[i][x]);
12        return mean/w(i, min, max);
13    };
14    auto var = [densityHistogram, mean](const int & i, const int & min, const int & max){
15        double var = 0;
16        for(int x = min; x < max; x++) var+=pow(x-mean(i, min, max), 2)*densityHistogram[i][x];
17        return var;
18    };
19    for(int i = 0; i < image.getBytes_per_pixel(); i++){
20        var_intra_classe = std::vector<double>(256,0);
21        for(int k = 0; k < 256; k++) var_intra_classe[k] = var(i, 0, k)+var(i, k, 256);
22        this->data[i] = std::min_element(var_intra_classe.begin(), var_intra_classe.end()) - var_intra_classe.begin();
23    }
24    return this->data;
25 }
```

L'algorithme peut appliquer la méthode sur plusieurs canaux. Pour chaque canal, l'histogramme est calculé.

L'algorithme sépare ces histogrammes en deux classes par un seuil k , puis calcule la variance intra-classe. C'est la somme des variances des données au sein d'une classe. Il gardera le seuil qui minimisera la variance intra-classe, ce même seuil permettra de binariser l'image.

J'ai implémenté cette méthode dans un style fonctionnel pour réduire le nombre de variables.

Listing 22 – Test de la méthode Otsu

```

1 Image &image = imageFactory.open("Images/lena24.bmp");
2 Otsu otsu = Otsu(ImageTreatment(image).convert_bits_per_pixel(GRAY_SCALE).get());
3 auto threshold = otsu.compute();
4 Image imageOtsu = otsu.apply(threshold);
5 ImageBmp(ImageTreatment(imageOtsu).convert_bits_per_pixel(BINARY).get()).save("Results/Segmentation/lenaOtsu.bmp");
```

Bien que l'algorithme puisse prendre en compte plusieurs canaux il est d'usage de la réduire en nuances de gris pour finir par la binariser.



FIGURE 9 – Image segmentée avec la méthode Otsu

1.11.2 K-mean(Classification multi-classe)

L'objectif de l'algorithme est de partitionner les pixels de l'image dans l'espace RGB en k catégorie.

L'algoritme est en 3 étape :

- 1) Initialiser les centres avec un point tiré aléatoirement dans l'espace.
- 2) Assosier chaque pixels à une catégorie tel que sa distance avec le centre de sa classe est minimal parmi toutes les catégories.
- 3) Calculer la moyenne pour chaque classe.

Tant que les centre des classes ne sont pas stable -> alterner l'étape 2 et 3.

KMean ne prend pas en compte la dispersion des pixels autour de son centre. J'implémente cette méthode parce que je pense qu'au vu des histogrammes, il n'est pas nécessaire d'ajuster d'autre paramètres (Variance, Probabilité des classes), les pixels formes des "sphères" dans l'espace.

Listing 23 – Etape 2 : Mesure de distance -> Norme 2

```

1 void KMean::updateValues(){
2     std::vector<int>tmp(this->data.size());
3     for(int x = 0; x < this->image.getSize_pixel_x(); x++){
4         for(int y = 0; y < this->image.getSize_pixel_y(); y++){
5             for(int cl = 0; cl < this->data.size(); cl++){
6                 tmp[cl] = 0;
7                 for(int cha = 0; cha < this->image.getBytes_per_pixel(); cha++){
8                     tmp[cl] += pow(this->data[cl].mean[cha] - this->image[x][y][cha], 2); //Distance
9                 }
10            }
11            //Choisi la catégorie qui minimise la distance du pixel avec son centre
12            this->data[std::min_element(tmp.begin(), tmp.end()) - tmp.begin()].addValue(x, y);
13        }
14    }
15 }
```

Listing 24 – Etape 3 : Calcul de la moyenne

```

1 void Cluster::updateMean(const Image & image, std::uniform_int_distribution<int> & distribution,
2     std::default_random_engine & generator){
3     mean = std::vector<double>(mean.size(), 0);
4     if(index.size() > 0)
5         for(int i = 0; i < image.getBytes_per_pixel(); i++){
6             for(int x = 0; x < index.size(); x++)
7                 mean[i] += image[index[x].first][index[x].second][i];
8             mean[i]/=(double)(index.size()); //Moyenne des pixels
9         } else {
10             for(int i = 0; i < image.getBytes_per_pixel(); i++)
11                 mean[i] = distribution(generator); //Si le cluster est vide -> Modifier aléatoirement son centre
12         }
13     index.clear();
14 }
```

Le nombre optimal de classe K n'est pas connu. Cette méthode graphique va nous permettre de le trouver.

Listing 25 – Optimisation de l'hyperparamètre k

```

1 Image &image = imageFactory.open("Images/imageTest.bmp");
2 KMean kMean(image);
3 std::vector<std::vector<double>> distances;
4 for(int k = 1; k < 3; k++){
5     kMean.setK(k);
6     auto cluster = kMean.compute();
7     Image image_segmentation = kMean.apply(cluster);
8     distances.push_back(SSD(image_segmentation, image));
9 }
10 std::cout << "[";
11 for(std::vector<double> distance : distances){
12     std::cout << "|";
13     for(double value : distance){
14         std::cout << value << ",";
15     }
16     std::cout << "]" , " << std::endl;
17 }
18 std::cout << "]" << std::endl;

```

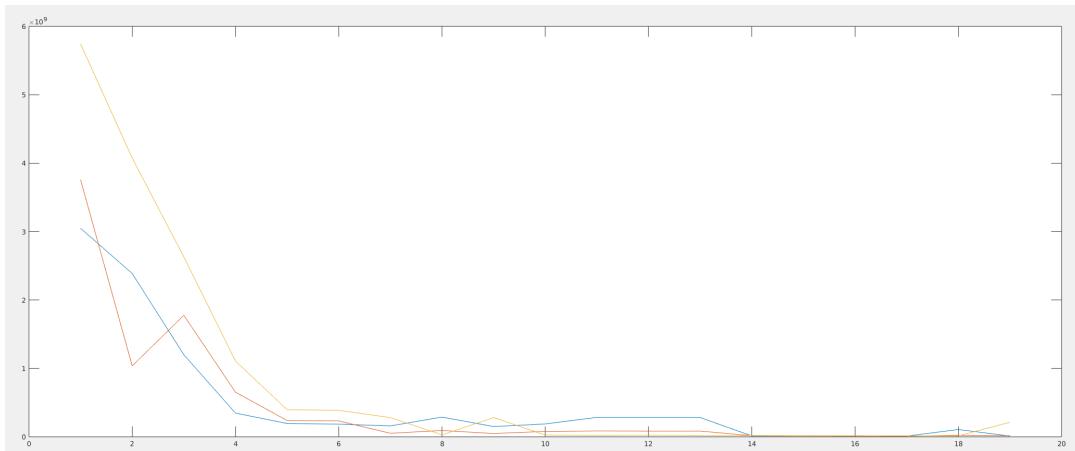


FIGURE 10 – Plot Matlab de la mesure SSD entre l'image et l'image segmenté

On essaye de minimiser à la fois k et l'erreur quadratique, un bon compromis est de prendre k = 8.

Listing 26 – Test de la segmentation avec k = 8

```

1 Image &image = imageFactory.open("Images/imageTest.bmp");
2 KMean kMean(image, 8);
3 auto cluster = kMean.compute();
4 Image image_segmentation = kMean.apply(cluster);
5 ImageBmp(image_segmentation).save("Results/Segmentation/imageTestKMean.bmp");
6 Image imageAbsoluteDifference(image.getBytes_per_pixel(), image.getSize(),
7     image.getSize_pixel_x(), image.getSize_pixel_y());
8
9 for(int x = 0; x < image.getSize_pixel_x(); x++)
10    for(int y = 0; y < image.getSize_pixel_y(); y++)
11        for(int i = 0; i < image.getBytes_per_pixel(); i++)
12            imageAbsoluteDifference[x][y][i] = abs(image[x][y][i]-image_segmentation[x][y][i]);
13 ImageBmp(imageAbsoluteDifference).save("Results/Segmentation/imageTestDiffKMean.bmp");
14 ImageBmp(ImageTreatment(imageAbsoluteDifference).convert_bits_per_pixel(GRAY_SCALE).threshold({20}, {255}).get())
15 save("Results/Segmentation/imageTestDiffKMeanBinary.bmp");

```

1.11.3 Résultats

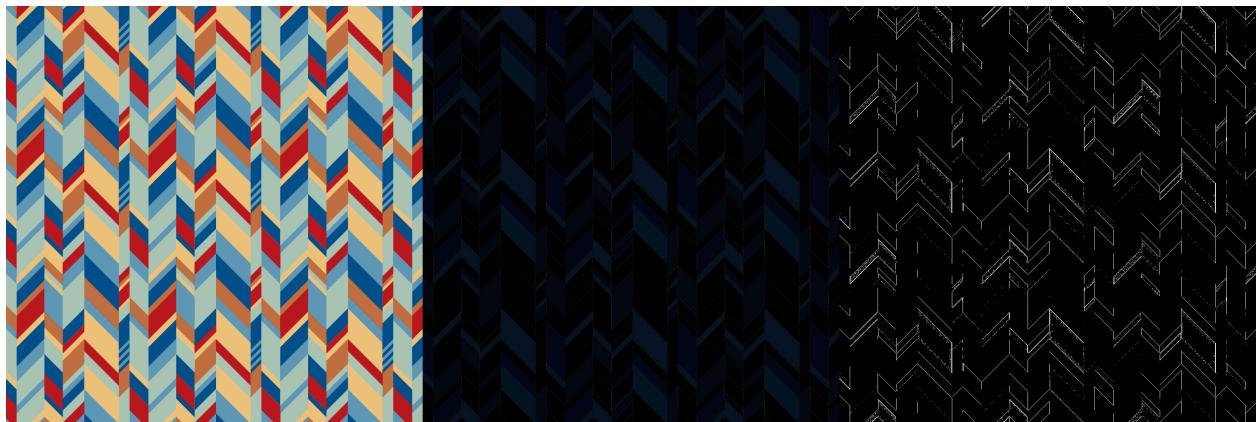


FIGURE 11 – Résultats de la segmentation

À gauche l'image segmentée, au milieu l'erreur absolue sur chaque canal, et à droite la binarisation des erreurs importantes.

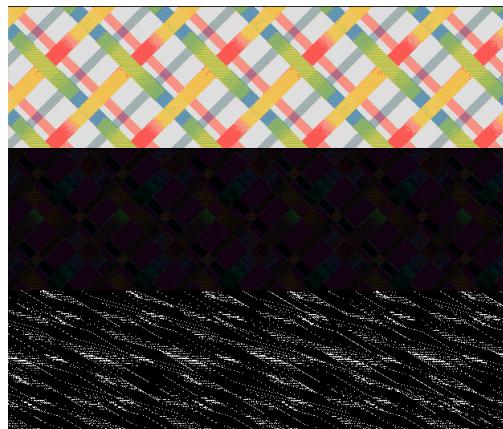


FIGURE 12 – Résultats de la segmentation

Pour $K > 2$ et pour un seul canal cette méthode ne converge pas tout le temps ? => Dans le cas où l'image est en nuances de gris l'initialisation à une grande importance.

1.11.4 EM(Classification multi-classe)

Ici encore le but est de partitionner l'espace RGB en k partitions.

Pour ce faire, les paramètres de K loi normale multidimensionnel (μ_k, π_k, Σ_k) vont être estimées.

L'objectif est d'associer à chaque pixel une probabilité d'appartenir à une classe particulière. La moyenne de la loi qui maximise cette mesure sera alors la nouvelle position du pixel.

Soit N le nombre de pixels, C la dimension de l'espace, et K le nombre de classes.

L'idée est de maximiser la log vraisemblance du modèle vis-à-vis des données : $\sum_{n=1}^N \ln(\sum_{k=1}^K \gamma(Z_{nk}))$

L'algorithme est basé sur les équations suivantes :

$$\begin{aligned}\gamma(Z_{nk})^{(N,K)} &= \frac{\pi_k \mathcal{N}(x|\mu_k, \Sigma_k)}{\sum_{j=1}^K \pi_j \mathcal{N}(x|\mu_j, \Sigma_j)} \quad \text{Expectation} \\ \mu_k^{(1,C)} &= \frac{1}{N_k} \sum_{n=1}^N \gamma(Z_{nk}) X_n \text{ avec } N_k = \sum_{n=1}^N \gamma(Z_{nk}) \quad \text{Maximization} \\ \pi_k &= \frac{N_k}{N} \\ \Sigma_k^{(C,C)} &= \frac{1}{N_k} \sum_{n=1}^N \gamma(Z_{nk})(X_n - \mu_k)(X_n - \mu_k)^T\end{aligned}$$

Les équations produisant les paramètres estimés μ_k, π_k, Σ_k maximise la log vraisemblance.

De la même façon que K-Mean, EM est un algorithme itératif en 3 étapes

- 1) Initialiser les paramètres des K lois multivariées.
- 2) Expectation : Calculer la probabilité qu'un pixel appartienne à une classe
- 3) Maximization : Calculer les paramètres μ_k, π_k, Σ_k fonction de $\gamma(Z_{nk})$

Tant que la log vraisemblance des données ne stagne pas -> Alterner l'étape 2 et 3.

La principale difficulté est de calculer la fonction de densité de la loi normale multidimensionnelle qui s'exprime :

$$f_{\mu, \Sigma}(x) = \frac{1}{(2\pi)^{\frac{C}{2}} |\Sigma|^{\frac{1}{2}}} \cdot \exp(-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu))$$

Dans l'expression nous avons le déterminant, la transposer, et l'inverse d'une matrice de covariance(non singulière).

Pour cela, le header **Matrix.h** est muni des fonctions transpose, déterminant_3_3 et invert_3_3 ainsi que les opérateurs qui convient. Pour simplifier seule les images en RGB seront acceptées.

Listing 27 – Calcul de la fonction de densité de la loi normale multivariée en x

```

1 void Parameter::setSigma(Matrix sigma){
2     Parameter::sigma = sigma;
3     Parameter::sigma_inverse = invert_3_3(sigma);
4     Parameter::norm = pow(2.50663, -3)*pow(determinant_3_3(sigma), -0.5);
5 }
6
7 double Parameter::density_loi_normal(Pixel x){
8     std::vector<double> tmp(x.size(), 0);
9     for(int i = 0; i < tmp.size(); i++)
10         tmp[i] = x[i]-mu[i];
11     Matrix m = convert(tmp);
12     Matrix quadform = transpose(m) * sigma_inverse * m;
13     return norm * exp(-0.5*quadform[0][0]);
14 }
```

Il est essentiel de précalculer les variables ne dépendant pas de x, c'est très coûteux, même sous cette forme les calculs sont très lourd.

Listing 28 – Etape 2 : Expectation, Formation de la matrice N*K des probabilités d'appartenance à une classe

```

1 void EM::expectation(){
2     auto gammaNormalise = [&](const Pixel x, const unsigned int k){
3         double num = this->data[k].L(x);
4         double denum = 0;
5         for(int i = 0; i < this->data.size(); i++) denum += this->data[i].L(x); //aka pi*density_loi_normal(x);
6         return num/denum;
7     };
8     for(int k = 0; k < this->data.size(); k++){
9         for(int x = 0; x < this->image.getSize_pixel_x(); x++){
10            for(int y = 0; y < this->image.getSize_pixel_y(); y++){
11                this->gamma[x*this->image.getSize_pixel_y() + y][k] = gammaNormalise(this->image[x][y], k);
12            }
13        }
14    }
15 }
16 }
```

Listing 29 – Etape 3 : Maximization, Calcul des paramètres μ_k, π_k, Σ_k fonction de $\gamma(Z_{nk})$

```

1 void EM::maximization(){
2     double Nk;
3     std::vector<double> mu_k;
4     Matrix sigma_k;
5     double pi_k;
6     auto NK = [&](const int k) {
7         double nK = 0;
8         for(int n = 0; n < this->gamma.size(); n++) nK+=this->gamma[n][k];
9         return nK;
10    };
11    for(int k = 0; k < this->data.size(); k++){
12        Nk = NK(k);
13        //Estimation de Mu_k
14        mu_k = std::vector<double>(this->data[k].mu.size(), 0);
15        for(int x = 0; x < this->image.getSize_pixel_x(); x++){
16            for(int y = 0; y < this->image.getSize_pixel_y(); y++){
17                mu_k = mu_k+gamma[x*this->image.getSize_pixel_y() + y][k]*this->image[x][y];
18            }
19        }
20        mu_k = 1/Nk*mu_k;
21        //Estimation de Sigma_k
22        sigma_k = Matrix(this->image.getBytes_per_pixel(), std::vector<double>(this->image.getBytes_per_pixel(), 0));
23        for(int x = 0; x < this->image.getSize_pixel_x(); x++){
24            for(int y = 0; y < this->image.getSize_pixel_y(); y++){
25                Matrix m = convert(this->image[x][y] - mu_k);
26                for(int i = 0; i < 3; i++){
27                    for(int j = 0; j < 3; j++){
28                        sigma_k[i][j]=sigma_k[i][j]+gamma[x*this->image.getSize_pixel_y() + y][k]*(m[i][0]*m[j][0]);
29                    }
30                }
31            }
32        }
33        sigma_k = 1/Nk*sigma_k;
34        //Estimation de pi_k
35        pi_k = Nk/(double)image.getSize();
36        this->data[k].mu = mu_k;
37        this->data[k].pi = pi_k;
38        this->data[k].setSigma(sigma_k);
39    }
40 }
41 }
```

Après convergence d'un maximum (Local ?) on peut exploiter les données de cette façon ;

Listing 30 – Segmentation

```

1 Image EM::apply(std::vector<Parameter> & parameters){
2     Image result = std::move(this->image);
3     for(int x = 0; x < this->image.getSize_pixel_x(); x++){
4         for(int y = 0; y < this->image.getSize_pixel_y(); y++){
5             std::vector<double> mean = this->data[std::max_element(
6                 this->gamma[x*this->image.getSize_pixel_y() + y].begin(),
7                 this->gamma[x*this->image.getSize_pixel_y() + y].end())
8                 -this->gamma[x*this->image.getSize_pixel_y() + y].begin()].mu;
9             result[x][y] = Pixel(mean.begin(), mean.end());
10        }
11    }
12    return result;
13 }
```

On cherche pour chaque pixel la classe la plus vraisemblable parmi les classes grâce à la matrice $\gamma(Z_{nk})$ calculé avec les paramètres estimés pour lui affecter le paramètre mu(Moyenne) de la classe.

1.11.5 Résultats

Après quelques tests il s'avère que la maximisation pose beaucoup de problèmes de convergence(Maximum local), l'étape Expectation est très lourde en calcul ce qui empêche d'avoir un résultat correct pour des $K > 3$.

Il faudrait optimiser le calcul de la densité de probabilité avec des bibliothèques spécialisées, pourquoi ne pas passer également par l'histogramme joint en retirant les zéros, ce qui aurait pour effet de factoriser les pixels.

Une solution pour avoir quand même un résultat du niveau de K-mean est d'initialiser les paramètres μ_k par l'algorithme K-Mean.

Listing 31 – Génère les résultats

```

1 Image &image = imageFactory.open("Images/imageTest.bmp");
2 KMean kMean(image, 8);
3 auto cluster = kMean.compute();
4 Matrix mean(cluster.size(), std::vector<double>(image.getBytes_per_pixel()));
5 for(int i = 0; i < cluster.size(); i++){
6     for(int j = 0; j < image.getBytes_per_pixel(); j++){
7         mean[i][j] = cluster[i].mean[j];
8     }
9 }
10 EM em = EM(image, mean);
11 auto parameter = em.compute();
12 Image image_segmentationEM = em.apply(parameter);
13 ImageBmp(image_segmentationEM).save("Results/Segmentation/imageTestEM.bmp");

```

On valide ainsi la partie Expectation puisqu'on retrouve bien la même image segmentée.

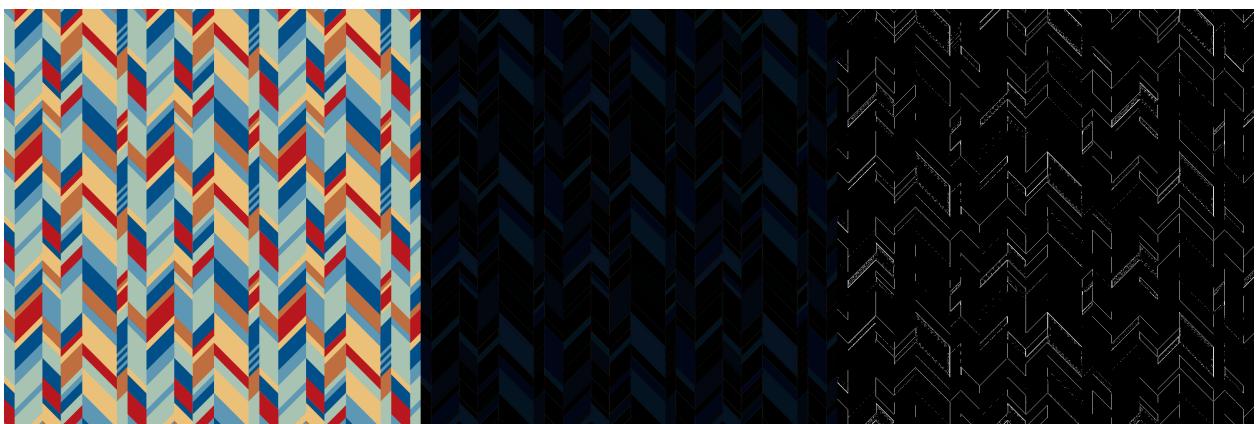


FIGURE 13 – Résultats

2 Critique Générale

Premièrement surcharger des opérateurs est un mauvais choix, j'ai plusieurs fois remis en cause l'implémentation de certains opérateurs, cela affecte massivement les fonctions du programme qu'il faut donc revalider. Également, surcharger des opérateurs sur des `typedef std ::vector<T>` s'avère désastreux, d'une part on modifie le comportement des opérateurs de la bibliothèque standard à partir du moment où l'on inclue **Image.h** dans un programme. De même les opérateurs `>=` et `<=` sont déjà défini sur ces types, les opérandes sont définis en non mutable, pour les redéfinir on ne peut plus mixé constant/non constant, d'où les lambda mutable qui ajoute quelque peu du désordre dans le code.

Une solution pour pallier à ces problèmes aurait été d'encapsuler ces `std ::vector<T>` dans des classes dédiées avec les opérateurs qui s'imposent.