

Université Clermont Auvergne

École Universitaire de Physique et d'Ingénierie

Diplôme : Master Imagerie et Technologie pour la Médecine
(TechMed)



Travail Encadré de Recherche

Présenté par : LEHARA Lyes

Thème : Bibliothèque de traitement d'images en C++

Dirigé par : Mme. PERY Emile

Année universitaire : 2024/2025

Table de matière

1.	Introduction.....	6
2.	Définition d'une image	6
2.1	Codage d'une image (image numérique).....	6
3.	Présentation de Projet	7
4.	Namespace version 1.0	8
4.1	Introduction :	8
4.2	Implémentation des fonctions.....	8
4.2.1	Allocation d'images	8
4.2.2	Création d'images particulières	8
4.2.2.1	Image Blanche.....	8
4.2.2.2	Image Damier.....	9
4.2.2.3	Image Sinusoïdale	9
4.2.3	Lecture et écriture des fichiers images au format brut (.RAW) :.....	10
4.2.3.1	Fichier Raw :	10
4.2.3.2	Lecteur de fichiers images au format brut.....	11
4.2.4	Ecriture de fichiers images au format brut.....	11
4.2.5	Conversion d'image	12
4.2.5.1	Conversion d'image d'un type à un autre	12
4.2.5.2	Conversion d'image de RGB vers niveau de gris	12
4.2.6	Conversion d'une image en fausses couleurs à l'aide d'une LUT	13
4.2.6.1	Fichier LUT.....	13
4.2.6.2	Chargement de fichier LUT (le filtre LUT)	13
4.2.6.3	Application de fausse couleur à l'aide de LUT	13
4.2.6.4	Sauvegarde des images en. PGM et. PPM	14
5.	Tester des filtres LUT sur les différentes images	15
6.	Namespace version 1.1	16
6.1	Introduction	16

6.2	Implémentation des fonctionnalités de namespace v1.0 dans le cadre d'une class	16
7.	Namespace version 2.0	18
7.1	Implémentation des traitements d'images	18
7.1.1	Processing 1&2	18
7.1.2	Addition des images.....	19
7.1.2.1	Addition d'une valeur scalaire à une image	19
7.1.2.2	Addition de deux images.....	21
7.1.3	Égalisation d'histogramme	23
7.1.4	Filtrage par convolution	25
7.1.4.1	Introduction	25
7.1.4.2	Les types des filtres (noyaux)	25
8.	Conclusion	27
9.	Perspectives.....	28
10.	Bibliographie.....	29

Liste des figures

Figure 1: Représentation d'image numérique.	7
Figure 2: Fonction d'allocation d'image.....	8
Figure 3: Création d'image blanche.....	9
Figure 4: Création d'image damier.....	9
Figure 5: Création d'image sinusoïdale.	10
Figure 6: Aperçu d'une image sinusoïdal.....	10
Figure 7: Aperçu d'une image damier	10
Figure 8: Aperçu d'une image Blanche	10
Figure 9: Lecteur d'un fichier brut.....	11
Figure 10: Ecriture d'un fichier brut.....	12
Figure 11: Conversion d'une image d'un type à un autre.	12
Figure 12: Conversion d'une image RGB en niveau de gris.....	13
Figure 13: Chargement de fichier LUT.	13
Figure 14 : Application de LUT sur l'image	14
Figure 15: Conversion d'un pixel gris en RGB via une LUT [9].....	14
Figure 16:Extrait de code pour la fonction sauvegarde .pgm.....	15
Figure 17 : Extrait de code pour la fonction sauvegarde .ppm	15
Figure 18: Résultat de l'application de LUT sur les différentes images.....	16
Figure 19: Aperçu du code de la class Image	17
Figure 20: Aperçu de code de la classe ImageRGB	18
Figure 21: Aperçu de code de la classe Processing 1	19
Figure 22: Aperçu de code de la classe Processing 2	19
Figure 23: Aperçu de code de la classe Addition avec un scalaire	20
Figure 24: le traitement de l'addition d'une image avec une valeur scalaire.....	20
Figure 25: Aperçu d'une image avant l'addition avec un scalaire	21
Figure 26: Addition d'une image après l'addition avec un scalaire	21
Figure 27: Aperçu de code de la classe addition de deux images	21
Figure 28: le traitement de l'addition de deux images	22
Figure 29: Résultat d'addition de deux images de même dimension	22
Figure 30: Résultat de l'addition de deux images de différentes dimensions.....	22

Figure 31 : Aperçu de code de la classe égalisation d'histogramme	23
Figure 32: Aperçu de calcul d'histogramme.....	24
Figure 33 Histogramme de l'image avant l'égalisation	24
Figure 34 Image avant l'égalisation d'histogramme	24
Figure 35: Histogramme de l'image après l'égalisation.....	24
Figure 36: Image après l'égalisation d'histogramme	24
Figure 37: image explicative sur la convolution.....	25
Figure 38: Aperçu de la class Convolution	27

1. Introduction

Le traitement d'images, qui est un sous-domaine du traitement du signal, englobe l'ensemble des méthodes et techniques appliquées aux images et vidéos dans le but d'extraire des informations pertinentes ou d'améliorer leur perception visuelle. Avant toute phase de traitement, un prétraitement est souvent requis pour optimiser la qualité des images. Cela comprend, par exemple, des opérations de rehaussement de contraste, de suppression de bruit, de correction du flou, ainsi que des techniques de segmentation ou d'extraction de contours visant à isoler les éléments significatifs d'une image.

Ce rapport fournit une synthèse des différentes versions évolutives de notre bibliothèque de traitement d'images en C++. La version de base de notre projet repose sur une architecture fonctionnelle utilisant des templates, tandis que les versions ultérieures introduisent une approche orientée objet, en intégrant deux classes, Image et ImageRGB, ainsi que des surcharges d'opérateurs.

La dernière étape de notre projet consiste à mettre en œuvre des méthodes de prétraitement (rehaussement de contraste, suppression de bruit, correction du flou), ainsi que des techniques de segmentation et d'extraction de contours, sur diverses images.

2. Définition d'une image

Une image, c'est une représentation visuelle d'une personne ou d'un objet, réalisée par des moyens comme la peinture, le dessin, la photo ou la vidéo. Elle peut également être définie comme un ensemble structuré d'informations qui, une fois affichées à l'écran, permettent à l'œil humain de les reconnaître [1].

2.1 Codage d'une image (image numérique)

Une image numérique est une représentation visuelle d'un objet ou d'une scène, composée d'une grille de petits éléments appelés pixels. Chaque pixel contient une information, comme une couleur ou un niveau de gris, qui permet de reconstituer l'image dans son ensemble.

Cette image peut provenir d'une photo, d'un dessin ou d'une vidéo, et devient numérique grâce à un processus appelé numérisation¹, qui convertit une image réelle (analogique) en une matrice de valeurs numériques.

Mathématiquement, une image numérique est représentée par une fonction à deux dimensions, $f(x, y)$, où chaque point (x, y) correspond à un pixel, et $f(x, y)$ indique l'intensité lumineuse à ce point. Cela signifie que chaque pixel est une mesure de la lumière captée à un endroit précis de l'image.

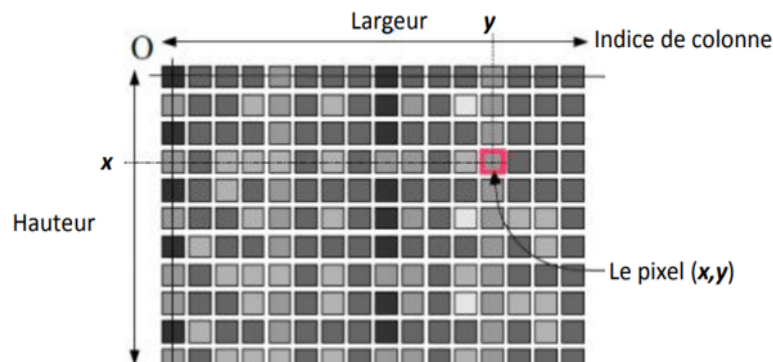


Figure 1: Représentation d'image numérique.

3. Présentation de Projet

La bibliothèque développée pour ce projet a été conçue sur structure organisée, facile à adapter et facile à comprendre, pour simplifier la création d'applications de traitement d'images en C++.

Chaque version de la bibliothèque est encapsulée dans un bloc **namespace** versionnés (**v1.0**, **v1.1** et **v2.0**) afin de bien séparer les différentes versions, éviter les problèmes de noms identiques et permettre d'ajouter de nouvelles fonctionnalités au fil du temps.

La bibliothèque est divisée en modules bien définis, chacun s'occupant d'un type précis de traitement (comme la conversion, le filtrage ou la génération d'images).

L'utilisation des **templates** permet de créer **des fonctions** et des **classes génériques**, pour manipuler les différents types de données d'images

Enfin, elle repose sur une organisation en **classes de base** et **classes filles**, qui permet de mieux structurer le code, de le rendre plus clair, plus facile à maintenir et à faire évoluer.

¹ La **numérisation** : est le processus qui consiste à convertir une information analogique en données numériques exploitables par un ordinateur.

4. Namespace version 1.0

4.1 Introduction :

La version 1.0 utilise des fonctions de base pour travailler avec des images. Elle permet de créer différentes images comme des images blanches, des damiers ou encore des images sinusoïdales. Elle peut aussi lire et enregistrer des fichiers d'image au format brut (. Raw). En plus, elle peut transformer des images de 16 bits ou plus, en images plus simples de 8 bits. Elle permet aussi d'appliquer des filtres de couleur LUT. Toutes ces opérations sont faites en utilisant des tableaux dynamiques **std::vector** en C++.

4.2 Implémentation des fonctions

4.2.1 Allocation d'images

Cette fonction sert à allouer dynamiquement une image sous forme d'un vecteur à une seule dimension dont la taille correspond à la largeur multipliée par la hauteur de l'image. C'est une fonction de type générique (template), elle peut fonctionner avec n'importe quel type de données (int, float, uint16_t, etc.).

```
17 namespace v1_0 {
18
19     template <typename T>
20     std::vector<T> allocationImage(size_t largeur, size_t hauteur) {
21         return std::vector<T>(largueur * hauteur);
22     }
23 }
```

Figure 2: Fonction d'allocation d'image.

4.2.2 Création d'images particulières

4.2.2.1 Image Blanche

La fonction **ImageBlanche** permet de créer une image blanche de taille **hauteur** multiplié par la **longueur**. Chaque pixel est initialisé avec la valeur maximale possible du type générique *T* (grâce à **std::numeric_limits<T>::max()**), ce qui correspond à la couleur blanche en niveaux de gris.

Exemple :

- Si *T* = **uint8_t**², alors max = 255 correspond à la couleur blanche

² **uint8_t** est un type de donnée entier non signé utilisé pour les images 8 bits.

- Si $T = \text{uint16_t}$ ³, alors $\text{max} = 65535$ correspond à la couleur blanche

```

23
24 // Image blanche
25 template <typename T>
26 std::vector<T> ImageBlanche(size_t largeur, size_t hauteur) {
27     return std::vector<T>(largueur * hauteur, std::numeric_limits<T>::max());
28 }
29

```

Figure 3: Création d'image blanche.

4.2.2.2 Image Damier

Cette fonction générique *ImageDamier* crée une image en damier de dimensions données (**hauteur** multiplié par **largeur** de l'image). Elle utilise deux boucles pour parcourir chaque pixel, et détermine si le pixel appartient à une case blanche ou noire en divisant les indices x et y par la taille de la case (**tailleCase**) modulo 2, puis en comparant leurs parités. Si la case est blanche, le pixel prend la valeur maximale 255 (blanc) ; sinon, il prend 0 (noir). L'image ainsi remplie est ensuite retournée.

```

45
46 // Damier
47 template <typename T>
48 std::vector<T> ImageDamier(size_t largeur, size_t hauteur, size_t tailleCase) {
49     std::vector<T> image(largeur * hauteur);
50
51     for (size_t y = 0; y < hauteur; ++y) {
52         for (size_t x = 0; x < largeur; ++x) {
53             bool estBlanc = ((x / tailleCase) % 2 == (y / tailleCase) % 2);
54             image[y * largeur + x] = estBlanc ? std::numeric_limits<T>::max() : 0;
55         }
56     }
57     return image;
58 }
59

```

Figure 4: Création d'image damier.

4.2.2.3 Image Sinusoïdale

La fonction *SinusoidaleImage* génère une image dont l'intensité des pixels varie selon une onde sinusoïdale. Elle prend en paramètre la largeur, la hauteur et la fréquence de l'onde qui représente le nombre de période de la fonction sinus.

- Pour chaque pixel, elle calcule une valeur sinus entre -1 et 1, puis la normalise entre 0 et 1 car les images utilisent des valeurs positives (0 à 255) :

```

double valeur = std::sin(2 * M_PI * frequence * x / largeur);
double valnormaliser = (valeur + 1.0) * 0.5;

```

³ `uint16_t` est un type de donnée entier non signé utilisé pour les images haute précision.

Puis la convertit en une intensité adaptée au type T (par exemple entre 0 et 255). L'image ainsi remplie est ensuite retournée.

```

29
30 // Mire sinusoïdale
31 template <typename T>
32 std::vector<T> SinusoidalImage(size_t largeur, size_t hauteur, double frequence) {
33     std::vector<T> image(largeur * hauteur); //Allocation de l'image
34
35 }
36
37 for (size_t y = 0; y < hauteur; ++y) {
38     for (size_t x = 0; x < largeur; ++x) {
39         double valeur = std::sin(x * 2 * M_PI * frequence * x / largeur);
40         double Valnormaliser = (valeur + 1.0) * 0.5;
41         image[y * largeur + x] = static_cast<T>(Valnormaliser * std::numeric_limits<T>::max());
42     }
43 }
44 return image;
45

```

Figure 5: Création d'image sinusoïdale.

❖ Les Resultats obtenus

Les figures ci-dessus montrent le résultat obtenu après l'appel aux fonctions.

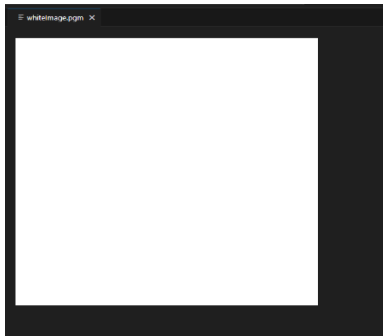


Figure 8: Aperçu d'une image Blanche

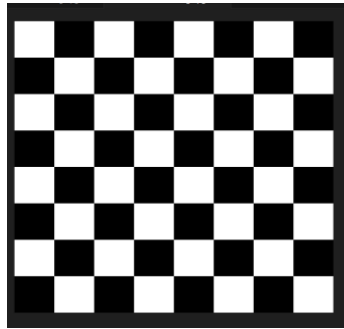


Figure 7: Aperçu d'une image dan

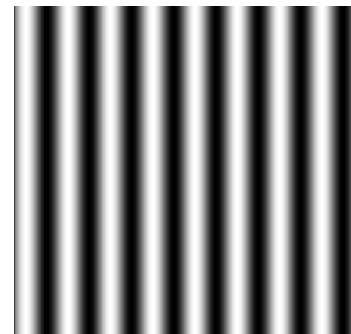


Figure 6: Aperçu d'une image sinusoïdal

4.2.3 Lecture et écriture des fichiers images au format brut (.RAW) :

4.2.3.1 Fichier Raw :

Un **fichier RAW** ⁴(du mot anglais *.raw* signifiant *brut*) est un type de fichier généré par un appareil photo numérique ou un scanner, qui contient les données **brutes** captées par le capteur, **sans aucun traitement**.

⁴ Un **fichier brut (RAW)** est un fichier qui contient uniquement les données sans aucune métadonnée (pas d'en-tête, pas de dimensions, pas d'information sur les canaux couleur).

4.2.3.2 Lecteur de fichiers images au format brut

La fonction *lireImageRaw* lit une image au format RAW et retourne ses pixels dans un vecteur. Elle prend en paramètre le nom du fichier **nomFichier**, les dimensions de l'image (la **hauteur** et la **longueur**) et **bigEndian**⁵ comme Boolean.

Big et **little endian** sont deux manières différentes de stocker les **octets** d'une valeur multi-octets (par exemple un entier sur 2 ou 4 octets) dans un fichier ou une mémoire.

- **Big-endian** → On stocke l'**octet le plus significatif** en premier (en tête).
- **Little-endian** → On stocke l'**octet le moins significatif** en premier.

Exemple : si on veut stocker la valeur **0x1234** (sur 2 octets) :

- En **big-endian** : 12 34
- En **little-endian** : 34 12

Ensuite, Elle lit les données pixel par pixel, en convertissant correctement les octets selon l'ordre spécifié, puis sauvegarde les valeurs dans un vecteur. Si le fichier ne s'ouvre pas, elle renvoie une image vide.

```

110 |
111 | template<typename T>
112 | std::vector<T> lireImageRAW(const std::string& nomFichier, int largeur, int hauteur, bool bigEndian = false) {
113 |     std::ifstream in(Str: nomFichier, Mode: std::ios::binary);
114 |     if (!in) throw std::runtime_error(Message: "Erreur d'ouverture du fichier : " + nomFichier);
115 |
116 |     std::vector<T> img(largeur * hauteur);
117 |     in.read(Str: reinterpret_cast<char*>(img.data()), Count: img.size() * sizeof(T));
118 |
119 |     if (bigEndian && sizeof(T) > 1) {
120 |         for (T& pixel : img) {
121 |             uint8_t* ptr = reinterpret_cast<uint8_t*>(&pixel);
122 |             std::reverse(First: ptr, Last: ptr + sizeof(T));
123 |         }
124 |     }
125 |
126 |     in.close();
127 |     return img;
128 | }
129 |

```

Figure 9: Lecteur d'un fichier brut

4.2.4 Ecriture de fichiers images au format brut

La fonction *ecrireFichierRaw* permet d'enregistrer une image (stockée dans un vecteur) dans un fichier brut (.Raw). Elle prend en paramètre l'image et le nom du fichier de destination

⁵ **Endianness** désigne l'ordre dans lequel les octets sont stocké en mémoire

```

130     template<typename T>
131     void ecrireFichierRaw(const std::vector<T>& image, const std::string& filename) {
132         std::ofstream out(Str: filename, Mode: std::ios::binary);
133         out.write(reinterpret_cast<const char*>(image.data()), Count: image.size() * sizeof(T));
134         out.close();
135     }
136

```

Figure 10: Ecriture d'un fichier brut

4.2.5 Conversion d'image

4.2.5.1 Conversion d'image d'un type à un autre

La fonction **convertImage** permet de convertir une image représentée sous forme de vecteur (`std::vector`) de pixels d'un type source (`SrcType` : type des pixels d'entrée comme `uint8_t`, `float`, etc.) vers un type destination (`DstType` : type des pixels de sortie `uint8_t`, `float`, etc.), avec une option pour ajuster dynamiquement⁶ la plage des valeurs (dynamique).

```

188 // Conversion de l'image
189 template<typename SrcType, typename DstType>
190 std::vector<DstType> convertImage(const std::vector<SrcType>& image, bool adjustDynamics) {
191     std::vector<DstType> imageCnvert(image.size());
192
193     SrcType minVal = *std::min_element(image.begin(), image.end());
194     SrcType maxVal = *std::max_element(image.begin(), image.end());
195
196     for (size_t i = 0; i < image.size(); ++i) {
197         if (adjustDynamics) {
198             imageCnvert[i] = static_cast<DstType>((image[i] - minVal) / static_cast<double>(maxVal - minVal)) * std::numeric_limits<DstType>::max());
199         } else {
200             imageCnvert[i] = static_cast<DstType>(image[i]);
201         }
202     }
203
204     return imageCnvert;
205 }
206
207
208
209
210

```

Figure 11: Conversion d'une image d'un type à un autre.

4.2.5.2 Conversion d'image de RGB vers niveau de gris

La fonction **convertRGB_Gris** transforme une image couleur **RGB** (stockée dans un `std::vector7<uint8_t>`) en une image en **niveaux de gris**. Elle parcourt chaque pixel de l'image puis, elle extrait les composantes rouge, verte et bleue, puis calcule une seule valeur de gris en appliquant une formule pondérée (0.299 pour le rouge, 0.587 pour le vert et 0.114 pour le bleu). Enfin elle retourne le résultat comme une image en noir et blanc.

⁶ L'**ajustement dynamique** consiste à répartir les valeurs de pixel de l'image source (par exemple en 16 bits) sur toute la plage disponible du type de destination (par exemple 8 bits).

⁷ **Vector** est un tableau (array) redimensionnable automatiquement (Il n'est pas nécessaire de définir sa taille au moment de la déclaration.)

```
150
151 std::vector<uint8_t> convertRGB_Gris(const std::vector<uint8_t>& rgbImage, size_t largeur, size_t hauteur) {
152     std::vector<uint8_t> imageGris(Count: largeur * hauteur);
153     for (size_t i = 0; i < largeur * hauteur; ++i) {
154         uint8_t r = rgbImage[i * 3 + 0];
155         uint8_t g = rgbImage[i * 3 + 1];
156         uint8_t b = rgbImage[i * 3 + 2];
157         imageGris[i] = static_cast<uint8_t>(0.299 * r + 0.587 * g + 0.114 * b);
158     }
159     return imageGris;
160 }
161
```

Figure 12: Conversion d'une image RGB en niveau de gris

4.2.6 Conversion d'une image en fausses couleurs à l'aide d'une LUT

4.2.6.1 Fichier LUT

Une **LUT** (*Look-Up Table*, ou table de correspondance en français), est un tableau de valeurs numériques utilisé en traitement d'image, vidéo ou graphisme. Elle permet de modifier facilement les couleurs ou la luminosité de l'image.[4]

4.2.6.2 Chargement de fichier LUT (le filtre LUT)

La fonction **chargerLUT** permet de lire un fichier binaire contenant une table de correspondance (Look-Up Table) et retourne un vecteur de `uint8_t` en sortie

```
137
138 You, seconds ago • Uncommitted changes
139 std::vector<uint8_t> chargerLUT(const std::string& fichier) {
140     std::vector<uint8_t> lut(Count: 256 * 3);
141     std::ifstream file(Str: fichier, Mode: std::ios::binary);
142     if (!file) {
143         std::cerr << "Erreur de lecture du fichier LUT\n";
144         return lut;
145     }
146     file.read(Str: reinterpret_cast<char*>(lut.data()), Count: lut.size());
147     return lut;
148 }
149
```

Figure 13: Chargement de fichier LUT.

4.2.6.3 Application de fausse couleur à l'aide de LUT

La fonction **applyLUT** transforme une image de niveaux de gris en image couleur en remplaçant chaque pixel gris par une couleur définie dans une table LUT, selon son intensité.

Les figures ci-dessous montrent le code de la fonction **applyLUT** ainsi que le schéma fonctionnel illustrant le calcul des valeurs **RGB** à partir d'un pixel en niveaux de gris à l'aide de la table LUT.

```

162
163 std::vector<uint8_t> applLUT(const std::vector<uint8_t>& imageGris, const std::vector<uint8_t>& lut) {
164     if (lut.size() % 3 != 0) {
165         throw std::runtime_error(Message: "LUT doit contenir un multiple de 3 valeurs RGB.");
166     }
167
168     size_t numColors = lut.size() / 3;
169     std::vector<uint8_t> imageRGB(count: imageGris.size() * 3);
170
171     for (size_t i = 0; i < imageGris.size(); ++i) {
172         uint8_t intensity = imageGris[i];
173
174         if (intensity >= numColors) {
175             throw std::runtime_error(Message: "le niveau de gris dépasse la plage de valeurs supportée par la LUT");
176         }
177
178         size_t lutIndex = intensity * 3;
179
180         imageRGB[3 * i + 0] = lut[lutIndex + 0]; // R
181         imageRGB[3 * i + 1] = lut[lutIndex + 1]; // G
182         imageRGB[3 * i + 2] = lut[lutIndex + 2]; // B
183     }
184
185     return imageRGB;
186 }
187

```

Figure 14 : Application de LUT sur l'image

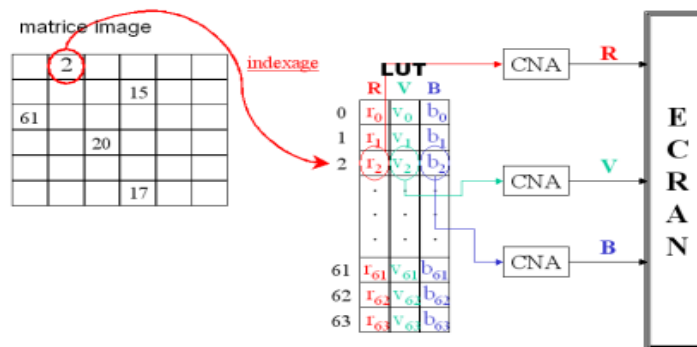


Figure 15: Conversion d'un pixel gris en RGB via une LUT [9]

4.2.6.4 Sauvegarde des images en. PGM et. PPM

- **Format .pgm** (Portable Graymap): est un format d'image en niveaux de gris, où chaque pixel est représenté par une valeur indiquant son intensité lumineuse entre 0 et 255. Le fichier commence par un en-tête texte "P5" (format binaire) suivi des dimensions de l'image et de la valeur maximale des niveaux de gris. Les données des pixels sont ensuite enregistrées directement, ligne par ligne, dans le fichier.
- **Format .PPM (Portable Pixmap)**: Ce format est utilisé pour visualiser des images en couleur (**RGB**). Le fichier commence par un en-tête texte "P6" (format binaire), suivi des dimensions et de la valeur maximale (255), puis tous les pixels RGB sont enregistrés à la suite en binaire.

Les figures suivantes présentent le code de sauvegarde d'une image aux formats **PGM** et **PPM**.

```
337  
338 template<typename T>  
339 void Image<T>::sauvegarderPGM( const std::string& fichier) const {  
340     std::ofstream ofs(Str: fichier, Mode: std::ios::binary);  
341     ofs << "P5\n" << _largeur << " " << _hauteur << "\n255\n";  
342     ofs.write(Str: reinterpret_cast<const char*>(_image.data()), Count: _image.size());  
343 }  
344
```

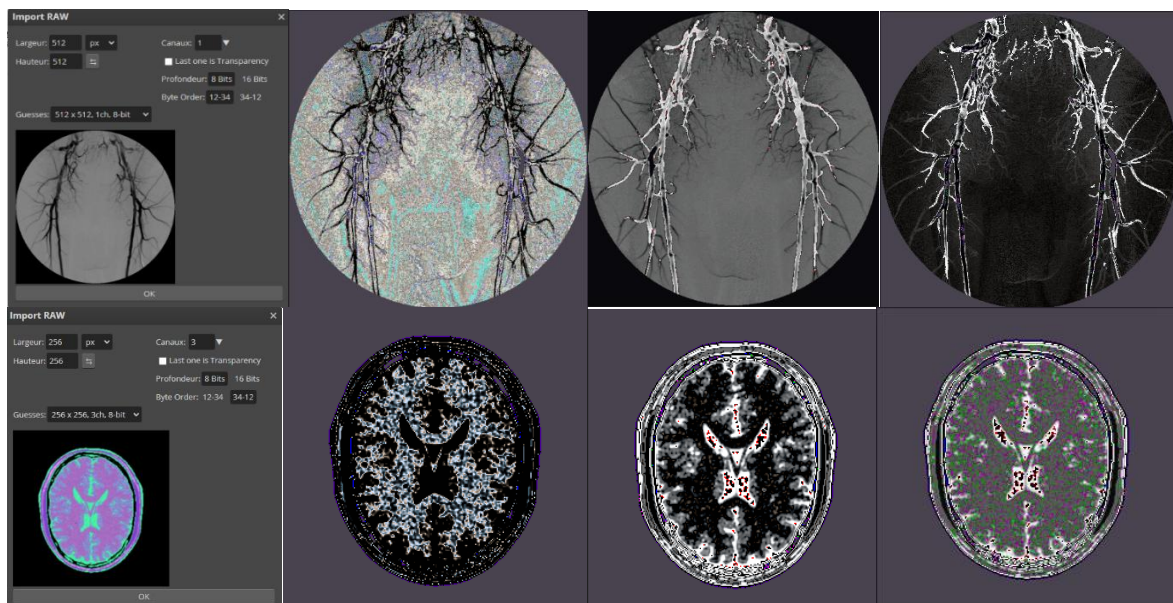
Figure 16: Extrait de code pour la fonction sauvegarde .pgm

```
441 void ImageRGB::sauvegarderPPM(const std::string& fichier) const {  
442     std::ofstream file(Str: fichier, Mode: std::ios::binary);  
443     if (!file) {  
444         std::cerr << "Erreur : impossible de créer le fichier " << fichier << std::endl;  
445         return;  
446     }  
447  
448     file << "P6\n" << getlargeur() << " " << gethauteur() << "\n255\n";  
449     file.write(Str: reinterpret_cast<const char*>(_ImageRGB.data()), Count: getlargeur() * gethauteur() * 3);  
450     file.close();  
451 }  
452
```

Figure 17 : Extrait de code pour la fonction sauvegarde .ppm

5. Tester des filtres LUT sur les différentes images

Les figures suivantes montrent le résultat obtenu après l'appel aux fonctions précédentes



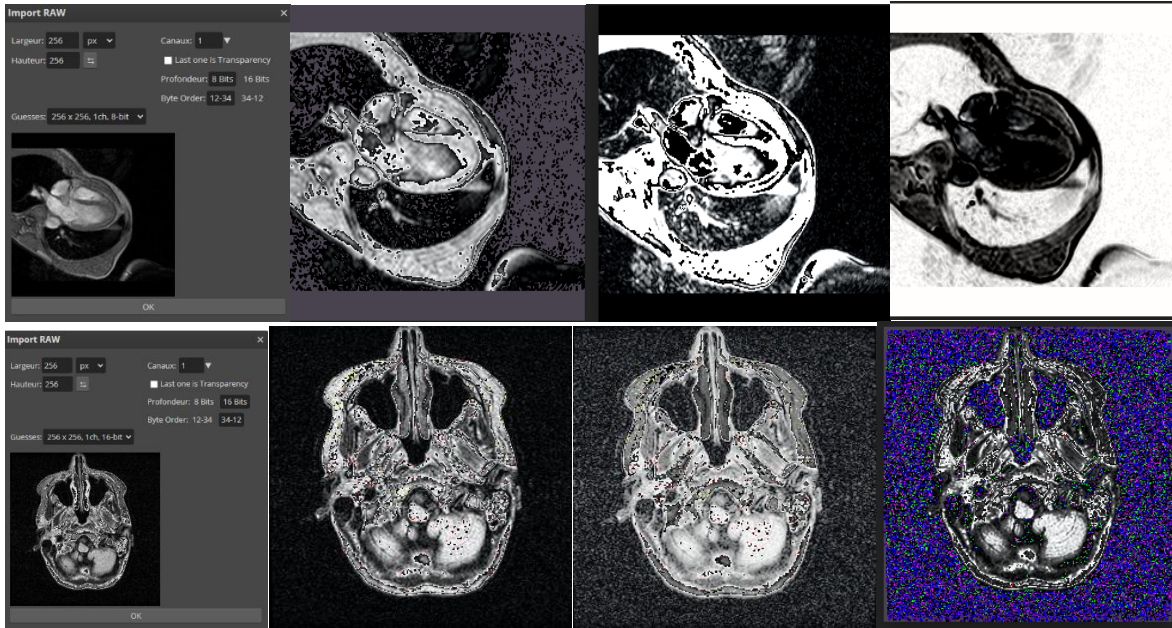


Figure 18: Résultat de l'application de LUT sur les différentes images

6. Namespace version 1.1

6.1 Introduction

L'objectif du namespace `v1_1` est de reprendre les mêmes fonctionnalités de `v1_0` en adoptant une approche orientée objet. On utilise une classe générique et `std::vector` pour coder des images, ce qui rend la gestion plus claire et structurée.

Deux classes principales sont définies : **Image**, qui gère les opérations de base (allocation d'image, création des images, lecture/écriture des fichiers bruts, conversion de type à un autre), et **ImageRGB**, qui hérite de la classe **Image** pour traiter les images couleur et appliquer des **LUT** spécifiques aux images en niveau de gris.

6.2 Implémentation des fonctionnalités de namespace `v1.0` dans le cadre d'une class

L'image ci-dessus représente la class `Image` de type *template* (un type générique), qui peut fonctionner avec n'importe quel type de données (`float`, `uint8_t`, `uint16_t`, ...), elle encapsule les dimensions de l'image (`_largeur`, `_hauteur`), ainsi que la structure de stockage des pixels avec `std::vector` qui sert à stocker tous les pixels de l'image sous forme d'un tableau comme attributs de la class.

Elle fournit aussi des méthodes pour la création d'images (**blanche**, **sinusoïdale**, **damier**), la lecture/écriture de fichiers (**PGM**, **RAW**), l'impression, la conversion de type d'image, et la surcharge de l'opérateur **parenthèse ()** pour accéder aux *pixel* (*i, j*).

```

252 namespace v1_1{
253
254     template<typename T>
255     class Image {
256     public:
257         Image(int largeur, int hauteur);
258         Image();
259
260         int getLargeur() const;
261         int getHauteur() const;
262
263         const std::vector<T>& getData() const;
264         void setData(const std::vector<T>& image);
265
266         void creerImageBlanche();
267         void creerSinusoidale(double frequence);
268         void creerDamier(int tailleCase);
269
270         void sauvegarderPGM(const std::string& fichier) const;
271         void ecrireFichierRaw(const std::string& fichier) const;
272
273         void printImage() const;
274
275         static Image<T> lireImageRAW(const std::string& fichier, size_t largeur, size_t hauteur, bool bigEndian = false);
276     };
277
278     template<typename Dst>
279     Image<Dst> convertirImage(bool ajuster = false) const;
280
281     T& operator()(int x, int y);
282     const T& operator()(int x, int y) const;
283
284 protected:
285     int _largeur, _hauteur;
286     std::vector<T> _image;

```

Figure 19: Aperçu du code de la class *Image*

La class⁸ *ImageRGB* est une class dérivé de la class *Image* conçue pour gérer les images en couleurs *RGB*, elle prend une propriété *_ImageRGB* pour accéder aux données de pixels d'une image *RGB*. Elle inclut également des méthodes pour lire/écrire des fichiers PPM, charger une LUT binaire, convertir une image RGB en niveaux de gris.

Cette classe propose aussi une **surcharge⁹ de constructeurs¹⁰** :

- Le premier constructeur permet de créer une image **RGB** vide à partir d'une taille spécifiée (largeur * hauteur).

⁸ Une **class** est un modèle qui sert à créer des objets, elle regroupe des attributs et des fonctions dans une structure cohérente

⁹ **Surcharge de méthodes** : consiste à définir plusieurs méthodes portant le même nom mais avec des paramètres différents

¹⁰ Un **constructeur** est une méthode prends le même nom de la class qui est appelée automatiquement lorsqu'un objet est créé.

- Le deuxième constructeur permet de convertir une image en niveaux de gris en une image **RGB** en utilisant une **LUT**.

```

423 You, seconds ago | 1 author (You)
424 class ImageRGB : public Image<uint8_t>{
425     public :
426         //Surcharge de constructeur
427         ImageRGB(const Image<uint8_t>& rgbImage, int largeur, int hauteur);
428         ImageRGB(const Image<uint8_t>& imageGris, size_t Largeur, size_t Hauteur, const std::vector<uint8_t>& lut);
429
430         const std::vector<uint8_t>& getImageRGB() const;
431         void sauvegarderPPM(const std::string& fichier) const;
432
433         static std::vector<uint8_t> chargerLUT(const std::string& fichier);
434         static Image<uint8_t> convertRGB_Gris(const std::vector<uint8_t>& imageRGB, int largeur, int hauteur);
435         static Image<uint8_t> lectureImageRawRGB(const std::string& fichier, size_t largeur, size_t hauteur, bool bigEndian = false);
436
437     private:
438         std::vector<uint8_t> _ImageRGB;
439
440     };
441
442

```

Figure 20: Aperçu de code de la classe *ImageRGB*

7. Namespace version 2.0

Cette dernière version de la bibliothèque contient les bases de traitement d'image tel que, l'addition des images, égalisation d'histogramme et le filtre convolution.

7.1 Implémentation des traitements d'images

7.1.1 Processing 1&2

Les traitements utilisés dans cette version sont implémentés sous forme d'une class **template** prenant en paramètre une ou deux images. Ces classes héritent soit de la classe abstraite (class de base) *Processing1*, pour les traitements qui nécessitent une seule image en entrée, soit de la classe *Processing2*, pour les traitements qui nécessitent deux images en entrée. La classe *Processing1* possède trois attributs :

- Une image d'entrée **imageInput_**
- Une image de sortie **imageOutput_**, accessible via la méthode **getOutput()**
- Un booléen **inPlace_** indiquant si le traitement doit écraser l'image originale.

Elle définit également une méthode virtuelle pure **Process()**, qui est appelée par la méthode **Update()** pour effectuer le traitement de base sur l'image

La figure suivante montre le code de la fonction Processing1 :

```

546
547     template<typename T>
      You, seconds ago | 1 author (You)
548     class Processing1 {
549     public:
550         Processing1(v1_1::Image<T>& input, bool inPlace = false);
551
552         virtual void Process() = 0;
553         void Update();
554
555         v1_1::Image<T>& getOutput();
556
557     protected:
558         v1_1::Image<T>& imageInput_;
559         v1_1::Image<T> imageOutput_;
560         bool inPlace_;
561     };
562

```

Figure 21: Aperçu de code de la classe Processing 1

La class Processing2 qui est utilisé pour les traitements qui nécessitent deux images en entrée, elle possède 4 attributs protégés :

- **imageInput_** : image d'entrée
- **imageOutput_** : image de sortie (accessible via la méthode **getOutput()**)
- **inPlace_** : booléen permettant d'indiquer si le traitement est effectué en place (modification directe de l'image d'entrée).

Elle possède également d'une méthode virtuelle pure **Process()** à redéfinir et une méthode **Update()** pour déclencher le traitement.

La figure suivante montre le code de la class processing2() :

```

567
568 { //Constructeur
569     Processing2(v1_1::Image<T>& input1, v1_1::Image<T>& input2, bool inPlace = false);
570 //Methodes
571     virtual void Process() = 0;
572
573     void Update();
574
575     v1_1::Image<T>& getOutput();
576
577     protected:
578     //attributs
579     v1_1::Image<T>& imageInput1_;
580     v1_1::Image<T>& imageInput2_;
581     v1_1::Image<T> imageOutput_;
582     bool inPlace_;
583 };
584

```

Figure 22: Aperçu de code de la classe Processing 2

7.1.2 Addition des images

7.1.2.1 Addition d'une valeur scalaire à une image

La class **AdditionScalaire** est une class Template qui hérite de la class **Processing1**, ce qui signifie qu'elle prends une seule image en entrée. Elle permet d'ajouter une valeur scalaire à chaque pixel de cette image. Elle contient un constructeur pour initialiser l'image, la valeur

scalaire (**valScalaire**) et **boolean inplace** qui permet d'indiquer si l'opération modifie l'image originale, elle contient aussi la méthode **Process** effectue le traitement principal en surchargeant la méthode virtuelle de la class mère. Elle dispose également d'une méthode statique **additionScalaire**, qui permet de faire le traitement sans qui créer des instances de la class.

```

617
618 // Addition avec un scalaire
619 template<typename T>
620 class AdditionScalaire : public Processing1<T> {
621 public:
622     AdditionScalaire(v1_1::Image<T>& input, T valScalaire, bool inplace = false);
623
624     void Process() override;
625     static v1_1::Image<T> additionScalaire(const v1_1::Image<T>& input, T valScalaire);
626 private:
627     T valScalaire_;
628 };
629
630
631

```

Figure 23: Aperçu de code de la classe Addition avec un scalaire

- La méthode **process()** qui permet de faire le traitement de l'addition d'une image avec une valeur scalaire. Voici le code correspondant :

```

598 template<typename T>
599 void AdditionScalaire<T>::Process() {
600     size_t largeur = this->imageInput_.getlargeur();
601     size_t hauteur = this->imageInput_.gethauteur();
602     v1_1::Image<T>& imageSortie = this->inPlace_ ? this->imageInput_ : this->imageOutput_;
603
604     for (size_t y = 0; y < hauteur; ++y) {
605         for (size_t x = 0; x < largeur; ++x) {
606             T val = this->imageInput_(x, y) + valScalaire_;
607
608             // Saturation (valeur maximale selon le type T)
609             if (std::numeric_limits<T>::is_integer) {
610                 val = std::min<T>(val, std::numeric_limits<T>::max());
611             }
612
613             imageSortie(x, y) = val;
614         }
615     }
616 }

```

Figure 24: le traitement de l'addition d'une image avec une valeur scalaire

Le résultat obtenu après l'addition d'une image avec une valeur scalaire est illustré dans les figures suivantes :



Figure 25: Aperçu d'une image avant l'addition avec un scalaire

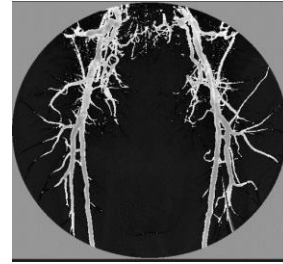


Figure 26: Addition d'une image après l'addition avec un scalaire

7.1.2.2 Addition de deux images

La class **Addition** est aussi une class **template** qui hérite de la class **Processing2**, indiquant qu'elle prend deux images en entrée. Son constructeur prend deux image **input1** et **input2** ainsi qu'un **boolean inplace** qui indique si le résultat doit écraser la première image. L'addition de deux images se fait comme suit :

- Si les deux images ont les mêmes dimensions, l'addition se fait pixel par pixel, pour avoir une image du même format en sortie.
- Si leurs dimensions sont différentes, l'image la plus petite est d'abord étendue par l'ajout des zéros (padding), pour correspondre à la taille de l'image la plus grande avant d'effectuer l'addition.

```
600
601 // Addition de deux images
602 template<typename T>
    You, 4 days ago | 1 author (You)
603 class Addition : public Processing2<T> {
604 public:
605 //constructeur
606     Addition(v1_1::Image<T>& input1, v1_1::Image<T>& input2, bool inplace = false);
607
608     void Process() override;
609
610     static v1_1::Image<T> addition(const v1_1::Image<T>& imageInput1, const v1_1::Image<T>& imageInput2);
611 };
612
613
```

Figure 27: Aperçu de code de la classe addition de deux images

- La méthode **process()** qui permet de faire le traitement de l'addition de deux images. Voici le code correspondant :

```

551     template<typename T>
552     void Addition<T>::Process() {
553         size_t largeur1 = this->imageInput1_.getlargeur();
554         size_t hauteur1 = this->imageInput1_.gethauteur();
555         size_t largeur2 = this->imageInput2_.getlargeur();
556         size_t hauteur2 = this->imageInput2_.gethauteur();
557
558         size_t largeurFinale = std::max(largeur1, largeur2);
559         size_t hauteurFinale = std::max(hauteur1, hauteur2);
560
561         v1_1::Image<T> imageTemp(largeurFinale, hauteurFinale);
562
563         for (size_t y = 0; y < hauteurFinale; ++y) {
564             for (size_t x = 0; x < largeurFinale; ++x) {
565                 T val1 = (x < largeur1 && y < hauteur1) ? this->imageInput1_(x, y) : 0;
566                 T val2 = (x < largeur2 && y < hauteur2) ? this->imageInput2_(x, y) : 0;
567
568                 imageTemp(x, y) = val1 + val2;
569             }
570         }
571         if (this->inPlace_) {
572             this->imageInput1_ = std::move(imageTemp);
573         } else {
574             this->imageOutput_ = std::move(imageTemp);
575         }
576     }

```

Figure 28: le traitement de l'addition de deux images

Les résultats obtenus après l'addition des deux images sont illustrés dans les figures suivantes

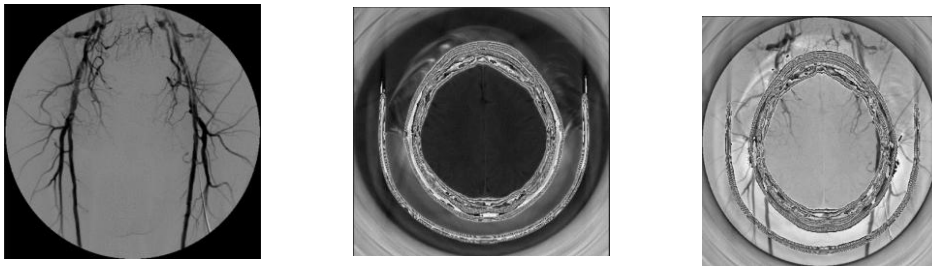


Figure 29: Résultat d'addition de deux images de même dimension



Figure 30: Résultat de l'addition de deux images de différentes dimensions

7.1.3 Égalisation d'histogramme

L'égalisation d'histogramme est une technique de traitement d'image utilisée pour améliorer le contraste des images. Elle consiste à répartir les niveaux de gris d'une image de manière uniforme, pour que tous les niveaux de luminosité soient mieux représentés.

Il est calculé en trois étapes principales :

❖ Calcul de l'histogramme :

Un histogramme d'image est une représentation graphique (vecteur) qui montre la répartition des pixels selon leur intensité lumineuse (de 0 à 255). Par exemple :

- Une image sombre aura un histogramme concentré à gauche.
- Une image très claire aura un histogramme concentré à droite

❖ Calcul de la fonction de distribution cumulative CDF

La CDF indique pour chaque niveau d'intensité (de 0 à 255), combien de pixels dans l'image ont des valeurs inférieures ou égales à ce niveau.

❖ Transformation des niveaux de gris (via LUT)

Consiste à créer une table LUT pour remplacer les anciennes valeurs de pixels par les nouvelles valeurs calculées

L'image suivante montre le code de la classe d'égalisation d'histogramme

```

767 // la class Égalisation d'histogramme
768 template<typename T>
769 ...
770 class egalisationHistogram : public Processing1<T> {
771 public:
772     egalisationHistogram(v1_1::Image<T>& input, bool inplace = false);
773
774     void Process() override;
775
776     v1_1::Image<uint8_t> getHistogramImage();
777
778     void computeHistogram(const v1_1::Image<T>& image);
779 private:
780     std::vector<int> histogramImage_;
781 };
782
783 You, 3 weeks ago • ajouter de namespace v2.0 à TER3
    
```

Figure 31 : Aperçu de code de la classe égalisation d'histogramme

Les étapes de calcul de l'histogramme des images sont implémentées dans la méthode **process()** et illustrées dans la figure suivante.

```

647     template<typename T>
648     void egalisationHistogram<T>::Process() {
649         static_assert(std::is_same<T, uint8_t>::value, "egalisation d'Histogram ne supporte que uint8_t");
650
651         size_t w = this->imageInput_.getlargeur();
652         size_t h = this->imageInput_.gethauteur();
653         size_t total = w * h;
654
655         v1_1::Image<T>& output = this->inPlace_ ? this->imageInput_ : this->imageOutput_;
656         output = v1_1::Image<T>(w, h);
657
658         // 1. Histogramme d'entrée
659         histogramImage_ = std::vector<int>(Count: 256, Val: 0);
660         for (auto val : this->imageInput_.getData()) {
661             histogramImage_[val]++;
662         }
663         // 2. CDF
664         std::vector<int> cdf(Count: 256);
665         cdf[0] = histogramImage_[0];
666         for (int i = 1; i < 256; ++i)
667             cdf[i] = cdf[i - 1] + histogramImage_[i];
668
669         // 3. LUT
670         std::vector<uint8_t> lut(Count: 256);
671         for (int i = 0; i < 256; ++i)
672             lut[i] = static_cast<uint8_t>(255.0 * cdf[i] / total);
673
674         // 4. Appliquer LUT
675         for (size_t y = 0; y < h; ++y)
676             for (size_t x = 0; x < w; ++x)
677                 output(x, y) = lut[this->imageInput_(x, y)];
678
679         // 5. Recalculer histogramme après égalisation
680         compterHistogram(image: output);
681     }

```

Figure 32: Aperçu de calcul de l'histogramme

Les figures suivantes montrent l'image originale et son histogramme :



Figure 34 Image avant l'égalisation d'histogramme

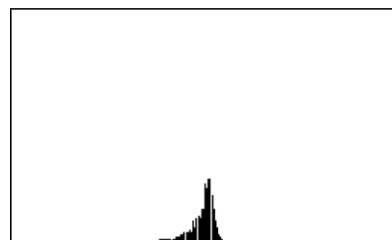


Figure 33 Histogramme de l'image avant l'égalisation

Son résultat après l'application de l'égalisation de l'histogramme :

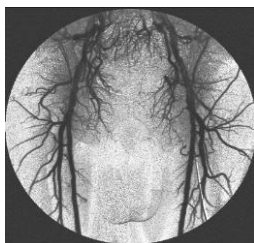


Figure 35: Image après l'égalisation d'histogramme

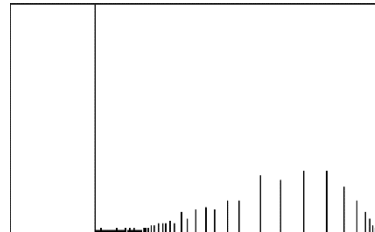


Figure 36: Histogramme de l'image après l'égalisation

7.1.4 Filtrage par convolution

7.1.4.1 Introduction

La convolution est une technique utilisée en traitement d'images. Elle consiste en une opération de multiplication de deux matrices de tailles différentes (généralement une petite et une grande), mais de même dimensionnalité semblable (1D, 2D ou 3D), produisant une nouvelle matrice. [12]

La convolution est donc le traitement d'une matrice image par une autre petite matrice appelée matrice de convolution, le filtre ou noyau (kernel).

Le filtre parcourt toute la matrice de l'image de manière incrémentale et génère une nouvelle matrice constituée des résultats de la multiplication.

Le filtre étudie successivement chacun des pixels de l'image. Pour chaque pixel, que nous appellerons pixel initial, il multiplie la valeur de ce pixel et de chacun des 8 pixels qui l'entourent par la valeur correspondante dans le noyau. Il additionne l'ensemble des résultats et le pixel initial prend alors la valeur du résultat final. Ainsi la formule de convolution est définie comme suit :

$$Y_{ij} = \sum_{u=-k}^k \sum_{v=-k}^k H_{u,v} \cdot X_{i-u,j-v}$$

La figure suivante montre l'exemple simple de produit de convolution.

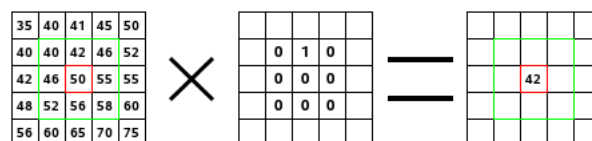


Figure 37: image explicative sur la convolution

7.1.4.2 Les types des filtres (noyaux)

En traitement d'image plusieurs types de filtres (noyaux) peuvent être utilisés selon le cas recherché. [13]

Voici quelques filtres les plus courants :

- **Filtre Moyenneur :**

Est utilisé pour réduire le bruit et obtenir un lissage uniforme de l'image, mais il peut flouter les contours.

Il remplace chaque pixel de l'image par la moyenne de ces voisins. Par exemple si on prend un noyau de 3x3 contient des coefficients égaux à 1/9. Sa forme est :

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

- **Filtre Gaussien :**

Ce filtre est appliqué lorsqu'on souhaite un flou plus naturel tout en préservant les bords. Il repose sur la distribution gaussienne, il applique des poids plus élevés au pixel central et plus faibles aux pixels plus éloignés. La matrice de son noyau est représentée par :

$$K = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

- **Filtre exponentiel :**

Le filtre exponentiel dit aussi filtre de lissage (filtrage lisseur), est principalement utilisé pour rajouter de flou dans image et pour réduire certains types de bruit. La matrice de son noyau est représentée par :

$$K = \frac{1}{32} \begin{bmatrix} 1 & 3 & 1 \\ 3 & 16 & 3 \\ 1 & 3 & 1 \end{bmatrix}$$

L'image suivante montre le code de la classe Convolution qui hérite de la classe **Processing1**. Elle contient un constructeur prenant en paramètre une image d'entrée, un noyau de convolution ainsi qu'un booléen optionnel **inPlace** pour déterminer si le traitement modifie directement l'image originale, elle contient aussi une méthode **Process()** qui permet d'appliquer l'opération de convolution. La classe propose également trois méthodes statiques : **creerMoyenneur()**, **creerGaussien()** et **creerExponentiel()**, qui permettent respectivement de générer des noyaux de convolution de type moyenneur, gaussien et exponentiel

```
885
886 template<typename T>
    You, seconds ago | 1 author (You)
887 class Convolution : public Processing1<T> {
888 public:
889     Convolution( v1_1::Image<T>& image, const v1_1::Image<float>& noyau, bool inplace = false);
890
891     void Process() override;
892
893     static v1_1::Image<float> creerMoyenneur(int taille);
894     static v1_1::Image<float> creerGaussien(int taille, float sigma);
895     static v1_1::Image<float> creerExponentiel(int taille, float lambda);
896
897 private:
898     v1_1::Image<float> noyau_;
899 };
900
```

Figure 38: Aperçu de la class Convolution

8. Conclusion

Le traitement d'image joue un rôle fondamental dans plusieurs domaines notamment la vision ordinateur, robotique et l'intelligence artificielle. Il sert à améliorer la qualité des images pour faciliter l'interprétation dans les traitements plus complexe comme la classification ou l'apprentissage automatique.

Le travail réalisé dans le cadre de ce rapport s'inscrit sur le développement d'une bibliothèque de traitement d'images en C++ structuré en trois version successives (namespace v1.0, namespace v1.1, et namespace v2.0).

Namespace v1.0 est une sous bibliothèque qui regroupe les fonctionnalités de base pour le traitement d'images. Elle inclut les fonctions pour allouer dynamiquement des images, la création des image blanche, damier et sinusoïdal, la lecture et l'écriture des fichier images en format brut (. RAW) selon leur type de codage **big** ou **little endian**, ainsi que la conversion entre types d'images. Elle contient également des fonctions qui permet de transformer des images en couleur **RGB** en niveau de gris (noir et blanc) en utilisant une table correspondance (**LUT**).

La version 1.1 (namespace v1.1) est une évolution de la version 1.0 vers une architecture orienté objet, afin de bien organiser et structurer le code. Cette version introduit une **class template Image** qui encapsule les fonctionnalités de bases liées à la gestion des images, telles que l'allocation des images, la création des images simple (images blanches, damiers, et sinusoïdale), la lecture et l'écriture des fichier image brut (.RAW), ainsi que la conversion entre ces différents types. Une class dérivée **ImageRGB** a également été introduite pour gérer les images en couleur (RGB). Cette architecture rend le code plus modulaire pour facilité leur utilisation dans la prochaine version.

Le namespace v2.0 représente la dernière version de cette bibliothèque, dont l'objectif l'implémentation de fonctionnalités plus avancées de traitement d'image. Cette version intègre des class Template ou abstract héritant de la class **Image**, afin de gérer des traitement complexe sur l'images. Elle permet d'additionner deux images de même ou de différentes dimensions, l'addition d'une image avec une valeur scalaire, le calcul d'histogramme pour améliorer le contraste, ainsi qu'une class pour le calcul de convolution, utilisée pour réduire le flou et le bruit dans les images.

9. Perspectives

Dans les perspectives d'améliorer cette bibliothèque de traitement d'image en C++ plusieurs fonctionnalités peuvent être ajouter :

- Filtrage par multiplication dans le domaine fréquentiel, avec des tests sur des images avec le filtre idéal et le filtre Butterworth
- Détection de contours par filtrage de Sobel
- Le multi seuillage ainsi que le seuillage d'Otsu
- Enfin, l'intégration d'algorithme de classification comme k-moyennes

Ces améliorations permettraient d'améliorer la capacité de la bibliothèque afin de mieux répondre aux besoins en traitement d'images

10. Bibliographie

- [1] <https://ticinformatique.wordpress.com/2019/10/22/la-notion-dimage>
- [2] [https://www.mediathequesludotheques.grandorlyseinebievre.fr/default/li
mage-au-temps-du-numerique.aspx?_lg=fr-FR](https://www.mediathequesludotheques.grandorlyseinebievre.fr/default/li
mage-au-temps-du-numerique.aspx?_lg=fr-FR)
- [3] [https://fac.umc.edu.dz/fstech/cours/Electronique/Master%20ST%C3%A9
l%C3%A9com/CoursImageProcessing1.pdf](https://fac.umc.edu.dz/fstech/cours/Electronique/Master%20ST%C3%A9
l%C3%A9com/CoursImageProcessing1.pdf)
- [4] <https://www.lcd-compare.com/definition-de-look-up-table.htm>
- [5] https://fr.wikipedia.org/wiki/Image_num%C3%A9rique
- [6] <https://perso.univ-lyon1.fr/erwan.guillou/files/CM%20-%20complet.pdf>
- [7] [https://www.umer-
lastig.fr/plf_homepage/cours/cours_ima/ima_numeriques.pdf](https://www.umer-
lastig.fr/plf_homepage/cours/cours_ima/ima_numeriques.pdf)
- [8] <https://perso.univ-lyon1.fr/erwan.guillou/files/CM%20-%20complet.pdf>
- [9] [http://ressources.unit.eu/cours/videocommunication/Transformation_ponc
tuelle_histogramme.pdf](http://ressources.unit.eu/cours/videocommunication/Transformation_ponc
tuelle_histogramme.pdf)
- [10] [https://openclassrooms.com/fr/courses/7137751-programmez-en-
orienté-objet-avec-c#table-of-content](https://openclassrooms.com/fr/courses/7137751-programmez-en-
orienté-objet-avec-c#table-of-content)
- [11] <https://chatgpt.com/auth/login>
- [12] <https://wp.unil.ch/risk/files/2015/12/8.-Filtres-et-convolution.pdf>
- [13] <https://mathinfo.alwaysdata.net/2016/11/filtres-de-convolution>
- [14] [https://labsticc.univ-
brest.fr/~rodin/FTP/Enseignements/L3/ProjetsIUP/old/ProjetImage/Filtra
ge.pdf](https://labsticc.univ-
brest.fr/~rodin/FTP/Enseignements/L3/ProjetsIUP/old/ProjetImage/Filtra
ge.pdf)
- [15] [https://medium.com/%40er_95882/convolution-image-filters-cnns-
and-examples-in-python-pytorch-bd3f3ac5df9c](https://medium.com/%40er_95882/convolution-image-filters-cnns-
and-examples-in-python-pytorch-bd3f3ac5df9c)
- [16] [https://openclassrooms.com/fr/courses/4470531-classez-et-
segmentez-des-donnees-visuelles/5026661-filtrez-une-image](https://openclassrooms.com/fr/courses/4470531-classez-et-
segmentez-des-donnees-visuelles/5026661-filtrez-une-image)
- [17] <https://perso.esiee.fr/~perretb/I5FM/TAI/convolution/index.html>
- [18] [https://perso.ensta-
paris.fr/~manzaner/Cours/Poly/Poly_Chap2_Filtrage.pdf](https://perso.ensta-
paris.fr/~manzaner/Cours/Poly/Poly_Chap2_Filtrage.pdf)
- [19] <https://github.com/daehli/image-processing>
- [20] <https://perso.esiee.fr/~perretb/I5FM/TAI/histogramme/index.html>

[21] [http://ressources.unit.eu/cours/videocommunication/UNIT_Image
%20Processing_nantes/Version%20FR/Chapitre%202/Ressources/Transf
ormation%20d'histogramme/Rchap2_TransfoHisto_FR\[final\].pdf](http://ressources.unit.eu/cours/videocommunication/UNIT_Image%20Processing_nantes/Version%20FR/Chapitre%202/Ressources/Transformation%20d'histogramme/Rchap2_TransfoHisto_FR[final].pdf)