Université Clermont Auvergne

École Universitaire de Physique et d'Ingénierie

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



Travaux 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

[**1.1** **Introduction :** 4](#_Toc197524791)

[**1.2** **Définition d’une image :** 4](#_Toc197524792)

[**1.3** **Codage d’une image (image numérique) :** 5](#_Toc197524793)

[Figure 1: Représentation d’image numérique. 5](#_Toc197524794)

[**1.4** **Présentation de Projet :** 5](#_Toc197524795)

[**1.5** **Namespace version 1.1 :** 5](#_Toc197524796)

[1.5.1 Définition : 5](#_Toc197524802)

[1.5.2 Implémentation des fonctions : 6](#_Toc197524803)

[1.5.3 Allocation d’images 6](#_Toc197524804)

[Figure 2: Fonction d’allocation d’image. 6](#_Toc197524805)

[1.5.4 Création d'images particulières : 6](#_Toc197524806)

[Figure 3: Création d’image blanche. 6](#_Toc197524807)

[Figure 4: Création d’image damier. 7](#_Toc197524808)

[Figure 5: Création d’image sinusoïdale. 7](#_Toc197524809)

[Figure 6: Création d’image sinusoïdale. 8](#_Toc197524810)

[1.5.5 Lecture et écriture de fichiers images au format brut (.raw) : 8](#_Toc197524811)

[Figure 7: Lecteur d’une image .Raw 9](#_Toc197524812)

[Figure 8: Ecriture d’une image .Raw 9](#_Toc197524813)

[1.5.6 Conversion d’image : 10](#_Toc197524814)

[Figure 9: Conversion d’une image d’un type à un autre. 10](#_Toc197524815)

[Figure 10: Conversion d’une image d’un type à un autre. 10](#_Toc197524816)

[1.5.7 Conversion d'une image en fausses couleurs à l'aide d'une LUT : 10](#_Toc197524817)

[Figure 11: Chargement de fichier LUT. 11](#_Toc197524818)

[Figure 12: Application de LUT sur l’image. 11](#_Toc197524819)

[1.5.8 Tester des filtres LUT sur les différentes images : 11](#_Toc197524820)

[Figure 13: Résultat de l’application de LUT sur les différents images 12](#_Toc197524821)

[**1.6** **Namespace version 1.1 :** 13](#_Toc197524822)

[1.6.1 Introduction 13](#_Toc197524823)

[1.6.2 Implémentation des fonctionnalités de namespace v1.0 dans le cadre d’une class 13](#_Toc197524824)

[Figure 14: Aperçu du code de la class Image 14](#_Toc197524825)

[Figure 15: Aperçu de code de la class ImageRGB 14](#_Toc197524826)

[**1.7** **Namespace version 2.0 :** 14](#_Toc197524827)

[**1.8** **Conclusion :** 14](#_Toc197524828)

Liste des figures

[Figure 1: Représentation d’image numérique. 5](#_Toc197524829)

[Figure 2: Fonction d’allocation d’image. 6](#_Toc197524830)

[Figure 3: Création d’image blanche. 6](#_Toc197524831)

[Figure 4: Création d’image damier. 7](#_Toc197524832)

[Figure 5: Création d’image sinusoïdale. 7](#_Toc197524833)

[Figure 6: Création d’image sinusoïdale. 8](#_Toc197524834)

[Figure 7: Lecteur d’une image .Raw 9](#_Toc197524835)

[Figure 8: Ecriture d’une image .Raw 9](#_Toc197524836)

[Figure 9: Conversion d’une image d’un type à un autre. 10](#_Toc197524837)

[Figure 10: Conversion d’une image d’un type à un autre. 10](#_Toc197524838)

[Figure 11: Chargement de fichier LUT. 11](#_Toc197524839)

[Figure 12: Application de LUT sur l’image. 11](#_Toc197524840)

[Figure 13: Résultat de l’application de LUT sur les différents images 12](#_Toc197524841)

[Figure 14: Aperçu du code de la class Image 14](#_Toc197524842)

[Figure 15: Aperçu de code de la class ImageRGB 14](#_Toc197524843)

## **Introduction :**

Le traitement d’images, sous-domaine du traitement du signal, regroupe l’ensemble des méthodes et techniques appliquées aux images et vidéos dans le but d’en extraire des informations pertinentes ou d’en améliorer la perception visuelle. Avant toute phase de traitement, un prétraitement est souvent nécessaire afin d’optimiser la qualité des images. Cela inclut, 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 présente 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 suivantes introduisent une approche orientée objet, en utilisant deux classes Image et **ImageRGB** et des surcharges d’opérateurs

La dernière étape de notre projet consiste à implémenter 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 différentes images.

## **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. C’est aussi un ensemble organisé d’informations qui, une fois affichées à l’écran, forment quelque chose que l’œil humain peut reconnaître.

D’un point de vue mathématique, une image peut être vue comme une fonction en deux dimensions, notée f(x, y), où chaque point (x, y) correspond à une certaine valeur. Cette valeur représente la lumière ou le niveau de gris à cet endroit précis de l’image.

## **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 numérisation, 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.

## **Présentation de Projet :**

Le projet a été conçu pour faciliter et structurer le code destiné au traitement d'images avec C++. Il est composé de plusieurs blocs de type **namespace** afin d'éviter les conflits de noms de fonctions utilisés. Ce projet est organisé en trois blocs **namespace** : **v1.0**, **v1.1** et **v2.0**.

## **Namespace version 1.1 :**



### Définition :

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++.

### Implémentation des fonctions :

### Allocation d’images

Cette fonction sert à allouer dynamiquement une image sous forme 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, unsigned char, etc.).

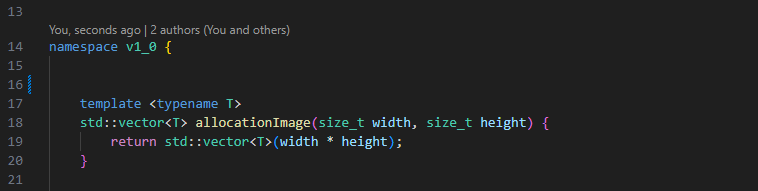


Figure 2: Fonction d’allocation d’image.

### Création d'images particulières :

#### Image Blanche

La fonction **ImageBlanche**  permet de créer une image blanche de taille **hauteur** (height) multiplié par la **longueur** (width)**.** 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 = **unsigned char**, alors max = 255
  + Si T = **unsigned short**, alors max = 65535

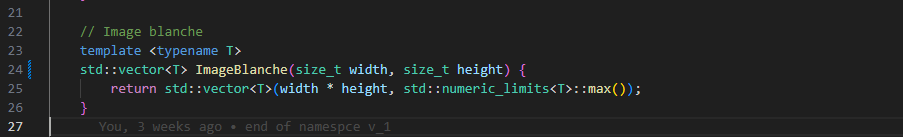
****

Figure 3: Création d’image blanche.

#### Image Damier

Cette fonction générique ***ImageDamier*** crée une image en damier de dimensions données (haueur multiplié par longueur 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 coordonnées *x* et *y* par la taille de la case(**tailleCase**), 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.

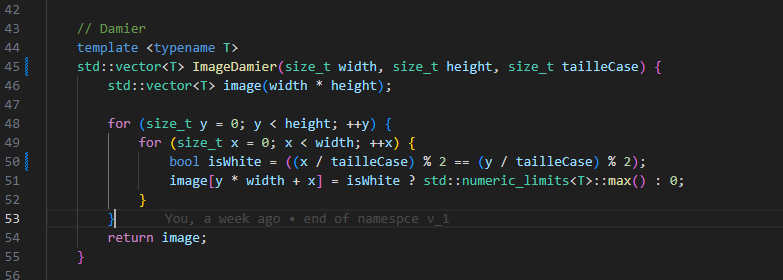


Figure 4: Création d’image damier.

#### Image Sinusoïdale

La fonction **SinusoïdaleImage** 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 valeur positive (0 à 255) :

**double valeur = std::sin(2 \* M\_PI \* frequency \* x / width);**

**double valnormaliser = (val + 1.0) \* 0.5;**

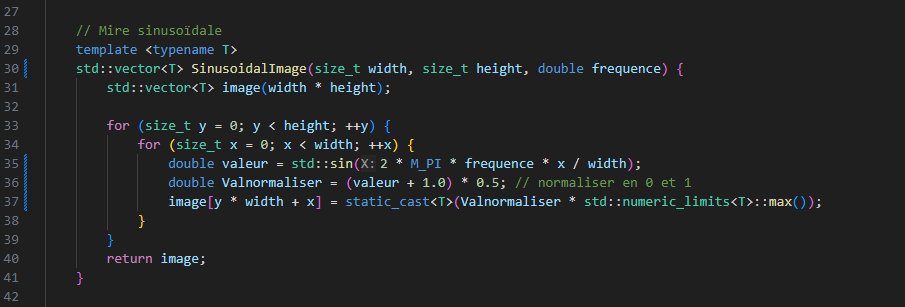
**** Puis la convertit en une intensité adaptée au type T (par exemple entre 0 et 255). L’image ainsi remplie est ensuite retournée.

Figure 5: Création d’image sinusoïdale.

* **Resultat obtenu**

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

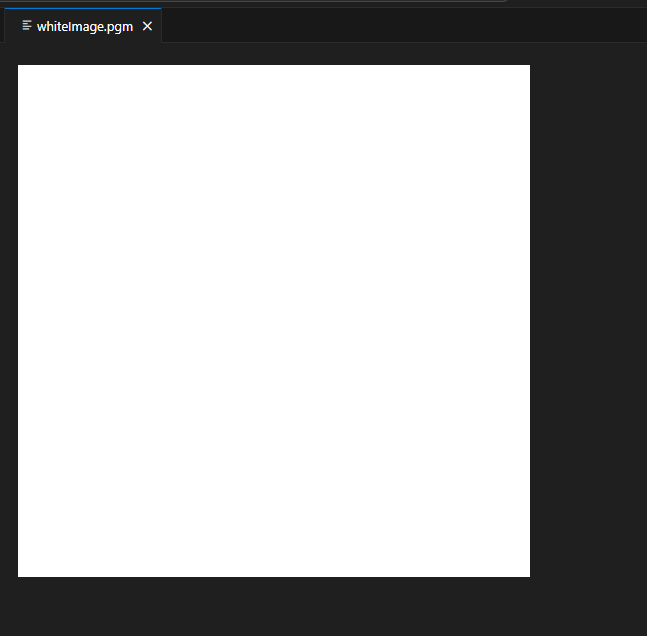
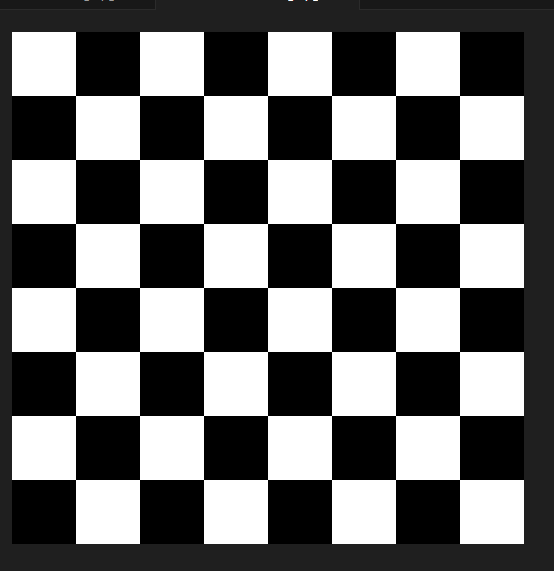


Figure 6: Création d’image sinusoïdale.

### Lecture et écriture de fichiers images au format brut (.raw) :

#### Lecteur de fichiers images au format brut

La fonction ***readRwImage*** lit une image au format RAW et retourne ses pixels dans un vecteur. Elle prend en paramètre le nom du fichier **filename**, les dimensions de l’image (la **haueur** et la **longueur**) et l’endianess (**big** ou **little endian**).

**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.

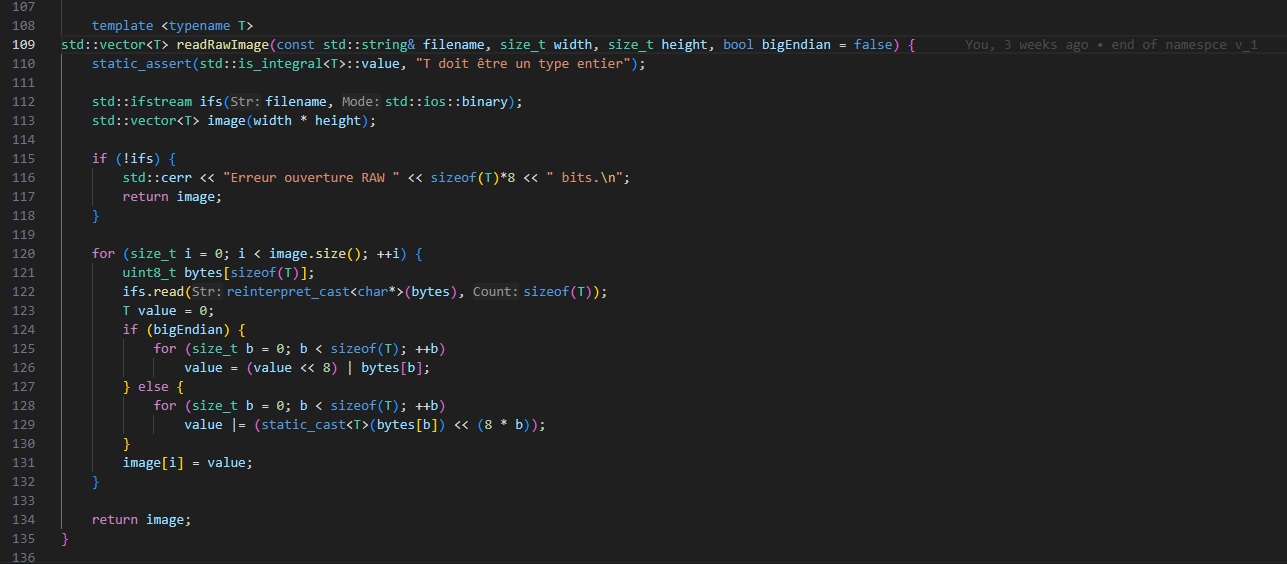


Figure 7: Lecteur d’une image .Raw

#### Ecriture de fichiers images au format brut

La fonction ***writeRawImage*** permet d’enregistrer une image (stockée dans un vecteur) dans un fichier brut (.Raw). Pour chaque pixel, elle découpe sa valeur en octets et les écrit dans le fichier soit en big-endian (octet fort en premier), soit en little-endian (octet faible en premier), selon le paramètre ***bigEndian***.

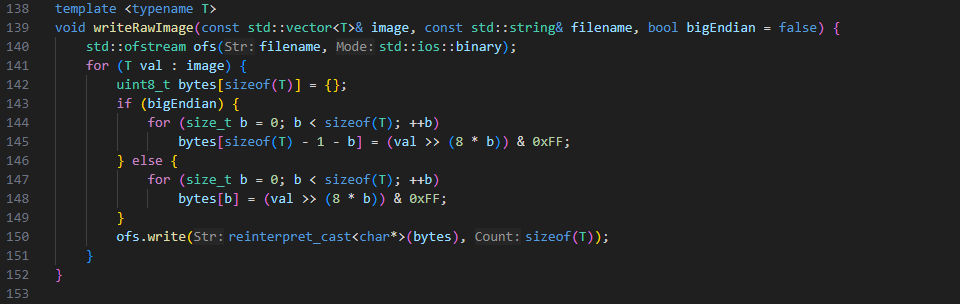


Figure 8: Ecriture d’une image .Raw

### Conversion d’image :

#### Conversion d’image d’un type à un autre :

La fonction ***converImage*** 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 dynamiquement la plage des valeurs (dynamique).

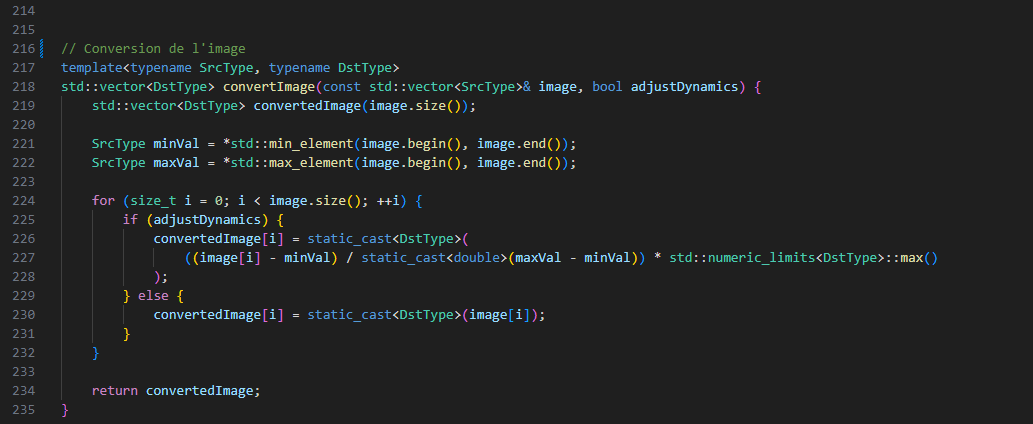


Figure 9: Conversion d’une image d’un type à un autre.

#### Conversion d’image de RGB vers niveau de gris :

La fonction ***convertRGBToGrayscale*** transforme une image couleur **RGB** (stockée dans un **std::vector<uint8\_t>)** en une image en **niveaux de gris**. Elle parcourt chaque pixel de l’image, 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.

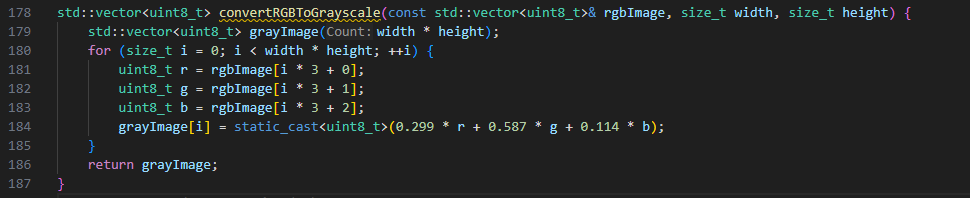
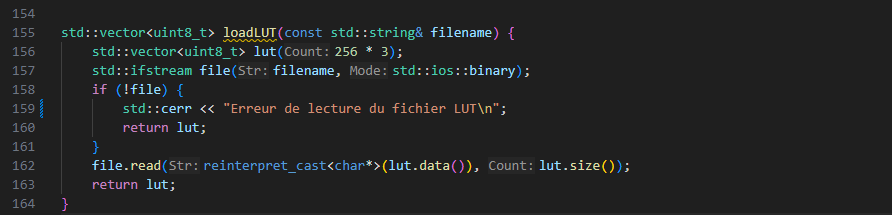


Figure 10: Conversion d’une image d’un type à un autre.

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

#### Chargement de fichier LUT (le filtre LUT) :

Figure 11: Chargement de fichier LUT.

#### Application de fausse couleur à l’aide de LUT

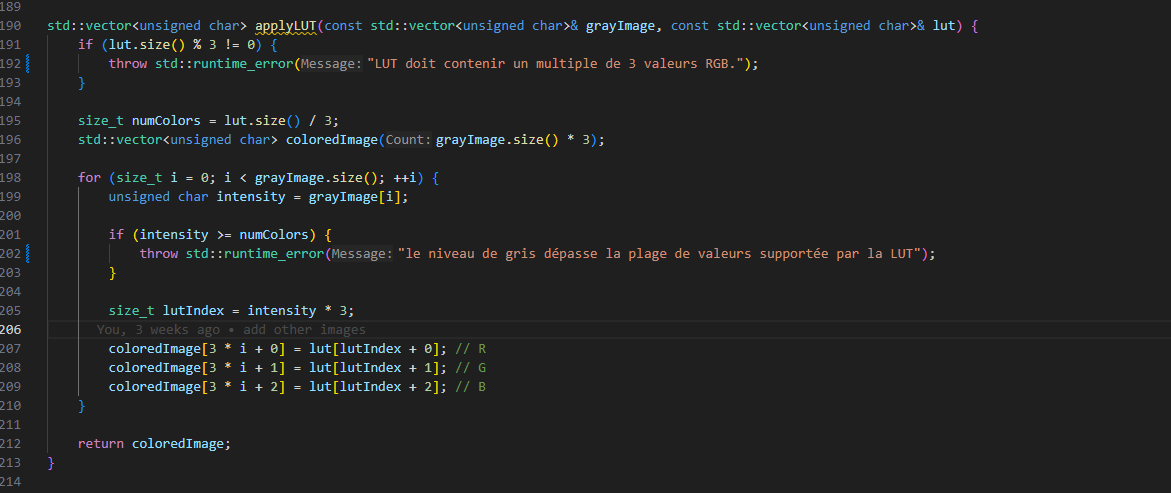
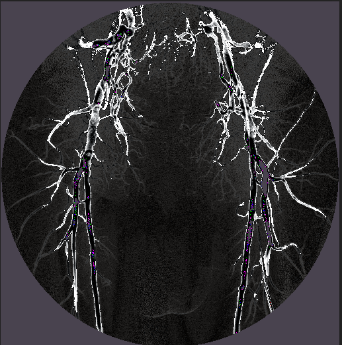
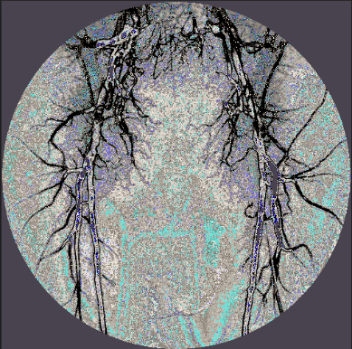
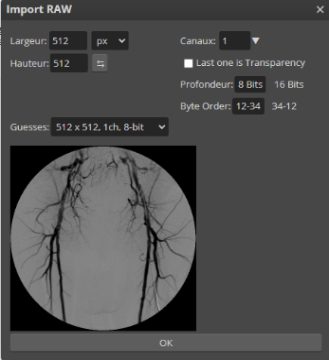
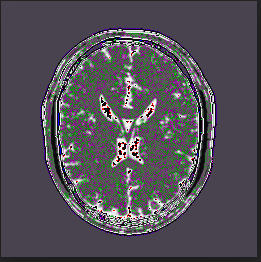
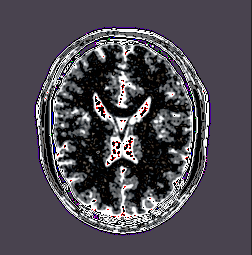
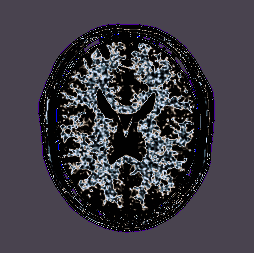
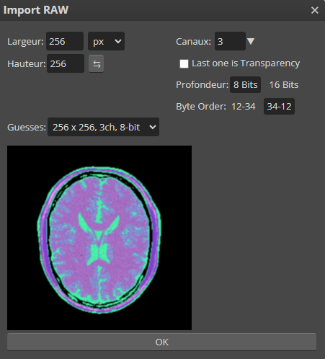
La fonction ***applyLUT*** transforme une image de niveaux de gris en image couleur en remplaçant chaque pixel gris par une couleur définie dans une table LUT, selon son intensité

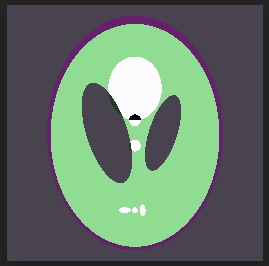
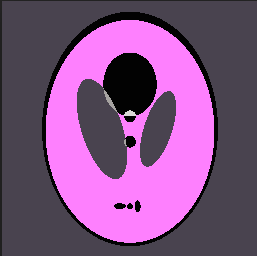
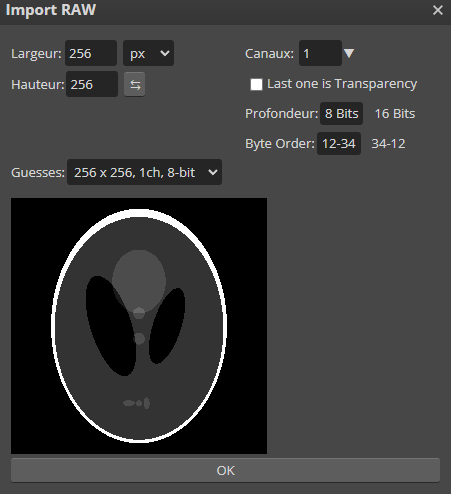
Figure 12: Application de LUT sur l’image.

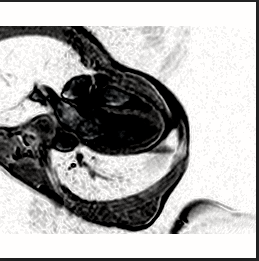
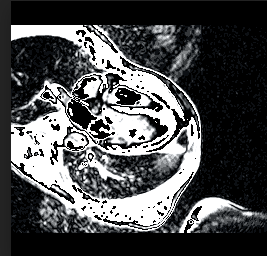
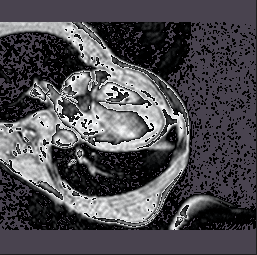
### Tester des filtres LUT sur les différentes images :

Les figures ci-dessus montrent le résultat obtenu après l’appel aux fonctions précédentes









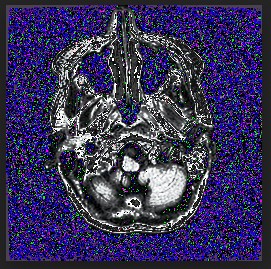
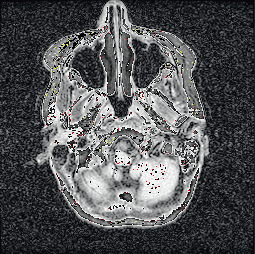
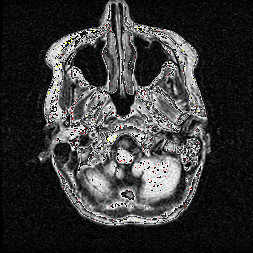
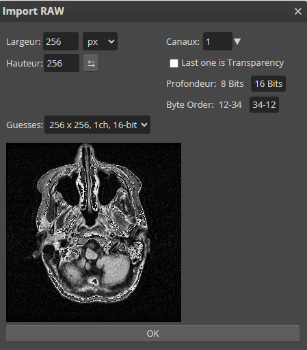


Figure 13: Résultat de l’application de LUT sur les différents images

## **Namespace version 1.1 :**

### 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 pour coder des images en utilisant ***std::vector*,** 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 fichier brut, conversion de type à un autre), et ***ImageRGB***, qui hérite de la class ***Image*** pour traiter les images couleur et appliquer des ***LUT*** spécifiques aux images en niveau de gris.

### 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, unsigned char, …), elle encapsule les dimensions de l’image (\_**width**, \_**height**), 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)***.

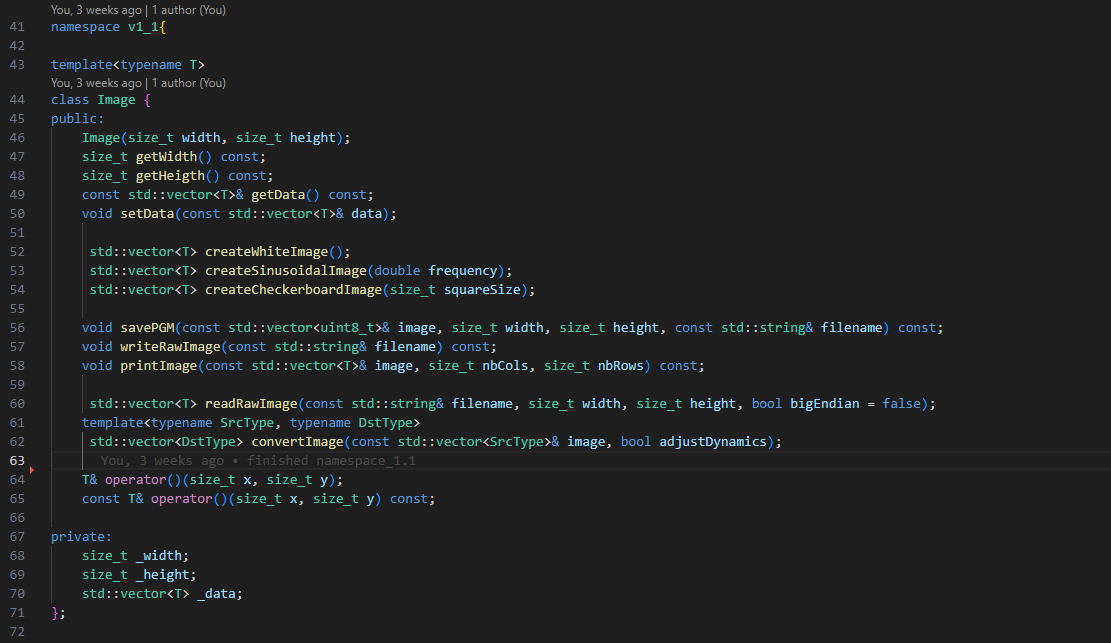


Figure 14: Aperçu du code de la class Image

La class ***ImageRGB*** est une class dérivé de la class ***Image*** qui gère les images en couleurs ***RGB***, elle prend une propriété ***\_rgbData*** pour accéder aux données de pixels d’une image ***RGB***. Elle inclut aussi des méthodes pour lire/écrire des fichiers PPM, charger une LUT binaire, convertir une image RGB en niveaux de gris. Elle contient aussi un constructeur pour créer des images en ***RGB*** à partir d’une image en niveaux de gris à l’aide de ***LUT***

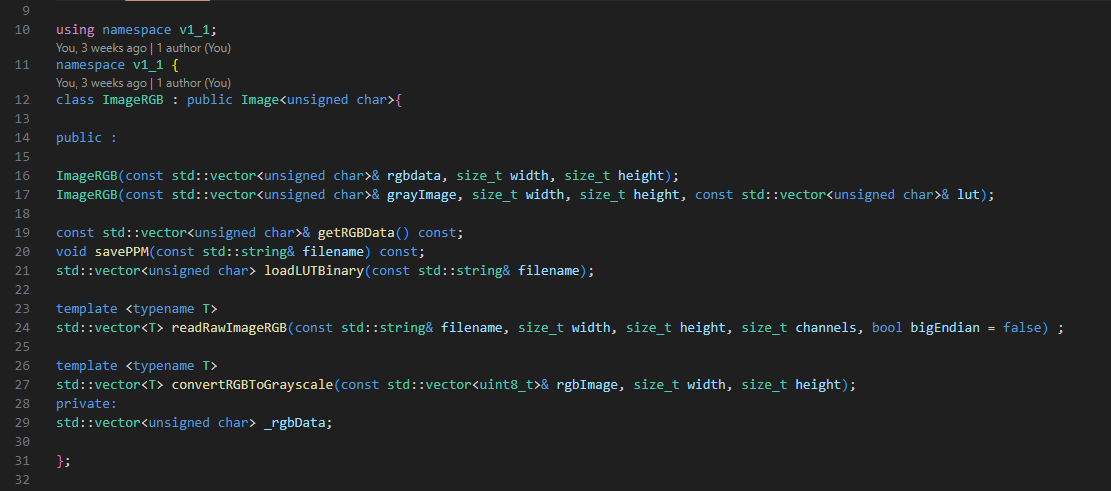


Figure 15: Aperçu de code de la class ImageRGB

## **Namespace version 2.0 :**

## **Conclusion :**