A Simple Image Library

Student ID: 12312710
Name: 胥熠/Brighton
Course: CS219: Advanced Programming
Date: 2025 年 5 月 12 日

目录

1 前言

Project4 作为一个保留节目，延续了前几个学期实现一个 C++ 矩阵类的传统要求，~~似乎是因为手老师似乎比较喜欢 cv::Mat~~。这个 Project 看似简单，实则暗藏玄机：内存是否安全申请和释放？类的设计是否高效优雅，还能满足鲁棒性，易用性？想要同时达成这些条件，其实相对来说是比较挑战的。以及如何在实现一个类的同时，引入一些有意思的东西呢？

2 功能展示

2.1 文件结构

本图像库主要由以下核心文件构成：

- `image.h`：图像类的头文件，包含了图像类的声明和相关函数的声明，模版类函数的实现。
- `image_proc.cpp`：图像处理函数的实现文件，包含了部分对图像进行处理的函数。
- `image_io.cpp`：图像输入输出函数的实现文件，包含了读取和保存图像的函数。

辅助文件包括：

- `demo.cpp`：一个演示程序，展示了库的主要功能和用法。
- `CMakeLists.txt`：用于构建库和演示程序的 CMake 配置文件。

2.2 运行环境

本库使用 C++17 标准编写。运行设备为 Lenovo Legion Y9000P 运行 x86 下的 Windows 11, 使用 Intel i9-13900HX(8 P-cores + 16 E-cores)，24GB DDR5 运存，支持 OpenMP 以及 AVX2 指令集。编译步骤为（上述文件均在 path 路径下，命令行为 Ubuntu 20.04 的 wsl）：

```
wsl *path*$ mkdir build
wsl *path*$ cd build
wsl *path*/build$ cmake .. -DCMAKE_BUILD_TYPE=Release
wsl *path*/build$ cmake --build .
wsl *path*/build$ ./demo
wsl *path*/build$ ./demo input.bmp input2.bmp
```

如果运行设备不支持 OpenMP 和 AVX2 指令集，程序不会编译错误，而会在编译时候提供一个 warning: #warning "AVX2 not enabled" [-Wcpp]。同时在运行时不会运行相关代码。

2.3 库调用指南

首先，我们可以导入一张 24 位无压缩 BMP 图像，获取其基本属性，并在处理后保存。

2.3.1 核心对象解释：Pixel 与 Image

库的核心是两个模板类：

- `ILib::Pixel<T, Channels>`：表示一个像素，其中 `T` 是每个通道的数据类型（如 `unsigned char`, `float`），`Channels` 是通道数。
- `ILib::Image<PixelType>`：表示一个图像，其中 `PixelType` 是具体的像素类型（例如 `ILib::Pixel<unsigned char, 3>` ）。

同时，库的 namespace 为 ILib。如果我们想要创造一个 24 位的 RGB 图像存储对象，我们可以使用 ILib::Image<ILib::Pixel<unsigned char,3»。也可以使用 ILib::ImageU8C3（内置 using 定义）来表示。

2.3.2 图像加载查询与保存

首先，如需要使用图像库，我们需要包含库头文件：#include ”image.h”。

图 1: input.bmp(后续处理使用的图像)

```
ILib::ImageU8C3 image = ILib::loadImageFromFile("input.bmp");
std::cout << "Image loaded: " << input_image.width() << "x" <<
input_image.height()
<< " Channels: " << input_image.getPixelTypeChannels()
<< " ColorSpace: " << static_cast<int>(input_image.getColorSpace()) <<
std::endl;
// 一些处理
ILib::saveImageToFile(image, "output.bmp");
// 其实库不需要用户手动销毁对象，对象会在生命周期结束后自动调用release方法。
image.release();
```

2.3.3 图像操作

亮度调整：支持通过 adjustBrightness、adjustBrightness_simd 函数来实现，也可以通过更简单的 += 操作符来实现。

```
    image += 50;
(or)image.adjustBrightness(50);
(or)image.adjustBrightness_simd(50);
```

图 2: 亮度调整

图 3: 图像融合

图像融合：支持对两张基本参数相同的图像进行混合，库提供的 `blend` 函数允许通过 alpha 权重控制两图的混合比例。操作符 + 也被重载用于两图的等权重混合。

```
ILib::ImageU8C3 image2 = ILib::loadImageFromFile("input2.bmp");
// 权重为: 70% image + 30% image2
ILib::ImageU8C3 blended_image = ILib::blend(image, image2, 0.7f);
ILib::ImageU8C3 blended_image2 = image + image2; //权重各50%
```

灰度转换：支持对 24 位 BMP 图像进行灰度化处理，处理后的图像类型为 8 位灰度图像（通道数为 1）。同时也支持保存为灰度图像。

```
ILib::ImageU8C1 image_gray = image.toGray(); // 转换为灰度图像
ILib::saveImageToFile(image_gray, "output_gray.bmp"); // 支持保存为灰度图像
```

图 4: 灰度图

图 5: ROI 操作

区域图像 (Region of Interest) 操作：可以高效的选中图像的特点子区域，而无需复制像素数据。同时支持基于此区域对原图进行修改。

```
//定义一个ROI区域，这里是图像的中心四分之一部分。
Rect roi = {image.width()/4, image.height()/4,
            image.width()/2, image.height()/4};
// 获取ROI区域
ILib::ImageU8C3 roi_view = image_for_roi.roi(rect);
roi_view += -50;
ILib::saveImageToFile(image, "output_roi.bmp");
```

2.3.4 卷积与滤波

支持通用的 2D 卷积操作 (`filter2D`)，并基于此提供了一系列预定义的图像滤波效果，如高斯模糊、锐化、Sobel 边缘检测和浮雕效果，部分预定核的参数可修改。用户也可以自定义卷积核参与计算。

```
// 5x5 高斯核，标准差 1.5，其中5，1.5可修改
ILib::ImageU8C3 blurred_image = ILib::gaussianBlur(image, 5, 1.5);
// 3x3 Sobel核，阈值 120，true为采用曼哈顿距离，false为欧几里得距离
ILib::ImageU8C1 image_gray = ILib::convertToGrayscale(image);
ILib::ImageU8C1 image_sobel_edge;
ILib::sobelMagnitude(image_gray, image_sobel_edge, 3, 120.0, true);
// 锐化
ILib::ImageU8C3 sharpened_image = ILib::sharpen(image);
// 浮雕
ILib::ImageU8C3 embossed_image = ILib::emboss(image);
// 自定义卷积核示例
ILib::Image<ILib::Pixel<float, 1>> my_krnl(3, 3, ILib::ColorSpace::UNKNOWN);
my_krnl.at(0,0) = {1.1f}; my_krnl(0,1) = {4.5f}; my_krnl(0,2) = {1.4f};
my_krnl.at(1,0) = {1.0f}; my_krnl(1,1) = {-9.0f};my_krnl(1,2) = {1.0f};
my_krnl.at(2,0) = {0.9f}; my_krnl(2,1) = {0.8f}; my_krnl(2,2) = {1.0f};
ILib::ImageU8C3 custom_result;
ILib::filter2D(input_image, custom_result, custom_kernel);
```

图 6: 高斯模糊

图 7: Sobel 边缘检测

图 8: 锐化

图 9: 浮雕

图 10: 自定义卷积核（已黑化）

图 11: 自定义卷积核（已飞升）

2.3.5 错误处理

库设计了很多编译阶段的检查，例如对于 Pixel 对象来说，会检查模板传入参数合理性：

```
static_assert(std::is_arithmetic_v<T> || std::is_enum_v<T>,
              "Pixel type must be arithmetic or enum");
static_assert(Channels > 0, "Pixel must have at least one channel");
```

如果传入的参数不合理，则会在编译阶段会报错。
在 include SIMD 或者 AVX2 指令集函数的时候，会检查 AVX2 指令集是否可用:

```
#ifdef __AVX2__ // 仅在支持 AVX2 时编译此部分
  #include <immintrin.h>
#else
  #warning "AVX2 not enabled"
#endif
```

在对应函数，会判断 ___AVX2___ 是否被定义，并选择性的运行代码：

```
#ifdef __AVX2__
  // ... SIMD 处理代码 ...
#else
  #warning "AVX2 not supported or enabled. Fail to run simd version"
#endif
```

除此之外，基本的函数都提供了运行时异常处理机制，例如在读取图像文件时，如果文件不存在或格式不支持，函数传入对象不符合要求，则都会抛出异常。

3 需求和设计思路

与 Project3 有显著不同的是，Project3 面向对象是纯粹的 24 位 BMP 图像，但这个 Project 的目标是构建一个更通用的图像库。图像可能有多种通道（Channel，我觉得这个词翻译的不好）：RGB 三通道的图像是最经典的，而对于灰度图像来说只有一个通道，还有四通道带透明度信息的图像。

图像还分不同的像素类型，RGB 就是用三个 unsigned char 来表示构成一个像素的三种颜色分量。但除此之外，还有 HSV、CMYK、YCbCr 等不同的图像类型。以 HSV 为例，它的三个通道分别是色相、饱和度和明度。而色相的取值范围是 0-360 的整数（嗯..?），饱和度和明度的取值范围是 0-1 的小数，显然不能用 char 来表示。

在设计核心图像类时，我们面临几个关键问题：如何表示不同类型的图像（灰度图、彩色图、不同颜色空间、不同数据深度）？如何高效且安全地管理像素数据内存？如何提供方便的像素访问和操作接口？在这个领域，C++ 的 Template 非常好用，可以只用写一次函数来支持不同的图像类型。

但实现不同图像类 IO 是相对麻烦的，鉴于这次 Project 主要关注于类的应用，我会用模版类将不同图片形式描述出来，但是 IO 方法只移植 24 位的 BMP 图像。（以及 8 位灰度图像，我会把它按照 24 位的逻辑存储）

3.1 像素表示：Pixel 结构体模板

在上一个 Project 中，我们注意到我们可以有两种存储像素的方式。一种是一条线性像素，整齐的摆放着 RGBRGBRG...（或者 BGR）。另一种多条线性存储，比如说一条负责存储 RRRRR，另外的负责存储其他颜色。乍一看两个存储方式都有优缺点，但是总的来说，第一种的优势更大，因为相对来说，只处理某一个通道的图像操作是比较少的，况且大多数文件格式也是交错方式存储的。在 OpenCV 中，cv::Mat 也是采用交错存储的方式。这里，我依旧把像素作为一个单独的结构体来存储。

又由于一个像素内包含的通道数可能不同，数据类型也会不同。为了实现通用性，我们设计了一个 `Pixel` 结构体模板：

```
template <typename T, size_t Channels> // T 是数据类型，Channels 是通道数。
struct Pixel {
```

```
public:
    // 编译期检查：确保 T 是算术或枚举，Channels > 0
    static_assert(...);

    T data[Channels]; // 存储通道数据

    // 默认构造函数 (初始化为0)
    Pixel() { /* ... */ }
    // 初始化列表构造函数，例如 pixel = {230, 255, 19}
    Pixel(const std::initializer_list<T>& list) { /* ... */ }

    // 通过 operator[] 访问通道 (带边界检查)
    T& operator[](size_t i);
    const T& operator[](size_t i) const;

    // 静态成员和类型别名，方便获取信息
    static constexpr size_t num_channels = Channels;
    using value_type = T;
};
```

这个设计我们可以通过模版存储不同类型的像素,比如说 24 位 RGB。我们就创建 Pixel<unsigned char,3>。对于灰色图或者说卷积核，可以为 Pixel<unsigned char/float,1>。

3.2 图像容器：Image 模版

由于上面我的像素使用了模版类，Image 也要使用模版类。如果把 Pixel 看成某种数据类型的话，我们的任务已经基本转换成了一个矩阵容器。一个矩阵类应该有什么元素？长，宽，以及指向数据数组的指针。但这三个要素还是显得有点不够用。

首先我们需要保证内存的安全性。考虑到一个图像数据可能会被多个对象共享，所以说销毁一个对象的同时，必须要确定这个内存是否还有其他对象在用，才能决定是否释放。手动去 counter++ 和 delete[] 来判断是一个非常折磨的事情，这里我们直接使用了 C++11 起支持的智能指针来管理内存。

其次，有些图像，我们只想要其中的一部分，比如说我们只想要图像的中间部分。我们可以通过深拷贝来实现这个功能，但这样子相对低效。我们可以再用一个指针来指向我们想要的区域的开头。同时，这样的话就需要提前保存图像每一行所占用的字节数。

这样，我们就确定了 Image 类所包含的数据：

- **宽度**：size_t 图像的宽度。
- **高度**：size_t 图像的高度。
- **智能指针**：`std::shared_ptr<T[]>` 指向原始图像数据的开头
- **数据开头指针**：T * 指向期望图像块数据的开头。

- **行大小**：size_t 每行的字节数，用于计算行偏移。
- **图像类型**：一种枚举类，区分这个图像是 RGB、灰度图还是其他类型。

3.3 图像类的操作

确定了 `Image` 类的数据成员后，我们需要给他赋予一些操作，使其真正成为一个易用且功能完备的图像容器。比如说浅拷贝，深拷贝，访问某一个像素的内容，选中 ROI 区域等等。

首先，创建一个对象除了 default 创建一个空图像以及传入长、宽、颜色 tag 的构造外，一个关键的构造函数就是创建某一个图像的区域图像。ROI 继承了原图像的智能指针，行大小以及图像类型，修改了长宽以及指向期望图像块的指针。可以在不用拷贝的情况下，直接对原图像进行操作，非常高效。

对于浅拷贝，也就是指针指向同一个数据块。我们甚至可以不用写代码，编译器会自动生成一个 default 的拷贝构造函数，帮我们把类里面的元素都拷贝一遍。（为了可读性，也可以专门标注出一行来。）而对于深拷贝，我们则需要在把成员变量拷贝一遍的同时，用 memcpy 方法把我们的图像数据拷贝一遍，因为这个方法的性能开销比较大，所以说用 = 赋值的，都是 default 浅拷贝，深拷贝则使用 clone() 方法。

对于像素数据的访问，通过 at(x,y) 来返回坐标处的像素引用。也可以通过 ptr(y) 来返回第 y 行的指针。

此外，还提供了如 `width()`, `height()` 等众多方法来获取图像的各种属性，这样设计的原因当然是不希望外部能够随意的修改 Image 类的成员变量。其中 `getPixelTypeChannels()` 是一个静态 `constexpr` 方法，它能根据模板参数 `PixelType` 在编译期推断出像素的通道数。

最后销毁一个对象则是用 `release()` 方法，这里我们显式的使用了 shared_ptr 的 reset() 方法来告知智能指针这里的引用次数减一，这样我们图像数据的释放就由智能指针来管理了。

集合上述设计要素，这是最终的 Image 类的设计：

```
template <typename PixelType>
class Image {
private:
    size_t width_ = 0;
    size_t height_ = 0;
    size_t steps_in_bytes_ = 0;
    std::shared_ptr<PixelType[]> data_holder_;
    PixelType* p_data_ = nullptr;              // 指向当前图像/ROI的起始数据
    ColorSpace color_space_ = ColorSpace::UNKNOWN;
    // 内部辅助函数，用于分配内存和初始化元数据
    void allocateAndInit(size_t width, size_t height, ColorSpace cs,
                         bool initialize_to_default = false);
public:
    Image() ... * 各种构造函数 *
    // 析构函数 (默认实现，shared_ptr会自动处理内存)
    ~Image() = default;
    // 赋值操作符 (默认浅拷贝)
```

```
        Image<PixelType>& operator=(const Image<PixelType>& other) = default;
        Image<PixelType>& operator=(Image<PixelType>&& other) noexcept = default;
        // 创建图像副本 (深拷贝)
        Image<PixelType> clone() const;
        // 将当前图像数据深拷贝到目标图像
        void copyTo(Image<PixelType>& dst) const;
        // 释放资源，图像变为空
        void release();
        // 重新创建图像 (会释放原有数据)
        void create(...);
        * 访问像素函数行指针函数*
        * 获取图像属性函数 *
        // ROI操作符
        Image<PixelType> roi(const Rect& roi_rect);
        Image<PixelType> operator()(const Rect& roi_rect);
        // 重载调整操作符
        Image<PixelType>& operator+=(int value_offset);
    };
}
```

写到后面的时候，Image 类已经在我眼里感觉比较庞大而且复杂了。但是，查阅 OpenCV 官方的 cv::Mat Class Reference 的使用文档后，我只觉得相形见绌。他的 Mat 单论构造方法就有二十几种，而且他的 Mat 类的设计也相当复杂，提供了相当多的功能以及 sample，的确是一个复杂且优秀的图像库。

3.4 图像处理方法

基本原理和上个 Project 相同，请见 reference:Project 3。

4 如何科学优化

本次 Project 很多优化方法和 Project2,3 所使用的方法相同，因此只是简略的介绍。

4.1 编译优化

在 CMake 构建系统中，针对 Release 模式配置了以下优化标志：

- `-O3`：启用 GCC 最高级别的常规优化。
- `-march=native`：指示编译器针对当前构建机器的 CPU 特性进行优化。
- `-mavx2`：显式启用 AVX2 (Advanced Vector Extensions 2) 指令集。

对于上述编译优化策略，我们在 CMakeLists.txt 中进行了如下配置，在非 Release 模式的时候，程序不会优化：

```
set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} -O3 -march=native -mavx2")
set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} -g -O0")
```

4.2 SIMD 优化

对于 addBrightness 功能，我们提供了两种实现方式：一种就是正常使用的 for 循环（addBrightness），另一种则是使用 AVX2 指令集来实现 (addBrightness_simd)。在实际测试中，AVX2 确实带来了部分的性能提升。

```
// 使用AVX2指令集实现示例，已简化逻辑的实现细节
void addBrightness_simd(int value_offset) {
    __m256i offset = _mm256_set1_epi8(value_offset);
    for (size_t i = 0; i < width_ * height_; i += 32) {
        __m256i pixels = _mm256_loadu_si256((__m256i*)&data_[i]);
        pixels = _mm256_add_epi8(pixels, offset);
        _mm256_storeu_si256((__m256i*)&data_[i], pixels);
    }
}
```

4.3 OpenMP 并行化

OpenMP 依旧伟大，能够几行代码加速几倍。库几乎在所有的处理函数中都使用了 OpenMP 来实现并行化。

```
#pragma omp parallel for schedule(static)
for (size_t y = 0; y < height; ++y) {
    // ... 循环处理某些元素 ...
}
```

4.4 软拷贝以及高效 ROI 实现

如前所述，如果不是明确要求深拷贝，Image 类的赋值操作符会使用默认的浅拷贝。这样做的好处是避免了不必要的内存拷贝，同时如果确实需要副本，也可以调用 clone() 函数来深拷贝。

同时，ROI 的实现用了一个非常巧妙的方式。我们指定一个区域，只需要修改图像的宽高和指向数据开头的指针即可，运行成本几乎可以忽略。具体原理 Monad 的 Report 中讲得比较详细，还有配图，可以见 reference [1]。

4.5 性能测试

我们的运行条件以及在上面说明，这里，我们延续使用 Project3 中的分辨率为 9725 * 4862 的 24 位无压缩 BMP 图像进行测试。我们分成以下三种 cases: 完全无任何优化（编译时使用 cmake .. -DCMAKE_BUILD_TYPE=debug），O3 + OpenMP(调用 addBrightness 函数), O3 + OpenMP + AVX2（调用 addBrightness_simd 函数）。程序运行五次时间取平均值。

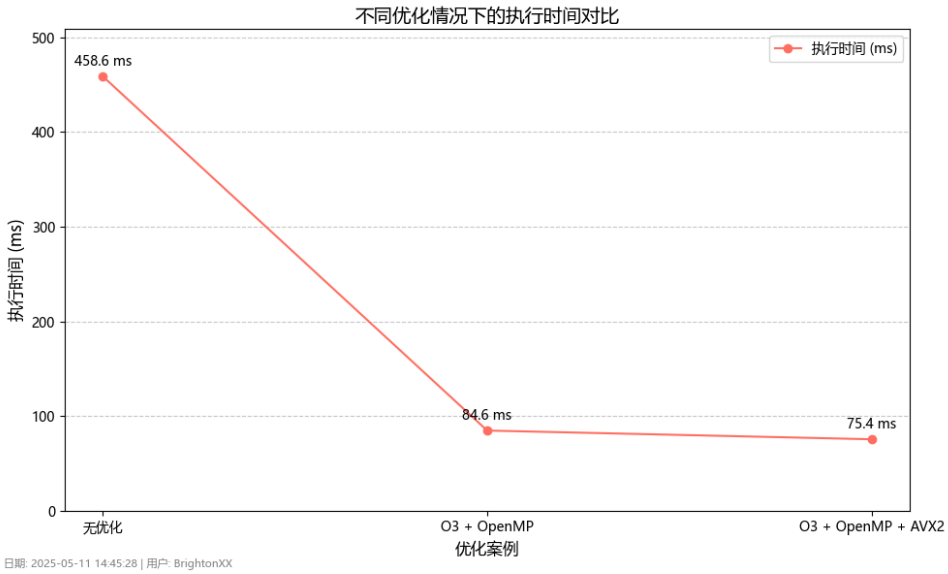

图 12: 性能测试结果

可以看到，手动实现的 SIMD 指令集确实能带来性能提升，但是比较小。但无论如何，我们还是看到对比起优化前的速度，程序运行速度提升了大约五倍。

5 AI 辅助了哪些工作

Move fast and break things.

C++ 语言实在是太复杂了，很多实用的语法其实我根本没接触过。但是在 Gemini 的辅助下，我能够极快速的上手 C++，虽然有些语法我并不清楚它是怎么实现的，但 anyway, it works。这在很多情况下就够用了。

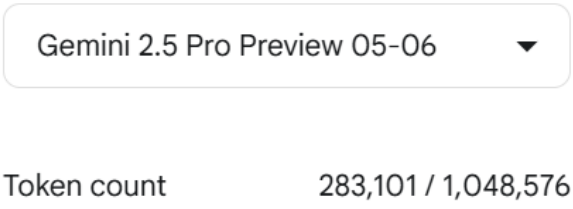

图 13: 30 万个 tokens 的上下文！

我这次写 Project 只使用了一个对话，看看 Gemini 2.5Pro 面对巨量文本，数据，要求的时候，它还能不能正确的反应。特别是写类相关的时候，AI 能不能基于前面设计好的内容，快速且正确的移植新方法：事实证明它做到了，而且目测免费的大模型只有它能做到，因为它支持 1M 的上下文。

比如说，在我设计好大致的 Pixel 和 Image 类，以及一些简单的方法后，对话上下文已经来到了十万多。我叫 Gemini 帮我移植 fliter2D，即通用卷积核的函数，它在第一遍的时候就提供了能够正确运行的函数。这其实是非常恐怖的，说明它不仅能够正确调用我设计的类的方法，

还能够提供一整个读取，卷积，保存的工作链。~~真的要失业子吗~~。

6 结语

在这个 Project 中，我感受到了 C++ 极其复杂的语言规则，里面有一堆很新很快的函数和方法。也体会到了写 template 的方便，对于不同的图像类型，我只用写一次函数，就能支持不同的图像类型，剩下的全部交给编译器来处理。

同时，shared_ptr 是一个非常伟大的发明。它真的一定程度上解决了跨对象内存管理的难点，在对象生命周期结束或不再被引用时自动释放所管理的内存的时候，它能够帮我自动释放内存，而不是我手动计数并且释放。~~说不定它早出几十年，Java 就会黯然失色。~~

总之，我个人对我的库的设计还是比较满意的。一个有趣的设计点是复用了 `Image` 类来存储卷积核的数据。这种做法简化了卷积操作的接口，也允许用户方便地创建和检视自定义的卷积核。当然，这也解释了为何像 OpenCV 这样的库将其核心图像容器命名为 `cv::Mat` 而非 `cv::Img`，因为矩阵更具通用性，可以自然地包含卷积核这类非典型“图像”的数据结构，从而减少了命名上的潜在歧义。

本项目仍存在一些待完善之处，例如未能支持更多的图像文件格式的 IO 操作，以及不同颜色空间之间的转换功能。若能补齐这些方面，还能实现更多很炫酷的操作。

同时，与先前 Project 一样，在 DDL 结束后我也会将本次 Project 中涉及到的所有源码开源，放在 https://github.com/BrightonXX/SUSTech-CPP-Project 下。

感谢各位能读到这里。

References

参考文献

[1] Yan W.Q. *CS205 Project5: Matrix Class*. Southern University of Science and Technology, 2022. [Source code]. Available: github.com/YanWQ-monad/CS205_Project5

[2] OpenCV (Open Source Computer Vision Library). *cv::Mat Class Reference*. Available: docs.opencv.org/master/class_Mat.html

[3] Brighton *Project3: A Simple Image Processor*. Southern University of Science and Technology, 2025. [Source code]. Available: github.com/BrightonXX/SUSTech-CPP-Project