Image Processing in C

TI202I - Algorithmic and Data Structures 1
C Programming Project

Proposed by : Rado Rakotonarivo

Efrei Paris
P1/P1-INT
2024 - 2025

General Instructions and Information

➔ This project is to be done exclusively in C. You must use the functions of the standard library of the C language.

➔ Files enclosed with the project:

Sample images has changed:
- A `barbara_gray.bmp` file in grayscale (8 bits) to test the functions of part 1.
- A `flowers_color.bmp` file in color (24 bits) to test the functions of part 2.

➔ Team organization:

- Projects are to be carried out in pairs (**only one trio is allowed, only for an odd number of students**).
- The list of teams is to be submitted to the teachers at the end of the first project follow-up session.
- Each team uses a Git repository to manage the project's source code.

➔ Project submission: Two submissions are required for this project.

- An intermediate submission containing a minimal version of the project (not required to be functional) with essentially the data structures and functions to be implemented.
- A final submission containing the complete and functional project.

➔ Key timestamps:

- Date of publication of the subject (Part 1): week of 17/03/2025.
- Date of follow-up session 1: week of 17/03/2025.
- Date of publication of the subject (Part 2): week of 31/03/2025.
- Date of publication of the subject (Part 3): week of 21/04/2025.
- Date of intermediate submission: Sunday **27/04/2025 at 11:59 PM**.
- Date of follow-up session 2: week of 28/04/2025.
- Date of follow-up session 3: week of 19/05/2025.
- Date of final submission: Sunday **25/05/2025 at 11:59 PM**.
- Date of defense: week of 26/05/2025.

➔ Submission methods will be communicated to you later.

➔ Evaluation modalities:

- A detailed evaluation grid will be provided to you.
- Final project grade = Defense grade + Source code grade.
- It is reminded that the project grade counts for 20% of the final grade of the module.
- It is not impossible for members of the same team to have different grades depending on the efforts made in carrying out the project.

Contents

Préambule

What is an image

From an algorithmic point of view, an *image* can be represented as a matrix, where each element corresponds to a pixel. Although this modeling is not an absolute truth, it is nevertheless a standard approach for the processing and manipulation of images in computer science. Thus, in a digital context, an image is often stored and manipulated as a two-dimensional array, each cell containing a value representing the light intensity (in the case of a grayscale image) or a triplet of values corresponding to the RGB components (red, green, blue) for a color image.

But what is a pixel in this context?

A *pixel* is the fundamental unit of a digital image, represented in a matrix structure where its coordinates determine its position in the image. Each pixel can be considered as an element of the matrix, indexed by two indices corresponding to its row and column. By zooming in on an image, one can observe this structure: as the scale increases, the image loses continuity to reveal a grid of small colored squares, which are none other than the pixels.

This matrix representation is essential in many image processing algorithms, whether it be filtering, edge detection, or compression. It allows mathematical transformations to be applied directly to the pixel values in order to extract, modify, or analyze the visual information contained in the image.

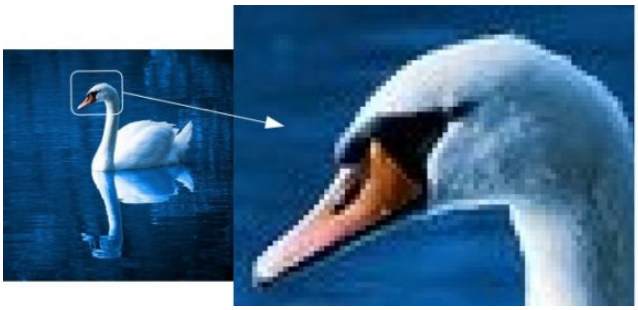


Figure 1 – Zooming into the image allows to observe the matrix structure in which lies the pixel data (Source : DataCorner).

Image processing

Image processing refers to the set of methods and techniques used to modify, analyze, and enhance digital images. It plays a central role in many fields, such as photography, artificial intelligence, medicine, or computer vision. Through these processes, it is possible to apply various treatments to images, such as color adjustment, contrast enhancement, noise reduction, or object detection and recognition.

Image processing is ubiquitous in our daily lives and plays a key role in concrete applications. For example:

- On smartphones: Instagram filters, automatic photo enhancement, or night mode that adjusts brightness are based on image processing algorithms.
- In video games: graphic rendering techniques optimize textures and visual effects for better immersion.
- In medicine: MRI and radiography analysis relies on specific treatments that improve diagnostic accuracy.
- In autonomous vehicles: onboard cameras use computer vision to identify obstacles, detect traffic signs, and ensure passenger safety.

This process is not recent. As early as the 1960s, it was used on powerful computers to analyze satellite and medical images. With the evolution of computing capabilities, these techniques have become more accessible and are now available on personal computers, and even directly on smartphones. Today, modern algorithms, often based on artificial intelligence, allow us to go even further. For example, neural networks are able to restore blurry photos, create ultra-realistic images from textual descriptions, or generate impressive special effects in cinema and animation.

(a) Original image **(b)** Grayscale image **(c)** Blurred image

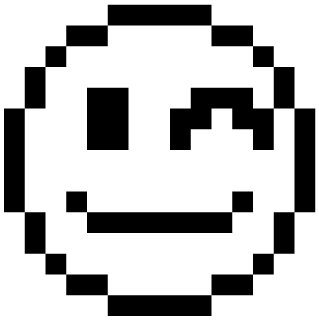
Figure 2 – Examples of image processing operations: conversion to grayscale and application of a blur filter.

Digital images representation

A digital image is a discrete representation of a visual image, that is, an image that is decomposed into *pixels*, which are the elementary points of the image. Each pixel is characterized by a color, which can be defined by a color model (RGB, CMYK, grayscale, etc.). The resolution of an image corresponds to the number of pixels that compose it, and determines the quality of the image. The higher the resolution, the more detailed and precise the image.

The image data is then stored as a matrix of pixels. Figure 2.3 shows a binary pixel art image, in which each pixel is represented in the image matrix: "1" indicates a black pixel and "0" indicates a white pixel. This binary image is of course a very simplified image format but there are others that can be much more complex.

The most common image formats are JPEG, PNG, GIF, BMP, TIFF, etc. Each image format has its own characteristics and specifications, which determine how the data is encoded and stored. Some formats are more suitable for image compression, others for transparency or animation. The choice of format depends on the use of the image and the constraints related to its distribution or storage.



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	1	1	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	1	0	0	1	1	0	0	0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0	1	1	0	1	1	0	1	0
1	0	0	0	1	1	0	0	1	0	0	0	1	0	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0
0	1	0	0	1	1	1	1	1	1	1	0	0	1	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	1	1	0	0	0	0	0	1	1	0	0	0	0
0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0

Figure 3 – Pixel art image and its matrix representation.

Applying transformations to an image therefore involves manipulating the pixel values of the matrix. This can be done by modifying the color components, applying filters, performing mathematical operations, or using image processing algorithms. These operations allow to modify the appearance of the image, extract information from it, or improve its quality.

In this project, we will choose the **BMP** image format to represent digital images.

The BMP format

The BMP format is an uncompressed bitmap image format that stores image data as pixels. It is simple to handle and understand, making it an ideal choice for implementing image processing algorithms. A BMP file consists of several distinct sections, each containing essential information for interpreting the image:

File header : contains general information about the file, such as its size, signature, and the location of the image data.

Image header : contains specific information about the image, such as its width, height, the number of bits per pixel, and the compression type.

Color table : contains the colors used in the image, as palettes or lookup tables (**This part is only for images with color depth less than or equal to 8 bits**).

Image pixel data : makes up the rest of the file and contains the pixel values of the image, stored in a pixel array.

File header and Image header

The header of a BMP image, composed of the file header and the image header (also called info header), occupies the first 54 bytes of a BMP image. It contains essential information about the image.

The Table 1 below summarizes how the information contained in the header of a BMP image is organized.

Field	Offset	Size	Description
Signature	0	2	File signature (BM)
File size	2	4	Total file size in bytes
Reserved 1	6	2	Reserved field (0)
Reserved 2	8	2	Reserved field (0)
Data offset (`offset`)	10	4	Indicates where the pixel matrix begins
Header size	14	4	Image header size (40)
Image width (`width`)	18	4	Image width in pixels
Image height (`height`)	22	4	Image height in pixels
Color planes	26	2	Number of color planes (1)
Color depth (`colorDepth`)	28	2	Number of bits per pixel (1, 4,16, 24 or 32)
Compression	30	4	Compression type (0 for uncompressed)
Data size (`dataSize`)	34	4	Image data size
Horizontal resolution	38	4	Horizontal resolution in pixels per meter
Vertical resolution	42	4	Vertical resolution in pixels per meter
Number of colors	46	4	Number of colors in the palette
Important colors	50	4	Important colors

Table 1 – BMP image header structure

For now the most important information in the header of a BMP image are the width and height of the image, the color depth, the data offset, and the data size. These information allow us to determine the size of the pixel matrix, as well as how the image data is stored.

It should be noted that if the color depth of the image is less than or equal to 8 bits, a color table is included in the image header. This table contains the colors used in the image, as palettes or lookup tables. It allows to associate a color value to each pixel, which is then used to display the image.

Image pixel data

Image data is stored in a pixel array, where each pixel is represented by a color value. This matrix starts at the `offset` indicated in the file header, and is organized to respect the order of the rows and columns of the image. Each pixel is represented by one or more bytes, depending on the color depth `colorDepth` of the image.

Basic operations on images

Image processing consists of applying operations on the pixels of the image, by modifying their values or combining them to obtain a specific result. These operations can be simple, such as inverting the colors or resizing the image, or more complex, such as edge detection or object segmentation.

Basic image operations can be classified into several categories, depending on their purpose and effect on the image. The following list presents some examples of common operations:

Color operations : modify the color components of the pixels, such as color inversion, conversion to grayscale, or hue correction.

Geometric operations : modify the geometry of the image, such as resizing, rotation, cropping, or scaling.

Filtering operations : apply filters to the image to improve its quality, such as blur, sharpening, contrast, or brightness.

Thresholding operations : segment the image based on predefined thresholds, to detect objects or contours.

Objectives and project breakdown

What is expected from you

In this project, you are required to implement an image processing program in C. A minimal version of the program should be able to read BMP images, apply transformations to these images, and save the results in new BMP files using a command-line interface.

To do this, it is necessary to consider the following points:

- The memory representation of a BMP image: how to store image data in memory, using appropriate data structures.
- Reading and writing BMP files: retrieve image data from a BMP file and save the results in a new BMP file.
- Developing basic image operations: implement image processing operations.
- Using a command-line interface (in the form of a menu) allowing the user to interact with the program.

From this minimal version, you will have the opportunity to extend the program's features by handling color images, implementing more advanced operations, and improving the user interface. You will also be able to explore more advanced image processing techniques, such as edge detection, object segmentation, or image restoration.

Project breakdown

The project will be divided into several parts, each corresponding to a specific set of features. Each part can be implemented independently and then integrated into the main program.

The project will be broken down as follows:

- **Part 1:** Grayscale image (8 bits) → Simplified representation of an image and getting along with basic operations.
- **Part 2:** Color image (24 bits) → More complex representation of an image and extension of basic operations to color images.
- **Part 3:** Advanced features and improvements.

At the final submission, you will have the choice between two implementations of the data structures storing the images. You can either keep two distinct types for grayscale and color images, or use a single type for both types of images and adapt the functions accordingly.

First user interface

User interface will be in the form of a command-line menu. The user will be able to choose from several options to perform operations on images, such as opening a file, applying filters, saving the results, etc. Each option will be associated with a specific functionality of the program, which will be implemented internally.

The menu should be clear and intuitive, displaying the different available options and guiding the user in their choices. It should allow the user to navigate easily between the different features of the program, offering the possibility to go back or quit at any time.

Here is a possible example of the menu for the program:

```
Please choose an option:
    1. Open an image
    2. Save an image
    3. Apply a filter
    4. Display image information
    5. Quit
>>> Your choice: 1
File path: barbara_gray.bmp
Image loaded successfully!
Please choose an option:
    1. Open an image
    2. Save an image
    3. Apply a filter
    4. Display image information
    5. Quit
>>> Your choice: 3
Please choose a filter:
    1. Negative
    2. Brightness
    3. Black and white
    4. Box Blur
    5. Gaussian blur
    5. Sharpness
    6. Outline
    7. Emboss
    8. Return to the previous menu
>>> Your choice: 1
Filter applied successfully!
Please choose an option:
    1. Open an image
    2. Save an image
    3. Apply a filter
    4. Display image information
    5. Quit
>>> Your choice: 2
File path: barbara_gray_negative.bmp
Image saved successfully!
Please choose an option:
    1. Open an image
    2. Save an image
    3. Apply a filter
    4. Display image information
    5. Quit
>>> Your choice: 5
```

1

Part 1 : 8-bit Grayscale Image Processing

This first part will help you get familiar with reading and writing images. It will be done exclusively with 8-bit grayscale images. Grayscale images are images that require only one color channel to store the color of each pixel.

Each pixel is therefore represented by a single byte (8 bits) that stores the value of the pixel's color (or intensity). The possible values for a grayscale pixel are between 0 and 255 where 0 represents black and 255 represents white. Intermediate values then represent different levels of gray.

All image processing functions in this part should be tested with the `barbara_gray.bmp` image provided to you. This image is an 8-bit grayscale image and does not require *data alignment* (this feature will be presented later).

1.1 The `t_bmp8` structure

We will define a structured type `t_bmp8` to represent an 8-bit grayscale image. This type is defined in the file `bmp8.h` as follows:

```
typedef struct {
    unsigned char header[54];
    unsigned char colorTable[1024];
    unsigned char * data;

    unsigned int width;
    unsigned int height;
    unsigned int colorDepth;
    unsigned int dataSize;
} t_bmp8;
```

- `header` : represents the BMP file header. This header is 54 bytes long.
- `colorTable` : represents the image's color table. This table is 1024 bytes long. **We remind you that for an 8-bit depth image, the color table is mandatory.**
- `data` : represents the image's data. This data is stored as an array of bytes.
- `width` : represents the image's width in pixels (Located at offset 18 of the header).
- `height` : represents the image's height in pixels (Located at offset 22 of the header).

- `colorDepth` : represents the image's color depth. For a grayscale image, this value is equal to 8 (Located at offset 28 of the header).
- `dataSize` : represents the size of the image's data in bytes (Located at offset 34 of the header).

1.2 Reading and Writing an 8-bit Grayscale Image

Once the `t_bmp8` type is defined, we can write the functions to read and write an 8-bit grayscale image. These functions will be defined in the `bmp8.c` file. Here is the prototype of the corresponding functions:

```
t_bmp8 * bmp8_loadImage(const char * filename);
void bmp8_saveImage(const char * filename, t_bmp8 * img);
void bmp8_free(t_bmp8 * img);
void bmp8_printInfo(t_bmp8 * img);
```

1.2.1 The `bmp8_loadImage` function

This function allows you to read an 8-bit grayscale image from a BMP file whose name (*path*) is provided by the `filename` parameter. This function will dynamically allocate memory to store an image of type `t_bmp8`, initialize the fields of this image, and return a pointer to this image. If an error occurs while reading the file, or if the image is not 8 bits deep, the function will display an error message and return `NULL`.

The `width`, `height`, `colorDepth`, and `dataSize` fields will be extracted from the header to initialize the corresponding fields of the `t_bmp8` image. Refer to Table 1 for the offsets of the header fields.

In order to write this function, refer to the `fopen`, `fread`, `fclose`, and `malloc` functions of the C standard library.

Accessing information stored in the BMP image file header

How to access the information stored in the BMP image file header? The following code shows how to read the image width from the BMP image file header:

```
// Assuming the BMP image file is open and the header is stored in the header array
unsigned char header[54];

// We know that the image width is stored at offset 18 of the header
unsigned int width = *(unsigned int *)&header[18];
```

This last line of code allows you to position yourself at offset 18 of the header and read the next 4 bytes (the size of an `unsigned int`) to obtain the image width. It is equivalent to:

```
unsigned int width = *(unsigned int *)(header + 18);
```

Actually, simple access to `header[18]` is not enough to read the image width because the width is stored on 4 bytes and the `header` array is an array of `unsigned char`, i.e. `header[18]` contains only one byte of data. To access the width, we must therefore read the next 4 bytes starting from offset 18. We can thus generalize this instruction to read the other fields of the header.

1.2.2 The `bmp8_saveImage` function

This function allows you to write an 8-bit grayscale image to a BMP file whose name (*path*) is provided by the `filename` parameter. This function takes a pointer to an image of type `t_bmp8` as a parameter and writes this image to the file. If an error occurs while writing the file, the function will display an error message.

In order to write this function, refer to the `fopen`, `fwrite`, and `fclose` functions of the C standard library.

1.2.3 The `bmp8_free` function

This function allows you to free the memory allocated to store an image of type `t_bmp8`. This function takes a pointer to an image of type `t_bmp8` as a parameter and frees the memory allocated for this image. This function should be called at the end of the program to free the memory allocated for an image.

1.2.4 The `bmp8_printInfo` function

This function allows you to display the information of an image of type `t_bmp8`. This function takes a pointer to an image of type `t_bmp8` as a parameter and displays the information of this image. The information to be displayed is as follows:

- The image width.
- The image height.
- The image color depth.
- The size of the image data.

An example of the expected display is as follows:

```
Image Info:
    Width: 800
    Height: 600
    Color Depth: 8
    Data Size: 480000
```

1.3 Image Processing Functions

Once your program is able to read and write an 8-bit grayscale image, you can start implementing image processing functions. In this part, we will implement the following functions:

```
void bmp8_negative(t_bmp8 * img);
void bmp8_brightness(t_bmp8 * img, int value);
void bmp8_threshold(t_bmp8 * img, int threshold);
```

1.3.1 The `bmp8_negative` function

This function inverts the colors of a grayscale image. It takes a pointer to an image of type `t_bmp8` as a parameter and subtracts the value of each pixel in this image from 255. For example, if the value of a pixel is 100, the value of this pixel after inversion will be 155.

(a) Original image.

(b) Result image.

Figure 1.1 – Applying an inversion with the `bmp8_negative` function.

1.3.2 The `bmp8_brightness` function

This function allows you to modify the brightness of a grayscale image. It takes a pointer to an image of type `t_bmp8` and an integer `value` (which can be negative) as parameters. For each pixel of the image, the function adds the `value` to the pixel value. Note that the value of a pixel cannot exceed 255 or be less than 0. If a pixel value exceeds 255, it will be set to 255. If a pixel value is less than 0, it will be set to 0.

(a) Original image.

(b) Result.

Figure 1.2 – Applying a brightness correction of `value`=50 with the `bmp8_brightness` function.

1.3.3 The `bmp8_threshold` function

This function transforms a grayscale image into a binary image. It takes a pointer to an image of type `t_bmp8` and an integer `threshold` as parameters. For each pixel of the image, if the pixel value is greater than or equal to `threshold`, the pixel value will be set to 255. Otherwise, the pixel value will be set to 0.

(a) Original image.

(b) Result.

Figure 1.3 – Turning an image into black and white with `threshold`=128 with the `bmp8_threshold` function.

1.4 Image filtering

Image filters are operations that apply a *mask* to an image to modify the pixel values. This mask, also called a *kernel*, is a square matrix of odd size. For each pixel of the image, the filter calculates the new pixel value

by applying the mask to the neighboring pixels of the pixel in question. The mask is chosen according to the desired effect. The most common masks are used to blur an image, detect edges, enhance edges, etc.

The operation that applies the mask to the image is called *convolution*. For each pixel of the image, the convolution calculates the new pixel value by multiplying the values of the neighboring pixels by the values of the mask, then summing the results. Take a look at this page to see the effect of different masks on an image.

The following formula expresses the convolution of an image I by a kernel K to obtain a new image I':

$$I'_{x,y} = \sum_{i=-n}^{n} \sum_{j=-n}^{n} I_{x-i,y-j} \times K_{i,j}$$

where $I'_{x,y}$ is the value of the pixel at position (x, y) of the resulting image, $I(x, y)$ is the value of the pixel at position (x, y) of the original image, $K(i, j)$ is the value of the (i, j) of the kernel K and n **is the size of the mask divided by 2.**

For instance, if we have two 3×3 matrices, the first one a portion of the image I and the second one a kernel K, the convolution reverses the rows and columns of the mask and performs multiplications on elements of the same indices then sums them up. The element at coordinates (x, y) of the resulting image is then:

$$I'_{x,y} = \begin{bmatrix} I_{x-1,y-1} & I_{x-1,y} & I_{x-1,y+1} \\ I_{x,y-1} & I_{x,y} & I_{x,y+1} \\ I_{x+1,y-1} & I_{x+1,y} & I_{x+1,y+1} \end{bmatrix} * \begin{bmatrix} K_{-1,-1} & K_{-1,0} & K_{-1,1} \\ K_{0,-1} & K_{0,0} & K_{0,1} \\ K_{1,-1} & K_{1,0} & K_{1,1} \end{bmatrix}$$

$$\begin{aligned} I'_{x,y} = {} & I_{x+1,y+1} \times K_{-1,-1} + I_{x+1,y} \times K_{-1,0} + I_{x+1,y-1} \times K_{-1,1} + \\ & I_{x,y+1} \times K_{0,-1} + I_{x,y} \times K_{0,0} + I_{x,y-1} \times K_{0,1} + \\ & I_{x-1,y+1} \times K_{1,-1} + I_{x-1,y} \times K_{1,0} + I_{x-1,y-1} \times K_{1,1} \end{aligned}$$

Here are the filters we will implement in this part:

- `box_ blur` with the following kernel :

$$\frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

- `gaussian_blur` with the following kernel :

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

- `outline` with the following kernel :

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

- `emboss` with the following kernelt :

$$\begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

- `sharpen` with the following kernel :

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

1.4.1 The `bmp8_applyFilter` function

Write a function with the following prototype:

```
void bmp8_applyFilter(t_bmp8 * img, float ** kernel, int kernelSize);
```

which applies a filter to a grayscale image. This function takes a pointer to an image of type `t_bmp8`, a matrix of floats `kernel` representing the filter kernel, and an integer `kernelSize` representing the kernel size. The kernel size is always odd. The function applies the filter to the image using convolution and modifies the pixel values of the image accordingly.

Remarks: In order to facilitate the implementation, you have the possibility to traverse the kernel using a nested loop whose indices vary from $-n$ to n. Since the mask is of odd size, the center of the mask is the element at index $(0,0)$. **For now we do not handle the edges of the image:** that is, the pixels at the edge of the image will not be modified by the filters. (We start with the pixels from $(1,1)$ and stop at $(width-2, height-2)$).

1.5 Usage Example

The following figures illustrate the application of the `bmp8_applyFilter` function with the different kernels presented earlier.

(a) `box_blur`. **(b)** `gaussian_blur`. **(c)** `outline`. **(d)** `emboss`. **(e)** `sharpen`.

Figure 1.4 – Applying different filters using the `bmp8_applyFilter` function.

2

Part 2 : 24-bit Image and Color Image Processing

2.1 24-bit Image

24-bits images are images that contain 3 color channels: red, green, and blue (RGB). Each channel is encoded on 8 bits, which gives 256 possible values for each channel. By combining these 3 channels, we obtain $256^3 = 16777216$ different colors. For a `.bmp` file, these 24 bits correspond to the value of the color depth (`colorDepth`).

This format stores the colors directly (therefore does not require a `color_table`), which means that each pixel contains the complete color information. It can be noted that a grayscale image stored in 24 bits is an image where the 3 color channels are equal. The data of a 24-bit image can therefore be viewed as an array of pixels, each pixel being represented by 3 bytes, one byte for each color channel.

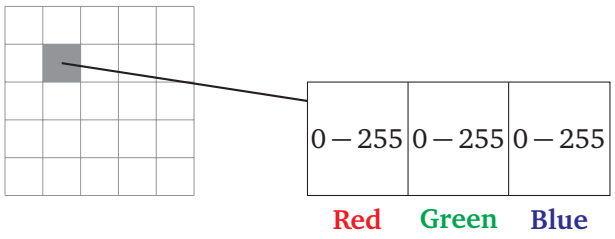


Figure 2.1 – Representation of a 24-bit image. The zoomed pixel is represented by 3 bytes, one for each color channel.

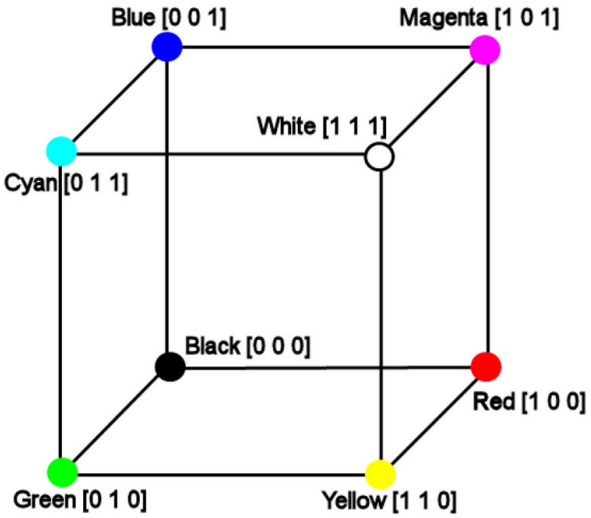


Figure 2.2 – RGB color space (source : ResearchGate)

2.2 The `t_bmp24` data type

Let us consider the data type `t_bmp24` as a structure that contains all the information necessary to represent a 24-bit image. This structure will allow us to easily manipulate the image data and access the different fields of the image. It will contain the following fields:

- `header` : of type `t_bmp_header` which contains the header of the image.
- `header_info` : of type `t_bmp_info` which contains the information of the image.
- `width` : of type `int` which contains the width of the image.
- `height` : of type `int` which contains the height of the image.
- `colorDepth` : of type `int` which contains the color depth of the image. (Here 24 bits)
- `data` : a matrix of `t_pixel` which contains the data of the image.

Thus the type `t_bmp24` is defined as follows:

```
typedef struct {
    t_bmp_header header;
    t_bmp_info header_info;
    int width;
    int height;
    int colorDepth;
    t_pixel **data;
} t_bmp24;
```

2.2.1 The `t_bmp_header` and `t_bmp_info` data types

The `t_bmp_header` and `t_bmp_info` types are types that represent the header and information of an image in BMP format, respectively, and detail the information in table 1. These types are defined as follows:

```
typedef struct {
    uint16_t type;
    uint32_t size;
    uint16_t reserved1;
    uint16_t reserved2;
    uint32_t offset;
} t_bmp_header;

typedef struct {
    uint32_t size;
    int32_t width;
    int32_t height;
    uint16_t planes;
    uint16_t bits;
    uint32_t compression;
    uint32_t imagesize;
    int32_t xresolution;
    int32_t yresolution;
    uint32_t ncolors;
    uint32_t importantcolors;
} t_bmp_info;
```

Where the `uint16_t`, `uint32_t` and `int32_t` types are unsigned integer types of 16 bits, 32 bits and 32 bits respectively. These types are defined in the `stdint.h` library.

Using these structured types to represent image headers is a common practice in C programming. They allow to store the information in an organized way and to facilitate the manipulation of image data. These fields will therefore replace the character array `header` that we had defined previously for the type `t_bmp8`.

2.2.2 The `t_pixel` data type

The `t_pixel` type is a type that represents a pixel of the image. It contains 3 fields: `red`, `green` and `blue` which represent the values of the red, green and blue channels of the pixel, respectively. These fields are of type `uint8_t` which is an unsigned integer type of 8 bits. The `t_pixel` type is defined as follows:

```
typedef struct {
    uint8_t red;
    uint8_t green;
    uint8_t blue;
} t_pixel;
```

2.2.3 Useful constants

In order to easily read the raw data of images, you can define the following constants:

```
// Offsets for the BMP header
#define BITMAP_MAGIC        0x00 // offset 0
#define BITMAP_SIZE         0x02 // offset 2
#define BITMAP_OFFSET       0x0A // offset 10
```

```
#define BITMAP_WIDTH        0x12 // offset 18
#define BITMAP_HEIGHT       0x16 // offset 22
#define BITMAP_DEPTH        0x1C // offset 28
#define BITMAP_SIZE_RAW     0x22 // offset 34

// Magical number for BMP files
#define BMP_TYPE            0x4D42 // 'BM' in hexadecimal

// Header sizes
#define HEADER_SIZE         0x0E // 14 octets
#define INFO_SIZE           0x28 // 40 octets

// Constant for the color depth
#define DEFAULT_DEPTH       0x18 // 24
```

2.3 Allocation and deallocation functions

Once the types `t_bmp24` and the types necessary for its implementation are defined, we can now write the functions that will allow us to manipulate them. The first functions you will need to write concern memory allocations as well as all the corresponding deallocation functions. Here are the prototypes of the expected functions:

```
t_pixel ** bmp24_allocateDataPixels (int width, int height);
void bmp24_freeDataPixels (t_pixel ** pixels, int height);

t_bmp24 * bmp24_allocate (int width, int height, int colorDepth);
void bmp24_free (t_bmp24 * img);
```

These function will be used for :

`bmp24_allocateDataPixels` Dynamically allocate memory for a `t_pixel` matrix of size `width` × `height`, and return the address allocated in the heap. If the allocation fails, the function displays an error message and returns `NULL`.

`bmp24_freeDataPixels` Free all the memory allocated for the `t_pixel` matrix `pixels`.

`bmp24_allocate` Dynamically allocate memory for a `t_bmp24` image. This function must call the function `bmp24_allocateDataPixels` to allocate the `data` matrix. Then it initializes the `width`, `height` and `colorDepth` fields of the image with the values received as parameters. Finally, it returns the address of the image allocated in the heap. If the allocation fails, the function frees all the memory already allocated for the matrix, then displays an error message and returns `NULL`.

`bmp24_free` Free all the memory allocated for the image `img` received as a parameter.

2.4 Features: Loading and Saving 24-bit Images

As for the grayscale image processing, we will need to implement the features that will allow us to load and save 24-bit images. These two functions will respectively read and write to a file. The expected functions are:

```
t_bmp24 * bmp24_loadImage (const char * filename);
void bmp24_saveImage (t_bmp * img, const char * filename);
```

The functions descibed in the following subsections will be useful in order to implement both functions.

2.4.1 Read and write functions

In order to read and write in a file, the following functions are provided. You can refer to the documentation of the `fseek()` function from the `stdio.h` library to have a better understanding on how the following functions operate on the `buffer` and the file descriptor `file`.

```
/*
 * @brief Set the file cursor to the position position in the file file,
 * then read n elements of size size from the file into the buffer.
 * @param position The position from which to read in file.
 * @param buffer The buffer to read the elements into.
 * @param size The size of each element to read.
 * @param n The number of elements to read.
 * @param file The file descriptor to read from.
 * @return void
*/
void file_rawRead (uint32_t position, void * buffer, uint32_t size, size_t n, FILE * file) {
    fseek(file, position, SEEK_SET);
    fread(buffer, size, n, file);
}

/*
 * @brief Set the file cursor to the position position in the file file,
 * then write n elements of size size from the buffer into the file.
 * @param position The position from which to write in file.
 * @param buffer The buffer to write the elements from.
 * @param size The size of each element to write.
 * @param n The number of elements to write.
 * @param file The file descriptor to write to.
 * @return void
*/
void file_rawWrite (uint32_t position, void * buffer, uint32_t size, size_t n, FILE * file) {
    fseek(file, position, SEEK_SET);
    fwrite(buffer, size, n, file);
}
```

For instance, if you want to read the header of a BMP file, you can use the `file_rawRead` function as follows:

```
t_bmp_header header;
file_rawRead(BITMAP_MAGIG, &header, sizeof(t_bmp_header), 1, file);
```

This function call will position the file cursor at the `BITMAP_MAGIC` position (at offset 0) in the file `file`, then read 1 element of size `sizeof(t_bmp_header)` into the buffer `header`.

2.4.2 Read and write pixel data

In order to read and write the pixel data of an image, you can use the following functions. These functions will allow you to read and write the image data in the file. Here are the prototypes of these functions:

```
void bmp24_readPixelValue (t_bmp * image, int x, int y, FILE * file);
void bmp24_readPixelData (t_bmp * image, FILE * file);

void bmp24_writePixelValue (t_bmp * image, int x, int y, FILE * file);
void bmp24_writePixelData (t_bmp * image, FILE * file);
```

The function `bmp24_readPixelValue` will read the pixel value at position `(x, y)` in the image and store it in the `data` field of the image. The function `bmp24_writePixelValue` will write the pixel value at position `(x, y)` in the image to the file.

Remarks:

- The data (the pixel matrix) of a BMP image starts at the offset `BITMAP_OFFSET` (10 bytes after the start of the header).
- In a BMP file, pixels are stored from bottom to top, meaning that the pixel at the bottom left of the image is the first pixel to be stored in the file. Therefore, you need to reverse the order of the rows when reading and writing data.
- Pixels are stored in BGR (Blue, Green, Red) order in the BMP file. Therefore, you need to reverse the order of the color channels when reading and writing data.
- In this part, **you do not need to manage padding: ensure that the width and height of the image are multiples of 4**.

2.4.3 The `bmp24_loadImage` function

The function `bmp24_loadImage` will read a BMP 24-bit image file and load it into a `t_bmp24` structure. Here are the specifications of this function:

- It opens the file `filename` in binary read mode. If the opening fails, it displays an error message and returns `NULL`.
- It raw reads the width, height, and color depth of the image (normally 24 bits) and then calls the function `bmp24_allocate` to allocate memory for the image.
- It then reads the headers of the file with the function `file_rawRead` and stores them in the `header` and `header_info` fields of the image.
- Finally, it reads the image data and initializes the pixel matrix with the function `bmp24_readPixelData`, then closes the file and returns the loaded image.

2.4.4 The `bmp24_saveImage` function

The function `bmp24_saveImage` will save a `t_bmp24` image into a file in BMP 24-bit format. Here are the specifications of this function:

- It opens the file `filename` in binary write mode. If the opening fails, it displays an error message and returns.
- It writes the headers of the image into the file using the function `file_rawWrite`.
- It then writes the image data into the file using the function `bmp24_writePixelData`.
- Finally, it closes the file.

2.5 Features: 24-bit Image Processing

You should now be able to manipulate 24-bit images. The functions you will need to implement are similar to those you implemented for grayscale images. The only difference is that the pixel data is now stored in a `t_pixel` structure instead of a single `uint8_t` value. These are the first feature functions to implement:

```
void bmp24_negative (t_bmp * img);
void bmp24_grayscale (t_bmp * img);
void bmp24_brightness (t_bmp * img, int value);
```

Recall that those functions have to :

`bmp24_negative` Invert the colors of the image: for each pixel, subtract the value of each color channel from 255. For example, if a pixel has an RGB value of (100, 150, 200), the negative value will be (155, 105, 55).

`bmp24_grayscale` Convert the image to grayscale: for each pixel, calculate the average value of the 3 color channels and assign this value to each channel. For example, if a pixel has an RGB value of (100, 150, 200), the grayscale value will be (150, 150, 150).

`bmp24_brightness` Adjust the brightness of the image: for each pixel, add a value `value` to each color channel. If the value exceeds 255, it is capped at 255. For example, if a pixel has an RGB value of (100, 150, 200) and `value` is 50, the new value will be (150, 200, 250).

(a) Image originale. **(b)** Négatif. **(c)** Niveaux de gris. **(d)** Luminosité +50.

Figure 2.3 – Application des fonctions `bmp24_negative`, `bmp24_grayscale` et `bmp24_brightness` avec un facteur 50.

2.6 Features: Convolution Filters

In order to apply the convolution filters described in section **??**, we propose to write a function that applies the convolution for a given pixel of an image.

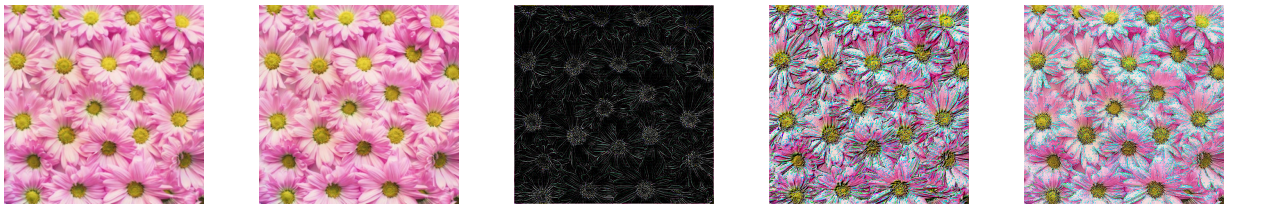
```
t_pixel bmp24_convolution (t_bmp24 * img, int x, int y, float ** kernel, int kernelSize);
```

This function will apply a convolution filter to a given pixel `(x, y)` of the image `img` using the convolution kernel `kernel` of size `kernelSize`. The function computes and returns the new value of the pixel after applying the filter.

Then implement the following features by calling the `bmp24_convolution` function for each pixel of the image. The convolution kernels remain the same as those described in section **??**.

- `bmp24_boxBlur` : Apply a box blur filter to the image.
- `bmp24_gaussianBlur` : Apply a Gaussian blur filter to the image.
- `bmp24_outline` : Apply an outline filter to the image.
- `bmp24_emboss` : Apply an emboss filter to the image.

- `bmp24_sharpen` : Apply a sharpen filter to the image.



(a) `bmp24_boxBlur`. **(b)** `bmp24_gaussianBlur`. **(c)** `bmp24_outline`. **(d)** `bmp24_emboss`. **(e)** `bmp24_sharpen`.

Figure 2.4 – Application de différents filtres de convolution

3

Part 3 : Histogram equalization

3.1 Image histogram

An *histogram* of a grayscale image is the graphical representation of the distribution of gray levels in the image. It is a table that counts the number of pixels for each gray level, ranging from 0 to 255 for an 8-bit image. The histogram is an essential tool in image processing, as it allows us to visualize the distribution of light intensities and analyze the contrast of the image.

The following figure shows the histogram of a grayscale image. The x-axis represents the gray levels (from 0 to 255), while the y-axis represents the number of pixels for each gray level.

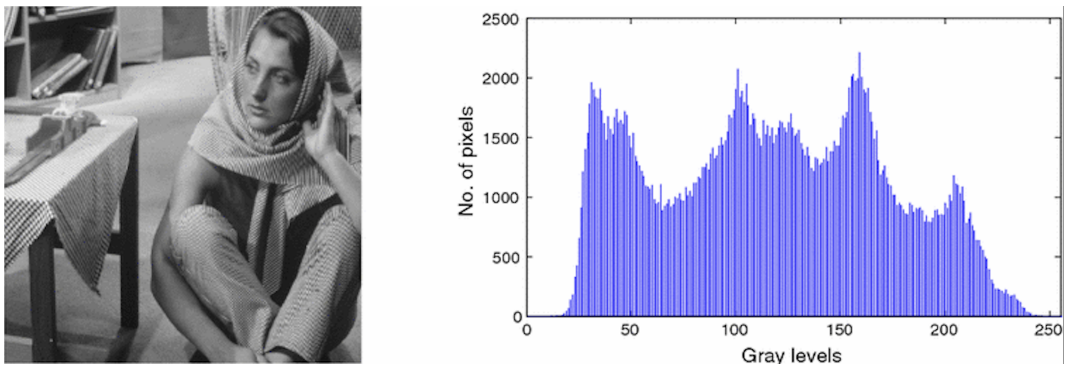


Figure 3.1 – Histogram of a grayscale image.

The left part of the histogram represents dark gray levels (0), while the right part represents light gray levels (255). Dark images will have a histogram that focuses on the left part, while light images will have a histogram that focuses on the right part. A well-distributed histogram indicates an image with good contrast, while a histogram concentrated in a small range of gray levels indicates low contrast.

The histogram can be calculated by iterating through each pixel of the image and incrementing the corresponding counter for the gray level of the pixel. For example, if a pixel has a value of 100, we increment the counter for gray level 100 in the histogram. Thus, for an 8-bit grayscale image, the histogram is an array of 256 integers, each representing the number of pixels with that gray level.

3.2 Histogram equalization

An *histogram equalization* is an image processing operation T that aims to improve the contrast of an image by redistributing the gray levels more uniformly. The main idea is to transform the histogram of the original

image into a uniform histogram, which allows for better visibility of details in the dark and light areas of the image.

Histogram equalization is a technique used to enhance the contrast of an image by redistributing the gray levels. It is particularly useful for images whose histogram is concentrated in a small range of gray levels, which can make certain parts of the image difficult to distinguish. By redistributing the gray levels, histogram equalization improves the overall contrast of the image.

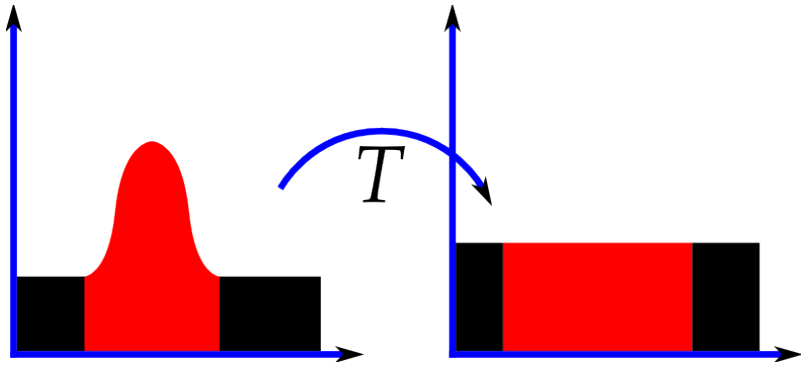


Figure 3.2 – Histogram equalization process.

Histogram equalization is performed by transforming the gray levels of the original image using a transformation function that depends on the distribution of gray levels in the image. This transformation is generally nonlinear and aims to extend the range of gray levels to obtain a more uniform distribution through a *cumulative distribution function (CDF)* of the original image's histogram. This operation can be performed both on grayscale images and color images.

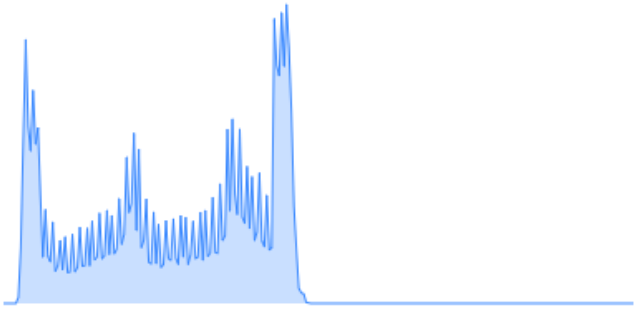
The CDF is a function that gives the cumulative probability of each gray level in the histogram. It is calculated by summing the values of the histogram from the lowest gray level to the current gray level. The CDF is then normalized to obtain a new mapping between the original gray levels and the new gray levels in the equalized image.

For grayscale images, the histogram equalization process can be summarized in the following steps:

1. Calculate the histogram of the original image.
2. Calculate the cumulative histogram using a CDF.
3. Normalize the CDF to obtain a new scale of gray levels.
4. Apply the transformation to each pixel of the original image using the new scale of gray levels.
5. Obtain the equalized image.

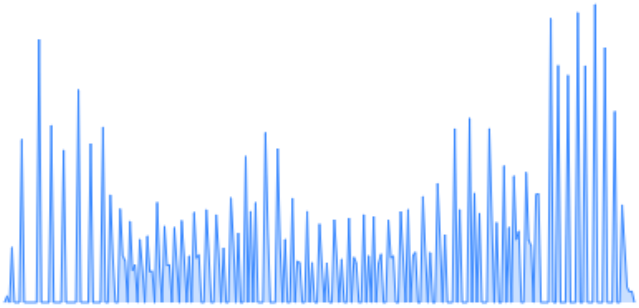
The following figures illustrate the histogram equalization process.

(a) Original image.



(b) Original histogram.

(a) Equalized image.



(b) Equalized histogram.

3.3 Implementation

As described in the previous section, histogram equalization is performed in several steps. In the first step, the functions described in this part will deal with 8-bit grayscale images. You are asked to implement the following functions:

```
unsigned int * bmp8_computeHistogram(t_bmp8 * img);
unsigned int * bmp8_computeCDF(unsigned int * hist);
void bmp8_equalize(t_bmp8 * img, unsigned int * hist_eq);
```

3.3.1 Compute histogram of a 8-bit grayscale image

The `bmp8_computeHistogram` function computes the histogram of an 8-bit grayscale image. It takes as input a pointer to an image `t_bmp8 * img` representing the image and returns an array of integers of size 256 containing the number of pixels for each gray level.

3.3.2 Compute cumulative normalized histogram

The cumulative histogram is a cumulative representation of the histogram of an image. The `bmp8_computeCDF` function takes as input an array of integers `hist` representing the histogram of a `t_bmp8` image and returns an array of integers of size 256 containing the normalized cumulative histogram `hist_eq`.

The cumulative histogram is calculated by summing the values of the original histogram. Specifically, the CDF is an array of the same size as the histogram `hist`, where each element represents the sum of the previous elements of the histogram. For a given histogram hist, the cumulative histogram cdf is defined as follows:

$$\begin{aligned} \text{cdf}[i] &= \sum_{j=0}^{i} \text{hist}[j] \\ &= \sum_{j=0}^{i-1} \text{hist}[j] + \text{hist}[i] \\ &= \text{cdf}[i-1] + \text{hist}[i] \end{aligned} \tag{3.1}$$

$\forall i \in [0, 255]$.

This cumulative histogram is then normalized to obtain a normalized histogram `hist_eq` that maps a gray level in the original image to its new value in the equalized image.

The normalization of the CDF is performed using the following formula:

$$\text{hist}_{eq}[i] = \text{round}\left(\frac{\text{cdf}[i] - \text{cdf}_{\text{min}}}{N - \text{cdf}_{\text{min}}} \times 255\right) \forall i \in [0, 255] \tag{3.2}$$

where cdf_{min} is the smallest non-zero value of the cumulative histogram CDF and N is the total number of pixels in the image. The `round` function is used to round the values.

For the implementation of this function, you can use the `round` function from the `math.h` library to round the values.

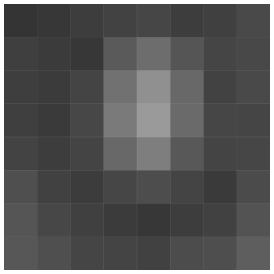
3.3.3 Equalize a 8-bit grayscale image

The `bmp8_equalize` function applies histogram equalization to an 8-bit grayscale image. It takes as input a pointer to an image `t_bmp8 * img` representing the original image and an array of integers `hist_eq` representing the normalized cumulative histogram. The function modifies the original image by applying the transformation defined by the histogram `hist_eq`. This function will apply the following principle:

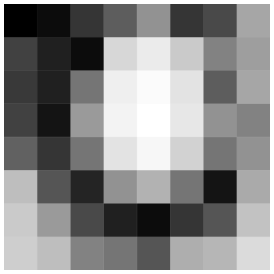
```
Function: bmp8_equalize(img, hist_eq)
Local var:
    hist
    hist_eq
Begin
    hist = bmp8_computeHistogram(img)
    hist_eq = bmp8_computeCDF(hist)
    For i <- 0 to img->dataSize - 1
        img->data[i] = hist_eq[img->data[i]]
    End For
End
```

3.3.4 Example

The following example of input image and expected output image illustrates the histogram equalization process. It will help you to test your implementation. The input image is a grayscale image with a low contrast, and the output image is the result of histogram equalization.



(a) Original image.



(b) Equalized image.

```
Original image :
 52  55  61  59  79  61  76  61
 62  59  55 104  94  85  59  71
 63  65  66 113 144 104  63  72
 64  70  70 126 154 109  71  69
 67  73  68 106 122  88  68  68
 68  79  60  70  77  66  58  75
 69  85  64  58  55  61  65  83
 70  87  69  68  65  73  78  90

Equalized image :
  0  12  53  32 190  53 174  53
 57  32  12 227 219 202  32 154
 65  85  93 239 251 227  65 158
 73 146 146 247 255 235 154 130
 97 166 117 231 243 210 117 117
117 190  36 146 178  93  20 170
130 202  73  20  12  53  85 194
```

The table 3.1 shows the values of the original histogram, the CDF, and the equalized histogram.

In the table 3.1, all gray levels not present in the original image are omitted, i.e., if `hist[i] = 0`, the gray level i is not represented in the table. For example, the gray level 0 is not present in the original image, so it does not appear in the table.

Note that the gray level 52 is the lowest value in the original image, and in the equalized image, it is mapped with 0. Similarly, the gray level 154 is the highest value in the original image, and in the equalized image, it is mapped with 255.

3.4 Equalization of color images

For color images, the histogram equalization method must be extended from grayscale to the three color components. It is possible to perform equalization independently on each component, but this degrades the colors and is therefore not used in practice.

Instead, one common approach is to convert the color image into a different color space that separates luminance (brightness) from chrominance (color information). This allows for equalizing the luminance component while preserving the chrominance components. The *YUV* color space is an example of a color space

Gray level `i`	**Histogram** `hist[i]`	**CDF** `cdf[i]`	**Equalized histogram** `hist_eq[i]`
52	1	1	0
55	3	4	12
58	2	6	20
59	3	9	32
60	1	10	36
61	4	14	53
62	1	15	57
63	2	17	65
64	2	19	73
65	3	22	85
66	2	24	93
67	1	25	97
68	5	30	117
69	3	33	130
70	4	37	146
71	2	39	154
72	1	40	158
73	2	42	166
75	1	43	170
76	1	44	174
77	1	45	178
78	1	46	182
79	2	48	190
83	1	49	194
85	2	51	202
87	1	52	206
88	1	53	210
90	1	54	215
94	1	55	219
104	2	57	227
106	1	58	231
109	1	59	235
113	1	60	239
122	1	61	243
126	1	62	247
144	1	63	251
154	1	64	255

Table 3.1 – For each gray level `i`, the histogram `hist[i]`, the cumulative histogram `cdf[i]` and the normalized cumulative histogram `hist_eq[i]`.

that allows separating luminance and chrominance. In this space, the Y component represents luminance, while the U and V components represent chrominance.

Equalizing the histogram of a color image is performed on the luminance component (Y) of the image, which allows to enhance the contrast of the image while preserving the color information. After equalizing the Y component, the image is converted back to the original color space (e.g., RGB) to obtain the final image.

The process of histogram equalization for color images can be summarized in the following steps:

1. Convert the color image to YUV color space.
2. Calculate the histogram of the Y component.
3. Calculate the cumulative histogram and normalize the CDF.
4. Apply histogram equalization to the Y component.
5. Convert the image back to the original color space (e.g., RGB).

3.4.1 Conversion from one color space to another

The conversion from RGB to YUV color space is performed using the following formulas:

$$\begin{aligned} Y &= 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \\ U &= -0.14713 \cdot R - 0.28886 \cdot G + 0.436 \cdot B \\ V &= 0.615 \cdot R - 0.51499 \cdot G - 0.10001 \cdot B \end{aligned} \tag{3.3}$$

where R, G and B are the red, green and blue values respectively, and Y, U and V are the luminance and chrominance values in the YUV color space. The conversion from YUV to RGB is performed using the following formulas:

$$\begin{aligned} R &= Y + 1.13983 \cdot V \\ G &= Y - 0.39465 \cdot U - 0.58060 \cdot V \\ B &= Y + 2.03211 \cdot U \end{aligned} \tag{3.4}$$

where R, G and B are the red, green and blue values respectively, and Y, U and V are the luminance and chrominance values in the YUV color space.

3.4.2 Equalization of the Y component

The equalization of the Y component of a color image is performed in the same way as the histogram equalization of a grayscale image. The histogram of the Y component is calculated, then the cumulative histogram is calculated and normalized. Finally, histogram equalization is applied to the Y component using the transformation defined by the equalized histogram.

3.4.3 Implementation

Write a function

```
void bmp24_equalize(t_bmp24 * img)}
```

that performs histogram equalization on a color image. The function takes as input a pointer to a color image `t_bmp24 * img` and modifies the original image by applying histogram equalization according to the principle described above.

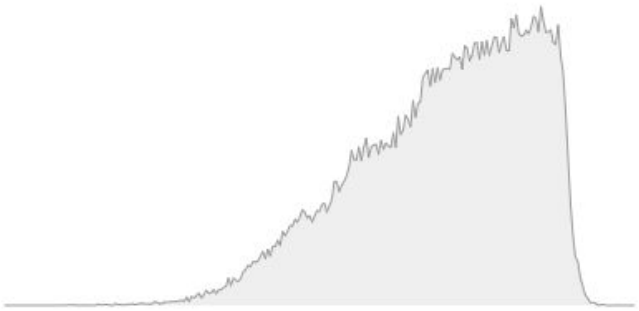
Remarks and clues:

- The conversion from RGB to YUV and the inverse conversion from YUV to RGB is performed pixel by pixel using the formulas described in equations 3.3 and 3.4.
- Note that the values of Y, U and V are floating-point values, while the values of the original image components are integers. Therefore, it is necessary to convert the floating-point values to integers after the inverse conversion and when calculating the histogram.
- It is possible to store the values of Y, U and V in a temporary array to facilitate the inverse conversion. You can also define a new type `t_yuv` to store the values of Y, U and V.
- It is important to ensure that the values of Y, U and V remain within the appropriate limits (0-255) during the inverse conversion.
- You can use the `round` function from the `math.h` library to round the values of Y, U and V to the nearest integer.

3.4.4 Example

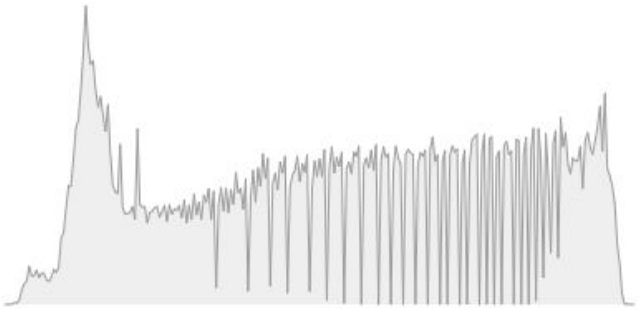
The following figures show the original image and the equalized image, along with their respective histograms.

(a) Original image.



(b) Original histogram of luminance (Y Component).

(a) Equalized image.



(b) Equalized histogram of luminance (Y Component).