Project for Computer Organization and Architecture

Title: Parallel Image Processing

Subject Code: 15B11CI313

4TH Semester

B. Tech

Batch: B15

Team Members: Aditya Agarwal 21104050

Mogish Hashmi 21104038

Vipul Verma 21104052

Yash Agarwal 21104039

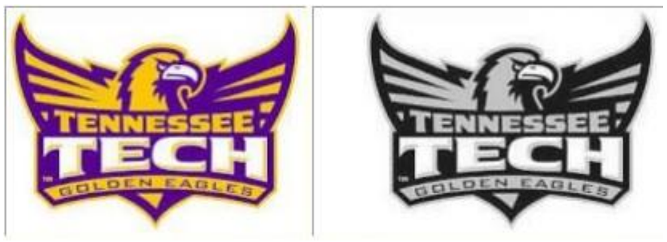
Index

Problem Statement

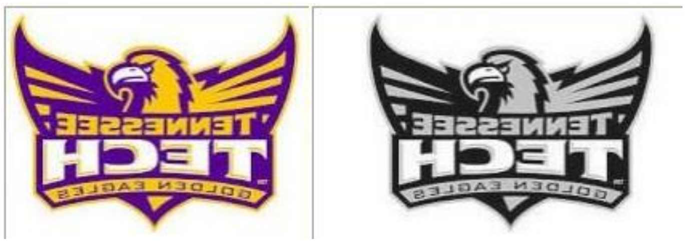
The problem statement is to develop an algorithm or software that can convert a normal image to its mirror form and then to its grey scale form. The goal is to create an efficient and accurate system that can automatically process images, reflect them horizontally to create a mirror image, and then convert the resulting image to a grey scale format. This system should be able to handle a wide variety of image types and sizes and produce high-quality output that accurately reflects the original image. The system should also be user-friendly, with a simple interface that allows users to easily upload and process their images. Overall, the goal of this project is to provide a fast and reliable tool for converting images to their mirror and grey scale forms, which can be used in a wide range of applications such as image editing, computer vision, and graphic design.

For example,

A normal image to grey scaled image



A flipped image to its grey scale form,



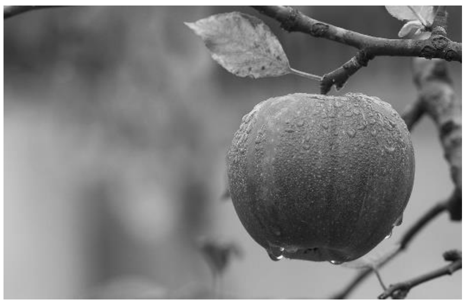
The goal is to solve the problem in least complexity and retain the best quality of original picture.

How to achieve this?

1.Load the image: The first step is to load the image you want to convert using an image processing library in your preferred programming language.

2.Convert to mirror image: To create the mirror image, you need to reflect the pixels of the original image horizontally. One way to do this is to iterate over each row of pixels in the image and swap the pixels on the left side of the image with those on the right side. This can be achieved using a nested loop or a built-in function depending on the library you're using.

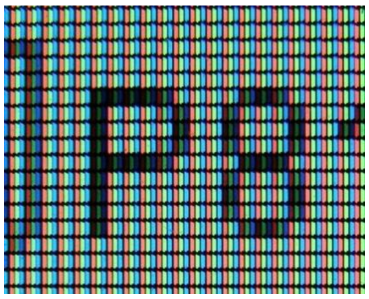
3.Convert to grey scale: To convert the mirror image to grey scale, you need to remove the colour information from the image and only retain the brightness values. This can be done by iterating over each pixel in the image and taking the average of the red, green, and blue values to obtain a single value representing the brightness of the pixel. The resulting image will be a grey scale version of the mirror image.

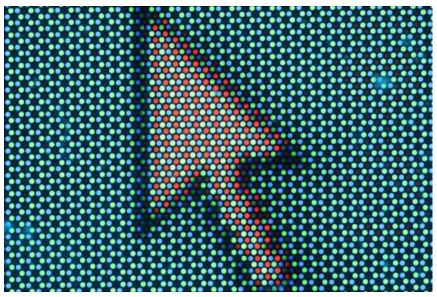


4.Save the image: Finally, you can save the resulting image in a file format of your choice using the image processing library you're working with.

Note that:

Images are represented as pixels. You can think of a pixel and an individual colour "dot" on your monitor screen.





The colour of the pixel is represented as a mixture of intensities of the colours red, green, and blue. Each intensity is represented as an 8-bit number in the ranging from 0 to 255. For example, the values (0,0,0) represent the colour black, the values (255,0,0) represent the colour red, and the values (255,255,0) represent the colour yellow. We call these intensities RGB values (for red, green, and blue).

Gray scaling an image represented as a series of RGB values is easy. Different methods exist, but an effective method is called the luminosity method. Given the I th pixel, you Grayscale that pixel. Then, for each i, set the red, green, and blue component of pixel[i] to Gray value[i] to Gray-scale the image.

To Flip an image or mirror it,

You need to reflect the pixels of the original image along a vertical axis. This means that the pixels on the left side of the image are moved to the right side, and the pixels on the right side of the image are moved to the left side. This will create a flipped version of the original image.

To achieve this, you can use an image processing library in your preferred programming language that provides functions for manipulating images. One common library used for image processing is OpenCV, which has functions for flipping images. In OpenCV, you can use the cv2.flip() function to flip an image horizontally by passing the image array and the value 1 for the flip Code parameter.

After flipping the image, you can save it to a file format of your choice, such as JPG. The resulting image will be a horizontally flipped version of the original RGB image.

It's important to note that the specific implementation may vary depending on the library and programming language you're using, and the method used to flip the image may differ slightly from library to library. However, the basic concept remains the same - reflecting the pixels of the original image along a vertical axis to create a flipped version of the image.

To Grey Scale an RGB image,

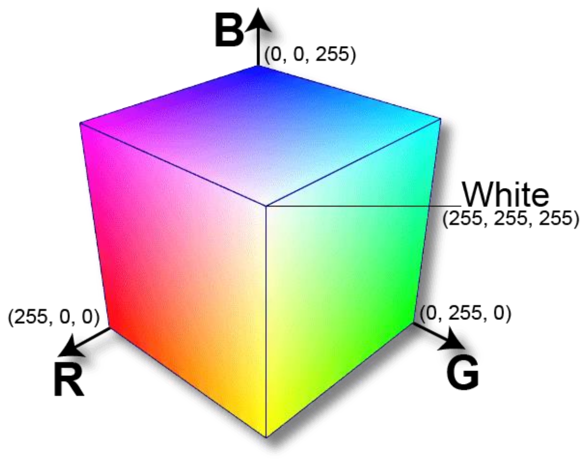
There are several methods you can grey scale an RGB image,

But first let’s define some useful terms,

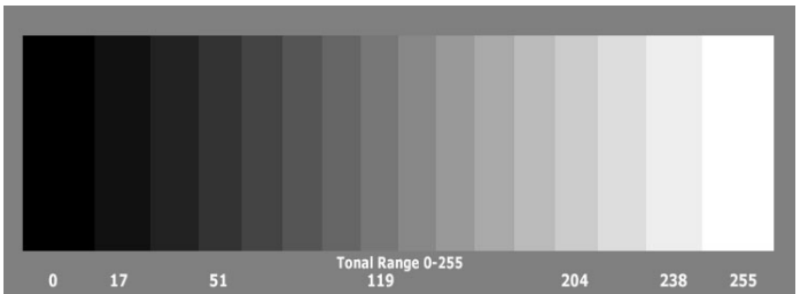
1.Color Models: Representing colours as numerical values is a necessary step in many applications. To do this, we use models that are mathematical models that describe ways of mapping colours to a set of numbers. Usually, a colour model defines three- or four-color components that are easily described through a coordinate system. Each colour that the model can represent corresponds to a point in this coordinate system.

2. RGB: The most well-known colour model is RGB which stands for Red-Green-Blue. As the name suggests, this model represents colours using individual values for red, green, and blue. The RGB model is used in almost all digital screens throughout the world.

Specifically, a colour is defined using three integer values from 0 to 255 for red, green, and blue, where a zero value means dark and a value of 255 means bright. Given the values, the final colour is defined when we mix these three basic colours weighted by their values.



3.Grayscale: Grayscale is the simplest model since it defines colors using only one component that is lightness. The amount of lightness is described using a value ranging from 0 (black) to 255 (white). On the one hand, grayscale images convey less information than RGB. However, they are common in image processing because using a grayscale image requires less available space and is faster, especially when we deal with complex computations.



Now to convert an RGB image to Greyscale, there are many methods like,

- **Lightness method**
  A very simple method is to take the average value of the components with the highest and lowest value:
  $$\text{grayscale} = \frac{\min(R,G,B) + \max(R,G,B)}{2}$$
  We can easily see that this method presents a very serious weakness since one RGB component is not used.

- **Average method**

Another method is to take the average value of the three components (red, green, and blue) as the grayscale value:

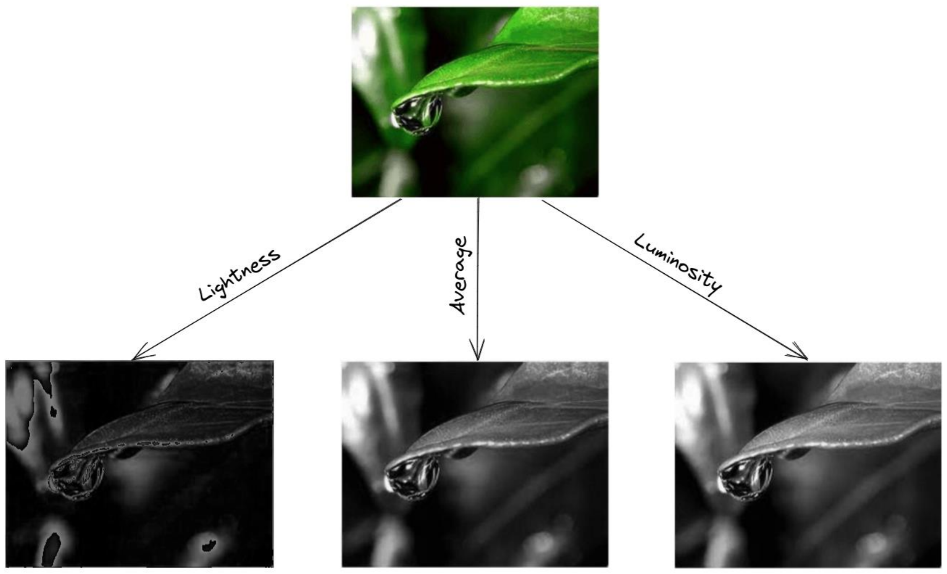
$$\text{grayscale} = \frac{R+G+B}{3}$$

Although we now consider all components, the average method is also problematic since it assigns the same weight to each component.

- **Luminosity method**

We will be using Luminosity Method, that successfully solves the problems of previous methods. Based on the aforementioned observations, we should take a weighted average of the components. The contribution of blue to the final value should decrease, and the contribution of green should increase. After some experiments and more in-depth analysis, researchers have concluded in the equation: Grayscale = 0.3 * R + 0.59 * G + 0.11 * B .

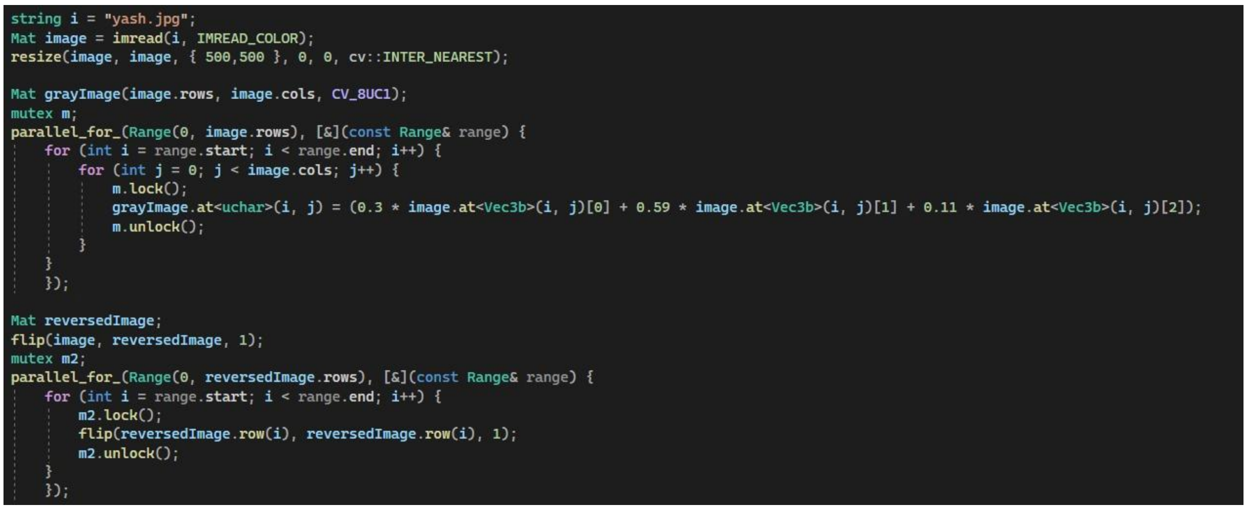
For a stock image,



The best results can be seen in **Luminosity method**.

Coding to achieve this Process

In Serial execution,

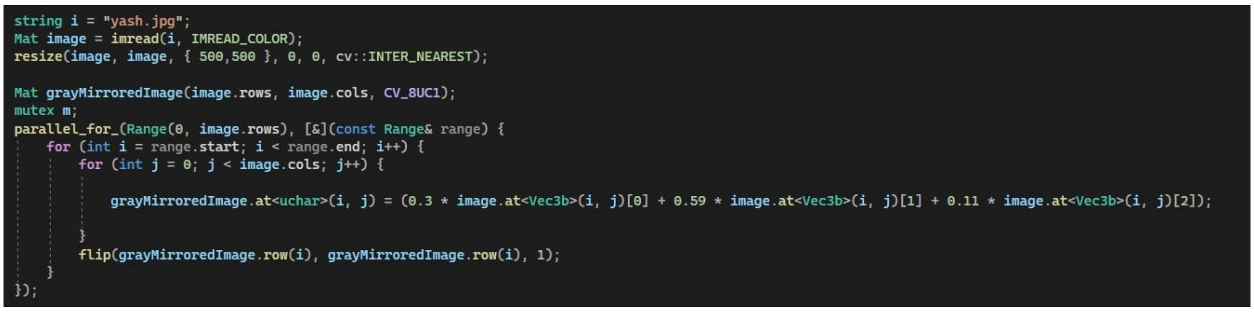


```
string i = "yash.jpg";
Mat image = imread(i, IMREAD_COLOR);
resize(image, image, { 500,500 }, 0, 0, cv::INTER_NEAREST);

Mat grayImage(image.rows, image.cols, CV_8UC1);
mutex m;
parallel_for_(Range(0, image.rows), [&](const Range& range) {
    for (int i = range.start; i < range.end; i++) {
        for (int j = 0; j < image.cols; j++) {
            m.lock();
            grayImage.at<uchar>(i, j) = (0.3 * image.at<Vec3b>(i, j)[0] + 0.59 * image.at<Vec3b>(i, j)[1] + 0.11 * image.at<Vec3b>(i, j)[2]);
            m.unlock();
        }
    }
    });

Mat reversedImage;
flip(image, reversedImage, 1);
mutex m2;
parallel_for_(Range(0, reversedImage.rows), [&](const Range& range) {
    for (int i = range.start; i < range.end; i++) {
        m2.lock();
        flip(reversedImage.row(i), reversedImage.row(i), 1);
        m2.unlock();
    }
    });
```

In parallel execution,



```
string i = "yash.jpg";
Mat image = imread(i, IMREAD_COLOR);
resize(image, image, { 500,500 }, 0, 0, cv::INTER_NEAREST);

Mat grayMirroredImage(image.rows, image.cols, CV_8UC1);
mutex m;
parallel_for_(Range(0, image.rows), [&](const Range& range) {
    for (int i = range.start; i < range.end; i++) {
        for (int j = 0; j < image.cols; j++) {

            grayMirroredImage.at<uchar>(i, j) = (0.3 * image.at<Vec3b>(i, j)[0] + 0.59 * image.at<Vec3b>(i, j)[1] + 0.11 * image.at<Vec3b>(i, j)[2]);

        }
        flip(grayMirroredImage.row(i), grayMirroredImage.row(i), 1);
    }
});
```

Output for the following code

We went from Image 1 to Image 2 through this process.

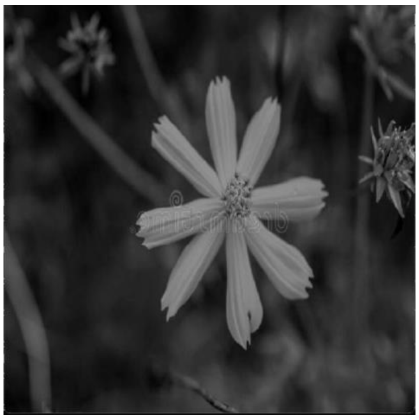
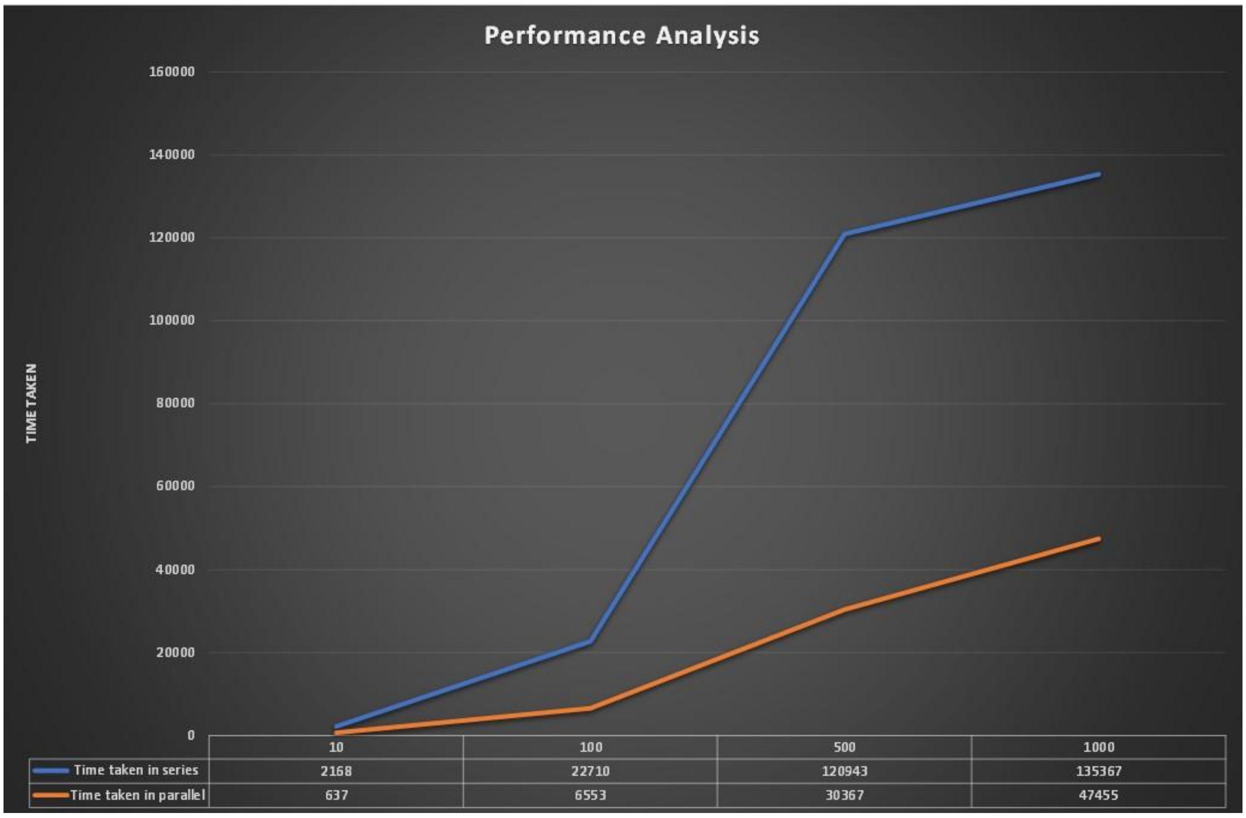


Image 1 **Image 2**

Performance Analysis



	10	100	500	1000
Time taken in series	2168	22710	120943	135367
Time taken in parallel	637	6553	30367	47455

Blue Line- >Time taken in Sequential.

Orange line->Time taken in Parallel.

Filtering the Image

>Filtering in image processing refers to the process of modifying an image by applying a mathematical operation or a set of operations to its pixels. These operations can be used to remove noise from the image, sharpen its edges, or enhance certain features.

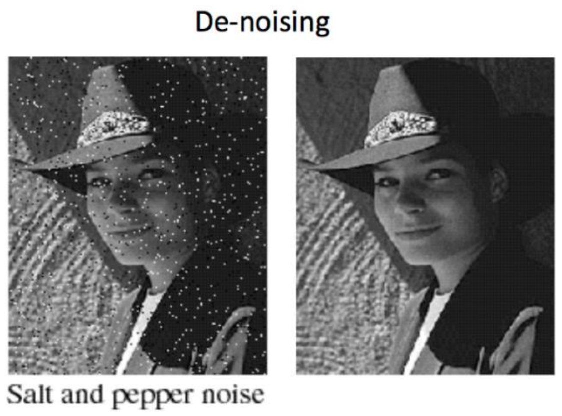
There are two main types of filters used in image processing: linear and non-linear filters. Linear filters operate on an image by performing a weighted sum of neighbouring pixels, and they are typically used for smoothing and blurring an image. Examples of linear filters include Gaussian filter, mean filter, and median filter.

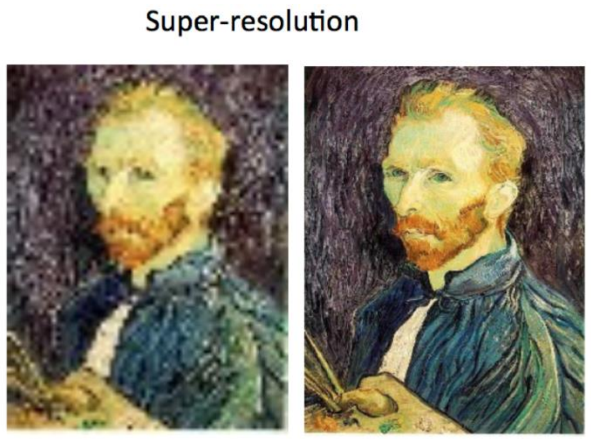
Non-linear filters, on the other hand, perform more complex operations that cannot be expressed as a weighted sum of neighbouring pixels. These filters are typically used for enhancing certain features in an image or removing specific types of noise. Examples of non-linear filters include Sobel filter, Laplacian filter, and Canny edge detection filter.

The choice of filter to use depends on the specific problem you are trying to solve and the characteristics of the image you are working with. A good understanding of image processing concepts and techniques is essential for choosing the right filter and applying it effectively to achieve the desired result.

>Filtering process is sequential since it is implemented using the filter2D function. The filter2D function applies the filter to each pixel in the input image sequentially, starting from the top left corner and moving across each row until the entire image has been filtered. Therefore, the output of the filter2D function is obtained sequentially and is stored in the Gray Image matrix.

Note that, depending on the hardware and software configuration, the implementation of the filter2D function might use parallel processing techniques, such as multithreading, to speed up the filtering process. However, from the code snippet provided, we cannot determine whether the implementation is parallel or not.





What is a filter 2D function?

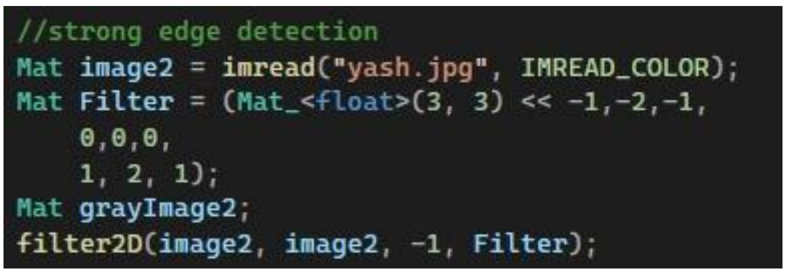
A 2D filter function is a mathematical operation that is applied to an image in order to modify its pixel values. It is a type of linear filter that operates on an image by performing a convolution of the image with a 2D kernel, also known as a filter mask or filter matrix.

The kernel is a small matrix of numerical values that are typically cantered around a specific pixel in the image. Each value in the kernel represents a weight that determines the influence of the corresponding pixel on the output value of the convolution operation. The kernel is moved across the image, pixel by pixel, and the convolution operation is performed at each step to generate a new pixel value for the output image.

The specific values in the kernel matrix depend on the type of filter being used and the specific application. Some common 2D filter functions include the Gaussian filter, which is used for smoothing an image and reducing noise, and the Sobel filter, which is used for edge detection.

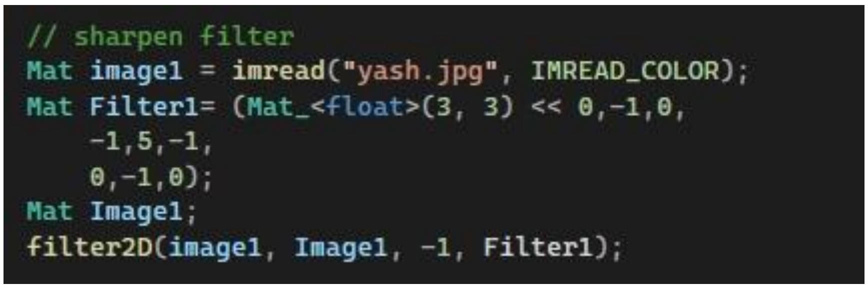
In image processing, 2D filter functions are often used to perform operations such as blurring, sharpening, and edge detection on images. They are implemented using specialized libraries or functions in programming languages such as Python, MATLAB, and C++.

Code for filter of Image



```
//strong edge detection
Mat image2 = imread("yash.jpg", IMREAD_COLOR);
Mat Filter = (Mat_<float>(3, 3) << -1,-2,-1,
    0,0,0,
    1, 2, 1);
Mat grayImage2;
filter2D(image2, image2, -1, Filter);
```

To Strong Edge Detection

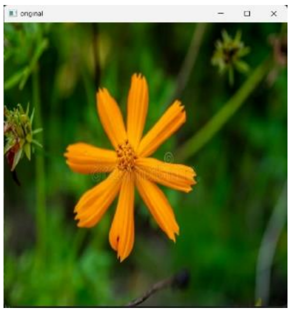


```
// sharpen filter
Mat image1 = imread("yash.jpg", IMREAD_COLOR);
Mat Filter1= (Mat_<float>(3, 3) << 0,-1,0,
    -1,5,-1,
    0,-1,0);
Mat Image1;
filter2D(image1, Image1, -1, Filter1);
```

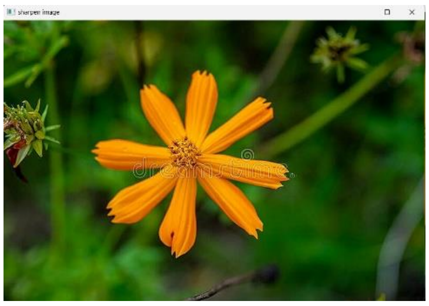
To Sharpen Filter

Output for the code

1.Image Sharpened,

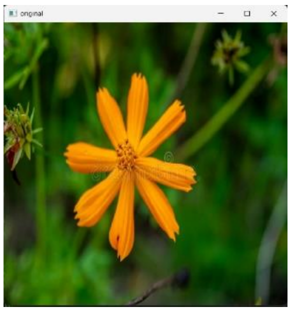


Original

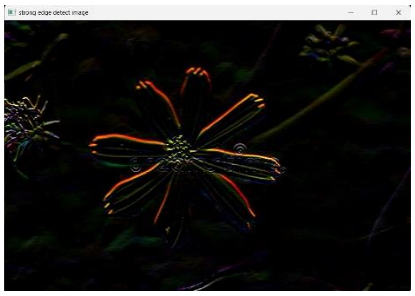


Filtered

2.Image Edge Detection,



Original



Filtered

Code for the operation performed

```
#include<opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc.hpp>
#include<iostream>
#include<time.h>
#include<vector>
using namespace cv;
using namespace std;

int main()
{
    clock_t start = 0;
    start = clock();

    // serial execution
    for (int k = 0; k <10; k++)
    {
        string i = "yash.jpg";
        Mat image = imread(i, IMREAD_COLOR);
        resize(image, image, { 500,500 }, 0, 0, cv::INTER_NEAREST);

        Mat grayImage(image.rows, image.cols, CV_8UC1);
        mutex m;
        parallel_for_(Range(0, image.rows), [&](const Range& range) {
            for (int i = range.start; i < range.end; i++) {
                for (int j = 0; j < image.cols; j++) {
                    m.lock();
                    grayImage.at<uchar>(i, j) = (0.3 * image.at<Vec3b>(i, j)[0] + 0.59 * image.at<Vec3b>(i, j)[1] +
0.11 * image.at<Vec3b>(i, j)[2]);
                    m.unlock();
                }
            }
            });

        Mat reversedImage;
        flip(image, reversedImage, 1);
        mutex m2;
        parallel_for_(Range(0, reversedImage.rows), [&](const Range& range) {
            for (int i = range.start; i < range.end; i++) {
                m2.lock();
                flip(reversedImage.row(i), reversedImage.row(i), 1);
                m2.unlock();
            }
            });
    }

    start = clock() - start;
    std::cout << "TIME TAKEN IN SEQUENTIAL EXECUTION : " << start<<endl;

    start = 0;
    start = clock();

    //parallel execution
    for (int k = 0; k <10; k++)
    {

        string i = "yash.jpg";
        Mat image = imread(i, IMREAD_COLOR);
        resize(image, image, { 500,500 }, 0, 0, cv::INTER_NEAREST);

        Mat grayMirroredImage(image.rows, image.cols, CV_8UC1);
        mutex m;
        parallel_for_(Range(0, image.rows), [&](const Range& range) {
            for (int i = range.start; i < range.end; i++) {
                for (int j = 0; j < image.cols; j++) {

                    grayMirroredImage.at<uchar>(i, j) = (0.3 * image.at<Vec3b>(i, j)[0] + 0.59 *
image.at<Vec3b>(i, j)[1] + 0.11 * image.at<Vec3b>(i, j)[2]);

                }
                flip(grayMirroredImage.row(i), grayMirroredImage.row(i), 1);
            }
        });

    }
    start = clock() - start;
    std::cout << " TIME TAKE IN PARALLEL EXECUTION : "<<start<<endl;


    string i = "yash.jpg";
    Mat image = imread(i, IMREAD_COLOR);
    resize(image, image, { 500,500 }, 0, 0, cv::INTER_NEAREST);

    Mat grayMirroredImage(image.rows, image.cols, CV_8UC1);
    mutex m;
    parallel_for_(Range(0, image.rows), [&](const Range& range) {
        for (int i = range.start; i < range.end; i++) {
            for (int j = 0; j < image.cols; j++) {

                grayMirroredImage.at<uchar>(i, j) = (0.3*image.at<Vec3b>(i, j)[0] + 0.59*image.at<Vec3b>(i, j)[1]
+ 0.11*image.at<Vec3b>(i, j)[2]) ;
            }
            flip(grayMirroredImage.row(i), grayMirroredImage.row(i), 1);
        }
        });


    // sharpen filter
    Mat image1 = imread("yash.jpg", IMREAD_COLOR);
    Mat Filter1= (Mat_<float>(3, 3) << 0,-1,0,
        -1,5,-1,
        0,-1,0);
    Mat Image1;
    filter2D(image1, Image1, -1, Filter1);

    //strong edge detection
    Mat image2 = imread("yash.jpg", IMREAD_COLOR);
    Mat Filter = (Mat_<float>(3, 3) << -1,-2,-1,
        0,0,0,
        1, 2, 1);
    Mat grayImage2;
    filter2D(image2, image2, -1, Filter);

    imshow("strong edge detect image",image2);

    imshow("sharpen image",Image1);

    imshow("original", image);

    imshow("gray mirrored image ", grayMirroredImage);

    waitKey(0);
    return 0;
}
```

Result and Analysis

Converting an image to its mirror and grayscale form can have different results and analysis based on the input image and the specific conversion process used. Here are some general observations and analysis that can be made:

Mirror Conversion:

The mirror conversion of an image will result in a horizontally flipped version of the original image.

This can be useful for certain applications, such as creating symmetrical images or correcting images that were flipped during the capture process.

However, if the original image contains text or other elements with directional orientation, flipping the image can make them difficult to read or recognize.

Grayscale Conversion:

The grayscale conversion of an image will result in a black-and-white version of the original image, where each pixel's value represents its brightness or luminance level.

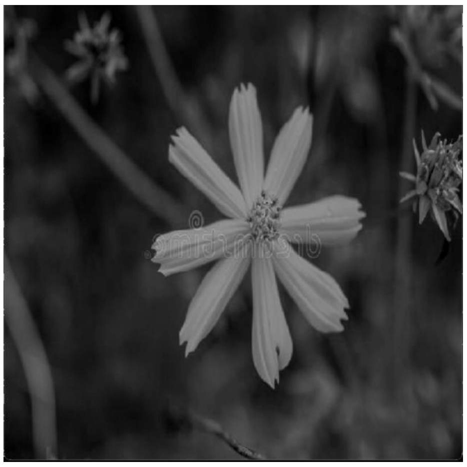
Grayscale conversion is often used for simplifying images and reducing their size, as well as improving the contrast and visibility of certain features in the image.

However, grayscale conversion can also result in a loss of information and detail, particularly in images with subtle colour variations or gradients.

When combining the two conversions together (mirror and grayscale), the resulting image will be a horizontally flipped black-and-white version of the original image. This can be useful for certain applications, such as in photography or design, but the specific use case will depend on the desired outcome.

It's important to note that the specific implementation of the conversion process may vary depending on the library and programming language used. The quality of the result also depends on the quality of the original image and the settings used during the conversion process. In general, it's always a good idea to experiment with different conversion methods and settings to find the best result for a given image.

Normal Image


Mirror and Grey-Scaled

Conclusion

In conclusion, converting an image to its mirror and grayscale form can have different effects and serve different purposes depending on the specific image and the desired outcome. Mirror conversion can be useful for creating symmetrical images or correcting flipped images, while grayscale conversion can be used to simplify an image and improve its contrast and visibility.

Combining the two conversions together can result in a horizontally flipped black-and-white version of the original image, which can be useful for certain applications. However, it's important to note that the specific implementation of the conversion process may vary and the quality of the result depends on the quality of the original image and the settings used during the conversion process.

Overall, understanding image processing concepts and techniques, as well as experimenting with different conversion methods and settings, is essential for achieving the desired result when converting an image to its mirror and grayscale form.

Appendix

LIBRARIES USED:

- opencv2/core/core.hpp: This library provides the basic building blocks for the OpenCV library. It includes data structures and functions for handling arrays, matrices, and other basic image-processing operations.
- opencv2/highgui/highgui.hpp: This library provides functions for displaying images and videos on the screen. It also includes functions for handling user input and displaying basic GUI elements.
- opencv2/imgproc.hpp: This library provides advanced image processing functions, such as filtering, thresholding, edge detection, and feature detection. It is the core of the OpenCV library and provides a wide range of functionality for image manipulation.
- iostream: This is the standard input/output library in C++. It provides functions for reading and writing data to the console.
- time.h: This library provides functions for working with time and date. In this specific code, it is likely being used to measure the execution time of certain operations.
- vector: This is a C++ container that provides a dynamic array-like data structure. It allows for efficient insertion and deletion of elements, as well as random access to elements.

FUNCTION USED:

- clock_t is a data type in C++ that represents the number of clock ticks elapsed since the program started running. It is used to measure the execution time of a certain portion of code.
- The imread() function is part of the OpenCV library and is used to read an image file from the disk.
- flip(image, reversedImage, 1); function takes an image, performs a flip operation on it based on the specified parameters, and returns the flipped image in the reversed Image variable.
- filter2D is a function provided by OpenCV that performs 2D convolution on an image. 2D convolution is a common operation in image processing and is used to perform tasks such as blurring, sharpening, edge detection, and more.
- parallel_for_(Range(0, image.rows),[&](const Range& range) { ... }, the function is being used to perform an operation on each row of an image in parallel. The Range(0, image.rows) specifies the range of rows to iterate over, in this case, from 0 to the total number of rows in the image. The & character before the lambda function indicates that it is being passed by reference, which allows it to modify variables outside of its scope. The lambda function is defined as [&](const Range& range) { ... }. The & character before the lambda function captures all variables used in the function by reference, allowing them to be modified inside the function. The const Range& range parameter specifies the range of rows to be processed by the function.

FLAGS, DATA STRUCUTRES AND DATA TYPES:

- IMREAD_COLOR flag indicates that the image should be loaded in the RGB color format.
- Mat is a matrix-like data structure in the OpenCV library that is used for storing and manipulating image data.

- mutexes can be used to protect shared resources in multithreaded applications
- uchar is an unsigned char data type that can store values in the range [0, 255]. It is commonly used to represent grayscale pixel values in OpenCV.
- Vec3b is a 3-channel (BGR) vector data type that can store three values of uchar type. It is commonly used to represent color pixel values in OpenCV. Each channel of the Vec3b type represents the intensity of one of the three-color channels: blue, green, and red.

References

https://www.baeldung.com/cs/convert-rgb-to-grayscale

https://in.mathworks.com/help/images/what-is-image-filtering-in-the-spatial-domain.html

https://www.isahit.com/blog/why-to-use-grayscale-conversion-during-image-processing#:~:text=%E2%80%8DWhat%20is%20grayscale%20in%20image,darkest%20of%20it%20being%20black.