OOPS

CLARITY - An Image Processing Pseudo - Application with OpenCV API integrated on linux (Ubuntu Distribution) and GUI Version Integrated with C++ Builder

Presented by :

Shiva Ranjane R (2023503528)
Raveena K (2023503553)
Hariharan R I (2023503555)

Job Roles :

- **Shivaranjane : Brightness Algorithm , GUI Integration (C++ Builder), Environment Management Head , Linux Integration**

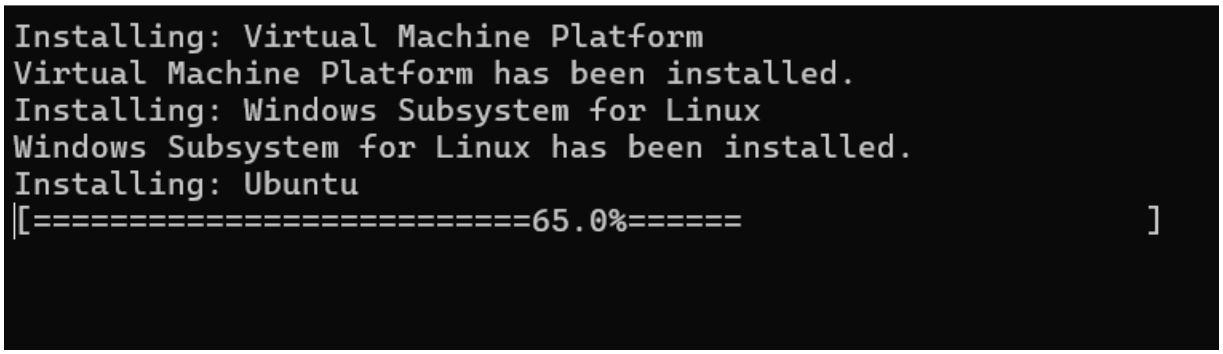
- **Raveena : Algorithm Developer , Greyscale Algorithm , Backend Head , Complexity and Effeciency Inspector, Code Enhancing Department**

- **Hariharan : Blur Algorithm , Testing , Debugging , Project Management, Co-ordinating Head**

Environment Installation

Windows Subsystem for Linux (WSL) :

Windows Subsystem for Linux (WSL) is a compatibility layer allowing Linux binaries to run natively on Windows 10 and Windows Server, enabling developers to work seamlessly with Linux tools and workflows within their Windows environment. It's utilized for development, cross-platform coding, system administration, and learning Linux, with each release bringing improvements, bug fixes, and expanded compatibility with different Linux distributions.



API Used : OpenCV

OpenCV (Linux Version for cpp Objects) :

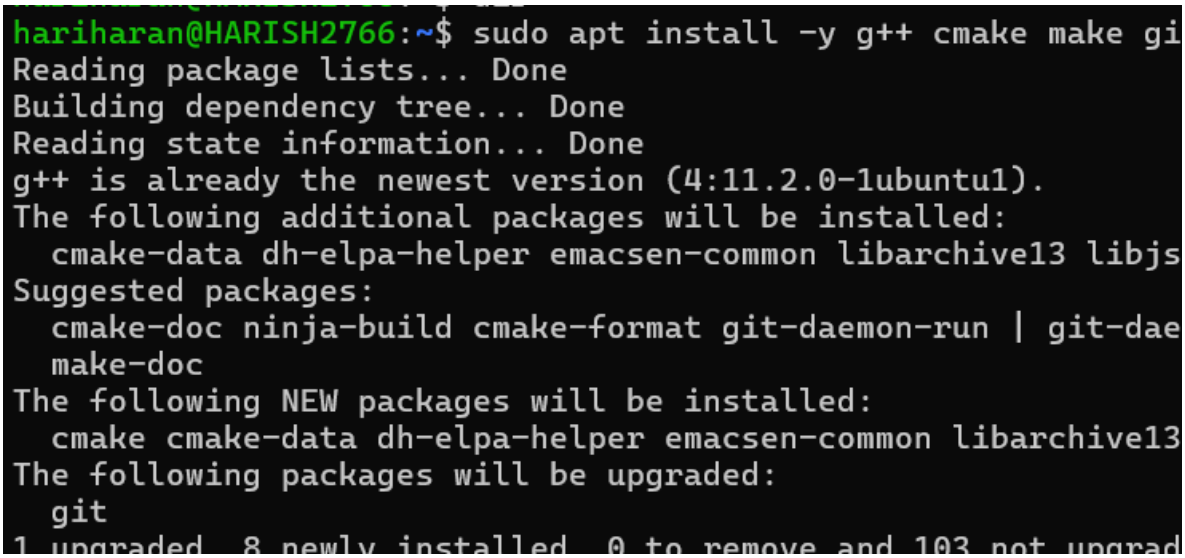
OpenCV for Linux is a widely utilized library in C++ for computer vision applications due to its robust functionality and efficiency. Its popularity stems from its comprehensive set of tools for image and video processing, making it invaluable for tasks such as object detection, facial recognition, and augmented reality. Developers favor OpenCV for its extensive collection of algorithms and functions, which enable complex image manipulation and analysis with ease. Its cross-platform compatibility ensures seamless integration with Linux systems, providing a stable environment for deploying computer vision solutions. Moreover, OpenCV's active community support and frequent updates ensure ongoing improvements and bug fixes, further solidifying its position as a cornerstone in the realm of computer vision development on the Linux platform.

OpenCV (Linux Version for cpp Objects) :

Installing OpenCV:

To install OpenCV, we will need to compile and install the package from the official repository.

Step 1: Installing required packages and tools
We need a c++ compiler to compile OpenCV, git to clone the sources from official repositories, and CMake, and make to execute the build system and some other dependencies. Enter the following command to get all these.



OpenCV (Linux Version for cpp Objects) :

Step 2: Download the source.
We need to clone the latest version of OpenCV.
To find the latest version, visit this website and click on the GitHub icon of the latest version.

```
hariharan@HARISH2766:~$ git clone https://github.com/opencv/opencv.git
Cloning into 'opencv'...
remote: Enumerating objects: 332355, done.
remote: Counting objects: 100% (1115/1115), done.
remote: Compressing objects: 100% (765/765), done.
Receiving objects:  20% (69110/332355), 127.36 MiB | 4.72 MiB/s
```

Step 3: Build the source
First, create the build directory and go into it

Next, generate the build scripts using cmake

```
hariharan@HARISH2766:~/build$ cmake ../opencv
-- 'Release' build type is used by default. Use CMAKE_BUILD_TYPE to specify build type
-- The CXX compiler identification is GNU 11.4.0
-- The C compiler identification is GNU 11.4.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detected processor: x86_64
-- Found PythonInterp: /usr/bin/python3 (found suitable version "3.10.12", minimum req
-- Could NOT find PythonLibs (missing: PYTHON_LIBRARIES PYTHON_INCLUDE_DIRS) (Required
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ModuleNotFoundError: No module named 'numpy'
-- Looking for ccache - not found
-- Performing Test HAVE_CXX_FSIGNED_CHAR
-- Performing Test HAVE_CXX_FSIGNED_CHAR - Success
```

OpenCV (Linux Version for cpp Objects) :

```
hariharan@HARISH2766:~/build$ make -j4
[  0%] Built target opencv_dnn_plugins
[  0%] Building C object 3rdparty/libjpeg-turbo/CMakeFiles/jpeg16-static.dir/src/jcapistd.c.o
[  0%] Building C object 3rdparty/libjpeg-turbo/CMakeFiles/jpeg12-static.dir/src/jcapistd.c.o
[  0%] Built target opencv_highgui_plugins
[  1%] Building C object 3rdparty/libjpeg-turbo/CMakeFiles/jpeg12-static.dir/src/jccolor.c.o
[  1%] Building C object 3rdparty/openjpeg/openjp2/CMakeFiles/libopenjp2.dir/thread.c.o
[  1%] Building C object 3rdparty/libjpeg-turbo/CMakeFiles/jpeg16-static.dir/src/jccolor.c.o
[  1%] Building C object 3rdparty/libjpeg-turbo/CMakeFiles/jpeg16-static.dir/src/jcdiffct.c.o
[  1%] Building C object 3rdparty/libjpeg-turbo/CMakeFiles/jpeg16-static.dir/src/jclossls.c.o
[  1%] Building C object 3rdparty/openjpeg/openjp2/CMakeFiles/libopenjp2.dir/bio.c.o
[  1%] Building C object 3rdparty/libjpeg-turbo/CMakeFiles/jpeg16-static.dir/src/jcmainct.c.o
[  1%] Building C object 3rdparty/openjpeg/openjp2/CMakeFiles/libopenjp2.dir/cio.c.o
[  1%] Building C object 3rdparty/libjpeg-turbo/CMakeFiles/jpeg16-static.dir/src/jcprepct.c.o
[  1%] Building C object 3rdparty/libjpeg-turbo/CMakeFiles/jpeg16-static.dir/src/jcsample.c.o
[  1%] Building C object 3rdparty/libjpeg-turbo/CMakeFiles/jpeg12-static.dir/src/jcdiffct.c.o
[  1%] Building C object 3rdparty/libjpeg-turbo/CMakeFiles/jpeg12-static.dir/src/jclossls.c.o
[  1%] Building C object 3rdparty/openjpeg/openjp2/CMakeFiles/libopenjp2.dir/dwt.c.o
[  1%] Building C object 3rdparty/libjpeg-turbo/CMakeFiles/jpeg12-static.dir/src/jcmainct.c.o
```

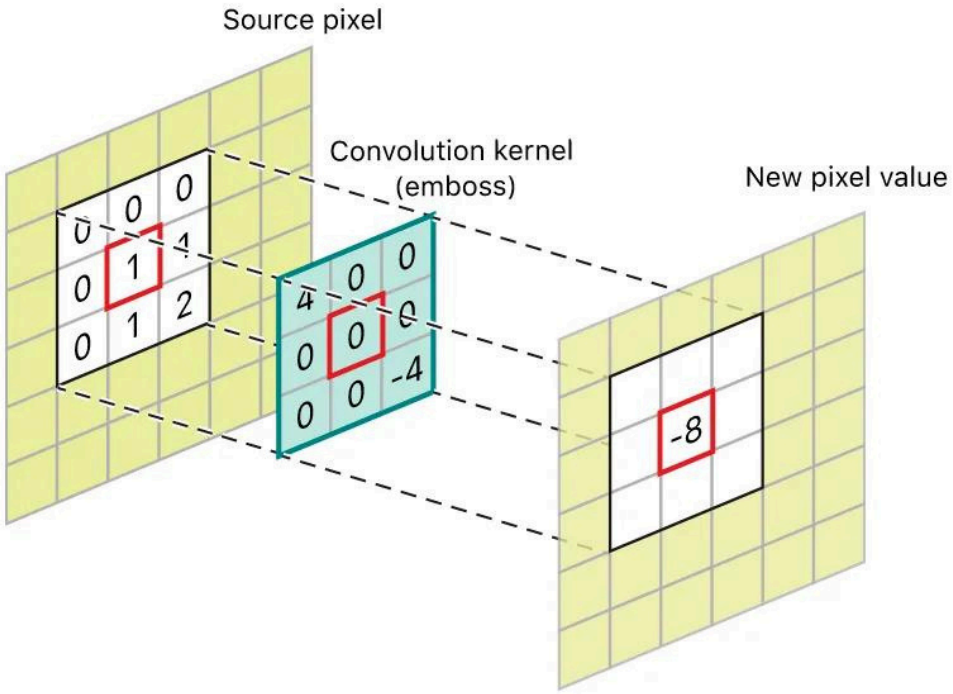
And That ‘ s How Opencv is installed in linux

Gaussian Blur Algorithm

Gaussian Blur :

Convolution :

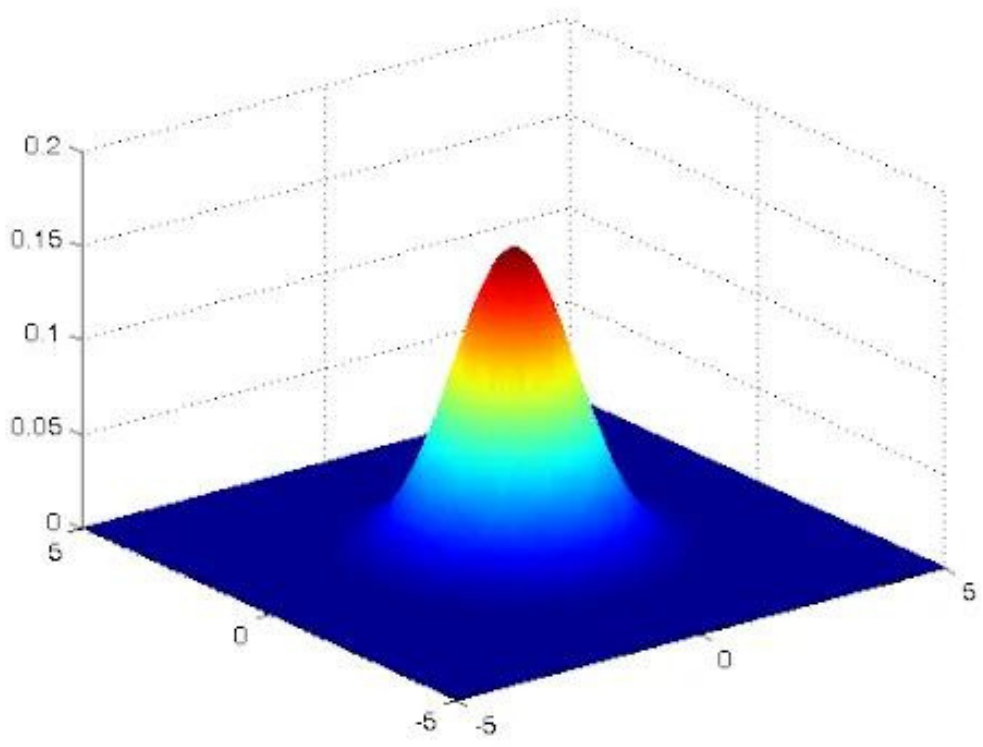
In simple terms, convolution is simply the process of taking a small matrix called the kernel and running it over all the pixels in an image. At every pixel, we'll perform some math operation involving the values in the convolution matrix and the values of a pixel and its surroundings to determine the value for a pixel in the output image.



Gaussian Blur :

Gaussian Distribution :

Next, let's turn to the Gaussian part of the Gaussian blur. Gaussian blur is simply a method of blurring an image through the use of a Gaussian function.
Below, you'll see a 2D Gaussian distribution. Notice that there is a peak in the center and the curve flattens out as you move towards the edges.



Gaussian Blur :

Gaussian Distribution :

Imagine that this distribution is superimposed over a group of pixels in an image. It should be apparent looking at this graph, that if we took a weighted average of the pixel's values and the height of the curve at that point, the pixels in the center of the group would contribute most significantly to the resulting value. This is, in essence, how Gaussian blur works.

In English, this means that we'll take the Gaussian function and we'll generate an n x m matrix. Using this matrix and the height of the Gaussian distribution at that pixel location, we'll compute new RGB values for the blurred image.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Gaussian Blur :

Gaussian Distribution :

The values from this function will create the convolution matrix / kernel that we'll apply to every pixel in the original image. The kernel is typically quite small — the larger it is the more computation we have to do at every pixel.
x and y specify the delta from the center pixel (0, 0). For example, if the selected radius for the kernel was 3, x and y would range from -3 to 3 (inclusive).
σ – the standard deviation – influences how significantly the center pixel's neighboring pixels affect the computations result.

Gaussian Blur :

Implementation :

We'll need to create a separate output image. We can't modify the source image directly because changing the pixel values will mess up the math for the adjacent pixel's computation in the next iteration.

Finally, we need to consider how we'll handle the edges. If we were looking at the very first pixel in an image, the kernel would extend beyond the bounds of the image. As a result, implementations will commonly ignore the outer most set of pixels, duplicate the edge, or wrap the image around.

In our case, for ease of implementation, we'll ignore it pixels on the edges.

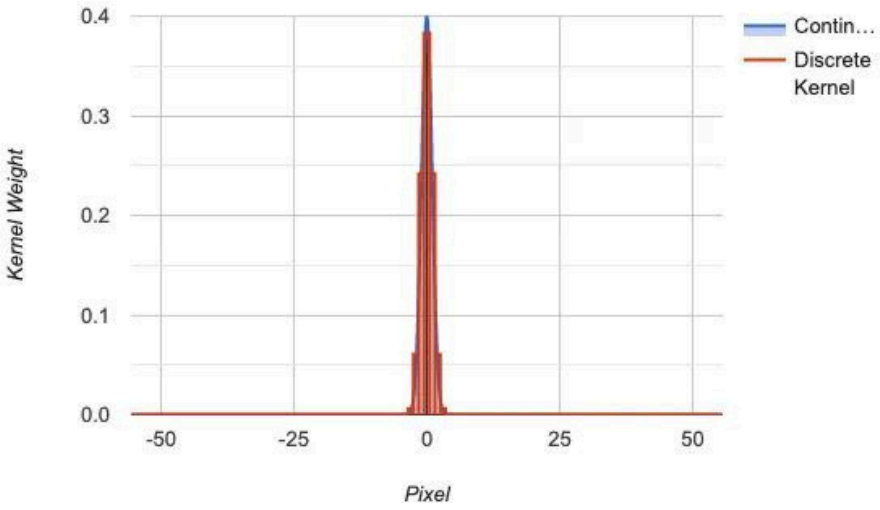
Let's start with implementing the Gaussian function. The first task is to identify reasonable values for x, y, and σ.

While the kernel can technically be an arbitrary size, we should scale σ in proportion to the kernel size. If we have a large kernel radius, but a small sigma, then all of the new pixels we're introducing with our larger radius aren't really affecting the computation.

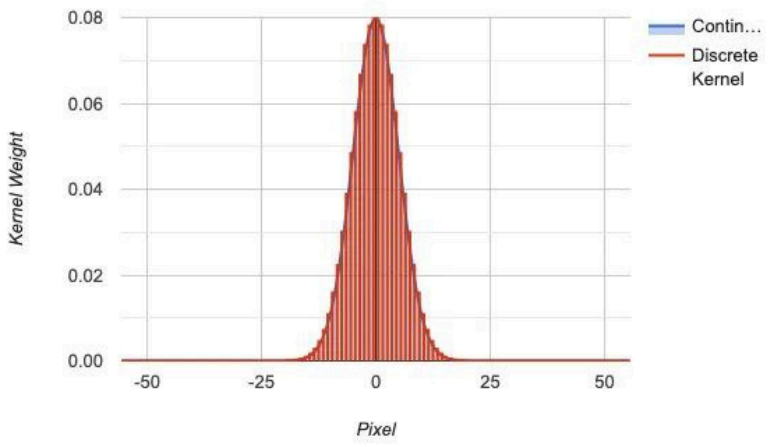
Gaussian Blur :

Implementation Problem :

Here’s an example of a large kernel radius, but a small sigma :-



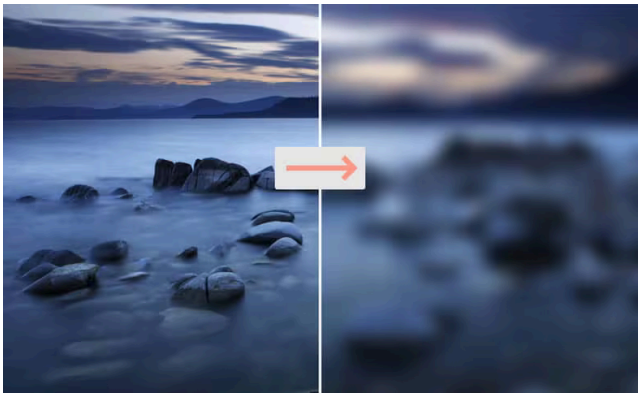
As opposed to Sigma 5, Kernel 111 :-



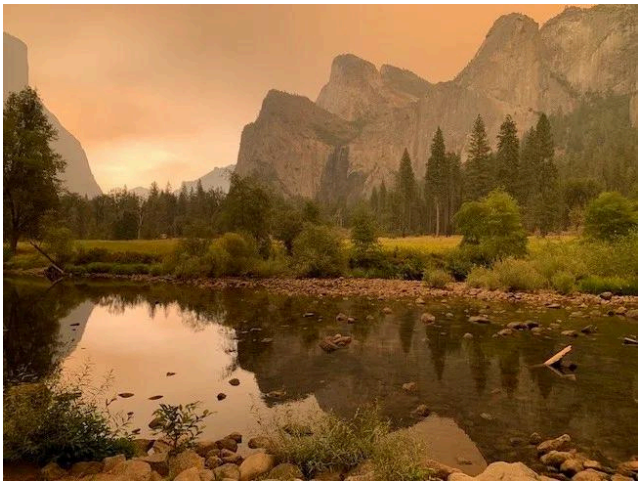
The weights are calculated by numerical integration of the continuous gaussian distribution

Gaussian Blur :

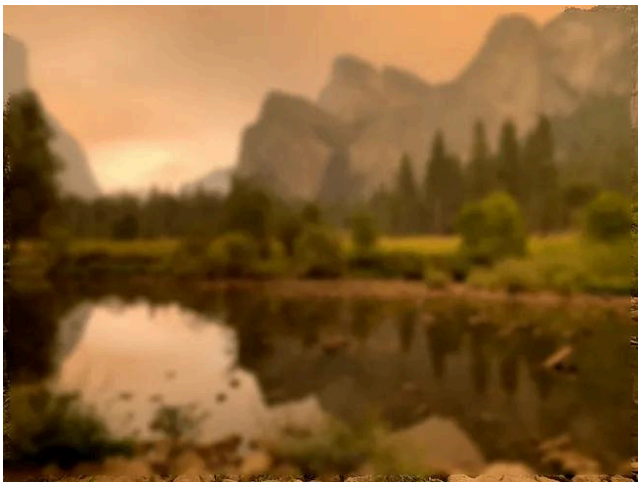
Output :



Original Image :



Blurred Image of Kernel Radius 11 :



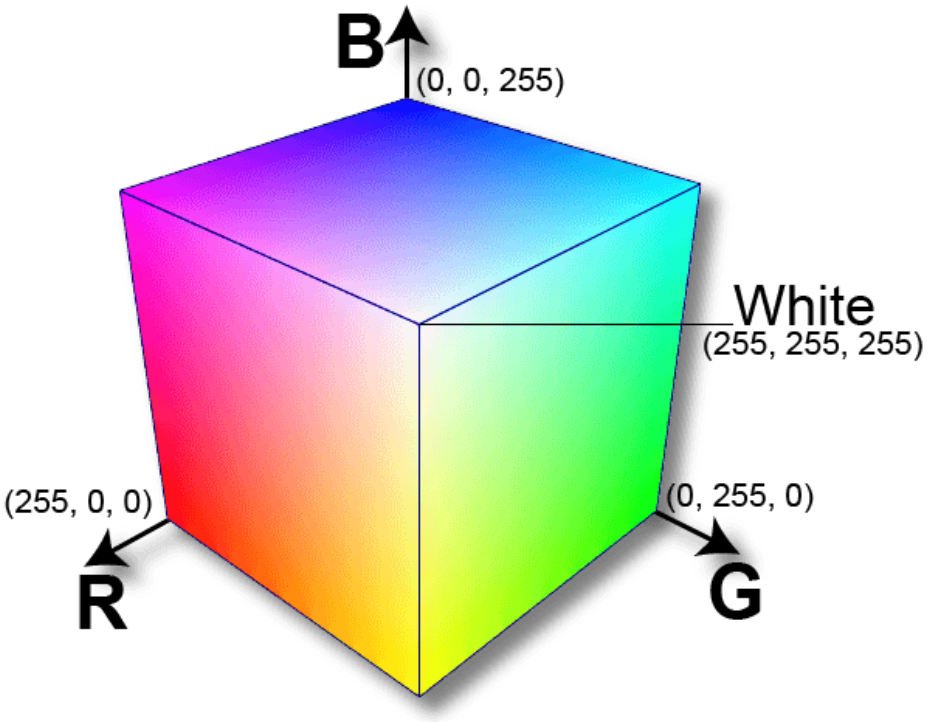
Greyscale Algorithm

Greyscale :

A grayscale algorithm converts a color image into shades of gray, where each pixel represents the brightness information of the original image

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

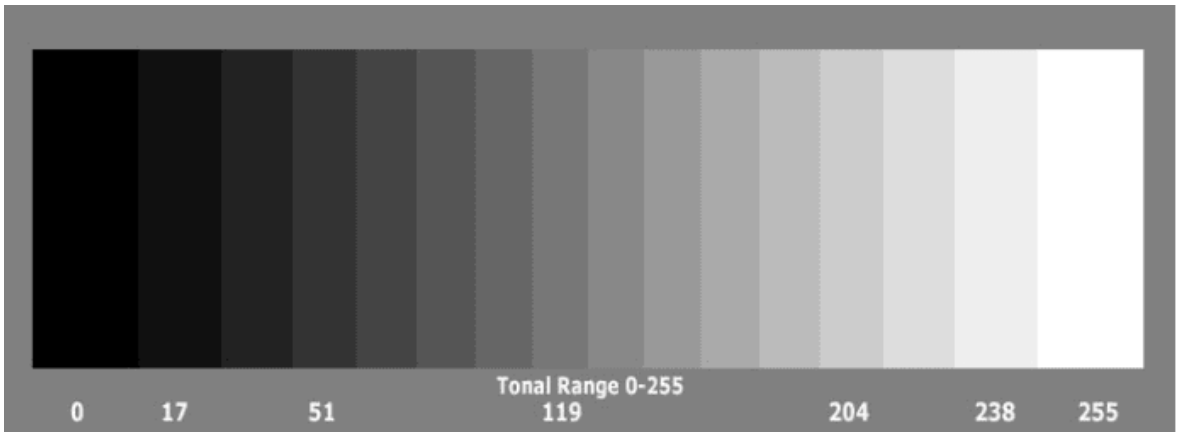
If we mix the three colors equally (RGB = (255, 255, 255)), we'll get white while the absence of all colors (RGB = (0, 0, 0)) means black.



1. Average Method

The average method calculates the grayscale value by averaging the red, green, and blue color values of each pixel.

Gray= {R + G + B}/3



Algorithm:

```
For Each Pixel in Image
{
  Red = Pixel.Red
  Green = Pixel.Green
  Blue = Pixel.Blue

  Gray = (Red + Green + Blue) / 3

  Pixel.Red = Gray
  Pixel.Green = Gray
  Pixel.Blue = Gray
}
```

The average method is also problematic since it assigns the same weight to each component. Based on research on human vision, we know that our eyes react to each color in a different manner. Specifically, our eyes are more sensitive to green, then to red, and finally to blue.

Greyscale:

Original Image:

Final Image:

Brightness Algorithm

Brightness:

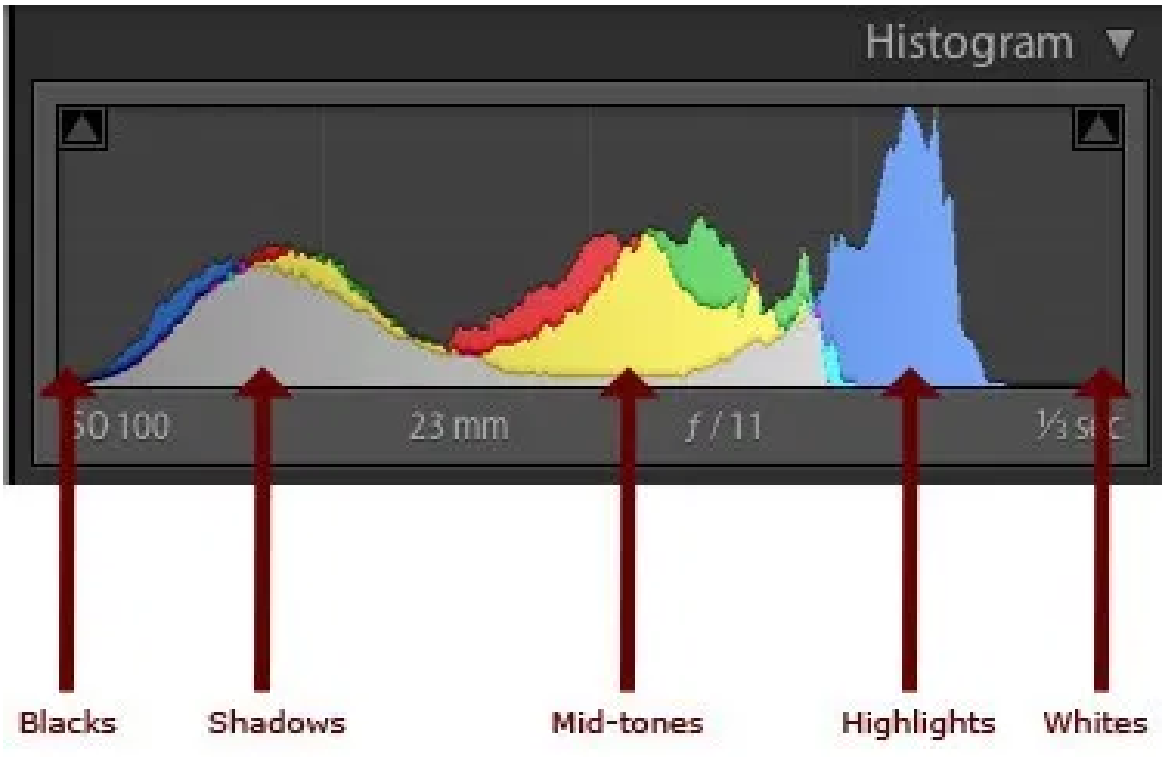
input: A constant reference to an OpenCV Mat object, which represents an input image.

output: A reference to an OpenCV Mat object, which will store the result of applying brightness adjustment to the input image.

b: An integer representing the brightness adjustment value.
The function iterates over each pixel of the input image using nested loops. For each pixel, it retrieves the pixel value as a 3-channel vector (Vec3b). Then, it iterates over each channel (Red, Green, Blue) and applies the brightness adjustment by adding the value of b to the corresponding channel intensity.

To ensure that the pixel values remain within the valid range of 0 to 255, it uses the min and max functions. Finally, the adjusted pixel values are assigned to the corresponding pixel in the output image.

Overall, this function performs a simple brightness adjustment on an input image by adding a constant value to each pixel's intensity in each channel.



Brightness:

Original Image

Brightness +50

Brightness -50

Source Code

Clarity 1D

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <cmath>
#include <vector>
using namespace std;
using namespace cv;
void generateGaussianKernel1D(int ks, double sigma, vector<double>& kernel)
{
   double sum = 0.0;
   int r = ks / 2;
   kernel.resize(ks);
   for (int i = -r; i <= r; ++i)
   {
      kernel[i + r] = exp(-(i * i) / (2 * sigma * sigma)) / (sqrt(2 * M_PI) * sigma);
      sum += kernel[i + r];
   }
   for (int i = 0; i < ks; ++i)
   {
      kernel[i] /= sum;
   }
}
```

```
void applyKernel1D(const Mat& src, Mat& dst, const vector<double>& kernel, bool horizontal)
{
   int ks = kernel.size();
   int r = ks / 2;
   dst = src.clone();
   if (horizontal)
   {
     for (int y = 0; y < src.rows; ++y)
     {
        for (int x = 0; x < src.cols; ++x)
        {
           double sumB = 0.0, sumG = 0.0, sumR = 0.0;
           for (int k = -r; k <= r; ++k)
           {
               int ix = x + k;
               if (ix >= 0 && ix < src.cols)
               {
                  Vec3b p = src.at<Vec3b>(y, ix);
                  double w = kernel[k + r];
                  sumB += p[0] * w;
                  sumG += p[1] * w;
                  sumR += p[2] * w;
               }
           }
          dst.at<Vec3b>(y, x)[0] = static_cast<unsigned char>(sumB);
           dst.at<Vec3b>(y, x)[1] = static_cast<unsigned char>(sumG);
           dst.at<Vec3b>(y, x)[2] = static_cast<unsigned char>(sumR);
         }
      }
   }
```

```
    else
    {
        for (int y = 0; y < src.rows; ++y)
        {
            for (int x = 0; x < src.cols; ++x)
            {
                double sumB = 0.0, sumG = 0.0, sumR = 0.0;
                for (int k = -r; k <= r; ++k)
                {
                    int iy = y + k;
                    if (iy >= 0 && iy < src.rows)
                    {
                        Vec3b p = src.at<Vec3b>(iy, x);
                        double w = kernel[k + r];
                        sumB += p[0] * w;
                        sumG += p[1] * w;
                        sumR += p[2] * w;
                    }
                }
                    dst.at<Vec3b>(y, x)[0] = static_cast<unsigned
char>(sumB);
                    dst.at<Vec3b>(y, x)[1] = static_cast<unsigned
char>(sumG);
                    dst.at<Vec3b>(y, x)[2] = static_cast<unsigned
char>(sumR);
            }
        }
    }
}
```

```
void blur(const Mat& input, Mat& output, int ks, double sigma)
{
   vector<double> kernel;
   generateGaussianKernel1D(ks, sigma, kernel);
   Mat temp;
   applyKernel1D(input, temp, kernel, true);
   applyKernel1D(temp, output, kernel, false);
}
void greyscale(const Mat& input, Mat& output)
{
   output = Mat::zeros(input.size(), CV_8UC1);
   for (int x = 0; x < input.rows; ++x)
   {
      for (int y = 0; y < input.cols; ++y)
      {
         Vec3b p = input.at<Vec3b>(x, y);
               uchar gray = static_cast<uchar>(( p[2] + p[1] + p[0])/3 );
         output.at<uchar>(x, y) = gray;
      }
   }
}
```

```
void brightness(const Mat& input, Mat& output, int b)
{
   for (int x = 0; x < input.rows; ++x)
   {
      for (int y = 0; y < input.cols; ++y)
      {
         Vec3b p = input.at<Vec3b>(x, y);
         for (int c = 0; c < 3; ++c)
         {
            int newValue = p[c] + b;
            newValue = min(max(newValue, 0), 255);
            output.at<Vec3b>(x, y)[c] = newValue;
         }
      }
   }
}
int main(int argc, char** argv)
{
   if (argc != 2)
   {
      cout << "Usage: ./blurImage <image_path>" << endl;
      return -1;
   }
   Mat inputImage = imread(argv[1]);
   if (inputImage.empty())
   {
        cerr << "Error: Could not open or find the image!" <<
endl;
      return -1;
   }
```

```
    Mat FinalImage = inputImage.clone();
    int c;
    cout<<”Enter the process:\n1.Blurring\n2.Greyscale\n
3.Brightness\n”;
    cin>>c;
    if(c==1)
    {
        int ks;
        double s;
        cout << "Enter Kernel size and sigma value: ";
        cin >> ks >> s;
        if (ks % 2 == 0)
        {
            ks += 1;
        }
        blur(inputImage, FinalImage, ks, s);
    }
    if(c==2)
    {
        greyscale(inputImage,FinalImage);
    }
    if(c==3)
    {
        int b;
        cout<<"Enter Brightness level(-255 to 255):";
        cin>>b;
        brightness(inputImage,FinalImage,b);
    }
    imshow("Original Image", inputImage);
    imshow("Final Image", FinalImage);
    imwrite("Final_image.jpg", FinalImage);
    waitKey(0);
    return 0;
}
```

Source Code

Clarity 2D

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <cmath>
using namespace std;
using namespace cv;
void gformula(int ks, double s, double** k)
{
   double sum = 0.0;
   int r = ks / 2;
   for (int x = -r; x <= r; ++x)
   {
      for (int y = -r; y <= r; ++y)
      {
         double e = -(x * x + y * y) / (2 * s * s);
         k[x + r][y + r] = exp(e) / (2 * M_PI * s * s);
         sum += k[x + r][y + r];
      }
   }
   for (int x = 0; x < ks; ++x)
   {
      for (int y = 0; y < ks; ++y) k[x][y] /= sum;
   }
}
void blur(const Mat &input, Mat &output, int ks, double s)
{
    int r = ks / 2;
    double** k = new double*[ks];
    for (int i = 0; i < ks; ++i)
    {
       k[i] = new double[ks];
    }
```

```
    gformula(ks, s, k);
    for (int x = 0; x < input.rows; ++x)
    {
        for (int y = 0; y < input.cols; ++y)
        {
            double R = 0.0, G = 0.0, B = 0.0;
            for (int kx = -r; kx <= r; ++kx)
            {
                for (int ky = -r; ky <= r; ++ky)
                {
                    int ix = x + kx;
                    int iy = y + ky;
                     if (ix >= 0 && ix < input.rows && iy >= 0 && iy <
input.cols)
                     {
                        Vec3b p = input.at<Vec3b>(ix, iy);
                        double w = k[kx + r][ky + r];
                        R += p[2] * w;
                        G += p[1] * w;
                        B += p[0] * w;
                    }
                }
            }
                output.at<Vec3b>(x,  y)[2]  =  static_cast<unsigned
char>(R);
                output.at<Vec3b>(x,  y)[1]  =  static_cast<unsigned
char>(G);
                 output.at<Vec3b>(x,  y)[0]  =  static_cast<unsigned
char>(B);
        }
    }
    for (int i = 0; i < ks; ++i)
    {
         delete[] k[i];
    }
    delete[] k;
}
```

```
 void greyscale(const Mat& input, Mat& output)
{
   output = Mat::zeros(input.size(), CV_8UC1);
   for (int x = 0; x < input.rows; ++x)
   {
      for (int y = 0; y < input.cols; ++y)
      {
         Vec3b p = input.at<Vec3b>(x, y);
        uchar gray =static_cast<uchar>((p[2] + p[1] + p[0])/3 );
         output.at<uchar>(x, y) = gray;
      }
   }
}

void brightness(const Mat& input, Mat& output, int b)
{
   for (int x = 0; x < input.rows; ++x)
   {
      for (int y = 0; y < input.cols; ++y)
      {
         Vec3b p = input.at<Vec3b>(x, y);
         for (int c = 0; c < 3; ++c)
         {
            int newValue = p[c] + b;
            newValue = min(max(newValue, 0), 255);
            output.at<Vec3b>(x, y)[c] = newValue;
         }
      }
   }
}
```

```
int main(int argc, char** argv)
{
   if (argc != 2)
   {
      cout << "Usage: ./blurImage <image_path>" << endl;
      return -1;
   }
   Mat inputImage = imread(argv[1]);
   if (inputImage.empty())
   {
        cerr << "Error: Could not open or find the image!" <<
endl;
      return -1;
   }

   Mat FinalImage = inputImage.clone();
   int c;
   cout<<”Enter the process:\n1.Blurring\n2.Greyscale\n
3.Brightness\n”;
   cin>>c;
   if(c==1)
   {
      int ks;
      double s;
      cout << "Enter Kernel size and sigma value: ";
      cin >> ks >> s;
      if (ks % 2 == 0)
      {
         ks += 1;
      }
      blur(inputImage, FinalImage, ks, s);
   }
```

```
 if(c==2)
 {
 greyscale(inputImage,FinalImage);
 }

if(c==3)
   {
      int b;
      cout<<"Enter Brightness level(-255 to 255):";
      cin>>b;
      brightness(inputImage,FinalImage,b);
   }
   imshow("Original Image", inputImage);
   imshow("Final Image", FinalImage);
   imwrite("Final_image.jpg", FinalImage);
   waitKey(0);
   return 0;
}
```

Thank you