# Biometrics
# Laboratory Report

Jakub Ciecierski

November 6, 2015

Date Performed: October 30, 2015

# Contents

|  |  |  |
|---|---|---|
| $-1$ | $-1$ | $-1$ |
| $-1$ | $9$ | $-1$ |
| $-1$ | $-1$ | $-1$ |

|  |  |  |
|---|---|---|
| $p_{(-1,-1)}$ | $p_{(-1,0)}$ | $p_{(-1,1)}$ |
| $p_{(0,-1)}$ | $p_{(0,0)}$ | $p_{(0,1)}$ |
| $p_{(1,-1)}$ | $p_{(1,0)}$ | $p_{(1,1)}$ |

Figure 1: Kernel matrix, and their relative positions

# 1 Objective

# 2 Convolution Filters

## 2.1 Introduction

The main idea of convolution filters is to modify a pixel value based on its neighbourhood. Value of a given pixel and its neighbours are multiplied by the corresponding element in the *kernel* matrix. We then sum all such products and normalize the final result. This final result is a new value for the given pixel.
Formally

$$I_{out}(x,y) = C_{off} + \frac{\sum_i \sum_j M(i,j) * I_{in}(x+i, y+j)}{D}$$

where:
$M$ - is the kernel matrix
$I_{in}$ - the input image
$I_{out}$ - the output image
$C_{off}$ - offset, usually 0
$D$ - divisor, for filters equal to $\sum_i \sum_j M(i,j)$, for others just 1

Kernel can be of any sizes (e.g. $(3 \times 3)$, $(5 \times 5)$, $(3 \times 5)$, $(15 \times 15)$). The most common ones are probably $(3 \times 3)$.
Notice that the summations interval of iterating the kernel matrix has been defined implicitly. To make it easier to compute corresponding pixel we fix a anchor point in the kernel matrix. The anchor point defines the origins of the kernel matrix. The figure 1 shows an example of $3 \times 3$ kernel matrix and matrix of indices of each corresponding element. In this example, the anchor point is fixed in the middle element of kernel matrix, i.e. under value 9.
It is probably appropriate to mention that convolution filters require quite a lot of computational power. Given a kernel matrix of size $n \times m$, calculation of a single pixel require $m * n$ multiplications. Thus for big kernel matrices the computations might become very slow.

## 2.2 Example

The example in figure 2 will hopefully make it clear how exactly convolution filters define value for a given pixel. The left most matrix presents an input image, the middle one is a kernel and the right most is a final result of the current pixel. The current pixel is marked with red border. The green border however, defines the neighbourhood of that pixel. It should be clear that the anchor point is again fixed under the middle point of the kernel matrix, thus defining its origins. Further, each pixel of the image matrix is multiplied by corresponding kernel value. All products are summed. Since the only pixel that is not multiplied by 0 has 42, the final result is simply 42.

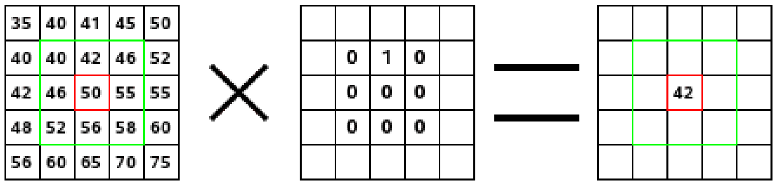


Figure 2: Example of convolution filter. The computation of a new value for pixel in red border. On the left: image matrix. On the right: kernel matrix. The right side is a new value for the current pixel

## 2.3 Special Cases

Finally special cases that might occur during computation of convolution filters are discussed. Firstly, however obvious, it is important that the final result is clipped to appropriate range. In case of standard RBG $[0, 255]$, we must ensure that the final values are within this range. Finally, what happens when we compute pixels that lay at the edge of the image. For instance, for the first pixel (starting from upper left corner), the corresponding pixel of kernel matrix will simply not exist. One of the following three solutions can be applied to fix that issue.

a. Compute as if there was additional row or column of pixel with values equal to the pixels at the edge.

b. Skip the pixels with neighbours outside the edge of an image.

c. The pixels in the neighbourhood outside the edge of an image assume to have value 0.

d. Wrap the image. pixel outside left edge use values of pixel on the right edge.

## 2.4 Implementation

The general algorithm flow of convolution filter is presented:

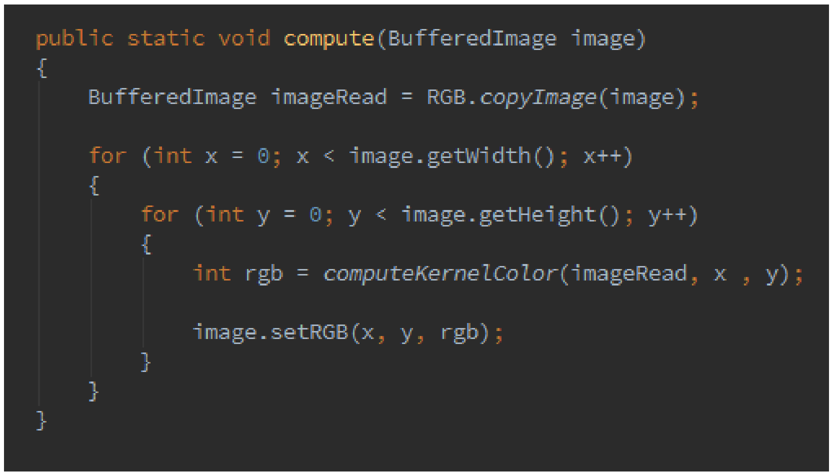
a. For *input* image, make a *output* copy.

b. For each pixel in the input image.

   (a) Compute new value using kernel matrix. The neighbourhood is also taken from input image.

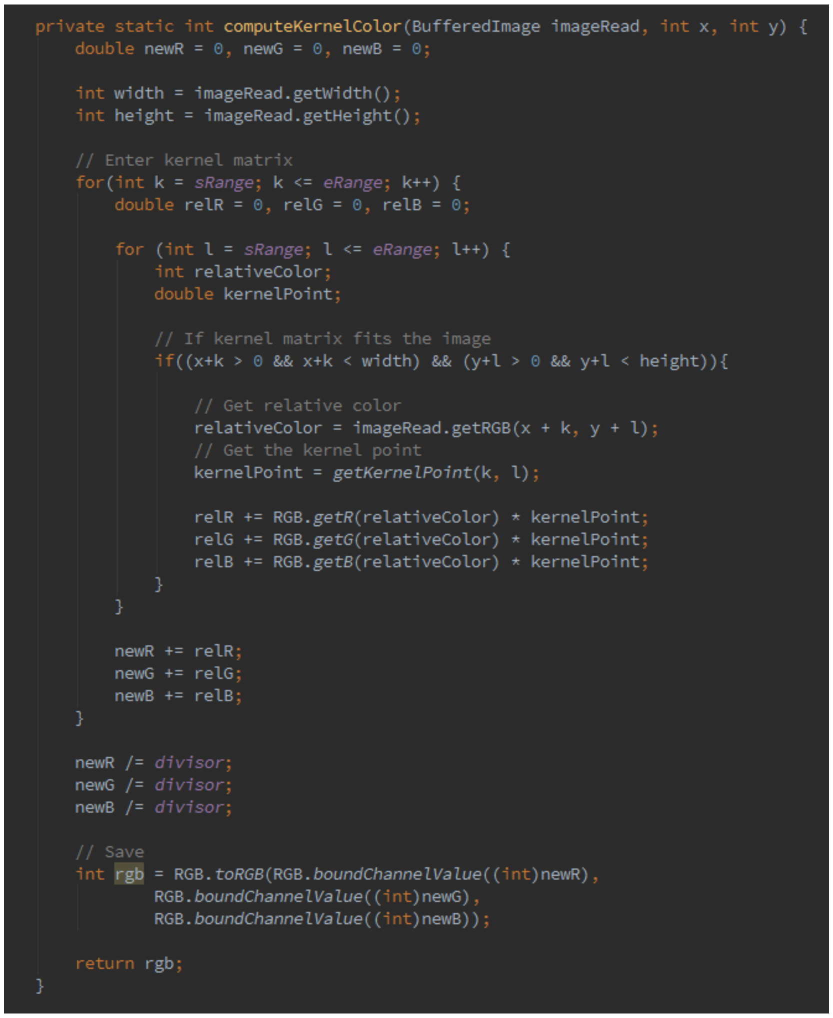
   (b) Save new value to output image.

First of all, the implementation of the main function called *compute* is shown in figure 3. The *compute* method calls *computeKernelColor* for each pixel. The implementation of this method is presented in figure **??**. The details of *computeKernelColor* method are now discussed. First of all, the two loops represent the summation over the kernel matrix. The *sRange* and *eRange*, standing for start range and end range respectively, corresponds to the indices of kernel matrix relative to the anchor point. The *getKernelPoint* is a simply method which returns the value of kernel value for indices relative to the anchor point. This method is shown in figure 5. Notice that *kernelMatrix* is one-dimensional vector.



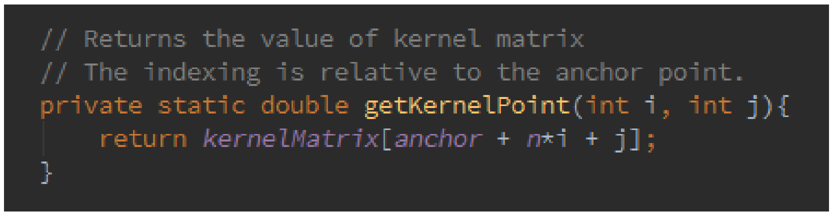
```
public static void compute(BufferedImage image)
{
    BufferedImage imageRead = RGB.copyImage(image);

    for (int x = 0; x < image.getWidth(); x++)
    {
        for (int y = 0; y < image.getHeight(); y++)
        {
            int rgb = computeKernelColor(imageRead, x , y);

            image.setRGB(x, y, rgb);
        }
    }
}
```

Figure 3: Main function of convolution filter



```java
private static int computeKernelColor(BufferedImage imageRead, int x, int y) {
    double newR = 0, newG = 0, newB = 0;

    int width = imageRead.getWidth();
    int height = imageRead.getHeight();

    // Enter kernel matrix
    for(int k = sRange; k <= eRange; k++) {
        double relR = 0, relG = 0, relB = 0;

        for (int l = sRange; l <= eRange; l++) {
            int relativeColor;
            double kernelPoint;

            // If kernel matrix fits the image
            if((x+k > 0 && x+k < width) && (y+l > 0 && y+l < height)){

                // Get relative color
                relativeColor = imageRead.getRGB(x + k, y + l);
                // Get the kernel point
                kernelPoint = getKernelPoint(k, l);

                relR += RGB.getR(relativeColor) * kernelPoint;
                relG += RGB.getG(relativeColor) * kernelPoint;
                relB += RGB.getB(relativeColor) * kernelPoint;
            }
        }

        newR += relR;
        newG += relG;
        newB += relB;
    }

    newR /= divisor;
    newG /= divisor;
    newB /= divisor;

    // Save
    int rgb = RGB.toRGB(RGB.boundChannelValue((int)newR),
            RGB.boundChannelValue((int)newG),
            RGB.boundChannelValue((int)newB));

    return rgb;
}
```

Figure 4: Compute Kernel



```java
// Returns the value of kernel matrix
// The indexing is relative to the anchor point.
private static double getKernelPoint(int i, int j){
    return kernelMatrix[anchor + n*i + j];
}
```

Figure 5: Get Kernel Point

## 2.5 High Pass Filter

High pass filters underscore components of the image with high frequency. Main objective of these filters is to sharpen an image. In other words, high pass filters preserve the high frequency information and reduce low frequency information.

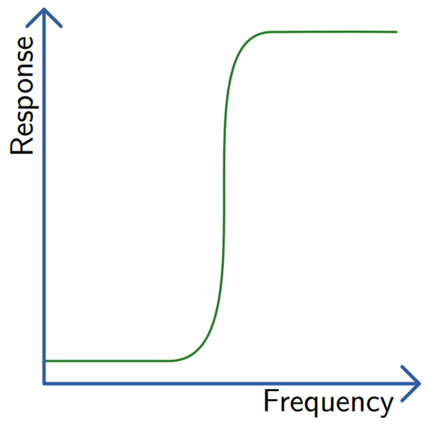
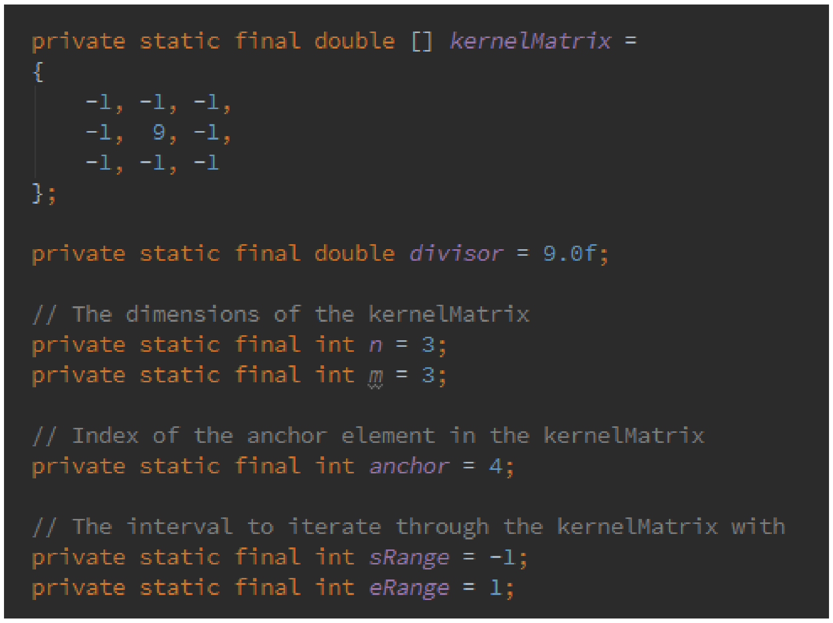


Figure 6: High Pass Filter

The implementation of an example of an high pass filter called a sharpen filter is shown in figure 7. The kernel is a $3 \times 3$ matrix with anchor point in the middle of that matrix. It was mentioned before that the kernel matrix is implemented using one dimensional vector. Thus it is clear to see that $kernelMatrix[anchor] = 9$



```java
private static final double [] kernelMatrix =
{
    -1, -1, -1,
    -1,  9, -1,
    -1, -1, -1
};

private static final double divisor = 9.0f;

// The dimensions of the kernelMatrix
private static final int n = 3;
private static final int m = 3;

// Index of the anchor element in the kernelMatrix
private static final int anchor = 4;

// The interval to iterate through the kernelMatrix with
private static final int sRange = -1;
private static final int eRange = 1;
```

Figure 7: High Pass Filter

## 2.6 Low Pass Filter

Low pass filter are used to smooth an image. By analogy, the low pass filters, preserve low frequency information and reduce the high frequency information. Implementation presented in figure 9.

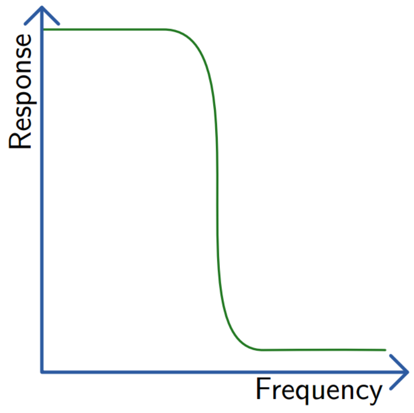
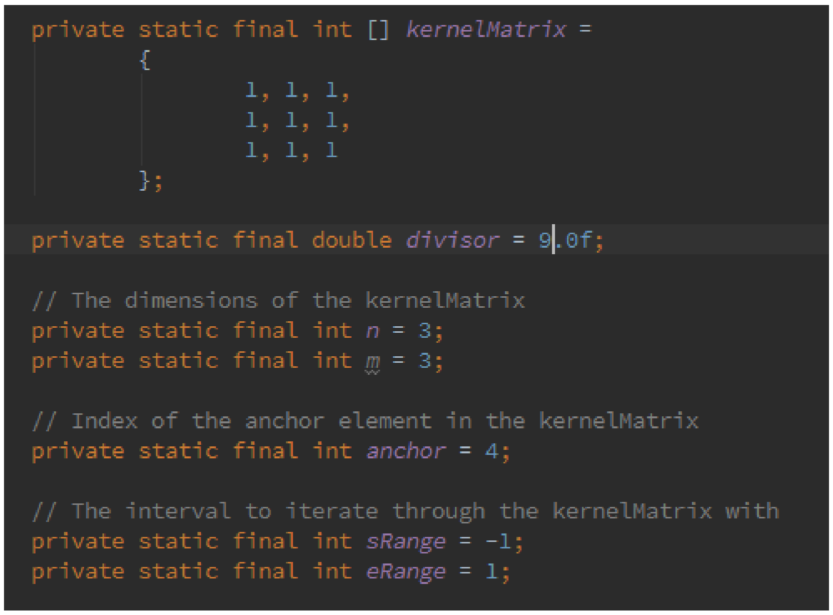


Figure 8: Low Pass Filter

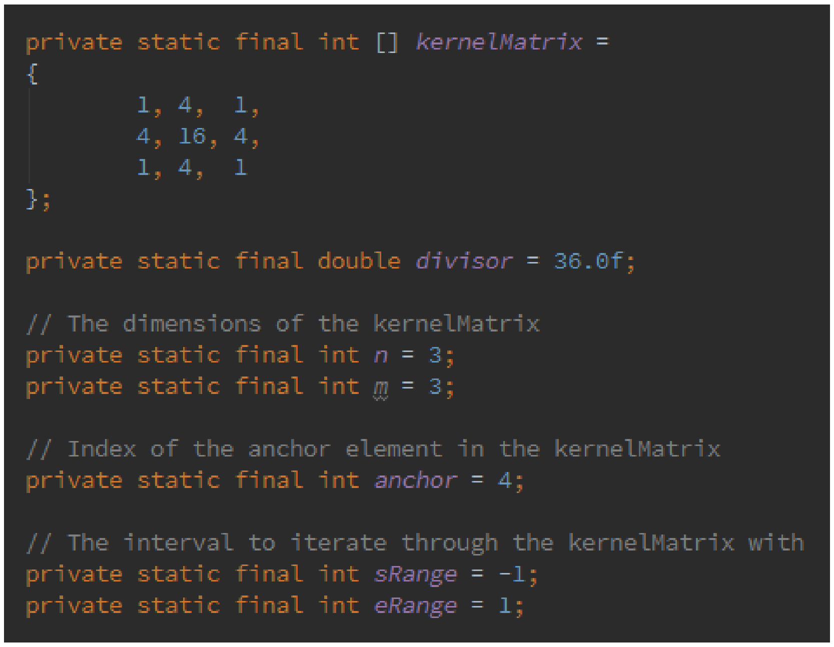


```java
private static final int [] kernelMatrix =
        {
                1, 1, 1,
                1, 1, 1,
                1, 1, 1
        };

private static final double divisor = 9.0f;

// The dimensions of the kernelMatrix
private static final int n = 3;
private static final int m = 3;

// Index of the anchor element in the kernelMatrix
private static final int anchor = 4;

// The interval to iterate through the kernelMatrix with
private static final int sRange = -1;
private static final int eRange = 1;
```

Figure 9: Low Pass Filter

## 2.7 Gaussian Filter

Gaussian filter is a low pass filter which is used to blur an image.
Figure 10 shows implementation.

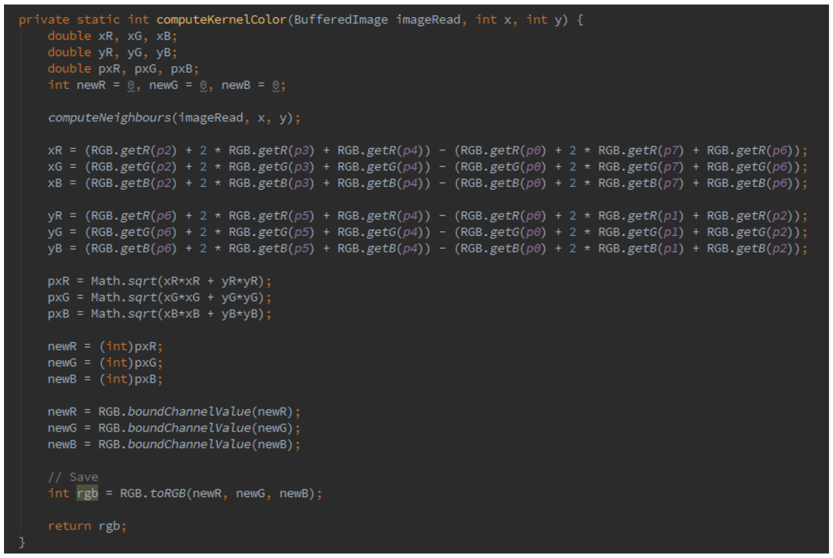


```java
private static final int [] kernelMatrix =
{
        1, 4,  1,
        4, 16, 4,
        1, 4,  1
};

private static final double divisor = 36.0f;

// The dimensions of the kernelMatrix
private static final int n = 3;
private static final int m = 3;

// Index of the anchor element in the kernelMatrix
private static final int anchor = 4;

// The interval to iterate through the kernelMatrix with
private static final int sRange = -1;
private static final int eRange = 1;
```

Figure 10: Gaussian Filter

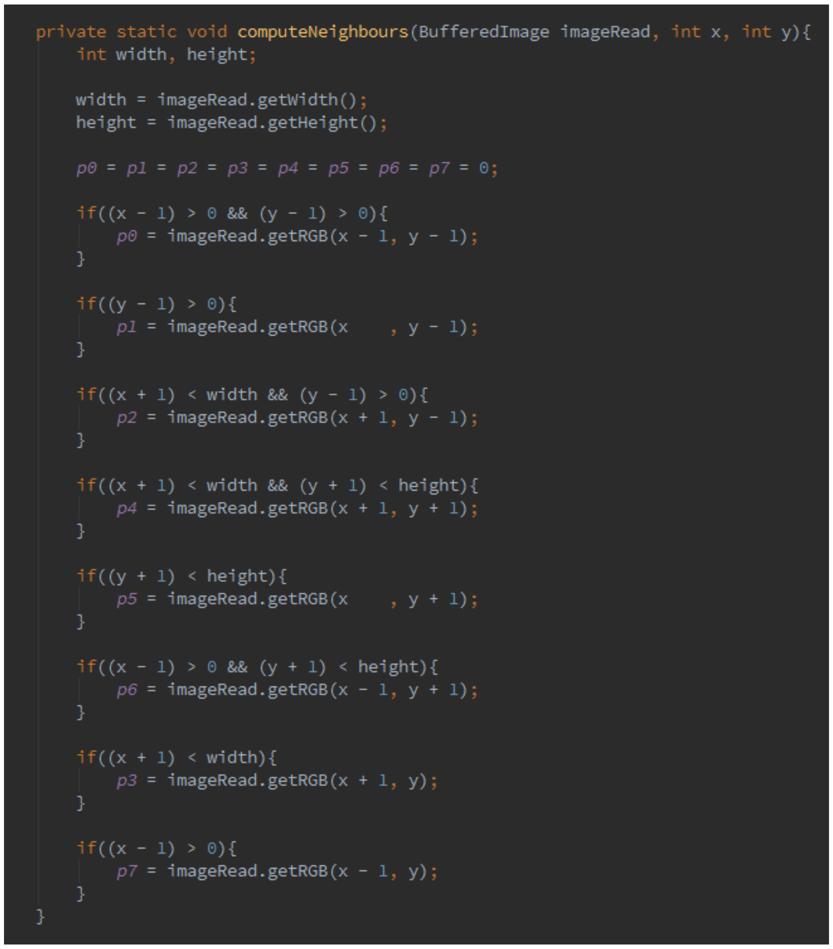
## 2.8 Sobel Filter

In case of Sobel filter, the implementation of *computeKernelColor* changes



```java
private static int computeKernelColor(BufferedImage imageRead, int x, int y) {
    double xR, xG, xB;
    double yR, yG, yB;
    double pxR, pxG, pxB;
    int newR = 0, newG = 0, newB = 0;

    computeNeighbours(imageRead, x, y);

    xR = (RGB.getR(p2) + 2 * RGB.getR(p3) + RGB.getR(p4)) - (RGB.getR(p0) + 2 * RGB.getR(p7) + RGB.getR(p6));
    xG = (RGB.getG(p2) + 2 * RGB.getG(p3) + RGB.getG(p4)) - (RGB.getG(p0) + 2 * RGB.getG(p7) + RGB.getG(p6));
    xB = (RGB.getB(p2) + 2 * RGB.getB(p3) + RGB.getB(p4)) - (RGB.getB(p0) + 2 * RGB.getB(p7) + RGB.getB(p6));

    yR = (RGB.getR(p6) + 2 * RGB.getR(p5) + RGB.getR(p4)) - (RGB.getR(p0) + 2 * RGB.getR(p1) + RGB.getR(p2));
    yG = (RGB.getG(p6) + 2 * RGB.getG(p5) + RGB.getG(p4)) - (RGB.getG(p0) + 2 * RGB.getG(p1) + RGB.getG(p2));
    yB = (RGB.getB(p6) + 2 * RGB.getB(p5) + RGB.getB(p4)) - (RGB.getB(p0) + 2 * RGB.getB(p1) + RGB.getB(p2));

    pxR = Math.sqrt(xR*xR + yR*yR);
    pxG = Math.sqrt(xG*xG + yG*yG);
    pxB = Math.sqrt(xB*xB + yB*yB);

    newR = (int)pxR;
    newG = (int)pxG;
    newB = (int)pxB;

    newR = RGB.boundChannelValue(newR);
    newG = RGB.boundChannelValue(newG);
    newB = RGB.boundChannelValue(newB);

    // Save
    int rgb = RGB.toRGB(newR, newG, newB);

    return rgb;
}
```

Figure 11: Compute Kernel



```java
private static void computeNeighbours(BufferedImage imageRead, int x, int y){
    int width, height;

    width = imageRead.getWidth();
    height = imageRead.getHeight();

    p0 = p1 = p2 = p3 = p4 = p5 = p6 = p7 = 0;

    if((x - 1) > 0 && (y - 1) > 0){
        p0 = imageRead.getRGB(x - 1, y - 1);
    }

    if((y - 1) > 0){
        p1 = imageRead.getRGB(x    , y - 1);
    }

    if((x + 1) < width && (y - 1) > 0){
        p2 = imageRead.getRGB(x + 1, y - 1);
    }

    if((x + 1) < width && (y + 1) < height){
        p4 = imageRead.getRGB(x + 1, y + 1);
    }

    if((y + 1) < height){
        p5 = imageRead.getRGB(x    , y + 1);
    }

    if((x - 1) > 0 && (y + 1) < height){
        p6 = imageRead.getRGB(x - 1, y + 1);
    }

    if((x + 1) < width){
        p3 = imageRead.getRGB(x + 1, y);
    }

    if((x - 1) > 0){
        p7 = imageRead.getRGB(x - 1, y);
    }
}
```

Figure 12: Compute Kernel

## 3 Pupil

To find the pupil on a image we must first remove as much noise as possible. To achieve this, blur filters are applied. Then, the image should be transformed to binary image - black and white. Further, Sobel filter will ensure that only edges of the pupil are left. At this point, the image should contain only one closed region - the pupil. Thus we need to apply an algorithm that finds a closed regions in an image. Once we have atleast one interior point we can apply scan line fill algorithm for that region. The longest line will represent diameter, and middle of that line will be the center of the eye.
Formally the flow of the algorithm for original color image is as follows:

a. Blur the original image - loose all the noise.

b. Convert to Grayscale

c. Convert Grayscale to Black and White image - binary image

d. Apply Sobel filter.

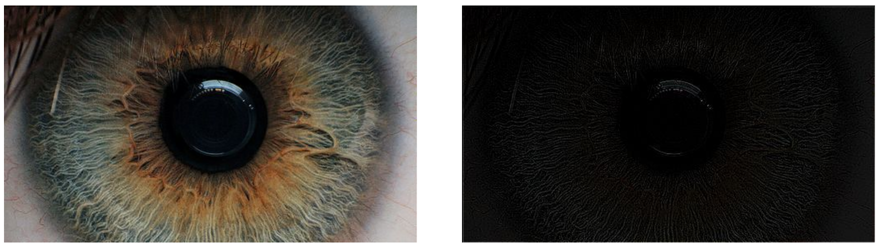
e. Find a closed region in the image - representing the pupil.

f. Use scan line fill algorithm to find diameter and centre - the longest line will be the diameter.

## 4 Results

### 4.1 Filters

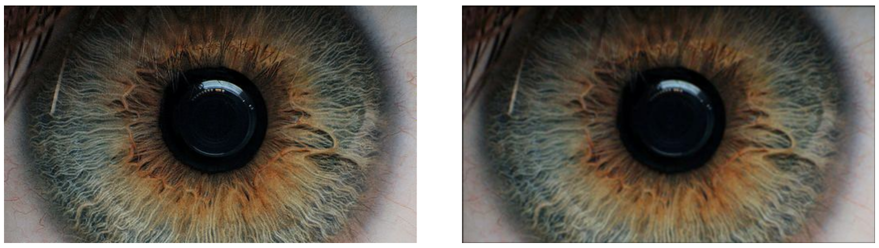
#### 4.1.1 High Pass



(a) Original image (b) High Pass filter

Figure 13: The original and High Pass filter

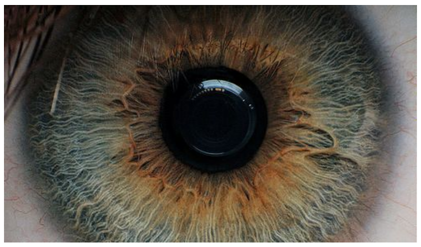
#### 4.1.2 Low Pass



(a) Original image (b) Low Pass

Figure 14: The original and Low Pass filter

### 4.1.3 Gaussian


(a) Original image

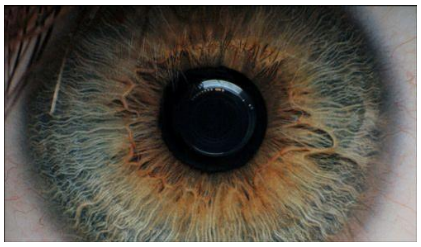
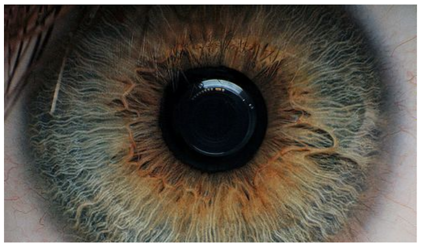

(b) Gaussian Filter

Figure 15: The original and gaussian filter

### 4.1.4 Sobel


(a) Original image

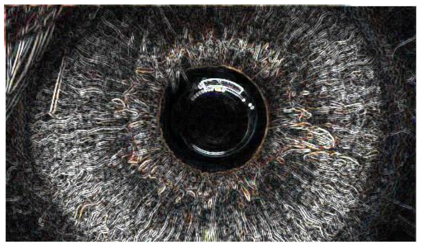
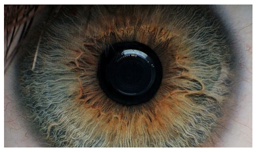
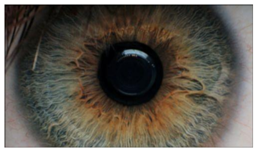

(b) Sobel filter

Figure 16: The original and Sobel filter

## 4.2 Pupil

The results of finding the pupil is now presented. First, the original image is subjected to low pass filtering. In this test case, the low pass filter was applied twice. Blurring is shown in figure 17. Further, figure 18 transforms the image to grayscale. By trail and error a threshold value 10 has been chosen to thresholding presented in figure 19. Finally, the last filter applied is the Sobel filter in figure 20.


(a) Original image


(b) Low Pass filter

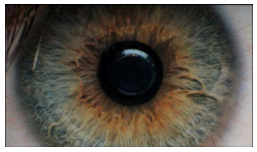
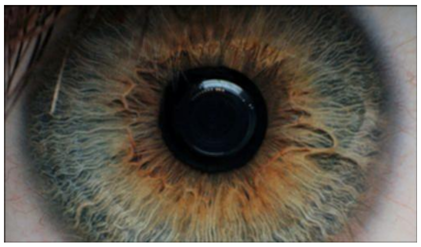
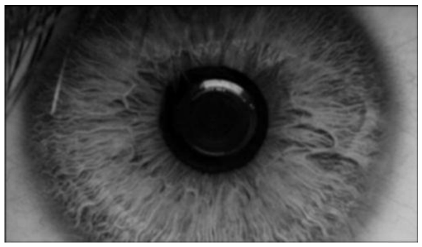

(c) Double Low Pass filter

Figure 17: Original to Double Low Pass filter

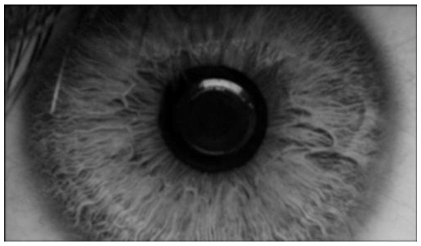


(a) Original image

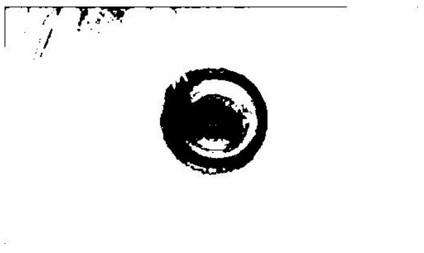


(b) Grayscale image

Figure 18: Low Pass to Grayscale

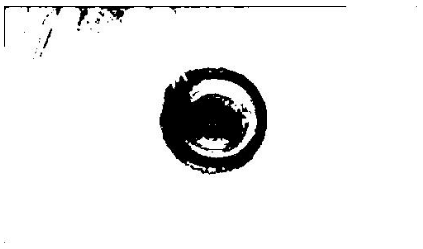


(a) Original image

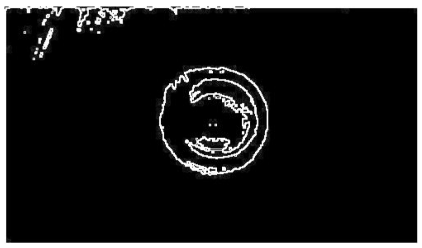


(b) Thresholding with threshold equal to 10

Figure 19: Grayscale to Threshold 10



(a) Original image



(b) Sobel filter

Figure 20: Binary to Sobel filter