Biometrics
Laboratory Report

Jakub Ciecierski

November 7, 2015

Date Performed: October 30, 2015

Contents

1 Objective

The objective of this study is to define an algorithm for finding a pupil in the eye image. In section 2 the tools required are showcased and implemented. Further, in section 3 the actual algorithm for extracting pupil is explained together with implementation. Results are presented in section 4. The last section is devoted to the conclusions of this study.

2 Methodology

2.1 Convolution Filters

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. (3×3), (5×5), (3×5), (15×15)). The most common ones are probably (3×3).
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 3×3 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.

-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

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.

2.1.1 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.

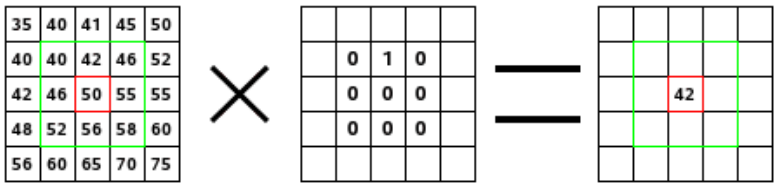


Figure 2: Example of convolution filter. The computation of a new value for pixel in red border. On the left: image matrix. In the middle: kernel matrix. The right side is a new value for the current pixel

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

The method of wrapping the image has been used in the following implementations.

2.1.3 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 4. 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 i, j;
            int relativeColor;
            double kernelPoint;

            i = x+k < 0 ? (width + (x+k)) : x+k;
            j = y+l < 0 ? (height + (y+l)) : y+l;
            i = i >= width ? (i - width) : i;
            j = j >= height ? (j - height) : j;

            // Get relative color
            relativeColor = imageRead.getRGB(i, j);
            // 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

2.2 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.

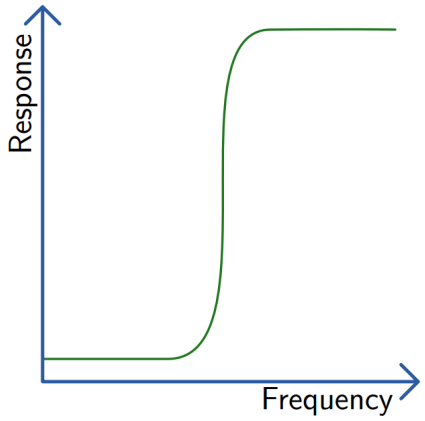
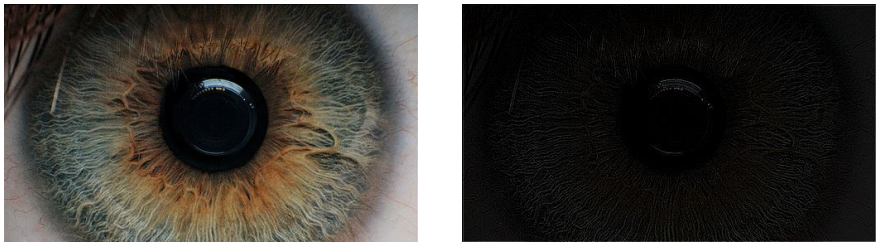


Figure 6: High Pass Filter

The implementation of an example of an high pass filter called a sharpen filter is shown in figure 7. The kernel is a 3×3 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 $kernelMatrix[anchor] = 9$. Result of applying High Pass filter is shown in figure 8.

```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



(a) Original image (b) High Pass filter

Figure 8: The original and High Pass filter

2.3 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 10. Example of blurring an image is presented in figure 11.

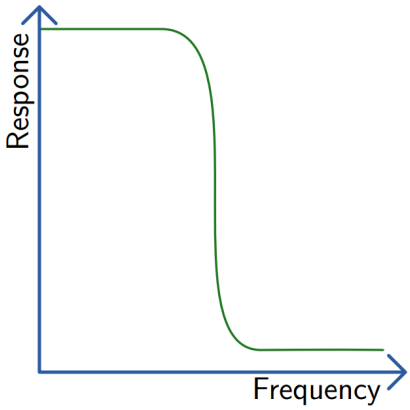
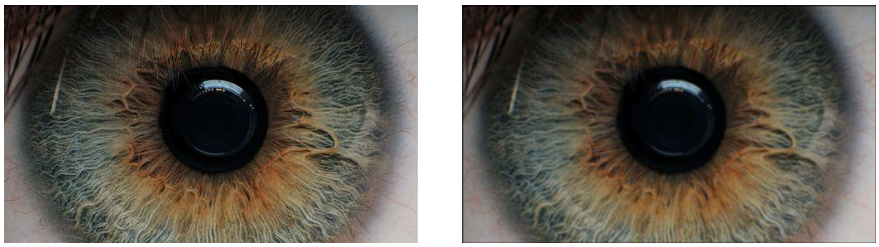


Figure 9: Low Pass Filter

```
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 10: Low Pass Filter



(a) Original image (b) Low Pass

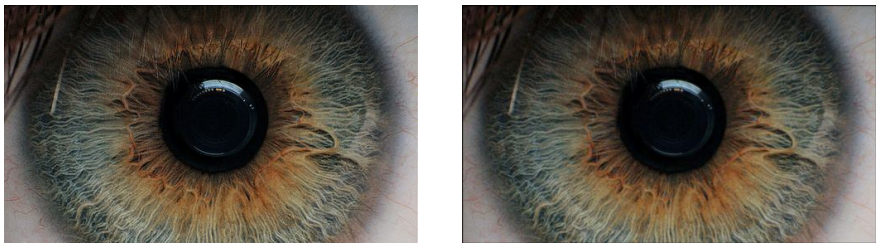
Figure 11: The original and Low Pass filter

2.4 Gaussian Filter

Gaussian filter is a low pass filter which is used to blur an image. It is usually used to reduce image noise and reduce detail.
Figure 12 shows implementation.
Figure 13 presents example of applyining the Gaussian filter.

```
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 12: Gaussian Filter



(a) Original image (b) Gaussian Filter

Figure 13: The original and gaussian filter

2.5 Sobel Filter

The Sobel filter is used to detect edges in the image.
Calculating a new value of pixel px is defined as follows:

$$x = (p_2 + 2 * p_3 + p_4) - (p_0 + 2 * p_7 + p_6)$$
$$y = (p_6 + 2 * p_5 + p_4) - (p_0 + 2 * p_1 + p_2)$$
$$px = \sqrt[2]{x^2 + y^2}$$

where pixels p_0 through p_7 are neighbours of p_x. The neighbourhood is defined in figure 14.
The figures 15 and 16 shows the implementation of the filter.
Finally the example of application is shown in figure 17.

p_0	p_1	p_2
p_7	p_x	p_3
p_6	p_5	p_4

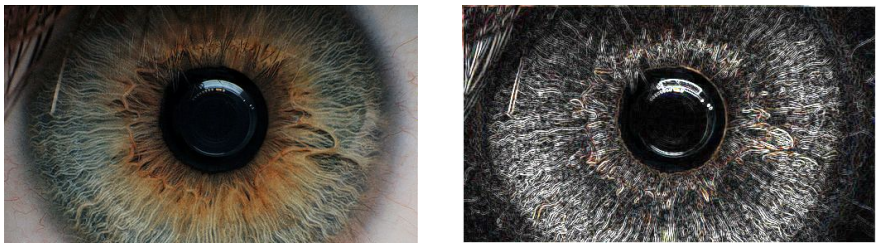
Figure 14: Neighbourhood in Sobel Filter.

```
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 15: Compute Kernel

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

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

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

    i = (x-1) < 0 ? 0 : x-1;
    j = (y-1) < 0 ? 0 : y-1;
    p0 = imageRead.getRGB(i, j);

    i = x;
    j = (y-1) < 0 ? 0 : y-1;
    p1 = imageRead.getRGB(i, j);

    i = (x+1) >= width ? width-1 : x+1;
    j = (y-1) < 0 ? 0 : y-1;
    p2 = imageRead.getRGB(i, j);

    i = (x+1) >= width ? width - 1 : x+1;
    j = (y+1) >= height ? height - 1 : y+1;
    p4 = imageRead.getRGB(i, j);

    i = x;
    j = (y+1) >= height ? height - 1 : y+1;
    p5 = imageRead.getRGB(i, j);


    i = (x-1) < 0 ? 0 : x-1;
    j = (y+1) >= height ? height - 1 : y+1;
    p6 = imageRead.getRGB(i, j);

    i = (x+1) >= width ? width-1 : x+1;
    j = y;
    p3 = imageRead.getRGB(i, j);

    i = (x-1) < 0 ? 0 : x-1;
    j = y;
    p7 = imageRead.getRGB(i, j);

}
```

Figure 16: Compute Kernel



(a) Original image (b) Sobel filter

Figure 17: The original and Sobel filter

2.6 Flood Fill

The main objective of Flood Fill algorithm is to color all the connected pixel with the same color. Starting from a source pixel, the image will be flooded with target color as long as all the pixel remain equal to the source one.
There exist different variations of Flood Fill:

a. Recursive - no explicit data structure is used, only pure recursion.

b. Iterative

   (a) Depth-first. Uses stack.

   (b) Breadth-first. Uses queue.

2.6.1 Implementation

The implemented version is the Iterative Breadth-first, using the queue. The figure 18 presents the method of Flood Fill. It enqueues the coordinate of the source pixel. Then, we dequeue that pixel and check whether it is equal to the background pixel. If so add four neighbours to the queue. This process is repeated until the queue is empty.

```
 */
public static void compute(BufferedImage image,
                           int startX, int startY,
                           Pixel labelPixel)
{
    int width, height;
    Pixel backgroundPixel;

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

    backgroundPixel = new Pixel(0, 0, 0);

    LinkedList<int[]> q = new LinkedList<>();

    q.addFirst(new int[]{startX, startY});

    while (!q.isEmpty()) {
        Pixel pixel;
        int[] node = q.removeLast();
        int x = node[0];
        int y = node[1];

        if ((x >= 0) && (x < width) &&
            (y >= 0) && (y < height) &&
            (new Pixel(image.getRGB(x, y)))
                .equals(backgroundPixel)) {
            image.setRGB(x, y, labelPixel.getColor());

            q.addFirst(new int[]{x + 1, y});
            q.addFirst(new int[]{x, y + 1});

            q.addFirst(new int[]{x, y - 1});
            q.addFirst(new int[]{x - 1, y});
        }
    }
}
```

Figure 18: The Flood Fill algorithm

3 Pupil Extraction

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 the interior points (points of the pupil) we can apply simple scan line algorithm to analyse the pupil. The longest line will represent diameter, and middle point of that line will be the center of the eye. By interior of the pupil we mean all the pixels belonging to the pupil. The exterior is everything else, i.e. the background.
Formally the flow of the algorithm for original color image is as follows:

a. Filter Phase - Leave only pupil's edges.
   (a) Apply Low Pass Filter to original image
   (b) Convert image to binary image:
      i. Apply Grayscale.
      ii. Apply Thresholding for constant threshold.
   (c) Apply Sobel Filter.

b. Find closed region. At this point, an image should contain a black background with white edges (a circle), representing interior of the pupil.
   (a) Apply Flood Fill algorithm on the image starting from the upper left pixel. Label the exterior with $labelColor$ - different to black or white. All none $labelColor$ pixels represent the interior of the pupil.
   (b) Fill the interior with pupil with $pupilColor$.
   (c) Now, the image is binary again, with pupil's interior position known.

c. Find Diameter. Apply simplified scan line algorithm
   (a) The longest line of the pupil will be the diameter.
   (b) Center of the eye is the center of diameter line.

There exist many problem with this algorithm. First of all, what blur filter to choose. Namely how much information we want to loose. Another, more obvious problem is choosing the constant for thresholding. Even though methods of estimating threshold exist, in this experiment a constant value will be chosen by method of trail and error.

3.1 Implementation

The java implementation of the algorithm for finding the pupil is now presented. The following subsections break down the algorithm into the corresponding steps.

3.1.1 Main

Let's start with figure 19 showing the parameters and the logic of the algorithm. The threshold is a constant value, in the experiment the value 4 is used. The constant $lowPassCount$ tells us how many times the image should be subjected to Low Pass filter. The $maxStartX$ and $maxEndX$ variables represent the start and end of a diameter line respectively. In analogy the $maxY$ is the y-coordinate of the diameter line. Further, the *diameter* and *center* are simply the length and center of the diameter line. The color to color the diameter and its center are $diameterPixel$ and $centerPixel$. Lastly, the exterior of the pupil will take color $labelPixel$ and the pupil it self will take $pupilPixel$.
The method *compute* calls three other methods defining each step of the algorithm.

```
private static final int threshold = 4;

// How many times Low Pass filter is applied
private static final int lowPassCount = 5;

// The beginning x-coordinate of pupil's diameter
private static int maxStartX;
// The end of x-coordinate pupil's diameter
private static int maxEndX;
// The y-coordinate of pupil's diameter
private static int maxY;

// The Diameter of the pupil
private static int diameter;
// The center of the pupil
private static int center;

// The color of the diameter
private static Pixel diameterPixel = new Pixel(255, 0, 0);
// The color of the center
private static Pixel centerPixel = new Pixel(0, 255, 0);

// The color of pupil's exterior. Everything that is not pupil is color with that pixel
private static Pixel labelPixel = new Pixel(125, 125, 125);
// The color of pupil. Pupil is colored with that pixel
private static Pixel pupilPixel = new Pixel(0, 0, 0);

// Starts computations for finding pupil in an image.
public static void compute(BufferedImage image){
    filterPhase(image);

    findCloseRegion(image);

    findDiameter(image);
}
```

Figure 19: The parameters and compute function for pupil algorithm

3.1.2 Filter Phase

Let's turn our attention to figure 20 presenting the Filter Phase. First of all, the original image is undertaken several Low Pass filters. Then the image is turned into Grayscale and the processes of blurring is repeated. Finally Theshold is computed turning the image into a binary one and Sobel filter is used to leave

only the edges of the pupil.

```
private static void filterPhase(BufferedImage image){
    for(int i = 0; i < lowPassCount; i++)
        LowPass.compute(image);

    Grayscale.compute(image);

    for(int i = 0; i < lowPassCount; i++)
        LowPass.compute(image);

    Treshold.compute(image, threshold);
    SobelFilter.compute(image);
}
```

Figure 20: The $filterPhase$ method

3.1.3 Find Closed Region

At this point, we assume to have a black and white picture, where white pixel represent the edges of the pupil. Although, it is possible that the pupil it self has some white noise inside it. The process of finding the closed region of the pupil is shown in figure 21. Firstly, the $findClosedRegion$ methods calls Flood Fill algorithm starting from the upper left edge of the image. This process will label the exterior of the pupil with $labelPixel$. Now, we simply fill the interior of the pupil with $pupilPixel$. Notice that the interior of the pupil is simply everything that has not been filled by the Flood Fill algorithm. The $fillPupil$ method explains exactly the process of filling the pupil.

```java
private static void findClosedRegion(BufferedImage image){
    // Compute flood fill.
    // All the pixel not covered by flood fill are a closes region
    int startX = 0;
    int startY = 0;

    FloodFill.compute(image, startX, startY, labelPixel);

    // Fill the closed region (i.e. the pupil) with black pixels
    fillPupil(image, labelPixel, pupilPixel);
}

// Fill all the pixel of closed regions with pupilPixel
private static void fillPupil(BufferedImage image,
                              Pixel skipPixel, Pixel pupilPixel) {
    for (int x = 0; x < image.getWidth(); x++) {
        for (int y = 0; y < image.getHeight(); y++) {

            Pixel pixel = new Pixel(image.getRGB(x, y));

            if(!pixel.equals(skipPixel)){
                image.setRGB(x, y, pupilPixel.getColor());
            }
        }
    }
}
```

Figure 21: The *findClosedRegion* method

3.1.4 Find Diameter

The final phase assumes to have an binary image with no noise it in. The image should contain only the exterior and interior pixels, with regards to the pupil. The figures 22 and 23 present the *findDiameter* and *findLongestScanLine* methods. Let's focus on the latter method. The *findLongestScanLine* will iterate through every row of the image, scanning from the left to the right edge of the image. Whenever the scanline encounters a *pupilPixel*, it will fetch the next pixel and see whether it is also a *pupilPixel*. In this easy manner it will calculate a length of each horizontal line of the pupil. The longest line is chosen to be the diameter. The *findDiamter* method after calculating the diameter will output the results and draw diameter in the binary image. The drawing methods are presented in figure 24.

```
private static void findDiameter(BufferedImage image){
    // Find the longest scan line
    findLongestScanLine(image, pupilPixel);

    drawLine(image, diameterPixel);

    // Draw the center.
    diameter = maxEndX - maxStartX;
    center = diameter/2 + maxStartX;
    SmartConsole.Print("Center: " + center + " Diameter: " + diameter);

    drawLineCenter(image, center, maxY, centerPixel);
}
```

Figure 22: The *findDiameter* method

```
private static void findLongestScanLine(BufferedImage image,
                                Pixel pupilPixel){
    Pixel currPixel, nextPixel;
    int currLineLength, maxLineLength;
    int startX;

    startX = 0;
    maxStartX = maxEndX = 0;

    maxLineLength = 0;
    maxY = -1;

    // For each row
    for (int y = 0; y < image.getHeight(); y++) {
        currLineLength = 0;
        for (int x = 0; x < image.getWidth() - 1; x++) {
            currPixel = new Pixel(image.getRGB(x, y));
            nextPixel = new Pixel(image.getRGB(x+1, y));

            // The line is ongoing. Potentially new line
            if(currPixel.equals(pupilPixel) &&
                nextPixel.equals(pupilPixel)) {
                // Check if it is a new line
                if(currLineLength == 0) {
                    startX = x;
                }
                currLineLength++;
            }
            // Line disconnected. Line Finished.
            else{
                // Check if it is the longest thus far
                if (currLineLength > maxLineLength) {
                    maxLineLength = currLineLength;
                    maxY = y;
                    maxStartX = startX;
                    maxEndX = x;
                }
                // Reset the length
                currLineLength = 0;
            }

        }
        if (currLineLength > maxLineLength) {
            maxLineLength = currLineLength;
            maxY = y;
            int x = image.getWidth() - 1;
            maxStartX = startX;
            maxEndX = x;
        }
    }
}
```

Figure 23: The *findLongestScanLine* method

```
private static void drawLineCenter(BufferedImage image,
                                   int x, int y,
                                   Pixel pixel){
    image.setRGB(x, y, pixel.getColor());
    image.setRGB(x-1, y, pixel.getColor());
    image.setRGB(x+1, y, pixel.getColor());

    image.setRGB(x, y + 1, pixel.getColor());
    image.setRGB(x-1, y + 1, pixel.getColor());
    image.setRGB(x+1, y + 1, pixel.getColor());

    image.setRGB(x, y - 1, pixel.getColor());
    image.setRGB(x-1, y - 1, pixel.getColor());
    image.setRGB(x+1, y - 1, pixel.getColor());
}

private static void drawLine(BufferedImage image, Pixel linePixel){
    // Draw the diameter line
    for (int x = maxStartX; x < maxEndX; x++) {
        image.setRGB(x, maxY, linePixel.getColor());
    }

}
```

Figure 24: The methods used to draw the diameter and center

4 Results

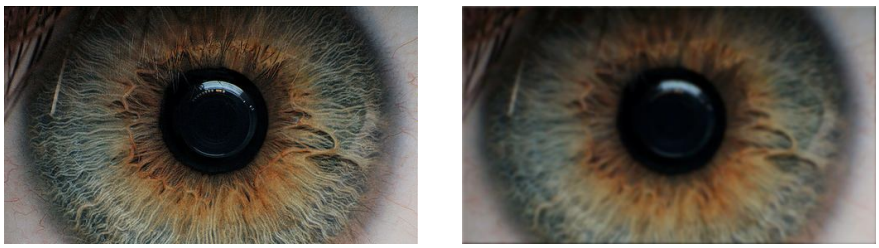
In this section the results of the pupil extraction are presented.

(a) Original 1 (b) Original 2 (c) Original 3

Figure 25: Three images to be tested

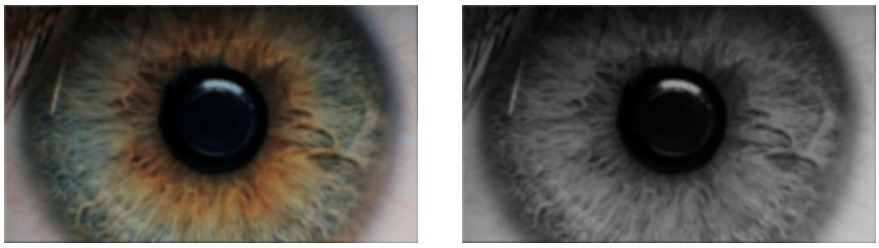
The figure 25 shows three eye images that will be tested with previously defined algorithm. The Original 1 image will be broken down into pieces, showing effect after each step of the algorithm. The other two will have the final result presented.
The result of finding the pupil have the following structure. Each figure will present a pair of images, first one prior to transformation and the second one after. Moreover, each figure's first image will be the previous figure's second

one. Hopefully the resulting effect will help to understand the process of finding the pupil interior.
The first phase of the algorithm is the filtering phase. The figure 26 presents the result of blurring the original image, after applying the Low Pass filter 5 times. In figure 27, the blurred image has been subjected to Grayscale transformation. Further, in figure 28 the Grayscale image has been blurred even more, by applying another 5 Low Pass filters. Such image is then turned to a binary image using thresholding filter with the threshold equal to 4 - figure 29. The final transformation in the first phase is the Sobel filter. The resulting image is presented in figure 30.
At this point, we constructed an image containing the edges of the pupil. The pupil it self still contains a lot noise. To clear that noise, the Flood Fill is used. The resulting effect is shown in figure 31. The exterior of the pupil is turned into gray color, while the interior has been cleaned to the black color.
Finally the simplified scanline algorithm is applied to find the diameter. The resulting diameter and the center of the eye is shown in the figure 32. The length of the diameter is equal to 128 pixels.
To compare the original image with the final result, figure 33 has been constructed.
The result of the two other original images are presented in figures 34 and 34,



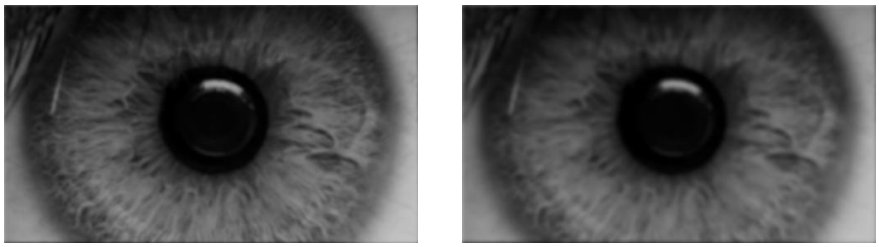
(a) Original image (b) Blurred image.

Figure 26: Original to Low Pass filter



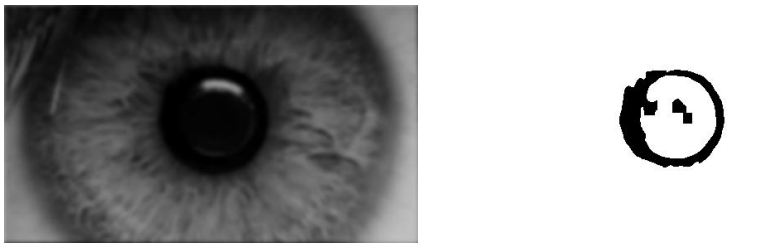
(a) Blurred image (b) Grayscale image

Figure 27: Low Pass to Grayscale



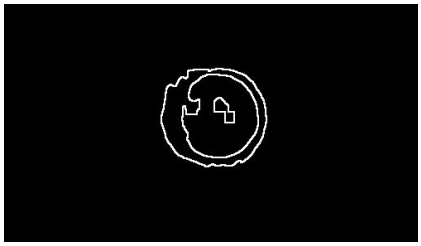
(a) Grayscale image (b) Blurred Grayscale

Figure 28: Further blurring of the Grayscale image



(a) Blurred Grayscale (b) Binary image

Figure 29: Threshold fitler with value 4



(a) Binary image (b) Sobel filter

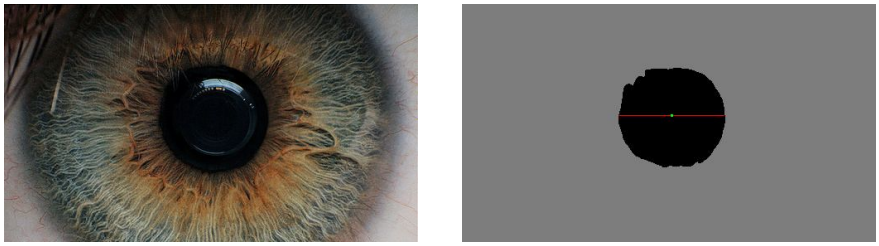
Figure 30: Binary to Sobel filter

(a) Sobel filter (b) The pupil

Figure 31: The pupil is found.

(a) The pupil (b) The diameter and center.

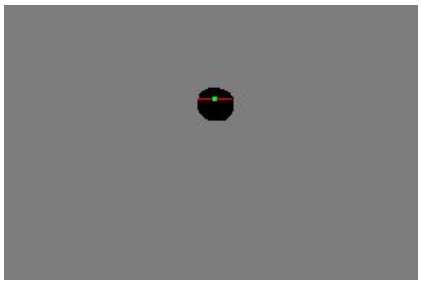
Figure 32: Drawing the diameter and center



(a) The original (b) The diameter and center.

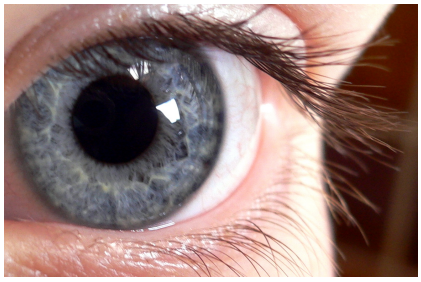
Figure 33: Original compared to Final result

(a) The original 2

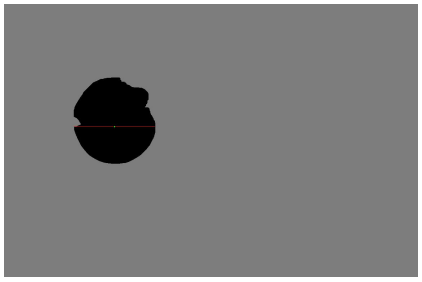


(b) The diameter and center.

Figure 34: Original 2 compared to Final result. Threshold = 4. Low Pass filter application count is 4.



(a) The original 3



(b) The diameter and center.

Figure 35: Original 3 compared to Final result. Threshold = 7, Low Pass filter application count is 10

5 Conclusions

The objective of this article focused on finding pupil in the eye image. The proper tools has been presented and finally the main algorithm for extracting pupil has been defined.
The algorithm has worked successfully for the given input image. Although the entire filtering phase has been prepared by trail and error. Further research into blurring methods and threshold evaluation is required.
Two other randomly chosen images of the eye have been presented and the results are quite good. One of the images required change of the threshold parameter and Low Pass filter application count.
The algorithm also assumes only one closed region (pupil) in the image. The Flood Fill phase could be easily generalized using multiple region labels. One for background and others for two eyes.