Analysis and Processing of Biometric Images
Laboratory 2

Bartomiej Dybisz

October 20, 2015

Date Performed:	October 16 , 2015
Instructor:	mgr Piotr Panasiuk

1 Objectives

Tresholding
To learn how to implement tresholding - in particular binarization. In addition, to see how it affects processed image.

Histogram Normalization
To learn how to implement two approaches of histogram normalization and to determine differences between output images.

Brightness
To learn how to implement brightness control over pixels of an image.

1.1 Definitions

Remark: throughout this report I will assume that each of red, green, blue channels of a pixel can take values from 0 to 1. It is imposed by JavaFX 8 - technology used to implement algorithms.

Tresholding A process of creating a black-and-white image out of a grayscale image consisting of setting exactly those pixels to white whose value is above a given threshold, setting the other pixels to black [1].

Image Histogram An image histogram is a graphical representation of the number of pixels in an image as a function of their intensity. Histograms are made up of bins, each bin representing a certain intensity value range. The histogram is computed by examining all pixels in the image and assigning each to a bin depending on the pixel intensity. The final value of a bin is the number of pixels assigned to it. The number of bins in which the whole intensity range is divided is usually in the order of the square root of the number of pixels [2].

Histogram Normalization In image processing, normalization is a process that changes the range of pixel intensity values. Applications include photographs with poor contrast due to glare, for example. Normalization is sometimes called contrast stretching or histogram stretching [3].

As one can find in literature, general equation for this process is as follows:

$$(\forall p \in pixels)(I(p) = (I(p) - minPix) * \frac{intMax - intMin}{maxPix - minPix} + intMin)$$

, where $intMax$ and $intMin$ correspond to the limits over which image intensity values will be extended, $minPix$ and $maxPix$ are values of minimal and maximal saturation of image's pixels and $I(p)$ defines intensity of pixel p.

However, since we discuss only standard 8-bit grayscale images, values will be stretched over $\langle 0; 1\rangle$ interval (as mentioned in remark to this section), hence $intMax = 1$ and $intMin = 0$, which will cause above equation to look as:

$$(\forall p \in pixels)(I(p) = (I(p) - minPix) * \frac{1}{maxPix - minPix}) \quad (1)$$

In our experiment we assumed that input picture is not grayscale, hence it is not true that for every channel saturation of red, green and blue channels must be the same. In consequence, we apply equation 1 to each channel separately. In addition, two approaches were adopt;

- $maxPix$ and $minPix$ were chosen for each channel separately (i.e. applying 1 to channel red, we consider minimum and maximum saturation of red channel of the whole image. Similarly for green and blue channel).
- $maxPix$ and $minPix$ were calculated as an average of minimal and maximal values of all three channels.

Brightness Image brightness (or luminous brightness) is a measure of intensity after the image has been acquired with a digital camera or digitized by an analog-to-digital converter[4].

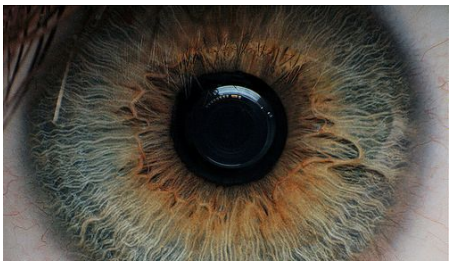
Brightness Manipulation We think of brightness manipulation as an increasing or decreasing intensity of each pixel. To express this we used following formula:

$$(\forall p \in pixels)(I(p) = I(p) + brigthness) \quad (2)$$

It is important to remember about clamping values of all pixels between 0 and 1 (convention assumed in remark to this section).

2 Experimental Data

Since laboratory concerned processing of images, it is not a surprise that our data were images. Although during session we check algorithms only for one picture, I'll demonstrate results on 3 additional ones to enrich this report.



(a)

(b)

(c)

(d)

Figure 1: Images used to check implemented algorithms.

3 Sample Code

This section contains code snippets with algorithms implementation. Each figures is provided with appropriate description either in caption or in method's comment.

3.1 Operations Class Diagram

Classes performing operations (tresholding, histogram normalization and brightness manipulation) inherit from Filter class. Such approach has been pursued because it may happen in future that polimorphism will be useful with e.g. dealing with list of filters, which should be applied to one image. Figure 2 depicts dependencies.

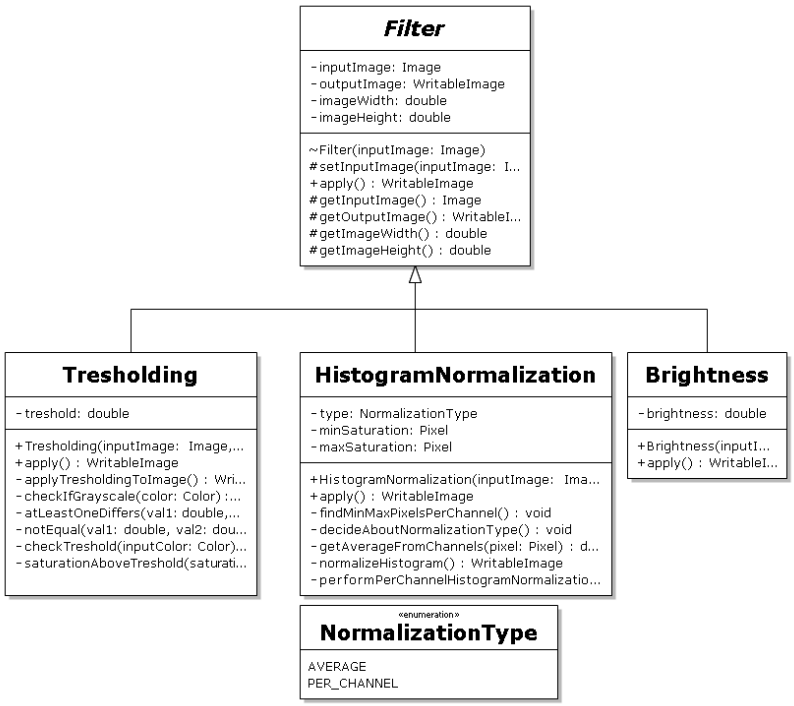


Figure 2: Class diagram of implemented operations.

3.2 Tresholding Implementation

```java
/**
 * Method applies thresholding (with treshold determined by{@link #treshold})
 * to the input inputImage (i.e. {@link #inputImage}. Treshold is assumed to
 * be from interval <0;255> for user convenience. Image must be grayscale.
 *
 * @return Image after tresholding ({@link #outputImage} field) or null in case
 * when error occurred.
 */
@Override
public WritableImage apply() {
    try {
        return applyTresholdingToImage();
    } catch (Exception e) {
        System.out.println("ERROR: " + e.getMessage());
    }
    return null;
}

private WritableImage applyTresholdingToImage() throws Exception {
    PixelReader inputReader = getInputImage().getPixelReader();
    PixelWriter outputWriter = getOutputImage().getPixelWriter();

    for (int x = 0; x < getImageWidth(); x++) {
        for (int y = 0; y < getImageHeight(); y++) {
            Color inputColor = inputReader.getColor(x, y);
            checkIfGrayscale(inputColor);
            Color outputColor = checkTreshold(inputColor);
            outputWriter.setColor(x, y, outputColor);
        }
    }
    return getOutputImage();
}
```

Figure 3: Overview of high abstraction level methods for tresholding. As one can see it may happens that input image is not in grayscale, in such a case appropriate exception is thrown (see figure 4).

```java
private void checkIfGrayscale(Color color) throws Exception {
    double red = color.getRed();
    double green = color.getGreen();
    double blue = color.getBlue();

    if (atLeastOneDiffers(red, green, blue)) {
        throw new NotGrayscaleException();
    }
}
```

Figure 4: Saturation on all channels is checked for equality.

```java
/**
 * @param inputColor Grayscale image.
 * @return Black or white color.
 */
private Color checkTreshold(Color inputColor) {
    Color outputColor;
    double inputOpacity = inputColor.getOpacity();
    double inputSaturation = inputColor.getRed();

    if (saturationAboveTreshold(inputSaturation)) {
        outputColor = new Color(0, 0, 0, inputOpacity);
    } else {
        outputColor = new Color(1, 1, 1, inputOpacity);
    }
    return outputColor;
}

private boolean saturationAboveTreshold(double saturation) {
    if (255.0 * saturation > treshold) {
        return true;
    } else {
        return false;
    }
}
```

Figure 5: Here we can see one of the results of remark given to section 1.1 - color saturation must be multiplied by 255.

3.3 Histogram Normalization Implementation

```java
/**
 * Normalizes histogram of input image ({@link #inputImage}) according to preferred
 * type ({@link #type}.
 *
 * @return Processed image, with normalized histogram; {@link #outputImage}.
 */
@Override
public WritableImage apply() {
    findMinMaxPixelsPerChannel();
    decideAboutNormalizationType();
    return normalizeHistogram();
}
```

Figure 6: Main, highly abstract method applying histogram normalization.

```
private void decideAboutNormalizationType() {
    switch (type) {
        /* To preserve generality of performPerChannelHistogramNormalization method,
           we set minimum and maximum for all "per channel operations" at the same value,
           which is an average of minimal and maximal saturation of channels, respectively */
        case AVERAGE:
            double min = getAverageFromChannels(minSaturation);
            double max = getAverageFromChannels(maxSaturation);
            minSaturation.setRGB(min, min, min);
            maxSaturation.setRGB(max, max, max);
            break;
    }
}

private double getAverageFromChannels(Pixel pixel) {
    return (pixel.getR() + pixel.getG() + pixel.getB()) / 3.0;
}
```

```
private WritableImage normalizeHistogram() {
    PixelReader inputReader = getInputImage().getPixelReader();
    PixelWriter outputWriter = getOutputImage().getPixelWriter();

    for (int x = 0; x < getImageWidth(); x++) {
        for (int y = 0; y < getImageHeight(); y++) {
            Color inputColor = inputReader.getColor(x, y);
            Pixel inputPixel = new Pixel(inputColor);
            Pixel outputPixel = performPerChannelHistogramNormalization(inputPixel);
            outputPixel.clampAllChannels();
            outputWriter.setColor(x, y, outputPixel.toColor());
        }
    }
    return getOutputImage();
}

private Pixel performPerChannelHistogramNormalization(Pixel pixel) {
    double redSaturation = (pixel.getR() - minSaturation.getR()) *
            (1.0 / maxSaturation.getR() - minSaturation.getR());
    double greenSaturation = (pixel.getG() - minSaturation.getG()) *
            (1.0 / maxSaturation.getG() - minSaturation.getG());
    double blueSaturation = (pixel.getB() - minSaturation.getB()) *
            (1.0 / maxSaturation.getB() - minSaturation.getB());
    return new Pixel(redSaturation, greenSaturation, blueSaturation);
}
```

Figure 7: Lower abstraction shows more about implementation and presents equations from section 1.1, concerning histogram normalization.

Brightness Manipulation Implementation

```java
/**
 * Modifies saturation of each pixel's channels by {@link #brightness} from
 * the input image i.e. {@link #inputImage}.
 *
 * @return Image with modified brightness along all channels; {@link #outputImage}.
 */
@Override
public WritableImage apply() {
    PixelReader pixelReader = getInputImage().getPixelReader();
    PixelWriter pixelWriter = getOutputImage().getPixelWriter();

    for (int x = 0; x < getImageWidth(); x++) {
        for (int y = 0; y < getImageHeight(); y++) {
            Color inputColor = pixelReader.getColor(x, y);
            Pixel pixel = new Pixel(inputColor);
            pixel.addToAllChannelsAndClamp(brightness);
            Color outputColor = pixel.createColor();
            pixelWriter.setColor(x, y, outputColor);
        }
    }

    return getOutputImage();
}
```

```java
public void addValueToAllChannelsAndClamp(double value) {
    double modifiedRed = clampChannelSaturation(getR() + value);
    double modifiedGreen = clampChannelSaturation(getG() + value);
    double modifiedBlue = clampChannelSaturation(getB() + value);

    setR(modifiedRed);
    setG(modifiedGreen);
    setB(modifiedBlue);
}

private double clampChannelSaturation(double value) {
    return Math.min(MAXIMUM_CHANNEL_SATURATION, Math.max(value, MINIMUM_CHANNEL_SATURATION));
}
```

4 Results and Conclusions

4.1 Tresholding Results

In case of tresholding, different pictures acts differently. To start with, figure 8a acts greatly in case of iris extraction. As one can see image 8b presents extraction of desired parts (treshold level 8). I have also notice that when treshold level is bigger than 40 (figure 8c), image starts to have a lot of distortions in form of additional information about white of the eye.
On the other hand picture 8d had rather average results comparing to all others. Although flower object can be distinguished from rest of the surrounding (tresh-

old level 105) but there is always a significant amount of artifacts in background and elevating treshold (picture 8f; level over 180) reveals only worse outcomes - flower borders are starting to disappear.
Figure 8g behaves best in terms of tresholding. Distinguishing between a planet and USS Enterprise NCC-1701-D can be easily done at (treshold) level 32. Increasing treshold causes lost of information about ship along with background artifacts. I think that by finding good ratio between lost of the background noise and preserving Enterprise shape we will be able to still recreate the ship (e.g using dilatation filter) and clear the picture out of noise (to some reasonable level). Image 8i presents treshold equal to 103.
Last one - figure 8a, also reacts nicely to the tresholding process. At level 105 (8k) it almost perfectly distinguish between hair and face, but slight changes to the treshold immediately causes lost of coherence, as can be seen on 8l.

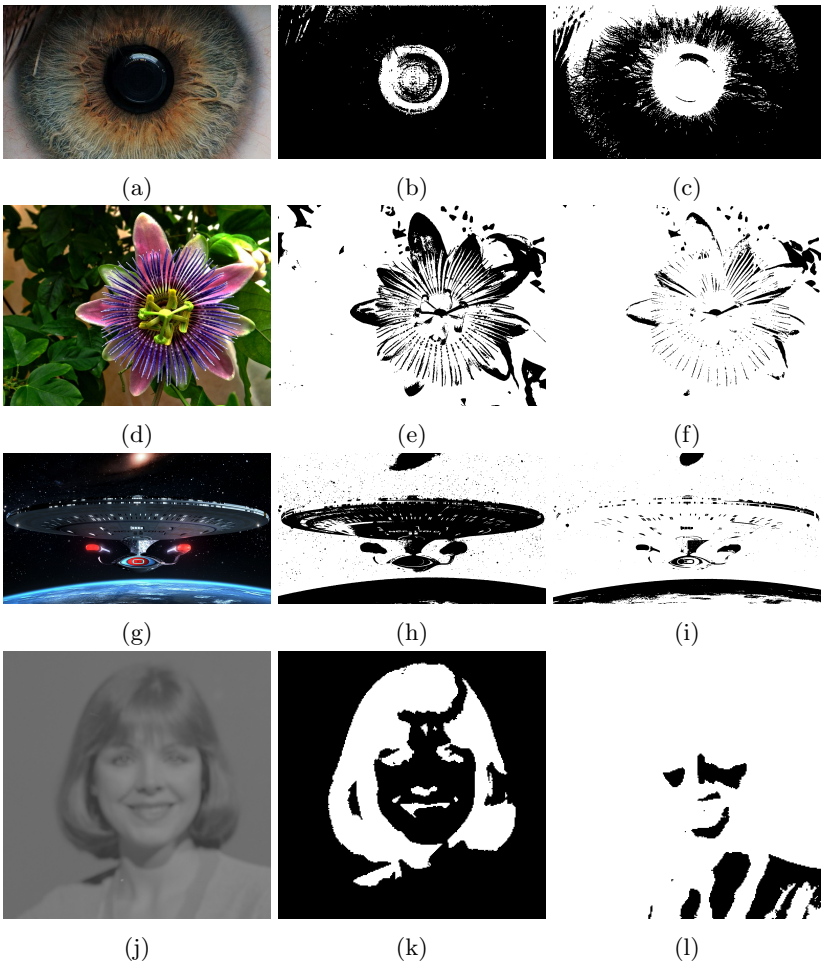


Figure 8: Results for tresholding using experimental data.

4.2 Histogram Normalization Results

Figure 9 represent results obtained by stretching histogram of test images. First column represents raw images (9a, 9d, 9g, 9j), next one corresponds to images with average approach (9b, 9e, 9h, 9k)and the last one (9c, 9f, 9i, 9l) per channel approach (as described in 1.1)
As one can easily see, the process does not affect most of the pictures. In higher resolution it can be see that colors of figure 9b are somehow more pleasant to the eye and quality of 9j has been drastically changed on 9k and 9l. On other two experimental data, no change has been observed.

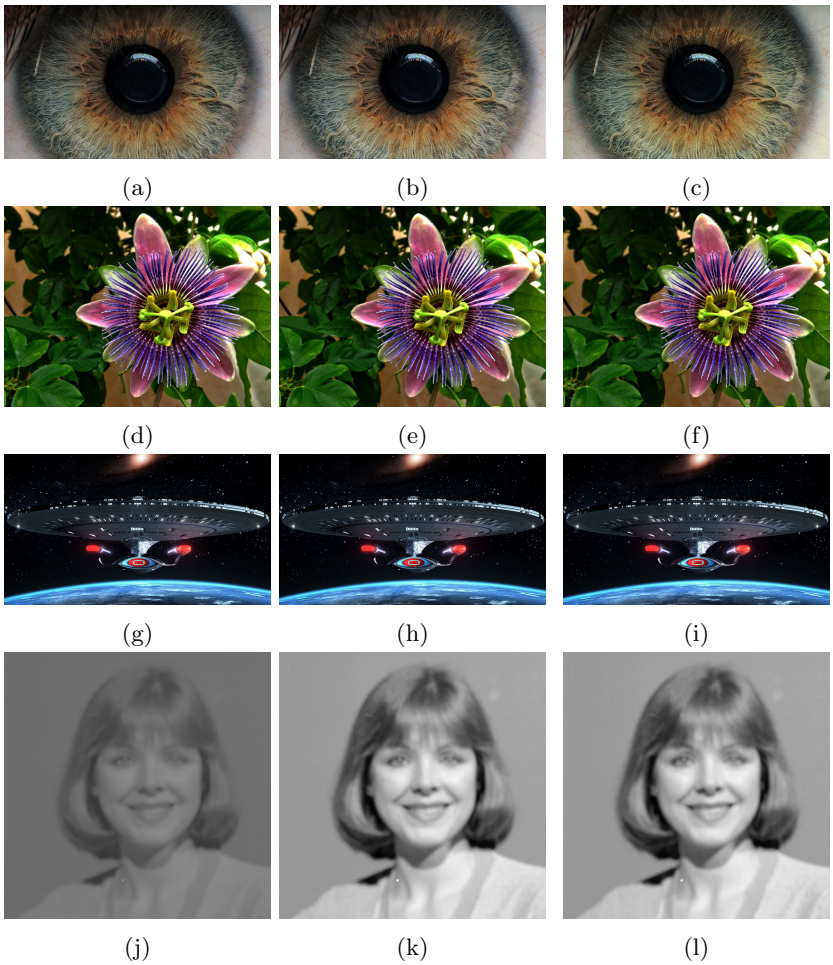


Figure 9: Results for histogram normalization of experimental data.

4.3 Brightness Manipulation Results

All pictures are acting rather the same - they have more and more light in process of adding brightness factor. All images were exposed to the same brightness level i.e 10b, 10e, 10h, 10k are at level 86 and 10c, 10f, 10i, 10l at level 186. From experimental data we see that only 10l is somehow all lost - it may be caused by the fact that it is in grayscale.

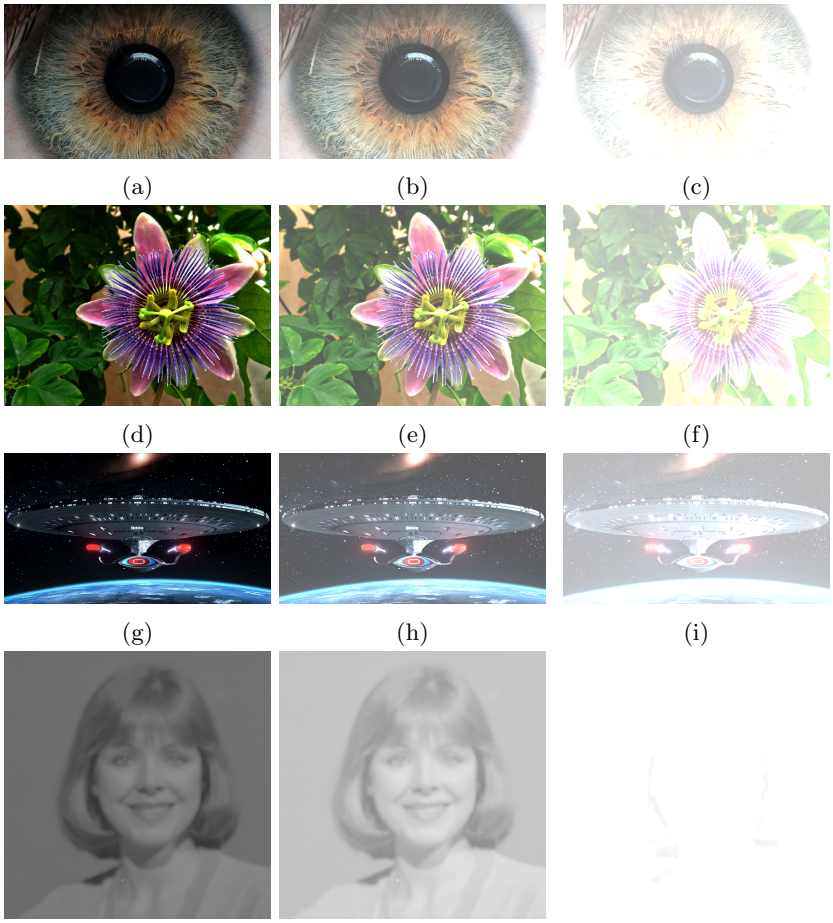


Figure 10: Results for brightness manipulation of experimental data.

To shortly sum up - all pictures were acting as assumed. Although histogram normalization required more work to implement, almost nothing can be seen on the experimental images. It may be caused by bad selection process - maybe 1b and 1c were already normalized? We will never know.

References

[1] https://www.wordnik.com/words/thresholding

[2] https://svi.nl/ImageHistogram

[3] https://en.wikipedia.org/wiki/Normalization_(image_processing)

[4] http://olympusmicro.com/primer/java/olympusmicd/digitalimaging/contrast/index.html