

Übungsaufgaben I, SBV1

Lisa Panholzer, Lukas Fiel

October 23, 2018

1 Gauss Filter

Es wurde ein Gauss Filter als ImageJ Filter implementiert. Die Behandlung der Randpixel wurde aus der Lehrveranstaltung übernommen. Gemeinsam mit dem Vortragenden Gerald Zwettler wurde die Java Klasse *Convolution-Filter* erweitert um auch die Randbereiche eines Bildes angemessen zu behandeln. In Heimarbeit wurde die Klasse um die Methode *GetGaussMask* erweitert. In dieser wird die Verteilung einer Gauss Kurve auf eine 2 dimensionale Maske übertragen.

Anschließend wurde eruiert welches Verhältnis von Sigma zum Radius der Maske eine klar zu erkennende Glocke darstellte. $\frac{2}{4}$ hat die gewünschte Eigenschaft.

Weiters wurde der Übergang von scharfen Kanten und Verläufen mit dem Gauss Filter gefiltert. Man bemerkt gut, dass bei einem Intensitätsverlauf kaum ein Filtereffekt sichtbar ist, während Kanten deutlich verschwommen erscheinen. Gewähltes Verhältnis: $\frac{\sigma}{radius} = \frac{2}{4}$ Interessant ist der Unterschied zum Median-Filter des nächsten Beispiels. Dieser stellt Kanten viel deutlicher dar und macht auch bei glatten Übergängen kaum einen bemerkenswerten Effekt.

1.0.1 Code

```
import ij.*;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;
import ij.gui.GenericDialog;

public class Gauss_ implements PlugInFilter {

    public int setup(String arg, ImagePlus imp) {
        if (arg.equals("about")) {
            showAbout();
            return DONE;
        }
        return DOES_8G + DOES_STACKS + SUPPORTS_MASKING
            ↪ ;
    } // setup
}
```

```

public void run(ImageProcessor ip) {
    int width = ip.getWidth();
    int height = ip.getHeight();
    int tgtRadius = getUserInput(4, "radius");
    int sigma = getUserInput(4, "sigma");

    double[][] resultImage = runFilter(ip,
        ↪ tgtRadius, sigma);

    ImageJUtility.showNewImage(resultImage, width,
        ↪ height, "mean_with_kernel_r=" + tgtRadius
        ↪ );

} // run

void showAbout() {
    IJ.showMessage("About_Template_...", "this_is_a
        ↪ PluginFilter_template\n");
} // showAbout

/**
 * Asks the user to input.
 *
 * @return value from user input. 0 if failed.
 */
public static int getUserInput(int defaultValue, String
    ↪ nameOfValue) {
    // user input
    System.out.print("Read_user_input: " +
        ↪ nameOfValue);
    GenericDialog gd = new GenericDialog("user_
        ↪ input:");
    gd.addNumericField("defaultValue", defaultValue
        ↪ , 0);
    gd.showDialog();
    if (gd.wasCanceled()) {
        return 0;
    }
}

```

```

    }
    int radius = (int) gd.getNextNumber();
    System.out.println(radius);
    return radius;
}

public static double[][] runFilter(ImageProcessor ip,
    ↪ int radius, int sigma) {
    // convert to pixel array
    byte[] pixels = (byte[]) ip.getPixels();
    int width = ip.getWidth();
    int height = ip.getHeight();
    int tgtRadius = radius;
    int size = 2 * radius + 1;

    int[][] inArr = ImageJUtility.
        ↪ convertFrom1DByteArr(pixels, width,
        ↪ height);
    double[][] inDataArrDouble = ImageJUtility.
        ↪ convertToDoubleArr2D(inArr, width, height
        ↪ );

    double[][] filterMask = ConvolutionFilter.
        ↪ GetGaussMask(tgtRadius, sigma);
    int[][] filterMaskInt = convert2Int(filterMask)
        ↪ ;
    ImageJUtility.showNewImage(filterMaskInt, size,
        ↪ size, "GaussMask");

    return ConvolutionFilter.ConvolveDoubleNorm(
        ↪ inDataArrDouble, width, height,
        ↪ filterMask, tgtRadius);
}

public static int[][] convert2Int(double[][] inMask) {
    double[][] tmpMask = inMask.clone();

```

```

        int size = inMask.length;
        int[][] maskInt = new int[size][size];
        int maxInt = 255;

        // get maximum
        double maxDouble = 0;
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                if (maxDouble < tmpMask[i][j] ) {
                    ↪ maxDouble = tmpMask[i][j]
                    ↪ ];}
            }
        }

        // scale mask
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {

                maskInt[i][j] = (int) (tmpMask[i
                    ↪ ][j] / maxDouble * maxInt);
            }
        }

        return maskInt;

    }

} // class FilterTemplate_

```

```

public class ConvolutionFilter {

    public static double[][] ConvolveDoubleNorm(double[][]
        ↪ inputImg, int width, int height, double[][]
        ↪ kernel, int radius, int numOfIterations) {
        double[][] returnImg = inputImg;
    }
}

```

```

    for(int iterCount = 0; iterCount < numOfIterations;
        ↪ iterCount++) {
        returnImg = ConvolutionFilter.
            ↪ ConvolveDoubleNorm(returnImg, width,
            ↪ height, kernel, radius);
    }

    return returnImg;
}

public static double[][] ConvolveDoubleNorm(double[][]
    ↪ inputImg, int width, int height, double[][]
    ↪ kernel, int radius) {
    double[][] returnImg = new double[width][height
        ↪ ];

    //step1: move mask to all possible image pixel
        ↪ positions
    for( int x = 0; x < width; x++) {
        for( int y = 0; y < height; y++) {

            double totalSum = 0.0;
            double maskCount = 0.0;
            //step2: interate over all mask
                ↪ elements
            for(int xOffset = -radius;
                ↪ xOffset <= radius ; xOffset
                ↪ ++) {
                for(int yOffset = -radius
                    ↪ ; yOffset <= radius
                    ↪ ; yOffset++) {
                    int nbX = x +
                        ↪ xOffset;
                    int nbY = y +
                        ↪ yOffset;

                    // step3: check

```

```

    ↪ range of
    ↪ coordinates
    ↪ in
    ↪ convolution
    ↪ mask
if(nbX >= 0 && nbX
    ↪ < width &&
    ↪ nbY >= 0 &&
    ↪ nbY < height
    ↪ ) {
        totalSum +=
            ↪
            ↪ inputImg
            ↪ [nbX
            ↪ ][nbY
            ↪ ] *
            ↪ kernel
            ↪ [
            ↪ xOffset
            ↪ +
            ↪ radius
            ↪ ][
            ↪ yOffset
            ↪ +
            ↪ radius
            ↪ ];
        maskCount
            ↪ +=
            ↪ kernel
            ↪ [
            ↪ xOffset
            ↪ +
            ↪ radius
            ↪ ][
            ↪ yOffset
            ↪ +
            ↪ radius
            ↪ ];

```

```

    }

    }

    }
    //step3.5 normalize
    totalSum /= maskCount;

    //step4: store result in output
    ↪ image
    returnImg[x][y] = totalSum;
    } // y loop
} // x loop

return returnImg;
}

public static double[][] ConvolveDouble(double[][]
    ↪ inputImg, int width, int height, double[][]
    ↪ kernel, int radius) {
    double[][] returnImg = new double[width][height
    ↪ ];

    //step1: move mask to all possible image pixel
    ↪ positions
    for( int x = 0; x < width; x++) {
        for( int y = 0; y < height; y++) {

            double totalSum = 0.0;
            //step2: interate over all mask
            ↪ elements
            for(int xOffset = -radius;
                ↪ xOffset <= radius ; xOffset
                ↪ ++ ) {
                for(int yOffset = -radius
                    ↪ ; yOffset <= radius
                    ↪ ; yOffset++) {
                    int nbX = x +
                        ↪ xOffset;

```



```

int nbY = y +
    ↪ yOffset;

// step3: check
    ↪ range of
    ↪ coordinates
    ↪ in
    ↪ convolution
    ↪ mask
if(nbX >= 0 && nbX
    ↪ < width &&
    ↪ nbY >= 0 &&
    ↪ nbY < height
    ↪ ) {
    totalSum +=
        ↪
        ↪ inputImg
        ↪ [nbX
        ↪ ][nbY
        ↪ ] *
        ↪ kernel
        ↪ [
        ↪ xOffset
        ↪ +
        ↪ radius
        ↪ ][
        ↪ yOffset
        ↪ +
        ↪ radius
        ↪ ];
}

}

//step4: store result in output
    ↪ image

```

```

        returnImg[x][y] = totalSum;
    } // y loop
} // x loop

return returnImg;
} // ConvolveDouble end

public static double[][] GetMeanMask(int tgtRadius) {
    int size = 2 * tgtRadius + 1;

    int numOfElements = size * size;
    double maskVal = 1.0 / numOfElements;
    double[][] kernelImg = new double[size][size];

    for(int i = 0; i < size; i++) {
        for(int j = 0; j < size; j++) {
            kernelImg[i][j] = maskVal;
        }
    }

    return kernelImg;
}

public static double[][] GetGaussMask(int tgtRadius,
    ↪ double sigma) {
    int size = 2 * tgtRadius + 1;

    double constant = 1 / (Math.PI * 2 * sigma * sigma);

    double[][] kernelImg = new double[size][size];

    for(int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            double diffI = i - size/2;
            double diffJ = j - size/2;

            kernelImg[i][j] = constant * Math
                ↪ .exp(-(diffI*diffI + diffJ

```

```

        ↪ *diffJ ) / (2*sigma*sigma))
        ↪ ;
    }
}

return kernelImg;
}

```

```

public static double[][] ApplySobelEdgeDetection(double
    ↪ [][] inputImg, int width, int height) {
    double[][] returnImg = new double[width][height];
    double[][] sobelV = new double[][]{{1.0, 0.0, -1.0},
    ↪ {2.0, 0.0, -2.0}, {1.0, 0.0, -1.0}};
    double[][] sobelH = new double[][]{{1.0, 2.0,
    ↪ 1.0}, {0.0, 0.0, 0.0}, {-1.0, -2.0,
    ↪ -1.0}};

    int radius = 1;
    double maxGradient = 1.0;

    // achtung! hier keine Normierung
    double[][] resultSobelV = ConvolveDouble(
    ↪ inputImg,width,height,sobelV,radius);
    double[][] resultSobelH = ConvolveDouble(
    ↪ inputImg,width,height,sobelH,radius);

    for( int x = 0; x < width; x++) {
        for( int y = 0; y < height; y++) {
            double vAbs = Math.abs(
            ↪ resultSobelV[x][y]);
            double hAbs = Math.abs(
            ↪ resultSobelH[x][y]);
            double resVal = vAbs + hAbs;
            returnImg[x][y] = resVal;

            // new max gradient?

```

```

        if(resVal >maxGradient)
            ↪ maxGradient = resVal;
    }
}

//finally normalize by max gradient value
double corrFactor = maxGradient/255.0;

for(int x = 0; x < width; x++) {
    for ( int y = 0; y < height; y++) {
        returnImg[x][y] /= corrFactor;
    }
}

return returnImg;
}
}

```

2 MedianFilter

Der MedianFilter kann leider nicht mittels der Klasse *ConvolutionFilter* implementiert werden, da die Maske für dieses Vorgehen konstant sein müsste. Das Prinzip ist allerdings sehr ähnlich. Es wird ein Pixel in Mitten einer quadratischen Umgebung betrachtet. Dieses Pixel soll im resultierenden Bild als der Median Wert der Umgebung gesetzt werden.

Implementiert wurde dies durch das Herausschneiden der interessanten Umgebung aus einer Kopie des Ursprungsbildes und anschließender Medianwertberechnung.

2.0.1 Code

```
import ij.*;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;
import ij.gui.GenericDialog;
import java.awt.Rectangle;
import java.util.Arrays;

import com.sun.net.httpserver.Authenticator.Success;

public class Median_ implements PlugInFilter {

    public int setup(String arg, ImagePlus imp) {
        if (arg.equals("about")) {
            showAbout();
            return DONE;
        }
        return DOES_8G + DOES_STACKS + SUPPORTS_MASKING
            ↪ ;
    } // setup

    public void run(ImageProcessor ip) {

        System.out.println("RUN: □Plugin□Median");
        int width = ip.getWidth();
        int height = ip.getHeight();
```

```

int radius = getUserInputRadius(4);
// int radius = 2; // default value for
    ↪ debugging

if (2 * radius > width || 2 * radius > height)
    ↪ {
        System.out.println("Be aware that double
            ↪ the radius has to fit in the
            ↪ image!");
    }

double[] [] resultImage = runFilter(ip, radius);

System.out.println("Now show the result image!"
    ↪ );
ImageJUtility.showNewImage(resultImage, width,
    ↪ height, "mean with kernel r=" + radius);
System.out.println("SUCCESS: MEDIAN FILTER DONE
    ↪ .");

} // run

public static double[] [] runFilter(ImageProcessor ip,
    ↪ int radius) {
    byte[] pixels = (byte[]) ip.getPixels();
    int width = ip.getWidth();
    int height = ip.getHeight();

    int[] [] inArr = ImageJUtility.
        ↪ convertFrom1DByteArr(pixels, width,
        ↪ height);
    double[] [] inDataArrDouble = ImageJUtility.
        ↪ convertToDoubleArr2D(inArr, width, height
        ↪ );

```

```

double[][] resultImage = inDataArrDouble.clone
    ↪ ();
int successIndex = 0;
int failureIndex = 0;
// step1: move mask to all possible image
    ↪ pixel positions
for (int x = 0; x < width; x++) {
    for (int y = 0; y < height; y++) {
        double[][] mask = inDataArrDouble
            ↪ .clone();
        try {

            // roi = new Rectangle(x
                ↪ - radius, y -
                ↪ radius, size -
                ↪ deltaX - 1, size);
            Rectangle roi = getROI(
                ↪ width, height, x, y,
                ↪ radius);
            mask = ImageJUtility.
                ↪ cropImage(mask, roi.
                ↪ width, roi.height,
                ↪ roi);
            double median = getMedian(
                ↪ mask,roi.width,roi.
                ↪ height);
            resultImage[x][y] = median
                ↪ ;

            successIndex++;
        } catch (java.lang.
            ↪ ArrayIndexOutOfBoundsException
            ↪ exc) {
            // TODO: error handling
                ↪ for edge cases

            resultImage[x][y] =
                ↪ resultImage[x][y];

```

```

failureIndex++;

    }

    }

    }
    // System.out.println("SUCCESS: run over
        ↪ picture. succeed: " + successIndex + ",
        ↪ failed: " + failureIndex
    // + ", sum: " + (int) (successIndex +
        ↪ failureIndex));
    return resultImage;
}

void showAbout() {
    IJ.showMessage("About_Template_...", "this_is_a
        ↪ PluginFilter_template\n");
} // showAbout

/**
 * get region of interest. defined by a Rectangle with
 * ↪ x and y coordinates of the
 * upper left corner and width and height as parameters
 * ↪ .
 *
 * @param width of the image
 * @param height of the image
 * @param x the x coordinate of the center of the mask
 * @param y the y coordinate of the center of the mask
 * @param radius of the mask
 * @return
 */
public static Rectangle getROI(int width, int height,
    ↪ int x, int y, int radius) {
    int xsize = 2 * radius + 1;
    int ysize = 2 * radius + 1;

```



```

        // special behaviour
        if (x - radius < 0) {
            xsize = xsize - (radius - x);
            x = radius;
        } // set minimum x
        if (y - radius < 0) {
            ysize = ysize - (radius - y);
            y = radius;
        } // set minimum y

        if (x + radius >= width) {
            int d = (radius - (width - x));
            xsize = xsize - d - 1 ;
        } // set maximum x
        if (y + radius >= height) {
            int d = (radius - (height - y));
            ysize = ysize - d - 1 ;
        } // set maximum y

        return new Rectangle(x - radius, y - radius,
            ↪ xsize, ysize);
    }

    public static double getMedian(double[][] inputImg, int
        ↪ width, int height) {
        int size = width * height;

        // fill array
        double[] arr = new double[size];
        int index = 0;
        for (int i = 0; i < width; i++) {
            for (int j = 0; j < height; j++) {
                arr[index] = inputImg[i][j];
                index++;
            }
        }
    }

```

```

        // sort array
        Arrays.sort(arr);
        // System.out.println("SUCCESS: getMedian.
        ↪ size: " + size);
        return arr[(int) (size / 2 + 1)];
    }

    /**
     * Asks the user to input a radius.
     *
     * @return radius from user input. 0 if failed.
     */
    public static int getUserInputRadius(int defaultValue)
        ↪ {
        // user input
        System.out.println("Read user input: radius");
        GenericDialog gd = new GenericDialog("user
        ↪ input:");
        gd.addNumericField("radius", defaultValue, 0);
        gd.showDialog();
        if (gd.wasCanceled()) {
            return 0;
        }
        return (int) gd.getNextNumber();
    }

} // class FilterTemplate_

```

2.0.2 resultierendes Bild



3 Steuerung des Filtereffekts

3.1 Code

```
import ij.*;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;
import ij.gui.GenericDialog;

public class FiltereffektEvaluierung_ implements PlugInFilter
↳ {

    public int setup(String arg, ImagePlus imp) {
        if (arg.equals("about")) {
            showAbout();
            return DONE;
        }
        return DOES_8G + DOES_STACKS + SUPPORTS_MASKING
↳ ;
    } // setup
```

```

public void run(ImageProcessor ip) {

    System.out.println("RUN: Time Evaluation");
    // convert to pixel array
    int width = ip.getWidth();
    int height = ip.getHeight();
    int tgtRadius = 4; // default value
    int sigma = 4;

    double[][] resultImage = new double[width][
        ↪ height];
    int [] iterations = {1,2,3,4,5};

    System.out.println("Please Input the radius of
        ↪ the mask for all the filters.");
    tgtRadius = getUserInput(tgtRadius, "radius");
    System.out.println("Please type a proper sigma
        ↪ value.");
    sigma = getUserInput(sigma, "sigma");

    // ----- MEAN -----
    long startTime = System.nanoTime();
    for (int j = 0; j < iterations.length; j++) {
        System.out.println("Run Mean Filter" +
            ↪ iterations[j] + " times.");
        startTime = System.nanoTime();
        for (int i = 0; i < iterations[j]; i++)
            ↪ {
                resultImage = Mean_.runFilter(ip,
                    ↪ tgtRadius); // for time
                    ↪ measurement the input
                    ↪ image is not important
            }
        System.out.println("Took:" + (System.
            ↪ nanoTime() - startTime) + "
            ↪ nanoseconds.");
    }
}

```

```

// ----- GAUSS -----

for (int j = 0; j < iterations.length; j++) {
    System.out.println("Run_Gauss_Filter_" +
        ↪ iterations[j] + "_times.");
    startTime = System.nanoTime();
    for (int i = 0; i < iterations[j]; i++)
        ↪ {
            resultImage = Gauss_.runFilter(ip
                ↪ , tgtRadius, sigma); // for
                ↪ time measurement the
                ↪ input image is not
                ↪ important
        }
    System.out.println("Took:" + (System.
        ↪ nanoTime() - startTime) + "_"
        ↪ nanoseconds.");
}

// ----- MEDIAN -----

for (int j = 0; j < iterations.length; j++) {
    System.out.println("Run_Median_Filter_"
        ↪ + iterations[j] + "_times.");
    startTime = System.nanoTime();
    for (int i = 0; i < iterations[j]; i++)
        ↪ {
            resultImage = Median_.runFilter(
                ↪ ip, tgtRadius); // for time
                ↪ measurement the input
                ↪ image is not important
        }
    System.out.println("Took:" + (System.
        ↪ nanoTime() - startTime) + "_"
        ↪ nanoseconds.");
}

//ImageJUtility.showNewImage(resultImage,
    ↪ width, height, "mean with kernel");

```

```

        System.out.println("SUCCESS:␣␣Time␣Evaluation:␣
        ↪ DONE.");

    } // run

    void showAbout() {
        IJ.showMessage("About␣Template_...", "this␣is␣a
        ↪ ␣PluginFilter␣template\n");
    } // showAbout

    /**
     * Asks the user to input.
     *
     * @return value from user input. 0 if failed.
     */
    public static int getUserInput(int defaultValue, String
    ↪ nameOfValue) {
        // user input
        System.out.print("Read␣user␣input:␣" +
        ↪ nameOfValue);
        GenericDialog gd = new GenericDialog("user␣
        ↪ input:");
        gd.addNumericField("defaultValue", defaultValue
        ↪ , 0);
        gd.showDialog();
        if (gd.wasCanceled()) {
            return 0;
        }
        int radius = (int) gd.getNextNumber();
        System.out.println(radius);
        return radius;
    }

} // class FilterTemplate_

```

3.2 Idee

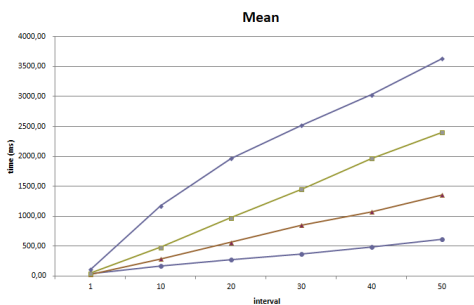
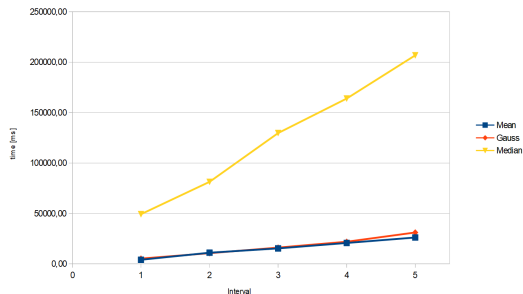
Das selbstgeschriebene Plugin *FiltereffektEvaluierung_* wurde geschrieben um die Laufzeiten der einzelnen Filter zu erfassen. Hierbei wurde darauf geachtet, dass mittels der Methode *System.nanoTime()* im Gegensatz zu *System.millis()* eine genauere Zeitmessung möglich ist. Als eine sehr große Maske wurde $radius = 40$ gewählt. Das Setzen der Größe *sigma* ist bei der Messung der Laufzeit irrelevant, da sie nur für das Erstellen der Maske ausschlaggebend ist. Dass die Berechnung von *sigma* eventuell mehr Zeit beansprucht als die Berechnungen die zum Erstellen einer Mean-Maske nötig ist, ist auch nicht von großer Bedeutung. Dies kann einfach durch den Fakt erklärt werden, dass die Maske immer nur ein mal vor dem Anwenden des Filters erstellt wird.

Beim Median-Filter wird gedanklich auch eine Maske über das Bild bewegt. In diesem Fall ist die Maske aber nicht im vorhinein berechnet worden, sondern grenzt lediglich den Bereich ein, in dem der Median-Wert berechnet werden soll.

3D Masken könnten neben dem Filtereffekt zum Beispiel bei einem RGB Bild auch Farbtöne verstärken/abschwächen. Dabei müssen aber im Falle des Median-Filters für jeden Pixel des dreidimensionalen Bildes alle 3 Dimensionen der Maske berechnet werden. Bei Mean- und Gauss-Filter hingegen reicht es einen zweidimensionalen Filter auf jede Ebene der dritten Dimension anzuwenden und die Ergebnisse weiterzuverarbeiten.

t_{gP}...time to get pixel value. Die Zeit einen Pixel aus der Berechnung mit einer 2D Maske
t_{cM}...time to calculate initial Mask
t_{cMM}...time to calculate Median Mask
d...Anzahl der Dimensionen

3.2.1 Tests



4 Histogrammeinebnung

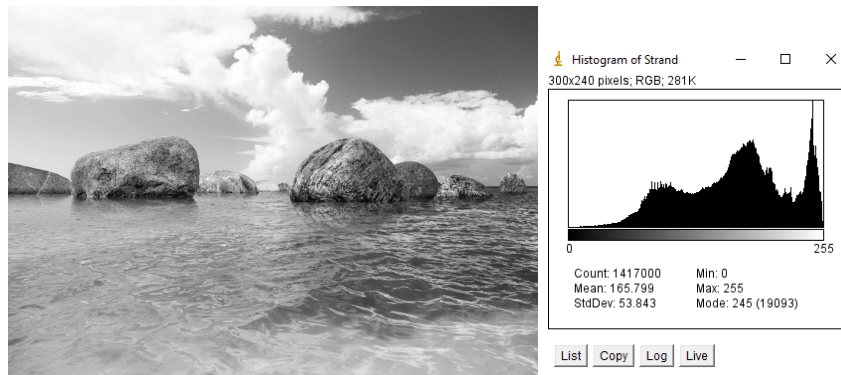
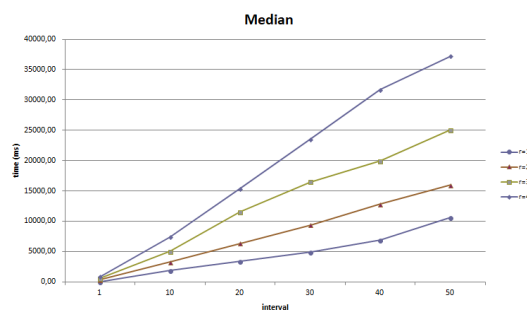
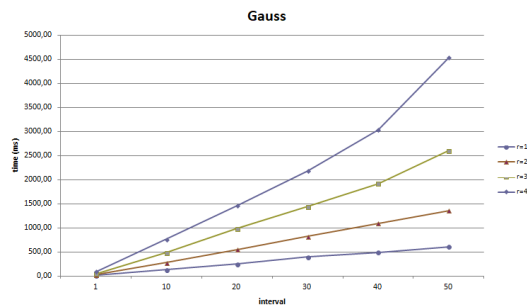
4.0.1 a) Ablauf und Lösungsidee

Die Histogrammeinebnung wurde mit der, wie im Foliensatz beschriebenen, Formel durchgeführt. Die genaue Implementierung findet sich im Bereich "Code".

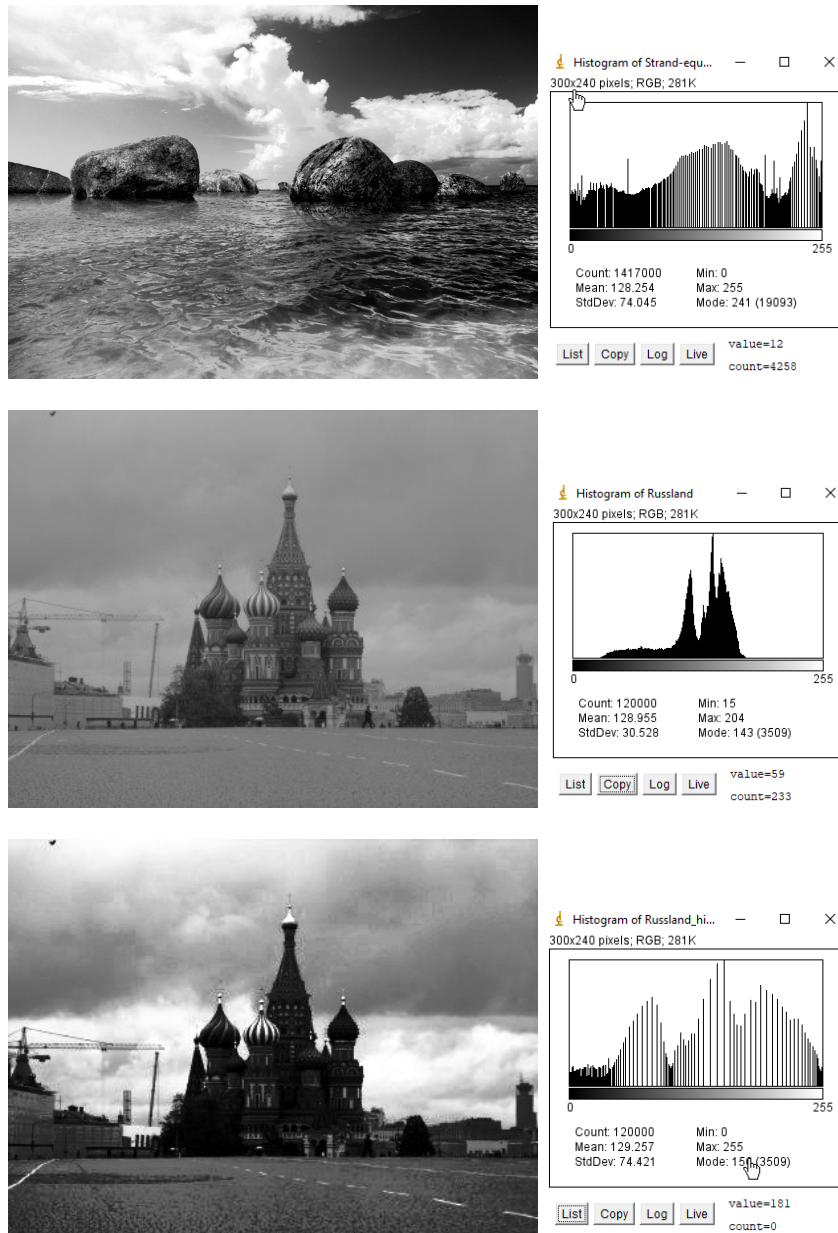
4.0.2 a) Tests

Die Implementierung wurde anhand der folgenden drei Bilder getestet:

1. Strand: Das Bild "Strand" enthält eine halbwegs gleichmäßige Verteilung der Grautöne. Nach Anwendung der Histogrammeinebnung kann man eine Verstärkung des Kontrastes erkennen.
2. Russland: Das Bild "Russland" enthält sehr wenig Kontrast und viele der Grautöne sind im Histogramm benachbart. Nach der Anwendung der Histogrammeinebnung verstärkt sich der Kontrast um ein vielfaches und selbst die Wokenformation sind nun detailliert sichtbar.

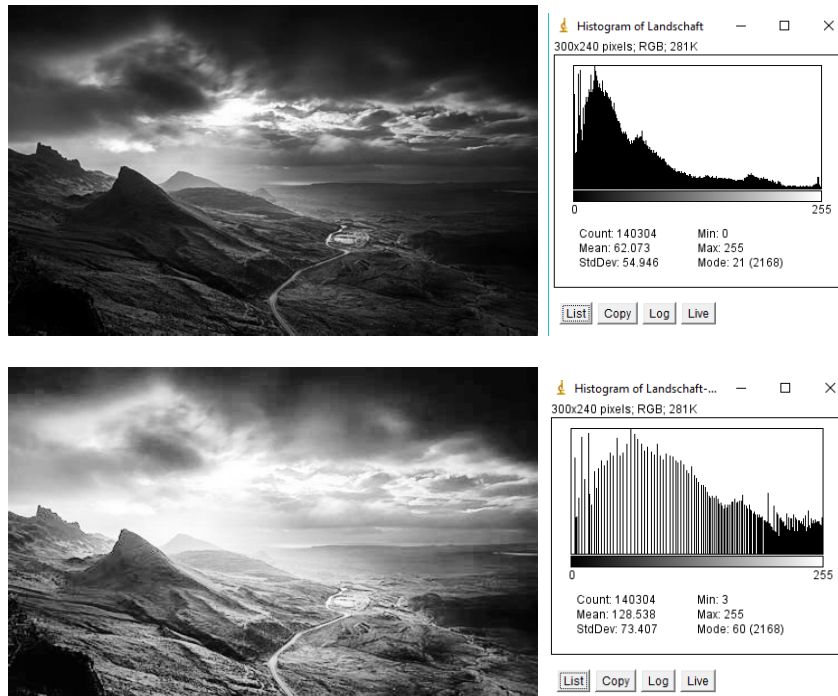


3. Landschaft: Das Bild "Landschaft" hat im Vergleich zu den ersten beiden Testbildern einen höheren Kontrast und ist dunkler. Bei der Anwendung der Histogrammeinebnung kann man nun beobachten, dass aufgrund der Gleichverteilung der Grautöne Richtung den Maximalwert (255), das Bild sich aufhellt und der Kontrast erhalten bleibt.



4.1 b) Diskussion Histogrammeinebnung

Es kann zu einer Verschlechterung der Bildqualität kommen, wenn die Histogrammeinebnung zum Beispiel auf ein stark überbelichtetes Bild angewandt wird. Bei folgender Aufnahme einer Straße mit Fußgängern sieht



man, dass der Großteil der Pixelintensitäten ca. über 200 liegt. Findet nun die Einebnung statt werden die Verläufe nicht mehr weich dargestellt. Die Unterschiede zwischen den Pixelintensitäten werden zu hoch und Teile des Bildes deshalb kantiger dargestellt.





4.2 Code

```
import ij.*;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;

public class HistogrammEqualization_BL_ implements
    ↪ PlugInFilter {

    public int setup(String arg, ImagePlus imp) {
        if (arg.equals("about")) {
            showAbout();
            return DONE;
        }
        return DOES_8G + DOES_STACKS + SUPPORTS_MASKING
            ↪ ;
    } // setup

    public void run(ImageProcessor ip) {
        byte[] pixels = (byte[]) ip.getPixels();
        int width = ip.getWidth();
        int height = ip.getHeight();

        final int MAXVAL = 255;
```

```

int[] [] inDataArrInt = ImageJUtility.
    ↪ convertFrom1DByteArr(pixels, width,
    ↪ height);

int[] tf2 = GetHistogramEqualizationTF2(MAXVAL,
    ↪ inDataArrInt, width, height);

int[] [] outDataArrInt2 =
    ↪ ImageTransformationFilter.
    ↪ GetTransformedImage(inDataArrInt, width,
    ↪ height, tf2);

ImageJUtility.showNewImage(outDataArrInt2,
    ↪ width, height, "Equalized_Image");

} // run

void showAbout() {
    IJ.showMessage("About_Template_...", "this_is_a
    ↪ PluginFilter_template\n");
} // showAbout

public static int[] GetHistogramEqualizationTF2(int
    ↪ maxValue, int[] [] inputImage, int width, int
    ↪ height) {

    int maxValueTF = maxValue - 0 + 1;
    int pixelCount = width * height;
    double probabilitySum = 0;

    int[] histogram = getHisto(inputImage, width,
    ↪ height, maxValue);
    int[] transferFunction = new int[maxValue + 1];

    for (int i = 0; i < histogram.length; i++) {
        probabilitySum+=((double)histogram[i])/
        ↪ pixelCount;
    }
}

```

```

        double tmpSum = probabilitySum *
            ↪ maxValueTF + 0;
        transferFunction[i] = (int)(Math.floor(
            ↪ tmpSum));

        if(transferFunction[i]>maxValue) {
            transferFunction[i] = maxValue;
        }
    }

    return transferFunction;

}

public static int[] getHisto(int[][] inImg, int width,
    ↪ int height, int maxValue) {
    int[] histogram = new int[maxValue + 1];
    // step1: get histogram
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            histogram[inImg[x][y]]++;
        }
    }
    return histogram;
}
} // class FilterTemplate_

```

5 Raster-Entfernung im Frequenzraum

5.1 Workflow

- Starten von *imageJ.exe*
- Öffnen eines Bildes
- $Process \rightarrow FFT \rightarrow FFT$
- Zuschneiden des interessanten Bereichs im FFT Bild
- $Process \rightarrow FFT \rightarrow inverse\ FFT$

5.2 Beispiele

5.2.1 Auge

Es wurde ein Bild gewählt, welches (wie bei einem Plakatdruck) Punkte in regelmässigen Abständen aufweist. Die eigentliche Bildinformation steckt in der Dicke der Punkte. Eine FFT Transformation zeigt deutlich ein periodisches Muster. Will man nur die eigentliche Bildinformation gewinnen, müssen hochfrequente Anteile des Bildes entfernt werden. Tabelle 4 zeigt deutlich dass durch ein Entfernen der Randbereiche (höhere Frequenzen) im FFT Bild und die anschließende Rücktransformation die eigentliche Bildinformation gewonnen werden konnte.

5.2.2 Elefant

In diesem Bild sind viele periodisch auftretende Elemente enthalten. Es wurde versucht die Schrift, die Gitterstäbe im Hintergrund und natürlich die beiden Tiere gut sichtbar zu erhalten. Da aber die Gitterstäbe selbst periodisch im Bild vorkommen und auch die Schrift sich wiederholende senkrechte Kanten hat, war dies nicht einfach. Ein Auslöschen der horizontalen und vertikalen Anteile aus dem Bild brachte in unseren Versuchen das beste Ergebnis. Hierbei ist aber zu beachten, dass das Zentrum des FFT Bildes die meiste Information enthält. Daher wurde diese bestehen gelassen. Auch die Randbereiche der FFT wurden belassen, da diese für scharfe Kanten im Bild verantwortlich sind. Ein Wegschneiden dieser Bereiche würde auch die Konturen des Elefanten und die Schrift unscharf machen.

5.2.3 Lochgitter

Hier handelt es sich um ein perspektivisch beläuchtetes Lochgitter. Die Löcher sind sechseckig. In der FFT erkennt man gut die Periodizität. Ein Wegschneiden der äusseren Bereiche der FFT und eine Rücktransformation zeigt deutlich die perspektivische Beläuchtung. Das Lochgitter konnte aber vollkommen entfernt werden. Interessant ist auch zu bemerken, dass im Rücktransformierten Bild eine Schrift "*colourbox*" deutlich zu erkennen ist. Bei genauerer Betrachtung des Ursprungsbildes ist diese hinter dem Gitter zu erkennen.

5.3 Analyse eines Frequenzmusters

Ein sich wiederholendes Muster in einem Bild ist mittels *FFT* gut vom eigentlichen Bildinhalt zu unterscheiden. So kann das Muster entfernt werden und das eigentliche Bild mittels *inverseFFT* ermittelt werden. Leider sind reale Bilder meist nicht genau horizontal ausgerichtet. Auch kann man nicht davon ausgehen, dass sich wiederholende Elemente in der Realität unverzerrt in einem Bild dargestellt sind. Kanten werden nur in den seltensten Fällen genau durch einen Pixel des Bildes dargestellt. All diese Umstände machen es schwer aus einem Alltagsfoto wiederkehrende Elemente herauszufiltern.

6 Anhang

```
public class ConvolutionFilter {

    public static double[][] ConvolveDoubleNorm(double[][]
        ↪ inputImg, int width, int height, double[][]
        ↪ kernel, int radius, int numOfIterations) {
        double[][] returnImg = inputImg;
        for(int iterCount = 0; iterCount < numOfIterations;
            ↪ iterCount++) {
            returnImg = ConvolutionFilter.
                ↪ ConvolveDoubleNorm(returnImg, width,
                ↪ height, kernel, radius);
        }
    }
}
```



```

    return returnImg;
}

public static double[] [] ConvolveDoubleNorm(double[] []
    ↪ inputImg, int width, int height, double[] []
    ↪ kernel, int radius) {
    double[] [] returnImg = new double[width][height
    ↪ ];

    //step1: move mask to all possible image pixel
    ↪ positions
    for( int x = 0; x < width; x++) {
        for( int y = 0; y < height; y++) {

            double totalSum = 0.0;
            double maskCount = 0.0;
            //step2: interate over all mask
            ↪ elements
            for(int xOffset = -radius;
            ↪ xOffset <= radius ; xOffset
            ↪ ++) {
                for(int yOffset = -radius
                ↪ ; yOffset <= radius
                ↪ ; yOffset++) {
                    int nbX = x +
                    ↪ xOffset;
                    int nbY = y +
                    ↪ yOffset;

                    // step3: check
                    ↪ range of
                    ↪ coordinates
                    ↪ in
                    ↪ convolution
                    ↪ mask

```

```

if(nbX >= 0 && nbX
    ↪ < width &&
    ↪ nbY >= 0 &&
    ↪ nbY < height
    ↪ ) {
    totalSum +=
        ↪
        ↪ inputImg
        ↪ [nbX
        ↪ ][nbY
        ↪ ] *
        ↪ kernel
        ↪ [
        ↪ xOffset
        ↪ +
        ↪ radius
        ↪ ][
        ↪ yOffset
        ↪ +
        ↪ radius
        ↪ ];
    maskCount
        ↪ +=
        ↪ kernel
        ↪ [
        ↪ xOffset
        ↪ +
        ↪ radius
        ↪ ][
        ↪ yOffset
        ↪ +
        ↪ radius
        ↪ ];
    }
}
//step3.5 normalize

```

```

        totalSum /= maskCount;

        //step4: store result in output
        ↪ image
        returnImg[x][y] = totalSum;
    } // y loop
} // x loop

return returnImg;
}

public static double[][] ConvolveDouble(double[][]
    ↪ inputImg, int width, int height, double[][]
    ↪ kernel, int radius) {
    double[][] returnImg = new double[width][height
        ↪ ];

    //step1: move mask to all possible image pixel
    ↪ positions
    for( int x = 0; x < width; x++) {
        for( int y = 0; y < height; y++) {

            double totalSum = 0.0;
            //step2: interate over all mask
            ↪ elements
            for(int xOffset = -radius;
                ↪ xOffset <= radius ; xOffset
                ↪ ++) {
                for(int yOffset = -radius
                    ↪ ; yOffset <= radius
                    ↪ ; yOffset++) {
                    int nbX = x +
                        ↪ xOffset;
                    int nbY = y +
                        ↪ yOffset;

```

```

// step3: check
    ↪ range of
    ↪ coordinates
    ↪ in
    ↪ convolution
    ↪ mask
if(nbX >= 0 && nbX
    ↪ < width &&
    ↪ nbY >= 0 &&
    ↪ nbY < height
    ↪ ) {
    totalSum +=
        ↪
        ↪ inputImg
        ↪ [nbX
        ↪ ][nbY
        ↪ ] *
        ↪ kernel
        ↪ [
        ↪ xOffset
        ↪ +
        ↪ radius
        ↪ ][
        ↪ yOffset
        ↪ +
        ↪ radius
        ↪ ];
    }

    }

    }

//step4: store result in output
    ↪ image
    returnImg[x][y] = totalSum;
    } // y loop
} // x loop

```

```

        return returnImg;
    } // ConvolveDouble end

    public static double[][] GetMeanMask(int tgtRadius) {
        int size = 2 * tgtRadius + 1;

        int numOfElements = size * size;
        double maskVal = 1.0 / numOfElements;
        double[][] kernelImg = new double[size][size];

        for(int i = 0; i < size; i++) {
            for(int j = 0; j < size; j++) {
                kernelImg[i][j] = maskVal;
            }
        }

        return kernelImg;
    }

    public static double[][] GetGaussMask(int tgtRadius,
        ↪ double sigma) {
        int size = 2 * tgtRadius + 1;

        double constant = 1 / (Math.PI * 2 * sigma * sigma);

        double[][] kernelImg = new double[size][size];

        for(int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                double diffI = i - size/2;
                double diffJ = j - size/2;

                kernelImg[i][j] = constant * Math
                    ↪ .exp(-( diffI*diffI + diffJ
                    ↪ *diffJ ) / (2*sigma*sigma))
                    ↪ ;
            }
        }
    }

```

```

        return kernelImg;
    }

public static double[][] ApplySobelEdgeDetection(double
    ↪ [] [] inputImg, int width, int height) {
    double[][] returnImg = new double[width][height];
    double[][] sobelV = new double[][]{{1.0, 0.0, -1.0},
    ↪ {2.0, 0.0, -2.0}, {1.0, 0.0, -1.0}};
    double[][] sobelH = new double[][]{{1.0, 2.0,
    ↪ 1.0}, {0.0, 0.0, 0.0}, {-1.0, -2.0,
    ↪ -1.0}};

    int radius = 1;
    double maxGradient = 1.0;

    // achtung! hier keine Normierung
    double[][] resultSobelV = ConvolveDouble(
    ↪ inputImg,width,height,sobelV,radius);
    double[][] resultSobelH = ConvolveDouble(
    ↪ inputImg,width,height,sobelH,radius);

    for( int x = 0; x < width; x++) {
        for( int y = 0; y < height; y++) {
            double vAbs = Math.abs(
            ↪ resultSobelV[x][y]);
            double hAbs = Math.abs(
            ↪ resultSobelH[x][y]);
            double resVal = vAbs + hAbs;
            returnImg[x][y] = resVal;

            // new max gradient?
            if(resVal >maxGradient)
                ↪ maxGradient = resVal;
        }
    }
}

```

```

        //finally normalize by max gradient value
        double corrFactor = maxGradient/255.0;

        for(int x = 0; x < width; x++) {
            for ( int y = 0; y < height; y++) {
                returnImg[x][y] /= corrFactor;
            }
        }

        return returnImg;
    }
}

```

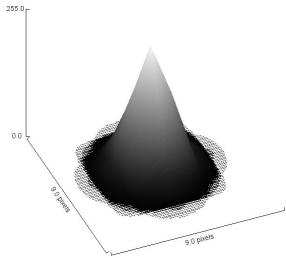

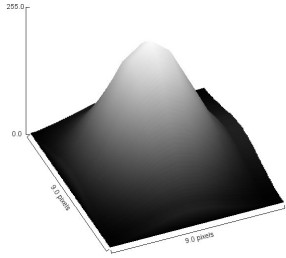

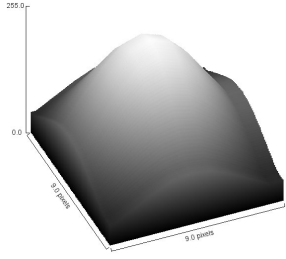

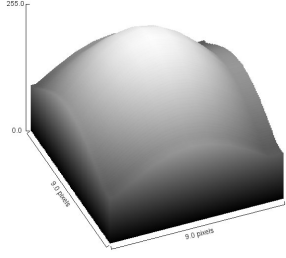

$\frac{\text{sigma}}{\text{radius}}$	Masken Surface Plot	gefiltertes Bild
$\frac{1}{4}$		
$\frac{2}{4}$		
$\frac{3}{4}$		
$\frac{4}{4}$		

Table 1: Gauss Filter Größen

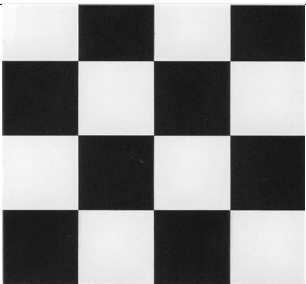
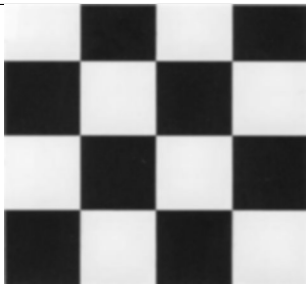
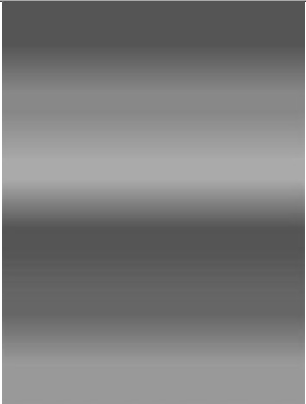
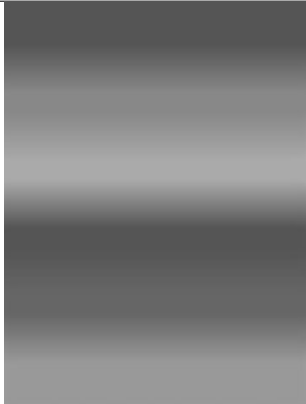
<i><u>sigma</u></i> <i><u>radius</u></i>	Masken Surface Plot	gefiltertes Bild
$\frac{1}{4}$		
$\frac{2}{4}$		

Table 2: Gauss Filter Evaluierung

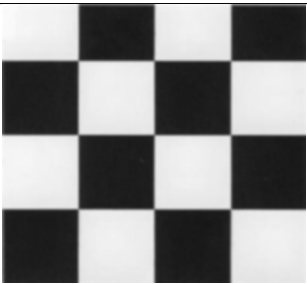
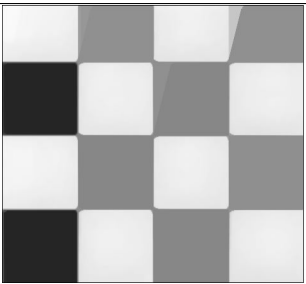
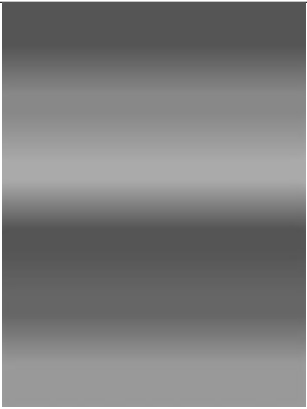

<i>$\frac{\sigma}{r}$</i>	Gauss gefiltertes Bild	Median gefiltertes Bild
$\frac{1}{4}$		
$\frac{2}{4}$		

Table 3: Gauss Filter vs. Median Filter

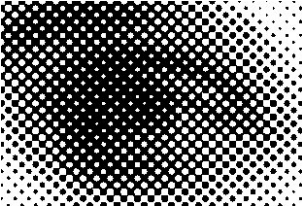
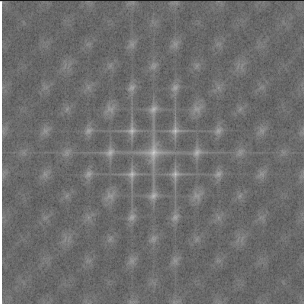

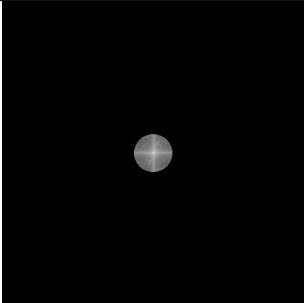
Bild	FFT
	
	

Table 4: Auswertung Auge

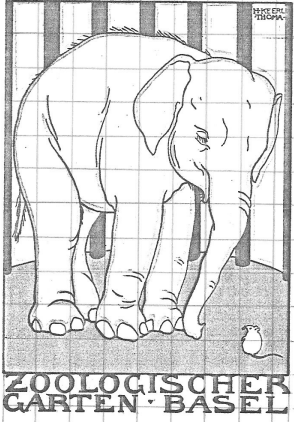
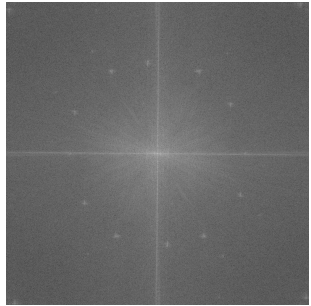
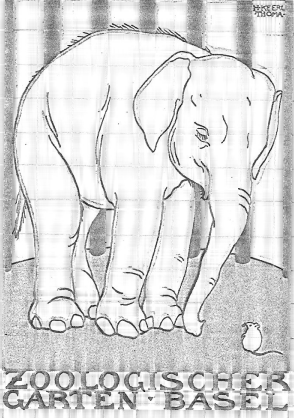
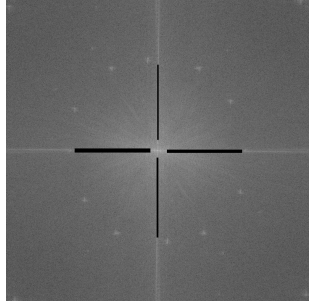
Bild	FFT
	
	

Table 5: Auswertung Elefant

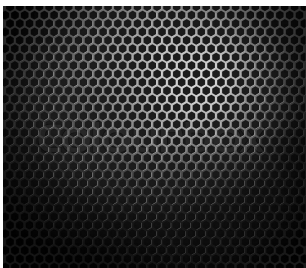
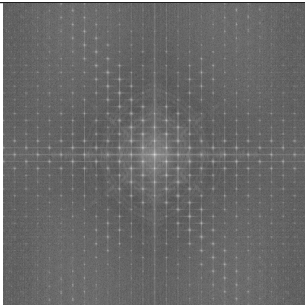

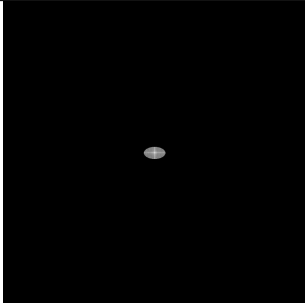
Bild	FFT
	
	

Table 6: Auswertung Lochgitter