

Übungsaufgaben I, SBV1

Lisa Panholzer, Lukas Fiel

October 29, 2018

1 Übungsaufgaben I

1.1 Gauss Filter

a) Implementierung

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 der Hausübung wurde die Klasse um die Methode *GetGaussMask* erweitert. In dieser wird die Verteilung einer Gauss Kurve auf eine 2 dimensionale Maske übertragen. Das Verhältnis von Sigma zur Masken-Abmessung wird detailliert in Punkt B beschrieben.

```
        ; columns
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_

```

; columns

```

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 numElements = size * size;
    double maskVal = 1.0 / numElements;
    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;
}

```

b) Darstellung der Gauss-Maske mittels Surface-Plot

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

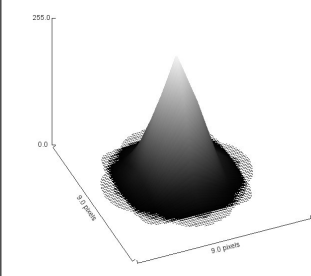
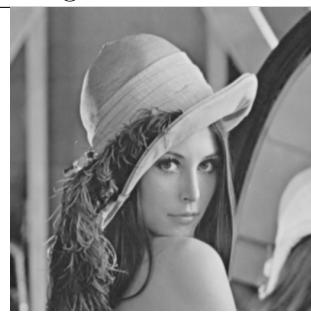
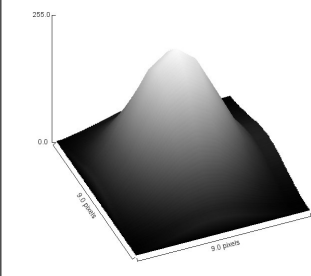

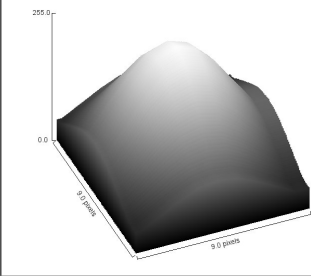
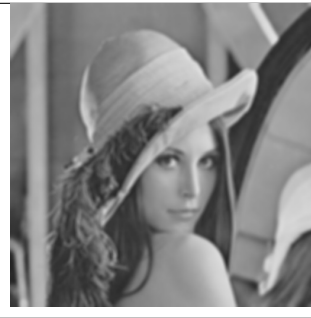
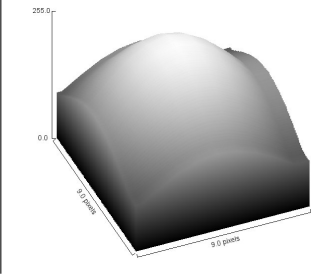
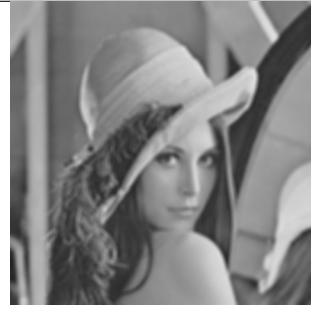
$\frac{\text{sigma}}{\text{radius}}$	Masken Surface Plot	gefiltertes Bild
$\frac{1}{4}$		
$\frac{2}{4}$		
$\frac{3}{4}$		
$\frac{4}{4}$		

Table 1: Gauss Filter Größen

c) **Auswirkungen im Bereich von Kanten und ansteigenden Intensitäten**

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: $\frac{\sigma}{radius} = \frac{2}{4}$

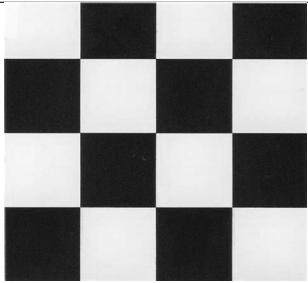
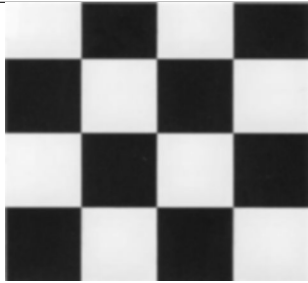
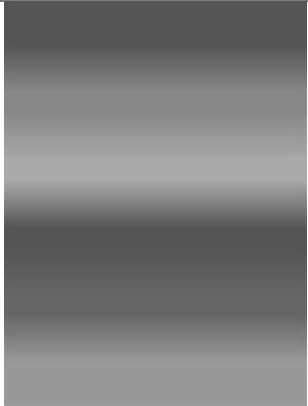
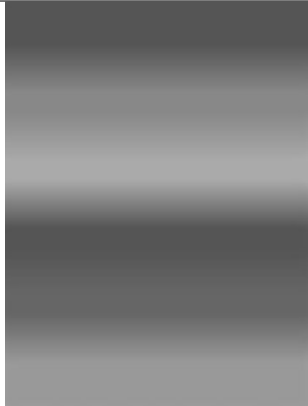
$\frac{\sigma}{radius}$	Masken Surface Plot	gefiltertes Bild
$\frac{1}{4}$		
$\frac{2}{4}$		

Table 2: Gauss Filter Evaluierung

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.

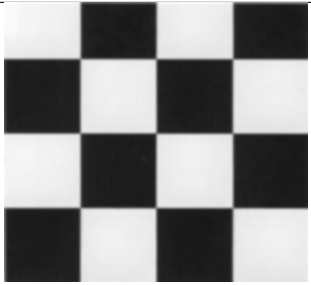
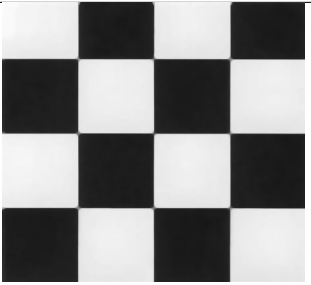
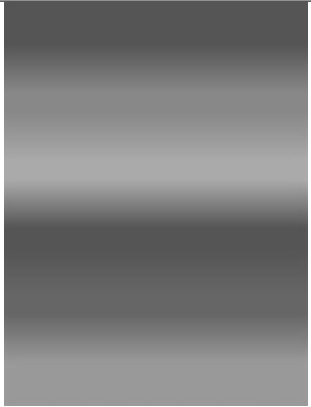
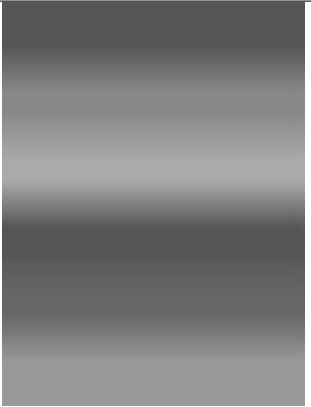
$\frac{\sigma}{r}$	Gauss gefiltertes Bild	Median gefiltertes Bild
$\frac{1}{4}$		
$\frac{2}{4}$		

Table 3: Gauss Filter vs. Median Filter

1.2 MedianFilter

a) Implementierung

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.

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

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.");

        System.out.println("Now_plot_4x4_to_see_filtereffect.");

        plot4x4(ip, resultImage);

    } // run

    private void plot4x4(ImageProcessor ip, double[][] filteredImg) {
        int segments = 4;

        byte[] pixels = (byte[]) ip.getPixels();
        int width = ip.getWidth();
        int height = ip.getHeight();
        int[][] inArr = ImageJUtility.convertFrom1DByteArr(pixels, width, height
        ↳ );
    }
}
```

```

        double[][] resultImg = ImageJUtility.convertToDoubleArr2D(inArr, width,
            ↪ height);

        int xCaroLength = width / segments;
        int yCaroLength = height / segments;

        // for every region
        for (int i = 0; i < segments; i++) {
            for (int j = 0; j < segments; j++) {
                if ((i+j) % 2 == 0) {
                    int xIndex = i*xCaroLength;
                    int yIndex = j*yCaroLength;
                    // calculate region
                    Rectangle roi = new Rectangle(xIndex, yIndex,
                        ↪ xCaroLength, yCaroLength);
                    double[][] tmpImg = ImageJUtility.cropImage(
                        ↪ filteredImg, roi.width, roi.height, roi);

                    // copy region to result image
                    for (int x = 0; x < xCaroLength; x++) {
                        for (int y = 0; y < yCaroLength; y++) {
                            resultImg[xIndex + x][yIndex + y
                                ↪ ] = tmpImg[x][y];
                        }
                    }
                }
            }
        }

        ImageJUtility.showNewImage(resultImg, width, height, "4x4_caro_for_
            ↪ filter_effect_evaluation");
    }

    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 = 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++) {

                Rectangle roi = getROI(width, height, x, y, radius);
                double[][] mask = ImageJUtility.cropImage(
                    ↪ inDataArrDouble, roi.width, roi.height, roi);
                double median = getMedian(mask, roi.width, roi.height);
                resultImage[x][y] = median;
            }
        }
        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);
    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_

```

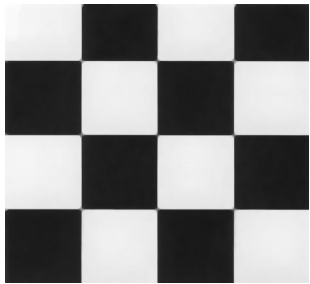
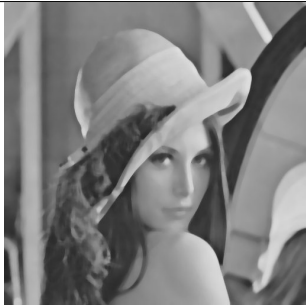
Median gefiltertes Schachbrett	Median gefilterte Bild "Lena"
	

Table 4: Median Filter

b) 4x4 Segment Darstellung und Statistische Werte



Figure 1: 4x4 Darstellung des Originalbildes überlagert mit Segmenten des gefilterten Bildes



Originalbild	gefiltertes Bild $radius = 8$
	
Mean: 127.46 StdDev: 46.6 Min: 29 Max: 243	Mean: 127.45 StdDev: 44.1 Min: 38 Max: 220

Table 5: Statistische Auswertung

c) Salt & Pepper Rauschen

Der Salt & Pepper Filter wird auf ein Testbild so oft angewendet, bis die Anzahl der Rausch-Pixel den Anteil der ursprünglichen Bild-Pixel übersteigt. Dies wird ca. bei einem Rausch-Anteil von über 50% stattfinden. Da bei der Anwendung des Median-Filters immer der mittlere Maskenwert herangezogen wird, kann bei einem Rausch-Anteil von über 50% nur mehr Schwarz oder Weiß auftreten. Ab diesem Zeitpunkt kann der Median-Filter die Störsignale des Salt & Pepper Filter nicht mehr korrigieren.



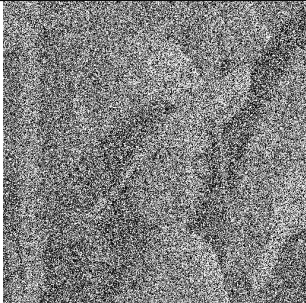
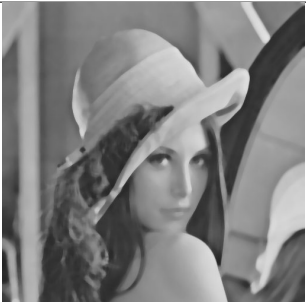


ein bisschen verrauscht	sehr verrauscht	fast unkenntlich
		
normaler Filtereffekt	erste Fehldarstellungen	ganze Bereiche sind schwarz/weiß
		

Table 6: Anwendung des Median Filters auf Bilder die mit Salt Pepper Noise verschlechtert wurden.

1.3 Steuerung des Filtereffekts

a) Vergleich

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 initiale Erstellen der Maske ausschlaggebend ist.

Ein Vergrößern der Maske steigert die benötigte Rechenzeit enorm. Man erkennt auch gut, dass eine große Filtermaske nicht unbedingt mit einem enormen Filtereffekt zu tun haben muss. Da hier ein kleines *sigma* gewählt wurde, ist auch die Auswirkung des Filters nicht groß, aber deutlich von der des Filters mit der kleinen Maske unterscheidbar (siehe Figure 7).

Zusätzlich zur Klasse *FiltereffektEvaluierung_.java* werden auch viele der anderen in diesem Papier besprochenen Klassen benötigt.

```
        ; columns
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_SG + 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_

```

Die so gewonnenen Daten wurden in eine Excel Tabelle eingetragen und in Figure 2 und 3 dargestellt. Man erkennt gut den linearen Zusammenhang

den eine mehrmalige Ausführung des Codes mit sich bringt. Zu beachten gilt auch dass Gauss und Mean Filter wie erwartet ähnliche Ergebnisse liefern da Sie auf den selben Methoden aufgebaut sind und lediglich die initiale Maskenerstellung die Filter unterscheidet. Unsere Messungen zeigten, dass auch der selbst geschriebene Median Filter eine lineare Laufzeit aufweist. Diese ist aber deutlich höher als die von Mean und Gauss Filter.

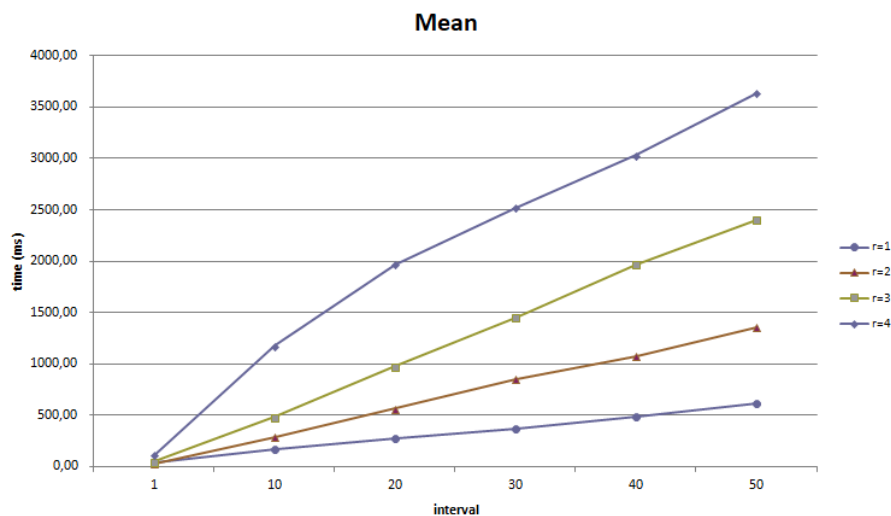


Figure 2: Mehrfachausführung des Mean Filters

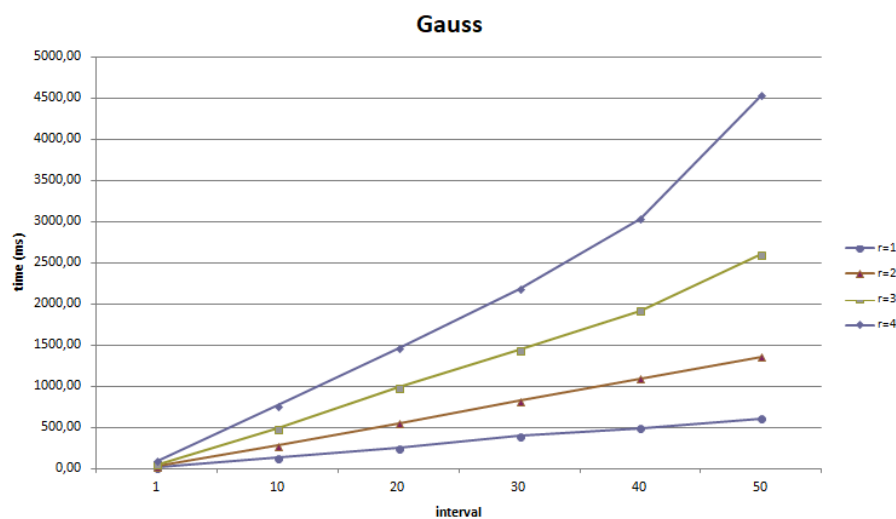


Figure 3: Mehrfachausführung des Gauss Filters

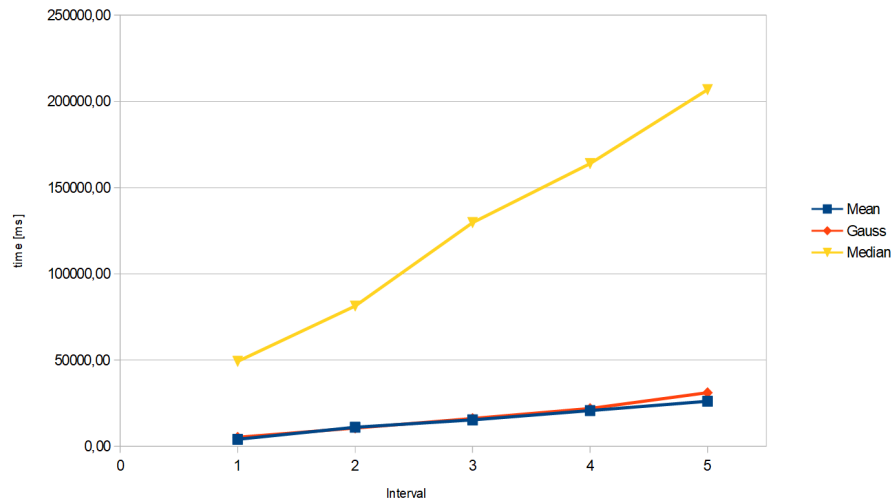


Figure 4: Zeitmessungen mit $radius = 40$

Ein Vergleich der Maskengrößen ergab auch hier einen linearen Zusammenhang bei allen 3 Filterarten. Siehe Figure 5.

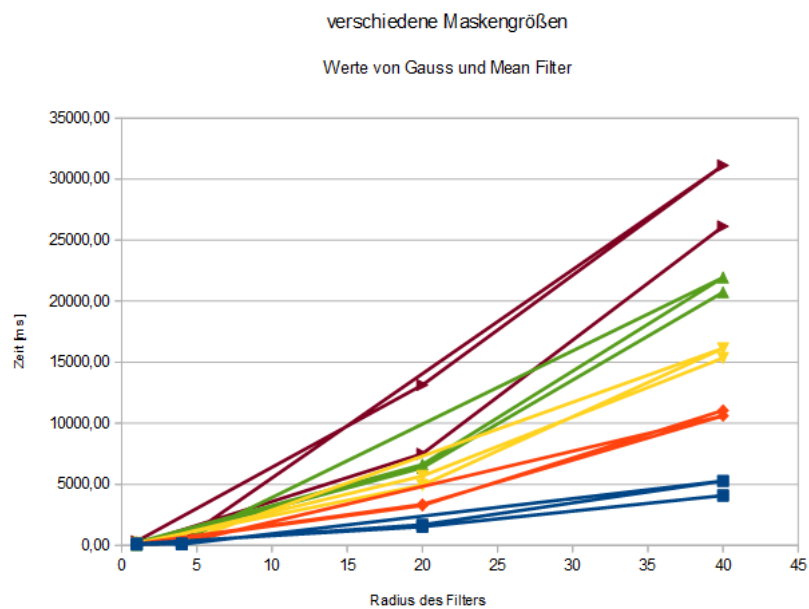


Figure 5: Vergleich verschiedener Maskengrößen

Allgemein würde ein mehrdimensionales Bild eine mehrfache Ausführung des 2D Filters erfordern. Da dieser Zusammenhang linear ist wurde bereits gezeigt. 3D Masken könnten aber neben dem Filtereffekt zum Beispiel bei einem RGB Bild auch Farbtöne verstärken/abschwächen. Es könnte zum Beispiel der Rot-Ton abhängig vom Grün-Ton verändert werden. Hierzu müsste für jeden Pixel im dreidimensionalen Bild die gesamte Information aller 3 Dimensionen im Bereich der Maske eingelesen und verarbeitet werden. Dies würde eine erhebliche Laufzeitsteigerung bedeuten.

t_{gP}	time to get pixel value. Die Zeit einen Pixel aus der Berechnung mit einer 2D Maske
$d = dimensions - 1$	Anzahl der zusätzlichen Einträge in der dritten Dimension
$size = length * width$	Anzahl der Pixel im zweidimensionalen Bild
$t_{2D} = size * t_{gP}$	Laufzeit eines 2D Bildes:
$t_{3D} = d * size * t_{gp}$	Laufzeit eines 3D Bildes
$t_{multiple3D} = d * size * (d * t_{gP})$ $= d^2 * size * t_{gP}$	quadratische Laufzeit da bei jedem Pixel alle Dimensionen berücksichtigt werden müssen

Figure 6: Berechnung der Laufzeit bei 3D Bildern.

b) Diskussion der Ergebnisse

Wie in Tabelle 7 gut erkennbar ist, kann durch wiederholtes Anwenden einer kleinen Maske nur sehr schwer das Ergebnis einer großen Maske erreicht werden. Es kommt daher darauf an, welchen optischen Effekt man erzielen möchte.

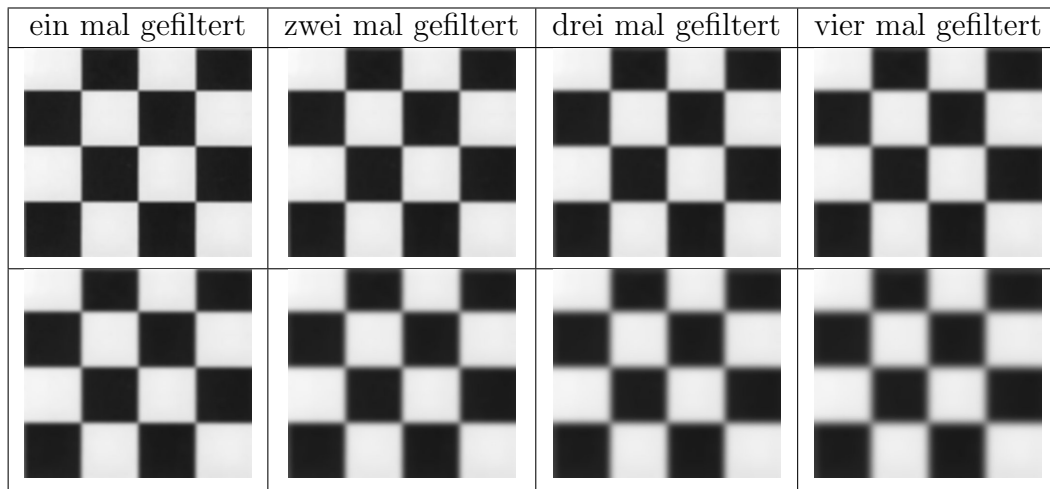


Figure 7: Wiederholte Anwendung des Gauss Filters mit $radius = 4$ (oben) und $radius = 40$ (unten)

1.4 Histogrammeinebnung

To-Do Lisa

a) Implementierung

```
; columns

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_

```

Die Implementierung wurde anhand der folgenden drei Bilder getestet:

1. Strand: Das Bild "Strand" (siehe Figure 8) enthält eine halbwegs gleichmäßige Verteilung der Grautöne. Nach Anwendung der Histogrammeinebnung (siehe Figure 9) kann man eine Verstärkung des Kontrastes erkennen.

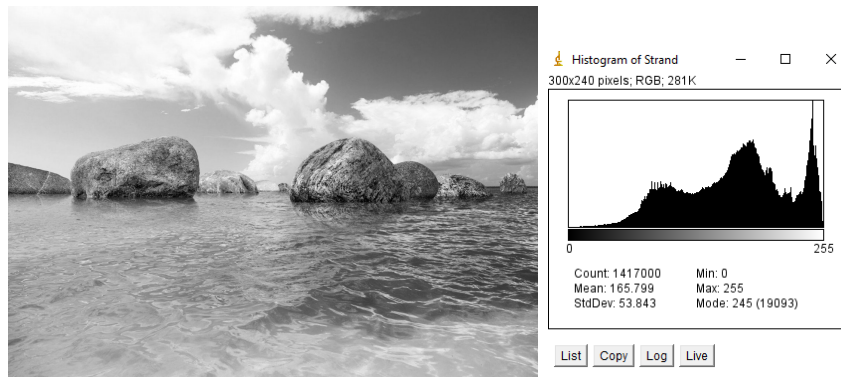


Figure 8: Original Test-Bild "Strand" mit dazugehörigem Histogramm

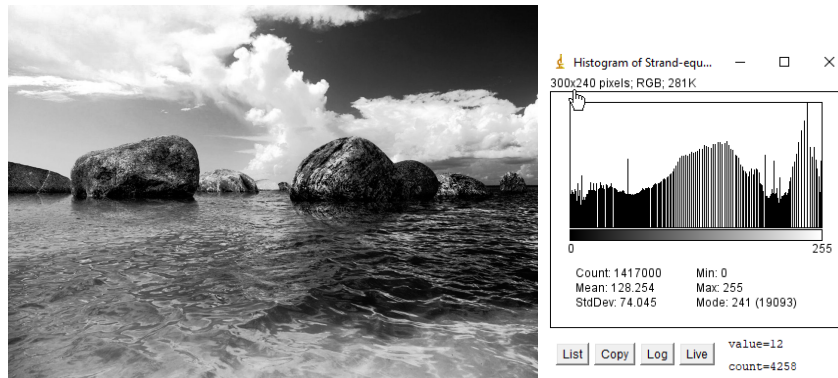


Figure 9: Test-Bild "Strand" mit dazugehörigem Histogramm nach der Histogrammeinebnung

2. Russland: Das Bild "Russland" (siehe Figure 10) enthält sehr wenig Kontrast und viele der Grautöne sind im Histogramm benachbart. Nach der Anwendung der Histogrammeinebnung (siehe Figure 11) verstärkt sich der Kontrast um ein vielfaches und selbst die Wolkenformation sind nun detailliert sichtbar.

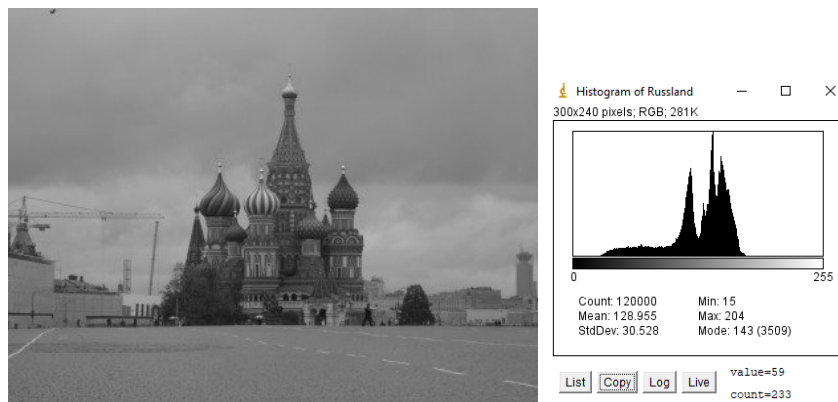


Figure 10: Original Test-Bild "Russland" mit dazugehörigem Histogramm

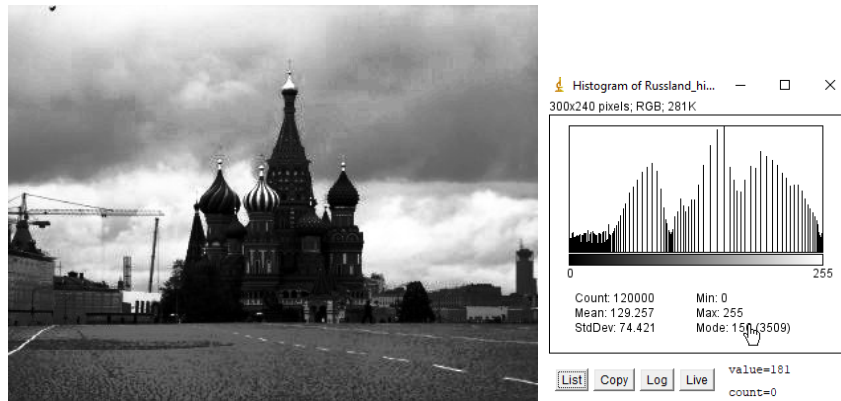


Figure 11: Test-Bild "Russland" mit dazugehörigem Histogramm nach der Histogrammeinebnung

3. Landschaft: Das Bild "Landschaft" (siehe Figure 12) hat im Vergleich zu den ersten beiden Testbildern einen höheren Kontrast und ist dunkler. Bei der Anwendung der Histogrammeinebnung (siehe Figure 13) kann man nun beobachten, das aufgrund der Gleichverteilung der Grautöne Richtung den Maximalwert (255), das Bild sich aufhellt und der Kontrast erhalten bleibt.



Figure 12: Original Test-Bild "Landschaft" mit dazugehörigem Histogramm



Figure 13: Test-Bild "Landschaft" mit dazugehörigem Histogramm nach der Histogrammeinebnung

b) Diskussion Histogrammeinebnung

Es kann zu einer Verschlechterung der Bildqualität kommen, wenn die Histogrammeinebnung zum Beispiel auf ein stark überbelichtetes Bild angewandt wird. Bei dem Testbild "Straße mit Fußgängern" (siehe Figure 14) sieht man, dass der Großteil der Pixelintensitäten ca. über 200 liegt. Findet nun die Einebnung (siehe Figure 15) statt werden die Verläufe nicht mehr weich dargestellt. Die Unterschiede zwischen den Pixelintensitäten werden zu hoch und Teile des Bildes deshalb kantiger dargestellt.



Figure 14: Original Test-Bild "Überbelichtung" mit dazugehörigem Histogramm

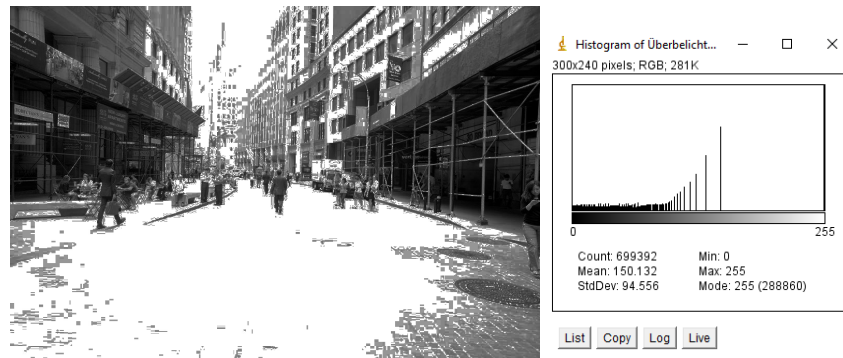


Figure 15: Test-Bild "Überbelichtung" mit dazugehörigem Histogramm nach der Histogrammeinebnung

1.5 Raster-Entfernung im Frequenzraum

a) Entfernen von horizontalen und vertikalen Balken

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

Testdatensatz 1)

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 (siehe Tabelle 7). Da aber die Gitterstäbe selbst periodisch im Bild vorkommen und auch die Schrift sich wiederholende senkrechte Kanten hat, ist 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 belassen. 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.

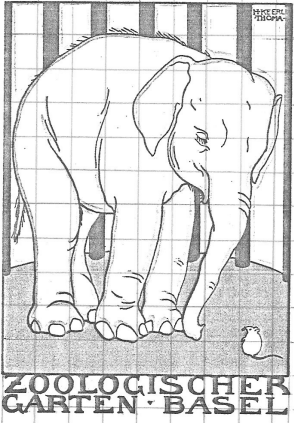
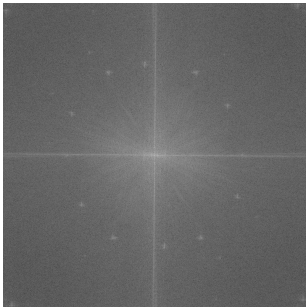
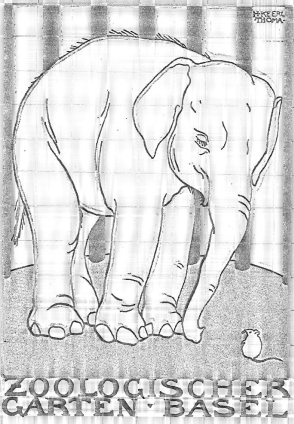
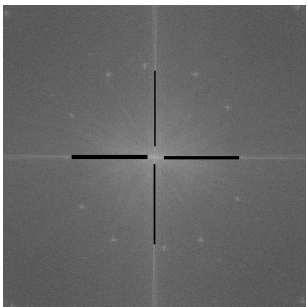
Bild	FFT
	
	

Table 7: Auswertung Elefant

Testdatensatz 2)

Zu Testzwecken 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 zeigt deutlich ein periodisches Muster. Will man nun die eigentliche Bildinformation gewinnen, müssen hochfrequente Anteile des Bildes entfernt werden. Tabelle 8 zeigt deutlich dass durch ein Entfernen der Randbereiche (höhere Frequenzen) im FFT Bild und die anschließende Rücktransformation die eigentliche Bildinformation gewonnen werden konnte.

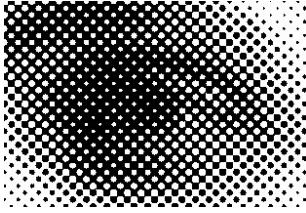
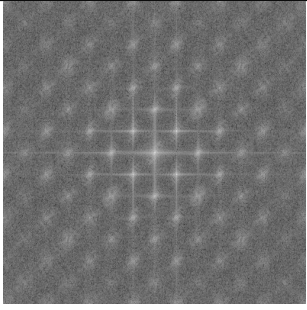

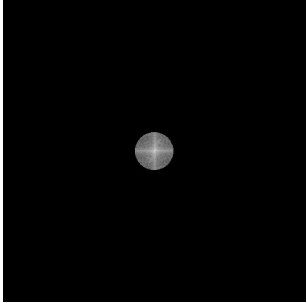
Bild	FFT
	
	

Table 8: Auswertung Auge

Testdatensatz 3)

Als weiteres Beispiel wurde ein perspektivisch beleuchtetes Lochgitter herangezogen (siehe Tabelle 9). 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 Beleuchtung. Das Lochgitter konnte vollkommen entfernt werden. Es ist auch anzumerken, dass im rücktransformierten Bild eine Schrift "*colourbox*" deutlich zu erkennen ist. Bei genauerer Betrachtung des Ursprungsbildes ist diese hinter dem Gitter zu erkennen.

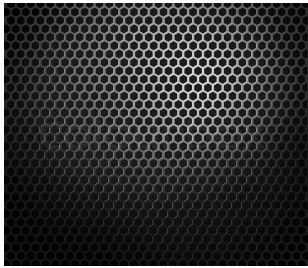
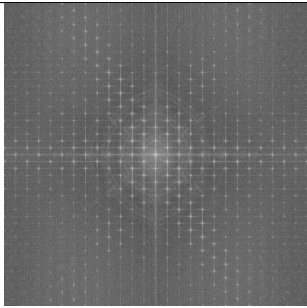


Bild	FFT
	
	

Table 9: Auswertung Lochgitter

b) 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 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.

Um den Raster möglichst gut zu entfernen sollten die Linien regelmäßig, in gleichen Abständen auf dem Bild vorhanden sein. Diese sollten auch sehr klar dargestellt werden, dann können diese möglichst exakt vom restlichen Bildinhalt getrennt werden.

1.6 Anhang

; columns

```
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[][] kernellImg = new double[size][size];

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

    return kernellImg;
}

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

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

    double[][] kernellImg = 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;

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

    return kernellImg;
}

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;
}

```