Übungsaufgaben I, SBV1

Lisa Panholzer, Lukas Fiel

October 22, 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 *ConvolutionFilter* 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.

Anschlißend 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 Filtereffeckt sichtbar ist, während Kanten deutlich verschwommen erscheinen. Gewähltes Verhältnis: $\frac{sigma}{radius} = \frac{2}{4}$

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;
    }


}
```

1.0.2 Test

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 coorinates of the
         * upper left corner and width and hight 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 coodrinate 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

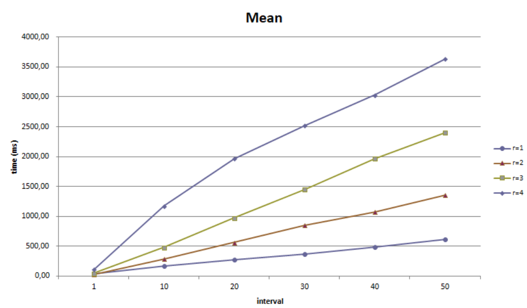
```
import ij.*;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;
import ij.gui.GenericDialog;


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

```

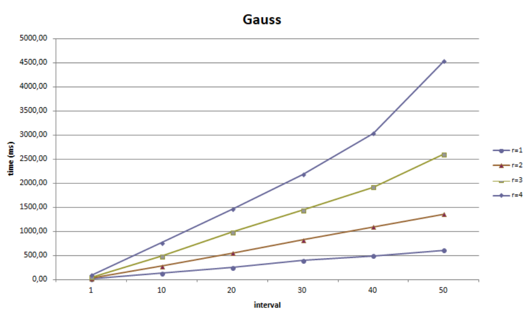
```
public void run(ImageProcessor ip) {

        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.");
        }
```

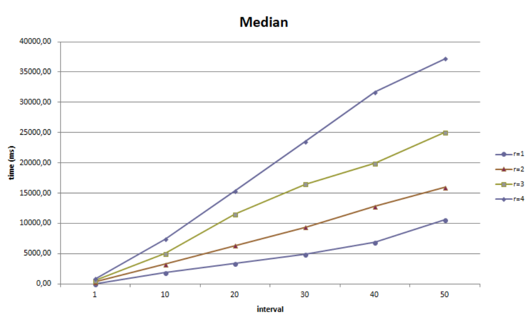
```
// ---------- GAUSS ----------

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_
```

3.1.1 Ablaufund Idee







3.1.2 Tests

4 Histogrammeinebnung

4.1 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 coorinates of the
 * upper left corner and width and hight 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 coodrinate 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_
```

4.2 Tests

5 Raster-Entfernung im Frequenzraum

5.1 Workflow

- Starten von *imageJ.exe*
- Öffnen eines Bildes
- *Process* → *FFT* → *FFT*
- Zuschneiden des interessanten Bereichs im FFT Bild
- *Process* → *FFT* → *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 er 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;
    }


}
```

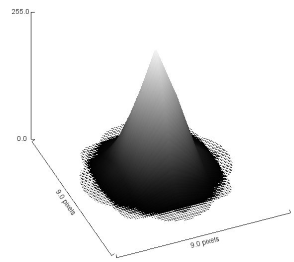
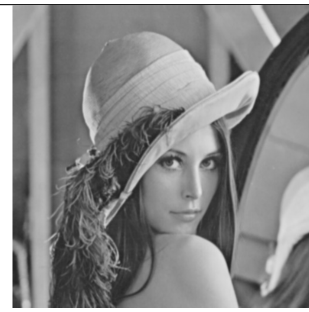
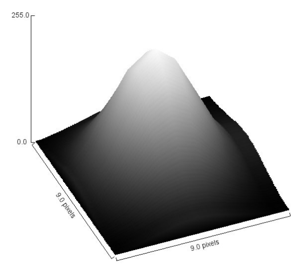
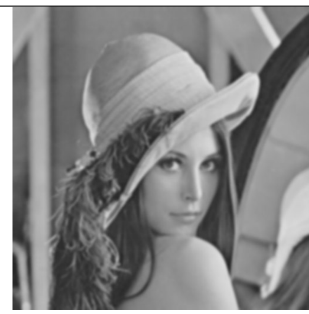
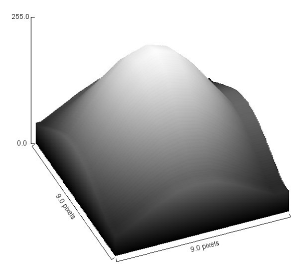
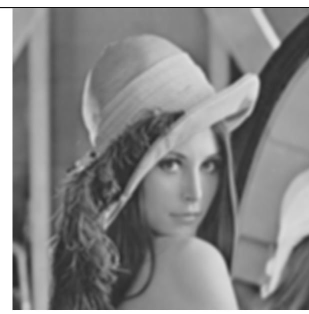
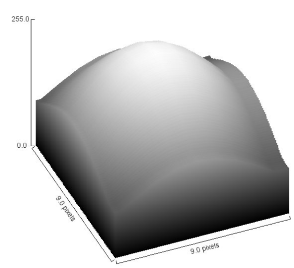
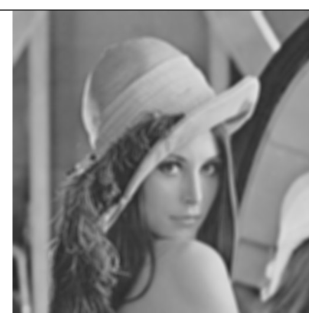
$\frac{sigma}{radius}$	Masken Surface Plot	gefiltertes Bild
$\frac{1}{4}$		
$\frac{2}{4}$		
$\frac{3}{4}$		
$\frac{4}{4}$		

Table 1: Gauss Filter Größen

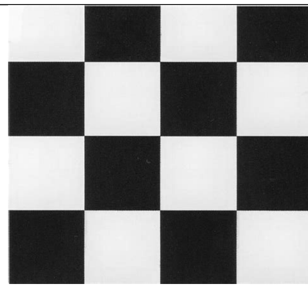
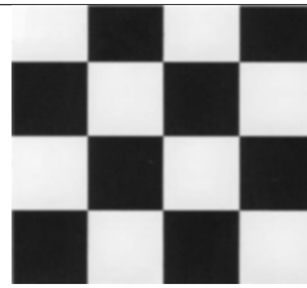
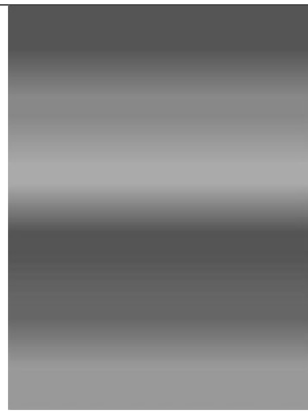
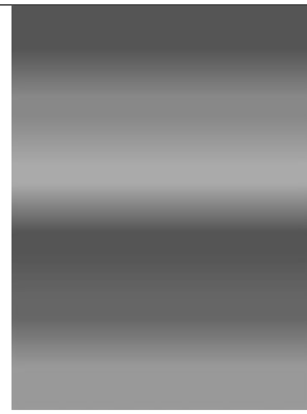
$\frac{sigma}{radius}$	Masken Surface Plot	gefiltertes Bild
$\frac{1}{4}$		
$\frac{2}{4}$		

Table 2: Gauss Filter Evaluierung

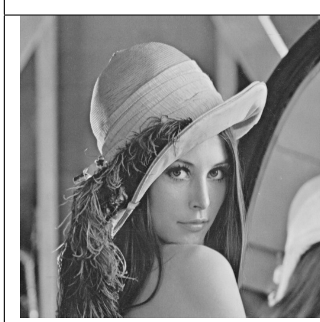
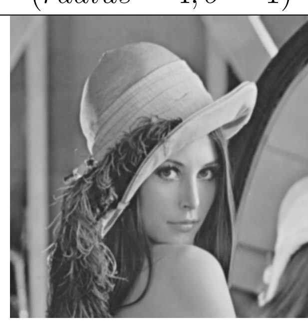
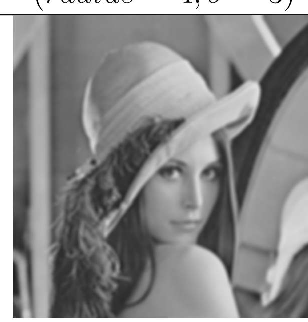
Ausgangsbild	gefiltertes Bild ($radius = 4, \sigma = 1$)	gefiltertes Bild ($radius = 4, \sigma = 3$)
		

Table 3: Test Gauss

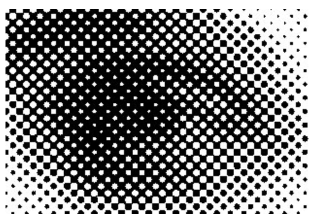
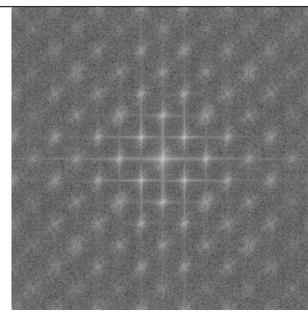
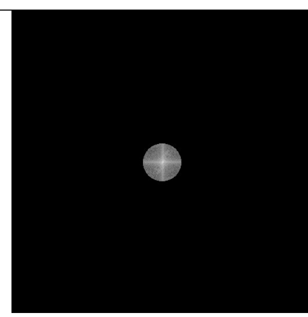
Bild	FFT
	
	

Table 4: Auswertung Auge

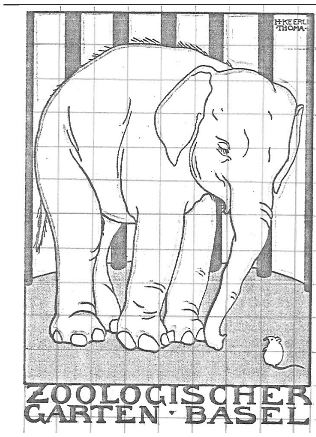
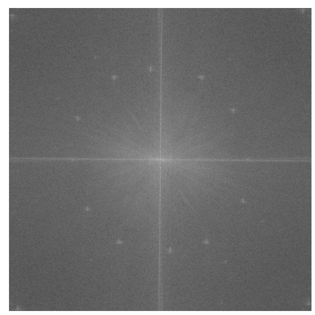
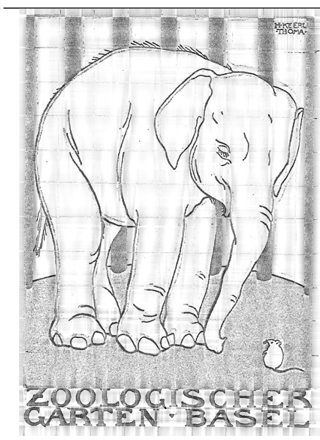
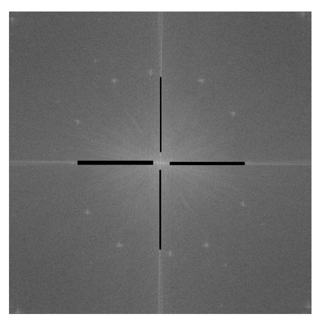
Bild	FFT
	
	

Table 5: Auswertung Elefant

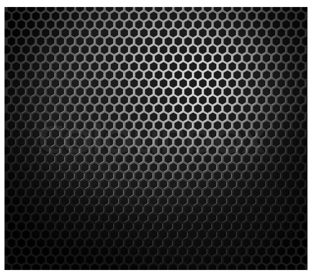
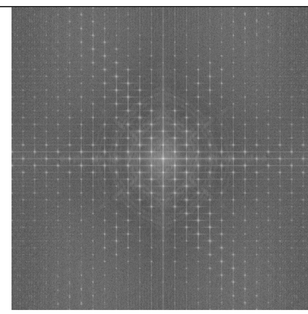
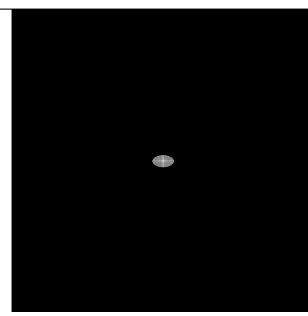
Bild	FFT
	
	

Table 6: Auswertung Lochgitter