Übungsaufgaben III, SBV1

Lukas Fiel, Lisa Panholzer

January 6, 2019

3 Übungsaufgaben III

3.1 Resampling und Bildüberlagerung

a) Zerteilen eines Bildes

Zur vertikalen Teilung eines Bildes wurde ein simpler Filter *ChopImgInHalf_* in *ImageJ* implementiert. Dieser definiert zuerst eine ROI (region of interest) welche die erste Hälfte des Bildes beinhaltet. Mittels *ImageJUtility.chopImage* kann dieser Bereich aus dem Ursprungsbild herausgeschnitten und angezeigt werden. Die Berechnung der zweiten Hälfte des Bildes unterscheidet sich lediglich durch die linke obere Koordinate des interessanten Bereichs (ROI).

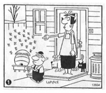
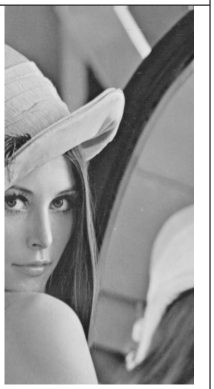
Testbild	erste Bildhälfte	zweite Bildhälfte
		
		

Table 1: Zerteilung eines Bildes anhand selbst definiertem Filter

; columns

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

```

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

                double widthHalf = width / 2.0;
                double[][] tmpImage = ImageJUtility.convertToDoubleArr2D(inDataArrInt,
                    ↪ width, height);
                Rectangle roi = new Rectangle(0, 0, (int)widthHalf, height);
                double[][] Img1 = ImageJUtility.cropImage(tmpImage, roi.width, roi.
                    ↪ height, roi);
                ImageJUtility.showNewImage(Img1, (int)widthHalf, height, "first␣half␣
                    ↪ image");
                roi = new Rectangle((int)widthHalf, 0, (int)widthHalf, height);
                double[][] Img2 = ImageJUtility.cropImage(tmpImage, roi.width, roi.
                    ↪ height, roi);
                ImageJUtility.showNewImage(Img2, (int)widthHalf, height, "second␣half␣
                    ↪ image");

        } //run

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

} //class FilterTemplate_
```

b) Transformation mittels Nearest Neighbor und Bilinearer Interpolation

; columns

```

import ij.*;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;
import java.awt.*;
import ij.gui.GenericDialog;

public class RegisterFinal_ implements PlugInFilter {

        boolean nnFlag = false;

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

        int widthHalf = (int) (width / 2.0);
                double[][] img1 = chopImgInHalf(inDataArrInt, width, height, widthHalf, true);
                double[][] img2 = chopImgInHalf(inDataArrInt, width, height, widthHalf, false);

                int[][] intImg2 = ImageJUtility.convertToIntArr2D(img2, widthHalf, height);

        // define transform
        double transX = getUserInput(0,"deltaX");
        double transY = getUserInput(0,"deltaY");
        double rotAngle = getUserInput(0,"rotation");

        //int[][] transformedImg = transformImage(inDataArrInt, width, height, transX, transY, rotAngle);
        int[][] transformedImg = transformImage(intImg2, widthHalf, height, transX, transY, rotAngle);

        ImageJUtility.showNewImage(transformedImg, widthHalf, height, "transformed image");

        } //run

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

        public static int getUserInput(int defaultValue, String nameOfValue) {
                // user input
                System.out.print("Read user input: " + nameOfValue);
                GenericDialog gd = new GenericDialog(nameOfValue);
                gd.addNumericField("please input " + nameOfValue + ": ", defaultValue, 0);
                gd.showDialog();
                if (gd.wasCanceled()) {
                        return 0;
                }
                int radius = (int) gd.getNextNumber();
                System.out.println(radius);
                return radius;
        }

        public double GetBilinearinterpolatedValue(int[][] inImg, double x, double y, int width, int height) {
                // calculate the delta for x and y
                double deltaX = x - Math.floor(x);
                double deltaY = y - Math.floor(y);

                // set calculation fragment
                int xPlus1 = (int) x + 1;
                int yPlus1 = (int) y + 1;

                //handling translation and rotation for x and y
                if(x < 0 || x >= width || y < 0 || y >= height || xPlus1 < 0 || xPlus1
                    ↪ >= width || yPlus1 < 0 || yPlus1 >= height) {
                        return 0;
                }

                // get 4 neighboring pixels
                int neighbor1 = inImg[xPlus1][(int) (y)];
                int neighbor2 = inImg[(int) (x)][yPlus1];
                int neighbor3 = inImg[xPlus1][yPlus1];
                int neighbor4 = inImg[(int) (x)][(int) (y)];

                // calculate weighted mean out of neighbors
                double weightedMean = ((1 - deltaX) * (1 - deltaY) * neighbor4) + (
                        ↪ deltaX * (1 - deltaY) * neighbor1)
                            + ((1 - deltaX) * deltaY * neighbor2) + (deltaX * deltaY
                                ↪  * neighbor3);

                return weightedMean;
        }

        public int[][] transformImage(int[][] inImg,int width, int height, double transX
                ↪ , double transY, double rotAngle) {

                //allocate result image
                int[][] resultImg = new int[width][height];

                // prepare cos theta, sin theta
                double cosTheta = Math.cos(Math.toRadians(-rotAngle));
                double sinTheta = Math.sin(Math.toRadians(-rotAngle)); // - weil
                        ↪ backgroundmapping

                double widthHalf = width / 2.0;
                double heightHalf = height / 2.0;


                //1) interate over all pixels and calc value utilizing backward-mapping
                for( int x= 0; x < width; x++) {
                        for (int y  =0; y< height; y++) {

                            double tmpposX = x - widthHalf;
                            double tmpposY = y - heightHalf;

                            //3) rotate
                            double posX = tmpposX * cosTheta + tmpposY * sinTheta;
                            double posY = - tmpposX * sinTheta + tmpposY * cosTheta;

                            //4) translate
                            posX -= transX;
                            posY -= transY;


                            // move origin back from center to top corner
                            posX = posX + widthHalf;
                            posY = posY + heightHalf;

                            //6) get interpolated value if flag is true
                            if (nnFlag) {
                                int nnX = (int) (posX + 0.5);
                                int nnY = (int) (posY + 0.5);

                                //6) assigne value from original img inImg if
                                    ↪ inside the image boundaries
                                if(nnX >= 0 && nnX <width && nnY >= 0 && nnY < height) {
                                    resultImg[x][y] = inImg[nnX][nnY];
                                }
                            }
                            else {
                                // if nearest neighbor flag is false, do bilinear interpolation
                                double resultVal = GetBilinearinterpolatedValue(inImg, posX, posY, width, height);

                                //set new rounded value for current location
                                resultImg[x][y] = (int) (resultVal + 0.5);

                            }


                        }
                }
                return resultImg;
        }


        public static double[][] chopImgInHalf(int[][] inDataArrInt, int width, int height, int widthHalf, boolean flag) {
                // store half of width in int var

                // create temporary image
                double[][] tmpImage = ImageJUtility.convertToDoubleArr2D(inDataArrInt, width, height);

                if (flag == true) {
                        // create region of interest
                        Rectangle roi = new Rectangle(0, 0, widthHalf, height);

                        // crop image and store first half in var
                        double[][] Img1 = ImageJUtility.cropImage(tmpImage, roi.width, roi.height, roi);
                        ImageJUtility.showNewImage(Img1, widthHalf, height, "first␣half␣image");

                        return Img1;
                } else {

                        // create region of interest
                        Rectangle roi = new Rectangle(0, 0, widthHalf, height);

                        // overwrite roi with values for second half, crop image and store second half
                        // in var
                        roi = new Rectangle(widthHalf, 0, widthHalf, height);
                        double[][] Img2 = ImageJUtility.cropImage(tmpImage, roi.width, roi.height, roi);
                        ImageJUtility.showNewImage(Img2, widthHalf, height, "second␣half␣image");
                        return Img2;
                }
        }


    } //class FilterTemplate␣
```

c) automatische Registrierung

Es wurde ein Filter in *ImgaeJ* implementiert, der zur automatischen Registrierung von Bildinhalten herangezogen werden soll. Dabei wurde von den gegebenen Testbildern ausgegangen.

Da diese mit einer Bildtiefe von $8bit$ nur Werte von 0 (schwarz) bis 255 (weiß) aufweisen, kann mittels SSE einfach ein Algorithmus geschrieben werden, der die Bilder voneinander subtrahiert und die Pixelwerte des Resultatbildes als Fitness heranzieht und aufsummiert. Der Hintergrund der gegebenen Bilder ist dabei meist weiß (255). Bei einer Verschiebung und anschließender Subtraktion entstehen aus diesem Grund aber schwarze Fragmente am Rand. Dieser Umstand kann leicht eliminiert werden, indem das Ursprungsbild zu Beginn invertiert wird. So ist der Hintergrund schwarz (0). Kanten werden dementsprechend weiß (255) dargestellt.

Das invertierte Bild wird anschließend, wie in Punkt a) beschrieben, zerteilt und die Einzelbilder dargestellt.

Die eigentliche Registrierung verschiebt nun Bild1 in x und y Richtiung und rotiert dieses auch um jeweils ein Inkrement. Jedes dieser transformierten Bilder wird nun von Bild2 abgezogen und erneut ein Fitneswert berechnet. Es ist davon auszugehen, dass ein schwarzer Hintergrund (0) abgezogen von einem schwarzen Hintergrund (0) wiederum 0 ergibt. Werden allerdings weiße Pixel von schwarzem Hintergrund abgezogen, oder schwarzer Hintergrund von weißen Linien abgezogen, so erhält man Werte abweichend von 0. Auch Negativwerte sind so denkbar, wesshalb diese Differenzwerte zum Quadrat genommen werden. Hierdurch sind Differenzwerte immer positiv.

Wird Bild1 irgendwann genau auf die Position geschoben an der sich Bild2 befindet so subtrahieren sich die weißen Linien im Idealfall zu 0. So kann ein eindeutiger Fitnesswert errechnet werden, der sein Optimum bei 0 findet.

Aus Ressourcengründen werden all die beschriebenen Berechnungen/Verschiebungen mit dem NearesNeighbor Algorithmus berechnet. Ist das Optimum gefunden wird anschließend nocheinmal die Transformation mit Bilinearer Interpolation berechnet und von Bild2 subtrahiert. Das Resultatbild wird zum Schluss für den User sichtbar dargestellt um den Erfolg des Filters zu veranschlaulichen.

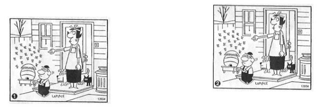
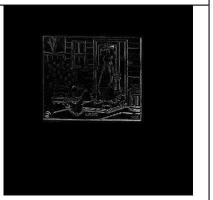
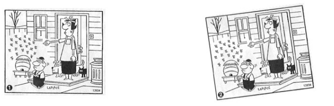
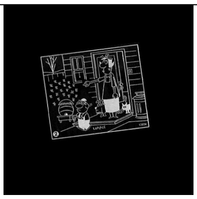
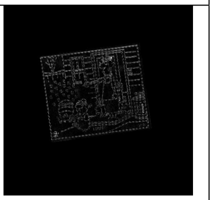
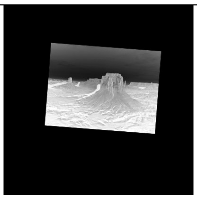
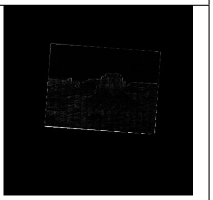
gegebenes Testbild	invertierter Bildausschnitt1	invertierter Bildausschnitt2	resultierendes Differenzbild
			
			
			

Table 2: Testfälle: automatische Registrierung

; columns

```

import ij.*;
import ij.plugin.filter.PlugInFilter;
import ij.process.*;
import java.awt.*;

public class AutoRegisterFinal_ 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) {
                // read image
                byte[] pixels = (byte[]) ip.getPixels();
                int width = ip.getWidth();
                int height = ip.getHeight();
                int[][] inDataArrInt = ImageJUtility.convertFrom1DByteArr(pixels, width,
                    ↪ height);

                // invert to set background to black
                int[] invertTF = ImageTransformationFilter.GetInversionTF(255);
                inDataArrInt = ImageTransformationFilter.GetTransformedImage(
                    ↪ inDataArrInt, width, height, invertTF);

        int widthHalf = (int) (width / 2.0);
        double[][] img1 = chopImgInHalf(inDataArrInt, width, height, widthHalf,
            ↪ true);
        double[][] img2 = chopImgInHalf(inDataArrInt, width, height, widthHalf,
            ↪ false);

        // initialize ranges
        int xRadius = 20;
        int yRadius = 20;
        int rotRadius = 20;

        // initialize arrays
        int[][] intImg1 = ImageJUtility.convertToIntArr2D(img1, widthHalf,
            ↪ height);
        int[][] intImg2 = ImageJUtility.convertToIntArr2D(img2, widthHalf,
            ↪ height);
        int[][] transformedImg;
        int[][] diffImg;
        int[][][] ssE = new int[2 * xRadius + 1][2 * yRadius + 1][2 * rotRadius
            ↪ + 1];

        // initial fitness
        diffImg = ImageJUtility.calculateImgDifference(intImg1, intImg2,
            ↪ widthHalf, height);
        int initialFitness = calculateSSE(diffImg, widthHalf, height);
        System.out.println("initiale␣Fitness:␣" + initialFitness);

        // fill ssE matrix and find minimum
        int minimum = initialFitness;
        int tmpSSE = 0;
        int minXind = 0;
        int minYind = 0;
        int minAngleInd = 0;
        for (int x = -xRadius; x < xRadius; x++) {
            for (int y = -yRadius; y < yRadius; y++) {
                for (int angle = -rotRadius; angle < rotRadius; angle++)
                    ↪ {
                    transformedImg = transformImage(intImg1,
                        ↪ widthHalf, height, x, y, angle, false);
                    diffImg = ImageJUtility.calculateImgDifference(
                        ↪ transformedImg, intImg2, widthHalf,
                        ↪ height);
                    tmpSSE = calculateSSE(diffImg, widthHalf, height
                        ↪ );
                    ssE[x + xRadius][y + yRadius][angle + rotRadius]
                        ↪ = tmpSSE;

                    // find minimum and save indices for later
                    if (tmpSSE < minimum) {
                        minimum = tmpSSE;
                        //System.out.println("current minimal
                            ↪ fitness: " + minimum);
                        minXind = x;
                        minYind = y;
                        minAngleInd = angle;
                    }
                }
            }
        }
        System.out.println("final␣Fitness:␣" + minimum);
        System.out.println("minXind:"+minXind+"minYind:"+minYind+"minAngleInd:"+
            ↪ minAngleInd);

        minXind = 22;
        minYind = -6;
        minAngleInd = -4;

        // plot difference image to proof the transformation
        transformedImg = transformImage(intImg1, widthHalf, height, minXind, minYind, minAngleInd, true);
        diffImg = ImageJUtility.calculateImgDifference(transformedImg, intImg2, widthHalf, height);
        ImageJUtility.showNewImage(diffImg, widthHalf, height, "fittest␣diff␣image");

    } // run

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

    public int[][] transformImage(int[][] inImg, int width, int height, double transX, double transY, double rotAngle,
            boolean interpolation) {

        // allocate result image
        int[][] resultImg = new int[width][height];

        // prepare cos theta, sin theta
        double cosTheta = Math.cos(Math.toRadians(-rotAngle));
        double sinTheta = Math.sin(Math.toRadians(-rotAngle)); // - weil backgroundmapping

        double widthHalf = width / 2.0;
        double heightHalf = height / 2.0;

        // 1) interate over all pixels and calc value utilizing backward-mapping
        for (int x = 0; x < width; x++) {
            for (int y = 0; y < height; y++) {

                double tmpposX = x - widthHalf;
                double tmpposY = y - heightHalf;

                // 3) rotate
                double posX = tmpposX * cosTheta + tmpposY * sinTheta;
                double posY = -tmpposX * sinTheta + tmpposY * cosTheta;

                // 4) translate
                posX -= transX;
                posY -= transY;

                // move origin back from center to top corner
                posX = posX + widthHalf;
                posY = posY + heightHalf;

                // 6) assigne value from original imag inImg if inside the image boundaries
                // get interpolated value if flag is true
                if (interpolation == false) {
                    int nnX = (int) (posX + 0.5);
                    int nnY = (int) (posY + 0.5);

                    // 6) assign value from original img inImg if inside the image boundaries
                    if (nnX >= 0 && nnX < width && nnY >= 0 && nnY < height) {
                        resultImg[x][y] = inImg[nnX][nnY];
                    }
                } else {
                    // if not nearest neighbor, do bilinear interpolation
                    double resultVal = GetBilinearinterpolatedValue(inImg, posX, posY, width, height);

                    // set new rounded value for current location
                    resultImg[x][y] = (int) (resultVal + 0.5);
                }
            }
        }
        return resultImg;
    }

    public int calculateSSE(int[][] diffImg, int width, int height) {
        int sse = 0;

        for (int x = 0; x < width; x++) {
            for (int y = 0; y < height; y++) {
                sse = sse + diffImg[x][y];
            }
        }

        return sse;
    }

    public static double[][] chopImgInHalf(int[][] inDataArrInt, int width, int height, int widthHalf, boolean flag) {
        // store half of width in int var

        // create temporary image
        double[][] tmpImage = ImageJUtility.convertToDoubleArr2D(inDataArrInt, width, height);

        if (flag == true) {
            // create region of interest
            Rectangle roi = new Rectangle(0, 0, widthHalf, height);

            // crop image and store first half in var
            double[][] Img1 = ImageJUtility.cropImage(tmpImage, roi.width, roi.height, roi);
            ImageJUtility.showNewImage(Img1, widthHalf, height, "first half image");

            return Img1;
        } else {

            // create region of interest
            Rectangle roi = new Rectangle(0, 0, widthHalf, height);

            // overwrite roi with values for second half, crop image and store second half
            // in var
            roi = new Rectangle(widthHalf, 0, widthHalf, height);
            double[][] Img2 = ImageJUtility.cropImage(tmpImage, roi.width, roi.height, roi);
            ImageJUtility.showNewImage(Img2, widthHalf, height, "second half image");
            return Img2;
        }
    }

    public double GetBilinearinterpolatedValue(int[][] inImg, double x, double y, int width, int height) {
        // calculate the delta for x and y
        double deltaX = x - Math.floor(x);
        double deltaY = y - Math.floor(y);

        // set calculation fragment
        int xPlus1 = (int) x + 1;
        int yPlus1 = (int) y + 1;


        //handling translation and rotation for x and y
        if(x < 0 || x >= width || y < 0 || y >= height || xPlus1 < 0 || xPlus1 >= width || yPlus1 < 0 || yPlus1 >= height) {
            return 0;
        }

        // get 4 neighboring pixels
        int neighbor1 = inImg[xPlus1][(int) (y)];
        int neighbor2 = inImg[(int) (x)][yPlus1];
        int neighbor3 = inImg[xPlus1][yPlus1];
        int neighbor4 = inImg[(int) (x)][(int) (y)];

        // calculate weighted mean out of neighbors
        double weightedMean = ((1 - deltaX) * (1 - deltaY) * neighbor4) + (deltaX * (1 - deltaY) * neighbor1)
                + ((1 - deltaX) * deltaY * neighbor2) + (deltaX * deltaY * neighbor3);

        return weightedMean;
    }

} // class FilterTemplate_
```

Figure 1: Resampling anhand bilinearer Interpolation und Skalierung um Faktor 2.0