

Übungsaufgaben III, SBV1

Lukas Fiel, Lisa Panholzer

January 16, 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
1 import ij.*;
2 import ij.plugin.filter.PlugInFilter;
3 import ij.process.*;
4 import java.awt.Rectangle;
5 import java.awt.*;
6 import ij.gui.GenericDialog;

```

```

7
8 public class ChopImgInHalf_ implements PlugInFilter {
9
10
11     public int setup(String arg, ImagePlus imp) {
12         if (arg.equals("about"))
13             {showAbout(); return DONE;}
14         return DOES_SG+DOES_STACKS+SUPPORTS_MASKING;
15     } //setup
16
17
18
19     public void run(ImageProcessor ip) {
20         byte[] pixels = (byte[])ip.getPixels();
21         int width = ip.getWidth();
22         int height = ip.getHeight();
23         int [][] inDataArrInt = ImageJUtility.convertFrom1DByteArr(pixels, width, height)
24             ↪ ;
25
26         double widthHalf = width / 2.0;
27         double [][] tmpImage = ImageJUtility.convertToDoubleArr2D(inDataArrInt,
28             ↪ width, height);
29         Rectangle roi = new Rectangle(0, 0, (int)widthHalf, height);
30         double [][] Img1 = ImageJUtility.cropImage(tmpImage, roi.width, roi.
31             ↪ height, roi);
32         ImageJUtility.showNewImage(Img1, (int)widthHalf, height, "first_half_
33             ↪ image");
34         roi = new Rectangle((int)widthHalf, 0, (int)widthHalf, height);
35         double [][] Img2 = ImageJUtility.cropImage(tmpImage, roi.width, roi.
36             ↪ height, roi);
37         ImageJUtility.showNewImage(Img2, (int)widthHalf, height, "second_half_
38             ↪ image");
39
40     } //run
41
42     void showAbout() {
43         IJ.showMessage("About_Template...",
44             "this_is_a_PluginFilter_template\n");
45     } //showAbout
46
47 } //class FilterTemplate_

```

b) Transformation mittels Nearest Neighbor und Bilinearer Interpolation

In einem separaten Filter *RegisterFinal_.java* wurden die Funktionalitäten implementiert, die für die automatische Registrierung in c) benötigt werden. Bei Aufruf des Filters wird als erstes die Funktionalität des *ChopImgInHalf_.java* Filter angewandt. Diese sorgt dafür, dass das Eingangsbild in zwei Hälften geteilt und das Originalbild A von Bild B getrennt wird. Abschließend werden neben dem bereits geöffneten Originalbild die zwei getrennten Bildhälften ausgegeben. Die Neuberechnung der Bilder erfolgt entweder mit der Nearest Neighbor- oder der Bilinearen Interpolation. Hierzu wurde eine boolsche Variable als Flag definiert, die je nach Wert eine der beiden Interpolationsmethoden verwendet.

Anschließend müssen ΔX und ΔY für die Translation angegeben werden. Für die Rotation muss zusätzlich noch der Rotationswinkel angegeben werden. Nach dem der User die Eingaben getätigt und bestätigt hat, wird das Bild B dementsprechend neu positioniert und gedreht.

Anbei folgen Testbilder, die die Funktionalität widerspiegeln:








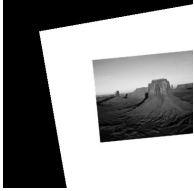







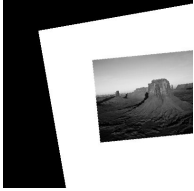
gegebenes Testbild	erste Bildhälfte	zweite Bildhälfte	resultierendes Ergebnis
 <p>deltaX=20, deltaY=20, rotation=45, nnFlag=false</p>			
 <p>deltaX=100, deltaY=50, rotation=10, nnFlag=false</p>			
 <p>deltaX=20, deltaY=20, rotation=45, nnFlag=true</p>			
 <p>deltaX=100, deltaY=50, rotation=10, nnFlag=true</p>			

Table 2: Testfälle:Transformation mittels NN und Bilinearer Interpolation

```

; columns
1
2 import ij.*;
3 import ij.plugin.filter.PlugInFilter;
4 import ij.process.*;
5 import java.awt.*;
6 import ij.gui.GenericDialog;
7
8 public class RegisterFinal_ implements PlugInFilter {
9
10     boolean nnFlag = false;
11
12     public int setup(String arg, ImagePlus imp) {
13         if (arg.equals("about"))
14             {showAbout(); return DONE;}
15         return DOES_8G+DOES_STACKS+SUPPORTS_MASKING;
16     } //setup
17
18
19
20     public void run(ImageProcessor ip) {
21         byte[] pixels = (byte[]) ip.getPixels();
22         int width = ip.getWidth();
23         int height = ip.getHeight();
24         int [][] inDataArrInt = ImageJUtility.convertFrom1DByteArr(pixels, width, height)
25             ↪ ;
26
27         int widthHalf = (int) (width / 2.0);
28         double [][] img1 = chopImgInHalf(inDataArrInt, width, height, widthHalf,
29             ↪ true);
30         double [][] img2 = chopImgInHalf(inDataArrInt, width, height, widthHalf,
31             ↪ false);
32
33         int [][] intImg2 = ImageJUtility.convertToIntArr2D(img2, widthHalf,
34             ↪ height);
35
36         // define transform
37         double transX = getUserInput(0,"deltaX");
38         double transY = getUserInput(0,"deltaY");
39         double rotAngle = getUserInput(0,"rotation");
40
41         //int [][] transformedImg = transformImage(inDataArrInt, width, height, transX,
42             ↪ transY, rotAngle);
43         int [][] transformedImg = transformImage(intImg2, widthHalf, height, transX,
44             ↪ transY, rotAngle);
45
46         ImageJUtility.showNewImage(transformedImg, widthHalf, height, "transformed_image
47             ↪ ");
48
49     } //run
50
51     void showAbout() {
52         IJ.showMessage("About_Template...",
53             "this_is_a_PluginFilter_template\n");
54     } //showAbout
55
56     public static int getUserInput(int defaultValue, String nameOfValue) {
57         // user input
58         System.out.print("Read_user_input:_ " + nameOfValue);
59         GenericDialog gd = new GenericDialog(nameOfValue);
60         gd.addNumericField("please_input_" + nameOfValue + ":", defaultValue,
61             ↪ 0);
62         gd.showDialog();
63         if (gd.wasCanceled()) {

```

```

56         return 0;
57     }
58     int radius = (int) gd.getNextNumber();
59     System.out.println(radius);
60     return radius;
61 }
62
63 public double GetBilinearinterpolatedValue(int [][] inImg, double x, double y,
64     ↪ int width, int height) {
65     // calculate the delta for x and y
66     double deltaX = x - Math.floor(x);
67     double deltaY = y - Math.floor(y);
68
69     // set calculation fragment
70     int xPlus1 = (int) x + 1;
71     int yPlus1 = (int) y + 1;
72
73     //handling translation and rotation for x and y
74     if(x < 0 || x >= width || y < 0 || y >= height || xPlus1 < 0 || xPlus1
75     ↪ >= width || yPlus1 < 0 || yPlus1 >= height) {
76         return 0;
77     }
78
79     // get 4 neighboring pixels
80     int neighbor1 = inImg[xPlus1][(int) (y)];
81     int neighbor2 = inImg[(int) (x)][yPlus1];
82     int neighbor3 = inImg[xPlus1][yPlus1];
83     int neighbor4 = inImg[(int) (x)][(int) (y)];
84
85     // calculate weighted mean out of neighbors
86     double weightedMean = ((1 - deltaX) * (1 - deltaY) * neighbor4) + (
87     ↪ deltaX * (1 - deltaY) * neighbor1)
88     ↪ + ((1 - deltaX) * deltaY * neighbor2) + (deltaX * deltaY
89     ↪ * neighbor3);
90
91     return weightedMean;
92 }
93
94 public int [][] transformImage(int [][] inImg, int width, int height, double transX
95     ↪ , double transY, double rotAngle) {
96
97     //allocate result image
98     int [][] resultImg = new int[width][height];
99
100     // prepare cos theta, sin theta
101     double cosTheta = Math.cos(Math.toRadians(-rotAngle));
102     double sinTheta = Math.sin(Math.toRadians(-rotAngle)); // - weil
103     ↪ backgroundmapping
104
105     double widthHalf = width / 2.0;
106     double heightHalf = height / 2.0;
107
108     //1) iterate over all pixels and calc value utilizing backward-mapping
109     for( int x= 0; x < width; x++) {
110         for (int y =0; y< height; y++) {
111
112             double tmpposX = x - widthHalf;
113             double tmpposY = y - heightHalf;
114
115             //3) rotate
116             double posX = tmpposX * cosTheta + tmpposY * sinTheta;
117             double posY = - tmpposX * sinTheta + tmpposY * cosTheta;

```

```

114
115
116         //4) translate
117         posX -= transX;
118         posY -= transY;
119
120
121         // move origin back from center to top corner
122         posX = posX + widthHalf;
123         posY = posY + heightHalf;
124
125         //6) get interpolated value if flag is true
126         if (nnFlag) {
127             int nnX = (int) (posX + 0.5);
128             int nnY = (int) (posY + 0.5);
129
130             //6) assigne value from original img inImg if
131             //    ↪ inside the image boundaries
132             if (nnX >= 0 && nnX < width && nnY >= 0 && nnY <
133                 ↪ height) {
134                 resultImg[x][y] = inImg[nnX][nnY];
135             }
136         }
137         else {
138             // if nearest neighbor flag is false, do
139             //    ↪ bilinear interpolation
140             double resultVal = GetBilinearinterpolatedValue(
141                 ↪ inImg, posX, posY, width, height);
142
143             //set new rounded value for current location
144             resultImg[x][y] = (int) (resultVal + 0.5);
145         }
146     }
147 }
148
149
150 public static double[][] chopImgInHalf(int[][] inDataArrInt, int width, int
151     ↪ height, int widthHalf, boolean flag) {
152     // store half of width in int var
153
154     // create temporary image
155     double[][] tmpImage = ImageJUtility.convertToDoubleArr2D(inDataArrInt,
156         ↪ width, height);
157
158     if (flag == true) {
159         // create region of interest
160         Rectangle roi = new Rectangle(0, 0, widthHalf, height);
161
162         // crop image and store first half in var
163         double[][] Img1 = ImageJUtility.cropImage(tmpImage, roi.width,
164             ↪ roi.height, roi);
165         ImageJUtility.showNewImage(Img1, widthHalf, height, "first_half_
166             ↪ image");
167
168         return Img1;
169     } else {
170         // create region of interest
171         Rectangle roi = new Rectangle(0, 0, widthHalf, height);
172
173         // overwrite roi with values for second half, crop image and

```



```

171 |                                     ↪ store second half
172 | // in var
173 | roi = new Rectangle(widthHalf, 0, widthHalf, height);
174 | double[][] Img2 = ImageJUtility.cropImage(tmpImage, roi.width,
175 |                                     ↪ roi.height, roi);
176 | ImageJUtility.showNewImage(Img2, widthHalf, height, "second_half
177 |                                     ↪ _image");
178 | return Img2;
179 | }
180 | } //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 Richtung und rotiert dieses auch um jeweils ein Inkrement. Jedes dieser transformierten Bilder wird nun von Bild2 abgezogen und erneut ein Fitnesswert 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, weshalb 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/Ver-

schiebungen mit dem NearesNeighbor Algorithmus berechnet. Ist das Optimum gefunden wird anschließend noch einmal 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 veranschaulichen.



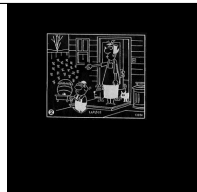


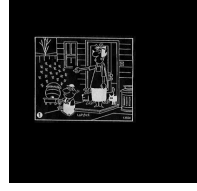
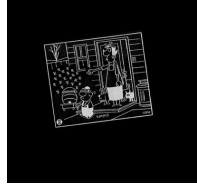
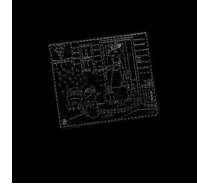


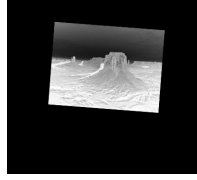
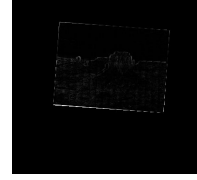
gegebenes Testbild	invertierter Bildausschnitt1	invertierter Bildausschnitt2	resultierendes Differenzbild
			
			
			

Table 3: Testfälle: automatische Registrierung

```

; columns
1
2 import ij.*;
3 import ij.plugin.filter.PlugInFilter;
4 import ij.process.*;
5 import java.awt.*;
6
7 public class AutoRegisterFinal_ implements PlugInFilter {
8
9     public int setup(String arg, ImagePlus imp) {
10         if (arg.equals("about")) {
11             showAbout();
12             return DONE;
13         }
14         return DOES_8G + DOES_STACKS + SUPPORTS_MASKING;
15     } // setup
16

```

```

17 public void run(ImageProcessor ip) {
18     // read image
19     byte[] pixels = (byte[]) ip.getPixels();
20     int width = ip.getWidth();
21     int height = ip.getHeight();
22     int[][] inDataArrInt = ImageJUtility.convertFrom1DByteArr(pixels, width,
23         ↪ height);
24
25     // invert to set background to black
26     int[] invertTF = ImageTransformationFilter.GetInversionTF(255);
27     inDataArrInt = ImageTransformationFilter.GetTransformedImage(
28         ↪ inDataArrInt, width, height, invertTF);
29
30     int widthHalf = (int) (width / 2.0);
31     double[][] img1 = chopImgInHalf(inDataArrInt, width, height, widthHalf,
32         ↪ true);
33     double[][] img2 = chopImgInHalf(inDataArrInt, width, height, widthHalf,
34         ↪ false);
35
36     // initialize ranges
37     int xRadius = 20;
38     int yRadius = 20;
39     int rotRadius = 20;
40
41     // initialize arrays
42     int[][] intImg1 = ImageJUtility.convertToIntArr2D(img1, widthHalf,
43         ↪ height);
44     int[][] intImg2 = ImageJUtility.convertToIntArr2D(img2, widthHalf,
45         ↪ height);
46     int[][] transformedImg;
47     int[][] diffImg;
48     int[][][] ssE = new int[2 * xRadius + 1][2 * yRadius + 1][2 * rotRadius
49         ↪ + 1];
50
51     // initial fitness
52     diffImg = ImageJUtility.calculateImgDifference(intImg1, intImg2,
53         ↪ widthHalf, height);
54     int initialFitness = calculateSSE(diffImg, widthHalf, height);
55     System.out.println("initiale_Fitness:~" + initialFitness);
56
57     // fill ssE matrix and find minimum
58     int minimum = initialFitness;
59     int tmpSSE = 0;
60     int minXind = 0;
61     int minYind = 0;
62     int minAngleInd = 0;
63     for (int x = -xRadius; x < xRadius; x++) {
64         for (int y = -yRadius; y < yRadius; y++) {
65             for (int angle = -rotRadius; angle < rotRadius; angle++)
66                 ↪ {
67                 transformedImg = transformImage(intImg1,
68                     ↪ widthHalf, height, x, y, angle, false);
69                 diffImg = ImageJUtility.calculateImgDifference(
70                     ↪ transformedImg, intImg2, widthHalf,
71                     ↪ height);
72                 tmpSSE = calculateSSE(diffImg, widthHalf, height
73                     ↪ );
74                 ssE[x + xRadius][y + yRadius][angle + rotRadius]
75                     ↪ = tmpSSE;
76
77                 // find minimum and save indices for later
78                 if (tmpSSE < minimum) {
79                     minimum = tmpSSE;
80                     //System.out.println("current minimal
81                         ↪ fitness: " + minimum);
82                     minXind = x;

```

```

68         minYind = y;
69         minAngleInd = angle;
70     }
71 }
72 }
73 }
74 System.out.println("final_Fitness:_"+ minimum);
75 System.out.println("minXind:"+minXind+"minYind:"+minYind+"minAngleInd:"+
    ↪ minAngleInd);
76
77 minXind = 22;
78 minYind = -6;
79 minAngleInd = -4;
80
81 // plot difference image to proof the transformation
82 transformedImg = transformImage(intImg1, widthHalf, height, minXind,
    ↪ minYind, minAngleInd, true);
83 diffImg = ImageJUtility.calculateImgDifference(transformedImg, intImg2,
    ↪ widthHalf, height);
84 ImageJUtility.showNewImage(diffImg, widthHalf, height, "fittest_diff_
    ↪ image");
85
86 } // run
87
88 void showAbout() {
89     IJ.showMessage("About_Template...", "this_is_a_PluginFilter_template\n"
    ↪ );
90 } // showAbout
91
92 public int[][] transformImage(int[][] inImg, int width, int height, double
    ↪ transX, double transY, double rotAngle,
93     boolean interpolation) {
94
95     // allocate result image
96     int[][] resultImg = new int[width][height];
97
98     // prepare cos theta, sin theta
99     double cosTheta = Math.cos(Math.toRadians(-rotAngle));
100    double sinTheta = Math.sin(Math.toRadians(-rotAngle)); // - weil
    ↪ backgroundmapping
101
102    double widthHalf = width / 2.0;
103    double heightHalf = height / 2.0;
104
105    // 1) iterate over all pixels and calc value utilizing backward-mapping
106    for (int x = 0; x < width; x++) {
107        for (int y = 0; y < height; y++) {
108
109            double tmpposX = x - widthHalf;
110            double tmpposY = y - heightHalf;
111
112            // 3) rotate
113            double posX = tmpposX * cosTheta + tmpposY * sinTheta;
114            double posY = -tmpposX * sinTheta + tmpposY * cosTheta;
115
116            // 4) translate
117            posX -= transX;
118            posY -= transY;
119
120            // move origin back from center to top corner
121            posX = posX + widthHalf;
122            posY = posY + heightHalf;
123
124            // 6) assigne value from original imag inImg if inside
    ↪ the image boundaries

```

```

125         // get interpolated value if flag is true
126         if (interpolation == false) {
127             int nnX = (int) (posX + 0.5);
128             int nnY = (int) (posY + 0.5);
129
130             // 6) assign value from original img inImg if
131             //     ↪ inside the image boundaries
132             if (nnX >= 0 && nnX < width && nnY >= 0 && nnY <
133                 ↪ height) {
134                 resultImg[x][y] = inImg[nnX][nnY];
135             }
136         } else {
137             // if not nearest neighbor, do bilinear
138             //     ↪ interpolation
139             double resultVal = GetBilinearinterpolatedValue(
140                 ↪ inImg, posX, posY, width, height);
141
142             // set new rounded value for current location
143             resultImg[x][y] = (int) (resultVal + 0.5);
144         }
145     }
146     return resultImg;
147 }
148
149 public int calculateSSE(int [][] diffImg, int width, int height) {
150     int sse = 0;
151
152     for (int x = 0; x < width; x++) {
153         for (int y = 0; y < height; y++) {
154             sse = sse + diffImg[x][y];
155         }
156     }
157
158     return sse;
159 }
160
161 public static double [][] chopImgInHalf(int [][] inDataArrInt, int width, int
162     ↪ height, int widthHalf, boolean flag) {
163     // store half of width in int var
164
165     // create temporary image
166     double [][] tmpImage = ImageJUtility.convertToDoubleArr2D(inDataArrInt,
167         ↪ width, height);
168
169     if (flag == true) {
170         // create region of interest
171         Rectangle roi = new Rectangle(0, 0, widthHalf, height);
172
173         // crop image and store first half in var
174         double [][] Img1 = ImageJUtility.cropImage(tmpImage, roi.width,
175             ↪ roi.height, roi);
176         ImageJUtility.showNewImage(Img1, widthHalf, height, "first_half_
177             ↪ image");
178
179         return Img1;
180     } else {
181         // create region of interest
182         Rectangle roi = new Rectangle(0, 0, widthHalf, height);
183
184         // overwrite roi with values for second half, crop image and
185         //     ↪ store second half
186         // in var
187         roi = new Rectangle(widthHalf, 0, widthHalf, height);

```

```

181         double[][] Img2 = ImageJUtility.cropImage(tmpImage, roi.width,
182             ↪ roi.height, roi);
183         ImageJUtility.showNewImage(Img2, widthHalf, height, "second_half
184             ↪ _image");
185         return Img2;
186     }
187     public double GetBilinearinterpolatedValue(int[][] inImg, double x, double y,
188         ↪ int width, int height) {
189         // calculate the delta for x and y
190         double deltaX = x - Math.floor(x);
191         double deltaY = y - Math.floor(y);
192
193         // set calculation fragment
194         int xPlus1 = (int) x + 1;
195         int yPlus1 = (int) y + 1;
196
197         //handling translation and rotation for x and y
198         if(x < 0 || x >= width || y < 0 || y >= height || xPlus1 < 0 || xPlus1
199             ↪ >= width || yPlus1 < 0 || yPlus1 >= height) {
200             return 0;
201         }
202
203         // get 4 neighboring pixels
204         int neighbor1 = inImg[xPlus1][(int) (y)];
205         int neighbor2 = inImg[(int) (x)][yPlus1];
206         int neighbor3 = inImg[xPlus1][yPlus1];
207         int neighbor4 = inImg[(int) (x)][(int) (y)];
208
209         // calculate weighted mean out of neighbors
210         double weightedMean = ((1 - deltaX) * (1 - deltaY) * neighbor4) + (
211             ↪ deltaX * (1 - deltaY) * neighbor1)
212             + ((1 - deltaX) * deltaY * neighbor2) + (deltaX * deltaY
213             ↪ * neighbor3);
214
215         return weightedMean;
216     }
217 } // class FilterTemplate_

```