

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 wird neben dem bereits geöffnet 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:












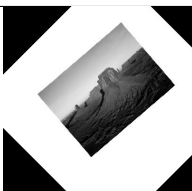




gegebenes Testbild	erste Bildhälfte	zweite Bildhälfte	resultierendes Ergebnis
			
deltaX=20, deltaY=20, rotation=45, nnFlag=false			
 deltaX=100, deltaY=50, rotation=10, nnFlag=false			
			
deltaX=20, deltaY=20, rotation=45, nnFlag=true			
 deltaX=100, deltaY=50, rotation=10, nnFlag=true			

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()) {
64             return 0;
65         }
66         int radius = (int) gd.getNextNumber();
67         System.out.println(radius);
68         return radius;
69     }
70
71     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
        ↪ >= width || yPlus1 < 0 || yPlus1 >= height) {
75         return 0;
76     }
77
78     // get 4 neighboring pixels
79     int neighbor1 = inImg[xPlus1][(int) (y)];
80     int neighbor2 = inImg[(int) (x)][yPlus1];
81     int neighbor3 = inImg[xPlus1][yPlus1];
82     int neighbor4 = inImg[(int) (x)][(int) (y)];
83
84     // calculate weighted mean out of neighbors
85     double weightedMean = ((1 - deltaX) * (1 - deltaY) * neighbor4) + (
        ↪ deltaX * (1 - deltaY) * neighbor1)
86         + ((1 - deltaX) * deltaY * neighbor2) + (deltaX * deltaY
        ↪ * neighbor3);
87
88     return weightedMean;
89 }
90
91 public int [][] transformImage(int [][] inImg, int width, int height, double transX
    ↪ , double transY, double rotAngle) {
92
93     //allocate result image
94     int [][] resultImg = new int[width][height];
95
96     // prepare cos theta, sin theta
97     double cosTheta = Math.cos(Math.toRadians(-rotAngle));
98     double sinTheta = Math.sin(Math.toRadians(-rotAngle)); // - weil
        ↪ backgroundmapping
99
100     double widthHalf = width / 2.0;
101     double heightHalf = height / 2.0;
102
103
104     //1) iterate over all pixels and calc value utilizing backward-mapping
105     for( int x= 0; x < width; x++) {
106         for (int y =0; y< height; y++) {
107
108             double tmpposX = x - widthHalf;
109             double tmpposY = y - heightHalf;
110
111             //3) rotate
112             double posX = tmpposX * cosTheta + tmpposY * sinTheta;
113             double posY = - tmpposX * sinTheta + tmpposY * cosTheta;
114
115             //4) translate
116             posX -= transX;
117             posY -= transY;
118
119
120             // move origin back from center to top corner
121             posX = posX + widthHalf;

```

```

122         posY = posY + heightHalf;
123
124         //6) get interpolated value if flag is true
125         if (nnFlag) {
126             int nnX = (int) (posX + 0.5);
127             int nnY = (int) (posY + 0.5);
128
129             //6) assigne value from original img inImg if
130             ↪ inside the image boundaries
131             if (nnX >= 0 && nnX < width && nnY >= 0 && nnY <
132             ↪ height) {
133                 resultImg[x][y] = inImg[nnX][nnY];
134             }
135         }
136         else {
137             // if nearest neighbor flag is false, do
138             ↪ bilinear 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 static double[][] chopImgInHalf(int[][] inDataArrInt, int width, int
150 ↪ height, int widthHalf, boolean flag) {
151     // store half of width in int var
152
153     // create temporary image
154     double[][] tmpImage = ImageJUtility.convertToDoubleArr2D(inDataArrInt,
155     ↪ width, height);
156
157     if (flag == true) {
158         // create region of interest
159         Rectangle roi = new Rectangle(0, 0, widthHalf, height);
160
161         // crop image and store first half in var
162         double[][] Img1 = ImageJUtility.cropImage(tmpImage, roi.width,
163         ↪ roi.height, roi);
164         ImageJUtility.showNewImage(Img1, widthHalf, height, "first_half_
165         ↪ image");
166
167         return Img1;
168     } else {
169
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
174         ↪ store second half
175         // in var
176         roi = new Rectangle(widthHalf, 0, widthHalf, height);
177         double[][] Img2 = ImageJUtility.cropImage(tmpImage, roi.width,
178         ↪ roi.height, roi);
179         ImageJUtility.showNewImage(Img2, widthHalf, height, "second_half_
180         ↪ _image");
181         return Img2;

```



```

176 |         }
177 |     }
178 |
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/Verschiebungen 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.











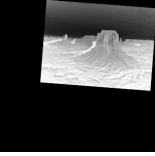
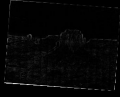
gegebenes Testbild	invertierter Bildausschnitt1	invertierter Bildausschnitt2	resultierendes Differenzbild
			
			
			

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_SG + 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,
            ↪ height);

```

```

23 // invert to set background to black
24
25 int[] invertTF = ImageTransformationFilter.GetInversionTF(255);
26
27 inDataArrInt = ImageTransformationFilter.GetTransformedImage(
28     ↪ inDataArrInt, width, height, invertTF);
29
30
31 int widthHalf = (int) (width / 2.0);
32
33 double[][] img1 = chopImgInHalf(inDataArrInt, width, height, widthHalf,
34     ↪ true);
35
36 double[][] img2 = chopImgInHalf(inDataArrInt, width, height, widthHalf,
37     ↪ false);
38
39
40 // initialize ranges
41
42 int xRadius = 20;
43
44 int yRadius = 20;
45
46 int rotRadius = 20;
47
48
49 // initialize arrays
50
51 int[][] intImg1 = ImageJUtility.convertToIntArr2D(img1, widthHalf,
52     ↪ height);
53
54 int[][] intImg2 = ImageJUtility.convertToIntArr2D(img2, widthHalf,
55     ↪ height);
56
57 int[][] transformedImg;
58
59 int[][] diffImg;
60
61 int[][][] ssE = new int[2 * xRadius + 1][2 * yRadius + 1][2 * rotRadius
62     ↪ + 1];
63
64
65 // initial fitness
66
67 diffImg = ImageJUtility.calculateImgDifference(intImg1, intImg2,
68     ↪ widthHalf, height);
69
70 int initialFitness = calculateSSE(diffImg, widthHalf, height);
71
72 System.out.println("initiale_Fitness:_" + initialFitness);
73
74
75 // fill ssE matrix and find minimum
76
77 int minimum = initialFitness;
78
79 int tmpSSE = 0;
80
81 int minXind = 0;
82
83 int minYind = 0;
84
85 int minAngleInd = 0;
86
87 for (int x = -xRadius; x < xRadius; x++) {
88     for (int y = -yRadius; y < yRadius; y++) {
89         for (int angle = -rotRadius; angle < rotRadius; angle++)
90             ↪ {
91                 transformedImg = transformImage(intImg1,
92                     ↪ widthHalf, height, x, y, angle, false);
93                 diffImg = ImageJUtility.calculateImgDifference(
94                     ↪ transformedImg, intImg2, widthHalf,
95                     ↪ height);
96                 tmpSSE = calculateSSE(diffImg, widthHalf, height
97                     ↪ );
98                 ssE[x + xRadius][y + yRadius][angle + rotRadius]
99                     ↪ = tmpSSE;
100
101                 // find minimum and save indices for later
102                 if (tmpSSE < minimum) {
103                     minimum = tmpSSE;
104                     //System.out.println("current minimal
105                         ↪ fitness: " + minimum);
106                     minXind = x;
107                     minYind = y;
108                     minAngleInd = angle;
109                 }
110             }
111         }
112     }
113
114 System.out.println("final_Fitness:_" + minimum);

```

```

75         System.out.println("minXind:"+minXind+"minYind:"+minYind+"minAngleInd:"+
76             ↪ minAngleInd);
77
78         minXind = 22;
79         minYind = -6;
80         minAngleInd = -4;
81
82         // plot difference image to proof the transformation
83         transformedImg = transformImage(intImg1, widthHalf, height, minXind,
84             ↪ minYind, minAngleInd, true);
85         diffImg = ImageJUtility.calculateImgDifference(transformedImg, intImg2,
86             ↪ widthHalf, height);
87         ImageJUtility.showNewImage(diffImg, widthHalf, height, "fittest_diff_
88             ↪ image");
89
90     } // run
91
92     void showAbout() {
93         IJ.showMessage("About_Template...", "this_is_a_PluginFilter_template\n"
94             ↪ );
95     } // showAbout
96
97     public int[][] transformImage(int[][] inImg, int width, int height, double
98         ↪ transX, double transY, double rotAngle,
99         boolean interpolation) {
100
101         // allocate result image
102         int[][] resultImg = new int[width][height];
103
104         // prepare cos theta, sin theta
105         double cosTheta = Math.cos(Math.toRadians(-rotAngle));
106         double sinTheta = Math.sin(Math.toRadians(-rotAngle)); // - weil
107             ↪ backgroundmapping
108
109         double widthHalf = width / 2.0;
110         double heightHalf = height / 2.0;
111
112         // 1) iterate over all pixels and calc value utilizing backward-mapping
113         for (int x = 0; x < width; x++) {
114             for (int y = 0; y < height; y++) {
115
116                 double tmpposX = x - widthHalf;
117                 double tmpposY = y - heightHalf;
118
119                 // 3) rotate
120                 double posX = tmpposX * cosTheta + tmpposY * sinTheta;
121                 double posY = -tmpposX * sinTheta + tmpposY * cosTheta;
122
123                 // 4) translate
124                 posX -= transX;
125                 posY -= transY;
126
127                 // move origin back from center to top corner
128                 posX = posX + widthHalf;
129                 posY = posY + heightHalf;
130
131                 // 6) assigne value from original imag inImg if inside
132                 ↪ the image boundaries
133                 // get interpolated value if flag is true
134                 if (interpolation == false) {
135                     int nnX = (int) (posX + 0.5);
136                     int nnY = (int) (posY + 0.5);
137
138                     // 6) assign value from original img inImg if
139                     ↪ inside the image boundaries

```

```

131         if (nnX >= 0 && nnX < width && nnY >= 0 && nnY <
132             ↪ height) {
133             resultImg[x][y] = inImg[nnX][nnY];
134         }
135     } else {
136         // if not nearest neighbor, do bilinear
137         ↪ interpolation
138         double resultVal = GetBilinearinterpolatedValue(
139             ↪ inImg, posX, posY, width, height);
140
141         // set new rounded value for current location
142         resultImg[x][y] = (int) (resultVal + 0.5);
143     }
144 }
145
146 public int calculateSSE(int [][] diffImg, int width, int height) {
147     int sse = 0;
148
149     for (int x = 0; x < width; x++) {
150         for (int y = 0; y < height; y++) {
151             sse = sse + diffImg[x][y];
152         }
153     }
154
155     return sse;
156 }
157
158 public static double [][] chopImgInHalf(int [][] inDataArrInt, int width, int
159     ↪ height, int widthHalf, boolean flag) {
160     // store half of width in int var
161
162     // create temporary image
163     double [][] tmpImage = ImageJUtility.convertToDoubleArr2D(inDataArrInt,
164         ↪ width, height);
165
166     if (flag == true) {
167         // create region of interest
168         Rectangle roi = new Rectangle(0, 0, widthHalf, height);
169
170         // crop image and store first half in var
171         double [][] Img1 = ImageJUtility.cropImage(tmpImage, roi.width,
172             ↪ roi.height, roi);
173         ImageJUtility.showNewImage(Img1, widthHalf, height, "first_half_
174             ↪ image");
175
176         return Img1;
177     } else {
178
179         // create region of interest
180         Rectangle roi = new Rectangle(0, 0, widthHalf, height);
181
182         // overwrite roi with values for second half, crop image and
183         ↪ store second half
184         // in var
185         roi = new Rectangle(widthHalf, 0, widthHalf, height);
186         double [][] Img2 = ImageJUtility.cropImage(tmpImage, roi.width,
187             ↪ roi.height, roi);
188         ImageJUtility.showNewImage(Img2, widthHalf, height, "second_half
189             ↪ _image");
190         return Img2;
191     }
192 }

```

```

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_

```