

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
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

```

; 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_SG+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.convertFrom1DByteArray(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,
72         ↪ int width, int height) {
73         // calculate the delta for x and y
74         double deltaX = x - Math.floor(x);
75         double deltaY = y - Math.floor(y);
76
77         // set calculation fragment
78         int xPlus1 = (int) x + 1;
79         int yPlus1 = (int) y + 1;

```

```

72 //handling translation and rotation for x and y
73 if(x < 0 || x >= width || y < 0 || y >= height || xPlus1 < 0 || xPlus1
74    ↪ >= 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) + (
86    ↪ deltaX * (1 - deltaY) * neighbor1
87    ↪ + ((1 - deltaX) * deltaY * neighbor2) + (deltaX * deltaY
88    ↪ * neighbor3);
89
90 return weightedMean;
91 }
92
93 public int [][] transformImage(int [][] inImg, int width, int height, double transX
94    ↪ , double transY, double rotAngle) {
95
96     //allocate result image
97     int [][] resultImg = new int[width][height];
98
99     // prepare cos theta, sin theta
100     double cosTheta = Math.cos(Math.toRadians(-rotAngle));
101     double sinTheta = Math.sin(Math.toRadians(-rotAngle)); // - weil
102     ↪ backgroundmapping
103
104     double widthHalf = width / 2.0;
105     double heightHalf = height / 2.0;
106
107     //1) iterate over all pixels and calc value utilizing backward-mapping
108     for( int x= 0; x < width; x++) {
109         for (int y =0; y< height; y++) {
110
111             double tmpposX = x - widthHalf;
112             double tmpposY = y - heightHalf;
113
114             //3) rotate
115             double posX = tmpposX * cosTheta + tmpposY * sinTheta;
116             double posY = - tmpposX * sinTheta + tmpposY * cosTheta;
117
118             //4) translate
119             posX -= transX;
120             posY -= transY;
121
122             // move origin back from center to top corner
123             posX = posX + widthHalf;
124             posY = posY + heightHalf;
125
126             //6) get interpolated value if flag is true
127             if (nnFlag) {
128                 int nnX = (int) (posX + 0.5);
129                 int nnY = (int) (posY + 0.5);
130
131                 //6) assigne value from original img inImg if
132                 ↪ inside the image boundaries

```

```

130         if(nnX >= 0 && nnX < width && nnY >= 0 && nnY <
131             ↪ height) {
132             resultImg[x][y] = inImg[nnX][nnY];
133         }
134     } else {
135         // if nearest neighbor flag is false, do
136         ↪ bilinear interpolation
137         double resultVal = GetBilinearinterpolatedValue(
138             ↪ inImg, posX, posY, width, height);
139
140         //set new rounded value for current location
141         resultImg[x][y] = (int) (resultVal + 0.5);
142     }
143 }
144 }
145 }
146 return resultImg;
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
171         // create region of interest
172         Rectangle roi = new Rectangle(0, 0, widthHalf, height);
173
174         // overwrite roi with values for second half, crop image and
175         ↪ store second half
176         // in var
177         roi = new Rectangle(widthHalf, 0, widthHalf, height);
178         double[][] Img2 = ImageJUtility.cropImage(tmpImage, roi.width,
179             ↪ roi.height, roi);
180         ImageJUtility.showNewImage(Img2, widthHalf, height, "second_half
181             ↪ _image");
182         return Img2;
183     }
184 }
185 }
186 } //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 Bilinear Interpolation berechnet und von Bild2 subtrahiert. Das Resultatbild wird zum Schluss für den User sichtbar dargestellt um den Erfolg des Filters zu veranschaulichen.











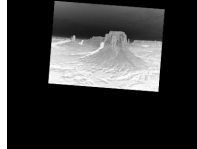
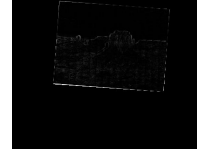
gegebenes Testbild	invertierter Bildausschnitt1	invertierter Bildausschnitt2	resultierendes Differenzbild
			
			
			

Table 2: 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);

```



```

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     // initialize ranges
34     int xRadius = 20;
35     int yRadius = 20;
36     int rotRadius = 20;
37
38     // initialize arrays
39     int[][] intImg1 = ImageJUtility.convertToIntArr2D(img1, widthHalf,
40         ↪ height);
41     int[][] intImg2 = ImageJUtility.convertToIntArr2D(img2, widthHalf,
42         ↪ height);
43     int[][] transformedImg;
44     int[][] diffImg;
45     int[][] ssE = new int[2 * xRadius + 1][2 * yRadius + 1][2 * rotRadius
46         ↪ + 1];
47
48     // initial fitness
49     diffImg = ImageJUtility.calculateImgDifference(intImg1, intImg2,
50         ↪ widthHalf, height);
51     int initialFitness = calculateSSE(diffImg, widthHalf, height);
52     System.out.println("initiale_Fitness:_" + initialFitness);
53
54     // fill ssE matrix and find minimum
55     int minimum = initialFitness;
56     int tmpSSE = 0;
57     int minXind = 0;
58     int minYind = 0;
59     int minAngleInd = 0;
60     for (int x = -xRadius; x < xRadius; x++) {
61         for (int y = -yRadius; y < yRadius; y++) {
62             for (int angle = -rotRadius; angle < rotRadius; angle++)
63                 ↪ {
64                 transformedImg = transformImage(intImg1,
65                     ↪ widthHalf, height, x, y, angle, false);
66                 diffImg = ImageJUtility.calculateImgDifference(
67                     ↪ transformedImg, intImg2, widthHalf,
68                     ↪ height);
69                 tmpSSE = calculateSSE(diffImg, widthHalf, height
70                     ↪ );
71                 ssE[x + xRadius][y + yRadius][angle + rotRadius]
72                     ↪ = tmpSSE;
73
74                 // find minimum and save indices for later
75                 if (tmpSSE < minimum) {
76                     minimum = tmpSSE;
77                     //System.out.println("current minimal
78                     ↪ fitness: " + minimum);
79                     minXind = x;
80                     minYind = y;
81                     minAngleInd = angle;
82                 }
83             }
84         }
85     }
86
87     System.out.println("final_Fitness:_" + minimum);
88     System.out.println("minXind:" + minXind + "minYind:" + minYind + "minAngleInd:" +
89         ↪ minAngleInd);
90
91     minXind = 22;
92     minYind = -6;

```

```

79         minAngleInd = -4;
80
81         // plot difference image to proof the transformation
82         transformedImg = transformImage(intImg1, widthHalf, height, minXind,
83             ↪ minYind, minAngleInd, true);
84         diffImg = ImageJUtility.calculateImgDifference(transformedImg, intImg2,
85             ↪ widthHalf, height);
86         ImageJUtility.showNewImage(diffImg, widthHalf, height, "fittest_diff_
87             ↪ image");
88     } // run
89
90     void showAbout() {
91         IJ.showMessage("About_Template...", "this_is_a_PluginFilter_template\n"
92             ↪ );
93     } // showAbout
94
95     public int[][] transformImage(int[][] inImg, int width, int height, double
96         ↪ transX, double transY, double rotAngle,
97         ↪ boolean interpolation) {
98
99         // allocate result image
100         int[][] resultImg = new int[width][height];
101
102         // prepare cos theta, sin theta
103         double cosTheta = Math.cos(Math.toRadians(-rotAngle));
104         double sinTheta = Math.sin(Math.toRadians(-rotAngle)); // - weil
105             ↪ backgroundmapping
106
107         double widthHalf = width / 2.0;
108         double heightHalf = height / 2.0;
109
110         // 1) iterate over all pixels and calc value utilizing backward-mapping
111         for (int x = 0; x < width; x++) {
112             for (int y = 0; y < height; y++) {
113
114                 double tmpposX = x - widthHalf;
115                 double tmpposY = y - heightHalf;
116
117                 // 3) rotate
118                 double posX = tmpposX * cosTheta + tmpposY * sinTheta;
119                 double posY = -tmpposX * sinTheta + tmpposY * cosTheta;
120
121                 // 4) translate
122                 posX -= transX;
123                 posY -= transY;
124
125                 // move origin back from center to top corner
126                 posX = posX + widthHalf;
127                 posY = posY + heightHalf;
128
129                 // 6) assigne value from original imag inImg if inside
130                 ↪ the image boundaries
131                 // get interpolated value if flag is true
132                 if (interpolation == false) {
133                     int nnX = (int) (posX + 0.5);
134                     int nnY = (int) (posY + 0.5);
135
136                     // 6) assign value from original img inImg if
137                     ↪ inside the image boundaries
138                     if (nnX >= 0 && nnX < width && nnY >= 0 && nnY <
139                         ↪ height) {
140                         resultImg[x][y] = inImg[nnX][nnY];
141                     }
142                 } else {

```

```

135 // if not nearest neighbor, do bilinear
136 ↪ interpolation
137 double resultVal = GetBilinearinterpolatedValue(
138 ↪ inImg, posX, posY, width, height);
139 // set new rounded value for current location
140 resultImg[x][y] = (int) (resultVal + 0.5);
141 }
142 }
143 return resultImg;
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         // create region of interest
179         Rectangle roi = new Rectangle(0, 0, widthHalf, height);
180
181         // overwrite roi with values for second half, crop image and
182 ↪ store second half
183         // in var
184         roi = new Rectangle(widthHalf, 0, widthHalf, height);
185         double [][] Img2 = ImageJUtility.cropImage(tmpImage, roi.width,
186 ↪ roi.height, roi);
187         ImageJUtility.showNewImage(Img2, widthHalf, height, "second_half
188 ↪ _image");
189         return Img2;
190     }
191 }
192
193 public double GetBilinearinterpolatedValue(int [][] inImg, double x, double y,
194 ↪ int width, int height) {
195     // calculate the delta for x and y
196     double deltaX = x - Math.floor(x);

```

```

190         double deltaY = y - Math.floor(y);
191
192         // set calculation fragment
193         int xPlus1 = (int) x + 1;
194         int yPlus1 = (int) y + 1;
195
196
197         //handling translation and rotation for x and y
198         if(x < 0 || x >= width || y < 0 || y >= height || xPlus1 < 0 || xPlus1
            ↪ >= width || yPlus1 < 0 || yPlus1 >= height) {
199             return 0;
200         }
201
202         // get 4 neighboring pixels
203         int neighbor1 = inImg[xPlus1][(int) (y)];
204         int neighbor2 = inImg[(int) (x)][yPlus1];
205         int neighbor3 = inImg[xPlus1][yPlus1];
206         int neighbor4 = inImg[(int) (x)][(int) (y)];
207
208         // calculate weighted mean out of neighbors
209         double weightedMean = ((1 - deltaX) * (1 - deltaY) * neighbor4) + (
210             ↪ deltaX * (1 - deltaY) * neighbor1)
                + ((1 - deltaX) * deltaY * neighbor2) + (deltaX * deltaY
                ↪ * neighbor3);
211
212         return weightedMean;
213     }
214
215 } // class FilterTemplate_

```

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