



UNIVERSITY
OF APPLIED SCIENCES
UPPER AUSTRIA

FH HAGENBERG

Signal- und Bildbearbeitung Uebung 01b

Julian Buchgeher/Julian Kohr

S2210745004/S2210745012

Übungsaufgaben I(b), SBV1

21.10.2022

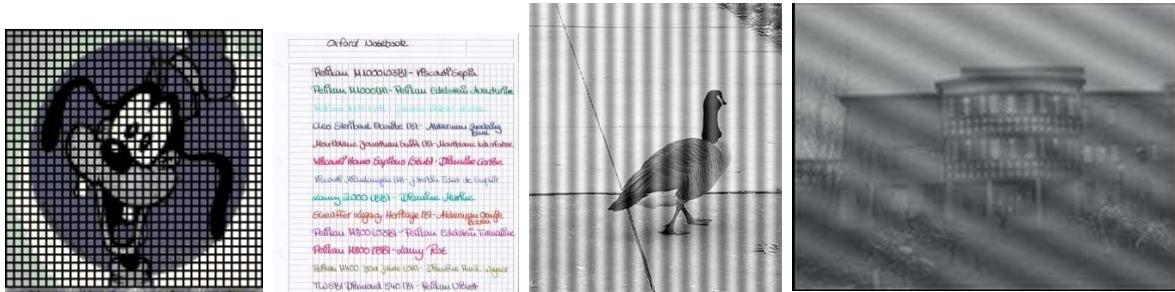
Abgabetermin: **21.11.2022, 23:55**

Gesamt 45 Punkte, Ziel: Ausarbeitungen für ca. 23 Punkte

Aufgabe 1.5 Raster-Entfernung im Frequenzraum [4]

- a) [4] Entfernen Sie bei Bilddaten mit regulärem Raster im Hintergrund die vertikalen und horizontalen Balken durch (manuelle) Manipulation im Frequenzraum. Bearbeiten Sie mindestens 3 **selbst gewählte** Datensätze und vergleichen Sie den erzielten Effekt. Wird bei dieser Korrektur auch die eigentliche Bildinformation in Mitleidenschaft gezogen?

BSP-Bilder:



Aufgabe 1.6 Anisotrope Diffusion [13]

- a) [8] Implementieren Sie Anisotrope Diffusion als ImageJ Plugin. Die Eingabe der Parameter für n und kappa erfolgt dabei durch den/die Benutzer*in. Testen Sie ausführlich und analysieren Sie, wie gut die Kanten erhalten bleiben.
- b) [3] Testen Sie die Auswirkungen von kappa k auf die Ergebnisqualität bei unterschiedlichen Eingangsbildern.
- c) [2] Visualisieren Sie die 8 Bilder für die lokale Diffusionskonstante c als separate Bilder und ebenso auch die 8 Gradienten-Bilder. Stellen Sie sicher, dass der Wertebereich [0;255] nicht verlassen wird (Vorzeichen bzw. Skalierung).

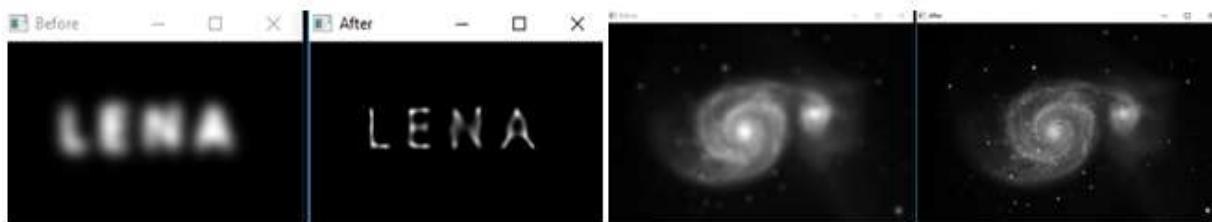
Aufgabe 1.7 Stressanalyse [12]

- a) [2] Recherchieren Sie, wie Stress aus EKG (ECG) und abgeleiteter Herzratenvariabilität abgeleitet werden kann. Steht dabei eine hohe oder niedrige Herzratenvariabilität für Stress und körperliche Belastung.
- b) [7] Analysieren Sie beigelegte EKG-Sequenzen (sowohl Ruhe als auch mit Bewegung) durch Anwendung von Baseline-Korrektur, Glättungsfilterung, Extraktion der Herz-Zyklen und Bestimmung des Pulses. Verwenden Sie dazu ein Tool Ihrer Wahl.
- c) [3] Leiten Sie aufbauend auf (b) die Herzratenvariabilität ab.

Dokumentieren und diskutieren Sie dabei ausführlich und vergleichen Sie dabei stets die „körperliche“ Ruhe mit „Stress / Bewegung“.

Aufgabe 1.8 Realisieren Sie eine RL-Deconvolution zur Bild-Restaurierung im Ortsraum [16]

- a) [10] Implementieren Sie eine RLD. Der Filterkern (Gauss, Mean oder ähnliche Tiefpass-Filter) kann dabei vorgegeben werden und ist folglich bekannt (PSF, point spread function). Glätten Sie damit das Eingangsbild und rekonstruieren Sie wieder bestmöglich das originale Bild mittels iterativer Annäherung. Methodensignatur: `private double[][] RDL(double[][] original, double[][] kernel, int width, int height, int radius, int iterations)`. Verwenden Sie zur Faltung die Methode "ConvolveDoubleNorm" aus der Übung – wenn der Kernel symmetrisch ist, muss kein „Flip“ erfolgen!
- b) [3] Testen Sie mit unterschiedlichen Varianten für die initiale Rekonstruktion g_0 , nämlich das gefaltete Eingangsbild, ein zufälliges Bild (Random), ein monotones Bild und ein anderes Bild (z.B. Graustufen-Clown). Welche Initialisierung liefert die besten Ergebnisse?
- c) [3] Verbessern Sie Ihre Implementierung der RLD dahingehend, dass bei unterschiedlichen Varianten gemäß (b) bzw. im Verlauf der Optimierung keine numerischen Probleme, wie etwa eine Division durch 0 usw. erfolgen und testen Sie ausführlich.



Inhaltsverzeichnis

1 Aufgabe 1.6 Anisotrope Diffusion	5
1.1 a	5
1.1.1 Implementierung	5
1.1.2 Beispielanwendungen	8
1.1.3 Analyse	11
1.2 b	11
1.3 c	14
2 Aufgabe 1.7 Stressanalyse	16
2.1 a	16
2.2 b	16
2.2.1 Matlab-File	16
2.3 Analyse	18
2.3.1 c	20
3 Aufgabe 1.8 RL-Deconvolution zur BildRestauration im Ortsraum	21
3.1 a	21
3.1.1 Filterklasse	21
3.1.2 Convolution Filterklasse	22
3.1.3 Dokumentation	23
3.1.4 Beispiele	24
3.2 b	26
3.3 c	29

1 Aufgabe 1.6 Anisotrope Diffusion

1.1 a

1.1.1 Implementierung

Die Pixelwerte werden in ein 2-dimensionales integer Array geladen. Für die weitere Verarbeitung werden allerdings Werte im Datentyp double benötigt, weshalb eine Konvertierung mittels der convertToDoubleArr2D Funktion aus der ImageJUtility durchgeführt wird. Danach wird ein Dialogfenster implementiert, das den Nutzer dazu auffordert Werte für Kappa und die Anzahl der Iterationen einzugeben. Zusätzlich kann ausgewählt werden, ob die Gradient- und Diffusionsbilder ausgegeben werden sollen, die während des Prozesses anfallen. Nach der Initiierung aller benötigten 2-dimensionalen Arrays (Masken in alle Richtungen, Gradient- und Diffusionsarrays) werden in jeder Iteration mithilfe der convolveDouble Funktion des Convolution Filters die Gradienten erstellt. Um die Diffusionen zu erstellen, werden diese mit dem eingegebenen Kappawert verrechnet:

$$Cn[x][y] = \text{Math.exp}(-0.5 * (\text{Math.pow}((\text{nablan}[x][y] / \text{kappa}), 2.0)))$$

Für das Ergebnisbild werden dann die Gradienten mit ihrer jeweiligen Gewichtung (diagonal: $1/\sqrt{2}$, rest: 1) und den Diffusionen verrechnet. Der Pixelwert des Ausgangsbildes wird zum Schluss darauf addiert. Der neu entstandene Wert wird in das Ergebnisarray an jeweiliger Stelle eingesetzt.

```

1 public class AnisotropicDiffusion_ implements PlugInFilter {
2
3     public int setup(String arg, ImagePlus imp) {
4         if (arg.equals("about"))
5             {showAbout(); return DONE;}
6         return DOES_8G+DOES_STACKS+SUPPORTS_MASKING;
7     } //setup
8
9
10    public void run(ImageProcessor ip) {
11        byte[] pixels = (byte[])ip.getPixels();
12        int width = ip.getWidth();
13        int height = ip.getHeight();
14
15        int[][] inDataArrInt = ImageJUtility.convertFrom1DByteArray(pixels, width, height)
16        ;
17        //Convert image values from integer to double for further processing
18        double[][] outImg = ImageJUtility.convert.ToDoubleArr2D(inDataArrInt, width,
19                      height);
20
21        //Radius of mask
22        int radius = 1;
23
24        //Weight based on pixel distance from center
25        double dx = 1.0;
26        double dy = 1.0;
27        double dd = 1.0 / Math.sqrt(2.0);
28
29        //Initiate variables for user input
30        int numOfIterations = 10;
31        double kappa = 10.0;
32        boolean dgimages = false;
33
34        //Get values in dialog from user
35        GenericDialog gd = new GenericDialog("Enter # of iterations and kappa");
36        gd.addNumericField("Iterations (n)", numOfIterations, 0);
37        gd.addNumericField("Kappa (k)", kappa, 0);
38        gd.addCheckbox("Show diffusion and gradient images", dgimages);
39        gd.showDialog();
40        if (gd.wasCanceled())
41            return;
42
43        //Set values to variables
44        numOfIterations = (int) gd.getNextNumber();

```

```

44     kappa = (int) gd.getNextNumber();
45     dgimages = gd.getNextBoolean();
46
47     //Initiate directional filter masks
48     double[][] hNO = new double[][] {{0.0, 0.0, 1.0}, {0.0, -1.0, 0.0}, {0.0, 0.0,
49     0.0}};
50     double[][] hO = new double[][] {{0.0, 0.0, 0.0}, {0.0, -1.0, 1.0}, {0.0, 0.0,
51     0.0}};
52     double[][] hN = new double[][] {{0.0, 1.0, 0.0}, {0.0, -1.0, 0.0}, {0.0, 0.0,
53     0.0}};
54     double[][] hSO = new double[][] {{0.0, 0.0, 0.0}, {0.0, -1.0, 0.0}, {0.0, 0.0,
55     1.0}};
56     double[][] hS = new double[][] {{0.0, 0.0, 0.0}, {0.0, -1.0, 0.0}, {0.0, 1.0,
57     0.0}};
58     double[][] hSW = new double[][] {{0.0, 0.0, 0.0}, {0.0, -1.0, 0.0}, {1.0, 0.0,
59     0.0}};
60     double[][] hW = new double[][] {{0.0, 0.0, 0.0}, {1.0, -1.0, 0.0}, {0.0, 0.0,
61     0.0}};
62     double[][] hNW = new double[][] {{1.0, 0.0, 0.0}, {0.0, -1.0, 0.0}, {0.0, 0.0,
63     0.0}};
64
65
66     //Initiate arrays for gradients
67     double[][] nablaN = new double[width][height];
68     double[][] nablaNO = new double[width][height];
69     double[][] nablaO = new double[width][height];
70     double[][] nablaSO = new double[width][height];
71     double[][] nablaS = new double[width][height];
72     double[][] nablaSW = new double[width][height];
73     double[][] nablaW = new double[width][height];
74     double[][] nablaNW = new double[width][height];
75
76
77     //Initiate arrays for coefficients
78     double[][] cN = new double[width][height];
79     double[][] cNO = new double[width][height];
80     double[][] cO = new double[width][height];
81     double[][] cSO = new double[width][height];
82     double[][] cS = new double[width][height];
83     double[][] cSW = new double[width][height];
84     double[][] cW = new double[width][height];
85     double[][] cNW = new double[width][height];
86
87
88     //Apply anisotropic diffusion n times
89     for (int iter = 0; iter < numOfIterations; iter++) {
90
91         //Do convolutions in all directions
92         nablaN = ConvolutionFilter.convolveDouble(outImg, width, height, hN, radius);
93         ;
94         nablaNO = ConvolutionFilter.convolveDouble(outImg, width, height, hNO,
95             radius);
96         nablaO = ConvolutionFilter.convolveDouble(outImg, width, height, hO, radius);
97         ;
98         nablaSO = ConvolutionFilter.convolveDouble(outImg, width, height, hSO,
99             radius);
100        nablaS = ConvolutionFilter.convolveDouble(outImg, width, height, hS, radius);
101        ;
102        nablaSW = ConvolutionFilter.convolveDouble(outImg, width, height, hSW,
103            radius);
104        nablaW = ConvolutionFilter.convolveDouble(outImg, width, height, hW, radius);
105        ;
106        nablaNW = ConvolutionFilter.convolveDouble(outImg, width, height, hNW,
107            radius);
108
109
110        //Get coefficients from gradients
111        for (int x = 0; x < width; x++) {
112            for (int y = 0; y < height; y++) {
113                cN[x][y] = Math.exp(-0.5 * (Math.pow((nablaN[x][y] / kappa), 2.0)));
114                cNO[x][y] = Math.exp(-0.5 * (Math.pow((nablaNO[x][y] / kappa), 2.0))
115                    );
116                cO[x][y] = Math.exp(-0.5 * (Math.pow((nablaO[x][y] / kappa), 2.0)));
117                cSO[x][y] = Math.exp(-0.5 * (Math.pow((nablaSO[x][y] / kappa), 2.0))
118                    );
119                cS[x][y] = Math.exp(-0.5 * (Math.pow((nablaS[x][y] / kappa), 2.0)));
120                cSW[x][y] = Math.exp(-0.5 * (Math.pow((nablaSW[x][y] / kappa), 2.0)))
121            }
122        }
123    }

```

```

99
100    );
101   cW[x][y] = Math.exp(-0.5 * (Math.pow((nablaW[x][y] / kappa), 2.0)));
102   cNW[x][y] = Math.exp(-0.5 * (Math.pow((nablaNW[x][y] / kappa), 2.0))
103   );
104 } //Height
105 } //Width
106
107 //Iterate through every pixel of outImg
108 for (int x = 0; x < width; x++) {
109   for (int y = 0; y < height; y++) {
110     //Perform anisotropic diffusion
111     outImg[x][y] = outImg[x][y] + ((1/6.8) * ((dx * ((c0[x][y] * nabla0[
112       x][y]) + (cW[x][y] * nablaW[x][y]))) + (dy * ((cN[x][y] * nablaN
113       [x][y]) + (cS[x][y] * nablaS[x][y]))) + (dd * ((cNO[x][y] *
114       nablaNO[x][y]) + (cSO[x][y] * nablaSO[x][y]) + (cSW[x][y] *
115       nablaSW[x][y]) + (cNW[x][y] * nablaNW[x][y]))));
116   } //Height
117 } //Width
118
119 } //Iterations
120
121 //Append naming of generated image by n and k values
122 String nstr = new String("n: " + numOfIterations + ", ");
123 String kappastr = new String("k: " + (int) kappa);
124
125 //Generate and output final image
126 ImageJUtility.showNewImage(outImg, width, height, "anisotropic diffusion, " +
127   nstr + kappastr);
128
129 //If user wants to get diffusion and gradient images
130 if (dgimages) {
131
132   //Get gradient images for all directions
133   getGradientImage(nablaN, width, height, "Gradient N");
134   getGradientImage(nablaO, width, height, "Gradient O");
135   getGradientImage(nablaS, width, height, "Gradient S");
136   getGradientImage(nablaW, width, height, "Gradient W");
137   getGradientImage(nablaNO, width, height, "Gradient NO");
138   getGradientImage(nablaNW, width, height, "Gradient NW");
139   getGradientImage(nablaSW, width, height, "Gradient SW");
140   getGradientImage(nablaSO, width, height, "Gradient SO");
141
142   //Get diffusion images for all directions
143   getDiffusionImage(cN, width, height, "Diffusion N");
144   getDiffusionImage(c0, width, height, "Diffusion O");
145   getDiffusionImage(cS, width, height, "Diffusion S");
146   getDiffusionImage(cW, width, height, "Diffusion W");
147   getDiffusionImage(cNO, width, height, "Diffusion NO");
148   getDiffusionImage(cNW, width, height, "Diffusion NW");
149   getDiffusionImage(cSW, width, height, "Diffusion SW");
150   getDiffusionImage(cSO, width, height, "Diffusion SO");
151
152 } //If
153 } //Run
154
155 public void getGradientImage(double[][] gradientArray, int width, int height, String
156   name) {
157   // Initiate new array
158   double[][] gradient = new double[width][height];
159
160   // Iterate through all pixels of image
161   for (int x = 0; x < width; x++) {
162     for (int y = 0; y < height; y++) {
163       //Make sure every pixel value is positive
164       gradient[x][y] = Math.abs(gradientArray[x][y]);
165     } //Height
166   } //Width
167
168   //Generate and output gradient image
169   ImageJUtility.showNewImage(gradient, width, height, name);
170 } //getGradientImage
171
172 public void getDiffusionImage(double[][] diffusionArray, int width, int height,
173   String name) {

```

```
163 // Initiate new array
164 double[][] diffusion = new double[width][height];
165
166 // Iterate through all pixels of image
167 for (int x = 0; x < width; x++) {
168     for (int y = 0; y < height; y++) {
169         //Multiply diffusion values by 255
170         diffusion[x][y] = diffusionArray[x][y] * 255;
171     } //Height
172 } //Width
173
174 //Generate and output diffusion image
175 ImageJUtility.showNewImage(diffusion, width, height, name);
176 } //getDiffusionImage
177
178 void showAbout() {
179     IJ.showMessage("About Template_...",
180                 "this is a PluginFilter template\n");
181 } //showAbout
182
183 } //Class AnisotropicDiffusion_
```

1.1.2 Beispieldaten



Abbildung 1: Ausgangsbild



Abbildung 2: Iterations: 10, Kappa: 10



Abbildung 3: Iterations: 30, Kappa: 10



Abbildung 4: Iterations: 10, Kappa: 30



Abbildung 5: Iterations: 30, Kappa: 30

1.1.3 Analyse

Anhand der Testbilder kann man erkennen, dass bei hohen Werten (30/30) das Bild ähnlich einer Gauss-Filterung verschwimmt, einige wenige prägnante Kanten aber weiterhin erkennbar bleiben. Bei kleiner gewählten Werten werden Flächen wie der Grund und die Bootsrümpfe geglättet. Kanten wie die Masten und der Schriftzug bleiben aber gut erkennbar. Ein höher gewählter Kappa-Wert führt zu einem höheren Blur des gesamten Bildes, während mehrere Iterationen mit einem kleineren Kappa-Wert zu weiterhin gut erkennbaren Kanten führen.

1.2 b



Abbildung 6: Eingangsbilde #1 - camera.png

Abbildung 7: camera.png mit $n=10, k=10$



Abbildung 8: camera.png mit $n=10, k=20$

Abbildung 9: camera.png mit $n=10, k=30$



Abbildung 10: Eingangsbilde #2 - clown.png



Abbildung 11: clown.png mit n=10, k=10



Abbildung 12: clown.png mit n=10, k=20



Abbildung 13: clown.png mit n=10, k=30



Abbildung 14: Eingangsbilde #2 - leaf.png



Abbildung 15: leaf.png mit n=10, k=10



Abbildung 16: leaf.png mit n=10, k=20



Abbildung 17: leaf.png mit n=10, k=30

Kappa steht für die Stärke, mit der die Anisotrope Diffusion angewandt wird. Zusammen mit der Anzahl der Iterationen kann man die Auswirkung des Filters auf ein Bild einstellen. In jeder der drei Versuchsreihen wurden 10 Durchgänge mit jeweils einem Kappawert von 10, 20 und 30 gewählt. Mit höherem Kappawert ist ein Blur zu erkennen, der allerdings bei kontrastreichen Kanten ausbleibt (Augen in Versuchsreihe clown.png).

1.3 c

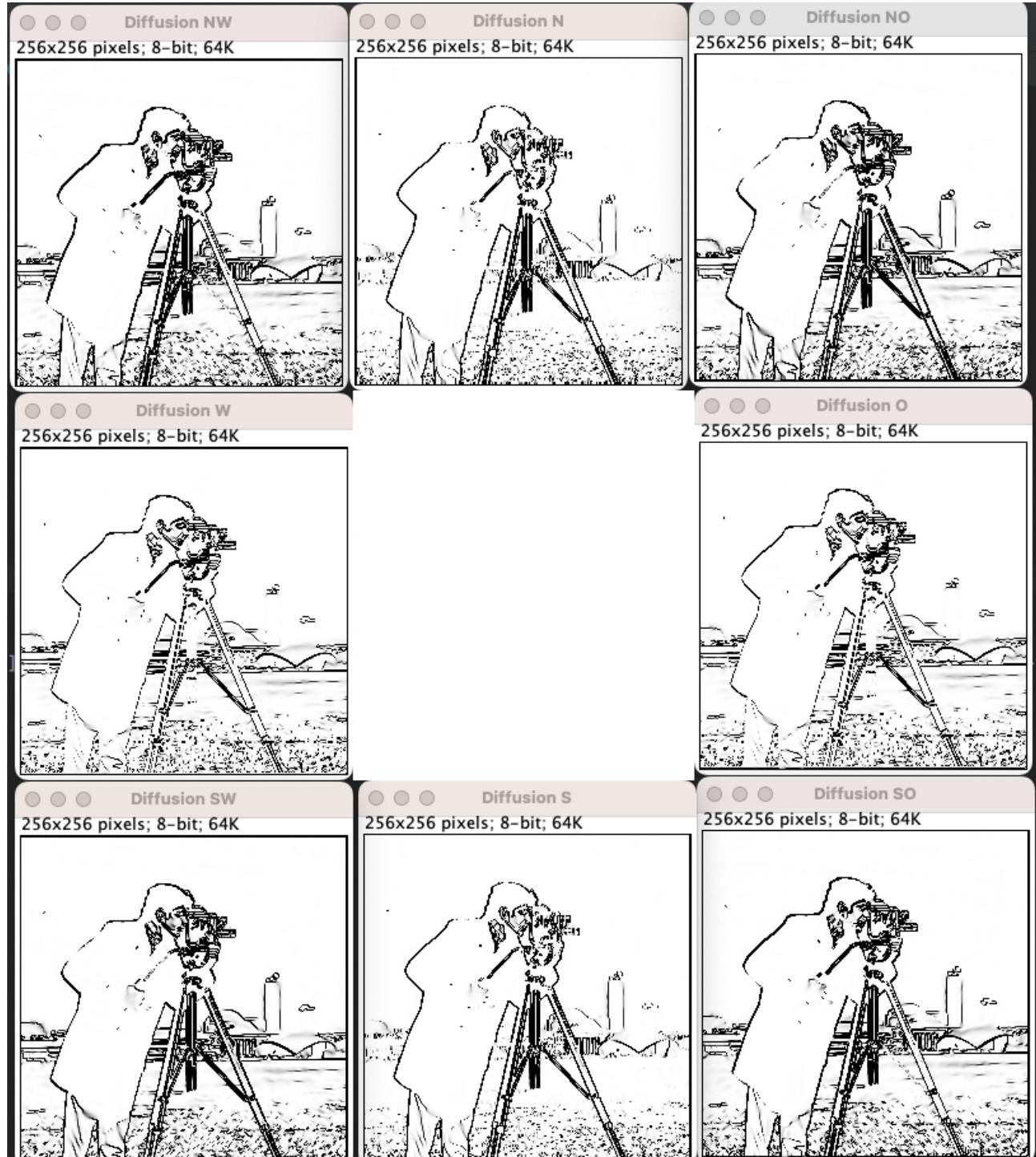


Abbildung 18: Bilder für die lokale Diffusionskonstante c

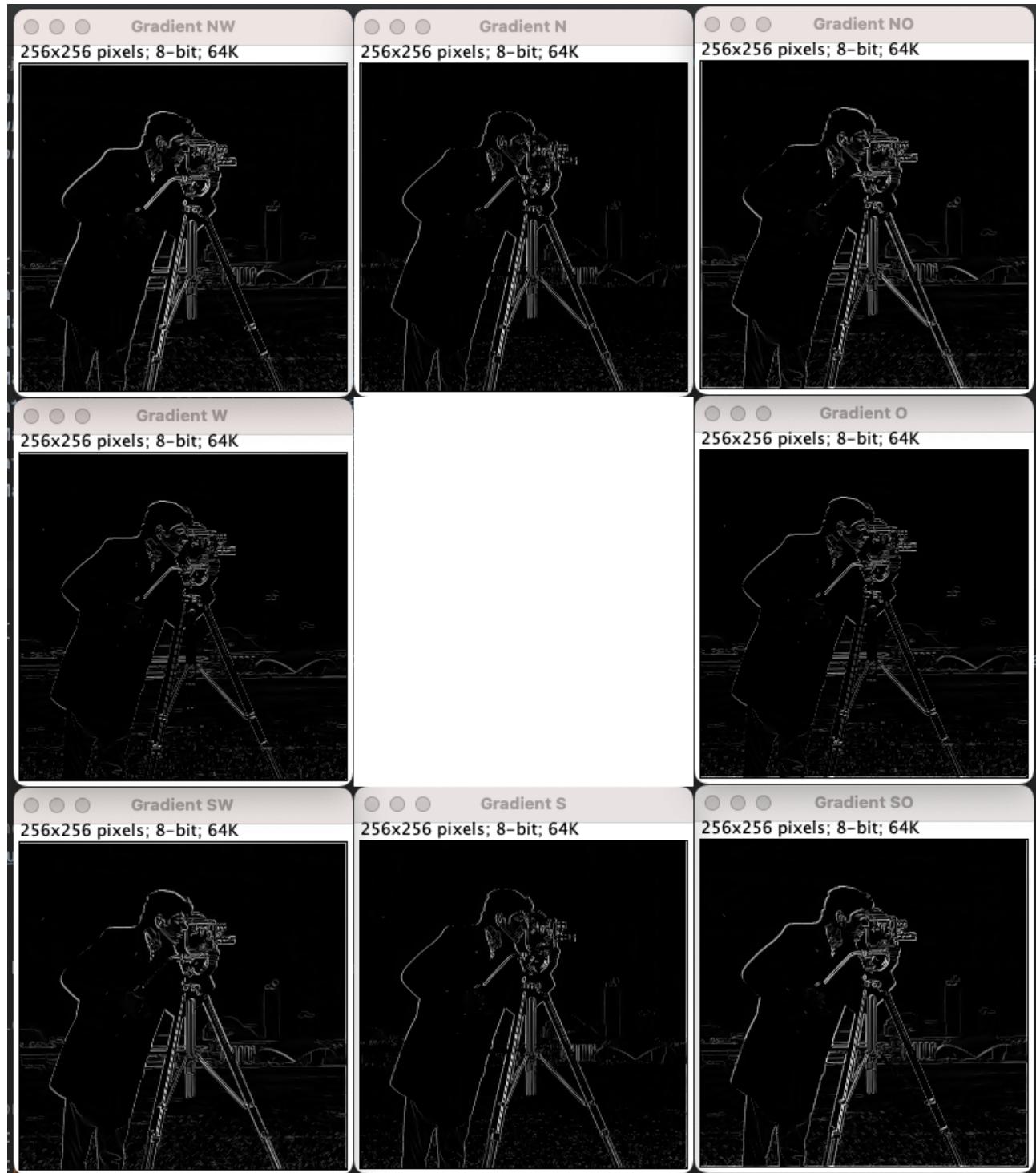


Abbildung 19: Gradienten-Bilder

2 Aufgabe 1.7 Stressanalyse

2.1 a

Aus dem EKG kann der Zeitabstand zwischen aufeinanderfolgenden Herzschlägen in Millisekunden abgelesen werden. Werden diese Zeitabstände kürzer deutet dies bereits auf Stress hin. Weiters kann die Herzratenvariabilität aus dem EKG ermittelt werden.

Die Herzratenvariabilität (kurz HRV) ist das physiologische Phänomen der Variation des Zeitabstands zwischen aufeinanderfolgenden Herzschlägen in Millisekunden. Dabei gilt eine hohe HRV als Zeichen für ein gesundes Herz und wird mit psychologischer Gesundheit, höherer Lebensqualität und geringerer Anfälligkeit für Krankheiten in Verbindung gebracht. Das liegt daran, dass das Herz im Normalfall, im Gegensatz zu einem Metronom, nicht immer im selben Takt schlägt. Eine niedrige Herzratenvariabilität weist auf Stress oder körperliche Aktivität hin, während eine hohe HRV auf Ruhe und Gesundheit hinweist.

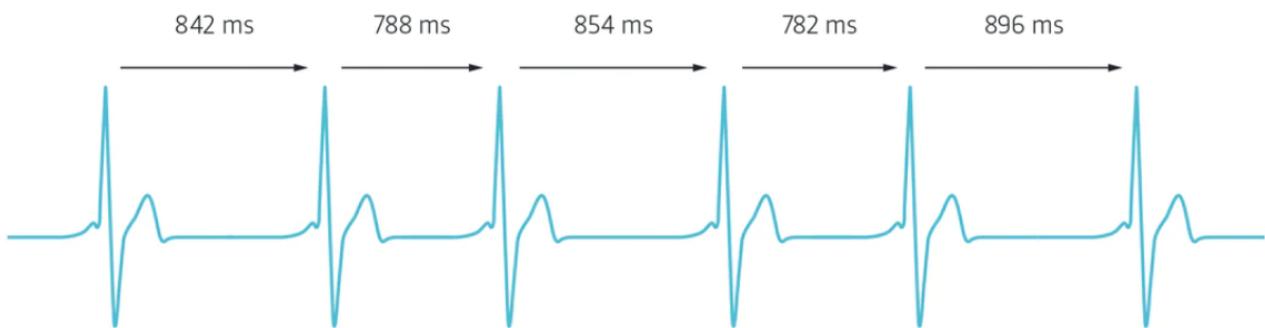


Abbildung 20: Beispiel zur Herzratenvariabilität anhand eines gesunden Herzens

2.2 b

2.2.1 Matlab-File

Listing 1: ..//Stressanalyse/EKG.m

```

1 close all;
2 hold on;
3 order = 4;
4 frameLEN = 13;
5
6 lx = 20;
7
8 % generate Signal
9 x_move = Time_Movement;
10 x_nomove = Time_NoMovement;
11
12 y_move = (Data_Movement)';
13 y_nomove = (Data_NoMovement)';
14
15 dy_move = diff(y_move);
16 dy_nomove = diff(y_nomove);
17 figure("name","Peaks of ECG with Movement");
18 findpeaks(y_move - medfilt1(y_move,50) ,x_move , 'Annotate' , 'extents' , 'MinPeakProminence' ,
19 ,0.008);
20 figure("name","Peaks of ECG without Movement");
21 findpeaks(y_nomove - medfilt1(y_nomove,50) ,x_nomove , 'Annotate' , 'extents' ,
22 'MinPeakProminence' ,0.008);
23 [pks_move,locs_move,peakWidth1_move,p_move] = findpeaks(y_move - medfilt1(y_move,50) ,
24 x_move , 'MinPeakProminence' ,0.008);

```

```
23 [pks_nomove ,locs_nomove ,peakWidth1_nomove ,p_nomove] = findpeaks(y_nomove -medfilt1( 
    y_nomove ,50),x_nomove ,'MinPeakProminence' ,0.008);
24
25 figure("name","ECG with Movement");
26 plot(x_move,y_move);
27 title("ECG with Movement");
28 xlabel("Time");
29 ylabel("mV");
30 figure("name","ECG with Movement and Baseline-Correction");
31 plot(locs_move,pks_move ,'r');
32 title("ECG with Movement and Baseline-Correction");
33 xlabel("Time");
34 ylabel("mV");
35 hold off;
36 figure("name","ECG without Movement");
37 plot(x_nomove,y_nomove);
38 title("ECG without Movement");
39 xlabel("Time");
40 ylabel("mV");
41 figure("name","ECG without Movement and Baseline-Correction");
42 plot(locs_nomove,pks_nomove ,'r');
43 title("ECG without Movement and Baseline-Correction");
44 xlabel("Time");
45 ylabel("mV");
46
47 % both ECGs
48 figure("name","ECG in Comparisson");
49 plot(x_move,y_move ,'b');
50 hold on;
51 plot(x_nomove,y_nomove);
52 legend("With Movement","Without Movement");
53 title("ECG in Comparisson without Baseline-Correction");
54 xlabel("Time");
55 ylabel("mV");
56
57 figure("name","ECG in Comparisson Baseline Correction");
58 plot(locs_move,pks_move ,'b');
59 hold on;
60 plot(locs_nomove,pks_nomove ,'r');
61 legend("With Movement","Without Movement");
62 title("ECG in Comparisson with Baseline-Correction");
63 xlabel("Time");
64 ylabel("mV");
```

2.3 Analyse

In den folgenden beiden Abbildungen wurden die EKGs (mit und ohne Bewegung) jeweils anhand deren Baseline korrigiert. Diese Korrektur erfolgte mit einem Matlab-Skript, das weiter oben eingefügt wurde. Bei der Baseline-Korrektur wird, anhand einer Medianfilterung, werden die Ausgangspunkte des Signals gegen die Basislinie Null gezogen, damit die einzelnen Segmente besser miteinander verglichen werden können. Vertikale Ausschläge bleiben dabei erhalten, um Arrhythmien trotzdem sehen zu können.

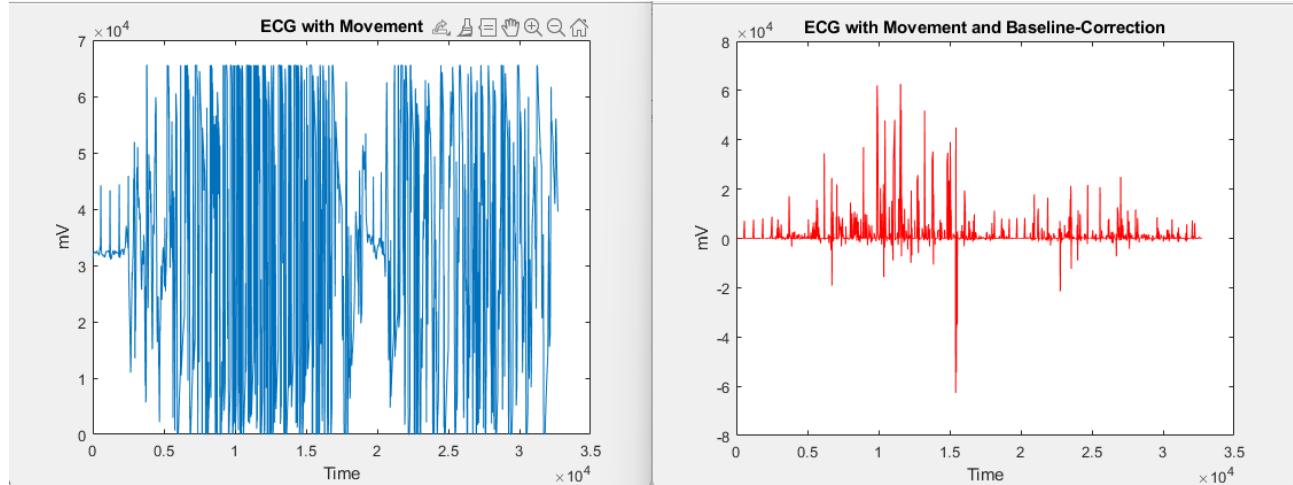


Abbildung 21: ECG mit Bewegung und anschließende Baseline-Korrektur

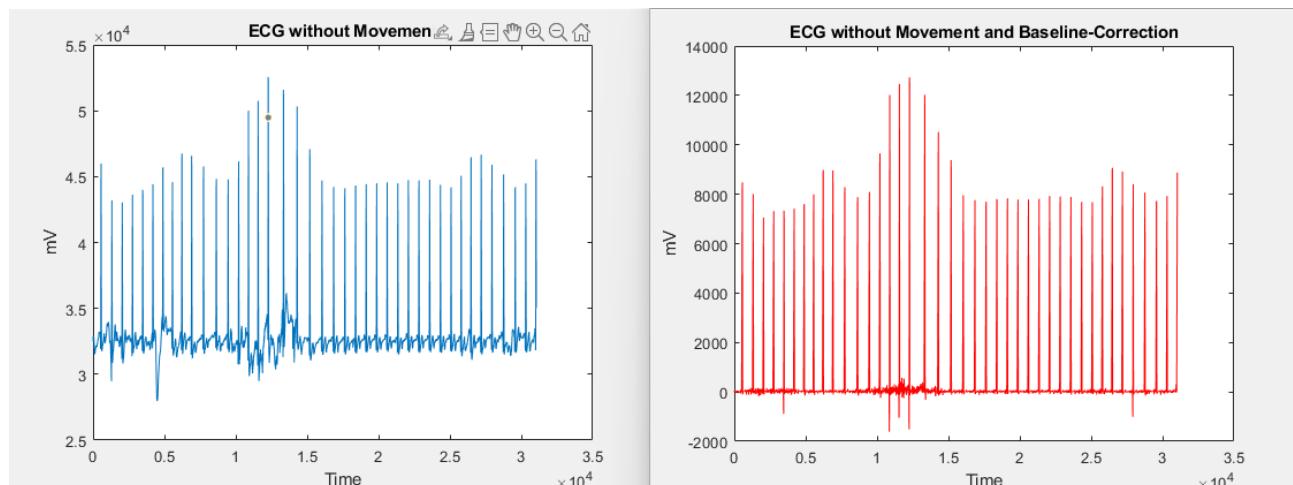


Abbildung 22: ECG ohne Bewegung und anschließende Baseline-Korrektur

Die nächsten beiden Abbildungen zeigen eine direkte Gegenüberstellung des EKGs mit Bewegung und dessen ohne Bewegung. Die deutlich niedrigeren Abstände der Ausschläge beim EKG mit Bewegung deuten dabei auf eine niedrige HRV hin, was wiederum bedeutet, dass die Person unter Stress steht. Beim EKG ohne Bewegung sind diese Abstände deutlich größer, was auf eine hohe HRV hindeutet. Die Person steht also unter keinem Stress.

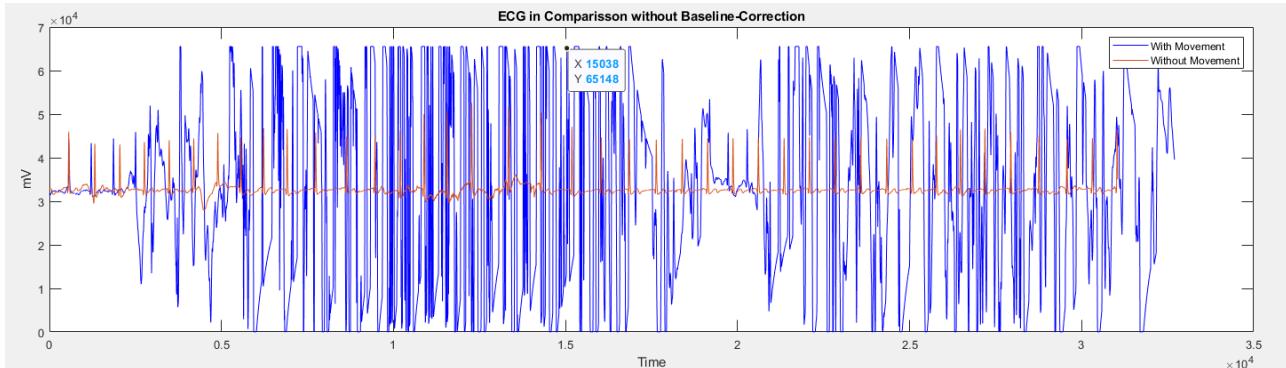


Abbildung 23: Vergleich der beiden EKGs ohne Baseline-Korrektur

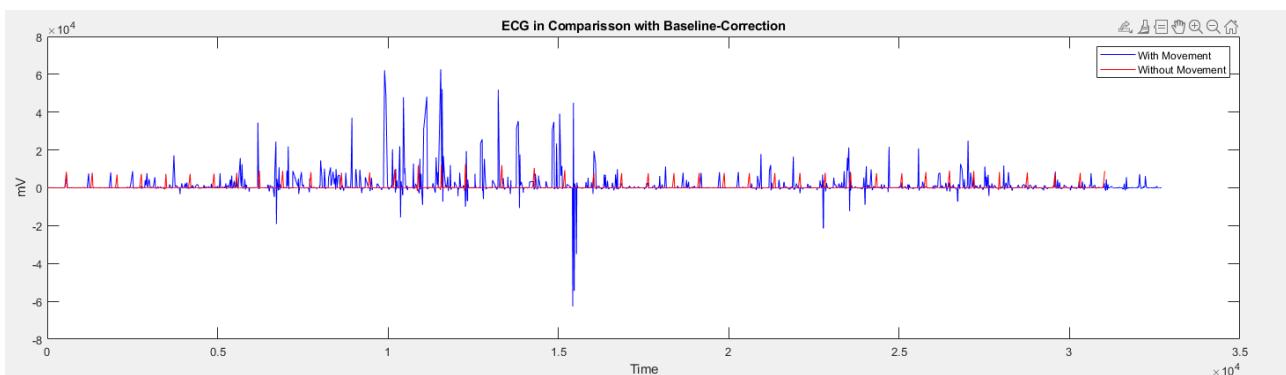


Abbildung 24: Vergleich der beiden EKGs mit Baseline-Korrektur

Im Anschluss wird noch die Herzfrequenz (ugs. Puls) der beiden EKGs zu den ersten beiden Schlägen zur Veranschaulichung berechnet. Dabei kann beobachtet werden, dass der Puls während der Bewegung deutlich höher ist als ohne Bewegung.

$$\text{Mit Bewegung: } 60000\text{ms}/(1215\text{ms}-569\text{ms}) = 92,88 \text{ bpm}$$

$$\text{Ohne Bewegung: } 60000\text{ms}/(1320\text{ms}-561\text{ms}) = 79,05 \text{ bpm}$$

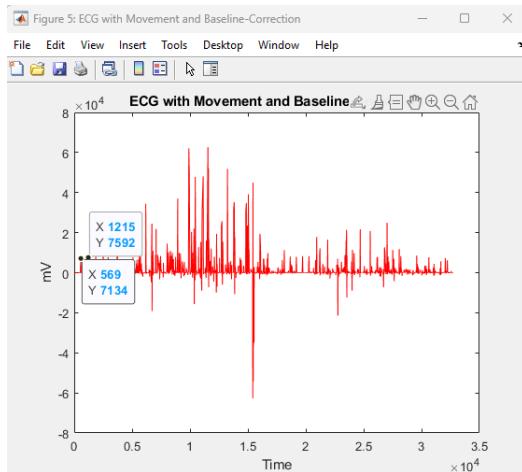


Abbildung 25: Pulse mit Bewegung

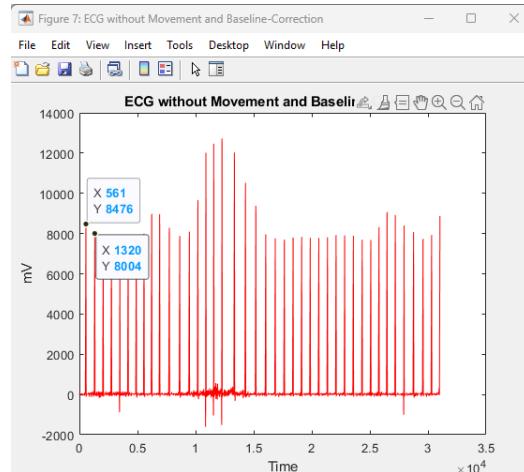


Abbildung 26: Puls ohne Bewegung

2.3.1 c

Die Herzratenvariabilität wurde ebenfalls via Matlab ermittelt und als Plot ausgegeben. Dabei wurden alle Hoch- und Tiefpunkte gleichermaßen in das Plot mit einbezogen. Zu beachten ist aber, dass durch das Rauschen sehr viele Punkte einbezogen wurden. Relevant für die Herzratenvariabilität sind nur die größeren Ausschläge des Diagramms, da nur diese für einen Herzschlag stehen. Die Ausschläge stehen für Distanz bis zum nächsten Schlag. Auch hier wird wieder deutlich, dass die Varianz beim EKG ohne Bewegung höher ist als bei dem mit Bewegung.

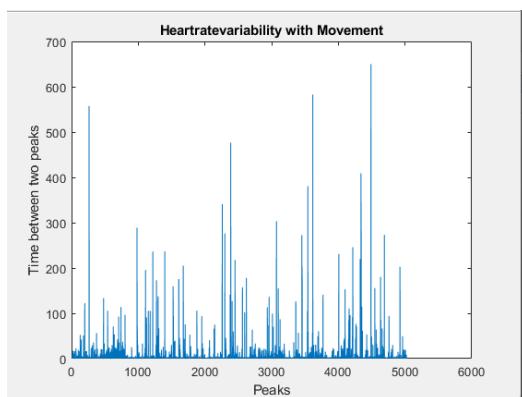


Abbildung 27: Herzratenvariabilität mit Bewegung

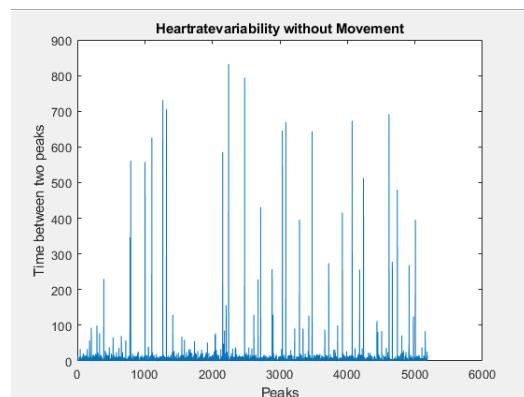


Abbildung 28: Herzratenvariabilität ohne Bewegung

3 Aufgabe 1.8 RL-Deconvolution zur BildRestaurierung im Ortsraum

3.1 a

3.1.1 Filterklasse

```

1  public class RichardsonLucyDeconvolution_ implements PlugInFilter {
2
3      public int setup(String arg, ImagePlus imp) {
4          if (arg.equals("about"))
5              {showAbout(); return DONE;}
6          return DOES_8G+DOES_STACKS+SUPPORTS_MASKING;
7      } //setup
8
9
10     public void run(ImageProcessor ip) {
11         byte[] pixels = (byte[])ip.getPixels();
12         int width = ip.getWidth();
13         int height = ip.getHeight();
14         int[][] inDataArrInt = ImageJUtility.convertFrom1DByteArr(pixels, width, height)
15             ;
16         double[][] inDataArrDouble = ImageJUtility.convert.ToDoubleArr2D(inDataArrInt,
17             width, height);
18         double[][] guessImg = ConvolutionFilter.GetGuessImage(inDataArrDouble, width,
19             height, 1);
20
21         int radius = 1;
22         // get number of iterations and radius
23         int iterations = 1;
24         GenericDialog gd = new GenericDialog("RLD config");
25         gd.addNumericField("Iterations", iterations, 0);
26         gd.addNumericField("Radius", radius, 0);
27         gd.showDialog();
28         if (gd.wasCanceled()) {
29             return;
30         }
31         iterations = (int)gd.getNextNumber();
32         radius = (int)gd.getNextNumber();
33
34         // point spread function
35         double kernel[][] = ConvolutionFilter.getMeanMask(radius);
36
37         // apply median filter
38         double[][] resultImg = ConvolutionFilter.RLD(inDataArrDouble, guessImg, kernel,
39             width, height, radius, iterations);
40
41         // show the result
42         ImageJUtility.showNewImage(resultImg, width, height, "RLD edited image");
43     } //run
44
45     void showAbout() {
46         IJ.showMessage("RLD",
47             "this is a PluginFilter RLD\n");
48     } //showAbout
49 } //class RichardsonLucyDeconvolution_

```

3.1.2 Convolution Filterklasse

```

1   public static double[][] GetGuessImage(double[][] originalImg, int width, int height
2     , int mode) {
3       double[][] guessImage = new double[width][height];
4
5       //uses the folded original image
6       if (mode == 0) {
7           for (int x = 0; x < width; x++) {
8               for (int y = 0; y < height; y++) {
9                   guessImage[x][y] = originalImg[x][y];
10              }
11          }
12      }
13
14      //generates an image with a random value between 1 and 255 for each pixel
15      else if (mode == 1) {
16          for (int x = 0; x < width; x++) {
17              for (int y = 0; y < height; y++) {
18                  guessImage[x][y] = (int) ((Math.random() * (255 - 1)) + 1);
19              }
20          }
21      }
22      return guessImage;
23
24      //generates a monotonous grey image
25      else if (mode == 2) {
26          for (int x = 0; x < width; x++) {
27              for (int y = 0; y < height; y++) {
28                  guessImage[x][y] = 127.0;
29              }
30          }
31      }
32      return guessImage;
33
34      else if (mode == 3) {
35          Opener opener = new Opener();
36          String imageFilePath = "clown.png";
37          ImagePlus imp = opener.openImage(imageFilePath);
38          ImageProcessor clown = imp.getProcessor();
39
40          byte[] pixels = (byte[])clown.getPixels();
41          int c_width = clown.getWidth();
42          int c_height = clown.getHeight();
43          int[][] inDataArrInt = ImageJUtility.convertFrom1DByteArr(pixels,
44                           c_width, c_height);
45          double[][] inDataArrDbl = ImageJUtility.convert.ToDoubleArr2D(
46                           inDataArrInt, c_width, c_height);
47
48          double[][] resampledImage = Resample_.getResampledImage(inDataArrDbl ,
49                           c_width, c_height, width, height, false);
50
51          for (int x = 0; x < width; x++) {
52              for (int y = 0; y < height; y++) {
53                  guessImage[x][y] = resampledImage[x][y];
54              }
55          }
56      }
57      return guessImage;
58  }
59
60
61  public static double[][] RLD(double[][] originalImg, double[][] guessImg, double[][]
62    kernel, int width, int height, int radius, int iterations) {
63      double[][] newGuess = new double[width][height];
64      double[][] cT = new double[width][height];
65      double[][] mT = new double[width][height];
66
67      for (int x = 0; x < width; x++) {

```

```

67         for (int y = 0; y < height; y++) {
68             double pixelRes = 0;
69             for (int xOffset = -radius; xOffset <= radius; xOffset++) {
70                 for (int yOffset = -radius; yOffset <= radius; yOffset++) {
71                     int nbX = x + xOffset;
72                     int nbY = y + yOffset;
73                     if ((nbX >= 0) && (nbY >= 0) && (nbX < width) && (nbY < height))
74                     {
75                         pixelRes += kernel[xOffset + radius][yOffset + radius] *
76                         guessImg[nbX][nbY];
77                     }
78                 }
79             }
80             if (pixelRes == 0) {
81                 pixelRes = 1;
82             } else if (pixelRes > 255) {
83                 pixelRes = 255;
84             }
85             mT[x][y] = originalImg[x][y] / pixelRes;
86         }
87     newGuess = convolveDoubleNorm(mT, width, height, kernel, radius);
88
89     for (int x = 0; x < width; x++) {
90         for (int y = 0; y < height; y++) {
91             if (guessImg[x][y] != 0) {
92                 newGuess[x][y] *= guessImg[x][y];
93             }
94             if (newGuess[x][y] > 255) { // no value above maximum
95                 newGuess[x][y] = 255;
96             }
97         }
98     }
99     if (iterations == 0) {
100        return newGuess;
101    } else {
102        return RLD(originalImg, newGuess, kernel, width, height, radius, iterations
103        - 1);
104    }
105 }
```

3.1.3 Dokumentation

Für die Implementierung wurde das Filter-Template aus der Übung genommen und damit die Klasse RichardsonLucyDeconvolution erstellt. Hier erfolgt die Abfrage zum Radius der Filtermaske und der Anzahl an Iterationen. Als Filterkern wurde im Beispiel der Mittelwertfilter, der in der Übung implementiert wurde, verwendet. Die Implementierung der eigentlichen Logik befindet sich in der ConvolutionFilter-Klasse. In einer Hifsfunktion GetGuessImg werden verschiedene Bilder generiert, die für den initialen Guess verwendet werden. Bei diesen Bildern handelt es sich um ein Randombild ein gleichmäßiges Graubild mit dem Wert 127 und einem Clown. Die Berechnungen für die Deconvolution wurden aus dem Foliensatz übernommen. Hauptaugenmerk liegt dabei auf den Randwerten, damit hier kein Überlauf entsteht. Die Iterationen werden durch eine Rekursion behandelt, wobei gleichermaßen auch eine Schleife verwendet werden könnte (Performance-technisch besser, aber weniger übersichtlich).

3.1.4 Beispiele

Für die Deconvolution des Clowns wurde das Graubild verwendet. Es ist eine deutliche Verbesserung zum gefilterten Bild zu erkennen, jedoch wird es mit nur 100 Iterationen dennoch nicht so gut wie das Ausgangsbild. Außerdem wächst die Verfälschung des Bildes (schwarze Linien) am Rand mit den Iterationen immer mehr an.



Abbildung 29: Ausgangsbild



Abbildung 30: Ausgangsbild nach 2-facher Filterung



Abbildung 31: Bild nach 50 Iterationen



Abbildung 32: Bild nach 100 Iterationen

Anhand der Boote ist zu erkennen, dass mit steigenden Iterationen, die Qualität des Guesses, dem des Originalbildes schon ziemlich nahe kommt. Auch die Boote wurden mit dem gleichen Graubild als Guess initialisiert. Der Fehlerrand ist hier etwas weniger auffällig als beim Clown.



Abbildung 33: Ausgangsbild



Abbildung 34: Ausgangsbild nach 2-facher Filterung



Abbildung 35: Bild nach 50 Iterationen



Abbildung 36: Bild nach 1000 Iterationen

3.2 b

Mit gleichbleibenden Graustufenbildern wurde bereits in den anderen beiden Unterpunkten getestet. Dabei kam es zu keinen Verzerrungen der Testbilder und sie lieferten generell zufriedenstellende Ergebnisse. Weiters wird ein zufälliges Graustufenbild verwendet. Um Unterschiede besser sehen zu können, wird immer das Bootsbild verwendet und jeweils mit 50 und 100 Iterationen getestet. Nach dem Test wird dasselbe noch einmal mit Bridge probiert.

Anders als die gleichmäßige Struktur bei der Anwendung mit gleichbleibenden Graustufenbildern, sieht man in den nachfolgenden Beispielen, dass durch die zufällig generierten Guesses, ein starkes Hintergrundrauschen entsteht.



Abbildung 37: Bild nach 50 Iterationen, ran- Abbildung 38: Bild nach 100 Iterationen, ran-
dom dom

Bei den Guessen vom Clown erzielt man ähnliche Ergebnisse. Auch hier bleibt im Hintergrund ein Rauschen bestehen, dass in dem Fall dem Clown ähnelt. Selbiges gilt für die Tests mit der Brücke. Sie liefern das gleiche Ergebnis wie der Test mit den Booten. Wird der Clown als Guess verwendet, so wird dieser durch Resampling an das Ausgangsbild angepasst. (aber dazu in der nächsten Übung mehr :D)



Abbildung 39: Bild nach 50 Iterationen, clown

Abbildung 40: Bild nach 100 Iterationen
clown



Abbildung 41: Bild nach 50 Iterationen ran-
dom



Abbildung 42: Bild nach 100 Iterationen ran-
dom



Abbildung 43: Bild nach 50 Iterationen clown



Abbildung 44: Bild nach 100 Iterationen
clown

Wird das Originalbild als Guess verwendet, so führt dies zwangsläufig zu den besten Ergebnissen. Dabei muss allerdings darauf geachtet werden, dass es zu keiner Überkorrektur kommt, da sonst die Ergebnisse wieder schlechter werden. Der Unterschied zwischen 50 und 100 Iterationen fällt hierbei fast nicht mehr ins Gewicht.



Abbildung 45: Bild nach 50 Iterationen original
Abbildung 46: Bild nach 100 Iterationen original



Abbildung 47: Bild nach 50 Iterationen original
Abbildung 48: Bild nach 100 Iterationen original

3.3 c

In diesem Codesnippet befinden sich alle Vorkehrungen um Operationen mit 0 zu vermeiden, damit keine schwarzen Bilder entstehen oder eine Nulldivision durchgeführt wird.

```

1      if (pixelRes == 0) {
2          pixelRes = 1;
3      } else if (pixelRes > 255) {
4          pixelRes = 255;
5      }
6      mT[x][y] = originalImg[x][y] / pixelRes;
7  }
8
9  newGuess = convolveDoubleNorm(mT, width, height, kernel, radius);
10
11 for (int x = 0; x < width; x++) {
12     for (int y = 0; y < height; y++) {
13         if (guessImg[x][y] != 0) {
14             newGuess[x][y] *= guessImg[x][y];
15         }
16         if (newGuess[x][y] > 255) { // no value above maximum
17             newGuess[x][y] = 255;
18         }
}

```

Im folgenden Beispiel wurde mit einem komplett schwarzem Guess (alles 0) initialisiert und bietet trotzdem gute Ergebnisse ohne Probleme. Interessant war bei diesem Test, dass die doppelte Anwendung von 50 Iterationen, bessere Ergebnisse lieferte als die Anwendung von 1000 Iterationen.



Abbildung 49: Ausgangsbild



Abbildung 50: Ausgangsbild nach 2-facher Filterung



Abbildung 51: Bild nach 2 mal 50 Iterationen



Abbildung 52: Bild nach 1000 Iterationen