

# SBV1, Signal- und Bildverarbeitung – WS 2022

## Übungsabgabe 3

Rudolf Hofmeister / David Lang

10. Januar 2023

### Zusammenfassung

Registrierung und Bildüberlagerung. **3.1 e) Kein Versuch!!!**

### 3.1 Aufgabe

#### a) Lösungsidee

Das Eingangsbild soll in der horizontalen Mitte geteilt und in 2 separate Hälften als neue Bilder ausgegeben werden. Dazu wird das Bild mit Länge und Breite der Funktion übergeben. Das Teilen eines Bildes mit gerader Breite ergibt auch zwei exakte Hälften, man muss sich jedoch eine Strategie überlegen, wenn die Breite ungerade Pixel groß ist. Man könnte das eine Pixel, das rechts zuviel ist, im neuen rechten Bild verwerfen oder ein Pixel im linken Bild hinzufügen. Als letztes könnte man noch in zwei Bilder teilen, wovon eines um ein Pixel breiter ist. Die letzte Variante werden wir umsetzen. Es soll vermieden werden, direkt in der Funktion die Bilder auszugeben, deshalb wird ein 3-dimensionales Array als Container für die zwei 2-dimensionalen Bilder dienen.

#### Source Code

```
1 private double[][][] getSplitedImage(double[][] inImg, int width, int height) {  
2     double[][][] resultImages = new double[2][][];  
3     int halfWidth = width / 2;  
4     double[][] leftImg = new double[halfWidth][height];  
5     // right image is one pixel wider if width is odd  
6     int oddWidth = width % 2 == 0 ? halfWidth : halfWidth + 1;  
7     double[][] rightImg = new double[oddWidth][height];  
8     resultImages[0] = leftImg;  
9     resultImages[1] = rightImg;  
10    for (int x = 0; x < width; x++) {  
11        for (int y = 0; y < height; y++) {  
12            if (x < halfWidth) {  
13                leftImg[x][y] = inImg[x][y];  
14            } else {  
15                rightImg[x - halfWidth][y] = inImg[x][y];  
16            }  
17        }  
18    }  
19}
```

```

17     }
18 }
19 }
20 return resultImages;
21 }

```

Anwendung der Funktion:

```

1 double[][][] splitImages = getSplitedImage(inDataArrDbl, width, height);
2 width = width / 2; // override width after split
3 ImageJUtility.showNewImage(splitImages[0], width, height, "left image");
4 ImageJUtility.showNewImage(splitImages[1], width+1, height, "right image");

```

Tests



Abbildung 3.1: Teilung eines Bildes mit ungerader Breite.

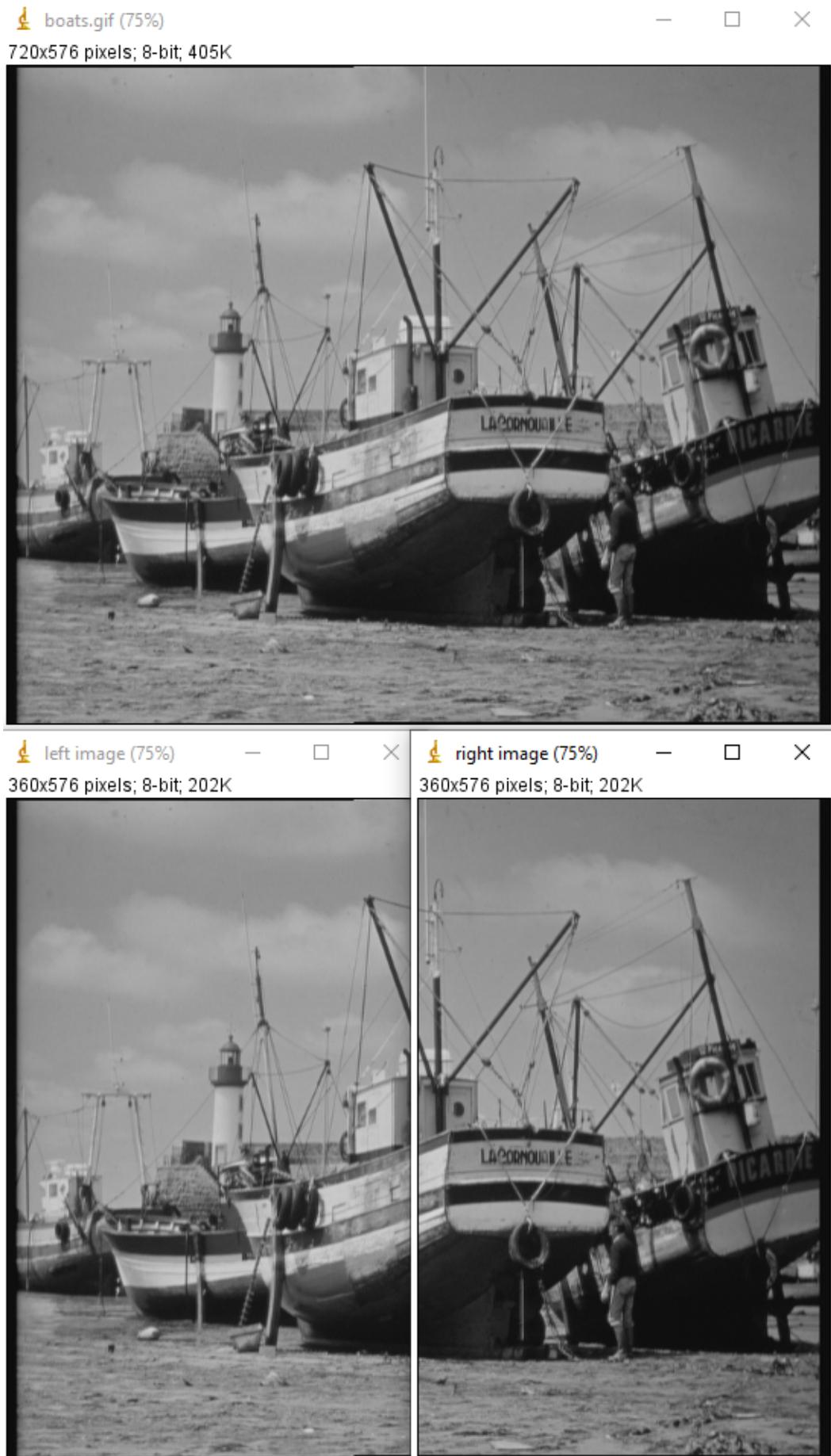
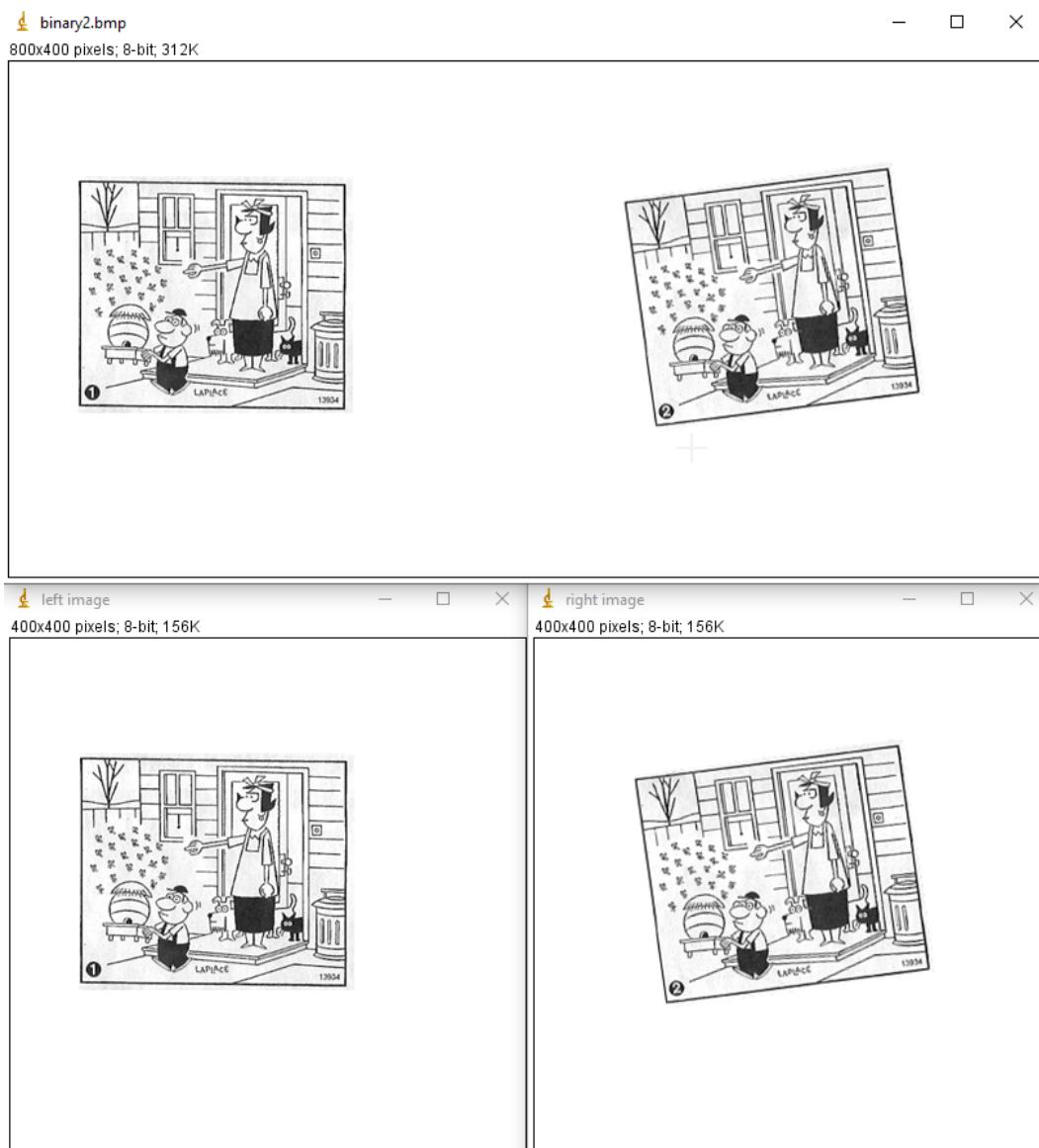


Abbildung 3.2: Teilung eines Bildes mit gerader Breite.



**Abbildung 3.3:** Teilung eines Bildes mit gerader Breite.

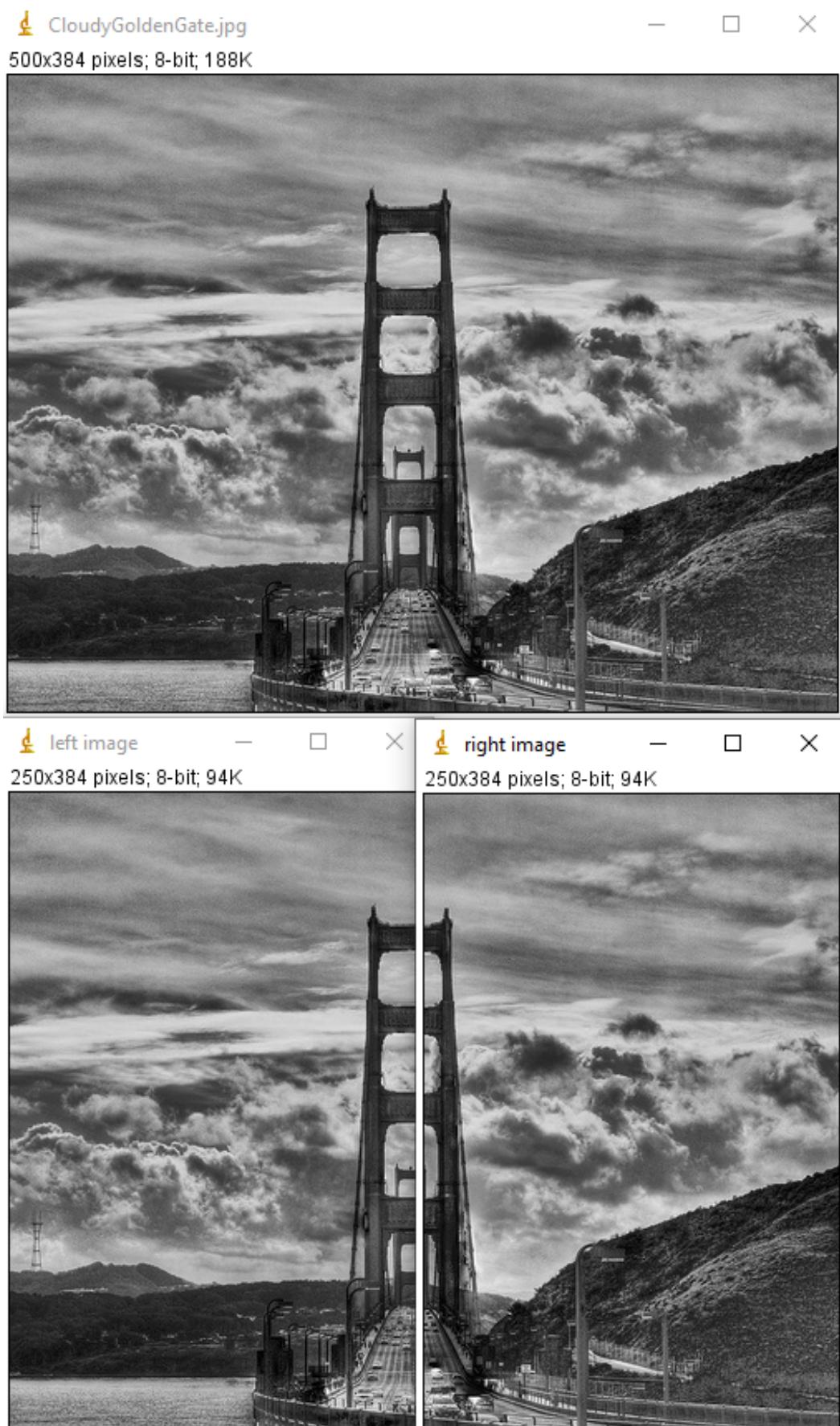


Abbildung 3.4: Teilung eines Bildes mit gerader Breite.

## b) Lösungsidee

Der Funktion für die Translation und Rotation eines Bildes soll die Parameter, um wie viel dass verschoben oder rotiert werden soll, als Funktionsparameter empfangen. Dann ist es später leichter diese Parameter als GenericDialog abzufragen. Als Rückgabe soll das transformierte Bild übergeben werden. Da es für den User intuitiver ist in Winkel zu denken, werden wir den Rotationsparameter in Grad übergeben und in der Funktion nach Rad umrechnen. Mit dem Winkel, unabhängig ob Grad oder Rad, können wir Sinus und Cosinus errechnen und transformieren mit der Entfernung vom Rotationsmittelpunkt multipliziert, wo der Pixel im neuen Bild gesetzt werden muss. Apropos Rotationsmittelpunkt, würde man einfach rotieren, dann würde  $P(0, 0)$  als Rotationsmittelpunkt herangezogen, das heißt, es wird um die linke obere Ecke rotiert, was bei einer Rotation von mehr als  $90^\circ$  das Bild aus dem sichtbaren Bereich verschwinden lässt. Deswegen muss die Bildmitte errechnet und als Rotationsmittelpunkt verwendet werden. Die Translation soll während der Iterationen laut Angabe die Nearest Neighbor Interpolation verwenden, da sie performanter ist. Das Ergebnis soll aber in Bilinearer Interpolation erstellt werden, um eine bessere Bildqualität zu erzielen. Damit man die Funktion nicht kopieren muss, soll ein boolscher Parameter angeben, welche der beiden Methoden verwendet werden soll.

### Post Implementation Anmerkung

Ich wüsste jetzt nicht, wo man beim alleinigen Transformieren Zwischenergebnisse gebraucht hätte! Kann es sein, dass hier bereit die Registrierung gemeint war? Dort haben wir es genauso umgesetzt.

### Source Code

```

1 public double[][] transformImg(double[][] inImg, int width, int height, double
2     transX, double transY, double rotAngle, boolean bilinear) {
3
4     //ATTENTION: change direction due to backward mapping
5     double radAngle = -rotAngle * Math.PI /180.0; // Math.toRadians is more expensive
6     double cosTheta = Math.cos(radAngle);
7     double sinTheta = Math.sin(radAngle);
8     double midX = width/2.0;
9     double midY = height/2.0;
10
11    //backproject all pixels of result image B
12    for(int x = 0; x < width; x++) {
13        for(int y = 0; y < height; y++) {
14            //1) move coordinate to center
15            double posX = x - midX;
16            double posY = y - midY;
17            //2) rotate
18            double newX = posX * cosTheta + posY * sinTheta;
19            double newY = -posX * sinTheta + posY * cosTheta;
20            //3) move coordinate back from center
21            newX = newX + midX;
22            newY = newY + midY;

```

```

23     //4) translate
24     posX = newX - transX;
25     posY = newY - transY;
26
27     double scalarVal;
28     if (bilinear) {
29         scalarVal = getBilinearInterpolatedValue(inImg, width, height, posX, posY);
30     } else {
31         scalarVal = getNNInterpolatedValue(inImg, width, height, posX, posY);
32     }
33     retArr[x][y] = scalarVal;
34 }
35 }
36
37 return retArr;
38 } //transformImg

```

### Tests

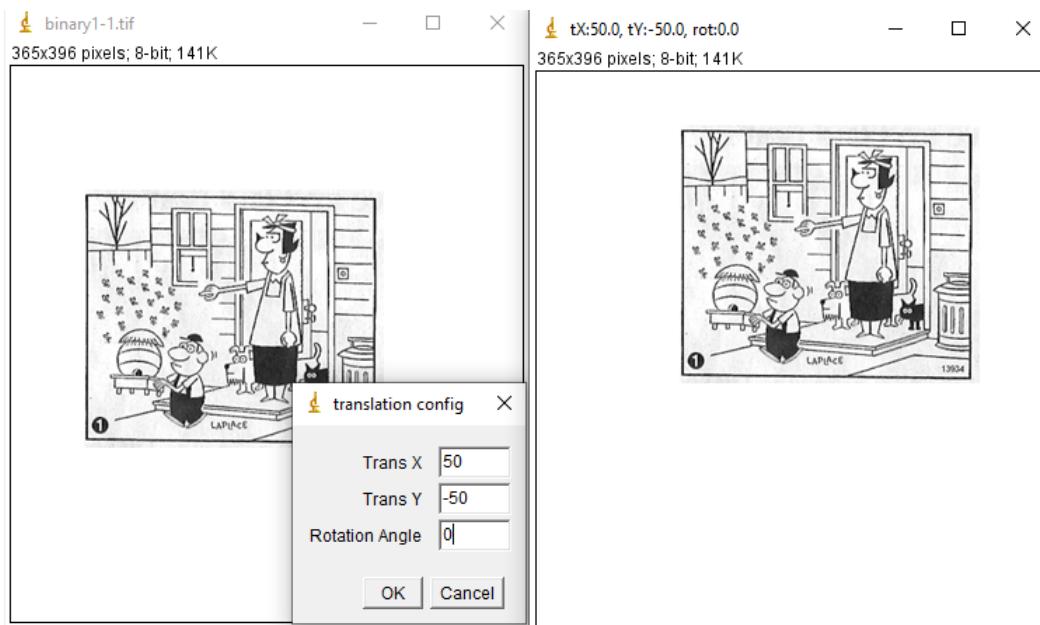
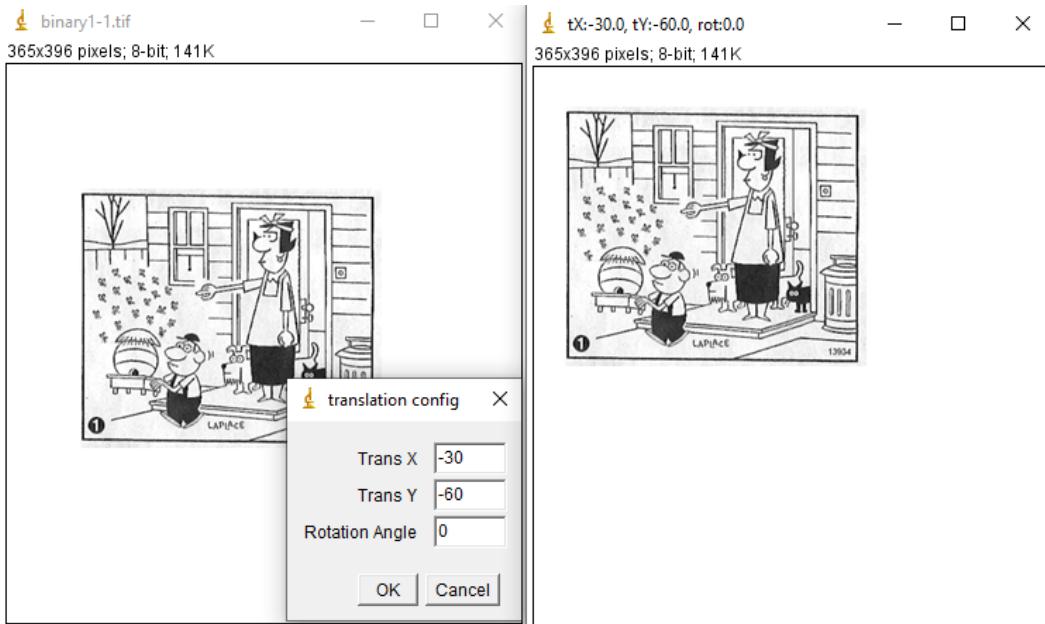
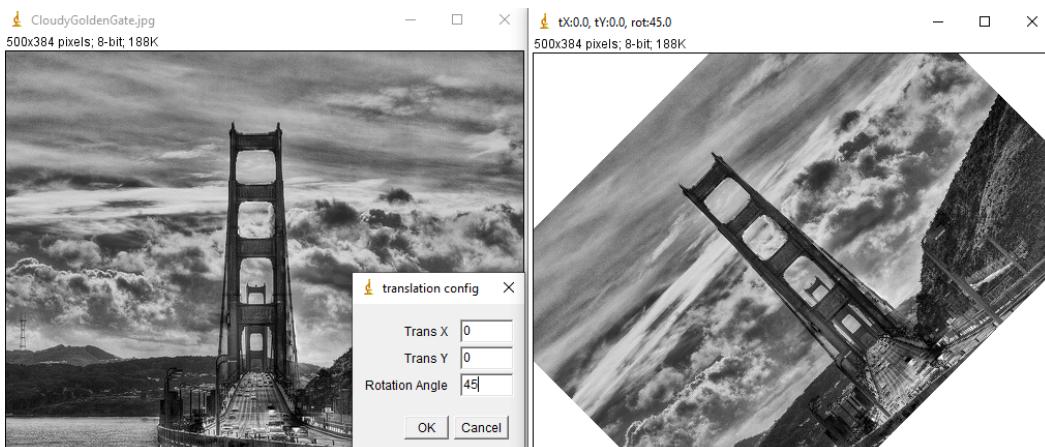


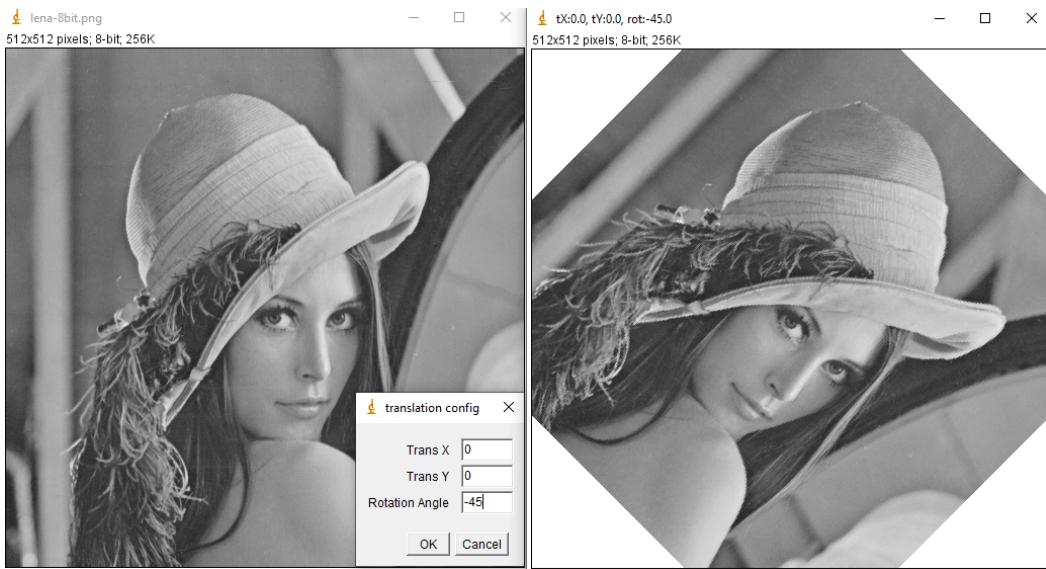
Abbildung 3.5: Translation mit NN Interpolation.



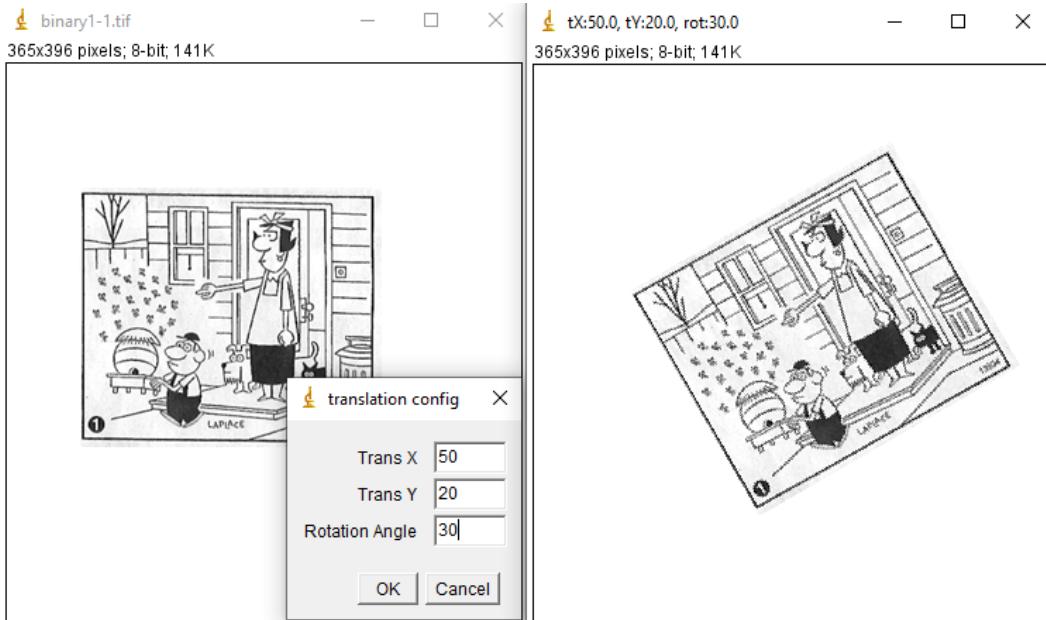
**Abbildung 3.6:** Translation mit NN Interpolation.



**Abbildung 3.7:** Rotation mit NN Interpolation.



**Abbildung 3.8:** Rotation mit NN Interpolation.



**Abbildung 3.9:** Translation und Rotation um die Bildmitte.

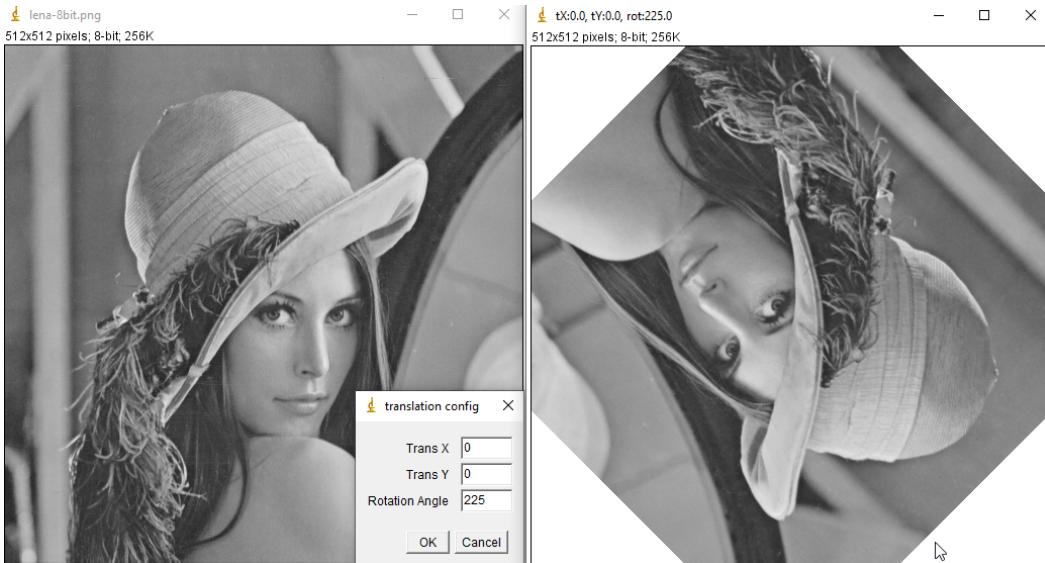


Abbildung 3.10: Rotation mit Bilinearer Interpolation.

### c) Lösungsidee

Es soll eine automatische Registrierung implementiert werden, was so viel bedeutet wie, es gibt 2 Bilder mit ähnlichem Inhalt aber mit unterschiedlicher Positionierung. Es gilt, eines davon so zu transformieren, dass es die gleiche Ausrichtung wie das Referenzbild bekommt. Im Idealfall sind sie kongruent und wenn nicht, dann zumindest mit einem geringen Errorwert. Dieser Errorwert wird mit der *Sum of squared errors* (SSE) Metrik berechnet. Das Finden der Position wird durch die zuvor implementierte Transformationsfunktion übernommen. Es wird jede mögliche Position, die die übergebenen Parameter erlauben, durchgeführt und jedes Mal nach einer erneuten Transformation die Errorrate neu berechnet. Wird ein neuer besserer Errorwert gefunden, dann werden die momentanen Parameter gespeichert und nach dem kompletten Durchlauf als Rückgabewerte in einem Array übergeben.

### Source Code

```

1 private double[] applyRegistration(double[][] inDataArrDbl, double[][] unregisteredImg, int width, int height, double[] registrationParams, String method) {
2
3     int numOfStepsTx = (int) registrationParams[0];
4     int numOfStepsTy = (int) registrationParams[1];
5     int numOfStepsRot = (int) registrationParams[2];
6     double stepWidthTx = registrationParams[3];
7     double stepWidthTy = registrationParams[4];
8     double stepWidthRot = registrationParams[5];
9
10    // size of search space: 11 x 11 x 11 ==> 1331 solutions
11    // one possible search candidate is (-4, 2.0, -8.0) for(Tx, Ty, Rot)
12    double minERR = 0.0;

```

```

13  if (method == "MI") {
14      minERR = getImgDiffMI(inDataArrDbl, unregisteredImg, width, height);
15  } else if (method == "SSE") {
16      minERR = getImgDiffSSE(inDataArrDbl, unregisteredImg, width, height);
17  }
18
19  double bestTx = 0.0;
20  double bestTy = 0.0;
21  double bestRot = 0.0;
22
23  double currTx = 0.0;
24  double currTy = 0.0;
25  double currRot = 0.0;
26
27  for (int txStep = -numOfStepsTx; txStep <= numOfStepsTx; txStep++) {
28      currTx = txStep * stepWidthTx;
29
30      for (int tyStep = -numOfStepsTy; tyStep <= numOfStepsTy; tyStep++) {
31          currTy = tyStep * stepWidthTy;
32
33          for (int rotStep = -numOfStepsRot; rotStep <= numOfStepsRot; rotStep++) {
34              currRot = rotStep * stepWidthRot;
35              double[][] registeredImg = transformImg(unregisteredImg, width, height,
36                  currTx, currTy, currRot, false);
37              double currErr = 0.0;
38              if (method == "MI") {
39                  currErr = getImgDiffMI(inDataArrDbl, registeredImg, width, height);
40              } else if (method == "SSE") {
41                  currErr = getImgDiffSSE(inDataArrDbl, registeredImg, width, height);
42              }
43
44              if ((method == "SSE" && currErr < minERR) || (method == "MI" && currErr >
45                  minERR)) { // check if better solution was found? if YES – replace the old solution
46                  minERR = currErr;
47                  bestTx = currTx;
48                  bestTy = currTy;
49                  bestRot = currRot;
50                  System.out.println(method + ": NEW best solution found with " + bestTx +
51                      ", " + bestTy + ", " + bestRot + " at ERR = " + minERR);
52              }
53          }
54      }
55  }

```

```

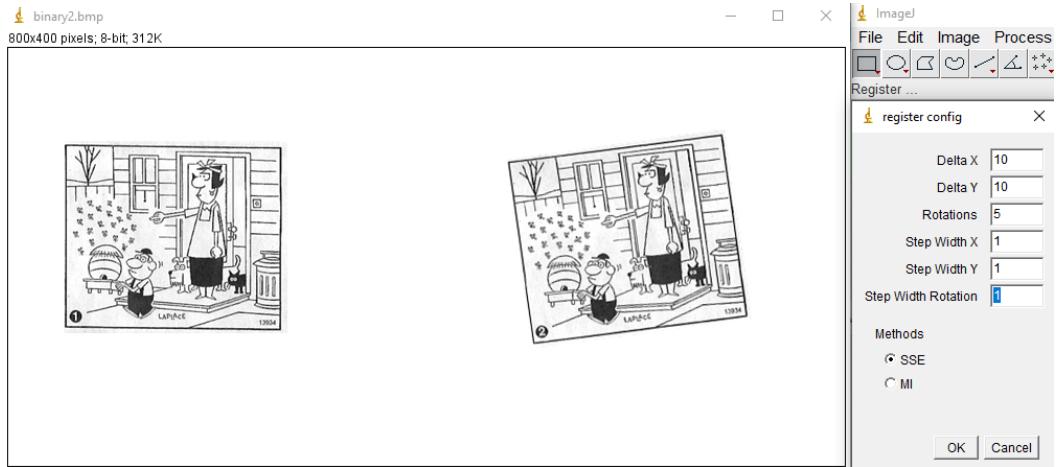
1 public double getImgDiffSSE(double[][] imgData1, double[][] imgData2, int width, int
2     height) {
3     double currentError = 0.0;
4     for(int x = 0; x < width; x++) {
5         for(int y = 0; y < height; y++) {
6             double diff = imgData1[x][y] - imgData2[x][y];
7             currentError += diff * diff;
8         }
9     }
10    return currentError;

```

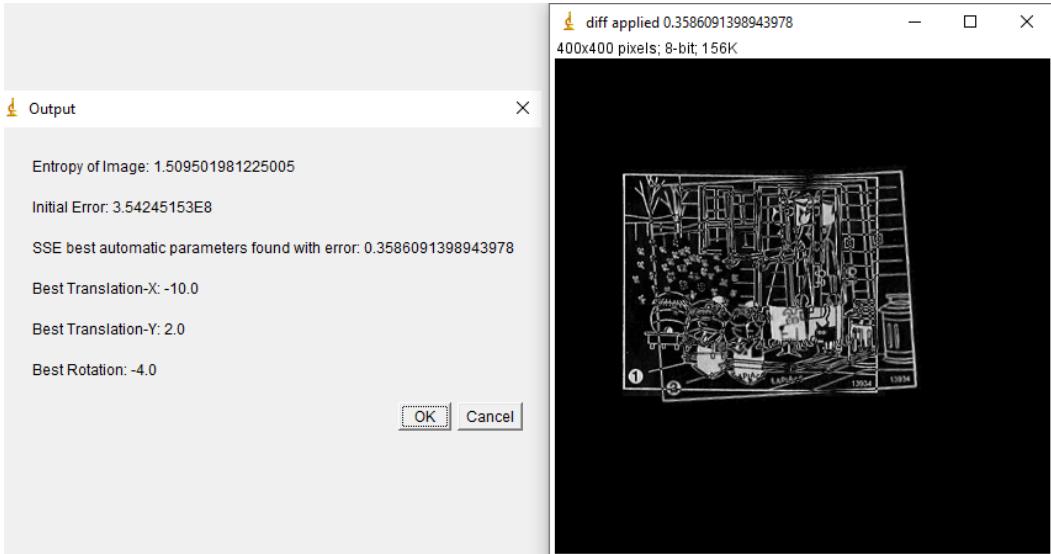
```
10 } //getImageDiffSSE
```

### Tests

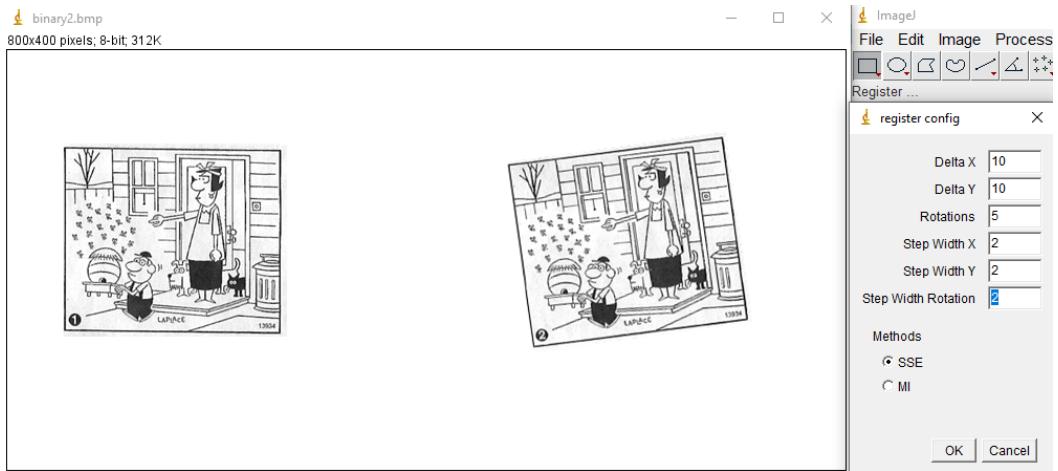
Das Ergebnis ist stark von den gewählten Parametern abhängig. Es werden die Bilder aus dem Projekt verwendet, geteilt und registriert.



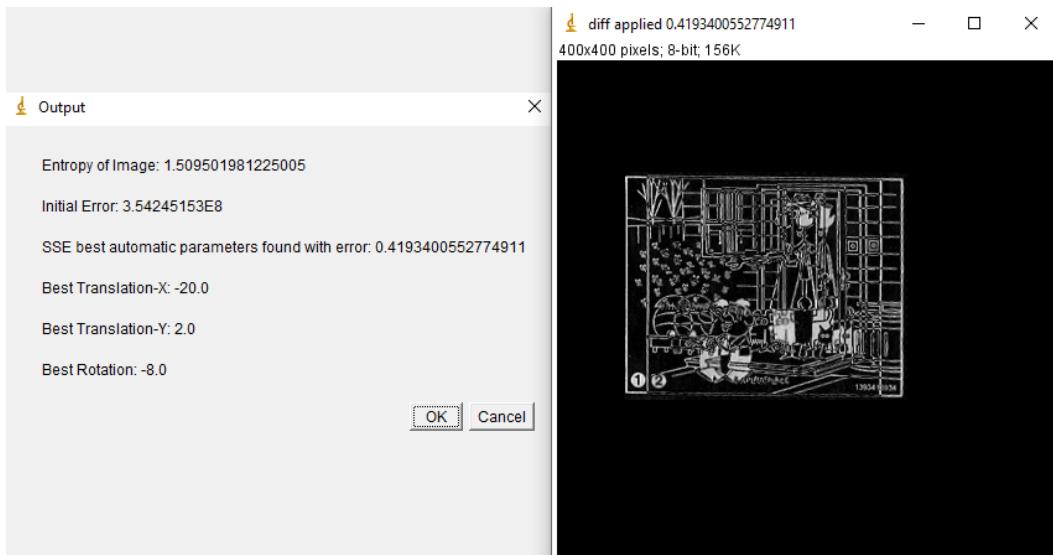
**Abbildung 3.11:** Experiment mit Schrittweite 1.



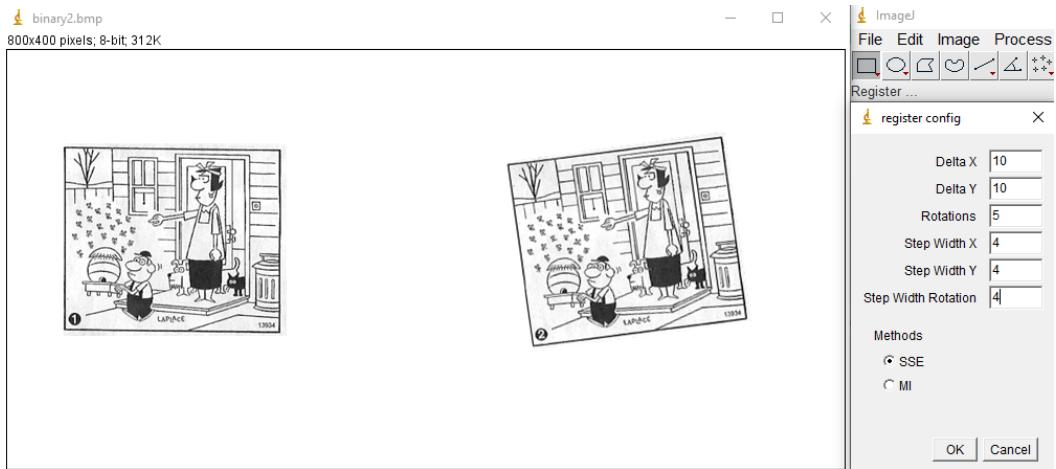
**Abbildung 3.12:** Ergebnis als Differenzenbild ist sowohl horizontal verschoben und verdreht.



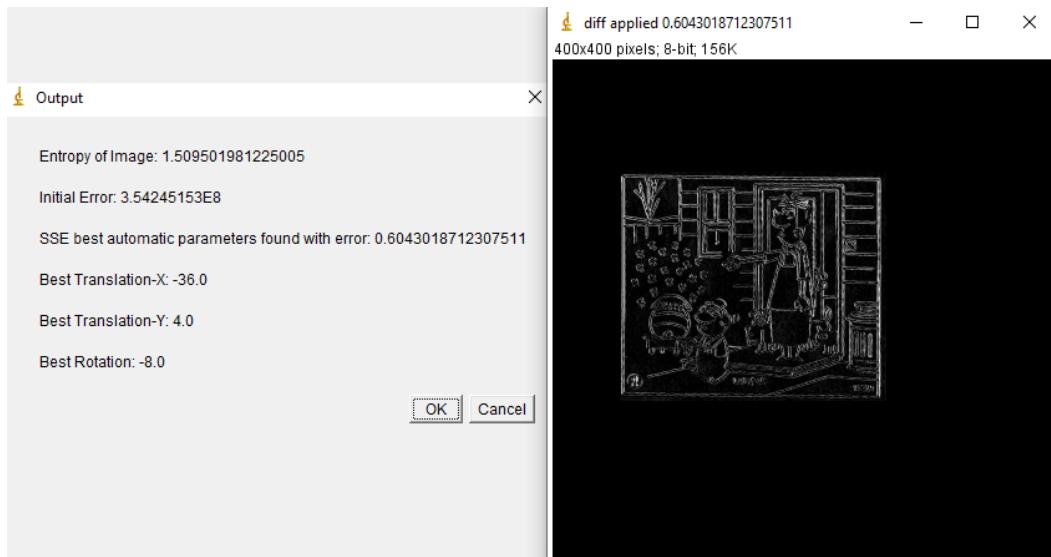
**Abbildung 3.13:** Experiment mit Schrittweite 2.



**Abbildung 3.14:** Ergebnis als Differenzenbild scheint bereits die korrekte Rotation aufzuweisen, jedoch noch horizontal verschoben.



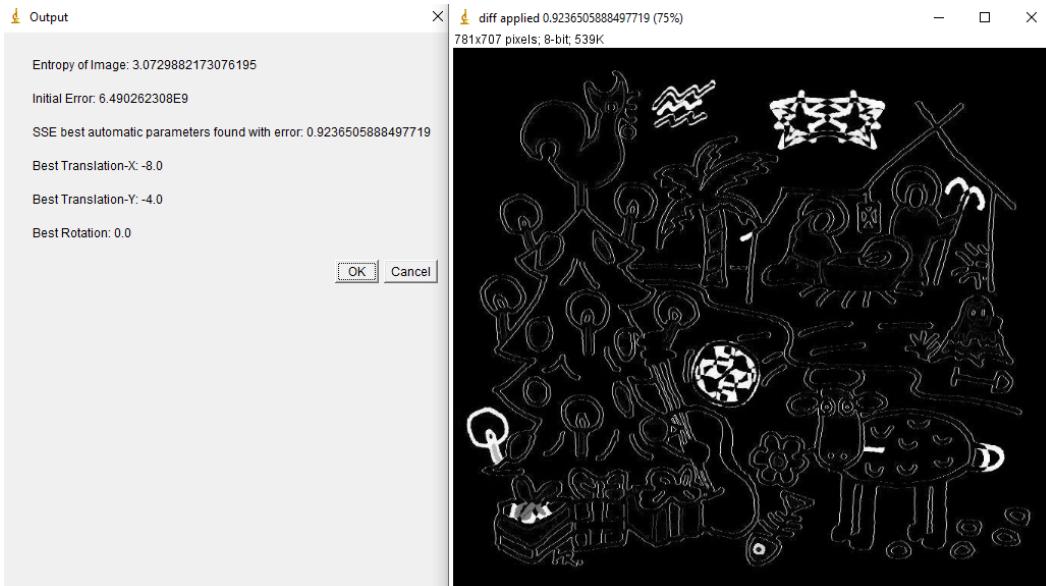
**Abbildung 3.15:** Experiment mit Schrittweite 4.



**Abbildung 3.16:** Sehr gute Überlagerung, noch nicht perfekt, aber das beste gefundene Resultat. Im Bereich der Bienen verdunkelt sich bereits das Differenzenbild.



**Abbildung 3.17:** Experiment mit Schrittweite 4 am Fehlersuchbild.



**Abbildung 3.18:** Die Fehler werden im Differenzenbild gut sichtbar.

#### d) Lösungsidee

Es soll nun anstelle der SSE Fehlermetrik die **Mutual Information** Metrik implementiert werden. Die bereits bekannte Shannon Entropie wird hier auf die Grauwerte des Bild-Histogramm errechnet. Dazu befinden sich im Template bereits passenden Methodenköpfe. Nämlich die Berechnung der Entropie eines Bildes `getEntropyOfImg` und von 2 Bildern `getEntropyOfImages`. Zum Berechnen der Entropie werden die Grauwerte auf ihr Auftreten gezählt und jeweils durch die Gesamtanzahl der im Bild vorhandenen Pixel dividiert. Damit erhält man die Wahrscheinlichkeit der einzelnen Grauwerte. Danach wird jede errechnete Wahrscheinlichkeit mit der Formel der Shannon-Entropie zur

Gesamtentropie aufsummiert. Diese Summe wird auch von der Funktion zurückgeliefert. Bei der Berechnung der Entropie von zwei Bildern, ist das ganz analog dazu, nur dass jetzt die Kombination der beiden Grauwerte an der gleichen Pixelposition der beiden Bilder gezählt wird. Dazu muss ein 2-dimensionales Array erstellt werden. Alles weitere unterscheidet sich nicht von der zuvor beschriebenen Methode.

Die Mutual Information tut jetzt nichts anderes als die Entropien der beiden einzelnen Bilder ( $H_a$  und  $H_b$ ) zu addieren und die Entropie der beiden gemeinsamen Bilder ( $H_{ab}$ ) abzuziehen. Je weniger Unordnung - Bilder stimmen überein - nun in  $H_{ab}$  ist, umso weniger wird abgezogen und der Ergebniswert wird größer. Das heißt, wir müssen bei Verwendung der Methoden unterscheiden. Muss bei SSE der Fehlerwert möglichst niedrig sein, so muss bei Mutual Information dieser Wert möglichst groß sein.

## Source Code

```

1 public double getImgDiffMI(double[][] imgData1, double[][] imgData2, int width, int
2   height) {
3   // Slides registration 5 – Slide 32:  $I(A,B)=H(B)-H(B/A)=H(B)+H(A)-H(A,B)$ 
4   double hA = getEntropyOfImg(imgData1, width, height);
5   double hB = getEntropyOfImg(imgData2, width, height);
6   double hAB = getEntropyOfImages(imgData1, imgData2, width, height);
7   return hB + hA - hAB;
8 } //getImgDiffMI
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38

```

```

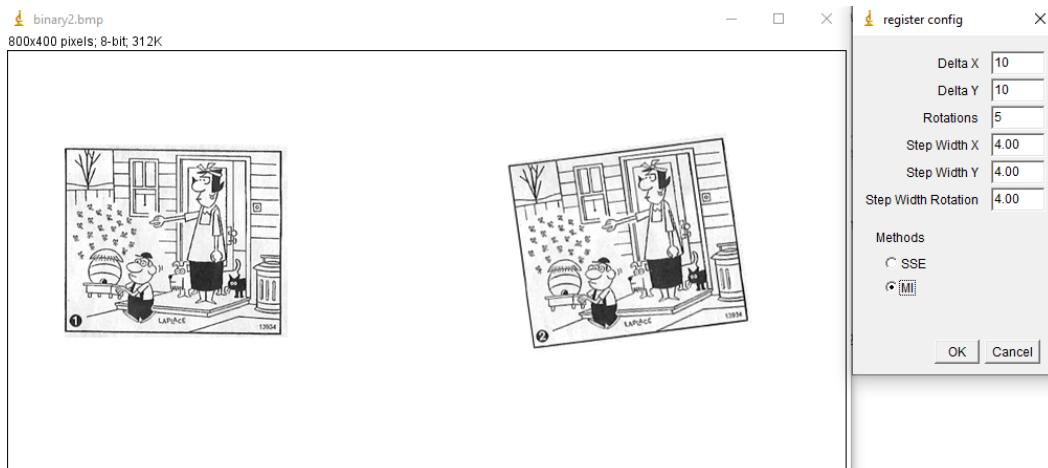
1 public double getEntropyOfImg(double[][] imgData, int width, int height) {
2   int maxValue = 255;
3   int[] greyValueCounts = new int[maxValue + 1];
4   for (int x = 0; x < width; x++) {
5     for (int y = 0; y < height; y++) {
6       int val = (int) imgData[x][y];
7       greyValueCounts[val]++;
8     }
9   }
10  double[] probabilities = new double[maxValue + 1];
11  double pixels = width * height;
12  for (int i = 0; i < probabilities.length; i++) {
13    probabilities[i] = greyValueCounts[i] / pixels;
14  }
15  // calc Entropy: sum(p * log_2 p)
16  double entropy = 0;
17  double logBase2 = Math.log(2);
18  for (int i = 0; i < probabilities.length; i++) {
19    double p = probabilities[i];
20    if (p > 0.0) {
21      double newP = p * (Math.log(p) / logBase2);
22      double newP = p * Math.log(p);
23      entropy += newP;
24    }
25  }
26  return -entropy;
27 } //getEntropyOfImg
28
29
30
31
32
33
34
35
36
37 public double getEntropyOfImages(double[][] imgA, double[][] imgB, int width, int
38   height) {
39   int maxValue = 255;
40 }
```

```

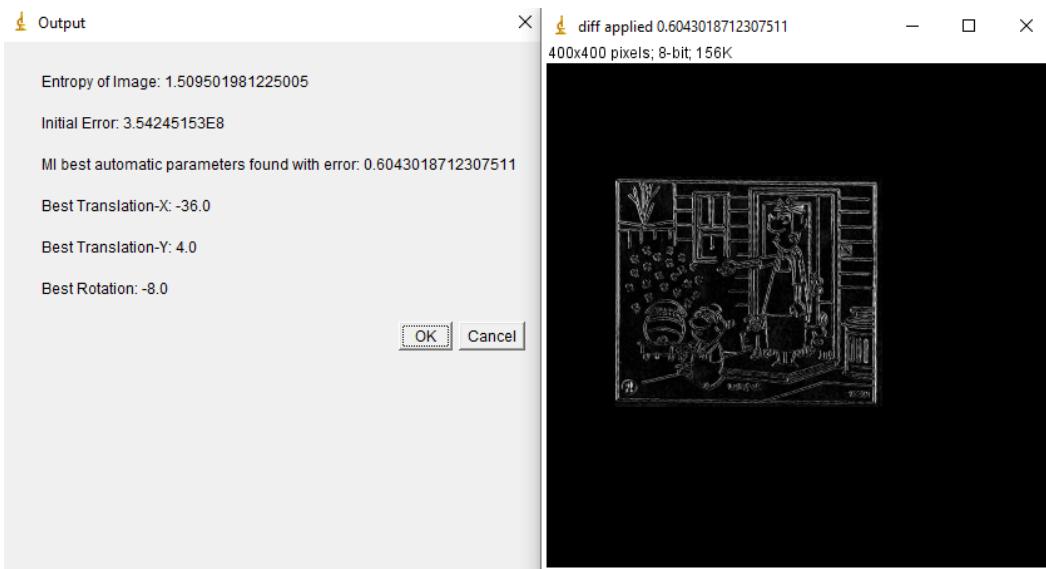
39  int[][] greyValueCounts = new int[maxValue + 1][maxValue + 1];
40  for (int x = 0; x < width; x++) {
41      for (int y = 0; y < height; y++) {
42          int valA = (int) imgA[x][y];
43          int valB = (int) imgB[x][y];
44          greyValueCounts[valA][valB]++;
45      }
46  }
47  double[][] probabilities = new double[maxValue + 1][maxValue + 1];
48  double pixels = width * height;
49  for (int i = 0; i < maxValue + 1; i++) {
50      for (int j = 0; j < maxValue + 1; j++) {
51          probabilities[i][j] = greyValueCounts[i][j] / pixels;
52      }
53  }
54  // calc Entropy: sum(p * log_2 p)
55  double entropy = 0;
56  double logBase2 = Math.log(2);
57  for (int i = 0; i < maxValue + 1; i++) {
58      for (int j = 0; j < maxValue + 1; j++) {
59          double p = probabilities[i][j];
60          if (p > 0.0) {
61              //double newP = p * (Math.log(p) / logBase2);
62              double newP = p * Math.log(p);
63              entropy += newP;
64          }
65      }
66  }
67  return -entropy;
68 }

```

## Tests



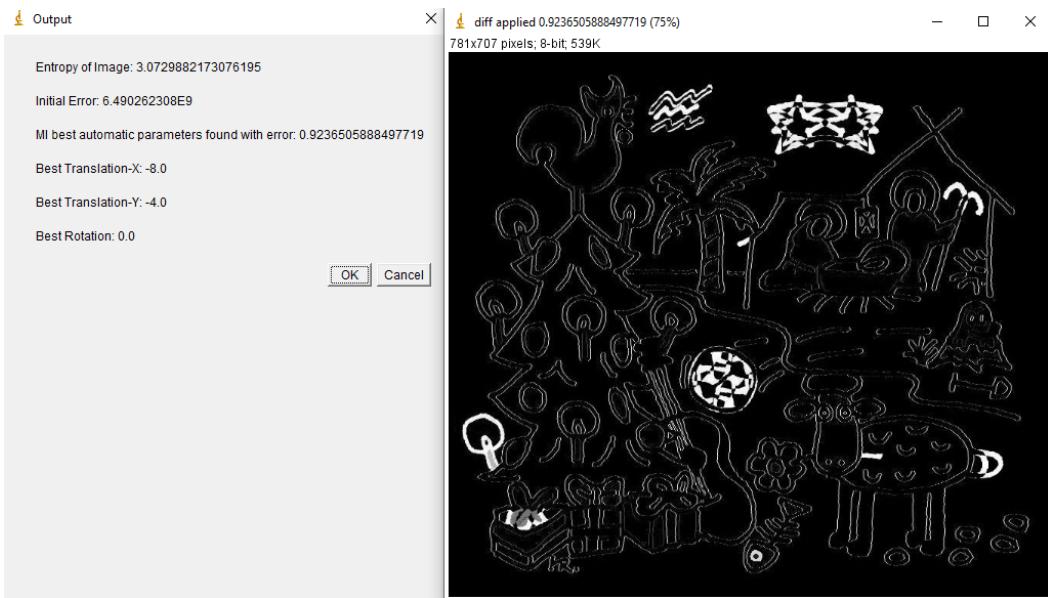
**Abbildung 3.19:** Mit den bereits gefundenen Parametern wird nun die Mutual Information Metrik zur Fehlerberechnung verwendet.



**Abbildung 3.20:** Selbes Resultat wie mit der SSE Metrik.



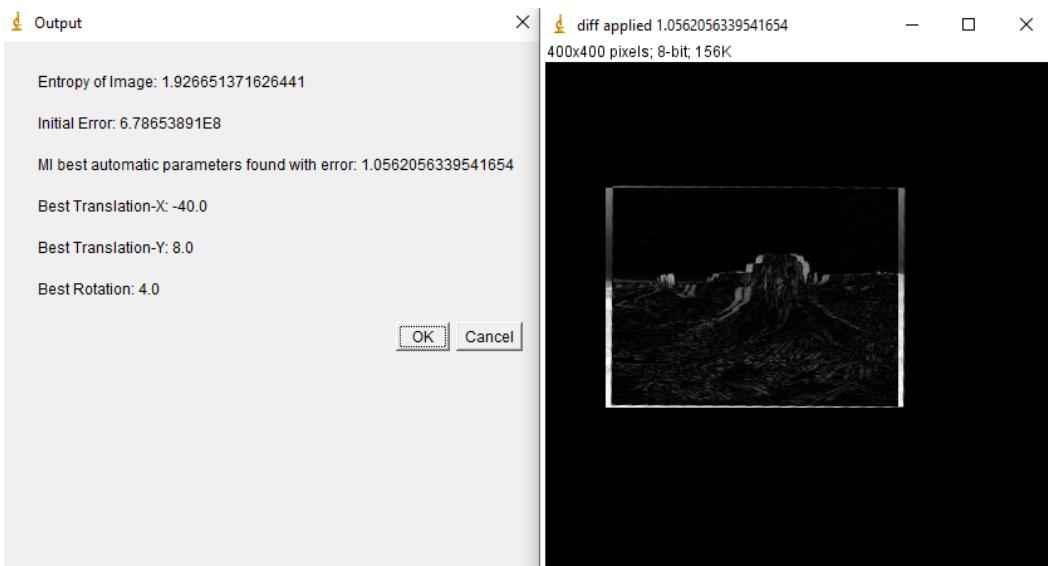
**Abbildung 3.21:** Experiment mit dem Fehlersuchbild.



**Abbildung 3.22:** Ergebnis als Differenzenbild auch hier kein Unterschied zur SSE Metrik zu erkennen.



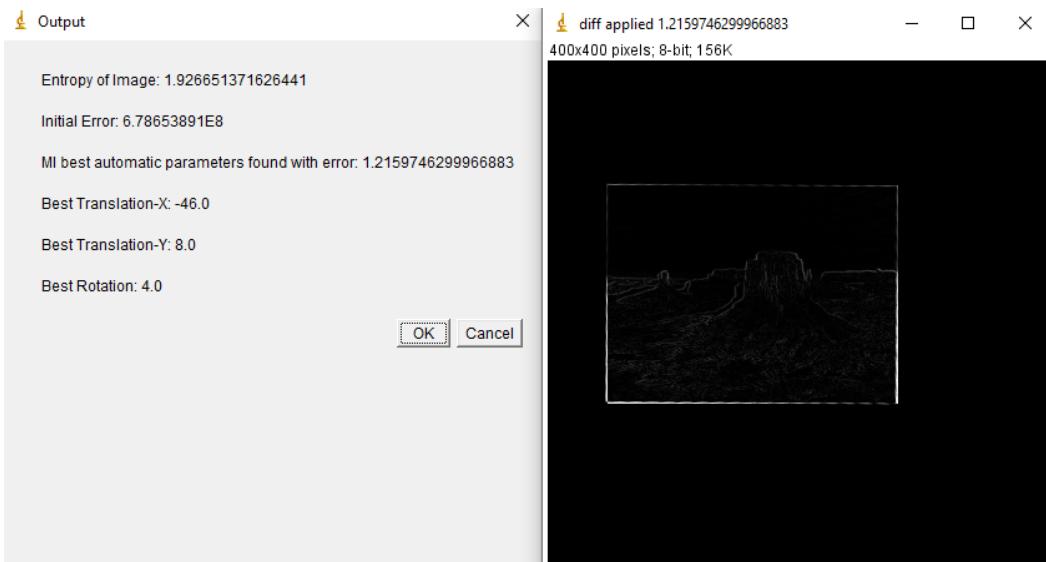
**Abbildung 3.23:** Experiment mit dem Canyon Bild.



**Abbildung 3.24:** Hier scheinen die Parameter, die zuvor gut gewählt waren, nicht ganz zu passen.



**Abbildung 3.25:** Experiment mit reduzierter Schrittweite und größerem Delta-X und -Y.



**Abbildung 3.26:** Ergebnis als Differenzenbild fällt jetzt besser aus.

e)

**Die Aufgabe mit der Distanz Map konnte nicht komplett implementiert werden, deshalb nicht als Versuch zählen!**

Ich habe alle relevanten Codeteile entfernt, wurde mit Prof. Zwettler besprochen.