Triangulierung / Tiefenwertberechnung

WS 22/23, Janis Reinelt

January 25, 2023

1 Einleitung

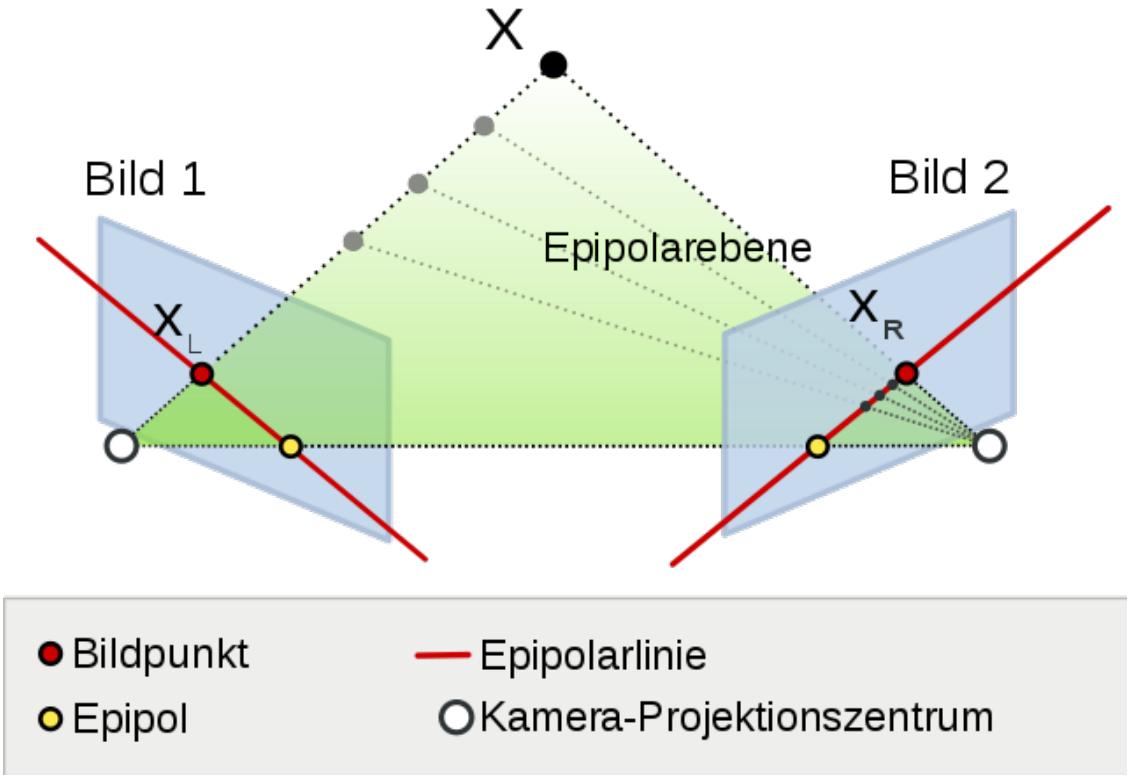
In der Bildverarbeitung stellt neben der Bildbearbeitung und der Computergrafik auch der Bereich der Computer Vision eine sehr große Rolle. Neben dem puren Erkennen von Objekten, stellt der Bereich der Tiefenwertberechnung ein weiteres komplexes Feld. Dabei werden Informationen aus einem oder mehreren Bildern dazu verwendet einen 2D-Punkt auf einem Bild einem 3D Punkt in der realen Welt zuzuordnen. Um die Auswertung zu erleichtern können auch Tiefensensoren verwendet werden. So wird zum Beispiel bei der Azure Kinect neben einer Kamera auch ein Tiefensensor verwendet um eine 3D-Konstruktion des Raumes oder die Pose eines Menschens zu erfassen.

Im Folgenden wird die Zuhilfenahme solcher Sensoren jedoch nicht berücksichtigt und sich einzig auf die Verwendung von korresponierenden Punkten in zwei Bildern beschränkt. Zudem wurden die verwendeten Kameras mittels eines Kalibrierungsmusters kalibriert und daraus deren Kameramatrizen berechnet.

In ihrem Buch *Multiple View Geometry*, beschreiben Hartley und Zisserman in Algorithmus 12.1 ein Verfahren, das mittels Triangulierung aus einem korrespondierenden 2D-Punktpaar einen möglichst genauen 3d-Punkt bestimmt. Dieser wurde im Folgenden implementiert.

2 Epipolargeometrie

Die hier zu Grunde liegende Theorie stammt aus der Epipolargeometrie. So wird jeder der beiden Punktlochkameras jeder 3D-Punkt durch die Linse auf den 2D Sensor projeziert insofern dieser im Sichtfeld liegt.



https://upload.wikimedia.org/wikipedia/commons/3/33/Epipolargeometrie3.svg

Ein Bildpunkt X wird dabei auf die Punkte X_R und X_L projeziert, indem ein Strahl von X zum jeweiligen Projektionszentrum O führt. Dies entspricht z.B. einem Lichstrahl, der ausgehend von X durch die Linse der Kamera auf den Sensor trifft. Die Bildebene auf der die Bildpunkte liegen sind dann die jeweigen Sensoren der Kameras.

Angenommen es wäre nur ein Bild verfügbar, so kann nicht genau bestimmt werden, wo X im zweiten Bild liegen würde.

Ist nun aber die Translation und Rotation der beiden Kameras zueinander bekannt, so kann der sog. Epipol bestimmt werden. Dieser gibt an, wo sich das jeweils andere Kamerazentrum im eigenen Bild (auf der Bildebene) befindet. Somit geben die Epipole (gelb) e_L und e_R die Projektion des rechten und linken Kamerazentrums auf die Bildebenen 1 und 2 an.

Daraus kann dann mittels X_L bzw. X_R und dem jeweiligen Epipol eine Epipolarlinie gebildet werden, auf der alle Punkte der Epipolarebene liegen. Aus den Vektoren $\vec{Oe}_L$ und $\vec{OX}_L$ lässt sich dann mittels Kreuzprodukt eine Normale $\vec{Oe}_L \times \vec{OX}_L$ bilden, für die dann wiederum $\vec{OX}_L \cdot (\vec{Oe}_L \times \vec{OX}_L) = 0$ gelten muss (senkrecht zueinander). Dieser Zusammenhang wird als *Epipolar Constraint* bezeichnet. Analog dazu auch bei der rechten Seite.

Diese Relation kann dann als

$$X_l^T \cdot T \cdot R \cdot X_r = 0$$

beschrieben werden, wobei R die Rotation und T die Rotation der linken zur rechten Kamera beschreiben und X_l und X_r Punkte im 3D Raum beschreiben. Weiter kann dann R und T als Essential Matrix $E = T \cdot R$ zusammengefasst werden. Durch dieses Verhältnis lassen sich nun 3D-Punkte aus dem Koordinatensystem von Kamera L zu dem von Kamera R und umgekehrt durch

E^{-1} berechnen. Eine Umkehrung von E mittels SVD ist einfach möglich, da T schiefsymmetrisch und R orthonormal ist.

Da X_L und X_L 3D-Punkte sind, die in den meisten Fällen nicht bekannt sein sollten, muss die Formel soweit umgeformt werden, sodass die 2D-Bildpunkte $x_L = [x_{L1}, x_{L2}, 1]^T$ und $x_R = [x_{R1}, x_{R2}, 1]^T$ transformiert werden können. Dafür wird die Projektion eines 3D-Punktes auf die 2D Bildebene benötigt. Diese Information enthält die Kameramatrix, auch Projektionsmatrix genannt. Da auch diese Parameter bei einem Bildpaar konstant sind, wird aus den beiden Kameramatrizen K_L, K_R und der Essentiellen Matrix dann die Fundamentalmatrix F gebildet. Somit gilt $E = K_L^T F K_R$. Eine Kalibrierung der Kamera und die Berechnung der Kameramatrix ist z.B. mittels eines Kalibrierungsmusters möglich. Dazu aber mehr im Kapitel Kalibrierung.

$$x_L^T \cdot K_L^{-T} \cdot E \cdot K_R^{-1} \cdot x_R = x_L^T \cdot F \cdot x_R = 0$$

Diese Fundementalmatrix projeziert somit einen Punkt des linken Bildes auf einen Punkt des rechten Bildes.

3 Problemstellung

In der Theorie gilt die oben gezeigte Epipolar Constraint immer für korrespondierende Punkte. Dies ist in der Realität durch Distortion, ungenaue Punkte, etc. häufig allerdings nicht der Fall. Schon bei einer minimalen Abweichung laufen die beidem Vektoren $\vec{x} = O\vec{X}_L$ und $\vec{x'} = O\vec{X}_R$ aneinander vorbei und bilden keinen Schnittpunkt X. Damit dennoch ein möglichst guter Punkt X im dreidimensionalen Raum berechnet werden kann, kommt das Prinzip der Triangulierung ins Spiel. Ein relativ einfacher Algorithmus zur Triangulierung eines Punktes X würde die zwei Vektoren aufstellen und dann jeweils die Stelle bestimmen, an denen der jeweils Andere senkrecht steht. Der Mittelpunkt dieser beiden Punkte würde dann als triangulierter Schnittpunkt X gelten.

[1] sehen das allerdings als eher schlecht an und schlagen den *Optimalen Triangulierungsalgorithmus (12.1)* vor. Dieser verwendet die Fundamentalmatrix um aus korrespondierenden Punktpaaren $x \leftrightarrow x'$ eine Fundamentalmatrix zu konstruieren, für die die Epipolbedingung zutrifft und somit besser geeignete homogene Punktpaare $\hat{x} \leftrightarrow \hat{x}'$ zu finden und diese dann zur Berechnung von X zu verwenden. # Experiment ## Vorbereitung Es wurden zwei Smartphones als Kameras verwendet, wobei eine Szene von beiden Kameras gleichzeitig erfasst wurde. Die Auflösung der erlangten Fotos wurde mittels *OpenCV* vor jeder Analyse auf $1600 \times 1200px$ skaliert. Beide Kameras speicherten bilder im $4:3$ Format. Die Ausrichtung zeigte von oben herab auf die Szene und die Epipole lagen nicht in der sichtbaren Bildebene.

3.0.1 Fundamentalmatrix

Zur Berechnung des Algorithmus wird eine initiale Fundamentalmatrix benötigt. Diese wird durch Verwenden der *OpenCV* Funktion `cv2.findFundamentalMat` und manuell ausgewählten Punktpaaren ($n = 10$) konstruiert.


Abb. 2: Das verwendete Bildpaar mit eingezeichneten Punkten.

Deweiteren wurde versucht analogh zu [2] mittels SIFT und Graubildern, korrespondierende Bildpunkte zu ermitteln. Aufgrund doch einiger fehlerhaften Zuordnungen wurden vorerst manuelle Punktpaare verwendet. Die in Kalibrierung verwendeten Ecken, wären ebenfalls mögliche Punktpaare.

Eine Berechnung aus den im Folgenden kalibrierten Kameramatrizen hätte auch verwendet werden können[1].

3.0.2 Kallibrierung

[1] geben an, dass etweder die Kameramatrizen oder F bekannt sein soll. Zur einfacheren Berechnung von X im letzten Schritt des Algorithmus, wurden beide Kameras kalibriert. Die erforderlichen Kameramatrizen könnten auch mittels einer Konstruktion via DLT und gemessenen 3D-Punkten ermittelt werden.

Zur Kalibrierung wurden die *OpenCV* Funktionen `cv2.findChessboardCorners` und `cv2.calibrateCamera` verwendet. Dafür wurde ein Schachbrettmuster ausgedruckt und in die Szene gelegt. Anschließend wurde ein Bild mit beiden Kameras gemacht. Anschließend wurde das Schachbrett in eine andere Position gebracht und erneut Fotos mit beiden Kameras gemacht. Dies wurde 10 mal durchgeführt und die Kamerapositionen nicht verändert.

Anschließend wurden mittels *OpenCV* die Ecken des Schachbretts identifiziert und an die Kalibrierungsfunktion übergeben. Diese ermittelte dann die intrinsische Kameramatrix, als auch die Rotations- und Translationsvektoren und Distortion der einzelnen Bilder. Da die verwendeten Smartphones bereits über eine einigermaßen gute Korrektur der Verzerrung haben, wurden die Bilder nicht entzerrt. Dies ist aber dennoch für den weiteren Verlauf zu empfehlen. Der Abstand der einzelnen Kacheln von $2cm$ wurde ebenfalls übergeben.

Aus den intrinsischen 3×3 Kameramatrizen K und K', und den bildspezifischen Translations (t)- und Rotationsvektoren(r) wurden dann die 3×4 Projektionsmatrizen P und P' berechnet, wobei die Rotationsmatrix R mittels Rodigues Rotationsformel aus r ermittelt wurde.

$$P = K \cdot W = K \cdot [R|t]$$

Das verwendete Bildpaar war Bildpaar 04 (Index 3).

Die beiden Projektionsmatrizen wurden kopiert und im Triangulierungsalgorithmus als Parameter verwendet.

3.1 Optimaler Triangulierungsalgorithmus

3.1.1 Allgemeiner Ablauf

Zum Start wurden die bereits erwähnten Bilder geladen und skaliert, die Kameramatrizen festgelegt und die korrespondierenden Punkte abgelegt.

Im Anschluss wurde mittels `HZTriangulation` eine Klasse für den Algorithmus mit den o.g. Parametern initialisiert. Bei der Initialisierung wird die Fundamentalmatrix F anhand der gegebenen Punkte berechnet. Das oben gezeigte Bild in Abb.2 wird hier anhand der gegebenen Bilder und Punkte mit Markern zur leichteren Auswertung gespeichert.

Für jedes 2D-Punktpaar wird der 3D-Punkt `X` berechnet und ausgegeben.

[25]:
```
pip install opencv-python numpy scipy
```

```
Requirement already satisfied: opencv-python in /opt/conda/lib/python3.10/site-
packages (4.7.0.68)
Requirement already satisfied: numpy in /opt/conda/lib/python3.10/site-packages
(1.23.5)
Requirement already satisfied: scipy in /opt/conda/lib/python3.10/site-packages
(1.10.0)
Note: you may need to restart the kernel to use updated packages.
```

[45]:
```
import cv2 as cv
import sys
import numpy as np
import hz_triangulation
from hz_triangulation import HZTriangulation
import importlib # importlib is a module from the standard library

np.set_printoptions(precision=7,suppress=True)
import hz_triangulation
importlib.reload(hz_triangulation)
from hz_triangulation import HZTriangulation

frame_size = (1600,1200)

imgL = cv.imread('assets/calibration/left-04.jpg')
imgR = cv.imread('assets/calibration/right-04.jpg')
imgL = cv.resize(imgL, frame_size)
imgR = cv.resize(imgR, frame_size)

K_l = np.array([
    [  -207.810101005,   -957.53264739 ,    919.420275048,     499058.
 ↪537873731],
```

```
    [   641.846642823,    177.844561471,    933.546841892,     625080.
 ↪797196914],
    [    -0.398103114,      0.004433558,      0.917329959,        776.
 ↪363780947]])
K_r = np.array([
    [   612.430485549,   -929.557380308,    804.197963442,     998847.
 ↪540699963],
    [   835.836177431,    546.965600104,    629.515655944,     476432.
 ↪545897397],
    [     0.000388   ,     -0.116635373,      0.993174727,        788.
 ↪766676859]])

pL = np.array([[714,834], [810,920], [850,1050], [580,365], [497,865],␣
 ↪[1022,1034], [991,829],  [968,1092], [596,1058], [618,796]], dtype=np.
 ↪float64)
pR = np.array([[859,547], [956,641], [989,782],  [734,46],  [627,571],␣
 ↪[1171,773],  [1151,554], [1113,830], [722,778],  [760,500]], dtype=np.
 ↪float64)


hz = HZTriangulation(imgL, imgR, pL, pR, K_l, K_r)
print("F:\n{hz.F}\n".format(**locals()))
hz.save_with_markers("assets/with-markers.jpg")

for i in range(0,len(pL)):
    X = hz.singlePointStep(i)
    print("{i} => x:\t{hz.x},\txp:\t{hz.xp},\tX: {X}".format(**locals()))
```

```
F:
[[-0.         0.0000009 -0.002265 ]
 [-0.0000006 -0.0000002  0.0017353]
 [ 0.0019145 -0.0020258  1.       ]]

0 => x: [714. 834.   1.],       xp:     [859. 547.   1.],       X: [  627.267571
-258.9133303 -1848.6675311     1.       ]
1 => x: [810. 920.   1.],       xp:     [956. 641.   1.],       X: [
454.5233996  -159.7912699 -1632.1528422     1.       ]
2 => x: [ 850. 1050.    1.],    xp:     [989. 782.   1.],       X: [
320.9613043  -115.4757738 -1475.1231199     1.       ]
3 => x: [580. 365.   1.],       xp:     [734.  46.   1.],       X:
[-8664.5147673  3151.0859893  8901.6578171     1.       ]
4 => x: [497. 865.   1.],       xp:     [627. 571.   1.],       X: [
645.1043037  -409.7981566 -1899.158308      1.       ]
5 => x: [1022. 1034.    1.],    xp:     [1171.  773.    1.],    X: [
305.3606612   -51.8577803 -1437.7224499     1.       ]
6 => x: [991. 829.   1.],       xp:     [1151.  554.    1.],    X: [
485.6241114   -72.9878131 -1627.1646123     1.       ]
```

```
7 => x: [ 968. 1092.    1.],    xp:     [1113.  830.    1.],    X: [  277.624249
-67.5876892 -1414.5166181     1.       ]
8 => x: [ 596. 1058.    1.],    xp:     [722. 778.   1.],       X: [
335.2991324  -214.4800002 -1507.0592085     1.       ]
9 => x: [618. 796.   1.],       xp:     [760. 500.   1.],       X: [
786.8408383  -382.4683594 -2058.3976973     1.       ]
```

3.1.2 Schritte für Punktpaare (Algorithmus)

Die Funktion `hz.singlePointStep(i)` nimmt jedes einzelen Punktpaar und evaluiert hier den 3D-Punkt X, sodass die Epipolarbedingung erfüllt ist.

Hierbei wird analog zu Hartley und Zisserman folgende Schritte ausgeführt: #### Erstellen von T und T' aus x und x' Wenn beide Bildpunkte mit den Epipolen korrespondieren, so liegen beide Punkte auf einer Linie zwischen den beiden Kamerazentren. Somit wäre es unmöglich, die genaue Position auf dieser Linie zu bestimmen. Deshalb wird hier angenommen, dass keiner der Bildpunkte auf dieser Linie liegt[1]. Durch den Versuchsaufbau ist dies zudem nicht möglich, da sich die Kameras nicht gegenseitig sehen.

Die Autoren merken zudem an, dass wenn nur ein Punkt auf dieser Linie liegt, sich dieser Punkt im Zentrum des anderen Kamerazentrums befinden muss. Deshalb gehen sie davon aus, dass keiner der beiden Bildpunkte auf einem Epipol liegt.

Somit können die beiden Bildpunkte auf den Urspurng $(0, 0, 1)^T$ mittels der Translationsmatrizen für x und x' verschoben werden.

Anwendung der Translation auf F Da sich durch die Translation der Bildpunkte in den Ursprung, diese auch auf die Fundamentalmatrix angewendet werden muss, wird F (`hz.Fp`) neuberechnet.

$$F' = {T'}^{-T} F T^{-1}$$

Berechnen der Epipole Die Epipole e und e' sollen zudem auf die x-Achse in den Punkten $(1, 0, f)^T$ und $(1, 0, f')^T$ transformiert werden.

Somit werden diese mittels SVD berechnet und auf $e_1^2 + e_2^2 = 1$ normalisiert.

Berechnung der Rotationsmatrizen Durch die berechneten Epipole können nun die Rotationsmatrizen R und R' aufgestellt werden.

Des weiternen, führte die Normalisierung dazu, dass $Re = (1, 0, e_3)^T$ und $R'e' = (1, 0, e_3')^T$ gilt. Die nun gebildeten Rotationsmatrizen rotieren somit die Epipole an die gewüschten Positionen $(1, 0, f)^T$ und $(1, 0, f')^T$ auf der x-Achse.

Anwendung der Rotation auf F' Analog zur Translation muss auch die Rotation der Bildpunkte in der Fundamentalmatrix widergespiegelt werden. Dies geschieht durch:

$$F'' = R' F' R^T$$

Die Form der Fundamentalmatrix entspricht somit der in [1] beschriebenen Form 12.3.

Setzen von Parametern Aufgrund der vorliegenden Form von F'' können die folgenden Parameter abgeleitet werden:

$$f = e3f' = e'_3 a = F''_{22} b = F''_{23} c = F''_{32} d = F''_{33}$$

Berechnen von Extrema Hartley und Zisserman verwenden nun den quadratischen Abstand (Squared Distance) um den Abstand d^2 einer Linie $l(t)$ durch den Punkt $(0, t, 1)^T$ und den Epipol $(1, 0, f)^T$ zum Ursprung in Abhängigkeit von t im linken Bild zu berechnen. Um die selbe Linie auch im rechten Bild zu erhalten wird die neue Fundamentalmatrix verwendet, sodass $l'(t) = F'' \cdot (0, t, 1)^T$. Auch hier wird eine Formel zur Berechnung des quadratischen Abstands aufgestellt.

Addiert man beide Abstandsgleichungen, so wird der Abstand minimal, wenn die Ableitung ein Minimum besitzt ($s'(t) = 0$). Die aus der Ableitung erhaltene polynomische Formel $g(t) = 0$ wird nun verwendet um 6 Nullstellen (roots) zu finden.

Berechnen der Kosten und t_{min} Der Realteil der berechneten Nullstellen t wird nun in die Kostenfunktion (Summe der quadratsichen Abstände $s(t)$) eingesetzt. Das t mit den geringsten Kosten wird als $t_{min} gewählt.

Somit ist die Linie $l(t)$ bei t_{min} am Nähesten zu den beiden Ursprüngen bzw der Punkt $(0, t_{min}, 1)$ der "optimale" zum Aufspannen von l und l'.

Ermitteln von $\hat{x}$ und $\hat{x}'$ Nun werden die optimalen Punkte $\hat{x}$ und $\hat{x}'$ gesucht. Diese weisen die kürzeste Distanz zum Urspurng auf, da x und x' im jeweilgen Ursprung liegen und deshalb auch nach dem Abstand zu dem jeweilgen Ursprung minimiert wurde.

Dafür wird für $l(t_{min})$ und $l'(t_{min})$ ermittelt und die Nähesten Punkte $\hat{x}$ und $\hat{x}'$ zum Ursprung mittels $(a, b, c) \rightarrow (-ac, -bc, a^2 + b^2)$ ermittelt.

Retransformation von $\hat{x}$ und $\hat{x}'$ Die zu Beginn vorgenommen Translationen und Rotationen werden nun an $\hat{x} = T^{-1} R^T \hat{x}$ und $\hat{x}' = {T'}^{-1} {R'}^T \hat{x}'$ rückgänig gemacht.

Die optimierten Koordinaten bleiben homogene Koordinaten.

Berechnung von $\hat{X}$ Um nun den 3D-Punkt $\hat{X}$ zu berechnen muss eine Projektion von 2D-Punkten zu 3D-Weltpunkten erfolgen.

Die von den Autoren genannte Option nennt die Direct Linear Transformation als Möglichkeit die intrinsischen und extrinsichen Eigenschaften zu schätzen.

Im Implementierungsbeispiel wurden allerdings die kalibrierten Kameramatrizen verwendet und A aufzustellen und mittels SVD zu lösen. Die resultierende homogene 3D Koordinate wurde normalisiert, sodass $\hat{X}_4 = 1$.

4 Auswertung

Für die Auswerung wurden die Ergebnisse aus Allgemeiner Ablauf verwendet.

4.1 Tiefenwertberechnung

Bei den Tiefenwerten fällt auf, dass sich die meisten zwischen $-2000 < \hat{X}_3 < -1200$ bewegen. Da die Kameramatrix in mm kalibriert wurde, würde das einer Entfernung von $1,2m$ bis $2m$ entsprechen. Ein Ausreißer mit komplett umgekehrten Vorzeichen und viel höheren Werten wurde auch verzeichnet.

Da aber alle Punkte auf dem Tisch liegen und eine Ansicht von Oben auf die Szene gewählt wurde, scheint die nicht korrekt wiedergespiegelt worden zu sein. Relativ konstante Werte ohne große Schwankungen für z wären hier zu erwarten gewesen.

Betrachtet man zusätzlich die Absolutwerte, fällt auf, dass alle Werte zu hoch liegen. So war der Abstand der Kameras zur Tischoberfläche nur ca $95cm$.

4.2 Verschiebung in X und Y Richtung

Die Maße des Schachbrettblattes entsprechen denen eines herkömmlichen Din A4 Blattes ($297mmx210mm$). Die Seiten des Schachbretts vermessen genau $160mm$. Vergleicht man die Distanz zwischen den einzelnen Punkten (6 oben rechts, 7 unten rechts, 8 unten links, 9 oben links) ergeben sich folgende Ergebnisse:

[30]:
```
print("6 to 7:\t", hz.dist(points[6],points[7]))
print("7 to 8:\t", hz.dist(points[7],points[8]))
print("8 to 9:\t", hz.dist(points[8],points[9]))
print("9 to 6:\t", hz.dist(points[9],points[6]))
```

```
6 to 7:  297.5101234030627
7 to 8:  182.94227025044992
8 to 9:  732.177663075534
9 to 6:  610.3045950650032
```

Hierbei fällt auf, dass vor allem Punkt 9 stark abweicht. Jedoch stimmen die gemessenen Längenverhältnisse nicht mit den relationen des Blattes überein. Dennoch liegen die Größenordnungen schon in dem richtigen Bereich.

4.3 Vergleich mit OpenCV

[50]:
```
points = hz.triangulateOpenCv().T
for i in range(0,points.shape[0]):
    points[i] = points[i]/points[i,-1]
points
```

[50]:
```
array([[-222.6716403,  170.5000864,  995.1109601,    1.       ],
       [ -21.5770931,   77.9548142, 1156.6826357,    1.       ],
       [ 264.8600346,  129.8966412, 1596.4579607,    1.       ],
       [-615.2533396,   69.8224837,  133.4201987,    1.       ],
       [-390.8671313,  646.8826604, 1332.9028462,    1.       ],
       [ 377.4314567, -216.8630453, 1440.3396845,    1.       ],
       [ -20.4272389, -257.0611301,  866.7872203,    1.       ],
       [ 471.6074753,  -73.2752909, 1674.6758365,    1.       ],
```

```
[  54.8282256,  766.2868738, 2031.7476504,    1.       ],
[-353.4672183,  300.8178766,  976.0993624,    1.       ]])
```

Hier wurden die 3D Koordinaten mittels `cv.triangulatePoints` ermittelt.

Auch hier liegen die Absolutwerte ähnlich weit weg von der erwarteten Entfernung in z Richtung und auch in die anderen beiden Richtungen.

Dies ist sogar noch extremer bemerkbar beim Betrachten der Abstände der Blattecken. Ein korrektes Verhältnis zwischen den Abständen ist auch hier nicht zu erkennen.

Auffällig ist, dass die meisten z-Werte hier positiv sind und sich über x und y mit positiven und negativen Werten mehr verteilen.

[51]:
```
print("6 to 7:\t", hz.dist(points[6],points[7]))
print("7 to 8:\t", hz.dist(points[7],points[8]))
print("8 to 9:\t", hz.dist(points[8],points[9]))
print("9 to 6:\t", hz.dist(points[9],points[6]))
```

```
6 to 7:  963.6178759836874
7 to 8:  1003.0303343456038
8 to 9:  1223.829998310355
9 to 6:  658.8579197681861
```

5 Diskussion

Der hier gezeigte Algorithmus ist stark von den gewählten Punkten und der Kamerakalibrierung abhängig. Da auch im *OpenCV* Ansatz einige Unstimmigkeiten zu finden sind, lässt dies auf ein Problem bei den Eingangsparametern schließen.

Eine mögliche Erklärung könnten einige ungenaue Angabe der korrespondierenden Punkte dienen. Jedoch brachte die Verwendung von SIFT in ersten Versuchen keine merkliche Besserung.

Eine weitere und sehr wahrscheinliche Ursache könnte die Kalibrierung der Kamera darstellen. Da beim Kalibrieren, das Blatt nicht allzu stark bewegt wurde und vor allem wenig seitlich gekippt, können die Projektionsmatrizen den Grund für die großen Unterschiede darstellen. Desweiteren war das verwendete Blatt nicht besonders stabil und bildete somit manchmal eine leichte Krümmung. Es wird daher empfohlen die Kalibrierung erneut durchzuführen.

Desweiteren kann eine Verzerrung durch die Kamera einen großen Einfluss auf die Triangulierung haben, weshalb ein Entzerren während der Kalibrierung ebenfalls eine Lösung darstellen könnte.

Zusätzlich war das Muster des verwendeten Schachbrett quadratisch, was eventuell auch einen Einfluss auf die Kalibrierung der Kamera haben könnte. Dazu konnte aber aktuell noch nichts gefunden werden.

6 Quellen

[1] Hartley/Zisserman, Multiple View Geometry, p.310-318

[2] https://www.youtube.com/watch?v=UZlRhEUWSas

[3] https://en.wikipedia.org/wiki/Triangulation_(computer_vision)

[4] https://gist.github.com/cr333/0d0e6c256f93c0ed7bd2

[5] https://uni-tuebingen.de/fakultaeten/mathematisch-naturwissenschaftliche-fakultaet/fachbereiche/informatik/lehrstuehle/autonomous-vision/lectures/computer-vision/

[6] https://docs.opencv.org/3.4/da/de9/tutorial_py_epipolar_geometry.html

[7] https://www.changjiangcai.com/files/text-books/Richard_Hartley_Andrew_Zisserman-Multiple_View_Geometry_in_Computer_Vision-EN.pdf

[8] https://medium.com/@insight-in-plain-sight/estimating-the-homography-matrix-with-the-direct-linear-transform-dlt-ec6bbb82ee2b

[9] https://glowingpython.blogspot.com/2011/06/svd-decomposition-with-numpy.html

[ ]: