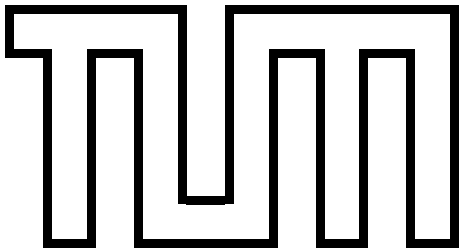


Technische Universität München

Fakultät für Informatik

Forschungs- und Lehreinheit Informatik VI

Master's Thesis in
Informatik

Direkte Schätzung der Lageveränderung mit einer monokularen Kamera

Elmar Mair

Aufgabensteller: Univ.-Prof. Dr.-Ing. Darius Burschka
Betreuer: Univ.-Prof. Dr.-Ing. Darius Burschka

Abgabedatum: 12. September 2007

Ich versichere, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Datum: Unterschrift:

Inhaltsverzeichnis

Kapitel 1

Kurzfassung (Abstract)

1.1 Deutsch

Um einen Roboter zu stabilisieren bzw. um grobe Bewegungen durchführen zu können, reicht es aus, die relative Lage des Roboters zu bestimmen. Bislang wurden in den meisten Anwendungen aktive Sensoren wie z.B. Lasersensoren verwendet. Oft wäre es aber von Vorteil anhand einer einfachen Kamera eine zuverlässige und echtzeitfähige relative Lokalisierung zu erreichen.
In Rahmen dieser Masterarbeit wurde ein Verfahren entwickelt, das die Rotationsänderung und Translationsrichtung aus einer Bildsequenz schätzt. Ziel des Algorithmus ist es dabei echtzeitfähig zu sein, im dreidimensionalen Raum zu operieren und keine Vorkenntnisse der Umgebung in Anspruch zu nehmen.
Die Rotation wird anhand des Arun-Algorithmus berechnet, die Translation über den Zentroiden der gewichteten Schnittpunkwolke des optischen Flusses. Beide Verfahren sind in den RANSAC-Algorithmus eingebettet. Gleichzeitig findet eine Hinderniserkennung statt, um eventuelle Gefahren in der Umgebung zu lokalisieren und eine mögliche Ausweichrichtung vorzuschlagen. Dem Algorithmus liegt die wichtige Annahme zugrunde, dass Punkte alleine durch Translation die diskretisierte Position im Bild nicht verändern, wenn sie weit genug von der Kamera entfernt sind.
In dieser schriftlichen Ausarbeitung wird der Algorithmus im Detail erklärt und mit ver-

wandten Arbeiten verglichen. Anschließend werden auch Simulations- und Testergebnisse vorgestellt.

1.2 English

To stabilize a robot or to execute a coarse movement, it is sufficient to get a relative position of the robot. So far, mainly active sensors, like a laser sensor, have been used to fullfill this task. However, often it would be of advantage and useful to calculate accurate and in real time the relative location with only one camera.
Within this Master's thesis an algorithm has been developed, which is able to calculate the rotation and the translation out of a frame sequence. This algorithm was aimed to operate in 3D space without any apriori knowledge and it should be real time capable.
The rotation is calculated by the so called Arun-algorithm and the Translation is calculated in a traditional way by the centroid of a weighted point cloud of the optical flow. Both practices are used in a RANSAC-framework. Furthermore, an obstacle recognition has been implemented to be able to avoid imminent dangers in the environment and to get a swerve direction. The algorithm is based on the assumption that points, far away from the camera, are affected in there pixel-discretized camera position only by the rotational component of any movement.
In this written work the algorithm becomes explained in detail and compared with some related work. The thesis closes with the results of some tests and simulations.

Kapitel 2

Danksagung

Hier gilt es zu allererst den Aufgabensteller, Betreuer und vor allem auch Mitwirkenden der Arbeit zu erwähnen, Prof. Darius Burschka. Durch seine freundliche und zuvorkommende Unterstützung habe ich nie die Freude an der Arbeit verloren. Zudem wäre der Algorithmus ohne seine genialen Einfälle so gar nicht zustande gekommen.
Weiters möchte ich den Mitarbeitern am Lehrstuhl VI (Robotik und Echtzeitsysteme) für Ihre Freundlichkeit danken. Ich habe mich jederzeit willkommen gefühlt.
Ich möchte mich an dieser Stelle auch bei allen Mitmenschen in meinem persönlichen Umfeld bedanken, die mich jederzeit unterstützt haben und die ich im Laufe der Arbeit sicher etwas vernachlässigt habe.

Danke.

Kapitel 3

Motivation

Robotern wird immer größere Bedeutung zugewiesen. Bereits seit längerem werden diese autonomen Maschinen erfolgreich in Fabriken eingesetzt. Die technologische Entwicklung und die daraus resultierende höhere Genauigkeit, Zuverlässigkeit und Lebensdauer der Geräte lässt die Zahl der Industrieroboter enorm ansteigen. Bis heute sind Manipulatoren weitgehend fix an einer Stelle verankert, von welcher aus sie operieren. So ist es auch nur notwendig, die Winkelstellung der Gelenke zu messen, um die Position des Roboters zu bestimmen. Erst seit kurzem finden auch zunehmend mobile, unbemannte Land-, aber auch Wasser- und Luftfahrzeuge ihren Einsatz. Eine größere Rechenleistung und leistungsfähigere Batterien ermöglichten diese neue Freiheit der Roboter. Meistens sind aber weiterhin Raum, Rechenleistung und Energieversorung stark begrenzt und es liegt an der Software aus wenig viel zu machen. Es gilt mehr denn je, effektive, zuverlässige Algorithmen zu entwickeln, die mit wenigen Ressourcen so viel Informationen wie nur möglich beschaffen.
Ein Bereich der Robotik bzw. Sensorik, der zunehmend an Bedeutung gewinnt, ist die computergestützte Bildverarbeitung. Eine Kamera stellt eine enorme Informationsquelle dar, dessen Potential uns durch die Augen und das Gehirn vorgehalten wird. Leider ist es der Wissenschaft noch nicht gelungen, in der Auswertung der Bildinformationen ähnliche Resultate wie beim Menschen zu erzielen. Zwar ist die Rechenleistung aktueller Prozessoren längst nicht vergleichbar mit jener des Gehirns, doch werden große Fortschritte in diesem Gebiet erzielt.

So galt es im Rahmen dieser Masterarbeit einen Algorithmus zu entwickeln, zu implementieren und zu testen, der mit geringem Rechenaufwand aus einer einzelnen Kamera möglichst viel Information über die Bewegung der Kamera herausholt.
Zunächst wird in diesem Kapitel auf den Hintergrund der Arbeit eingegangen. Gleichzeitig werden die verschiedenen Punkte aufgezeigt, die bei der Lösung des Problems in Betracht gezogen wurden. Die genaue Funktionsweise und die Details über den Aufbau des Algorithmus werden erst in Kapitel 5 beschrieben. Zuvor wird in Kapitel 4 noch auf grundlegendes Wissen hingewiesen, um das Verständnis des Verfahrens zu erleichtern. In Kapitel 6 wird auf die Entwicklung der Software eingegangen und das Vorgehen bei der Implementierung erklärt. Vor der Zusammenfassung und dem Resümee werden die Testergebnisse in Kapitel 7 vorgestellt und entsprechende Schlüsse gezogen.

3.1 Hintergrund der Arbeit

Das DLR (Deutsches Luft- und Raumfahrtszentrum) beschäftigt sich seit einiger Zeit mit der Entwicklung von Dronen. In diesem Rahmen wurde ein Quadrokopter und ein Zeppelin entwickelt, die jeweils über eine Kamera verfügen. Doch vor allem beim Quadrokopter, der nur in der Lage ist, geringe Lasten zu tragen, wird die begrenzte Rechenleistung zum Problem. Die Steuerungsalgorithmen werden daher größtenteils in der Bodenstation durchgeführt und die Sensor- und Steuersignale zwischen Drone und Kontrollrechner übertragen.
Doch was passiert, wenn diese Verbindung vielleicht nur kurz oder aber auch für längere Zeit ausfällt? Die Regelschleife alleine reicht nicht aus, um das Fluggerät zu stabilisieren. Es muss den UAVs (unmanned air vehicles) also möglich sein, über einige Zeit auch ohne externe Signale auszukommen, wobei die einzige Information, die man erhält, die Kameradaten sind. Ein Hauptaugenmerk gilt dabei auch der begrenzten Rechenleistung, die es bei der Entwicklung einer Lösung stets zu berücksichtigen gibt. Den Robotern soll ermöglicht werden, folgende Navigations-Aufgaben unabhängig von der Bodenstation durchführen zu können:

a) eine Eigenstabilisierung

(a) Quadrokopter des DLR (b) Zeppelin des DLR

Abbildung 3.1: Auf diesen beiden Fluggeräten soll das vorgestellte Verfahren in naher Zukunft laufen.

b) ein vordefiniertes Verhalten auszuführen (z.B. Position halten, landen, ...)

c) Hindernissen erfolgreich auszustellen

Man könnte auch sagen, dass sozusagen die Grundlage für ein reaktives Verhalten erzeugt werden soll, das es dem Flugobjekt ermöglicht, allein über die Daten einer Kamera die Position zu halten bzw. Hindernissen auszustellen und sicher zu landen.
Dieser Hintergrund gilt somit als initiale Anregung einen Algorithmus zu entwickeln, der die oben gestellte Aufgabe lösen, aber nicht auf diese beschränkt sein soll. So wird das Verfahren möglichst generisch gehalten, um es für jegliche Navigations-Aufgabe in 3D und 2D verwenden zu können. Die allgemeine Aufgabe, die somit dieser Masterarbeit zugrunde liegt, wird im folgenden Abschnitt beschrieben.

3.2 Problemformulierung

Ziel ist es, einen Algorithmus zu entwickeln, dem es möglich ist, aus der Bildsequenz einer **einzelnen, optischen Kamera** die eigene Bewegung zu schätzen. Dabei dürfen keine komplexen Algorithmen zum Einsatz kommen, da mit einer begrenzten Rechenleistung zu rechnen ist und die **Echtzeitfähigkeit** des Verfahrens vorausgesetzt wird. Weiters sollen **keine Rückschlüsse auf Position und Ziel** notwendig sein. Der Algorithmus soll

zudem **keine Anfangsinformation** benötigen.
Da ohne irgendwelche Vorkenntnisse über die Umgebung keine absolute Positionsbestimmung mit nur einer Kamera möglich ist, soll die Verwendung zusätzlicher Sensorik nicht ausgeschlossen werden. Um die gewonnenen Daten in einem Kalmanfilter mit anderen Sensordaten erfolgreich fusionieren zu können, ist es notwendig, dass das Verfahren ein Zuverlässigkeitsmaß seiner Berechnung liefert.
Es gilt somit folgende Teilprobleme zu lösen:

a) Bestimmung der Rotation

b) Bestimmung der Translationsrichtung

c) Erkennen von Hindernissen

d) Maß für die Zuverlässigkeit der Schätzung, sei es für die Rotation, wie auch für die Translation

3.2.1 Biologie als Vorbild

Beim Sammeln von Lösungsmöglichkeiten wurde ein Abstecher in die Biologie gemacht, denn Mutter Natur hat bereits des öfteren (einfachste) Antworten auf technische Probleme geliefert. Und auch für den beschriebenen Anwendungsfall gibt es Beispiele in der Welt der Tiere: Wie bereits erwähnt, gilt es eine begrenzte Rechenleistung und die Echt-

Abbildung 3.2: Ein Insektenauge besteht aus bis zu 30.000 lichtempfindlichen Kegeln (Ommatidien). Je mehr dieser Kegel vorhanden sind, desto höher ist die Auflösung und desto besser (genauer und schneller) kann der optische Fluss wahrgenommen werden.

zeitfähigkeit zu berücksichtigen - dieselben Bedingungen wie wir sie bei Insekten vorfinden. Eine Wespe hat ungefähr eine Million Neuronen und hat sich bereits über Jahrtausende hinweg im Ausweichen von Hindernissen und im Navigieren im 3D-Raum bewiesen. Biologen konnten mit Hilfe aufwendiger Tests mit großer Sicherheit feststellen, dass Bienen, Fliegen, Wespen usw. den optischen Fluss nutzen, um ihren Flug zu stabilisieren und um Objekten auszustellen [Göt68]. So versuchen diese Insekten den optischen Fluss konstant zu halten, was bei einer konstanten Fluggeschwindigkeit eine konstante Höhe zur Folge hat (siehe. Abschnitt 4.3).
Effekte wie z.B. das Ertrinken von Bienen wenn sie über stille Gewässer ohne Kontur fliegen oder das Landen beim Fliegen mit starkem Gegenwind sind logische und beobachtbare Folgerungen dieser These [Weh81].

3.3 Lösungsansätze

Zunächst gilt zu klären bis zu welchem Grad das Problem sich mit nur einer Kamera und ohne Vorkenntnisse überhaupt lösen lässt. Um Entfernungen exakt bestimmen zu können, muss man entweder zwei Kameras verwenden und deren relative Lage zueinander kennen oder eine Bildsequenz mit der absolvierten Bewegung der Kamera haben. Andere Methoden, wie z.B. Objekt-Erkennung und Abschätzung der Dimensionen dieses Objektes (z.B. eines Baumes) sind zu rechenaufwendig und nicht echtzeitfähig.
Im beschriebenen Fall hat man nur eine Kamera und kennt auch die exakte durchgeführte Bewegung nicht - Längen, Abstände usw. können somit nur bis zu einem Skalierungsfaktor bestimmt werden. Für eine Stabilisierung bzw. grobe Kurskontrolle ist die Entfernung von Objekten aber nicht zwingend notwendig:
Das Lokalisierungsproblem lässt sich in relative und absolute Lokalisierung unterteilen. Während bei der absoluten Positionsbestimmung die Erkennung von Landmarken notwendig ist, ist bei der relativen Positionsbestimmung nur die relative Veränderung der sechs Freiheitsgrade von Interesse. Diese reicht aber bereits aus, um einem Roboter eine gewisse Autonomie, wie z.B. Stabilisierung im Raum, grobes Einhalten einer Richtung, usw. zu verleihen. Natürlich aber mit Verlust einer Dimension und zwar der Länge der erfolgten Translation.

In vergangenen Arbeiten wurden vor allem folgende Eigenschaften als Informationsquelle für die Bewegungsschätzung verwendet:

- *Fluchtpunkte*: Berechnung über z.B. einen Garbor-Filter [RK05] oder der Cascaded-Hough-Transformation[1]. Das Bestimmen der Fluchtpunkte ist jedoch sehr rechenaufwendig.
- *Motion Blur*: Das Verschmieren von Punkten ist nur bei hohen Geschwindigkeiten bzw. entsprechend langen Belichtungszeiten sinnvoll und somit nicht echtzeitfähig für Roboter mit üblichen Geschwindigkeiten bis $20\frac{km}{h}$. [Hor86]
- *Reflexionen*: Um mit Hilfe von Reflexionen Aussagen über die Umgebung treffen zu können, muss man die Position und die Art der Lichtquellen kennen (oder gar kontrollieren können) und über die beleuchtete Oberfläche Bescheid wissen. [TV98]

Dadurch, dass man aber nicht nur ein Bild hat, sondern eine Bildsequenz, erhält man zusätzliche Inhalte:

- *optischer Fluss*: Den optischen Fluss (kurz: OF) kann man entweder für jedes einzelne Pixel und so flächendeckend oder nur für bestimmte Punkte berechnen. Für Ersteres gibt es zwar eigene Chips (bzw. Sensoren), womit die Berechnung des OF zwar schnell ist, doch die Weiterverarbeitung der einzelnen Vektoren in Echtzeit Schwierigkeiten bereitet. Zudem benötigt man so einen eigenen Sensor und ist nicht mehr nur auf die Kamera als Datenquelle angewiesen. Mit Hilfe eines Feature Trackers wie den Kanade-Lucas-Tomasi-Tracker (siehe Appendix B) lassen sich robuste, markante Punkte verfolgen und somit der OF für diese Punkte bestimmen. Mehr zum OF findet man in Abschnitt 4.3. [TV98, Hor86]
- *Tracken von Texturwechsel*: Um Texturwechsel verfolgen zu können ist die Anwendung komplizierter Filter (z.B. Wavelet-Filter) notwendig und somit der Rechenaufwand wesentlich höher als bei einem Merkmals-Tracker (vgl. Kapitel B). [NHSA06, FDT98]

Für die Entwicklung des Verfahrens wurden zwar keine Einschränkungen, aber folgende Annahmen getroffen:

[1] http://homes.esat.kuleuven.ac.be/~tuytelaa/CHT.html

Annahmen:

- Die von der Kamera beobachtete Welt ist statisch - die Objekte in der Welt bewegen sich nicht. Diese Annahme wird gemacht, obwohl das entwickelte Verfahren in der Lage ist, einzelne bewegte Objekte bzw. Störungen rauszufiltern.
- Der Roboter befindet sich nicht in einem kleinen geschlossenen Raum, sondern findet vor allem in größeren Räumen und in der freien Natur seinen Einsatz.
- Die Kamera ist parallel und nicht senkrecht zum Untergrund ausgerichtet. Zunächst wird von einer einfachen optischen Kamera ausgegangen, es soll aber auch die Verwendung einer katadioptrische (omnidirektionale) Kamera nicht ausgeschlossen werden.

Somit kann man mit einer Kamera zwar keine absoluten Aussagen über die eigene Position oder die Position anderer Objekte treffen, jedoch gibt es bereits mehrere Verfahren, um aus einem Monokamerasystem Rückschlüsse auf Rotation und Translation zu ziehen. Bereits vorhandene Algorithmen und Methoden wurden daher zwar in Betracht gezogen, doch keine erfüllte alle geforderten Auflagen (siehe Abschnitt 3.4 und Abschnitt 5.6).
Im entwickelten Verfahren wird versucht, alle vorhandenen Informationen im optischen Fluss der Bildsequenz bestmöglich zu nutzen, um möglichst effizient und zuverlässig die relative Bewegung zu schätzen.

3.4 Verwandte Arbeiten

Bisher wurden zur Berechnung der Fundamentalmatrix und somit den Zusammenhängen zwischen 2 korrespondierenden Bildern der 8-Punkt-Algorithmus verwendet [Hor86, TV98]. Vor allem der normalisierte 8-Punkt-Algorithmus, eine robustere Abwandlung des ursprünglichen Verfahrens, wird in der Praxis ziemlich erfolgreich verwendet [Hor86, TV98]. Der Algorithmus wird dabei meist in ein iteratives Verfahren eingebunden, wie z.B. in RANSAC (siehe Abschnitt 4.4), wobei aus der Punktmenge zufällig 8 Punkte zur Fundamentalmatrix-Schätzung herausgenommen werden.

Trotz Normalisierung birgt diese Methode aber auch ihre Probleme: Zunächst dürfen die

Punkte nicht auf einer Ebene liegen, da das Verfahren ansonsten keine Lösung liefert. Mit der Anzahl der verschiedenen Ebenen und der Punkte steigt auch die Genauigkeit des Algorithmus. In [LF93] wird folgender Zusammenhang von Zuverlässigkeit der Lösung, Anzahl der Punkte und Anzahl der Ebenen dargestellt:

	Anzahl der Punkte		
Anzahl der Ebenen	10	40	70
2	73	47,8	46,6
5	60,2	27,5	22,2
50	35,4	7,7	8,4

Tabelle 3.1: Auszug (Eckpunkte) aus einer Tabelle in [LF93] um den Zusammenhang zwischen der Anzahl der Ebenen bzw. der Anzahl der Punkte und dem durchschnittlichen relativen Fehler (in Prozent) darzustellen. Bei den jeweils 100 Testläufen wurde ein 0,2 Pixel Rauschen simuliert.

Die Tabelle 3.1 macht klar, dass die Genauigkeit und Zuverlässigkeit des Ergebnisses stark von der Anzahl der Ebenen abhängt. Leider sind in vom Menschen erschaffenen Umgebungen oft nur sehr wenige Ebenen, aus denen die Punkte stammen, in manchen Fällen sogar nur eine. Die Anzahl der Punkte kann man eventuell erhöhen, indem man einen besseren Tracker bzw. eine bessere Kamera verwendet, doch auf die Anzahl der Ebenen hat man im Normalfall keinen Einfluss. Probleme gibt es auch, wenn sogenannte „gefährliche Flächen“ gemessen werden. Dabei handelt es sich um spezielle Raumquadriken, die trotz beliebiger Anzahl von Korrespondenzen mehrere Fundamentalmatrizen zulassen [LF94]. Da die Epipole als eine einfache Funktion der Kameraverschiebung gesehen werden können, kann man daraus schließen, dass die Stabilität der Fundamentalmatrix-Berechnung mit der Stabilität der Bewegungsschätzung korreliert [LF94]. Somit können folgende drei Zusammenhänge nachgewiesen werden, die auf eine nicht stabile Fundamentalmatrix-Berechnung schließen lassen:

- kleine Translation
- Translationsrichtung parallel zur Bildebene (die Epipole liegen im Unendlichen)
- reine Translation (die Fundamentalmatrix ist antisymmetrisch)

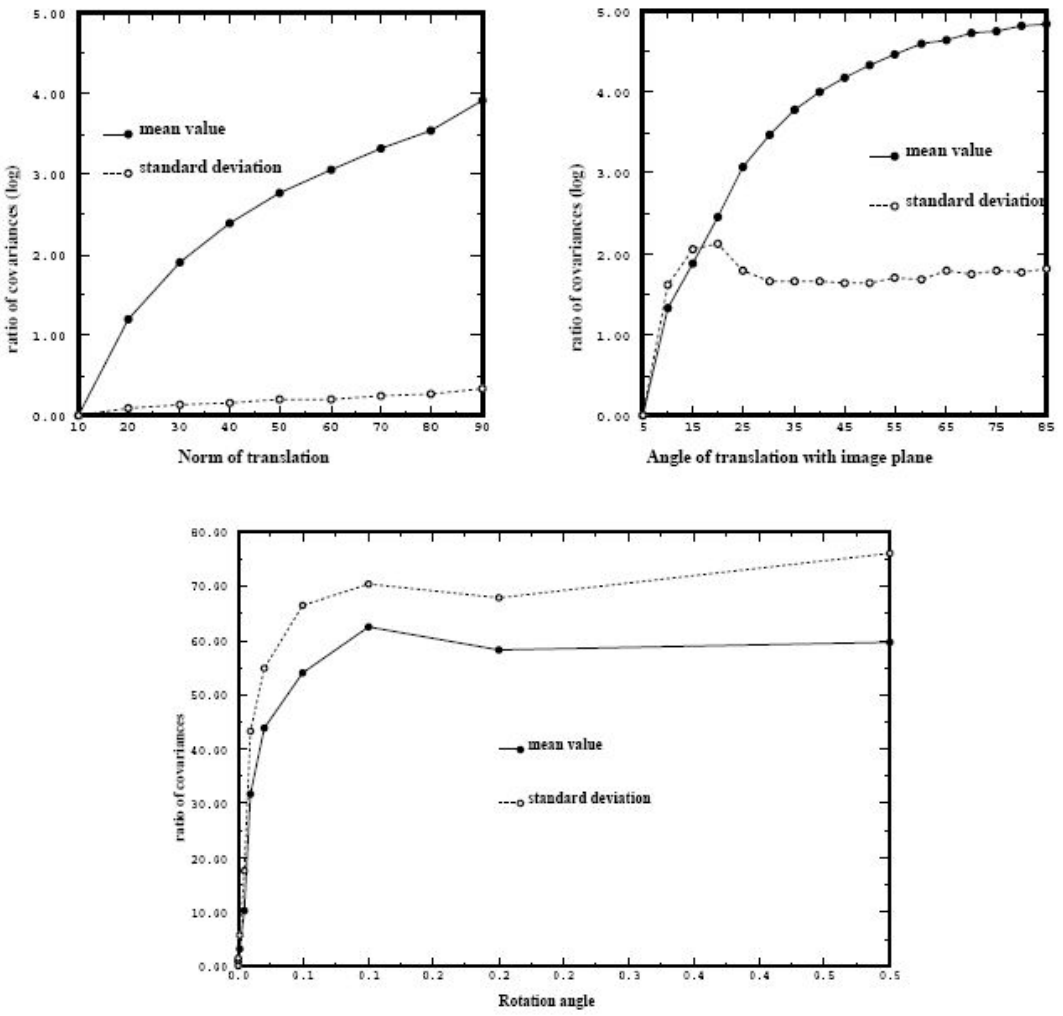


Abbildung 3.3: Die Epipol-Stabilität als Funktion der Bewegung. Je höher der Punkt in Y-Richtung, desto stabiler sind die Epipole.

Ein weiterer Nachteil des 8-Punkt-Algorithmus ist die Tatsache, dass das Verfahren keine Information über die Güte der Lösung gibt. Somit kann die Lösung nur sehr schlecht in Kombination mit einem Kalmanfilter verwendet werden und eine Fusion mit anderen Sensoren ist nur sehr ineffizient möglich.
In Abschnitt 5.6 wird der 8-Punkt-Algorithmus dem hier beschriebenen Verfahren gegenübergestellt.

Als weitere wichtige verwandte Werke sind sicherlich noch die Arbeiten von D. Nister [Nis04] und A. J. Davison [Dav03] zu nennen.
Nister entwickelte einen Algorithmus, um über drei bekannte Punkte in der Welt und den dazugehörigen Bildpunkten für eine **nichtspezifische**, kalibrierte Kamera die 3D-Lage in allen sechs Freiheitsgraden der Kamera zu berechnen. Als Ergebnis erhält man bis zu acht Lösungen (Schnittpunkte eines Kreises mit einer biquadratischen Fläche). Durch Einschränkung der Blickrichtung der Kamera kann man jeweils zwei Paare der Lösungen zusammenfassen und erhält somit nur mehr maximal vier Lösungen und einen formschlüssigen Lösungsweg. Das Finden der Schnittpunkte entspricht dem Berechnen der Nullstellen eines Polynoms achten Grades.
Für dieses echtzeitfähige Verfahren ist aber die genaue Position von mindestens drei sichtbaren und erkannten Punkten in der Welt erforderlich.

Davison entwickelte eine Art Bayes-Filter, der sowohl für SLAM (**S**imultanous **L**ocalisation and **M**apping) als auch „Structure from Motion“ eingesetzt werden kann. Ausgehend von einer kalibrierten Kamera und einer kleinen Anzahl bekannter 3D-Punkte im ersten Bild ist es möglich, die Kamera frei im Raum zu bewegen und zu lokalisieren. Dies geschieht, indem gute Merkmale im Raum gewählt werden, die Tiefe der Punkte über mehrere Bilder geschätzt wird und sobald ausreichend Genauigkeit erreicht ist, wird der 3D-Punkt in die Karte aufgenommen. Dabei werden die Punkte nicht verworfen und nur neue Punkte in Bildbereichen gesucht, die noch keine bzw. wenig bereits gefundene Punkte enthalten. So erreicht man durch wenige Punkte eine Flächenabdeckung der Bilder und der SLAM-Algorithmus wird echtzeitfähig. Aus den aufgenommenen 3D-Punkten lässt sich anschließend weitmaschig die Struktur des Raumes erkennen.

Welche Vor- und Nachteile diese Verfahren gegenüber dem hier entwickelten Algorithmus haben, wird in Abschnitt 5.6 diskutiert.

Kapitel 4

Mathematische Grundlagen des Verfahrens

4.1 Abbildungseigenschaften bei monokularen Kameras

Bei einer monokularen Kamera werden Punkte in der 3D-Welt gemäß dem Strahlensatz über folgende Projektion auf die Bildebene abgebildet:

$$\begin{pmatrix} x_B \\ y_B \end{pmatrix} = \frac{f}{z_W} \begin{pmatrix} x_W \\ y_W \end{pmatrix} \tag{4.1}$$

Die 2D-Punkte im Bild enthalten somit nicht nur die Information ihrer Koordinaten, sondern können als Endpunkte der Richtungsvektoren vom Weltpunkt über das optische Zentrum gesehen werden.

$$\vec{k}_i = \begin{pmatrix} x_{B_i} \\ y_{B_i} \\ 1 \end{pmatrix} \tag{4.2}$$

Durch eine Punktspiegelung der optischen Ebene um das optische Zentrum (vgl. Abbildung 4.1) werden diese Vektoren invertiert und zeigen so in Richtung der 3D-Punkte in

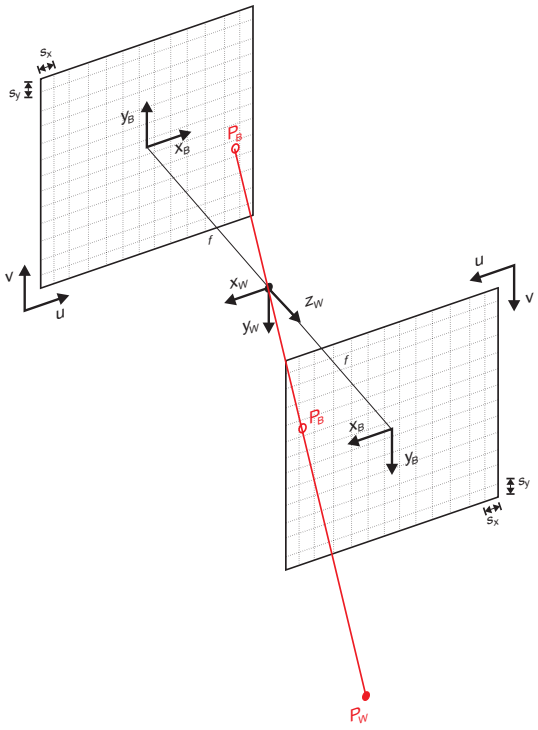


Abbildung 4.1: Um sich ein Koordinatensystem zu sparen, wird das Weltkoordinatensystem und das Kamerakoordinatensystem zusammengelegt mit Ursprung im optischen Zentrum. Zur Vereinfachung und um Vorzeichenfehler zu vermeiden, wird die punktgespiegelte Bildebene angenommen.

der Welt. Indem die Punkte auf Einheitsbrennweite („unit focal“) normiert werden (Division der Punkte durch die Brennweite), wird die Bildebene verschoben und es ergibt sich eine Eins als z-Koordinate der Bildpunkte.

$$\vec{n}_i = \frac{\vec{k}_i}{\left\|\vec{k}_i\right\|} \tag{4.3}$$

Jeder 3D-Punkt befindet sich somit auf dem entsprechenden Strahl. Die radiale Entfernung, also $\left\|\vec{k}_i\right\| = \lambda$ geht jedoch durch die projektive Abbildung verloren.

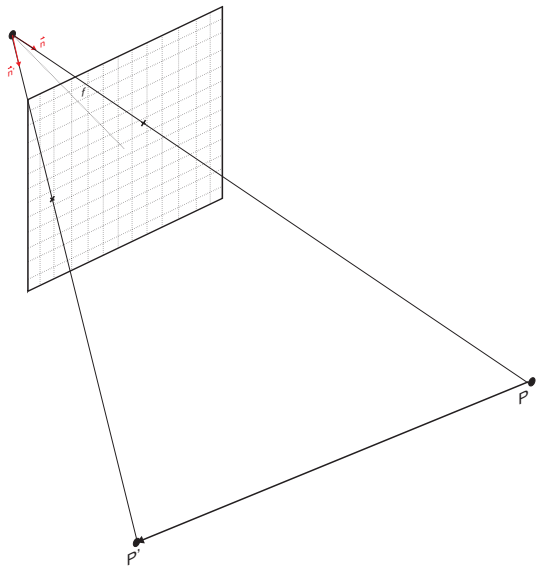


Abbildung 4.2: Ein Bildpunkt repräsentiert einen Richtungsvektor zum eigentlichen Punkt in der Welt. Die radiale Entfernung geht durch die projektive Abbildung verloren.

4.2 Bewegungsgleichung

Jede Bewegung (Positionsänderung) im dreidimensionalen Raum kann als Kombination einer Rotation um die drei Achsen des kartesischen Koordinatensystems und einer Translation in eine bestimmte Richtung betrachtet werden.

$$\vec{P'} \;=\; R(\vec{P} \;+\; \vec{T}) \tag{4.4}$$

Dabei wird angenommen, dass zuerst die Translation ausgeführt wird und erst anschließend die Rotation auf den verschobenen Punkt durchgeführt wird. Diese Sichtweise erleichtert in Abschnitt 5.1 die theoretische Herleitung des Vorgehens bei der Aufteilung der Punkte.

Da jeder Punkt durch einen Vektor im Raum beschrieben werden kann

$$\vec{P_i} \;=\; \lambda_i \vec{n}_i \tag{4.5}$$

und die Rotation als orthonormale Abbildung die Länge λ der Vektoren nicht verändert, kann die Gleichung 4.4 durch Einsetzen von 4.5 wie folgt geschrieben werden:

$$\lambda'_i \, \vec{n}'_i \;=\; R \; (\lambda_i \vec{n}_i \;+\; \vec{T}) \tag{4.6}$$

Diese Gleichung gilt als Ausgangspunkt für die Aufteilung in translationsinvariante und translationsabhängige Punkte - der grundlegenden Idee des Verfahrens (siehe Abschnitt 5.1).

4.3 Optischer Fluss

Als optischen Fluss (kurz: OF) bezeichnet man ein Vektorfeld, das die 2D-Bewegungsrichtung und -geschwindigkeit in einer Bildsequenz wiederspiegelt. Der optische Fluss kann dabei für jedes Pixel oder aber auch nur für bestimmte Pixel bestimmt werden. Natürlich kann man die Start- und Endpunkte der Vektoren auch subpixelgenau angeben.
Der optische Fluss ist für diese Arbeit insofern von Bedeutung, da diese Vektoren zusammen mit den Kameraparametern, wie u.a. die Brennweite, die einzige Informationsquelle für den Algorithmus darstellen. Aus diesen Daten wird die relative Bewegung berechnet. In einer Bildsequenz erhält man den OF dadurch, dass jeder neue Punkt (und somit Endpunkt) des Bildes *i*, im nächsten Bild *i+1* als alter Punkt (und somit Ausgangspunkt) verwendet wird. Vorausgesetzt natürlich, es wird die entsprechende Korrespondenz gefunden. Die Datentypen, die im Algorithmus verwendet werden, sind somit geordnete Tupel der Form:

$$(B_{ialt},\ B_{ineu}) \tag{4.7}$$

Da es sich um Bildpunkte handelt wird die Abkürzung B verwendet. Im Folgenden wird dieses Datenstrukt als OF-Tupel oder OF-Vektor bezeichnet.
Die Länge eines OF-Vektors ist dabei proportional zur Geschwindigkeit der Kamera und umgekehrt proportional zum Quadrat der Entfernung des Punktes in der Welt [TV98].

$$l_{OFV} \ = \ F \frac{Geschwindigkeit}{Entfernung^2} \ \propto \ \frac{Geschwindigkeit}{Entfernung^2} \tag{4.8}$$

F steht für einen Faktor, der sich aus der Entfernung des Punktes von der optischen Achse und der Brennweite der Kamera ergibt.

4.4 RANSAC

Der RANSAC-Algorithmus (RANdom SAmple Consensus) ist ein generischer Algorithmus, der vor allem wegen seiner Robustheit bekannt und beliebt ist. Er wird vor allem bei Datensätzen mit vielen Ausreißern verwendet. Im Gegensatz zum LMS (Least Median of Squares) kann dieses Verfahren auch dann noch ein gutes Ergebnis liefern, wenn mehr als die Hälfte eines Datensatzes Ausreißer sind. Beim Erkennen der translationsinvarianten Punkte ist diese Eigenschaft von entscheidender Bedeutung, da mehr translationsabhängige Punkte vorhanden sein können.
Grob beschrieben, geht der RANSAC-Algorithmus so vor, dass er über eine vorgegebene Anzahl an Iterationen eine zufällige Teilmenge der gegebenen Gesamtmenge aussucht und für diese Punkte entsprechend einer vorgegebenen Methode ein passendes Modell berechnet. Das Modell wird anschließend auf den gesamten Datensatz angewendet und überprüft, wie viele der restlichen Punkte im Rahmen eines maximalen Fehlers liegen. Ist die Anzahl dieser „Treffer“ größer als ein vorgegebener unterer Grenzwert, wird über alle dieser „passenden“ Punkte ein neues Modell errechnet und der durchschnittliche Fehler mit dem besten Fehler der vergangenen Iterationen verglichen. Ist der Fehler kleiner, wird das aktuelle Modell als neues bestes Modell übernommen. Stellt sich in den restlichen Iterationen heraus, dass es sich bei diesem Modell um die Anpassung mit dem kleinsten durchschnittlichen Fehler handelt, wird dieses als Ergebnis des Verfahrens zurückgegeben (vgl. Abbildung 4.3). Der generische RANSAC-Code ist in Appendix A dargestellt.
Dieser Algorithmus ist dabei so entworfen, dass er für jedes beliebige Problem anwendbar ist. Ihn direkt ohne Anpassungen zu übernehmen, wäre daher nicht optimal, da meist noch zusätzliche Informationen vorhanden sind, die so verloren gehen würden. Bei der Entwicklung des hier beschriebenen Verfahrens wurde besonders darauf geachtet, alle Informationen zu verwenden, die man aus den Daten bekommen bzw. voraussetzen kann. Somit sind einige Zusatzinformationen in den ursprünglichen RANSAC-Algorithmus integriert. Das Verfahren ist so angepasst, dass eine große Effizienzsteigerung und Stabilitätszunahme erreicht wird. Auf die einzelnen Änderungen wird zu Beginn der Kapitel 5.2 und 5.3 eingegangen.

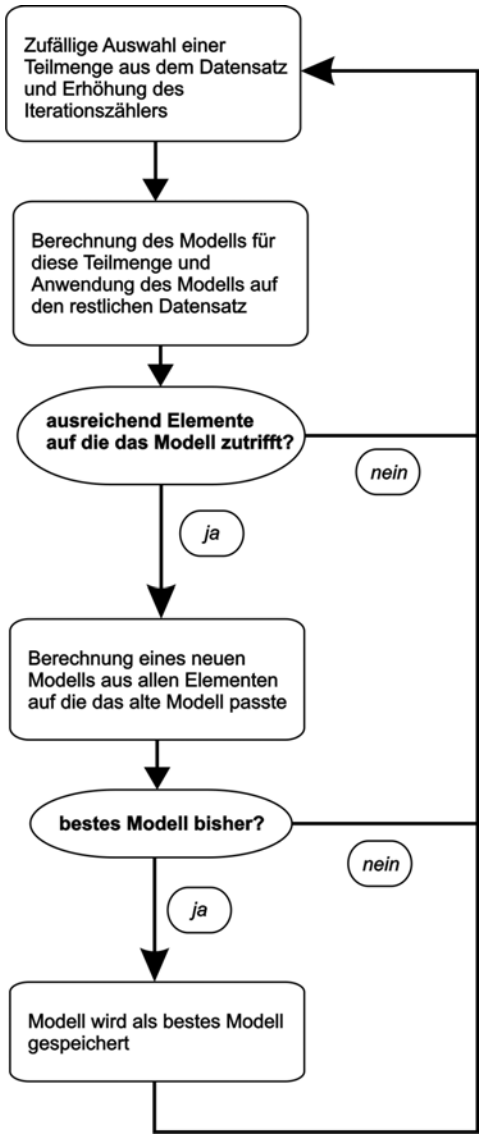


Abbildung 4.3: Hier dargestellt ist ein grobes Ablaufdiagramm des generischen RANSAC-Algorithmus. Der generische RANSAC-Code ist in Appendix A dargestellt.

Kapitel 5

Theorie und Gliederung des Verfahrens

Als Ziel des zu entwerfenden Algorithmus galt es, Rotation und Translationsrichtung aus den Korrespondenzen in zwei Bildern bzw. einer Bildsequenz zu berechnen. Zudem sollten Hindernisse erkannt und lokalisiert werden, sodass eine Ausweichtrajektorie berechnet werden kann.
Getrennt lassen sich Translation und Rotation einfach, geschlossen und robust berechnen. Doch leider handelt es sich nur selten um eine reine Translation bzw. reine Rotation. Da man in der Praxis mit Rauschen und Ausreißern rechnen muss, werden auch bei sehr stabilen Verfahren iterative Algorithmen wie RANSAC oder LMS verwendet, um diese Störungen zu umgehen. Im hier besprochenen Algorithmus wird der RANSAC-Algorithmus (siehe Abschnitt 4.4) nicht nur angewandt, um Ausreißer zu erkennen, sondern er erfüllt die wichtige Aufgabe den Datensatz der korrespondierenden Punkte in zwei Teilmengen zu trennen (siehe Abschnitt 5.1), nämlich:

- die weit entfernten translationsinvarianten Punkte und
- die nahen translationsabhängigen Punkte.

Gleichzeitig unterdrückt RANSAC das Rauschen bzw. auch Ausreißer (Fehlkorrespondenzen) und wird daher nicht nur in der Rotations- sondern auch Translationsberechnung verwendet. Der Gesamtalgorithmus kann somit grob in folgende Teilaufgaben gegliedert

werden:

1. Berechnung der Rotation mit einem angepassten RANSAC-Verfahren und gleichzeitige Aufteilung der Daten in Rotations-Inlier und -Outlier (siehe Abschnitt 5.2).
2. Berechnung der Translation über die zurückrotierten Outlier der Rotationsberechnung. Auch hier wird wieder ein angepasster RANSAC-Algorithmus verwendet (siehe Abschnitt 5.3).
3. Die Hindernis-Erkennung. Sie wird auch über die zurückrotierten Outlier der Rotationsberechnung bestimmt (siehe Abschnitt 5.4).

5.1 Aufteilung der Punkte

Wie zu Beginn dieses Kapitels bereits erwähnt, ist eine Aufteilung der Punkte in translationsinvariante und translationsabhängige Punkte notwendig. In der kontinuierlichen Welt gibt es natürlich keine realen (endlich entfernten) Punkte, deren Projektion auf eine Ebene sich durch eine Translation überhaupt nicht verändert. Doch je weiter ein Punkt von der Kamera entfernt ist, desto kleiner wird die Differenz seiner projektiven Abbildungen während einer translatorischen Bewegung der Kamera.
Wegen der Diskretisierung durch die Pixelgröße s des Kamerasensors kann man aber jene Punkte, die eine Entfernung z_∞ nicht unterschreiten, als translationsinvariant ansehen.

$$z_\infty = \frac{f}{s_m}\, T_m \tag{5.1}$$

Dabei entsprechen s_m und T_m entweder s_x (Ausdehnung der Pixel in x-Richtung) und T_x oder s_y (Ausdehnung der Pixel in y-Richtung) und T_y, je nachdem welches Verhältnis $\frac{T_m}{s_m}$ kleiner ist. T_x und T_y sind dabei die x- bzw. y-Komponenten der Abbildung der Translation. Abbildung 5.2 veranschaulicht die Formel, wie man diese Komponenten getrennt berechnen kann:

$$\Delta T_{s_m} \;=\; \cos\psi_m \, \Delta T \tag{5.2}$$

$$\Delta T_m \;=\; \frac{T_{s_m}}{\cos\varphi_m} \;=\; \frac{\cos\psi_m}{\cos\varphi_m}\, \Delta T \tag{5.3}$$

$$\Delta T_x \;=\; \frac{\cos\psi_x}{\cos\varphi_x}\, \Delta T \qquad bzw. \qquad \Delta T_y \;=\; \frac{\cos\psi_y}{\cos\varphi_y}\, \Delta T \tag{5.4}$$

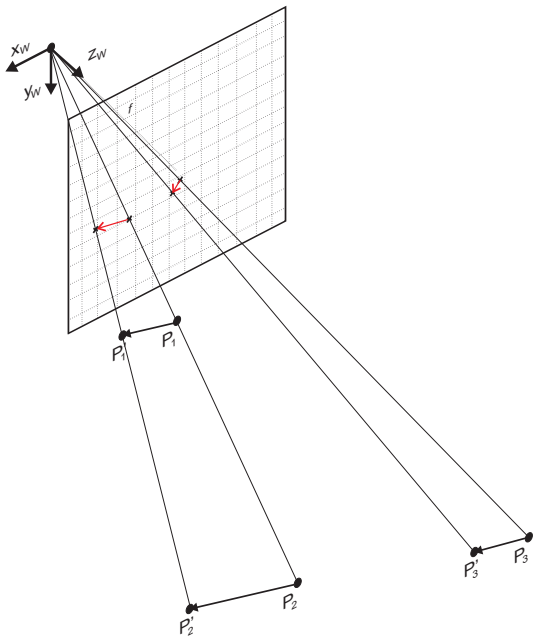


Abbildung 5.1: Punktvektor 1 (P_1, P_1') ist kürzer als Punktvektor 2, aber näher beim optischen Zentrum. Darum ist es möglich, dass die entsprechenden OF-Vektoren die identische Abbildung haben. Punktvektor 3 ist gleich lang wie Punktvektor 1, aber weiter vom optischen Zentrum entfernt - die Abbildung ist dementsprechend kürzer. Sie ist sogar so kurz, dass die Abbildung der Punkte P_3 und P_3', in einem Pixel liegen. Würden diese OF-Vektoren allein durch eine Translation zustande kommen, wäre der Punkt 3 somit translationsinvariant.

ψ ist dabei der Winkel zwischen Translation und der Senkrechten zum Abbildungsstrahl des Punktes. φ entspricht dem Winkel zwischen Punkt und optischer Achse. Somit ist bei einer Translation der OF-Vektor umso länger, je senkrechter die Translationsrichtung zum Abbildungsstrahl des jeweiligen Punktes ist und je näher am Bildrand der Punkt abgebildet wird. Das bedeutet, dass bei Bewegungen in Richtung der optischen Achse die translationsinvarianten Punkte viel näher liegen, da die messbare Translationskomponente im Allgemeinen sehr klein wird.

Beispiel: Bei einer Bewegungsgeschwindigkeit von 2 km/h **parallel** zur Bildebene und einer Kamera mit $f = 8,6\ mm$ Brennweite, $s = 9,6\ \mu m$ Pixelgröße und 30 Hz Bildrate

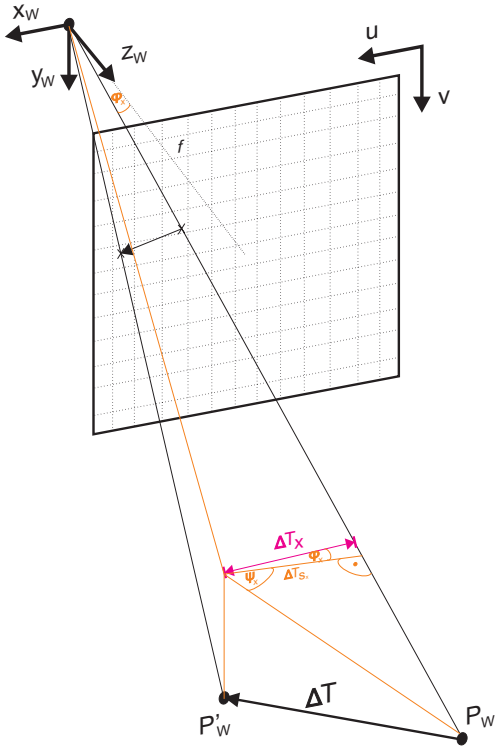


Abbildung 5.2: Diese Skizze soll veranschaulichen, wie die messbare Translationskomponente ΔT_x berechnet werden kann. Salop kann man sagen, dass die OF-Vektoren länger werden, je paralleler die Translation zur Bildebene ist und je näher sich die Vektoren am Bildrand befinden. Die orangen Hilfslinien stellen die Projektionen auf die x-z-Ebene dar.

kann folgende Entfernung für z_∞ berechnet werden:

$$v \;=\; 2\ km/h \;=\; 0,556\ m/s \quad , \quad fr \;=\; 30\ 1/s \qquad \Rightarrow\; T \;=\; \frac{v}{fr} \;=\; 0,185$$

$$\Rightarrow\; z_\infty \;=\; \frac{f}{s}\,T \;=\; 16,6\ m$$

Bei einer hohen Bildrate und somit kleiner Translation zwischen den einzelnen Messungen, sind bereits sehr nahe Punkte translationsinvariant.

Natürlich nimmt die Entfernung z_∞ der translationsinvarianten Punkte bei subpixelgenauer Merkmalsfindung zu. Auf der anderen Seite kann in Umgebungen mit hauptsächlich nahen Punkten, eine Zusammenfassung mehrerer Pixel (z.B. über Bildpyramiden) s_m

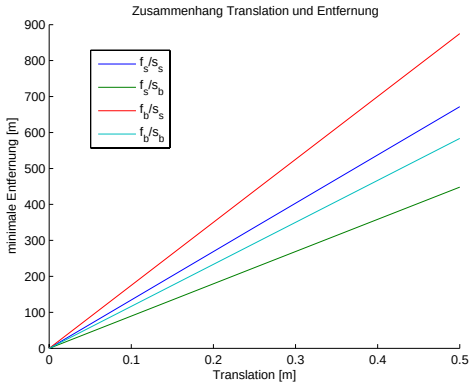


Abbildung 5.3: Umso kleiner die Brennweite und umso größer die Pixel, desto näher dürfen die translationsinvarianten Punkte liegen bzw. desto größer darf die Translation zwischen den Bildern sein. Bei einer Seitenlänge der quadratischen Pixel von $s_s = 6,4\ \mu m$ bzw. $s_b = 9,6\ \mu m$ und einer Brennweite von $f_s = 8,6\ mm$ bzw. $f_b = 11,2\ mm$ ergeben sich die hier dargestellten Zusammenhänge.

künstlich erhöhen und somit auf Kosten eines größeren Fehlers eine Bewegungsschätzung ermöglichen (siehe Kapitel 8).
Nun wurde gezeigt, dass man in der Theorie die Punkte nach dem z_∞-Kriterium in zwei Teilmengen $\{P_{T_{inv}}\}$ und $\{P_{T_{abh}}\}$ unterteilen kann, doch es stellt sich immer noch die Frage, wie diese Punkte in der Praxis unterschieden werden können. Da durch die projektive Abbildung keine Tiefeninformation mehr vorhanden ist, gibt es nur ein (triviales) Unterscheidungsmerkmal für die beiden theoretisch durchschnittfreien Mengen: Alle Punkte die sich nach dem Zurückrotieren mit der geschätzten Rotation an ihrer alten Position befinden sind translationsinvariant, da die Translation entweder keinen Auswirkung auf die Punkte hatte oder keine Translation vorhanden ist. Somit werden im translationsfreien Fall alle Punkte als translationsinvariant angesehen, was aber keine Auswirkung auf das Verfahren hat.
Die Trennung der Punkte erfolgt somit gleichzeitig mit der Berechnung der Rotation.

5.2 Berechnung der Rotation

Die Rotationsbestimmung wird ausschließlich über die weit entfernten und somit translationsinvarianten Punkte $\{P_{Tinv}\}$ durchgeführt. Die Unterscheidung der Punkte erfolgt dabei nach dem einfachen Kriterium, ob die Länge des optischen Flusses nach der Rückrotation nahe Null ist. Die Frage stellt sich, bei welcher Rotation. Es ist weder die Rotation, Translation noch die Entfernung der Punkte bekannt. Und hier kommen die Vorteile des RANSAC Algorithmus doppelt zum Einsatz: Durch das iterative Verfahren wird zunächst die Rotation bestimmt, anhand derer anschließend die Punkte aufgeteilt werden (siehe Abschnitt 5.1).
Zufällig werden drei Punkte aus dem Datensatz genommen und die passende Rotation, wie in Abschnitt 5.2.1 beschrieben, bestimmt. Kann dieses erste Modell erfolgreich auf einen großen Anteil der anderen Punkte angewendet werden, handelt es sich offensichtlich um drei translationsinvariante Punkte und eine gute Rotationsschätzung (Zeile 23 im generischen Code, vgl. Appendix A). Die OF-Tupel, dessen Endpunkte sich nach der Rückrotation nahe dem Ausgangspunkt befinden und somit dem ersten Modell entsprechen, werden als Inlier bezeichnet. Ein OF-Vektor ist somit Rotations-Inlier, falls er weit entfernt und somit translationsinvariant ist oder es keine Translation gibt bzw. wenn es sich um einen Zufall handelt (Ausreißer, Rauschen). Die Menge der Inlier wird nun verwendet, um eine neue optimale Rotation über alle diese Punkte zu berechnen - die erste Berechnung beruhte schließlich nur auf drei Punkte. Die neu berechneten Rotationsparameter sind somit genauer und weniger rauschanfällig. OF-Tupel, die dieser zweiten Berechnung nicht mehr genügen, werden aus der Inlier-Menge entfernt. Die durchschnittliche Abweichung vom Ausgangspunkt wird als Fehlermaß verwendet und nur wenn dieses kleiner ist, als der bislang beste Fehler und noch genügend Inlier vorhanden sind, wird das zweite Modell als bestes gespeichert (Zeile 27-29 im generischen Code, vgl. Appendix A).

Es stellt sich nur mehr die Frage, wie viele Iterationen notwendig sind, um sicher zu gehen, dass bei der zufälligen Wahl der *maybeinliers* (vgl. Appendix A) auch ein Set mit drei

translationsinvarianten OF-Tupeln dabei ist.

$$p_{set} = p_{Tinv}^3 \tag{5.5}$$

$$p_{mind1} = iter * p_{set} \tag{5.6}$$

Zur Orientierung: Bei den für die Tests verwendete Bilder (siehe Abschnitt 7.1) waren im Durchschnitt ungefähr 60% der verfolgten Punkte Rotations-Inlier (Merkmale am Horizont können durch den großen Texturspung gut verfolgt werden). Das untere 5%-Quantil lag bei ungefähr 30%.

$$p_{set} = p_{Tinv}^3 = 0,3^3 = 0,027$$

$$p_{mind1} = iter * p_{set} = 40 * 0,027 = 1,08$$

In diesem konkreten Fall, mit 30 Iterationen, würde man für 95% der Bilder mit 100%iger Wahrscheinlichkeit ein Set mit drei translationsinvarianten OF-Tupeln im Datensatz finden. Über diese Gleichungen kann man natürlich auch die Anzahl der Iterationen ausrechnen, um mit einer gewissen Wahrscheinlichkeit einen Satz Inlier zu finden. Abbildung 5.4 stellt den Zusammenhang der notwendigen Iterationen und den Prozentsatz der Rotations-Inlier dar.

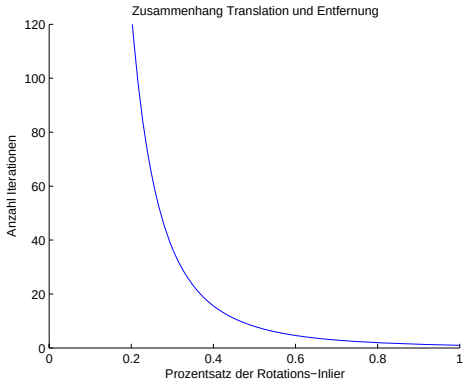


Abbildung 5.4: Hier dargestellt ist der Zusammenhang der notwendigen Iterationen, um bei einem gegebenen Prozentsatz an Rotations-Inlier eine Trefferwahrscheinlichkeit von 100% zu erreichen.

Doch man verfügt über zusätzliches Wissen, das die Suche nach den entfernten Punkten wesentlich beschleunigt: Wenn das Verfahren auf eine Bildsequenz angewendet wird,

kann das Wissen um die vorherigen Inlier verwendet werden, um diese bei der eigentlich zufälligen Wahl der *maybeinlier* im nächsten Schritt zu bevorzugen und den anderen Punkten vorzuziehen. Man kann davon ausgehen, dass weit entfernte Punkte des vorangegangenen Schrittes wiederum weit entfernt und somit translationsinvariant bzw. Inlier sind (siehe Abschnitt 4.3). Durch dieses Wissen lässt sich die Anzahl der notwendigen Iterationen drastisch verringern. Wahrscheinlichkeitstheoretisch würde eine Iteration ausreichen, da man mit sehr hoher Sicherheit annehmen kann, dass es sich um ein Set mit ausschließlich Rotations-Inliern handelt.

Somit sind folgende Parameter für den angepassten RANSAC-Algorithmus notwendig:

Eingabe:

- *data:* Datensatz der verfolgten Punkte. Der Datensatz besteht aus den alten und den neuen Punkte-Paaren sowie den entsprechenden Indizes.
- *model:* Das Modell, das es zu berechnen gilt, ist die Rotation. Die Rotation wird dabei, wie in Aschnitt 5.2.1 beschrieben, aus drei bzw. mehreren Punkten berechnet.
- *n:* Die minimale Anzahl an Punkten für die Rotationsbestimmung ist drei.
- *k:* Die Anzahl der Iterationen, in denen neue Modelle berechnet werden, richtet sich nach der Anzahl der Punkte, der relativen Häufigkeit von guten Punkten und *n*. Abbildung 5.4 stellt die berechneten Zusammenhänge dar.
- *t:* Der Grenzwert für den tolerierten Fehler der Inlier ist abhängig von der Pixelgröße und dem Bildrauschen.
- *d:* Minimaler Prozentsatz der Punkte in data, die dem *maybe-Modell* entsprechen müssen, um dieses für die Rotationsberechnung verwenden zu können (Anteil der *alsoinlier* in der gesamten OF-Menge). Dieser Parameter ist abhängig von der Anwendung und der verwendetetn OF-Quelle (z.B. dem Tracker).

Ausgabe: *bestfit:* Die beste gefundene Rotationsmatrix bzw. eine Einheitsmatrix, falls keine passende Rotation gefunden werden konnte.

Die eingeführten Unterschiede zum generischen Algorithmus (siehe Appendix A) bestehen demnach aus folgenden zwei Punkten:

- Beim Test, ob es sich um ein passendes Modell handelt und somit genügend passende OF-Tupel gefunden wurden, wird nicht nur die Anzahl der *alsoinliers*, sondern die Summe von *also-* und *maybeinliers* mit dem Prozentsatz verglichen. Dies ermöglicht eine bessere Angabe des minmalen Prozentsatzes, dem die Inlier entsprechen müssen. Gleichzeitig wird verhindert, dass bei einer geringen OF-Vektoren-Zahl die drei OF-Tupel der *maybeinlier* ins Gewicht fallen.
- Weiters werden nach der Rotationsschätzung über die gesamten Inlier, alle OF-Tupel, die der gemeinsamen Rotationsbestimmung nicht mehr genügen, aus der Inlier-Menge entfernt. Anschließend wird erneut überprüft, ob genügend Inlier vorhanden sind, bevor die Parameter übernommen werden können. So wird auch gewährleistet, dass beim nächsten Bild der Sequenz nicht ungenaue oder gar falsche Rotations-Inlier bevorzugt werden.

5.2.1 Berechnung der Rotationsmatrix

Für die Rotationsberechnung wurde ein Verfahren verwendet, das der Rotationsberechnung von Berthold K.P. Horn et al. in [Hor87, BKPHN87] sehr ähnlich ist, jedoch im Gegensatz dazu nicht auf der Eigenwertzerlegung von symmetrischen Matrizen basiert, sondern auf der Singulärwertzerlegung einer beliebigen Matrix. Der formschlüssige Algorithmus wurde unabhängig zu Horn von K.S. Arun entwickelt und wird in [AHB87] im Detail beschrieben. Das Verfahren ermöglicht die Berechnung der Rotation zwischen zwei korrespondierenden Punktwolken $\{P_i\}$ und $\left\{\bar{P}_i\right\}$. Die so berechnete orthonormale Rotationsmatrix ist bezüglich der Summe der Fehlerquadrate optimal.
Dieser Algorithmus ist gegenüber Rauschen robust. Nur wenn alle Punkte auf einer Linie liegen, liefert diese Methode unendlich viele Lösungen und ist somit nicht zu gebrauchen. Dieser Fall kann daran erkannt werden, dass zwei der drei singulären Werte der Matrix $\tilde{M}$ (vgl. Formel 5.14) gleich sind.

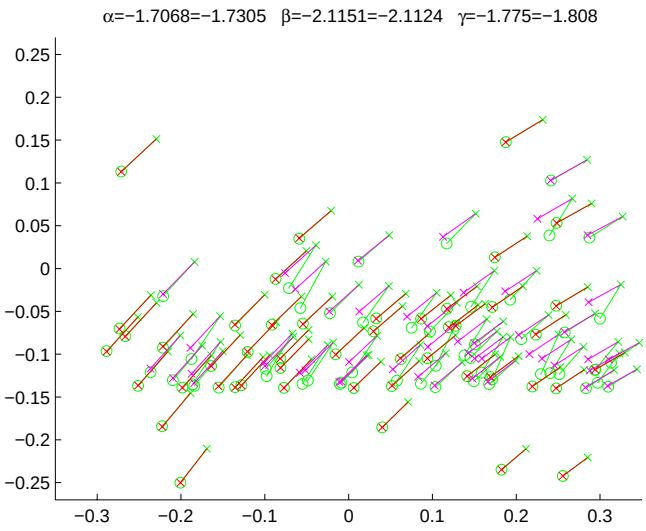


Abbildung 5.5: Hier dargestellt ist das Ergebnis einer Rotationsberechnung in Matlab. Die grünen Linien stellen den optischen Fluss dar - Kreise signalisieren die Ausgangspunkte der Vektoren, Kreuze die Endpunkte. Die Inlier sind rot markiert, Outlier haben die Farbe Magenta. Rote Kreuze, also die zurückrotierten Rotations-Inlier, liegen in den grünen Kreisen - der OF-Vektor kam somit ausschließlich durch die Rotation zustande. Mehr zur farblichen Kodierung wird in Abschnitt 7.2 beschrieben.

Aruns Algorithmus [AHB87]

Bei einem Bildpunkt handelt es sich eigentlich um einen Lichtstrahl, der von einem Objekt in der Welt ausgeht, durch das optische Zentrum der Kamera läuft und auf die Bildebene auftrifft. Eine Dimension geht zwar durch diese projektive Abbildung verloren, doch kann der Bildpunkt weiterhin als Strahl interpretiert werden. Ist es möglich diesen Bildpunkt über ein oder mehrere Bilder zu verfolgen und nimmt man an, dass es sich bei der Bewegung der Kamera um eine reine Rotation handelt, ist auch der entsprechende Strahl nur rotiert.

$$\lambda_i \, \vec{n_i'} = R^T \, \lambda_i \, \vec{n_i} = \lambda_i \, R^T \, \vec{n_i} \tag{5.7}$$

Das Verfahren basiert, wie bereits erwähnt, auf einer Punktwolke. Eine Verschiebung des Ursprungs des Koordinatensystems beeinflusst das Ergebnis der Berechnungen wesentlich (vgl. [TV98]). Das beste Ergebnis wird unter folgenden zwei Bedingungen erzielt: Der

Ursprung befindet sich im Zentroiden der Punktwolke. Die Punkte, die als Richtungsvektoren gesehen werden können (vgl. Gleichung 4.2), werden auf eine Sphäre mit Radius Eins und Mittelpunkt im Ursprung verschoben. So erhält man Einheits-Richtungsvektoren. Es ist von großer Bedeutung, dass für das weitere Vorgehen diese normierten Punkte bzw. Vektoren $\vec{n}$ verwendet werden.
Normierung auf Länge Eins:

$$\vec{k} = P_C = \begin{pmatrix} x_C \\ y_C \\ 1 \end{pmatrix} \tag{5.8}$$

$$\vec{n} = \frac{\vec{k}}{\left\|\vec{k}\right\|} \tag{5.9}$$

Verschiebung des Ursprungs in die Zentroiden $\bar{P}$ bzw. $\bar{P^*}$ der Punktwolken:

$$\bar{P} = \frac{1}{n}\sum_{i=1}^{n} P_i, \quad \bar{P^*} = \frac{1}{n}\sum_{i=1}^{n} P_i^* \tag{5.10}$$

$$P_i' = P_i - \bar{P}, \quad {P^*}_i' = P_i^* - \bar{P^*} \tag{5.11}$$

Nun wird die nicht normierte, gekreuzte Kovarianz-Matrix berechnet:

$$\tilde{M} = \sum_{i=1}^{n} {P^*}_i' \, {P_i'}^T \tag{5.12}$$

Somit wäre $\frac{1}{n}\tilde{M}$ die gekreuzte Kovarianz-Matrix zwischen $\{P_i\}$ und $\left\{\bar{P}_i\right\}$.
Nun kann gezeigt werden, dass die optimale Annäherung (bzgl. der Summe der Fehlerquadrate) der Rotation $\tilde{R}$ an die wahre Rotation R folgende Gleichung erfüllen muss:

$$\tilde{R} = argmax_{\tilde{R}}\, tr(R\tilde{M}) \tag{5.13}$$

Ist $(\tilde{U}, \tilde{\Sigma}, \tilde{V})$ eine Singulärwertzerlegung von $\tilde{M}$

$$\tilde{M} \;=\; (\tilde{U}, \tilde{\Sigma}, \tilde{V}) \tag{5.14}$$

dann kann $\tilde{R}$ wie folgt berechnet werden:

$$\tilde{R} = \tilde{V}\,\tilde{U}^T \tag{5.15}$$

$\tilde{R}$ ist somit notwendigerweise orthonormal, symmetrisch und positiv definit. Es kann aber vorkommen, dass in koplanarer Anordnung der Punkte eine Spiegelungsmatrix und nicht eine Rotationsmatrix berechnet wird. Dies ist genau dann der Fall, wenn $det(\tilde{R}) = -1$ und nicht $+1$ ist. In diesem Fall kann die Rotationsmatrix über

$$\tilde{R}' = \tilde{V}'\tilde{U}^T, \quad \text{wobei } \tilde{V}' = \begin{pmatrix} v_1 \\ v_2 \\ -v_3 \end{pmatrix} \tag{5.16}$$

berechnet werden. Dabei sind v_1, v_2 und v_3 die Zeilen in V entsprechend den Singulärwerten λ_1, λ_2 und λ_3, wobei $\lambda_1 > \lambda_2 > \lambda_3 = 0$ ist.

Ist das Rauschen sehr groß und liegen alle Punkte in einer Ebene, kann es vorkommen, dass $det(\tilde{R}) = -1$, doch keiner der singulären Werten Null ist. In diesem Fall kann über dieses Verfahren keine Lösung gefunden werden.

Wiederholt wird darauf hingewiesen, dass sich im Set, das der Rotationsberechnung zur Verfügung steht, nur OF-Tupel befinden dürfen, die ausschließlich durch die Kamera-Rotation entstanden sind. Jegliche Translations-Komponente würde das Ergebnis verfälschen.
Eigentlich ist Aruns Verfahren vorgesehen auch die Translation und Skalierung der Kamera zu bestimmen, jedoch müsste in diesem Fall der Ursprung beider Punktwolken identisch sein, was bei einer bewegten Kamera leider normal nicht der Fall ist. Auch bei diesem Verfahren sind natürlich mindestens drei Punkte notwendig, um die Rotation im dreidimensionalen Raum korrekt zu berechnen.

5.2.2 Berechnung des Rotations-Fehlers

Die Abweichung eines OF-Vektors von einem gegebenen Modell ergibt sich aus dem euklidischen Abstand des Startpunktes und des zurückrotierten neuen Endpunktes eines

OF-Tupels.

$$B^{*\prime}{}_{i_{neu}} = R^T \begin{pmatrix} x_{B_{i_{neu}}} \\ y_{B_{i_{neu}}} \\ 1 \end{pmatrix} \tag{5.17}$$

$$B^*_{i_{neu}} = \frac{B^{*\prime}{}_{i_{neu}}}{z_{B^{*\prime}{}_{i_{neu}}}} \tag{5.18}$$

Da die Rotationsmatrix orthonormal ist, entspricht die dazugehörige inverse Matrix ihrer Transponierten. Zur Erinnerung: Durch die Normierung der Punkte auf die Einheits-Brennweite (vgl. Gleichung 4.2) ist die z-Koordinate der Punkte Eins.
Somit ergibt sich folgender Fehler für jeden Punkt

$$err_i \;=\; \sqrt{\left(x_{B_{i_{alt}}} - x_{B'_{ineu}}\right)^2 + \left(x_{B_{i_{alt}}} - x_{B'_{ineu}}\right)^2} \tag{5.19}$$

und ein gemeinsamer, durchschnittlicher Fehler von

$$\bar{err} \;=\; \frac{1}{n} \sum_{i=1}^{n} err_i \tag{5.20}$$

Dieser Fehler wird zur Bewertung der berechneten Rotationsmatrix herangezogen (Zeile 27 im generischen Code, vgl. Appendix A).

5.3 Berechnung der Translation

Durch die Rotationsberechnung werden die OF-Tupel bereits indirekt in eine translationsinvariante und translationsabhängige Teilmenge unterteilt. Die Menge der Rotations-Outlier besteht somit nur mehr aus nahen, translationsabhängige OF-Vektoren und natürlich evtl. Ausreißern. Diese werden mit der berechneten Rotation zurückrotiert und enthalten somit nur mehr die Translationskomponente (vgl. Gleichung 4.4). Dem Translationsalgorithmus zur Verfügung gestellt, kann dieser aus diesem Satz OF-Tupel die Translation schätzen.

$$\vec{P}'^{*} \;=\; R^T\,\vec{P}' \;=\; R^T\,(\,R\,(\,\vec{P}\;+\;\vec{T}\,)\,) \;=\; \vec{P}\;+\;\vec{T} \tag{5.21}$$

Dabei ist $\vec{T}\;\neq!\;0$ weil nur Rotations-Outlier verwendet werden.
Wieder wird der RANSAC-Algorithmus verwendet, um eventuelle Ausreißer zu unterdrücken. Die Wahl der *maybeinliers* wird aber auch dieses Mal nicht ganz dem Zufall überlassen. Lange Vektoren bieten generell genauere Ergebnisse, da der relative Rauschanteil geringer ist und werden daher im Algorithmus bevorzugt. Dies geschieht dadurch, dass die Vektoren bezüglich ihrer Länge geordnet und anschließend zuerst die Kombinationen der längsten OF-Tupel zur Berechnung der ersten Translationsparameter, dem *maybe-Modell*, herangezogen werden. Dieselben Änderungen wie bei der Rotationsberechnung (siehe Abschnitt 5.2) finden auch bei der Translationsberechnung ihre Anwendung. Das heißt, die Angabe der minimalen Anzahl der Inlier erfolgt als Prozentsatz und bei der Berechnung des durchschnittlichen Fehlers wird auch wieder überprüft, ob die *alsoinliers* immer noch der zweiten, über alle Inlier berechneten, Translation entsprechen.
Eine weitere Abänderung hat sich bei Tests im Laufe der Entwicklung als sinnvoll erwiesen: Der maximale Abstand der Ist- zur Soll-Translation wird auf die Länge des Vektors bezogen und demnach als Prozentsatz angegeben. Dies ist daher sinnvoll, da der eigentliche Fehler ein Richtungswinkel-Fehler ist und so ein großer Abstand bei langen Vektoren den gleichen Fehler bedeutet wie ein kleinerer Abstand bei kleineren OF-Vektoren.
Der so abgeänderte RANSAC-Algorithmus wird aber noch in eine äußere Schleife gepackt, die folgenden Zweck erfüllt: Kleine OF-Vektoren sind doppelt problematisch für die Translationsberechnung. Dasselbe Rauschen bei kurzen Vektoren bedeutet eine wesentlich größere Richtungsänderung des OF-Vektors als bei langen. Weiters handelt es sich bei kleinen Vektoren oft auch nur um translationsinvariante Punkte, die jedoch durch Rauschen oder andere Störungen den Rotations-Inlier-Grenzwert überschritten haben. Aus diesen beiden Gründen sollten kurze OF-Vektoren möglichst nicht zur Translationsberechnung herangezogen werden - es ist sinnvoll eine Mindestlänge der Vektoren zu fordern. Gleichzeitig bedeutet das aber auch, dass bei geringer Translation keine Richtung bestimmt werden kann, da die meisten Vektoren unter dem geforderten Schwellwert liegen und jene, die lang genug wären, zum größten Teil Ausreißer sind, die die Berechnung verfälschen. Als Lösung wird eine schrittweise, lineare Verringerung des Grenzwertes bis Mindestlänge Null durchgeführt. Nur wenige Schleifendurchläufe (in den Tests wurden drei oder fünf

verwendet) sind notwendig, um unabhängig von der Länge der OF-Vektoren, ein robustes Ergebnis zu erzielen. Für den eigentlichen RANSAC-Translations-Algorithmus werden nur mehr OF-Tupel verwendet, die der Längeneinschränkung genügen. Sobald ein Ergebnis gefunden ist, wird die Schleife verlassen.

Somit sind folgende Parameter für die einhüllende Schleife notwendig:

Eingabe:

- *steps:* Anzahl der Schritte, in denen die Mindestlänge der verwendeten OF-Tupel verringert wird.
- *minDist:* Ausgangswert für die Mindestlängen der zu verwendenden OF-Tupel.

Folgende Parameter werden für den angepassten RANSAC-Algorithmus verwendet:

Eingabe:

- *data:* Der durch das jeweilige *minDist* eingeschränkte Datensatz (OF-Tupel inklusive entsprechender Indizes).
- *model:* Das Modell, das es zu berechnen gilt, ist die Translation. Die Translation wird dabei, wie in Abschnitt 5.3.1 beschrieben, aus zwei oder drei bzw. mehreren Punkten berechnet und benötigt zwei Parameter (*accuracyDist*, *accuracyAngle*).
- *n:* Die minimale Anzahl an Punkten für die Translationsbestimmung ist zwei. Es hat sich aber herausgestellt, dass man über drei OF-Tupel eine robustere Ausgangstranslation erhält.
- *k:* Die Anzahl der Iterationen, in denen neue Modelle berechnet werden richtet sich nach der Anzahl der Ausreißer im Datensatz.
- *t:* Der Grenzwert für den tolerierten Fehler der Inlier ist abhängig von der Pixelgröße und vom Rauschen
- *d:* Minimaler Prozentsatz an Punkten in data, die für die Translationsberechnung verwendet werden (sogenannte Inlier). Dieser Parameter ist abhängig vom prozentuellen Anteil der Ausreißer.

Ausgabe: *bestfit:* Der beste gefundene Richtungsvektor der Translation bzw. ein Nullvektor, falls kein passendes Modell gefunden wurde.

Der Unterschied zum generischen Algorithmus liegt demnach in folgenden Punkten:

- Umschließende Schleife, welche die Menge der OF-Tupel auf eine Mindestlänge einschränkt. Dieser Grenzwert wird iterativ linear verringert.
- Für die Abschätzung des Fehlers wird nicht ein konkreter Wert verwendet, sondern eine Prozentangabe, die sich auf die Länge des Vektors bezieht, da der Fehl-Abstand mit der Länge der OF-Tupel zunimmt.
- Beim Test, ob es sich um ein passendes Modell handelt und somit genügend passende OF-Tupel gefunden wurden, wird nicht nur die Anzahl der *alsoinliers*, sondern die Summe von *also-* und *maybeinliers* mit dem Prozentsatz verglichen. Dies ermöglicht eine bessere Angabe des minmalen Prozentsatzes, dem die Inlier entsprechen müssen. Gleichzeitig wird verhindert, dass bei einer geringen OF-Vektoren-Zahl die OF-Tupel der *maybeinlier* ins Gewicht fallen.
- Weiters werden die Punkte, die der gemeinsamen Translationsbestimmung nicht mehr genügen, aus der Inlier-Menge entfernt. Anschließend wird erneut überprüft, ob genügend Inlier vorhanden sind, bevor die Parameter übernommen werden können.

5.3.1 Berechnung des Richtungsvektors

Zur Berechnung der Translation wird ein weitverbreitetes, triviales Verfahren verwendet. Zur Erinnerung: Der Algorithmus erhält bereits zurückrotierte und translationsabhängige OF-Tupel $\left(B_{ialt},\ B^*_{i_{neu}}\right)$.
In Gleichung 5.18 wurde $B^*_{i_{neu}}$, der zurückrotierte Enpunkt des OF-Vektors, bereits hergeleitet. Der Epipolargeometrie entsprechend liegt der optische Fluss auf den epipolaren Geraden und die OF-Vektoren schneiden sich so theoretisch (rausch- und störungsfrei) in einem Punkt, dem Epipol. Der Vektor zwischen Epipol und optischen Zentrum der Kamera bildet somit den Richtungsvektor der Translation. Je nachdem ob sich die neuen bwz.

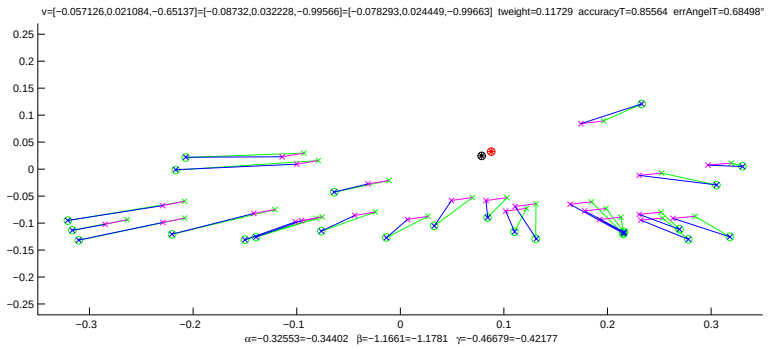


Abbildung 5.6: Hier dargestellt ist das Ergebnis einer Translationsberechnung in Matlab. Die grünen Linien stellen den optischen Fluss dar - Kreise signalisieren die Ausgangspunkte der Vektoren, Kreuze die Endpunkte. Die Rotation wird magentafarben dargestellt und die magentafarbenen Kreuze stellen somit die neuen Endpunkte für die Translationsberechnung dar. Blaue Kreuze symbolisieren die in die geschätzte Translationsrichtung zurückverschobenen Endpunkte. Diese liegen bei erfolgreicher Berechnung in den grünen Kreisen. Mehr zur farblichen Kodierung wird in Abschnitt 7.2 beschrieben.

die alten Bildpunkte näher beim Epipol befinden, handelt es sich um eine Translation nach hinten bzw. nach vorne. Für die Berechnung der Schnittpunkte der Vektoren würde sich die Darstellung der Geraden in der Hesse-Normalform eignen:

$$ax + by + c = 0 \tag{5.22}$$

$$\text{wobei: } a^2 + b^2 = 1 \tag{5.23}$$

Durch die untere Bedingung wird gewährleistet, dass man durch einfaches Einsetzen von Koordinaten den Abstand der Punkte zu der Geraden erhält.
Diese Form ermöglicht zwar eine problemfreie Darstellung von Geraden jeglicher Steigung und das Bestimmen des gemeinsamen Schnittpunktes kann als einfaches Eigenwertproblem gelöst werden, doch leider bringt dieses Verfahren ein gravierendes Problem mit sich. OF-Vektoren kurzer Länge bzw. die Schnittpunkte, die sich aus zwei Geraden mit kleinem Winkel ergeben, sind äußerst ungenau bzw. schlecht konditioniert. Da durch das Lösen des Eigenwertproblems nur einzelne Geraden und somit die Vektoren nur aufgrund ihrer Länge gewichtet werden können, wird der Winkel zwischen den Vektoren vernachlässigt. Das Ergebnis der Epipol-Berechnung ist so oft sehr ungenau und die Aussage über die Zuverlässigkeit der Berechnung korreliert nicht mit dem Fehler der Schätzung. Es muss

also möglich sein, die einzelnen Schnittpunkte der Vektoren aufgrund ihrer Länge und vor allem des einschließenden Winkels zu gewichten. Um das zu gewährleisten, wird die herkömmliche Geradendarstellung mit y als Funktion von x verwendet und alle möglichen Schnittpunkte der Geraden bezüglich ihrer Länge und dem eingeschlossenen Winkel gewichtet.
Dabei werden zwei Zuverlässigkeits-Grenzwerte verwendet: der eine für die minimale Länge der Vektoren (*accuracyDist*), der andere für den minimalen Winkel (*accuracyAngle*). Bei Werten, die darüber liegen, wird der Faktor Eins verwendet, darunter wird entsprechend der Maße ein Faktor kleiner Eins berechnet. Tests in der Entwicklungsphase haben gezeigt, dass durch diese Variante eine wesentlich robustere und genauere Translationsschätzung möglich wird. Zudem hat sich herausgestellt, dass eine sin^2-Gewichtung einer linearen Gewichtung vorzuziehen ist.

$$y = m_i x + b_i \tag{5.24}$$

$$m_i = \frac{y_{B^*_{i_{neu}}} - y_{B_{i_{alt}}}}{x_{B^*_{i_{neu}}} - x_{B_{i_{alt}}}}, \quad \text{wobei: } x_{B^*_{i_{neu}}} - x_{B_{i_{alt}}} \neq 0 \tag{5.25}$$

$$b_i = y_{B^*_{i_{neu}}} - x_{B^*_{i_{neu}}} m_i = y_{B_{i_{alt}}} - x_{B_{i_{alt}}} m_i \tag{5.26}$$

Natürlich gibt es bei dieser Darstellung Probleme bei Geraden, die senkrecht zur x-Achse verlaufen ($m = \pm\infty$). Dieses Problem der Darstellung von vertikalen Geraden ist der Preis der Genauigkeit der Schnittpunkte und kann mittels einfacher Abfragen von $x_{B^*_{i_{neu}}} - x_{B_{i_{alt}}} = 0$ bei der Berechnung der Steigung behoben werden.

$$y^*_k = m_i x^*_k + b_i \ , \quad y^*_k = m_j x^*_k + b_j \quad \Rightarrow \quad x^*_k = \frac{b_i - b_j}{m_j - m_i} \tag{5.27}$$

$$y_k = f_k \ (m_i x^*_k + b_i) \tag{5.28}$$

$$x_k = f_k \ x^*_k \tag{5.29}$$

f_k ist dabei die Gewichtung, die mit Hilfe von *accuracyDist* und *accuracyAngle* bestimmt wird. Diese beiden Parameter hängen vom Rauschen und von der Auflösung der Bilder sowie der Rechengenauigkeit des Hostcomputers ab.

Aus dieser gewichteten Schnittpunkt-Wolke wird nun der Zentroid bestimmt:

$$f_{sum} = \sum_{k=1}^{\frac{n(n-1)}{2}} f_k \tag{5.30}$$

$$x_E = \frac{1}{f_{sum}} \sum_{k=1}^{\frac{n(n-1)}{2}} x_k \ , \qquad y_E = \frac{1}{f_{sum}} \sum_{k=1}^{\frac{n(n-1)}{2}} y_k \tag{5.31}$$

$$\bar{f} = \frac{f_{sum}}{\left(\frac{n(n-1)}{2}\right)} \tag{5.32}$$

Der so berechnete Zentroid entspricht demnach dem gemittelten Epipol. $\bar{f}$ wird als Zuverlässigkeitsmaß der Translationsschätzung verwendet.

Nun wird überprüft in welche Richtung die einzelnen OF-Vektoren orientiert sind.

$$s = \sum_{i=1}^{n} sign\left(\sqrt{\left(x_E - x_{B^*_{i_{neu}}}\right)^2 + \left(y_E - y_{B^*_{i_{neu}}}\right)^2} - \sqrt{\left(x_E - x_{B_{i_{alt}}}\right)^2 + \left(y_E - y_{B_{i_{alt}}}\right)^2}\right) \tag{5.33}$$

Zeigt die Mehrheit zum Epipol ($s > 0$), handelt es sich um eine Vorwärtsbewegung, zeigt die Mehrheit vom Epipol weg ($s < 0$), handelt es sich um eine Rückwärtsbewegung. Sind gleich viele Vektoren nach außen bzw. innen gerichtet ($s = 0$), was bei nur zwei OF-Tupel durchaus der Fall sein kann, so ist der berechnete Epipol ungültig.

5.3.2 Berechnung des Translations-Fehlers

Die Abweichung eines Punktes von einem gegebenen Modell ergibt sich aus der Abweichung des zurückrotierten OF-Endpunktes von der Geraden, die der Epipol und der OF-Ausgangspunkt erzeugen.

Dazu wird der normierte Vektor $\vec{n}$, der senkrecht zur Geraden ist, die durch den Epipol und den Endpunkt des OF-Vektors aufgespannt wird, berechnet. Durch das Skalarprodukt erhält man die Länge der Projektion des OF-Vektors auf $\vec{n}$ und so den kleinsten

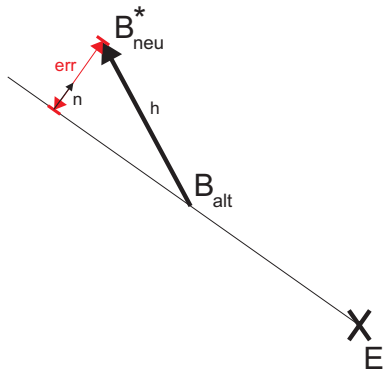


Abbildung 5.7: Da der Epipol über alle OF-Vektoren gemittelt wird, liegen diese in der Regel nicht genau in Richtung des Epipols. Der Fehler ist dabei das Verhältnis des Abstands *err* zur Länge des Vektors *h*.

euklidischen Abstand des Vektor-Enpunktes von der Geraden (vgl. Abbildung 5.7).

$$\vec{n} = norm \begin{pmatrix} -(y_E - y_{B_{alt}}) \\ x_E - x_{B_{alt}} \end{pmatrix} \tag{5.34}$$

$$\vec{h} = \begin{pmatrix} x_{B^*_{neu}} - x_{B_{alt}} \\ y_{B^*_{neu}} - y_{B_{alt}} \end{pmatrix} \tag{5.35}$$

$$err = abs\left(\vec{h} \cdot \vec{n}\right) \tag{5.36}$$

Anschließend wird der Fehler durch die Länge des Vektors geteilt, sodass man eine prozentuelle Angabe des Fehlers erhält. Dies ist notwendig, da der Fehlerwinkel und nicht der effektive Abstand von Bedeutung ist. Durch dieses Verhältnis ist laut Strahlensatz gewährleistet, dass bei gleichem Fehler-Winkel der gleiche Fehlerwert resultiert - unabhängig von der Länge des Vektors bzw. dem Abstand des Enpunktes von der Geraden. Stimmt jedoch das Vorzeichen der Richtung des berechneten neuen Punktes mit dem der Translationsrichtung nicht überein, wird ein möglichst großer Wert (z.B. *INT_MAX*) zurückgegeben - der Punkt ist vermutlich ein Ausreißer und kommt in die Menge der Translations-Outlier. Dabei wird wie in Gleichung 5.33 vorgegangen.

5.4 Die Hinderniserkennung

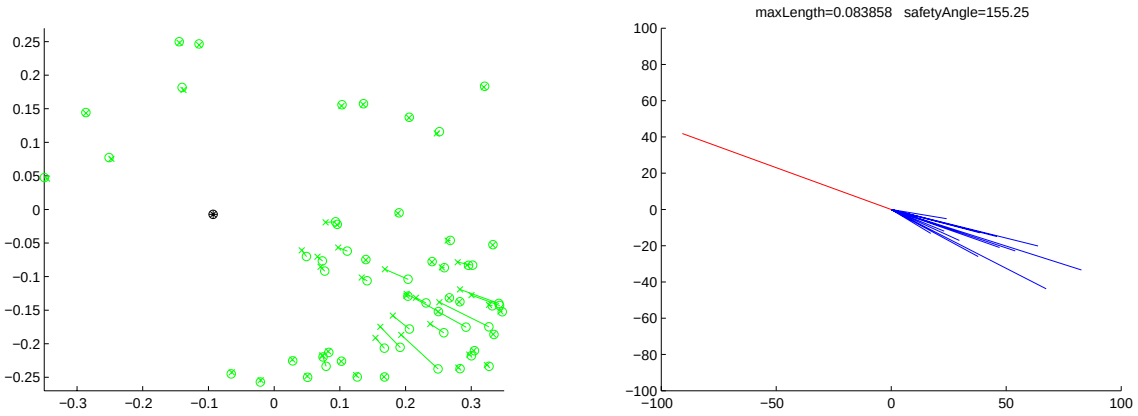
Bei der Hinderniserkennung wird versucht die bestmöglichste Information über eventuelle Hindernisse aus den Bilddaten zu erhalten. Da sich die Kamera in alle sechs Freiheitsgrade bewegen kann, ist es schwer die Richtung zu bestimmen, aus der die Gefahr droht. Trotzdem ist es wünschenswert, die Informationen aus den Bildern bei jeder Bewegung der Kamera deuten zu können. Die Rotation enthält dabei keine nutzbare Information über die Entfernung der Punkte. Ein OF-Vektor, der nur mehr aus der Translationskomponente besteht, ist aber proportional zu folgendem Verhältnis (vgl. Gleichung 4.8):

$$l_i \;=\; f_k h_i \frac{v_k}{z_i^2} \;\propto\; \frac{v_k}{z_i^2} \tag{5.37}$$

Die Brennweite f_k ist eine Konstante der Kamera und der Abstand von der optischen Achse h_i kann vernachlässigt werden. Durch den Öffnungswinkel der Kamera muss dieser sehr klein sein.
Dieser Zusammenhang der Länge l des OF-Vektors ist die einzige Information über die Entfernung z_i eines Punktes, die man aus einem OF-Tupel erhält. Die Länge des Vektors hängt aber nicht nur von der z-Koordinate des Punktes ab, sondern auch von der Geschwindigkeit der Kamera v_k. Aus diesem Grund wird bei der Hinderniserkennung ein Grenzwert für die „gefährliche“ Länge der OF-Tupel verwendet, sodass dieser Wert, falls gewünscht, ständig an die Geschwindigkeit der Kamera angepasst werden kann. Zwei Überlegungen können hier verfolgt werden. Entweder bleibt der Grenzwert konstant, nach der Überlegung, dass bei einer geringen Geschwindigkeit erst nähere Objekte gefährlich werden als bei hoher Geschwindigkeit. Oder die Geschwindigkeit wird über einen anderen Sensor gemessen und der Grenzwert-Parameter wird durch ein übergeordnetes Hostprogramm kontinuierlich angepasst, um Hinderniswarnungen für Objekte bis zu einer gleichbleibenden Entfernung zu erhalten.
Der Algorithmus erhält demnach, wie der Translations-Algorithmus, die zurückrotierten Punkte des optischen Flusses $\left(B_{ialt},\ B^*_{i_{neu}}\right)$.

Der relative Vergleich gilt jedoch bei jeder Geschwindigkeit und jeder Entfernung: Lange OF-Vektoren bedeuten, dass dieser Bereich näher ist als der Bereich mit kürzeren. Da sich die Kamera in alle Richtungen bewegen kann, muss man als Ausgangspunkt für eine Bereichsangabe die Kamera selbst verwenden. Der einzige sinnvolle Referenzpunkt ohne

zusätzliches Vorwissen ist somit die Richtung, in der sich die Kamera bewegt bzw. der Epipol. Das Bild stellt nur einen kleinen sichtbaren Bereich der gesamten Umgebung dar, trotzdem möchte man die erhaltenen OF-Tupel auswerten. Eine sinnvolle Aussage kann daher nur über Winkelbereiche mit dem Epipol als Zentrum getroffen werden. Die OF-Tupel sind in einem gewissen Winkel bezüglich der x-Achse zum Epipol geneigt. Alle Winkel, die einen zu langen OF-Vektor enthalten, werden als „gefährlich“ markiert.



(a) Die OF-Vektoren die zur Hinderniserkennung verwendet werden und der daraus berechnete Epipol.

(b) Das Ergebnis der Hinderniserkennung - rot dargestellt der Ausweichwinkel.

Abbildung 5.8: Hier dargestellt ist das Ergebnis einer Hinderniserkennung in Matlab. Es werden gefährliche Objekte im rechten unteren Quadranten simuliert. Die farbliche Kodierung ist in Abschnitt 7.2 beschrieben.

Der Algorithmus zur Hinderniserkennung besteht somit aus folgenden Schritten:

1. *Berechnung des Epipols*: Um Rechenzeit zu sparen, wird eine vereinfachte Variante des Translatinosbestimmungs-Algorithmus (siehe Abschnitt 5.3.1) zur Epipolberechnung der Hinderniserkennung verwendet. Die Vereinfachung besteht darin, auf die Gewichtung der Schnittpunkte bezüglich der Länge und des einschließenden Winkels der Vektoren zu verzichten. Nicht allzugroße Fehler in der Epipolbestimmung haben auch nur kleine Auswirkungen auf die Definition des Winkelbereichs.

2. *Filtern der OF-Vektoren, die potentielle Hindernisse sein könnten*: Wie bereits erwähnt, wird ein Grenzwert *minLength* verwendet, um zu bestimmen welche OF-

Vektoren nah genug sein könnten, um eine Gefahr darzustellen. Dieser Grenzwert muss dem Sicherheitsabstand und der aktuellen Geschwindigkeit der Kamera angepasst werden.

3. *Eintragen in das Winkel-Array*: Um den Winkelbereich rund um den Epipol darzustellen, wird ein Array verwendet, das je nach Genauigkeit der Einteilung der Winkel aus entsprechend vielen Elementen besteht (z.B. erhält man mit 720 Elementen eine Genauigkeit von einem halben Grad). Nun wird von allen OF-Vektoren, die über dem Grenzwert liegen, der entsprechende Winkel zum Epipol berechnet und so die Position im Winkel-Array bestimmt. Die Länge des Vektors wird berechnet und falls die Winkelposition leer oder der Eintrag kleiner als die berechnete Länge ist, wird der Wert in das Array eingetragen.

4. *Berechnung des Ausweichwinkels*: Nun hat man zwar die Information, wo sich potentielle Hindernisse befinden, aber man weiß nicht, wie man diesen Hindernissen ausweichen kann. Dafür wird der größte Freiraum, also der Bereich mit den meisten leeren Einträgen gesucht und die mittlere Position dieses Bereichs im Winkel-Array bestimmt. Dabei wird angenommen, dass es sich um einen zirkulären Speicher handelt. Der Winkel, welcher der mittleren Position dieses größten Winkelbereichs entspricht, wird als Ausweichwinkel vorgeschlagen.

Der Algorithmus gibt somit folgende drei Werte zurück:

- *Das Winkel-Array*: Dieses Array enthält die Informationen über Hindernisse in den entsprechenden Winkelbereichen um den Epipol.
- *Der Ausweichwinkel*: Entspricht dem Winkel, der am weitesten von den erkannten Hindernissen entfernt ist.
- *Die größte gemessene Länge*: Dieser Wert entspricht der Länge des längsten OF-Vektors und kann als Angabe verwendet werden, wie akut die Situation ist. Entsprechen kann man diesen Wert auch als Maß verwenden, wie stark in den Ausweichwinkel eingelenkt werden soll.

5.5 Zusammenhänge zwischen den einzelnen Berechnungen

Bei der Entwicklung des Algorithmus wurde darauf geachtet, alle vorhandenen Informationen sinnvoll zu nutzen und jegliches erhaltbares Wissen in das Verfahren zu integrieren. Die Rotationsberechnung erfolgt für jedes Bild, da kürzere Translationen mehr translationsinvariante Punkte erwarten lassen. Auf der anderen Seite soll die Translationsberechnung erst nach möglichst vielen Bildern erfolgen, da bei großen Translationen die Richtung genauer und robuster bestimmbar ist. Dafür werden die Indizes der translationsabhängigen Merkmale über die Bildsequenz verfolgt und die Rotation, die für jedes Bild bestimmt wird aufsummiert.
Gleichzeitig muss darauf geachtet werden, dass die translationsabhängigen Merkmale, die im Bild verfolgt werden, sich über die gewünschte Anzahl der Frames überhaupt verfolgen lassen. Da sie sehr nahe an der Kamera sind, können sie leicht aus deren Blickfeld geraten. Zudem hat sich gezeigt, dass bei zu großen Rotationen die Genauigkeit der Translation stark abnimmt, da der Rotations-Restfehler erfahrungsgemäß größer ist. Aus diesem Grund wurde ein Grenzwert für den maximalen Winkel der gemessenen Rotation eingeführt, bei dessen Überschreiten eine Translationsschätzung stattfindet, auch wenn noch nicht die für die Translation geforderte Anzahl an Bilder erreicht wurde. Die Anzahl der Bilder, die für die Translationsberechnung zusammengefasst werden, hängt somit von der Geschwindigkeit der Kamera, der Bildrate und der Stärke der berechneten Rotation ab.

Die Translationsberechnung und Hinderniserkennung ist nur möglich, wenn die Rotationsbestimmung erfolgreich war. Darum wird bei einer nicht erfolgten Rotationsberechnung die Translation über die bisher gespeicherte Bildsequenz ausgeführt, um die bis zu diesem Punkt angesammelte Translations-Information nicht zu verlieren.
Die Hinderniserkennung soll möglichst schnell und mit geringem Rechenaufwand erfolgen. Indem man aber nicht für alle, sondern nur für je zwei Bilder diese Berechnungen durchführt, kann ein Großteil der Ausreißer herausgefiltert werden. Die Ausführung der Hinderniserkennung auf je zwei Bilder hat sich in den Tests als sehr erfolgreich erwiesen. Auf die Rotationswinkel-Abfrage und alternative Vorgehensweise bei misslungener

Rotationsberechnung kann durch diese kleine Bildfolge verzichtet werden.

5.6 Bewertung

Da der Algorithmus eigentlich aus drei Teilalgorithmen und einem übergeordneten Kontrollprogramm besteht (vgl. Abbildung 6.4 und C.1), können die verschiedenen Teilverfahren auch einzeln und unterschiedlich gewertet werden. Zunächst ist zu erwähnen, dass der Algorithmus sehr viele anwendungsspezifische Parameter benötigt, was die Berechnungen anpassungsfähig und robust macht, gleichzeitig aber etwas Vorwissen, Vorarbeit und vor allem Tests benötigt. Eine passende Parametrisierung ist bei der folgenden Bewertung vorausgesetzt.
Die Rotationsbestimmung, also grob Aruns Algorithmus in einer RANSAC-Hülle, ist sehr robust und liefert sehr gute Ergebnisse auch bei verrauschten Bildern (siehe Kapitel 7). Das Verfahren von Arun funktioniert im Gegensatz zum 8-Punkt-Algorithmus auch wenn alle Punkte in einer Ebene sind und versagt erst, wenn alle Punkte auf einer Linie liegen. Dadurch, dass nicht die gesamte Fundamentalmatrix, sondern nur die Rotation berechnet wird, gibt es auch keine sogenannten „gefährlichen Flächen“ wie beim 8-Punkt-Algorithmus. Vor allem gibt es keine Probleme mit reiner Translation.
Bei der Berechnung der Translation gibt es aber auch bei diesem Verfahren Probleme mit zu kleinen Bewegungen, da in diesen Fällen die durch das Rauschen bedingte Ungenauigkeit zu groß wird. Bei Translationen, die stark parallel zur Bildebene sind, kann es zu größeren Ungenauigkeiten kommen. Der Epipol befindet sich in diesen Fällen weit weg vom Ursprung des Bildebenen-Koordinatensystems (Richtung Undendlich), sodass das System einfach schlecht konditioniert ist. Dadurch, dass die Schnittpunkte bezüglich der Länge und des einschließenden Winkels der erzeugenden Vektoren gewichtet werden, wird diesem Problem entgegengewirkt. Im Gegensatz zum 8-Punkt-Algorithmus, bei dem alle OF-Tupel gleich bewertet werden, erhält man ein Maß für die Zuverlässigkeit der Schätzung. Dieses Zuverlässigkeitsmaß ist insofern von besonderer Bedeutung, da sich damit dieses Verfahren sehr gut für den Einsatz mit einem Kalman-Filter eignet. Der 8-Punkt-Algorithmus liefert keine Information über die Genauigkeit bzw. Zuverlässigkeit der Schätzung und ist daher ungeeignet für den Einsatz in einem Kalman-Filter.

Zudem erhält man über den RANSAC-Algorithmus den durchschnittlichen Fehler für die Modellberechnung. Es läge somit nahe, den Zuverlässigkeitswert der Translationsberechnung mit dem durchschnittlichen Fehler der Rotationsbestimmung zu gewichten. Schlechte Rotationsparameter haben zur Folge, dass die OF-Vektoren nur entsprechend ungenau zurückrotiert werden können und die Translationsschätzung so auch größere Fehler aufweist. Da RANSACs bester durchschnittlicher Fehler jedoch nur schlecht mit dem tatsächlichen Fehler korreliert, wird die Zuverlässigkeit der Translationsberechnung nicht mit dem errechneten Fehlerwert der Rotationsbestimmung gewichtet (vgl. (d)-Bilder in den Simulationsergebnissen in Abschnitt 7.3).

Für den gesamten Algorithmus ist keinerlei Wissen über die gefundenen Punkte in der Umgebung notwendig, wie es bei Nister oder aber auch Davison der Fall ist. Das Verfahren zielt auf eine Anwendung in der freien Natur, wobei größere Strecken zurückgelegt werden und die Kamera nicht nur lokal zirkuliert. In diesem Fall hätte Davisons SLAM-Verfahren seine Schwierigkeiten, weil immer neue Punkte in den Zustandsvektor aufgenommen werden müssten und alte Referenzen nicht zur Stabilisierung der Karte und Einschränkung der Ungenauigkeiten verwendet werden können.

Ziel des Algorithmus ist es nicht eine Umgebungs-Karte aufzubauen, sondern die relative Bewegung der Kamera zu schätzen, was auch mit großer Zuverlässigkeit, Genauigkeit und wenig Rechenzeit gelingt.

Kapitel 6

Software Entwicklung

Bei der Entwicklung des Algorithmus wurde folgende Strategie verfolgt: Theoretische Grundlagen und Skizzen wurden zu Beginn auf Papier gebracht. Anschließend wurden die Ideen in Matlab implementiert (vgl. Abbildung 6.1), um die Funktionsweise und Richtigkeit der Routinen bzw. des gesamten Verfahrens zu überprüfen. Schlussendlich wurde der gesamte Algorithmus in C++ unter Eclipse-Fedora programmiert.
Somit wurden alle Methoden sowohl in Matlab als auch in C++ geschrieben. Diese Vorgehensweise ist zwar aufwendig, bietet aber folgende Vorteile:

- In Matlab lassen sich einfach und schnell neue Ideen umsetzen, Ergebnisse visualisieren und Testdaten erzeugen.
- Korrespondierende Funktionen können effizient verglichen werden und die Fehlersuche wird erleichtert, da beide Varianten, bis auf Rundungsungenauigkeiten (Matlab hat eine höhere Auflösung der Zahlen) dasselbe Ergebnis liefern müssen.
- Die Implementierung in C++ ist echtzeitfähig und auf beliebigen Systemen einsetzbar. Eine der Vorgaben der Arbeit war es, das Verfahren echtzeitfähig zu halten und in C++ zu implementieren.

Bei der C++ Variante wurde eine Schnittstelle implementiert, die es ermöglichte Datensätze von OF-Tupel, die von Matlab im Rahmen einer Simulation generiert und in eine Datei geschrieben wurden, auszulesen und als Datenquelle zu verwenden (vgl. Ab-

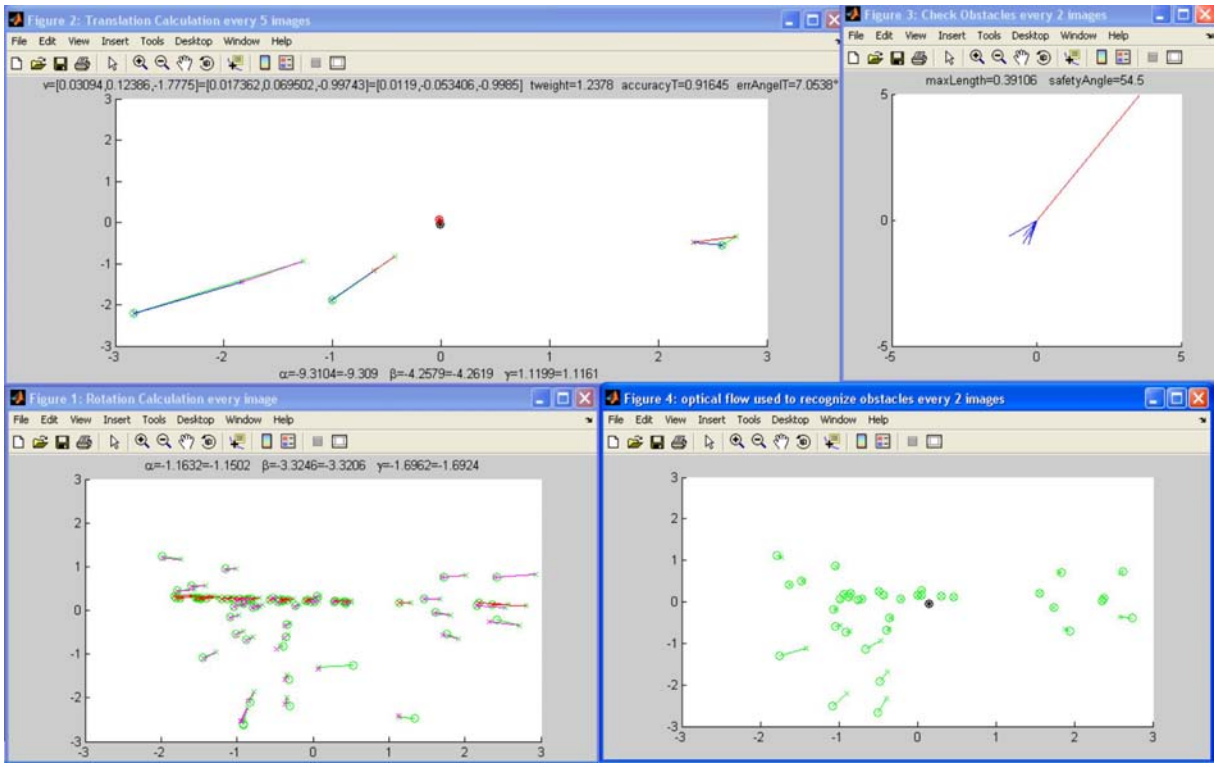


Abbildung 6.1: Dieser Screenshot wurde während einer Simulation mit der Matlab-Implementierung gemacht. Matlab stellt bereits eine enorme Bibliothek mit mathematischen Routinen zur Verfügung, auch die Visualisierung ist wesentlich einfacher. So konnten Ideen schnell umgesetzt und mit einer Simulationsumgebung verifiziert werden.

bildung 6.2). Analog wurde für Matlab eine Schnittstelle generiert um Dateien, die mit den Daten des KLT-Tracker beschrieben sind, auszulesen und zu verarbeiten. Durch diese Überkreuzung der Varianten war es möglich, die Genauigkeit und Fehlerfreiheit des C++ Programms mit den Simulationsdaten aus Matlab zu überprüfen. Gleichzeitig konnten die Ergebnisse der C++ Variante bei realen Bildern über Matlab verifiziert werden.
Um das Verfahren in den verschiedensten Situationen testen zu können, wurde in Matlab ein umfangreiches Umgebungsmodell geschaffen, dass es zulässt, Rauschen, Ausreißer, Pixeldiskretisierung, entfernte Punkte, nahe Punkte, Hindernisse in den verschiedenen Quadranten des Bildes, usw. zu simulieren und beliebig zu kombinieren.

Viel Energie wurde auch für eine ausführliche Visualisierung aufgewendet. So wird in der Matlab-Umgebung das Rotations-Ergebnis, das Translations-Ergebnis sowie der optische

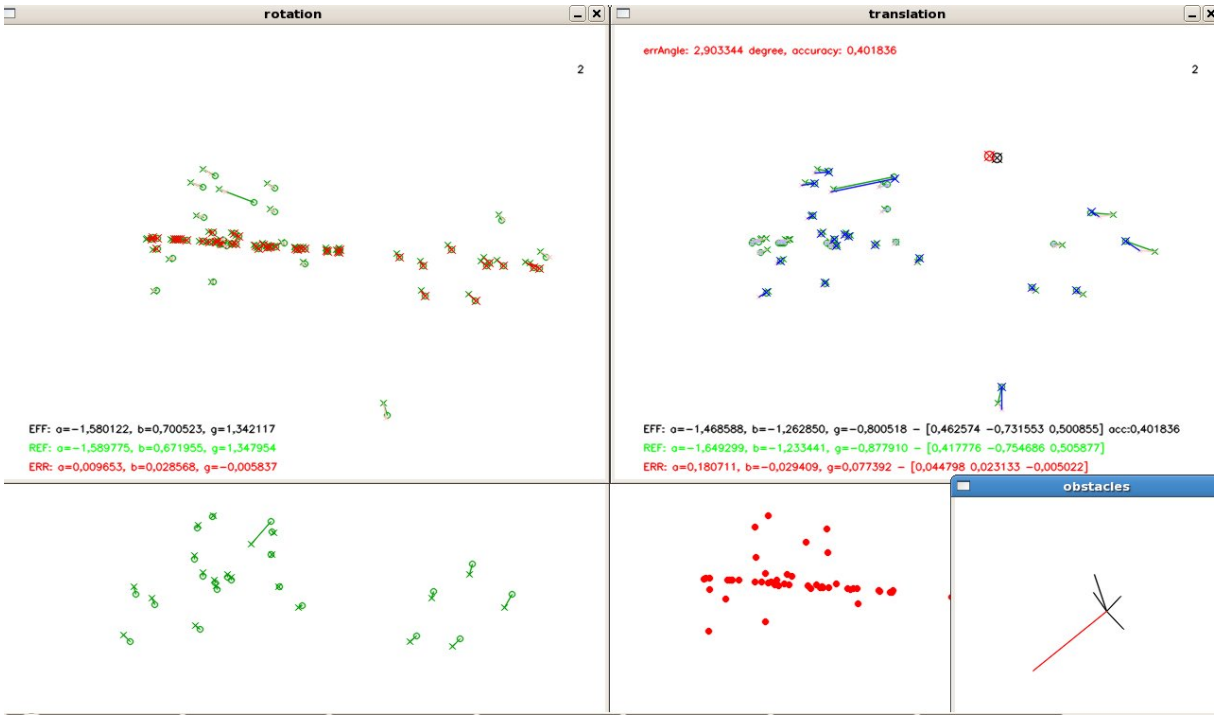


Abbildung 6.2: Dieser Screenshot wurde während einer Simulation des C++-Programm gemacht. Dabei werden nicht Bilder als OF-Quelle verwendet, sondern mit Matlab generierte Simulationsdaten.

Fluss, der für die Hinderniserkennung verwendet wird, und das Ergebnis der Hinderniserkennung selbst angezeigt. Beispiele dafür finden sich in Kapitel 5.
Für die C++-Variante wurde zudem eine aufschlussreichere Visualisierung implementiert, da diese hauptsächlich bei realen Bildern Anwendung findet (siehe Abbildung 6.3). Die Visualisierung des Ergebnisses ist von größter Bedeutung, wenn es darum geht einen Algorithmus zu verbessern, Schwachstellen zu erkennen und Parameter zu bestimmen. Die Kodierung der Farben wird in Abschnitt 7.2 beschrieben.

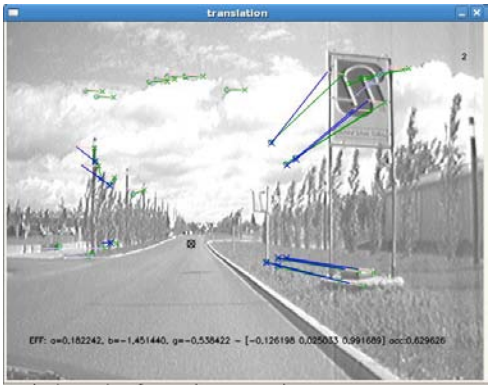
Um die OF-Tupel aus den Bildern zu extrahiern wurde der KLT-Tracker verwendet (siehe Appendix B). Um Vektoren und Matrizen einfach handhaben zu können, wurden Teile der XVision2-Bibliothek[1] mit eingebunden.
Eine Schnittstellendefinition für die Datenquelle dient dazu, den Algorithmus bei beliebigen Anwendungen unter verschiedensten Bedingungen einsetzen zu können. Das Projekt

[1]http://www.cs.jhu.edu/CIPS/xvision/

(a) Ergebnis des Trackers - die roten Punkte konnten verfolgt werden.

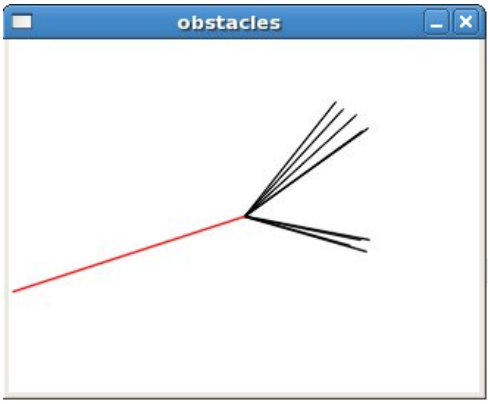
(b) Für die roten Punkte in 6.3(a) kann der optischer Fluss bestimmt werden.

(c) Ergebnis der Rotationsbestimmung.



(d) Ergebnis der Translationsbestimmung.

(e) Optischer Fluss der zur Hinderniserkennung herangezogen wird.



(f) Ergebnis der Hinderniserkennung.

Abbildung 6.3: Hier dargestellt die Visualisierung der Ergebnisse bei einer Fahrt durch das Forschungsgelände in Garching (mehr in Abschnitt 7.4).

ist sehr modular gehalten, um die Übersicht zu wahren und Komponenten einfach austauschen zu können (siehe Abbildung 6.4). So könnte z.B. die Translations- oder Rotationsberechnung einfach ausgetauscht werden, ohne das restliche Programm zu beeinflussen.

Im Rahmen dieser Arbeit wurden um die 10.000 Zeilen C++-Code und ca 5.000 Zeilen Matlab-Code geschrieben. Zur besseren Weiterverarbeitung und Verwendung wurde eine Doxygen[2]-Dokumentation des C++-Programms erstellt. In Appendix C wird nochmals genauer auf die einzelnen Klassen der Software eingegangen und Abbildung C.1 stellt ein vereinfachtes Klassendiagramm der C++-Software dar.

[2]http://www.stack.nl/ dimitri/doxygen/

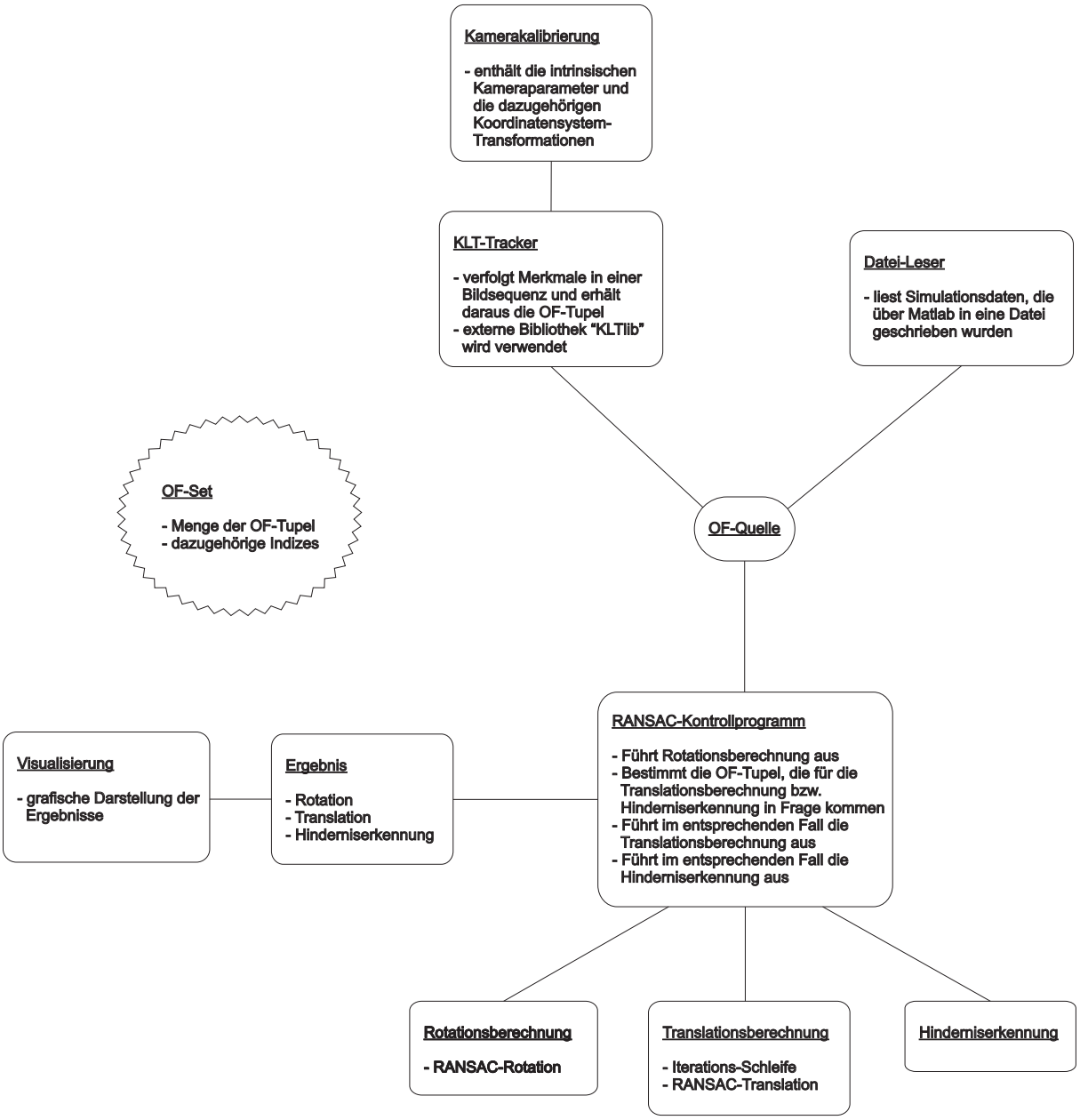


Abbildung 6.4: Diese Grafik soll den modularen Aufbau der Software veranschaulichen und einen kurze Erklärung zu den einzelnen Modulen liefern. Das Modul bzw. die Klasse OF-Set stellt den Datentyp der optischen Fluss Vektoren dar, die im gesamten Programm verwendet werden.

Kapitel 7

Ergebnisse

Es ist sinnlos irgendwelche Theorien aufzustellen oder Schlussfolgerungen zu treffen, wenn man keine Beweise dafür aufbringt. Aus diesem Grund wurden einige Simulationen mit dem entworfenen Algorithmus durchgeführt, die seine Genauigkeit und Geschwindigkeit belegen sollen. Gleichzeitig wurde das Verfahren an realen Bildsequenzen getestet, die mit verschiedenen Kameras in verschiedenen Umgebungen durchgeführt wurden. Die aufschlussreichsten Ergebnisse dieser Simulationen und Tests werden in diesem Kapitel präsentiert.

Damit eine optimale Funktionalität des Verfahrens gewährleistet werden kann, müssen die passenden Parameter dafür gefunden werden. Da die Testergebnisse stark von der Anwendung bzw. der Quelle der RANSAC-Daten abhängig sind, empfiehlt es sich für jede Anwendung eigene Tests durchzuführen, um die passenden Parameter zu finden.

7.1 Parameterwahl

In diesem Abschnitt werden die verwendeten Parameter für die Tests aufgelistet und Erfahrungen bzw. Begründungen für die Parameterwahl angegeben.

Parameter des Gesamtalgorithmus

- *numTrans:* Maximale Anzahl an Bilder, nach denen eine Translationsberechnung er-

folgen soll. Dieser Wert ist stark abhängig von der Bildrate - so wurden in der Simulation Wert von 3 bis 20 verwendet. Bei den realen Bildern, die mit einer Frequenz von ca. einem Herz aufgenommen wurden, durften maximal drei Bilder zwischen jeder Translationsberechnung vergehen, um noch genügend translationsabhängige Merkmale über die Bildsequenz verfolgen zu können.

- *angLimitTrans:* Wird dieser Grenzwert von einem der Winkel der aufsummierten Rotationsmatrix erreicht, wird unabhängig von *numTrans* sofort eine Translationsberechnung durchgeführt. In der Simulation konnten mit einem Grenzwert von 6° noch gute Translationsberechnungen erzielt werden.
- *numCheckObstac:* Maximale Anzahl an Bilder, nach denen eine Hinderniserkennung erfolgen soll. Es hat sich herausgestellt, dass zwei Bilder ausreichend sind, um Ausreißer zu unterdrücken, gleichzeitig wird eine schnelle Hinderniserkennung gewährleistet.

Parameter der Rotationsberechnung

- *maxIterR:* Anzahl an Iteration, die für den RANSAC-Rotations-Algorithmus verwendet werden. Wie in Abschnitt 5.2 berechnet, ist man mit 40 Iterationen auf der sicheren Seite. Dies trifft nicht nur theoretisch zu, sondern auch in der Praxis wurden mit diesem Wert stets Rotationen gefunden und es wurden auch mit weitaus weniger Schritte, z.B. 10, noch gute Ergebnisse erzielt, da die Information über vorangegangene Inlier verwendet wird.
- *minReqT:* Drei OF-Tupel sind für Aruns Algorithmus notwendig und ausreichend.
- *threshFitR:* Je nach Rauschen muss ein entsprechender Wert gewählt werden. Meistens reicht ein Wert aus, welcher der Länge von ein bis zwei Pixeln entspricht.
- *minClosedPercR:* Bei 717 Bildern, die bei einer Fahrt mit dem Auto durch eine Kleinstadt (Garching bei München) und durch das Forschungsgelände der TUM gemacht wurden, konnten im Schnitt 57 Punkte von einem Bild zum nächsten verfolgt werden. Von diesen Punkten konnten wiederum ca. 35, das heißt 61% als Rotations-Inlier, identifiziert werden. In 95% der Fälle waren mindestens 30% der getrackten Punkte Rotations-Inlier.

Parameter der Translationsberechnung

- *maxStepsT:* Damit man auch bei geringerer Translation eine Lösung findet, wird der Wert, der durch *minDistT* vorgegeben wird, in *maxStepsT* Schritten verringert. Sobald eine Lösung gefunden wurde, wird diese Schleife verlassen. Hier reichten bei den realen Bildern Werte von 3 bis 5 aus, um gute Ergebnisse zu erzielen. Umso feiner die Unterteilung gemacht wird, desto weniger läuft man Gefahr, dass Rauschen auf die Berechnung Einfluss nimmt.

- *minDistT:* Dieser Grenzwert gibt an, wie lang die OF-Vektoren im ersten Durchgang sein müssen. Je nach Bildrate, Anzahl der Bilder, die für die Translationsberechnung zusammengefasst werden, Auflösung und Geschwindigkeit der Kamera muss eine entsprechende Länge gewählt werden. Bei der Simulation in Matlab wurde eine Länge von 30 Pixel verwendet.

- *maxIterT:* Anzahl an Iterationen, die für den RANSAC-Translations-Algorithmus durchgeführt werden sollen. Da nur mehr Rotations-Outlier und so translationsabhängige OF-Tupel vorhanden sind, würde theoretisch auch nur eine Iteration ausreichen. Je nach Rausch und vor allem Ausreißer-Anteil ist es jedoch sinnvoll 10 bis 20 Iterationen zu verwenden.

- *minReqT:* Mindestens zwei OF-Tupel sind für die Translationsberechnung notwendig. Tests während der Entwicklung haben jedoch gezeigt, dass die Verwendung von drei OF-Vektoren schneller ein Ergebnis liefert.

- *threshFitPercT:* Je nach Rauschen muss der entsprechende Wert gewählt werden. In den Tests hat sich eine Toleranz von 10% (der Länge der Vektoren) bewährt.

- *minClosedPercT:* Theoretisch müsste der Datensatz nur mehr aus translationsabhängigen OF-Tupeln bestehen, doch leider ist der Prozentsatz an Translations-Inlier je nach Rauschen und Ausreißer-Verhältnis weit unter 100%. Bei den realen Bildern konnte ein Anteil von mindestend 60% als Inlier eingestuft werden.

- *accuracyDistT:* Dieser Wert wird für die Berechnung des Richtungsvektors benötigt. Ist ein OF-Vektor kürzer als dieser Wert vorgibt, wird der Schnittpunkt über eine quadratische Sinusfunktion mit einem Faktor kleiner Eins gewichtet. Für diesen

Parameter können ähnliche Werte wie für *minDistT* verwendet werden.

- *accuracyAngleT:* Auch dieser Wert wird für die Berechnung des Richtungsvektors benötigt. Ist der eingeschlossene Winkel zwischen zwei OF-Vektoren kleiner als dieser Wert vorgibt, wird der Schnittpunkt über eine quadratische Sinusfunktion mit einem Faktor kleiner Eins gewichtet. Der Zuverlässigkeitswert ergibt sich so aus dem Produkt dieses Faktors und der Gewichtung, die sich aus der Länge der einzelnen OF-Vektoren und *accuracyDistT* ergibt.

Parameter der Hinderniserkennung

- *exactnessCO:* Der Winkelbereich um den Epipol wird in einzelne Teile zerlegt. Dieser Parameter gibt an, wie genau diese Einteilung erfolgen soll und somit auch die Größe des Winkel-Array.
- *minLengthCO:* Dieser Grenzwert bestimmt, ab welcher Länge ein OF-Vektor als gefährlich eingestuft wird.
- *minFreeAngleCO:* Mit diesem Parameter kann angegeben werden, wie groß die Hindernislücke im Winkel-Array mindestens sein muss, um den Sicherheitswinkel anzugeben. Die Angabe erfolgt in der Anzahl der Winkelteile und ist somit abhängig von *exactnessCO*.

7.2 Visualisierung

Um nachfolgende Ergebnis-Visualisierungen zu verstehen, wird hier kurz auf die farbliche Kodierung eingegangen:

- *grüne Linie*: Grün wird der eigentliche OF-Vektor dargestellt. Ein Kreis deutet dabei auf den Ausgangspunkt hin, ein Kreuz auf den Endpunkt des Vektors.
- *rote/magentafarbene Linie*: Diese beiden Farben signalisieren, dass die Linie durch die Rückrotation zustande gekommen ist. Der Endpunkt, der dem Ergebnis der Rückrotation enspricht, wird stets als Kreuz in der entsprechenden Farbe markiert. Rot bedeutet dabei, dass es sich bei diesem OF-Tupel um einen Rotations-Inlier handelt, Magenta steht für einen Rotations-Outlier.

- *hellblaue/blaue Linie*: Diese beiden Farben signalisieren, dass die Linie durch die Rücktranslation zustande gekommen ist. Wie bei der Rotation wird der Endpunkt, der dem Ergebnis der Rücktranslation enspricht, stets als Kreuz in der entsprechenden Farbe markiert. Analog zur Rotation steht Blau für einen Translations-Inlier, während Hellblau auf einen Translations-Outlier hinweist.

- *roter Stern (Kreuz mit Kreis)*: Das Zeichen symbolisiert den eigentlichen Epipol, also die simulierte Translation. Diese Angabe kann und wird nur bei den mit Matlab generierten Simulationsdaten angegeben.

- *schwarzer Stern (Kreuz mit Kreis)*: Dieses Symbol steht für den berechneten Epipol, also die geschätzte Translation.

Beide Sterne können außerhalb des Bildes liegen (z.B. bei Seitwärts-Bewegungen) und daher nicht sichtbar sein.
Hinweis: Bei einer erfolgreichen Bewegungsschätzung muss somit das blaue Kreuz im grünen Kreis liegen.

Matlab-Simulation: Beim Ergebnis-Bild der Rotation wird in der Textangabe zunächst der simulierte Winkel und anschließend der berechnete Winkel angegeben. Bei der Translation wird am unteren Bildrand die summierte Rotation im selben Format angeführt. Am oberen Bildrand stehen die translationsspezifischen Werte: der Translationsvektor, der normierte Translationsvektor, die berechnete Translationsrichtung, der längste Translations-Inlier, die berechnete Zuverlässigkeit der Messung und der Winkel zwischen dem simulierten und dem berechneten Translationsvektor. Das Koordinatensystem ist dabei so festgelegt, dass die x-Achse nach rechts zeigt, die y-Achse nach unten und die z-Achse in das Bild und somit in Kamerarichtung (vgl. Abbildungen in Kapitel 6).
C++-Variante: In C++ werden bei der Visualisierung ähnliche Werte angegeben. Am unteren Bildschirmrand steht dabei in Schwarz die berechneten Werte, darunter in Grün die simulierten Werte und anschließend in Rot die Differenz, also der Fehler. Die restlichen Angaben verstehen sich aus der Beschriftung. Das Tracker-Ergebnis besteht aus roten und grünen Punkten, wobei die grünen Punkte die verfolgten Merkmale darstellen und die roten die neu hinzugenommenen Merkmale signalisieren (vgl. Abbildungen 6.2 und 7.12).

Bei der Hinderniserkennung werden die gefährlichen Winkel schwarz und der Ausweichwinkel rot dargestellt.

7.3 Simulation

Um den Algorithmus kontrolliert testen zu können, wurde in Matlab ein Umgebungsmodell implementiert, in dem kontrolliert weißes Rauschen, Ausreißer sowie Pixel-Diskretisierung hinzugefügt werden können. Reale Umstände können kontrolliert simuliert und das Verhalten des Algorithmus beobachtet werden. So konnte die Zuverlässigkeit und die Genauigkeit unter verschiedensten Umständen getestet werden.
Das Umgebungsmodell wird mit über 100 zufälligen Punkten in der Welt realisiert, die folgende Werte einnehmen können: In z-Richtung (Entfernung) von 0 bis 1000 Meter, in x- Richtung (Breite) den gesamten Bereich des Öffnungswinkel und in y-Richtung (Höhe) von 0 bis 100 Meter, wobei die Kamera in 50 Meter Höhe angenommen wird. Die Kamera wurde mit einer Brennweite von 8,6 mm, einer Auflösung von 768 x 576 und einer Pixelgröße von 8,6μm x 8,6μm simuliert. Das ergibt eine Chip-Größe von 6,4mm x 4,9mm. Diese Werte wurden auf Einheits-Brennweite (f=1) normiert, um Messwerte zu erhalten, die unabhängig von der Brennweite sind („unit focal“).
Bei der Kamera wurden folgende zufällige Bewegungen simuliert: 0 bis 0,5° Rotation pro Bild und Achse und 0 bis 0,05 Meter Translation pro Bild für die x- und die y-Achse und 0,02 bis 0,1 Meter in z-Richtung. So wird vor allem eine Bewegung in Blickrichtung simuliert. Die Translationsberechnung erfolgt spätestens nach jedem zehnten Bild. Das bedeutet bei einer Bildrate von 30Hz, dass jede Sekunde eine Rotation bis 15° erfolgen kann bzw. eine Translation in x- und y-Richtung von 1,5 Meter. In z-Richtung wurde eine Geschwindigkeit von bis zu 3 $\frac{m}{s}$ erlaubt.
Für die Geschwindigkeitsmessung wurde die C++-Implementierung des Algorithmus verwendet, ansonsten wurden die Berechnungen mit der Matlab-Variante durchgeführt.

7.3.1 Genauigkeit

Für die Berechnung der Genauigkeit wurden folgende Situationen simuliert:

1. *ideale Situation*: kein Rauschen, keine Ausreißer und keine Diskretisierung.
2. *geringes Rauschen*: weißes Rauschen ($\sigma^2 = 0.05$), keine Ausreißer und keine Diskretisierung.
3. *starkes Rauschen*: weißes Rauschen ($\sigma^2 = 0.1$ bzw. $\sigma^2 = 0.2$), keine Ausreißer und keine Diskretisierung.
4. *Ausreißer*: kein Rauschen, 5% Ausreißer und keine Diskretisierung.
5. *Diskretisierung*: kein Rauschen, keine Ausreißer und Pixel-Diskretisierung.
6. *schlechter Fall*: weißes Rauschen ($\sigma^2 = 0.05$), 2% Ausreißer und Pixel-Diskretisierung.

Um das weiße Rauschen zu realisieren, wurde nach der Kameraprojektion bei der x- und der y-Komponente jeweils ein normalverteilter, mittelwertfreier Zufallswert mit entsprechender Varianz hinzugefügt. Ausreißer wurden durch zufällige OF-Tupel in der Bildebene mit einem Maximalwert von 10 Pixel pro Bild realisiert. Der KLT-Tracker würde durch die Einschränkung des Suchraums auch nicht längere Ausreißer liefern, doch kann sich dieser Fehler durch das Aufsummieren der Bilder auch entsprechend vergrößern - 10 Bilder à 10 Pixel bedeutet eine maximale Länge der Ausreißer von 100 Pixel. Bei der Diskretisierung wurde die neue Position der Merkmale wie folgt berechnet:

$$x_{neu} = \left(\left(\frac{x_{alt}}{s_x} \; modulo \; 1 \right) + 0.5 \right) s_x \tag{7.1}$$

$$y_{neu} = \left(\left(\frac{y_{alt}}{s_y} \; modulo \; 1 \right) + 0.5 \right) s_y \tag{7.2}$$

Für jeden Testlauf wurde jeweils eine Sequenz von 2.000 Bilder verwendet. Es wird für jede Situation nur ein Durchlauf aufgezeigt, obwohl für jeden Fall mehrere Läufe mit einer unterschiedlichen Initialisierung des Zufallsgenerators durchgeführt wurden. Die präsentierten Ergebnisse entsprechen dabei nicht dem besten Durchlauf (z.B. hat der beste Durchlauf bei idealen Bedingungen knapp 1° durchschnittlichen Translationsfehler betragen), sondern sind so gewählt, dass auch auf Probleme des Algorithmus hingewiesen

werden kann.
Da bei der Translationsschätzung nicht die Länge, sondern nur die Richtung der Verschiebung bestimmt wird, kann der Fehler auch nur als Richtungsabweichung und somit in Grad und nicht in Metern angegeben werden. Daher ist im Folgenden stets die Richtungsabweichung der geschätzten von der simulierten Translation gemeint, wenn von einem Translationsfehler die Rede ist.

1. Ideale Situation
Bei subpixelgenauer Merkmalsbestimmung werden ohne Rauschen und ohne Ausreißer folgende Ergebnisse erzielt:

durchschnittliche Anzahl gefundener OF-Tupel	89.7725
durchschnittlicher Rotationsfehler um die x-Achse	0,0110°
durchschnittlicher Rotationsfehler um die y-Achse	0,0102°
durchschnittlicher Rotationsfehler um die z-Achse	0,0117°
durchschnittlicher Rotationsfehler um alle Achsen	0,0109°
Anzahl nicht erfolgter Rotationsberechnungen	0 von 2000
Anzahl nicht erfolgter Translationsberechnungen	0 von 200
durchschnittlicher Richtungsfehler	1,7111°

Tabelle 7.1: Simulationsergebnisse unter idealen Verhältnissen, ohne Rauschen, ohne Ausreißer und mit subpixelgenauer Positionsbestimmung.

Bei fünf der 200 Translationsberechnungen war der Fehler des Richtungsvektors größer als 7°. Ein Wert war sogar über 20° - Abbildung 7.1 zeigt die Situation bei dieser Translationsberechnung. Relativ große Fehler in der Rotationsberechnung (knapp 0,1° in x- und z-Richtung), eine sehr kleine Translation (4 cm in x-Richtung, und ungefähr je 2 cm in y- und z-Richtung) und eine zur Bildebene parallele Verschiebung (z-Komponente macht nur ca. 21% der Gesamtbewegung aus) sind die Ursache dafür, dass die Berechnung schlecht konditioniert ist. Die Ausreißer können jedoch gut über den durchschnittlichen Fehler der Translationsschätzung isoliert werden, wie in Abbildung 7.2(b) ersichtlich wird.
Weiters lässt sich erkennen, dass bei der Rotationsberechnung ein größerer Fehler bei der z-Achse auftritt. Dieser Effekt kommt zustande, da die Kamera nur in einer Richtung der

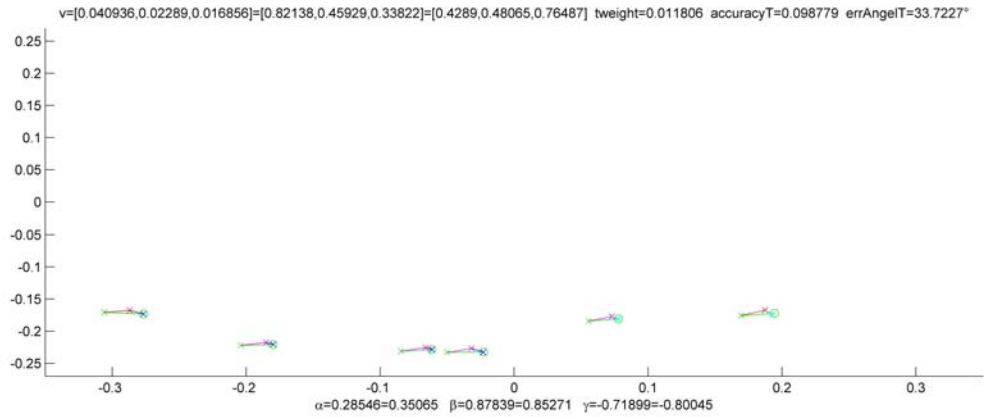
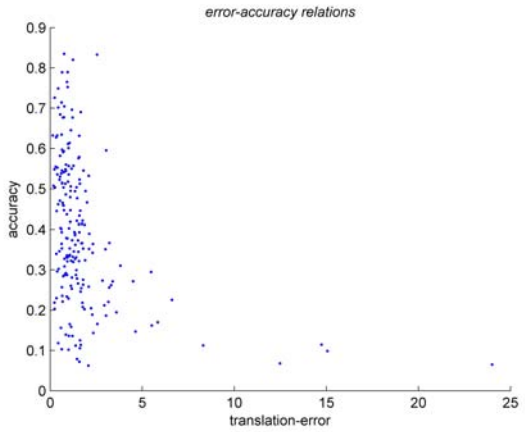
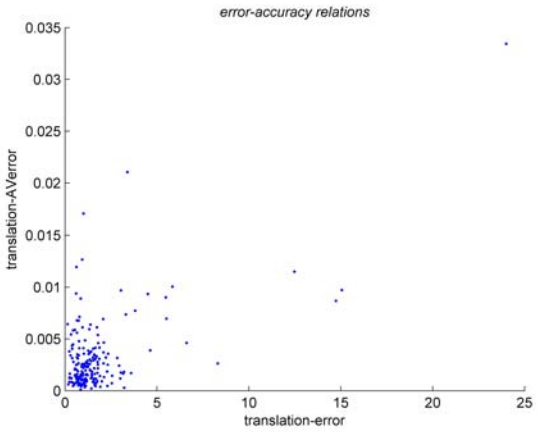


Abbildung 7.1: Grafische Darstellung der Translationsbestimmung, die den Ausreißer in Abbildung 7.2(a) erzeugt. Vor allem eine kleine und zur Bildebene parallele Translation bedingt den relativ großen Fehler in der Translationsschätzung.

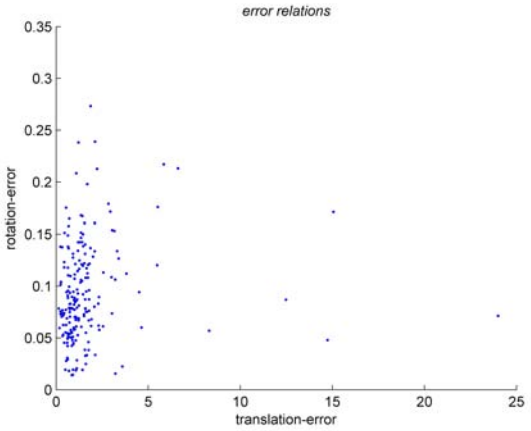
z-Achse Punkte liefert, nämlich in der Blickrichtung der Kamera. Die Drehung um diese Achse ist somit schlecht konditioniert und kann nur mit einer „größeren“ Ungenauigkeit berechnet werden. Tests, in denen auch die anderen Achsen entsprechend konditioniert wurden, indem man Punkte auf nur einer der entsprechenden Halbachsen simulierte, haben ähnliche Fehler wie bei der z-Achse ergeben und folglich die Annahme bestätigt.



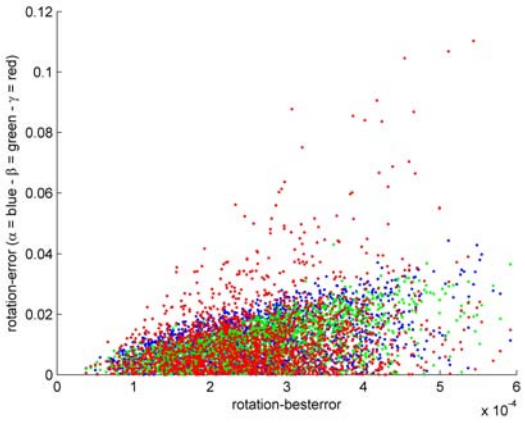
(a) Translationsfehler - Zuverlässigkeit der Translationsschätzung



(b) Translationsfehler - durchschnittlicher Translationsfehler der Berechnung



(c) Translationsfehler - Rotationsfehler



(d) Rotationsfehler - durchschnittlicher Rotationsfehler der Berechnung (x-Achse=blau, y-Achse=grün, z-Achse=rot)

Abbildung 7.2: Simulation idealer Verhältnisse.

2. Geringes Rauschen

Bei subpixelgenauer Merkmalsbestimmung werden bei geringem Rauschen ($\sigma^2 = 0.05$), aber ohne Ausreißer folgende Ergebnisse erzielt:

durchschnittliche Anzahl gefundener OF-Tupel	89.7725
durchschnittlicher Rotationsfehler um die x-Achse	0,0113°
durchschnittlicher Rotationsfehler um die y-Achse	0,0107°
durchschnittlicher Rotationsfehler um die z-Achse	0,0231°
durchschnittlicher Rotationsfehler um alle Achsen	0,0150°
Anzahl nicht erfolgter Rotationsberechnungen	0 von 2000
Anzahl nicht erfolgter Translationsberechnungen	0 von 200
durchschnittlicher Richtungsfehler	2,5888°

Tabelle 7.2: Simulationsergebnisse mit $\sigma^2 = 0.05$ Rauschen.

Die eine Translation, die nicht geschätzt werden konnte, wird in Abbildung 7.3 dargestellt.

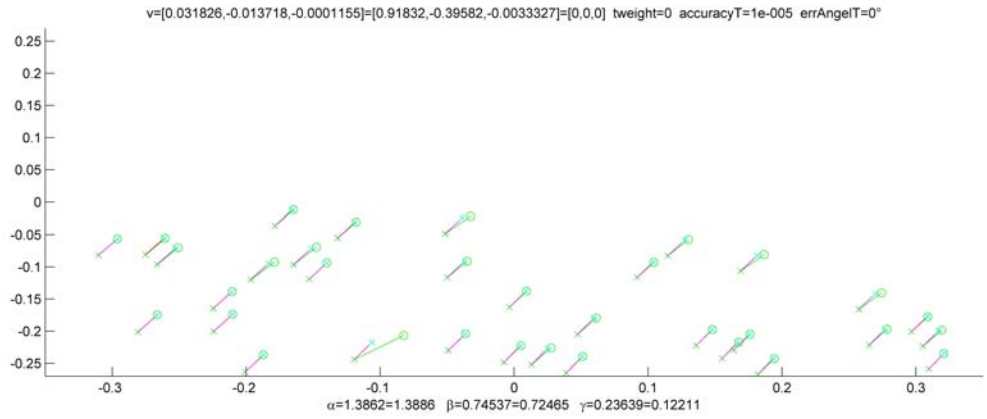
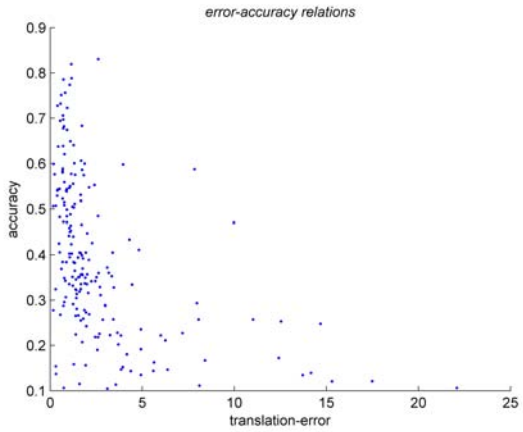
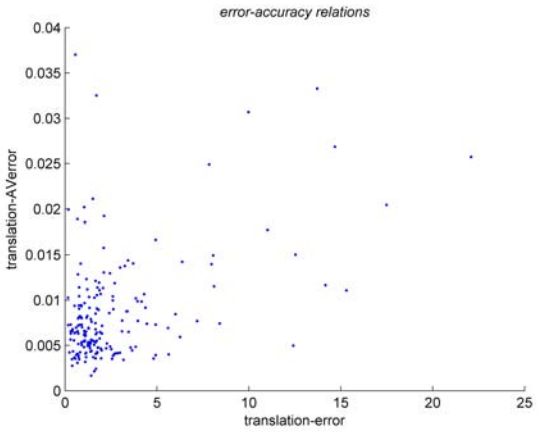


Abbildung 7.3: Grafische Darstellung der Translationsbestimmung, die nicht geschätzt werden konnte (vgl. Tabelle 7.2).

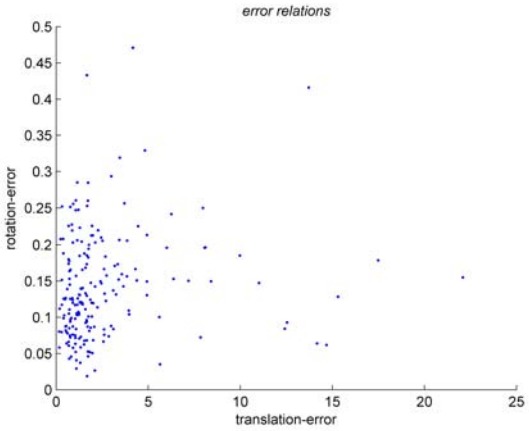
Ähnlich wie in Abbildung 7.1 ist auch hier eine zur Bildebene parallele (z-Komponente der Translation beträgt 0,25%) und vor allem kleine Translation (ca. 4 cm) für das Fehlschlagen der Berechnung verantwortlich. Um solche kurze Bewegungen erkennen zu können, wird auch stets die Länge des längsten Translations-Inliers im Ergebnis zurückgegeben.



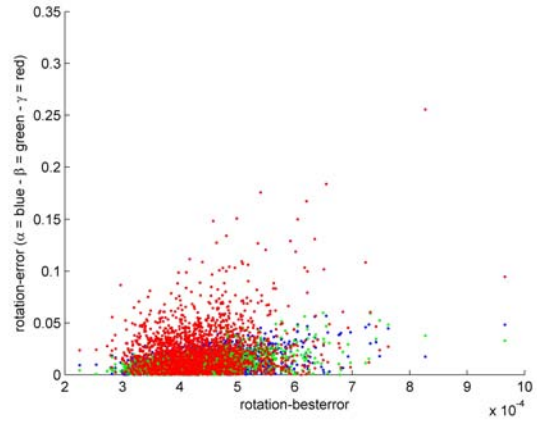
(a) Translationsfehler - Zuverlässigkeit der Translationsschätzung



(b) Translationsfehler - durchschnittlicher Translationsfehler der Berechnung



(c) Translationsfehler - Rotationsfehler



(d) Rotationsfehler - durchschnittlicher Rotationsfehler der Berechnung (x-Achse=blau, y-Achse=grün, z-Achse=rot)

Abbildung 7.4: Simulation mit geringem Rauschen ($\sigma^2 = 0.05$).

3. Stärkeres Rauschen

Bei subpixelgenauer Merkmalsbestimmung, stärkerem Rauschen ($\sigma^2 = 0.1$) und ohne Ausreißer werden folgende Ergebnisse erzielt:

durchschnittliche Anzahl gefundener OF-Tupel	89.7725
durchschnittlicher Rotationsfehler um die x-Achse	0,0131°
durchschnittlicher Rotationsfehler um die y-Achse	0,0123°
durchschnittlicher Rotationsfehler um die z-Achse	0,0343°
durchschnittlicher Rotationsfehler um alle Achsen	0,0199°
Anzahl nicht erfolgter Rotationsberechnungen	1 von 2000
Anzahl nicht erfolgter Translationsberechnungen	2 von 200
durchschnittlicher Richtungsfehler	3,2307°

Tabelle 7.3: Simulationsergebnisse mit $\sigma^2 = 0.1$ Rauschen.

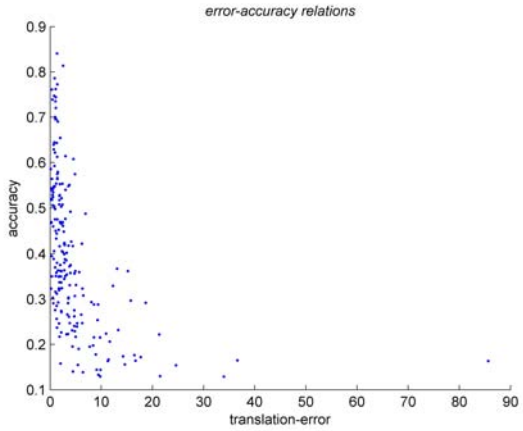
Wird das weiße Rauschen auf $\sigma^2 = 0.2$ erhöht, erhält man folgende Resultate:

durchschnittliche Anzahl gefundener OF-Tupel	89.7725
durchschnittlicher Rotationsfehler um die x-Achse	0,0173°
durchschnittlicher Rotationsfehler um die y-Achse	0,0158°
durchschnittlicher Rotationsfehler um die z-Achse	0,0555°
durchschnittlicher Rotationsfehler um alle Achsen	0,0295°
Anzahl nicht erfolgter Rotationsberechnungen	1 von 2000
Anzahl nicht erfolgter Translationsberechnungen	1 von 200
durchschnittlicher Richtungsfehler	4,7389°

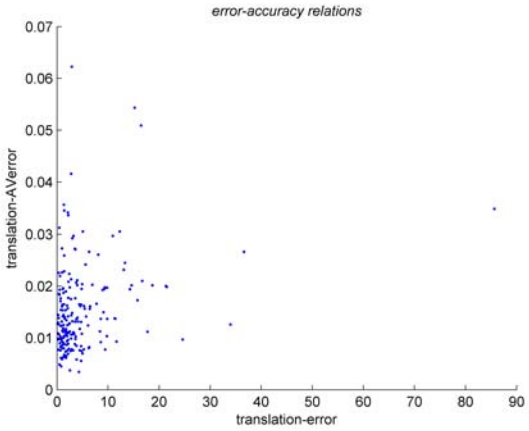
Tabelle 7.4: Simulationsergebnisse mit $\sigma^2 = 0.2$ Rauschen.

Die Tabellen 7.3 und 7.4 lassen erkennen, dass es sich beim Ausreißer der Translationsberechnung mit einem Fehler von ca. 85° um eine Fehlschätzung handelt, die erst durch das größere Rauschen zustandegekommen ist. Beim $\sigma^2 = 0.1$-Rauschen konnten bereits zwei Translationen nicht geschätzt werden, während beim $\sigma^2 = 0.2$ -Testdurchlauf nur mehr eine Translation nicht berechnet werden konnte. Das bedeutet, dass das erhöhte Rauschen die Translationsberechnung in diesem einen Fall erst ermöglicht hat - die Schätzung ist jedoch mit großer Wahrscheinlichkeit sehr ungenau. Ansonsten sind bis auf das obere 3%-Quantil alle Fehler der Translationsberechnung unter 20°; 90% der Fehler sind sogar kleiner als 10°.

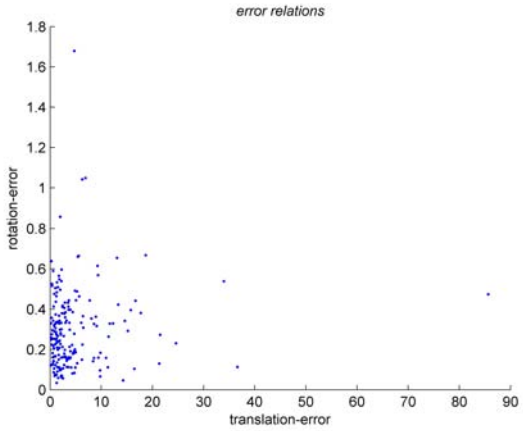
Der gemessene als auch der effektive Fehler bei der Rotationsschätzung hat im Vergleich zu den Testdurchläufen mit geringem Rauschen zugenommen.



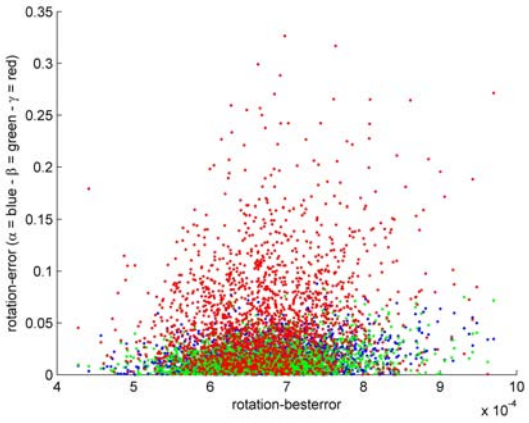
(a) Translationsfehler - Zuverlässigkeit der Translationsschätzung



(b) Translationsfehler - durchschnittlicher Translationsfehler der Berechnung



(c) Translationsfehler - Rotationsfehler



(d) Rotationsfehler - durchschnittlicher Rotationsfehler der Berechnung (x-Achse=blau, y-Achse=grün, z-Achse=rot)

Abbildung 7.5: Simulation mit stärkerem Rauschen ($\sigma^2 = 0.2$).

3. Ausreißer

Ohne Rauschen, jedoch mit 5% Ausreißer und bei subpixelgenauer Merkmalsbestimmung werden bei der selben Parametrisierung wie bei den ersten beiden Testläufen folgende Ergebnisse erzielt:

durchschnittliche Anzahl gefundener OF-Tupel	94.6170
durchschnittlicher Rotationsfehler um die x-Achse	0,0112°
durchschnittlicher Rotationsfehler um die y-Achse	0,0104°
durchschnittlicher Rotationsfehler um die z-Achse	0,0116°
durchschnittlicher Rotationsfehler um alle Achsen	0,0111°
Anzahl nicht erfolgter Rotationsberechnungen	0 von 2000
Anzahl nicht erfolgter Translationsberechnungen	137 von 200
durchschnittlicher Richtungsfehler	4,1898°

Tabelle 7.5: Simulationsergebnisse mit 5% Ausreißer und herkömmlichen Parametern.

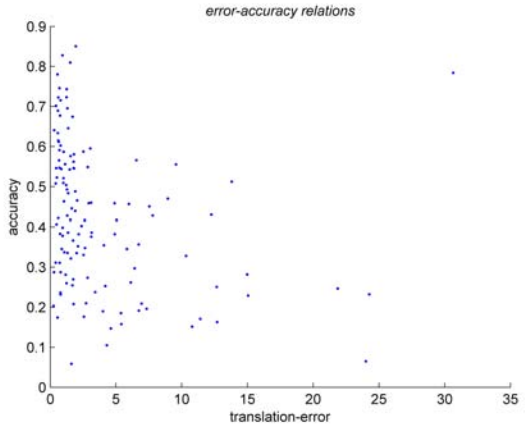
137 von 200 Translationsberechnungen konnten nicht erfolgreich durchgeführt werden. Die Ausreißer bewirken, dass keine passenden Paare für die Translationsschätzung gefunden werden. Durch Verringern des notwendigen Inlieranteils bei der Translationsberechnung erhält man zwar eine weit größere Erfolgsrate, doch sind es beinahe ausschließlich Fehlschätzungen, die gewonnen werden. Verdoppelt man jedoch die Anzahl der Iterationen des Translations-RANSAC-Algorithmus von 20 auf 40, erhält man folgendes Ergebnis:

durchschnittliche Anzahl gefundener OF-Tupel	72.9780
durchschnittlicher Rotationsfehler um die x-Achse	0.0089°
durchschnittlicher Rotationsfehler um die y-Achse	0.0096°
durchschnittlicher Rotationsfehler um die z-Achse	0,0110°
durchschnittlicher Rotationsfehler um alle Achsen	0,0098°
Anzahl nicht erfolgter Rotationsberechnungen	2 von 2000
Anzahl nicht erfolgter Translationsberechnungen	10 von 201
durchschnittlicher Richtungsfehler	2,6255°

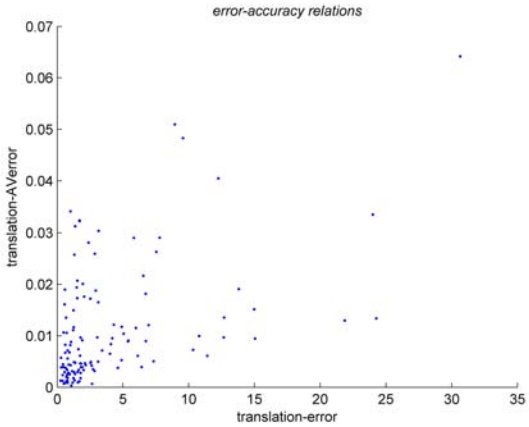
Tabelle 7.6: Simulationsergebnisse mit 5% Ausreißer und angepassten Parametern.

Während die Rotationsschätzung durch die Ausreißer nicht beeinflusst wird, müssen je nach Ausreißer-Prozentsatz die Translations-Parameter angepasst werden. Durch den Ausreißeranteil bei den Rotations-Outlier sinkt der Anteil der Translations-Inlier in dieser Menge. Das muss beim Parameter $minClosedPercT$ berücksichtigt werden. Da Ausreißer

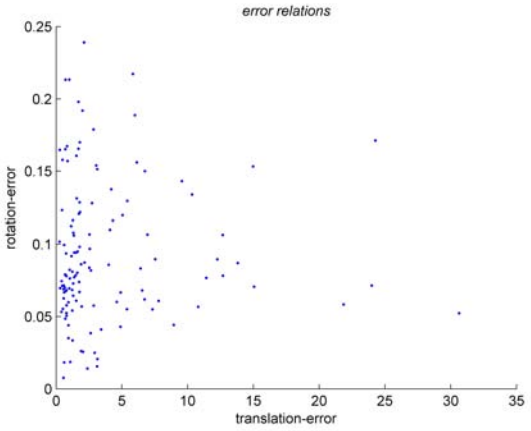
oft sehr lange Vektoren sind und somit beim RANSAC-Algorithmus als *maybeinlier* bevorzugt werden, sollte bei einem schlechten Tracker die Anzahl der Iterationen (und somit $maxIterT$) erhöht werden.



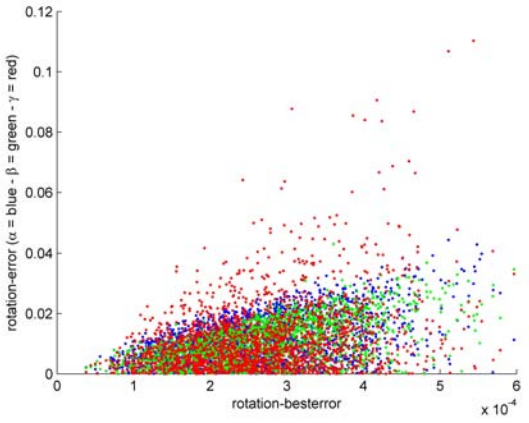
(a) Translationsfehler - Zuverlässigkeit der Translationsschätzung



(b) Translationsfehler - durchschnittlicher Translationsfehler der Berechnung



(c) Translationsfehler - Rotationsfehler



(d) Rotationsfehler - durchschnittlicher Rotationsfehler der Berechnung (x-Achse=blau, y-Achse=grün, z-Achse=rot)

Abbildung 7.6: Simulation mit 5% Ausreißer.

Durch Ausreißer, die der effektiven Translation gut entsprechen, wird die Translationsbestimmung verfälscht. In der Simulation wurde daher ein etwas schlechteres Ergebnis als

beim idealen Testlauf erzielt. Vor allem die schlechteste Schätzung der Translationsberechnung in Abbildung 7.6(a) mit hoher Zuverlässigkeit ist die Folge von Ausreißern, die zufällig so angeordnet sind, dass sie selbst eine Translation beschreiben.

4. Diskretisierung

Ohne Rauschen , ohne Ausreißer, doch bei Diskretisierung der Merkmalspositionen auf die Pixelgröße werden bei der selben Parametrisierung wie bei den ersten beiden Testläufen folgende Ergebnisse erzielt:

durchschnittliche Anzahl gefundener OF-Tupel	86.9535
durchschnittlicher Rotationsfehler um die x-Achse	0,0124°
durchschnittlicher Rotationsfehler um die y-Achse	0,0119°
durchschnittlicher Rotationsfehler um die z-Achse	0,0314°
durchschnittlicher Rotationsfehler um alle Achsen	0,0186°
Anzahl nicht erfolgter Rotationsberechnungen	43 von 2000
Anzahl nicht erfolgter Translationsberechnungen	30 von 236
durchschnittlicher Richtungsfehler	2,8753°

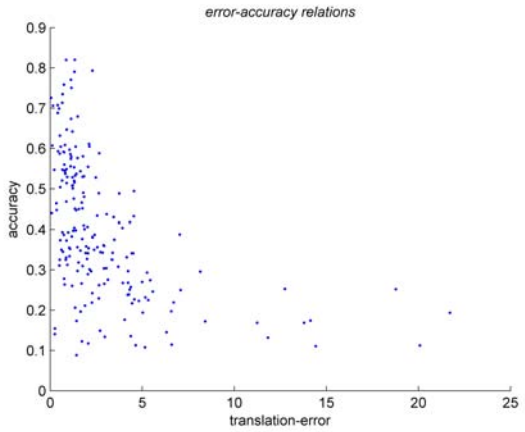
Tabelle 7.7: Simulationsergebnisse bei Pixel-Diskretisierung mit bisherigen Parametern.

Viele Rotationsberechnungen konnten nicht erfolgreich durchgeführt werden, da die Toleranzen für die Rotationsbestimmung, die durch den Parameter *threshFitR* bestimmt werden, zu klein sind. Wird diese Toleranz von 2 auf 3 Pixellängen vergrößert, führt das zu folgendem Ergebnis:

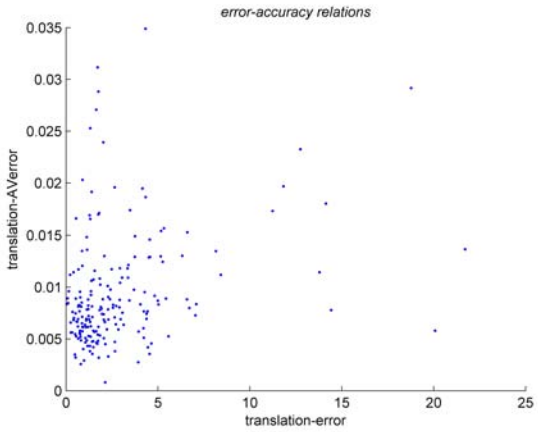
durchschnittliche Anzahl gefundener OF-Tupel	89,7725
durchschnittlicher Rotationsfehler um die x-Achse	0,0145°
durchschnittlicher Rotationsfehler um die y-Achse	0,0132°
durchschnittlicher Rotationsfehler um die z-Achse	0,0286°
durchschnittlicher Rotationsfehler um alle Achsen	0,0188°
Anzahl nicht erfolgter Rotationsberechnungen	0 von 2000
Anzahl nicht erfolgter Translationsberechnungen	2 von 200
durchschnittlicher Richtungsfehler	2,8246°

Tabelle 7.8: Simulationsergebnisse bei Pixel-Diskretisierung mit verbesserten Parametern.

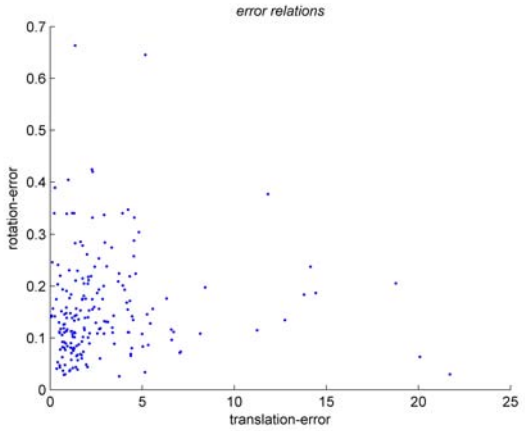
Die Diskretisierung bewirkt im Grunde nur, dass die Messdaten ungenauer werden. Als natürliche Konsequenz werden auch unsere Messergebnisse ungenauer und um weiterhin Schätzungen zu erhalten müssen die Toleranzen gelockert werden, vor allem der Parameter *threshFitR* muss entsprechend vergrößert werden.



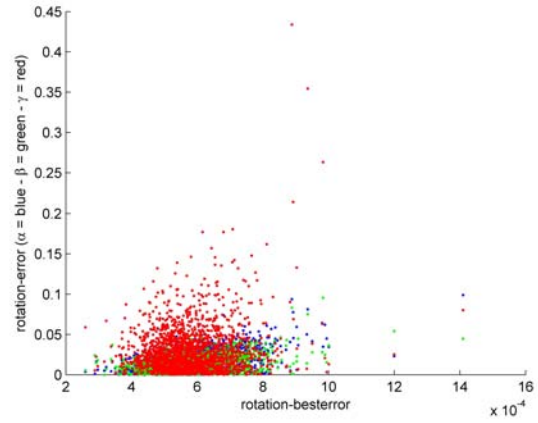
(a) Translationsfehler - Zuverlässigkeit der Translationsschätzung



(b) Translationsfehler - durchschnittlicher Translationsfehler der Berechnung



(c) Translationsfehler - Rotationsfehler



(d) Rotationsfehler - durchschnittlicher Rotationsfehler der Berechnung (x-Achse=blau, y-Achse=grün, z-Achse=rot)

Abbildung 7.7: Simulation mit Pixel-Diskretisierung.

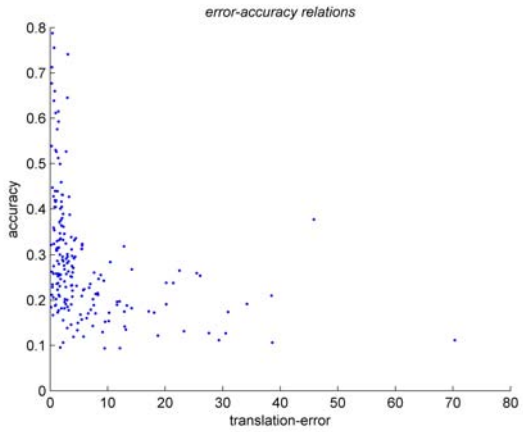
5. Schlechter Fall

In diesem Testlauf wurde geringes Rauschen ($\sigma^2 = 0.05$), 2% Ausreißer gemeinsam mit der Diskretisierung der Merkmalspositionen simuliert. Um das Beispiel realistischer zu machen und eine bessere Konditionierung für die Translationsberechnung zu erhalten, wurden beim Zufallsgenerator die Wahl für eine Vorwärtsbewegung um 25% bevorzugt, sodass größere OF zustande kommen und sich nicht so einfach zu Null summieren. Da

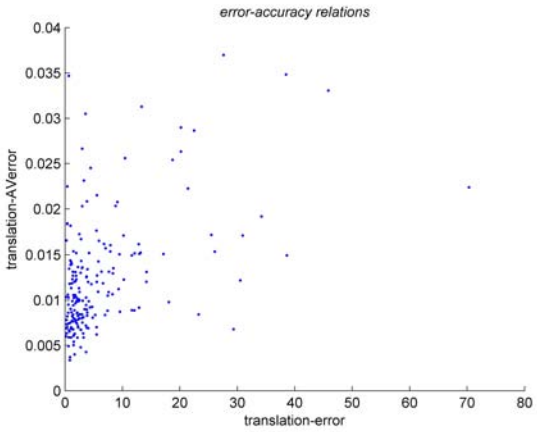
nun mit einer Vorwärtsbewegung zu rechnen ist, wurde der Grenzwinkel für die Rotationsberechnung von 65° auf 85° angehoben. Folgende Ergebnisse wurden erzielt:

durchschnittliche Anzahl gefundener OF-Tupel	97,795
durchschnittlicher Rotationsfehler um die x-Achse	0,0153°
durchschnittlicher Rotationsfehler um die y-Achse	0,0135°
durchschnittlicher Rotationsfehler um die z-Achse	0,0349°
durchschnittlicher Rotationsfehler um alle Achsen	0,0213°
Anzahl nicht erfolgter Rotationsberechnungen	1 von 2000
Anzahl nicht erfolgter Translationsberechnungen	1 von 200
durchschnittlicher Richtungsfehler	6,0969°

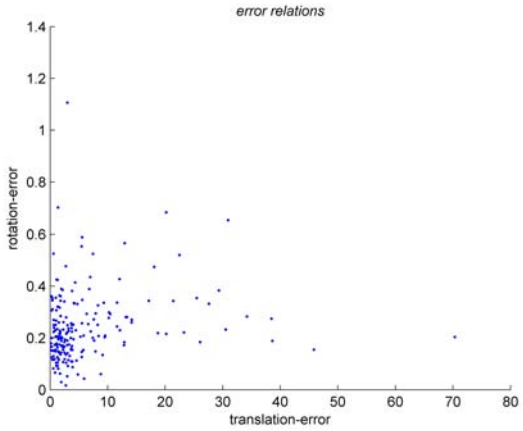
Durch die Parameteranpassung war es dem Algorithmus möglich sowohl Rotation als auch Translation in den meisten Fällen sehr gut zu schätzen. Nur wenige Ausreißer bei der Translationsbestimmung sind zu erkennen.



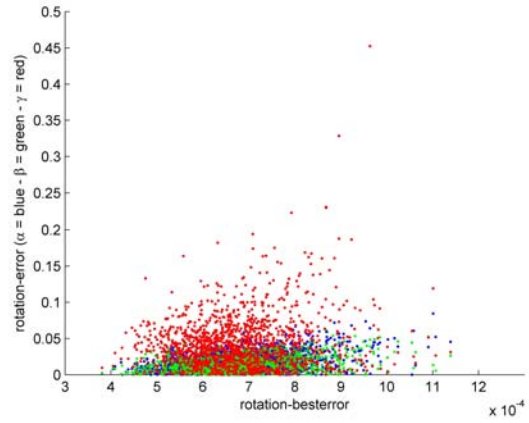
(a) Translationsfehler - Zuverlässigkeit der Translationsschätzung



(b) Translationsfehler - durchschnittlicher Translationsfehler der Berechnung



(c) Translationsfehler - Rotationsfehler



(d) Rotationsfehler - durchschnittlicher Rotationsfehler der Berechnung (x-Achse=blau, y-Achse=grün, z-Achse=rot)

Abbildung 7.8: Simulation mit Rauschen ($\sigma^2 = 0.05$), 2% Ausreißer und Pixel-Diskretisierung.

Ergebnis der Simulation

Die Tests haben gezeigt, dass auch bei der Simulation von Rauschen, Ausreißern und Pixel-Diskretisierung das Verfahren äußerst sinnvolle Schätzungen liefert.

	Simulierte Situationen					
Durchschnittswerte	ideal	wenig R.	starkes R.	Ausreißer	Disktret.	alles
Anzahl OF-Tupel	89,7725	89,7725	89,7725	72,9780	89,7725	97,795
Rot.fehler (x-Achse)	0,0110°	0,0113°	0,0173°	0,0089°	0,0145°	0,0153°
Rot.fehler (y-Achse)	0,0102°	0,0107°	0,0158°	0,0096°	0,0132°	0,0135°
Rot.fehler (z-Achse)	0,0117°	0,0231°	0,0555°	0,0110°	0,0286°	0,0349°
Rot.fehler (alle)	0,0109°	0,0150°	0,0295°	0,0098°	0,0188°	0,0213°
fehlerh. Rot.berech.	0/2000	0/2000	1/2000	2/2000	0/2000	1/2000
fehlerh. Tra.berech.	0/200	1/200	1/200	10/201	2/200	1/200
Richtungsfehler	1,7111°	2,5888°	4,7389°	3,8152°	2,8246°	6,0969°

Tabelle 7.9: Hier sind die Ergebnisse der verschiedenen Genauigkeitstests gegenübergestellt. Durch die Verwendung eines uneingeschränkten Zufallsgenerators können die Translationen der einzelnen Bilder in umgekehrter Richtung wirken und sich so zu kleinen Werten aufsummieren. Trotzdem gelingt es dem Verfahren sehr gute Ergebnisse zu erzielen.

Die Rotation konnte in allen Testläufen fast immer und mit sehr hohen Genauigkeiten (weit unter 0,1°) bestimmt werden. Bei den nicht so guten Schätzungen der Translation liefert der Zuverlässigkeitswert wie gewünscht nur einen kleinen Wert (vgl. (a)-Bilder der Serie).
Bei den Tests wurde bis auf die zwei Änderungen, die angeführt sind keine Parameter verändert. Daraus kann man schließen, dass die passende Parametrisierung des Algorithmus zwar wichtig ist und das Ergebnis deutlich verbessern kann, doch das Verfahren reagiert nicht sensibel auf kleine Änderungen der äußeren Einflüsse bzw. der Parameter.
Weiters haben die Tests gezeigt, dass kein Zusammenhang zwischen Rotationsfehler und Translationsfehler besteht, wie in den (c)-Bildern der jeweiligen Testläufe ersichtlich wird. Auch gibt es nur einen eher schwach korrelierten Zusammenhang zwischen Rotationsgenauigkeit und gemittelten Rotationsfehler, daher ist eine zusätzliche Gewichtung der Zuverlässigkeit der Translationsberechnung mit dem durchschnittlichen Rotationsfehler nicht sinnvoll. Trotzdem eignet sich auch die Rotationsbestimmung gut für den Einsatz mit einem Kalmanfilter, da diese bei passender Parametrisierung nur dann eine Lösung findet, wenn das Ergebnis auch zutreffend ist, das heißt, in den Toleranzen der Parameter

liegt.

7.3.2 Simulation von zu nahen Punkten

In diesem Test wird das Verhalten von zu nahen Punkten überprüft. Es dürfen somit nur wenige bzw. gar keine Punkte der Gleichung 5.1 genügen. Eigentlich ist die minimale Translation ausschlaggebend für z_∞. Doch da auch keine Translation vorkommen kann und somit die minimale Verschiebung Null ist, wird die maximale Translation in x- bzw. y-Richtung als Anhaltspunkt genommen. Für das bislang verwendete Simulationsmodell ergibt sich eine maximale Verschiebung von 0,05 m. Entsprechend Gleichung 5.1 bedeutet das für die simulierte Kamera (normiert auf Brennweite $f = 1$):

$$z_\infty = \frac{1}{0.00094}\ 0,05m\ \approx\ 53m \tag{7.3}$$

Folgender Test wurde bei idealen Verhältnissen (ohne Rauschen, ohne Ausreißer und bei subpixelgenauer Positionsbestimmung) mit ausschließlich nahen Punkten durchgeführt. Um eine größere Fehlertoleranz bei den Ergebnissen der Rotationsbestimmung zu ermöglichen, wurde der entsprechende Grenzwert bei dieser Simulation vergrößert (*threshFitR*=3 Pixel). Auch der Prozentsatz der nötigen *alsoinliers* wurde auf 20 (bzw. bei 5 m auf 10) verringert (*minClosedPercR*). Dies erhöht zwar die Wahrscheinlichkeit von Fehlberechnungen, jedoch ist der Anteil an translationsinvarianten Punkten, bedingt durch die ausschließlich nahen Punkte, nur mehr sehr gering.

	maximale Entfernung in Meter				
Durchschnittswerte	50	20	10	5-a	5-b
Anzahl OF-Tupel	63.833	70.118	84.912	90.609	81.844
Rotationsfehler (x-Achse)	0.049°	0.128°	0.187°	0.291	0.379°
Rotationsfehler (y-Achse)	0.053°	0.124°	0.189°	0.322	0.370°
Rotationsfehler (z-Achse)	0.032°	0.101°	0.113°	0.193	0.340°
Rotationsfehler (alle)	0.045°	0.118°	0.163°	0.269	0.363°
fehlerhafte Rot.berechnungen	1	4	268	1130	60
fehlerhafte Trans.berechnungen	6/200	7/201	145/395	826/1154	65/242
Richtungsfehler	4.432°	10.679°	8.174°	10.788	22.336°

Tabelle 7.10: Hier sind die Ergebnisse der verschiedenen maximalen Entfernungen gegenübergestellt. Der Prozentanteil der translationsinvarianten Punkte wurde von 30% auf 20% und bei 5b nochmals auf 10% verringert. Auch hier wurden wieder je Simulationslauf 2000 Bilder verwendet.

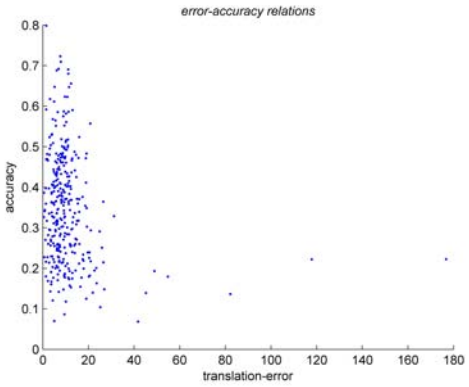
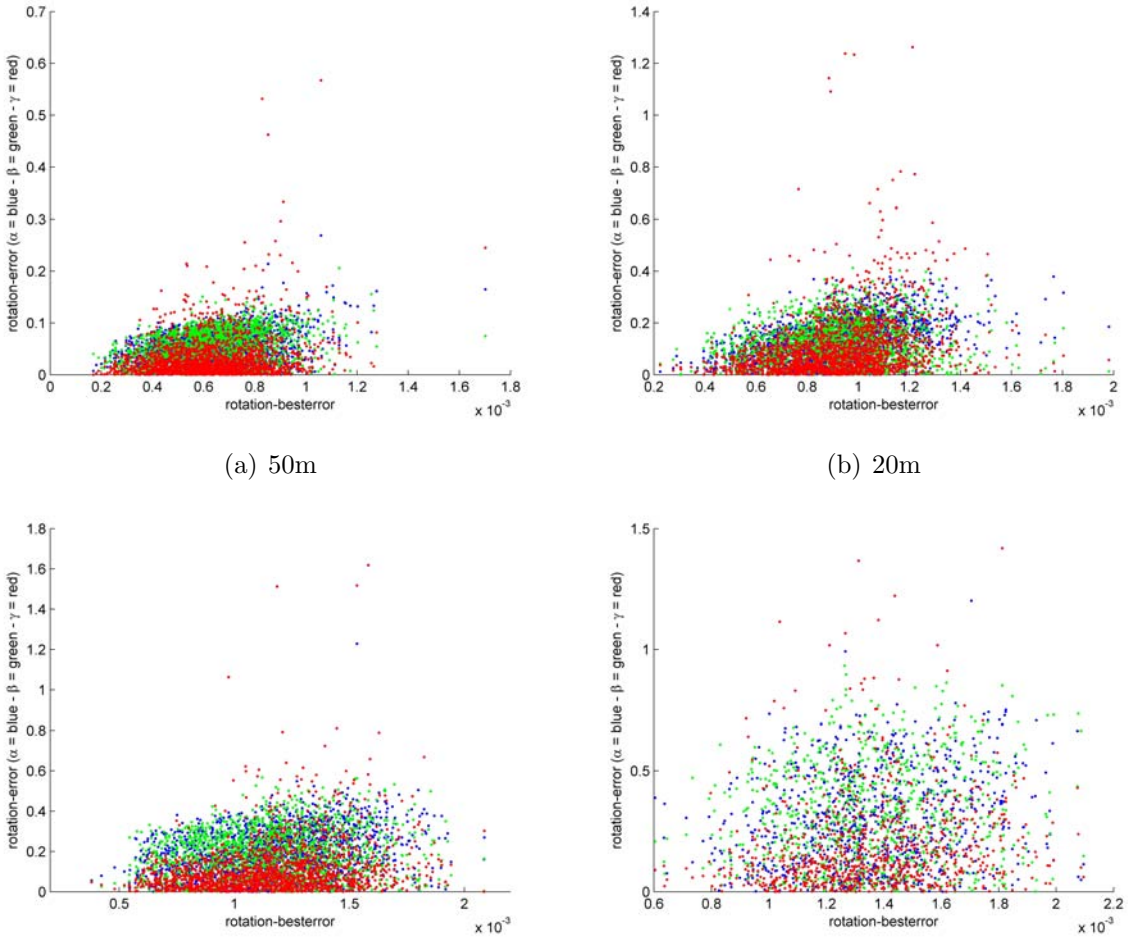


Abbildung 7.9: Ergebnis der Translationsberechnung der Simulation mit Punkten bis zu 5 m Entfernung und $minClosedPercR$=20% (Test 5-a).

Trotz der hohen Fehlerrate bei der Rotationsbestimmung im Testfall 5-a in Tabelle 7.10, konnte bei den verbleibenden Bildern eine relativ gute Translationsschätzung gemacht werden (vgl. Abbildung 7.9). Wie in Abschnitt 5.3 beschrieben, wird bei einem Rotationsfehler die Translation, der bis zu diesem Bild aufsummierten Sequenz, berechnet. So geht keine Information über die relative Bewegung verloren.



(a) 50m

(b) 20m

(c) 10m

(d) 5m -a

Abbildung 7.10: Darstellung des Rotationsfehlers bei der Simulation des Algorithmus in geschlossenen Räumen (bei ausschließlich nahen Punkten). Die Längenangaben stehen für die maximale Entfernung der Punkte in der Simulation.

7.3.3 Geschwindigkeit

Die Geschwindigkeit des Algorithmus wurde auf einem Intel Core Duo Prozessor T2300 gemessen (1,66GHz, 667MHz FSB, 2MB L2-Cache) mit einem Gigabyte RAM. Der Code ist nicht auf ein Multi-Prozessor-System optimiert, sodass das Programm nur auf einem der beiden Prozessoren ausgeführt wird.

Zum Testen wurde „gprof“[1] verwendet, eine GNU Bibliothek zum Profilieren von Programmen. Mit diesem Werkzeug können unter anderem die Ausführungszeiten der einzelnen Funktionen gemessen werden.

Die Parametrisierung entspricht jener bei den Genauigkeitstests. Somit wurden folgende laufzeitrelevanten Parameter verwendet:

- maxIterR=40
- maxIterT=20
- maxStepsT=5

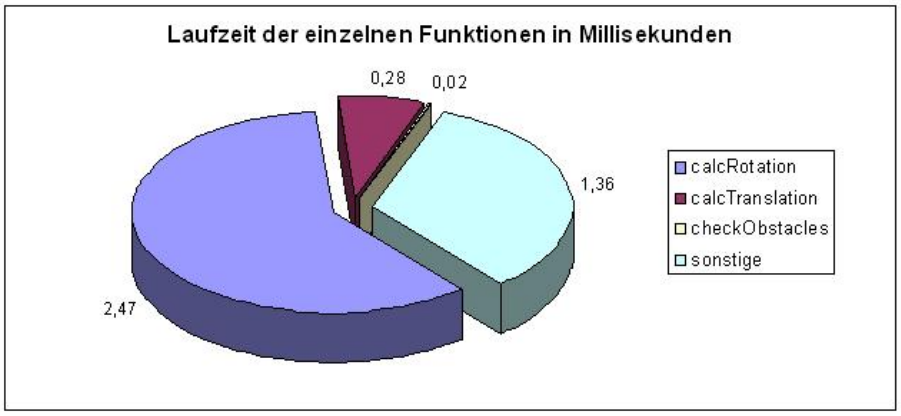


Abbildung 7.11: Der Laufzeittest der einzelnen Funktionen zeigt klar, dass das Verfahren echtzeitfähig ist. Mit einer durchschnittlichen Gesamtzeit von 4,13 Millisekunden auf einem 1,6 GHz Rechner liegt die Laufzeit weit unter den 33,3 ms, die ein echtzeitfähiger Algorithmus bei einer Bildrate von 30 Bildern die Sekunde benötigen darf. Die laufzeitrelevanten RANSAC-Parameter sind: maxIterR=40, maxIterT=20, maxStepsT=5. Unter den Anteil „sonstige“ fallen die Kontroll- und Steuerroutinen des Verfahrens.

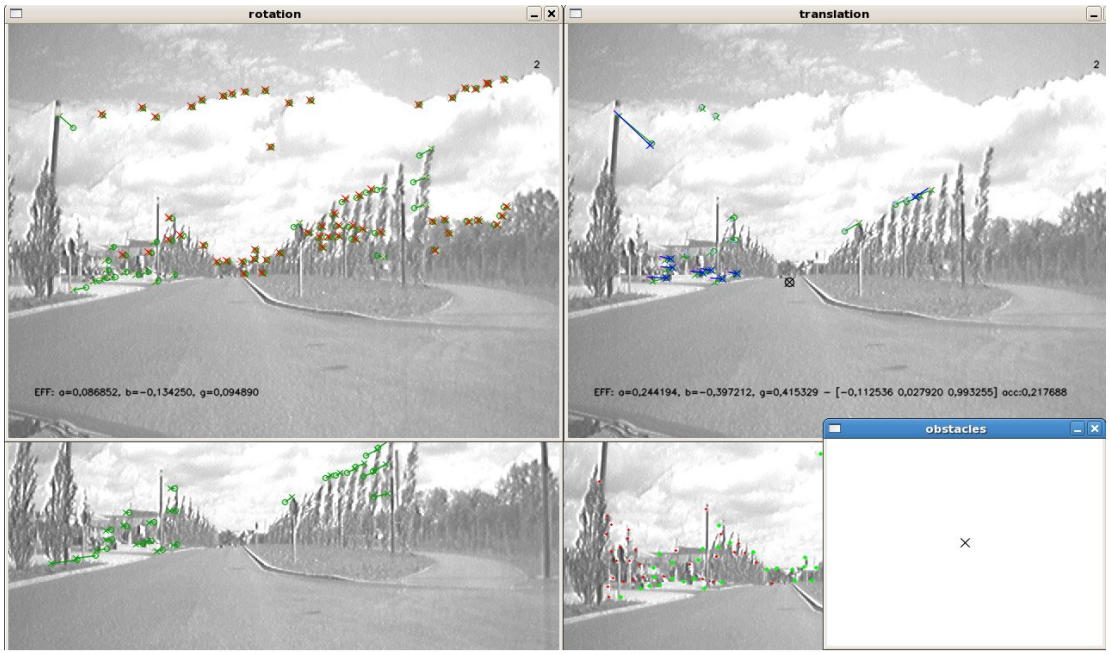
Das Ergebnis wird in der Abbildung 7.11 dargestellt. Es wird ersichtlich, dass die Rotationsbestimmung einen wesentlichen Anteil der Laufzeit einnimmt. Dies lässt sich damit begründen, dass für die Singulärwert-Zerlegung der Rotationsbestimmung die SVD-Funktion der OpenCV-Bibliothek[2] verwendet wird, mit dem Vorteil, dass sie eine hohe Genauig-

[1] http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html
[2] http://sourceforge.net/projects/opencv/

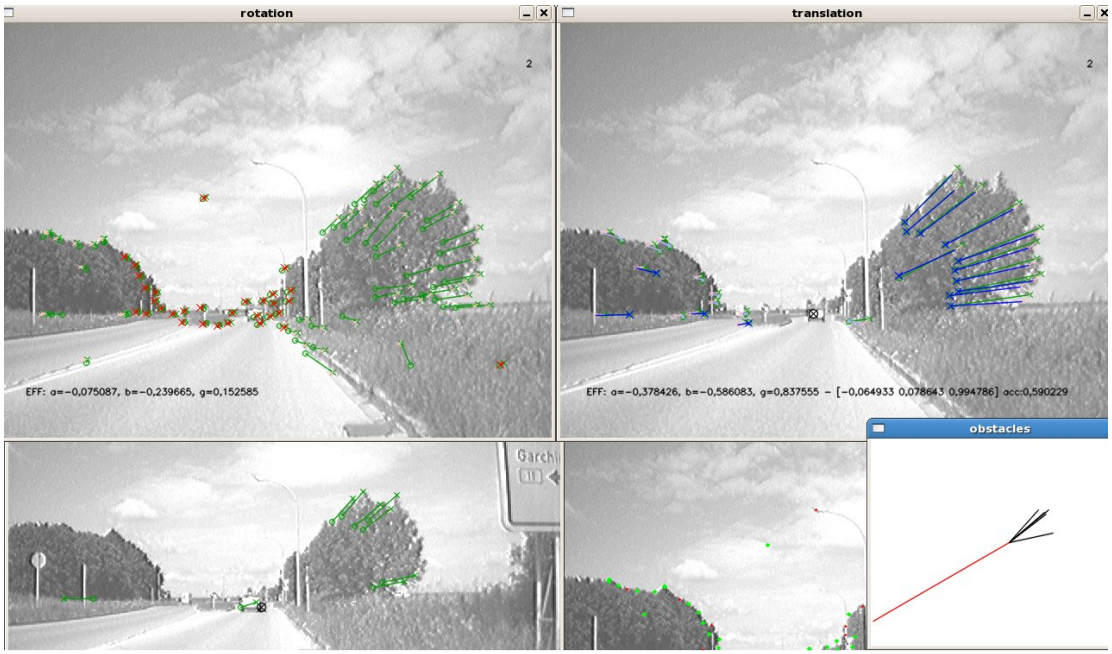
keit aufweist, was für eine genaue Rotationsberechnung von großer Bedeutung ist. Diese Genauigkeit hat jedoch eine lange Rechenzeit zum Preis.

7.4 Test an realen Bildern

Für die Tests an realen Bildern wurden mehrere Testfahrten durchgeführt: Eine Fahrt mit einem kurzfristig umdisponierten Speisewagen durch das MI-Gebäude der TU München, zwei weitere in einem Auto durch die Kleinstadt Garching bei München bzw. durch das Forschungszentrum in Garching. Folgende Bilder sind Auszüge aus den beiden letzteren Bildreihen - weitere Auszüge sind in Appendix D aufgeführt.



(a) Autofahrt im Forschungsgelände Garching



(b) Autofahrt zur Stadt Garching

Abbildung 7.12: Dargestellt sind zwei Screenshots der Ergebnis-Visualisierung der beiden Testfahrten (siehe oben). Die Bedeutung der Visualisierung ist in Abschnitt 7.2 beschrieben.

Kapitel 8

Weiterführende Arbeiten

Obwohl der Algorithmus sehr gute Ergebnisse erzielt, gibt es natürlich immer noch Verbesserungsmöglichkeiten. Einige davon sind:

- Aruns Algorithmus zur Bestimmung der Rotation wurde in [RV97] nochmals verbessert, dadurch dass eine Produkt-Singulärwert-Zerlegung (PSVD) verwendet wird und das Verfahren in ein gemischtes least-square und total-least-square (LS-TLS) Problem gegliedert wird. Diese Variante könnte zur Verbesserung der Rotationsberechnung implementiert werden.

- Um auch in engen Räumen eine Rotationserkennung zu ermöglichen, könnte die Auflösung des Bildes verringert werden. Somit wäre die Entfernung für translationsinvariante Punkte künstlich verkürzt. Praktisch kann das so umgesetzt werden, dass bei einer nicht erfolgten Rotationserkennung über mehrere Stufen, wie bei einer Bildpyramide, die Auflösung verringert wird. So ist es möglich auch bei sehr nahen Punkten noch eine Rotationsschätzung zu machen. Das Ergebnis wäre dann zwar etwas ungenauer, aber für einen Kalmanfilter immer noch sinnvoll zur Stabilisierung der Prädiktion.

- Analog dazu könnte im umgekehrten Fall, bei vielen sehr weit entfernten Punkten, eine subpixelgenaue Merkmalsextraktion erfolgen und diese, nur im Fall dass keine Rotation erkannt wird, schrittweise auf kleinere Genauigkeiten gerundet werden.

Das Verfahren soll demnächts bei der Rekonstruktion eines Wespenflugs Anwendung finden. Darüber hinaus soll überprüft werden, wie genau und zuverlässig die Epipolargeometrie aus den Ergebnissen berechnet werden kann.
Natürlich wird auch weiterhin das ursprüngliche Ziel verfolgt, nämlich den Algorithmus beim Quadrokopter und Zeppelin des DLR einzusetzen (siehe Abschnitt 3.1).

Kapitel 9

Zusammenfassung

Im Rahmen dieser Masterarbeit wurde ein Algorithmus entwickelt, der die relative Bewegung der Kamera schätzt und Hindernisse in der Umgebung entdeckt. Die hohe Genauigkeit und Zuverlässigkeit dieser Methode wurde in einer ausführlichen Test- und Simulationsreihe erfolgreich nachgewiesen.

Ein wesentlicher Bestandteil des Verfahrens ist der RANSAC-Algorithmus. Wie jeder iterative Algorithmus hat auch RANSAC einen negativen Beigeschmack, da iterative Methoden meist eine lange, nicht deterministische Rechenzeit in Anspruch nehmen und oft ungenaue Resultate liefern. Diese Vorurteile treffen hier jedoch aus folgenden Punkten nicht zu: Das Verfahren ist echtzeitfähig, wie der Test in Abschnitt 7.3.3 bewiesen hat. Die Iterationen im Algorithmus können als Parameter $maxIterR$ und $maxIterT$ bzw. $maxStepsT$ angegeben werden und daraus lässt sich eine maximale Rechenzeit berechnen. Wird $maxStepsT$=1 verwendet, so ist die Anzahl der Iterationen sogar deterministisch. Da der Rechenaufwand mit 4,13 Millisekunden auf einem 1,6 GHz Prozessor sehr gering ist, können ohne weiteres auch höhere Iterationszahlen verwendet werden. Für die Rotationsschätzung lässt sich jedoch sehr gut berechnen, wieviele Schritte nötig sind, um die Rotation mit ausreichend hoher Wahrscheinlichkeit zu berechnen. Somit ist der Einsatz dieser iterativen Methode ohne Bedenken über Einschränkungen zu vertreten.
Die hohe Parameteranzahl des Verfahrens erfordert ein gutes Verständnis des Algorithmus, um bei einem unerwünschten Verhalten den oder die entsprechenden Werte anzupassen. Hat man jedoch das Wissen um die Funktionsweise des Programms, so erweist sich die

zunächst so nachteilig wirkende Anhäufung an Parametern als effizientes Werkzeug, um die perfekten Einstellung an die aktuelle Situation zu ermöglichen. Im Grunde sind die Parameter sehr intuitiv, was das Einlernen und die Handhabung wesentlich erleichtert.
Der Algorithmus besteht aus drei Einzelteilen, die jeweils eine Aufgabe lösen. Die Methoden können so den entsprechenden Bedürfnissen angepasst werden und müssen nicht aufeinander abgestimmt sein. Dieser bedeutende Vorteil ermöglicht den Einsatz von äußerst effizienten Teil-Algorithmen, die unabhängig voneinander Translation und Rotation schätzen und Hindernisse erkennen. Hier spiegelt sich gleichzeitig der entscheidende Unterschied zu den Konkurrenzalgorithmen wieder. Z.B. schätzt der 8-Punkt-Algorithmus Rotation und Translation gemeinsam, was bei schlechter Konditionierung einer Komponente zum Scheitern nicht nur dieses einen Teils, sondern der gesamte Berechnung führt.

Die ursprüngliche Aufgabenstellung der Masterarbeit konnte somit erfolgreich gelöst werden und dem Einsatz dieses Verfahrens in realen Anwendungen steht nichts mehr im Wege.

Anhang A

Der generische RANSAC-Code

Der hier dargestellte generische Code ist einer freien Internet-Enzyklopädie[1] entnommen, entspricht der Beschreibung in [FB81] und ist selbsterklärend:

Code-Ausschnitt A.1: Der generische RANSAC-Code

```
1 Eingabe:
2   data - Datensatz (die beobachteten Punkte)
3   model - ein Modell, das dem Datensatz zugrunde liegt
4   n - die minimale Anzahl an Punkten, die für die Modellberechnung
      notwendig sind
5   k - Anzahl der Iterationen, in denen neue Modelle berechnet werden
6   t - Fehler-Grenzwert, wann ein Punkt dem Modell entspricht
7   d - minimale Anzahl der Punkte, die einem Modell entsprechen
      müssen, damit es angenommen wird
8 Ausgabe:
9   bestfit - Modell-Parameter, die dem besten Modell entsprechen,
      oder nil, falls kein passendes Modell gefunden werden konnte
10
11 iterations := 0
12 bestfit := nil
13 besterr := unendlich
```

[1] www.wikipedia.org, am 10. Mai 2007

```
14 solange iterations < k
15     maybeinliers := n zufällig gewählte Punkte aus dem Datensatz (
        data)
16     maybemodel := Modellparameter, die aus den maybeinliers
        berechnet wurden
17     alsoinliers := leerer Datensatz
18
19     für jeden Punkt in data und nicht in maybeinliers
20         wenn der Punkt dem maybemodel bis auf einen Fehler kleiner
            als t entspricht
21             füge den Punkt den alsoinliers hinzu
22
23     wenn die Anzahl der Elemente in alsoinliers > d /*das bedeutet,
        dass ein gutes Modell für den Datensatz gefunden wurde und
        getestet werden muss wie gut dieses Modell ist*/
24         bettermodel := Modell Parameter, die aufgrund aller Punkte
            in maybeinliers und alsoinliers errechnet wurden
25         thiserr := ein Maß, wie gut das Modell diesen Punkten
            entspricht
26         wenn thiserr < besterr
27             bestfit := bettermodel
28             besterr := thiserr
29
30     inkrementiere iterations
31
32 gib bestfit zurück
```

Anhang B

KLT Tracker

Der „***K**anade-**L**ucas-**T**omasi Feature Tracker*“ ist das Ergebnis jahrelanger Forschung der Erfinder Takeo Kanade, Bruce D. Lucas und Carlo Tomasi. Ziel der Forschung war es, einen Algorithmus zu entwickeln, der möglichst robuste Merkmale in einem Bild findet. Robust bedeutet dabei, dass sich diese Merkmale über entsprechende Kriterien sehr gut vom Rest des Bildes unterscheiden lassen. Zudem soll der Algorithmus möglichst zuverlässig und schnell diese Merkmale in einem zweiten Bild wiederfinden.
Etwas genauer: Gute Merkmale werden anhand des kleinsten Eigenwertes jeder 2x2 Gradientenmatrix ausgewählt. Diese Punkte werden anschließend mittels einer Newton-Raphson Methode, welche die Differenzen zwischen den beiden Fenstern minimiert, über weitere Bilder verfolgt. Dabei werden Bildpyramiden verwendet, um auch größere Verschiebungen der Punkte schnell zu finden.
Dieser Tracker erfreut sich großer Beliebtheit in der gesamten Computer-Vision-Community, doch würde es den Rahmen dieser Arbeit sprengen im Detail darauf einzugehen. In diesem Sinne wird hier auf [LK81, TK91, ST94, Bir96] verwiesen.

Der KLT-Tracker wurde im Rahmen der Masterarbeit zur Erzeugung der OF-Tupel und somit als eine der Datenquellen für den Algorithmus genutzt.

(a) Bild 17. 111 von 150 Punkte konnten vom letzten Bild übernommen werden.

(b) Bild 18. 112 von 150 Punkte konnten vom letzten Bild übernommen werden.

Abbildung B.1: Ergebnis des KLT-Trackers auf eine Bildsequenz, gemacht aus dem MI-Gebäude im Forschungszentrum Garching der TU München. Grün sind alle Merkmale dargestellt, die vom vorangegangenen Bild verfolgt werden konnten. Rot bedeutet, dass die Punkte neu hinzugenommen wurden. In jedem Bild wurden 150 Merkmale extrahiert.

Anhang C

Klassendiagramm

Abbildung C.1 stellt ein grobes Klassendiagramm des C++-Programms mit den Schlüsselfunktionen und Schlüsselattributen dar. Auf eine detailliertere Darstellung und Beschreibung wird hier verzichtet und auf die Doxygen-Dokumentation, die für den Programmcode erstellt wurde, verwiesen.

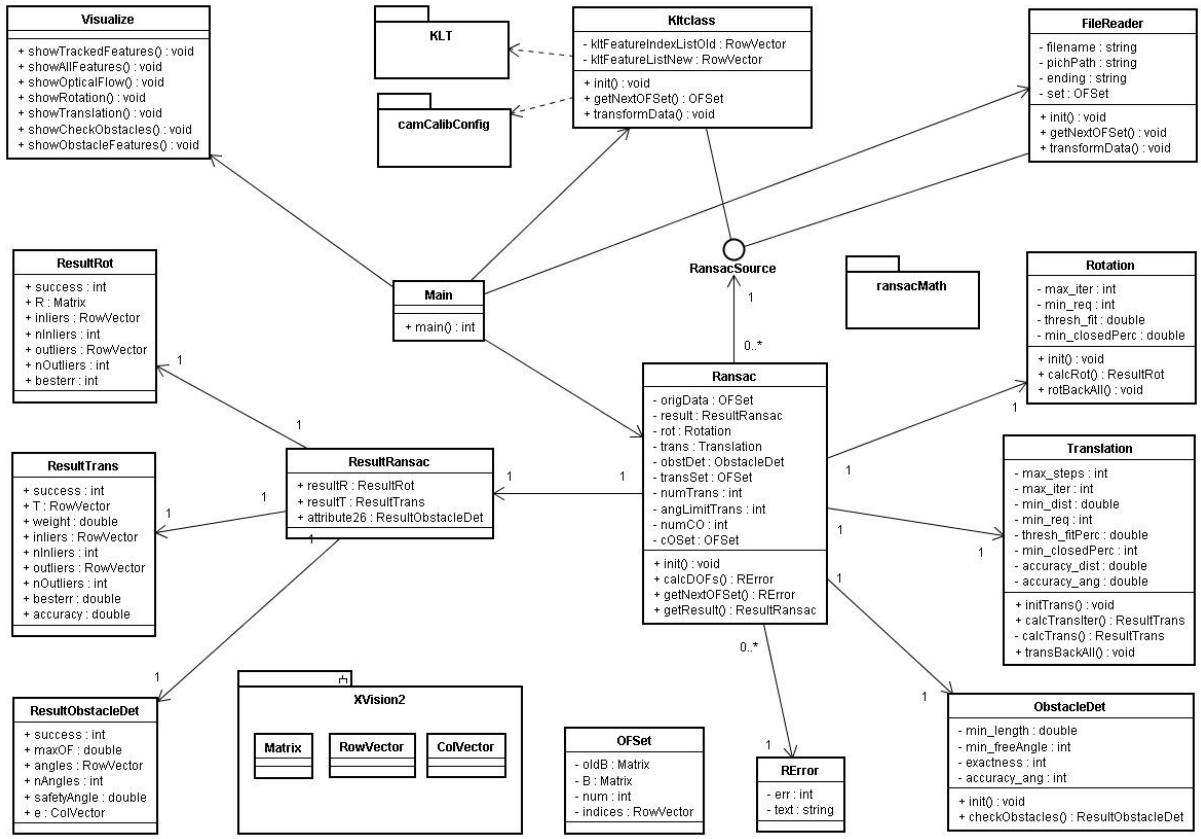


Abbildung C.1: Das dargestellte Klassendiagramm ist vereinfacht und enthält nur Schlüsselfunktionen und -attribute. Die Klasse „OFSet“ und die Datei „ransacMath“ werden beinahe von allen Klassen verwendet - deren Abhängigkeiten wurden daher der Übersichtlichkeit wegen weggelassen.

Anhang D

Testergebnisse an realen Bildern

Dargestellt sind Screenshots der Ergebnis-Visualisierung der beiden Testfahrten durch das Forschungszentrum der TU München in Garching und durch die Stadt Garching selbst. Die Bedeutung der Visualisierung ist in Abschnitt 7.2 beschrieben.
Die verwendete Kamera ist eine einfache Firewire-Kamera. Da die Bilder etwas unscharf waren, wurden sie im Anschluss mit einem Bildverarbeitungsprogramm geschärft. Die Auflösung, das softwareseitige Schärfen und die langsame Bildrate (manuell ausgelöst - etwa alle halben bzw. jede volle Sekunde) führten zu eher schlechten Test-Verhältnissen. Trotzdem lieferte der Algorithmus sehr gute Ergebnisse, solange der KLT-Tracker nicht versagte.

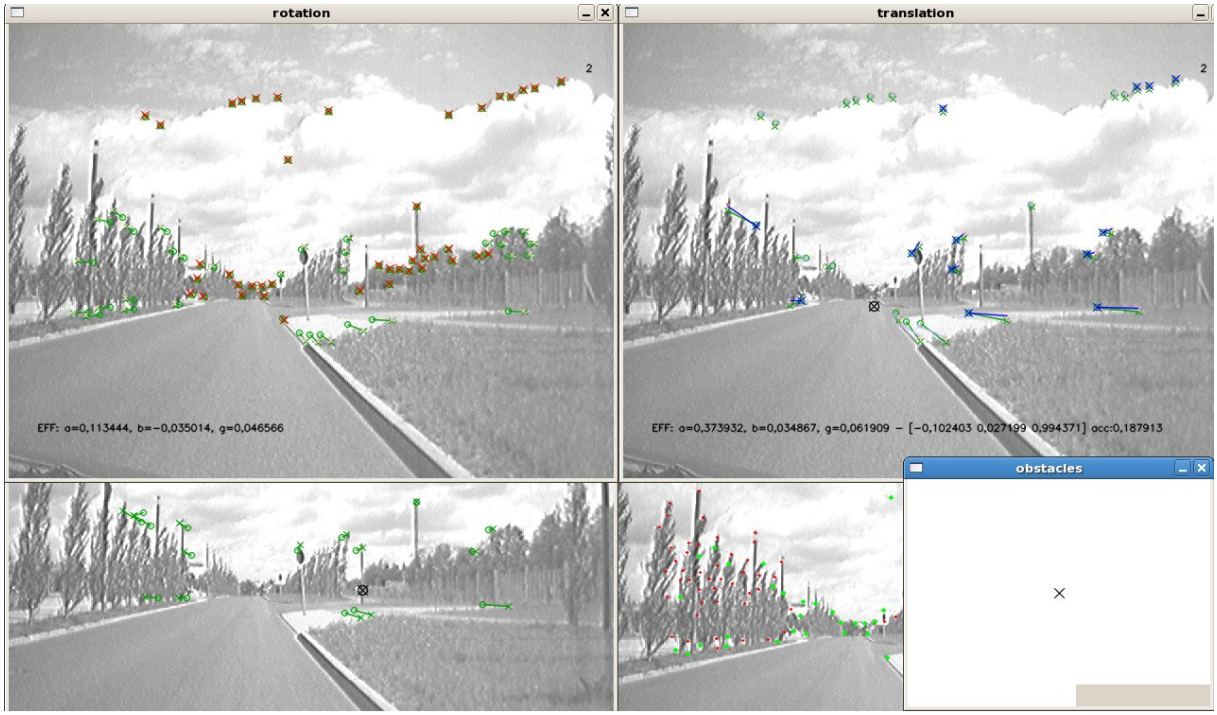


Abbildung D.1: Screenshot während der Berechnung der Bildsequenz, die bei der Testfahrt durch das Forschungsgelände der TUM in Garching gewonnen wurde. Kontrastreiche Merkmalspunkte an den Konturen der Wolken können sehr gut verfolgt werden und stellen gute Punkte für die Rotationsberechnung dar.

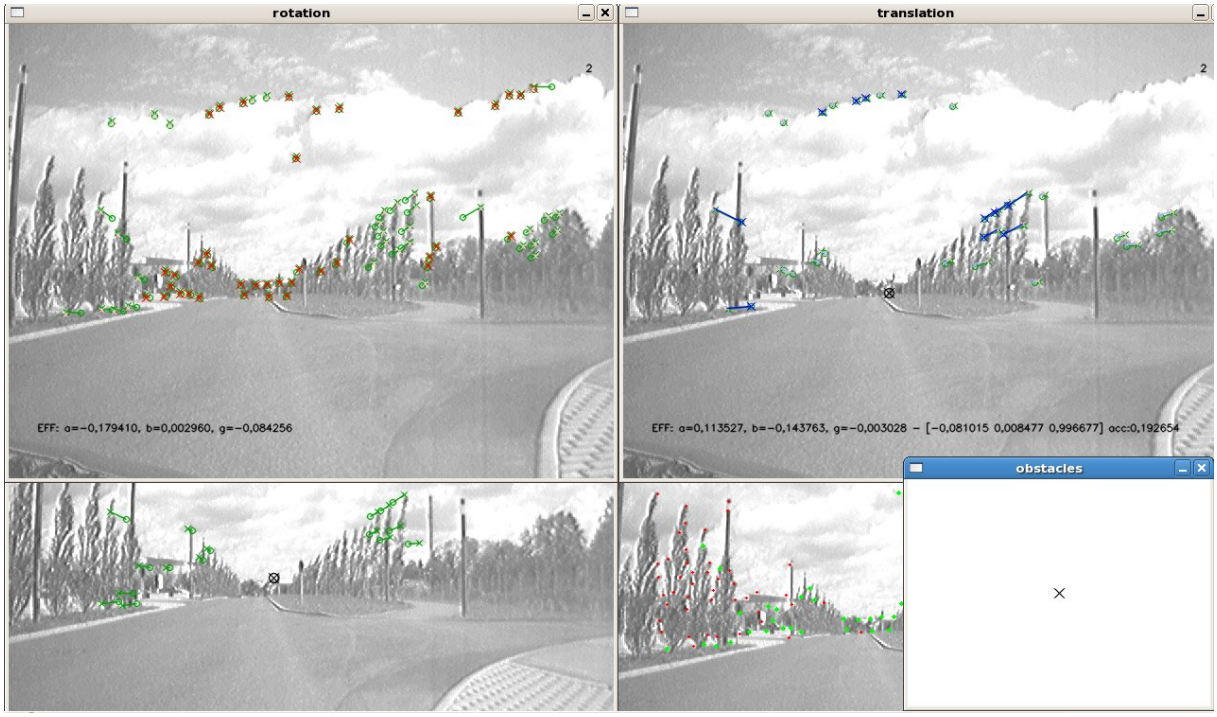


Abbildung D.2: Screenshot während der Berechnung der Bildsequenz, die bei der Testfahrt durch das Forschungsgelände der TUM in Garching gewonnen wurde.

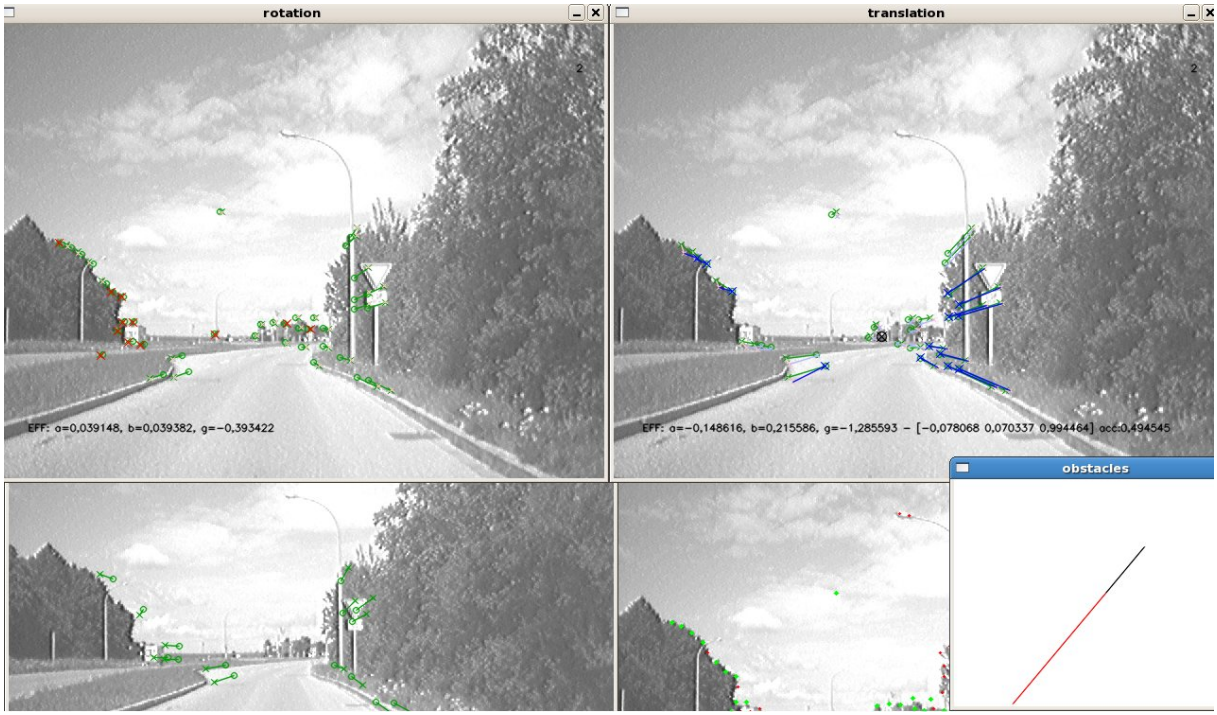


Abbildung D.3: Screenshot während der Berechnung der Bildsequenz, die bei der Testfahrt zur Stadt Garching gewonnen wurde. Das Schild an der rechten Straßenseite wird als Hindernis erfolgreich erkannt.

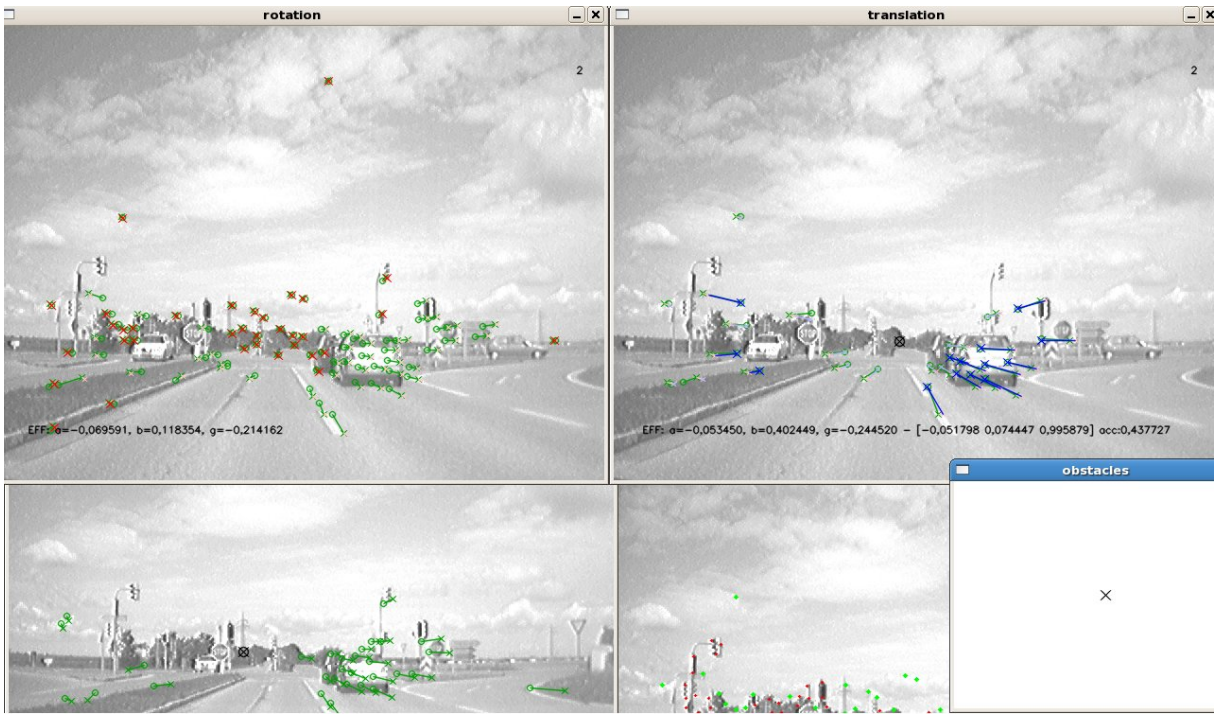


Abbildung D.4: Screenshot während der Berechnung der Bildsequenz, die bei der Testfahrt zur Stadt Garching gewonnen wurde. Obwohl das Auto näher ist als das Schild in Abbildung D.3, wird das Auto nicht als Hindernis erkannt, da die Kamera (das Auto) durch das Abbremsen bei der Kreuzung eine geringere Geschwindigkeit hat und die OF-Vektoren daher kürzer sind.

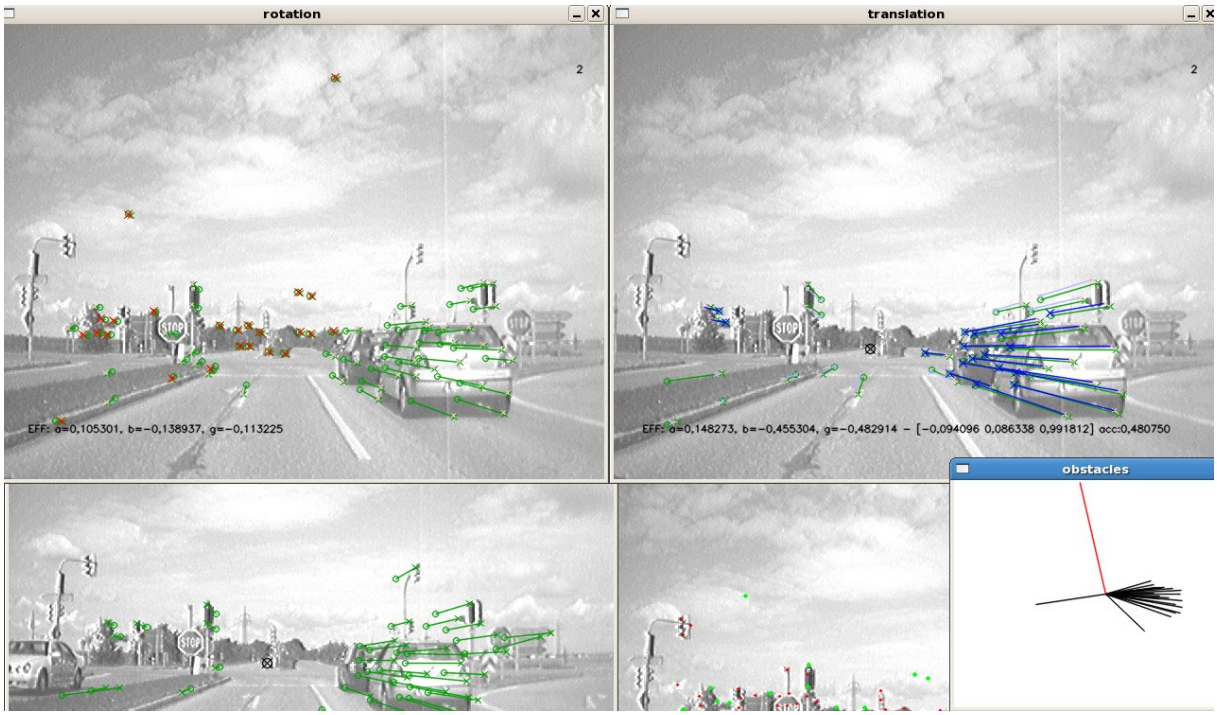


Abbildung D.5: Screenshot während der Berechnung der Bildsequenz, die bei der Testfahrt zur Stadt Garching gewonnen wurde. Das Auto wird nun als Hindernis erfolgreich erkannt.

Literaturverzeichnis

[AHB87] K. S. Arun, T. S. Huang, and S. D. Blostein. Least-squares fitting of two 3-d point sets. *IEEE Trans. Pattern Anal. Mach. Intell.*, 9(5):698–700, September 1987.

[Bir96] Stan Birchfield. Derivation of kanade-lucas-tomasi tracking equation. unpublished, 1996.

[BKPHN87] H.M. Hilden B. K. P. Horn and S. Negahdaripour. Closed-formsolution of absolute orientation using orthonomal matrices, 1987.

[Dav03] Andrew J. Davison. Real-time simultaneous localisation and mappingwith a single camera. In *Proceedings International Conference on Computer Vision - Volume 2*, pages 1403–1412, 2003.

[FB81] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.

[FDT98] C. Thorpe F. Dellaert and S. Thrun. Super-resolved texture tracking of planar surface patches. In *International Conference on Intelligent Robots and Systems - Volume 1*, pages 197–203, 1998.

[Göt68] K. G. Götz. Flight control in drosophila by visual percetion of motion. *Kybernetik, Band 4, Heft 6*, 1968.

[Hor86] B. K. P. Horn. *Robot Vision*. MIT Press, 1986.

[Hor87] B. K. P. Horn. Closed-formsolution of absolute orientation using unit quaternion. *Journal of the Optical Society of America A, Vol. 4, page 629-642*, 1987.

[LF93] Quang-Tuan Luong and Olivier D. Faugeras. Determining the fundamental matrix with planes. In *CVPR '93*, pages 489–494, 1993.

[LF94] Quang-Tuan Luong and Olivier D. Faugeras. A stability analysis of the fundamental matrix. In *ECCV (1)*, pages 577–588, 1994.

[LK81] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *International Joint Conference on Artificial Intelligence*, pages 674–679, 1981.

[NHSA06] Nguyen, Hieu, Smeulders, and Arnold. Robust tracking using foreground-background texture discrimination. *International Journal of Computer Vision*, pages 277–293, September 2006.

[Nis04] David Nister. A minimal solution to the generalised 3-point pose problem. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition 04*, 2004.

[RK05] C. Rasmussen and T. Korrah. On-vehicle and aerial texture analysis for vision-based desert road following. In *IEEE International Workshop on Machine Vision for Intelligent Vehicles*, 2005.

[RV97] Jose A. Ramos and Erik I. Verriest. Total least squares fitting of two 3-d point sets in m-d. In *Proceeding of the 36th Conference on Decision and Control*, pages 5048–5053, 1997.

[ST94] Jianbo Shi and Carlo Tomasi. Good features to track. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 593–600, 1994.

[TK91] Carlo Tomasi and Takeo Kanade. Determining the fundamental matrix with planes. Technical report, Carnegie Mellon University, 1991.

[TV98] E. Trucco and A. Verri. *Introductory Techniques for 3-D Computer Vision.* Prentice Hall, 1998.

[Weh81] R. Wehner. *Himmelsnavigation bei Insekten.* Orell Füssli, 1981.

Abbildungsverzeichnis

Tabellenverzeichnis

Code-Verzeichnis

Index