Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. habil. P. Levi
Betreuer:	Dr. rer. nat. Oliver Zweigle
begonnen am:	23. Juli 2012
beendet am:	17. Januar 2013
CR-Klassifikation:	I.2.9

Diplomarbeit Nr. 3376

Evaluierung von Verfahren zum optischen Lokalisieren und Kartographieren (SLAM) mit Eignung für den Einsatz auf UAVs

Eric Price

Institut für Parallele und
Verteilte Systeme
Universität Stuttgart
Universitätsraße 38
D–70569 Stuttgart

Zusammenfassung

In der Robotik werden kleine UAVs (unmanned aerial vehicles) dank zunehmender Hardwareminiaturisierung immer interessanter. Jedoch gelten auf diesen Plattformen wie Quadcoptern oder Modellflugzeugen zusätzliche Beschränkungen wie Größe, Bauform und Gewicht, die die zur Verfügung stehende Rechnerleistung für Anwendungen der autonomen Robotik nach wie vor einschränken. Thema dieser Diplomarbeit ist die Evaluierung von Verfahren zum simultanen Lokalisieren und Kartographieren an Hand optisch erfasster Bilddaten von einem mikro-UAV aus, wobei besonderes Augenmerk auf der Eignung der SLAM Verfahren für autonome optische Navigation in einer 3D Outdoor-Umgebung liegt.

Inhaltsverzeichnis

Kapitel 1

Einführung

In der Robotik werden kleine UAVs (unmanned aerial vehicles) dank zunehmender Hardwareminiaturisierung immer interessanter. Jedoch gelten auf diesen Plattformen wie Quadcoptern oder Modellflugzeugen zusätzliche Beschränkungen wie Größe, Bauform und Gewicht, die die zur Verfügung stehende Rechnerleistung für Anwendungen der autonomen Robotik nach wie vor einschränken. Thema dieser Diplomarbeit ist die Evaluierung von Verfahren zum simultanen Lokalisieren und Kartographieren an Hand optisch erfasster Bilddaten von einem mikro-UAV aus, wobei besonderes Augenmerk auf der Eignung der SLAM Verfahren[DWB06, BDW06] für autonome optische Navigation in einer 3D Outdoor-Umgebung liegt.

Im Rahmen einer vorangegangenen Studienarbeit[Pri12] wurden von einem UAV während des Fluges Sensordaten erfasst, welche nun zur Evaluierung passender Auswertungsalgorithmen zur Verfügung stehen. Diese werden wir derart aufarbeiten, dass sie in der selben Form und zeitlicher Abfolge zur Verfügung stehen, wie sie auch in Echtzeit während eines Fluges anfallen. Die verarbeitende Software arbeitet somit auf einer sehr realistischen Datenbasis.

Hierfür entwickeln wir eine Softwareschnittstelle, die sowohl zur Anbindung an existierende oder noch zu entwickelnde On-Board Sensor-Hardware geeignet ist, als auch aufgezeichnete Daten in entsprechender Form einlesen und synchronisieren kann. Damit stehen diese Daten weiteren Auswertungskomponenten, wie einem zu evaluierenden SLAM Algorithmus, in einer von der Datenquelle unabhängigen Form zur Verfügung. Mit Hilfe dieser Schnittstelle werden wir außerdem die aufgezeichneten Daten weiter analysieren, um Eigenschaften, die für die Auswahl und Funktion geeigneter SLAM Algorithmen relevant sind, zu extrahieren.

Für die SLAM Evaluierung werden im Rahmen dieser Arbeit zwei Ansätze verfolgt:

- Zum einen betrachten wir die Möglichkeit der Bestimmung von Tiefeninformation aus dem optischen Fluss. Diese soll es uns ermöglichen, generische 3D SLAM Algorithmen zu evaluieren, die auf quasi beliebigen 3D Messpunktwolken arbeiten und etwa im Zusammenhang mit SLAM auf LASER-Scanner-Daten zum Einsatz kommen. Wie wir sehen werden ist dies zwar möglich, jedoch nicht effizient genug. Für eine effiziente Realisierung wäre Spezial-Hardware erforderlich, die parallele Datenverarbeitung zulässt, wie etwa FPGAs[Tri94]. Wir werden daher diesen Ansatz, obwohl er uns wertvolle Erkenntnisse liefert, nicht direkt zu einem funktionstüchtigen SLAM Algorithmus verfolgen können.
- Ohne Tiefeninformationen ist die Wahl möglicher SLAM-Algorithmen deutlich eingeschränkter. Im zweiten Ansatz betrachten wir daher eine direkte Auswertung von 2D Videodaten in Verbindung mit den aufgezeichneten Sensordaten an Hand einer effizienten Implementierung von MonoSLAM [DRMS07], und evaluieren diese.

Als Referenzhardware zum Ansteuern der Sensoren dient das schon bei der Studienarbeit verwendete und ausführlich beschriebene OpenPilot System[Ank09].

1.1 Verwandte Arbeiten

Diese Arbeit ist die logische Fortführung der vorangegangenen Studienarbeit des Autors[Pri12], welche dem Zweck diente, für die Erprobung geeigneter SLAM Algorithmen geeignete Daten zu sammeln.

Sie reiht sich damit auch ein in eine Reihe von Arbeiten mit ähnlicher Zielsetzung, wie dem 2011 von Achtelik et al. durchgeführten optischen SLAM von einem Quadcopter mit nach unten gerichteter Kamera[AAWS11], oder den von Konstantin Schauwecker 2012 auf dem Autonomous Mobile Systems Konferenz präsentierten optischen Navigation mittels horizontaler Stereo-Vision[SKSZ12].

1.2 Übersicht der Arbeit

1.2.1 Datenbasis

Wir beginnen mit einer Betrachtung der verfügbaren Datenbasis in Kapitel 2. Dabei unterscheiden wir zwischen von Sensoren stammenden Rohdaten in Abschnitt 2.1 und Daten die bereits einer Filterung unterworfen wurden in Abschnitt 2.2. Ebenfalls getrennt betrachtet werden die auch getrennt aufgezeichneten Videodaten in Abschnitt 2.3.

1.2.2 Sensordatenakquise & Synchronisation

In Kapitel 3 entwickeln wir die nötigen Software-Komponenten um die vorhandenen Datenaufzeichnungen[Pri11] in Echtzeit abzuspielen und dabei verschiedene Datenquellen zu synchronisieren (Abschnitt 3.3).

1.2.3 State of the art SLAM

Kapitel 4 widmet sich einem Überblick über verfügbare SLAM Verfahren auf aktuellem Stand der Technik. In Abschnitt 4.3 betrachten wir dabei die für unsere Arbeit erforderlichen Eigenschaften und Ansprüche.

1.2.4 Entwicklung & Analyse von SLAM Ansätzen

In Kapitel 5 entwickeln und evaluieren wir schließlich die als geeignet identifizierten Ansätze.

Abschnitt 5.1 widmet sich der Bestimmung von Tiefeninformationen aus optischem Fluss, und den daraus gewonnenen Erkenntnissen, bis hin zur Entwicklung eines eigenen effizienten Verfahrens zur Bestimmung des optischen Flusses an Hand von Template Matches im Unterabschnitt 5.1.3.

In Abschnitt 5.2 evaluieren wir schließlich das MonoSLAM[DRMS07] Verfahren, welches wir mit Hilfe des RT-SLAM Frameworks[RGS+11] implementieren.

1.2.5 Ergebnisse und Ausblick

Wir schließen die Arbeit in Kapitel 6 mit einer Zusammenstellung der gewonnenen Erkenntnisse, sowie deren Implikationen auf eine noch zu entwickelnde Lösung zur optischen Navigation.

1.2.6 Anhang

Der Quellcode der in dieser Arbeit beschriebenen Entwicklungen verteilt sich auf verschiedene öffentlich zugängliche Open-Source Quellen, die im Anhang A aufgelistet sind.

Kapitel 2

Datenbasis

Zur Evaluierung von SLAM Verfahren stehen uns die im Rahmen der Studienarbeit[Pri12] erfassten Daten zur Verfügung. Diese beinhalten nebst Videodaten einer Kamera und aufgezeichneten Sensorwerten auch bereits gefilterte Zustandsinformationen wie Ausrichtung, Geschwindigkeit und Position im Raum, jedoch wurden diese ohne die Zuhilfenahme optischer Informationen mit einem Extended Kalman Filter[WB95] ermittelt und sind dementsprechend fehlerbehaftet.

Alle Daten, mit Ausnahme der Videos sind im UAVObject Format aufgezeichnet. Auch in Echtzeit übermittelte Daten wären in diesem Format verfügbar, da das von OpenPilot verwendete Übertragungsprotokoll (UAVTalk) genau diese überträgt. Beide Formate sind in der vorangegangenen Arbeit im Detail beschrieben[Pri12].

Während wir uns in der vorangegangenen Arbeit auf die erfassenden Sensoren konzentriert haben, genügt uns hier eine Auflistung der erfassten Daten und ihrer Einheiten sowie die Zeitintervalle in denen sie gemessen wurden.

2.1 Rohdaten

Die folgenden Daten sind über die OpenPilot Hardware direkt als Sensorwerte verfügbar, welche nicht, oder nur trivial korrigiert werden. Korrekturen die vom OpenPilot System bereits durchgeführt werden sind etwaige Multiplikationen und Additionen um gemessene Sensorwerte in physikalische Werte zu konvertieren, wobei konstante Korrekturkoeffizienten zum Einsatz kommen die vorher im Rahmen eines Kalibrierungsvorganges ermittelt wurden. Etwaige angewandte Filter wie Tiefpässe filtern nur Frequenzen die deutlich über der Messfrequenz liegen, wie dies im Rahmen der Analog-Digital-Wandlung üblich ist um Samplingartefakte zu vermeiden[Uns00]. Die einzige Ausnahme ist hierbei das Gyroskop, da sich der Nullpunkt dieses Sensors temperaturabhängig verschiebt und ständig nachjustiert werden muss, siehe Abschnitt 2.2.2.

2.1.1 3 Achsen Accelerometer

Dieser Sensor misst die Beschleunigung in 3 Achsen im lokalen Koordinatensystem des UAV, also relativ zur aktuellen Ausrichtung.

In Echtzeit sind Messwerte von diesem Sensor in Intervallen von 1 ms verfügbar, Aufgezeichnet wurden diese Daten jedoch nur alle 30 ms.

Die Daten sind fehlerbehaftet, sowohl mit Rauschen das sich näherungsweise als Gaußsche Normalverteilung darstellen lässt, als auch mit systematischen Fehlern bedingt durch Kalibrierfehler.

Der gemessene Wert ist ein dreidimensionaler Vektor in der Einheit „Meter pro Sekunde im Quadrat“ (m/s^2).

Siehe auch Abbildung 2.1.

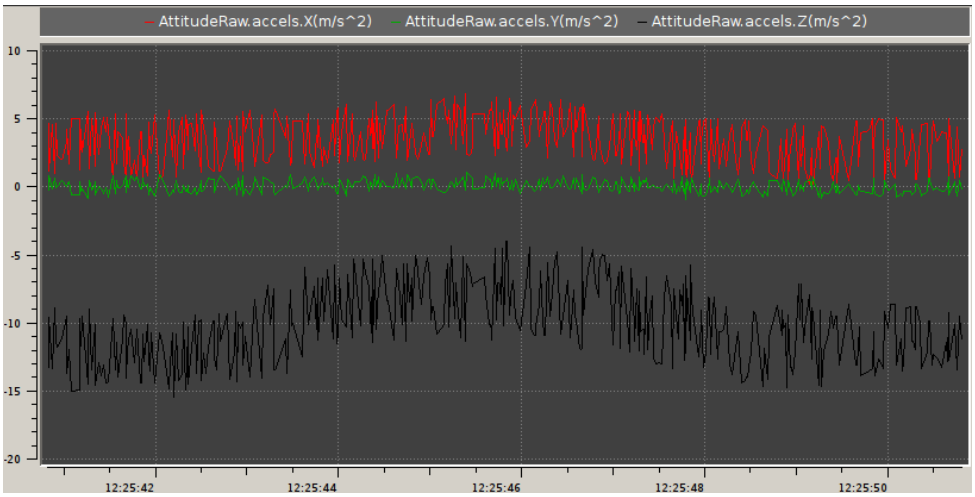

Abbildung 2.1: Während des Fluges aufgezeichnete Accelerometer Messwerte über einen Zeitraum von 10 Sekunden.

2.1.2 3 Achsen Gyroskop

Dieser Sensor misst die Rotationsgeschwindigkeit in 3 Achsen im lokalen Koordinatensystem des UAV, also relativ zur aktuellen Ausrichtung.

In Echtzeit sind Messwerte von diesem Sensor in Intervallen von 1 ms verfügbar, Aufgezeichnet wurden diese Daten jedoch nur alle 30 ms.

Die Daten sind fehlerbehaftet, sowohl mit Rauschen das sich näherungsweise als Gaußsche Normalverteilung darstellen lässt, als auch mit systematischen Fehlern bedingt durch Kalibrierfehler. Die Daten des Sensors selbst sind darüber hinaus mit einem temperaturabhängigen Fehler behaftet, jedoch wird dieser bereits, wie in 2.2.2 beschrieben, durch On-Board Filter korrigiert.

Der gemessene Wert ist ein dreidimensionaler Vektor in der Einheit „Grad pro Sekunde (°/s)“.

Siehe auch Abbildung 2.2.

2.1.3 3 Achsen Magnetometer

Dieser Sensor misst die Stärke des lokalen Magnetfeldes in 3 Achsen im lokalen Koordinatensystem des UAV, also relativ zur aktuellen Ausrichtung.

In Echtzeit sind Messwerte von diesem Sensor in Intervallen von 1 ms verfügbar, Aufgezeichnet wurden diese Daten jedoch ebenfalls nur alle 30 ms.

Die Daten sind fehlerbehaftet, sowohl mit Rauschen das sich näherungsweise als Gaußsche Normalverteilung darstellen lässt, als auch mit systematischen Fehlern bedingt durch Kalibrierfehler.

Statische Magnetfelder die ihren Ursprung auf dem UAV selbst haben werden bereits durch die Kalibrierung weitestgehend kompensiert. Damit erlaubt dieser Sensor die Bestimmung der Ausrichtung relativ zu externen Magnetfeldern, in der Regel sind diese beim Flug außerhalb von Gebäuden vom Erdmagnetfeld dominiert. Wenn die Ausrichtung des Erdmagnetfelds bekannt ist kann somit auch die absolute Ausrichtung des UAV im Raum ermittelt werden.

Der gemessene Wert ist ein dreidimensionaler Vektor in der Einheit „Milligauß“.

2.1.4 Statischer Drucksensor

Dieser Sensor misst den atmosphärischen Druck der am UAV anliegt, sowie die aus diesem und der Lufttemperatur errechnete barometrische Höhe.

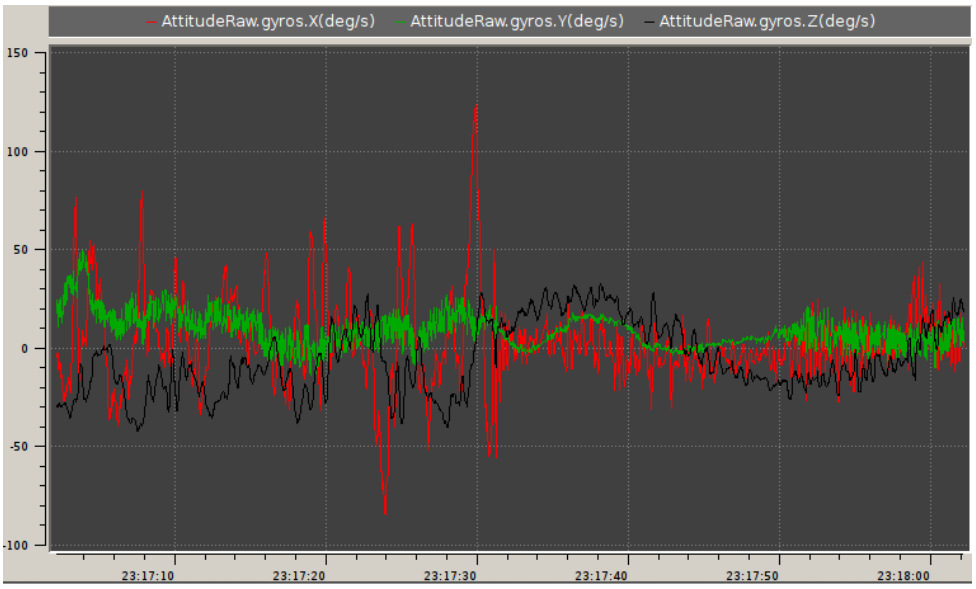

Abbildung 2.2: Während des Fluges aufgezeichnete Gyroskop Messwerte über einen Zeitraum von 60 Sekunden.

In Echtzeit sind Messwerte von diesem Sensor in Intervallen von 5 ms verfügbar, Aufgezeichnet wurden diese Daten alle 30 ms.

Die Daten sind fehlerbehaftet, sowohl mit Rauschen das sich näherungsweise als Gaußsche Normalverteilung darstellen lässt, als auch mit systematischen Fehlern, etwa einem möglichen Druckabfall oder Anstieg bei hohen Fluggeschwindigkeiten bedingt durch aerodynamische Effekte. Größter Faktor ist jedoch der insbesondere vom Wetter abhängige barometrische Druck, der sich als nahezu konstante statische Abweichung äußert.

Der Sensor liefert eine Komponente der absoluten Positionierung, jedoch ausschließlich in der Vertikalen.

Der gemessene Wert ist der Druck in der Einheit „Millibar“ sowie die daraus errechnete Höhe über Normal Null in der Einheit „Meter über Normal Null“.

Die Aufzeichnung dieses Sensors sind auch in Abbildung 2.3 dargestellt.

2.2 Interpretierte Daten

Diese Informationen sind das Resultat einer Filterung, entweder auf dem Sensor selbst oder auf der OpenPilot[Ank09] Hardware, die sowohl für die Aufzeichnung der Daten als auch zur Hardware Ansteuerung beim Einsatz in Echtzeit zum tragen kommt.

2.2.1 GPS Sensor

Dieser Sensor misst die absolute Position im Raum durch Vergleich der Signallaufzeiten zwischen mehreren Satelliten und dem Empfänger. Neuere Sensoren messen zusätzlich auch noch die absolute Geschwindigkeit über Frequenzabweichungen die auf dem Dopplereffekt beruhen[BGWZ00], diese Werte sind aber in den offline zur Verfügung stehenden Daten nicht enthalten.

In Echtzeit sind Messwerte von diesem Sensor alle 25 ms verfügbar, das Intervall der Aufzeichnung betrug wegen der Verwendung eines älteren Sensors jedoch 100 ms.

Die Daten sind fehlerbehaftet. Im GPS Sensor arbeitet in der Regel ein EKF[GWA01], ausgelesen wird dessen Zustandsannahme die jedoch mit einem relativ großen Fehler behaftet sein kann. Dieser Filter eliminiert in der Regel hochfrequente Fehler und Rauschen, jedoch kann die Position dennoch um mehrere Meter abweichen. Lediglich bei der Mittlung über mehrere Stunden können wir davon ausgehen, dass

der Fehler gegen 0 geht. Der Fehler ist dabei empfangsabhängig. Schlechterer Empfang führt aber wegen dem im GPS Empfänger arbeitenden EKF nicht sofort zu einer schlechteren Position, vielmehr führt dieser zu einer graduell divergierenden Zustandsannahme. Umgekehrt braucht der GPS Empfänger bei verbessertem Empfang auch einige Zeit um sich wieder auf die korrekte Position „einzupendeln“.

Eine genauere Analyse oder Optimierung der Vorgänge innerhalb des GPS Moduls ist nicht Bestandteil dieser Arbeit, sie scheitert an proprietären und nicht offengelegten Algorithmen sowie der begrenzten Zeit.

Der Sensor liefert die 3D Position als Punkt über dem WGS84 Referenzellipsoid[Mal96], gegeben durch Längengrad und Breitengrad sowie Höhe über dem Referenzellipsoid und die Abweichung des WGS84 Geoids vom Ellipsoid an der aktuellen Position. Des weiteren liefert der GPS Empfänger die aktuelle, auf ganze Sekunden abgerundete Uhrzeit, und eine Abschätzung der Empfangsqualität. Der Sensor liefert die aktuelle Bewegungsrichtung und -geschwindigkeit in 2D. Mit dem neueren Sensor ist darüber hinaus auch die Geschwindigkeit im Raum als dreidimensionaler Vektor verfügbar.

Die gemessenen Werte sind daher:

- 2d Position bestehend aus Latitude und Longitude als Ganzzahl in der Einheit „$\frac{1}{10^7}$ Grad“.
- Höhe über dem Referenzellipsoid in der Einheit „Meter“.
- Abweichung des WGS84 Geoids vom Referenzellipsoid an der aktuellen Position in der Einheit „Meter“.
- Die Uhrzeit in der Einheit „Jahr“, „Monat“, „Tag“, „Stunde“, „Minute“, „Sekunde“.
- Eine Abschätzung der Empfangsqualität als „DOP“ Wert[GWA01].
- Die Bewegungsrichtung in der horizontalen in der Einheit „Grad“.
- Die eindimensionale Bewegungsgeschwindigkeit in der Horizontalen in der Einheit „Meter pro Sekunde“.

Sowie mit neueren Sensoren und nicht Bestandteil der Aufzeichnungen:

- Die Bewegungsgeschwindigkeit als 3D Vektor in der Einheit „Meter pro Sekunde“.

Siehe Abbildungen 2.3 und 2.4.

2.2.2 Extended Kalman Filter

Die Sensorfusion auf der OpenPilot Hardware wird mittels eines Extended Kalman Filters durchgeführt. Bei der Aufzeichnung im Rahmen der Studienarbeit lief dieser auf einem dort beschriebenen dedizierten AHRS System, neuere Versionen führen den selben Algorithmus in einem Prozess auf dem Flight Controller selbst durch. Der Filter ist im Detail im Paper von Prof. Schinstock beschrieben[Sch10].

Als Messwerte stehen diesem die bereits beschriebenen Daten von Accelerometer, Gyroskop, Magnetometer, Barometer, und dem GPS zur Verfügung, deren aktuellen Werte jeweils aus den UAVObjekten ausgelesen werden.

Intern wird ein 13-dimensionaler Systemzustand verwaltet, dieser besteht aus

- Position (3D) im absoluten Koordinatensystem NED (north, east, down), in Metern relativ zu einer Referenzposition. (Anmerkung: In den aufgezeichneten Daten ist die Position noch als Ganzzahl in Zentimetern gespeichert.)
- Geschwindigkeit (3D) im absoluten Koordinatensystem NED in Metern pro Sekunde.
- Ausrichtung im Raum (als Quaternion (4D))[Eva77].
- Nullpunkt der Gyroskope (3D) für die lokalen Messachsen X, Y und Z in „Grad pro Sekunde“.

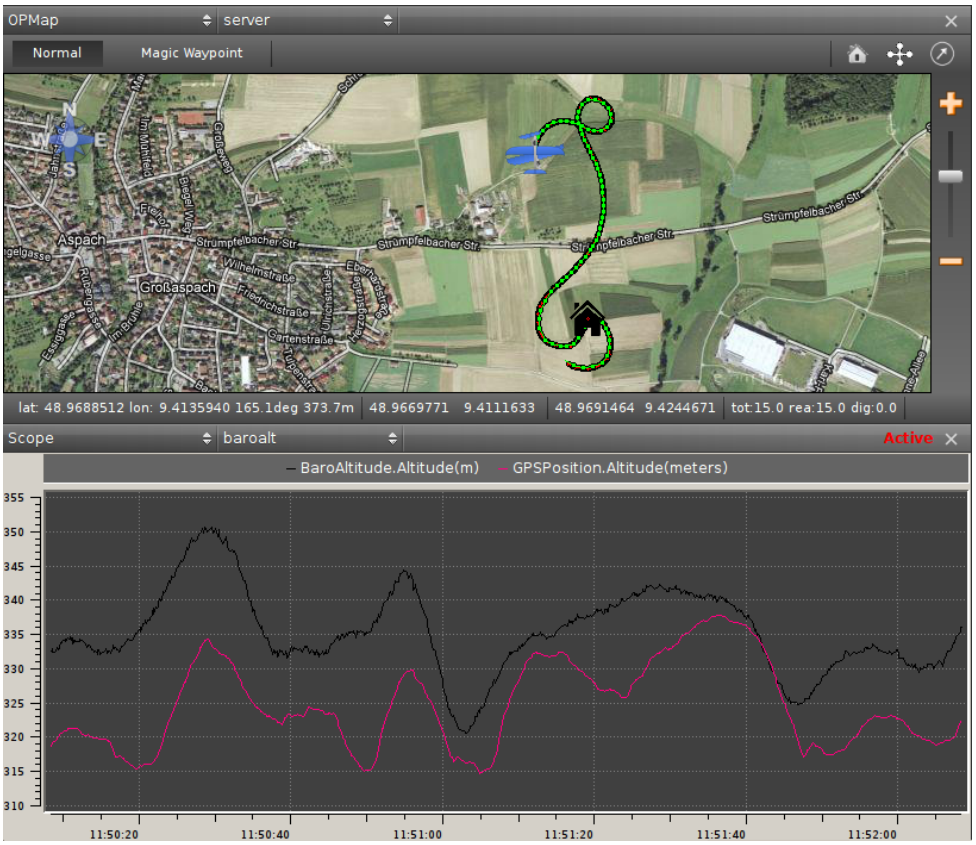

Abbildung 2.3: Während des Fluges aufgezeichnete Positionsdaten aus GPS und Höhenmesser.

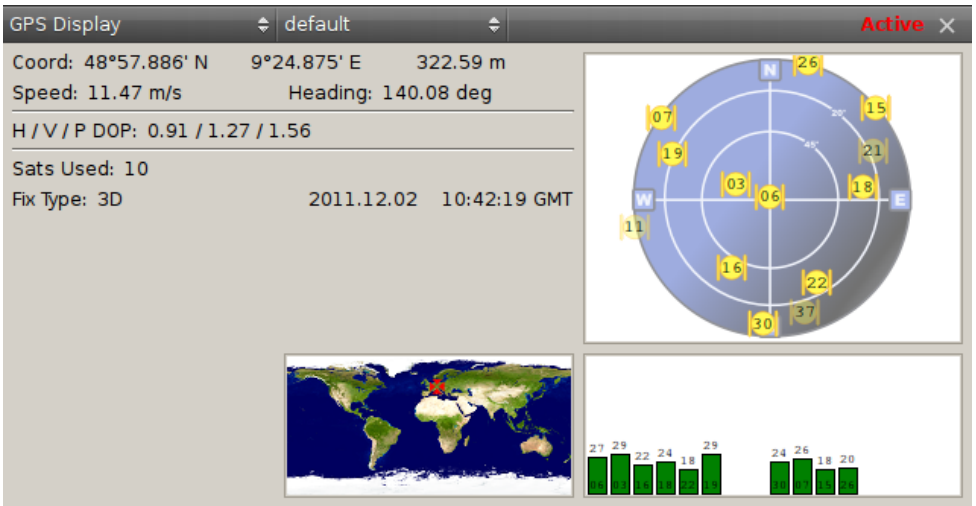

Abbildung 2.4: Während des Fluges aufgezeichnete Daten des GPS Sensors inklusive Uhrzeit, Position, sowie Empfangs- und Orbitaldaten der GPS Satelliten in Reichweite.

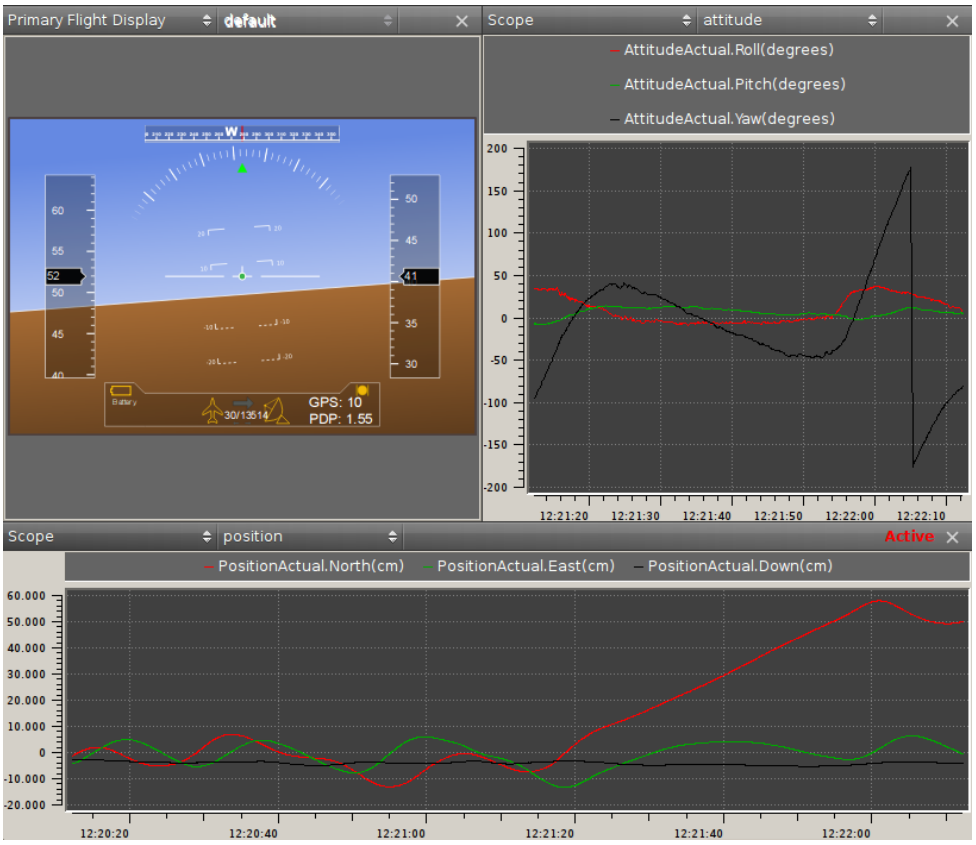

Abbildung 2.5: Während des Fluges aufgezeichnete Positionsdaten und Ausrichtungsdaten als Ergebnis des Extended Kalman Filters.

Letzterer ist Dank der Verfügbarkeit der Magnetometer-Messwerte beobachtbar, und fließt direkt in den Messvorgang des Gyroskopsensors in 2.1.2 ein.

Die wahrscheinlichste Position, Geschwindigkeit und Ausrichtung im Raum werden jeweils in separaten UAVObjekten abgespeichert. Hierbei wird zusätzlich die Darstellung der Ausrichtung als Rotationsquaternion in die Darstellung in Euler Winkeln (Roll, Pitch, Yaw)[Eva77] überführt, beide Darstellungen sind daher für die weitere Verarbeitung verfügbar.

In Echtzeit sind diese Werte mit einer Frequenz von 1000 Hz (Periode 1 ms) verfügbar. Aufgezeichnet wurden diese Daten nur alle 30 ms.

Das Ergebnis der EKF Filterung ist in Abbildung 2.5 sichtbar. Auch die Visualisierungen in späteren Kapiteln, etwa Abbildung 5.26 basiert auf diesen Daten.

2.3 Videodaten

Im Rahmen der Studienarbeit[Pri12] wurden Videodaten mit einer auf dem Flugzeug montierten Kamera aufgezeichnet, die diese intern auf einer Speicherkarte abspeicherte. Die Auflösung betrug 640 auf 480 Bildpunkte, die angegebene Bildrate der Kamera war 30 Bilder pro Sekunde. Wie sie jedoch bei einer genaueren Analyse des Bildmaterials in Abschnitt 3.3.2 herausstellte, war diese sowohl in der Dokumentation als auch im Dateikopf der Videodaten falsch angegeben. Beim Abspielen tatsächlich gezählt werden konnten letztlich nur etwa 18 Bilder pro Sekunde.

Eine Bestimmung der genauen Brennweite sowie der genauen Ausrichtung der Kamera gegenüber den anderen Sensoren fand im Rahmen der Studienarbeit nicht statt. Die Ausrichtung war auch wegen zwischenzeitlich aufgetretenen Deformationen am Rumpf des verwendeten Flugzeugs nicht mehr bestimmbar, daher muss diese Information im Rahmen der Evaluierung aus den aufgezeichneten Videodaten bestimmt werden.

Ein Kamerabild ist in Abbildung 2.6 zu sehen.

Abbildung 2.6: Während des Fluges aufgezeichnetes Video mit nach vorne gerichteter Kamera.

Kapitel 3

Sensordatenakquise & Synchronisation

Wir suchen eine Methode die aufgezeichneten Daten derart abzuspielen, dass diese einem verarbeitenden Programm in der selben Abfolge vorliegen, als würde dieses auf einem On-Board Computer ausgeführt und mit Echtzeitdaten versorgt.

Im Falle der Telemetriedaten von OpenPilot lägen diese Daten sehr wahrscheinlich in Form eines Telemetrie-Datenstroms im UAVTalk Format vor. Auch das OPL Format besteht aus einem in Pakete zerlegten und mit Zeitstempeln versehenen UAVTalk Telemetriedatenstrom. (Dieses wird im Detail in der vorangegangenen Studienarbeit beschrieben [Pri12]). Die Aufgabe war also relativ trivial lösbar, es musste nur ein Programm geschrieben werden dass die in der OPL Datei gespeicherten UAVTalk Pakete in der richtigen zeitlichen Abfolge an einen Empfänger schickt. Für diesen ist dann nicht mehr unterscheidbar ob diese von echten Sensoren oder von einer Aufzeichnung stammen.

Realisiert wurde dies durch eine einfache Schleife, die vor dem Versand jedes Pakets wartet ob der im Zeitstempel angegebene Zeitpunkt bereits erreicht bzw. überschritten ist. Hierbei kam das für diesen Zweck erstellte Programm replay_udp.c (im Verzeichnis flight/Logging) zum Einsatz (siehe Anhang A), welches eine sehr hohe Exaktheit in der zeitlichen Abfolge erreicht. Für längere Wartezeiten wird die POSIX Funktion nanosleep() verwendet[But97], für Wartezeiten unterhalb der Zeit-Granularität des abspielenden Systems wird auf eine simple Warteschleife zurückgegriffen. Ein Ausschnitt des Quellcodes ist in Code Listing „Algorithmus 3.1“ dargestellt.

Das empfangende Programm benötigt die Fähigkeit, UAVTalk Datenströme zu parsen. Hierfür bietet sich insbesondere das in C geschriebene Framework von OpenPilot selbst an. Dies läuft üblicherweise auf einem Embedded-Prozessor unter Benutzung des Echtzeit-Betriebssystems FreeRTOS[Bar03]. Allerdings steht mit „SimPosix“ ein Framework bereit, das die Fähigkeiten von FreeRTOS sowie den Hardware Abstraction Layer von OpenPilot, PiOS, auf einem Linux System emuliert. SimPosix ist seit 2010 Bestandteil von OpenPilot und wurde maßgeblich vom Autor dieser Arbeit entwickelt.

Damit steht nicht nur ein Parser für UAVTalk zur Verfügung, sondern auch eine zuverlässige Multitasking-Umgebung, die auf geänderte UAVObjekte über Ereignisroutinen und Ereigniswarteschlangen zeitkritisch reagieren kann, in der selben Art wie dies an Board des UAVs selbst stattfindet.

Zur Vollständigkeit werden wir in diesem Rahmen einige grundlegende Funktionen der Firmware von OpenPilot kurz vorstellen. Anders als bei der vorangegangenen Studienarbeit, in der hauptsächlich die funktionalen Module vorgestellt wurden, liegt unser Fokus diesmal auf dem zu Grunde liegenden Framework selbst, da wir außer dem Telemetriemodul zum Empfang des UAVTalk Datenstroms keine Funktionalität des Flight-Controllers benötigen.

Abbildung 3.1 zeigt den Informationsfluss zwischen den beteiligten Komponenten.

3.1 SimPosix Framework

Das Framework besteht aus einer Linux-Emulation von FreeRTOS, einem abgespeckten emuliertem Hardware Abstraction Layer PiOS[Pri12], der in erster Linie Daten Ein/Ausgabe Funktionen übers Netzwerk mittels UDP bereitstellt, sowie den OpenPilot Kernkomponenten.

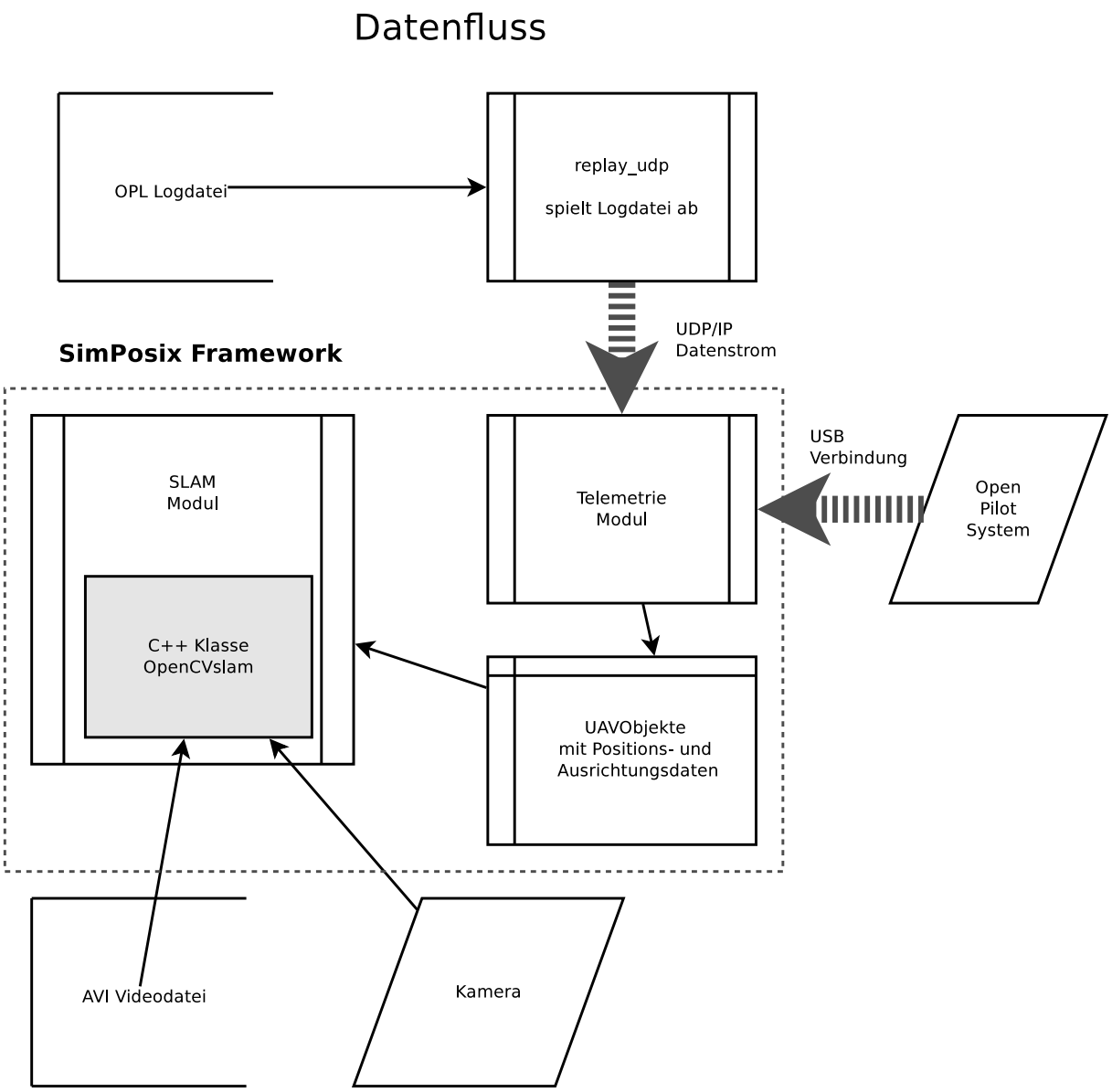

Abbildung 3.1: Diagramm des Datenflusses und der beteiligten Komponenten beim Abspielen der Logdaten.

Algorithmus 3.1 Exakte Warteschleife zur Zeitsynchronisation des abgespielten Datenstroms in replay_udp.c.

```
static inline void dosleep(struct timeval * starttime, unsigned long
    sleeptime) {
        // sleep for the remaining time (if any)
        // if remaining time is smaller than  granularity
        // we are evil and do busy waiting
        struct timeval ctime;
        static struct timespec x;
        static unsigned long timed;
        gettimeofday(&ctime,NULL);
        timed=timediff(starttime,&ctime);
        while (timed<sleeptime) {
                if (sleeptime-timed>granularity) {
                        x.tv_sec=0;
                        x.tv_nsec=granularity*1000/2;
                        nanosleep(&x,NULL);
                }
                gettimeofday(&ctime,NULL);
                timed=timediff(starttime,&ctime);
        }
}
```

3.1.1 UAVObjekte

UAVObjekte sind generische Datencontainer für die Verwaltung von Daten an Bord von OpenPilot. Sie dienen sowohl der Interprozesskommunikation als auch dem Datenaustausch über Telemetrie, inklusive der Telemetriedatenaufzeichnung (logging). UAVObjekte speichern die aktuellen Ist-Werte aller Arten von Sensor-, Konfigurations-, Zustands- und Steuerungsdaten. Das Format und die Fähigkeiten der Objekte sind in [Pri11] detailliert beschrieben.

3.1.2 FreeRTOS Event Queues.

Ein wichtiges Feature von FreeRTOS sind sogenannte Event Queues[Bar03]. Diese erlauben einem Task auf ein Ereignis zu warten, die Programm-Abarbeitung wird unterbrochen bis dieses Ereignis eintritt oder eine optionale Zeitschranke überschritten wird. Hierbei können mehrere Tasks auf das selbe Ereignis warten und werden dann nacheinander abgearbeitet.

3.1.3 OpenPilot UAVObjekt Manager und Event Scheduler

Der UAVObjekt Manager legt Event Queues für alle dem System bekannten UAVObjekte an, deren Ereignisse jeweils bei einem Schreibzugriff auf das jeweilige UAVObjekt ausgelöst werden. Anderen Modulen bzw. deren Tasks ist es so möglich gezielt auf die Änderung eines oder mehrerer UAVObjekte zu reagieren, wodurch die gesamte Programmstruktur ereignisorientiert aufgebaut werden kann.

Darüber hinaus hört der Event Scheduler in einem eigenen Task auf die Änderung beliebiger UAVObjekte und ruft in diesem Fall eine Liste von Callback-Funktionen auf, die vorher von andere Modulen registriert werden können. Diese Rückruf-Funktionen werden dann jedoch im Kontext des Event Scheduler Task ausgeführt.

3.1.4 Modul Task

Die Implementierung eines SLAM Algorithmus, sowie der dafür erforderlichen Vorarbeiten, wie das Einlesen und die Synchronisation von Video und Telemetriedaten, findet daher bevorzugt in einem innerhalb

von SimPosix implementierten Modul Task statt. Dieser kann sowohl die von FreeRTOS und OpenPilot zur Verfügung gestellten Funktionen zur Zeitsynchronisation, als auch beliebige Linux System- und Bibliotheksfunktionen verwenden.

3.2 Videodaten

Zur Verarbeitung von Bild und insbesondere Videodaten bietet sich das OpenCV Framework an[BK08]. Dieses dient als Grundlage für eine Vielzahl von Forschungsprojekten der Computer Vision und stellt Funktionen zur Bild und Video Ein/Ausgabe und Zugriff auf sowohl Kameras als auch Videodateien zur Verfügung. OpenCV bietet darüber hinaus vordefinierte Filterklassen und vereinfachte mathematische Operatoren zur Verarbeitung von Bildern, Vektoren und Matrizen.

3.2.1 Schnittstelle zu C++

OpenCV bietet für die meisten Funktionen ein abwärtskompatibles C Interface, da frühere Versionen dieser Bibliothek noch in C geschrieben waren. Gerade die für effiziente Implementierung von Matrixoperationen erforderlichen Operatoren sind jedoch nur in C++ verfügbar, da das Überladen von Operatoren in C nicht möglich ist[Str97]. Da SimPosix und das für unsere Zwecke erstellte Modul in C vorliegen war es daher erforderlich das OpenPilot Framework dahingehend zu erweitern, dass auch C++ Code compiliert und vom Linker[CT03] in der selben ausführbaren Datei zusammengeführt wird. Hierzu ist es erforderlich eine C++ Klasse von einer zu C kompatiblen Funktion zu instantiieren, die wiederum in einer C Header Datei deklariert ist um den Aufruf aus anderen C Funktionen zuzulassen. Siehe Code Listing „Algorithmus 3.2“.

3.2.2 Einlesen von Videodaten

Da OpenCV direkt auf die Videoquelle zugreift muss an dieser Stelle unterschieden werden ob die Videodaten von einer Film-Datei oder von einem Video-Device stammen. Da die weitergehende Entwicklung jedoch sowieso andauernd neue Kompiliervorgänge erfordert, sprach an dieser Stelle nichts gegen eine sehr einfache Lösung:

Die jeweils nicht benötigte Videoquelleninstantiierung wurde vor dem Kompiliervorgang jeweils auskommentiert. Auch die verwendete Videodatei wurde hart kodiert.

Siehe Code Listing „Algorithmus 3.3“.

3.3 Synchronisation

Bedingt durch die separate Aufzeichnung der Video und sonstigen Daten müssen diese jedoch für die Zwecke dieser Arbeit zusammengefügt und synchronisiert werden.

Beide Datenformate, das für Videos verwendete AVI Containerformat[Jac07], sowie das von OpenPilot verwendete OPL Log Format[Pri12] erlauben ein zeitlich korrektes Abspielen in Echtzeit dank eingebetteter Zeitstempel. Werden beide Dateien simultan eingelesen und die Daten in dieser Form zeitkorrigiert wiedergegeben, reduziert sich der erforderliche Synchronisationsaufwand daher auf die Bestimmung eines konstanten Offsets, da die Aufzeichnungen nicht zum selben Zeitpunkt beginnen.

Wie bereits im Rahmen der Studienarbeit festgestellt wurde[Pri12] ist die grobe Bestimmung dieses Offsets an Hand eindeutiger Merkmale wie Start und Landung möglich, eine Feinjustierung erfordert die Korrelation einzelner Ereignisse, die sowohl im Video als auch in den Telemetriedaten erkennbar sind. Das am einfachsten manuell bestimmbare Merkmal ist die Ausrichtung des Horizonts, bzw. deren zeitliche Änderung, die sich mit den gespeicherten Daten für die Ausrichtung im Raum, insbesondere in der Längsachse (Roll Winkel) visuell abgleichen lässt.

Algorithmus 3.2 Kombination von C und C++ im selben Programm.

```
// C header file: opencvslam.h
...
void opencvslam(SLAMSettingsData* settings);
...

// C++ header file: opencvslam.hpp
...
class OpenCVslam {
public:
        OpenCVslam(SLAMSettingsData * newsettings);
        void run();
...

// C++ code file: opencvslam.cpp
...
OpenCVslam::OpenCVslam(SLAMSettingsData * newsettings) {
        // initialisations...
}
OpenCVslam::run() {
        // stuff...
}
...
/**
 * bridge to C code
 */
void opencvslam(SLAMSettingsData *settings) {
        OpenCVslam *slam = new OpenCVslam(settings);
        slam->run();
}
```

Algorithmus 3.3 Initialisierung der Videoquelle ist hart kodiert.

```
        /* Initialize OpenCV */
        VideoSource = cvCaptureFromFile("test.avi");
        //VideoSource = cvCaptureFromCAM(0);
...
        // retrieve video position if any, then delay
        double pos=cvGetCaptureProperty(VideoSource, CV_CAP_PROP_POS_MSEC);
...
```

3.3.1 Video Synchronisation auf Software Ebene

Die Implementierung einer Kompensation des Zeitversatzes zwischen Video und Telemetriedaten erfolgte zunächst im Abspielprogramm für die Telemetrie Logdateien, replay_udp.c, wie wir es am Anfang von Kapitel 3 beschrieben haben. Diese Vorgehensweise war naheliegend, da bei der Aufzeichnung die Telemetrieaufzeichnung jeweils bereits mit dem Einschalten der Stromversorgung begann, also deutlich vor dem Beginn der Videoaufzeichnung. Das Programm modifizieren wir also dahingehend, dass Telemetriedaten beim Abspielen entsprechend des vorher zu ermittelnden Zeitversatzes übersprungen werden.

Die Videodaten werden jedoch von einem SimPosix Task eingelesen, welcher die jeweils aktuellen Werte der abgespielten Telemetriedaten aus UAVObjekten ausliest. Hier muss sicher gestellt werden, das das Einlesen von Videodaten exakt synchron mit dem Empfang der Telemetriedaten beginnt, ohne etwaige Startverzögerungen bedingt durch das Laden von Dateien. Dies wurde dadurch realisiert, dass die Videodatei zunächst beim Programmstart geöffnet und der erste Frame eingelesen wird, noch bevor das Programm zum Abspielen der Telemetriedaten gestartet wurde. Dann wird jedoch die Ausführung des Modul Tasks so lange gestoppt bis eine Änderung an einem UAVObjekt den beginnenden Empfang der Telemetriedaten signalisiert. Erst in diesem Moment wird der Timer für das zeitkorrekte Abspielen der Videodaten wieder gestartet und läuft somit hinreichend synchron zur Wiedergabe der Telemetriedaten.

3.3.2 Einlesen von der Kamera

Das Einlesen der Videodaten unterscheidet sich geringfügig für das Einlesen von Kameras und das Einlesen von Videodateien.

Bei Kameras wird jeweils eine feste Wartezeit entsprechend der eingestellten Bildwiederholrate gewartet und dann ein neues Bild von der Kamera angefordert.

Bei Videodateien hingegen muss zuerst der Zeitstempel des nächsten in der Datei gespeicherten Videobildes abgefragt werden, um daraufhin genau diesen Zeitpunkt abzuwarten bevor das Bild dann verarbeitet wird. Die hierfür verwendete Funktion cvGetCaptureProperty() ist in Code Listing „Algorithmus 3.3“ erwähnt.

Erst bei der Implementierung und dem Test dieser Vorgehensweise wurde dabei festgestellt, das die im Rahmen der Studienarbeit angefertigten Videos nicht in den vom Hersteller angegebenen 30 Bildern pro Sekunde aufgezeichnet wurden, sondern, obwohl im Video-Dateikopf die Wiederholrate von 30 Bildern pro Sekunde hinterlegt war, nur durchschnittlich 18.3 Bilder pro Sekunde tatsächlich abgespeichert waren. Die verzeichneten Aufnahmeintervalle der Einzelbilder schwankten dabei recht stark zwischen 30 Millisekunden bis zu 100 Millisekunden, vermutlich bedingt durch die Bildverarbeitung und Ein/Ausgabeprozesse auf der Kamera. Ein Screenshot der Einzelbildintervalle und deren ermittelter Mittelwert ist in Abbildung 3.2 abgedruckt.

3.3.3 Auslagerung in separate Ein/Ausgabe Threads

An dieser Stelle gibt es allerdings ein Problem mit der Taskverwaltung von SimPosix bzw. FreeRTOS. FreeRTOS ist für einen einzelnen CPU Kern ausgelegt und lässt jeweils nur einen einzelnen Task zur selben Zeit ablaufen. Im Rahmen der Task-Preemption von FreeRTOS wird dieser Task in der „SimPosix“ Implementierung mittels eines Signal-Handlers regelmäßig angehalten, was wiederum den Task-Thread solange in einer Semaphore pausieren lässt bis der FreeRTOS Scheduler diesen erneut zur Ausführung freigibt.

Beim Zugriff auf Videoquellen ist dieses Verhalten nicht wünschenswert. Die Granularität der FreeRTOS Simulation ist gegenüber der Linux Scheduling Granularität recht grob, dadurch kommt es bei Ein/Ausgabefunktionen zu Synchronisationsproblemen und Verzögerungen. Darüber hinaus besteht während der vergleichsweise langen Wartezeiten die Gefahr, dass Ein/Ausgabepuffer überlaufen.

Es war daher notwendig, einige Aufrufe von OpenCV, die zur Ansteuerung von Hardware dienen, wie eben das Einlesen von Videodaten oder die Anzeige von Bildern auf dem Bildschirm über die Debug Funktionen von OpenCV, in einen separaten Thread auszulagern. Hierfür wurde eine Wrapperfunktion geschrieben, die das selbe Interface bereitstellt wie die entsprechende Funktion von OpenCV. Die eigentliche Funktionalität aber in einen neu erstellten temporären Thread auslagert, mit dem bei dessen Beendigung wieder synchronisiert wird.

```
...
frame 5572 took 0,040119 ms (24,925846 fps) average 0,054653 (18,297136 fps)
frame 5573 took 0,042801 ms (23,363940 fps) average 0,054651 (18,297847 fps)
frame 5574 took 0,041162 ms (24,294253 fps) average 0,054649 (18,298657 fps)
frame 5575 took 0,052788 ms (18,943699 fps) average 0,054648 (18,298769 fps)
frame 5576 took 0,043300 ms (23,094689 fps) average 0,054646 (18,299450 fps)
frame 5577 took 0,058044 ms (17,228309 fps) average 0,054647 (18,299245 fps)
frame 5578 took 0,038802 ms (25,771866 fps) average 0,054644 (18,300197 fps)
frame 5579 took 0,057709 ms (17,328319 fps) average 0,054645 (18,300014 fps)
frame 5580 took 0,041851 ms (23,894292 fps) average 0,054642 (18,300782 fps)
frame 5581 took 0,058016 ms (17,236625 fps) average 0,054643 (18,300580 fps)
frame 5582 took 0,068227 ms (14,656954 fps) average 0,054646 (18,299764 fps)
frame 5583 took 0,042553 ms (23,500106 fps) average 0,054643 (18,300490 fps)
...
```

Abbildung 3.2: Intervalle und Wiederholraten bei Einzelnen Bildern im Video (Screenshot).

Algorithmus 3.4 Auslagerung von Ein/Ausgabefunktionen in separaten POSIX Thread mit voll gesetzter Signalmaske.

```
...
static void* retrieveThread(void* arg) {
        sigset_t set;
        sigfillset(&set);
        pthread_sigmask(SIG_BLOCK, &set, NULL);
        return cvRetrieveFrame((CvCapture*)arg,0);
}
...
/*call OpenCV function cvRetrievFrame() in separate thread, then wait */
IplImage* backgroundRetrieveFrame(CvCapture *VideoSource) {
        IplImage* result;
        pthread_t retriever;
        pthread_attr_t threadAttributes;
        pthread_attr_init( &threadAttributes );
        pthread_attr_setdetachstate( &threadAttributes,
           PTHREAD_CREATE_JOINABLE );
        pthread_create( &retriever, &threadAttributes,
                        &retrieveThread, VideoSource);
        pthread_join(retriever,(void**)&result);
        return result;
}
...
```

Dieser Thread, kreiert mittels der POSIX Funktion pthread_create()[But97], muss zudem von der Taskverwaltung von SimPosix, die ebenfalls auf POSIX Threads basiert, entkoppelt werden. Dessen Signalmaske wird entsprechend angepasst um die vom FreeRTOS Scheduler ausgesandten Signale zu ignorieren, womit der neue Thread unabhängig von FreeRTOS und außerhalb der Kontrolle dessen Schedulers abläuft, aber dennoch mit SimPosix Tasks Daten austauschen kann.

Siehe Code Listing „Algorithmus 3.4“.

3.3.4 Bestimmung des Zeitversatzes

Um manuell den exakten Zeitversatz eines Paares aus Video und Telemetriedaten zu bestimmen, benötigt es eine Visualisierung eines visuell im Bild leicht abgleichbaren Merkmals, in diesem Fall der Neigung des Horizonts. Dies wurde realisiert in dem zum einen das gerade verarbeitete Videobild auf dem Bildschirm angezeigt wurde, zum anderen dieses mit einer gelben Linie entsprechend des Horizonts überlagert wurde. Diese Linie wurde in Realisierung eines einfachen künstlichen Horizonts zuerst entsprechend dem Neigewinkel (Pitch) nach oben bzw. unten verschoben, und dann entsprechend dem Rollwinkel verdreht. Siehe Abbildung 3.3.

Dieses Vorgehen in Verbindung mit einem verlangsamten Abspielen von sowohl Video als auch Telemetrie erlaubte es, den exakten Versatz manuell auf wenige Millisekunden genau zu bestimmen.

3.3.5 Bestimmung des Rotationsversatzes

Für die exakte Bestimmung des erforderlichen Pixel-Versatzes pro Grad Pitch-Neigung wäre hierzu ein exaktes Kameramodell und die Bestimmung der Brennweite erforderlich gewesen, dies wurde an dieser Stelle jedoch noch nicht bestimmt, stattdessen erfolgte die Bestimmung des Verhältnisses von Drehung zu Pixelversatz zunächst experimentell durch Beobachtung des sichtbaren Horizonts.

Dabei wurde aber bereits sichtbar, dass die Kamera auf dem Modell merklich verdreht angebracht war, die Blickachse der Kamera war dabei gegenüber der Rumpfausrichtung um etwa 5 Grad nach rechts geneigt,

Abbildung 3.3: Ein ins Video eingeblendeter gelber Balken, der die Horizontlinie gemäß des künstlichen Horizonts wiedergibt, erlaubt die visuelle Synchronisierung von Video und Telemetriedaten.

um etwa 10-15 Grad nach unten zeigend. Dieses Phänomen war durchaus erwartet, die Kamera wurde absichtlich leicht nach unten blickend angebracht, um im Horizontalflug mehr kontrastreiches Gelände aufnehmen zu können. Es war jedoch an dieser Stelle bereits klar, dass zur realistischen Evaluierung des SLAM Algorithmus eine exaktere Bestimmung des Kameraversatzes nötig sein würde.

Darauf werden wir in 5.2.7.2 noch einmal zurückkommen.

Kapitel 4

State of the art SLAM

Bei SLAM geht es darum, sowohl die Position von Landmarken, also von Sensoren erkannten Merkmalen, zu bestimmen, als auch die Position des Beobachters an Hand der selben Messung relativ zu diesen Landmarken zu berechnen. Die Lösung des einen Problems bedingt hierbei jeweils die Lösung des anderen. Nur wenn die eigene Position erkannt ist kann die Position von relativ zum Sensor gemessenen Landmarken bestimmt werden, nur wenn die Position dieser Landmarken bekannt ist, kann die des Sensors berechnet werden. Beides ist also nur in einem kombinierten Ansatz möglich der beide Probleme löst. Bewegungen des Sensors, von Landmarken und Messfehler bringen Unsicherheit in dieses System die Schritt für Schritt reduziert werden muss.

Für unsere Zwecke kommen ausschließlich Echtzeit online Verfahren in Frage, bei denen das SLAM Problem kontinuierlich an Hand der jeweils aktuellen Sensordaten gelöst wird. Offline Verfahren, die im Nachhinein den wahrscheinlichsten Systemzustand auf Grund ganzer Messdatensätze berechnen sind für autonome Steuerung von Flugobjekten bei denen schnelle Reaktionen wichtig sind, offensichtlich nur bedingt geeignet.

Ein Überblick in SLAM Verfahren generell wurde bereits in der vorangegangenen Studienarbeit[Pri12] gegeben, wir konzentrieren uns daher an dieser Stelle auf zwei Ansatztypen. Zum einen generische SLAM Ansätze auf 3D Daten[BDW06, ENT05, DWB06], die sich hauptsächlich in der benutzten Kartendarstellung und Methoden zum Loop Closing unterscheiden, zum anderen MonoSLAM[AAWS11, DRMS07] und vergleichbare, die in der Lage sind 2D Daten aus einer Bilddatenquelle direkt zu verarbeiten.

4.1 Generische 3D SLAM Verfahren

Wie unter anderem in der Diplomarbeit von Aichele[Aic07] im Detail aufgeführt, verwalten 3D SLAM Algorithmen Landmarken im Raum, die neben einer Positionsabschätzung und Angaben zur Unsicherheit auch Daten enthalten die eine Wiedererkennung erlauben sollen. Im Fall von EKF basierten Algorithmen werden zusätzlich auch Kovarianzen von Landmarken untereinander verwaltet.

Es wird jedoch davon ausgegangen, dass eine direkte Messung der Position einer Landmarke im 3D Raum möglich ist, etwa über Laser-Entfernungsmessung[Axe99], Sonar-Laufzeiten[Mág84] oder mittels Parallaxenvergleich bei Stereobildern[SKSZ12]. Lediglich die eindeutige Zuordnung von Messung zu Landmarke muss eventuell durch eine Abschätzung bestimmt werden. Hierbei unterscheiden sich die Methoden wiederum nach Datenquelle, bei Laserscannerdaten wird zum Beispiel häufig die dreidimensionale Struktur analysiert[Axe99].

Da wir über eine Kamera verfügen, haben wir eine Vielzahl von Methoden aus dem Bereich Bildverstehen zur Verfügung um Landmarken eindeutig zu erkennen. Beispielhaft erwähnt sei hier Template Matching[Bru09] oder der Abgleich von Farbhistogrammen[SS94]. Was uns dagegen fehlt ist die Fähigkeit Tiefeninformationen aus dem Bild zu extrahieren.

Um generische SLAM Algorithmen zu evaluieren sollten wir also zuerst eine Möglichkeit finden, eine wenn auch fehlerbehaftete Abschätzung der Tiefeninformation zu erhalten.

Abbildung 4.1: SLAM6D Beispielbild, Quelle: OpenSLAM.org `http://openslam.org/slam6d.html`

Postulieren wir, dass uns dies gelingt, haben wir wiederum eine Vielzahl von Algorithmen und verfügbaren Implementierungen, die wir im Rahmen dieser Arbeit evaluieren könnten. In Frage kommende Implementierungen etwa auf OpenSLAM.org[SFG12] wären unter Anderem:

- SLAM6D[NLS+12, Nüc09], ein Verfahren zur Arbeit auf 3D Punktwolken. Siehe Abbildung 4.1.
- TreeMap[Fre07, Fre12], ein für SLAM geeignetes Inferenzverfahren für Gaußverteilungen. Siehe Abbildung 4.2.
- Thin Junction Tree Filter for SLAM[Pas02, Pas12], ein Verfahren zur effizienten Inferenz von partiellen Kalman-Filter basierten Zustandsabschätzungen (Abbildung 4.3).

4.1.1 PTAM - Parallel Tracking and Mapping

Auch der von Konstantin Schauwecker auf der Autonomous Systems 2012 vorgestellte[SKSZ12] und auf der Arbeit von Klein und Murray aufbauende PTAM Ansatz[KM07] fällt in diese Kategorie. PTAM ist dadurch charakterisiert, dass die Lokalisation und das Mapping parallel und unabhängig voneinander durchgeführt werden, wobei prinzipiell auch völlig unabhängige Ansätze verfolgt werden können. Hierbei setzten jedoch sowohl Klein und Murray, als auch Schauwecker auf einen Stereokamera Ansatz, bei dem zu jedem bei diesem Ansatz für den Mapping-Schritt relevanten Keyframe auch Tiefeninformation vorhanden ist. Ein wichtiges Feature dieses Ansatzes ist, dass Mapping eben nicht in jedem Bild durchgeführt wird, sondern nur ausgehend von Schlüsselbildern, die ausreichend Perspektivenunterschied aufweisen. Der Lokalisierungsschritt wird kontinuierlich zu jedem Kamerabild ausgeführt, jedoch nur basierend auf wenigen bekannten Landmarken, was dessen Rechenaufwand begrenzt.

Anders als bei EKF basierten SLAM Ansätzen kann hier eine zum Schritt k stattfindende Verbesserung der Lokalisation die Position von in Schritten $l < k$ kartografierten Landmarken nicht mehr verbessern. Es ist vielmehr Aufgabe eines dedizierten Loop-Closing Schrittes, basierend auf Keyframes in Schritt $f > k$ und dessen Positionsabschätzung der selben Landmarken die Karte gegebenenfalls zu korrigieren.

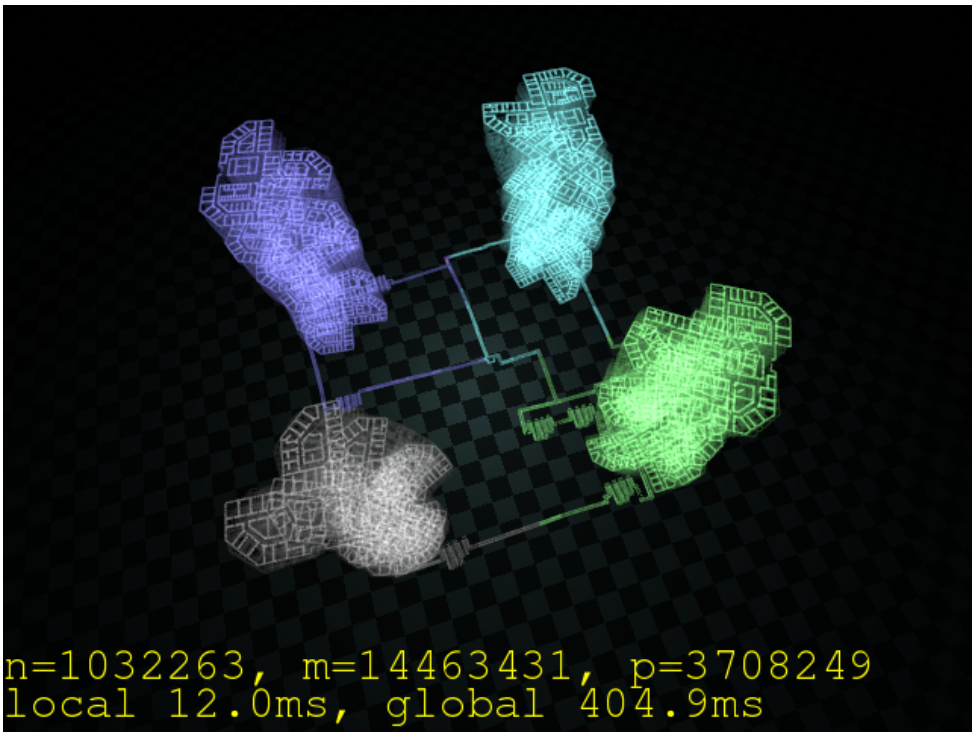

Abbildung 4.2: TreeMap Visualisierung, Beispielbild, Quelle: OpenSLAM.org `http://openslam.org/treemap.html`

Da das Mapping bei PTAM mit deutlich geringerer Frequenz durchgeführt wird, können in diesem Fall teilweise auch Algorithmen die für Offline-SLAM entwickelt wurden eingesetzt werden[KM07].

Dieses Vorgehen ist jedoch in dieser Form, worauf Klein und Murray in ihrem Paper[KM07] ausdrücklich hinweisen, nur für kleine Karten und nicht für offene Explorationsaufgaben geeignet. Schauwecker hingegen verzichtet auf das Loop-Closing und berechnet die Karte immer nur an Hand des aktuellen Keyframes, da dessen Arbeit[SKSZ12] kein dauerhaftes Mapping zum Ziel hat und zur Lageregelung des dort verwendeten Quadcopters lediglich „optische Odometrie“ benötigt.

4.2 SLAM mit integrierter 2D zu 3D Bildinterpretation

Die fehlende Tiefeninformation in Bildern einer einzelnen Kamera (monokulare Vision) ist ein Phänomen, das selbst wiederum als Unsicherheitsproblem abgebildet werden kann. Erst bei Sichtung der selben Landmarke aus unterschiedlichen Blickwinkeln kann die 3D Position eindeutig bestimmt werden. Das Problem ist also mit dem SLAM Problem prinzipiell sehr nahe verwandt. Es liegt daher auch nahe einen Algorithmus zu finden, der beide Probleme gleichzeitig löst. Die Bestimmung der Distanz zum Sensor wird so Teil des Mapping-Problems und die Landmarke wird nicht mehr als 3D Messung in der Welt detektiert, sondern zweidimensional im Kamerabild.

4.2.1 MonoSLAM

Wird als SLAM Verfahren ein Extended Kalman Filter verwendet, lässt sich dieser recht einfach erweitern. Etwa in dem pro Landmarke eine Ausrichtung, also die Normale der Textur bei erster Sichtung, im Zustand kodiert wird, was es erlaubt diese wiederum auch bei abweichenden Blickwinkeln in die Bildebene zu projizieren, und hier die Abweichung von der gemessenen Position zur erwarteten im 2D Kamerabild zu messen. Diese Abweichung entspricht dann genau der "Innovation" im EKF Korrekturschritt, der Rest ist nur Mathematik:

Sei $f(\hat{x}, ...)$ die Zustandsübergangsfunktion, und $h(\hat{x})$ die Funktion für die erwartete Messung für angenommen Zustand $\hat{x}$, so ist im Extended Kalman Filter definiert[WB95]

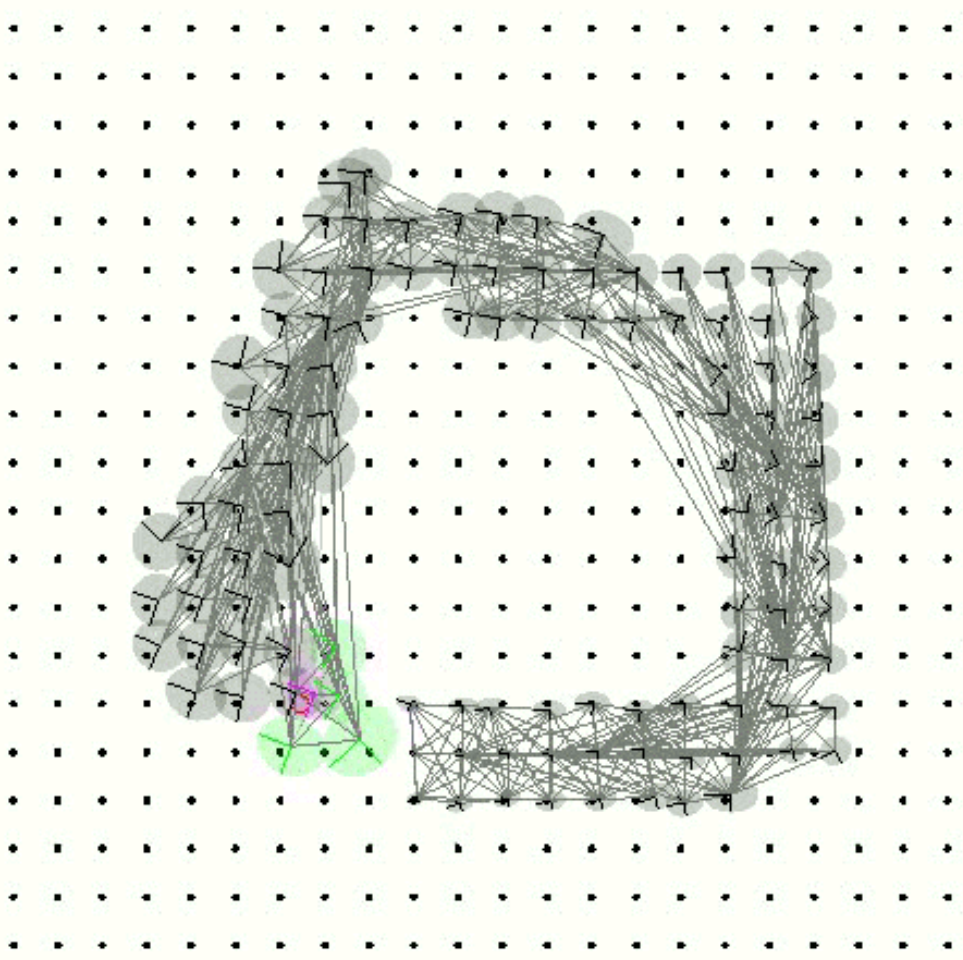

Abbildung 4.3: Thin Junction Tree Filter zur Inferenz von partiellen Zustandsabschätzungen, Beispielbild, Quelle: OpenSLAM.org `http://openslam.org/tjtf.html`

Zustandsschätzung: $\hat{x}_{k|k-1} = f\left(\hat{x}_{k-1|k-1}, ...\right)$

Messvorhersage: $\hat{z}_k = h\left(\hat{x}_{k|k-1}\right)$

Innovation: $\tilde{y}_k = z_k - \hat{z}_k$

Korrektur: $\hat{x}_{k|k} = \hat{x}_{k|k-1} + K * \tilde{y}_k$

(Hierbei ist K der Kalman-Gain und setzt sich prinzipiell zusammen aus der Ableitung von $h()$, deren Umkehrfunktion, und einer Gewichtung, die auf der aktuellen Kovarianzabschätzung beruht.)

Dies ist genau der Kernansatz des von Davison et al 2007 vorgestellten MonoSLAM Algorithmus[DRMS07].

Prinzipiell ist dieses Vorgehen auch mit anderen SLAM Ansätzen als EKF SLAM denkbar. Auch verschiedene Mischformen wurden schon implementiert, wie etwa eine Kombination aus Partikelfilter und EKF[BB09].

Für MonoSLAM spricht seine gezielte Ausrichtung auf Monokulare Videodaten. Der Algorithmus hat jedoch den Nachteil das er ohne weitergehende Optimierungen schlecht skaliert, da er, wie auch der reguläre EKF SLAM, Kovarianzen zwischen allen Landmarken abspeichert und berechnet.

Es werden daher Ansätze gesucht, den Rechenaufwand von MonoSLAM durch Heuristiken zu reduzieren wie dies unter anderem im Ansatz von RT-SLAM gemacht wurde[RGS+11].

4.3 Erforderliche Eigenschaften.

Ein für uns in Frage kommender SLAM Algorithmus muss die folgenden Eigenschaften vereinen.

4.3.1 Online Echtzeit Berechnung

Es muss sich um ein echtzeitfähiges online SLAM Verfahren handeln, also ein Verfahren, welches eine Zustandsabschätzung jeweils an Hand der aktuellsten Sensormessungen berechnen kann. Dabei muss die benötigte Rechenzeit unterhalb der vom selben Verfahren benötigten Messperiode liegen. Eine aktuelle Zustandsabschätzung muss dabei ebenfalls mit für Navigationsaufgaben hinreichender Frequenz zur Verfügung stehen.

4.3.2 Geringer Rechenaufwand

Ein übliches UAV, wie in der Studienarbeit[Pri12] verwendet, bewegt sich mit einer Geschwindigkeit in der Größenordnung von 15 m/s. Um einem im Abstand von 10 Meter erkanntes Hindernis noch ausweichen zu können benötigt es daher Reaktionszeiten von deutlich unter einer Sekunde. Das Gesamtsystem des Flugzeugs, bestehend aus Autopilot, Lageregelung und Servos, benötigt jedoch zusätzliche Zeit um einen Steuerbefehl auszuführen und auch eine Flugbahnänderung selbst kann nicht instantan erfolgen. Deshalb müssen wir realistisch eine Arbeitsfrequenz von 10 Herz oder höher voraussetzen um später auch anspruchsvollere Navigationsaufgaben wie Kollisionsvermeidung, automatische Starts und Landungen durchführen zu können.

Dies muss auf einer Plattform die für On-Board Einsätze geeignet ist erreicht werden. In der vorangegangenen Studienarbeit kam hierzu eine ARM11 CPU zum Einsatz, inzwischen sind aber schnellere und bessere Embedded Lösungen sowohl auf ARM als auch auf X86 Architektur Basis verfügbar. So bietet etwa Kontron[Kon12] Kompakte Embedded-Boards mit Mehrkern Intel CPUs bis zu 2 GHz, wie sie auch die ETH Zürich auf ihren Pixhawk[MTH+12] UAV-Projekten für Computer-Vision-Zwecke einsetzt.

Für die Evaluierung des Algorithmus steht ein Lenovo R61 Laptop mit einem 1.8 GHz Intel Core2 Dual Core CPU zur Verfügung. Dank der aktuellen raschen Entwicklung im Embedded Bereich können wir davon ausgehen, dass für den On-Board Einsatz hinreichend gleichwertige Systeme zur Verfügung stehen oder in absehbarer Zeit stehen werden. Die Performance auf dem Offline Test System können wir also auch bezüglich Rechengeschwindigkeit als hinreichend repräsentativ für den Einsatz auf einem Live System ansehen. Siehe Abbildung 4.4.

Abbildung 4.4: Laptop für die SLAM Evaluierung. Lenovo R61.

4.3.3 Große offene Karte

Da ein UAV außerhalb geschlossener Gebäude operiert, muss der Algorithmus in der Lage sein sehr viele Objekte in einem sehr großen Skalenbereich zu verwalten. Die visuell erkennbaren Strukturen reichen von Objekten im Zentimeterbereich im Nahumfeld des UAV über für die unmittelbare Kollisionsvermeidung relevanten Objekte im Umfeld von 10 bis 50 Meter, für die Navigation relevante Landmarken in einer Distanz von mehreren hundert Metern bis wenigen Kilometern, bis zu Wolken und fernen Strukturen am Horizont, die viele Kilometer entfernt sein können, und sogar stellare Objekte, wie Sonne, Mond und Fixsterne, deren Distanz gegenüber der Kamera unmessbar groß ist.

Kapitel 5

Entwicklung & Analyse von SLAM Ansätzen

Um die Performance und das Verhalten eines SLAM Algorithmus überprüfen zu können müssen wir diesen im Rahmen des unter 3 vorgestellten Frameworks implementieren. Zuerst versuchen wir hierbei einen Ansatz zu finden, aus einer Serie von Kamerabildern Tiefeninformationen zu extrahieren, da diese uns eine größere Zahl von unter 4 vorgestellten Algorithmen erschließen würden.

5.1 Bestimmung von Tiefeninformationen aus Optischem Fluss

Ohne eine zweite Kamera wie in [SKSZ12], die über räumliche Parallaxe Tiefeninformationen liefern würde, bleibt bei nur einer Bildquelle nur der Vergleich zweier zeitlich aufeinanderfolgenden Einzelbilder. Auch diese weisen bei geeigneten Bewegungen einen räumlichen Versatz auf, aber auch Verdrehungen.

Dieser räumliche Versatzvektor pro Pixel und Zeiteinheit wird als optischer Fluss bezeichnet. Aus diesem lässt sich prinzipiell die Raumtiefe für jeden Pixel berechnen, sofern die Bewegung im Raum bekannt ist[Pra80]. Umgekehrt lässt sich die Bewegung im Raum aus dem optischen Fluss von Pixeln bekannter Entfernung berechnen.

Dieses Problem ist prinzipiell rückführbar auf das SLAM Problem selbst. Zur Veranschaulichung genügt es, wenn wir an dieser Stelle die jeweiligen Pixel als „Landmarken“ betrachten.

5.1.1 Bestimmen des optischen Flusses

Bevor wir jedoch an die Implementierung eines SLAM Verfahrens basierend auf dem optischen Fluss gehen können benötigen wir zunächst eine Möglichkeit den optischen Fluss zu bestimmen.

5.1.2 Verfahren von OpenCV

OpenCV bringt zwei Funktionen zur Bestimmung des optischen Flusses mit. Dies wäre einmal calcOpticalFlowPyrLK() die sich des Iterativen Lucas-Kanade Verfahrens mit Hilfe von Bildpyramiden[Bou01] bedient. Des weiteren implementiert OpenCV das Verfahren zur Bestimmung des globalen optischen Fluss nach Gunnar Farneback[Far03].

Ein Test dieser Verfahrens ergab allerdings keine zufriedenstellenden Ergebnisse. Beide Verfahren arbeiten auf Monochrombildern. Die von dem Kamera gelieferten bzw. auf Video aufgezeichneten Daten sind jedoch Bilder im RGB Farbraum. Um die Verfahren anzuwenden müssen diese also zuerst konvertiert werden, was wertvolle Prozessorlaufzeit mit Aufwand O(x*y) in Anspruch nimmt (wobei x und y die Bildauflösung in x und y Richtung darstellt, im Fall der aufgezeichneten Videos also 640*480).

Die aufgezeichneten Videos zeigen Felder in Grün- und Brauntönen, sowie einen grau wolkenverhangenen Himmel. Nach der Konvertierung ins Monochrome sind diese so kontrastarm, dass keiner der von OpenCV

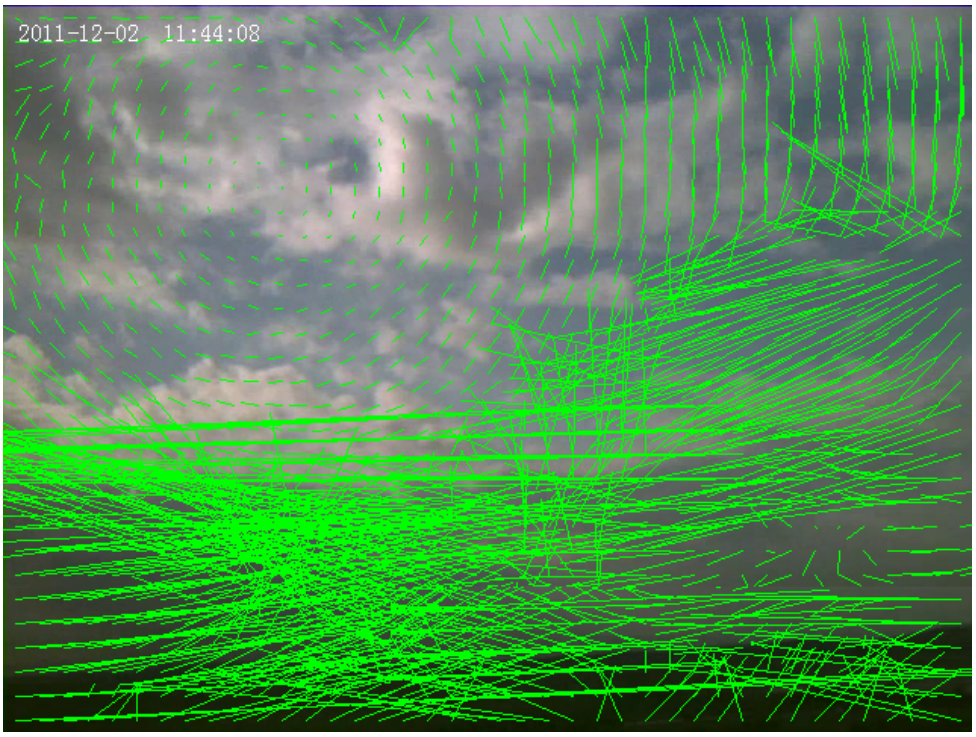

Abbildung 5.1: Optische Flussbestimmung nach Lucas-Kanade mit Bildpyramiden auf dem Flugvideo. Einer der seltenen Momente mit teilweise korrekter Flussbestimmung.

zur Verfügung gestellten Algorithmen noch brauchbare Werte zum optischen Fluss liefert. Dies wird in Abbildungen 5.1,5.2 und 5.3 sichtbar, in denen der von diesen Verfahren ermittelte optische Fluss sichtbar gemacht ist.

Gleichzeitig ist die Laufzeit beider Methoden zu hoch. Auf dem für die Implementierung herangezogenen Intel Core2 CPU mit 1800MHz war die Berechnung des Flusses bei 18 fps mit beiden Implementierungen bereits nicht mehr in Echtzeit möglich. Angesichts des gesteckten Ziels, prinzipiell SLAM auf einer langsameren CPU an Bord durchführen zu können, konnten diese Algorithmen also nicht verwendet werden.

5.1.3 Optische Flussermittlung per Template Matching

Für einen einzelnen Pixel ist der optische Fluss nicht berechenbar[BFB94]. Wir müssen stets die Umgebung eines Pixels mit in die Berechnung einbeziehen. Hierbei müssen aber zwangsweise Annahmen getroffen werden. Die einfachste Annahme wäre, dass der gesamte beobachtete Bildausschnitt den selben optischen Fluss aufweist. Da zwei aufeinanderfolgende Kamerabilder nebst Translationen auch Verdrehungen aufweisen können, wäre diese Annahme aber für Bildregionen die nahe des Rotationszentrums liegen sicher zu kurz gegriffen.

5.1.3.1 Näherung des Optischen Flusses als Rotation plus Translation

Wenn wir jedoch stattdessen von der naheliegenderen Annahme ausgehen, dass hinreichend nahe beieinanderliegende Pixel eine hinreichend identischer Entfernung zur Kamera aufweisen, können wir den optischen Fluss in einem Bildbereich annähernd beschreiben durch eine Verschiebung plus Verdrehung, also den Tupel:

$$\begin{Bmatrix} Translation\,in\,X\,Richtung \\ Translation\,in\,Y\,Richtung \\ Drehung\,im\,Uhrzeigersinn \end{Bmatrix} = \begin{Bmatrix} \hat{x} \\ \hat{y} \\ \alpha \end{Bmatrix}$$

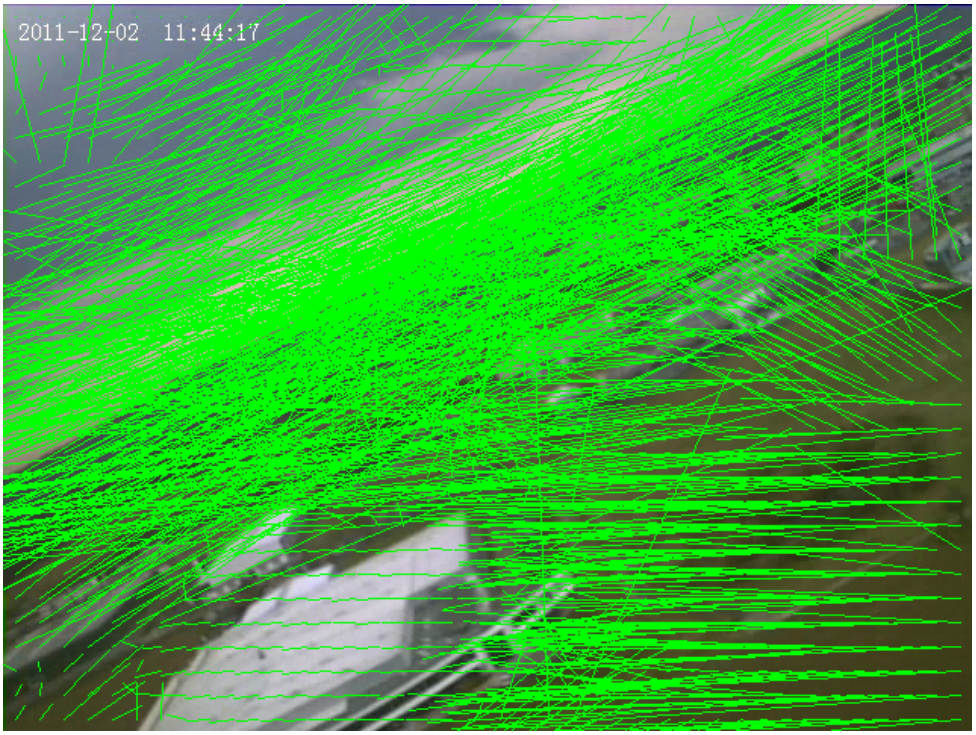

Abbildung 5.2: Optische Flussbestimmung nach Lucas-Kanade mit Bildpyramiden auf dem Flugvideo. Nur an wenigen Stellen mit starkem Kontrast ist der Fluss auch nur annähernd korrekt.

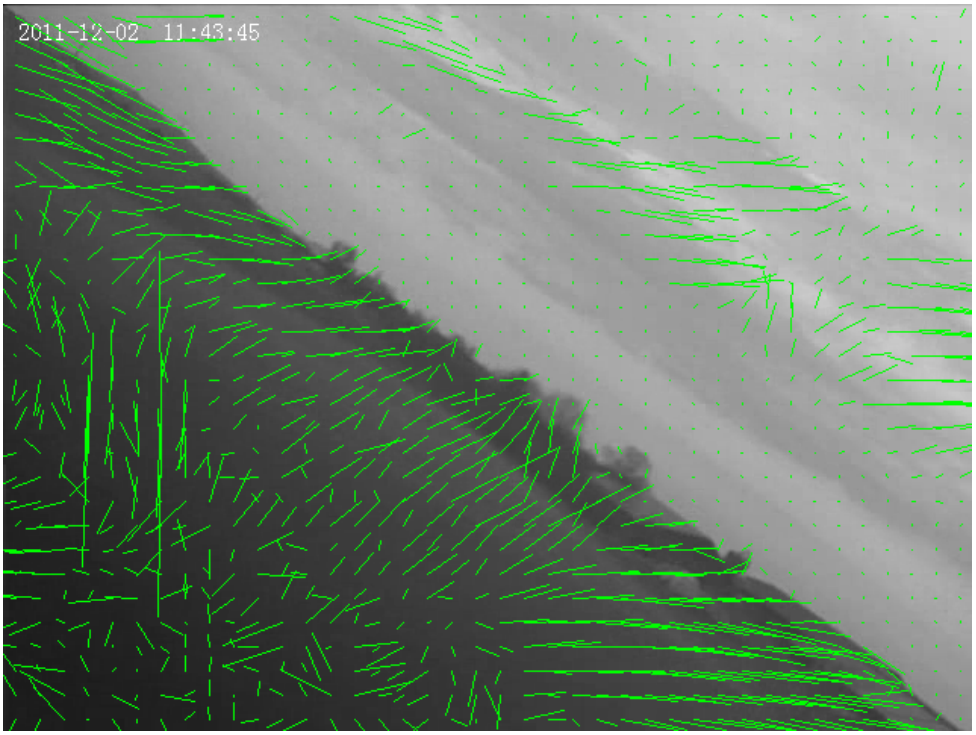

Abbildung 5.3: Optische Flussbestimmung nach Gunnar Farneback auf dem Flugvideo. Dieses Verfahren ist nicht nur zu langsam sondern auch völlig ungeeignet auf dieser Art von Material den optischen Fluss zu berechnen.

Offensichtlich muss hierbei festgelegt werden ob die Drehung vor oder nach der Translation durchgeführt wird, da diese Operationen nicht assoziativ sind, auch wenn sich beide Varianten ineinander überführen lassen. Wir entscheiden uns dabei für die Ausführung der Translation vor der Rotation.

Dieser näherungsweise Fluss kann auch in Matrixform dargestellt werden, und zwar als Matrix R wobei

$$\begin{Bmatrix} \tilde{x} \\ \tilde{y} \end{Bmatrix} = R * \begin{Bmatrix} x \\ y \\ 1 \end{Bmatrix} = \begin{Bmatrix} cos(\alpha) & sin(\alpha) & \hat{x} \\ -sin(\alpha) & cos(\alpha) & \hat{y} \end{Bmatrix} * \begin{Bmatrix} x \\ y \\ 1 \end{Bmatrix}$$

Die dabei betrachtete Umgebung beträgt minimal die unmittelbare Umgebung eines Pixel und maximal das gesamte Bild. Die Annahme, dass die Tiefe der betrachteten Pixel näherungsweise identisch ist, ist bei größeren Bildausschnitten im Allgemeinen natürlich nicht mehr haltbar.

Wenn wir nun zwei Bildausschnitte aufeinanderfolgender Kamerabilder vergleichen, erwarten wir eine Übereinstimmung, wenn der Bildausschnitt des ersten Bildes entsprechend des angenommenen entsprechend der obigen Annahme genäherten optischen Flusses gedreht und verschoben wird.

5.1.3.2 Bewertung der Näherung über Template Matching

Es sollte also möglich sein über die Bestimmung des Unterschieds des rotierten Ausschnitts des ersten Bildes mit dem Referenzausschnitt des zweiten Bildes ein Fehlermaß für den angenommenen optischen Fluss zu finden.

Wir benötigen also ein schnelles Vergleichsverfahren, das problemlos auf RGB Bildern arbeitet oder auf diese erweitert werden kann. Ein bekanntermaßen schnelles Verfahren um Bilder zu vergleichen ist Template Matching[Bru09]. Hierbei wird aus zwei Bildern schlicht der quadratische Fehler aller Pixel summiert. Sind zwei Bilder identisch, ist die Fehlersumme null. Jede zusätzliche Abweichung lässt ihn ansteigen. Dies funktioniert auch mit Mehrfarbbildern, da der quadratische Abstand sich in beliebig vielen Dimensionen einfach berechnen lässt. Sei $C_b(x, y, c)$ der Wert eines Pixels in Bild b an Position x,y im Farbkanal c, so ist der quadratische Pixel-Fehler für einen RGB Pixel an der Position x,y

$$PixelFehler^2 = (C_1(x, y, \text{"rot"}) - C_2(x, y, \text{"rot"}))^2 + (C_1(x, y, \text{"grün"}) - C_2(x, y, \text{"grün"}))^2 + (C_1(x, y, \text{"blau"}) - C_2(x, y, \text{"blau"}))^2$$

und damit der quadratische Template Abstand allgemein:

$$Abstand^2 = \sum_{x,y,c} (C_1(x, y, c) - C_2(x, y, c))^2$$

Unter der Annahme, dass sich der optische Fluss im untersuchten Bildbereich tatsächlich entsprechend der unter 5.1.3.1 vorgestellten Rotation und Translation beschreiben lässt, können wir auch annehmen, dass dieses Fehlermaß für die bestmögliche Näherung des optischen Flusses minimal ist.

Die Annahme, dass der Fehler für alle anderen möglichen Flussnäherungen größer ist als für die korrekte, ist dagegen offensichtlich nicht allgemein gegeben. Das einfachste Gegenbeispiel wären zwei einfarbig schwarze Bilder, etwa bei Aufnahmen mit geschlossener Verschlusskappe. Auf diesen wäre offensichtlich kein optischer Fluss bestimmbar, aber das Beispiel veranschaulicht, dass wir uns mit allgemeingültigen Aussagen auf Bildern sehr schwer tun.

Dennoch sind plausible Aussagen über die Qualität dieser Näherung möglich:

Gibt es einzelne Pixel im Ausgangsbild für die der angenommene Fluss nicht zutrifft, dann finden sich diese an einer anderen Stelle im Referenzbild. Diese örtliche Substitution erhöht potentiell das Fehlermaß. Der Anstieg ist nicht generell bestimmbar, da dieser von der farblichen Abweichung zur Umgebung an der jeweils korrekten und falschen Position abhängig ist.

Allerdings kann selbst unter ungünstigsten Bedingungen ein von einem ansonsten korrekt angenäherten Fluss abweichender Pixel das Fehlermaß nur um den maximalen Fehler von 2 Pixeln ansteigen lassen, wohingegen der maximale Fehler einer falschen Flussannahme sehr viel größer ist, da diese unter Umständen alle Pixel des untersuchten Bildausschnitts betreffen würde.

Zufällige Änderungen an einzelnen Pixeln des Referenzausschnitts, etwa durch Rauschen, Kompressionsartefakte, oder sonstige Änderungen an Objekten im Bild betreffen dagegen nur die betroffenen Pixel selbst und tragen dementsprechend deutlich geringer zum Gesamtfehler bei. Außerdem erhöhen diese das

Fehlermaß gleichermaßen für korrekte und falsche Flussannahmen, daher ändern sie den Charakter des Minima nicht. Dies gilt auch für Änderungen die das gesamte Bild betreffen, wie eine Beleuchtungsänderung oder Verschlusszeitänderungen, die einen Fehlerwert für alle Pixel generieren.

Wir können also auch plausibel annehmen, dass das globale Fehlerminimum den besten Näherungswert für den untersuchten Bildausschnitt, bzw. die Mehrzahl der darin enthaltenen Pixel darstellt.

5.1.3.3 Interpolation auf Sub-Pixel Ebene

Sowohl Rotationen als auch Translationen über reelle Werte sorgen für eine Situation, in der es keine 1 zu 1 Beziehung zwischen Pixeln im Ausgangsbild und im rotierten und verschobenen Bild gibt. Vielmehr wird einem Pixel im verschobenen Bild eine Koordinate im Ausgangsbild zugeordnet, die zwischen Pixeln liegt.

Dementsprechend muss für die Bestimmung des Farbwerts an dieser Stelle der Ausgangswert interpoliert werden. Dieses Problem lösen wir auf traditionelle Weise durch lineare Interpolation, da sich der Rechenaufwand für diese Interpolationsfunktion in Grenzen hält.

5.1.3.4 Bestimmbarkeit des optischen Flusses

Ein Algorithmus der das Minimum der Template-Differenz bestimmt, wäre also prinzipiell zur Bestimmung des optischen Flusses geeignet, wobei diese Methode bedingt durch die gemachten Annahmen nur heuristischen Charakter hätte und nicht allgemeingültig ist. Wir listen die erforderlichen Annahmen.

Der optische Fluss kann mittels Template Matching bestimmt werden wenn:

- Sich der optische Fluss innerhalb des untersuchten Abschnitts mittels einer parametrisierten Funktion, in unserem Fall Rotation und Translation, annähern lässt. Wobei natürlich auch kompliziertere Näherungsfunktionen denkbar sind.
- Der Fehler, der durch die Summe aller Pixel mit abweichendem Fluss erzeugt wird, kleiner ist als der Fehler der durch falsche Flussannahmen erzeugt wird.
- Der Fehler, der durch Rauschen oder sonstige Änderungen die nicht flussbedingt sind erzeugt ist, invariant gegenüber der Flussannahme ist und das Minimum nicht verschiebt.
- Der Fluss überhaupt eindeutig bestimmbar ist, es also ein eindeutiges globales Fehlerminimum gibt. (Siehe das in Beispiel von der geschlossenen Verschlusskappe in 5.1.3.2)

5.1.3.5 Hierarchische Flussbestimmung

Der Fluss im Kamerabild wird, insbesondere wenn die sichtbaren Objekte weit entfernt und die Translation der Kamera im 3D Raum klein gegenüber dieser Entfernung ist, primär von Drehungen der Kamera bestimmt. (Hiervon können wir bei Aufnahmen im Außenbereich, insbesondere aus der Luft, ausgehen). Die Drehung der Kamera erzeugt wiederum ein optisches Flussfeld über das Bild, das sich über die in 5.1.3.1 angegebene Näherung aus Translation und Rotation hinreichend bestimmen lässt und von der Entfernung der sichtbaren Objekte unabhängig ist. Siehe Abbildung 5.4.

Von Translationsbewegungen erzeugter optischer Fluss erzeugt wiederum lokale Abweichungen von diesem globalen Flussfeld, die verglichen mit diesem klein sind. Lokale Oberflächenstrukturen die sich nur unwesentlich vom Hintergrundobjekten abheben, wie etwa Bäume auf einer Ebene, erzeugen wiederum lokale Abweichungen im Flussfeld, die wiederum im Verhältnis zu denen größerer Strukturen klein sind. Siehe Abbildung 5.5.

Dies legt Nahe, den Fluss nicht in einem Schritt für jeden Bildpunkt zu bestimmen, sondern, hierarchisch vorzugehen. Wir bestimmen zunächst die gemittelte Flusshypothese für das gesamte Bild, errechnen daraus den erwarteten Fluss für Teilbilder und benutzen diesen als Startwert für die Flussermittlung für immer kleinere Ausschnitte.

Dies gelingt am besten durch Erzeugung einer Gauß-Pyramide, wie sie auch im Verfahren von Bouguet bei der Anwendung des Lucas-Kanade Trackers verwendet wird[Jäh05, Bou01], siehe Abbildung 5.6. Diese

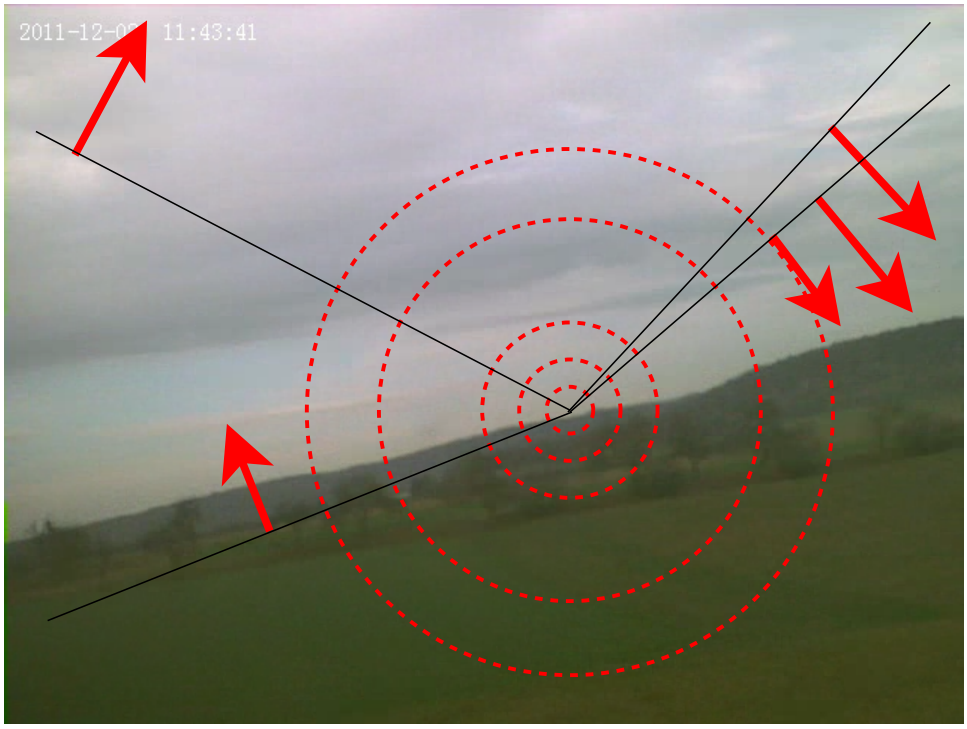

Abbildung 5.4: Die Rotation der Kamera bestimmt in der Regel den dominierenden Anteil des optischen Flusses. Dieses Flussfeld ist homogen, unabhängig vom Bildinhalt oder Objekten, oder deren Distanz und kann global bestimmt werden. Die Bestimmung dieses Feldes ergibt bereits eine gute Näherung für jeden einzelnen Pixel.

liefert verkleinerte Teilbilder, die für die Bestimmung des gemittelten Flusses verwendet werden können. Die damit verbundene Weichzeichnung verändert jedoch die Eigenschaften der Bildausschnitte in einer für die Flussbestimmung per Template Matching relevanten Art.

Da das Bild geglättet wird, werden Übergänge zwischen Pixeln weicher, was sich auf den Fehler beim Template Matching auswirkt. So sind die Farbwerte nahe beieinanderliegender Pixel grundsätzlich ähnlich, was folgende Auswirkungen hat:

- Kleine Abweichungen zwischen dem untersuchten optischen Fluss und dem Optimum erzeugen auch nur kleine zusätzliche Fehler.
- Es gibt damit einen Fehler-Gradienten, der auf das Optimum hinzeigt.
- Kleine Abweichungen einzelner Pixel von der Näherung erhöhen den Fehler wegen der ähnlichen Farbwerte ebenfalls nur geringfügig.
- Abweichungen durch Rauschen und ihr Einfluss werden reduziert.

Es stellt sich Zwangsläufig die Frage der optimalen Größe eines zu testenden Bildausschnittes. Dieser sollte so klein wie möglich sein um den Rechenaufwand gering zu halten, gleichzeitig aber groß genug um eindeutige Ergebnisse liefern zu können.

Wir wählen eine Ausschnittsgröße von 20x15 Pixeln, da dieses Ausschnittsfenster den kleinsten gemeinsamen ganzzahligen Teiler der Bildgröße von 640 auf 480 Bildpunkten darstellt und sich so durch 5-maliges Verkleinern mittels einer binominalen Faltungsmaske[MKG+97, SKRS11, Jäh05] $\frac{1}{32}$ der Originalgröße einstellt.

5.1.3.6 Suche des Optimums

Über den Template Vergleich selbst ist es zunächst nur möglich zwei Hypothesen für den optischen Fluss zu vergleichen, jedoch können wir damit das Optimum noch nicht finden.

Abbildung 5.5: Die Bewegung der Kamera erzeugt in Näherung Optischen Fluss relativ zu einem Fluchtpunkt. Dieser Flussanteil ist in der Regel sehr viel kleiner als der durch Rotation verursachte Fluss. Die Stärke des Flusses ist hierbei von der Distanz der Objekte zur Kamera und dem Winkel gegenüber der Bewegungsachse abhängig. Dieses Flussfeld ist mit Einschränkungen ebenfalls homogen, es sei denn, nahe Objekte verdecken einen weit entfernten Hintergrund. Die Bestimmung des Flusses in einer Region ergibt also wiederum eine gute Näherung für den Fluss in Teilregionen.

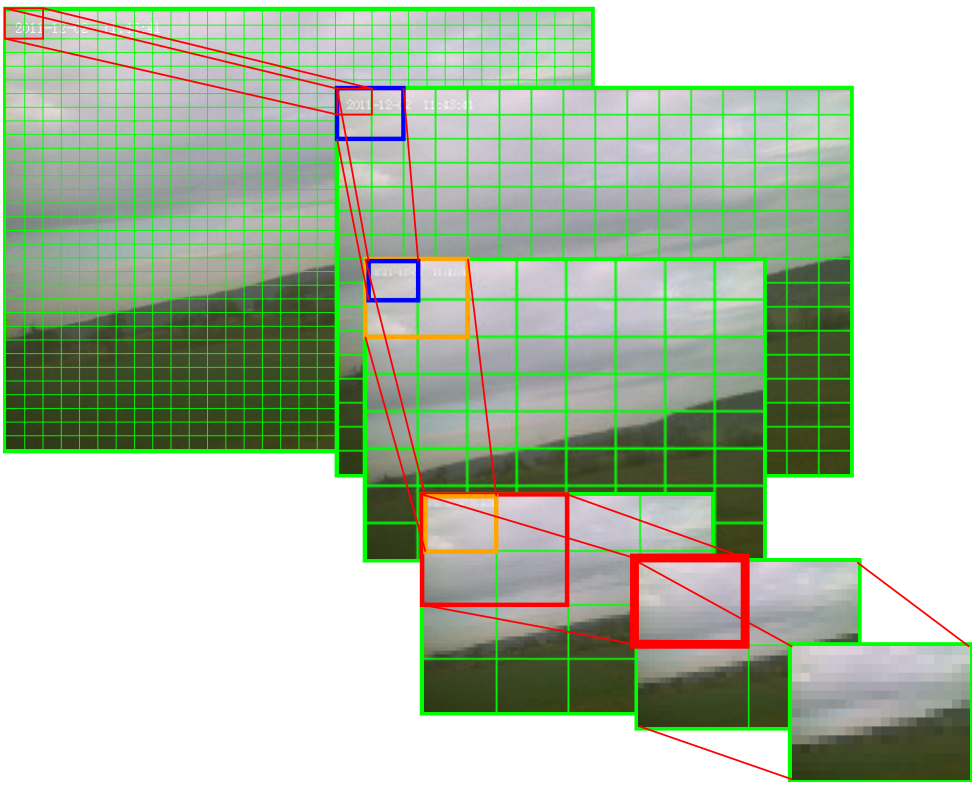

Abbildung 5.6: Bildpyramide zur Bestimmung des optischen Flusses im Template Matching Verfahren.

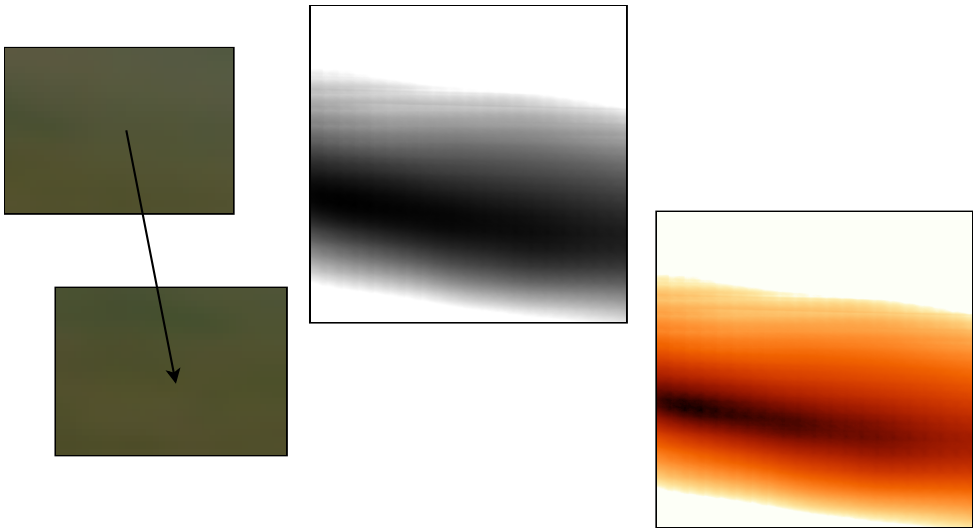

Abbildung 5.7: Gradient des Templateunterschieds zwischen zwei Frame-Ausschnitten in Abhängigkeit vom x,y Versatz, mit und ohne Hervorhebung des Minimums durch Falschfarben.

Im Rahmen diese Arbeit wurden mehrere Verfahren implementiert, getestet und optimiert um das Optimum mittels mehrerer Vergleiche zu finden.

5.1.3.6.1 Partikel Filter Ansatz Bei diesem Ansatz wird eine größere Anzahl Hypothesen für den Fluss aufgestellt, die zufällig verteilt um einen Wahrscheinlichen Anfangswert für diesen sind. Aus diesen wird wiederum die wahrscheinlichste (also die mit dem geringsten Fehler) ausgewählt und in einem kleineren Umfeld um diese Werte erneut gesucht. Dieses Verfahren führt zum Ziel, die optimale Abschätzung des optischen Flusses kann also auf diese Art und Weise ermittelt werden. Jedoch sind dazu für eine zuverlässige Flussbestimmung in etwa 200 Template Vergleiche pro Bildausschnitt erforderlich, bei geringeren Anzahlen wird die Abschätzung für den optischen Fluss signifikant fehleranfälliger.

Dieses Verfahren ist leider viel zu langsam um den optischen Fluss für alle Pixel, oder auch nur die gesamte Gaußsche Pyramide bis hinab auf die 20x15 Pixel großen Ausschnitte des Bildes in Originalgröße, in Echtzeit zu berechnen.

Der Quellcode dieses Verfahrens befindet sich in zwei Varianten in den Methode temperMatch() und particleMatch() in der Klasse CCFlow in der Datei flight/Modules/SLAM/ccflow.cpp in Anhang A.1.

5.1.3.6.2 Vollständige Analyse Um den Zustandsraum des Template Matching Fehlers zumindest zweidimensional zu visualisieren, wurde für einzelne Ausschnitte der Fehler in Abhängigkeit vom Versatz in x und y Richtung sowie in eine der Versatzrichtungen und vom Winkel dargestellt.

Die Visualisierung in Abbildungen 5.7 und 5.8 belegt, dass die getroffenen Annahmen mit einigen implementierungsspezifischen Einschränkungen richtig waren.

- Der Fehler weist einen deutlichen Gradienten auf, welcher auf das oder die Maxima zeigt.
- Form und Ausprägung des Maximums hängen stark von den sichtbaren Features im Bild ab und können stark variieren. Es kann auch lokale Maxima geben, wenn der Bildausschnitt etwa sich wiederholende Features aufweist, die mehrmals "passen".
- Der Gradient ist nicht homogen sondern weist sichtbare Sprungstellen auf. Vor und nach der Sprungstelle verläuft der Gradient jeweils in die selbe Richtung. Dies ist der Implementierung geschuldet, aller Wahrscheinlichkeit nach dem verwendeten linearen Interpolationsverfahren für Sub-Pixel Werte, da dieses ebenfalls Inhomogenitäten an Pixelgrenzen aufweist.

Darüber hinaus gibt es noch größere Sprungstellen am Bildrand, die dann auftreten wenn wegen des Versatzes auf Pixel zurückgegriffen werden muss, die außerhalb des Bildes liegen. Bildstellen bei denen

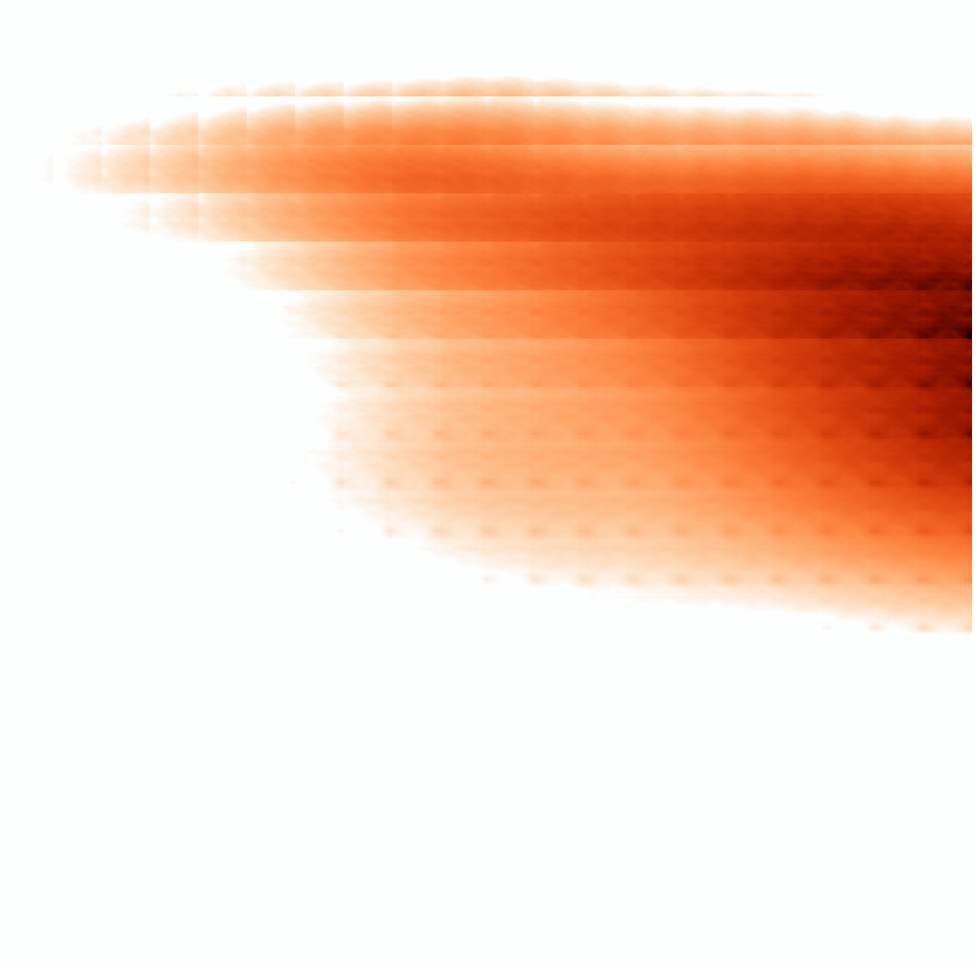

Abbildung 5.8: Ein weiterer Templateabstandsgradient mit deutlicheren Inhomogenitäten.

dies der Fall ist können nicht einfach ignoriert werden oder als identisch angesehen werden, da dies etwa den Gesamtfehler für komplett aus dem Bild verschobene Bildausschnitte auf null verringern würde. Auch die Annahme eines maximalen Fehlers ist nicht sinnvoll, da dies wiederum teilweise aus dem Bild verschobene Bereiche über Gebühr benachteiligen würde. Stattdessen erhöhen wir der Fehler um einen vergleichsweise geringen konstanten Betrag pro Pixel, der in etwa dem Durchschnittsfehler entspricht. Dies behebt das Problem der Inhomogenität nicht, behält aber zumindest in den meisten Fällen die Position des globalen Maximums bei.

Dieses Verfahren ist offensichtlich nicht zum Praxiseinsatz geeignet, hilft aber durch die Visualisierung des Problems, eine elegante Lösung zu finden.

Der Quellcode dieses Verfahrens befindet sich in der Methode fullMatch() in der Klasse CCFlow in der Datei flight/Modules/SLAM/ccflow.cpp in .

5.1.3.6.3 Gradientenabstiegsverfahren Die Beobachtungen in 5.1.3.6.2 legen nahe, die optimale Flussnäherung in einem Gradientenabstiegsverfahren zu ermitteln, vergleichbar gängiger Ansätze in der Videokompression[LF96]. Um den Gradienten an einer Flusshypothese $\phi = \begin{Bmatrix} \hat{x} \\ \hat{y} \\ \alpha \end{Bmatrix}$ zu ermitteln wird hierbei ein Templatevergleich für die Hypothesen ψ_k mit $k = [1-3]$durchgeführt in einem minimalen Abstand Δ_k.

$$\psi_1 = \begin{Bmatrix} \hat{x} + \Delta_1 \\ \hat{y} \\ \alpha \end{Bmatrix}$$

$$\psi_2 = \begin{Bmatrix} \hat{x} \\ \hat{y} + \Delta_2 \\ \alpha \end{Bmatrix}$$

$$\psi_3 = \begin{Bmatrix} \hat{x} \\ \hat{y} \\ \alpha + \Delta_3 \end{Bmatrix}$$

Δ muss so klein wie möglich gewählt werden, so das der Gradient innerhalb des Vektors $\begin{Bmatrix} \Delta_1 \\ \Delta_2 \\ \Delta_3 \end{Bmatrix}$ als konstant angenommen werden kann. Ein unteres Limit stellt jedoch die Genauigkeit der für die Implementierung benutzten Variablentypen dar. Zu kleine Schritte führen zu Verfälschungen bedingt durch die begrenzte Genauigkeit. Für die lineare Interpolation der Werte zwischen Pixeln wurde aus Performance-Gründen eine Festkommaarithmetik gewählt mit 5 Bit Nachkommastellen. Die kleinste signifikante Einheit ist also $\frac{1}{2^5} = \frac{1}{32}$ Pixel, also $\Delta_1 = \Delta_2 = \frac{1}{32}$. Dies stellt damit auch die größte erreichbare Genauigkeit für die Flusshypothese dar, die sich jedoch bei Bedarf durch eine Anpassung der Bitbreite verfeinern ließe.

Die Bestimmung von geeigneten Werten für Δ_3 ist schwieriger, da sich eine Rotation auf nahe am Zentrum gelegene Pixel kaum auswirkt, auf weit davon entfernte jedoch um so mehr. Mit einem Wert für Δ_3 von $\frac{1}{8}$ Grad wurden jedoch brauchbare Ergebnisse erzielt.

Sei $E(\phi) = E(\begin{Bmatrix} \hat{x} \\ \hat{y} \\ \alpha \end{Bmatrix})$ der ermittelte Templateabstand für die Flusshypothese ϕ, so liefert

$$G = \begin{Bmatrix} E(\psi_1) - E(\phi) \\ E(\psi_2) - E(\phi) \\ E(\psi_3) - E(\phi) \end{Bmatrix} \approx \frac{\delta E(\phi)}{\delta \phi}$$

den zugehörigen Gradienten für die Flusshypothese ϕ, eine Näherung deren Ableitung.

Um diese zu ermitteln sind 4 Template-Vergleiche erforderlich zur Ermittlung des Abstandes $E(\phi)$ sowie $E(\psi_{[1-3]})$

Intuitiv würden wir nun, nachdem der Gradient $G(\phi)$ ermittelt wurde, eine Neue Hypothese ϕ' wählen, die gegenüber ϕ in Richtung des Vektors $G(\phi)$ um eine Schrittweite s verschoben ist. Hierbei würden

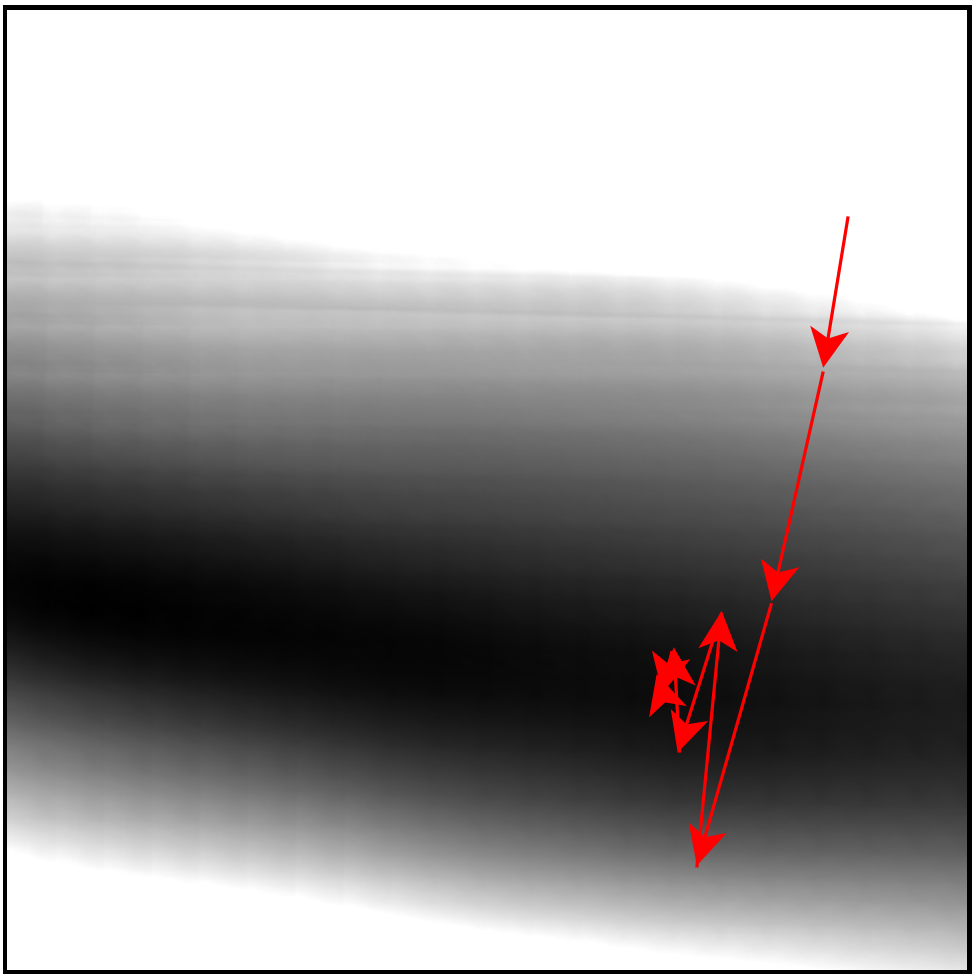

Abbildung 5.9: Trivialer Gradientenabstieg folgt dem Gradienten. Dieses konvergiert bei entsprechend langgestreckten Minimas unter Umständen sehr langsam.

wir s in jedem Schritt entsprechend eines Erfolgskriteriums verringern oder vergrößern, solange bis eine Abbruchbedingung erfüllt ist.

Dies führt auch zum Ziel, allerdings ist dieses Vorgehen nicht optimal.

Generell laufen Gradientenabstiegsverfahren Gefahr statt des globalen Minimas in einem lokalen Minima stecken zu bleiben[FP63]. Dies erwies sich aber nicht als das Hauptproblem des Ansatzes. Vielmehr besteht das Hauptproblem in der oft ungleichen Ausprägung des Gradienten in verschiedenen Dimensionen. Vereinfacht gesagt ist das gesuchte Minimum oft sehr lang gestreckt entlang einer Gerade, wobei der Gradient annähernd orthogonal auf diese Gerade zustrebt, wohingegen die Komponente entlang der Gerade sehr schwach ausgeprägt ist.

Ein Abstiegsverfahren das nur die Richtung des Gradienten an der aktuellen Hypothese betrachtet gerät hier in eine Zickzackbewegung die nicht schnell zum Ziel führt. Mehr noch, da sich der Absolutwert der Hypothese trotz großer zurückgelegter Strecken kaum verändert, endet das Verfahren häufig am Talboden, ohne dem dort durchaus vorhandenen, zum globalen Minimum führenden Gradienten zu folgen. Dieses Verhalten ist aus Abbildung 5.9 ersichtlich.

Dieses Problem kann umgangen werden, indem die normalisierten Gradienten der aktuellen und der vorangegangenen Hypothese vektoriell addiert werden, und die nächste Hypothese in Richtung dieser Vektoraddition gewählt wird, wie in Abbildung 5.10 dargestellt.

Derart führt das Verfahren auch bei Gradienten die über mehrere Dimensionen ungleich verteilt sind zum Ziel. Es muss jedoch noch eine geeignete Abbruchbedingung, sowie ein geeignetes Verfahren zur Reduzierung der Schrittweite gefunden werden.

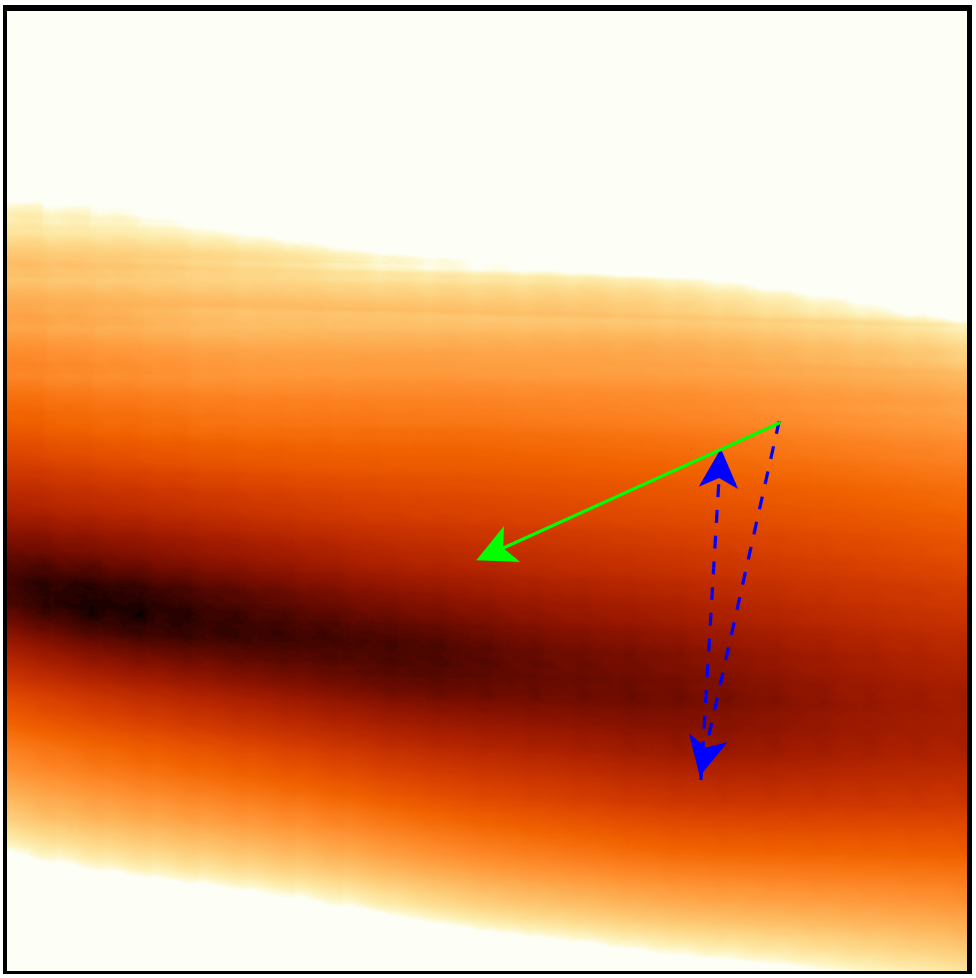

Abbildung 5.10: Ein effizienteres Gradientenabstiegsverfahren zieht die Gradienten der letzten zwei Hypothesen zur Bestimmung des nächsten Schrittes heran.

Hierfür bilden wir das Skalarprodukt aus dem normalisierten aktuellen und vorangegangenen Gradienten. Ist dies negativ, hat sich der Gradient seit der letzten Hypothese um mehr als 90 Grad gedreht, das Minimum wurde also "überschossen". In diesem Fall sollte die Schrittweite reduziert werden, insbesondere dann, wenn sich der Gradient annähernd vollständig gedreht hat, das Skalarprodukt also nahe -1 ist.

Dies wurde gemäß „Algorithmus 5.1“ realisiert.

Algorithmus 5.1 Gradientenabstiegsverfahren. Die Schrittlänge wird reduziert, wenn der Winkel zwischen den letzten zwei Gradienten mehr als 90 Grad beträgt. Die Schrittrichtung bestimmt sich aus der Vektorsumme der letzten zwei Gradienten.

```
...
TransRot current = initial;
TransRot last = current;
TransRot step = initialRange;
Vec4f prev=Vec4f(0,0,0,bestMatchScore);
...
while (step[0]>1./8. && maxcount-->0) {
        ...
        Vec4f grad = correlateGradient(test,reference,current);
        ...
        Vec3f combined = Vec3f(prev[0]+grad[0],prev[1]+grad[1],prev[2]+grad
            [2]);
        ...
        combined *= 1.0/length(combined); // normalize

        float success = (grad[0]*prev[0]+grad[1]*prev[1]+grad[2]*prev[2]);
            // dot product
        if (success<0) step *= 1.0+(2*success/3);

        current[0] = last[0] + step[0] * combined[0];
        current[1] = last[1] + step[1] * combined[1];
        current[2] = last[2] + step[2] * combined[2];

        last=current;
        prev=grad;
}
...
```

Die Schrittweite wird mit einen Faktor $k < 1$ multipliziert, wenn das Skalarprodukt der letzten beiden Gradienten $(grad \cdot prev) < 1$ beträgt. (Eine Schrittweitenvergrößerung findet also nicht statt.)

Die Formel zur Berechnung des Faktors beträgt $1.0 - \left(\frac{2}{3} * -(grad \cdot prev)\right)$ wobei sich das Skalarprodukt zwischen 0 und -1 bewegt. Die Schrittlänge bleibt also maximal gleich und wird minimal auf $\frac{2}{3}$ ihrer Länge verkürzt.

Die Abbruchbedingung wurde gesetzt auf eine Schrittlänge von s $< \frac{1}{8}$ Pixel

Zusätzlich wird bei Erreichen einer maximalen Schrittanzahl abgebrochen, in diesem Fall gilt die Flussnäherung als nicht ermittelbar. Im Fall eines unbrauchbaren Gradienten mit Länge null, oder bei erreichen einer Hypothese deren Übereinstimmung deutlich schlechter ist als die Starthypothese wird das Verfahren mit einer zufälligen Starthypothese neu gestartet, allerdings ohne den Zähler für die maximale Schrittzahl neu zu starten.

Beim Einsatz dieses Gradientenabstiegsverfahren zur Ermittlung des optischen Flusses auf RGB Bildern wird nach 3000 analysierten Einzelbildern die Abbruchbedingung nach mittleren Schrittzahlen entsprechend folgender Tabelle erreicht. Hierbei ist jeweils die mittlere Schrittzahl mit und ohne Einbeziehung von fehlgeschlagenen Gradientenabstiegen angegeben. Ein Abstieg gilt als fehlgeschlagen, wenn die Abbruchbedingung auch nach 50 Schritten nicht erreicht war.

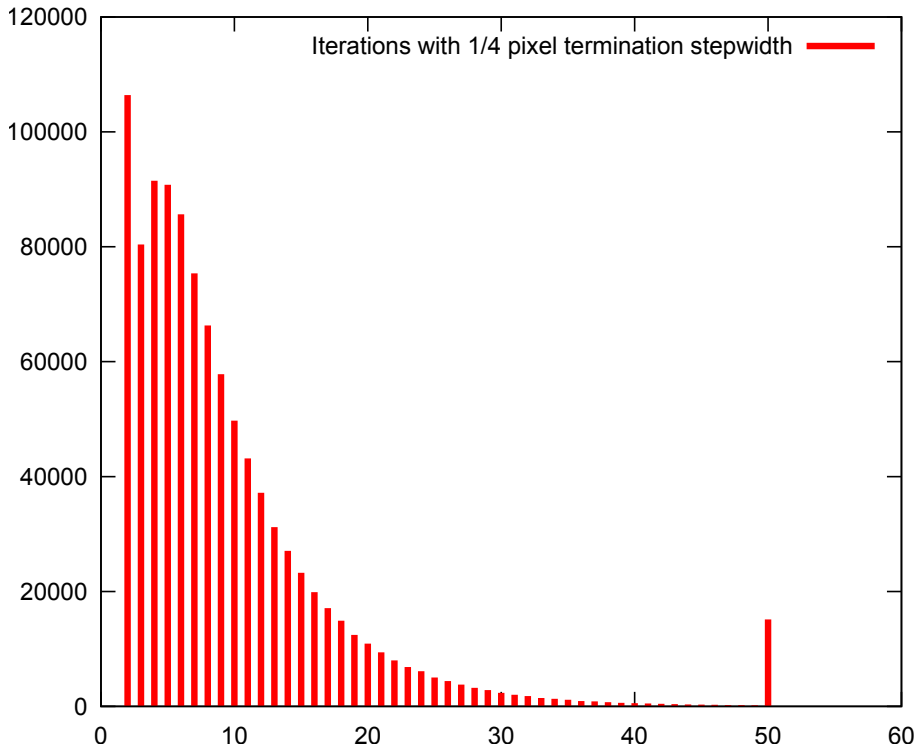

Abbildung 5.11: Verteilung der Konvergierungsdauer in Iterationen für Gradientenabstieg mit einer Abbruchschrittweite von $\frac{1}{4}$Pixel.

Abbruch Schrittweite $\leq x$	Mittlere Schrittzahl	Mittlere Schrittzahl inkl. Fehlschläge
$\frac{1}{4}Pixel$	8,97	9,58
$\frac{1}{8}Pixel$	13,44	14,21
$\frac{1}{16}Pixel$	17,59	18,70

Siehe auch Abbildungen 5.11, 5.12 und 5.13.

Dieser Wert liegt deutlich unter den Werten die mit anderen Verfahren erreichbar sind und liegt auch deutlich unter den vor der Implementierung für möglich gehaltenen Werten.

Das Verfahren ist immun gegenüber den unter 5.1.3.6.2 gefundenen inhomogenen Sprungstellen in der Template-Übereinstimmung, sofern diese nur statistisch selten genug getroffen werden, da ein einzelner Schritt in die falsche Richtung im nächsten Schritt wieder kompensiert wird.

Eine weitere Optimierung scheint an dieser Stelle Dank der sehr vielversprechenden Werte wenig aussichtsreich.

5.1.3.6.4 Erfolgsbestimmung Wie unter 5.1.3.6.2 ausgeführt führt dieses Verfahren nicht in allen Fällen zum Ziel. In homogenen Bildbereichen oder Bildbereichen deren Strukturen in der aktuellen Vergrößerung zu verwaschen sind ist der Fluss nicht ermittelbar. Dennoch kann es ein Minimum im Templateabstand geben, dieses kann aber eine deutlich schlechtere Abschätzung des optischen Flusses sein als ein bereits bestimmter Fluss in der nächst gröberen Pyramidenstufe.

Auch für die Verwendung der Flussabschätzung für einen darauf zu implementierenden SLAM Algorithmus ist es hilfreich einen Anhaltspunkt für die Qualität des ermittelten Flusses zu haben. Ein wahrscheinlich falsch gemessener Fluss kann einen Filter eventuell divergieren lassen und eine ansonsten korrekte Zustandsabschätzung stören. Es wäre daher sinnvoll, diesem eine deutlich geringere Gewichtung zu geben als einer Flussabschätzung die mit hoher Wahrscheinlichkeit korrekt ist.

Wir sind also an einem Erfolgsmaß für das Template Matching - Gradientenabstiegsverfahren interessiert. Ein triviales Kriterium wäre der Templateabstand am ermittelten Minimum. Dieses sagt jedoch noch

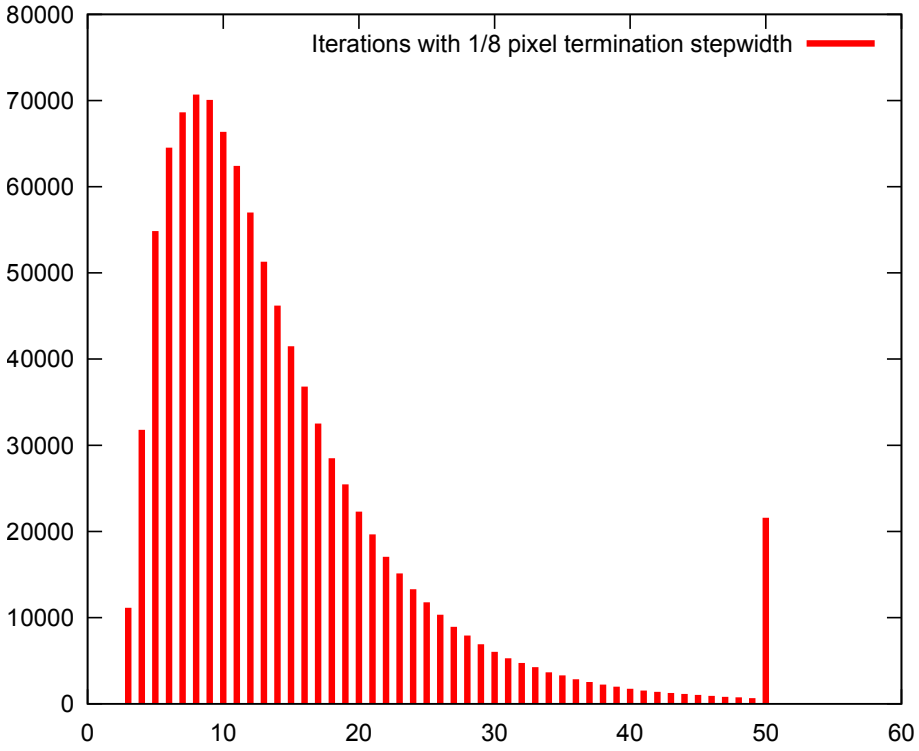

Abbildung 5.12: Verteilung der Konvergierungsdauer in Iterationen für Gradientenabstieg mit einer Abbruchschrittweite von $\frac{1}{8}$Pixel.

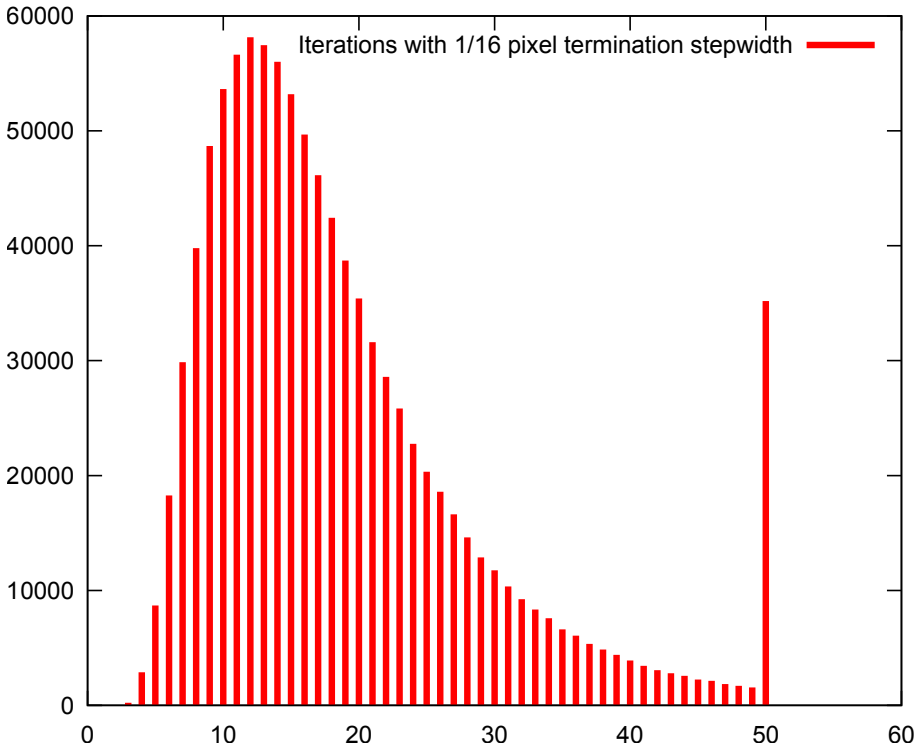

Abbildung 5.13: Verteilung der Konvergierungsdauer in Iterationen für Gradientenabstieg mit einer Abbruchschrittweite von $\frac{1}{16}$Pixel.

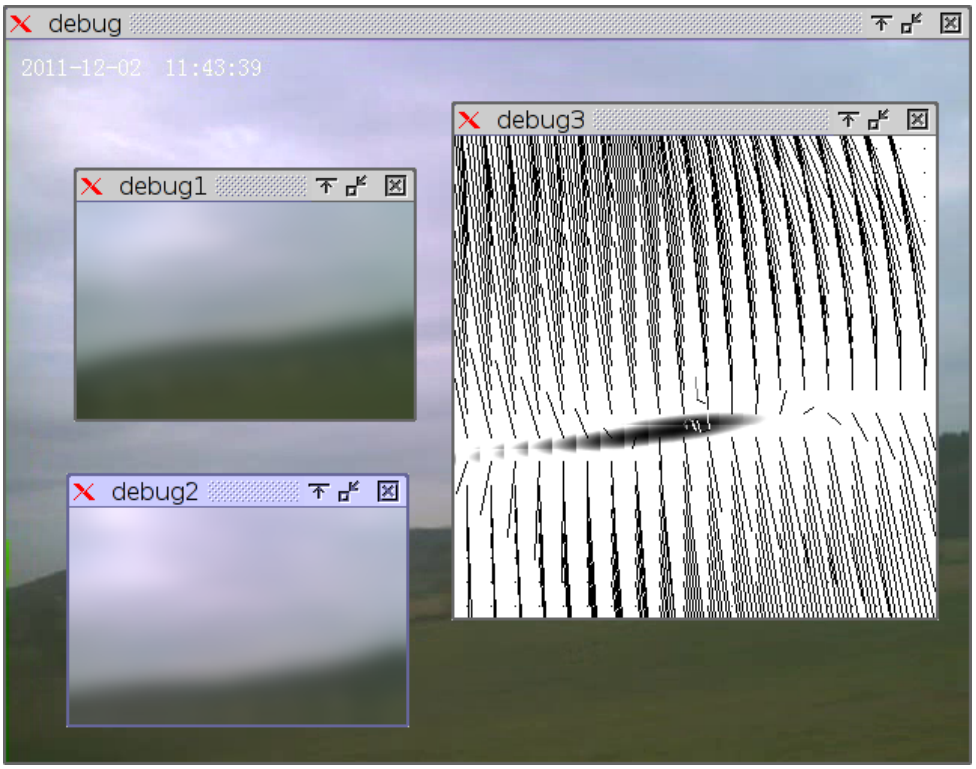

Abbildung 5.14: Visualisierung des Gradientenabstiegs beim Abgleich zweier Bilder, inklusive des Gradienten und des vom Verfahren beim Abstieg zurückgelegten Wegs.

nichts über dessen Eindeutigkeit aus. Aussagekräftiger wäre hier die Tiefe des Minimums gegenüber der Umgebung, oder auch die Steilheit des darauf hinweisenden Gradienten. Je steiler der Gradient im Umkreis des Minimums, desto eindeutiger ist dieses.

Ein mögliches Qualitätskriterium wäre also die Änderung des Abstandes über die zurückgelegte Strecke, also geteilt durch die Länge des beim Gradientenabstieg beschrittenen Weges. Dies erwies sich jedoch, wohl auch bedingt durch die Inhomogenitäten im Gradienten, als nicht sehr zuverlässig.

$$Q_1 = \left| \sum_k \left(\|G(\phi_k)\| \,/\, \|\phi_k - \phi_{(k-1)}\| \right) \right|$$

Nach ausprobieren mehrerer möglicher Erfolgskriterien erwies sich stattdessen die Änderung der Gradientenlänge dividiert durch das Integral der Differenzen der normierten Gradienten als brauchbarere Heuristik:

$$Q_2 = \left| \sum_k \left(\|G(\phi_k)\| - \|G(\phi_{(k-1)})\| \right) / \sum_k \left(\|G(\phi_k) - G(\phi_{(k-1)})\| \right) \right|$$

Diese ist um so größer, je stärker sich die Stärke des Gradienten während des Abstiegs ändert, und um so kleiner, je mehr dieser seine Richtung ändert.

Veranschaulicht liefert diese Heuristik die größten Werte für einen geradlinigen Gradientenabstieg auf ein entferntes Maxima, bei dem der Gradient zunächst steil ist und dann in der Talsohle am Minimum abflacht.

Die kleinsten Werte ergeben sich für Gradientenabstiege, bei denen die Gradientenrichtung stark schwankt, und der Gradient in der Stärke größtenteils gleich bleibt, was wir auch als zielloses Umherirren beschreiben könnten.

Was in diese Heuristik noch nicht einfließt, ist die Qualität der Starthypothese. Liegt die Starthypothese sehr nahe am Minimum, sind die beim traversieren des Gradientenabstiegspfades besuchten Hypothesen nicht aussagekräftig bezüglich der Qualität des Gradienten in größerer Entfernung.

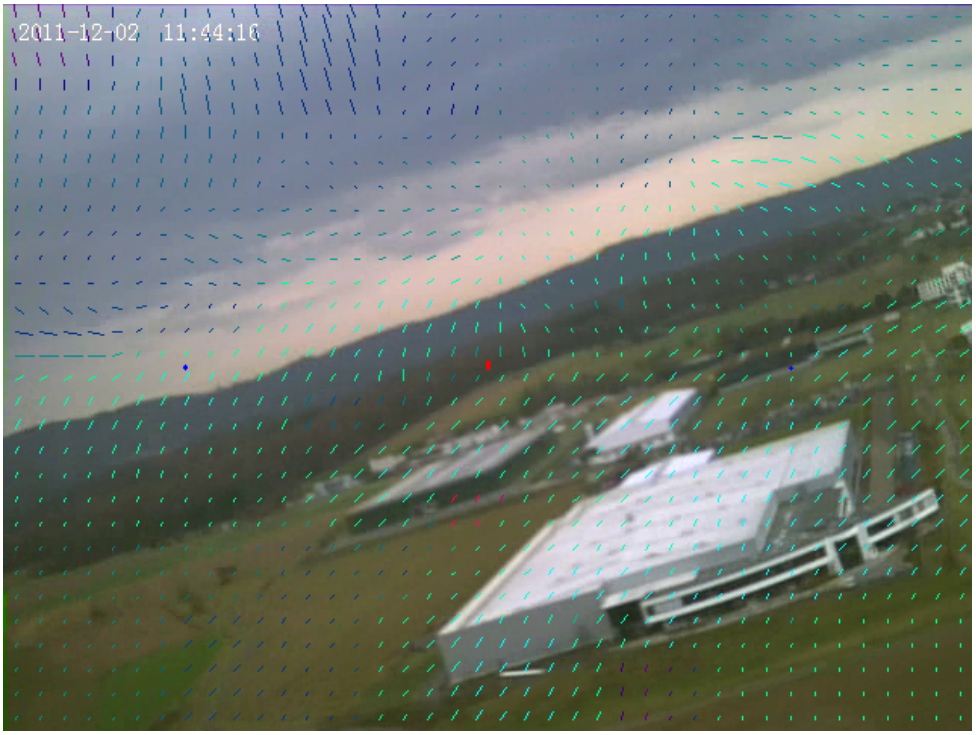

Abbildung 5.15: Mit Template Matching ermittelter optischer Fluss inklusive Qualitätsabschätzung gemäß 5.1.3.6.4. Aussagekräftige Flusshypothesen sind Rot oder Türkis, niedrige Qualitätsmaße blau.

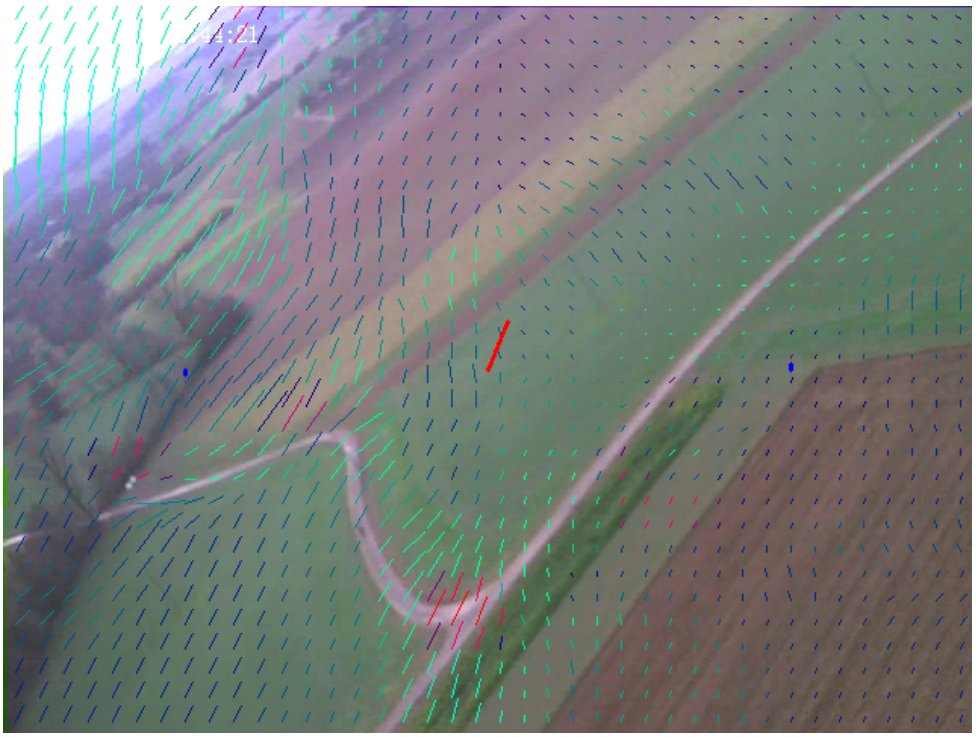

Abbildung 5.16: Mit Template Matching ermittelter optischer Fluss inklusive Qualitätsabschätzung gemäß 5.1.3.6.4. Aussagekräftige Flusshypothesen sind Rot oder Türkis, niedrige Qualitätsmaße blau.

Auch wenn wir mit dieser Heuristik gute Erfolge erzielen können, und damit belegt ist, dass eine Bewertung des Gradienten generell möglich ist, steht eine tiefer gehende Analyse und aussagekräftiges Bewertungskriterium an dieser Stelle noch aus.

5.1.3.7 Bestimmung von 3D Informationen aus dem gewonnenen Fluss

Nachdem wir eine Methode gefunden haben, den optischen Fluss selbst zu ermitteln, versuchen wir im nächsten Schritt diesen zur Bestimmung der Tiefeninformation zu verwenden.

Wir berechnen die Flussnäherung explizit nur für jedes 20x15 Pixel Rechteck im Bild, jedoch können wir aus diesen Daten, die ja insbesondere auch Informationen über die Rotation enthalten, sowie den Flussnäherungen für umliegende Rechtecke den Fluss interpolieren.

5.1.3.7.1 Trivialer Filter Zunächst werden wir dabei überprüfen ob die Qualität des ermittelten optischen Flusses überhaupt ausreicht um brauchbare Informationen über die Tiefe zu erlangen. Hierfür benötigen wir noch keinen allgemeinen Filter, es reicht einen Algorithmus zu erstellen, der den Fluss direkt in ein Tiefenbild umwandelt, unter der Annahme, dass ein größerer Fluss eine geringere Tiefe impliziert.

Da wir zunächst keine Information über Rotations- oder Translationsbewegungen verarbeiten, ist eine absolute Bestimmung der Tiefe nicht möglich, stattdessen wird die Tiefe normiert zwischen 0 und 1 dargestellt, wobei 0 der geringste im Bild gemessene Fluss ist und 1 der größte.

Wird dies in jedem Schritt direkt errechnet, erlangen wir jedoch ein Tiefenbild, das von Rauschen dominiert wird und das nicht sichtbar mit der für einen menschlichen Betrachter ersichtlichen räumlichen Tiefe eines Videos korreliert. Offensichtlich ist der optische Fluss also ohne weitere Filterung zur Tiefenbestimmung ungeeignet.

5.1.3.7.2 Tiefpass Um dem Rauschen Herr zu werden müssen wir filtern. Ein erster Ansatz ist hierbei ein simpler Tiefpass, also eine Durchschnittsbildung über eine längere Zeit.

Das Problem hierbei ist, dass die Tiefe eines Bildpixels nicht konstant ist, vielmehr bewegen sich die durch Pixel repräsentierten Objekte im Bild ja genau gemäß des ermittelten optischen Flusses. Um die Tiefe eines solchen bewegten Pixels also über eine längere Zeit zu beobachten muss dessen Bewegung im 2D Bildraum entsprechend des optischen Flusses berücksichtigt werden.

Um die gefilterte Tiefe eines Pixels zu bestimmen, kombinieren wir daher den aktuellen Flusswert dieses Pixels mit der ermittelte Tiefe eines Pixels aus dem vorangegangenen Bildübergang, dessen ermittelter Fluss wiederum zur Position des aktuellen Pixel zeigt. (Vereinfacht gesagt, die gefilterte Tiefe schwimmt im Fluss mit).

Hierbei ist es uns jetzt auch möglich, den ermittelten Qualitätswert des Flusses aus 5.1.3.6.4 in die Berechnung einfließen zu lassen. Ist der ermittelte Fluss nicht zuverlässig, ändern wir dementsprechend die Zustandshypothese nur wenig.

Die Anwendung dieses Verfahrens ist in Abbildung 5.17 und 5.18 dargestellt.

Der Quellcode findet sich im GIT Branch corvuscorax/opencvslam in Datei flight/Modules/SLAM/opencvslam.cpp in der Methode OpenCVslam::run() (Anhang A.1).

Abbildung 5.17: Optischer Fluss bei seitlicher Kamerabewegung. Aus dem Fluss können Rückschlüsse auf die räumliche Tiefe gewonnen werden. Die Flussermittlung gelang jedoch in einigen homogenen Bildbereichen nicht korrekt.

Abbildung 5.18: Aus dem optischen Fluss in 5.17 ermittelte räumliche Tiefe mit dem Verfahren aus 5.1.3.7.2. Die Tiefenermittlung ist nur relativ der sichtbaren Gegenstände möglich. Nahe Gegenstände sind hell, entfernte Gegenstände sind dunkel. Wie wir sehen wird für Bildbereiche mit homogener Färbung, für die die Flusserkennung fehlschlug, eine zu weite Entfernung angenommen.

Leider gelang die Bestimmung dieses Tiefenbildes in voller Auflösung bereits nur noch mit etwa 2 Bildern pro Sekunde, die Performanz des Ansatzes ist also in dieser Form nicht ausreichend für den Echtzeiteinsatz an Bord.

5.1.3.7.3 Erweiterter Filter Deutlich bessere Werte sind zu erwarten, wenn für die Ermittlung der Tiefeninformationen weitere Informationen verwendet werden, wie etwa Sensorinformationen über Rotation und die Zustandshypothese des Filters bezüglich seitlicher Bewegung / Translation.

Dies würde es erlauben einen Erwartungswert für den optischen Fluss abhängig von der Tiefe zu berechnen, bzw. aus dem Istwert einen Absolutwert für die Tiefe zu berechnen.

Des weiteren wäre mit dem Erwartungswert auch die Richtung des Flussvektors vorgegeben, weicht dieser seitlich von diesem ab, wäre dies ein Hinweis auf eine Eigenbewegung des gesichteten Objektes oder aber eine missglückte Flussbestimmung. In beiden Fällen sollte der Fluss nicht, oder nur mit niedriger Gewichtung in die weitere Berechnung einfließen, da sonst etwa beim EKF die Gefahr besteht, dass der Filter divergiert.

Diese Rückkopplung des SLAM Filters in die Flussbestimmung wurde jedoch nicht mehr implementiert, da hierfür bereits ein funktionstüchtiger SLAM Algorithmus erforderlich gewesen wäre, die Performance der Flussbestimmung jedoch hierfür ungenügend war.

Die Evaluierung eines solchen erweiterten Flussfilters steht somit noch aus.

5.1.4 Evaluierung

Den optischen Fluss für das gesamte Bild zu errechnen ist mit den zur Verfügung stehenden Algorithmen zu ungenau oder zu langsam, wenn diese Bestimmung auf einer regulären CPU durchgeführt wird. Um diese Methode in der Praxis einzusetzen muss sie um mindestens den Faktor 10 beschleunigt werden. Auch eine höhere örtliche Auflösung wäre hilfreich, da eine Tiefenauflösung von 32 auf 32 Bildpunkten deutlich unter der 2D Auflösung liegt.

Die durchgeführten Berechnungen beim Template Matching sind in hohem Maße parallelisierbar. Spezialhardware wie ein FPGA[Tri94], oder auch nur parallele Ausführung auf einer GPU[Gla52, AHAS08] würden diesen Ansatz ohne weiteres realistisch durchführbar machen. Embedded ARM CPUs für Video-Smartphones[War05] verfügen oft über dedizierte Hardwarekomponenten zur Videokompression, die auch Hilfskomponenten enthalten um den optischen Fluss zu bestimmen, deren Programmierung ist aber wegen geschlossener Schnittstellen und proprietärer Algorithmen nicht trivial und wurde daher im Rahmen dieser Arbeit nicht angegangen.

Das in 5.1.3.7.2 erkennbare Tiefenbild ist aussagekräftig für nahe an der Kamera gelegene Objekte. Weiter entfernte Bereiche bei denen der Unterschied im optischen Fluss sehr klein ist werden dagegen nicht sichtbar aufgelöst und gehen im Rauschen unter. Selbst der eingesetzte Tiefpass kann dies nicht kompensieren, hier wäre eine Beobachtung über deutlich längere Zeiträume erforderlich. Da aber die Kamera insbesondere bei fliegenden Aufnahmen, wie beim Kurvenflug oft Drehungen unterworfen ist, wandern Objekte hierfür zu früh aus dem Kamerabild aus.

Hier ist es also zwingend erforderlich deren Zustand von Objekten im dreidimensionalen Raum um die Kamera weiter zu betrachten und zu berechnen. Eine Tiefenbestimmung allein auf den Pixeln des zweidimensionalen Kamerabildes ist somit nur sehr eingeschränkt und im Nahbereich möglich, ähnlich wie auch eine Stereokamera nur in einer begrenzten Tiefe aussagekräftige Tiefeninformationen liefert, da ab eines bestimmten Abstandes zur Linse der Versatz nicht mehr messbar ist.

Trotz allem kann die Bestimmung des optischen Flusses sicherlich hilfreich sein um einem SLAM Algorithmus zusätzliche Informationen über sichtbare Landmarken zu geben. Findet diese Bestimmung nicht hierarchisch für das gesamte Bild statt, sondern nur an der Position vermuteter Landmarken und mit dem Startwert des aus der Zustandshypothese errechneten Flusses, ist auch der Rechenaufwand durchaus vertretbar. Damit hätten wir eine zusätzliche Datenquelle um einem Algorithmus wie etwa MonoSLAM mit weiteren Messwerten zu versorgen. Gleichzeitig steht mit Template Matching ein effizientes Verfahren zum Erkennen der Position von Landmarken im 2D RGB Raum zur Verfügung.

5.2 MonoSLAM Implementierung basierend auf RT-SLAM

Da die Tiefenbestimmung aus optischem Fluss nicht in der Lage war in ausreichend performanter Form Tiefenwerte zu liefern, sind wir in der Wahl des SLAM Algorithmus auf Algorithmen beschränkt, die direkt auf Bilddaten arbeiten.

Als vielversprechendste Alternative bietet sich hier eine Implementierung von MonoSLAM[DRMS07] an, da dieser Algorithmus, trotz einiger Nachteile die wir in 4.2.1 beschrieben haben, speziell für den Einsatzzweck mit einer einzelnen Kamera hin entwickelt wurde.

Um Zeit zu sparen beginnen wir mit einer Recherche nach flexiblen existierenden Implementierungen die wir für unsere Zwecke anpassen können. Viele im Rahmen von Forschungsarbeiten geschriebene SLAM Algorithmen sind mehr "proof of concept" Code und, wie etwa das MonoSLAM Beispiel auf openslam.org[SFG12], in Matlab[HL97] geschrieben.

Dies erfüllt kaum unsere Anforderungen nach einem praxistauglichen Algorithmus für ein On-Board Echtzeitsystem. Wir benötigen eine effiziente und gleichzeitig vielseitige Implementierung von MonoSLAM. Eine solche finden wir in Form von RT-SLAM[RGS+11] auf der Webseite von OpenRobots.org, einem Projekt der Université de Toulouse am Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) unter der Leitung von Cyril Rousillon.

5.2.1 RT-SLAM

RT-SLAM[RGS+11] steht für Real Time SLAM und ist somit ein für Echtzeitanwendungen gedachtes Framework. RT-SLAM implementiert MonoSLAM[DRMS07] sowie die dafür erforderlichen Techniken wie einen Extended Kalman Filter, Funktionen für das Bildverstehen wie den Harris Detektor[HS88] und mathematische Hilfsfunktionen wie das RANSAC[CMK03] Verfahren in einem objektorientierten Umfeld. Weitere Bestandteile des Frameworks sind Funktionen zur Visualisierung oder eine Demo-Anwendung, bei der SLAM nur an Hand einer Kamera und ohne sonstige Sensorinformationen durchgeführt werden kann.

RT-SLAM ist dabei flexibel genug um mehr als nur ein Weltmodell zu unterstützen und zusätzliche Sensorinformationen in seine Zustandsberechnung einfließen zu lassen.

5.2.2 Architektur

RT-SLAMs Fähigkeiten gehen teilweise deutlich über die einer einfachen MonoSLAM Implementierung hinaus. Wir betrachten daher dessen Architektur in einem kurzen, in Abbildung 5.19 dargestellten Überblick.

5.2.2.1 Landmarken

RT-SLAM ist in der Lage, Landmarken entweder als zusammenhängende Segmente oder als Punkte im Raum zu modellieren. Ersteres ist leider noch nicht ausgereift und unzureichend dokumentiert, daher beschränken wir uns auf die Betrachtung der zweiten Darstellungsweise.

RT-SLAM implementiert ein Verfahren zur Effizienzsteigerung, bei dem Landmarken, deren Positionsabschätzung eine angegebene Genauigkeit überschritten haben, vereinfacht repräsentiert werden. Initialisiert werden Landmarken wegen der zunächst sehr ungenauen Tiefe als „verankerte Homogene Punkte mit inverser Tiefenparametrisierung“[Sol10] die jedoch zur Darstellung eine 6-dimensionale Repräsentation in der Zustandsmatrix benötigen. Ist der Zustand allerdings weit genug konvergiert, kann diese Darstellung vereinfacht werden zu einer dreidimensionalen euklidischen Repräsentation der jeweiligen Landmarke.

5.2.2.2 Roboter

RT-SLAM modelliert Roboter als Teil der Karte, die sowohl über Zustände (Position und Ausrichtung im Raum) als auch Sensoren verfügen. Die Sensoren machen wiederum Beobachtungen (observations) bezüglich des Roboters oder bezüglich Landmarken.

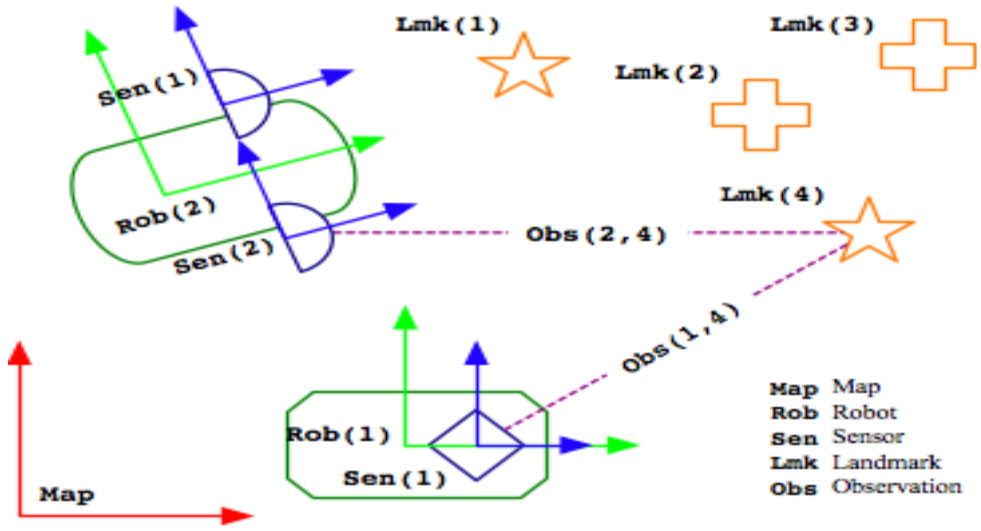

Abbildung 5.19: Weltmodell von RT-SLAM, aus [RGS+11] Seite 2.

Ein Roboter kann dabei über ein beliebiges Bewegungsmodell verfügen, welches den Zustandsübergang von einem Schritt zum nächsten beschreibt.

In RT-SLAM sind bereits mehrere Bewegungsmodelle für Roboter implementiert für die unterschiedliche Sensoren erforderlich sind.

5.2.2.2.1 Konstante Geschwindigkeit Das einfachste Modell ist das eines frei beweglichen Körpers mit Masseträgheit, welcher seine Geschwindigkeit und seine Rotation abgesehen von einem Störeinfluss beibehält. Der Zustand ist gegeben durch den Tupel $\vec{x} = [\vec{p}, \vec{q}, \vec{v}, \vec{w}]$ wobei $\vec{p} = [x, y, z]$, $\vec{q} = [q_0, q_1, q_2, q_3]$ die Ausrichtung im Raum als Quaternion, $\vec{v} = [v_x, v_y, v_z]$v die Geschwindigkeit und $\vec{w} = [w_x, w_y, t_z]$ die Rotationsgeschwindigkeit um die eigene Achse darstellt.

5.2.2.2.2 Konstante Geschwindigkeit auf Roboter zentriert. Dieses Modell ist fast identisch mit 5.2.2.2.1 jedoch mit dem Unterschied, dass die Karte auf den Roboter zentriert ist. D.h. der Roboter bleibt im Ursprung der Karte und es werden stattdessen alle Landmarken bewegt. Dieses Bewegungsmodell eignet sich in erster Linie wenn keine dauerhafte Karte angefertigt wird, sondern nur die Lokalisierung an Hand gerade sichtbarer Landmarken stattfinden soll.

5.2.2.2.3 Roboter mit Odometriemessung Mathematisch ist dieses Modell sogar einfacher als 5.2.2.2.1 Der Zustand besteht lediglich aus einem Tupel $\vec{x} = [\vec{p}, \vec{q}]$. Allerdings wird für die Zustandsübergangsfunktion $\vec{x}_k = f(\vec{x}_{k-1}, \vec{u}_{k-1})$ eine Messung $\vec{u} = \left[\vec{dp}, \vec{dq}\right]$ der tatsächlich zurückgelegten Strecke $\vec{dp}$ und des Winkels $\vec{dq}$ benötigt.

5.2.2.2.4 Roboter mit Inertialsystem Dieses ist das komplexeste in RT-SLAM implementierte Robotermodell. Es ähnelt stark dem EKF des OpenPilot 2.2.2 und beherbergt eine 19 dimensionale Zustandshypothese. $\vec{x} = \left[\vec{p}, \vec{q}, \vec{v}, \vec{a}, \vec{b}, \vec{g}\right]$ wobei $\vec{p}$ die 3D Position im Raum, $\vec{q}$ die Ausrichtung als Quaternion, $\vec{v}$ die Geschwindigkeit als 3D Vektor, $\vec{a}$ und $\vec{b}$ die Nullpunkt-Hypothesen angeschlossener Gyroskop- und Accelerometersensoren, und $\vec{g}$ die Hypothese der Gravitationskraft ist. Letztere wird dabei als konstant,

aber unbekannt angenommen, womit das Kartenkoordinatensystem beliebig gegenüber dem Gravitationsvektor gedreht sein kann. Dies unterscheidet dieses Modell von OpenPilot, dessen EKF Weltmodell mit einer konstanten und bekannten Gravitation rechnet und auch das Accelerometer als korrekt kalibriert voraussetzt.

Dieses Modell benötigt zum Zustandsübergang $f(\vec{x}_{k-1}, \vec{u}_{k-1})$Messwerte eines Gyroskop und Accelerometer Sensors $\vec{u} = \left[\vec{dv}, \vec{dq}\right]$.

5.2.2.3 Karte

Sowohl Landmarken als auch Roboter sind Bestandteil eines abstrakten Kartenobjekts. RT-SLAM basiert zunächst auf einer generischen EKF-SLAM[Aic07, RGS+11] Implementierung, die Karte wird damit repräsentiert als Liste aller Zustandsvariablen aller Landmarken und Roboter sowie einer sehr großen aber dünn besetzten Kovarianzmatrix.

RT-SLAM verwendet zur Speicherung und Berechnung dieser Strukturen die Klassen des Boost UBLAS Frameworks[WKW02], die eine Ressourcen schonende Verwaltung dieser Strukturen zulässt.

5.2.2.4 Welt

Das Weltmodell beinhaltet wiederum ein oder mehrere Karten, wobei bisher eine Mehrkartendarstellung in RT-SLAM zwar angedacht, aber noch nicht implementiert wurde. Die dafür in RT-SLAM vorgesehene Funktion enthält als Code bisher lediglich ein „{ // TODO }“ in der vorgesehenen Klasse in rtslam/include/rtslam/mapManger.hpp. Siehe Anhang A.2.

5.2.2.5 Sensoren

Jeder Roboter verfügt über verschiedene Sensoren. Diese sind in RT-SLAM sowohl abstrakt als mathematisches Modell für verschiedene Sensortypen, wie Kameras oder Sensoren zur absoluten Positionsbestimmung, implementiert, als auch als explizite Treiberklassen, die zum Zugriff auf Hardwareschnittstellen dienen. RT-SLAM beinhaltet etwa Klassen zum Lesen von Monochrom-Bildern von hochwertigen Firewire Kameras wie sie für Forschungszwecke eingesetzt werden[And99], und Code zur Anbindung an externe Positionsbestimmungssysteme.

Im Rahmen der Sensorauswertung wird auch die Innovation bezüglich Landmarken bestimmt. RT-SLAM setzt dabei auf den Harris Detektor[HS88] um "Points of Interest"[Jäh05] in Bildern zu erkennen und wieder zu finden. Diese werden dann als Landmarken schließlich in die Karte eingetragen. RT-SLAM verwendet die OpenCV Bibliothek zur Verwaltung von 2D Bilddaten, allerdings wird OpenCV nicht für Matrixoperationen verwendet, da RT-SLAM hierzu Boost UBLAS[WKW02] einsetzt.

5.2.2.5.1 Kamera Das von RT-SLAM unterstützte Kameramodell ist mit Abstand der komplexeste Sensortyp. Zunächst ermittelt dieser ein 2D Bild und verfügt über ein Abbildungsmodell um 3D in 2D Koordinaten zu überführen. Dabei kommt zunächst das auch bei OpenCV gebräuchliche Lochkameramodell[KB06] zum Einsatz, das nach dem Strahlensatz 3D Koordinaten linear auf eine Ebene abbildet. Zusätzlich unterstützt RT-SLAM auch ein Modell für Linsenkorrektur, welches an das von OpenCV verwendete angelehnt ist, jedoch eine einfachere Korrekturformel verwendet. Siehe Abschnitt 5.2.7.2.

RT-SLAM erstellt dabei niemals ein komplettes entzerrtes Bild. Vielmehr wird bei der Umwandlung von 3D in 2D Koordinaten oder umgekehrt jeweils die Korrekturformel angewandt. Hierdurch entfällt dieser ansonsten in der Robotik verbreitete aber rechenaufwändige Schritt.

Der Kamerasensor liefert letztendlich Innovationen für die Position von Landmarken im 2D Bildraum, allerdings wird er auch für zusätzliche Schritte, wie die Erstellung neuer Landmarken verwendet.

Auf unterster Ebene benötigt der Kamerasensor lediglich ein Monochrombild als Eingabe.

Abbildung 5.20: Visualisierung von RT-SLAM 2D Daten, Videobild mit erkannten Landmarken und deren Positionsvarianz.

5.2.2.5.2 Positionssensoren Eine weitere von RT-SLAM unterstützte Sensorklasse sind Positionssensoren. Diese erlaubt das Einlesen von Informationen über Roboter, wie Position und Ausrichtung im Raum, und wird instantiiert von einer ganzen Reihe von Sensoren für Lokalisationssysteme aller Art wie etwa GPS aber auch externe visuelle Positionierungssysteme. Wie bei der Kamera werden die von diesen Sensoren gelieferten Innovationswerte während des EKF Korrekturschritts verarbeitet.

5.2.2.6 Estimatoren

Estimatoren liefern ebenfalls Messwerte, werden von RT-SLAM aber anders eingesetzt. Anders als Sensoren werden diese nicht während des Korrekturschritts eingesetzt, um dem EKF Innovationsdaten zu liefern, sondern sie liefern Daten die direkt während des Zustandsübergangs im Rahmen des Roboter Bewegungsmodells zur nächsten Zustandshypothese verwendet werden. Beispiele wären hier etwa Odometriedaten für den Roboter mit Odometriemessung wie in 5.2.2.2.3 oder aber Messungen der Inertialsensoren gemäß 5.2.2.2.4.

5.2.2.7 Visualisierung

RT-SLAM verfügt neben Debug-Ausgaben an der Kommandozeile über zwei Visualisierungssysteme.

Eine auf Qt aufbauende Bibliothek, genannt *Qdisplay,* gibt 2D Bilddaten der Kamera oder der Kameras mit entsprechenden Überlagerungen, die erkannten Landmarken entsprechen, aus. Siehe Abbildung 5.20.

Des weiteren erlaubt die *GDHE* Bibliothek, die Karte mit gesichteten Landmarken, der aktuellen Position des oder der Roboter und deren zurückgelegtem Weg dreidimensional auszugeben.

Diese Ansicht erlaubt es den Sichtwinkel und die Distanz interaktiv mittels Mauseingabe frei zu bestimmen (Abbildung 5.21).

Beide Ansichten zusammen erlauben es, die erkannte Position von Landmarken mit der Realität abzugleichen, und so das Resultat der simultanen Lokalisierung und Kartografierung zu beurteilen.

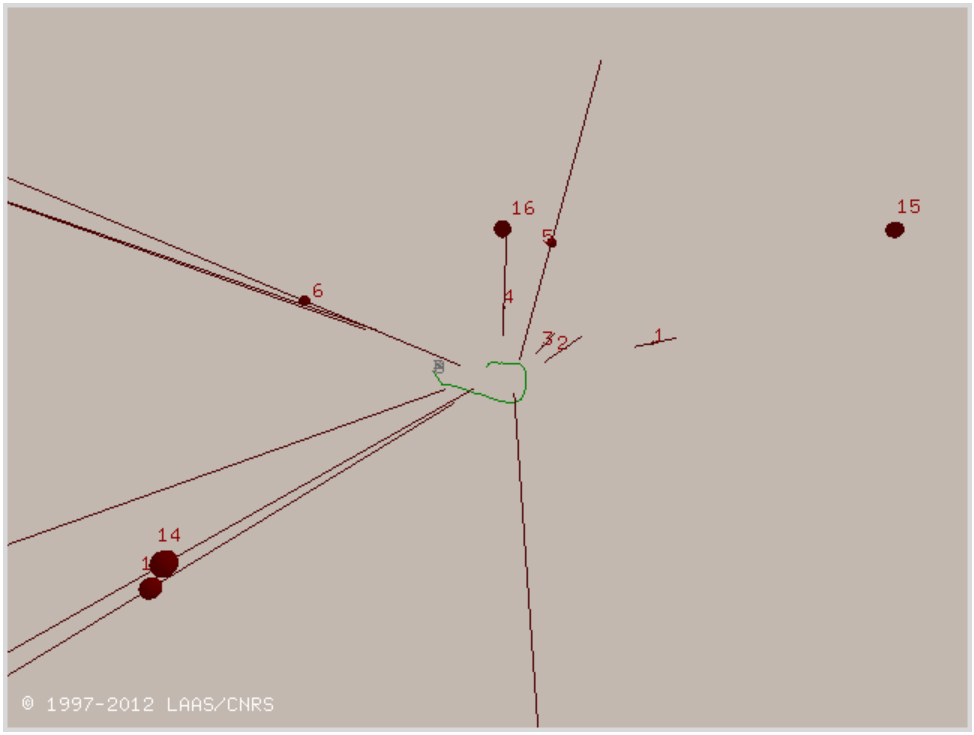

Abbildung 5.21: Visualisierung von RT-SLAM 3D Daten, Erkannte Landmarken, deren Sichtachse zum Zeitpunkt der Erkennung und wahrscheinliche Position, zurückgelegte Strecke der Kamera in Grün.

5.2.3 Test als eigenständige Applikation

In einem ersten Schritt der Evaluierung versuchen wir nun RT-SLAM überhaupt zum Laufen zu bekommen. Als eine Variante von MonoSLAM ist RT-SLAM generell in der Lage, ausschließlich mit einer einzelnen Kamera als Sensor zu arbeiten. Es sollte also möglich sein, die beiliegende demo_slam Applikation mit einer USB Kamera zum Laufen zu bringen.

Dies scheitert zunächst daran, dass RT-SLAM in der zum Zeitpunkt der Erstellung dieser Arbeit vorliegenden Version keine Treiberklasse für generische USB Kameras mitbringt. Diese lassen sich, wie die meisten Videoquellen unter Linux mit dem Video4Linux2[SDVR99] Treiber ansteuern. Zudem bietet das auch von RT-SLAM zur Verwaltung von 2D Bilddaten verwendete OpenCV Framework die Möglichkeit auf beliebige Videoquellen, sowohl Kameras als auch Videodateien, zuzugreifen. Von dieser Möglichkeit haben wir ja schon in Abschnitt 3.2.2 Gebrauch gemacht.

Der modulare Aufbau von RT-SLAM erlaubt uns aber das Schreiben einer weiteren Kamera-Sensorklasse mit sehr vertretbarem Aufwand. Diese wurde implementiert, und damit das Einlesen von Bildern von V4L2 Geräten ermöglicht. Der Code hierfür findet sich in der Klasse HardwareSensorCameraOpenCV::, siehe Anhang A.2.

Ein erster Test von RT-SLAM mit Standardeinstellungen ergab einen funktionierenden Filter, der jedoch bei Bewegungen schnell divergiert.

Dies ist auch nicht weiter verwunderlich. Es genügt schließlich, dass für einen relativ kurzen Zeitraum nicht ausreichend viele Landmarken erkannt werden. Auch eine unzureichende Kalibrierung der Kamera und daraus resultierende Fehler in der Positionsabschätzung können leicht zu einem Zustand führen, in dem die Unsicherheit bezüglich Position und Ausrichtung, aber auch die Unsicherheit der Geschwindigkeit und Rotationsgeschwindigkeit stark zunimmt.

Eine auch nur über kurze Zeit fortgeführte falsche Zustandsannahme bezüglich der Rotationsgeschwindigkeit führt aber zu einer massiv falschen Annahme der Ausrichtung im Raum, und damit auch einer inkorrekten Linearisierung des EKF. Siehe Abbildung 5.22.

An diesem Punkt divergiert der Filter, selbst wenn einzelne Landmarken wieder gesehen werden, da RT-SLAM sie an der falschen Stelle vermutet und somit auch erneut erkannte Landmarken allerhöchstens

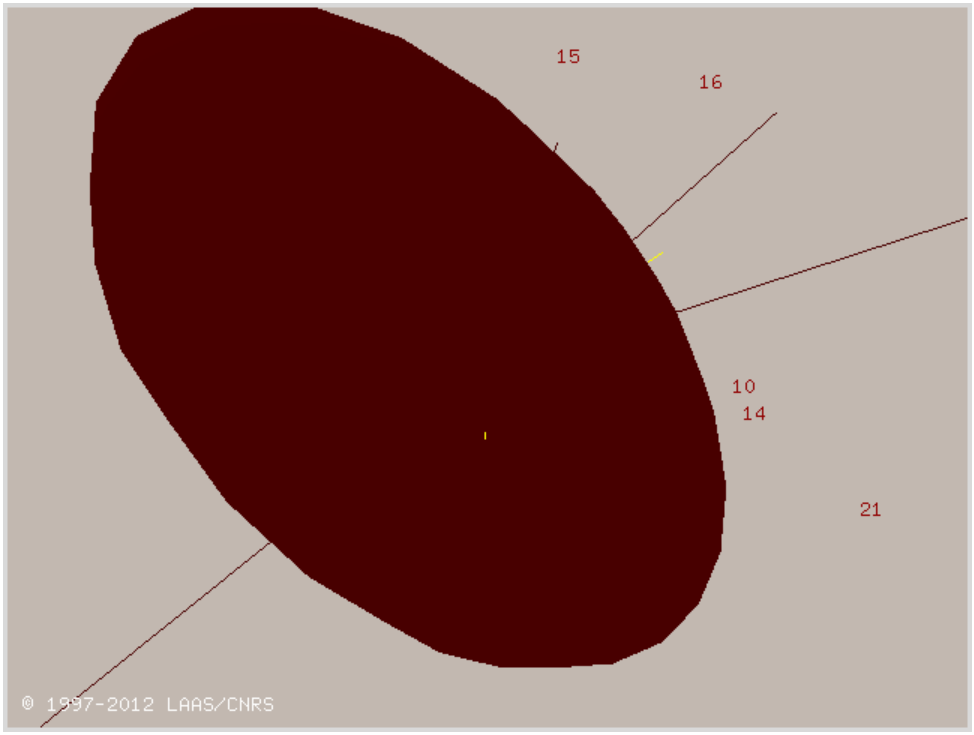

Abbildung 5.22: Visualisierung von RT-SLAM 3D Daten. Divergiert der Filter, so können Landmarken nicht mehr effektiv lokalisiert werden, womit auch die Bestimmung der eigenen Position fehlschlägt.

als neue Landmarken erkennt und erneut an anderer Stelle in die Karte einträgt.

Die Dokumentation von RT-SLAM empfiehlt für die ausschließliche Verwendung einer Kamera eine Bildfrequenz von wenigstens 60 Bildern pro Sekunde.

Ein Testdurchlauf von RT-SLAM auf dem während des Fluges aufgenommenen Video mit 18 fps ergab dementsprechend auch nur eine sehr schnelle Divergierung des Filters, der prompt ein Absturz des RT-SLAM Programms folgte.

Dieses Problem dürfte aber für unsere Zwecke eine untergeordnete Rolle spielen, da wir ja Dank der OpenPilot Hardware bereits über eine Positions- und Ausrichtungshypothesen, sowie über Beschleunigungs- und Rotationssensoren verfügen. Es gilt also zu untersuchen wie RT-SLAM mit diesen zusätzlichen Informationsquellen zurechtkommt.

5.2.4 Anbindung an SimPosix und OpenCV

Mögliche Beispiele für die korrekte Instantiierung der diversen existierenden RT-SLAM Komponenten finden sich in der RT-SLAM beiliegenden demo_slam Applikation. Um diese aus SimPosix heraus zu instantiieren (siehe 3.1) waren jedoch einige Modifikationen erforderlich.

5.2.4.1 Build Environment

RT-SLAM setzt zur Kompilierung auf qmake, wobei die jeweiligen Unterprojekte wie GDHE oder Qdisplay als dynamische Bibliotheken gebaut werden.

SimPosix ist dagegen ein mit GNU Makefiles gebautes monolithisches Projekt, das alle eigenen Bestandteile statisch ein"linkt". Dies um zusätzlichen zu compilierenden Code, auch zu C++ Code, zu ergänzen ist uns in 3.2.1 bereits gelungen jedoch ist die Menge an Quellcode die für RT-SLAM compiliert werden muss in einer anderen Größenordnung.

Dabei traten Probleme auf, etwa Überlagerungen von Funktionen und Klassennamen zwischen den verschiedenen Komponenten, trotz des Einsatzes von C++ Namespaces. Auch identisch benannte Dateien

waren ein Problem, da diese zu identisch benannten Objektdateien führten, die sich während des Link-Vorgangs gegenseitig überschrieben. Dies konnte nur durch Umbenennungen der betreffenden Funktionen und teilweise durch Umbenennen von Dateien gelöst werden.

Teilweise beinhaltete der RT-SLAM Quellcode auch Fehler wie Methoden und Funktionsdefinitionen in Header Dateien, die während des Link-Vorgangs zu Kollisionen führten, weil die betroffene Methoden deswegen mehrfach compiliert wurde, sofern die Header Datei mehr als einmal eingebunden war. Dies konnte gelöst werden indem die funktionalen Komponenten in die entsprechenden C++ Dateien verschoben wurden, wo sie eigentlich hingehören.

Alles in allem erwies sich das RT-SLAM Framework, obwohl funktional durchdacht, von der Code Qualität als problematisch, fehlerbehaftet und nicht 100% ausgereift. Hierauf werden wir in Abschnitt 5.2.5 weiter eingehen.

Zuletzt müssen noch sämtliche Systembibliotheken die RT-SLAM benötigt, wie etwa Boost, Qt oder OpenCV in die LDFLAGS Variable des Makefiles aufgenommen werden. Damit kann RT-SLAM als Bestandteil von SimPosix innerhalb der OpenPilot Firmware kompiliert werden (mittels „make sim_posix“).

5.2.4.2 RT-SLAM Instanzenklasse

In der demo_slam Applikation wird RT-SLAM aus einer nicht objektorientierten main() Funktion heraus aufgerufen. Um RT-SLAM von SimPosix aus zu instantiieren sind wir jedoch an einer instantiierbaren Hauptklasse interessiert, über die auch die Kommunikation und Synchronisation zwischen SimPosix und RT-SLAM abgewickelt werden kann.

Diese wurde realisiert als die Klasse RTSlam::, in der Datei flight/Modules/SLAM/rtslam.cpp (Anhang A.2). Prinzipiell handelt es sich dabei um eine Reimplementierung von demo_slam, dem jedoch einige nicht benötigte Funktionen genommen wurden. Die üblicherweise aus Kommandozeilenoptionen ausgelesenen Konfigurationsparameter werden ebenfalls hart kodiert.

Diese Klasse wird wiederum von der bereits für unsere Arbeiten in 3.3.1 und 5.1.1 verwendeten Klasse OpenCVslam:: (flight/Modules/SLAM/opencvslam.cpp) instantiiert und aufgerufen, jedoch in der in A.2 referenzierten Revision.

5.2.4.3 Parallelität

RT-SLAM verwendet die Multi-Thread Funktionen zur parallelen Ausführung und Interprozesskommunikation von Boost[WKW02]. Wir müssen also einen Weg finden, diese Boost Threads auszuführen, ohne dass es dabei zum Konflikt zwischen diesen und den Tasks des von SimPosix simulierten FreeRTOS kommt.

Der dabei eingeschlagene Weg ist dabei die Ausführung von Boost Threads außerhalb und parallel zu dem von FreeRTOS verwalteten Task. Dies erreichen wir indem wir zuerst einen separaten Thread für die Initialisierung von RT-SLAM starten, mittels der POSIX Funktion pthread_create()[But97]. Dieser braucht eine entsprechende Signalmaske um nicht auf die von SimPosix verwendeten Signale zur Tasksteuerung zu reagieren, und arbeitet so echt parallel, so wie wir dies bereits für die Funktionen zur Ein/Ausgabe von OpenCV in Abschnitt 3.3.3 implementiert hatten.

Aus dieser Umgebung können dann wiederum von RT-SLAM beliebig Boost Threads initialisiert werden, ohne das es zum Konflikt kommt.

5.2.4.4 Datenweitergabe

Die im SLAM Modul von SimPosix bereits zur Verfügung stehenden Sensordaten müssen in geeigneter Weise an RT-SLAM weitergegeben werden. Die Schwierigkeit dabei ist, dass RT-SLAM in separaten Threads asynchron zur Ausführung von SimPosix abläuft. Dies wird jedoch von den in RT-SLAM bereits implementierten Treiberklassen schon berücksichtigt, bei denen Sensordaten direkt aus der Hardware ausgelesen werden und jeweils mit einem Zeitstempel versehen in einen Puffer geschrieben werden. Auf diesen Puffer greift RT-SLAM dann im nächsten Rechenschritt zu, sobald alle Daten für diesen Schritt vorhanden sind.

Die Lösung ist daher, neue Treiberklassen zu schreiben, wie dies in 5.2.3 bereits für das Einlesen von Bildern von OpenCV geschah, jedoch ohne einen eigenen Thread für die Ein/Ausgabe. Stattdessen wird eine öffentliche („public“) Methode kreiert und exportiert, so dass diese von SimPosix aus angesprungen werden kann. Die Bild- oder sonstigen Sensordaten werden dabei übergeben.

Das Füllen des Puffers findet somit innerhalb des SimPosix FreeRTOS Tasks statt, der ebenfalls asynchron zum Abarbeiten des RT-SLAM Filters abläuft, das Auslesen des Puffers durch RT-SLAM verläuft dagegen wieder analog zur Verarbeitung im alleinstehenden Betrieb von RT-SLAM.

Allerdings arbeitet auch RT-SLAM auf Monochrombildern, so das in diesem Schritt eine Konvertierung in ein Graustufenbild stattfinden muss. Diesen Schritt wollen wir natürlich vermeiden, jedoch ist die Erweiterung von RT-SLAM auf Farbbilder nicht ohne erheblichen Aufwand möglich, so dass diese Vorgehensweise an diesem Punkt unvermeidlich ist.

5.2.5 Korrekturen an RT-SLAM

Ein Framework von der Größe und Komplexität RT-SLAMs ist zwangsläufig nicht fehlerfrei. Einige dieser Fehler wurden während der Arbeiten an dieser Diplomarbeit entdeckt und mussten behoben werden um weiter arbeiten zu können. Die Fehler, beziehungsweise Korrekturen wurden über die Mailingliste des OpenRobots Projekts an die Entwickler von RT-SLAM weitergeleitet, womit die vorgenommenen Korrekturen auch in die weitere Entwicklung von RT-SLAM einfließen können und teilweise auch schon sind.

5.2.5.1 Fließkommaarithmetik

Wie bereits bei den ersten Tests unter 5.2.3 festgestellt, gibt es Situationen in denen das unmodifizierte RT-SLAM mit entsprechenden Ausnahmen abbricht. Dies waren meist mathematische Funktionen des Boost Frameworks, die mit ungültigen Werten aufgerufen wurden, etwa Nullen oder den Fließkomma-Metawerten Inf und -NaN[Gol91].

Dies kam insbesondere dann vor, wenn der Filter divergierte, jedoch in Einzelfällen auch spontan beim Anlegen einer neuen Landmarke oder deren Reparametrisierung.

Eine einfache Korrektur war in diesem Fall nicht möglich. Offensichtlich wurde an einer oder mehreren Stellen in den Berechnungen des Filters vergessen entsprechende Fallunterscheidungen einzubauen um Division durch Null, Wurzelziehung aus negativen Zahlen und ähnliche unzulässige Operatoren zu unterbinden.

Diese liefern dann als Ergebnis die Fliesskomma-Metawerte Inf (unendlich) bzw. -NaN (not a number, keine Zahl) die leider die unangenehme Eigenschaft haben, in weiteren Berechnungen zu propagieren. So liefert jede mathematische Fließkommaoperation mit -NaN wieder -NaN. Ein solcher Wert in einem beliebigen Eingangswert des Filters, selbst wenn dieser ansonsten keine messbaren Auswirkungen hätte, endet so in der Kovarianzmatrix, und im nächsten Schritt auch in den Zustandsvariablen, was wiederum zum Absturz bzw. Abbruch führt.

Bei Implementierung eines Filters, der jeweils auf dem aktuellen Zustand den nächsten berechnet ist generell darauf zu achten, das ein -NaN Wert nicht in die Berechnung gelangen kann, dies wurde bei RT-SLAM offensichtlich nicht ausreichend berücksichtigt.

Einen -NaN Wert im Nachhinein zurückzuverfolgen ist dabei nahezu unmöglich, insbesondere bei einem so komplexen Softwarepaket wie RT-SLAM, die einzige Lösung war daher, die Ausführung sofort zu unterbrechen, wenn eine Operation zu einem -NaN Resultat führte.

Dies ist möglich, da bei in Hardware ausgeführten Fließkommaoperationen, die CPU eine Unterbrechung auslöst, wenn diese zu ungültigen Werten führen, die sogenannte Gleitkommaausnahme. Anders als bei Integer-Operationen, bei denen eine Division durch Null zur Ausnahmebehandlung und unter Umständen zum Abbruch des Programms führt ist dies bei Gleitkommaoperationen normalerweise nicht der Fall. Dieses Verhalten lässt sich auf x86 Architekturen jedoch aktivieren, mittels der POSIX C Funktion feenableexcept()[SRR92], definiert in fenv.h.

Daraufhin waren jedoch immer noch eine Vielzahl von Testläufen und Korrekturen an etwa ein Dutzend Stellen im RT-SLAM Code und im Quellcode der Hilfsbibliotheken erforderlich, an denen notwendige

Fallunterscheidungen etwa in numerischen Lösungsverfahren, Interpolationsverfahren und anderen mathematischen Operationen nicht vorhanden waren.

Dabei war auffällig, dass einige Bestandteile des Codes deutlich ausgereifter erschienen und über derartige Fallunterscheidungen verfügten, die allerhöchstens in Einzelfällen vergessen wurden. Ein Beispiel hierfür wäre etwa die Klasse QuickHarrisDetector:: die mit der GIT Revision 8b4769c23c5374e75e5f503a03741e28b122d9b9 korrigiert wurde. Andere Klassen hingehend, auch in von RT-SLAM benutzten mathematischen Hilfsbibliotheken, schienen teilweise ohne jegliche Überprüfung programmiert worden zu sein, wie etwa einige der in der "jmath" Bibliothek in GIT Revision d4c956f3f0df3a1cc0f47a7a7bc0e1d91c76ce9a behobenen Fehler. Auch an der "correl" genannten Bibliothek die von RT-SLAM verwendet wird mussten Korrekturen vorgenommen werden. Siehe Anhang A.2.

5.2.5.2 Falsche Berechnung der Innovation

In RT-SLAM war bereits eine Sensorklasse implementiert, die das Messen der Roboterausrichtung im Raum zulässt. Der Winkel des Roboters wird dabei in 3 Eulerschen Winkelangaben gemessen, in einer der Darstellung von OpenPilot vergleichbaren Form (Roll, Pitch, Yaw). Diese wurde nun auch für das Einlesen unserer Zustandsdaten in 5.2.4.4 verwendet.

Bei der Berechnung der Innovation wurde jedoch von den RT-SLAM Entwicklern übersehen, dass diese Winkel Sprungstellen aufweisen. Beispielsweise würde der Erwartungswert von Yaw=-179 Grad und ein Messwert von Yaw=+179 Grad einer Innovation von -2 Grad entsprechen, die naive Berechnung liefert jedoch einen Wert von 179 - (-179) = 358 Grad. Eine solch hohe Innovation führt logischerweise zu drastischem Fehlverhalten des Filters.

Die Lösung ist jedoch einfach. Prinzipiell ist jedes Winkelpaar mehrdeutig, da die Differenz auf mehreren Wegen erreicht werden kann. Wir müssen daher einfach grundsätzlich annehmen, dass die direkteste Variante gewählt wurde, und müssen so Innovationen von mehr als 180° grundsätzlich ausschließen. Der Fehler in RT-SLAM selbst wurde in GIT Revision 9ba444a64d92e8ed7f0c6bb7ed87c66dab27c4ff behoben.

5.2.6 Konfiguration

RT-SLAM bietet, wie wir das bei einem Filter dieser Komplexität erwarten, eine große Anzahl von Konfigurationsmöglichkeiten. Angefangen vom Roboter- und Kartenmodell über die maximale Anzahl von gespeicherten Landmarken, Parameter für die Ausrichtung der Kamera und für Linsenkorrekturfaktoren bis hin zu Gewichtungen für verschiedene Filter lassen sich über Konfigurationsdateien einstellen, leider in vielen Fällen ohne nennenswerte Dokumentation.

In einigen Fällen waren auch Anpassungen am Quellcode erforderlich um passende Modelle auszuwählen, bzw. um die Verwendung der unter 5.2.4.4 geschriebenen eigenen Sensortreiber zuzulassen.

Für jede mögliche Konfigurationsoption musste dabei ermittelt werden, was diese genau bewirkt, um dann einen geeigneter Wert mittels Überlegungen, Messungen wie im Fall der Kameraparameter, und in manchen Fällen durch Ausprobieren zu finden.

5.2.6.1 estimation.cfg

data/estimation.cfg beinhaltet diverse Parameter, die das Verhalten von RT-SLAM im Detail beeinflussen.

CORRECTION_SIZE Dieser Parameter gibt die Anzahl der Parameter für die Kamerakorrektur an, und damit die Wertigkeit des Polynoms für die Verzerrungskorrektur. Der passende Wert richtet sich nach der Genauigkeit der Kamerakalibrierung, siehe Abschnitt 5.2.7.2.

MAP_SIZE Die Größe der Karte in Zustandsvariablen. Dies beschränkt damit auch die maximale Größe der Kovarianzmatrix. Die maximale Anzahl verwaltbarer Landmarken hängt davon ab ob diese bereits reparametrisiert wurden und wie viele Zustandsvariablen der Roboter selbst benötigt. Der passende Wert richtet sich nach Geschwindigkeit und Rechenkapazität des Systems, wobei wir diesen Wert so hoch wie möglich setzen möchten, um große Karten zuzulassen.

PIX_NOISE Präzision der Kamera, Ortsvarianz eines Pixels, und damit einer Landmarke im 2D Bildraum. Der Vorgabewert der Beispielkonfigurationsdatei beträgt 1.0.

D_MIN Initialisierungswert des inversen Tiefen-Mittelwerts für Landmarken. Dies ist die zu erwartende Entfernung zur Kamera für neue Landmarken. Für einen fliegenden Roboter der sich die meiste Zeit in über 20 Meter Höhe befindet und Sicht auf kilometerweit entfernte Objekte erlangt, ist der Standardwert von 0.5 Metern eher ungeeignet. Ein Wert von mindestens 20 erzielt hier bessere Ergebnisse. Siehe auch 5.2.7.3.1.

REPARAM_TH Der Schwellwert für die Reparametrisierung von Landmarken. Je kleiner dieser Wert, desto genauer muss eine Landmarke erkannt sein um die Reparametrisierung anzustoßen.

GRID_HCELLS, GRID_VCELLS Die Anzahl von Suchzellen pro Bild für das Suchgitter bei der Suche nach neuen Landmarken. RT-SLAM versucht Landmarken im Bild gleichmäßig zu verteilen, sind in einem Quadranten keine Landmarken vorhanden und noch freier Kartenplatz, wird versucht mittels des Harris Detektor[HS88] neue brauchbare Landmarken zu finden.

GRID_MARGIN Minimaler Abstand vom Bildrand bei der Suche nach neuen Landmarken.

GRID_SEPARATION Minimaler Abstand zu existierenden Landmarken bei der Suche nach neuen Landmarken

KILL_SEARCH_SIZE Die Anzahl vergeblicher Suchen nach einer Landmarke bevor diese aus der Karte gelöscht wird.

Sonstige: Es gibt noch diverse andere Konfigurationsoptionen, die sich auf die Verwaltung von Landmarken beziehen, insbesondere das Verhalten des Harris Detektors und die Filterung von Landmarkenmessungen mittels des RANSAC Algorithmus[CMK03]. Leider sind diese weder in der Dokumentation von RT-SLAM, noch in dessen Quellcode ausreichend dokumentiert um deren Funktion hier zuverlässig wiedergeben zu können. Die in der Beispielkonfiguration angegebenen Werte liefern jedoch brauchbare Ergebnisse.

RELEVANCE_TH

MAHALANOBIS_TH

N_UPDATES_TOTAL

N_UPDATES_RANSAC

N_INIT

N_RECOMP_GAINS

RANSAC_LOW_INNOV

RANSAC_NTRIES

HARRIS_CONV_SIZE

HARRIS_TH

HARRIS_EDDGE

DESC_SIZE

MULTIVIEW_DESCRIPTOR

DESC_SCALE_STEP

DESC_ANGLE_STEP

DESC_PREDICTION_TYPE

PATCH_SIZE

MAX_SEARCH_SIZE

MATCH_TH

5.2.6.2 setup.cfg

Setup.cfg beeinflusst hauptsächlich das Verhalten der demo_slam Applikation die RT-SLAM als Anschauungs- und Demonstrationsmaterial mitgeliefert wird.

Einige der Optionen beziehen sich auf die mit demo_slam möglichen Simulationsläufe mittels des von RT-SLAM unterstützten Simulations-Frameworks, welches wir hier nicht weiter betrachten.

Viele der Konfigurationsoptionen sind in der an SimPosix gekoppelten Implementierung, welche gezielt an ein bestimmtes Szenario angepasst wurde daher ohne Relevanz. Einige sind jedoch erforderlich, da sie wichtige Parameter wie Ausrichtung von Sensoren gegenüber dem Roboter, oder auch die Parameter des Kameramodells kodieren. Nur diese sind hier wiedergegeben.

GPS_POSE Position des absoluten Positionssensors relativ zur Kamera, sowie die Ausrichtung der Kamera gegenüber dem absoluten Koordinatensystem. Da in unserem Fall die Kamera gegenüber den absoluten Sensoren verschoben ist, muss dieser Wert entsprechend angepasst werden.

GPS_VARIANCE Die Varianz der vom GPS Sensor gelieferten Position und Geschwindigkeit. Dieser Parameter ist ursprünglich nicht Bestandteil von RT-SLAM sondern wurde während der Implementierung der OpenPilot Sensorklasse hinzugefügt um die Sensorvarianz setzen zu können. Siehe Abschnitt 5.2.4.4.

IMG_WIDTH, IMG_HEIGHT, INTRINSIC, DISTORTION Die Parameter des Kameramodells. Dies beinhaltet die Bildgröße, die Brennweite, und die Parameter des Verzerrungsmodells, siehe Abschnitt 5.2.7.2.

5.2.7 Evaluierung

Wir konzentrieren uns zunächst auf ein Globales Kartenmodell aus 5.2.2.3, da dieses bei kreisförmigen Flügen über dem selben Gebiet, wie sie während des Evaluationsmaterials stattfinden, ein Loop Closing über wiederentdeckte Landmarken zulässt.

5.2.7.1 Bewegungsmodell

Für das Bewegungsmodell wählen wir zunächst das Modell mit konstanter Geschwindigkeit, da dieses mit nur vertretbarem Implementierungsaufwand verbunden ist. In diesem Fall sind die Daten über Rotationsgeschwindigkeit und Beschleunigungen nicht relevant, stattdessen übergeben wir die Daten über Geschwindigkeit, Ausrichtung und Position, die bereits vom OpenPilot EKF gefiltert wurden als Sensordaten an den RT-SLAM Filter.

Hierfür werden zwei Sensorklassen implementiert, zunächst hardwareSensorCameraOpenPilot::, eine Kameraklasse, die entsprechend den Überlegungen in 5.2.4.4 Bilddaten an RT-SLAM überträgt, sowie die Klasse hardwareSensorStateOpenPilot::, eine Sensorklasse für absolute Positionierung, die die Daten des PositionActual und AttitudeActual UAVObjekts an RT-SLAM weitergibt.

Dabei müssen wir unter anderem die Koordinatensysteme korrekt ineinander überführen. Das Koordinatensystem von RT-SLAM ist gegenüber dem von OpenPilot gedreht. OpenPilot verwendet NED (north east down) als globales Koordinatensystem, wohingehend RT-SLAM eine nach oben zeigende Z Achse aufweist.

Auch die Rotationsrichtungen müssen dabei überprüft werden.

Eine einfache Überprüfung der Sensorklassen war möglich indem zunächst ausschließlich der Positionssensor verwendet wurde, aber lediglich ein konstant schwarzes Bild an RT-SLAM übertragen wurde. Auf diesem werden keine Landmarken erkannt, daher folgt der Roboter ausschließlich seinem Bewegungsmodell und den Sensorinformationen, die sich nach einigen Korrekturen auch deckten, siehe Abschnitt 5.2.5.2. Dies zeigt auch, dass das Bewegungsmodell für unsere Zwecke zunächst hinreichend ist (Abbildung 5.23).

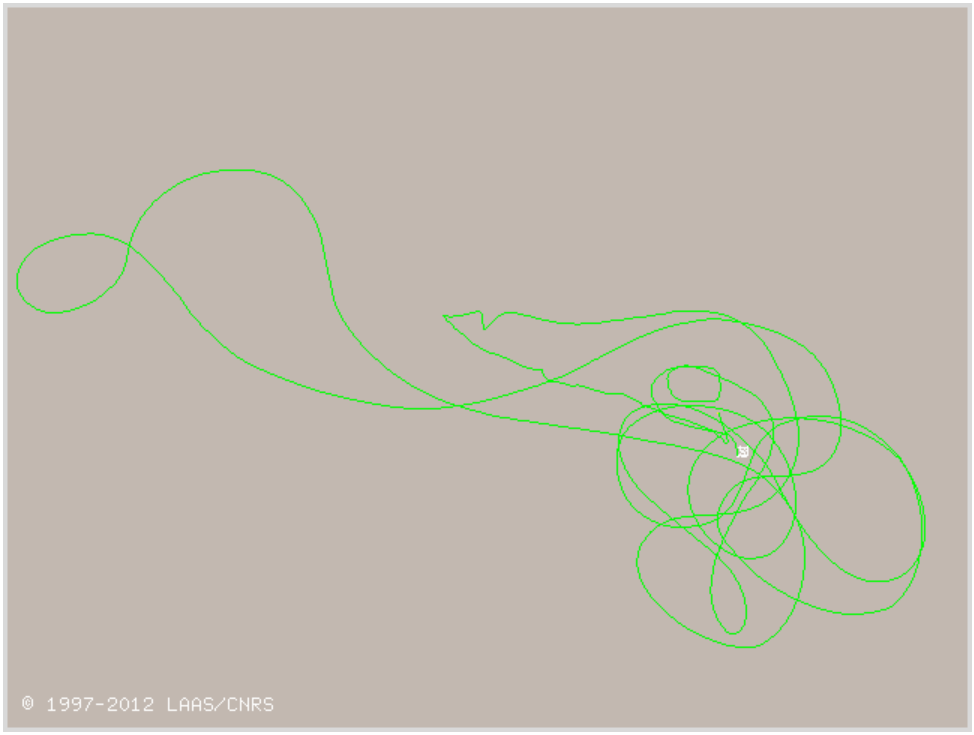

Abbildung 5.23: Visualisierung von RT-SLAM 3D Daten. Die Filterung basiert ausschließlich auf Positionsdaten des OpenPilot EKF, wenn die Kamera nur schwarze Bilder liefert.

5.2.7.2 Kameramodell und Kalibrierung

Im nächsten Schritt müssen wir nun die optischen SLAM Fähigkeiten testen.

Erste Tests verliefen frustrierend, da RT-SLAM wiederholt abstürzte. Dies konnte jedoch durch erhöhen der Stabilität behoben werden wie in Abschnitt 5.2.5.1 erläutert.

Jetzt fand RT-SLAM zwar Landmarken, aber die Positionierung war noch sehr ungenau, wodurch diese auch schnell wieder verworfen oder falschen Features zugeordnet wurden. Der Fehler lag sowohl in einem falsch eingeschätzten Kamerawinkel, als auch in einem unzutreffenden Kameramodell.

RT-SLAM modelliert Kameras als lineare Abbildungen gemäß einer Lochkamera mit fester Brennweite, die um ein polynomiales radiales Verzerrungsmodell ergänzt wird:

Sei $\vec{p} = \begin{Bmatrix} x \\ y \\ z \end{Bmatrix}$ die Position in 3D vor der Kamera, wobei die z-Achse Orthogonal zur Bildebene steht. $\begin{Bmatrix} hcenter \\ vcenter \end{Bmatrix}$ der Ursprung in der Bildebene gemäß des Kameramodells, und $\begin{Bmatrix} hfocal \\ vfocal \end{Bmatrix}$hfokal/vfocal die Brennweite in horizontaler und vertikaler Richtung in Pixeln und k_{1-n}die Verzerrungskoeffizienten, so ist die

Projektion $\vec{\gamma} = \begin{Bmatrix} x' \\ y' \end{Bmatrix} = \begin{Bmatrix} \frac{x}{z} \\ \frac{y}{z} \\ z \end{Bmatrix}$

Radius $r = \|\vec{\gamma}\|$

Radiale Verzerrung $v = 1 + \sum_{i\epsilon[1-n]} \left(r^{(2*i)} * k_i\right)$

Verzerrte Projektion $\vec{\delta} = \begin{Bmatrix} x" \\ y" \end{Bmatrix} = v * \vec{\gamma}$

Position in der Bildebene $\vec{p'} = \begin{Bmatrix} u \\ v \end{Bmatrix} = \begin{Bmatrix} hcenter + hfocal * x" \\ vcenter + vfocal * y" \end{Bmatrix}$

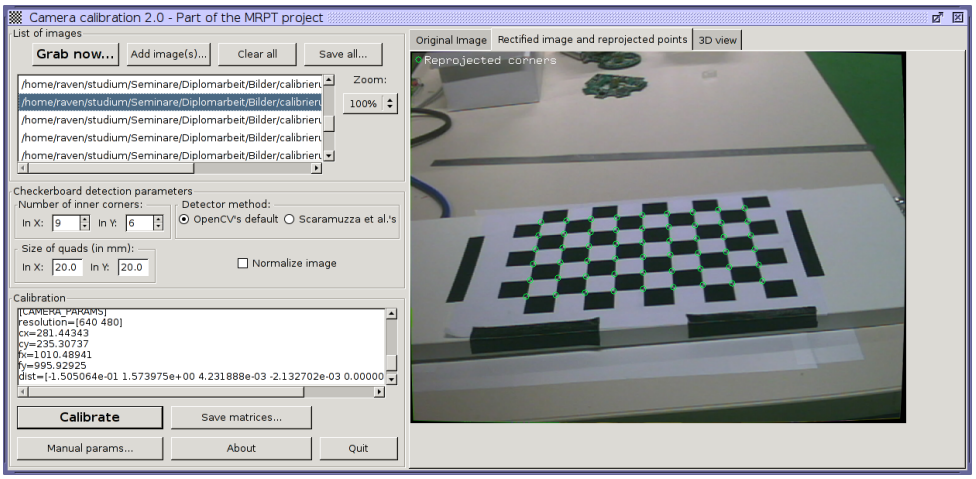

Abbildung 5.24: Kamerakalibrierung. Das Kalibrierungsprogramm des MRPT erkennt die Schnittpunkte in einem Schachbrettmuster...

Diese Berechnungen finden sich in RT-SLAM in der Datei include/rtslam/pinholeTools.hpp. Siehe Anhang A.2.

Sowohl die Brennweite, als auch die Koeffizienten des Verzerrungsmodells mussten nun bestimmt werden. Für diese Aufgabe gibt es diverse existierende Softwarelösungen, die meisten orientieren sich allerdings an dem Verzerrungsmodell von OpenCV, welches leider geringfügig komplexer ist als das von RT-SLAM, und zusätzliche Koeffizienten festlegt[unb12].

OpenCV bestimmt zusätzlich zur radialen Verzerrung v, gegeben durch Koeffizienten k_{1-3} eine Tangentiale Verzerrung, gegeben durch Koeffizienten $p_{1,2}$. Damit errechnet sich

$$\vec{\delta'} = \begin{Bmatrix} x" \\ y" \end{Bmatrix} = v * \vec{\gamma} + \begin{Bmatrix} 2p_1x'y' + p_2\left(r^2 + 2x'^2\right) \\ p_1\left(r^2 + 2y'^2\right) + 2p_2x'y' \end{Bmatrix}$$

Wir haben in diesem Fall leider keine andere Wahl, als die ermittelten tangentialen Koeffizienten zu ignorieren.

Wie sich herausstellte ist die Bestimmung, insbesondere der höherwertigen Koeffizienten an Hand von Beispielaufnahmen jedoch alles andere als eindeutig. Zwei Sätze an Beispielaufnahmen der selben Kamera liefern teilweise deutlich abweichende Koeffizientenkombinationen.

Die Kalibrierung mit dem Kalibrierungsprogramm des Mobile Robot Programming Toolkit (MRPT)[Bla11], welches das Kameramodell von OpenCV einsetzt, lieferte uns jedoch zumindest brauchbare und reproduzierbare Werte für die Brennweite (Abbildungen 5.24 und 5.25).

Was nun noch fehlt ist eine brauchbare Bestimmung der Winkelabweichung der Kamera.

Diese kann wie in 3.3.2 erwähnt nur optisch an Hand der Videobilder bestimmt werden. Der unter 3.3.5 implementierte gelbe Balken wird daher ersetzt durch eine Gitterdarstellung der D = 0 Ebene im NED Koordinatensystem, die im Idealfall parallel zum Erdboden verlaufen sollte. Dabei kommt eine Reimplementierung des RT-SLAM Kameramodells zum Einsatz die mit den ermittelten Faktoren bestückt wird.

Der Quellcode findet sich im Branch corvuscorax/rt-slam im Verzeichnis flight/Libraries/inc/geometrics/... (Anhang A.2).

Durch diese Visualisierung sind auch vergleichsweise kleine Abweichungen zwischen der gemessenen Ausrichtung und dem Videobild sichtbar, und können gegebenenfalls durch Nachjustierung am Kamerawinkel, korrigiert werden.

Dabei zeigen sich auch Fehler in der von OpenPilot aufgezeichneten Ausrichtung und Position, die Kamera kann also mit diesem Verfahren nicht genauer justiert werden als der Fehler der Positions- und Ausrichtungsbestimmung des Systems selbst. Auch Höhenunterschiede im Gelände wie Hügel und Täler erschweren die Abschätzung, so dass im Endeffekt nur eine auf etwa 1 Grad genaue Ausrichtung bestimmt werden konnte. Siehe Abbildungen 5.26 und 5.27.

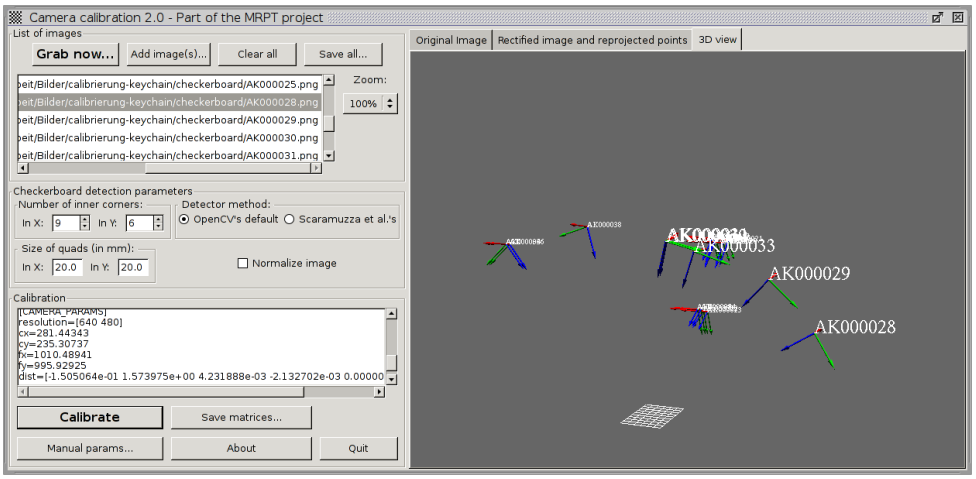

Abbildung 5.25: Kamerakalibrierung. Das Kalibrierungsprogramm des MRPT berechnet Brennweite und Verzerrungskoeffizienten für das Kameramodell von OpenCV, nachdem es die Positionen der Kamera relativ zum erkannten Muster bestimmt hat.

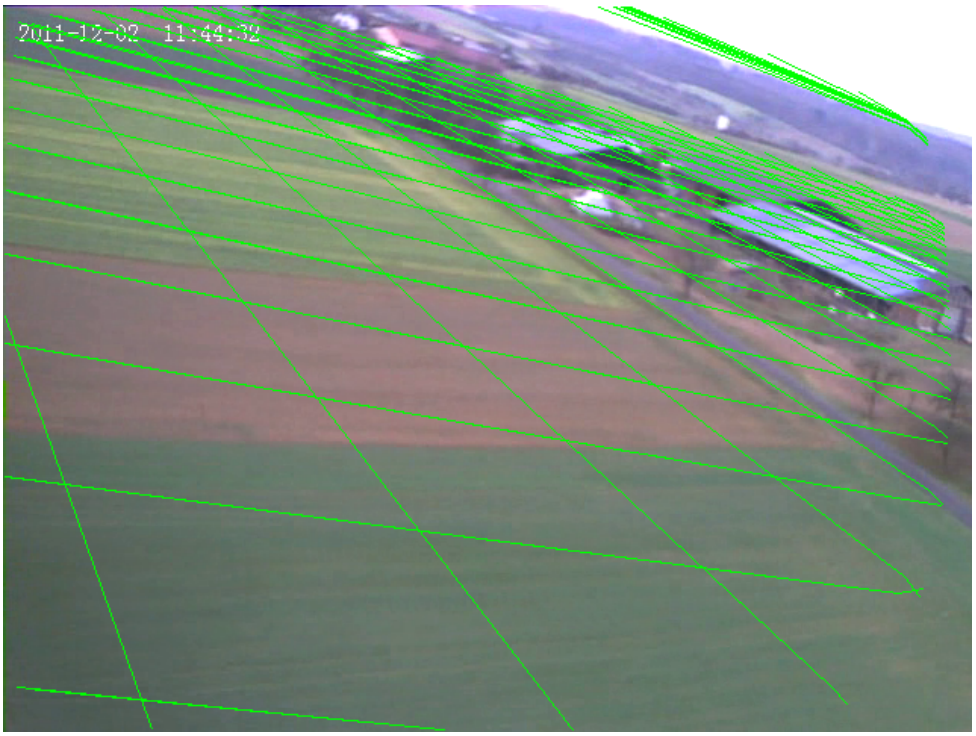

Abbildung 5.26: Kamerabild, überlagert mit einem Gitter für den Erdboden. Dieses erlaubt, insbesondere in der Bewegung, eine visuelle Kontrolle der Kamerakalibrierung, sowie eine Feinjustierung der Kamera-Ausrichtung.

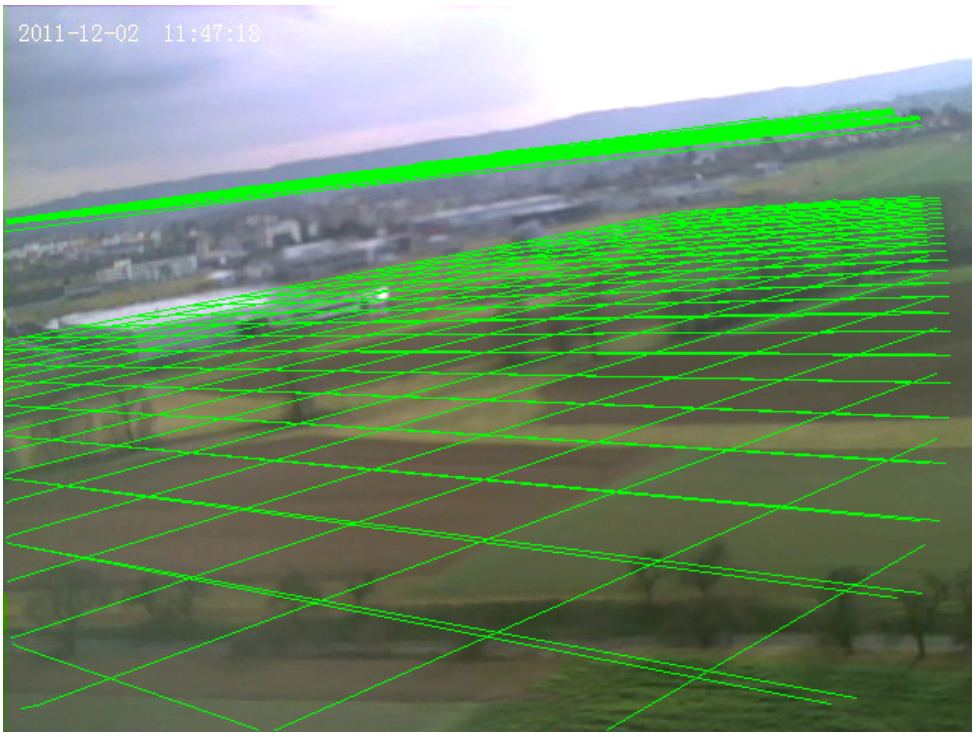

Abbildung 5.27: Kamerabild, überlagert mit einem Gitter für den Erdboden. Dieses erlaubt, insbesondere in der Bewegung, eine visuelle Kontrolle der Kamerakalibrierung, sowie eine Feinjustierung der Kamera-Ausrichtung.

5.2.7.3 Kartografierung

Nachdem nun alle implementierungsbedingten Hindernisse beseitigt wurden, kann die Fähigkeit zum Mapping beurteilt werden. Diese äußert sich in der Fähigkeit, Landmarken zu identifizieren, zu lokalisieren und wieder zu finden.

5.2.7.3.1 Tiefeninitialisierung von Landmarken Dabei mussten jedoch noch einige Konfigurationsanpassungen vorgenommen werden, siehe auch Abschnitt 5.2.6. Für rein optischen SLAM etwa spielt die initiale Entfernungsabschätzung für Landmarken eine untergeordnete Rolle. Diese geben mit einer willkürlich auf diesen Wert gesetzten Tiefe für die ersten gesichteten Landmarken schlicht die ansonsten allein mittels optischer Sensoren nicht erkennbare Skalenbasis der Karte vor. Weitere Landmarken werden damit automatisch mit einer vergleichbaren Tiefe initialisiert, welche damit meist zumindest in der richtigen Größenordnung liegt.

Anders bei optischem SLAM mit einer absoluten Referenzgröße, die in unserem Fall in Form der Positions- und Geschwindigkeitsmessung der Schnittstelle zum OpenPilot EKF vorliegt. In diesem Fall ist etwa die Standard-Tiefenschätzung von unter einem Meter völlig ungeeignet, da diese bei den geflogenen Geschwindigkeiten und der vorliegenden Bildwiederholrate zu einer völlig falschen Abschätzung der Landmarkenposition führt, und daher Landmarken auch nicht, bzw. falsch wiedergefunden werden.

Eine initiale Tiefenschätzung von etwa 75 Metern zeigte sich für die meisten Landmarken als sehr viel realistischer, und führt zu korrekter Verwaltung von Landmarken, die sich im Abstand von etwa 10 bis 200 Metern befinden. Jedoch werden auch entfernte Objekte wie Horizont und Wolken mit dieser Distanz initialisiert, und hier führt der Initialisierungswert letztendlich zu einem Verwerfen der Landmarke, da Erwartungs- und Messwerte zu weit voneinander entfernt liegen.

Zwar ist RT-SLAM mit einer geeignet gesetzten Initialabschätzung einsetzbar und funktioniert, stößt aber auch an sichtbare Grenzen.

5.2.7.3.2 Performance Bei weiteren Tests zeigte sich schnell, dass auch die ohne Performance-Einbußen verwaltbare Landmarkenmenge des EKF zu gering ist, um größere Außengebiete zu verwalten. RT-SLAM erkennt auch in den qualitativ minderwertigen Videobildern eine Vielzahl von Landmarken, die auch dank der von OpenPilot kommenden Positionierungsinformationen mit hinreichender Korrektheit in der Karte eingezeichnet werden. Jedoch stößt die durch die Kartengröße begrenzte Landmarkenmenge schnell an ihre Grenze. Das Problem ist auch, dass auf große Distanz erkannte Landmarken erst nach relativ langer Flugzeit aus ausreichend unterschiedlichen Winkeln genau genug lokalisiert werden können, damit RT-SLAM seine Reparametrisierung durchführt. Dies verbraucht weiteren wertvollen Platz in der Kovarianzmatrix, da nicht reparametrisierte Landmarken eine höhere Zustandsdimensionalität haben.

Den Reparamerisierungs-Threshold in 5.2.6.1 höher zu setzen führt jedoch zu Problemen, da zu früh reparametrisierte Landmarken wegen der fehlenden Information und hoher Varianz oft falsch zugeordnet werden. Hier fehlt in RT-SLAM ein Algorithmus um Landmarken auch aus abweichenden Winkeln noch zuverlässig wiedererkennen zu können.

Ab einer erlaubten Kartengröße von etwa 200 Zustandsvariablen kommt es in RT-SLAM zudem zu einem Effekt, bei dem keine Bearbeitung von Landmarken mehr stattfindet, obwohl der Positionszustand noch fortgeschrieben wird. Es handelt sich dabei möglicherweise um einen Implementierungsfehler, dessen Ursache aber jedoch in der zur Verfügung stehenden Zeit nicht mehr gefunden, und der daher nicht mehr behoben werden konnte. Die wahrscheinlichste Ursache ist, dass die Berechnung der Kovarianzmatrix nicht mehr im Zeitintervall eines Bildes möglich ist, dann jedoch nicht mehr angewandt wird, da bereits neue Daten vorliegen.

Wird die Karte dagegen auf einen kleineren Wert begrenzt, ist diese schon nach sehr kurzer Zeit, die im Video etwa einem Halbkreis beim Kurvenflug entspricht, voll. Danach werden nur noch vereinzelt neue Landmarken initialisiert, nämlich dann, wenn existierende Landmarken verworfen werden. Die findet in RT-SLAM wiederum nur dann statt, wenn Landmarken gemäß der Kameraausrichtung zwar sichtbar sein sollten, jedoch im Bild nicht gefunden werden. Dies bedeutet, dass eine große Anzahl momentan nicht sichtbarer Landmarken „hinter der Kamera“ RT-SLAM blockiert. Der Effekt tritt bei der erwähnten kreisförmigen Flugbahn zwangsläufig auf, sobald die erstmalig gesehenen Landmarken aus dem Bild ausgewandert sind.

Ein Ausweg ist, die Menge zu initialisierender Landmarken pro Bild zu begrenzen. In diesem Fall wird nur eine sehr kleine Menge an Landmarken überhaupt initialisiert und verfolgt, dies jedoch über den gesamten Flug. Die Karte ist dabei allerdings viel zu dünn besiedelt, um optische Navigation zu erlauben und demonstriert lediglich RT-SLAMs Fähigkeit, Landmarken mit den vorhandenen Daten zuverlässig zu verfolgen.

Demnach zeigt uns das erfolgreiche Mappen dieser wenigen Landmarken über einen längeren Zeitraum, dass der Ansatz generell funktioniert, wenn dessen Limitierungen umgangen werden können. Das heißt MonoSLAM ist prinzipiell ein geeigneter Ansatz um die vorliegenden Daten effizient zu verarbeiten. Der Algorithmus muss jedoch erweitert werden um die Fähigkeit weitere Landmarken, eventuell außerhalb der Kovarianzmatrix, in einer geeigneten Kartendarstellung zu speichern.

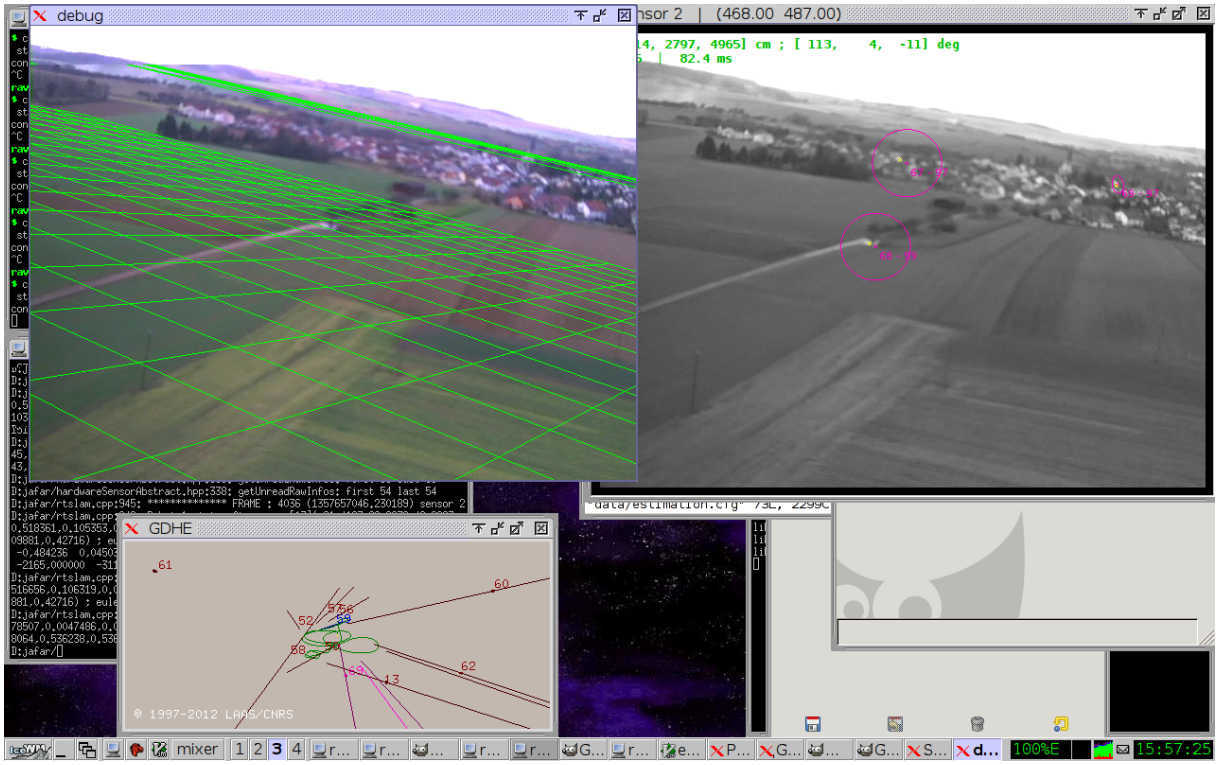

Abbildung 5.28: Echtzeit-SLAM auf den aufgezeichneten Daten mittels SimPosix, OpenCV und RT-SLAM.

Kapitel 6

Ergebnisse und Ausblick

Einen SLAM Algorithmus zu finden, der direkt und unmodifiziert für den Einsatz auf einem UAV geeignet ist, ist uns nicht gelungen, wir haben jedoch Ergebnisse erzielt, die darlegen welche Art von Algorithmus für diesen Zweck geeignet ist, und wie dieser implementiert werden muss.

6.1 Partielle Bildanalyse

Wie die Arbeiten zur Flussanalyse in 5.1.1 gezeigt haben, sind Verfahren die das Bild in jedem Schritt lückenlos bearbeiten, selbst bei relativ kleinen Auflösungen wie der verwendeten, nicht performant genug um auf den für diesen Zweck zur Verfügung stehenden Rechnerarchitekturen ohne Spezialhardware zu arbeiten. Der korrekte Ansatz ist dagegen der von RT-SLAM verfolgte, nämlich konsequent nur ausgewählte Bildbereiche zu betrachten, die entweder gestützt durch einen Schätzwert, beobachtete Objekte wie Landmarken enthalten, oder in denen gezielt nach diesen gesucht werden soll.

Dies erlaubt, auch bei größeren Auflösungen und mehreren Kameras, Computer Vision zu betreiben, denn abgesehen von den Ressourcen die für das Einlesen des oder der Bilder selbst benötigt werden sind wir nur begrenzt durch die Menge an Objekte oder Landmarken die in jedem Schritt verarbeitet werden können. Hier ist es jedoch wiederum nicht erforderlich in jedem Schritt alle Landmarken der Karte zu bearbeiten. Stattdessen sollte es ausreichen, eine relevante Untermenge dieser zu betrachten, etwa vergleichbar der Herangehensweise des PTAM Ansatzes[KM07].

6.2 Optischer Fluss auf RGB Template Matching

Wir haben, wie in Abschnitt 5.1.3.6.3 dargelegt, wertvolle Erkenntnisse erlangt, wie mit sehr einfachen Mitteln und auch performant ein lokaler optischer Fluss bestimmt werden kann. Diese Erkennung lässt sich im Rahmen einer ebenfalls Template basierten Identifizierung oder Lokalisierung von Landmarken durchführen und liefert zusätzliche Messwerte, die ein SLAM Verfahren schneller konvergieren lassen können.

Zudem erlaubt dieses Verfahren ein Arbeiten auf Farbbildern, was die meisten Algorithmen, inklusive dem Harris Detektor[HS88] in der bei RT-SLAM implementierten Form, nicht unterstützen. Dies erlaubt aber gerade in kontrastärmeren Umgebungen eine sehr viel eindeutigere Identifizierung von Landmarken.

6.3 EKF SLAM

EKF-SLAM, insbesondere mit den bei RT-SLAM verwendeten Optimierungen wie inverser Tiefenparametrisierung und Reparametrisierung, scheint in Zusammenhang mit Monokularem SLAM ein sehr vielversprechender Ansatz zu sein, auch für den Einsatz auf UAVs. Er muss aber erweitert werden um einen Ansatz der die Zahl verwaltbarer Landmarken stark erhöht.

Dies kann etwa erreicht werden indem zunächst eine geeignete Kartendarstellung für große Außenkarten, wie etwa ein Octree, gewählt wird[Pri11]. Aus dieser würden wir dann nur situationsbedingt einzelne Landmarken auswählen und in einer Kovarianzmatrix bearbeiten, wenn sich deren Zustandsabschätzung massiv verbessern lässt.

Dies hält den rechenaufwändigen erweiterten Kalman Filter (EKF) performant, ohne die Größe der Karte zu begrenzen. Eine geeignete und performante Auswahlstrategie der dafür verwendeten Landmarken muss jedoch noch gefunden werden.

Ein möglicher Ansatz wäre etwa, vergleichbar mit PTAM[KM07] MonoSLAM nur mit wenigen Landmarken durchzuführen, jedoch jede in MonoSLAM verwaltete Landmarke als eine Referenzmarke für eine Vielzahl in der Nähe befindlicher weiterer Landmarken eines separaten Mapping-Prozesses zu verwenden. Ändert sich die Position einer Referenzmarke entsprechend des EKF Korrekturschritts, würden an diese gekoppelte, in der Nähe befindliche Landmarken ebenfalls verschoben, womit Loop-Closing effizient implementierbar wäre.

Die von RT-SLAM durchgeführte Landmarken-Tiefenparametrisierung mit nur einer Starthypothese reicht nicht aus, wie wir in Abschnitt 5.2.7.3.1 gesehen haben. Bei Außenszenen, in denen Landmarken in Entfernungen von wenigen Zentimetern bis zu mehreren Lichtjahren entfernt sein können, müssen mehrere Hypothesen betrachtet werden. Uns bietet sich hier ein Ansatz an, bei dem etwa vergleichbar mit einem Partikelfilter mehrere Hypothesen in unterschiedlichen Größenordnungen betrachtet werden und somit auch Landmarken mit massiv abweichender Entfernung zuverlässig wiedergefunden werden können.

6.4 Integrierte Zustandsabschätzung

Die Lokalisierung an Hand von GPS lässt je nach Empfang deutlich zu wünschen übrig. Die Lokalisierung durch SLAM kann demnach bei brauchbaren Landmarken genauere Ergebnisse liefern, und sollte unbedingt in die Zustandsabschätzung des UAV überführt werden. Eine derartige Rückkopplung war bei der Evaluierung mittels aufgezeichneter Daten nicht möglich, ist aber beim Echtzeitbetrieb äußerst wünschenswert. Dies würde umgekehrt die vom UAV an den SLAM Filter geleiteten Positionsinformationen entsprechend dessen eigenen Erkenntnissen verbessern, selbst wenn dieser nur, wie in 5.2.7.1 implementiert, Position und Ausrichtung und nicht die hochfrequenten Rohdaten auswertet.

6.5 Codequalität und Softwarezuverlässigkeit

Viele Entwicklungen aus Forschungsprojekten, insbesondere aus dem akademischen Umfeld, haben eher Proof of Concept Charakter, als dass sie produktiv einsatzfähig wären. Obwohl RT-SLAM vom Ansatz und Einsatzzweck qualitativ deutlich ausgereifter anzusetzen ist als einige der Referenzimplementierungen etwa auf OpenSLAM.org [SFG12], ist dessen Stabilität nicht ausreichend für den Einsatz auf einem UAV, speziell wenn der SLAM Algorithmus für autonome Navigation eingesetzt werden soll.

Für diesen Zweck wird es daher unter Umständen unvermeidbar sein, einen Algorithmus von Grund auf neu zu implementieren, jedoch in deutlich zuverlässigerer Art und unter grundsätzlicher Berücksichtigung aller Ausführungszweige und Eventualitäten, um Fehlfunktionen wie sie bei RT-SLAM auftraten (5.2.5,5.2.7.3) von vornherein auszuschließen.

Auch die Vielzahl unterschiedlicher Technologien, die bei der Kombination von RT-SLAM mit SimPosix zum Einsatz kommen, wie Boost und FreeRTOS ist ungünstig und verkompliziert das Gesamtsystem, was auch das Timing-Verhalten schwer beherrschbar macht. Dies kann bei einer gezielten Neuimplementierung vermieden werden.

6.6 Ausblick

Der nächste logische Schritt wäre basierend auf den in dieser Arbeit gewonnenen Erkenntnissen die Ausarbeitung eines integrierten SLAM Verfahrens in Verbindung mit einer Kartenverwaltung für weitläufige Außenkarten.

- Dieses sollte in der Lage sein, Landmarken auch zu Oberflächen zu verbinden, wie dies ansatzmäßig bereits von RT-SLAM beherrscht wird. So können in einem späteren Schritt Hindernisse von ebenem Boden und geeigneten Landeplätzen unterschieden werden.
- Nur eine geeignete Unterauswahl von Landmarken sollte dabei zu jedem Zeitpunkt mittels eines Extended Kalman Filters oder einer vergleichbaren Technologie verarbeitet werden um deren Position exakter zu bestimmen, oder Loop-Closing zu betreiben.
- Die Initialisierung neuer Landmarken sollte bei Außenszenen mehr als eine Tiefenhypothese in unterschiedlichen Entfernungsgrößenordnungen aufstellen, da sonst die Ungenauigkeit bezüglich einzelner Landmarken zu groß ist, um diese wieder zu erkennen.
- Die Erkennung optischen Flusses kann dabei dazu dienen, sowohl die Qualität des SLAM zu verbessern, als auch bewegte Objekte zu erkennen, was für Kollisionsvermeidung relevant ist, jedoch sollte dieser nur an ausgewählten Positionen ausgewertet werden um den Rechenaufwand zu minimieren.
- Die Verarbeitung des gesamten Bildes sollte, analog zum Vorgehen von RT-SLAM, aus Performancegründen vermieden werden. Dies erlaubt den Einsatz qualitativ deutlich höherwertiger Kameras und höherer Auflösungen, bei gleichzeitiger Fokussierung der Berechnung auf Bildbereiche für die Landmarken und Hypothesen existieren. Es muss aber dennoch, vergleichbar mit dem in RT-SLAM verwendeten Harris Detektor, ein Verfahren existieren, das neue Landmarken in unbekannten Bildbereichen initialisiert.
- Wenn möglich sollten auch Farbinformationen mit ausgewertet werden, da diese gerade im Außenbereich monochrom sehr kontrastarme Strukturen optisch unterscheidbar machen.
- Besonderen Wert müssen wir dabei auf die Stabilität und Codequalität legen, da ein Softwarefehler an Board eines UAV wortwörtlich zum „Absturz“ führen kann. Aus der Luft- und Raumfahrt geläufige Verfahren der Softwarevalidierung und -verifizierung sollten dabei, soweit vertretbar, angewendet werden.

Ist dies gelungen, und existiert eine aus SLAM gewonnene Umgebungskarte, können sich daran weitere Arbeiten zur autonomen Navigation, Kollisionsvermeidung, Start- und Landemanöver und vergleichbares anschließen.

Anhang A

Quellcode

Der Quellcode dieser Arbeit findet sich, soweit Änderungen vorgenommen wurden, in den GIT[Swi08] Repositories des OpenPilot Projekts[Ope11], erreichbar unter der URL git://git.openpilot.org/OpenPilot.git.

A.1 Hierarchische Bestimmung des optischen Flusses basierend auf Template Matching

Der Code zu Abschnitt 5.1.1 findet sich im Branch corvuscorax/opencvslam, GIT Revision e7197fb20f52ce7c4a25021ac2dfb6f46047d1ac

Die Firmware kann mit make sim_posix gebaut werden, der relevante Code für das SLAM Evaluationsmodul findet sich im Unterverzeichnis

flight/Modules/SLAM

A.2 Implementierung von RT-SLAM

Der Code zu Kapitel 5.2 findet sich im Branch corvuscorax/rt-slam, GIT Revision 00cb658b9ec83f6c163444a415f216180f927933

Zusätzlich sind die modifizierten Quellen für RT-SLAM und dessen Abhängigkeiten erforderlich. Diese befinden sich in den Branches:

corvuscorax/rtslam/correl 309c0398f851170f48ae0b8cc12ed6c4e4bccde4

corvuscorax/rtslam/gdhe 8a77bb80c8ffb436f72301e571ddeb8ac9dfb055

corvuscorax/rtslam/jmath d4c956f3f0df3a1cc0f47a7a7bc0e1d91c76ce9a

corvuscorax/rtslam/qdisplay f8097ad2b79e4df9ac9ef2827af00041efc9d01c

corvuscorax/rtslam/rtslam 8b4769c23c5374e75e5f503a03741e28b122d9b9

und werden entsprechend des Howtos in flight/jafar/rtslam_howto.txt ausgecheckt. Dies gilt auch für die dort erwähnten weiteren Abhängigkeiten von RT-SLAM, hierbei sei auch auf die Installationsdokumentation von RT-SLAM verwiesen[RGS+11]. Die Verzeichnisse lauten dann etwa flight/jafar/rtslam bzw. flight/jafar/gdhe, etc.

Von den RT-SLAM Abhängigkeiten wurde jeweils der master Branch verwendet, zum Zeitpunkt der Verwendung waren dies die in der folgenden Tabelle (A.1) aufgelisteten Repositories und Revisionen.

Die Firmware kann mit make sim_posix gebaut werden, der relevante Code für das SLAM Evaluationsmodul findet sich ebenfalls im Unterverzeichnis

flight/Modules/SLAM

Typ	Repository	Branch	Revision	Tag
GIT	http://trac.laas.fr/git/robots/jafar/jafar.git	master	2500fccbd4895c4670c2273c868e5069efbd092c	
GIT	http://trac.laas.fr/git/robots/jafar/modules/image.git	master	30b7b0304c5960caa7878c49363976a9248dd41b	image-2.2
SVN[CSFP04]	http://svn.boost.org/svn/boost/sandbox/numeric_bindings	/	r79943	

Tabelle A.1: Quellcode des RT-SLAM Projekts

Abbildungsverzeichnis

Literaturverzeichnis

[AAWS11] Achtelik, M. ; Achtelik, M. ; Weiss, S. ; Siegwart, R.: Onboard IMU and monocular vision based control for MAVs in unknown in- and outdoor environments. In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on* IEEE, 2011, S. 3056–3063

[AHAS08] Allusse, Y. ; Horain, P. ; Agarwal, A. ; Saipriyadarshan, C.: GpuCV: an opensource GPU-accelerated framework forimage processing and computer vision. In: *Proceedings of the 16th ACM international conference on Multimedia* ACM, 2008, S. 1089–1092

[Aic07] Aichele, Fabian: *SLAM zur Navigation im Gebäude für Serviceroboter*, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, Diploma Thesis, September 2007. `http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2575&engl=1`. – 150 S.

[And99] Anderson, D.: *FireWire system architecture: IEEE 1394a*. Addison-Wesley Longman Publishing Co., Inc., 1999

[Ank09] Ankers, David et a.: *OpenPilot*. `http://openpilot.org/`. Version: 2009

[Axe99] Axelsson, P.: Processing of laser scanner data–algorithms and applications. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 54 (1999), Nr. 2-3, S. 138–147

[Bar03] Barry, R.: *Real Time Application Design Using FreeRTOS in small embedded systems*. 2003

[BB09] Brooks, A. ; Bailey, T.: HybridSLAM: Combining FastSLAM and EKF-SLAM for reliable mapping. In: *Algorithmic Foundation of Robotics VIII* (2009), S. 647–661

[BDW06] Bailey, T. ; Durrant-Whyte, H.: Simultaneous localization and mapping (SLAM): Part II. In: *Robotics & Automation Magazine, IEEE* 13 (2006), Nr. 3, S. 108–117

[BFB94] Barron, J.L. ; Fleet, D.J. ; Beauchemin, SS: Performance of optical flow techniques. In: *International journal of computer vision* 12 (1994), Nr. 1, S. 43–77

[BGWZ00] Bevly, D.M. ; Gerdes, J.C. ; Wilson, C. ; Zhang, G.: The use of GPS based velocity measurements for improved vehicle state estimation. In: *American Control Conference, 2000. Proceedings of the 2000* Bd. 4 IEEE, 2000, S. 2538–2542

[BK08] Bradski, G. ; Kaehler, A.: *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Incorporated, 2008

[Bla11] Blanco, JL: *Mobile Robot Programming Toolkit (MRPT)*. `http://mrpt.org/Application%3Acamera-calib`. Version: 2011

[Bou01] Bouguet, J.Y.: Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. In: *Intel Corporation* (2001)

[Bru09] Brunelli, R.: *Template matching techniques in computer vision: Theory and practice*. Wiley, 2009

[But97] Butenhof, D.R.: *Programming with POSIX threads*. Addison-Wesley Professional, 1997

[CMK03] Chum, O. ; Matas, J. ; Kittler, J.: Locally optimized RANSAC. In: *Pattern Recognition* (2003), S. 236–243

[CSFP04] Collins-Sussman, B. ; Fitzpatrick, B. ; Pilato, C.: *Version control with subversion.* O'Reilly Media, Incorporated, 2004

[CT03] Chamberlain, S. ; Taylor, I.L.: *Using ld: the GNU Linker.* 2003

[DRMS07] Davison, A.J. ; Reid, I.D. ; Molton, N.D. ; Stasse, O.: MonoSLAM: Real-time single camera SLAM. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2007), S. 1052–1067

[DWB06] Durrant-Whyte, H. ; Bailey, T.: Simultaneous localization and mapping: part I. In: *Robotics & Automation Magazine, IEEE* 13 (2006), Nr. 2, S. 99–110

[ENT05] Estrada, C. ; Neira, J. ; Tardós, J.D.: Hierarchical SLAM: Real-time accurate mapping of large environments. In: *Robotics, IEEE Transactions on* 21 (2005), Nr. 4, S. 588–596

[Eva77] Evans, D.J.: On the representatation of orientation space. In: *Molecular Physics* 34 (1977), Nr. 2, S. 317–325

[Far03] Farnebäck, G.: Two-frame motion estimation based on polynomial expansion. In: *Image Analysis* (2003), S. 363–370

[FP63] Fletcher, R. ; Powell, M.J.D.: A rapidly convergent descent method for minimization. In: *The Computer Journal* 6 (1963), Nr. 2, S. 163–168

[Fre07] Frese, U.: Efficient 6-DOF SLAM with treemap as a generic backend. In: *Robotics and Automation, 2007 IEEE International Conference on* IEEE, 2007, S. 4814–4819

[Fre12] Frese, Udo: *TreeMap.* `http://openslam.org/treemap.html`. Version: 2012

[Gla52] Glaser, AH: The pitot cylinder as a static pressure probe in turbulent flow. In: *Journal of Scientific Instruments* 29 (1952), S. 219

[Gol91] Goldberg, D.: What every computer scientist should know about floating-point arithmetic. In: *ACM Computing Surveys (CSUR)* 23 (1991), Nr. 1, S. 5–48

[GWA01] Grewal, M.S. ; Weill, L.R. ; Andrews, A.P.: *Global positioning systems, inertial navigation, and integration.* Bd. 2. Wiley Online Library, 2001

[HL97] Hanselman, D. ; Littlefield, B.C.: *Mastering MATLAB 5: A comprehensive tutorial and reference.* Prentice Hall PTR, 1997

[HS88] Harris, C. ; Stephens, M.: A combined corner and edge detector. In: *Alvey vision conference* Bd. 15 Manchester, UK, 1988, S. 50

[Jac07] Jack, K.: *Video demystified: a handbook for the digital engineer.* Newnes, 2007

[Jäh05] Jähne, B.: *Digitale Bildverarbeitung.* Springer Verlag, 2005

[KB06] Kannala, J. ; Brandt, S.S.: A generic camera model and calibration method for conventional, wide-angle, and fish-eye lenses. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 28 (2006), Nr. 8, S. 1335–1340

[KM07] Klein, G. ; Murray, D.: Parallel tracking and mapping for small AR workspaces. In: *Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on* IEEE, 2007, S. 225–234

[Kon12] Kontron: *Kontron web shop.* `http://emea.kontron.com`. Version: 2012

[LF96] Liu, L.K. ; Feig, E.: A block-based gradient descent search algorithm for block motion estimation in video coding. In: *Circuits and Systems for Video Technology, IEEE Transactions on* 6 (1996), Nr. 4, S. 419–422

[Mág84] Mágori, V.: Ultraschall-Distanzsensoren zur Objektidentifizierung und Lageerkennung. In: *VDI-Berichte* 509 (1984), S. 27–31

[Mal96] Malys, S.: The WGS84 Reference Frame. In: *National Imagery and Mapping Agency* (1996)

[MKG+97] Mendis, S.K. ; Kemeny, S.E. ; Gee, R.C. ; Pain, B. ; Staller, C.O. ; Kim, Q. ; Fossum, E.R.: CMOS active pixel image sensors for highly integrated imaging systems. In: *Solid-State Circuits, IEEE Journal of* 32 (1997), Nr. 2, S. 187–197

[MTH+12] Meier, L. ; Tanskanen, P. ; Heng, L. ; Lee, G.H. ; Fraundorfer, F. ; Pollefeys, M.: PIXHAWK: A micro aerial vehicle design for autonomous flight using onboard computer vision. In: *Autonomous Robots* (2012), S. 1–19

[NLS+12] Nuechter, Andreas ; Lingemann, Kai ; Sprickerhof, Jochen ; Borrmann, Dorit ; Elseberg, Jan ; Schneider, Peter ; Qui, Deyuan: *SLAM6D*. `http://openslam.org/slam6d.html`. Version: 2012

[Nüc09] Nüchter, A.: *3D robotic mapping: the simultaneous localization and mapping problem with six degrees of freedom*. Bd. 52. Springer, 2009

[Ope11] OpenPilot: *OpenPilot GIT Repository*. `http://wiki.openpilot.org/display/Doc/Getting+the+code`. Version: 2011

[Pas02] Paskin, M.A.: *Thin Juinction Tree Filters for Simultaneous Localization and Mapping*. Berkley, 2002

[Pas12] Paskin, Mark A.: *Thin Junction Tree Filters for SLAM*. `http://openslam.org/tjtf.html`. Version: 2012

[Pra80] Prazdny, K.: Egomotion and relative depth map from optical flow. In: *Biological cybernetics* 36 (1980), Nr. 2, S. 87–102

[Pri11] Price, Eric: *Erstellung von weiträumigen Umgebungskarten mit Hilfe von Octrees*. 9 2011

[Pri12] Price, Eric: *Evaluierung von Verfahren zur Bilddatenerfassung zum autonomen Mapping mit UAVs*. Studienarbeit: Universität Stuttgart, Institut für Parallele und Verteilte Systeme, Bildverstehen. Version: Januar 2012. `http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=STUD-2344&engl=0`

[RGS+11] Roussillon, C. ; Gonzalez, A. ; Solà, J. ; Codol, J.M. ; Mansard, N. ; Lacroix, S. ; Devy, M.: RT-SLAM: a generic and real-time visual SLAM implementation. In: *Computer Vision Systems* (2011), S. 31–40

[Sch10] Schinstock, D.E.: *GPS-aided INS Solution for OpenPilot*. `http://wiki.openpilot.org/display/Doc/INSGPS+Algorithm`. Version: 2010

[SDVR99] Schimek, M.H. ; Dirks, B. ; Verkuil, H. ; Rubli, M.: Video for Linux Two API Specification. (1999)

[SFG12] Stachniss, Cyrill ; Frese, Udo ; Grisetti, G: *OpenSLAM.org*. `http://openslam.org/`. Version: 2012

[SKRS11] Senthil Kumar, K. ; Ramesh, G. ; Srinivasan, KV: First Pilot View (FPV) Flying UAV Test Bed for Acoustic and Image Data Generation. (2011)

[SKSZ12] Schauwecker, K. ; Ke, N.R. ; Scherer, S.A. ; Zell, A.: Markerless Visual Control of a Quad-Rotor Micro Aerial Vehicle by Means of On-Board Stereo Processing. In: *Autonomous Mobile Systems 2012* (2012), S. 11–20

[Sol10] Sola, J.: Consistency of the monocular ekf-slam algorithm for three different landmark parametrizations. In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on* IEEE, 2010, S. 3513–3518

[SRR92] Stevens, W.R. ; Rago, S.A. ; Ritchie, D.M.: *Advanced programming in the UNIX environment*. Bd. 4. Addison-Wesley New York., 1992

[SS94] Stricker, M. ; Swain, M.: The capacity of color histogram indexing. In: *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on* IEEE, 1994, S. 704–708

[Str97] Stroustrup, B.: *The C++ programming language.* Addison-Wesley Longman Publishing Co., Inc., 1997

[Swi08] Swicegood, T.: *Pragmatic version control using Git.* Pragmatic Bookshelf, 2008

[Tri94] Trimberger, S.: *Field-programmable gate array technology.* Springer, 1994

[unb12] unbekannt: *Camera Calibration and 3D Reconsruction.* `http://opencv.willowgarage.com/documentation/camera_calibration_and_3d_reconstruction.html`. Version: 2012

[Uns00] Unser, M.: Sampling-50 years after Shannon. In: *Proceedings of the IEEE* 88 (2000), Nr. 4, S. 569–587

[War05] Ward, M.: Developing video phones with ARM processor-based solutions. In: *Technology in-depth* 4 (2005), Nr. 4, S. 16–18

[WB95] Welch, G. ; Bishop, G.: *An introduction to the Kalman filter.* 1995

[WKW02] Walter, J. ; Koch, M. ; Winkler, G.: *uBLAS: boost basic linear algebra.* `http://www.boost.org/doc/libs/1_52_0/libs/numeric/ublas/doc/index.htm`. Version: 2002

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfaßt und nur die
angegebenen Quellen benutzt zu haben.

_____

(Eric Price)