Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Falk Böschen

Geschwindigkeitsoptimierung eines visuell geführten Knickarmroboters

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Falk Böschen

Geschwindigkeitsoptimierung eines visuell geführten Knickarmroboters

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Andreas Meisel
Zweitgutachter: Prof. Dr.-Ing. Franz Korf

Abgegeben am 30. August 2011

Falk Böschen

Thema der Bachelorarbeit

Geschwindigkeitsoptimierung eines visuell geführten Knickarmroboters

Stichworte

Visual Servoing, Optical Flow, Lucas Kanade, SIFT, SURF, OpenCV, Matlab

Kurzzusammenfassung

Im Bereich der Robotik wird zur Steuerung häufig auf optische Sensoren als Input zurückgegriffen. Optische Sensoren produzieren jedoch einen hohen Datendurchsatz. Um eine echtzeitfähige Steuerung zu erreichen werden in dieser Arbeit mehrere Ansätze zur Objektverfolgung mit Verfahren des Optischen Flusses verglichen. Die effektivste Verfolgung wird dann mit einer Merkmalsdetektion kombiniert. Diese Synthese wird im Anschluss in eine Testumgebung, für die Simulierung eines Anwendungsfalles, implementiert.

Title of the paper

Speed optimization of a visually guided articulated arm robot

Keywords

Visual Servoing, Optical Flow, Lucas Kanade, SIFT, SURF, OpenCV, Matlab

Abstract

Optical sensors are often used in robotics as data input for controlling. However, these sensors produce a high data throughput. In this work, several approaches for object tracking using optical flow methods are compared, to achieve a real-time control. The most effective method from these tests is combined with a feature detection algorithm. Finally the synthesized function is tested in a given use-case szenario.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Informatik hat in den vergangenen 20 Jahren viele Bereiche unseres Lebens verändert. Durch die Einführung des Internets am 06. August 1991 wurden neue Wege zur Kommunikation geschaffen und eine gigantische Welt voller Informationen eröffnet. Auf diese Weise konnten soziale Netzwerke, wie beispielsweise Facebook, das gesamte Leben vernetzen. Durch den in den letzten Jahren steigenden Absatz von Smartphones ist auch die mobile Vernetzung stark gestiegen und somit ist die Informatik im alltäglichen Leben allzeit präsent.

Im Fachgebiet der Robotik wurden seit den 70er Jahren, in denen die ersten Roboter, primär in der Industrie, zum Einsatz kamen, diverse Fortschritte erreicht. Kleinere Roboter, die auf spezielle Einsatzgebiete beschränkt sind, werden heutzutage schon in Serie produziert. Doch selbst diese arbeiten nach aktuellen Tests [1] immer noch nicht fehlerfrei. Die gewünschte Zeitersparnis durch den Einsatz von Haushaltsrobotern wird durch die Schwächen in Technik und Software häufig verhindert. Eine wirkliche Revolution unseres alltäglichen Lebens, wie in der (digitalen) Kommunikation und Vernetzung, ist deswegen bisher ausgeblieben. Derzeit haben Roboter oftmals den Status eines Spielzeugs und nicht den eines praktischen Helfers. Ein weiterer Faktor der die Verbreitung von Robotern im normalen Leben hemmt, ist dass die steigende Autonomie der Roboter die Sicherheit der Umgebung gefährdet und zusätzlicher Entwicklungsaufwand erforderlich ist.

Die meisten Roboter vollführen momentan nur fest einprogrammierte Abläufe und können nicht auf Umgebungsänderungen reagieren. Um eine höhere Autonomie der Roboter zu erreichen, wird häufig auf optische Sensoren als Dateninput zurückgegriffen, da diese kostengünstig erwerbbar sind und einen hohen Datendurchsatz haben. Eine Behinderung oder Gefährdung von Menschen muss dementsprechend durch den Einsatz von echtzeitfähigen Algorithmen vermieden werden, was bei dem hohen Datendurchsatz optischer Sensoren eine Herausforderung ist.

[1] c't Magazin für Computer Technik Heft 18 vom 15.08.2011

1.2 Idee

In dieser Arbeit wird auf eine vorangegangene Masterarbeit von Benjamin Wagner [Wag11] aufgebaut. Diese realisiert einen Hybriden Visual Servoing Ansatz in der Entwicklungsumgebung MATLAB. Mit Visual Servoing werden Regelungsverfahren bezeichnet, die als Dateninput auf Optische Sensoren zurückgreifen, um Roboter zu steuern. Die Realisierung strukturiert einen einfachen Greifvorgang in vier Abschnitte: Suchen, Grobanfahrt, Feinanfahrt und Greifen. Für die Feinanfahrt wurde eine bildbasierte Regelung verwendet, die jedoch keine Echtzeitfähigkeit erlaubt. Dies war allerdings auch nicht der Schwerpunkt dieser Arbeit, sondern der Funktionsnachweis der entwickelten Methodik, unabhängig von der Performance des Systems. Basierend auf der Neulokalisierung der Merkmale in jedem Bild durch den SIFT Algorithmus, wurde ein stabiles Verfahren implementiert, anstatt sich mit der Problematik instabiler Verfolgungsverfahren zu beschäftigen. Im Rahmen dieser Arbeit soll nun genau diese Problematik analysiert und Möglichkeiten zur Geschwindigkeitssteigerung untersucht und realisiert werden. Dabei wird auf die Methoden des Optischen Flusses zurückgegriffen.

1.3 Aufbau der Arbeit

In dieser Ausarbeitung werden zunächst die soft- und hardwaretechnischen Grundlagen kurz dargestellt, um für die folgenden Tests die Funktionsbasis bereitzustellen. Das nächste Kapitel beschäftigt sich mit den Ansätzen der Erkennung von Objekten und stellt einen Vergleich zwischen zwei merkmalsbasierten Algorithmen auf. Die anschließende Sektion behandelt den Optischen Fluss, dessen Grundlagen und verschiedenen Implementierungen basierend auf den differentiellen Verfahren von Horn-Schunk und Lucas-Kanade. Es wird ebenfalls ein Vergleich verschiedener Implementierungen vorgenommen, um die bestmögliche Verfolgung zu realisieren. Der Schwerpunkt liegt dabei auf dem Vergleich der benötigten Zeiten und der Robustheit der Methoden. Basierend auf den Testergebnissen wird eine Synthese der effektivsten Komponenten in eine kombinierte Objektverfolgung realisiert. Die mit Hilfe des Optischen Flusses verfolgten Punkte, werden in regelmäßigen Abständen relokalisiert. Im Anschluss erfolgt noch die Integration der Funktion in die vorhandene Software [Wag11], um ein Testszenario in einer realitätsnahen Umgebung zu simulieren. Dies soll die Leistungsfähigkeit der Entwicklung verifizieren. Abschließend wird noch ein Ausblick auf mögliche Weiterentwicklungen und Anwendungsszenarios gegeben.

2 Einrichten der Test und Entwicklungsumgebung

In diesem Kapitel wird die Entwicklungs- und Testumgebung dargestellt, die in den weiteren Abschnitten verwendet wird.

2.1 Hardware

Die Tests werden auf einem System mit einem Intel i5-430m Prozessor (2,26 Ghz, 3MB L3 Cache) und 4 GB RAM unter Windows7 in MATLAB durchgeführt.

Das Zielsystem besteht aus einem Intel D 2 x 3,0 Ghz Prozessor mit 2 GB RAM unter Windows XP. Im Weiteren wurde die nachfolgende Hardware für den späteren Integrationstest in die Arbeit von [Wag11] genutzt. Für die anderen Testabläufe wurde auf Bilder der in dem System installierten Kamera zurückgegriffen.
Die Basis bildet ein Knick-arm Roboter des Typs Katana 450-6M180 von Neuronics.

Abbildung 1: Katana 450-6M180 mit Kontrollbox

Der Katana besitzt fünf Drehgelenke, die durch eine Kette von Gliedern verbunden sind. Auf eine Erläuterung der Funktionsweise der Kinematik wird in dieser Arbeit verzichtet, da der Schwerpunkt dieser Arbeit im Bereich der Bildverarbeitung liegt und die Steuerung des Katana aus der Arbeit von Benjamin Wagner übernommen wird.

Am letzten Glied des Knickarm-Roboters befindet sich ein zweiteiliger Greifer, der über ein sechstes Drehgelenk gesteuert werden kann. Der Greifer besitzt keinerlei Art von Sensoren und kann ein Gewicht von bis zu 250 g heben.

Abbildung 2: Greifer (3D)

Auf Grund der am Endeffektor montierten Kamera muss das maximale Tragegewicht jedoch auf 80 g reduziert werden. Der erreichbare Arbeitsraum wird in der nächsten Abbildung dargestellt.

Abbildung 3: Katana Arbeitsbereich (3D)

Zur Ansteuerung stellt Neuronics eine C++-Bibliothek (Katana Native Interface kurz KNI) zur Verfügung. Über diese Schnittstelle lassen sich die Gelenkeinstellungen direkt modifizieren.

Die am Endeffektor montierte Kamera ist vom Modell Guppy F-080B. Es handelt sich somit um ein Eye-in-Hand System. Die Guppy ist in der Lage 30 Bilder pro Sekunde mit 1032 x 778 Pixeln und einer schwarz-weiß Darstellung zu liefern. Das Gewicht der Kamera beträgt 50 g.

Abbildung 4: Guppy F-080B

Als Objektiv für die Kamera ist ein Produkt der Firma PENTAX mit der Bezeichnung H416 (KP) angebaut. Es handelt sich um ein Weitwinkelobjektiv mit 4,2 mm Brennweite und 120 g Gewicht.

Abbildung 5: Pentax H416 (KP)

2.2 Software

In dieser Ausarbeitung werden zwei Entwicklungsumgebungen eingesetzt. Zum einen Visual Studio zur Entwicklung von Anwendungen in C/C++ und zum anderen MATLAB, denn in dieser ist die Arbeit von Benjaming Wagner [Wag11] realisiert. Als Schnittstelle zwischen den beiden Programmiersprachen dienen sogenannte MEX-Dateien. Diese kapseln C/C++-Code derartig, dass ein Aufruf aus MATLAB möglich wird. Neben den Standard C-Bibliotheken wird noch eine weitere frei verfügbare Bibliothek zur Bildverarbeitung, die OpenCV-Library, eingebunden. Dies erfordert erweiterten Aufwand bei der Einrichtung der Entwicklungsumgebung.

2.2.1 Visual Studio

Als C-Entwicklungsumgebung wurde in dieser Arbeit Visual Studio 2010 Ultimate genutzt. Die OpenCV Bibliothek kann ab Version 2.1 genutzt werden. Am einfachsten ist es den Win32 Installer [2] zu nutzen. Bei der Installation ist darauf zu achten, dass die Option “add to Path for all Users” gewählt wird.

Im Folgenden sind nun die Schritte aufgelistet, die notwendig sind damit MEX-Dateien erstellt werden können :

1. Visual Studio starten
2. Erstellung eines Projektes für eine C++ Win32 Konsolenanwendung
3. Umstellung der Konfiguration auf “Release”
4. Einrichtung der Projekteigenschaften gemäß Anhang B.4
5. Moduldefinitionsdatei (B.3) zum Projekt hinzufügen (Projekt $\rightarrow$ Neues Element hinzufügen $\rightarrow$ DEF-Datei)
6. Einbindung des Headers “mex.h” mit vorangehender ifdef Anweisung (B.1 Z.3-5) um Dateitypenkonflikt mit “yvals.h” (nur unter VS2010) zu vermeiden
7. Ersetzung der Main-Methode durch den MEX-Funktionskopf (vgl. B.1 Z.19)
8. Erstellung einer weiteren C/C++ Datei, sowie eines Headers um beide zu verbinden
9. Einbindung der benötigten OpenCV-Header (Standardmäßig “cxcore.h”, “cv.h” und “highgui.h”) in die zweite Datei

[2] `http://sourceforge.net/projects/opencvlibrary/files/opencv-win/`

Nach Abschluss der Einrichtung der Entwicklungsumgebung kann man die beiden Dateien mit der benötigten Funktionalität füllen. Die erzeugten Dateien sollten im Projekt Ordner unter Release zu finden sein. Von den erstellten Dateien werden nur die .mexw32 und .lib Datei benötigt. Diese müssen anschließend in den MATLAB Projektordner kopiert werden.

2.2.2 MATLAB

Als weitere Entwicklungs- und Testumgebung wurde auf MATLAB in der Version R2009b(32Bit) zurückgegriffen, damit eine Kopplung mit den eingebundenen 32-Bit OpenCV Funktionen möglich ist. Aufgrund der durchgeführten Einstellungen in Visual Studio sind keine weiteren Anpassungen in MATLAB nötig.

Alternativ gibt es auch die Möglichkeit die Mex-Datei teilweise mit MATLAB zu kompilieren. In dem Fall würde die Einbindung in Visual Studio entfallen und es müsste mit Visual Studio nur die .obj-Datei erzeugt werden, die die OpenCV Funktionen enthält. Danach muss in MATLAB nur noch erst mit *mex -setup* der Compiler ausgewählt werden. Im Anschluss kann mit dem Befehl *mex mex.cpp opencv.obj* die ausführbare .mexw32-Datei erzeugt werden. In dieser Ausarbeitung wird auf diesen Entwicklungspfad verzichtet und es wird für die Kreierung der Mex-Datei nur auf Visual Studio zurückgegriffen, aufgrund der einfacheren Erzeugung und besseren C/C++ Umgebung.

Damit die Datei verwendet werden kann, müssen auf dem Zielsystem, falls es sich vom Entwicklungssystem unterscheidet, alle Abhängigkeiten vorhanden sein. Befindet sich die Mex-Datei nun im Projektordner auf dem Zielsystem, muss dafür gesorgt werden, dass dort ebenfalls, falls nicht vorhanden, die OpenCV Bibliothek installiert wird. Alternativ benutzt man das Programm *Dependency Walker* [3]. Es kann direkt ohne Installation gestartet werden. Unter Datei $\rightarrow$ Öffnen wird die Mex-Datei ausgewählt und das Programm teilt einem die fehlenden Abhängigkeiten mit. Diese DLLs (Dynamic Link Library) müssen per Hand direkt in den Projektordner kopiert werden.

[3] `http://www.dependencywalker.com`

3 Erkennung von Objekten

Zu Beginn soll hier auf die Objekterkennung genauer eingegangen werden, denn ohne die Erkennung ist eine Verfolgung des Objektes nicht realisierbar.

Es gibt verschiedene Verfahren um Objekte zu registrieren. Die Identifikation lässt sich über Farbe, Form, spezielle Objektmerkmale oder über extra am Objekt angefügte Marker realisieren. Eine Lokalisierung des Objektes im Bild ist bei eindeutigen Farben schnell und einfach realisierbar, jedoch setzt dies voraus, dass eine Farbkamera vorhanden ist und jedes Objekt eine einmalige Farbgebung hat. Die Extrapolation von Formen ist ein sehr aufwändiges Verfahren und wird nur selten eingesetzt. Auch das Identifizieren von zusätzlich angebrachten Markern ist nicht alltagstauglich, weshalb hier im Folgenden auf die Erkennung von Objekten mit Hilfe von Merkmalen weiter eingegangen wird. Aus diesem Bereich werden zwei Verfahren dargestellt und getestet. Als Erstes wird das SIFT Verfahren betrachtet und im Anschluss die daraus entstandene Weiterentwicklung SURF.

3.1 SIFT

Der Begriff SIFT steht für Scale Invariant Feature Transform und bezeichnet damit ein Verfahren das Merkmale aus einem Bild extrapoliert, die auch aus anderen Ansichten wiedererkannt werden können. Das Verfahren wurde 1999 von David Lowe publiziert [Low99].
Der Algorithmus bestimmt sogenannte Features oder Merkmale, die invariant in Hinsicht auf Translation, Skalierung und Rotation, sowie teil-invariant gegenüber Beleuchtungsänderungen, affinen und homographischen Projektionen sind. Invarianz bedeutet in diesem Sinne, dass die genannten Bildmodifikationen keinen Einfluss auf die Wiedererkennung der Merkmale haben. Außerdem wird der Einfluss von Bildrauschen durch die Anwendung von Gaußfiltern verringert.

Das Verfahren beginnt damit, dass aus einem Bild I eine Gauss Pyramide L aufgebaut wird. Hierzu wird das Bild mit dem Gaussfilter $g(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-x^2}{2\sigma^2}}$ bearbeitet, bei einem Faktor σ von $\sqrt{2}$, der bei jeder weiteren Stufe mit dem Faktor $k = \sqrt{2}$ multipliziert wird. Dies bildet die Abbildung 6 ab.

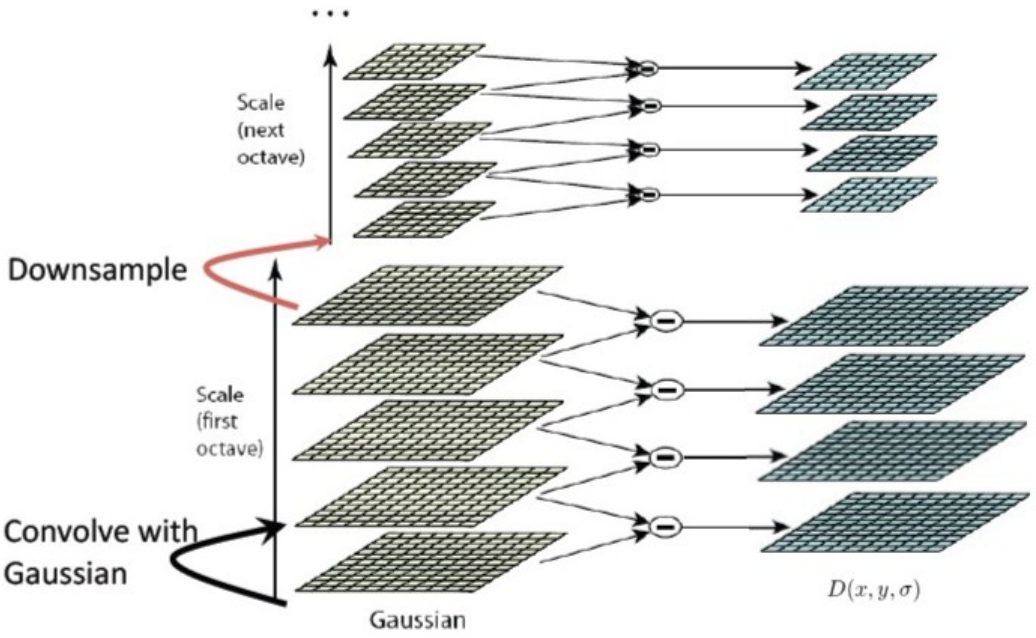


Abbildung 6: SIFT Gauss Pyramide

Aus der Gausspyramide wird dann eine Pyramide von *Difference of Gaussian (DoG)* erzeugt, indem zwei Gaussebenen von einander subtrahiert werden : $D(x,y,\sigma) = L(x,y,k_i\sigma) - L(x,y,k_j\sigma)$. Sobald diese DoG-Pyramide erstellt ist, werden dort Minima und Maxima über alle Ebenen gesucht.

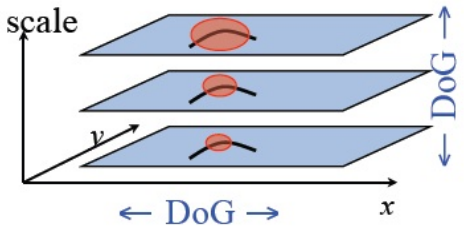


Abbildung 7: Suche in DoG in Raum und Skalierung

Die Bestimmung von Maxima und Minima erfolgt durch den Vergleich mit der direkten Acht-Punkte-Nachbarschaft auf der gleichen Ebene, sowie mit der Neun-Punkte-Nachbarschaft auf der darüber und der darunter liegenden Ebene. Ist der Wert ein Maximum oder Minimum, dann wird er zur Kandidatenliste hinzugefügt. Aus dieser Liste werden im Anschluss die schwachen Merkmale entfernt, indem die Punkte mit der quadratischen Taylorfunktion interpoliert werden.

Sind die Positionen der Merkmale bekannt wird der sogenannte *Feature Descriptor* aufgebaut. Dazu werden zunächst die Orientierung und die Stärke der Features berechnet :

$$\theta(x,y) = \arctan\left(\frac{L(x,y+1)-L(x,y-1)}{L(x+1,y)-L(x-1,y)}\right)$$

$$m(x,y) = \sqrt{(L(x+1,y)-L(x-1,y))^2+(L(x,y+1)-L(x,y-1))^2}$$

Sind diese Werte für jeden Pixel berechnet, wird die 16x16 Umgebung um jedes Merkmal genommen und die Haupt-Orientierung berechnet. Schwache Orientierungen, die unterhalb eines gesetzten *threshold* (Grenzwert) sind, werden nicht betrachtet. Dieses 16x16 Fenster wird dann auf ein 4x4 Fenster projiziert, sodass für jede Zelle des 4x4 Fensters eine Orientierung bestimmt wird. Die folgende Abbildung 8 zeigt dies für einen 2x2 Ausschnitt.

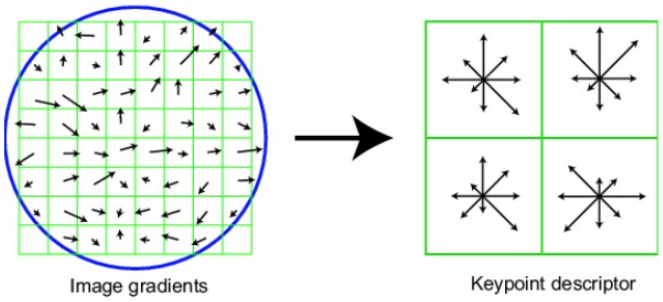


Abbildung 8: SIFT Feature Descriptor 2x2 Ausschnitt

Dies ergibt dann ein 4x4 Fenster, dass somit aus 16 Zellen mit jeweils acht Orientierungen besteht und somit hat der *Feature Descriptor* eine Dimension von $16 \times 8 = 128$.

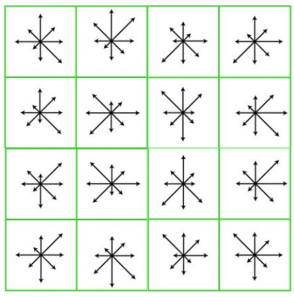


Abbildung 9: SIFT Feature Descriptor (vollständig)

Mit Hilfe dieses 128-dimensionalen Vektors lässt sich der Punkt eindeutig identifizieren und wiedererkennen.

3.2 SURF

Dieses Unterkapitel erläutert kompakt das Prinzip einer Weiterentwicklung des SIFT Algorithmus, das Speeded Up Robust Features Verfahren, kurz SURF. Es wurde 2006 von Herbert Bay, Andreas Ess, Tinne Tuytelaars und Luc Van Gool in *Proceedings of the 9th European Conference on Computer Vision* vorgestellt und dann zwei Jahre später erneut in [BETG08]. Das Verfahren entwickelt die Methodik des SIFT Algorithmus dahingehend weiter, dass die Gaussfilter durch Mittelwertfilter ersetzt werden. Durch die Verwendung von Integralbildern lassen sich die Merkmale in konstantem Zeitaufwand berechnen und erlauben somit eine schnelle Verarbeitung. Integralbilder werden aufgebaut, indem für jeden Pixel im Integralbild $I_{\sum}(\mathbf{x})$ mit $\mathbf{x} = (x,y)^T$, alle Pixel im Originalbild die im Rechteck, dass aus Ursprung und Pixel aufgespannt wird, aufsummiert werden $I_{\sum}(\mathbf{x}) = \sum_{i=0}^{i\leq x}\sum_{j=0}^{j\leq y} I(i,j)$.

Mit Hilfe dieser Integralbildern wird die Gauss Ableitung zweiter Ordnung bestimmt, um die Determinante der Hesse Matrix[4] ohne großen Rechenaufwand zu approximieren. Es wird im Anschluss ein Gewichtungsfaktor aus dem Unterschied zwischen der Approximierung zweiter Ordnung und dem tatsächlichen Gaussfilter errechnet. Die Determinante wird für jeden Pixel im Bild auf verschiedenen Skalierungsebenen, wie beim SIFT Verfahren, berechnet, um dann über alle Ebenen Maxima zu suchen. Die auf diese Weise detektierten Merkmale werden durch einen 64-dimensionalen Vektor beschrieben. Dafür wird zunächst eine wiedererkennbare Orientierung aus einer kreisförmigen Umgebungsregion bestimmt. Im Anschluss wird eine quadratische Region anhand der Orientierung ausgewählt und daraus der *SURF Descriptor* berechnet.

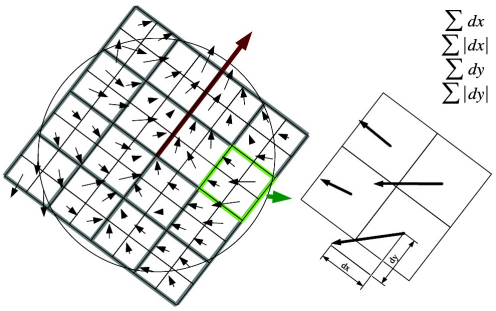


Abbildung 10: *SURF Descriptor* Bestimmung [BETG08]

Für detaillierte Informationen wird hier auf Kapitel 3 und 4 der ursprünglichen Publikation [BETG08] verwiesen.

[4] Die Hesse-Matrix fasst die partiellen zweiten Ableitungen einer mehrdimensionalen Funktion in einer Matrix zusammen. Quelle : `http://de.wikipedia.org/wiki/Hesse-Matrix`

3.3 Vergleich

Nachdem nun die Grundlagen des SIFT und des SURF Verfahrens erläutert wurden, werden hier zwei Implementierungen unter MATLAB verglichen. Hierbei wird die vl_ feat Toolbox[5] für die SIFT Berechnung verwendet, da diese Toolbox schon in der Arbeit von [Wag11] genutzt wurde. Für die Bestimmung der SURF Merkmale wird eine SURFmex[6] genannte Implementierung verwendet, die die OpenCV2.1 SURF Funktion kapselt. Die Bilder entstammen der unter Kapitel 2.1 beschriebenen Kamera.

Die benötigte Rechenzeit und die Anzahl der gefundenen Merkmale werden auf verschiedenen Bildauflösungen, in 10%-Schritten, einhundert Mal berechnet und daraus das arithmetische Mittel kalkuliert. Das Ergebnis ist in den Abbildungen 11 und 12 zu betrachten.

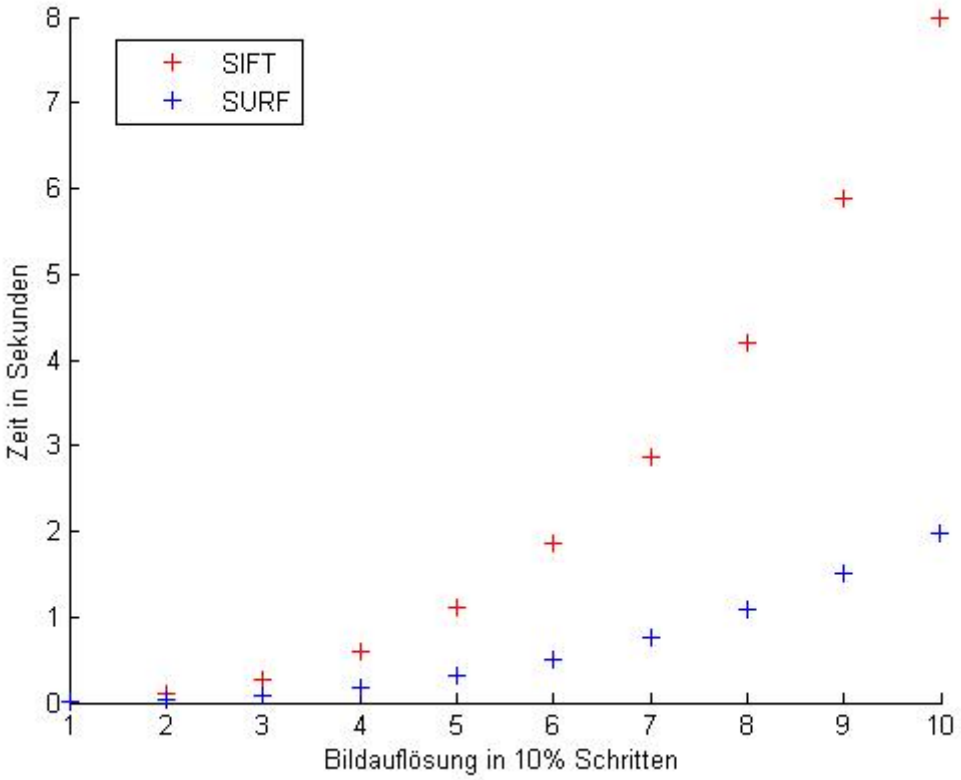


Abbildung 11: SIFT/SURF Rechenzeitbedarf bei verschiedenen Auflösungen

[5] http://www.vlfeat.org/
[6] http://www.maths.lth.se/matematiklth/personal/petter/surfmex.php

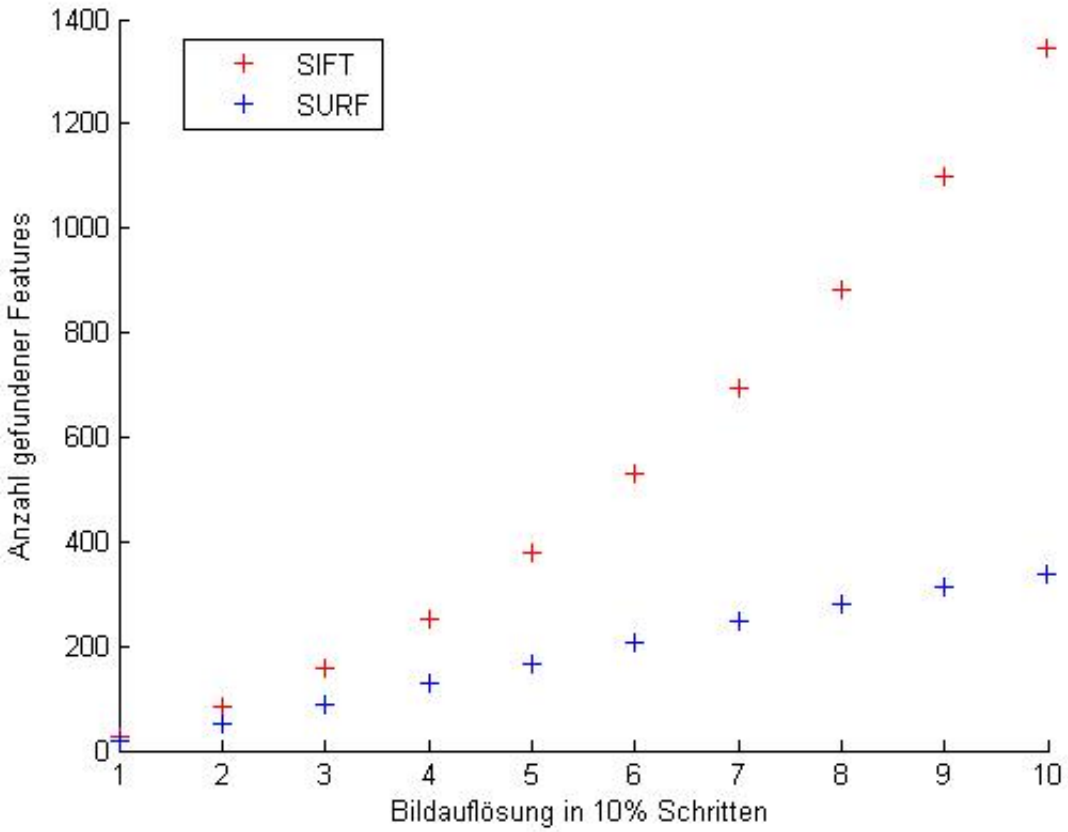


Abbildung 12: Anzahl SIFT/SURF Merkmale bei verschiedenen Auflösungen

Wie man den Grafiken 11 und 12 entnehmen kann, findet das SIFT Verfahren deutlich mehr Feature Punkte, benötigt dementsprechend aber auch deutlich mehr Zeit, um diese zu finden, unabhängig von der Auflösung.

Die Anzahl der Merkmale gibt keine Aussage über die Robustheit des Verfahrens, deshalb wurden zusätzlich die Punktkorrespondenzen betrachtet. Hierfür wurde zunächst ein Bild auf das Objekt reduziert, damit nur Merkmale bestimmt werden die zur Objektoberfläche gehören. Auf der nächsten Seite werden die mit dem SIFT und SURF Algorithmus bestimmten Punkten auf dem Objekt dargestellt.

Abbildung 13: Objekt mit SIFT Merkmalen

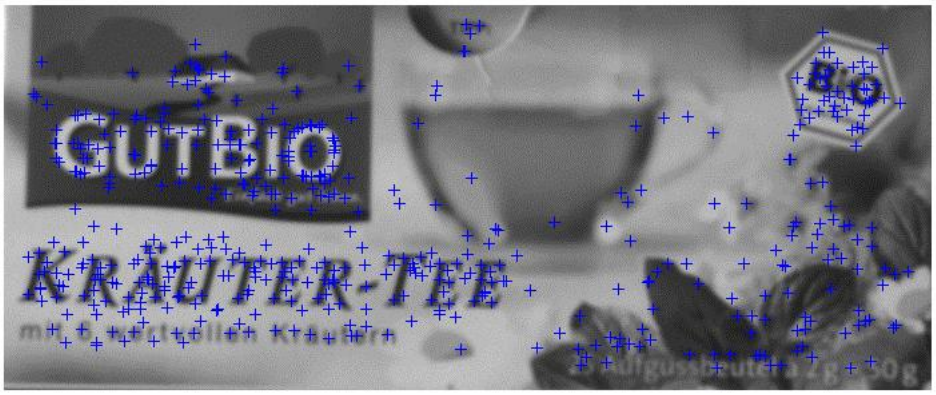


Abbildung 14: Objekt mit SURF Merkmalen

Wie man im Unterschied zwischen Abbildung 13 und 14 erkennen kann, ist wie bereits oben im Test analysiert, die Punktzahl bei SIFT deutlich höher. Diese errechneten Referenzmerkmale wurden nun wieder auf den verschiedenen Bildauflösungsstufen mit den dort erfassten Merkmalen verglichen.

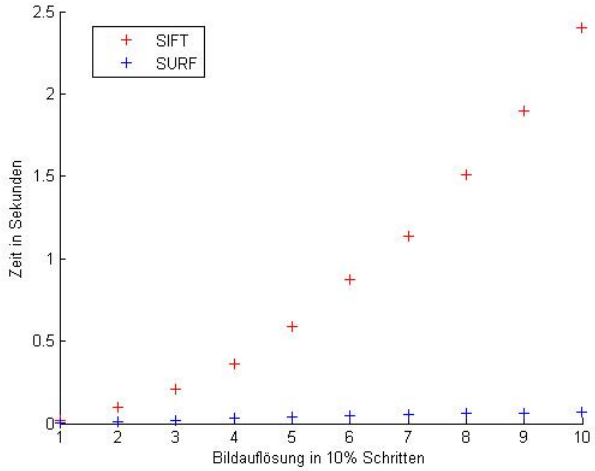


Abbildung 15: Rechenzeitbedarf der Korrespondenzsuche bei verschiedenen Auflösungen

Überraschend ist, dass bei unter 40 prozentiger Auflösung SURF die höhere Anzahl an Treffer liefert, bei durchgehend deutlich niedrigerem Zeitbedarf im Vergleich zu SIFT.

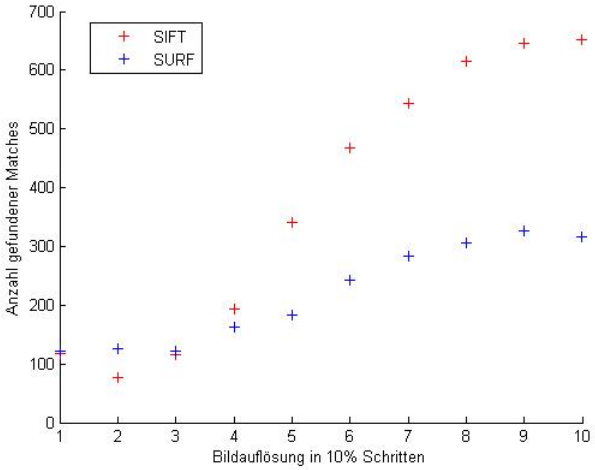


Abbildung 16: Anzahl Matches bei verschiedenen Auflösungen

Visualisiert man nun die SURF und SIFT Merkmale auf voller Auflösung, erkennt man das SIFT aufgrund der vielen Treffer sogar falsche Zuordnungen liefert, etwas das bei SURF seltener passiert. Das SIFT Verfahren produziert dafür die größere Auswahl an Punkten für die Korrespondenzsuche.

Abbildung 17: SIFT Ergebnis der Korrespondenzsuche

Abbildung 18: SURF Ergebnis der Korrespondenzsuche

Folglich liefert das SURF Verfahren die robusteren Punkte und diese auch in deutlich geringerer Zeit, wenn auch dafür deutlich weniger Merkmale. Damit ist das Verfahren bei nicht allzu kleinen Objekten vorzuziehen.

4 Optischer Fluss

4.1 Definition

In diesem Kapitel werden die Grundlagen der beiden bekanntesten Methoden zur Bestimmung des Optischen Flusses dargestellt. Hierfür muss jedoch erstmal der Optische Fluss genauer definiert werden :

Der Optische Fluss wird durch ein Vektorfeld repräsentiert, dass die Bewegungsrichtung und Geschwindigkeit eines jeden Pixels darstellt. Dieses Vektorfeld kann nicht exakt berechnet, sondern nur geschätzt werden. Hierfür gibt es mehrere Verfahren, die alle auf der Auswertung von Helligkeitsinformationen im Bild beruhen. Damit eine akkurate Schätzung möglich ist, betrachten diese Verfahren nicht nur die einzelnen Pixel, sondern ziehen zur Approximierung die Umgebung des Pixels mit ein.

Ein Problem das hierbei auftreten kann, ist das sogenannte Aperturproblem, das daraus resultiert, dass in einem lokalen Bereich nur der Anteil des Optischen Fluss Vektors eindeutig bestimmt werden kann, der parallel zum Helligkeitsgradienten im Ausschnitt verläuft. Somit lässt sich zum Beispiel an einer senkrechten einheitlichen Kante im aktuellen Bildausschnitt nur eine horizontale, jedoch keine vertikale Bewegung feststellen.

Letztendlich gibt es zwei Klassen von Verfahren, die auf dieser Basis arbeiten. Zum Einen die blockweise arbeitenden Verfahren die in der Bildkompression zum Einsatz kommen und die differentiellen Verfahren auf die hier im Weiteren eingegangen wird, da eine Punktverfolgung zu realisieren ist.

Diese differentiellen Verfahren basieren auf dem 1981 von *Berthold K.P. Horn* und *Brian G. Schunk* entwickelten Verfahren [HS80]. Unter der Annahme einer konstanten Helligkeit E in einer Bildersequenz ergibt sich für die Verschiebung eines Punktes (x,y,t) mit der Intensität $E(x,y,t)$ um die Werte δx , δy und δt die Gleichung :

$$E(x,y,t) = E(x+\delta x, y+\delta y, t+\delta t) \tag{4.1}$$

Daraus folgt mit der Approximierung durch die Taylorreihe unter der Annahme, dass die Bewegung gering ist :

$$E(x,y,t) = E(x,y,t) + \frac{\partial E}{\partial x}\delta x + \frac{\partial E}{\partial y}\delta y + \frac{\partial E}{\partial t}\delta t + H.O.T.^{7} \tag{4.2}$$

[7] Higher Order Terms

Nun subtrahiert man $E(x,y,t)$ von beiden Seiten und teilt durch δt. Unter der Voraussetzung, dass die H.O.T. bei $\delta t \to 0$ ebenfalls 0 werden, erhält man:

$$\frac{\partial E}{\partial t} + \frac{\partial E}{\partial x}\frac{\delta x}{\delta t} + \frac{\partial E}{\partial y}\frac{\delta y}{\delta t} = 0 \tag{4.3}$$

Im Folgenden werden nun zwei Verfahren zur Bestimmung des Optischen Fluss näher dargestellt.

4.2 Verfahren und Theorie

Dieses Kapitel beschreibt zunächst die hinter den verschiedenen Verfahren liegenden mathematischen Prinzipien, bevor dann im nächsten Kapitel verschiedene Implementierungen getestet werden. Die behandelten Verfahren sind Horn-Schunk, Lucas-Kanade in zwei Ausführungen und SIFT-Flow.

4.2.1 Horn-Schunck Verfahren

Als Erstes wird die Funktionsweise des 1981 entwickelten Verfahrens, das nach seinen Erschaffern Horn-Schunck Verfahren [HS80] genannt wurde, erläutert werden.

Die Gleichung (4.3) kann man zunächst umformen, indem man $u = \frac{\delta x}{\delta t}$ und $v = \frac{\delta y}{\delta t}$ setzt, sowie $E_x = \frac{\partial E}{\partial x}$, $E_y = \frac{\partial E}{\partial y}$ und $E_t = \frac{\partial E}{\partial t}$

$$E_x u + E_y v + E_t = 0 \tag{4.4}$$

Dies ist eine lineare Gleichung mit zwei Unbekannten und kann nicht ohne weitere Bedingungen gelöst werden. Unter der Annahme, dass die Umgebung eines Pixels einer ähnlichen Verschiebung unterworfen ist, ist es möglich eine Bedingung aufzustellen, die die Geschwindigkeit eines Pixel im Vergleich zu seiner Nachbarschaft beschränkt. Um das zu erreichen minimiert man die Summe der Quadrate der Laplace X- und Y-Komponenten, die wie folgt definiert sind :

$$\begin{aligned} \Delta^2 u &= \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \\ \Delta^2 v &= \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \end{aligned} \tag{4.5}$$

Mit diesen Komponenten lässt sich jetzt das Residuum der Funktion aufstellen, die zur Approximierung des Optischen Flusses dient :

$$f = \int\int [(E_x u + E_y v + E_t)^2 + \alpha^2(\|\Delta x\|^2 \|\Delta y\|^2)]\mathrm{d}x\mathrm{d}y \tag{4.6}$$

Wobei α eine Regelkonstante ist. Desto größer sie ist, desto glatter ist der berechnete Fluss. Minimiert man diese Funktion erhält man :

$$\begin{aligned} E_x(E_x u + E_y v + E_t) - \alpha^2 \Delta u = 0 \\ E_y(E_x u + E_y v + E_t) - \alpha^2 \Delta v = 0 \end{aligned} \tag{4.7}$$

Ersetzt man nun noch Δu durch $\bar{u} - u$, wobei $\bar{u}$ eine gewichtete Mittlung über die Nachbarschaft ist (analog für Δv das Gleiche), erhält man :

$$\begin{aligned} ({E_x}^2 + \alpha^2)u + E_x E_y v &= \alpha^2 \bar{u} - E_x E_t \\ E_x E_y u + ({E_y}^2 + \alpha^2)v &= \alpha^2 \bar{v} - E_y E_t \end{aligned} \tag{4.8}$$

Dies sind lineare Gleichungen für u und v die sich für jeden Pixel berechnen lassen.
Da sich nach einem Durchlauf die Nachbarschaft geändert hat, muss die Berechnung dann wiederholt werden. Hieraus entwickelt sich das iterative Schema :

$$\begin{aligned} u^{i+1} &= \bar{u}^i - \frac{E_x(E_x\bar{u}^i + E_y\bar{v}^i + E_t)}{\alpha^2 + {E_x}^2 + {E_y}^2} \\ v^{i+1} &= \bar{v}^i - \frac{E_y(E_x\bar{u}^i + E_y\bar{v}^i + E_t)}{\alpha^2 + {E_x}^2 + {E_y}^2} \end{aligned} \tag{4.9}$$

In dieser Form lassen sich die Gleichungen lösen und folglich ist eine Implementierung möglich.

4.2.2 Lucas-Kanade Verfahren

Einen anderen Ansatz präsentieren *Bruce D. Lucas* und *Takeo Kanade* in [LK] und [Luc84]. Bei diesem wird davon ausgegangen, dass sich die Umgebung eines Pixels p im fast gleichen Verhältnis wie der Pixel selbst zwischen zwei Frames ändert. Nimmt man nun alle Punkte im Fenster um das Zentrum p und die Gleichung aus (4.3), dann lässt sich ein überstimmtes Gleichungssystem aufstellen, um u und v zu bestimmen.

$$\begin{aligned} E_x(q_1)u + E_y(q_1)v &= -E_t(q_1) \\ E_x(q_2)u + E_y(q_2)v &= -E_t(q_2) \\ &\vdots \\ E_x(q_n)u + E_y(q_n)v &= -E_t(q_n) \end{aligned} \tag{4.10}$$

$q_1, q_2, \ldots, q_n$ sind die Punkte innerhalb des Fensters und $E_x(q_m)$, $E_y(q_m)$ und $E_t(q_m)$ sind die partiellen Ableitungen nach x, y und t im Punkt q_m.
Dies lässt sich nun auch in Matrixform darstellen

$$\mathbf{Ax} = \mathbf{b} \tag{4.11}$$

mit

$$\mathbf{A} = \begin{bmatrix} E_x(q_1) & E_y(q_1) \\ E_x(q_2) & E_y(q_2) \\ \vdots & \vdots \\ E_x(q_n) & E_y(q_n) \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} u \\ v \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} -E_t(q_1) \\ -E_t(q_2) \\ \vdots \\ -E_t(q_n) \end{bmatrix}$$

Dieses System lässt sich nun mit der Methode der kleinsten Quadrate lösen :

$$\mathbf{x} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b} \tag{4.12}$$

Und man erhält ausformuliert :

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} E_x(q_i)^2 & \sum_{i=1}^{n} E_x(q_i)E_y(q_i) \\ \sum_{i=1}^{n} E_x(q_i)E_y(q_i) & \sum_{i=1}^{n} E_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_{i=1}^{n} E_x(q_i)E_t(q_i) \\ -\sum_{i=1}^{n} E_y(q_i)E_t(q_i) \end{bmatrix} \tag{4.13}$$

Dies ist die einfache Lösung, die man implementieren könnte. Oft ist es jedoch sinnvoller eine Gewichtung der Pixel mit einzubeziehen. Indem man die Pixel in der Nähe des Zentrums des Fenster stärker gewichtet als die im Randbereich des Fensters.
Mit Gewichtung erhält man dann das folgende Gleichungssystem :

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} w_i E_x(q_i)^2 & \sum_{i=1}^{n} w_i E_x(q_i)E_y(q_i) \\ \sum_{i=1}^{n} w_i E_x(q_i)E_y(q_i) & \sum_{i=1}^{n} w_i E_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_{i=1}^{n} w_i E_x(q_i)E_t(q_i) \\ -\sum_{i=1}^{n} w_i E_y(q_i)E_t(q_i) \end{bmatrix} \tag{4.14}$$

Eine genauere Beschreibung einer Weiterentwicklung des Lucas-Kanade Verfahren zu einem auf Pyramiden basierten iterativen Merkmalsverfolgungsalgorithmus erfolgt in [Bou00] und wird im Weiteren erläutert. Eine bündige Zusammenfassung ist am Ende des Unterkapitels zu finden.

Es seien zwei Bilder I und J gegeben, sowie ein Punkt **u** aus I dessen korrespondierender Punkt **v** aus J gesucht ist. Dabei sind n_x und n_y die Abmessungen der Bilder. Das Ziel ist die Verschiebung **d** zwischen **u** und **v** zu bestimmen. Der erste Schritt ist die Bildpyramiden I^L und J^L aufzubauen. Hierfür wird $I^0 = I$ gesetzt. Die weiteren Ebenen der Pyramide werden nach der folgenden Vorschrift berechnet :

$$\begin{aligned} &I^L(x,y) = \tfrac{1}{4}I^{L-1}(2x,2y)+ \\ &\tfrac{1}{8}(I^{L-1}(2x-1,2y) + I^{L-1}(2x+1,2y) + I^{L-1}(2x,2y-1) + I^{L-1}(2x,2y+1))+ \\ &\tfrac{1}{16}(I^{L-1}(2x-1,2y-1) + I^{L-1}(2x+1,2y+1) + I^{L-1}(2x-1,2y+1) + I^{L-1}(2x+1,2y-1)) \end{aligned} \tag{4.15}$$

Für J^L wird dies analog ausgeführt. Damit für die Bildkanten keine separate Betrachtung erfolgen muss, wird ein 1-Pixel breiter Rahmen um das Bild definiert, der die gleichen Daten enthält wie die äußersten Pixel im Bild. Daraus resultiert für die Abmessungen der verschiedenen Ebenen, dass diese die größten ganzzahligen Werte annehmen, die die folgenden Bedingungen erfüllen :

$$\begin{aligned} n_x^L &\leq \tfrac{n_x^{L-1}+1}{2} \\ n_y^L &\leq \tfrac{n_y^{L-1}+1}{2} \end{aligned} \tag{4.16}$$

Mit diesen Bedingungen sind I^L und J^L bestimmbar. Nachdem nun die Pyramide auf den Ebenen $L = 0, ..., L_m$ bestimmt ist, muss auch der Punkt **u** auf allen Ebenen bestimmt werden. Die Punktpositionen $\mathbf{u}^L = \begin{bmatrix} {u_x}^L & {u_y}^L \end{bmatrix}^T$ werden berechnet mit :

$$\mathbf{u}^L = \frac{\mathbf{u}}{2^L} \tag{4.17}$$

Das Verfahren läuft nun so ab, dass auf der untersten Pyramidenebene L_m der Optische Fluss bestimmt wird und das Ergebnis dann als initiale Annahme $\mathbf{g}^{L-1} = 2(\mathbf{g}^L + \mathbf{d}^L)$ an die nächst höhere Ebene L_{m-1} weitergegeben wird. Dies wird wiederholt, bis man auf der obersten Ebene $L = 0$ ankommt. Für die tiefste Ebene wird $g^{L_m} = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$ gesetzt.

Beginnend auf der niedrigsten Ebene L_m wird nun für jede Ebene der Optische Fluss und damit die Verschiebung **d** berechnet. Die Bestimmung des Optischen Flusses erfolgt mit der iterativen Lucas-Kanade Methode. Dafür muss als Erstes die nächste Funktion minimiert werden:

$$\varepsilon(\mathbf{d}^L) = \sum_{x=u_x^L-w_x}^{u_x^L+w_x} \sum_{y=u_y^L-w_y}^{u_y^L+w_y} (I^L(x,y) - J^L(x+g_x^L+d_x^L, y+g_y^L+d_y^L))^2 \tag{4.18}$$

Wobei w_x und w_y das Suchfenster festlegen. Nach der Erweiterung durch Ableitung erhält man :

$$\frac{\partial \varepsilon(\mathbf{d}^L)}{\partial \mathbf{d}^L} = -2 \sum_{x=u_x^L-w_x}^{u_x^L+w_x} \sum_{y=u_y^L-w_y}^{u_y^L+w_y} (I^L(x,y) - J^L(x+g_x^L+d_x^L, y+g_y^L+d_y^L)) \begin{bmatrix} \frac{\partial J^L}{\partial x} & \frac{\partial J^L}{\partial y} \end{bmatrix} \tag{4.19}$$

Nun wird $J^L(x+g_x^L+d_x^L, y+g_y^L+d_y^L)$ durch die Taylor-Reihen-Approximierung um den Punkt $\mathbf{d}^L = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$ ersetzt :

$$\frac{\partial \varepsilon(\mathbf{d}^L)}{\partial \mathbf{d}^L} \approx -2 \sum_{x=u_x^L-w_x}^{u_x^L+w_x} \sum_{y=u_y^L-w_y}^{u_y^L+w_y} (I^L(x,y) - J^L(x,y) - \begin{bmatrix} \frac{\partial J^L}{\partial x} & \frac{\partial J^L}{\partial y} \end{bmatrix} \mathbf{d}^L) \begin{bmatrix} \frac{\partial J^L}{\partial x} & \frac{\partial J^L}{\partial y} \end{bmatrix} \tag{4.20}$$

Dabei gilt :

$$\frac{\partial J^L}{\partial x} = I_x(x,y) = \frac{I^L(x+1,y) - I^L(x-1,y)}{2} \tag{4.21}$$

$$\frac{\partial J^L}{\partial y} = I_y(x,y) = \frac{I^L(x,y+1) - I^L(x,y-1)}{2} \tag{4.22}$$

Um die Ableitungen zu bestimmen wird der Sharr-Operator verwendet, wodurch die Gleichung (4.20) in die nächste Form übergeht :

$$\frac{1}{2}\left[\frac{\partial\varepsilon(\mathbf{d}^L)}{\partial\mathbf{d}^L}\right]^T \approx \sum_{x=u_x^L-w_x}^{u_x^L+w_x}\sum_{y=u_y^L-w_y}^{u_y^L+w_y}\left(\begin{bmatrix} {I_x}^2 & I_xI_y \\ I_xI_y & {I_y}^2 \end{bmatrix}\mathbf{d}^L - \begin{bmatrix}\delta II_x \\ \delta II_y\end{bmatrix}\right) \tag{4.23}$$

Setzt man nun

$$G = \sum_{x=u_x^L-w_x}^{u_x^L+w_x}\sum_{y=u_y^L-w_y}^{u_y^L+w_y}\begin{bmatrix} {I_x}^2 & I_xI_y \\ I_xI_y & {I_y}^2 \end{bmatrix} \quad \text{und} \quad \overline{b} = \sum_{x=u_x^L-w_x}^{u_x^L+w_x}\sum_{y=u_y^L-w_y}^{u_y^L+w_y}\begin{bmatrix}\delta II_x \\ \delta II_y\end{bmatrix} \tag{4.24}$$

kann man die Gleichung (4.23) in der Form

$$\frac{1}{2}\left[\partial\varepsilon(\mathbf{d}^L)\partial\mathbf{d}^L\right]^T \approx G\mathbf{d}^L - \overline{b} \tag{4.25}$$

notieren und umgeformt ergibt sich somit für den Optischen Fluss :

$$\mathbf{d}_{opt}^L = G^{-1}\overline{b} \tag{4.26}$$

Damit dies nun iterativ berechnet werden kann, muss diese Gleichung noch modifiziert werden. Sei $\overline{\eta}^k$ der Optische Fluss $\mathbf{d}^L$ bei Iteration k, dann gilt :

$$\overline{\eta}^k = G^{-1}\overline{b}_k \tag{4.27}$$

mit

$$\overline{b}_k = \sum_{x=u_x^L-w_x}^{u_x^L+w_x}\sum_{y=u_y^L-w_y}^{u_y^L+w_y}\begin{bmatrix}\delta I_kI_x \\ \delta I_kI_y\end{bmatrix} \tag{4.28}$$

und

$$\delta I_k = I^L(x,y) - J^L(x+g_x^L+(\mathbf{d}^L)_x^{k-1}, y+g_y^L+(\mathbf{d}^L)_y^{k-1}) \tag{4.29}$$

Somit ist $\mathbf{d}^L = \sum_{k=1}^K \overline{\eta}^k$.
Für detailliertere Informationen wird auf die entsprechende Literatur [Bou00] verwiesen.

Ist $\mathbf{d}^0$ bestimmt, lässt sich der gesuchte Punkt $\mathbf{v}$ mit $\mathbf{v} = \mathbf{u} + \mathbf{d} = \mathbf{u} + \mathbf{g}^0 + \mathbf{d}^0$ berechnen.

Das gesamte Verfahren ist hier noch einmal als Pseudo-Code aufgeführt.

Gegeben : Zwei Bilder I und J, sowie ein Punkt **u** aus I.
Gesucht : Zu **u** korrespondierender Punkt **v** aus J

Pyramidenaufbau : $\{I^L\}_{L=0,...,L_m}$ und $\{J^L\}_{L=0,...,L_m}$
Initiale Pyramiden Annahme : $\mathbf{g}^{L_m} = \begin{bmatrix} {g_x}^{L_m} & {g_x}^{L_m} \end{bmatrix}^T = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$
for $L = L_M \to 0$ **do**
  Position von u in I^L : $\mathbf{u}^L = \begin{bmatrix} p_x & p_y \end{bmatrix}^T = \frac{\mathbf{u}}{2^L}$
  Ableitung von I^L nach x : $I_x(x,y) = \frac{I^L(x+1,y) - I^L(x-1,y)}{2}$
  Ableitung von I^L nach y : $I_y(x,y) = \frac{I^L(x,y+1) - I^L(x,y-1)}{2}$
  Räumliche Gradienten Matrix : $G = \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} {I_x}^2(x,y) & I_x(x,y)I_y(x,y) \\ I_x(x,y)I_y(x,y) & {I_y}^2(x,y) \end{bmatrix}$
  Initiale Annahme für L-K : $(\mathbf{d}^L)^0 = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$
  for $k = 1 \to K$ or $\|\overline{\eta}^k\| < threshold$ **do**
    Bildunterschied : $\delta I_k(x,y) = I^L(x,y) - J^L(x + g_x^L + (\mathbf{d}^L)_x^{k-1}, y + g_y^L + (\mathbf{d}^L)_y^{k-1})$
    Bildfehler Vektor : $\overline{b}_k = \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} \delta I_k(x,y) I_x(x,y) \\ \delta I_k(x,y) I_y(x,y) \end{bmatrix}$
    Optischer Fluss : $\overline{\eta}^k = G^{-1}\overline{b}_k$
    Annahme für nächste Iteration : $(\mathbf{d}^L)^k = (\mathbf{d}^L)^{k-1} + \overline{\eta}^k$
  end for
  Annahme für nächste Iteration L-1 : $\mathbf{g}^{L-1} = \begin{bmatrix} {g_x}^{L-1} & {g_y}^{L-1} \end{bmatrix}^T = 2(g^L + \mathbf{d}^L)$
end for
Endgültiger Optischer Fluss Vektor : $\mathbf{d} = g^0 + \mathbf{d}^0$
Punktkoordinaten in J : $\mathbf{v} = \mathbf{u} + \mathbf{d}$

4.2.3 SIFT-Flow Verfahren

Eine modifizierte Version des Optischen Flusses wird in einem als SIFTFlow[8] bezeichneten Verfahren verwendet.
Dieses in [LYT08] vorgestellte Verfahren arbeitet statt auf den RGB Werten der Bilder auf den SIFT Daten. Das heißt es wird zu Beginn des Verfahrens für beide Bilder für jeden Pixel der SIFT Beschreibungsvektor berechnet und als so genannte SIFT Bilder abgespeichert.

Die Zielfunktion dieser Implementierung lautet :

$$\begin{aligned} &E(w) = \textstyle\sum_q \min(\|s_1(q) - s_2(q + x(q))\|_1, t) + \\ &\textstyle\sum_q \eta(|u(q)| + |v(q)|) + \\ &\textstyle\sum_{q,p\in\varepsilon} \min(\alpha|u(q) - u(p)|, d) + \min(\alpha|v(q) - v(p)|, d) \end{aligned} \tag{4.30}$$

mit s_1,s_2 als SIFT Bilder, $p = (x,y)$, $w(p) = (u(p), v(p))$ und ε gleich der Menge der Punkte der direkten Nachbarschaft, sowie t und d als Grenzwerte. Die Komplexität der Methode liegt bei $\mathrm{O}(\mathrm{L}^2)$. Auf eine ausführliche Beschreibung der Korrespondenzsuche wird an dieser Stelle verzichtet. Interessierte können dies im oben genannten Artikel nachlesen.

[8] http://people.csail.mit.edu/celiu/SIFTflow/

4.3 Test verschiedener Implementierungen

Der Abschnitt dieser Arbeit stellt als Erstes die Testgrundlagen dar. Danach folgen die Tests verschiedener Implementierungen und als Abschluss folgt ein kurzes Fazit.

4.3.1 Testgrundlagen

Das Ziel dieser Arbeit ist eine echtzeitfähige Verfolgung von Objekten, um eine Geschwindigkeitsoptimierung zu erhalten.

Aufgrund dieser Zielsetzung prüfen die ersten Testreihen zu jeder der folgenden Implementierungen den Zeitbedarf. Dies geschieht abhängig von der Bildauflösung und den funktionsspezifischen Parametern.

Scheinen diese Ergebnisse annähernd verwendbar, wird versucht die optimalen Parameter zu ermitteln. Optimale Parameter bedeutet, dass möglichst wenige Punkte das Objekt verlassen, die Distanz zwischen berechneter und tatsächlicher Position möglichst klein ist und dabei die Rechenzeit noch möglichst gering gehalten wird.

Möglichst wenige Punkte außerhalb des Objektes und eine möglichst kleine Distanz ist notwendig, da eine punktgenaue Regelung optimiert werden soll. Dabei sollte die Zeit natürlich geringer, als die Bisherige sein.

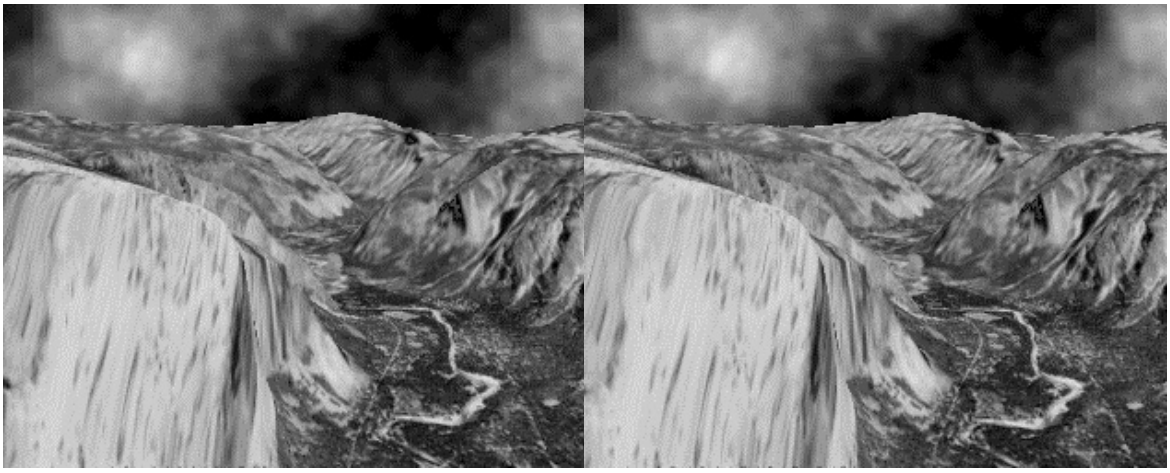
Ergibt ein Test in Bezug auf diese drei Kerngedanken positive Ergebnisse, dann muss noch die Robustheit des Verfahrens bei translatorischen, rotatorischen und skalierungstechnischen Bildveränderungen überprüft werden. Außerdem muss die Verfolgung in stark strukturierter Umgebung getestet werden.

4.3.2 Horn-Schunck

Als erster Versuch zur Objektverfolgung dient eine Implementierung des Horn-Schunck Verfahrens aus 4.2.1. Diese Implementierung in MATLAB[9] wurde von *Mohd Kharbat* bei *Cranfield Defence and Security* im Oktober 2008 realisiert und im Januar 2009 noch einmal überarbeitet.

Die Methode braucht als Eingabeparameter, wie in Kapitel 4.2.1 entwickelt, die beiden Eingangsbilder, einen Glättungsfaktor α und die Anzahl der durchzuführenden Iterationen. Außerdem kann optional eine Annahme für die Komponenten *u* und *v* des Optischen Flusses übergeben werden.

Ein erster Test mit den mitgelieferten Demo-Grauwert-Bildern (Auflösung 316x252) erzeugt annehmbare Ausgaben.



(a) Erstes Bild (b) Zweites Bild

Abbildung 19: Mitgelieferte Beispiel Bilder (Verschiebung nach links vorne von Bild 1 auf 2)

Die mitgelieferte Visualisierung des Optischen Flusses zeigt eindeutig die Verschiebung zwischen den beiden Beispielbildern. Die Pfeile des Optischen Flusses verdecken allerdings bei voller Auflösung die Grafik, weswegen in der folgenden Abbildung nur ein Ausschnitt, der vorderen Gebirgskette, gezeigt wird.

[9] `http://www.mathworks.com/matlabcentral/fileexchange/22756-horn-schunck-optical-flow-method`

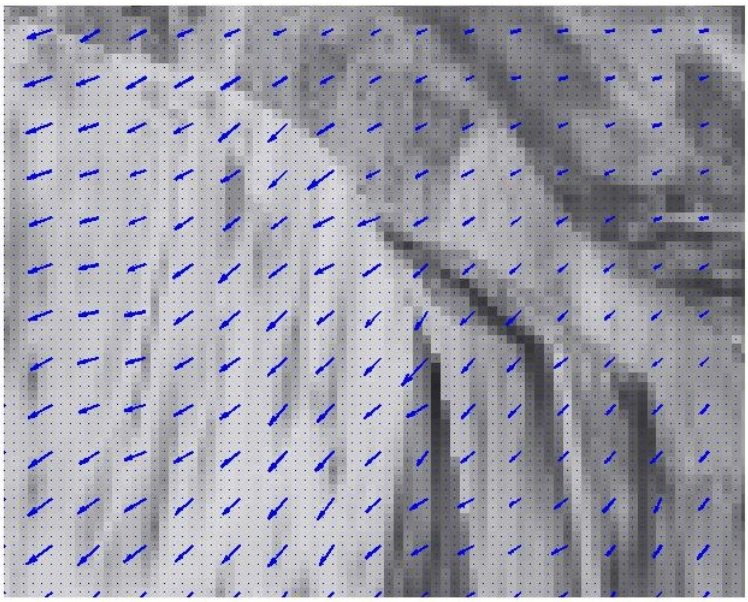


Abbildung 20: Ausschnitt des Vektorfeldes des Optischen Flusses

Die benötigte Rechenzeit beträgt 1,58 Sekunden inklusive dem Anzeigen der Visualisierung. Aufgrund der positiven Ergebnisse mit den Beispielbildern folgen weitere Tests mit Bildern, die mit der zur Verfügung stehenden Hardware aufgenommen wurden.

Das Ziel dieser Arbeit ist eine nahezu echtzeitfähige Verfolgung von Objekten, daher wird zunächst der Einfluss der verschiedenen Parameter auf die benötigte Rechenzeit betrachtet. Um alle Parametermöglichkeiten zu analysieren werden vier Testreihen durchgeführt. In der Ersten wird die Bildauflösung von 10 bis 100 Prozent in Zehner Schritten abgeändert und jeweils der Optische Fluss mit Hilfe des Horn-Schunk Verfahrens bestimmt und die Zeit gemessen. Die zweite und dritte Testreihe modifizieren beide den Glättungsfaktor α. Dies ist erforderlich, da der default Wert bei 1 liegt und somit nicht eindeutig ist, ob eine Modifikation nach oben oder unten nötig ist. Die zweite Reihe deckt den unteren Bereich ab, indem α im Bereich von 0.1 bis 1 variiert wird, in 0.1 Schritten, während der dritte Test den oberen Bereich von 1 bis 10 testet. Die letzte Versuchsreihe modifiziert die Anzahl von Iterationen im Bereich von 10 bis 100 in Zehner Schritten. Die gemessen Zeiten werden in der nächsten Abbildung dargestellt.

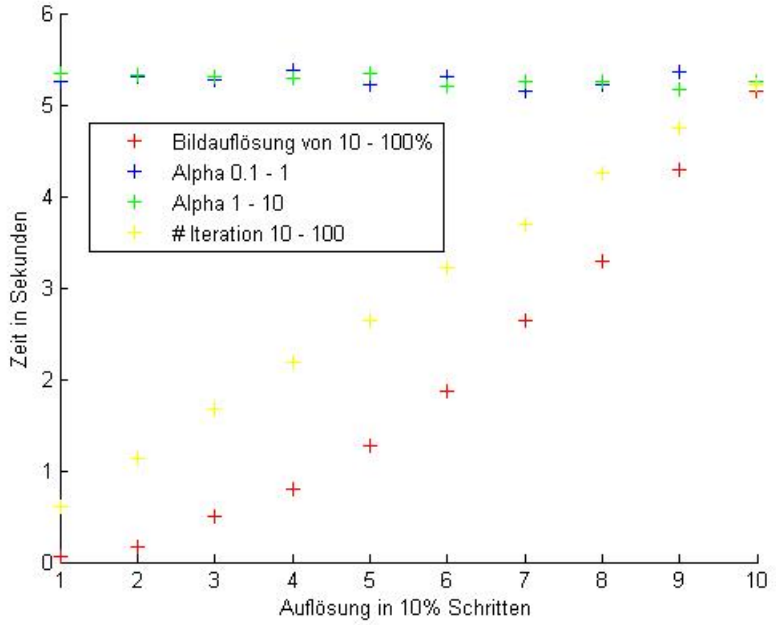


Abbildung 21: Gemessene Zeiten bei unterschiedlicher Parameterwahl

Wie die Grafik 21 verdeutlicht, bedeuten Änderungen am Glättungsfaktor keine Verringerung oder Erhöhung der Rechenzeit. Reduziert man jedoch die Anzahl der Iterationen oder die Bildauflösung verringert sich gleichzeitig die Rechenzeit.

Als weitere Ergebnisse der Testreihe wurden die soweit besten Parameter ermittelt :

- Bildauflösung 100% (trotz höherer Rechenzeit bessere Punktverteilung)
- Glättungsfaktor $\alpha = 10$
- Iterationen = 100

Mit diesen Einstellungen erhält man diese Ergebnisse :

- Punkte Wiedererkennung : 403 von 669, somit ein Punktverlust von 39,8% in einem Durchgang
- Punkte außerhalb des Objektes : 4 (entspricht ca. 1%).
- Durchschnittliche Punktverschiebung 250,6 Pixel
- Benötigte Zeit 5,3 Sekunden

Die durchschnittliche Verschiebung wird bestimmt, indem die korrekten Positionen mit SIFT errechnet werden und dann mit dem Satz des Pythogaros $c = \sqrt{a^2 + b^2}$ die Distanz ermittelt wird. Dieser Abstand wird für jeden Punkt kalkuliert und aufsummiert und dann durch die Anzahl Punkte geteilt, womit man die durchschnittliche Verschiebung erhält. Dieser Wert ist von Interesse, da für die punktgenaue Verfolgung, ein möglichst geringer Unterschied zwischen ermittelter und tatsächlicher Position erwartet wird. Das Ergebnis wird im nächsten Bild 22 präsentiert, dabei sind die SIFT Punkte rot, während die mit Horn-Schunk bestimmten Positionen grün eingezeichnet sind. Die korrespondierenden Punkte sind mit Linien verbunden.

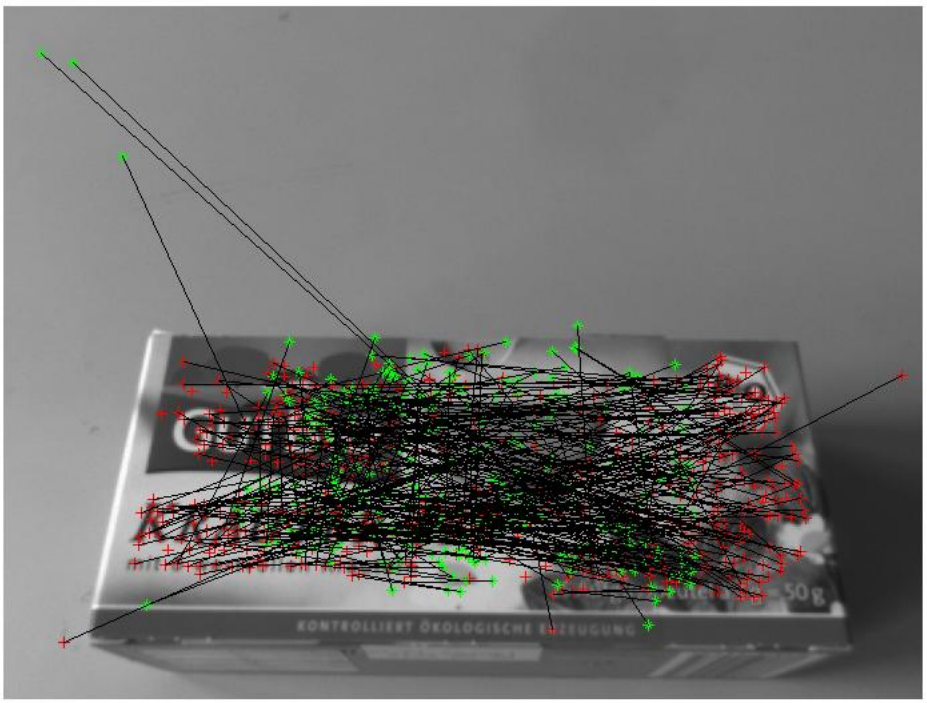


Abbildung 22: Ergebnis bei ermittelten optimalen Parametern

Die Visualisierung zeigt, dass keine eindeutige Orientierung der Verschiebung erkannt wurde. Zudem zeigen die langen schwarzen Verbindungslinien, dass die erkannten Korrespondenzen weit auseinander liegen. Damit erscheint diese Implementierung nicht für das Ziel der Arbeit tauglich, insbesondere da die benötigte Rechenzeit von 5 Sekunden nicht der beabsichtigten Echtzeit entspricht.

4.3.3 Hierarchisches Lucas-Kanade

Als zweites wird eine Implementierung[10] des Lucas-Kanade Verfahrens getestet, auf Basis der Ergebnisse des durchgeführten Vergleiches von Optischen Fluss Verfahren in [BFB94].

Diese Realisierung in MATLAB baut auf 4.2.2 und [LK] auf. Hierbei wird ein hierarchischer Ansatz verfolgt, der das Bild in verschiedenen Auflösungen betrachtet.
Als Input dienen zwei aufeinander folgende Bilder der Sequenz, die Fenstergröße, die Anzahl der Durchläufe und Pyramidenebenen.

Der erste Testschwerpunkt dieser Implementierung ist wiederum der Zeitfaktor. Folglich ermitteln die ersten Versuchsreihen den Einfluss der einzelnen Parameter auf den Rechenbedarf.

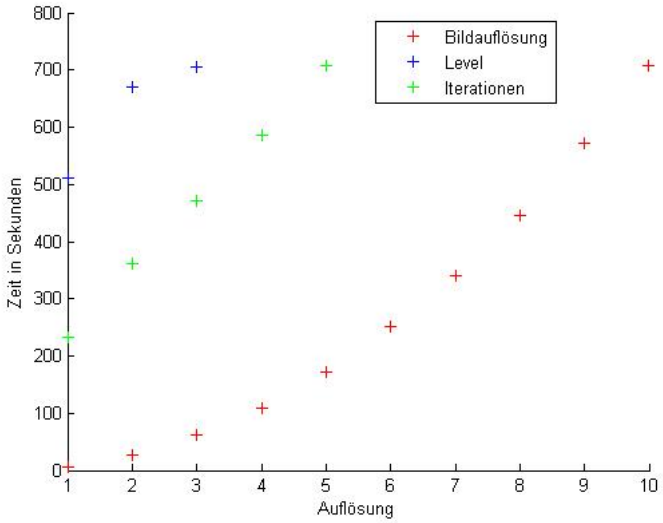


Abbildung 23: Zeitbedarf bei Parametern Variierung

Aus den Graphen in 24 geht hervor, dass alle Parameter Einfluss auf den Zeitbedarf haben. Insbesondere die Bildauflösung beeinflusst diesen, die Iterationsanzahl ebenfalls. Die wichtigste Erkenntnis der Testreihe ist jedoch, dass selbst bei geringsten Einstellungen, die gemessene Rechenzeit weit über den Ergebnissen von 4.3.2 liegt. Sie beträgt sogar mehr als bei der Lokalisierung durch SIFT.

[10] `http://www.cs.ucf.edu/~vision/Code/Optical_Flow/Lucas%20Kanade.zip`

Es wird trotzdem versucht, einen verwendbaren Datensatz als Ausgabe zu erhalten. Hierfür wird die Bildauflösung, die Fenstergröße und die Iterationszahl auf Maximum gesetzt, da bei den vorherigen Tests auf Zeit zuviele Punkte verloren gingen. Nur die Level Anzahl wird auf 2 gesetzt, da diese gleichwertige Punktemengen erzeugt.
Parameterwahl :

- Bildauflösung : 100%
- Level : 2
- Iterationen : 5
- Fenstergröße : 4

Letztendlich ist das mit den “besten” Parametern erzeugte Ergebnis nicht brauchbar.

Abbildung 24: Ergebnis mit SIFT(rot) und Lucas Kanade (grün) Punkten

Die erzeugte Verteilung der Punkte lässt keinerlei Rückschlüsse auf eine Bewegung zu und ist damit vollkommen ungeeignet. Hinzu kommt der hohe Zeitbedarf, der dieses Verfahren ebenfalls als unbrauchbar für das gesetzte Ziel einordnet.

4.3.4 SIFT-Flow

Die benötigte Zeit wird bei dieser Implementierung nur im Vergleich zur Bildauflösung betrachtet. Das Ergebnis zeigt die Abbildung 25.

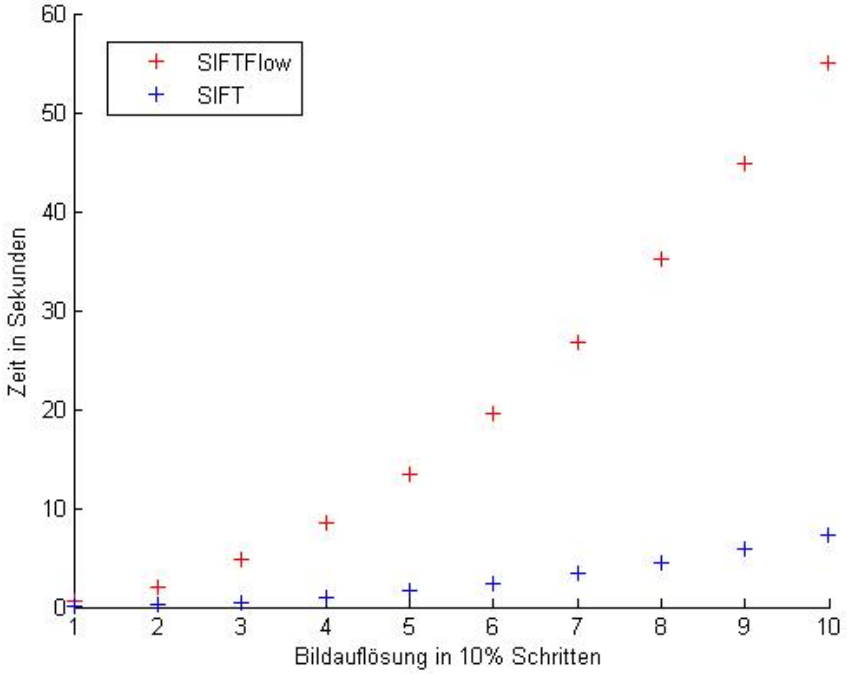


Abbildung 25: Rechenzeit bei verschiedenen Auflösungen

Die gemessenen Zeiten sind höher als beim Test des Horn-Schunk Verfahrens, was darauf zurückzuführen ist, dass neben der Flussberechnung auch eine SIFT Berechnung nötig ist. Die Messung bei voller Auflösung ergibt die folgenden Daten :

- Punkte Wiedererkennung : 403 von 669, somit ein Punktverlust von 39,8% in einem Durchgang (wie bei 4.3.2)
- Punkte außerhalb des Objektes : 3 (weniger als 1%).
- Durchschnittliche Punktverschiebung 244,9 Pixel
- Benötigte Zeit 62,5 Sekunden

Damit ähneln die ermittelten Daten, genauso wie die erzeugte Ausgabe, denen des Horn-Schunk Algorithmus.

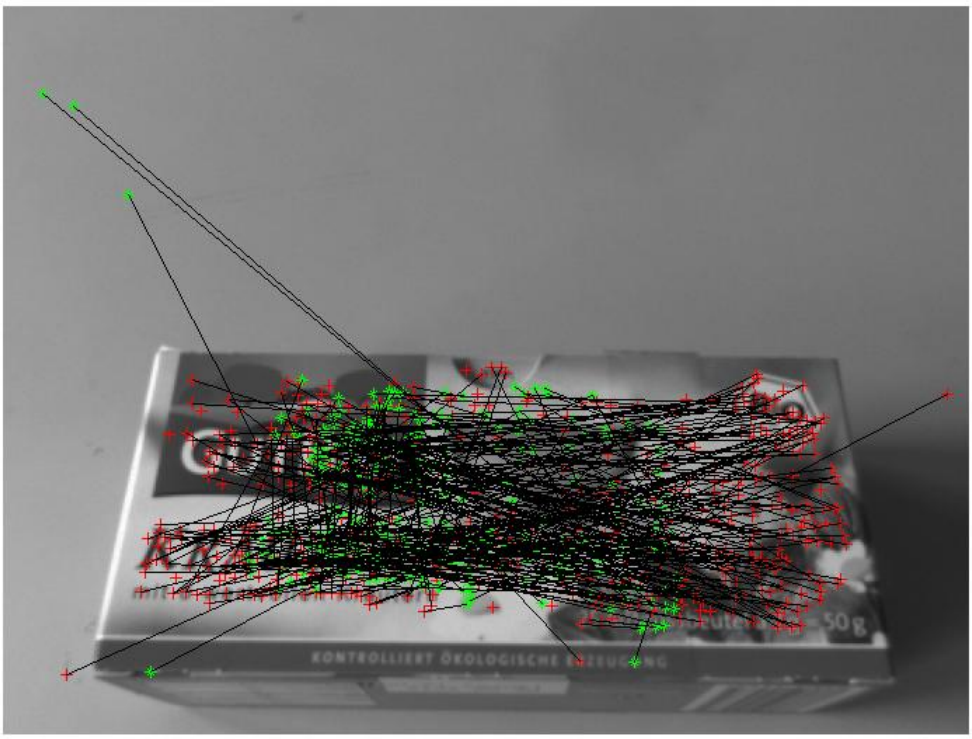


Abbildung 26: SIFTFlow Ergebnis - SIFT(rot) - SIFTFLOW (grün)

Durch ein größeres Suchfenster und der Suche auf mehreren Pyramidenebenen, ergibt sich eine ansatzweise brauchbare Wiedererkennung des Objektes. Die benötige Zeit ist geringer als das Lucas-Kanade Verfahren, jedoch ist es nicht schneller als die Horn-Schunk Implementierung.

4.3.5 Iteratives Lucas-Kanade

Nachdem alle bisherigen Testreihen keine Echtzeitfähigkeit (Optischer Fluss Berechnung in deutlich unter 1 Sekunde) nachweisen, wird als Viertes ein Ansatz in C verfolgt. Hierbei wird das Lucas-Kanade Verfahren mit Pyramiden aus der OpenCV Bibliothek[11] verwendet.

Im Handbuch der Bibliothek [Int01] erfolgt eine kurze Beschreibung der Funktionsparameter, die im Folgenden wiedergegeben wird. Neben den beiden Bildern der Sequenz wird der Funktion eine Liste von Punkten übergeben, die verfolgt werden sollen, sowie die Suchfenstergröße, die Anzahl der zu berechnenden Pyramidenebenen, die Anzahl der Feature Punkte und zwei optionale Buffer, die ansonsten zur Laufzeit allokiert werden. Desweiteren besteht die Möglichkeit noch weitere Kriterien und Flags zu setzen, zum Beispiel wenn schon Teile vorberechnet sind. Dies wird aber nicht in dieser Arbeit verwendet. Als Rückgabe erhält man die neuen Punktkoordinaten, sowie die Fehlerdistanz und eine Liste, die je nachdem, ob der Punkt wiedergefunden wurde oder nicht, Nullen und Einsen enthält.

Diese Umsetzung erlaubt eine Punktverfolgung von unter einer drittel Sekunde Dauer und wird deswegen in eine MEX-Datei integriert, die dann in MATLAB verwendet werden kann. Damit die OpenCV Implementierung mit korrekter Funktionsweise in MATLAB aufgerufen werden kann, muss aber in MATLAB eine Konvertierung der Bilddaten erfolgen, da die Bildrepräsentation zwischen MATLAB und OpenCV divergiert. MATLAB speichert Bilder in der Form $height \times width \times colorchannels$, während OpenCV dies in der Form $(height \times colorchannels) \times width$ erledigt. Dies lässt sich jedoch mit einem einfachen Permutierungs-Befehl in MATLAB lösen.

Bevor der eigentliche Test erfolgt, wird hier im Weiteren noch kurz auf den Aufbau der MEX-Datei eingegangen. Wie schon in 2.2.1 beschrieben wird ein Zwei-Dateien-Ansatz gewählt, wie in [SE08] vorgeschlagen, um eine übersichtliche Codegestaltung zu erreichen.

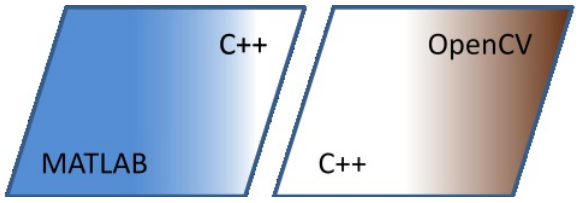


Abbildung 27: Zweigeteilter Aufbau : Links Mex-Datei; Rechts OpenCV Interface [SE08]

[11] http://opencv.jp/opencv-2.1_org/cpp/motion_analysis_and_object_tracking.html

Die Hauptdatei (B.1) dient somit als eine Art Interface und ist ein Wrapper für die eigentliche Programmroutine. In ihr werden die Daten, die aus MATLAB übergeben werden, ausgelesen und in nutzbare Datenstrukturen für C verwandelt. Diese Zeiger werden an die Funktion in der zweiten Datei übergeben, die dann die eigentliche Berechnung ausführt. Zuvor werden ebenfalls die Zeiger auf die Ausgabevariablen der Mex-Datei erzeugt und der Funktion mitgegeben.

In der zweiten Datei (B.2) werden dann zunächst die erhaltenen Daten in OpenCV Datenformate verwandelt. Dazu werden aus den Höhen, Breiten und den Bilddaten zwei sogenannte *IplImages* erzeugt, die von der Optischer-Fluss-Methode der OpenCV Bibliothek verwendet werden können. Außerdem werden aus den Merkmalsdaten eine Liste von *CvPoint2D32f* generiert. Mit diesen Daten kann nun die Funktion aufgerufen werden. Im Anschluss müssen die neuen Punkte vom *CvPoint2D32f* Format zurück in *double* Werte konvertiert werden, die dann wiederum dem Ausgabezeiger zugewiesen werden, womit die Daten in MATLAB verwendbar sind.

Nachdem der Aufbau der MEX-Datei erläutert ist, wird hier ebenfalls in einer Testreihe der Einfluss der Bildauflösung auf die Rechenzeit betrachtet.

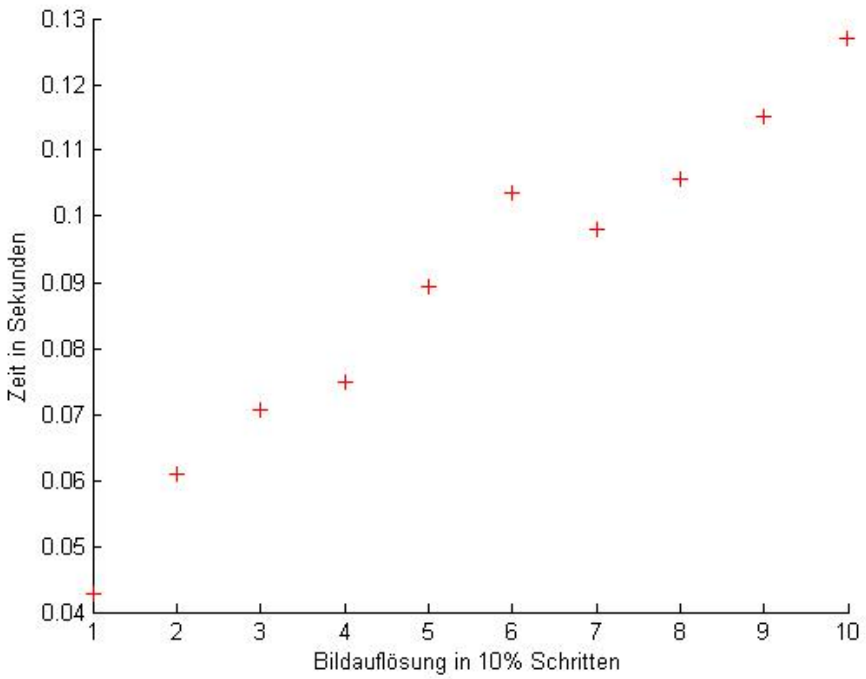


Abbildung 28: Rechenzeit bei verschiedenen Auflösungen

Die Grafik 28 zeigt einen linearen Anstieg der Zeit. Der Ausschlag bei 60% resultiert aus der CPU Zuteilung, da die Rechenzeit im Millisekundenbereich liegt und der Ausschlag sich bei jeder Berechnung ändert. Da die benötigte Zeit sehr gering ist, wird im Weiteren mit voller Auflösung gearbeitet. Eine Verfolgung über eine Bildsequenz von zehn Bildern ergab die unten gelisteten Ergebnisse. Die dargestellten Abbildungen repräsentieren Ausschnitte nach verschiedenen Iterationsschritten. Hierfür werden im ersten Bild Merkmale auf dem Objekt in einem festen Muster ausgewählt, da SIFT und SURF teilweise bei der initialen Bestimmung Outlier, das sind Punkte die nicht mehr auf dem Objekt liegen, produzieren.

Abbildung 29: Erstes Bild der Sequenz mit Startmuster

Die 11×11 ausgewählten Merkmale sollen dann durch eine Bildabfolge von zehn Grafiken (Gesamte Sequenz im Anhang C.1) verfolgt und die benötige Zeit gemessen werden, sowie die Anzahl an Outlier erfasst werden. Die Bilder werden zur Laufzeit geladen, da diese auch in der späteren Implementierung von der Kamera eingelesen werden.

(a) Ergebnis nach 1. Iteration

(b) Ergebnis nach 5. Iteration

Abbildung 30: OpenCV Lucas-Kanade : Verfolgungssequenzausschnitte

Abbildung 31: Ergebnis nach 10. Iteration : Zeitaufwand 0.65s und Outlier 2,5%

Im Vergleich zu den vorherigen getesteten Verfahren, gibt es bei der OpenCV Umsetzung so gut wie keine Punkte die das Objekt verlassen. Auch wenn sich das Muster nach und nach auflöst, was daran liegt, dass nicht alle Punkte gleich robust sind, da es keine mit SIFT, SURF oder ähnlichen Verfahren bestimmten Merkmale sind, so wird doch immerhin die *Region of Interest* und damit das Objekt ausreichend genau bestimmt. Die Berechnungszeiten sind selbst bei 10 Bildern noch weit unter einer Sekunde und damit ist das Verfahren ausreichend echtzeitfähig.

4.3.6 Fazit

Wie die in den letzten Unterkapiteln durchgeführten Testreihen zeigen, erkennen das Horn-Schunk und das SIFT-Flow Verfahren die Objekte wieder, wenn auch ohne lokale Korrespondenzerkennungen und bei einem zu hohem Rechenbedarf. Diese Anforderungen löst die Implementierung aus der OpenCV Bibliothek deutlich besser und viel schneller, während die andere Lucas-Kanade Implementierung nicht eine brauchbare Korrespondenz zustande bekommt. Da das Ziel dieser Arbeit eine Geschwindigkeitsoptimierung ist, fiel die Entscheidung zu Gunsten der OpenCV Umsetzung aus. Diese Punktverfolgung wird mit einer Merkmalsdetektion verbunden, um dann in die bestehende Anwendung integriert und getestet zu werden.

5 Kombinierte Verfolgung mit Erkennung

Anhand der aus der Analyse der Testergebnisse der vorherigen Kapitel hervorgegangenen Daten wird in diesem Kapitel nun eine kombinierte Punktverfolgung in MATLAB realisiert.

Zur Merkmalsextraktion kann das SURF Verfahren angewendet werden, da dies die besseren Ergebnisse in Kapitel 3.3 liefert. Aber es besteht dennoch die Möglichkeit auf SIFT umzustellen, da dies eine höhere Punktanzahl produziert und häufiger verwendet wird. Damit ist eine Integration in verschiedene Anwendungen möglich.

Für die Verfolgung der Merkmale wird auf die unter Kapitel 4.3.5 getestete OpenCV Implementierung zurückgegriffen. Der Test ergibt, dass über eine Sequenz von 10 Bildern eine Verfolgung relativ verlustfrei möglich ist. Danach sollte aber eine Neubestimmung der Merkmalspositionen ausgeführt werden. Deshalb muss diese Implementierung wissen in welcher Iteration sie sich befindet, damit entschieden wird, ob die Verfolgung durchgeführt wird oder ob eine Relokalisierung mit SURF oder SIFT nötig ist. Dies wird über ein Struct realisiert, das die Funktion als Input erhält und modifiziert wieder ausgibt. Neben der Iterationsnummer werden auch andere Daten benötigt, sodass das Struct diesen Aufbau hat :

- ITERATION : Iterationsanzahl
- IMG : Referenz Bild
- OFIMG : Bild des letzten Durchganges, benötigt für Optischen Fluss
- FEATURES : Merkmale
- DESC : Merkmalsvektoren
- RDESC : Merkmalsvektoren der Referenzpunkte
- MATCHES : Korrespondenzzuordnungen zwischen vorherigen und neuen Punkten

Der Funktionsaufruf lautet damit :

```
[struct] = combined_tracking(mode,img,struct);
```

Die Variable *mode* muss entweder mit 'SIFT' oder 'SURF' belegt sein. Zur korrekten Initialisierung der Methode muss im Struct entweder das Referenzbild oder eine Feature und Deskriptor Liste vorhanden sein. Alle im Struct enthaltenen Daten sind frei verwendbar, jedoch sollte das Struct selbst, nach der Initialisierung, nicht mehr modifiziert werden um einen korrekten Ablauf zu gewährleisten.

5.1 Test

Nachdem die Funktion komplettiert ist, wird ein abschließender Test der entwickelten Funktion, unabhängig von der umgebenden Funktionalität, in Bezug auf den Zeitbedarf durchgeführt. Dies dient zur Bereitstellung von Referenzdaten zum Vergleich mit der in den Kontext anderer Anwendungen integrierten Funktion.

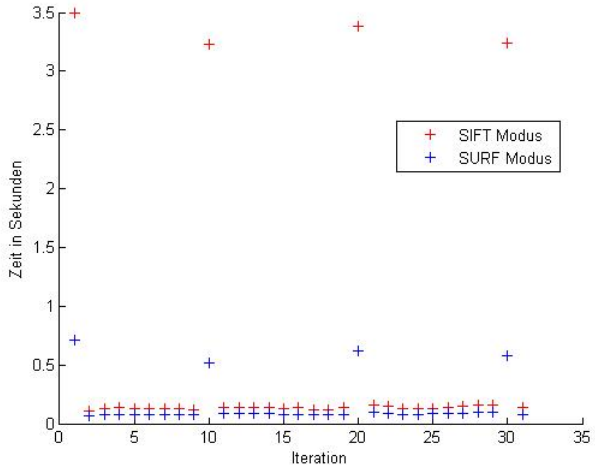


Abbildung 32: Rechenzeit für jeden Iterationsschritt in einer 30 Bildersequenz

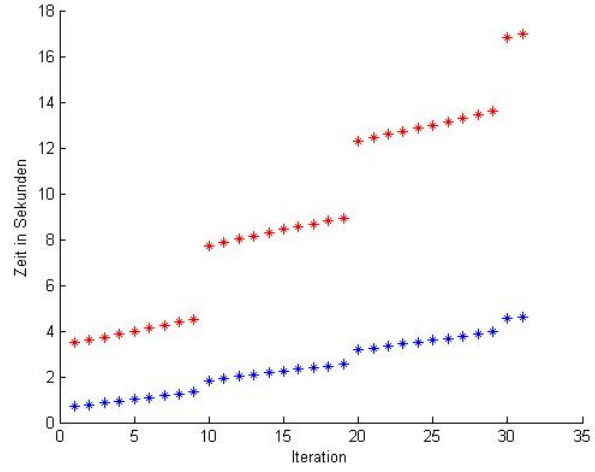


Abbildung 33: Aufsummierte Rechenzeit für eine 30 Bildersequenz

Die Grafiken 32 und 33 stellen den, basierend auf den vorherigen Tests, angenommen Zeitbedarf dar. So ist ein konstanter Rechenaufwand für den Optischen Fluss erkennbar, sowie ein signifikant höherer Bedarf für SIFT und SURF, wobei SIFT vier Mal soviel Zeit benötigt wie SURF. Um bei späteren Tests vergleichbare Daten zu erhalten, sollte folglich immer der Zeitraum von 10 Iterationen betrachtet werden.

Da der zeitliche Aspekt nun ausreichend geprüft ist, wird nun noch die Robustheit des Verfahrens getestet. Hierfür werden die Transformationen Translation, Rotation und Skalierung betrachtet. Außerdem wird die Verfolgung in einer stark texturierten Umgebung überprüft, da in den bisherigen Versuchsreihen sich das Objekt in schwach texturierter Umgebung befand.

Um die Referenzpunkte zu generieren wird auf die Grafik 14 zurückgegriffen. Damit wird zunächst eine Punktkorrespondenzsuche mit einem Bild 34 mit schwach texturierter Umgebung ausgeführt.

Abbildung 34: Ausgangsposition des Objektes für weitere Tests

Ausgehend von dieser Lage, werden die oben genannten Szenarios getestet. Als Erstes wird die Objektverfolgung bei Translation in Nähe eines weiteren Objektes und damit in einer stärker strukturierten Umgebung untersucht. Das Ergebnis zeigt die Abbildung 35. Wie man sieht, gehen nur wenige Punkte “verloren” und das, obwohl das Objekt vergleichbare Strukturen besitzt.

Abbildung 35: Translation des Objektes mit weiterem Objekt im Bild

Erfolgt nun jedoch eine stärkere Rotation des zu verfolgenden Objektes, dann versagt die Verfolgung komplett.

Abbildung 36: Rotation des Objektes mit weiterem Objekt im Bild

Ausführlichere Tests mit dem alleinstehenden Objekt ergaben, dass dies allein durch die Rotation des Objektes bedingt und unabhängig von der Umgebung ist. Bei steigendem Rotationswinkel gehen immer mehr Korrespondenzen verloren, was bei einer späteren echtzeitfähigen Anwendung auf Grund der kleinen Bildunterschiede irrelevant werden sollte.

Als letztes wird der Einfluss der Skalierung auf die Verfolgung betrachtet. Dabei erfolgt zugleich eine Translation, sowie eine Verstärkung der Umgebungstextur.

Abbildung 37: Skaliertes Objektes in stark strukturierter Umgebung

Bei diesem Test werden nur wenige falsche Positionen berechnet. Zusammenfassend lässt sich sagen, dass die Funktion robust in Bezug auf Translation und Skalierung ist, sowie teilrobust bei steigender Umgebungsstrukturierung. Für Rotationen gilt dies nur bei kleinen Änderungen.

6 Softwareintegration

Dieses Kapitel beschreibt die Integration des Optischen Flusses zur Objektverfolgung in die Arbeit von Benjamin Wagner [Wag11].

Dabei wird ebenfalls auf auftretende Schwierigkeiten eingegangen. Letztendlich wird noch ein Vergleich der ursprünglichen Umsetzung mit der neuen optimierten Lösung durchgeführt, um das Ziel dieser Ausarbeitung bestätigen zu können.

6.1 Visual Servoing

Unter Visual Servoing, auch Vision-Based Robot Control, werden Verfahren zusammen gefasst, die optische Sensoren zur Steuerung der Bewegung von Robotern nutzen. *Sanderson* und *Weiss* führten 1980 eine Klassifizierung [SW80] in zwei Kategorien ein :

1. Hierarchie des Regelsystems
    - Direkte Gelenksteuerung aus den Bildinformationen
    - Ausgang des Bildverarbeitungsreglers entspricht Eingang des Gelenkreglers
2. Definition der Regelabweichung
    - Die Regelabweichung wird aus den Bilddaten bestimmt
    - Die Regelung erfolgt über aus Bildinformationen gewonnenen Lagedaten

Hierbei kristallisierte sich heraus, dass die direkte Gelenkansteuerung weniger praktikabel ist. Gewichtige Gründe sind Folgende :

1. Hohe Abtastraten der Kamera nicht möglich
2. Kinematische Singularitäten
3. Kein einfacher Austausch von Komponenten (Bildregler, Gelenkregler)

Die hierarchischen Verfahren können als einer von drei Typus klassifiziert werden.

Die erste Kategorie dieser Anwendungen sind bildbasierte Visual Servoing Verfahren (IBVS - Image Based Visual Servoing). Hierbei wird aus einer Regelabweichung, die aus den Bilddaten gewonnen wird, eine Steuerung der Kamerabewegung berechnet, mit der diese Abweichung minimiert wird.

Ein Nachteil dieser Lösungen wird in [Cha98] beschrieben. Bei Konvergenz- und Stabilitätsprüfungen wurden Konfigurationen gefunden, bei denen die Wahl der Merkmale und Kameraparameter dazu führte, dass sich die Kamera immer weiter vom Zielpunkt entfernte, anstatt sich auf ihn zu zubewegen. Dieser Effekt wird mit *camera retreat* bezeichnet.

Die zweite Kategorie sind positionsbasierte Verfahren, die eine Lageschätzung anhand der Bildinformationen durchführen und dann eine Lageanpassung des Roboters ausführen. Die ersten Algorithmen hatten eine hohe Rechenzeit, wodurch es zu sogenannten *look and move* Aktionen kam. Dies konnte jedoch mit dem Einsatz verschiedener Verfahren, wie dem KALMAN-Filter gelöst werden. Viele Arbeiten greifen auf künstliche Marker am Objekt zurück um dieses verfolgen zu können.

Die dritte Kategorie ist letztlich eine Kombination der beiden vorherigen Ansätze und wird deshalb auch Hybrider Ansatz genannt. Durch die Vereinigung beider Grundideen wird versucht die Nachteile zu minimieren. Dazu werden meistens die translatorischen und rotatorischen Gelenke separat behandelt um eine bildbasierte und positionsbasierte Regelung durchzuführen.

Ein als 2,5D VS bezeichneter Ansatz nutzt die Epipolar-Geometrie um eine Rekonstruktion der Szene zu schaffen und damit die Kamerabewegung bis auf einen Skalierungsfaktor zu bestimmen. Für diese skalierungsunabhängigen und somit nur rotatorischen Bewegungsregelung wird dann ein rein positionsbasiertes Verfahren gewählt. Dafür werden jedoch Korrespondenzpunkte in zwei aufeinander folgenden Bildern benötigt.
Der Skalierungsfaktor wird dann mit einer bildbasierten Regelung aufgehoben. Dabei wird meist mit relativen und nicht mit absoluten Distanzen zwischen den aktuellen Merkmalen und den *Teach-In* Merkmalen gearbeitet. Bei der Bestimmung der Homographiematrix kann es jedoch immer noch, durch die Technik der optischen Sensoren und den daraus resultierenden Störungen im Bild, zu Fehlberechnungen kommen.

6.2 Vorhandene Software

Der in der [Wag11] verfolgte hybride Ansatz kombiniert eine direkte Ansteuerung mit einem bildbasierten Reglersystem. Die Umsetzung erfordert ein vorherigen *Teach-In* Schritt für jedes Objekt. Zunächst wird der Greifvorgang in zwei Phasen unterteilt: Die Positionierung und das Greifen des Objektes.

Der erste Abschnitt des Positioniervorganges ist das Suchen des Objektes im Bildbereich. Dabei fährt der Katana-Roboter in einem Halbkreis über den Arbeitsbereich und versucht das Objekt durch Merkmalswiedererkennung zu lokalisieren. Als Merkmale wurden SIFT Punkte gewählt (Begründung s. [Wag11] Kapitel 2.2). Sobald dies geschehen ist, wird eine erste direkte Ansteuerung vorgenommen, indem auf Basis einer Teilrekonstruktion die Objektposition in Relation zur Kamera bestimmt wird. Damit wird eine schon sehr nahe Übereinstimmung der aktuellen Ansicht mit der Referenzansicht erreicht. Für die restliche Angleichung, bis auf eine definierte Fehlermarge, wird nun auf die bildbasierte Regelung zurückgegriffen. Ist die Referenzansicht erreicht, kann der vorher per *Teach-In* Schritt angelernte Greifvorgang durchgeführt werden.

Vorteile des Verfahrens

- Vermeidung des *camera retreat* durch direkte Steuerung
- Robustheit gegenüber Bildstörungen und Kalibrierungsungenauigkeiten
- Keine Vollrekonstruktion des Objektes nötig
- Schneller als reine Bildregelung

Nachteile des Verfahrens

- *Teach-In* nötig für Greifvorgang
- Objekt wird nur bezüglich der Referenzansicht erkannt
- Module lassen sich schwer austauschen, durch Kopplung von Regelung und direkter Steuerung

6.3 Integration

Als Erstes wird nun die vorhandene Implementation analysiert um den exakten Part zu finden, der ersetzt werden muss. Aus [Wag11] geht hervor, dass der Suchvorgang in drei Abschnitte geteilt ist. Zunächst die Lokalisierung des Objektes im Arbeitsbereich, zweitens die direkte Ansteuerung und drittens dann die Regelung, um eine möglichst exakte Positionierung zu ermöglichen. Eine erste Lokalisierung der Komponenten zeigt, dass der gesamte Suchvorgang in einer Datei nacheinander abgearbeitet wird. Zunächst wird die Routine wiederholt aufgerufen bis das Objekt lokalisiert wird. Dabei wird der Arbeitsbereich in 3-Grad-Schritten des ersten Gelenkes abgesucht. Ist das Objekt lokalisiert, wird der zweite Teil der Funktion ausgeführt, die direkte Ansteuerung. Im Anschluss wird dann zur Regelung übergegangen. Der dritte Abschnitt ist somit der Bereich, der modifiziert werden muss. Aufgrund der durchgängigen Verwendung wird auf eine Separierung der Datei verzichtet. Der bisherige Ablauf der Regelung wird im folgenden vereinfacht dargestellt.

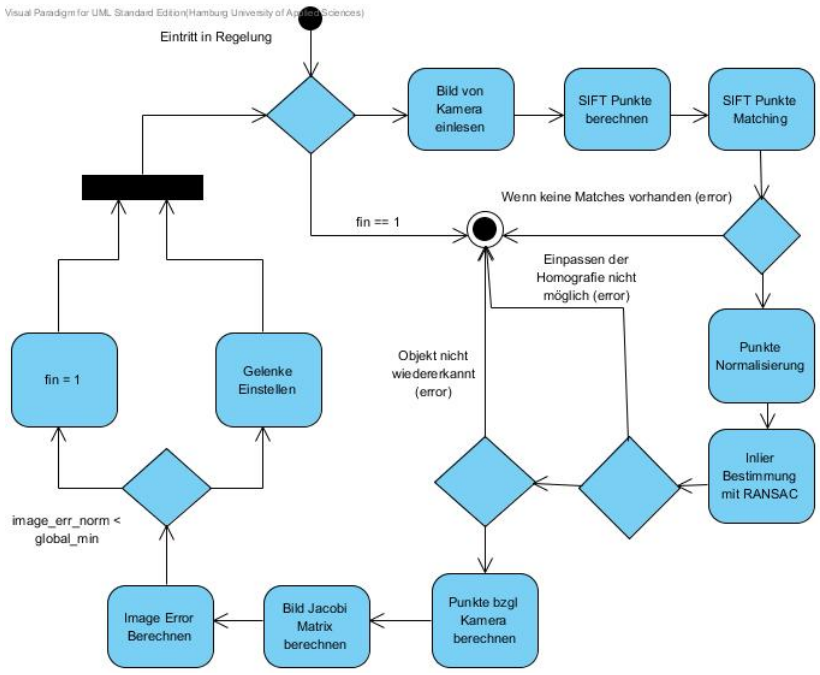


Abbildung 38: Regelungsablauf in der Wagner Implementierung

Im Vergleich dazu die modifizierte Regelung mit eingebauter *Combined Tracking* Funktion. Die Abbildung zeigt, dass nur im ersten Abschnitt, bis zur Bestimmung der Bild-Jacobi Matrix, Änderungen vorgenommen wurden.

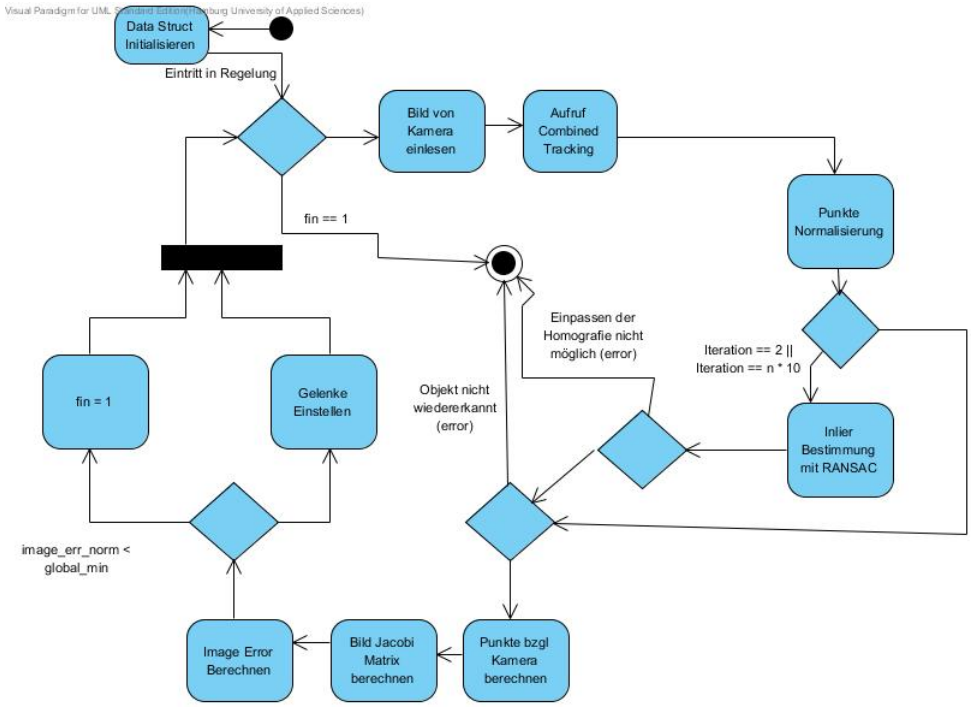


Abbildung 39: Modifizierter Regelungsablauf

6.4 Vergleich der alten und neuen Regelung

Ein erster Vergleich der Lokalisierung und der Verfolgung in einem Regelungsschritt ergibt eine zeitliche Verbesserung um 50% von im Mittel 8,8 auf 4,4 Sekunden, die aber noch nicht der gewünschten Echtzeitnähe entspricht. Eine detailliertere Analyse ergibt, dass von den 8,8 beziehungsweise 4,4 Sekunden 1,2 Sekunden auf die Kalkulierung und Einstellung der Gelenkwinkel entfällt. Damit entfallen 7,6 Sekunden auf die Punktlokalisierung, während die Punktverfolgung 3,2 Sekunden dauert. Dies überrascht, da in 4.3.5 gezeigt wird, dass die Berechnung im letzteren Fall bei weit unter einer Sekunde liegt.

Bei einer noch genaueren Betrachtung der Zeiten ergeben sich die in der nächsten Tabelle dargestellten durchschnittlichen Zeiten für die beiden unterschiedlichen Regelungsschrittarten.

	Lokalisierung(SIFT)	Verfolgung(OF)
Bild einlesen	0,35 Sekunden	0,35 Sekunden
Punktneubestimmung	4,5 Sekunden (SIFT)	0,06 Sekunden (OF)
Punktnormalisierung	3 Sekunden	3 Sekunden
Gelenkberechnung und Einstellung	1,2 Sekunden	1,2 Sekunden

Hiermit wird die Normalisierung der Punkte, die für die Berechnung der Bild-Jacobi-Matrix benötigt werden, als zweitgrößter Zeitfaktor identifiziert.

Weiterhin wird neben dem erhöhten Zeitbedarf, im Vergleich zur ursprünglichen Version, bei vereinzelten Durchläufen der sogenannte *camera retreat* beobachtet, der gemäß [Wag11] durch die direkte Ansteuerung nicht mehr eintreten sollte. Daneben tritt in ungefähr der Hälfte der Versuche ein weiteres Problem bei der Regelung mit Optischem Fluss auf, dass bei reiner SIFT Regelung ebenfalls nicht auftritt. Nachdem mehrere Schleifendurchgänge ohne Probleme abgearbeitet werden, tritt bei der Bestimmung der Bild-Jacobi-Matrix keine lokale asymptotische Stabilität ein. Daraus resultiert eine fehlerhafte Berechnung der extrinsischen Kameraparameter in einem späteren Regelungsschritt. Die ungültige Kalkulation resultiert in einem *Not a Number* Ergebnis. Eine weitere Regelung, auch mit den Daten vorheriger Iterationen, ist nicht möglich. Manchmal stellt sich auch eine Konvergenz der Regelung nicht mehr ein.

Somit ergeben sich nun drei Problemstellungen :

1. Zeitaufwand der Normalisierung der Punkte
2. *camera retreat* in manchen Fällen
3. Teilweise keine Konvergierung des Verfahrens

Diese drei Probleme lassen sich jedoch nicht ohne weiteres reproduzieren und sind numerischer und nicht bildverarbeitungstechnischer Natur und werden in dieser Arbeit nicht weiter betrachtet.

Trotz der auftretenden Fehler, war es möglich eine komplette Regelung mit der in 5 entwickelten Methodik zu erreichen. Dabei konvergiert das Verfahren nach 51 Regelungsschritten und einem Zeitaufwand von etwas mehr als 4 Minuten. Eine im Anschluss durchgeführte Regelung mit der Implementierung von Benjamin Wagner mit gleichen Ausgangsparametern konvergiert nach 69 Iterationen und einer benötigen Zeit von rund 12 Minuten und 15 Sekunden. Damit ergibt sich basierend auf den Messungen für 10 Durchläufe bei beiden Umsetzungen ein Zeitvorteil von 54,8% für die neue Entwicklung im Vergleich mit der Ursprünglichen. Durch die höhere Iterationsanzahl bei Letzterem konvergiert das neue Verfahren sogar in nur 25% der Zeit des Vorherigen. Da die Schrittanzahl nicht konstant bei gleichen Ausgangsparametern ist, ist der zeitliche Vorteil von 54,8% der realistischere Wert.

In Anbetracht dessen, dass es sich um eine bildbasierte Regelung handelt, wird hier nun auch ein Vergleich der Konvergierung der Umsetzungen aufgeführt. Hierfür ist der Bildfehler, auch *image error* genannt, in Abhängigkeit vom Regelungsschritt zu betrachten. Dies wird durch die nächsten beiden Graphen visualisiert, die den Verlauf des Bildfehler bei den beiden oben erfolgreich durchgeführten Durchläufen wiedergeben :

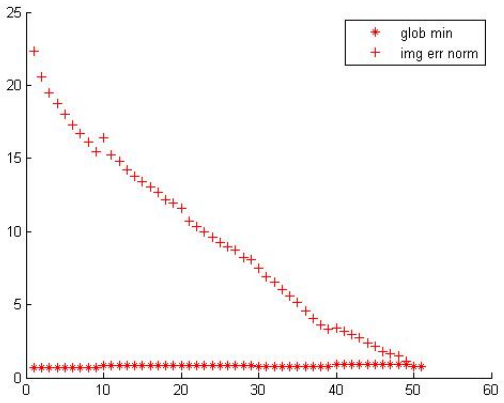


Abbildung 40: Bildfehlerverlauf bei einer erfolgreichen Regelung mit Combined Tracking

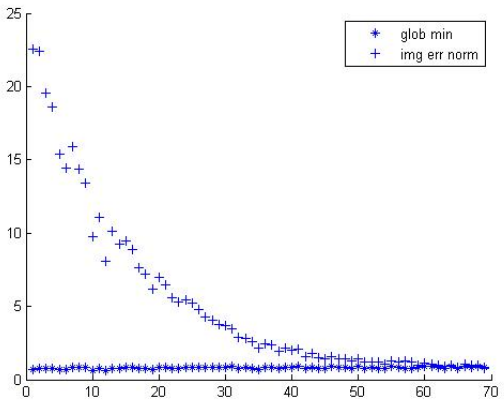


Abbildung 41: Bildfehlerverlauf bei alter Regelung

Auffallend ist, dass die neue Umsetzung konstanter und linearer konvergiert, während die alte Lösung eher einen logarithmischen Abfall gemäß der Formel $\log_{\frac{1}{2}} x + b$ produziert und insbesondere zu Beginn stärker schwankt. Desweiteren ist das bestimmte globale Minimum, bei dessen Unterschreitung die Regelung terminiert, bei der neuen Implementierung für jeweils 10 Iterationen konstant, da sich die Punktanzahl nur beim SIFT/SURF Schritt ändert.

Dies zeigt auch der Vergleich zweier Aufnahmen an einem Regelungsschrittmoduswechsel. Eine umfassendere Sequenz ist im Anhang C.2 zu finden.

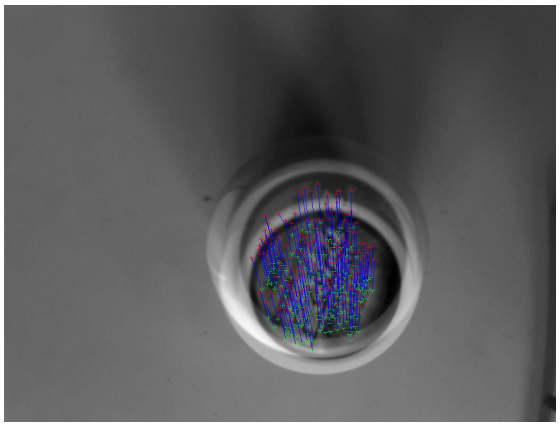


Abbildung 42: Punktkorrespondenzen bei Iteration 29

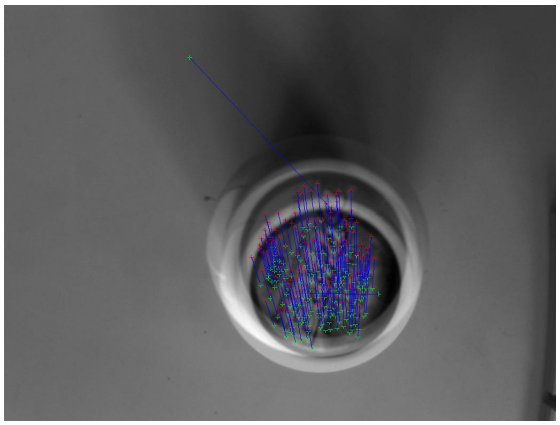


Abbildung 43: Punktkorrespondenzen bei Iteration 30

Hierbei wird verdeutlicht, dass die gewählte SIFT Implementierung, Outlier entfernen und generieren kann, somit zu besseren und schlechteren Punktkorrespondenzen führt, je nach Ergebnis der Punktkorrespondenzsuche. Daraus ergibt sich ein Optimierungspotenzial bei den Korrespondenzen beim SIFT Verfahren. Da es neben der gewählten SIFT Implementierung noch weitere schnellere Realisierungen gibt, sollte bei späterer Verwendung des Verfolgungsalgorithmus eine Substitution oder die Verwendung von SURF angestrebt werden.

7 Zusammenfassung und Ausblick

In dieser Ausarbeitung wurde erfolgreich gezeigt, dass sich der Regelungsvorgang eines bildgestützten kinetischen Systems mit Hilfe des Optischen Flusses beschleunigen lässt. Hierfür wurden verschiedene Implementierungen analysiert und in Bezug auf Punktgenauigkeit und Rechenzeit getestet. Das effektivste Verfahren wurde mit einer Merkmalerkennung kombiniert, um eine anwendungsunabhängige Umsetzung zu erzeugen. Diese ermöglichte in einer praktischen Anwendung mehr als eine Halbierung der benötigten Zeit. Dabei konvergierte die Regelung zudem konstanter und in weniger Schritten.

In Bezug auf die Arbeit von Benjamin Wagner stellt sich selbst mit den durchgeführten Optimierungen nicht die erhoffte Echtzeitfähigkeit, weswegen es hier noch Potenzial für weitere Arbeiten gibt, die eine Beschleunigung im numerischen Abschnitt der von Benjamin Wagner vorgestellten Methodik vornehmen müssten. Dabei sollten dann gleichzeitig die im vorherigen Kapitel genannten Problemstellungen analysiert und behoben werden. Auch scheinen die ausgeführten Gelenkeinstellungen in zu kleinen Schritten zu erfolgen. Eine weitere Ergänzungsmöglichkeit ist die Beschränkung des Koordinatensystem auf fünf Freiheitsgrade, um daraus resultierende überflüssige Gelenkeinstellungstests zu vermeiden und die inverse Kinematik zu beschleunigen. Damit ließe sich auch eine Beschädigung des Roboters durch Beschränkung der inversen Kinematik realisieren. Außerdem stehen immer noch die von Herrn Wagner selbst vorgeschlagenen Verbesserungen in Bezug auf Objektwiedererkennung aus verschiedenen Perspektiven, sowie die Vereinfachung und Reduzierung des *Teach-In* Schrittes aus.

Unabhängig von der Arbeit von Benjamin Wagner, ließe sich die hier entwickelte Objektverfolgung auf multiple Objekte erweitern und es könnten Hindernisvermeidungsalgorithmen für mobile Vehikel entwickelt werden. Dabei sollte generell das implementierte SIFT Verfahren überarbeitet werden, falls es weiter genutzt wird, da dieses häufiger als der SURF Algorithmus weit von der tatsächlichen Position entfernte Zuordnungen findet. Zudem sind modernere Realisierungen von SIFT deutlich schneller. Weiterhin wäre eine dynamischere Wahl des Regelschritttypus praktisch, da zum Beispiel bei Bildern mit vielen Strukturen eine häufigere Neulokalisierung vonnöten ist, um die Verfolgung zu gewährleisten. Dabei sollte betrachtet werden, ob die Modusauswahl von extern angestoßen werden soll oder die Funktion dies selbst intelligent erfasst.

Zusammenfassend kann gesagt werden, dass in dieser Arbeit die Grundlagen für eine Vielzahl von regelungsbasierten Objektverfolgungen in Echtzeit gelegt wurde.

Literatur

[BETG08] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *Computer Vision and Image Understanding (CVIU)*, volume 110, pages 346–359, 2008.

[BFB94] J.L. Barron, D.J. Fleet, and S.S. Beauchemin. Performance of optical flow techniques. *International Journal of Computer Vision*, 1994.

[Bou00] Jean-Yves Bouguet. Pyramidal implementation of the lucas kanade feature tracker description of the algorithm, 2000.

[Cha98] F. Chaumette. Potential problems of stability and convergence in image-based and position-based visual servoing. In *The Confluence of Vision and Control LNCIS*, volume 237, pages 66–78. Springer Verlag, 1998.

[HS80] Berthold K. P. Horn and Brian G. Schunk. Determining optical flow. A.I. Memo, April 1980.

[Int01] Intel Corporation. *Open Source Computer Vision Library - Reference Manual*, 2001.

[LK] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of Imaging Understanding Workshop*.

[Low99] David G. Lowe. Object recognition from local scale-invariant features. In *The Proceedings of the Seventh IEEE International Conference on Computer Vision*, pages 1150–1157, 1999.

[Luc84] Bruce D. Lucas. *Generalized Image Matching by the Method of Differences*. PhD thesis, Carnegie-Mellon University, 1984.

[LYT08] Ce Liu, Jenny Yuen, and Antonio Torralba. Sift flow: Dense correspondence across scenes and its applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2008.

[SE08] Yoad Snapir and Rachely Esman. Opencv and mex files quick guide, 2008.

[SW80] A.C. Sanderson and L.E. Weiss. Image-based visual servo control using relational graph error signals. In *Proceedings of IEEE Int. Conf. Robot. Autom.*, 1980.

[Wag11] Benjamin Wagner. Kamerageführte Positionierung eines Roboterarms für Handhabungsaufgaben. Master's thesis, HAW Hamburg, 2011.

Abbildungsverzeichnis

A Quellenverzeichnis

- Abbildung 1 :
  http://www.neuronics.com/cms_de/mediabase/pdf_brochure/Neuronics_LinuxRobot_UniKit_DE.pdf
- Abbildung 2 :
  http://www.neuronics.com/cms_de/mediabase/downloads/downloads_table/3D/8_Winkelgreifer_mit_Finger.jpg
- Abbildung 3 :
  http://www.neuronics.com/cms_de/mediabase/downloads/downloads_table/3D/2_Katana450_6M180.jpg
- Abbildung 4 :
  http://www.alliedvisiontec.com/de/produkte/kameras/firewire/guppy/f-080bc.html
- Abbildung 5 :
  http://security-systems.pentax.de/de/product/C60402KP/ssd_products_image_processing.php
- Abbildungen 6, 7, 9 und 8 :
  Vorlesungsunterlagen zu Camera Based Interaction (Kapitel 2& 3)von Prof. Florian Vogt (https://pub.informatik.haw-hamburg.de/home/pub/prof/vogt/course/lectures/(HAW pub - eingeschränkter Zugang)) - Original Quelle unbekannt
- Abbildung 19 :
  http://www.mathworks.com/matlabcentral/fileexchange/22756-horn-schunck-optical-flow-method

B Visual Studio

B.1 MEX-File

```
1  #include <stdio.h>
2  #include <yvals.h>
3  #if (_MSC_VER >= 1600)
4  #define __STDC_UTF_16__
5  #endif
6  #include "mex.h"
7  #include "OF.h"
8  #include <malloc.h>
9
10 #define IN_IMAGE_A prhs[0]
11 #define IN_DIMENSIONS_A prhs[1]
12 #define IN_IMAGE_B prhs[2]
13 #define IN_DIMENSIONS_B prhs[3]
14 #define IN_FN prhs[4]
15 #define IN_FX prhs[5]
16 #define IN_FY prhs[6]
17
18
19 void mexFunction(int nlhs, mxArray **plhs, int nrhs, const mxArray
       **prhs) {
20         bool intInput = true;
21
22         if(nrhs != 7)
23                 mexErrMsgTxt("Usage: BA_V2(image1, dim_image1,
                       image2, dim_image2, f_n, fx, fy)");
24
25         if( mxIsUint8(IN_IMAGE_A) || mxIsUint8(IN_IMAGE_A) )
26                 intInput = true;
27         else if( mxIsSingle(IN_IMAGE_A) || mxIsSingle(IN_IMAGE_B))
28                 intInput = false;
29         else
30                 mexErrMsgTxt("Input should be a matrix of uint8 or
                       single precision floats.");
31
32         if(( mxGetNumberOfElements(IN_DIMENSIONS_A) != 3 )||(
               mxGetNumberOfElements(IN_DIMENSIONS_B) != 3 ))
```

```
33                  mexErrMsgTxt("Dimension vector should contain two
                        elements: [width, height, widthstep].");
34
35
36          /*Get Data of Image 1*/
37          char *ImageA = (char *)mxGetData(IN_IMAGE_A);
38          double *imgSizeA =  mxGetPr(IN_DIMENSIONS_A);
39          double I1WS= *(imgSizeA+2);
40
41          /*Get Data of Image 2*/
42          double *imgSizeB = mxGetPr(IN_DIMENSIONS_B);
43          char *ImageB = (char *)mxGetData(IN_IMAGE_B);
44          double I2WS = *(imgSizeB+2);
45
46          /*Get Features*/
47          double* featuresInX = mxGetPr(IN_FX);
48          double* featuresInY= mxGetPr(IN_FY);
49          double* InN = mxGetPr(IN_FN);
50
51          /*Get Output Pointer*/
52          plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
53          double* OnN = mxGetPr(plhs[0]);
54          plhs[1] = mxCreateDoubleMatrix(1, *InN, mxREAL);
55          double* featuresOutX = mxGetPr(plhs[1]);
56          plhs[2] = mxCreateDoubleMatrix(1, *InN, mxREAL);
57          double* featuresOutY = mxGetPr(plhs[2]);
58
59          /*Calculate Optical Flow*/
60          OF(featuresInX,featuresInY,InN,featuresOutX,featuresOutY,
                OnN, ImageA, imgSizeA, I1WS, ImageB, imgSizeB, I2WS);
61
62  }
```

B.2 OpenCV Interface Funktion

```
1  #include <stdio.h>
2  #include <malloc.h>
3  #include <cv.h>
4  #include <cxcore.h>
5  #include <highgui.h>
6  #include <cvaux.h>
7  #include <ml.h>
8  #include <cxmisc.h>
9  #include <cvwimage.h>
10 #include "OF.h"
11
12 const int MAX_FEATURES = 50000;
13
14 void createFeatureList(CvPoint2D32f* flist, double* fdataX, double*
       fdataY, int fsize);
15 void cvpoint2double(CvPoint2D32f* fdata, double* flistX, double*
       flistY, int fsize);
16
17 void OF(double* featuresInX, double* featuresInY, double* InN,
       double* featuresOutX, double* featuresOutY, double* OnN, char*
       im1,double* I1S,double I1WS, char* im2, double* I2S,double I2WS)
18 {
19     /*Image A*/
20     size_t width  = (size_t) malloc(sizeof(size_t));
21     size_t height = (size_t) malloc(sizeof(size_t));
22     width  = (size_t) I1S[0];
23     height = (size_t) I1S[1];
24     CvSize size;
25     size.height = height;
26     size.width = width;
27     IplImage *imgA = cvCreateImageHeader(size, IPL_DEPTH_8U, 1);
28     imgA->imageData = im1;
29     size_t widthStepI1 = (size_t) I1WS * (sizeof(unsigned char));
30     imgA->widthStep = widthStepI1;
31     imgA->imageDataOrigin = imgA->imageData;
32     /*Image B*/
33     width  = (size_t) I2S[0];
34     height = (size_t) I2S[1];
```

```
35      size.height = height;
36      size.width = width;
37      IplImage *imgB = cvCreateImageHeader(size, IPL_DEPTH_8U, 1);
38      imgB->imageData = im2;
39      size_t widthStepI2 = (size_t) I2WS * (sizeof(unsigned char));
40      imgB->widthStep = widthStepI2;
41      imgB->imageDataOrigin = imgB->imageData;
42
43      CvSize img_sz = cvGetSize( imgA );
44      int win_size = 15;
45
46      // Get the features for tracking
47      IplImage* eig_image = cvCreateImage( img_sz, IPL_DEPTH_8U, 1 );
48      IplImage* tmp_image = cvCreateImage( img_sz, IPL_DEPTH_8U, 1 );
49
50      int feature_count = MAX_FEATURES;
51      int f_s = *(InN);
52      CvPoint2D32f* featuresA = new CvPoint2D32f[ f_s];
53      createFeatureList(featuresA, featuresInX,featuresInY, f_s);
54      CvPoint2D32f* featuresB = new CvPoint2D32f[ f_s ];
55
56      // Call Lucas Kanade algorithm
57      char* features_found = (char*) malloc(sizeof(char)* f_s );
58      float* feature_errors = (float*) malloc(sizeof(float)* f_s);
59
60      cvCalcOpticalFlowPyrLK( imgA, imgB, NULL, NULL, featuresA,
           featuresB, f_s,
61                  cvSize( win_size, win_size ), 5, features_found,
                       feature_errors,
62                   cvTermCriteria( CV_TERMCRIT_ITER | CV_TERMCRIT_EPS
                        , 20, 0.3 ), 0 );
63      /* Create Output */
64
65      int fsize = 0;
66      for(int i = 0; i < f_s; i++)
67      {
68          if(features_found[i] == 1)
69              fsize++;
70      }
71
```

```
72      cvpoint2double(featuresB, featuresOutX,featuresOutY, fsize);
73      *(OnN) = fsize;
74  }
75
76  void createFeatureList(CvPoint2D32f* flist, double* fdataX, double*
        fdataY, int fsize)
77  {
78      for(int i = 0; i < fsize; i++)
79          flist[i] = cvPoint2D32f( *(fdataX+i), *(fdataY+i) );
80  }
81
82
83  /*Converts a cvpoint back to double format*/
84  void cvpoint2double(CvPoint2D32f* fdata, double* flistX, double*
       flistY, int fsize)
85  {
86      for(int i = 0; i < fsize; i++)
87          {
88          *(flistX+i) = fdata[i].x;
89          *(flistY+i) = fdata[i].y;
90          }
91  }
```

B.3 Moduldefinitionsdatei

```
LIBRARY "{Projektname}.mexw32"
EXPORTS mexFunction
```

B.4 Visual Studio Projekteinstellungen

Es wird angenommen das die Installationen unter *C:\OpenCV2.1* und *C:\Program Files\MATLAB* liegen.

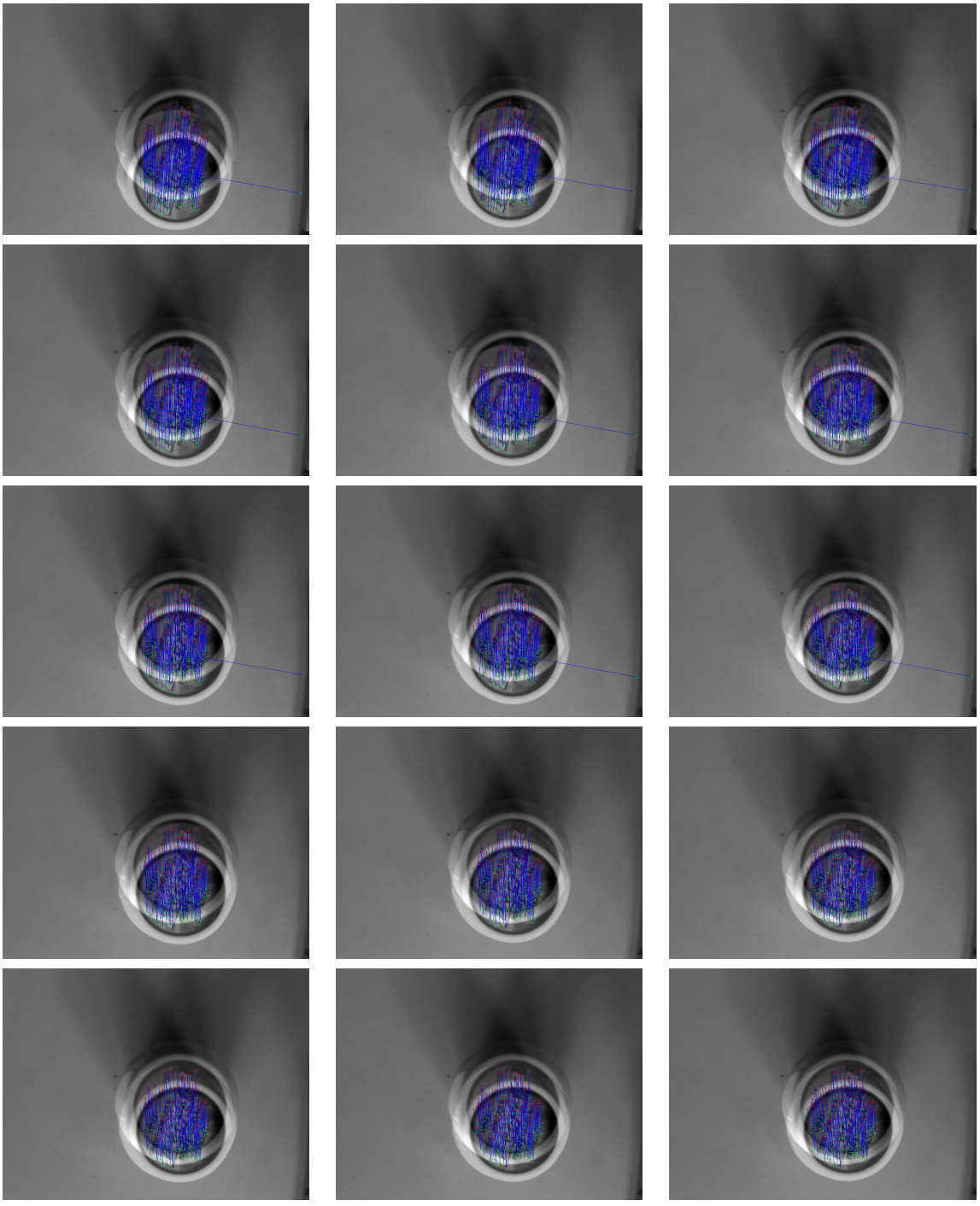
- Allgemein
  - Zielerweiterung
    - .mexw32
- VC++-Verzeichnisse
  - Includeverzeichnisse
    - C:\Program Files\MATLAB\Version\extern\include
    - C:\OpenCV2.1\include\opencv
  - Bibliotheksverzeichnisse
    - C:\Program Files\MATLAB\version\extern\lib\win32\microsoft
    - C:\OpenCV2.1\lib
  - Quellverzeichnisse
    - C:\OpenCV2.1\src\cv
    - C:\OpenCV2.1\src\cvaux
    - C:\OpenCV2.1\src\cxcore
    - C:\OpenCV2.1\src\highgui
    - C:\OpenCV2.1\src\ml
- C/C++
  - Allgemein
    - Zusätzliche Includeverzeichnisse
      - C:\Program Files\MATLAB\version\extern\include
  - Präprozessor
    - Präprozessordefinitionen
      - WIN32
      - MATLAB_ MEX_ FILE
  - Vorkomipilierter Header
    - Vorkompilierter Header
      - Vorkompilierter Header nicht verwenden

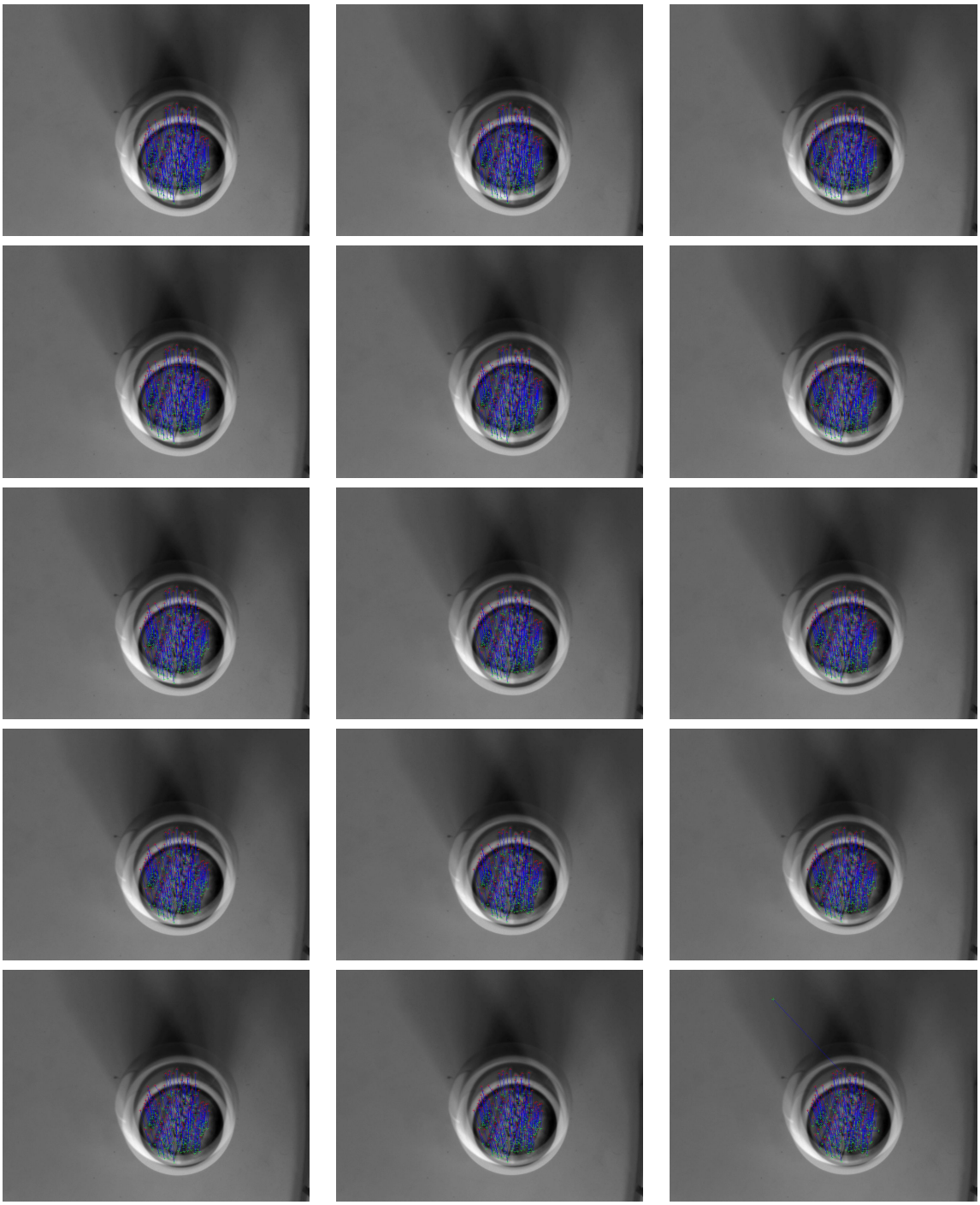
- Linker
  - Allgemein
    - Ausgabedatei
      - $ (OutDir) $ (TargetName).mexw32
    - Zusätzliche Bibliotheksverzeichnisse
      - C:\Program Files\MATLAB\version\extern\lib\win32\microsoft
  - Eingabe
    - Zusätzliche Abhängigkeiten
      - libmx.lib
      - libmex.lib
      - libmat.lib
      - cv210.lib
      - cxcore210.lib
      - highgui210.lib
    - Modulsdefinitionsdatei
      - mex.def

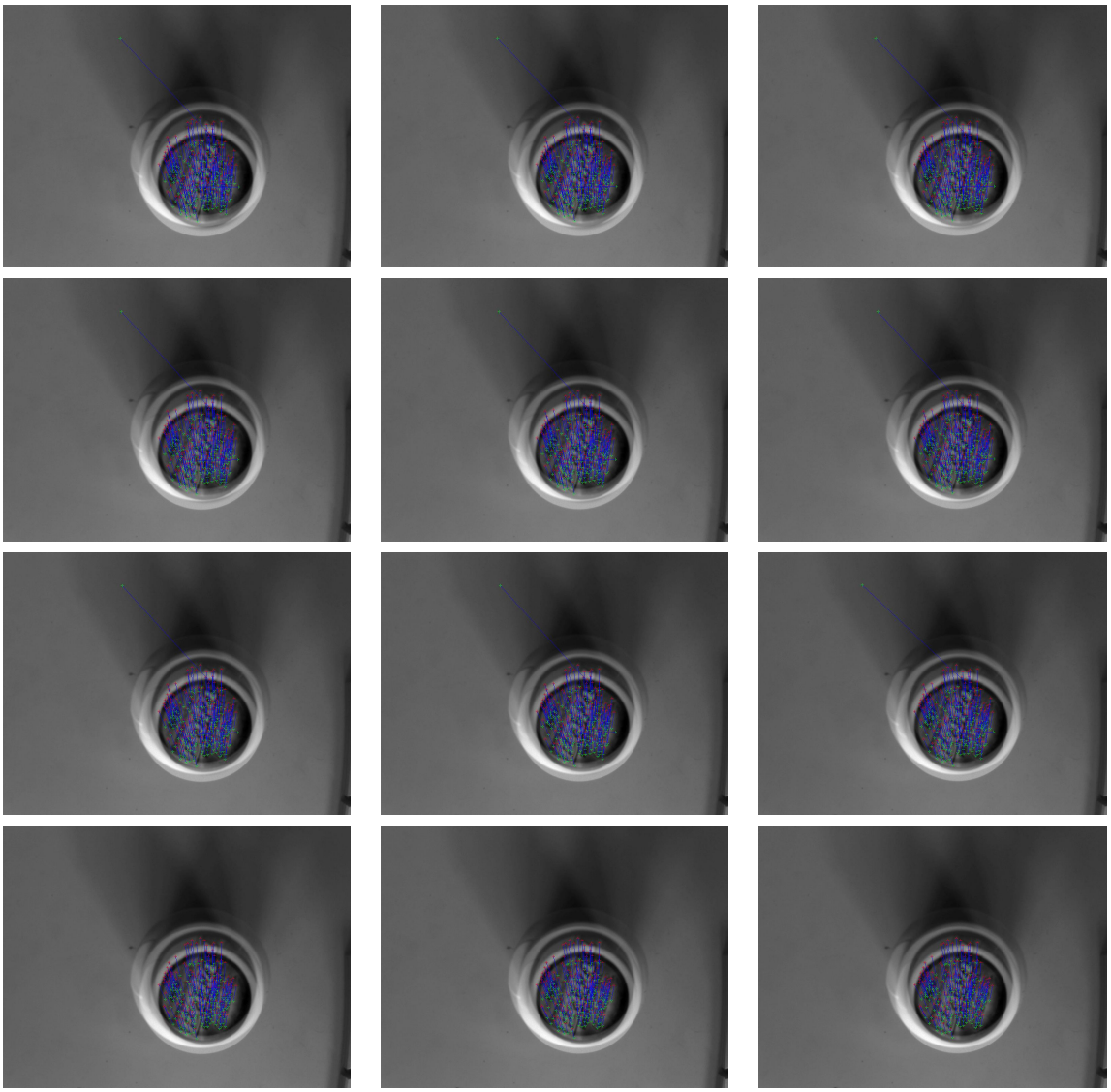
C MATLAB

C.1 Testreihen-Bildersequenz

C.2 Ausschnitt der Bilder einer Regelung (Iteration 1 - 42)







C.3 Code : Combined Tracking

```
1  function [ data_out ] = combined_tracking( mode, img, data)
2  %COMBINED_TRACKING Combined Feature Tracking with SIFT/SURF/Optical
       Flow
3  %   [ data_out ] = combined_tracking( img, data, parameters)
4  %      INPUT:
5  %           IMG  : actual image
6  %           DATA : Struct :
7  %                  DATA.FEATURES  : Last Features Locations
8  %                  DATA.DESC      : Last Features Descriptors
9  %                  DATA.IMG       : Ref Image
10 %                  DATA.OFIMG     : Image from last iteration
11 %                  DATA:ITERATION : Actual Iteration
12 %                  MODE : 'SIFT' or 'SURF'
13 %           OUTPUT:
14 %                  DATA_OUT : Struct :
15 %                  DATA_OUT.FEATURES  : Last Features Locations
16 %                  DATA_OUT.DESC      : Last Features Descriptors
17 %                  DATA_OUT.IMG       : Ref Image
18 %                  DATA_OUT.OFIMG     : Image from last iteration
19 %                  DATA_OUT:ITERATION : Next Iteration
20 %   Dependencies :
21 %       vl_feat
22 %       SURF_mex
23 %
24 %   Literature :
25 %
26 %   Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool.
27 %   Surf: Speeded up robust features.
28 %       In Computer Vision and Image Understanding (CVIU),
29 %   volume 110, pages 346?359, 2008.
30 %
31 %   Jean-Yves Bouguet.
32 %       Pyramidal implementation of the lucas kanade feature
       tracker,   2000.
33 %
34 %   David G. Lowe.
35 %       Object recognition from local scale-invariant features.
```

```
36 %    In The Proceedings of the Seventh IEEE International
                                Conference on Computer Vision,
37 %    pages 1150?1157, 1999.
38
39
40
41 %% Input argument check
42 if(nargin < 3)
43     error('Too few arguments.');
44 end
45
46 if(isfield(data, 'iteration'))
47     if(data.iteration == 1)
48         if(isfield(data,'features')&& isfield(data,'desc'))
49             ref_m = 'features';
50             data.rdesc = data.desc;
51         elseif(isfield(data, 'img'))
52             ref_m = 'img';
53         else
54             error('No Reference View found in struct "data".');
55         end
56     else
57         if(((~isfield(data, 'features'))&&(~isfield(data, 'desc')))
                ||...
58            ((~isfield(data, 'features'))&&(~isfield(data, 'ofimg'))
                ))
59             error('Missing data in struct "data". Possibly init is
                    required.');
60         else
61             if(size(data.features,1)~=2)
62                 error('Wrong feature structure. Format 2xn needed.'
                        );
63             end
64         end
65     end
66 else
67     data.iteration = 1;
68     if(isfield(data,'features')&& isfield(data,'desc'))
69         ref_m = 'features';
70         data.rdesc = data.desc;
```

```
71      elseif(isfield(data, 'img'))
72          ref_m = 'img';
73      else
74          error('No Reference View found in struct "data".');
75      end
76  end
77  %% Calculation
78  if(data.iteration == 1 || mod(data.iteration,10) == 0)
79      if(strcmp(mode,'SIFT'))
80          % SIFT Parameters and Modifications
81          peak_thresh = 0; % umso höher, desto weniger features
82          edge_thresh = 100; % umso höher, desto mehr features
83          scale_dist = 1.4;
84          if(size(img,3) == 3)
85              img = rgb2gray(img);
86          end
87          img2 = single(img);
88          % Calculate Features
89          [f,d] = vl_sift(img2, 'PeakThresh', peak_thresh, '
                edgethresh', edge_thresh);
90          if(data.iteration == 1)
91              if(strcmp(ref_m,'img'))
92                  if(size(data.img,3) == 3)
93                      img3 = rgb2gray(data.img);
94                  else
95                      img3 = data.img;
96                  end
97                  img3 = single(img3);
98                  [data.features,data.desc] = vl_sift(img3, '
                        PeakThresh', peak_thresh, 'edgethresh',
                        edge_thresh);
99                  data.rdesc = data.desc;
100             end
101         end
102         % Matching
103         [matches, scores] = vl_ubcmatch(data.rdesc, d, scale_dist);
104         % Output Creation
105         data_out.features = f(1:2,matches(2,:));
106         data_out.desc = d(:,matches(2,:));
107         data_out.ofimg = img;
```

```
108         data_out.matches = matches;
109         data_out.rdesc = data.rdesc;
110     elseif(strcmp(mode,'SURF'))
111         % SURF Modifications
112         if(size(img,3) == 3)
113             img = rgb2gray(img);
114         end
115         % Calculate Features
116         [f, d] = surfpoints(img);
117         if(data.iteration == 1)
118             if(size(data.img,3) == 3)
119                 img2 = rgb2gray(data.img);
120             else
121                 img2 = data.img;
122             end
123             [data.features,data.desc] = surfpoints(img2);
124             data.rdesc = data.desc;
125         end
126         % Matching
127         [matches] = surfmatch(data.rdesc, d);
128         % Output Creation
129         data_out.features = f(1:2,matches(2,:));
130         data_out.desc = d(:,matches(2,:));
131         data_out.ofimg = img;
132         data_out.matches = matches;
133         data_out.rdesc = data.rdesc;
134     else
135         error('Bad parameters.');
136     end
137 else
138     % Modify Image 1
139     tim1 = permute( data.ofimg(:,:,:), [2 1 3] );
140     im1_dim = [size(tim1,1), size(tim1,2)];
141     im1_ws = size(tim1,1) + mod( size(tim1,1), 4 );
142     t2im1 = uint8(zeros(im1_ws, size(tim1,2)));
143     t2im1(1:size(tim1,1), 1:size(tim1,2)) = tim1;
144     tim1 = t2im1;
145     % Modify Image 2
146     tim = permute( img(:,:,:), [2 1 3] );
147     im_dim = [size(tim,1), size(tim,2)];
```

```
148     im_ws = size(tim,1) + mod( size(tim,1), 4 );
149     t2im = uint8(zeros(im_ws, size(tim,2)));
150     t2im(1:size(tim,1), 1:size(tim,2)) = tim;
151     tim = t2im;
152     % Use Optical Flow
153     f_n  = size(data.features,2);
154     [f_n, m2x, m2y] = ocv_pyrlk(tim1,[im1_dim im1_ws], tim,[im_dim
            im_ws],f_n,data.features(1,:),data.features(2,:));
155     % Output Creation
156     data_out.features = [m2x;m2y];
157     data_out.ofimg = img;
158     data_out.matches = data.matches;%[1:size(data_out.features,2)
            ;1:size(data_out.features,2)];%
159     data_out.desc = data.desc;
160     data_out.rdesc = data.rdesc;
161 end
162 data_out.iteration = data.iteration + 1;
163 end
```

D Inhalt der CD-ROM

Die beiliegende beinhaltet die folgenden Ordner :

```
pdf
projekt_ct
projekt_wagner
```

Der Ordner *pdf* enthält die Arbeit im PDF-Format.

Das Verzeichnis `projekt_ct` enthält diese Ordner :

```
Combined Tracking
Modifizierte Wagner Dateien
SURFmex
vlfeat
```

Der Ordner *Combined Tracking* enthält die entwickelte Funktionalität. *SURFmex* und *vlfeat* sind die dafür benötigten Toolboxen. Der vierte Ordner enthält die modifizierten Dateien zur Ersetzung in der Regelung von Benjamin Wagner.

Das Verzeichnis *projekt_wagner* enthält die von Benjamin Wagner entwickelte Software.

*Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 30. August 2011

Ort, Datum

Unterschrift