Fakultät Informationstechnik

Studiengang Softwaretechnik und Medieninformatik

Bachelorarbeit

Umsetzung einer Optischen Flusserkennung in einem mobilen Zigbee- Videostream

Kurzbeschreibung

Diese Bachelorarbeit beschäftigt sich mit der Konzeption und Umsetzung eines Sensors, zur Erfassung des Lagezustands über den optischen Fluss und Berechnung daraus resultierender Regelgrößen. Sie bildet damit die Grundlage und Wissensbasis zur Stabilisierung des Flugverhaltens, in erster Linie des Schwebefluges eines Flugapparates. Zur Auswertung der von der Kamera gelieferten Bilder kommt die OpenCV-Bibliothek zum Einsatz, in Folgendem werden alle ihre Möglichkeiten einen Bewegungsfluss zu erkennen aufgezeigt und bewertet. Der Transport der Bilder zu dem stationären Rechenserver wird mit dem ZigBee Funknetz-Standart realisiert, es wird auf seine Möglichkeiten und Grenzen hingewiesen und Vorschläge zur Optimierung seiner Funktionalität gebracht. Neben der Einführung die einem die Grundlagen der einzelnen Technologien näher bringt, beschäftigt sich der Hauptteil der Arbeit mit der Realisierung, Modifikation und Erweiterung der einzelnen Segmente des Systems. Dazu gehört unter anderem die Kommunikation zwischen dem Flugapparat und dem Rechenserver, das Framework der Flusserkennung und das bereitstellen der der Steuerdaten für den Regler. Ein Einblick in die zukünftigen Einsatz- und Erweiterungsmöglichkeiten des Systems rundet die Arbeit ab.

Inhaltsverzeichniss

1. Grundlagen zum optischen Fluss

1.1. Optischer Fluss

Als Definition für den optischen Fluss liefert die freie Onlineenzyklopädie, Wikipedia folgende Beschreibung:

Als Optischer Fluss (eng. Optical Flow) wird in der Bildverarbeitung und in der optischen Messtechnik ein Vektorfeld bezeichnet, das die Bewegungsrichtung und -Geschwindigkeit für jeden Bildpunkt einer Bildsequenz angibt. Der Optische Fluss kann als die auf die Bildebene projizierten Geschwindigkeitsvektoren von sichtbaren Objekten verstanden werden.

Optischer Fluss ist ein Phänomen, das uns stets in alltäglichen Leben begleitet. Es ist nichts Anderes als die Bewegung unserer Umwelt, relativ zu unseren Augen, die wir wahrnehmen, wenn wir beispielsweise durch die Stadt laufen, oder mit dem Auto unterwegs sind. Blicken wir aus dem Fenster eines Fahrenden Zuges, so sehen wir verschiedenste Landschaften, ob Gebäude in der Stadt, oder Bäume und Felder auf dem Land, egal ob überfüllt von Menschenmassen oder Tieren, oder scheinbar leer und verlassen, alle diese Objekte rasen in einer scheinbar, fast homogenen Masse an uns vorbei. Diese Bewegung ist der besagte optische Fluss. Zu den Funktionen des optischen Flusses, gehört nicht nur das Erfassen von Bewegungsrichtungen, sondern auch das Erkennen von Entfernungen, so bewegen sich zum Beispiel große, weit entfernte Objekte wie Wolken oder Hochhäuser, in Relation zu näheren und kleineren Objekten wie Bäumen, nur sehr langsam.

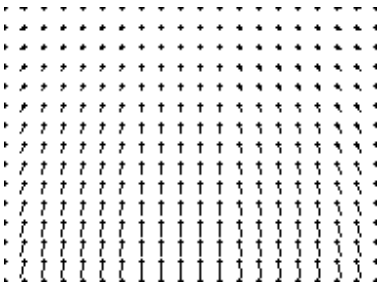


Abbildung 1-1: Vektorfeld für die Vorwertsbewegung

Es gibt eindeutige, mathematische Beziehungen Zwischen der Entfernung zum betrachteten Objekt und dem optischen Fluss. Fahren Sie die doppelte Geschwindigkeit so verdoppelt sich auch die Geschwindigkeit des Flusses. Platziert man jetzt das Objekt in einer doppelten Entfernung, so halbiert sich die sichtbare Geschwindigkeit wieder. Natürlich gibt es auch einen Zusammenhang zwischen dem optischen Fluss und dem Betrachtungswinkel relativ zur Bewegungsrichtung. Der Optische Fluss ist am größten, wenn das betrachtete Objekt sich auf Ihrer Höhe befindet, seitlich, ober- oder unterhalb von Ihnen, also Blickrichtung 90° zur Bewegungsrichtung. Vor oder hinter der 90° Marke nimmt der erkennbare optische Fluss ab.

Platziert man das betrachtete Objekt aber direkt in der Bewegungsrichtung, also vor ihnen, so wird man keinen optischen Fluss erkennen, wir merken, direkt auf der Bewegungslinie ist der optische Fluss gleich 0. Allerdings werden die Kanten eines realen Objekts nie direkt auf der Linie der Bewegung liegen und haben so mit, einen minimalen optischen Fluss, den wir als das Größerwerden des Objekts wahrnehmen können.

Folgende Abbildung veranschaulicht die Verteilung des optischen Flusses bei unterschiedlicher Bewegung. Bei einer gradlinigen Bewegung in eine bestimmte Richtung verteilt sich der optische Fluss, wie oben bereist beschrieben, direkt auf der Bewegungsrichtung ist kein Fluss zu erkennen, auf der Höhe des Betrachters sehen wir den größten optischen Fluss, mit entgegen gesetzter Flussrichtung zur Bewegung des Betrachters. Anders bei einer Drehbewegung, dort sieht man auf den ersten Blick einen konstanten optischen Fluss, auf der zweidimensionalen Abbildung erkennt man allerdings nur die Pole des optischen Flusses. Die Nullpunkte, die sich bei einer gradlinigen Bewegung auf der Bewegungsgeraden befinden, sind bei einer Drehung, ober- und unterhalb der Drehfläche, was einem mehr oder weniger logisch erscheint, wenn man sich an die Rechter-Daumen-Regel oder die Umfassungsregel aus dem Physikunterricht erinnert. Mit der Umfassungsregel wird aus der Drehbewegung wieder eine Gerade, die senkrecht zur Drehfläche steht, als Drehachse bezeichnet werden kann und wie bei gradliniger Bewegung die Nullpunkte des optischen Flusses markiert.

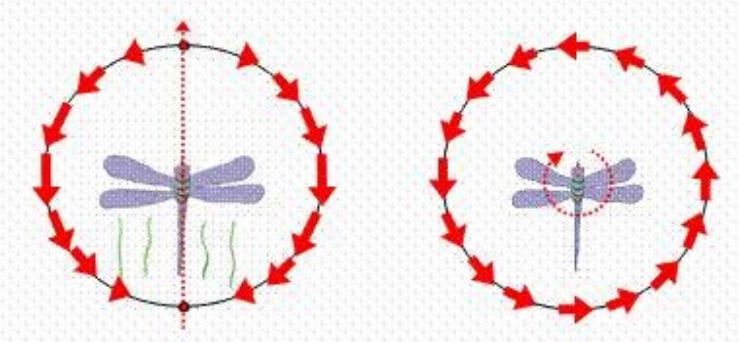


Abbildung 1-2: Charakteristische Vektorbilder für eine Vorwertsbewegung und eine Drehung

Die Eigenschaften der Drehbewegung werden eindeutiger wenn man die nächste Abbildung betrachtet. Es handelt sich dabei um jeweils eine Kugel, im Raum um den Betrachter, die in der Fläche aufgerollt wurde, Pfeile repräsentieren hier mit ihrer Länge, die Geschwindigkeit des optischen Flusses, wobei die Punkte oben und unten, immer einen und den selben Punkt meinen, nämlich den oberen und den unteren Punkt auf der Drehachse.

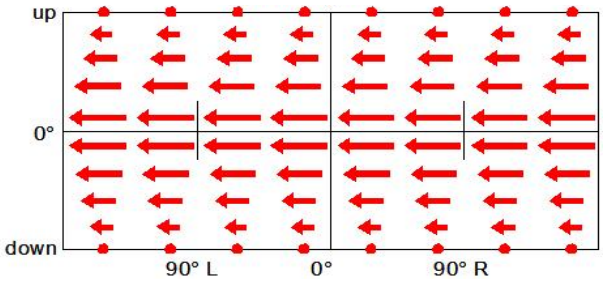


Abbildung 1-3: Aufgeklappte Vektorkugel für die Drehung nach rechts

Das nächste Bild zeigt eine solche Kugel, in einer gradlinigen Bewegung. Der Punkt in der Mitte markiert hier die Bewegungsrichtung, mit den Vektoren des optischen Flusses, die von der Mitte ausgehen, zur 90°-Marke immer größer werden, weiter hinten wieder an Länge verlieren und in der 180°-Marke in einem Punkt münden.

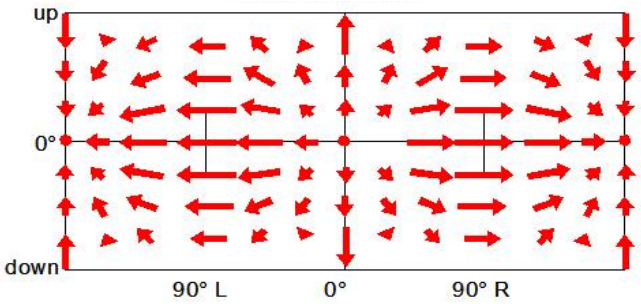


Abbildung 1-4: Aufgeklappte Vektorkugel für die Vorwertsbewegung

1.1.1. Bewegung zur Grauwertveränderung

In der Literatur zur Bildverarbeitung liest man oft den Begriff der Grauwertveränderung/ Grauwertverschiebung. Der Grauwert ist der Helligkeits- oder Intensitätswert eines Pixels, ohne die Berücksichtigung der Farben. In einem RGB-Bild könnte die Bestimmung des Grauwerts folgendermaßen aussehen:

$$\text{Grauwert} = 0{,}5 \cdot \text{Rot} + 0{,}3 \cdot \text{Grün} + 0{,}2 \cdot \text{Blau}$$

Anhand der Bewegung eben dieser Grauwerte lässt sich der optische Fluss berechnen. Was dabei aber fälschlicherweise, angenommen werden kann, ist die falsche Tatsache, dass die Grauwertveränderung mit der Bewegung in der Szene gleich zu setzen ist. Nun spielen in der Grauwertveränderung die Beleuchtung und die Objekte, eine gleich große Rolle wie die Bewegung selbst. Angenommen eine Kugel mit einer sehr glatten und gleichmäßigen Oberfläche dreht sich vor der Kamera. Die Grauwerte wären für den Betrachter regungslos. Andererseits, würde man eine solche Kugel von einer sich bewegenden Lichtquelle beleuchten lassen, könnte man anhand des bewegten Schattens, zu einer falschen Erkenntnis kommen, dass die Kugel sich bewegt.

Abbildung 1-5: Zusammenstellung der Grauwertveränderung

So kann man resultierend sagen, dass die Grauwertveränderung ein Ergebnis aus der Verbindung der Bewegung und der Beleuchtung ist, welches seine Uhrsprungskomponente nicht immer eindeutig widerspiegelt.

1.1.2. Perspektivisches Bewegungsmodell

Mit dem Perspektivischen Bewegungsmodell lassen sich Geschwindigkeiten der Szenenpunkte, relativ zur Kamera, als Beobachter, ziemlich anschaulich berechnen. Das Perspektivische Bewegungsmodell beruht auf der dreidimensionalen Starkörperbewegung mit sechs Freiheitsgraden. Zu einem sind es drei Bewegungsrichtungen der Translation mit

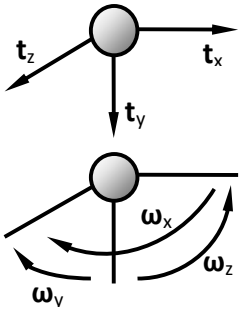
$$\vec{t} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

Formel 1-1

und drei weitere Bewegungsrichtungen der Rotation

$$\vec{\omega} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

Formel 1-2



Translatorische Bewegungsvektoren

Rotatorische Bewegungsvektoren

Abbildung 1-6: Freiheitsgrade des Perspektivischen Bewegungsmodells

Ausgegangen von den sechs Freiheitsgraden und den Koordinaten des Szenepunktes $\vec{P}$, mit

$$\vec{P} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Formel 1-3

läst sich für die Geschwindigkeit eines Punktes der Szene folgende Gleichung aufstellen:

$$\dot{\vec{P}} = -\vec{t} - \vec{\omega} \times \vec{P} = \begin{bmatrix} -t_x - \omega_y \cdot Z + \omega_z \cdot Y \\ -t_y - \omega_z \cdot X + \omega_x \cdot Z \\ -t_z - \omega_x \cdot Y + \omega_y \cdot X \end{bmatrix}$$

Formel 1-4 [3]

Die vielen Minuszeichen kommen daher, da es von der Bewegung der Kamera ausgegangen wird, Bewegungen der Objekte in der Szene wirken dem genau entgegen.

Um die Geschwindigkeit eines Punktes zu bestimmen, sind seine Koordinaten nach der Zeit abzuleiten. Dies kann auf einzelne Komponenten der Bewegungsrichtung explizit angewandt werden. Hier die x-Koordinate, mit

$$x = \frac{f}{Z} \cdot X$$

Formel 1-5

wo f die Brennweite der Kamera und Z den Abstand des Punktes zur Linse darstellen. Abgeleitet nach der Zeit sieht die Gleichung so aus:

$$\frac{dx}{dt} = f \cdot \left(\frac{\partial x}{\partial Z} \cdot \frac{\partial Z}{\partial t} + \frac{\partial x}{\partial X} \cdot \frac{\partial X}{\partial t} \right)$$

Formel 1-6

oder

$$\dot{x} = f \cdot \left(\frac{X}{Z^2} \cdot \dot{Z} + \frac{1}{Z} \cdot \dot{X} \right)$$

Formel 1-7

und für y entsprechend

$$\dot{y} = f \cdot \left(\frac{Y}{Z^2} \cdot \dot{Z} + \frac{1}{Z} \cdot \dot{Y} \right)$$

Formel 1-8

Durch Einsetzen der Formel 1-7 und Formel 1-8 in die Formel 1-4 [3], so erhält man die Formel für das Geschwindigkeitsfeld $\dot{\vec{p}}$:

$$\dot{\vec{p}} = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \frac{1}{Z} \cdot \begin{bmatrix} t_z \cdot x - f \cdot t_x \\ t_z \cdot y - f \cdot t_y \end{bmatrix} + \omega_x \cdot \begin{bmatrix} \frac{1}{f} \cdot x \cdot y \\ \frac{1}{f} \cdot y^2 + f \end{bmatrix} - \omega_y \cdot \begin{bmatrix} \frac{1}{f} \cdot x^2 + f \\ \frac{1}{f} \cdot x \cdot y \end{bmatrix} - \omega_z \cdot \begin{bmatrix} -y \\ x \end{bmatrix}$$

Formel 1-9

Diese Darstellung lässt sich durch Normierung weiter vereinfachen. Man setze die Brennweite f=1 und substituiere mit

$$A = \begin{bmatrix} -1 & 0 & x \\ 0 & -1 & y \end{bmatrix}$$

und

$$B = \begin{bmatrix} x \cdot y & -x^2 - 1 & y \\ y^2 + 1 & -x \cdot y & -x \end{bmatrix}$$

so erhält man für das Perspektivische Bewegungsmodell, folgende Schreibweise:

$$\dot{p} = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = A \cdot \frac{t}{Z} + B \cdot \omega = \begin{bmatrix} \frac{1}{Z} \cdot A & B \end{bmatrix} \cdot \begin{bmatrix} t \\ \omega \end{bmatrix}$$

Formel 1-10 [3]

Man merkt die Abhängigkeit der Geschwindigkeitsverteilung, von der Entfernung Z des Szenenpunktes bis zur linse der Kamera. Es bestätigt sich das Phänomen, dass weit entfernte Objekte sich langsamer zu bewegen scheinen, als näher gelegene. Dieser Zusammenhang Z(p) wird als Tiefenfunktion bezeichnet.

1.2. Probleme der Flusserkennung

1.2.1. Das Blendenproblem (Aperturproblem)

Die Analyse des optischen Flusses beruht also auf räumlichen und zeitlichen Grauwertänderungen. Diese Grauwerte werden innerhalb bestimmter Radien, in zwei aufeinander folgenden Bildern wieder erkannt und einander zugeordnet. Nicht immer sind markante Merkmale dabei die sich sofort zuordnen lassen, doch auch wenn es der Fall ist, kann das Ergebnis mehrdeutig werden. Ein Solches Problem bezeichnet man als das Blendenproblem. Es taucht dann auf, wenn zwar ein Abschnitt des Suchfensters/Maske, Grauwertmäßig sich eindeutig vom Rest der Pixel unterscheidet, doch keine fassbaren Merkmale bittet, an hand derer die Bewegung eindeutig erkannt werden kann, weil das Objekt, zum Beispiel viel größer als das Suchfenster ist.

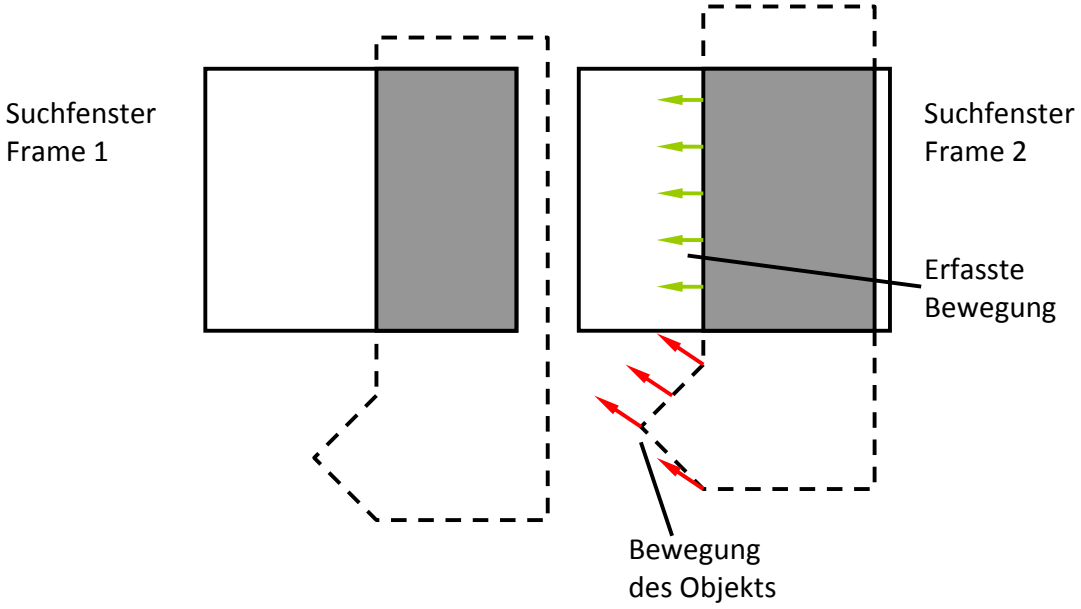


Abbildung 1-7: Veranschaulichung des Blendenproblems

Wir können lediglich die senkrecht zur Kante liegende Komponente des Verschiebungsvektors bestimmen, während die parallel zur Kante liegende unbekannt bleibt. Diese Mehrdeutigkeit (das Blendenproblem) kann z.B. aufgelöst werden, wenn die Operatormaske die Ecke eines Objektes einschließt.

1.2.2. Korrespondenzproblem

Das Korrespondenzproblem beschreibt alle arten des Informationsverlustes einer Bewegung, das zuvor erwähnte Blendenproblem ist ein Spezialfall des Korrespondenzproblems. Das Korrespondenzproblem liegt dann vor, wenn wir keine eindeutig mit einander korrelierender

Punkte, in zwei aufeinander folgenden Bildern einer Sequenz, bestimmen können. In folgendem sind einige Beispiele des Korrespondenzproblems aufgeführt.

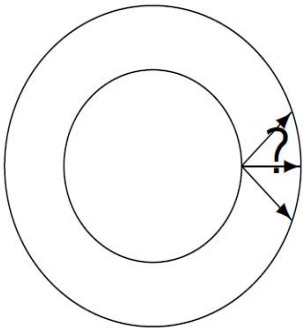


Abbildung 1-8: Korrespondenzproblem 1

Es fehlen eindeutige Merkmale, deren Bewegung man Verfolgen könnte.

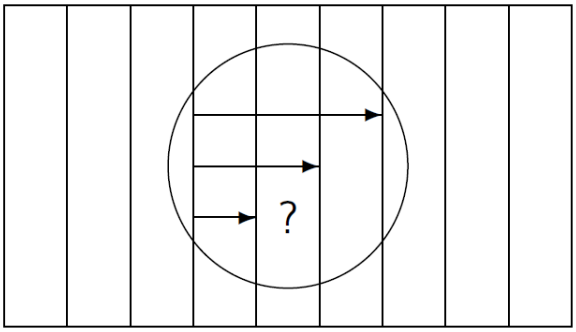


Abbildung 1-9: Korrespondenzproblem 2

Bei Periodischen Texturen, wie Netz oder Gitter, kann die Bewegung nie eindeutig bestimmt werden, solange man nur ein abschnitt des Objekts betrachtet und die Bewegung ein Vielfaches der Maschenweite ist. Man erkennt die Bewegung erst dann eindeutig, wenn man den Rand des Gitters im Suchfenster hat.

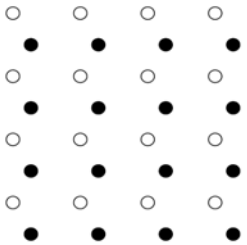


Abbildung 1-10: Korrespondenzproblem 3

Bei Bildern mit vielen Objekten, die eine sehr ähnliche Form haben, kann das korrespondierende Teilchen im nächsten Bild nicht ermittelt werden.

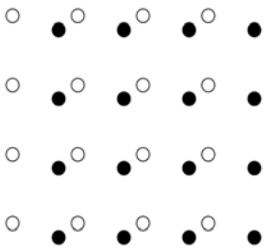


Abbildung 1-11: Lösung durch Verkürzen des Zeitinterwals

Eine Lösung für Solche Probleme wäre das Verkürzen des Zeitinterwals, man würde einen solchen Interwal einstellen, bei dem man sich sicher sein kann, dass in der Zeit ein Teilchen nur einen kleinen Teil des mittleren Teilchenabstandes überbrücken kann.

1.3. Erkennen des optischen Flusses

Die Erfassung des Optischen Flusses ist eine Disziplin der künstlichen Intelligenz. Eine Videokamera liefert ca. 30 Bilder in der Sekunde, der Unterschied zwischen den einzelnen Frames stellt hierbei die wichtigste Informationsquelle dar. Bewegt man die Kamera relativ zur Umgebung, so erkennt man am Bildschirm eine dementsprechende Bewegung, den zuvor erwähnten optischen Fluss. Den optischen Fluss stellt man oft als einen Vektorfeld dar, dabei beschreibt Länge und die Richtung jedes Vektors $\vec{V}(x_i, y_i)$ die derzeitige Bewegung, des Pixels $P(x_i, y_i)$. In der Abbildung 1-12: Vektorfelder des Zooms, der Drehung und ihrer Kombination sieht man einige mögliche Vektorfelder und die dazugehörigen Transformationen, Zoom, Drehung und die Kombination as den beiden.

Abbildung 1-12: Vektorfelder des Zooms, der Drehung und ihrer Kombination

Eine Bildrate von 30 Bilder/Sekunde entspricht einem Zeitintervall von $\frac{1}{30}$ Sekunde, zwischen zwei aufeinander folgenden Bildern. Es wird stets versucht die Intervalle so klein wie Möglich zu halten, damit die Bewegung der Bildinhalte nur wenige Pixel ausmacht[1]. Andererseits muss die Zeit, die dem Algorithmus bleibt, um den Fluss der Grauwerte zu berechnen, groß genug sein, um beim nächsten Frame das Ergebnis vorzulegen und mit der nächsten Berechnung fortzufahren.

1.4. Korrelationsbasierte Verfahren

1.4.1. Block Matching

Die korrelationsbasierte Verfahren zur Erfassung des optischen Flusses, versuchen in zwei auf einander folgenden Bildern Bereiche zu finden, die einander eindeutig zuordenbar sind. Dabei vergleichen sie Muster, mittels eines Ähnlichkeitsmaßes, aus hellen und dunkeln Pixeln, in

[1] Die meisten Algorithmen zur Bestimmung des optischen Flusses, basieren auf der Annahme, dass die einzelnen Bewegungen im Bild nie eine große Distanz zurücklegen, somit sucht man ein bestimmtes Muster nicht in der gesamten Bildebene, sondern nur in der unmittelbaren Umgebung des gesuchten Grauwerts. Das heißt, umso kleiner der Zeitliche Abstand zweier auf einander folgender Bilder, desto kleiner kann der Suchbereich eingestellt werden, ohne den Verlust der Wahrscheinlichkeit den Vektor richtig zu bestimmen.

lokalen Bereichen der Bilder (Blöcke). Die zuvor definierten Blöcke müssen so gewählt werden, dass der Algorithmus bis zum nächsten Bildpaar das Ergebnis vorlegen kann, das heißt möglichst kleine Abschnitte mit wenigen Pixeln. Andererseits dürfen die Blöcke auch nicht zu klein gewählt werden, da sonst die Gefahr besteht, dass das Suchmuster sich zum Zeitpunkt in dem das Zweite Bild gemacht wird, schon außerhalb des Blocks befindet und nicht gefunden werden kann.

Da die verfolgten Objekte, an den der optische Fluss bestimmt werden soll, meist größer sind als die Blöcke, tritt beim Suchen der Ähnlichkeiten meist das Aperturproblem auf, da die Objekte nur an ihren Rändern eindeutige Bewegungen erkennen lassen. Dieses Problem wird dadurch gelöst, dass man einfach annimmt, dass die Blöcke die keine eindeutig zu erkennende Muster enthalten, zu einem Objekt mit daneben liegenden Blöcken zusammen gefasst werden können. Weiterhin wird angenommen, dass alle Blöcke die zu einem Objekt gehören, gleiche Geschwindigkeiten besitzen, also in einem homogenen Vektorfeld liegen, in dem alle Vektoren die selbe Richtung und Länge haben.

Dazu existieren physikalische Verfahren, zur besseren Vorstellung der Problematik [4], dabei stellt man sich zwischen den benachbarten Blöcken gespannte Federn vor, deren Federkonstante jeweils davon abhängig ist, wie ähnlich die Blöcke sich sind. Zwei ähnlich aussehende Blöcke haben demnach eine Verbindungsfeder mit einer großen Federkonstante, Blöcke die sich stark unterscheiden, oder zwischen denen sich eine Kante befindet, eine mit einer kleinen Federkonstante.

Das Ähnlichkeitsmaß ist in aller Regel, eine Korrelation zwischen zwei Folgebildern. Es gibt verschiedene Methoden um die Ähnlichkeit zweier Blöcke zu bestimmen.

SAD: Summe der Differenzen der Absolutbeträge („Sum of Absolut Differences“)

$$SAD = \sum_{r \in N} \left| I'(p' + r) - I(p + r) \right|$$

SSD: Summe der quadratischen Differenzen der Absolutbeträge („Sum of Squared Differences“)

$$SSD = \sum_{r \in N} \left(I'(p' + r) - I(p + r) \right)^2$$

NCC: Normalisierte Kreuzkorrelation („Normalized Cross Correlation“)

$$NCC = \frac{\sum_{r \in N} I'(p' + r) \cdot I(p + r)}{\sqrt{\sum_{r \in N} I'^2(p' + r)} \cdot \sqrt{\sum_{r \in N} I'^2(p + r)}}$$

Dies sind drei am häufigsten eingesetzte Verfahren, um die Ähnlichkeit Zweier Blöcke zu bestimmen. Die Verfahren SAD und SSD zeigen die Unterschiede der Blöcke auf, also umso größer das Ergebnis, desto unterschiedlicher sehen sich die Blöcke, um die beste Korrelation der Blöcke zu finden sucht man daher nach dem minimalen Ergebnis. Im Gegensatz dazu sucht man mit dem NCC die Ähnlichkeit, so stellt der Maximalwert dieses Verfahrens die beste Übereinstimmung der Blöcke dar.

Methoden zur Ähnlichkeitsuntersuchung gibt es ebenfalls zu genüge, sie alle versuchen auf dem kürzestem Wege, die Größte Übereinstimmung zweier Blöcke heraus zu finden. Das einfachste Vorgehen dabei wäre die Blöcke so in einander zu verschieben, dass am ende des Vorgangs jeder Pixel des ersten Blockes, jedem Pixel aus dem Zweiten mindestens einmal gegenübergestellt worden wäre und zu jeder dieser Konstellation einen Ähnlichkeitsfaktor zu bestimmen. Mit dieser Methode wäre es sichergestellt, dass das Ergebnis mit der größten Ähnlichkeit, tatsächlich die optimale Lösung darstellt. Betrachtet man aber die Tatsache, dass man dabei mindestens N mal einen Ähnlichkeitsfaktor bestimmen muss, dessen Komplexität, je nach dem wie Weit die Blöcke in einander verschoben sind, variiert. Für einen solchen Vergleich zweier Blöcke, dessen Kante 8 Pixel groß ist, wären 224 Schritte nötig, wobei im Schnitt, bei jedem Schritt 16 Faktoren bestimmt werden müssen. Abbildung 1-13 beschreibt ein solches Verfahren, wobei man die Fläche unter der Kurve als Komplexität des Verfahrens betrachten kann.

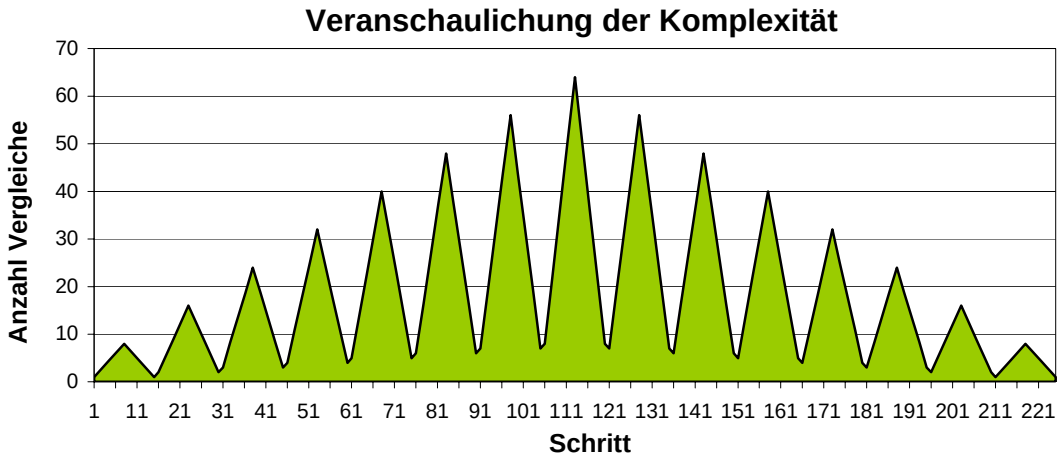


Abbildung 1-13: Komplexität der einfachen Ähnlichkeitsbestimmung

Zur Lösung dieses Problems gibt es allerdings einige Vorgehensweisen, die nicht so Zeit- und Rechenintensiv sind, zum Beispiel der Three-Step Search (TSS) Algorithmus oder der Two Dimensional Logarithmic Search (TDL) Algorithmus[5], die Ich in Folgendem, kurz erläutere.

Three-Step Search (TSS)

Der Three-Step Search Algorithmus zeichnet sich dadurch aus, dass es sehr simpel und schnell ist, nur wenig Arbeitsspeicher benötigt, dabei aber nahezu optimale Ergebnisse liefert. Der Name Three-Step Search spricht in dem Fall für sich, die Suche des optimalen Blockausschnitts, mit dem größten Ähnlichkeitsfaktor, passiert hier in drei Schritten.

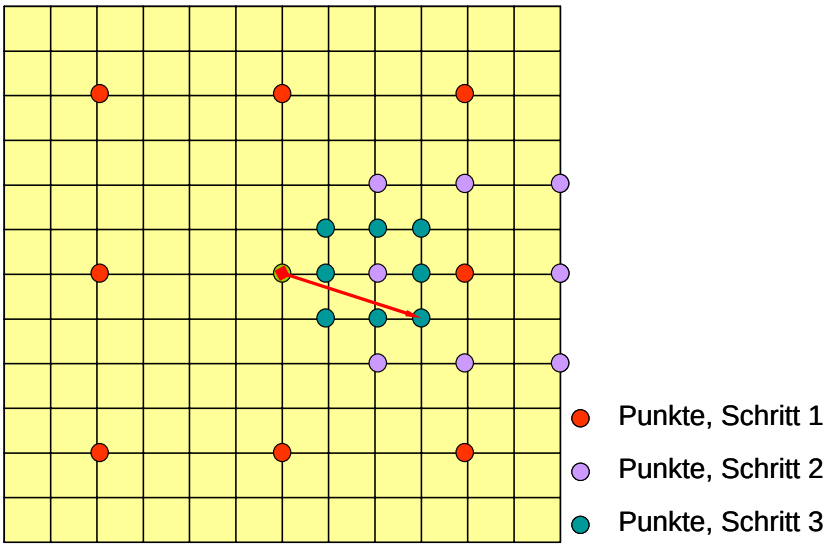


Abbildung 1-14: Three-Step Search

Zu Beginn werden die zu untersuchenden Frames in neun gleichgroße bereiche Geteilt. Die Mitten dieser Bereiche sind in der Abbildung 1-14 mit roten Punkten markiert. Im ersten Sritt wird Frame 1 mit seinem Zentrum jeweils zu jedem dieser Punkte angelehnt und aus den dabei entstandenen Konstellationen, jeweils die Ähnlichkeitsfaktoren bestimmt. So kriegt jeder, der Mittelpunkte einen eigenen Ähnlichkeitsfaktor zugewiesen. Es wird bestimmt welche dieser Konstellationen die geringste Abweichung ergab, der dazugehörige Mittelpunkt des Bereiches wird nun als Ausgangspunkt für den Schritt zwei. Um den, neu bestimmten Zentrum der Suche werden wieder acht Punkte verteilt, die Entfernung der Punkte von einander wird im Vergleich zum Schritt 1 halbiert. Die Vorgehensweise aus dem ersten Schritt wird, diesmal mit den neuen Punkten wiederholt. Man erhält wieder ein neues Zentrum und legt acht Punkte um ihn, wieder mit der Halben Entfernung zwischen den Punkten im Vergleich zum vorigen Schritt. Im Schritt drei wird genauso vorgegangen, diesmal bezeichnet der Punkt mit dem größten Ähnlichkeitsfaktor die Position der Spitze, des Bewegungsvektors, ausgehend vom Zentrum des Blockes.

Two Dimensional Logarithmic Search (TDL)

Diese Suche ist der zuvor erklärten Three-Step Suche sehr ähnlich, schließt aber nicht aus auch mehr Schritte als nur drei zu machen, was sie für größere Bildausschnitte geeigneter macht.

Dazu kann man bei der TDL eine größere Genauigkeit erwarten, da sie soweit rechnet bis keine Verkleinerung des Suchbereichs mehr möglich ist.

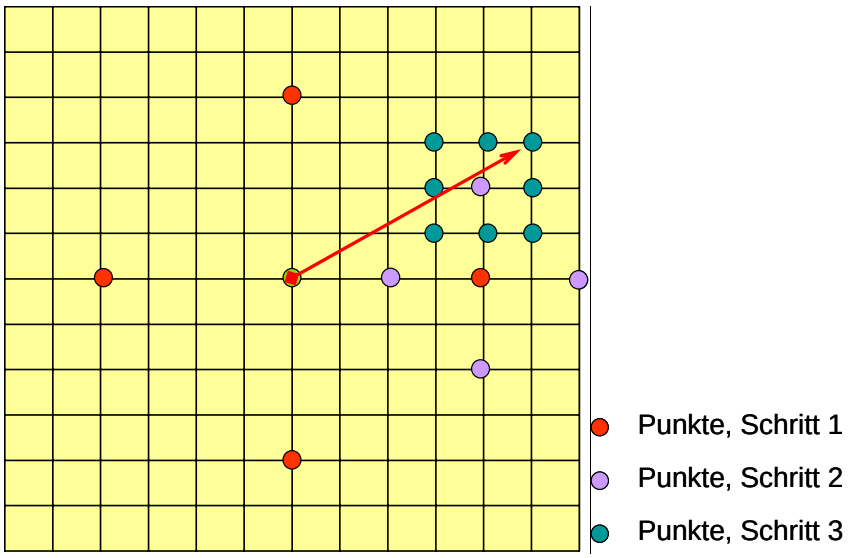


Abbildung 1-15: Two Dimensional Logarithmic Search

Im ersten Schritt bildet man ein Kreutz aus fünf Punkten, in einem Bildabschnitt, genau so wie bei der Three-Step Suche werden diese Punkte jeweils mit dem Mittelpunkt des Anderen Frames Abgefahren und zu jedem dieser Punkte Ähnlichkeitsfaktoren bestimmt. Der Punkt mit der größten Ähnlichkeit wird wieder das neue Zentrum der Suche. Die Abstände der Punkte werden Halbiert. Dieses Vorgehen wird so lange wiederholt bis der Abstand zwischen den Punkten einen Pixel beträgt, dann werden alle neun, den Mittelpunkt umschließende Pixel abgefahren und der, mit der größten Ähnlichkeit wird zur Spitze des Vektors der Bewegung.

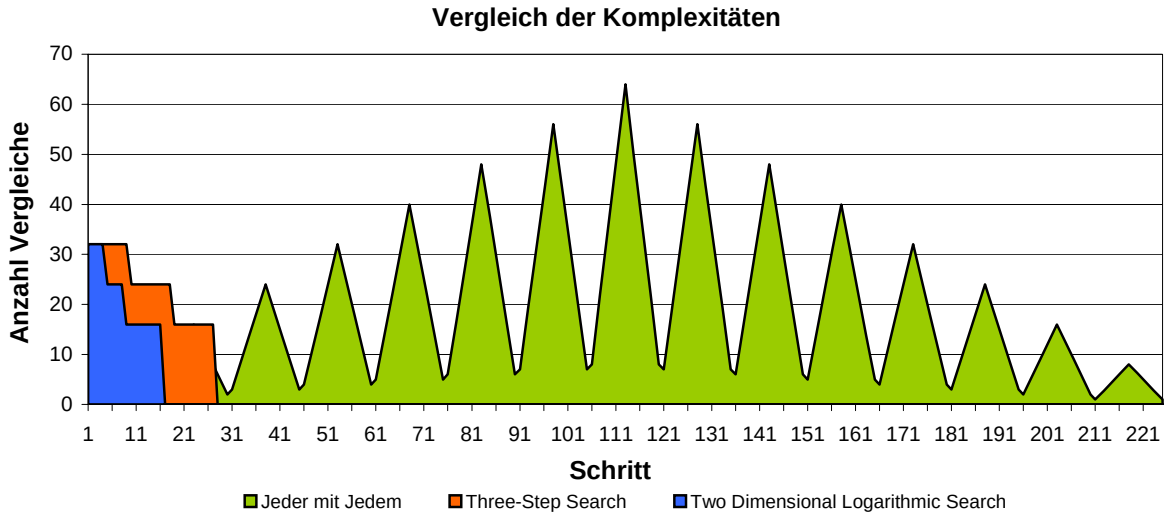


Abbildung 1-16: Komplexität der Methoden im Vergleich

In direktem Vergleich merkt man den unterschied der Suchalgorithmen, wehrend die Methode „jeder mit jedem“ je nach Bildüberlappung, mehr oder weniger Berechnungen anstellen muss, sind die beiden Fortschrittlicheren Algorithmen gleich von Anfang an mit vielen Berechnungen belastet, da diese aber stichprobenartig durchgeführt werden, sind sie dementsprechend auch in wenigen Schritten beim gesuchten Pixel, oder in seiner unmittelbaren Nähe.

1.4.2. Vor- und Nachteile der Korrelationsverfahren

Zu den größten Vorteilen der Korrelationsbasierten Verfahren zur Erfassung des optischen Flusses gehört ihre relativ gute Verständlichkeit, der unwesendliche mathematische Hintergrund und geringe Komplexität, was die Realisierung einfacher und weniger Fehleranfällig macht. Ist aber gerade diese Einfachheit von Block Matching, das was für diese Methode zum Nachteil wird. Denn die Suche nach dem ähnlichen Block, im zweiten Bild ist sehr rechenintensiv und benötigt eine Menge Zeit und Speicher. Was im Gegenzug einen relativ exaktes Ergebnis erwarten läst, doch auch das wäre eine Täuschung, denn die Blockweise Suche nach Ähnlichkeiten ist sehr unzuverlässig und anfällig für die bekannten Probleme der Bewegungsschätzung, Korrelations- und das Blendenproblem lassen Fehleinschätzungen des Flusses sehr oft entstehen.

1.5. Gradientenverfahren

Die Annahme, die bei allen Gradientenverfahren gültig ist, ist die dass ein Objekt im Bild immer eine gleich bleibende Intensität besitzt. Ein Punkt zur Zeit t, an der Stelle (x,y), befindet sich zur Zeit t + dt an der Stelle (x+dx,y+dy). Würde sich die Intensität zeitlich verändern, würde die Formel 1-11 gelten.

$$I(x+dx, y+dy, t+dt) = I(x, y, t) + \frac{\partial I}{\partial x} \cdot dx + \frac{\partial I}{\partial y} \cdot dy + \frac{\partial I}{\partial t} \cdot dt$$

Formel 1-11

Da die Annahme, der Konstanten Intensität gilt, bedeutet dies, dass

$$\frac{\partial I}{\partial x} \cdot dx + \frac{\partial I}{\partial y} \cdot dy + \frac{\partial I}{\partial t} \cdot dt = 0$$

Formel 1-12

Mit der Substitution der beiden Geschwindigkeiten u und v, mit

$$u = \frac{dx}{dt}$$

und

$$v = \frac{dy}{dt}$$

Erhält man die Formel, die Horn & Schunk, 1981 der Welt präsentierten.

$$\frac{\partial I}{\partial x} \cdot u + \frac{\partial I}{\partial y} \cdot v = -\frac{\partial I}{\partial t}$$

Formel 1-13: Differentieller Ansatz zur Bestimmung des optischen Flusses (Helligkeitskonstanzgleichung) [3]

An dieser Stelle setzen die beiden Verfahren von Lucas & Kanade und Horn & Schunck an. Es gilt eine Gleichung zu lösen die Zwei Unbekannte besitzt, also brauch es noch eine zusätzliche Bedingung um eine eindeutige Lösung zu kriegen.

1.5.1. Horn & Schunck Algorithmus

Die zusätzliche Bedingung, die zur Lösung beitragen sollte war bei Horn & Schunck ist die so genannte Glattheitsbedingung. Die besagt, dass das Lokale Vektorfeld eines kleinen Bildabschnitts fast konstant ist, d.h. Bewegungsvektoren unterscheiden sich vom Pixel zu Pixel nur geringfügig, in ihrer Richtung und Länge (glattes Vektorfeld). Zu Beginn nimmt man an, dass das Vektorfeld konstant ist und es einige Abweichungen vom allgemeinen Richtung und Länge des Feldes gibt, die wie folgt beschrieben werden können

$$e_s(u,v) = \int\int \|\Delta u\|^2 + \|\Delta v\|^2 dxdy = \int\int \left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 dxdy$$

Formel 1-14: Abweichung von der H&S-Bedingung [3]

Die Helligkeitskonstanzgleichung, Formel 1-13 kann man hierfür auch aufschreiben als:

$$e_d(u,v) = \iint \left(\frac{\partial I}{\partial x} \cdot u + \frac{\partial I}{\partial y} \cdot v + \frac{\partial I}{\partial t}\right)^2 dxdy$$

Formel 1-15

Aus den Gleichungen in Formel 1-14 und Formel 1-15 ergibt sich das zu minimierende Gesamtfunktional der Methode nach Horn & Schunck:

$$e_{H\&S}(u,v) = e_d(u,v) + \lambda \cdot e_s(u,v)$$

Formel 1-16: Gesamtfunktional der Methode nach Horn & Schunck [3]

Die beispielhafte Lösung der Funktion findet man in der Präsentation, zur Veranstaltung „Bildverarbeitung und Computer Vision“, der Universität Münster, im Kapitel 13 – Optische Flüsse (Literatur [7]).

1.5.2. Lucas & Kanade Algorithmus

Die Bedingung, die Lucas & Kanade in ihrer Berechnung des Optischen Flusses zur Annahme machen, ist die weiterführende Glattheitsbedingung des Horn & Schunck Algorithmuses. Diese besagt, dass der optische Fluss innerhalb eines kleinen Bildbereichs konstant ist. So erhalten wir eine Nachbarschaftsmatrix, mit der Kantenlänge m, mit Pixels für die alle, die Helligkeitskonstanzgleichung gilt, was zu einem System von $N=m^2$ linearen Gleichungen führt. Für unsere bisherige Gleichung aus Formel 1-13 heißt es, wenn die Bewegungskomponenten u und v zu einem Vektor zusammengefasst werden, erhält man die Formel

$$\left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right] \cdot \begin{bmatrix} u \\ v \end{bmatrix} = -\frac{\partial I}{\partial t}$$

Formel 1-17

Wendet man darauf die Methode „Kleinste- Quadrat- Minimierung“[1] an, so erhält man eine Energiegleichung die es zu minimieren gilt.

$$E = \sum_{k \in N} \left\| \left[\frac{\partial I_k}{\partial x}, \frac{\partial I_k}{\partial y}\right] \cdot \begin{bmatrix} u \\ v \end{bmatrix} + \frac{\partial I_k}{\partial t} \right\|^2 \stackrel{!}{=} \min$$

Formel 1-18

Zur Lösung Führt dann folgende Formel:

[1] Die Methode der kleinsten Quadrate (englisch: method of least squares) ist das mathematische Standardverfahren zur Ausgleichungsrechnung. Dabei wird zu einer Datenpunktwolke eine Kurve gesucht, die möglichst nahe an den Datenpunkten verläuft. Die Daten können physikalische Messwerte, wirtschaftliche Größen oder Ähnliches repräsentieren, während die Kurve aus einer parameterabhängigen problemangepassten Familie von Funktionen stammt. Die Methode der kleinsten Quadrate besteht dann darin, die Kurvenparameter so zu bestimmen, dass die Summe der quadratischen Abweichungen der Kurve von den beobachteten Punkten minimiert wird. [6]

$$\begin{bmatrix} \sum_{k\in N}\left(\frac{\partial I_k}{\partial x}\right)^2 & \sum_{k\in N}\frac{\partial I_k}{\partial x}\cdot\frac{\partial I_k}{\partial y} \\ \sum_{k\in N}\frac{\partial I_k}{\partial x}\cdot\frac{\partial I_k}{\partial y} & \sum_{k\in N}\left(\frac{\partial I_k}{\partial y}\right)^2 \end{bmatrix} \cdot \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_{k\in N}\frac{\partial I_k}{\partial x}\cdot\frac{\partial I_k}{\partial t} \\ \sum_{k\in N}\frac{\partial I_k}{\partial y}\cdot\frac{\partial I_k}{\partial t} \end{bmatrix}$$

Formel 1-19: Lösungsgleichung zum Lucas & Kanade Algorithmus [3]

1.6. Vor- und Nachteile der Gradientenverfahren

Gradientenverfahren zur Schätzung des optischen Flusses werden meistens mit ihrer, relativ hohen Genauigkeit in Verbindung gebracht. Tatsächlich liefern sowohl das Verfahren von Horn & Schunck als auch der Lucas & Kanade Algorithmus, sehr genaue Ergebnisse, was für sie spricht. Was aber schon die mathematische Komplexität vermuten läst, sind diese Methoden zu rechenintensiv, was eine Echtzeittauglichkeit ausschließen läst. Zudem kommt noch die Tatsache, dass Gradientenverfahren nur eine lokal begrenzte Bewegungsschätzung erlauben, die meistens nur wenige Pixel groß ist.

Diese Probleme lasen sich aber mit relativ wenig Aufwand in den Griff kriegen. Zur Minimierung des zeitlichen und rechnerischen Aufwandes, kommen so genannte Featuretracker zum Einsatz. Dies sind Verfahren, die bereits vor der eigentlichen Bewegungsschätzung Pixelmuster in den Bildern suchen und zur weiteren Verfolgung nominieren. Die Grauwertbewegung wird danach nur in der unmittelbaren Nähe der Features geschätzt, was ein erhebliches Ersparnis an Rechenleistung herbeiruft. Es muss natürlich bedacht werden, dass dabei kein zusammenhängendes Vektorfeld entsteht, was Bewegungen jedes Pixels erfasst. Was aber unseren Anforderungen an den optischen Fluss vollkommen genügt.

1.6.1. Gaußpyramiden

Abbildung 1-17: Gaußpyramide

Zur Eliminierung des Problems, dass sich die Gradientenverfahren nicht über viele Pixel hinweg Objekte einander zuordnen können und so mit keinen optischen Fluss erkennen, kommen die Gaußpyramiden zum Einsatz.

Die Funktion der Gaußpyramiden ist sehr einfach aber effizient, durch die Verringerung der Auflösung wird das gesamte Bild mit weniger Pixel dargestellt, so mit haben die Gradientenverfahren keine Probleme auch Bewegungen zu erfassen, die über weite Strecken gehen.

1.6.2. Feature Selektion

Zur Verwendung von Features bei der Suche nach optischen Flüssen, griff man wegen mehrerer Umstände, der ausschlaggebende war das Aperturproblem und das allgemeine Korrelationsproblem. Zur Erinnerung, das Aperturproblem besagt dass bei einer geraden Kannte des Objekts, nur der teil der Bewegung erfasst werden kann, der senkrecht zur Kannte verläuft. Die Idee der Forscher war, nur Ecken oder Regionen mit vielen, eindeutigen Texturen zu verfolgen, solche Regionen des Bildes nennt man “Trackabe Features“.

Das Verfahren der Featuresuche, welches mit der Funktion *cvGoodFeaturesToTrack(),* in der OpenCV-Bibliothek implementiert ist, wurde von Carlo Tomasi und Takeo Kanade, im April 1991, in dem Bericht „Detection and Tracking of Point Features“ [8] veröfentlicht. Dort werden mathematische Herleitung und die Zusammenhänge der Featuresuche erklärt. Im Grunde ist es eine Untersuchung, der in der Formel 1-19: Lösungsgleichung zum Lucas & Kanade Algorithmus [3] vorgestellten Gleichung zur Erfassung des optischen Flusses. Angewandt auf ein Bildausschnitt, liefert die 2x2 Matrix Indikationswerte, die im Verhältnis zu einander, Aussagen über den Inhalt des Fensters treffen lassen. So kann man eine Kante von einer Ecke oder einen Bereich ganz ohne zuordenbare Features unterscheiden.

In Verbindung mit den Gaußpyramiden und den Featuretrackern zeigen die Gradientenverfahren sehr gute Ergebnisse, und erlauben ohne hochgesteckte Anforderungen an die Rechenleistung zu haben, auch Echtzeitproblematiken anzugehen. In der Literatur findet man Zahlreiche Aussagen darüber, wie radikal sich die Anwendung von Gaußpyramiden auf die Fähigkeiten und Anforderungen der Gradientenverfahren auswirkt. Und obwohl die meisten, die Meinung verträten, dass der Algorithmus von Horn & Schunck in Kombination mit den Pyramiden bessere Ergebnisse liefert, wird in der OpenCV-Bibliothek nur der Ansatz von Lucas & Kanade um die Gaußpyramiden erweitert, wohl aus dem Grunde, dass der Lucas & Kanade Algorithmus eine Vereinfachung des von Horn & Schunck ist und schon in der unmodifizierten Version Weniger Zeit- und Speicheraufwändig ist.

1.7. Kalman-Filter

Um möglichst genaue Messergebnisse vorlegen zu können, bedarf es eines Algorithmus der ohne viel Rechenaufwand, das Rauschen eines Signals heraus filtert und mit Statistischen Methoden an den tatsächlichen Wert möglichst nahe herankommt. Für eine solche Aufgabe

bittet sich der, heute wohl am weitesten verbreitete Algorithmus zur Zustandsschätzung linearer und nichtlinearer Systeme an, der Kalman-Filter.

Der Kalman-Filter ist ein nach seinem Entdecker Rudolf E. Kálmán benannter Satz von mathematischen Gleichungen. Obgleich die Benennung des Filters nach Rudolf E. Kálmán erfolgte, wurden bereits zuvor nahezu identische Verfahren durch Thorvald N. Thiele und Peter Swerling veröffentlicht. Auch existierten zur selben Zeit bereits allgemeinere, nichtlineare Filter von Ruslan L. Stratonovich, die das Kalman-Filter und weitere lineare Filter als Spezialfälle enthalten. Ebenso erwähnenswert sind Vorarbeiten und gemeinsame Publikationen von Kálmán mit Richard S. Bucy, insbesondere für den Fall zeitkontinuierlicher dynamischer Systeme. Daher wird häufig die Bezeichnung Kalman-Bucy-Filter und gelegentlich auch Stratonovich-Kalman-Bucy-Filter in der Fachliteratur benutzt.

Der erste erfolgreiche Einsatz des Filters erfolgte in Echtzeitnavigations- und Leitsystemen, die im Rahmen des Apollo-Programms der NASA unter Federführung von Stanley F. Schmidt entwickelt wurden und in der Navigation und Steuerung der Raumkapsel verbaut wurden.

Mithilfe dieses Filters sind bei Vorliegen lediglich fehlerbehafteter Beobachtungen Rückschlüsse auf den exakten Zustand von vielen der Technologien, Wissenschaften und des Managements zugeordneten Systemen möglich. Die Kernaussage ist: „Ist-Wert des Systems muss gleich Soll-Wert zu einer bestimmten Zeit sein.“ Vereinfacht dient das Kalman-Filter zum Entfernen der von den Messgeräten verursachten Störungen. Dabei müssen sowohl die mathematische Struktur des zugrundeliegenden dynamischen Systems als auch die der Messverfälschungen bekannt sein.

Eine Besonderheit des 1960 von Kálmán vorgestellten Filters bildet seine spezielle mathematische Struktur, die den Einsatz in Echtzeitsystemen verschiedenster technischer Bereiche ermöglicht. Dazu zählen u.a. die Auswertung von Radarsignalen zur Positionsverfolgung von sich bewegenden Objekten (Tracking) aber auch der Einsatz in elektronischen Regelkreisen allgegenwärtiger Kommunikationssysteme wie etwa Radio und Computer.

Mittlerweile existiert eine große Bandbreite von Kalman-Filtern für die unterschiedlichsten Anwendungsgebiete.

Im Gegensatz zu den klassischen FIR- und IIR-Filtern der Signal- und Zeitreihenanalyse basiert das Kalman-Filter auf einer Zustandsraummodellierung, bei der explizit zwischen der Dynamik des Systemzustands und dem Prozess seiner Messung unterschieden wird.

Bevor man einen Kalman-Filter erfolgreich einsetzen kann, sind vom System einige notwendige Bedienungen zu erfühlen, diese wären:

- das System ist linear,
- das Störsignal ist ein weises Rauschen und

- das Störsignal ist gaußscher Natur.

die Erste bedeutet, dass jeder Zustand des Systems zum Zeitpunkt k, ähnlich wie eine Markow-Kette erster Ordnung, nur vom Zustand k-1 abhängt, also aus der Multiplikation einiger Matrizen mit dem Zustand k-1 ermittelt werden kann.

$$X_k = F_{k-1}X_{k-1} + B_{k-1}u_{k-1} + v_{k-1}$$

Die Anderen zwei sagen aus, dass das Rauschen in unserer Messung nicht mit der Zeit korrelieren darf, also Zeitunabhängig ist und mit Zwei Werten exakt beschrieben werden kann, nämlich mit einem Durchschnittswert und einer Kovarianz.

$$v_k \approx WN(\bar{x}, Q_k)$$

Kurz zusammengefasst, erhöhen wir die Genauigkeit der Messwerte nach der eigentlichen Messung, mit Hilfe der vorangegangenen Ergebnisse. Wir berechnen dafür den aktuellen Zustand des Systems unter Berücksichtigung der aktuellen Messwerte, und der Werte von vorherigen Messungen und der dazugehörigen Wahrscheinlichkeiten und Varianzen. Somit bittet der Kalman-Filter eine gute Möglichkeit entweder von mehreren Quellen, oder von einer Quelle zur unterschiedlichen Zeiten erfasste Werte, mit einander zu kombinieren, um zu einem statistisch genaueren Ergebnis zu kommen.

Bis Jetzt war die Rede von Messungen eines ruhenden Systems, d.h. die einzelnen Messwerte wären im Idealfall genau gleich, was ist aber wenn sich das Objekt dessen Zustand erfasst wird, zwischen der ersten und der zweiten Messung bewegt. Um diesen Effekt im griff zu bekommen müssen wir in der so genannten „prediction phase“ eine Vorhersage treffen. In der „prediction phase“ verwenden wir alles was wir über dass System wissen um ihren Zustand zur Zeit der nächsten Messung zu ermitteln. Nehmen wir an unser Quadrokopter bewegt sich mit einer konstanten Geschwindigkeit, so können wir die einzelnen Messwerte (Lage im Raum) nicht einfach auf einander beziehen, doch mit der Kenntnis des Systems und des Zustands in dem es sich zur Zeit *t* befindet, können wir bestimmen wie sich der Ort des Quadrokopters bis zur zweiten Messung zur Zeit *t + dt* ändert. So vergleichen wir die Messung zur Zeit *t + dt* nicht einfach mit dem alten Zustand, sondern mit dem alten Zustand projiziert in die Zeit *t +dt.* Dabei unterscheidet kalmansche Lehre drei grundlegende Bewegungsarten.

1.7.1. Dynamical motion

Dynamical motion oder die Dynamische Bewegung, ist eine Bewegungsart bei der wir die Annahme treffen, dass das System eine gleich bleibende und seit der letzten Messung nicht veränderte Bewegung ausführt. Es sei, zum Beispiel x, die zu letzt gemessene Position des Systems zum Zeitpunkt t und v die aktuelle Geschwindigkeit, dann resultiert daraus, dass zur

Zeit $t + dt$ das System sich an der Position $x + v * dt$ befinden wird und sich mit der Geschwindigkeit v weiter bewegt.

1.7.2. Controll motion

Controll motion, eine Kontrollierte Bewegung. Wie der Name schon vermuten läst, ist es eine Bewegungsart die wir vorwiegend bei Systemen anwenden, die von Außen gestört werden und geregelt werden müssen. Das bedeuten anders ausgeduckt, wir kriegen eine Reaktion des Systems auf eine beliebige Bewegungsart die wir selber initiieren. Am Beispiel unseres Quadrokopters heißt es, gibt man zum Zeitpunkt t dem Quadrokopter, der sich an der Position x befindet, ein Befehl zum Beschleunigen, so erwarten wir den Quadrokopter nicht an der Position $x + v * dt$, am Zeitpunkt $t + dt$, sondern ein Stück weiter, wegen der Beschleunigung die wir vorgeben und dem entsprechend auch mit einer höheren Geschwindigkeit $v + dv$.

1.7.3. Random motion

Der Name spricht auch in deisem Fall für sich, die Random motion oder die zufällige Bewegung, schließt alle die Bewegungen mit ein, die wir nicht vorhersehen können. Sei es ein Windstoß oder ein Rottorbruch.

1.8. OpenCV

OpenCV ist eine quelloffene Programmbibliothek unter BSD-Lizenz[1], kann also auch kommerziell frei genutzt werden. Sie ist für die Programmiersprachen C und C++ geschrieben und enthält Algorithmen für die Bildverarbeitung und maschinelles Sehen. Das CV im Namen steht für Computer Vision. Die Entwicklung der Bibliothek wurde von Intel initiiert und wird heute hauptsächlich von Willow Garage gepflegt. Im September 2006 wurde die Version 1.0 herausgegeben, Ende September 2009 folgte nach längerer Pause die Version 2.0.0, welche die Bezeichnung "Gold" trägt. Die Stärke von OpenCV liegt in ihrer Geschwindigkeit und in der großen Menge der Algorithmen aus neuesten Forschungsergebnissen. Die Bibliothek umfasst unter anderem Algorithmen für Gesichtsdetektion, 3D-Funktionalität, Haar-Klassifikatoren, verschiedene sehr schnelle Filter (Sobel, Canny, Gauß) und Funktionen für die Kamerakalibrierung. [1]

OpenCV-Bibliothek ist frei im Internet erhältlich, und eignet sich für die unterschiedlichsten Bildverarbeitungsaufgaben.

[1] BSD-Lizenz, steht für Berkeley Software Distribution - Lizenz und bezeichnet eine Gruppe von Lizenzen aus dem Open-Source-Bereich. Der Urtyp der Lizenz stammt von der University of California, Berkeley. Software unter BSD-Lizenz darf frei verwendet, kopiert, verändert und verbreiten werden. Einzige Bedingung ist, dass der Copyright-Vermerk des ursprünglichen Programms nicht entfernt werden darf. Somit eignet sich unter einer BSD-Lizenz stehende Software auch als Vorlage für kommerzielle Produkte.

Beschäftigt man sich mit OpenCV eine kurze Zeit, merkt man, dass es ein starkes Werkzeug ist, wenn es darum geht Informationen Bildern oder Videosequenzen zu entlocken. Folgende Disziplinen Zählen zu den wichtigsten OpenCV-Aufgaben:

- Mensch-Computer Interaction (HCI)
- Object Identifikation, Segmentierung und Erkennung
- Gesichtserkennung
- Gestenerkennung
- Objektverfolgung
- Structure From Motion (SFM)[1]
- Mobile Robotics

1.8.1. Aufbau von OpenCV

Die OpenCV-Bibliothek läst sich in 4 wesendliche Teile aufgliedern.

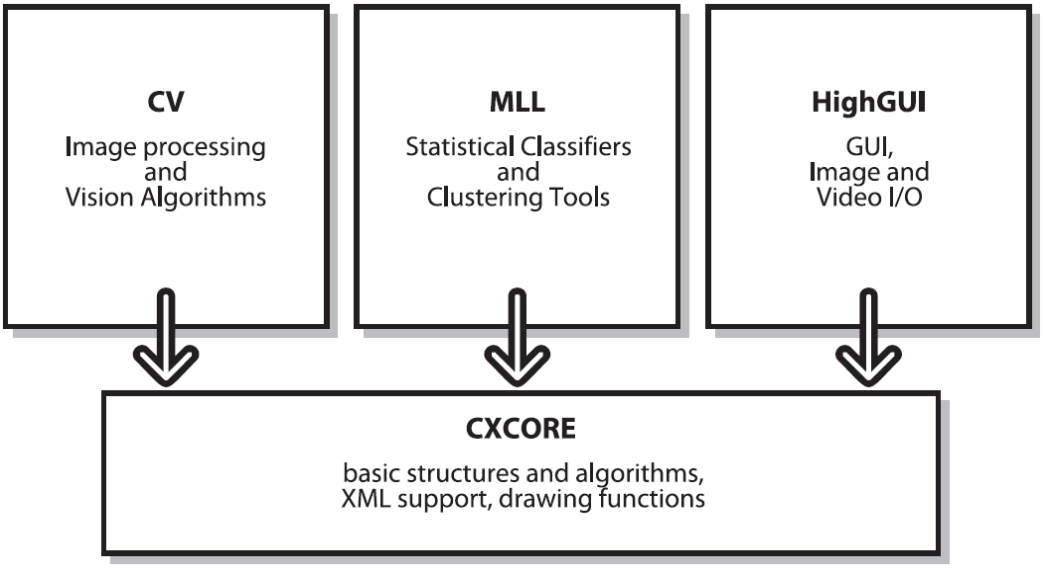


Abbildung 1-18: Aufbau der OpenCV-bibliothek (Quelle[2])

[1] Structure From Motion, Nennt man in der Wissenschaft das Aufbauen von Dreidimensionalen Räumen, aus Zweidimensionalen Bildsequenzen. Dabei verlässt man sich im Wesentlichen auf die Tatsache, dass sich weit entfernte Gegenstände, optisch langsamer bewegen als diejenigen die sich in der Nähe der Kamera aufhalten.

CV steht für *Computer Vision* und benennt alle Algorithmen zur Bild- und Videoverarbeitung, so wie das Verzerren der Bilder (scheren, drehen, skalieren), oder Konvertierungen in verschiedene Bildformate.

MLL heißt *Mashine Learning Library*, es beinhaltet statistische Methoden, Klassifikatoren, Clustering Tools und markiert somit den künstlichen Intellekt von OpenCV.

HighGUI ist für die Kommunikation mit dem Benutzer zuständig, Fensterroutine, Geometrie, aber auch das Laden, das Lesen und Schreiben der Dateien gehört zu ihren Aufgaben.

CXCORE stellt den Datenverkehr und Kommunikation zum Betriebssystem dar und sorgt für einen reibungslosen Arbeitsablauf.

1.8.2. Installation und Einstellungen von OpenCV

Es existieren sehr viele Beschreibungen der Installation von OpenCV, ob unter Windows oder Mac Systemen, auf Visual Studio oder Eclipse, für C oder C++. Die Meisten von ihnen sind sehr akkurat und detailliert beschrieben und enthalten genügend Bildmaterial um den Benutzer sicher zum Ziel zu führen. [10][11]

Die Installation ist simpel und Schnell erledigt, bei der Vorbereitung braucht man nichts verstellen, auch den Installationspfad sollte man mit „C:\OpenCVX.X“ belassen. Mit rund 130 MB nimmt die Bibliothek nicht viel Platz auf der Festplatte ein und ist sofort einsatzbereit.

Beim Arbeiten mit Eclipse sollten folgende Enstellungen gemacht werden:

Die Einstellungen in Visual Studio sehen so aus:

S. 356 learning OpenCV

2. Ausgangssituation

2.1. Problembeschreibung

2.1.1. Quadrocopterprojekt

Abbildung 2-1: Quadrocopter der Hochschule Esslingen

2.2. Beschreibung des Uhrsprungprogramms

Das zugrunde liegende Programm, ist ein Beispielcode, des DARPA Grand Challenge[1] Teams von Standfort University. Die Programmiersprache ist C, mit Funktionen der Freien Bibliothek OpenCV. Es implementiert die Erfassung optischer Flüsse in einer AVI- Videodatei. Das Verfahren, das dabei zum Einsatz kommt ist der Lucas & Kanade Algorithmus, in Verbindung mit einem Feature Tracker und den Gauß-Auflösungspyramiden.

[1] Die DARPA Grand Challenge ist eine von der Technologieabteilung Defense Advanced Research Projects Agency des US-amerikanischen Verteidigungsministeriums gesponserter Wettbewerb für unbemannte Landfahrzeuge. Mit der Ausschreibung des Preises soll die Entwicklung vollkommen autonom fahrender Fahrzeuge vorangetrieben werden.

Der Ablauf des Programms sieht folgendermaßen aus. In der Anfangsphase werden Variablen deklariert, Pointer auf das zu untersuchende Video angelegt, je nach Auflösung der Frames, Platz im Speicher reserviert. Der wesentliche Teil des Ablaufs passiert in einer endlosen while-Schleife. Dort werden, pro Durchgang zwei Frames aus dem Videofile gelesen und in Schwarzweißbilder umgewandelt, spätere Schätzung des optischen Flusses erfolgt mittels Erfassung der Grauwertverschiebungen, dafür reichen Bilder mit einer Tiefe von acht Bit pro Pixel.

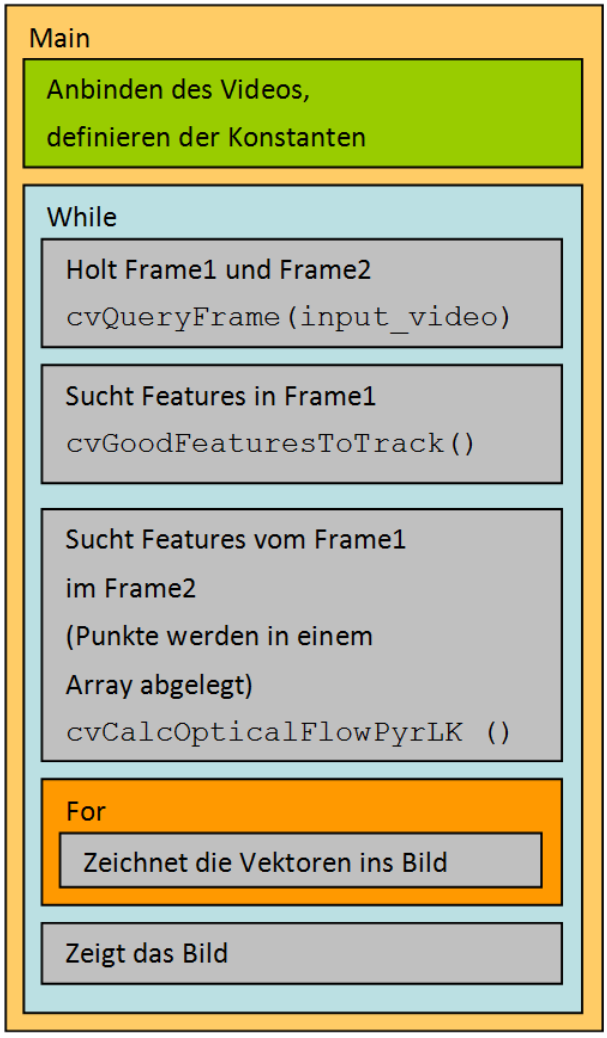


Abbildung 2-2: Grober Programmaufbau

Die Strategie, der Schätzung des optischen Flusses sieht aus, wie folgt.

Ein Array von Koordinatenpunkten (CvPoint) wird angelegt, mit der zuvor festgelegten Anzahl an gesuchten Punkten (Features).

```
CvPoint2D32f frame1_features[Anzahl_Features];
```

Abbildung 2-3: Featurearray

Der nach dem Prinzip von Shi & Tomasi Algorithmus arbeitende Feature Tracker, `cvGoodFeaturesToTrack()` füllt dieses Array mit Koordinaten von Punkten, die die festgelegten Rahmenbedingungen erfühlen. Diese wären, das Feature darf eine bestimmte Intensitätsgrenze nicht unterschreiten und es muss ein minimaller Abstand zwischen den Features bestehen.

Die Funktion berechnet zunächst den minimalen Eigenwert für jeden Bildpunkt der Quelle, mit der Funktion `cvCornerMinEigenVal()`und legt ihn in `eig_image` ab.

```
public static void cvGoodFeaturesToTrack(
        IntPtr image,
        IntPtr eigImage,
        IntPtr tempImage,
        IntPtr corners,         // frame1_features
        ref int cornerCount,
        double qualityLevel,
        double minDistance,
        IntPtr mask,
        int blockSize,
        int useHarris,
        double k
)
```

Abbildung 2-4: Definition von cvGoodFeaturesToTrack

Danach wird alles, außer den lokalen Maxima in 3x3 Nachbarschaft, entfernt, so bleiben nur die Ecken und Stellen mit einem großen Wiedererkennungswert, zur weiteren Betrachtung. Im nächsten Schritt werden die Stellen entfern, die einen minimalen Eigenwert von weniger als `quality_level*max(eigImage(x,y))` besitzen. Schließlich sorgt die Funktion dafür, dass alle Ecken eine bestimmte Distanz von einander einhalten `min_distance`. Dabei lässt die Funktion bevorzugt die Features fallen, die später gefunden wurden, und aus diesem Grund, als schwächer gelten. Nach einem erfolgreichen Durchlauf des Algorithmus, werden die Koordinaten von den gefundenen Featurepunkten in dem Punktearray `corners` abgelegt.

Ein weiteres Array von Punkten wird angelegt, diesmal für das zweite Frame `frame2_features[Anzahl_Features]`. Die zwei bestehenden Arrays werden mit den zu untersuchenden Bildern an die Funktion `cvCalcOpticalFlowPyrLK()` übergeben. Diese Funktion berechnet den optischen Fluss nach dem Lucas & Kanade und Gaußpyramiden Verfahren. Dabei such der Algorithmus eine Pixelwanderung unmittelbar in der Nähe der übergebenen Punkte des ersten Arrays, die Größe der Suchregion wird von `winSize` vorgegeben. Die Gaußpyramiden werden in zwei temporären Bildern aufgebaut (`prevPyr` und

currPyr), die Anzahl der Pyramidenschichten wird von der Variable level vorgegeben. Das Bytearray status enthält immer bei den Elementen eine Null, bei denen der Algorithmus keine Übereinstimmung zum jeweiligen Feature aus dem ersten Bild, im Bild zwei finden konnte. Im Gegensatz dazu enthalten im Floatarray trackError alle nicht gefundenen Elemente eine Einz.

```
public static void cvCalcOpticalFlowPyrLK(
        IntPtr prev,
        IntPtr curr,
        IntPtr prevPyr,
        IntPtr currPyr,
        PointF[] prevFeatures, // frame1_features
        PointF[] currFeatures, // frame2_features
        int count,
        Size winSize,
        int level,
        byte[] status,
        float[] trackError,
        MCvTermCriteria criteria,
        LKFLOW_TYPE flags
)
```

Abbildung 2-5: Definition von cvCalcOpticalFlowPyrLK

Das criteria besteht aus zwei Variablen und sagt aus wann die suche erfolgreich ist, und wann nicht, die erste Variable CV_TERMCRIT_ITER besagt wie viele Widerholungen bei der Suche nach Features gemacht werden dürfen, bevor die suche abgebrochen wird, die zweite CV_TERMCRIT_EPS besagt wie genau die Übereinstimmung der Features sein soll, die erreicht werden muss, um sagen zu Können dass das Feature gefunden wurde. Die Funktion füllt, die dabei entstandene Ergebnisse in das Zweite Punktearray. Alles was noch zu tun bleibt, ist die Punkte aus den Arrays jeweils zu verbinden

$$v \approx \begin{bmatrix} p_{Arr1i} \\ p_{Arr2i} \end{bmatrix}$$ mit 0 < i ≤ Anzahl der Features.

In einer For-Schleife werden die einzelnen Vektoren ins Bild gezeichnet. So erhält man ein, mehr oder weniger dichtes Vektorfeld, mit einer größeren Konzentration an Stellen mit vielen Kontrasten und erkennbaren Bewegungen.

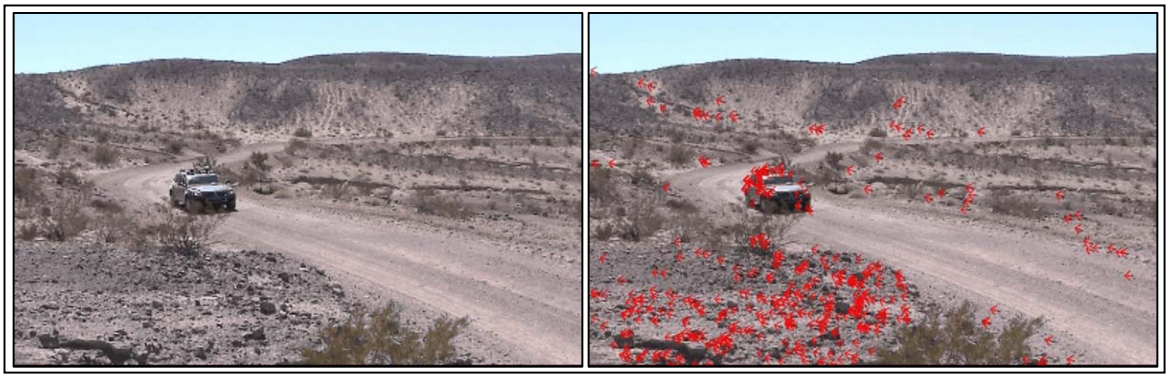


Abbildung 2-6: Input- und Output- Bilder des Programms, des Stanford DARPA Teams

Das Programm zeigt sehr gut die Funktion, der Flusserkennung. In zahlreichen Kommentaren werden die einzelnen Schritte, relativ detailliert erklärt. Die Aufgabe, einen an die Problematik der Flusserkennung und an ihre Lösung mit der OpenCV-Bibliothek, möglichst schnell heran zu führen erfüllt das Beispielprogramm sehr gut. Man kriegt schnell ein Gefühl für den Aufwand der von Nöten ist, um den optischen Fluss zu erfassen, der zwar in einer niedrigen Qualität und relativ hohem Risiko belastet ist, Falsche Ergebnisse zu liefern, der allerdings ziemlich früh die eigenen Erfolge feiern läst, da man vieles vorgezeigt bekommt und nicht alles blind ausprobieren muss.

2.3. Benötigte Zeichenfunktionen

Die Zeichenfunktionen [9] von OpenCV haben zwar nicht viel mit der Funktionalität der Algorithmen zur Schätzung des optischen Flusses zu tut, doch sind sie sehr Hilfreich wenn es darum geht, Ergebnisse und Zwischeninformationen anschaulich zu präsentieren.

Während der Arbeit mit OpenCV benötigte ich im wesendlichen, drei Zeichenmethoden, die ich im Weiteren kurz beschreibe.

2.3.1. cvLine()

```
void cvLine(
      CvArr * img,
      CvPoint pt1,
      CvPoint pt2,
      CvScalar color,
      int thickness=1,
      int lineType=8,
      int shift=0
)
```

Abbildung 2-7: Definition von cvLine

Die Funktion `cvLine()` zeichnet eine Linie in das übergebene Bild `img`, die beiden Punkte `pt1` und `pt2` stehen dabei für den Anfang und das Ende der Linie. Die Variablen `color`, `thickness`, `lineType` und `shift` verändern ihr Aussehen.

Da es in OpenCV keine explizite Funktion gibt, die Pfeile zeichnen könnte, müssen Pfeile aus Linien aufgebaut werden, dazu braucht man die Konstante pi als:

```
static const double pi = 3.14159265358979323846;
```

Abbildung 2-8: Definition von pi

Bildung der Pfeile funktioniert dann folgendermaßen:

```
angle2 = atan2( (double) b.y - d.y, (double) b.x - d.x );
cvLine( frame1, b, d, main_line_color, main_line_thickness, CV_AA, 0 );
b.x = (int) (d.x + 9 * cos(angle2 + pi / 4));
b.y = (int) (d.y + 9 * sin(angle2 + pi / 4));
cvLine( frame1, b, d, main_line_color, main_line_thickness, CV_AA, 0 );
b.x = (int) (d.x + 9 * cos(angle2 - pi / 4));
b.y = (int) (d.y + 9 * sin(angle2 - pi / 4));
cvLine( frame1, b, d, main_line_color, main_line_thickness, CV_AA, 0 );
```

Abbildung 2-9: Pfeil bilden aus Linien

2.3.2. cvCircle()

```
void cvCircle (
      CvArr * img,
      CvPoint Zentrum,
      int radius,
      CvScalar color,
      int thickness=1,
      int lineType=8,
      int shift=0
)
```

Abbildung 2-10: Definition von cvCircle

Diese Funktion zeichnet einen Kreis, ins Bild `img`, dabei ist der Punkt `Zentrum` der Mittelpunkt des Kreises und die Variable `radius` steht für den Radius. Die anderen Übergabeparameter steuern, analog zum `cvLine` das Aussehen der Linie, des Kreises.

Diese Methode verwendete ich um Features die von der `cvGoodFeatureToTrack()` Funktion gefunden wurden, zu markieren.

2.3.3. cvPutText()

```
void cvPutText(
      CvArr* img,
      const char* text,
      CvPoint org,
      const CvFont* font,
      CvScalar color
)
```

Abbildung 2-11: Definition von cvPutText

Die Funktion `cvPutText()` schreibt einen beliebigen Text `text`, ins Bild `img`. Der Punkt `org`, definiert die Position an der, der untere, linke Rand des Textes beginnt, `cvPoint(0,0)` steht dabei für die linke obere Ecke des Bildes. Man kann die Farbe des Textes mit der Variable `color` einstellen, mit

```
CvScalar color = CV_RGB(0,255,0);
```

Abbildung 2-12: Beispiel zum CvScalar, für grüne Farbe

Diese Funktion bittet eine gute Möglichkeit unterschiedliche Ergebnisse auszugeben, so wie Zwischenzeiten des Algorithmus, oder X- und Y-Komponente des erfassten optischen Flusses.

2.4. Modifikationen des Programms

Gefordert ist eine Softwarelösung zur Bewegungsschätzung, um später eine darauf basierende Bewegungsregelung für den hochschuleigenen Quadrocopter zu entwickeln. Das bedeutet es bedarf kein flächendeckendes Vektorfeld, sondern nur einen möglichst genauen Vektor, genauer seine X- und Y-Komponente, die als Steuergrößen im Regler verarbeitet werden können.

So war mein erster Schritt zur Lösung, einen Durchschnittsvektor zu bilden. So rechnete ich in der For-Schleife alle X- und Y-Komponenten der Teilvektoren zusammen und teilte das Ergebnis durch die Anzahl der Features, oder Teilvektoren. Hierbei sollte man von der Anzahl der gefundenen Features ausgehen und nicht von der ursprünglich angesetzten Anzahl der Features. dazu dienen die beiden Arrays `status` und `trackError`, und die Variable `count`. Zur Erinnerung, das i-te Element des Arrays `status` enthält eine Null, falls das dazugehörige Feature nicht verfolgt werden konnte, das Array `trackError` wird in diesem Element eine Eins enthalten. Die Variable `count` enthält die Anzahl der Features, sie wird während der Laufzeit des Algorithmus stets nach unten korrigiert, falls ein Feature im zweiten Bild nicht gefunden wurde, so enthält die Variable immer die aktuelle Featureanzahl.

Um eine gewisse Übersicht im Code zu gewährleisten, trennte ich das bisherige Programm in mehrere kleinere Teile auf und führte zur Vereinfachung und größerer Verständlichkeit, neue Variablentypen ein. Solche wie

```
struct TwoFrames {
        IplImage *frame1;
        IplImage *frame2;
        };
```

Abbildung 2-13: Definition der Struktur TwoFrames

und

```
struct Pointer {
        CvPoint p1;
        CvPoint p2;
        };
```

Abbildung 2-14: Definition der Struktur Pointer

Diese Strukturen dienen nur einer besseren Übersicht und Kompaktheit des Codes. Der eigentliche Anstoß zur Einführung dieser Strukturen, war die Überlegung zwei anschauliche Objekte, als Eingang und Ausgang zu besitzen, die während dem Ablauf des Programms in einander umgerechnet werden. Aus zwei Bildern wird ein Pfeil.

Zur Testzwecken war es notwendig Längenangaben des Pointers erfassen zu können, dazu dienen die Funktionen `LaengePtr`, `LaengePtrX` und `LaengePtrY`, mit

```
int LaengePtr (struct Pointer P){
        return sqrt(square(LaengePtrX (P)) + square(LaengePtrY (P)));
        }
int LaengePtrX (struct Pointer P){
        return (P.p1.x > P.p2.x)? (P.p1.x - P.p2.x) : (P.p2.x - P.p1.x);
        }
int LaengePtrY (struct Pointer P){
        return (P.p1.y > P.p2.y)? (P.p1.y - P.p2.y) : (P.p2.y - P.p1.y);
        }
```

Abbildung 2-15: Funktionen zur Ausgabe der Längen eines Pfeils

und

```
inline static double square(int a){
        return a * a;
        }
```

Abbildung 2-16: Funktion square

Des Weiteren wanderte die gesamte Zeichen-Prozedur, die für das Zeichnen der Pfeile zuständig war, in eine eigene Funktion `FlowPointer()`, die ein Pfeil `ptr` in die Mitte des Übergebenen Frames `frame` zeichnet.

```
int FlowPointer(
        IplImage *frame,
        struct Pointer ptr,
        CvSize frame_size,
        double angle,
)
```

Abbildung 2-17: Definition von FlowPointer

Für die Berechnung des Bewegungsvektors wurden eigene Methoden eingeführt, die unterschiedliche OpenCV- Algorithmen zur Bestimmung des optischen Flusses benutzen, diese werden im eigenen Kapitel behandelt.

Durch die genannten Modifikationen im Programmablauf wurde die `main` sehr entlastet und gewann an Übersichtlichkeit, was zum Beispiel die Zeitmessung wesentlich einfacher macht.

2.5. Methoden zur Erfassung des Optischen Flusses

Die OpenCV- Bibliothek Realisiert sowohl die Korrelationsbasierte Verfahren mit Block Matching, als auch Gradientenverfahren mit dem Horn & Schunck Algorithmus und dem Algorithmus von Lucas & Kanade, zur Erfassung des optischen Flusses. Insgesamt sind es vier Methoden, eine davon, `cvCalcOpticalFlowPyrLK()` wurde bereits in dem Beispielprogramm von Stanford University realisiert und im Abschnitt 2.2 vorgestellt. Weitere Methoden sind `cvCalcOpticalFlowBM()`, `cvCalcOpticalFlowLK()` und `cvCalcOpticalFlowHS()`, die ich in Folgendem näher beschreibe.

2.5.1. cvCalcOpticalFlowBM

Die Funktion `cvCalcOpticalFlowBM()`funktioniert nach dem Block-Matching Algorithmus und gehört somit zu den Korrelationsbasierten Verfahren der Flusserkennung.

```
void cvCalcOpticalFlowBM(
      const CvArr* prev,
      const CvArr* curr,
      CvSize blockSize,
      CvSize shiftSize,
      CvSize max_range,
      int usePrevious,
      CvArr* velx,
      CvArr* vely
)
```

Abbildung 2-18: Definition von cvCalcOpticalFlowBM

Die Variablen `prev` und `curr` sind zwei Bilder, an denen der Optische Fluss detektiert wird, `blockSize` ist die Größe der Blöcke, in die das Bild aufgeteilt wird. Variable `shiftSize` ist die Schrittweite, oder Häufigkeit, mit der die Blöcke im Bild verteilt werden, `max_range` gibt vor, wie weit um die Blöcke herum nach Ähnlichkeiten gesucht wird. Die Variable `usePrevious` sagt aus, ob vorher ermittelte Ergebnisse mitberücksichtigt werden sollen, dann sucht der Algorithmus zu erst in der Richtung, wo der letzte Vektor des Blocks hin gezeigt hat. Die 32-bit floating-point, single-channel Bildmatrizen `velx` und `vely` enthalten die Geschwindigkeitskomponenten, jeweils in X- und Y-Richtung, für jeden Block des Bildes.

```
velx = cvCreateImage(CvSize(120,160), IPL_DEPTH_32F, 1)
```

Abbildung 2-19: Beispiel der Allokation von velx

Für jeden Block gibt es also ein Pixel in `velx` und in `vely`. Die Bildung der Höhe und Breite von `velx` und `vely` funktioniert so:

$$Höhe_{VEL_{X/Y}} = \left[\frac{Höhe_{prev} - Höhe_{blockSize}}{Höhe_{shiftSize}} \right], \; Breite_{VEL_{X/Y}} = \left[\frac{Breite_{prev} - Breite_{blockSize}}{Breite_{shiftSize}} \right]$$

Formel 2-1: Bildung der Größe, von velx und vely, bei Block Matching

2.5.2. cvCalcOpticalFlowLK

Die Funktion `cvCalcOpticalFlowLK()` funktioniert nach dem Algorithmus von Lucas & Kanade, gehört also zu den Gradientenverfahren.

```
void cvCalcOpticalFlowLK(
      const CvArr* prev,
      const CvArr* curr,
      CvSize winSize,
      CvArr* velx,
      CvArr* vely
)
```

Formel 2-2: Definition von cvCalcOpticalFlowLK

Die Bilder `prev` und `curr`, sind das zu untersuchende Bildpaar. Die Variable `winSize` steht für die Größe der Blöcke, in den der Fluss als konstant angenommen wird. `velx` und `vely` sind ähnlich wie beim `cvCalcOpticalFlowBM()`, X- und Y-Komponente der Bewegung. Diese Bildmatrizen sind hier aber genauso groß wie die übergebenen Bilder, `prev` und `curr`, sind jedoch in jedem Fall 32-bit floating-point, single-channel Bilder. Denn auch wenn die Annahme herrscht, dass in einem bestimmten Block von der Größe `winSize` ein konstanter Fluss existiert, wird für jeden Pixel des Bildes ein eigener Vektor, in Form von seiner X- und Y-Komponente abgelegt.

2.5.3. cvCalcOpticalFlowHS

Die Funktion `cvCalcOpticalFlowHS()` funktioniert nach dem Algorithmus von Horn & Schunck und gehört neben der `cvCalcOpticalFlowLK()` zu den Gradientenverfahren.

```
void cvCalcOpticalFlowHS(
      const CvArr* prev,
      const CvArr* curr,
      int usePrevious,
      CvArr* velx,
      CvArr* vely,
      double lambda,
      CvTermCriteria criteria
)
```

Abbildung 2-20: Definition von cvCalcOpticalFlowHS

Die Pointer `prev` und `curr` zeigen, wie bei den vorangegangenen Methoden auf die zwei übergebenen Bilder. Die Variable `usePervious` entscheidet ob die zuvor ermittelten Ergebnisse in die Rechnung miteinbezogen werden, oder nicht. `velx` und `vely` stehen auch hier für die in jedem Pixel ermittelte Geschwindigkeit, da der Algorithmus von Horn & Schunck wirklich für jeden Pixel ein Bewegungsvektor erstellt, müssen die Matrizen genauso groß sein wie die Bilder `prev` und `curr`. Die Variable `lambda` repräsentiert den, aus der Formel 1-16 bekanten Faktor des Rauschens.

$$e_{H\&S}(u,v) = e_d(u,v) + \boldsymbol{\lambda} \cdot e_s(u,v)$$

Formel 2-3: Rauschfaktor in der Formel von Horn & Schunck

Das `criteria` legt fest wann die Suche terminiert, es ist eine Struktur, bestehend aus zwei Variablen, die erste besagt wie viele Widerholungen gemacht werden, so lange es keine ausreichende Ähnlichkeit gibt, die andere ist ein Faktor der Ähnlichkeit, besagt also wie genau die Übereinstimmung sein muss, um sagen zu können das die Pixelwanderung gefunden wurde.

3. Test

3.1. Aufbau der Testumgebung

Die Testumgebung, ist ein wenig modifizierter, schon aus dem Physiklabor bekannter Versuchsaufbau, mit dem sich Beschleunigungen und die Erdanziehungskraft messen lassen. An vorderster Stelle stand die Anforderung, die Tests möglichst reproduzierbar zu machen. Der

Aufbau bittet eine sehr gute Möglichkeit eine identische Bewegung vom test zu test zu gewährleisten. Der Versuchsaufbau ist in folgendem Bild dargestellt.

Abbildung 3-1: Versuchsaufbau

Eine herkömmliche Digitalkamera, befestigt an den Beweglichen Wagen des Aufbaus, wird über eigens dafür angefertigte texturreiche Untergründe gefahren. Die Texturen sind sehr Symmetrisch, was ihre spätere Reproduktion leichter machen Dürfte. Relativ viele Gewichte am Wagen, sorgen dafür, dass die Kamera möglichst stabil gelagert ist und die Bewegung nicht so anfällig ist, für kleine Störfaktoren, wie zum Beispiel das Gewicht des Kabels und der Zug der durch den Kabel, mit dem die Kamera zum Rechner verbunden ist, verursacht wird. Der Wagen ist über einen Pfaden mit einem Gewicht, am ende der Testbahn befestigt und wird von der auf diesen Gewicht wirkenden Erdanziehungskraft Beschleunigt. Das beschleunigende Gewicht ist sehr klein gehalten, damit die Geschwindigkeit nicht übertrieben groß wird und die Bewegung längere Zeit andauert um möglichst viele Messwerte zu bekommen. So kriegen wir eine weitestgehend konstante Beschleunigung, die einen Anschaulichen Vergleich der, von den Algorithmen gelieferten Ergebnisse erlaubt.

3.2. Probleme beim Testen

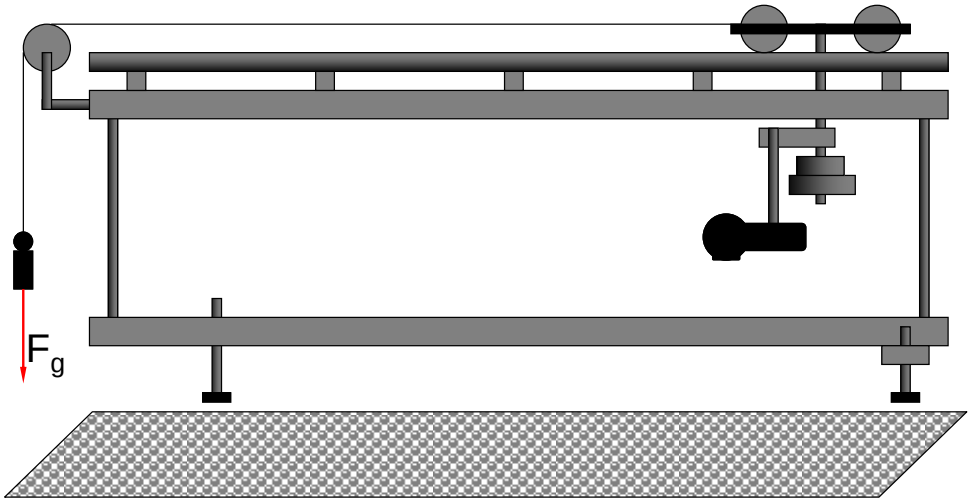


Abbildung 3-2: Skizze des Versuchaufbaus

3.3. Ergebnisse der Tests

Während den Tests untersuchte ich die einzelnen Algorithmen auf ihre Tauglichkeit für die geforderte Aufgabe, den Quadrokopter zu stabilisieren. Jeder von OpenCV gegebenen Algorithmen zur Flusserkennung, besitzt eigene Eigenschaften in Form von Variablen, deren Einstellungen die Qualität des gelieferten Ergebnisses in unterschiedlichem Masse beinträchtigen können. Diese Einflusse, aufs Ergebnis versuchte ich im Ersten Teil der Tests herauszufinden, um gegebenenfalls eine optimale Einstellung der Algorithmen Vorschlagen zu können.

3.3.1. cvCalcOpticalFlowPyrLK

Die Funktion `cvCalcOpticalFlowPyrLK()` besitzt folgende Parameter, die auf das Ergebnis Einfluss nehmen können:

- `int count,`
- `Size winSize,`
- `int level,`
- `MCvTermCriteria criteria`

3.3.2. cvCalcOpticalFlowBM

- `CvSize blockSize,`
- `CvSize shiftSize,`
- `CvSize max_range`

3.3.3. cvCalcOpticalFlowLK

Die einzige Möglichkeit bei `cvCalcOpticalFlowLK()` den Algorithmus zu beeinflussen ist das:

- `CvSize winSize`

Zur Errinerung, der Algorithmus von Lucas & Kanade setzt voraus das der optische Fluss in festen Blöcken, einen gleichen Fluss hat, und nicht wie beispielsweise, beim Horn & Schunck, für jeden Pixel berechnet wird. Das winSize steht genau für diese Blockgröße, wobei es ganz bestimmte Werte gibt, die man für die Kantenlänge der Blöcke einsetzen kann, es sind die 1, 3, 5, 7, 9, 11 und die 13.

Gleich zu beginn Zeigte sich die relativ hohe Rauschanfälligkeit des Algorithmus, was mit nachfolgender Abbildung verdeutlicht wird.

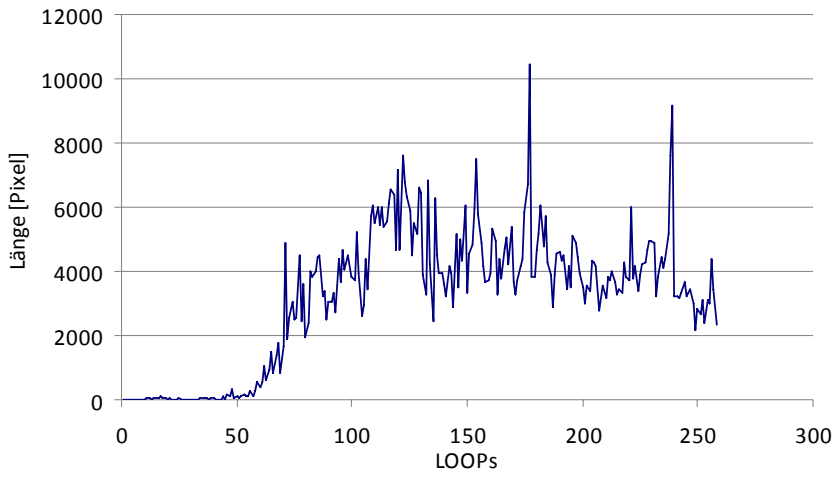


Abbildung 3-3: Pfeilänge beim Algorithmus von Lucas und Kanade, rechteckiges Muster

beim Testen des winSize empfahl sich die Methode, stehende Messungen durchzuführen, also nicht die Kamera fahren zu lassen, sondern in Ruhestellung, nur das Rauschen aufnähmen und analysieren. Folgende Kurve zeigt den dabei ermittelten Einfluss der Blockgröße, auf den Zeigerausschlag.

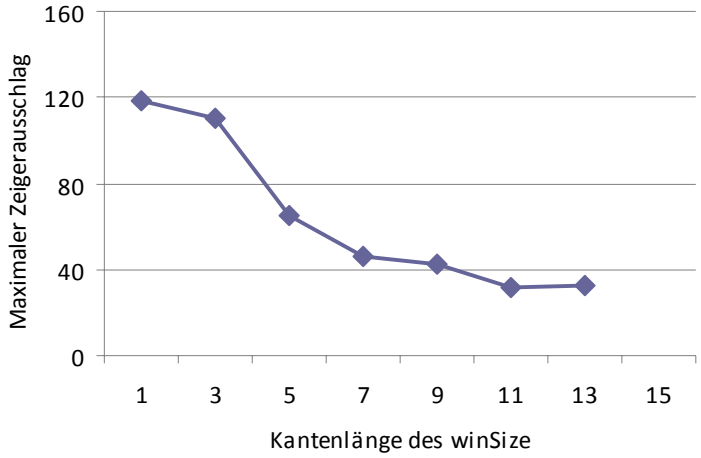


Abbildung 3-4: Verhältnis zwischen winSize und Rauschanfälligkeit, beim Lucas & Kanade Algorithmus

3.3.4. cvCalcOpticalFlowHS

Die Methode `cvCalcOpticalFlowHS()` hat folgende Einstellmöglichkeiten:

- `double lambda,`
- `CvTermCriteria criteria`

Wie Oben bereits erwähnt ist `lamda` der Rauschfaktor, es bestätigt sich auch im Test, um so größer das `lamda` gemacht wird, desto unzuverlässiger wird auch der Ausschlag des Pfeils. Beim Test nach der Abhängigkeit von `lamda` variiere ich ihre Werte von 1 bis 0,001, das Ergebnis sieht wie folgt aus:

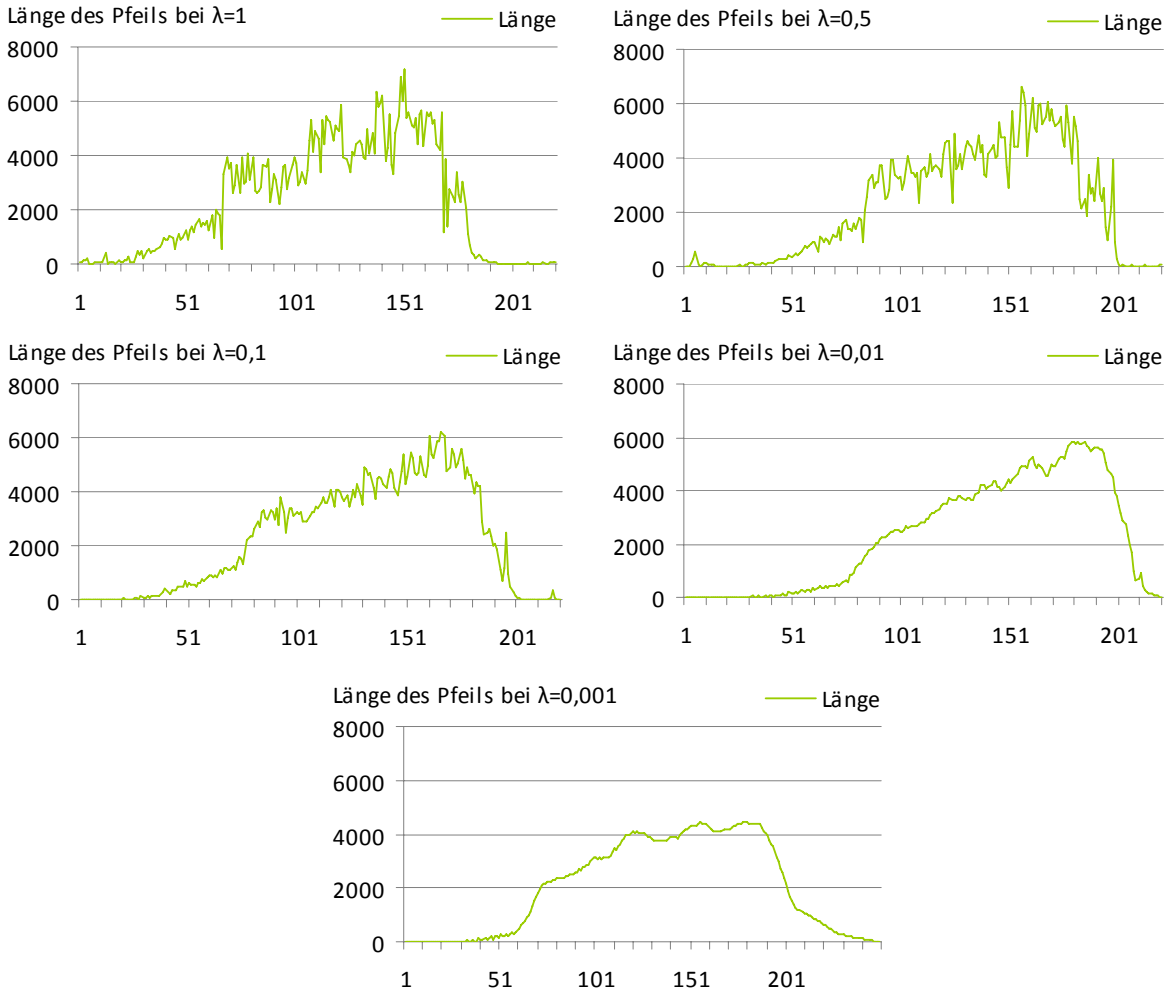


Abbildung 3-5: Variierung von lamda, im Algorithmus von Horn & Schunck

Dabei Steht die Y-Achse für die Länge des errechneten Pfeils, in Pixel und die X-Achse Zeigt die während dessen verstrichenen Loops. Was einem sofort ins Auge sticht ist, dass die Letzten zwei Kurven deutlich glätteren Verlauf haben als die anderen. Die Letzte Kurve, die einen Wert von `lamda`, von 0,001 entspricht, zeigt einen Kurvenverlauf der kaum realistisch zu sein scheint. Es besitzt kaum Rauscheffekte und sprungartige Bewegungen des Pfeils. Was auch auffällig ist, ist dass die letzte Kurve rund ein Drittel kürzere Maximallänge des Pfeils ausgibt, was einen zu dem Entschluss kommen lässt, dass das `lamda`, wenn es zu klein gewählt ist, Dämpfungseffekte besitzt, es bestätigt sich auch mit dem Abklingen der Kurve.

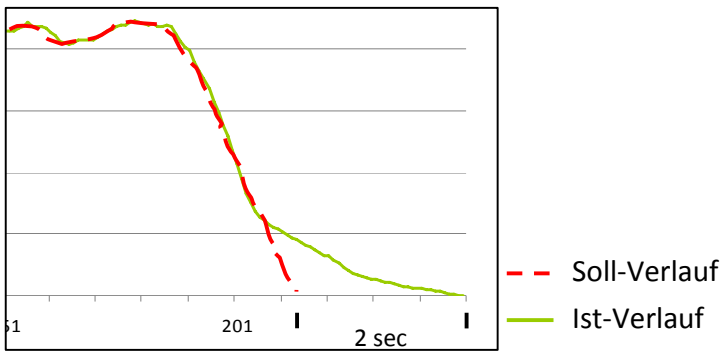


Abbildung 3-6: Abweichung des Kurvenverlaufs von der Wirklichkeit

Bei einem lamda von 0,001 klingt die Kurve nicht ab, wie man es erwarten würde, nämlich so wie der Wagen sich bewegt, sondern nur ganz langsam. Es Entsteht ein Nachschwingen von ziemlich genau 2 Sekunden, was für eine Regelung des Flugapparats große Schwierigkeiten bereiten würde, wenn es, es nicht gar unmöglich macht.

In Anbetracht dieses Ergebnisses bin ich zu dem Entschluss gekommen, dass es die beste Variante wäre den Algorithmus, mit einem lamda = 0,01 zu betreiben. Auch wenn, bei dieser Einstellung keine so glatte Kurve heraus kam, ist das Ergebnis trotzdem genau genug um damit weiter arbeiten zu können. Eine Länge des Pfeils von 6000 entspricht in dem Fall einer Geschwindigkeit von 0,106 m/s, das Rauschen was dabei entsteht, bewegt sich in Bereich von 20 – 40 Pixel, also 0,0005 m/s, verschwindend klein.

4. Literaturverzeichnis

[1] http://de.wikipedia.org/wiki/OpenCV

[2] Learning OpenCV (Gary Bradski und Adrian Kaehler)

[3] Navigation und Regelung eines Luftschiffes mittels optischer, inertialer und GPS Sensoren, Doktorarbeit, Universität Stuttgart (Martin Fach)

[4] Bewegungserkennung in Videosequenzen zur Erfassung von Verkehrsteilnehmern, Diplomarbeit, Hochschule Esslingen (Ralph Scharpf)

[5] http://www.ece.cmu.edu/~ee899/project/deepak_mid.htm

[6] http://de.wikipedia.org/wiki/Methode_der_kleinsten_Quadrate

[7] Vorlesungspräsentation, der Veranstaltung „Bildverarbeitung und Computer Vision, Kapitel 13 – Optische Flüsse, Uni-Münster (von Prof. Dr. Xiaoyi Jiang, Michael Schmeing und Lucas Franek)

[8] Detection and Tracking of Point Features, (Carlo Tomasi und Takeo Kanade)

[9] http://opencv.willowgarage.com/documentation/drawing_functions.html

[10] http://blog.aguskurniawan.net/post/OpenCV-210-with-Visual-Studio-2010.aspx , Installation von OpenCV und Integration in Visual Studio 2010

[11] http://www.jestinstoffel.com/?q=node/112 , Integration von OpenCV in Eclipse