Tigers Mannheim

Prädiktion von Roboter- und Ball-Positionen in der RoboCup Small-Size-League

Implementierung und Validierung

STUDIENARBEIT
des Studienganges Informationstechnik
an der Dualen Hochschule Baden-Württemberg Mannheim

Peter Birkenkampf	**4224606**
Birgit Dölle	**4266423**
Maren Künemund	**4292597**
Marcel Sauer	**4267753**

Mannheim, den 17. Juni 2011

DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Bearbeitungszeitraum:	28.03.2011 - 19.06.2011
Kurs:	TIT08AIN
Ausbildungsfirma:	Deutsches Zentrum für Luft- und Raumfahrt
Betreuer:	Dr.-Ing. Martin Friedmann

Ehrenwörtliche Erklärung

Hiermit versichern wir, dass wir die vorliegende Arbeit selbstständig
und nur unter Benutzung angegebener Quellen und Hilfsmittel angefertigt haben.

Peter Birkenkampf

Birgit Dölle

Maren Künemund

Marcel Sauer

Mannheim, den 17. Juni 2011

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Die Small Size League des RoboCup dient Forschern internationaler Teams als Plattform zur Entwicklung eines intelligenten Robotersystems. Die durch die Teams entwickelten Systeme treten in einem Fußballspiel gegeneinander an. Vollständig autonom agierende Roboter werden hier nicht genutzt. Die Steuerung der Feldspieler erfolgt zentral vom Server des jeweiligen Teams.
Die Lokalisierung der Roboter auf dem Spielfeld erfolgt durch zwei über dem Feld montierten Kameras. Aufgenommene Bilder werden durch die in der Small Size League standardisierten Bildverarbeitungssoftware SSL-Vision [17] analysiert. Als Resultat werden die Positionen der Roboter und des Balls über Ethernet an die verschiedenen Teamserver versendet.
Dabei entsteht ein Messrauschen durch die Nutzung von Kameras in einer realen Umgebung. Außerdem kommt es durch die Versendung der Positionsdaten über Ethernet zu Verzögerungen. In [12] wurde gezeigt, dass sich ein Roboter in der Small Size League in dieser sehr kurzen Latenzzeit bereits bis zu 47cm bewegen kann. Daher ist das Entfernen dieser externen Einflüsse aus den Eingangsdaten essentiell wichtig für die korrekte Arbeit der künstlichen Intelligenz.
In [4] wurden die theoretischen Grundlagen probabilistischer Verfahren erläutert, welche dazu genutzt werden sollen, aus den eingehenden Daten eine zukünftige

Spielposition zu prädizieren. Die vorgestellten Verfahren sind das Extended Kalman Filter und das Partikelfilter. Des Weiteren wurde eine Modellierung der Objektbewegungen in der Small Size League beschrieben, welche durch die zu verwendenden probabilistischen Verfahren genutzt werden sollen.

In der vorliegenden Arbeit wird die Implementierung eines Softwaremoduls für den Teamserver des RoboCup Teams Tigers Mannheim vorgestellt, welches die in [4] vorgestellten Verfahren umsetzt. Im Anschluss wird die korrekte Funktion des Softwaremoduls anhand verschiedener Testszenarien validiert.

Kapitel 2

Implementierung

In diesem Kapitel wird die Implementierung einer Softwarekomponente zur Filterung und Prädiktion von Roboter- und Ball-Positionen beschrieben. Diese wird im Folgenden als Worldpredictor-Modul bezeichnet und ist Teil des Softwaresystems *Sumatra* des RoboCup-Teams Tigers Mannheim.
Im ersten Teil 2.1 dieses Kapitels wird auf den Kontext des Vorhersagemoduls näher eingegangen. Hier wird der Worldpredictor in das Gesamtsystem eingeordnet und der Rahmen der Implementierung erläutert.
Der nächste Teil 2.2 stellt ein dem Worldpredictor vorgeschobenes Modul zur Ermittlung von Höhendaten vor. Dieses ermittelt aus zweidimensionalen Kameradaten, welche von SSL-Vision übermittelt werden, auf Basis perspektivischer Verzerrungen, die Höhe fliegender Bälle und passt deren Koordinaten entsprechend an.
Der dritte Teil 2.3 befasst sich mit dem Aufbau des Worldpredictors. Dabei wird die Arbeitsweise des Moduls vorgestellt. Dieses Teilkapitel beschreibt auch die Position und Aufgabe der Filterverfahren. Auf die Implementierung dieser wird in 2.4 eingegangen.
Teilkapitel 2.5 beschreibt die Implementierung einer Matrix-Klasse, welche für effektive Matrixoperationen innerhalb der Filter und der Bewegungsmodelle verwendet wird.

Der letzte Teil 2.6 dieses Kapitels befasst sich mit den implementierten Komponenten zur Auswertung der korrekten Funktion des Worldpredictors. Diese ermöglichen es Aussagen über die Qualität des Moduls zu treffen und bilden die Basis für die spätere Validierung.

2.1 Kontext des Vorhersagemoduls

Dieser Abschnitt der Arbeit befasst sich mit dem Kontext der Implementierung des Worldpredictors. Dabei wird in 2.1.1 *Sumatra*, das Softwaresystem der Tigers Mannheim, eingeführt. Dieses verwendet das in 2.1.2 beschriebene Modulsystem *Moduli*. Innerhalb von Sumatra basiert alle Kommunikation zwischen Modulen auf dem Observer-Entwurfsmuster. Aus diesem Grund wird dieses Muster in 2.1.3 kurz beschrieben. Der letzte Teil dieses Abschnittes 2.1.4 befasst sich mit der grafischen Benutzeroberfläche von *Sumatra*.

2.1.1 Sumatra

Sumatra, das Softwaresystem der Tigers Mannheim, erhält die Kameradaten von SSL-Vision und die Entscheidungen der Schiedsrichter und generiert daraus Steuerbefehle für die Roboter. Die Software ist modular aufgebaut und in der Programmiersprache Java entwickelt. Die Module beinhalten Implementierungen verschiedener Aufgaben. So ist es die Aufgabe des Cam-Moduls, die Daten von SSL-Vision, der in der Small Size League standardisierten Software zur Auswertung der Kameradaten, entgegenzunehmen und für die Verarbeitung in *Sumatra* vorzubereiten. Das Worldpredictor-Modul ist dafür zuständig, die vom Cam-Modul bereitgestellten Daten zu filtern und auf die zukünftige Spielfeldsituation zu schließen. Mit dieser Information wird im KI-Modul eine geeignete Strategie ausgewählt und das zukünftige Verhalten der eigenen Roboter bestimmt. Das letzte Modul in der Arbeitskette von *Sumatra* ist der Bot-Manager. Dieser nimmt die Befehle aus dem KI-Modul entge-

gen, wandelt sie in Roboterbefehle um und leitet diese an die Roboter weiter. Dieser Ablauf ist in Abbildung 2.1 visualisiert.

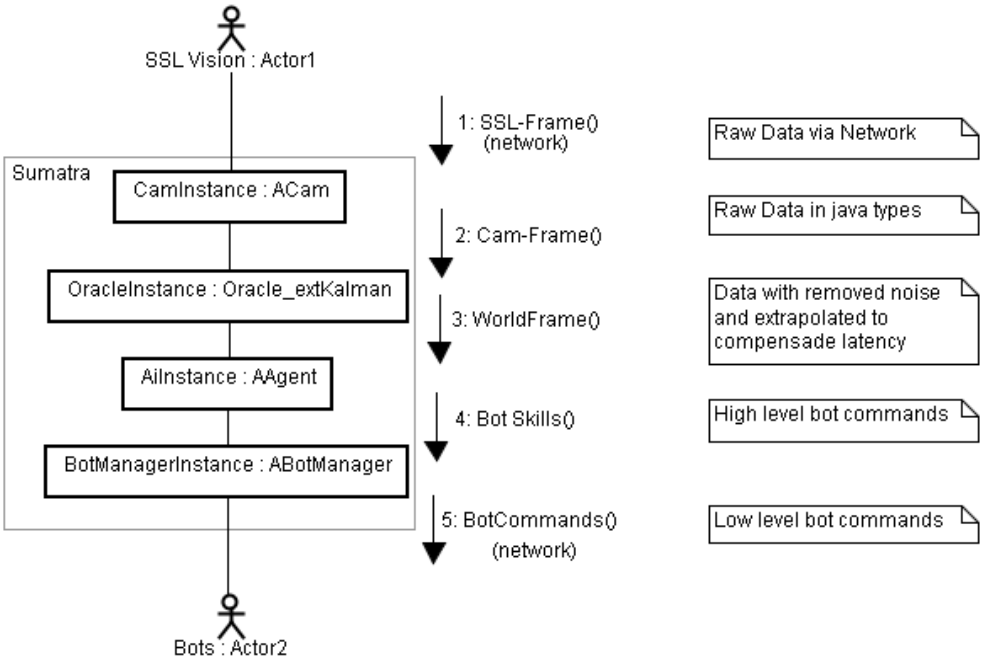


Abbildung 2.1: Ex- und interne Kommunikation von *Sumatra*

2.1.2 Moduli

Sumatra nutzt das Opensource Modulsystem *Moduli* [1]. In einer Konfigurations-Datei ist hinterlegt, welche Implementierungen von Cam-, Worldpredictor-, KI- und anderen Modulen geladen werden sollen. Beim Start von *Sumatra* wird diese Datei eingelesen und die entsprechenden Module geladen. Nach der Initialisierung der Komponenten werden durch *Moduli* Kommunikationsverbindungen unter den Modulen hergestellt. Über abstrakte Klassen und Interfaces[1] wird bestimmt, wie die einzelnen Module aufgebaut sein müssen. Dies bedeutet weiterhin, dass andere Module

[1]abstrakte Klassen ohne Implementierungen, Javaklassen können nur von einer Klasse erben, aber mehrere Interfaces implementieren

auf ein Modul zugreifen können, ohne dass Details über die Implementierung bekannt sind. Als ein von *Moduli* nutzbares Modul müssen von einer Klasse zusätzlich Methoden bereitgestellt werden, um auf ein Starten und ein Beenden des Gesamtsystems reagieren zu können. Dabei werden im ersten Schritt Instanzen der Module erzeugt. Im zweiten Schritt dürfen Verknüpfungen untereinander hergestellt werden. Darauf folgend sind die Module bereit zum Arbeiten. Beim Beenden müssen zuerst die Verknüpfungen aufgehoben werden. Im Anschluss werden die Instanzen der Module aus dem Speicher gelöscht.

2.1.3 Observer

Die Kommunikation innerhalb von *Sumatra* erfolgt mit Hilfe des Observer-Entwurfsmusters. Beim diesem Entwurfsmuster existiert eine beobachtbare Klasse, die als *observable* bezeichnet wird. Diese kann von Instanzen anderer Klassen beobachtet werden, den *Observern*. Dafür besitzt eine Instanz einer beobachtbaren Klasse eine Liste solcher Beobachter. In diese Liste können sich *Observer* eintragen oder auch wieder entfernen lassen. Die beobachtbare Klasse reagiert auf Ereignisse, indem sie eine Methode jedes registrierten Beobachters aufruft. Die Methode ist durch ein Interface definiert, welches von allen *Observern* implementiert werden muss.
Auf *Sumatra* bezogen ist beispielsweise `SSLVisionCam` eine Implementierung des Cam-Moduls. Dieses ist *observable* und leitet an alle registrierten *Observer* `CamFrame`s weiter. Die *Observer*, wie z.B. das Worldpredictor-Modul implementieren das Interface `ICamObserver`. Beim Starten der Software trägt sich der Worldpredictor bei der `SSLVisionCam` ein. Die nötige Referenz zur Instanz wurde über *Moduli* an den Worldpredictor übergeben. Bei einem eingehenden `CamFrame` wird nun immer die Methode `onNewCamDetectionFrame` des Worldpredictors aufgerufen. Auf diese Art wird die Kommunikation zwischen Modulen ermöglicht.
Durch die Observer-Struktur innerhalb von *Sumatra* ist es möglich, ohne Wissen über die Implementierung von anderen Modulen Daten zu empfangen oder Daten an

andere Module weiterzuleiten. Zudem können Ergebnisse eines Moduls problemlos an mehrere Stellen geschickt werden. Zum Beispiel könnten ausgehende Daten ohne Änderungen an den eigentlichen Modulen vorzunehmen durch einen zusätzlichen *Observer* abgegriffen und dann in eine Ausgabe kopiert oder visualisiert werden.

2.1.4 Grafische Benutzeroberfläche

Zur Steuerung von Sumatra wurde eine grafische Benutzeroberfläche erstellt, über die u.a. das System gestartet, gestoppt oder weitere Einstellungen vorgenommen werden können. Die Benutzeroberfläche zeigt zu jedem Modul Daten an und erlaubt eine anschauliche Visualisierung komplexer Sachverhalte. Da auch die Elemente der grafischen Benutzeroberfläche nur das Observer-Entwurfsmuster und *Moduli* kommunizieren, ist es einfach möglich, unter Wettkampfbedingungen die grafische Benutzeroberfläche zu Gunsten der Performance abzuschalten.

2.2 Erkennung fliegender Bälle

Ein wichtiges Thema der RoboCup Small Size League ist die Erkennung und Vorhersage von fliegenden Bällen, da von immer mehr Teams sogenannte Chip-Kicker verwendet werden. (vgl. [16]) Mit dieser Vorrichtung ist es möglich, Flanken, wie beim menschlichen Fußball zu schießen. Um mit solchen Bällen umgehen zu können, müssen diese zunächst erkannt werden. Daher wird zwischen Cam- und Worldpredictor-Modul das „BallCorrector“-Modul eingefügt. Dessen Umsetzung wird in diesem Kapitel beschrieben.

Als Grundlage dienen dem Modul die Daten, welche die Bilderkennungssoftware SSL-Vision ausgibt. SSL-Vision liefert, neben den Positionen und Ausrichtungen der Roboter, die Positionen des Balls, die zum jeweiligen Zeitschritt erkannt wurde. Der Ball befindet sich mit einer Höhe von Null allerdings immer auf dem Boden, da die Software fliegende Bälle nicht erkennen kann. Die von der Kamera aufgenommenen

Ballpositionen sind im Fall eines Fluges inkorrekt, da nicht die wirkliche Position, sondern die auf das Spielfeld projizierte Position ausgegeben wird. Aus diesen Daten gilt es die wirkliche Position mit der Höhe wieder herzustellen. Die dabei zugrunde gelegten theoretischen Betrachtungen sind in [4] beschrieben. Ist die Rückrechnung gelungen, wird der fehlerhafte Null-Wert der Ballhöhe des Kameradatensatzes durch die korrekte Höhe ersetzt. Außerdem werden die auf das Spielfeld projizierten Positionen mit der errechneten, im Optimalfall realen, Position substituiert.

Für jeden Kameradatensatz wird folgendes Szenario durchlaufen: Das Modul extrahiert aus dem Datensatz der Kamera die Roboterpositionen und -ausrichtungen und die Ballposition. Da der Ball in den meisten Fällen nur von einer Kamera erfasst wird, liefert im Schnitt nur jeder zweite Kameradatensatz Daten zum Ball. Die Datensätze ohne Ball werden von diesem Modul nicht weiter betrachtet und ohne Korrektur an das Modul zur Vorhersage weitergegeben. Für Datensätze, in denen ein Ball enthalten ist wird überprüft, ob sich der Ball in der Kickerzone eines Roboters befindet. Diese Zone bezeichnet den Bereich vor dem Roboter, in welchem der Ball durch den Chip-Kicker erreicht werden kann. Jeder Flug eines Balls in einem regulären Spiel muss demnach in einer Kickerzone beginnen. Daher werden Bälle, die eine Kickerzone verlassen dahingehend untersucht, ob sie einen hohen Schuss darstellen. Der Ball befindet sich innerhalb der Kickerzone eines Roboters, wenn sein Abstand zum Roboter nicht größer als 10 cm und der Winkel zwischen der Ausrichtung des Roboters und der Strecke zwischen Robotermittelpunkt und Ballposition $\pi/8$ Rad nicht überschreitet. In diesem Fall wird der internen Instanz *Altigraph* die entsprechende Roboter- und die Ballposition mitgeteilt, woraufhin eine neue Instanz der Klasse *Flug* angelegt wird. Der Altigraph ist der Verwalter potentieller Flüge und ist somit die Schnittstelle zwischen einzelnen potentiellen Flügen und dem Ball-Corrector, welcher die Daten der SSL-Vision korrigiert.

Nachdem der Abschuss des Balls als solcher erkannt wurde, muss anhand der Bewegung des Balls detektiert werden, ob sich dieser auf dem Spielfeld befindet oder

ob er in der Luft fliegt. Dazu übergibt der BallCorrector den Ball, den der aktuelle Kameradatensatz liefert, dem *Altigraphen*. Dieser ist dafür verantwortlich, jedem der potentiellen Flüge den Ball zur Kontrolle zu übergeben. Jeder Instanz von *Flug* besteht unter anderem aus einer Flug-Startposition und -Ausrichtung. Diese Startposition entspricht der Position des Chip-Kickers des Roboters, in dessen Nähe sich der Ball in einem vorherigen Datensatz befunden hat. Bei einem fliegenden Ball entspricht die Ausrichtung des Roboters zum Schusszeitpunkt der Flugrichtung. Durch Kenntnis der Flugrichtung wird durch jede *Flug*-Instanz der auf das Spielfeld projizierte Ball auf seine Flugrichtung zurückgerechnet. Bei dieser Berechnung muss beachtet werden, von welcher der beiden Kameras der Ball aufgenommen wurde. Ist die Ballposition innerhalb gewisser Parameter des *Fluges*, kann eine Rückrechnung durchgeführt werden. Der *Flug* meldet dem *Altigraphen*, wenn der Ball, der ergänzt werden sollte, außerhalb der Parameter liegt. Dies ist beispielsweise der Fall, wenn die errechnete Ballposition außerhalb des Spielfelds liegt. Dies wiederum bedeutet, dass der jeweilige potentielle Flug höchstwahrscheinlich kein hoher Ball war und vom *Altigraph* gelöscht werden. Alle anderen potentiellen Flüge bleiben erhalten und warten auf den nächsten Kameradatensatz.

Im nächsten Schritt ermittelt der *Altigraph*, ob der Ball im Moment fliegt. Dazu nutzt er den ersten und somit ältesten *Flug* in der internen Liste. Der älteste *Flug* ist derjenige, welcher am wahrscheinlichsten ein wirklicher Flugball ist, da zu ihm die meisten Bälle gepasst haben. Wenn dieser mehr als 4 Bälle besitzt, Bedingung des Kleinste-Quadrate-Verfahrens zur Ermittlung der Regressionsparabel, erfolgt zum ersten Mal die Berechnung der möglichen Flugbahn mittels Ausgleichsrechnung.

Es wird mit der Methode der kleinsten Quadrate eine Regressionsparabel durch die Ballflugpositionen des ältesten *Fluges* gelegt. Zur effizienten Berechnung wird das dreidimensionale Problem auf ein zweidimensionales reduziert. Neben der Flughöhe erfolgt die Betrachtung der Positionen nicht mit deren absoluten Koordinaten. Es wird stattdessen der Abstand betrachtet, der zwischen der errechneten Ballpositi-

on und der Abschussposition liegt. Durch die Betrachtung der Parabel in der Form $a \cdot (x + d)^2 + e$ lassen sich direkt die erwartete Flugweite $-2d$ und die erwartete maximale Flughöhe e ablesen. Wenn die Flughöhe über einem zuvor festgelegten Grenzwert von 400 mm liegt und die erwartete Flugdistanz ebenfalls den definierten Grenzwert von 300 mm überschreitet, wird davon ausgegangen, dass der Ball fliegt. Der BallCorrector erfragt vom Altigraphen im Falle eines Fluges die reale Position und Höhe des Balls und ersetzt mit diesem Wert die ursprünglich von der SSL-Vision angenommene Position und Flughöhe. Mit der Anpassung des Frames ist die Aufgabe des BallCorrectors erledigt. Falls der Ball nicht fliegt, muss keine Korrektur vorgenommen werden und die Daten können ohne eine Anpassung direkt dem Worldpredictor übergeben werden.

Das Modul benötigt einige Parameter zur korrekten Erkennung von fliegenden Bällen. Dazu gehört ein maximal zulässiger Winkel zwischen der Ausrichtung des Roboters und der Strecke zwischen potentieller Abschussposition und Ball. Dieser Winkel ist bei einem Schuss auf dem Boden unter realen Bedingungen theoretisch Null. Durch die perspektivische Verzerrung bei einem fliegenden Ball weicht dieser Winkel von Null ab. Mit dem Parameter `START_BOT2BALL_MAX_ANGLE` kann der Grenzwinkel, also die maximal erlaubte Abweichung, sodass der Ball noch als potentieller Flugball bewertet wird, festgesetzt werden. Weitere wichtige Parameter werden in Tabelle 2.2 dargestellt. Insbesondere der Parameter `BALL2BALL_MIN_DISTANCE` ist für die Berechnung der Flugparabel wichtig. Mit ihm wird sichergestellt, dass die Positionen der Bälle aufeinander folgender Aufnahmen einige Zentimeter entfernt sein müssen. Ein möglicher Flug, der ausschließlich aus beinah deckungsgleichen Ballaufnahmen direkt vor dem Roboter, also der Zeitpunkt kurz vor dem Schuss, bestehen könnte, wird somit ausgeschlossen. Diese Flüge sind problematisch, da deren Höhe und Flugweite sehr hohe Werte annehmen können. Mit diesem Parameter wird allerdings sichergestellt, dass der Ball in Bewegung ist.

Parameter	Wert	Beschreibung
MIN_FLY_HEIGHT	300mm	Minimale erwartete Fluglänge
MIN_FLY_LENGTH	400mm	Minimale erwartete Flughöhe
START_BOT2BALL-MAX_ANGLE	$\pi/8$	Maximaler Winkel zwischen der Ausrichtung des Roboters und der Strecke zwischen potentieller Abschussposition und Ball beim Abschuss
RUN_BOT2BALL-MAX_ANGLE	$\pi/6$	Maximaler Winkel zwischen der Ausrichtung des Roboters und der Strecke zwischen potentieller Abschussposition und Ball beim Flug
BALL2BALL-MIN_DISTANCE	400mm	Minimal zulässiger Abstand zwei aufeinanderfolgender Bälle

Abbildung 2.2: Parameter zur Erkennung fliegender Bälle

2.3 Aufbau des Vorhersagemoduls

In diesem Abschnitt wird das Worldpredictor-Modul vorgestellt. Dabei wird zunächst in 2.3.1 auf den Datenfluss und die wichtigsten Klassen innerhalb des Moduls eingegangen. Dabei wird auf die allgemeinen Aufgaben der Klassen eingegangen. In Abschnitt 2.3.2 werden Implementierungsdetails vorgestellt, welche der Präzision, der Laufzeitverbesserung, numerischen Stabilität und einfacher Implementierung von Filtern und Bewegungsmodellen dienen.

2.3.1 Datenfluss und Klassen im Worldpredictor

Das Worldpredictor-Modul soll aus verrauschten Kameradaten auf Spielsituationen in der Zukunft schließen. Um dies zu ermöglichen werden zwei Threads verwendet. Der erste Thread wartet auf `CamFrames` und speichert diese in einem Puffer, der zweite Thread dient als arbeitender Thread. Dieser entnimmt aus dem Puffer `CamFrames`, wertet die Informationen aus, führt die jeweiligen probabilistischen Ver-

fahren durch und erstellt aus den gewonnenen Informationen einen `WorldFrame`. Eine weitere Aufgabe des zweiten Threads ist es, regelmäßig auf neue und aus dem Spiel genommene Objekten zu reagieren.
Basierend auf diesem Überblick lassen werden im Folgenden die verwendeten Klassen mit ihren Aufgaben aufführen.

Oracle_ExtKal Die `Oracle_ExtKal`-Klasse ist die Hauptklasse des Moduls. Sie implementiert die Schnittstellen zum Cam-Modul und zur KI. Weiterhin reagiert sie auf *Moduli*-Kommandos zum Starten bzw. Stoppen des Moduls. Dabei wird der Arbeiterthread gestartet bzw. ihm ein Terminierungssignal geschickt. Die `Oracle_ExtKal`-Klasse selbst bildet den ersten beschriebenen Thread mit der Aufgabe `CamFrames` anzunehmen.

TrackingManager Die Klasse `TrackingManager` organisiert das Hinzufügen und Entfernen von Robotern und Bällen auf das Spielfeld. Dafür werden vier Listen eingesetzt. Eine Liste enthält Informationen über bislang unbekannte eigene Roboter auf dem Spielfeld. Befindet sich in dieser Liste genügend Beobachtungen eines neuen Roboters um sicherzustellen das der Roboter wirklich auf dem Spielfeld zu finden ist und es sich um keine falsch positive Erkennung handelt, wird die Information über den Roboter in die Liste der bekannten eigenen Roboter eingepflegt. Diese Liste beinhaltet alle eigenen Roboter für die Vorhersagen erzeugt werden sollen. Inhalt dieser Liste sind Implementierungen des `IFilter`-Interfaces, welche auf Instanzen des `TigersMotionModel` zurückgreifen. Diese Instanzen werden beim Übergang zwischen den Listen initialisiert. Gab es für ein Element der zu verfolgenden eigenen Roboter zu lange keine neuen Positionsdaten durch die Kameras, wird dieses aus der Liste gelöscht. Analog dazu werden gegnerische Roboter in den verbleibenden zwei Listen gehalten. Für diese wird jedoch das `FoodMotionModel` in der Liste der zu verfolgenden gegnerischen Roboter genutzt. Für die Datenhaltung des Balls

existieren keine Listen da immer nur ein Ball im Spiel ist. Das Tracking mehrerer Bälle ist zwar möglich, wurde aus Performance-Gründen jedoch nicht umgesetzt. Für die Filterung der Balldaten wird das `BallMotionModel` eingesetzt. Der Ablauf des `TrackingManagers` ist in Abbildung 2.3 dargestellt.

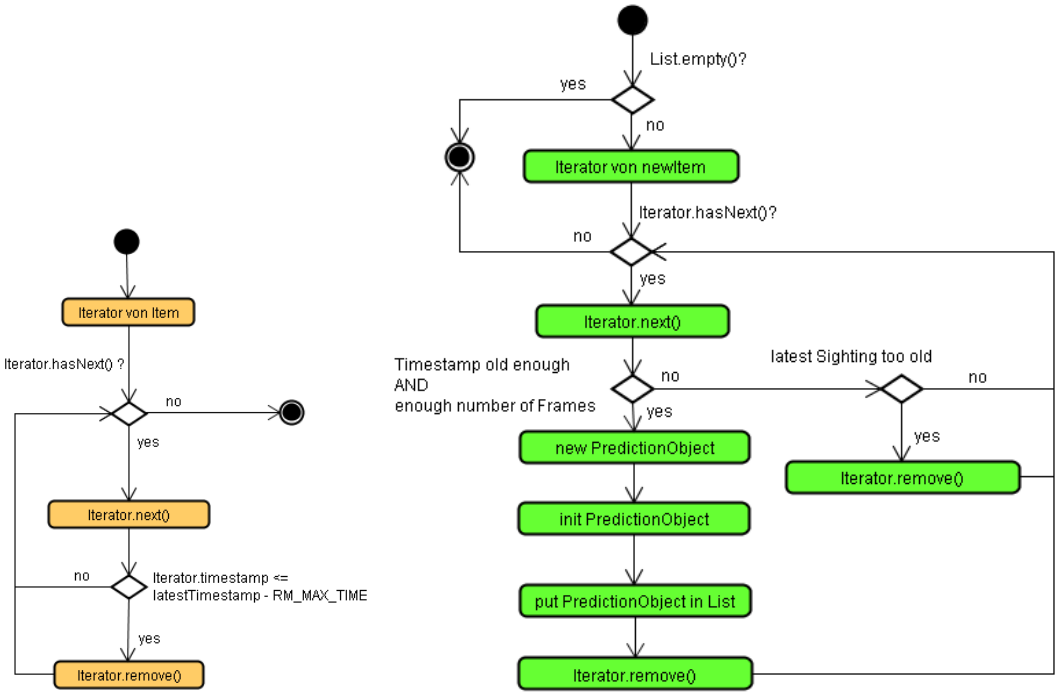


Abbildung 2.3: TrackingManager – grün: Überprüfen der Listen mit neuen Objekten; beige: Überprüfen der Listen mit bereits verarbeitenden Objekten

SyncedCamFrameBuffer Die Klasse `SyncedCamFrameBuffer` stellt den oben erwähnten Puffer bereit. Zudem enthält die Klasse zusätzliche Logik zum parallelen Prozessieren mehrerer `CamFrames`, der Verhinderung der Bearbeitung alter `CamFrames` und der Vermeidung von Aufstauungen von zu vielen unbearbeiteten `CamFrames`. Diese Klasse und die Auswirkungen durch die interne Logik werden in 2.3.2 beschrieben.

Director Der `Director` ist die Implementierung des arbeitenden Threads im Worldpredictor. Dieser wird mit Modulstart gestartet und erhält ein Terminierungssignal wenn das Modul gestoppt wird. Die Aufgabe dieses Threads ist es in einer Schleife Informationen aus dem `SyncedCamFrameBuffer` abzufragen. Ein eingehender `CamFrame` wird vom `Director` genutzt um `Ball-` und `BotProzessor` anzuweisen, die jeweiligen probabilistischen Verfahren durchzuführen. Dadurch wird Rauschen gefiltert und ein zukünftiger Zustand der Objekte auf dem Spielfeld berechnet. Dieser wird an den `WorldFramePacker` übergeben, welcher die Daten in einen von der KI interpretierbaren Container, den `WorldFrame` speichert. Dieser `WorldFrame` wird von der `Director`-Klasse über die `Oracle_ExtKal`-Klasse an die eingetragenen Beobachter (siehe Kapitel 2.1.3) weitergeleitet. Im letzten Schritt wird aus der `Director`-Klasse der `TrackingManager` gestartet. Dieser Schritt wird nur nach Überschreiten einer konfigurierten Minimalzeit seit dem letzten Aufruf durchgeführt.

Prediction Context Eine Instanz der Klasse `PredictionContext` wird im Modul verwendet um die Listen der neuen und bekannten Objekte, sowie der Informationen über den Ball zu halten. Diese Instanz wird von fast allen anderen Klassen des Moduls referenziert.

Ball- und Botprozessor Die Klassen `BallProzessor` und `BotProzessor` nehmen Informationen aus den `CamFrames` entgegen und ordnen diese den jeweiligen bekannten oder neuen Objekten in `PredictionContext` zu. Neue Objekte werden in die betreffenden Listen eingepflegt. Bei bekannten Objekten werden die zugehörigen Daten als Beobachtung dem jeweiligen Filter zur Verfügung gestellt. Weiterhin bieten die Klassen die Funktionalität, die Vorhersagen zu erstellen. Diese werden mit Hilfe der Filter und der darin referenzierten Bewegungsmodelle erstellt.

IMotionModel Das Interface `IMotionModel` stellt die Basis für alle Bewegungsmodelle dar. Es bietet Methoden um aus einem Status einen zukünftigen Folgestatus

zu berechnen. Weiterhin existieren Methoden um modellabhängige Größen in die Filter eingehen zu lassen. Dazu zählen u.a. Jacobi-Matrizen und Kovarianzen. Speziell für das Partikelfilter existieren zusätzlich Methoden zur Generierung von Samples. Implementierungen dieses Interfaces sind die Klassen
`TigersMotionModel` für die Bewegung eines omnidirektionalen Roboters mit Ansteuerungsinformationen, `FoodMotionModel` für gegnerische omnidirektionale Roboter zu denen keine Ansteuerungsinformationen zur Verfügung stehen und
`BallMotionModel`, welches die meist gradlinigen Bewegungen des Balls modelliert.

IFilter Das Interface `IFilter` definiert die Schnittstelle zu den Implementierungen der verschieden Filter. In diesem sind Methoden definiert um Korrekturschritte und Prädiktionen durchzuführen. Die Filter werden mit einem Bewegungsmodell (`IMotionModel`), welches das zugrunde liegende System modelliert, initialisiert. Implementierungen dieses Interfaces sind die Klassen `ExtKalmanFilter` und `ParticleFilter`.
Aus den einzelnen Klassen lässt sich das Gesamtsystem zusammensetzten. Dieses Gesamtsystem des Moduls ist vereinfacht in Abbildung 2.4 zu sehen. Dabei werden die konkreten Implementierungen von Filtern und Bewegungsmodellen nicht dargestellt. Diese werden in Kapitel 2.4 näher erläutert.

2.3.2 Konzepte

Bei der Umsetzung des Moduls wurden unterschiedliche Konzepte verwendet um Stabilität, Wiederverwendbarkeit, Austauschbarkeit und Genauigkeit zu ermöglichen. Im Folgenden werden diese beschrieben. Dabei wird zuerst ein Konzept zur optimierten Verarbeitung von `CamFrames` vorgestellt. Zur Verringerung von Abhängigkeiten zwischen den Laufzeiten von SSL-Vision und *Sumatra* wird darauf folgend die Nutzung von Kamerazeitstempeln beschrieben. Ein weiteres Konzept nimmt eine Anpassung der intern verwendeten Einheiten vor, um einen möglichst guten Ausgangs-

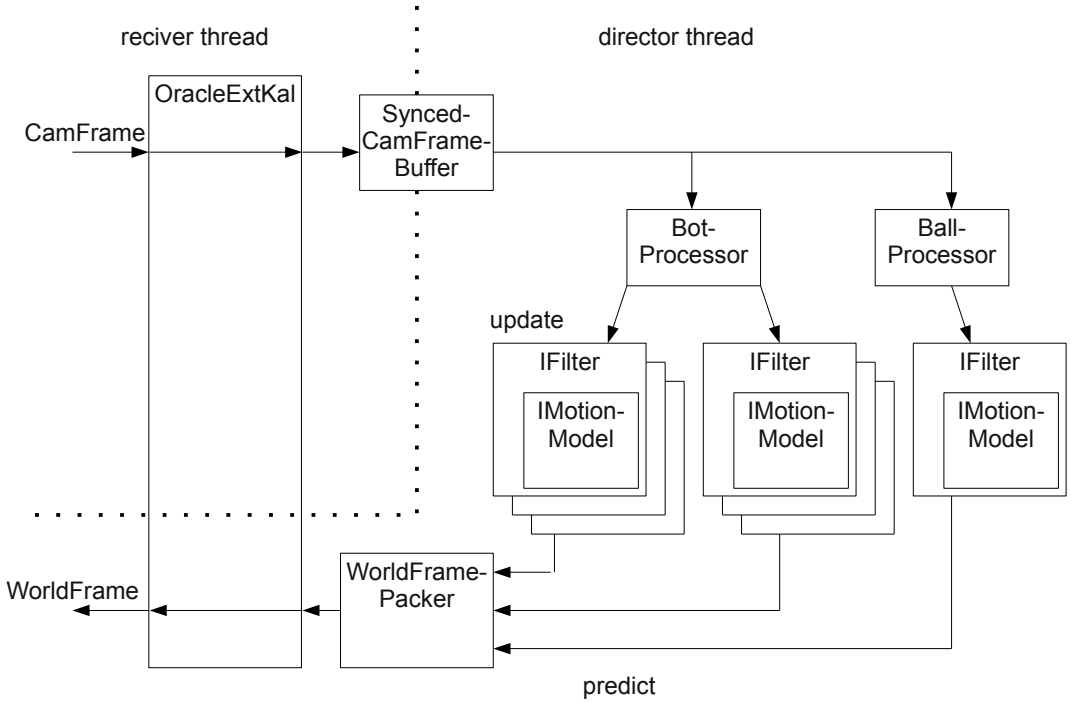


Abbildung 2.4: Datenfluss im Worldpredictor-Modul

punkt für die numerische Stabilität der Filterverfahren zu generieren. Das nächste Konzept beschreibt kurz die Umsetzung von austauschbaren Bewegungsmodellen und Filterverfahren. Das letzte Konzept beschreibt die Kollisionserkennung und dafür erforderliche schrittweise Prädiktion.

Optimierte Verarbeitung von Kameradaten

In der RoboCup Small Size League wird das Spielgeschehen mit Hilfe von zwei über dem Spielfeld befindlichen Kameras aufgezeichnet. Die Daten zu den jeweiligen Spielfeldhälften werden zuerst von der SSL-Vision Software verarbeitet und schließlich an die Teams weitergereicht. Innerhalb dieser Verarbeitungsschritte kann es zu verschiedenen Verzögerungen kommen, die ein asynchrones Empfangen der Kameradaten verursachen können. Somit ist nicht gewährleistet, dass zum Zeitpunkt der Verarbeitung die Daten zu beiden Spielfeldhälften vorhanden oder in der richtigen Reihenfolge sind. Aus diesem Grund ist es wichtig, ankommende Daten zu überprüfen und mit eventuell vorhandenen Kameradaten der anderen Spielfeldhälfte zu kombinieren. Für diese Thematik steht der `SyncedCamFrameBuffer` zur Verfügung, der im Folgenden näher beschrieben wird.
Dafür wird zunächst kurz die Funktionsweise des Cam-Merging zusammengefasst: Sobald der erste `CamFrame` vorhanden ist, wird dieser verarbeitet. Am Ende wird überprüft, ob währenddessen neue Daten von einer anderen Kamera im Puffer liegen. Ist dies der Fall, werden diese Daten ebenfalls verarbeitet und aus dem Puffer entfernt. Dabei werden die Vorhersagen der Objekte des vorherigen Frames auf den jetzigen angeglichen. Diese Schritte wiederholen sich so lange, bis entweder keine weiteren Kameradaten vorhanden sind oder die Daten von allen Kameras bereits einmal verarbeitet wurden. Im Anschluss folgt das Packen der `WorldFrames`. Umgesetzt wurde `SyncedCamFrameBuffer` mit Hilfe eines Arrays für die einkommenden `CamFrame`-Daten, eines weiteren zur Speicherung des Status der jeweiligen Kameradaten, sowie einer Warteschlange, um die Reihenfolge der Abarbeitung festzuhalten.

Sobald ein Frame in den Puffer geschrieben werden soll, wird der Zeitstempel geprüft. Ist dieser zu alt, werden die Daten verworfen. Anschließend werden im Array der `CamFrames` die neuen Daten am entsprechenden Index aktualisiert und der Status der Kamera neu gesetzt.
Jede Kamera besitzt vier Status:

NOTSEEN ist der Status, wenn noch keine Daten während eines Verarbeitungsdurchlaufs von dieser Kamera im Puffer lagen

QUEUED wird einer Kamera zugewiesen, wenn sie in die Warteschlange aufgenommen wurde

PROCESSED ist der Status einer Kamera, wenn ihre Daten verarbeitet wurden

PROCESSAGAIN wird einer Kamera zugewiesen, wenn sie vorher im Status *PROCESSED* war und währenddessen erneut empfangen wurden. Das heißt, dass sie für die nächste Verarbeitungskette als *QUEUED* vorgemerkt wird.

Sind die Status korrekt gesetzt, wird auch die Warteschlange aktualisiert. Bei einem neuen Eintrag wird die Kamera-ID ans Ende der Schlange geschrieben. Eine Kamera, die bereits in der Warteschlange ist und von der neue Daten vorliegen, wird zurück an das Ende der Warteschlange gesetzt. Daten von einer Kamera, die bereits verarbeitet wurden, werden nicht in die Warteschlange aufgenommen. Wird auf den `SyncedCamFrameBuffer` zugegriffen, wird die in der Warteschlange ganz vorne stehende ID genutzt, um auf das Array der `CamFrames` zuzugreifen und abschließend der entsprechende Eintrag aus der Warteschlange gelöscht. Hier wird auch der Status auf *PROCESSED* gesetzt.
Um zu überprüfen, ob nach den Korrekturschritten der Filter ein weiterer Frame zur Verfügung steht, wird lediglich durch die Status der Kameras iteriert. Sobald eine Kamera mit dem Status *QUEUED* vorhanden ist, kann der Prozessierungsvorgang mit den neuen Daten wiederholt werden.

Nachdem schließlich alle Kameradaten verarbeitet wurden, wird abschließend die Liste der Daten zurückgesetzt. Das heißt, die Warteschlange wird geleert, alle Kameras mit den Status *PROCESSAGAIN* werden auf *QUEUED* gesetzt und sofort in die Warteschlange aufgenommen und die restlichen Daten auf *NOTSEEN* gesetzt.

Nutzung von Kamerazeitstempeln

Seit 2010 kommt die standardisierte Software SSL-Vision in der Small Size League zum Einsatz. Diese wertet Kameraaufnahmen des Spielfelds aus und versendet über eine Ethernet-Verbindung mittels UDP-Protokoll Positionsinformationen. Dabei werden bei versendeten Paketen die Zeitstempel der Kameraaufnahmen mitgesendet. Diese geben weit mehr Aufschluss über die tatsächlich vorherrschende Zeit als andere verfügbare Zeitstempel wie der Absendezeitpunkt aus SSL-Vision oder die Ankunftszeit in Sumatra. Dies liegt darin begründet, dass der Zeitstempel zum Absendezeitpunkt einen nicht konstanten Fehler basierend auf der Bearbeitungszeit von SSL-Vision enthält. Bei der Ankunftszeit in *Sumatra* addiert sich zu diesem noch ein weiterer nicht konstanter Fehler basierend auf der nicht deterministischen Übertragung von Paketen über UDP und Ethernet.
Diese Fehler lassen sich umgehen indem nur Kamerazeitstempel ins System einfließen. Da letztendlich innerhalb des Moduls nur Zeitdifferenzen betrachtet werden, ist dies problemlos möglich.

Interne Einheiten

Die Rahmenbedingungen für die Filter und die Bewegungsmodelle sind nicht optimal. Das größte Problem bilden die Zeitstempel. Diese sind in *Sumatra* in Nanosekunden seit Systemstart angegeben. Rein mathematisch ist es für die Filter und die Bewegungsmodelle egal in welcher Einheit die Daten zur Verfügung stehen. Numerisch kann dies allerdings schnell zu Problemen führen. Zudem sind Bezugseinheiten nicht konsistent. So werden Ansteuerungsdaten von Robotern in $\frac{m}{s}$ übergeben. Po-

sitionsdaten haben die Einheit mm. Zeitstempel sind in ns gegeben. Dies heißt bei einer direkten Verwendung, dass vor allem in den Bewegungsmodellen viele Umrechnungen stattfinden müssen und somit unnötige Komplexität einfließt.
Um diese Komplexität zu vermeiden und numerisch stabilere Berechnungen durchzuführen werden innerhalb des Moduls eigene Einheiten verwendet. Basierend auf Umrechnungsfaktoren welche in der Modul-Konfigurationsdatei festgehalten sind werden Zeiten und Längen konvertiert. Die Zeiten werden zudem so verschoben, dass die erste eingehende Messung zum Zeitpunkt Null stattfindet.
Statt die große Menge verschiedener Einheiten innerhalb der Filter oder der Bewegungsmodelle zu kennen, werden beim Zugriff auf `CamFrames` oder Ansteuerungen alle Daten in die internen Einheiten konvertiert. Somit stehen den Filtern und Bewegungsmodellen nur Daten aus den Einheiten τ (interne Zeit) und ι (interne Strecke) und aus diesen zusammengesetzte Einheiten wie $\frac{\iota}{\tau}$ für Geschwindigkeiten und $\frac{\iota}{\tau^2}$ für Beschleunigungen zur Verfügung. Ausnahmen bilden nur Winkel und deren zeitlichen Ableitungen. Diese sind definiert als Rad, $\frac{Rad}{\tau}$, $\frac{Rad}{\tau^2}$.
Vor der Ausgabe von Daten aus dem Modul zu weiteren Modulen werden die Daten gemäß der in den Interfaces vorgeschriebenen Einheiten konvertiert. Der Prozess der Umwandlung ist von außen nicht erkennbar, bietet aber intern die Vorteile, dass keine Betrachtungen der Einheiten in Filtern und Bewegungsmodellen nötig sind und per Konfigurationsdatei die Einheiten nach numerischen Gesichtspunkten skaliert werden können.

Trennung von Filter und Bewegungsmodell

Bei einer statischen Implementierung von Filterverfahren zur Rauschunterdrückung und Prädiktion von Bewegungen ist in der Filterklasse fest definiert welche Dimensionen Statusvektoren und Messungsvektoren besitzen. Weiterhin ist das Bewegungsmodell innerhalb der Methoden der Filterklasse implementiert und notwendige Angaben wie beispielsweise Varianzen sind als lokale Variablen verfügbar.

Diese Implementierung hat allerdings viele Nachteile. Zum Beispiel ist es nicht möglich das Bewegungsmodell, welches das Filter verwendet, einfach auszutauschen oder für andere Filter zu verwenden. Des Weiteren stellt ein Filter einen Algorithmus da, der einen Statusvektor und einen Messvektor benötigt, aber nicht festschreibt in welcher Dimension diese vorliegen müssen. Diese sind definiert durch das modellierte System und lassen sich vom Filter abkapseln. Auch Angaben über Varianzen sind keine Eigenschaften von Filtern sondern des Systems.
Zur Umgehung der oben genannten Abhängigkeiten wird in dem Modul eine strikte Trennung zwischen der Implementierung eines Filters und der eines Bewegungsmodells durchgeführt.
Die Implementierung eines Filters, die dem Interface `IFilter` folgt, definiert nur den Filteralgorithmus. Damit ein Filter angewendet werden kann wird diesem ein Bewegungsmodell, definiert über das Interface `IMotionModel`, übergeben. Dieses Bewegungsmodell hat als direkte Aufgabe die Modellierung der Bewegung. Zudem enthält das Bewegungsmodell Informationen über Varianzen und andere Systemparameter. Weiterhin kann es auch Messungen zu Messvektoren konvertieren und aus Statusvektoren Ausgabeobjekte erzeugen.
Die Filter können arbeiten ohne Wissen über das System oder die Art der modellierten Bewegung zu haben. So lassen sich Filter und Bewegungsmodelle beliebig zusammen verwenden.

Kollisionserkennung und schrittweise Prädiktion

Für die Erkennung von Kollisionen wird nicht eine Prädiktion pro Objekt vorgenommen, stattdessen werden schrittweise Vorhersagen erzeugt. Pro Prädiktionsschritt können Objekte auf Überlappung überprüft werden. Bei einer Überlappung kann eine Behandlung der Kollision erfolgen. Dieser Ansatz wurde bisher nur in Teilen umgesetzt. Die schrittweise Prädiktion ist bereits implementiert. Die Schnittstelle um auf Kollisionen zu reagieren wurde aber bisher noch nicht implementiert.

2.4 Filterverfahren

In diesem Abschnitt werden die verschiedenen Filterverfahren vorgestellt. Diese Kapitel basiert im Wesentlichen auf [4]. Die Bewegungsmodelle für Roboter und Bälle wurden übernommen und an das Interface `IMotionModel` angepasst. Dafür wurden weitere Systemparameter mit dem Bewegungsmodell zusammengelegt. Die Filter verwenden diese Bewegungsmodelle. Hier wird näher auf deren Implementierung eingegangen. Im ersten Teil 2.4.1 wird das Extended Kalman Filter erläutert. Der zweite Teil 2.4.2 beschäftigt sich mit der Implementierung des Partikelfilters.

2.4.1 Extended Kalman Filter

Das Extended Kalman Filter ist ein probabilistisches Verfahren, dessen Funktionsweise in [4] beschrieben wurde. In diesem Kapitel wird die grundlegende Implementierung des Filters beschrieben und ein Verfahren zur Erkennung von falsch positiven Messungen vorgestellt.

Ablauf Der Ablauf des Extended Kalman Filters besteht aus den Schritten „Vorhersage“ und „Korrektur“. Im Vorhersageschritt wird aus dem aktuellem Zustand eine Vorhersage eines zukünftigen Zustands generiert. Der Korrekturschritt dient der Einbindung von Messungen in das Filter. Durch die Vorhersage steigt die Ungewissheit des prädizierten Zustands, durch die Korrektur nimmt diese ab.
Hier vorgestellt wird die Klasse `AExtendedKalmanFilter`. Diese beinhaltet die für das Extended Kalman Filter benötigten Funktionen. Innerhalb des Moduls wird eine ähnliche Klasse verwendet, welche die gleiche Funktionalität bietet und dem `IFilter` Interface folgt. Die Funktionen f, G, R, h, H und Q werden durch ein Bewegungsmodell (`IMotionModel`) bereitgestellt.

```
1  public abstract class AExtendedKalmanFilter {
2    private Matrix  xs; //state vector
3    private Matrix  Ps; //covariance matrix
4
5    protected abstract  Matrix  f(double dt, Matrix x); //noiseless dynamics
```

```
6     protected abstract  Matrix  G(double dt);           //jacobian of f w.r.t. state
7     protected abstract  Matrix  R(double dt);           //covar. of dynamics noise
8     protected abstract  Matrix  h(Matrix x);            //noiseless measurement
9     protected abstract  Matrix  H();                    //jacobian of h w.r.t. state
10    protected abstract  Matrix  Q();                    //covar. of measurement noise
11
12    protected AExtendedKalmanFilter(Matrix x, Matrix P) {
13      xs = x;
14      Ps = P;
15    }
16
17    public Matrix predict_state(double dt, Matrix x)  {
18      return f(dt, x);
19    }
20
21    public Matrix predict_covariance(double dt, Matrix P) {
22      Matrix G = G(dt);
23      Matrix R = R(dt);
24      return G.times(covar).times(G.transpose()).plus(R);
25    }
26
27    public void update(double dt, Matrix z) {
28      Matrix pred_x = predict_state(dt, xs);
29      Matrix pred_P = predict_covariancex(dt, xs, Ps);
30      Matrix H      = H();
31      Matrix Q      = Q();
32
33      //calculate kalman gain
34      Matrix K = pred_P.times(H.transpose()).
35                  times(H.times(pred_P).times(H.transpose()).
36                        plus(Q)).inverse());
37      //correct state
38      xs = pred_x.plus(K.times(z.minus(h(pred_x))));
39      //correct covariance
40      int dim = pred_P.getRowDimension();
41      Ps = (Matrix.identity(dim, dim).minus(K.times(H))).times(pred_P);
42    }
43  }
```

Listing 2.1: Beispielimplementierung eines Extended Kalman Filters

Der aktuelle Systemzustand wird in Form des Zustandsvektors `xs` und der Kovarianzmatrix `Ps` als `private` Klassenvariablen gespeichert. Das objektorientierte Prinzip der Kapselung ist hier besonders wichtig, da der Systemzustand ausschließlich durch das Filter bestimmt werden soll. Jeglicher Einfluss von Außen ist hier unerwünscht und wird durch den `privaten` Status verhindert.

Alle vom Filter benötigten Matrizen, d.h. die Übergangs- und Messmatrix, die zuge-

ordneten Kovarianzmatrizen und die benötigten Jacobimatrizen, sind abhängig vom zu modellierenden Filterverhalten, d.h. z.B. der abzubildenden Roboterbewegung. Daher werden diese Matrizen in der konkreten Implementierung vom Bewegungsmodell, teils in Abhängigkeit von der letzten Messung oder der letzten Ansteuerung, angefordert.

Der Konstruktor erwartet zur korrekten Initialisierung einen initialen Zustandsvektor `x` und eine initiale Kovarianzmatrix `P`. Die konkrete Implementierung erwartet hier das Bewegungsmodell und eine Messung. Mit Hilfe des Bewegungsmodells wird aus der Messung ein initialer Zustandsvektor gebildet. Auch wird die initiale Kovarianzmatrix vom Bewegungsmodell in Abhängigkeit des Zustandes gebildet.

Im Anschluss werden die Methoden `predict_state` und `predict_covariance` implementiert, durch welche der Vorhersageschritt des Extended Kalman Filters abgebildet wird. Die erste Methode setzt die Vorhersage des Zustands um und erwartet als Parameter einen Zeitunterschied `dt` und einen Ausgangszustand `x`. Dieser Zustand wird dann mit Hilfe der abstrakten Methode `f`, durch welche das linearisierte Bewegungsmodell abgebildet wird, in den `dt` Zeitschritte späteren Zeitpunkt transformiert. Ähnlich funktioniert die Methode `predict_covariance`. Hier wird die übergebenen Kovarianzmatrix `P` in den `dt` Zeitschritte späteren Zeitpunkt prädiziert. Dies erfolgt mithilfe der Systemmatrix `A` und der Kovarianzmatrix des Systemrauschens `R`. Alle hierfür benötigten Matrizen werden in der konkreten Implementierung durch das Bewegungsmodell bereitgestellt.

Die letzte Methode, `update`, implementiert den Korrekturschritt des Extended Kalman Filters. Zunächst wird der aktuelle Zustand durch Aufruf der Vorhersage-Methoden in den Zeitpunkt der Messung prädiziert. Danach werden die benötigten Systemmatrizen mit Hilfe der zugehörigen abstrakten Methoden geladen. Im Anschluss erfolgt die Berechnung des Kalman-Faktors `K`, welcher bestimmt, wie stark die Messung in den korrigierten Zustand eingeht. Mit diesem wird zunächst der Zustandsvektor `xs` und dann die Kovarianzmatrix `Ps` des Systemzustands korrigiert.

Somit wird in der abstrakten Klasse `AExtendedKalmanFilter` der gesamte Algorithmus des Extended Kalman Filters implementiert. Die praktische Anwendung des Filters ist durch Implementierung der Funktionen f, G, R, h, H und Q, welche vom Bewegungsmodell bereitgestellt werden, gegeben.

Erkennung falsch positiver Messungen Bei falsch positiven Messungen handelt es sich um inkorrekte Messungen, welche mit einer hohen Gewissheit aufgenommen werden. Sie stellen eine Form von nicht-normalverteiltem weißen Rauschen dar und verschlechtern daher die Resultate eines Extended Kalman Filters, da dieses allein von normalverteiltem Rauschen ausgeht. Deshalb werden solche falsch positiven Messungen mit Hilfe des in [5] beschriebenen Ansatzes erkannt und für das Update des Extended Kalman Filters gesperrt. Zur Entscheidung, ob eine Messung abgelehnt oder akzeptiert werden soll, muss zunächst die bedingte Wahrscheinlichkeit $P\left(z_k|\overline{\mu}_k, \overline{\Sigma}_k\right)$ berechnet werden. Durch diese wird die Wahrscheinlichkeit bestimmt, die Messung z_k im (vorhergesagten) Zustand $\overline{\mu}_k$ mit der Kovarianzmatrix $\overline{\Sigma}_k$ zu erhalten. Die Wahrscheinlichkeitsfunktion ist dabei eine Normalverteilung (im n-dimensionalen Raum des Zustandsvektors) mit dem Erwartungswert des Ausgangszustands $\overline{\mu}_k$ und den Kovarianzen von $\overline{\Sigma}_k$. Beide Komponenten wurden vorher durch die Matrix C_k in den Messungsraum transformiert. Gleichung (2.1) fasst diesen Sachverhalt zusammen.

$$P\left(z_k|\overline{\mu}_k, \overline{\Sigma}_k\right) = \frac{1}{\left(2\pi|C_k\overline{\Sigma}_k C_k^T + R|\right)^{\frac{1}{n}}} \exp\left(-\frac{1}{2}\left(z - C_k\overline{\mu}_k\right)^T C_k^{-1}\left(z - C_k\overline{\mu}_k\right)\right) \tag{2.1}$$

Durch Festlegung eines Mindestwertes kann nun einfach entschieden werden, ab wann Messungen als zu unwahrscheinlich angelehnt werden. Der Vorteil des Extended Kalman Filters, von der Vorhersage abweichende Messungen weniger stark im Update-Schritt zu berücksichtigen, wie zur Vorhersage passende Werte, wird

dadurch nicht beschnitten. Es wird lediglich diese Robustheit verbessert. Da durch übersprungene Messungen die Kovarianzen bei der Vorhersage immer stärker ansteigen, ist zudem sichergestellt, dass andauernde richtige Messungen, welche weit von der Vorhersage entfernt liegen, schließlich übernommen werden. In einem solchen Fall wird der Zustand durch die großen Kovarianzen sofort auf die neue Messung springen. Das vorgestellte Verfahren zeichnet sich durch seine hohe numerische Effektivität und Robustheit bei der Entfernung falsch positiver Messungen aus ohne auf weitere Filterverfahren zurückgreifen zu müssen. (vgl. [5])

2.4.2 Partikelfilter

Das Partikelfilter ist ein weiteres probabilistischen Verfahren, welches im Worldpredictor implementiert wurde. Grundlagen zu diesem Verfahren wurden in [4] erläutert. Dieses Kapitel befasst sich mit der Implementierung und beschreibt die verwendeten Verfahren zur Gewichtung und zum resampeln der Partikel. Weiterhin wird in diesem Kapitel kurz die Wahl des verwendeten Zufallszahlgenerators begründet.

Ablauf des Partikelfilters Der grundsätzliche Ablauf einer Iteration setzt sich wie folgt zusammen:

1. Entnahme von Partikelproben anhand der Vorhersage und der Messung
2. Gewichtung der Partikel anhand der Messungswahrscheinlichkeit
3. Normierung der Gewichtung
4. Berechnung des effektiven Samplingfaktor
5. Resampling bei Bedarf

Im ersten Schritt werden Proben genommen. Diese erhält das Partikelfilter über das Bewegungsmodell. Dazu benötigt das Modell einen Zustandsvektor und eine Messung, die über die Filterschnittstelle bereitgestellt wird. Da Messungen nicht in

regelmäßigen Abständen stattfinden, muss das Partikel zum Zeitpunkt der Messung vom letzten vorhergesagten Zustand aus berechnet werden. Die Vorhersage erfolgt ebenfalls über das Bewegungsmodell. Hier wird dem Bewegungsmodell die Informationen über den Zustandsvektor und die Ansteuerungsdaten mitgegeben.
Im Anschluss folgt die Gewichtung. Dazu werden die Gewichte des vorherigen Zustandes mit der aktuellen Messungswahrscheinlichkeit multipliziert. Die Messungswahrscheinlichkeit liefert wieder das Bewegungsmodell. Diesem werden der Zustandsvektor, die Messung und die Zeitdifferenz seit der letzten Messung übergeben. Jedes der berechneten Gewichte für die einzelnen Partikel wird aufsummiert, sodass im Weiteren jedes Gewicht normiert werden kann.
Danach wird der effektive Samplingfaktor berechnet. Dieser erlaubt es dann mittels eines Schwellwertes zu entscheiden, ob ein Resampling stattfinden sollte, oder nicht.

Resampling Das Partikelfilter lässt mehrere Möglichkeiten zu, das Resampeln zu vollziehen. Die erste Variante ist ein auf Zufall basiertes Ziehen. Hierbei wird das Gewicht, welches auf eins normiert ist, mit einer uniform verteilten Zufallszahl verglichen. Ist das Gewicht des Partikels größer, so kommt es für den Resampling-Algorithmus weiter in Frage. Ist es jedoch kleiner, verschwindet es aus der Partikelmenge. Aus der ursprünglichen Menge werden solange Partikel gezogen, bis die neuen Partikel eine neue Menge der gleichen Größe ergeben. Hierbei kann es vorkommen, dass Partikel mit hoher Gewichtung mehrfach vorkommen.
Eine andere Methode ist die Gruppenauswahl. Diese soll anhand des in Listing 2.2 aufgeführten Quellcodes näher erläutert werden. In [4] wurden bereits die Grundlagen der Gruppenauswahl beschrieben. Listing 2.2 zeigt auf, wie dieses Verfahren in Java umgesetzt wurde. Hierbei wurde das Array `rsCount` und die Variable `weightCumulative`, die die Summe in Abhängigkeit des Schleifendurchgangs aufsummiert, angelegt. Das Array dient zur Speicherung, wie oft das k-te Partikel nach dem Resampling noch vorhanden sein soll. Dementsprechend erfolgt die Abfrage zum

```
1  int k = 0;
2  double weightCumulative = particle[0].weight;
3
4  for (int i = 0; i < nParticle; i++)
5    rsCount[i] = 0;
6  int cntPartikel = 0;
7  while (j < nParticle)
8  {
9    while (weightCumulative > (((double) j) / nParticle) && j < nParticle)
10   {
11     rsCount[k]++;
12     cntPartikel++;
13     j++;
14   }
15   k++;
16   if (k == nParticle || j == nParticle)
17   {
18     rsCount[k - 1] = nParticle - cntPartikel + 1;
19     break;
20   }
21   weightCumulative += particle[k].weight;
22 }
```

Listing 2.2: Stratified Resampling

Hochzählen des k-ten Elements über die die aufsummierte Summe und $\frac{j}{N}$, wobei `j` der Zähler für den Schleifendurchgang ist und `N` die Anzahl der Partikel beinhaltet. Nachdem also mit diesem Algorithmus festgelegt wurde, welche Partikel und wie viele durch das Resampling weiterhin verwendet werden, folgt ein Neuverteilen der Partikel auf das Array, sodass weiterhin wie gewohnt auf diese zugegriffen werden können. Abbildung 2.5 zeigt an Hand eines gewählten Beispiels, wie dieser Algorithmus funktioniert. Dabei sind 5 Partikel gegeben und wie folgt belegt:

Partikelindex	0	1	2	3	4
Gewichtung	0,1	0,4	0,3	0,05	0,15

In der linken Tabelle wird das Beispiel auf den in Listing 2.2 angegeben Code angewendet. In der rechten Tabelle wird anhand des Arrays `rsCount` das Array `rsIndices` belegt. Die Inidizes dieses Arrays geben die Indizes der resampleten Partikel an, während die Werte des Arrays die alten Indizes angeben. Der passende Quellcode hierzu wird unter Listing 2.3 angegeben.

```
1  j=0;
2  for (int i = 0; i < nParticle; ++i) {
3    if (rsCount[i] > 0) {
4      rsIndices[i] = i;
5
6      while (rsCount[i] > 1) {
7        while (rsCount[j] > 0)
8            ++j; // find next free spot
9        rsIndices[j++] = i; // assign index
10       --rsCount[i]; // decrement number of remaining offsprings
11     }
12   }
13 }
```

Listing 2.3: Neuanordnung der Partikel im Array

		rsCount					weightCumulative
j	k	0	0	0	0	0	0,1
1	0	0	0	0	0	0	0,5
1	1	0	1	0	0	0	
2	1	0	2	0	0	0	
2	1	0	2	0	0	0	
3	2	0	2	1	0	0	0,8
4	2	0	2	2	0	0	
4	2	0	2	2	0	0	0,85
4	3	0	2	2	0	0	1
5	4	0	2	2	0	1	

j - Anzahl der vergebenen Partikel
k - Index des zu untersuchenden Partikel

i	j	rsCount					rsIndices				
		0	2	2	0	1	0	0	0	0	0
0	0	0	2	2	0	1	0	0	0	0	0
0	0	0	2	2	0	1	0	0	0	0	0
1	0	0	1	2	0	1	0	1	0	0	0
1	1	0	0	2	0	1	1	1	0	0	0
2	2	0	0	1	0	1	1	1	2	0	0
2	3	0	0	0	0	1	1	1	2	2	0
3	4	0	0	0	0	1	1	1	2	2	0
4	4	0	0	0	0	0	1	1	2	2	4

i - Index des zu untersuchenden Partikel
j - Index, an dem Partikel eingefügt werden soll

Abbildung 2.5: Resampling-Algorithmus anhand eines Beispiels

Random Generator Das Partikelfilter stellt hohe Anforderungen an einen Zufallszahlengenerator. Zum einen wird das Partikelfilter in einer Echtzeitanwendung eingesetzt und muss dementsprechend so schnell wie möglich Zugriff auf Zufallszahlen haben. Zum anderen werden innerhalb einer Iteration mehrere Zufallszahlen benötigt. Da eine Verarbeitung der Kameraframes bis zu 120 mal in einer Sekunde stattfinden kann, muss der Zufallszahlengenerator eine hohe Periodendauer haben. In [9] wurde sich mit den Anforderungen an Zufallsgeneratoren ausführlich auseinandergesetzt und verschiedene Aspekte betrachtet. Anhand von [13] kann man sehen, dass dies nicht für die Java-Klasse Random gegeben ist. Auch die SecurityRandom-Klasse erfüllt nicht die Anforderungen, da sie nicht performant genug ist. Stattdessen wird die Java-Bibliothek, die in [6] vorgestellt wird, verwendet.

2.5 Matrizen

Zur Umsetzung der Filter werden reelle Matrizen benötigt. Für die Programmiersprache Java, welche in diesem Projekt genutzt wird, existieren bereits vorgefertigte Matrix-Bibliotheken. Die zu Beginn des Projekts eingesetzte Bibliothek *Jama* [7] erwies sich als nicht ausreichend performant. Andere performante Bibliotheken [3], wie *UJMP* [2] waren für die Verwendung nicht ausreichend dokumentiert. Daher entstand die Notwendigkeit zur Entwicklung einer eigenen Matrix-Klasse mit hohen Performance-Ansprüchen.
Sie muss numerisch stabil sein und zudem schneller arbeiten als andere Bibliotheken. Der minimale Funktionsumfang ist durch die mit ihr zu implementierenden Verfahren vorgegeben. Es handelt sich um Addition, Subtraktion und Multiplikation von Matrizen. Des Weiteren wird die Funktionalität des Transponierens benötigt. Die aufwendigste der benötigten Funktionen ist die Bildung der Inversen einer Matrix. Eine geschickte Möglichkeit dies zu umgehen, ist die Ermittlung der Inversen einer Matrix anhand der Lösung eines Gleichungssystems. In den folgenden Abschnitten

wird erläutert, welche Besonderheiten die Implementierung der eigenen Matrixklasse besitzt.

Nach [11] ist für die Performance von Software sinnvoll, komplexe Datenstrukturen zu vermeiden. Insbesondere sind zweidimensionale Arrays in der Programmiersprache Java langsamer als eindimensionale. Daher erfolgt die Speicherung der Daten in eindimensionalen Arrays und nicht, wie beispielsweise bei der Bibliothek *Jama*, in zweidimensionalen Arrays. Bei der Implementierung des Zugriffs auf das eindimensionale Arrays ist zu beachten, dass die eigentliche Zeilennummer zunächst mit der Spaltenanzahl der Matrix multipliziert werden muss. Auf diesen Wert wird anschließend die gewünschte Spaltennummer addiert. Das Ergebnis liefert den Index des Elements, auf das zugegriffen werden soll. Der Zugriff auf die komplette Matrix wird im Allgemeinen mittels zwei ineinander verschachtelter for-Schleifen durchgeführt. Die äußere Schleife sorgt für den Zeilenwechsel und die innere iteriert über die Einträge in den Spalten. Um bei der Iteration durch eine Zeile, gleichbleibender Zeilenindex und veränderter Spaltenindex nicht ständig neu das Produkt von Zeilennummer und Spaltenanzahl der Matrix zu bilden, wird dieser Wert zwischen der ersten und der zweiten For-Schleife berechnet und in einer Variablen gespeichert. In der inneren for-Schleife für jeden Zugriff lediglich eine Addition nötig.

Bei manchen Operationen, wie Addition, Subtraktion oder Transposition von Matrizen, ist es möglich, die Ausgangsmatrix zur Speicherung des Ergebnisses zu nutzen. Dadurch muss kein zusätzlicher Speicherplatz vom System angefordert werden. Um diesen Sachverhalt nutzen zu können, muss sichergestellt werden, dass die Ausgangsmatrix nicht für weitere Operationen benötigt wird. Dies liegt in der Verantwortung des Programmierers, der die Matrizen einsetzt. Um festzulegen, ob die Werte der Ausgangsmatrix überschrieben werden dürfen oder nicht, muss ein Flag des Typs *boolean* gesetzt werden. Setzt der Programmierer das Flag auf `true` erlaubt er explizit, die erste der Funktion übergebene Matrix mit dem Ergebnis der Berechnungen zu überschreiben. Das Setzen auf `false` verbietet das Überschrieben der alten Ma-

trix. Setzt der Programmierer das Flag nicht, wird davon ausgegangen, dass die Matrix nicht überschrieben werden darf und automatisch das Flag auf `false` gesetzt.

Beim Anlegen von Matrizen kann ebenfalls eine Optimierung des Speicherbedarfs erfolgen. Gibt ein beliebiger Algorithmus ein eindimensionales, zeilenweise abgelegtes Array zurück, welches speziell zur Erzeugung einer Matrix erstellt wurde, kann der Speicher direkt genutzt werden. Sollen vom gleichen Array mehrere Matrizen erzeugt werden oder wird auf dem Array nach der Erstellung der Matrix weiter gearbeitet, ist es notwendig, neuen Speicher für die Matrix vom System anzufordern und die Daten des alten Arrays in das Neue zu kopieren. Der Programmierer entscheidet wiederum mit Hilfe eines Flags, ob der Speicher genutzt werden darf oder nicht. Dieses Vorgehen ist nur bei eindimensionalen Arrays, die dem Konstruktor übergeben werden, möglich. Zweidimensionale Arrays müssen aufgrund der internen Datenhaltung zunächst in die eindimensionale Form gebracht werden. Daher ist in diesem Fall das Anfordern von neuem Speicher nicht zu umgehen.

Eine bei der Nutzung von Matrizen sehr aufwendige Funktion ist die Invertierung. Eine mathematisch korrekte Möglichkeit ist Berechnung mit Hilfe von Unterdeterminanten und Adjunkten [10], wobei dieses Verfahren in der Praxis eher ungeeignet ist. Da es sich bei den für das Extended Kalman Filter zu invertierenden Matrizen immer mindestens um positiv semi definite Matrizen handelt, kann auf die alternative Invertierung mittels Cholesky zurückgegriffen werden. Um jedoch auch jede beliebige invertierbare Matrix mit der entwickelten Matrixklasse invertieren zu können, wurde eine allgemeinere Form der Invertierung implementiert. Somit werden dem Programmierer von der Matrixklasse zwei verschiedene Methoden zum Invertieren zur Verfügung gestellt. Die erste Methode heißt `inverse()`. Für quadratische Matrizen M der Dimensionen $m < 5$ ist die Bestimmung der Inversen für jedes Einzelelement $m_{j,k}$ (mit j als Zeilen- und k als Spaltenindex) explizit angegeben. Ist die Matrix größer, wird die Inverse mittels LR-Zerlegung ermittelt. Für den Fall, dass

dem Programmierer allerdings bekannt ist, dass seine Matrizen mindestens symmetrisch positiv semi definit sind, kann die zweite Methode zur Invertierung genutzt werden. Diese besitzt den Namen `inverseByCholesky()` und ist für Matrizen mit den eben genannten Eigenschaften effektiver.

2.6 Auswertungskomponenten

Um die Software effizient und ausgiebig testen zu können, hilft es nicht, sich nur auf Augenmaß zu verlassen. Es müssen genaue und jederzeit an- bzw. abschaltbare Auswertungstools vorhanden sein, damit die Güte der Daten zuverlässig überprüft werden kann, gleichzeitig aber auch die Performance der Software im Einsatz nicht beschränkt wird. Zu diesem Zweck wurden verschiedene Auswertungskomponenten implementiert, deren Umsetzung in diesem Kapitel näher beschrieben wird.

2.6.1 Aufnahme und Wiedergabe von CamFrames

Das Aufnehmen und Abspielen von durch *Sumatra* empfangenen Kameradaten ist wichtig für die experimentelle Erprobung unterschiedlicher Module. Durch diese Komponente kann ein Szenario aufgenommen und im Anschluss zur Evaluierung unterschiedlicher Konfigurationen des Gesamtsystems verwendet werden. Aus diesem Grund spielt sie u.a. eine wichtige Rolle zur Auswertung der Filterverfahren und derer Leistungsfähigkeit.
Die Implementierung dieser Komponente wurde nicht direkt im Worldpredictor-Modul vorgenommen, sondern das Cam-Modul erweitert, da nur hier die Kameradaten direkt und unverarbeitet vorliegen. Um Kameradaten erfolgreich aufzunehmen, werden sie beim Empfangen in *Sumatra* in ein serialisierbares Objekt[2] konvertiert. Über einen Stream werden sie schließlich regelmäßig in einer Datei persistent ge-

[2]Serializable ist ein Interface in Java, welches Klassen ermöglicht, ihre Daten und Referenzen durch Nutzung der writeObject()-Methode in Dateien zu schreiben. Näheres hierzu findet sich in [14] Kapitel 14.12

```
1  try
2  {
3    object = (SSLVisionData) in.readObject();
4  } catch (ClassNotFoundException err)
5  {
6    err.printStackTrace();
7  }
8  arg0.setData(object.getData());
9
10 ThreadUtil.parkNanosSafe(object.getTimestamp() - oldTimestamp);
```

Listing 2.4: Lesen der Kameradaten und Abwarten eines genauen Zeitintervalls

speichert. Damit können Kameradaten aufgenommen werden.

Zur Wiedergabe von aufgenommenen Daten wurde eine Klasse entwickelt, die das Interface `IReceiver` implementiert. Durch die Interface-Methode `receive` wird festgelegt, welche Daten empfangen werden. Bei Nutzung dieser Klasse wird sichergestellt, dass die vorher aufgenommenen Kameradaten verwendet werden und nicht die von Simulator oder SSL-Vision übertragenen Daten.

Neben den empfangenen Daten werden auch die Zeitdifferenzen beim Empfang der Kameradaten gespeichert. Dies ermöglicht, dass die Daten beim Einlesen in den gleichen Zeitabständen zum Worldpredictor gelangen wie zum Zeitpunkt der Aufnahme. Listing 2.4 zeigt, wie der ausführenden Thread genau für das angegebene Zeitintervall in den Schlafzustand gebracht wird.

Wie an diesem Code weiterhin ersichtlich, können Daten über die `readObject()` Methode eingelesen werden. Hierbei muss das gelesene Objekt als das ursprünglich geschriebene Objekt gecastet werden. Ein dynamisches Typecasting ist dabei nicht möglich.

Mit Hilfe dieser Komponente ist es somit möglich, Spielverläufe aufzunehmen und abzuspielen. Es ist jedoch nicht möglich, Roboteransteuerungen aufzunehmen, da diese nicht durch das Cam-Modul bearbeitet werden.

2.6.2 Schreiben von Vorhersagedaten in eine Datei

Um Vorhersagedaten dauerhaft festzuhalten, soll es die Möglichkeit geben, formatierte Daten der Vorhersage in eine Datei zu schreiben. Im Nachhinein können diese Daten mit einer Visualisierungssoftware wie *gnuplot* (vgl. [15]) verbildlicht und analysiert werden. Für das Aufnehmen der Daten wurde die Klasse `Precision` implementiert. Sie benutzt die Daten der Vorhersageobjekte unmittelbar bevor sie mit den neuen Kameradaten aktualisiert werden. Zum Vergleich der gefilterten und vorhergesagten Zustände, greift `Precision` direkt auf die Kameradaten des vorherigen Schritts zurück. Da die Übersicht bei der Darstellung mehrerer Objekte verloren geht, werden lediglich die Daten eines Objekts geschrieben. Welches Objekt und in welche Datei geschrieben wird ist frei konfigurierbar.
Zur internen Speicherung der Daten des beobachteten Objektes in der `Precision` Klasse, existieren zu den Robotern und Bällen jeweils Funktionen für die Behandlung von Kameradaten und Vorhersageobjekten. Diese beiden Funktionen speichern die entsprechenden Daten in ein Array mit einer festgelegten Größe. Diese Größe ist standardmäßig auf 10000 festgelegt. Sobald das Array gefüllt ist oder das Modul beendet wird, werden die Daten am Stück in die Datei geschrieben. Somit soll einerseits ein ständiger Datenfluss auf die Festplatte vermieden werden, da diese I/O-Befehle sehr zeitaufwändig sind. Andererseits wäre ein Schreiben erst nach Beenden des Moduls unvorteilhaft, da so das Array zu einer unbegrenzten Größe anwächst und den Speicher unnötig belastet.
Bevor die Daten überhaupt im Array gespeichert werden, werden verschiedene Abfragen gestartet. Zum einen muss das Objekt die richtige ID haben. Für den Ball ist das die ID Null, für die Roboter der Tigers Mannheim eine Zahl von 100 bis 199 und für die gegnerischen Roboter eine Zahl von 200 bis 299. Da Daten von mehreren Kameras kommen, wird auch der Zeitstempel des vorherigen Kameradatums geprüft. Sind diese beiden identisch, wird auf ein Speichern der Daten verzichtet.
Beim Hinausschreiben werden die verschiedenen Arrays der einzelnen Objektinfor-

mationen zusammengeführt. Ein Datensatz besteht aus den Kameradaten und den Daten der N Vorhersagen. Nachfolgend ist die Reihenfolge in der Datei aufgeführt:

Id, Vorhersage0(Zeitstempel, Position(x,y), Rotation), Vorhersage1(Zeitstempel, Position(x,y), Rotation), ..., VorhersageN(Zeitstempel, Position(x,y), Rotation)

Die einzelnen Daten werden mit Leerzeichen voneinander getrennt, einzelne Datensätze mit einem Zeilenumbruch. Das ermöglicht ein einfaches Interpretieren der Daten von Visualisierungsprogrammen.

2.6.3 Echzeitvisualisierung in der Sumatra-GUI

In der grafischen Benutzeroberfläche von *Sumatra* wurde ein Panel für beliebige Worldpredictor-Implementierungen angelegt. Dieses soll in Echtzeit die Vorhersagen des Worldpredictor-Moduls mit den von SSL-Vision kommenden Daten abgleichen. Dazu folgt es, wie alle Elemente der grafischen Oberfläche, dem MVP-Entwurfsmuster (Modell View Presenter-Pattern). Im Falle des Worldpredictor-Panels heißt das, dass eine Klasse geschrieben wurde die nur für die Anzeige der grafischen Oberfläche zuständig ist. Diese enthält bis zu 7 Diagramme für absolute Positionen, Winkel und Fehlerabschätzungen. Des Weiteren sind Elemente vorhanden um die Frequenz eingehender Kameradaten und ausgehender `WorldFrames` zu vergleichen. Neben den Anzeigeelementen sind verschiedene Bedienelemente vorhanden um beispielsweise das zu beobachtende Objekt auszuwählen. Abbildung 2.6 zeigt eine Aufnahme des beschriebenen Panels in der grafischen Benutzeroberfläche von *Sumatra*.

Die Logik, also was passiert wenn ein Bedienelement ein Signal sendet oder wie die Daten auf den Anzeigeelementen dargestellt werden sollen, ist im zum Worldpredictor-Panel gehörenden *Presenter* untergebracht. Dieser greift als ein *Observer* auf das Cam- und das Worldpredictor-Modul zu, um einen Vergleich der Kameradaten zu den Prädiktionen durchzuführen.

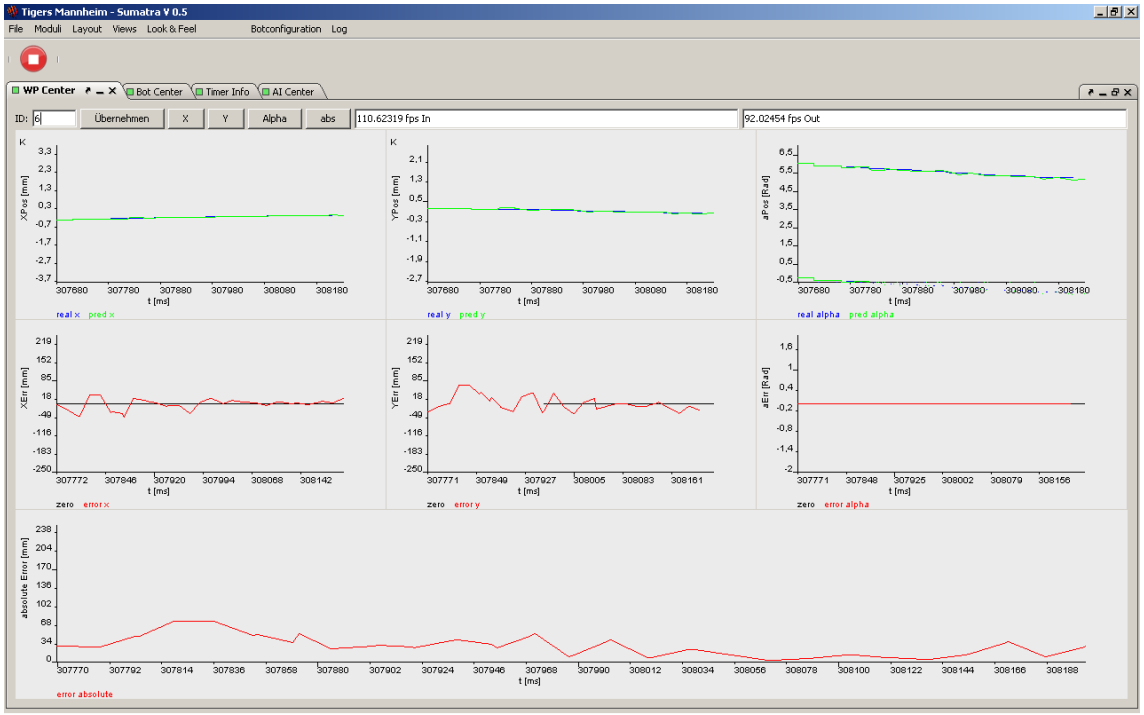


Abbildung 2.6: Worldpredictor-GUI in *Sumatra*

Die *View* der grafischen Benutzeroberfläche des Worldpredictors ist in Form einer Klasse implementiert, welche von `JPanel`[3] abgeleitet wurde. Auf diesem werden verschiedene Diagramme angezeigt. Diese Tragen die Position, oder Fehlerwerte gegenüber der Zeit ab.

Die Diagramme zur Position und zum Winkel werden direkt mit den Daten aus den `CamFrames` (eingehende Daten) und den `WorldFrames` (ausgehende Daten) gefüllt. Allgemein werden auf diesen Diagrammen zwei Graphen angezeigt. Ein Graph für die Daten aus den `Cam-Frames` und ein Graph für die Daten aus den Vorhersagen. Dabei wird die x-Position (die Zeit) direkt aus Zeitstempeln der Frames abgelesen. Daraus folgt, das im Vergleich zur aktuellen Zeit ein eingehender `CamFrame` weiter links auf der Zeitachse eingetragen wird (Aufnahmezeit des Bildes) und der daraus generierte `WorldFrame` rechts der aktuellen Zeit eingetragen wird (Zeitstempel der

[3]aus *Swing* [8], das für die grafische Benutzeroberfläche von *Sumatra* genutzte GUI-Toolkit

Zeit für die die Voraussage gelten soll). Dem gegenüber wird die Position oder die Ausrichtung abgetragen.
Für die Graphen in den Fehlerabschätzungen werden eingehende `CamFrames` mit vorherigen `WorldFrames` verglichen. Um dies zu ermöglichen, wird eine Liste an Referenzen auf `WorldFrames` zwischengespeichert. Bei einem eingehenden `CamFrame` kann über den Zeitstempel entschieden werden, welche Vorhersagen um diese Zeit herum getroffen wurden. Aus den dazugehörigen `WorldFrames` wird durch lineare Interpolation ein Fehler zur Position aus den `CamFrames` bestimmt.

Anmerkungen Da die grafische Benutzeroberfläche unabhängig von der Implementierung des Worldpredictors sein soll ist es nicht möglich, auf die rauschunterdrückten Werte des Filters (d.h. noch ohne Vorhersage) zuzugreifen. So wird das Rauschen der `CamFrames` direkt an die Positions- und Fehlergraphen weitergegeben. Auch ist es nicht möglich einen `CamFrame` direkt mit einem `WorldFrame` zu vergleichen da zu einem Zeitstempel eines `WorldFrames` kein `CamFrame` eingehen muss.

Kapitel 3

Experimentelle Erprobung

In diesem Kapitel wird die vorgestellte Implementierung der verschiedenen Filterverfahren erprobt. Hierzu wurden diverse Testszenarien durchgeführt, die in den einzelnen Abschnitten näher erläutert werden. Im Anschluss daran folgt eine ausführliche Beschreibung der Ergebnisse und welche Schlussfolgerungen daraus gezogen werden können.

Um für verschiedene Anforderungen (z.B. Verarbeitung einmal mit Kalman-Filter, ein anderes Mal mit Partikelfilter) auf den gleichen Datensätzen zu arbeiten, wurde die in Kapitel 2.6.1 vorgestellte Komponente zur Aufnahme von Kameradaten in *Sumatra* verwendet. Weiterhin wird die Auswertungskomponente aus Kapitel 2.6.2 verwendet, um die Daten für anschließende Visualisierungen zu speichern.

Zur Auswertung der Daten folgt neben der grafischen Darstellung auch eine statistische Gegenüberstellung. Für diese werden Prädiktionen realen und gefilterten Kameradaten gegenübergestellt. Prädiziert wird dabei $50ms$ in die Zukunft. Dieser Wert ergab sich aus Messungen der gesamten Systemlatenz. Bei der Gegenüberstellung wird der euklidische Abstand zwischen Vorhersage und linearer Interpolation der nicht prädizierten Daten betrachtet. Dieser Sachverhalt ist in Abbildung 3.1 dargestellt.

Hierbei werden die Datensätze anhand folgender statistischer Messzahlen zu den

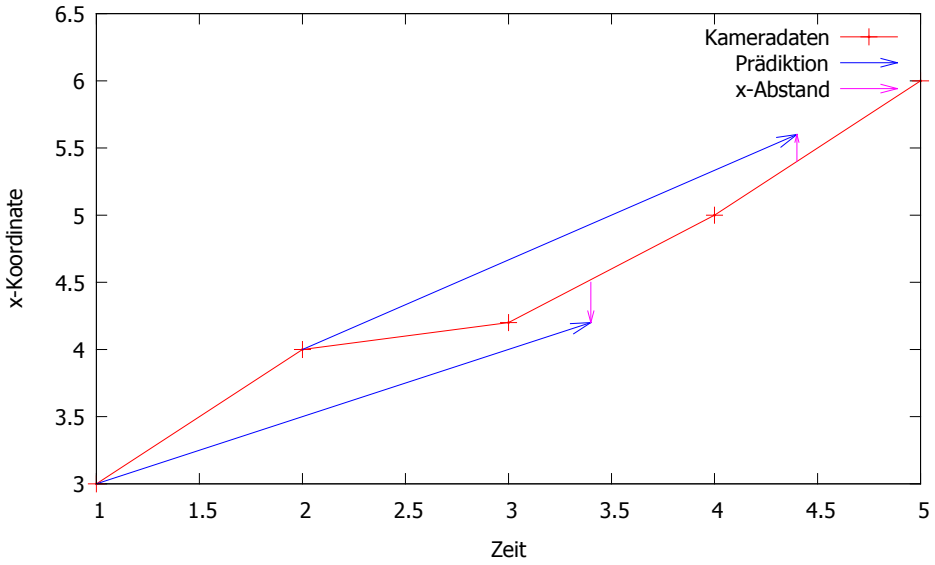


Abbildung 3.1: Basis der Validierungsrechnung

euklidischen Abständen gekennzeichnet:

- $\varnothing$ – arithmetisches Mittel
- m – Median
- min – minimaler Abstand
- max – maximaler Abstand
- var – Varianz
- σ – Standardabweichung

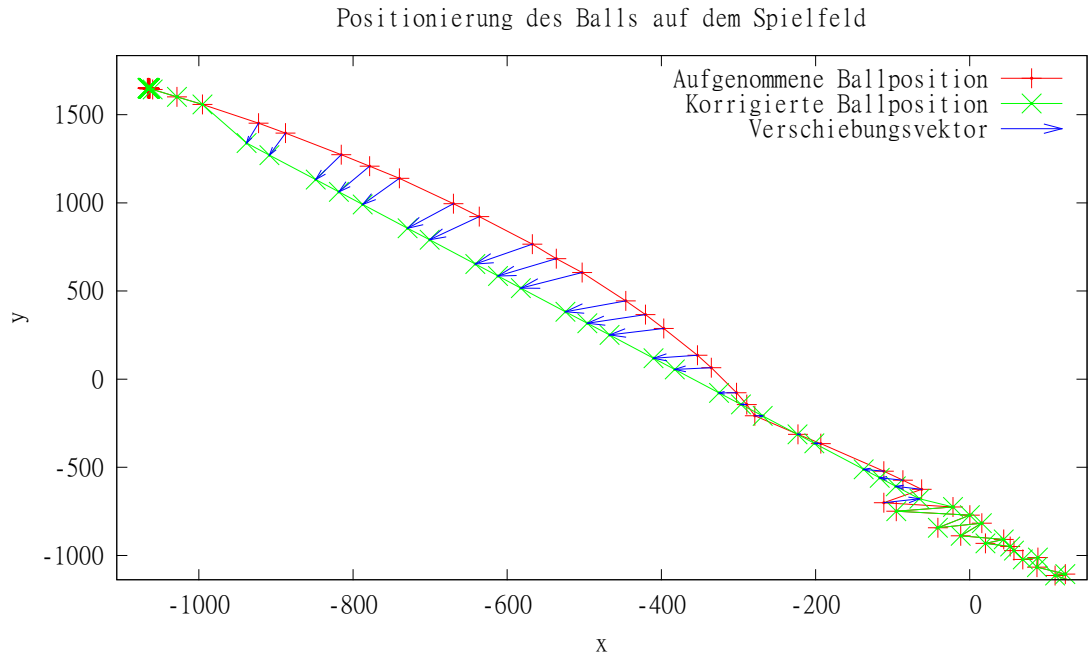
Um reale Daten aufzunehmen wurde die Testumgebung so gestaltet, dass einflussnehmende Faktoren weitestgehend ausgeschlossen sind. Reale Daten wurden auf einem $4,7m \times 3,54m$ großen Spielfeld aufgenommen. Über diesem hängen in $2,9m$ Höhe zwei Kameras der Art AVT Marlin F-046C. Für eine annähernd gleichmäßige Beleuchtung sorgen vier 1000 Watt starke Scheinwerfer. Die Bildverarbeitungssoft-

ware SSL-Vision ist hinreichend kalibriert und wertet die Kameradaten aus. Der Anschluss des Vision-Servers an den *Sumatra*-Rechner erfolgt durch Ethernet.
Für die folgenden Betrachtungen sind alle Längen in mm, alle Zeiten in s und alle Winkel in *Rad* gegeben sofern keine Einheit explizit angegeben ist.

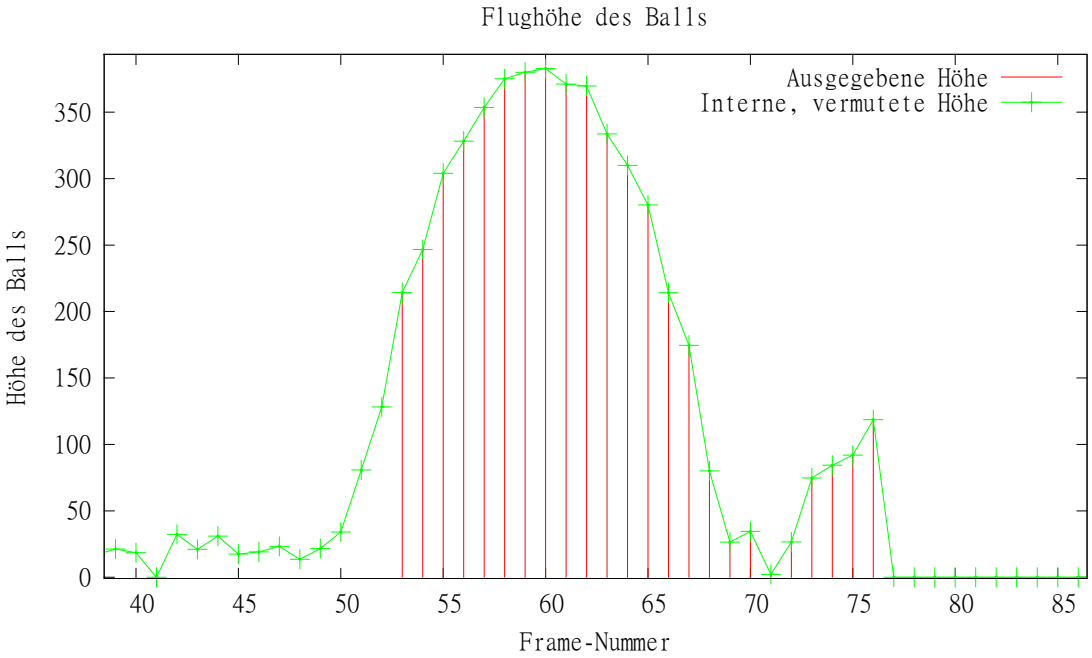
3.1 Detektion fliegender Bälle

Anhand verschiedener Testszenarien wurde die korrekte Funktionsweise des Moduls zur Erkennung fliegender Bälle validiert. Während der Entwicklung wurde mit JUnit-Testfällen kontrolliert, dass die aufgerufenen Methoden wie gewünscht arbeiten. Die in Abschnitt 2.2 erwähnten Parameter wurden in Realtests heuristisch bestimmt. Diese Tests deckten umfangreiche Szenarien ab. Die Wichtigsten werden nachfolgend erläutert.
Der grundlegende Test ist das Bewegen eines Balls auf dem Boden. In diesem Fall soll der Ball als nicht fliegend erkannt werden und daher auch keine Korrektur vorgenommen werden. Wenn der Ball in der Kickerzone eines Roboters liegt, wird versucht, eine Flugbahn zu berechnen. Da diese allerdings nicht innerhalb der vorgegebenen Parameter (vgl. Kapitel 2.2) liegt, wird der Flug sofort verworfen.
Bei beiden nachfolgend beschriebenen Testszenarien befinden sich die Kameras an den Positionen $[-1136; -99]$ bzw. $[1175; -82]$[1]
Die nächste Teststufe ist das Erkennen eines hohen Balls bei einem Schuss auf einer Spielfeldhälfte. Somit ist gewährleistet, dass die Daten nur von einer Kamera kommen. Ein solches Szenario zeigt die Abbildung 3.2. Das Bild 3.2a zeigt die Position des Balls auf dem Spielfeld. Der von der Kamera aufgezeichnete Weg des Balls ist in rot gekennzeichnet. Die Daten, die dieses Modul dem Worldpredictor-Modul übergibt, sind in grün dargestellt. Existiert eine Abweichung zwischen diesen beiden Datensätzen wird die Positionsänderung mit einem blauen Pfeil gekennzeichnet. Die

[1] [x;y] Positionsangaben in mm. Der Ursprung des Koordinatensystems liegt in der Mitte des Spielfelds. Die y-Achse bildet die Mittellinie, die x-Achse verbindet demnach beide Tore.



(a) Position des Balls auf dem Spielfeld



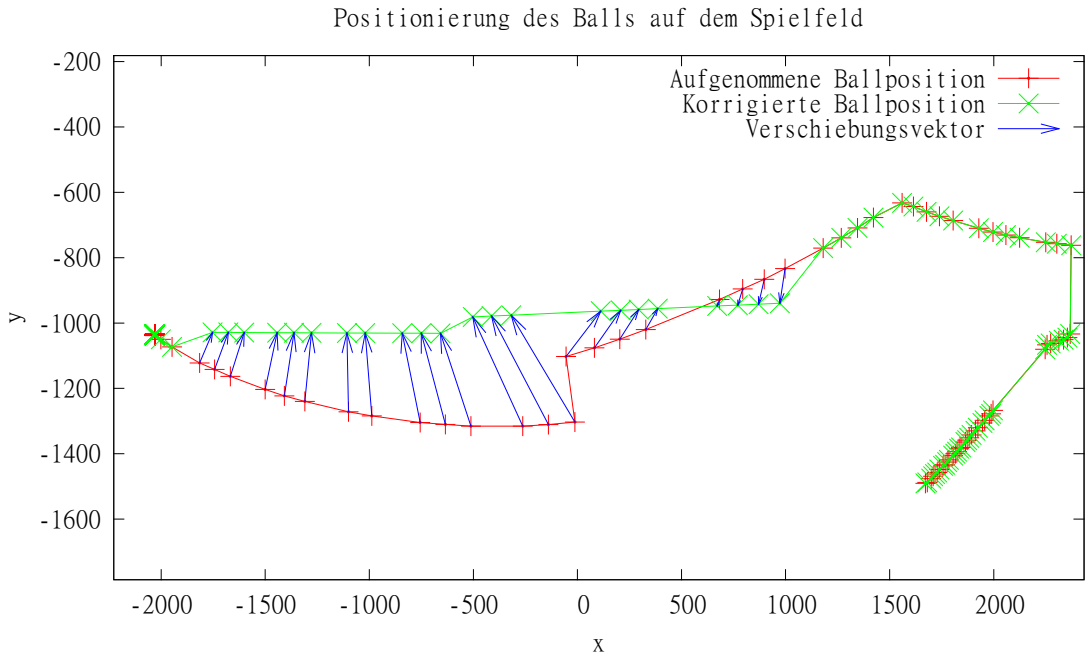
(b) Flughöhe des Balls

Abbildung 3.2: Position und Höhe eines fliegenden Balls (über eine Spielfeldhälfte)

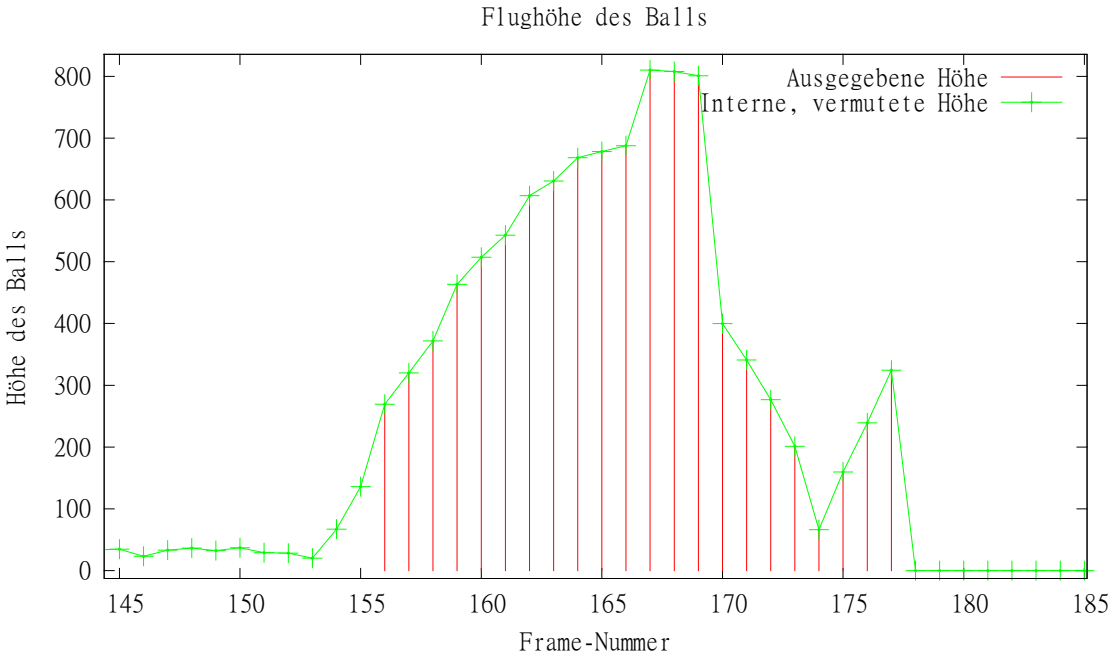
Abschussposition befindet sich bei ca. [-1130; 1650]. Von dort aus ist die Flugparabel, die von der Kamera wahrgenommen wird, gut zu sehen. Bereits nach 4 Messungen hat das Modul den Flug erkannt und rechnet die Position des Balls zurück. Eine Zurückrechnung der Ballposition (blauer Pfeil) bedeutet auch, dass die Höhe des Balls geändert wird. Dies ist in Abbildung 3.2*b* zu sehen. Die grüne Kurve zeigt, die interne, vermutete Ballhöhe an. Diese dargestellte Höhe entspricht der Höhe, die für die letzte ankommende Messung im am längsten vorhandenen Flug errechnet wurde. Das wiederum bedeutet, dass modulintern bereits eine Höhe ermittelt und vermutet wird. Eine Vermutung wird bereits erstellt, wenn zur Berechnung der Flugparabel und daran anschließender Validierung der Flugparameter anhand der vorgegebenen Grenzwerte noch keine ausreichende Anzahl von Messungen vorhanden ist. Dies erklärt das scheinbare Rauschen des Höhenwertes in der Abbildung 3.2*b* vor dem Flug. Mit einer ausreichenden Anzahl von Messungen und einem innerhalb der Parameter liegenden Flug werden die Höhendaten des Balls von Null auf den rot dargestellten Wert korrigiert. Jeder rote Balken entspricht somit zeitlich einem blauen Positions-Korrekturvektor. Der Flug gilt als beendet, wenn der von der Abschussposition zurückgelegte Weg der Strecke zwischen den Schnittpunkten der errechneten Flugparabel mit der Abszissenachse überschreitet. Im Optimalfall entspricht dies der Landung des Balls auf dem Boden, was bei den Aufnahmen der Daten von nur einer Kamera zutrifft. Im Anschluss an den ersten Flug sind weitere kleine Flugansätze zu sehen, welche jedoch schnell abgebrochen werden. Wie in Kapitel 2.2 erläutert, werden unter Umständen mehrere Flüge in einer List angelegt. Wenn ein Flug beendet wird, rücken die anderen nach. Somit sind nach Beenden des einen Fluges noch weitere Flüge mit ähnlichen Parametern wie der erste Flug in der Liste. Diese können unter Umständen noch zulässig sein, wenn der erste Flug bereits beendet ist. Sie werden aber, ebenso wie der Hauptflug, nun relativ schnell beendet. Die Höhe des Balls beträgt, wenn sich kein passender Flug in der Liste befindet, nach dem letzten Flug wieder Null. Das Rauschen der Ballpositionen im unteren

rechten Bildteil der Abbildung 3.3*a* ist durch die Aufnahme eines Balls von zwei verschiedenen Kameras aus begründet. Dieser Überlappungsbereich stellt eine große Hürde bei der Erkennung von Flugbällen dar, da in Testläufen unter Umständen Abweichungen der aufgenommenen Ballpositionen von bis zu 20 cm bei einen auf dem Spielfeld ruhenden Ball gemessen wurden.

Das zweite wichtige Testszenario ist der Flug eines Balls von einer Spielfeldhälfte in die andere. Dabei wird der Überlappungsbereich der Kameras durchquert, in dem es zu großen Abweichungen bezüglich der gemessenen Ballpositionen kommen kann. Abbildung 3.3 zeigt einen solchen Flug über zwei Spielfeldhälften. Er beginnt bei ca. [-2000;-1050]. Die Darstellung erfolgt in gleicher Art und Weise, wie beim vorherigen Szenario. Der visualisierte Flug besteht aus drei einzelnen Flügen. Der erste Flug ist der längste. Er liegt vollständig auf der linken Spielfeldhälfte. Die ersten 11 Korrekturen werden vom ersten Flug durchgeführt. Der zweite Flug besitzt eine leicht geänderte Abschussrichtung. Anhand dieses Fluges werden die nächsten drei Korrekturen vorgenommen. Die letzten acht Korrekturen erfolgen auch mit diesem Flug, allerdings stammen die Ballpositionsdaten von der Kamera der rechten Spielfeldhälfte, was an dem Knick in der projizierten Flugbahn (rot dargestellte Ballpositionen mit Verbindung) ersichtlich ist. Mit dem Beenden dieses Fluges ist der Gesamtflug beendet, da kein weiterer Flug mit passenden Parametern mehr in der Liste vorhanden ist. Bei ungünstigen Umständen kann die veränderte Wahrnehmung aufgrund des Kamerawechsels auch zum Abbruch der Flugerkennung führen. Problematisch ist die Erkennung, wenn der Ball direkt unter der Kamera entlang fliegt. In diesem Fall werden die Ballpositionen zwar ebenso auf das Spielfeld projiziert, beschreiben aber keine gekrümmte Flugbahn mehr. Vielmehr wird aus Kamerasicht eine Häufung von Bällen an Abschuss- und Landeposition auftreten. Solche Flüge wurden in den Tests in ca. 50% der Druchführungen erkannt.



(a) Position des Balls auf dem Spielfeld



(b) Flughöhe des Balls

Abbildung 3.3: Position und Höhe eines fliegenden Balls (über beide Spielfeldhälften)

3.2 Filterung von Rauschen

Das Filtern des Rauschens ist der erste Testfall der umgesetzten Filterverfahren. Hierbei soll herausgefunden werden, wie gut die implementierten Filterverfahren das Rauschen entfernen. Zu diesem Zweck wird in diesem Testszenario ein statisches Objekt wie zum Beispiel der Ball auf eine Position gelegt und nicht bewegt. Dies geschieht für etwa 20 Sekunden. Danach wird eine weitere Position ausgesucht. Das gleiche Vorgehen wird auch hier vollzogen. Das ganze wird für unterschiedliche Positionen auf dem Spielfeld durchgeführt.
Zur Evaluation werden die Datensätze mit EV1XXX bezeichnet. Die drei hinteren Stellen geben an (in dieser Reihenfolge) durch welche Quelle die Kameradaten bereitgestellt[2] und welche Art der Verarbeitung[3] vorgenommen wurde. Die letzte Stelle dient zur Durchnummerierung der Datensätze jedes Szenarios.

Untersuchung des Rauschens bei realen Messungen Zur Untersuchung des Rauschens anhand von realen Daten wurde der Ball auf das $4,7m \times 3,54m$ großen Spielfeld gelegt und nicht bewegt. Da zwei Kameras über dem Spielfeld hängen und diese ein Weitwinkelobjektiv besitzen, sind vor allem der Überlappungsbereich und die Ränder für die Betrachtungen interessant.
Nachfolgende Tabelle zeigt die Zusammenfassung der aufgenommenen Daten. Angegeben sind neben den Positionen auch die Varianzen in x- und y-Richtung.
Die Positionen wurden nicht in absoluten Werten angegeben, sondern geben die relative Position auf dem Spielfeld an. Die Koordinate $[0; 0]$ bezeichnet so den Mittelpunkt des Spielfelds. Die x-Achse des Koordinatensystems liegt auf der Mittellinie. Dementsprechend verbindet die x-Achse die beiden Tore. Ein Wert von 1 bzw. -1 gibt an, dass der Ball auf der Spielfeldbegrenzungslinie platziert wurde.
Die Sichtbereiche der Kameras überlappen in einem Korridor auf der [0; y]-Linie.

[2] 1-reales Spielfeld mit SSL-Vision; 2-Simulator
[3] 0-keine Filterung; 1-Extended Kalman Filter; 2-Partikelfilter

Demzufolge befinden sich alle Positionen im Überlappungsbereich deren x-Koordinate sehr nahe bei 0 liegt. Messungen mit einer x-Komponente von 0.5 sind sicher außerhalb des Überlappungsbereichs.

Testfall	x-Position	y-Position	Varianz x	Varianz y
EV1100	-1	1	0,1656	0,1924
EV1101	0	-1	410,4204	1,3851
EV1102	0	0	0,9900	111,7827
EV1103	0	1	140,4900	210,4851
EV1104	0	0,5	25,6843	264,3504
EV1105	0	-0,5	28,0207	23,7881
EV1106	-0,5	0	0,0547	0,2300
EV1107	-1	0	0,5091	0,3336
EV1108	-0,5	0,5	0,2483	0,4223
EV1109	0,5	0	0,6025	0,2075
EV110A	1	0	1,5161	0,1135
EV110B	1	-1	0,2136	0,2546
EV110C	0,5	-0,5	0,2240	0,3998

Wie anhand der Tabelle zu erkennen ist, sind die Varianzen in dem Überlappungsbereich (d.h. $x = 0$) wesentlich höher als in den restlichen Bereichen des Spielfeldes. Abbildung 3.4 stellt die Tabelle bildlich dar. Die blauen Rechtecke geben die aufgenommenen Positionen an, rot dargestellt sind die Abweichungen. Je länger sich diese Balken erstrecken, umso höher ist die Varianz für die x- bzw. y-Position. Sie geben jedoch keine maßstabsgetreue Abweichung an. Der gelb angedeutete Bereich repräsentiert den Überlappungsbereich. Auch hier ist deutlich zu erkennen, dass innerhalb dieses Bereichs die Varianzen wesentlich höher sind. Allerdings kann kein Zusammenhang der x- bzw. y-Varianz mit den betrachteten Positionen hergestellt werden. Viele Faktoren wie Justierung der Kameras und Kalibrierung der Bildverarbeitungssoftware können sich hier auswirken. Diese Zusammenhänge weiter zu

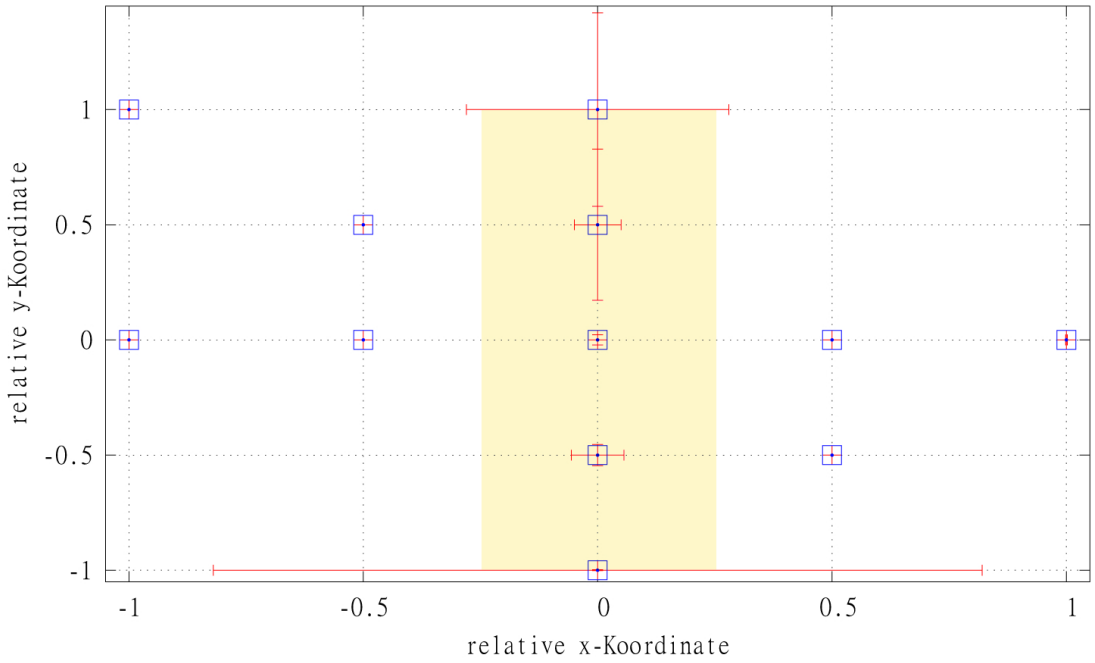


Abbildung 3.4: Ruhender Ball an verschiedenen Positionen mit Varianzen

untersuchen wurde in diesen Betrachtungen nicht vorgesehen. Diese Tatsache verdeutlicht allerdings, dass robuste Filterverfahren für den praktischen Einsatz bei RoboCup unbedingt benötigt werden.
Neben dem angesprochenen Überlappungsbereich waren auch die Ränder für die Rauschbetrachtungen interessant. Wie an dieser Auswertung zu erkennen, nehmen die Verzerrungen durch die Weitwinkelobjektive kaum Einfluss auf die Güte der Positionen. Somit kann man zusammenfassend sagen, dass sich das Rauschen nur in dem Überlappungsbereich verändert und sich sehr unvorhersehbar verhält. Aus diesem Grund sollten spätere Weiterentwicklungen der Filter diesen Bereich des Spielfelds gesondert betrachten.

Rauschfilterung an Hand von simulierten Daten Um die Güte der Rauschfilterung des Extended Kalman Filters und des Partikelfilters zu testen, wurden Daten im Simulator aufgenommen und über ein Flag die Varianz des Rauschens festgelegt.

Bei diesem Szenario wurde der Ball auf drei Positionen gelegt. Die erste ist der Spielfeldmittelpunkt und die anderen beiden sind Punkte auf der linken und rechten Spielfeldhälfte. Zur Auswertungen wurden immer diese drei Aufnahmen herangezogen. Die Datensätze zu den Auswertungen EV1210 bis EV1212 wurden mit einem Rauschen mit einer Standardabweichung von einem halben Zentimeter überlagert. Bei den Datensätzen zu EV1213 bis EV1215 wurden ein Rauschen mit einer Standardabweichung von einem viertel Zentimeter hinzugefügt.

Im Folgenden soll die Rauschfilterung des Extended Kalman Filters näher untersucht werden. Die hierbei berechneten statistischen Kennzahlen können in folgender Tabelle nachgelesen werden.

Testszenario	$\varnothing$	m	min	max	var	σ
EV1210	3,628892	3,382625	0,029803	17,110898	3,764984	1,940357
EV1211	4,228144	3,962501	0,206577	21,451823	5,077516	2,253334
EV1212	5,782475	5,329720	0,175934	29,335747	10,684012	3,268641
EV1213	1,660213	1,523853	0,033171	8,512472	0,897177	0,947194
EV1214	2,025390	1,825679	0,023753	18,826906	1,687821	1,299162
EV1215	2,753133	2,546277	0,018737	13,380801	2,339316	1,529482

Wie man an diesen Messdaten sehen kann, kann das Rauschen durch das Kalman-Filter stark vermindert werden. Vor allem in den Testfällen EV1210 und EV1213 konnte das Rauschen auf ein Minimum gefiltert werden. Dies ist das Szenario, in dem der Ball auf die Mittelposition gelegt wird. Anders als in der realen Aufnahme konnte durch vielfaches Updaten der Position diese besser und genauer angegeben werden.

Diese Ergebnisse zeigen deutlich, dass das Kalman-Filter sehr gut das Rauschen eines ruhenden Objektes kompensieren kann. Warum jedoch in Abhängigkeit zur Position des Balls (linke bzw. rechte Spielfeldhälfte) solch starken Unterschiede vorliegen, konnte nicht herausgefunden werden. Dies weiter zu untersuchen scheint jedenfalls nicht erträglich zu sein, da bereits die Tests auf einem realen Spielfeld ergeben haben,

dass die Rauschunterdrückung unabhängig von der Spielfeldhälfte ist.
Als nächstes sollen gleiche Untersuchungen auch für das Partikelfilter angestellt werden. Hierbei gelten die gleichen Voraussetzungen und Angaben zu Rauschen. Nachstehende Tabelle zeigt die hierfür ausgewerteten Daten.

Testszenario	$\varnothing$	m	min	max	var	σ
EV1220	5.433844	5.026777	0.144556	18.227050	8.614769	2.935093
EV1221	4.219003	3.859570	0.079896	14.923112	5.926316	2.434403
EV1222	5.669706	5.344591	0.108011	20.374581	8.354252	2.890372
EV1223	5.341106	5.035740	0.076395	20.188014	8.476018	2.911360
EV1224	4.146607	3.790931	0.067651	16.761044	5.295653	2.301229
EV1225	3.649185	3.501403	1.656620	6.359877	2.386868	1.544949

Die Anzahl der Partikel für diese Testfälle beträgt 100, das eingestellte Streuen der Partikel beträgt $8mm$. Dabei deutlich zu erkennen ist, dass dieses Streuen großen Einfluss auf die allgemeine Varianz nimmt. Bei einem Rauschen mit einem halben Zentimeter Standardabweichung, konnte dadurch die Abschätzung der Position verbessert werden. Ist das Rauschen jedoch geringer eingestellt, wirkt sich dies negativ auf die berechnete Position des Balls aus. Daraus lässt sich schlussfolgern, dass ein Kennen des Rauschens und das korrekte Einstellen der Streuung essentiell für das Partikelfilter sind.
Weiterhin wurde untersucht, wie sich die Anzahl der Partikel auf die Güte der Filterung auswirkt. Hierbei wurden ein Rauschen mit einer Standardabweichung von einem halben Zentimeter und eine Partikelstreuung von acht Millimetern eingestellt. Zu untersuchen war die Abschätzung bei einer Partikelanzahl von 100, 500 und 1000. Die Ergebnisse fasst die folgende Tabelle zusammen.

Partikelanzahl	∅	m	min	max	var	σ
100	5.669706	5.344591	0.108011	20.374581	8.354252	2.890372
500	1.351285	1.209837	0.022375	5.225560	0.640436	0.800272
1000	0.727458	0.645666	0.015785	3.080630	0.207882	0.455941

Hier ist ein deutlicher Trend zu sehen. Je höher die Anzahl der Partikel gewählt wird, desto besser wird die Positionsbestimmung. Bei einer Partikelanzahl von 500 kann dieses Filterverfahren bereits bessere Ergebnisse erzielen als die vorliegende Konfiguration des Extended Kalman Filters. Ein großer Nachteil ist hierbei jedoch die Laufzeit der Rechnung, die ab 1000 Partikel bereits an eine kritische Grenze stößt, ab der die Kameradaten nicht mehr in Echtzeit verarbeitet werden können. Da in den für diese Auswertung genutzten Datensätzen neben dem Ball nur ein weiterer Roboter auf dem Spielfeld war, wird eine weitere Verschlechterung der Performance unter Wettbewerbsbedingungen vermutet.

Dieses Testszenario hat gezeigt, dass das Kennen des Rauschens zum Einstellen der Filter sehr wichtig ist. Auch ist die Erkenntnis wichtig, dass sich das Rauschen im Überlappungsbereich der Kamerasichtfelder anders verhält als auf dem Rest des Spielfelds. Die Messergebnisse haben gezeigt, dass das Partikelfilter ab einer Partikelanzahl von 500 dem Extended Kalman Filter bei der Rauschunterdrückung überlegen ist. Jedoch leidet unter diesen Einstellungen die Performance des gesamten Systems dramatisch. Aus diesem Grund kann das Partikelfilter ohne Optimierung mit einer solchen Partikelanzahl nicht unter den Echtzeitbedingungen der Small Size League eingesetzt werden. Bei einer Partikelanzahl, die die Rechenzeit weniger stark beansprucht, liefert das Extended Kalman Filter bessere Ergebnisse.

3.3 Testen des Bewegungsmodells anhand von idealen Daten

Zum Testen des Bewegungsmodells ist es notwendig, bestimmte einflussnehmende Faktoren auszustellen, wie verrauschte oder unregelmäßige Kameradaten. Aus diesem Grund werden diese Tests mit Hilfe des Simulators durchgeführt, bei dem das Rauschen deaktiviert werden kann und die Kameradaten genügend gleichmäßig in *Sumatra* eintreffen, da der Simulator auf demselben System läuft.
Für diese Tests wurde folgender Ablauf festgelegt:

Testen des Ballbewegungsmodells Der Ball soll anfänglich langsam und gleichmäßig über das Spielfeld gerollt werden. Im Anschluss daran soll er mit hoher Geschwindigkeit rollen, um so einen Schuss durch einen Roboter zu simulieren. Abschließend soll der Ball gegen einen anderen Roboter prallen. Dieses Szenario wurde viermal aufgenommen werden (CF21X0 bis CF21X3).

Testen des Roboterbewegungsmodells Hier wird eine feste Abfolge von Wegpunkten durch die Roboter abgefahren. Dieses Szenario wurde fünfmal wiederholt (CF22X0 bis CF22X4).

3.3.1 Ballbewegungsmodell

In Folgendem wird das Ballbewegungsmodell untersucht. Hierfür wurden mehrere Testdurchläufe vorgenommen. Einige sprechende und interessante Datensätze werden nun im Näheren vorgestellt und untersucht.
Die mit Kalman-Filter ausgewerteten Datensätze bieten folgende statistische Kennzahlen:

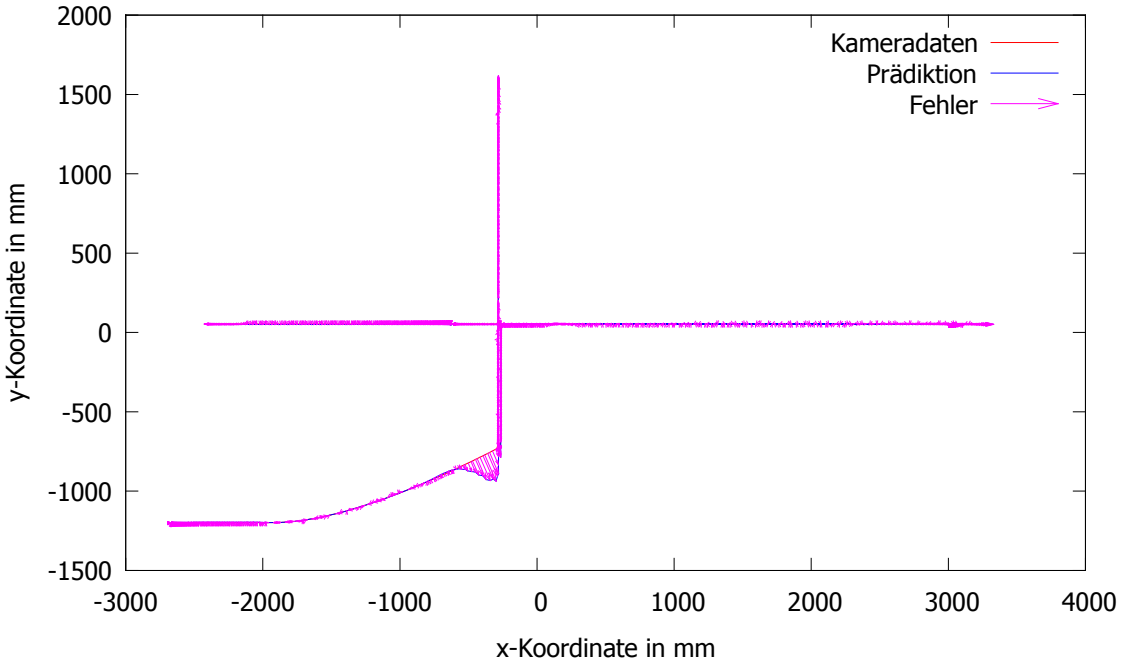


Abbildung 3.5: Simulierte Ballbewegungen ohne Rauschen und Vorhersage mit Kalman-Filter

Testszenario	∅	m	min	max	var	σ
EV2110	6,3251	2,1966	0,0000	185,4364	262,6183	16,2055
EV2111	7,1032	2,5493	0,0490	536,5521	479,6552	21,9010
EV2112	9,6108	2,7665	0,0569	640,3431	768,2024	27,7165
EV2113	7,6377	2,4825	0,0040	257,4319	462,8866	21,5148

Eine allgemeine Beurteilung zum Bewegungsmodell kann jedoch aus diesen Kennzahlen nicht folgen. Aus diesem Grund wird im Folgenden näher auf einzelne Bereiche der Datensätze eingegangen. Im Speziellen wird der Durchlauf EV2110 untersucht, welcher auf den Kameradaten CF2100 beruht.

Abbildung 3.5 zeigt den Verlauf des Balls. Hierbei bewegte dieser sich mit hoher Geschwindigkeit von links Mitte nach ganz rechts, danach langsam wieder zurück in die Mitte, ein kleines Stück nach unten, dann nach oben ausholend, um mit hoher Geschwindigkeit schließlich gegen einen Roboter zu prallen und nach links wieder

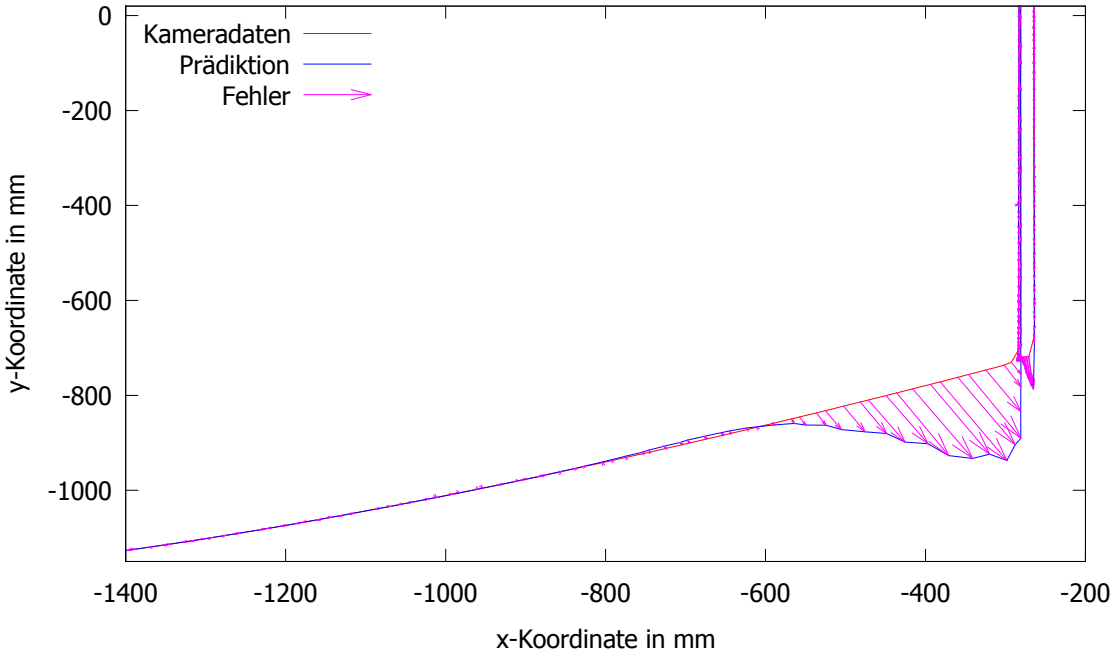


Abbildung 3.6: Abpraller des Balls an einem Roboter

auszurollen.

Besonders interessant ist die Situation, in der der Ball vom Roboter abprallt, siehe hierzu Abbildung 3.6.

Im Graphen grün dargestellt ist die durch die ideale Kamera des Simulators gelieferte Ballposition. Dies sind gleichzeitig die Eingangsdaten des Worldpredictors. Da hier kein Rauschen vorhanden ist, können diese Positionen als die realen Positionen angesehen werden. In rot dargestellt ist die Vorhersage des Filters. Die blauen Pfeile geben die Differenz zur Vorhersage und des passenden Kameraframes.

Wie man an diesem Graphen sehr gut erkennen kann, prädiziert das Filter zum Zeitpunkt des Abprallens noch in die vorherige Bewegungsrichtung des Balls. Erst nach einigen Messungen, wird die geänderte Bewegungsrichtung übernommen und es erfolgt eine Angleichung an der wirklichen Bahn des Balls. Nach einem sehr geringem Überschwinger bei $x = -600$ bis -800, stimmt die Bahn schließlich wieder sehr genau überein. Trotz der hohen Geschwindigkeit des Balls und der abrupten

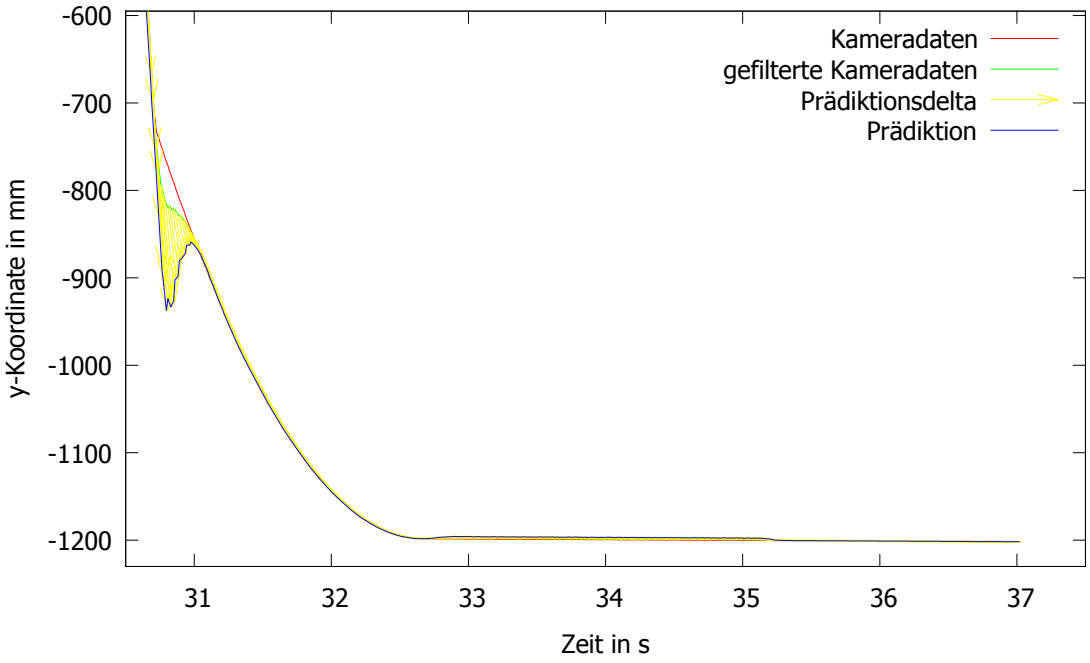


Abbildung 3.7: zeitlicher Verlauf des Abprallers des Balls in y-Richtung

Richtungsänderung weicht das Filter nur zehn bis zwanzig Zentimeter ab. Weiterhin ist an dieser Abbildung sehr gut zu erkennen, dass das Filter einen sich langsam ausrollenden Ball (die linke Hälfte im Graphen nach dem Abprallen) nahezu perfekt prädiziert. Dies ist an den sehr kleinen purpurnen Pfeilen zu sehen. Je kleiner sie sind, desto besser stimmen die vorhergesagten mit den realen Positionen überein. Aber auch ein sich schnell bewegender Ball (in der Abbildung die erste Senkrechte von links) wird sehr gut prädiziert.

Wichtig zu erfahren ist nun, wie diese Situation im zeitlichen Verlauf aussieht. Hierzu wird Abbildung 3.7 bereitgestellt. Der rote Graph stellt die idealen Kameradaten des Simulators dar. Der Grüne die durch das Extended Kalman Filter gefilterten Daten und der Blaue zeigt die vorhergesagten Positionen des Balls.

Wie bereits zu erwarten war, zeigt das Filter auch hier eine sehr gute Genauigkeit. Sobald sich der Ball auf einer Bahn befindet, kann das Filter diese sehr genau prädizieren. Die rote, grüne und schwarze Linie scheinen ineinander zu laufen. Wie

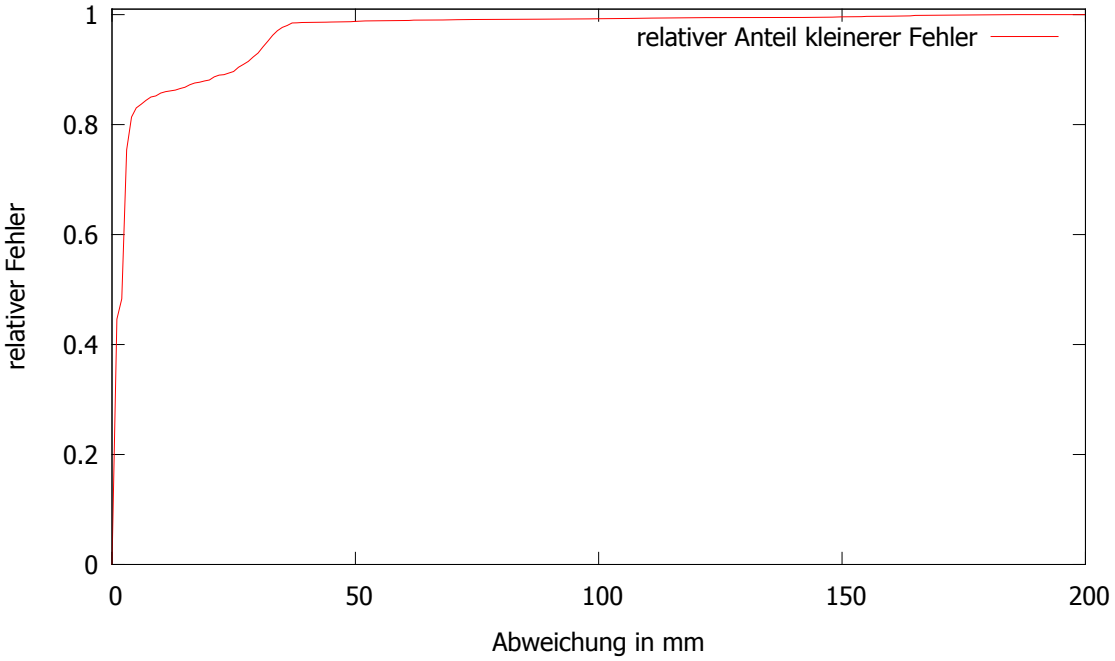


Abbildung 3.8: Fehlerverteilung des Durchlaufs EV2110

auch zuvor bereits gesehen, treten höhere Abweichung während des Aufprallens auf. Hier entstehen über einen Zeitraum von 0,3s Differenzen von in etwa 17cm auf. Daraus kann man schließen, dass das Bewegungsmodell bereits soweit entwickelt ist, gleichmäßige Bewegungen des Balls zu modellieren. Trotz des Wissens über ein mögliches Hindernis, sind Abpraller an diesem noch nicht modelliert und führen deswegen zu größeren Abweichungen in solchen Situationen

Abbildung 3.8 zeigt, wie hoch der Anteil der jeweiligen Fehler ist. Daran ist zu sehen, dass die Mehrzahl aller Abweichungen sich in einem sehr geringen Bereich befinden. Höhere Abweichung treten nur sehr selten auf. Da in diesem Testdurchlauf nur ein Abpraller an einem Roboter stattgefunden hat, ist dies nicht verwunderlich. Bei sehr viel mehr Interaktionen würde sich demnach der relative Anteil der größeren Abweichung deutlich erhöhen.

Da Abpraller aufwändig zu modellieren sind, wurde das Partikelfilter implementiert. Mit diesem ist es möglich, mehrere Hypothesen zur Position des Balls zu verfolgen.

Besonders interessant ist dies bei Abprallern des Balls an einem Hindernis. Da viele Faktoren, wie Auftreffwinkel, Material und Geschwindigkeit der Akteure eine Rolle spielen, ist es schwierig, dies über ein Modell zu erfassen. Aus diesem Grund wird die Monte-Carlo Methode angewendet und mehrere Hypothesen verfolgt. Dadurch könnte ein sehr viel einfacheres Modell implementiert werden, welches genau diese Problematik löst.

Doch bevor dieser Aspekt näher untersucht wird, folgt erst eine Analyse wie gut das Partikelfilter die vorher beschriebenen Testdurchläufe verarbeitet.

Folgende Tabelle zeigt die statistische Auswertung der Datensätze, die mit einem Partikelfilter mit 50 Partikeln verarbeitet wurden.

Testszenario	$\varnothing$	m	min	max	var	σ
EV2120	20.1975	11.2022	0.1345	440.7090	1041.0304	32.2650
EV2121	79.8736	33.5154	0.6455	1691.1299	21836.2111	147.7708
EV2122	32.9348	9.7283	0.1151	5007.1542	27884.4227	166.9863
EV2123	44.0084	13.2127	0.2320	2754.8776	17913.4317	133.8411
EV2124	30.9373	14.8989	0.4893	1311.5937	3649.2142	60.4087

Diese Tabelle zeigt bereits, dass das Partikelfilter nicht an die Genauigkeit des Kalman-Filters herankommt. Trotzdem soll im Folgenden wie bereits vorher beim Kalman-Filter der Datensatz CF2100 näher untersucht werden.

Hierzu werden wieder zwei Abbildungen bereitgestellt. Abbildung 3.9 gibt einen Gesamteindruck des Testdurchlaufs und Abbildung 3.10 zeigt einen näheren Ausschnitt dessen.

Abbildung 3.9 lässt leicht größere Unterschiede zwischen realen Daten und der vom Filter angenommenen Position erkennen. Dies wird deutlich durch den höheren Blauanteil im Graphen. Dieser stellt in Form von Pfeilen die Abweichungen zwischen realen und prädizierten Positionen dar. Das heißt, je deutlicher diese Pfeile erkennbar sind, desto schlechter ist die Vorhersage durch das Filter.

Ein Heranzoomen an die bereits bekannte Stelle dieses Szenarios zeigt nun, wie stark

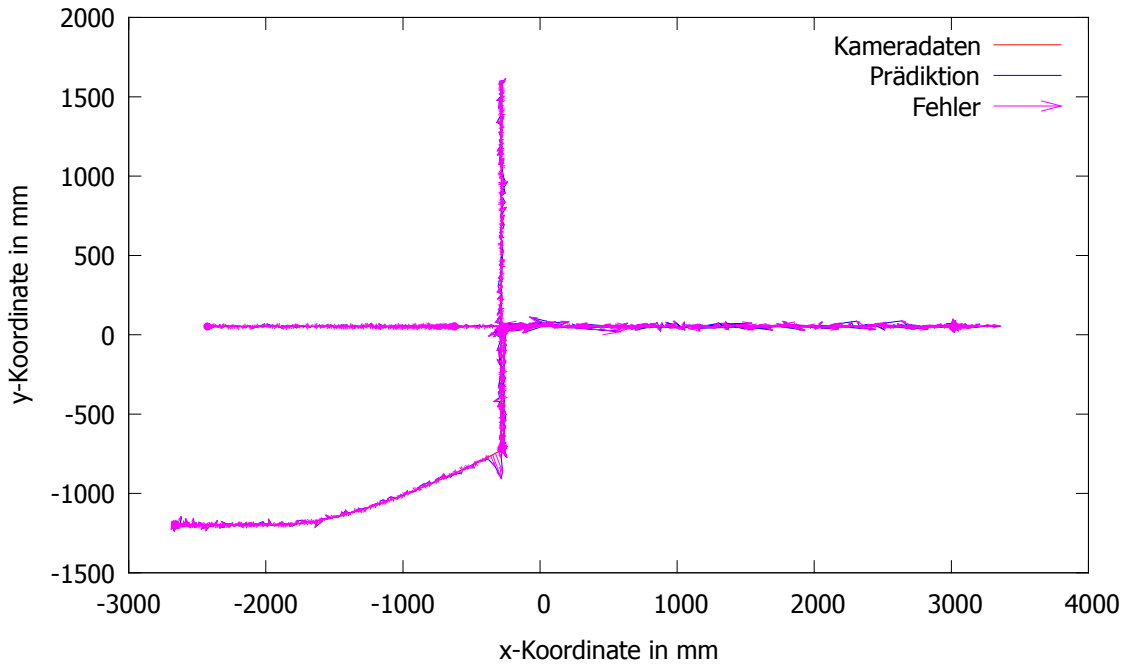


Abbildung 3.9: simulierte Ballbewegungen ohne Rauschen und Vorhersage mit Partikelfilter

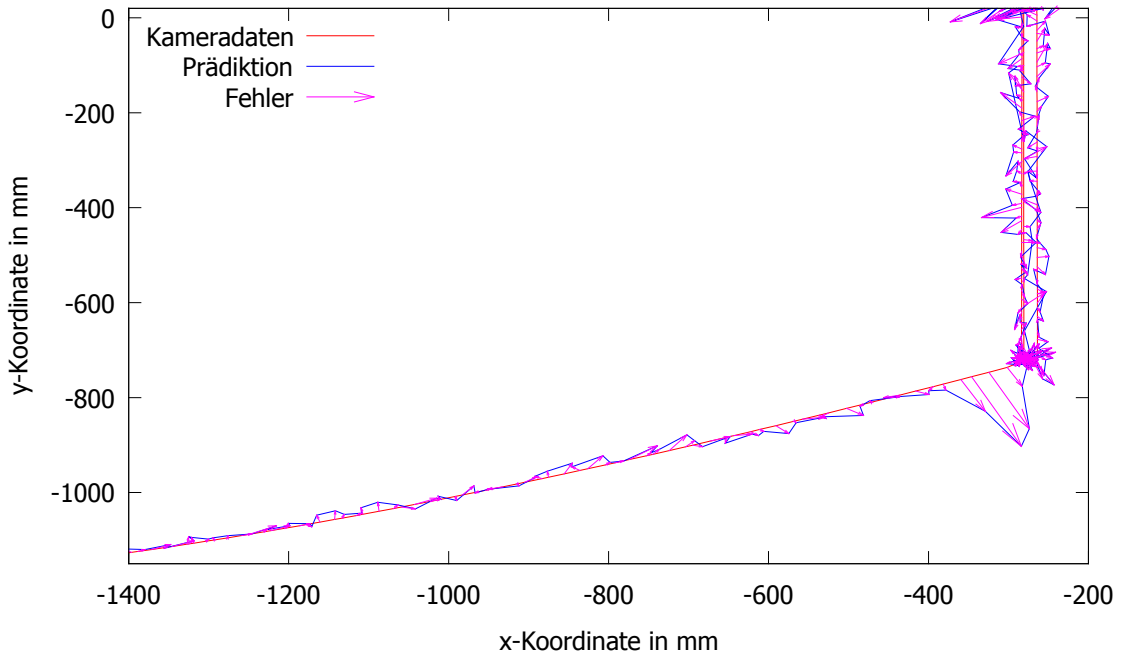


Abbildung 3.10: Abpraller des Balls an einem Bot gefiltert mit Partikelfilter

das Partikelfilter um die reale Position schwankt. Diese Schwankung ist zurückzuführen auf die zufällige Streuung der Partikel. Da nur 50 Partikel zum Vorhersagen benutzt wurden, wirkt sich die geringe Anzahl auf die Prädiktion aus. Statt zu filtern wird ein Rauschen noch hinzugegeben. Um dies zu verhindern, muss die Partikelanzahl deutlich erhöht werden. Dies wiederum hat einen Einbruch der Performance zur Folge. Das heißt, dass nicht alle Kameradaten verarbeitet werden können. In diesen Zeitintervallen ohne Einbeziehung der Messung, liefert das Filter sehr schlechte Positionsabschätzungen.

Weiterhin ist auch hier ein Überschwingen zum Zeitpunkt der Reflektion zu erkennen. Da noch keine Implementierung für eine Reflektion an einem Hindernis vorhanden ist, wurde dies erwartet.

Eine Problematik, die sich bei der Auswertung ergab, war die Einbeziehung der Geschwindigkeit für den Ball. Kann diese nicht zuverlässig bestimmt werden, leidet auch der Aktualisierungsschritt der Partikel darunter. Da noch keine ausreichend gute Methode implementiert wurde, um die Geschwindigkeit aufzunehmen, kann das Partikelfilter dementsprechend auch keine guten Abschätzungen über die Position liefern. Für das Kalman-Filter trifft dies nicht zu, weil die Geschwindigkeit durch den Filteralgorithmus automatisch mitbestimmt wird, wohingegen man sich beim Partikelfilter mit dieser Problematik weitergehend auseinander setzen muss.

Diese Auswertungen haben gezeigt, dass das Partikelfilter noch nicht performant genug ist, um dieses auch für den Produktiveinsatz zu nutzen. Ausschlusskriterien sind vor allem die Verarbeitungsgeschwindigkeit und der Mangel einer Implementierung zur Bestimmung der Ballgeschwindigkeiten. In beiden Fällen können diese Probleme durch weitere Recherchen und Implementierungen behoben werden.

Da das Extended Kalman Filter bereits gute Ergebnisse zeigt, ist die Prädiktion der Ballposition durch den Worldpredictor gut genug gewährleistet. Das Kalman-Filter kann für den Produktiveinsatz genutzt werden, während weiterhin das Partikelfilter verbessert wird. Mit Einbeziehung von Abprallern an Hindernissen kann das Parti-

kelfilter noch eine große Rolle spielen und sollte aus diesem Grund weiter untersucht werden.

3.3.2 Roboter-Bewegungsmodell

In diesem Kapitel wird die Prädiktion von Roboterbewegungen mit Hilfe des Extended Kalman Filters untersucht. Dabei werden zwei Fälle unterschieden: Zum einen die Prädiktion unter Bekanntheit der Ansteuerungsinformationen und zum anderen die Prädiktion ohne Ansteuerungsinformationen. Der hierfür abgefahrene Pfad ist in Abbildung 3.11 zu sehen.

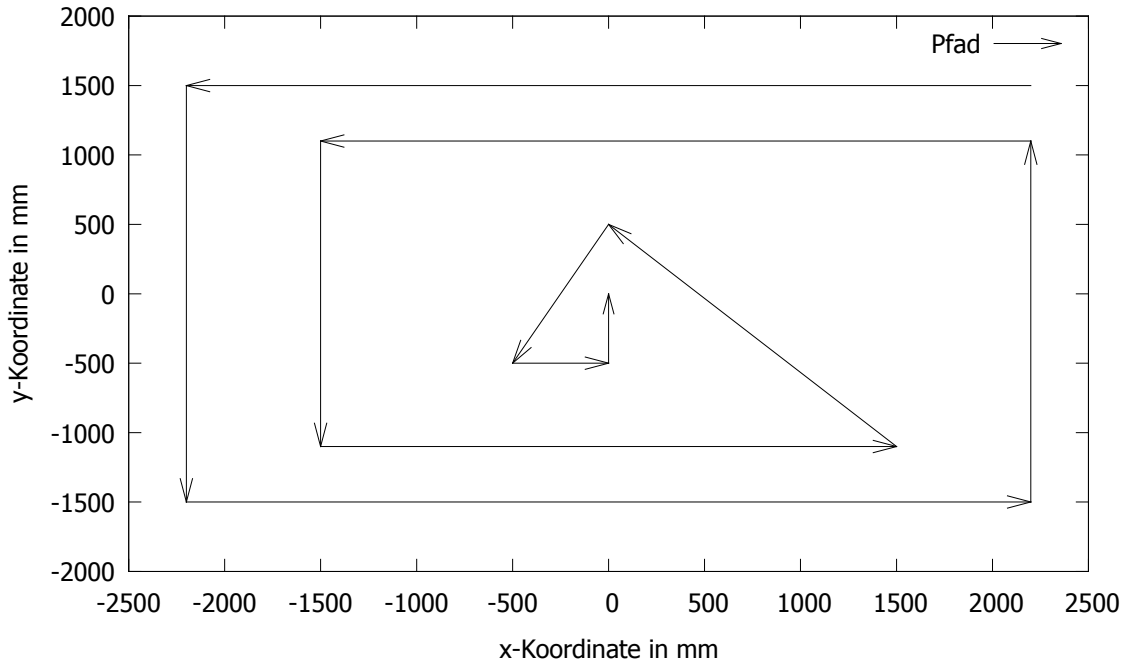


Abbildung 3.11: Pfad für Robotertests

Eine Auswertung der gesamten Testdurchläufe folgt in den nachfolgenden zwei Tabellen. Die erste Tabelle zeigt die statistischen Kennzahlen in Bezug auf die Position des Roboters, die zweite die Ausrichtung des Roboters. Hierbei wurden jeweils als erstes die Auswertungen mit Ansteuerungen (EV221X) und anschließend ohne Ansteuerungen (EV222X) nach den Kameradaten gruppiert.

Position:

Testdurchlauf	∅	m	min	max	var	σ
EV2210	12.2886	3.9593	0.0005	128.8740	336.2469	18.3370
EV2220	6.7925	7.3447	0.0001	55.4328	32.9014	5.7360
EV2211	12.0107	4.3025	0.0004	111.2012	281.0091	16.7633
EV2221	7.5028	7.7363	0.0008	60.8079	43.8730	6.6237
EV2212	11.0772	3.1782	0.0011	538.3464	509.2890	22.5674
EV2222	7.3623	7.3436	0.0004	69.9251	43.0043	6.5578
EV2213	11.4255	3.1291	0.0005	341.1046	399.1118	19.9778
EV2223	7.1611	7.3304	0.0003	68.0787	43.9338	6.6283
EV2214	10.9369	2.8802	0.0001	163.7685	249.6984	15.8019
EV2224	6.3374	6.6041	0.0004	36.0222	29.6367	5.4440

Anhand dieser Tabellen kann man sehen, dass das Einbeziehen der Ansteuerungsdaten die Prädiktion nicht verbessert. Bei simulierten Daten liegen die prädizierten Zustände mit Ansteuerungsdaten im Durchschnitt schlechter, wohingegen sie beim Median deutlich besser sind. Der Grund dafür könnte bei der Simulation der Spielfeldumgebung liegen. Da in einer Simulation die Realität auf ein Minimum vereinfacht wird und deswegen die reale Welt nicht perfekt nachbilden kann, verhalten sich simulierte Objekte mit auf die Realität justierter Ansteuerung nicht genauso wie in der realen Testumgebung. Um diese Unstimmigkeiten zu beseitigen, müssten diese Ansteuerungen auch für den Simulator angepasst werden. Dies ist jedoch nicht zielführend und wird deshalb nicht weiter in Betracht gezogen. Die nächsten Tabellen fassen alle Testdurchläufe wieder getrennt nach Position und Ausrichtung zusammen. Weiterhin gibt Abbildung 3.12 eine grafische Darstellung der Verteilung des relativen Fehlers.

Diese Tabellen zeigen, dass bei idealen Daten eine Verbesserung durch Ansteuerungen nicht stattfindet. Auch die Fehlerverteilung lässt den gleichen Schluss zu. Lediglich im Bereich mit sehr geringen Abweichungen können Daten besser prädiziert werden. Das lässt den Schluss zu, dass Ansteuerungen, die eine geringe Veränderung der Roboterbahn verursacht zu einer besseren Prädiktion führt als bei größeren

Winkel:

Testdurchlauf	$\angle\varnothing$	$\angle m$	$\angle min$	$\angle max$	$\angle var$	$\angle\sigma$
EV2210	0.0152	0.0027	0.0000	0.1790	0.00058	0.0241
EV2220	0.0042	0.0018	0.0000	0.0508	0.00003	0.0056
EV2211	0.0174	0.0030	0.0000	0.1596	0.00065	0.0255
EV2221	0.0047	0.0024	0.0000	0.0344	0.00004	0.0059
EV2212	0.0118	0.0014	0.0000	0.3264	0.00047	0.0218
EV2222	0.0032	0.0009	0.0000	0.0281	0.00002	0.0045
EV2213	0.0137	0.0019	0.0000	0.2663	0.00051	0.0226
EV2223	0.0037	0.0014	0.0000	0.0289	0.00003	0.0050
EV2214	0.0148	0.0016	0.0000	0.1652	0.00064	0.0252
EV2224	0.0038	0.0009	0.0000	0.0344	0.00003	0.0056

Position:

Durchschnittswerte	$\varnothing(\varnothing)$	$\varnothing(m)$	$\varnothing(var)$	$\varnothing(\sigma)$
mit Ansteuerungen	11.547771	3.489838	355.071041	18.689485
ohne Ansteuerungen	7.031206	7.271794	38.669840	6.197925

Veränderungen. Dies jedoch lässt noch keinen Schluss auf die Güte des Gesamtsystems zu. In diesem Fall wurde auf idealen Daten gearbeitet. Wie sich das System bei realen Daten verhält, muss weiterhin getestet werden.

Die Abweichungen einer leichten Kurvenfahrt in X-Richtung ist in Abbildung 3.13 gegeben. Die rote Linie ist anfänglich noch relativ gut auf der idealen Linie des Roboters. Doch mit den Ansteuerungsdaten kann sie den Kurvenverlauf nicht sehr gut nachverfolgen. Dahingegen kann das Filter die Position wesentlich besser filtern, wenn es ohne Ansteuerungsdaten arbeitet. Hierbei wird sich lediglich auf die Daten der Kamera verlassen, die nicht verrauscht sind und deswegen die ideale Position bereits angeben. Nach dem Kurvenverlauf nähern sich beide Filtervarianten der

Winkel:

Durchschnittswerte	$\varnothing(\angle\varnothing)$	$\varnothing(\angle m)$	$\varnothing(\angle var)$	$\varnothing(\angle\sigma)$
mit Ansteuerungen	0.014576	0.002108	0.000571	0.023845
ohne Ansteuerungen	0.003919	0.001473	0.000029	0.005339

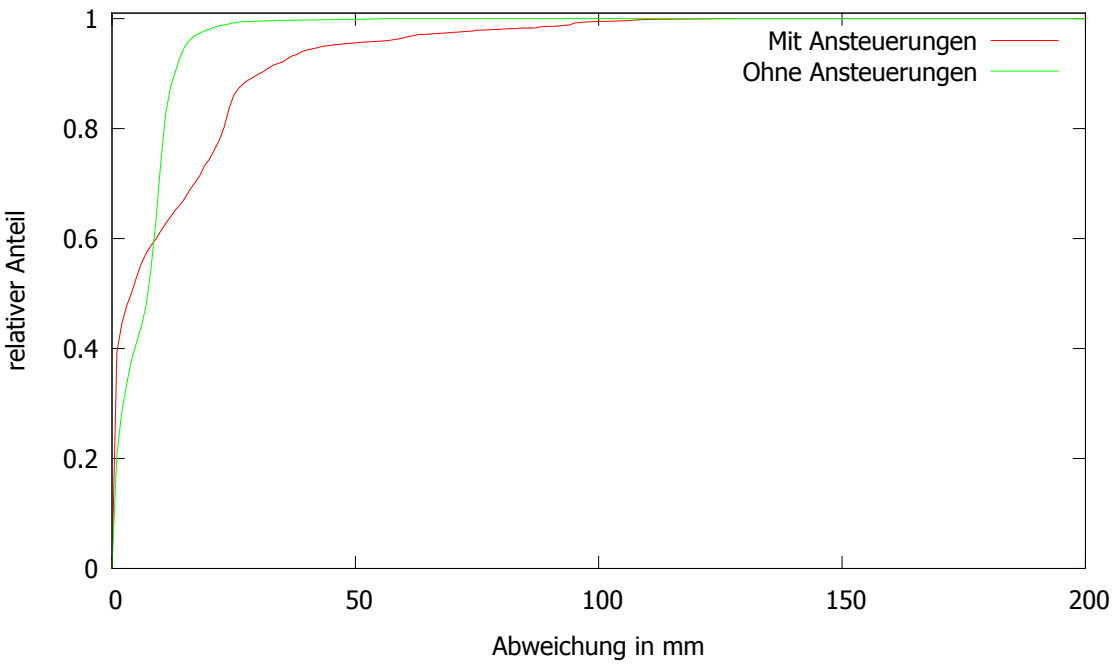


Abbildung 3.12: Fehlerverteilung mit und ohne Ansteuerungsinformationen bei idealen Daten

idealen Spur wieder sehr gut an.

Zum Abschluss folgt Abbildung 3.14, die über die Roboterfahrt ein Gesamteindruck liefert. Hierbei wurden die Ansteuerungen miteinberechnet. Zu erkennen ist, dass besonders in den Eckpunkten des Pfades die Roboterbewegung sehr schlecht prädiziert werden konnte. Wie bereits dargestellt helfen die Ansteuerungsdaten bei idealen Kameraframes wenig und führen eher zu einer Verschlechterung der Prädiktion. Aus diesem Grund folgt eine nähere Untersuchung unter realen Testbedingungen im anschließenden Kapitel.

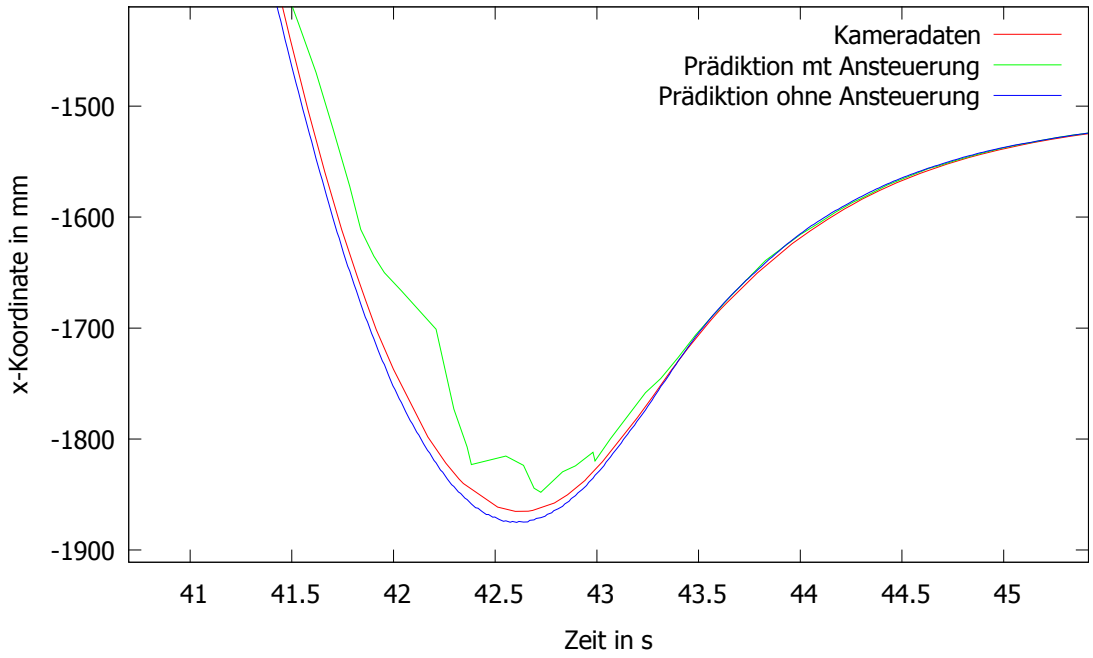


Abbildung 3.13: Vergleich einer Kurvenfahrt eines Roboters in X-Richtung mit Ansteuerungsdaten und ohne

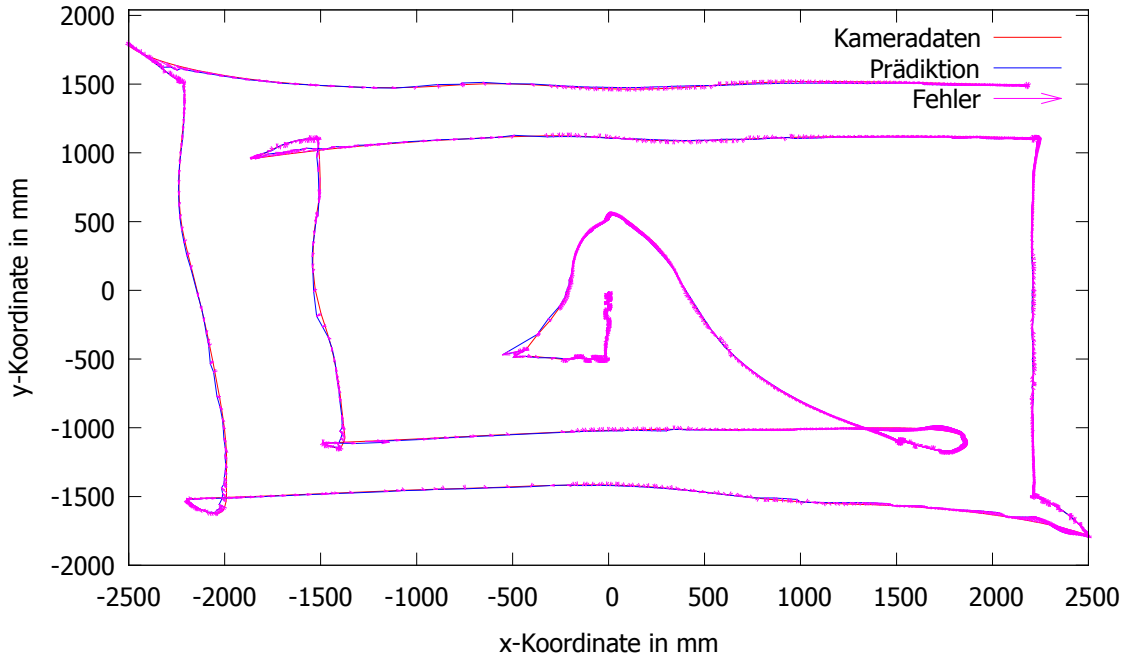


Abbildung 3.14: Prädiktion eines Roboters mit Ansteuerungen und idealen Daten

3.4 Testen des Bewegungsmodells anhand von realen Daten

In diesem Testfall wird das Extended Kalman Filter unter Wettkampfbedingungen auf einem realen Spielfeld getestet. Dabei werden die Daten wie in Kapitel 3.3 beschrieben aufgenommen und ausgewertet. Die aufgenommenen Kameradaten erhalten dementsprechend folgende Nummer: CF31X für die Balldaten und CF32X für die Roboterdaten. Hierbei werden für die Evaluierung des Ball-Bewegungsmodells mehrere Szenarien aufgenommen und für das Roboter-Bewegungsmodell ein Szenario fünf mal durchlaufen.
Eine Betrachtung des Partikelfilters wird hier nicht mehr vorgenommen. Aus den Ergebnissen in 3.2 und 3.3 geht hervor, das in der bisherigen Implementierung bezüglich Laufzeit und Genauigkeit das Extended Kalman Filter zu bevorzugen ist.

3.4.1 Ball-Bewegungsmodell

Zum Testen des Ball-Bewegungsmodells wurde ein Ball auf dem realen Spielfeld von umher geschossen.
Abbildung 3.15 zeigt ein Beispiel für die Aufnahmen eines umher gespielten Balls aus dem Datensatz CF31C.
Die nachfolgenden Statistiken betrachten entweder den Abstand zu den Kameradaten oder zu gefilterten Kameradaten. Die gefilterten Datensätze liegen näher an der realen Position liegen, da bei diesen das Messrauschen der Kameras entfernt wurde. Bei schnellen Geschwindigkeitsänderungen ist die erfolgt die Übernahme jedoch verzögert. Abbildung 3.16 zeigt den gefilterten Pfad gegenüber dem aufgenommenen Pfad. Abbildung 3.17 zeigt Ausschnitte aus Abbildung 3.16 zur Verdeutlichung der oben genannten Vor- und Nachteile der gefilterten Daten.
Abbildung 3.18 zeigt den rechten Ausschnitt aus 3.17 inklusive Prädiktion und Fehlervektoren zu den Kameradaten. Das Einschwingen des Extended Kalman Filters

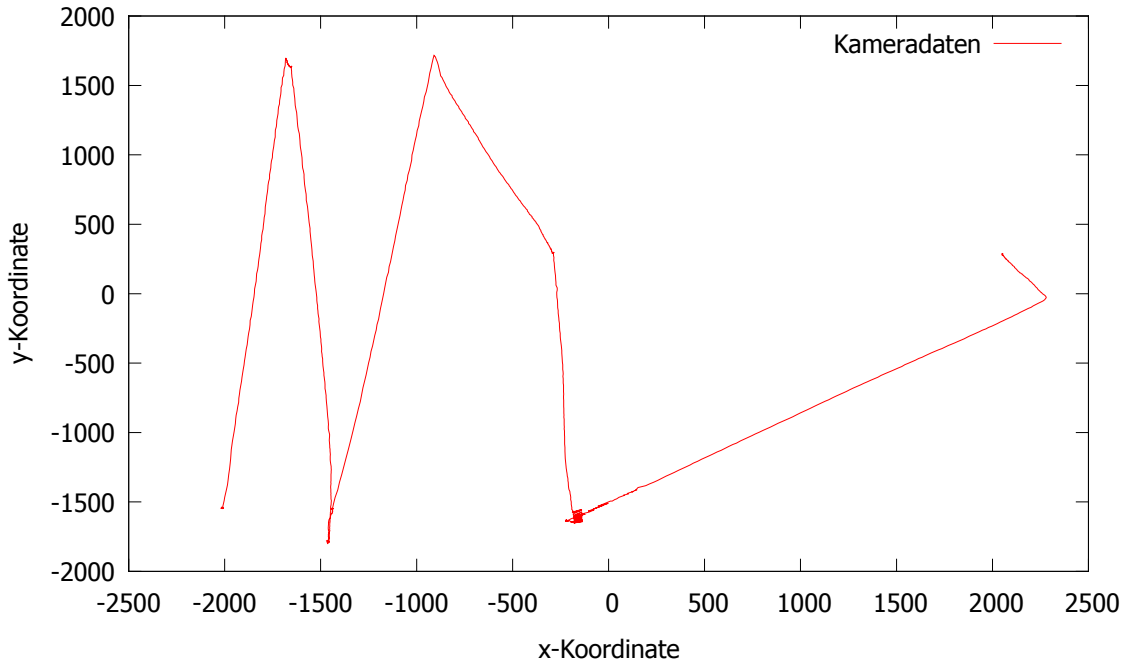


Abbildung 3.15: Ballposition aus Datensatz CF31C

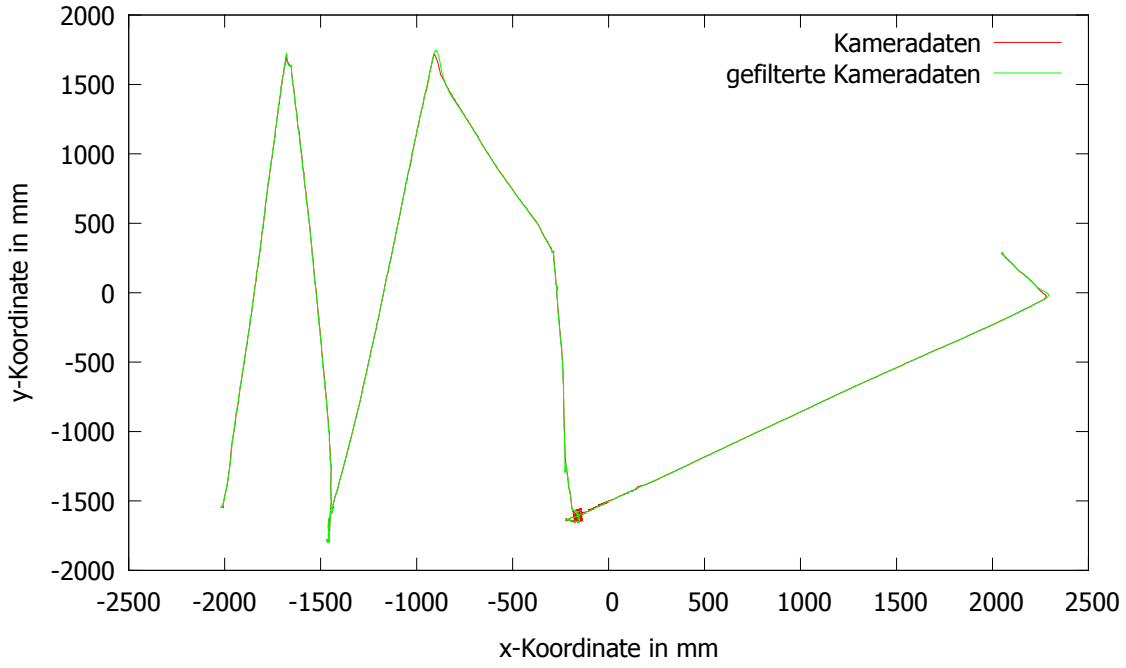


Abbildung 3.16: Kameraaufnahmen und gefilterte Daten der Ballposition (CF31C)

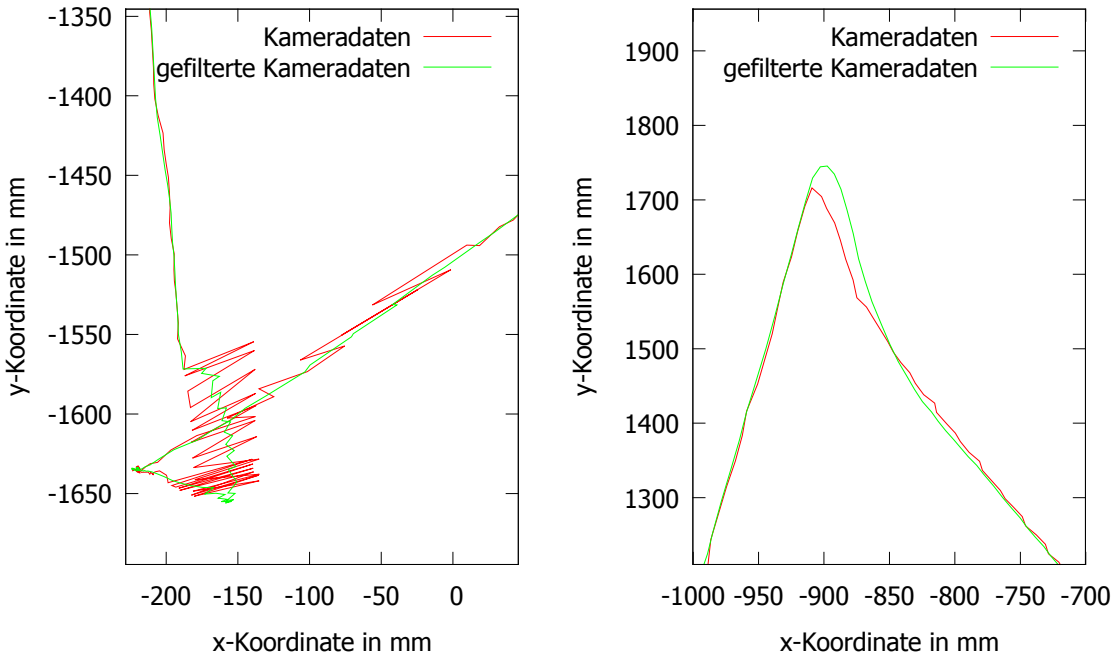


Abbildung 3.17: Vor- und Nachteile gefilterter Daten (CF31C)

bei einer unerwarteten Richtungsänderung ist deutlich zu erkennen.

Zur weiteren Auswertung wurden weitere Fehlergrößen zu den Zeitpunkten der Messungen bestimmt. Dafür wird der Positionsunterschied von den Kameradaten und den gefilterten Kameradaten zur Prädiktion berechnet. Zusätzlich wird der Positionsunterschied betrachtet, welcher entstehen würde wenn ungefilterte und unprädizierte Daten als Ausgabe genutzt würden[4]. Abbildung 3.19 zeigt die verschiedenen Fehlerausprägungen. Damit diese erkennbar sind, ist die y-Koordinate gegenüber der Zeit abgebildet. Zudem wurde ein Ausschnitt gewählt bei dem sich die Geschwindigkeit in y-Richtung stark ändert. Die Pfeile symbolisieren die y-Komponente des Fehlers.

In Abbildung 3.20 ist die Verteilung der Fehlerbeträge ablesbar. Die Graphen geben

[4]Dieser Fall entstände wenn Daten aus der Kamera direkt verwendet würden und kein Filter oder Bewegungsmodell verwendet würde. Bei dieser Verwendung ohne Worldpredictor würden im System keine Geschwindigkeitsinformationen der Objekte berechnet werden, welche von den nachfolgenden Modulen benötigt werden.)

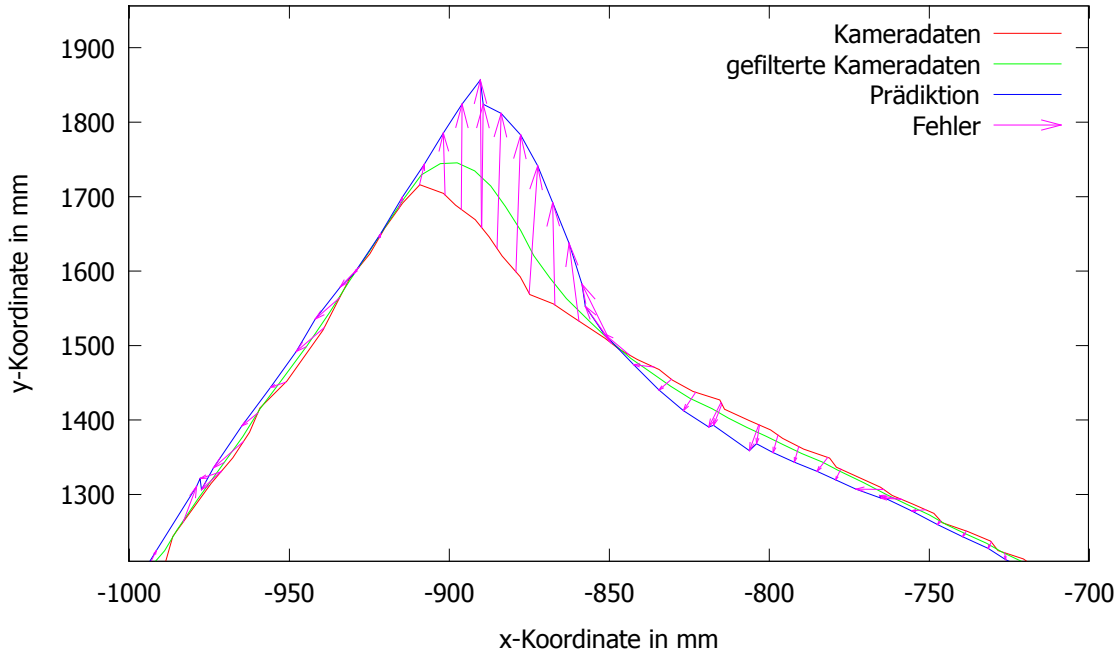


Abbildung 3.18: Ausschnitt inklusive Prädiktion (CF31C)

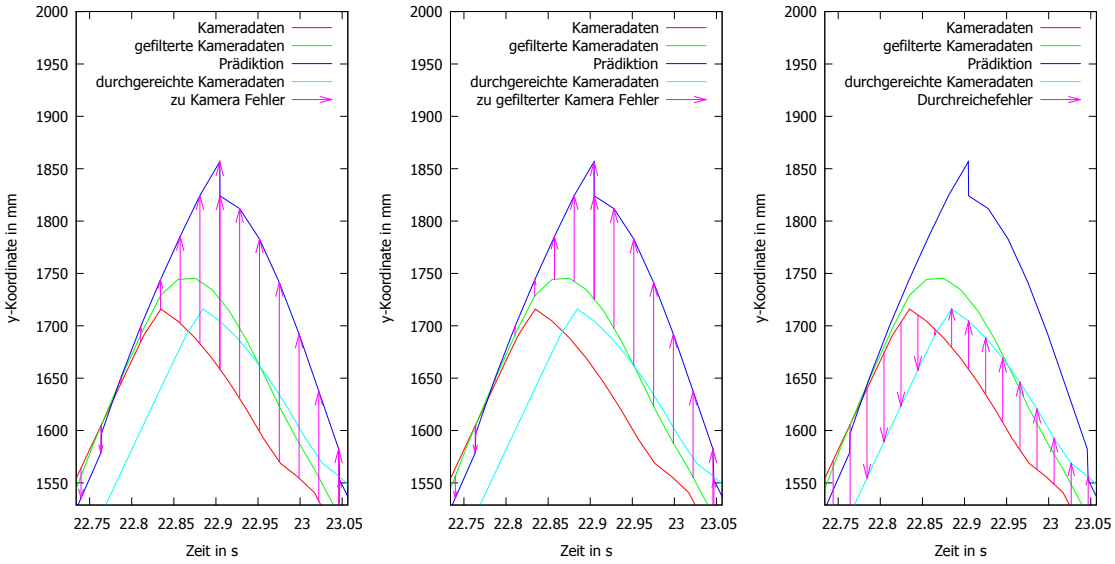


Abbildung 3.19: Verschiedene Fehlertypen (CF31C)

zu den verschiedenen Fehlertypen den relativen Anteil der Daten in den Datensätzen an, welche einen kleineren Fehler aufweisen. Repräsentativ für alle aufgenommen Datensätze ist stets der Fehler zu den gefilterten Kameradaten der geringste. Zu den real aufgenommen Kameradaten sind die Fehler tendenziell größer. Werden Kameradaten nur durchgereicht entsteht ein großer Fehler, welcher auf die fehlende Geschwindigkeitsbetrachtung zurückzuführen ist.

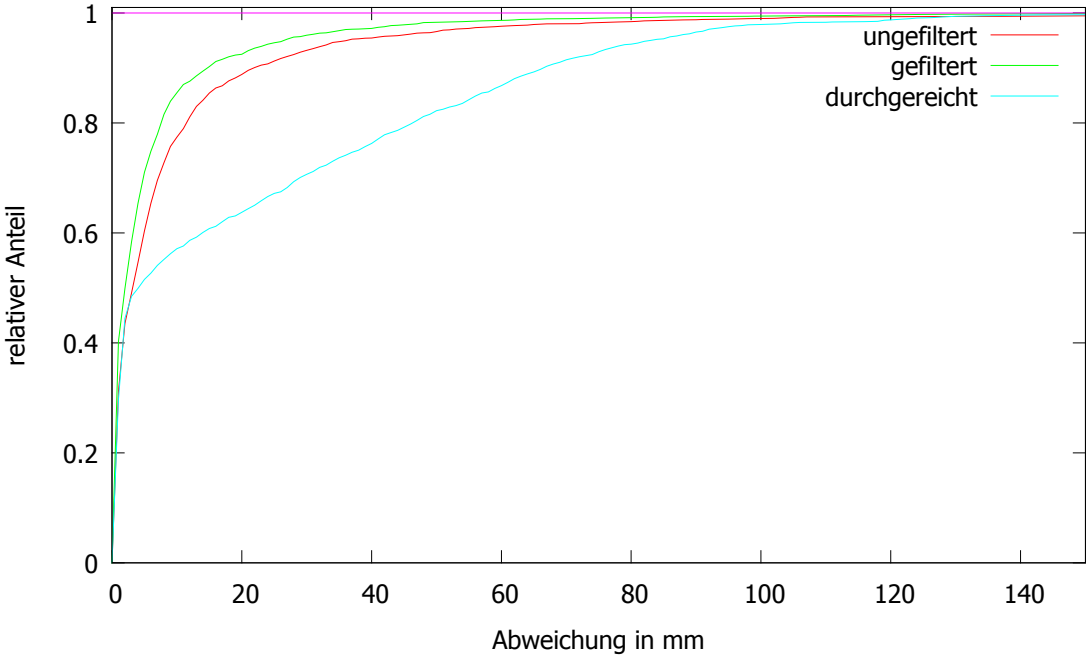


Abbildung 3.20: Fehlerverteilung (CF31C)

Aus dem bisher betrachten Datensatz CF31C sind folgende Größen berechnet worden:

CF31C	$\varnothing$	m	min	max	var	σ
ungefiltert	9.2705	3.1630	0.0040	248.1281	419.1378	20.4729
gefiltert	6.3036	2.0296	0.0088	182.4713	202.4500	14.2285
durchgereicht	21.9021	4.0376	0.0000	220.5099	917.6363	30.2925

Aus der Mittelung über 16 unterschiedliche Datensätzen ergeben sich folgende Durchschnittsgrößen:

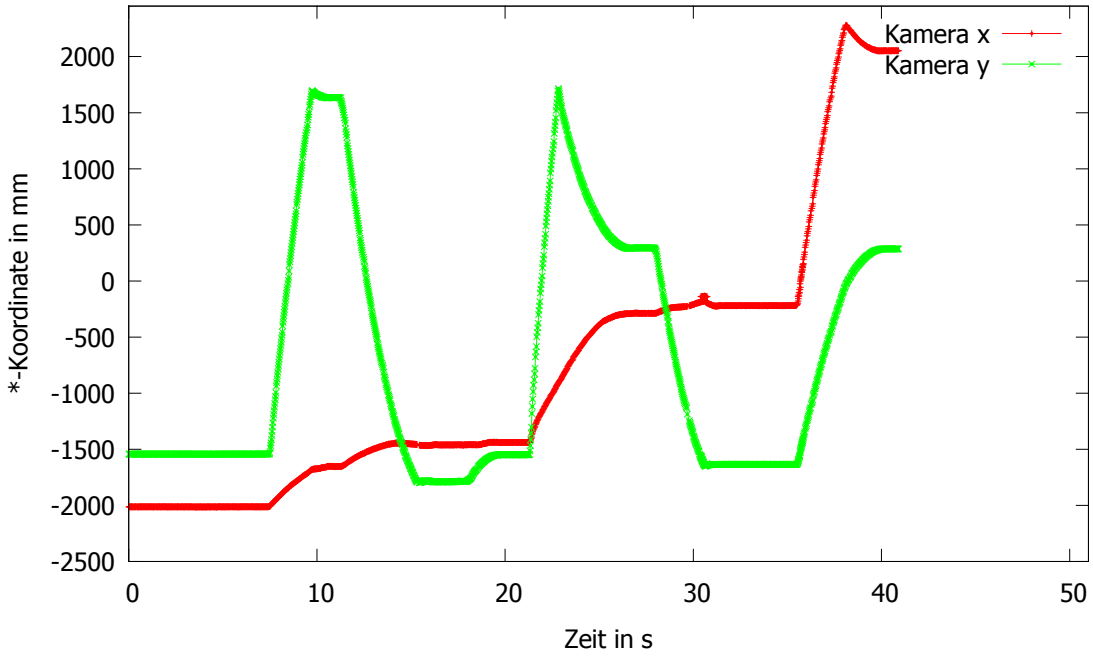


Abbildung 3.21: Kameradaten aus CF31C, gut geeignet

Ball-Tests	$\varnothing(\varnothing)$	$\varnothing(m)$	$\varnothing(var)$	$\varnothing(\sigma)$
ungefiltert	17.2935	6.4886	1619.6521	37.1785
gefiltert	11.3857	3.9727	832.4275	26.3000
durchgereicht	36.3669	24.7914	1855.8994	41.8794

Es ist erkennbar, dass in der Mittelung die Fehler deutlich größer sind als beim betrachteten Beispieldatensatz CF31C. In diesem Testdatensatz herrschten für das Filter und Bewegungsmodell gut geeignete Bedingungen. Diese sind maßgeblich abhängig von der Regelmäßigkeit der eingehenden Daten und den Bewegungsgeschwindigkeiten. Abbildung 3.21 zeigt die eingehenden Daten aus CF31C. Dem entgegen sind in Abbildung 3.22 die Daten aus CF315 gezeigt. Diese enthalten deutlich höhere Geschwindigkeiten und zudem längere Phasen von bis zu einer Sekunde ohne Aktualisierung.

Dem entsprechend zeigen auch die Statistiken zu dem Datensatz CF315 größere Fehler:

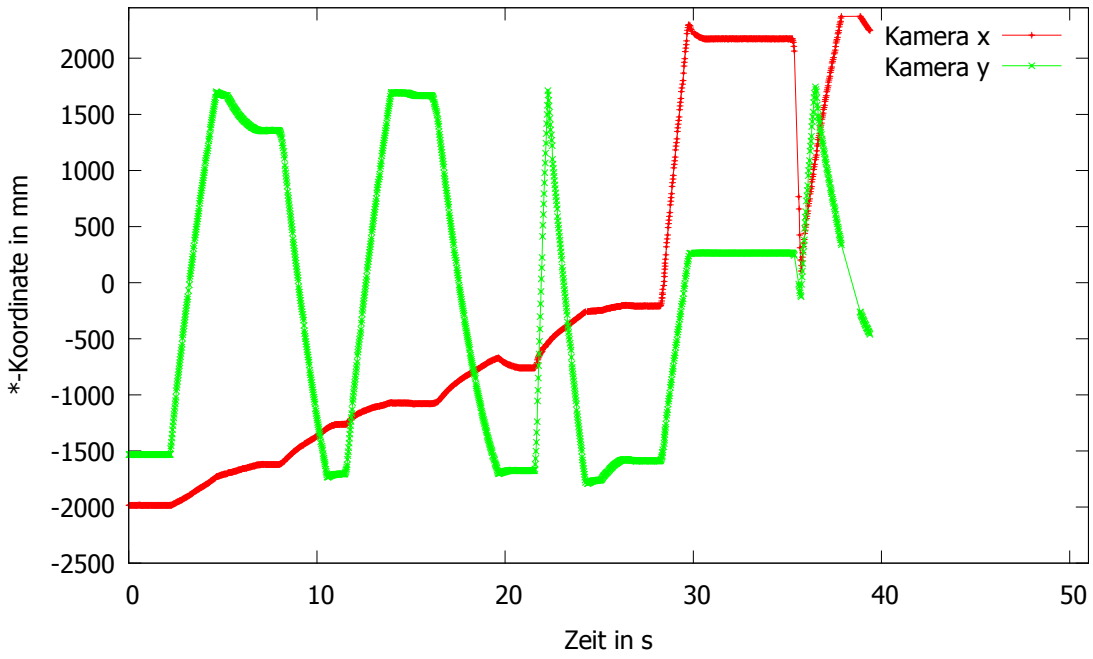


Abbildung 3.22: Kameradaten aus CF315, hohe Geschwindigkeiten, fehlende Frames, daher schlecht geeignet

CF315	∅	m	min	max	var	σ
ungefiltert	20.4258	5.5880	0.0087	1597.4988	4028.5737	63.4711
gefiltert	14.8507	3.7732	0.0033	1548.8613	2845.5050	53.3433
durchgereicht	39.0130	12.5470	0.0000	356.2104	2543.7554	50.4357

Als durchschnittliche Fehler über alle Datensätze ergeben sich $17mm$ zu den ungefilterten Kameradaten und $11mm$ zu den gefilterten Daten. Der reale durchschnittliche Fehler lässt sich nicht ermitteln da keine Idealdaten vorhanden sind (und auch nicht vorhanden sein können). In die ungefilterten Fehler geht Rauschen ungedämpft ein, die gefilterten Daten reagieren nicht sofort auf Geschwindigkeitsänderungen.
Aus den Vergleichen geht zudem hervor, dass der Median des Fehlers deutlich unter dem Erwartungswert liegt. Dies ist begründet durch einen sehr geringen Fehler bei freien Bewegungen. Große Fehlerwerte entstehen erst bei abrupten Beschleunigungen oder Abbremsungen.
Mit einem Fehler-Median von weniger als $7mm$ zur Kamera bzw. unter $4mm$ zur den gefilterten Daten hin erweist sich die Kombination aus Filter und Bewegungsmodell für den Ball als wettbewerbstauglich.

3.4.2 Roboter-Bewegungsmodell

Für die Validierung des Roboter-Bewegungsmodells wurde ähnlich vorgegangen, wie in Kapitel 3.4.1. Die Bewegung der Roboter erfolgte durch Abfolge von Bewegungsbefehlen, durch welche die Roboter feste Wegpunkte ansteuerten. Der abzufahrende Pfad ist in Abbildung 3.11 abgebildet.
Am Beispiel des Testdatensatzes CF321 sind in Abbildung 3.23 neben dem Pfad die real aufgenommenen Positionen des Roboters abgebildet. Dabei lassen sich mehrere Probleme erkennen. Ein Problem ist, dass der Roboter nicht exakt dem Pfad folgt. Im Gegensatz dazu entstehen sogar große Differenzen zwischen dem Pfad und der zurückgelegten Strecke. Dies ist eine Folge aus der noch nicht idealen Bewegungs-Ansteuerung des Roboters. Besonders bei Fahrten orthogonal zur Ausrichtung des

Roboters treten große Abweichungen auf, wie bei Bewegungen parallel zur y-Achse zu erkennen ist. Durch die abweichende Fahrt lässt sich allerdings das Bewegungsmodell für Kurvenfahren validieren ohne künstlich Kurven vorzugeben. Ein anderes erkennbares Problem ist starkes Kamerarauschen im Überlappungsbereich der Kameras um $x = 0$. Dies würde ohne den Einsatz von Filtern zu unbrauchbaren Prädiktionen führen.

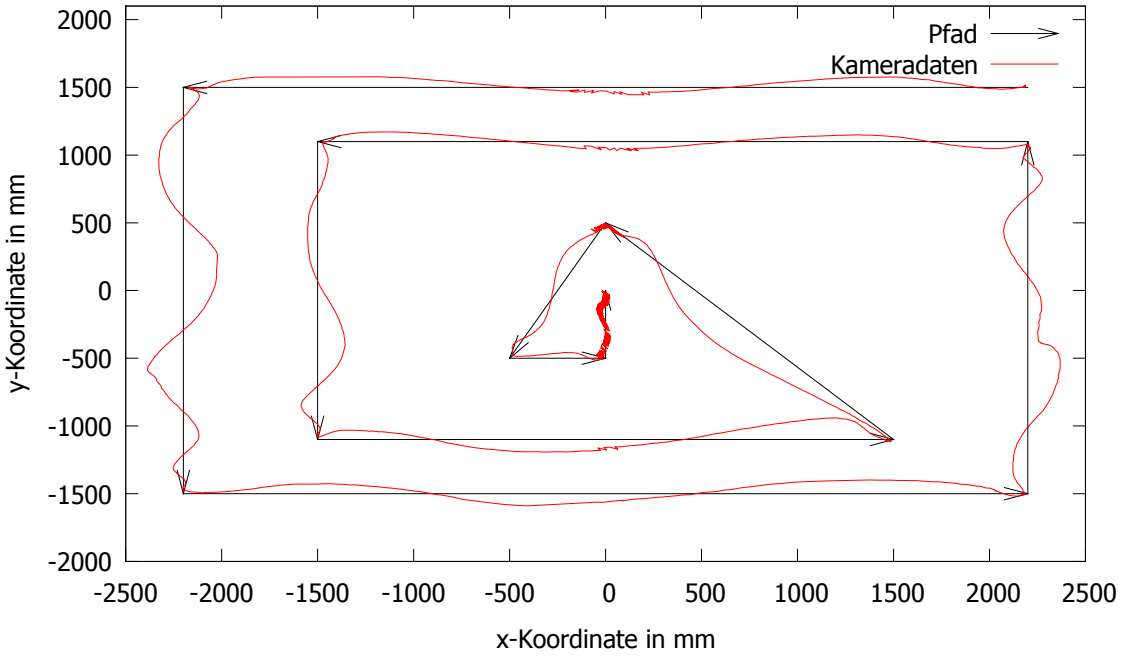


Abbildung 3.23: Folge des Testpfades (CF321)

Durch die in Kapitel 2.6.1 vorgestellte Softwarekomponente ist es möglich, empfangene Daten von SSL-Vision aufzuzeichnen und beliebig abzuspielen. Auf dieser Basis wurden die Testdaten direkt aufgenommen, mit Ansteuerungen prozessiert und herausgeschrieben. Für gegnerische Roboter wird prinzipiell das gleiche Bewegungsmodell verwendet wie für eigene Roboter. Da zu gegnerischen Robotern keine Ansteuerungsdaten vorliegen, sind die beeinflussenden Teile aus dem Bewegungsmodell entfernt. Als Testdaten für dieses Bewegungsmodell ohne Ansteuerungen wurden die aufgenommenen SSL-Frames der Tests des Bewegungsmodells mit Ansteuerungs-

daten verwendet. Dadurch sind die Resultate der Vorhersagen direkt miteinander vergleichbar.

Abbildung 3.24 zeigt die Nutzung der Ansteuerungsdaten. Bewegt sich der Roboter mit annähernd konstanter Geschwindigkeit, liegen beide Bewegungsmodelle nahe beieinander. Bei einer Änderung der Geschwindigkeit wird das Modell mit Ansteuerungen sofort beeinflusst, reagiert noch bevor die Änderung in den Kameradaten sichtbar ist und deckt sich mit später aufgenommenen Kameradaten. Das andere Modell dagegen folgt durch die Natur des Extended Kalman Filters der Geschwindigkeitsänderung erst verzögert. In der Abbildung ist auch erkennbar, dass ein Durchreichen der Kameradaten bei einer Bewegung immer zu einem Fehler führt.

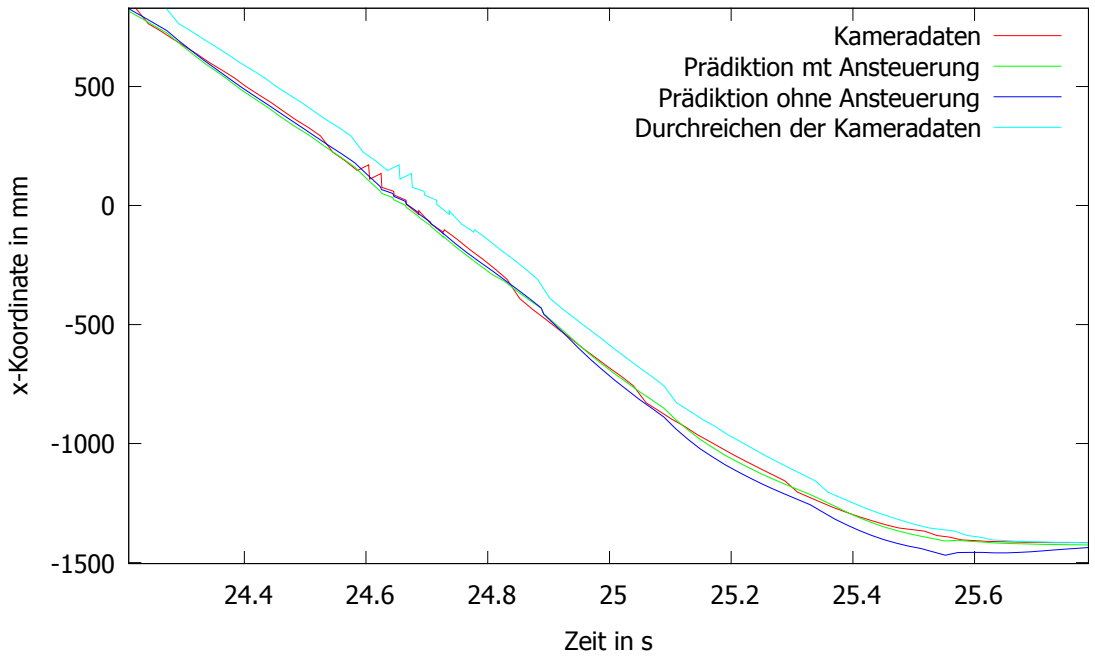


Abbildung 3.24: Vergleich mit und ohne Ansteuerung (CF321)

Analog zu Kapitel 3.4.1 lassen sich für die Position des Roboters Fehler bestimmen. Da durch zwei Bewegungsmodelle auch verschiedene Prädiktionen erfolgen gibt es fünf betrachtbare Fälle. Diese sind:

- *Anst. Kam.*: Prädiktion mit Ansteuerung gegenüber der Kamera
- *Anst. Fil.*: Prädiktion mit Ansteuerung gegenüber der gefilterten Kameradaten
- *Kam.*: Prädiktion ohne Ansteuerung gegenüber der Kamera
- *Fil.*: Prädiktion ohne Ansteuerung gegenüber der gefilterten Kameradaten
- *durchg.*: Durchreichen der Kameradaten

Dabei ist zu beachten, dass das Filtern das Bewegungsmodell verwendet. Dementsprechend sind die gefilterten Daten in *Anst. Fil.* und *Fil.* unterschiedlich.
Für die verschiedenen Fehlerbetrachtungen ergeben sich unterschiedliche Fehlerverteilungen. Diese sind in Abbildung 3.25 gezeigt. Die Fehler für das Bewegungsmodell mit Ansteuerungen zu den gefilterten Kameradaten sind am geringsten. Die Fehler mit Ansteuerungen zu den Kameradaten sind vergleichbar mit den Fehlern ohne Ansteuerung zu den Filterdaten. Dies spiegelt sich in allen Testfällen wieder obwohl kein Zusammenhang zwischen den Fällen bekannt ist. Weitaus größer sind die Fehler ohne Ansteuerung zu den Kameradaten. Die schlechtesten Ergebnisse werden bei einem reinen Durchreichen der Kameradaten erreicht.
Der betrachte Datensatz CF321 führt zu den folgenden statistischen Größen:

CF321	$\varnothing$	m	min	max	var	σ
Anst. Kam.	12.9536	10.8557	0.1199	106.1055	102.3472	10.1167
Anst. Fil.	7.2204	5.4940	0.0350	100.2173	49.2574	7.0184
Kam.	21.7626	17.5774	0.1868	284.7471	331.1721	18.1981
Fil.	11.4462	8.0697	0.0772	280.6107	144.2968	12.0124
durchge.	34.3031	30.1097	0.0000	177.1939	813.7533	28.5264

Da ein Roboter zusätzlich zu seiner Position auch eine Orientierung besitzt, sind auch für diese Fehlerwerte vorhanden und wurden ausgewertet. Daraus bildet sich folgende Tabelle:

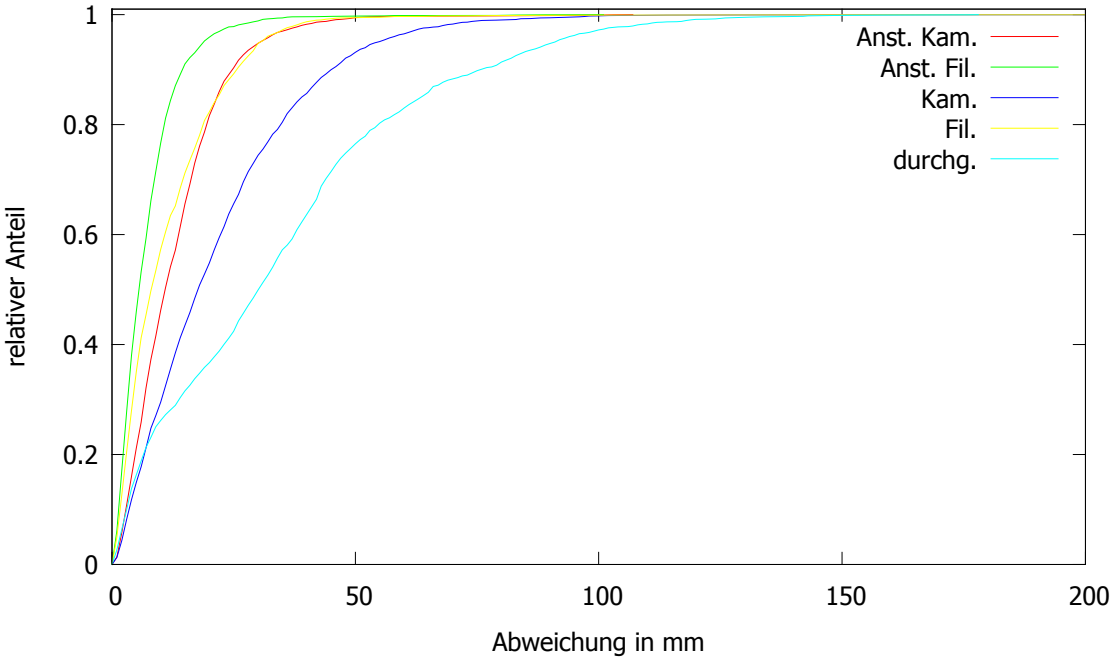


Abbildung 3.25: Vergleich der Fehlerverteilungen (CF321)

CF321	$\angle\varnothing$	$\angle m$	$\angle min$	$\angle max$	$\angle var$	$\angle\sigma$
Anst. Kam.	0.0227	0.0182	0.000000	0.1295	0.0004	0.0190
Anst. Fil.	0.0104	0.0074	0.000002	0.0910	0.0001	0.0104
Kam.	0.0228	0.0186	0.000008	0.1585	0.0004	0.0191
Fil.	0.0119	0.0088	0.000002	0.1376	0.0001	0.0116
durchge.	0.0185	0.0149	0.000017	0.0987	0.0002	0.0154

Die Fehler hierbei belaufen sich auf umgerechnet weniger als $1,5°$ im Durchschnitt. Auffällig jedoch ist das entgegengesetzt zu den bisherigen Betrachtungen auch durchgereichte Ergebnisse teils als besser zu bewerten sind als Daten aus dem Bewegungsmodell. Dies ist darauf zurückzuführen, dass die Ausrichtung des Roboters stark vom Schlupf beim Fahren abhängig ist. Da die Ansteuerung der Roboter zum Zeitpunkt der Datenaufnahme noch nicht ideal war, konnte es vorkommen, dass Räder durchdrehten oder vom Boden abhoben, wodurch es zu spontanen Drehungen des Roboters kam. Diese unvorhersehbaren Bewegungen werden derzeit vom Bewegungs-

modell nicht erfasst. Für spätere Betrachtungen sollte die Berechnung des Winkels weiter untersucht werden.

Im Allgemeinen verliefen alle Testfälle sehr ähnlich mit vergleichbaren Ergebnissen. Aus diesem Grund wird auf die anderen Testfälle nicht weiter eingegangen. Zum Überblick sollen folgende Tabellen dienen:

Roboter-Tests	$\varnothing(\varnothing)$	$min(\varnothing)$	$max(\varnothing)$	$\varnothing(m)$	$min(m)$	$max(m)$
Anst. Kam.	12.3627	11.9273	12.9536	10.2222	9.7095	10.8557
Anst. Fil.	6.9469	6.6391	7.2204	5.1176	4.6000	5.4940
Kam.	21.5991	20.9492	22.3382	18.4706	17.5774	18.8989
Fil.	11.2082	10.8991	11.6251	8.3054	8.0697	8.5753
durchge.	33.2505	30.3545	36.0778	28.5482	25.6117	31.7152

Roboter-Tests	$\varnothing(var)$	$min(var)$	$max(var)$	$\varnothing(\sigma)$	$min(\sigma)$	$max(\sigma)$
Anst. Kam.	104.9999	90.0105	120.3260	10.2359	9.4874	10.9693
Anst. Fil.	51.9968	41.4636	65.1750	7.1918	6.4392	8.0731
Kam.	281.2147	234.0081	331.1721	16.7351	15.2973	18.1981
Fil.	121.0015	94.4469	153.7488	10.9452	9.7184	12.3996
durchge.	796.1509	724.0713	851.8541	28.2064	26.9086	29.1865

Durch die Minima und Maxima ist erkennbar, dass die Testläufe innerhalb enger Grenzen bezüglich der betrachteten Parameter verliefen. Einzig die Intervalle der Varianz wirken groß. Dies entsteht durch die Quadrierung bei dem Berechnen der Varianz aus der Standardabweichung. In den Tabellen ist erkennbar, dass diese nur gering variiert.

Zusammenfassend weist das Bewegungsmodell für Roboter zusammen mit dem Extended Kalman Filter gute Ergebnisse auf. Sehr deutlich ist auch zu erkennen, dass das Wissen um die Ansteuerungsdaten einen großen Genauigkeitsvorteil bringt. In den durchgeführten Tests konnte mit den Ansteuerungen der Fehler bei den Vorhersagen nahezu halbiert werden. Die aus den Tests gewonnenen Daten weisen darauf

hin, dass das System für Roboter einsatzfähig ist. Die Fehler sind sehr gering. Je nach Betrachtungsweise muss man im Median mit weniger als $11mm$ Abweichung von eigenen und ca. $19mm$ Abweichung bei gegnerischen Robotern in Bezug auf die Kameradaten rechnen. Betrachtet man die Fehler gegenüber den gefilterten Kameradaten, reduzierten sich die Fehler im Median auf $5mm$ mit und $8mm$ ohne Ansteuerungen. Wie auch schon bei der Validierung des Ball-Bewegungsmodells anhand realer Daten ist auch hier aus Mangel an unverfälschten Positionsdaten der reale Fehler nicht berechenbar. Die Betrachtung zu den Kameradaten und die zu den gefilterten Kameradaten bilden allerdings gute Näherungswerte.

3.5 Performance der Matrixoperationen

Mit Hilfe von JUnit-Testfällen wurde die Performance der Java-Bibliothek *Jama* und der eigenen Matrix-Klasse verglichen. Die Initialisierung von Matrizen auf verschiedenste Art und Weise und die benötigten Operationen auf Matrizen (Addition, Subtraktion, Multiplikation, Transposition und die Invertierung) sind Gegenstand der Untersuchung. Sämtliche Tests fanden auf verschiedenen Systemen statt und lieferten sehr ähnliche Ergebnisse. Abweichend waren nur die Gesamtlaufzeiten. Die Verhältnisse der Laufzeiten der Test untereinander von einem System zu einem anderen System wichen um weniger als 5% ab. Die in den nächsten Abschnitten vorgestellten Ergebnisse lieferte ein System mit Windows 7 Professional (64-Bit), Intel(R) Core(TM)2 Duo CPU, P8400 mit 2.26 GHz und 3GB RAM. Diese Ergebnisse werden hier normiert präsentiert.

Initialisierung

Zur Initialisierung von Matrizen werden von der Matrixklasse verschiedene Konstruktoren angeboten. Die wohl einfachste Art, eine Matrix zu erzeugen, ist der Aufruf des Konstruktors mit zwei Werten vom Typ `Integer`. Diese geben die Di-

mension der Matrix an. Die Matrix wird, wenn kein Wert durch einen dritten Parameter vom Typ `Double` übergeben wird, initial an allen Stellen mit dem Wert Null belegt. Bei dieser Operation ist die neue Matrixklasse $\approx$ 1.5-mal so schnell, wie die *Jama*-Bibliothek. Wird dem Konstruktor zur Erzeugung der Matrix ein eindimensionales Array übergeben, liefert die neue Klasse die Matrix um den Faktor 1.7 schneller zurück. Darf der mit dem Array übergebene Speicher direkt für die Matrix genutzt werden, kann zusätzlich eine Verbesserung um den Faktor 1.5 (Gesamtfaktor: $\approx$ 2.5) erzielt werden. Lediglich bei der Initialisierung mit einem zweidimensionalen Array ist die neue Matrixklasse um den Faktor 2 langsamer. Dies ist darin begründet, dass *Jama* automatisch den Speicher des übergebenen zweidimensionalen Arrays nutzt und die eigene Matrixklasse die übergebenen Daten zunächst in ein eindimensionales Array konvertieren muss.
Auf dem oben angegebenen System können in $1ms$ ca. 8000 Matrizen der Dimension ($3x3$) mit einem zweidimensionalen Array initialisiert werden. Sollen hingegen Matrizen der Größe $3x4$ erzeugt werden, welche mit dem Wert Null vorbelegt sind, werden bis zu 27.000 in einer Millisekunde von der neuen Matrixklasse bereitgestellt. Bei gleichen Voraussetzungen konnten mit *Jama* nur 16.000 Matrizen generiert werden. Ähnliche Laufzeiten sind auch bei der Erzeugung von Identitätsmatrizen zu beobachten. *Jama* lieferte 15.000 in einer Millisekunde, die neue Matrixklasse hingegen 24.000.

Grundlegende Rechenoperationen

Bei den grundlegenden Rechenoperationen wie Addition, Subtraktion und Multiplikation konnte, im Vergleich zu *Jama*, eine Verbesserung um den Faktor 1.7 erreicht werden. Dies lässt sich auf die geringere Zugriffszeit bei eindimensionalen Arrays zurückführen. Des Weiteren ist es bei der Addition und der Subtraktion möglich, das Endergebnis wieder in der Ausgangsmatrix zu speichern und somit das Anlegen einer neuen Ergebnismatrix zu vermeiden. Hierdurch kann ein zusätzlicher Faktor

von 2.5 (Gesamtfaktor: ≈ 4.2) erreicht werden. Die vierte grundlegende Rechenoperation ist die Transposition. Hier erzielt die eigene Matrixklasse eine Verbesserung um den Faktor 2.5.
In absoluten Zahlen bedeutet dies, dass 2000 Additionen mit Matrizen der Größe $4x4$ mit *Jama* in einer Millisekunde addiert werden können. Mit der neuen Matrixklasse konnten rund 3300 Additionen in der gleichen Zeit durchgeführt werden. Bei der Subtraktion verhält es sich ähnlich. Große Unterschiede sind wiederum bei der Transposition zu beobachten gewesen. Mit der neuen Matrixklasse konnten 5600 Transpositionen in einer Millisekunde durchgeführt werden. Die *Jama*-Bibliothek konnte in der gleichen Zeit nur 2200 Matrizen transponieren.

Invertierung

Die Invertierung ist die aufwendigste aller Operationen. Es können mit der neu Implementierten Matrixklasse in einer Millisekunde 3000 symmetrisch positiv definite Matrizen der Dimension $6x6$ invertiert werden. Die *Jama*-Implementierung liegt geringfügig darunter. Für sehr kleine Matrizen ist die neue Bibliothek allerdings wesentlich schneller, da für Dimensionen kleiner gleich vier für jedes Matrixelement die Berechnungsformel direkt angegeben wurde. Die Invertierung einer $2x2$ Matrix kann auf diese Weise 11500 mal in einer Millisekunde erfolgen. Wohingegen bei *Jama* nur 1000 Invertierungen in der gleichen Zeit möglich sind.

Die gemessenen Werte bestätigen eindeutig, dass die eigene Matrixklasse performanter ist, als die zunächst verwendete Bibliothek *Jama*. Es konnte für die meisten Operationen eine Zeitersparnis von 40% erreicht werden. Bei der Wiederverwendung des gegebenen Speicherbereiches bei Operationen wie der Addition konnte sogar eine Verbesserung gegenüber *Jama* um 80% erzielt werden.

Kapitel 4

Zusammenfassung und Ausblick

In dieser Arbeit wurde die Implementierung eines Softwaremoduls für den Teamserver des RoboCup Teams Tigers Mannheim vorgestellt. Dieses wird dafür verwendet, um aus verrauschten und mit zeitlicher Verzögerung ankommende Beobachtungsdaten der Roboter und des Balls auf dem Spielfeld, Vorhersagen zum zukünftigen Spielsituationen zu generieren. Dafür wurde zunächst das Modul Worldpredictor in *Sumatra*, dem Softwaresystem der Tigers Mannheim, definiert. In diesem wurde eine Umgebung zur optimalen Verarbeitung der eingehenden Daten durch probabilistische Verfahren geschaffen. Als Verfahren zur Rauschreduktion und Prädiktion wurde ein Extended Kalman Filter und ein Partikelfilter mit den in [4] beschriebenen Bewegungsmodellen implementiert.
Durch verschiedene Testszenarien konnte nachgewiesen werden, dass durch die implementierten Verfahren das Messrauschen sehr gut entfernt werden kann. Des Weiteren ist eine genaue Prädizierung der Objektbewegungen auf einem realen Spielfeld möglich. Lediglich bei Kollisionen von Objekten treten Abweichungen auf, da diese derzeit noch nicht gesondert behandelt werden. Die implementierten Verfahren liefern bei der Nutzung des Simulators schlechte Ergebnisse, weil die verschiedenen Parameter der Filter und der Roboteransteuerung auf die realen Bedinungen eingestellt wurden. Da das System für den Wettkampfbetrieb beim Robocup ausgelegt

ist, werden diese Einschränkungen in Kauf genommen. Die wichtigste Erweiterung, welche zukünftig vorgenommen werden sollte, ist die Erkennung und Behandlung von Kollisionen. Da gerade die Interaktionen verschiedener Objekte auf dem Spielfeld für das Spielgeschehen interessant sind, sollte daher bei der Weiterentwicklung besonders dieser Schwerpunkt forciert werden.

Das Partikelfilter benötigt in der derzeitigen Konfiguration für eine genaue Vorhersage eine große Anzahl Partikel. Da die Nutzung von vielen Partikeln aufgrund der Echtzeitanforderungen des Systems nicht möglich ist, muss auf den Einsatz des Partikelfilters derzeit verzichtet werden. Bei Optimierung der Performance und Nutzung geeigneter Verfahren zur Bestimmung der Objektgeschwindigkeit, kann das Verfahren in der Zukunft aber relevant für die Vorhersage des Balls werden.

Weil fliegende Bälle können durch die in der Small Size League standardisierten Bildverarbeitungssoftware SSL-Vision nicht erkannt werden, wurde eine Softwarekomponente in *Sumatra* erstellt, welche dies leistet. Durch sie wird aufgrund der bei fliegenden Bällen entstehenden perspektivischen Verzerrung auf die Flugbahn geschlossen und die Beobachtungsdaten, welche an das Worldpredictor-Modul weitergeleitet werden entsprechend korrigiert. Die Funktionsweise dieser Komponente wurde durch Versuchsreihen auf einem realen Spielfeld nachgewiesen. Problematisch gestalten sich hier aufgrund der verrauschten Kameradaten Bälle, die über beide Spielfeldhälften fliegen.

Da sowohl für die benutzten probabilistischen Verfahren als auch für die Erkennung fliegender Bälle eine Vielzahl an Matrixoperationen durchgeführt werden muss, wurde eine eigene, hochperformante Matrixklasse implementiert. Durch diese konnte, je nach Operation, eine Zeitersparnis von 40-80% gegenüber der vorher genutzten *Jama*-Bibliothek erreicht werden.

Literaturverzeichnis

[1] *Moduli*. http://moduli.sourceforge.net/, Abruf: 24. Juni 2011

[2] Arndt, Holger: *UJMP: Universal Java Matrix Package*. http://www.ujmp.org/, Abruf: 30. April 2011

[3] Arndt, Holger: *The Universal Java Matrix Package: Everything is a Matrix*. München : Technische Universität, 2010

[4] Birkenkampf, Peter ; Dölle, Birgit ; Künemund, Maren ; Sauer, Marcel: *Prädiktion von Roboter- und Ball-Positionen in der RoboCup Small-Size-League. Theoretische Grundlagen verwendeter Verfahren und Modellierung von Objektbewegungen*. Duale Hochschule Baden-Wï¿½rttemberg Mannheim, 2011. – Studienarbeit

[5] Browning, Brett ; Bowling, Michael ; Veloso, Manuela: Improbability Filtering for Rejecting False Positives. In: *Proceedings of ICRA-02, the 2002 IEEE International Conference on Robotics and Automation*, 2002

[6] Dyer, Dan: *A Programmer's Guide to Random Numbers*. http://blog.uncommons.org/2008/04/03/a-java-programmers-\guide-to-random-numbers-part-1-beyond-javautilrandom/. Version: 2008, Abruf: 17.Juni 2011

[7] Hicklin, Joe ; Moler, Cleve ; Webb, Peter ; Boisvert, Ronald F. ; Miller, Bruce ; Pozo, Roldan ; Remington, Karin: *JAMA : A Java Matrix Package*. `http://math.nist.gov/javanumerics/jama/`, Abruf: 30. April 2011

[8] Loy, Marc ; Eckstein, Robert ; Wood, Dave ; Elliott, James ; Cole, Brian: *Java Swing*. Bd. 2. Sebastopol : O'Reilly Media, Inc., 2002. – ISBN 978–0–596–00408–8

[9] Marsaglia, G.: *Diehard Battery of Tests of Randomness*. Florida State University, 1995

[10] Papula, Lothar: *Mathematik für Ingenieure und Naturwissenschaftler*. Wiesbaden : Vieweg + Teubner, 2009. – ISBN 978–3–8348–0564–5

[11] Shirazi, Jack: *Java Performance Tuning*. `www.javaperformancetuning.com/tips/rawtips.shtml`, Abruf: 25.11.2010

[12] Srisabye, Jirat ; Wasuntapichaikul, Piyamate u. a.: *Skuba 2009 Extended Team Description*. Bangkok : Kasetsart University, 2009

[13] Tyler, Tim: *Sun Redefines Randomness*. `http://www.alife.co.uk/nonrandom/`, Abruf: 17. Juni 2011

[14] Ullenboom, Christian: *Java ist auch eine Insel: Programmieren mit der Java Standard Edition Version 6*. 8. Bonn : Galileo Press, 2009 `http://openbook.galileocomputing.de/javainsel8/`. – ISBN 978–3–8362–1371–4

[15] Williams, Thomas ; Kelley, Colin: *gnuplot 4.4. An Interactive Plotting Program*. `http://www.gnuplot.info/docs_4.4/gnuplot.pdf`. Version: 2010, Abruf: 25.Juni 2011

[16] Yonghai Wu, Xingzhong Q. u. a.: *Extended TDP of ZjuNlict 2009*. Zhejiang University, 2009

[17] Zickler, S. ; Laue, T. ; Birbach, O. ; Wongphati, M. ; Veloso, M.: SSL-Vision: The shared vision system for the RoboCup Small Size League. In: Baltes, J. (Hrsg.) ; Lagoudakis, M.G. (Hrsg.) ; Naruse, T. (Hrsg.) ; Shiry, S. (Hrsg.): *RoboCup 2009: Robot Soccer World Cup XIII*. Berlin : Springer, 2010 (Lecture Notes in Artificial Intelligence 5949)