

Datenbankprojekt WS19/20:
Distributed Streaming EKF SLAM
Fachgebiet Datenbanksysteme und
Informationsmanagement
Technische Universität Berlin

Jestram, Johannes Livert, Benedikt
`jestram@posteo.de` `benedikt.livert@gmx.de`
Paranskij, Mark
`mark.paranskij@gmail.com`

3. Februar 2020

Inhaltsverzeichnis

1	Einleitung	3
1.1	Motivation	4
1.2	Extended Kalman Filter	5
1.2.1	Prediction-Step	6
1.2.2	Update-Step	6
1.3	Simultaneous Localization and Mapping	6
2	Methodik	7
2.1	Verwendete Software	8
2.1.1	Apache Flink	8
2.1.2	InfluxDB	9
2.1.3	Colt	10
2.2	Victoria Park Datensatz	10
2.2.1	Messungen	10
2.2.2	Probleme mit den Daten	11
2.2.3	Ausreißer	12
2.3	Implementierung	13
2.3.1	Simulation mehrerer Fahrzeuge	13
2.3.2	Umsetzung in Apache Flink	14
2.3.3	EKF Implementierung	15
2.3.4	SLAM Implementierung	17
3	Evaluation	19
3.1	Qualität des EKF	20
3.2	Evaluation von SLAM	21
4	Zusammenfassung und Diskussion	22

Zusammenfassung

In verschiedenen technischen Disziplinen ist es nützlich, auf Grundlage des bisherigen Verhaltens eines Systems eine Vorhersage über dessen künftigen Zustand treffen zu können. Ein Beispiel dafür ist die Navigation: Mithilfe der bisherigen Bewegungsrichtung und der aktuellen Geschwindigkeit lässt sich die Position des Objektes zu einem zukünftigen Zeitpunkt schätzen. Eine Methode zum Schätzen eines Status ist der sog. Kalman Filter [1]. Ein weiteres interessantes und vor allem durch die Robotik geprägtes Problem ist Simultaneous Localization And Mapping (SLAM). Hierbei soll ein Objekt zeitgleich sowohl seine eigene Position bestimmen, als auch die ihm unbekannte Umgebung kartografieren. Eine fortgeschrittene Version des Kalman Filters, der Extended Kalman Filter (EKF) kann um weitere Funktionalität erweitert zu werden um SLAM zu gewährleisten. [2] In diesem Projekt simulieren wir einen EKF mit SLAM basierend auf dem Victoria Park Datensatz. Dieser besteht aus Daten, die von einem Fahrzeug aufgezeichnet wurde, was ca. 25 Minuten durch den Victoria Park in Sydney gefahren ist. Die Implementierung geschieht in Apache Flink, wobei besonderer Fokus auf die parallele Ausführung des EKF für mehrere simulierte Fahrzeuge gelegt wird. Außerdem werten wir unsere Ergebnisse dahingehend aus, welche Schwierigkeiten sich hinsichtlich Komplexität bei der Implementierung von SLAM ergeben und welche Umstände zu schlechten Vorhersagen des Kalman Filters führen. Hierbei zeigt sich, dass die Komplexität von SLAM maßgeblich von der Anzahl gemessener Objekte in der zu kartographierenden Umgebung abhängt. Verschiedene Ansätze zur Komplexitätsreduzierung sind zu diskutieren.

1 Einleitung

Um eine möglichst verlässliche Vorhersage treffen zu können, müssen verschiedene Aspekte des betreffenden Systems gemessen werden. Da Messungen jedoch stets, wenn auch in unterschiedlichem Maße, fehlerbehaftet sind, sollten diese Messfehler auch in die Betrachtung des Systemzustandes einbezogen werden. Weiterhin unterscheidet sich die Komplexität der Methoden zur Vorhersage des zukünftigen Zustandes je nach betrachtetem System. Möglicherweise wird der Zustandsübergang des Systems durch komplexe Funktionen mit vielen Parametern dargestellt. Zuletzt erscheint es auch als sinnvoll, die getroffene Vorhersage anhand neuer Messungen zu überprüfen und gegebenenfalls anzupassen. Eine Methode, um solche Zustandsübergänge vorherzusagen und zu überprüfen, ist der sogenannte Kalman Filter [1]. Der Kalman Filter besteht im wesentlichen aus 2 Schritten.

Einem sog. *Prediction-Step*, bei dem der Status in einem künftigen Zeitpunkt mithilfe von internen Messungen geschätzt wird. Die daraus resultierende Schätzung wird dann im sog. *Update-Step* mithilfe einer externen Messung aktualisiert.

Ein weiteres, vor allem durch die moderne Robotik geprägtes, Problem ist das Kartographieren einer Umgebung. Zum einen muss der autonome Roboter stets seine eigene Position innerhalb der unbekannten Umgebung kennen, zum anderen soll er eben diese Umgebung erkunden und aufzeichnen. Die Erkundung der Umgebung basiert in der Regel auf Messungen mit eingebauten Sensoren, daher ist die Positionsbestimmung der Erkannten Objekte stets relativ zur eigenen Position. Andererseits ortet sich der messende Roboter innerhalb der soeben kartierten Umgebung. Die Genauigkeit der eigenen Positionsbestimmung und der erkundeten Objekte in der Umgebung hängen also in beide Richtungen unmittelbar zusammen. Beide Messungen unterliegen jedoch Fehlern, möglicherweise gibt es sogar Aussetzer bei der Erfassung der Sensordaten. Die Problemstellung erinnert an ein Henne-Ei Problem. Die beschriebene Problemstellung wird im englischen als ‘Simultaneous Localization and Mapping’ (SLAM) bezeichnet.

Es ist möglich, den Kalman Filter zur Lösung des SLAM Problems zu nutzen [2]. Wie im Folgenden beschrieben werden wird, ist die Berechnung der Vorhersagen und deren Vergleiche mit den Messungen ein rechenaufwändiger Algorithmus.

1.1 Motivation

In diesem Projekt wird als Datengrundlage der Victoria Park Datensatz betrachtet. Diesem Datensatz liegt die Bewegung eines Fahrzeugs durch besagten Park zugrunde. Der Zustand (d.h. die Position und Odometriedaten) eines Fahrzeugs wird durch eingebaute Sensoren sowie GPS regelmäßig gemessen. Darüber hinaus ermittelt das Fahrzeug mittels eines Lasersensors vor ihm liegende Objekte.

Basierend auf diesen Messungen soll der EKF/SLAM Algorithmus implementiert werden. Da die reine Implementierung des Verfahrens basierend auf dem gegebenen Datensatz bereits erfolgt ist [3], wird es im Rahmen dieses Projektes darum gehen, wie ein solche Algorithmus parallelisiert werden kann. Um das EKF-SLAM Verfahren skalierbar zu machen, ist es folglich interessant dafür notwendigen Berechnungen zu parallelisieren. Es geht zum einen darum zu analysieren, inwiefern die Berechnungen selbst parallelisierbar sind. Weiterhin sind in den Anwendungsgebieten des Kalman Filter auch Szenarien denkbar, in denen Daten von mehr als einem System (z.B. Fahr-

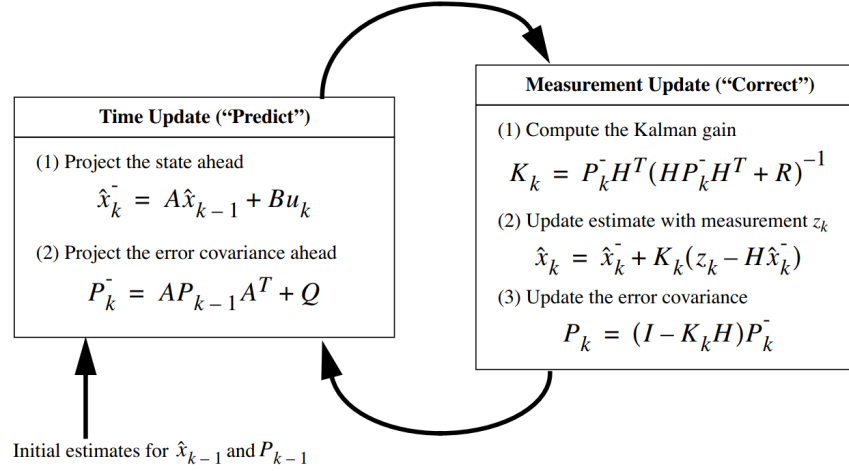


Abbildung 1: Extended Kalman Filter Algorithmus [5].

zeug) verarbeitet werden sollen [4]. Für solche Szenarien stellt sich die Frage, ob und wie Daten von mehreren Fahrzeugen möglichst effektiv verarbeitet werden können.

Dieses Projekt zielt folglich darauf ab, die parallele Implementierung des EKF SLAM Algorithmus umzusetzen und zu evaluieren. Dabei wird die Software für parallele und verteilte Verarbeitung von Datenströmen Apache Flink ¹ eingesetzt.

1.2 Extended Kalman Filter

Der Kalman Filter wurde im Jahr 1960 von Rudolph E. Kalman im Rahmen seines Artikels "A New Approach to Linear Filtering and *Prediction-Step* Problems" [1] vorgestellt. Er ist ein Ansatz um Zustände zu schätzen. Der Kalman Filter minimiert die Fehlervarianz der Schätzung. Der grundlegende Algorithmus funktioniert wie folgt [Abb. 1]: Zunächst wird im sog. *Prediction-Step* der Zustand eines Systems zu einem Zeitpunkt t auf Grundlage vorhergegangener Zustandsmessungen und einer Zustandsübergangsfunktion geschätzt. Diese Schätzung nennt man A-Priori Schätzung. Im sog. *Update-Step* (auch *Correction-Step*) wird die A-Priori Schätzung nach dem Erhalt neuer Messdaten korrigiert und der Algorithmus beginnt von vorne. Resultat des *Update-Step* ist die sog. A-Posteriori Schätzung.

¹<https://flink.apache.org/>

Eine attraktive Eigenschaft dieses Algorithmus ist seine rekursive Natur; zur Berechnung des geschätzten neuen Zustands sind nur die Daten der vorherigen Messung nötig, da sich in dieser alle vorherigen Messungen zusammengefasst wiederfinden. Der Extended Kalman Filter unterscheidet sich vom Diskreten Kalman Filter darin, dass die Zustandsübergänge durch nichtlineare Funktionen abgebildet werden. Die Nichtlinearität verkompliziert den Algorithmus, wie im folgenden gezeigt wird.

1.2.1 Prediction-Step

Der Zustand eines Systems kann über beliebig viele Parameter dargestellt werden. Um einen Zustand \hat{x}_t in einen Zustand \hat{x}_{t+1} zu überführen, muss für jeden Parameter eine Funktion definiert werden, die diesen Übergang abbildet. Im Falle eines Fahrzeuges erfolgt dies über die Erstellung eines Bewegungsmodells. Auf Grundlage der Sensorik und vorhandenen Messwerte muss das Bewegungsmodell den Zustand, korrekt und dargestellt in die gewünschten Metriken, berechnen.

Doch das Bewegungsmodell ist nicht der einzige Einflussfaktor für den Zustandsübergang: Messfehler und äußere Einflussfaktoren sorgen für Ungenauigkeiten. Diese Ungenauigkeiten werden im Extended Kalman Filter berücksichtigt.

1.2.2 Update-Step

Der im Prediction-Step geschätzte Zustand wird nun basierend auf neuen, externen Beobachtungen des Zustandes korrigiert. Jedoch ist es nicht so, dass einfach die externe Beobachtung als Schätzung verwendet wird. Vielmehr wird die Korrektur um den sog. *Kalman Gain* gedämpft. Dieser berechnet sich sowohl aus Messunsicherheiten, als auch aus einer Kovarianzmatrix, die die Abhängigkeiten zwischen den einzelnen Parametern des Zustands abbildet. Die genaue Berechnung der notwendigen Variablen wird in Abschnitt 2.3.3 erläutert.

1.3 Simultaneous Localization and Mapping

Simultaneous Localization And Mapping (SLAM) ist ein Verfahren in der Robotik. Das Ziel von SLAM ist, dass ein Aktor zu jedem beliebigen Zeitpunkt seine eigene Position bestimmen kann, aber zeitgleich die ihm unbekannte Umgebung kartographiert. In unserem Fall ist der Aktor ein Fahrzeug und die Umgebung der Victoria Park. Die grundlegende Problemstellung von SLAM erinnert an ein Henne-Ei-Problem. Wie ortet sich ein Aktor, wenn

er die Umgebung nicht kennt und wie erkundet er die Umgebung, wenn er seine eigene Position nicht kennt? Die Lösung liegt im Kalman Filter. Der Zustand, welcher vom Filter geschätzt werden soll, wird um die sog. Landmarks, also interessante Objekte in der Umgebung, erweitert. Pro Landmark werden dem Zustand zwei Variablen hinzugefügt, nämlich die jeweilige X und Y Koordinate. Auch die zum Zustand gehörende Kovarianzmatrix wird dementsprechend erweitert.

Dieser Schritt ist jedoch sehr komplex. Zum einen müssen der Zustandsvektor und die Kovarianzmatrix dynamisch wachsen und, wie später beschrieben, theoretisch auch schrumpfen können. Zum anderen erreicht die Kovarianzmatrix sehr schnell hohe Dimensionen, was Berechnungsschritte im Kalman Filter zu sehr aufwendigen Operationen macht. Bei unserer Simulation erreichte die Kovarianzmatrix schon nach ca. 1500 Datenpunkte eine Größe von 1000×1000 und verlangsamte dadurch die Ausführung signifikant. Aber nicht nur die Repräsentation des Zustandes ändert sich bei EKF-SLAM, auch der *Prediction-* und *Update-Step* des EKF verändern sich. Im *Prediction-Step* wird im zuvor beschriebenen Fall die Position und Fahrtrichtung des Fahrzeuges geschätzt. Bei EKF-SLAM erweitert sich diese Schätzung um die zum Fahrzeug relativen Positionen der bereits gesehenen Landmarks.

Im *Update-Step* wird bei EKF-SLAM nicht mehr auf eine GPS Messung zurückgegriffen. Stattdessen wird das Erkennen von nahen Landmarks als externe Messung verwendet. Aus dem Wissen über die Position zuvor gesehener Landmarks und der gemessenen relativen Position des Fahrzeuges zu diesen Landmarks, kann eine absolute Position des Fahrzeuges bestimmt werden, mithilfe derer die zuvor geschätzte Position korrigiert wird.

2 Methodik

Im folgenden werden zunächst die benutzten softwareseitigen Werkzeuge zur Umsetzung erklärt (Abschnitt 2.1). An dieser Stelle wird der Fokus auf den für das Projekt relevanten Funktionalitäten der jeweiligen Software liegen. Anschließend werden wir ausführlich auf die bereitgestellten Daten eingehen und deren Spezifika erläutern, da dieses Wissen Grundlage der Implementierung des Algorithmus ist (Abschnitt 2.2). Als nächstes werden wir ausgewählte Designentscheidungen der Implementierung vorstellen (Abschnitt 2.3). Das beinhaltet unter anderem die konkrete Umsetzung des Kalman Filter für die gegebene Problemstellung. Die Ergebnisse bzw. die Ausgabe unserer Implementierung werden im Anschluss dokumentiert,

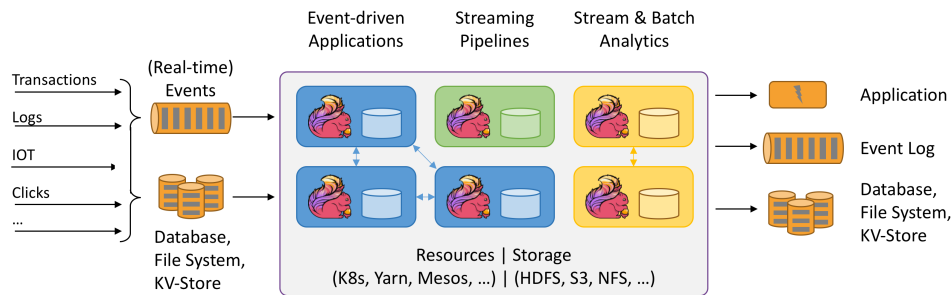


Abbildung 2: Überblick über die Funktionen von Apache Flink [7].

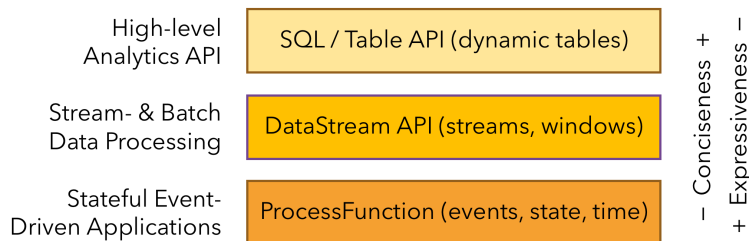


Abbildung 3: Flink's APIs auf verschiedenen Abstraktionsniveaus [8].

analysiert und bewertet(Abschnitt 3).

2.1 Verwendete Software

2.1.1 Apache Flink

Flink ist eine Software zur verteilten und parallelen Verarbeitung von Stream- oder Batchdaten. Es ist aus dem Stratosphere Projekt² [6] der TU-Berlin hervorgegangen und ein Top-Level-Projekt der Apache Foundation³. Die Software ist per Open Source Lizenz verfügbar und aufgrund der hohen Performanz [6] ein beliebtes Werkzeug für verteilte und skalierbare Datenverarbeitung.

Flink basiert auf der Java Virtual Machine (JVM) und ist in Scala und Java implementiert⁴. Der Funktionsumfang umfasst die parallele und verteilte Verarbeitung von Datenströmen, sowie einmaligen (in der Regel sehr

²<http://stratosphere.eu>

³<https://apache.org>

⁴<https://github.com/apache/flink>

rechenaufwändigen) sog. Batch-Aufträgen [Abb. 3]. Dazu stehen verschiedene Application Programming Interfaces (APIs) auf unterschiedlichen Abstraktionsebenen zur Verfügung [Abb. 2]. Flink unterscheidet Arbeitsaufträge in Streaming- und Batch-Aufträge. Diese unterscheiden sich in der Art und Weise der Parallelisierung der Berechnungen: bei Streams müssen beispielsweise Aggregationen mit einem Fenster versehen werden, da man sie bei einem potentiell unendlich langen Datenstrom nicht auf die gesamten Daten ausführen kann. Bei Batch-Daten ist das im Gegensatz möglich.

Da das Ziel dieses Projektes die Anwendung von EKF-SLAM auf mehrere Fahrzeuge in Echtzeit ist, wird vor allem die *DataStream API* genutzt. Sie liefert vorgefertigte Funktionalitäten für viele übliche Aufgaben in der Verarbeitung von Datenströmen. Diese umfassen unter anderem die automatisierte Parallelisierung der Verarbeitung von Datenströmen. Weiterhin ist es möglich, eingehende Daten mit sog. Schlüsseln (engl. *keys*) zu versehen und die weitere Verarbeitung der Daten im Folgenden abhängig von dem zugeordneten Key zu machen. Dies erfolgt über die Klassen *KeyedDataStream* und *KeyedDataPoint*. Ein *KeyedDataStream* verarbeitet generische Datentupel, welche jeweils mit einem Zeitstempel und einem Schlüssel versehen sind. Dabei liefern *KeyedDataPoint* eine Art Blaupause für den Inhalt eines einzelnen Datentupels.

Mit Hilfe solcher geschlüsselter Datenströme ist es möglich, für jeden Schlüssel einen eigenen Zustand (engl. *state*) durch die Klasse *ValueState* zu definieren. Folglich ist es mit Schlüsseln und Zustand möglich, Daten und darauf basierende Berechnungen nach deren Quelle oder Zugehörigkeit klar voneinander zu trennen.

2.1.2 InfluxDB

InfluxDB⁵ ist eine für Zeitreihendaten optimierte Datenbank. Die Interaktion erfolgt in einer Anfragesprache ähnlich zu SQL. Es existieren APIs für verschiedene Programmiersprachen, unter anderem Java.

Jedes zu speichernde Datentupel muss mit einem Zeitstempel versehen sein. Wie in Flink ist auch hier möglich, eingehende Daten anhand eines Schlüssels zu unterscheiden. Daten können in unterschiedliche Datenbanken gespeichert werden und innerhalb einer Datenbank nach Messreihen unterschieden werden. Einzelne Messungen werden in sog. Feldern gespeichert. Anzahl und Typ des Inhalts der einzelnen Felder können durch den Nutzer festgelegt werden. Dank der genannten Funktionen bietet InfluxDB eine

⁵<https://www.influxdata.com/products/influxdb-overview>

leistungsfähige Lösung für die anwendungsspezifische Speicherung von Echtzeitdaten.

2.1.3 Colt

Bei der Implementierung des Extended Kalman Filter müssen auch Operationen auf Matrizen programmiert werden. Java bietet in der Version 1.8, die wir verwendet haben, keine native Möglichkeit, auf Matrizen und Vektoren (außer in Form von Listen oder Arrays) zu arbeiten. Dafür benötigt man eine externe Bibliothek. Wir haben uns für die Colt 1.2 Library ⁶ entschieden, welche am CERN entwickelt wurde und alle für uns notwendigen Matrix- und Vektoroperationen durch Colt zur Verfügung gestellt werden.

2.2 Victoria Park Datensatz

Die Inhalte dieses Datensatzes ⁷ wurden im Jahr 2006 im Victoria Park in Sydney gesammelt. Es wurde die Bewegung eines Fahrzeugs durch den Park aufgezeichnet und währenddessen vom Fahrzeug Objekte in der nahen Umgebung erfasst. Die Dauer der Aufzeichnungen ist rund 25 Minuten.

2.2.1 Messungen

Es existieren drei verschiedene Quellen für Messungen. Die internen Sensordaten liefern die Odometrie. Diese setzt sich aus dem Lenkwinkel α , sowie der Geschwindigkeit v , gemessen am linken Hinterrad, zusammen. Außerdem wurde regelmäßig die Position des Fahrzeuges via GPS bestimmt. Die Messung der in der Umgebung befindlichen Objekte erfolgt mittels eines Lasersensors.

Neben den rohen Messungen existieren unterschiedlich aufgearbeitete Datensätze. Diese unterscheiden sich in der Anzahl der enthaltenen Datenpunkte sowie der Formatierung der Werte. Die Odometriedaten werden stets gemeinsam im Format Zeitstempel, Lenkwinkel, Geschwindigkeit angegeben. Es existiert eine Datei mit rund 66.000 Messungen und eine mit rund 4.000.

Auch die GPS-Daten werden in mehreren Versionen geliefert. Grundsätzlich gehen die GPS-Messungen in regelmäßigen Abständen von 200ms ein. Jedoch sind teils starke zeitliche Lücken in den Messungen vorhanden, so dass über einen längeren Zeitraum keine Positionsbestimmung erfolgt. Auch enthält der Datensatz einige Ausreißer. Diese erkennt man daran, dass das

⁶<https://dst.lbl.gov/ACSSoftware/colt>

⁷https://www.mrpt.org/Dataset_The_Victoria_Park

Auto, falls diese Messungen korrekt wären, von einem zum nächsten Punkt eine viel zu große Distanz zurücklegen würde. Beide genannten Mängel wurden auch von den Autoren des Datensatzes erkannt und benannt.

Der Lasersensor ist über der vorderen Stoßstange des Fahrzeugs positioniert und vermisst einen 180-Grad Winkel. Dieser Blickwinkel wird mit 361 Messpunkten von rechts nach links (in Fahrtrichtung) abgetastet. Für jeden einzelnen Messpunkt speichert der Laser den Abstand des gemessenen Objektes. Die Messungen sind für Objekte, die sich höchstens 80m entfernt befinden, hinreichend präzise.

2.2.2 Probleme mit den Daten

Im Laufe des Projektes stellte sich der Umgang mit den Daten als das zeit- aufwändigste Problem heraus. Vor allem die unterschiedlichen Versionen desselben Datensatzes sorgten regelmäßig für Probleme.

Auf der Website des Datensatzes ⁸ befindet sich ein Datensatz bestehend aus 4832 Datenpunkten, der laut Dokumentation mit Hilfe eines externen Tools, des Rawlog Viewers ⁹, als Textdatei extrahiert und visualisiert werden kann. Diese 4832 Datenpunkte setzen sich zur Hälfte aus Inkrementen zwischen zwei Messungen in Richtung der X-Achse, Y-Achse und der Drehung des Fahrzeuges zusammen, sowie 2416 Lasermessungen. Problematisch bei diesem Datensatz ist hierbei jedoch die Tatsache, dass die komplexe Geometrie zwischen Geschwindigkeitsmessung und Laser- und GPS-Messung am Fahrzeug nicht näher dokumentiert ist, was zu einem hohen Maß an Ambiguität bezüglich der Verarbeitung des Datensatzes geführt hat.

Darüber hinaus befindet sich auf der originalen Website für den Datensatz der Publikation ¹⁰ ein Datensatz im ASCII Format. Dieser ist aus 69941 Datenpunkten zusammengesetzt, von denen 61763 Odometrie Daten sind, die jeweils einen Zeitstempel, die Geschwindigkeit in Metern pro Sekunde und den Winkel der Vorderachse des Fahrzeuges beinhalten. Dazu kommen 948 Datenpunkte für GPS-Messungen, sowie 7230 Lasermessungen.

Ebenfalls befindet im originalen Download ¹¹ ein MatLab Datensatz, den wir mit Hilfe eines Python-Skriptes ausgelesen haben, welcher sich aus 61945 gleich zusammengesetzten Odometrie Datenpunkten, 4466 GPS-Messungen und 7249 Lasermessungen, insgesamt also 73660 Datenpunkten, ergibt.

⁸https://www.mrpt.org/Dataset_The_Victoria_Park

⁹<https://www.mrpt.org/list-of-mrpt-apps/rawlogviewer>

¹⁰http://www-personal.acfr.usyd.edu.au/nebot/victoria_park.htm

¹¹https://www.mrpt.org/Dataset_The_Victoria_Park

Aufgrund eines Mangels an definitiver Information und inkohärenter Datensätze wurde der Projektfortschritt lange durch das Auswerten dieser Datensätze hinausgezögert. Hierbei haben wir versucht durch die Visualisierung der Daten diese zu validieren. Das führte bei dem kleinen Datensatz, bestehend aus 4832 Datenpunkten, zu keinem Erfolg, da das Format der Inkremente nicht ausreichend dokumentiert ist, und wir erst mit viel Recherche und durch interne Informationen aus dem Rawlog Viewer ¹² auf das zugrundeliegende Bewegungsmodell des Datensatzes ¹³ gestoßen sind.

In visualisierter Form ähnelten lediglich die GPS-Messungen des MatLab Datensatzes dem erwarteten Pfad des Fahrzeugs, weshalb wir uns entschlossen haben den EKF auf den GPS Daten aus diesem Datensatz auszuführen um das Projekt voranzutreiben. Gesamtheitlich betrachtet machten die genannten Probleme mit dem Datensatz circa die Hälfte der Projektarbeit aus. Selbst am Ende des Projektes kam es noch dazu, dass wir verschiedene Eingangsdaten ausprobieren mussten, um valide und funktionierende Ergebnisse zu erzielen, da beispielsweise der ASCII Datensatz gravierend vom MatLab Datensatz abweicht und aus unserer Perspektive als falsch anzusehen ist. Die GPS-Messungen der beiden Datensätze weichen gänzlich voneinander ab und die Odometrie-Daten, durch den gleichen EKF-Algorithmus verarbeitet, resultieren in komplett verschiedenen Ergebnissen.

2.2.3 Ausreißer

Wie bereits aus der Quelle für den Datensatz ¹⁴ hervorgeht, sind die GPS-Messungen des MatLab Datensatzes fehlerhaft und unvollständig und wurden daher in der ursprünglichen Implementierung nicht weiter zur Berechnung von SLAM verwendet. Da es aufgrund der unterschiedlichen Datensätze dennoch notwendig wurde den EKF unsererseits in der einfachsten Form mit GPS zu implementieren, mussten wir dennoch auf den MatLab Datensatz mit Ausreißern und Lücken zurückgreifen. Damit es trotzdem vergleichbare und repräsentative Ergebnisse für den EKF mit GPS geben konnte, entschieden wir uns dazu, den GPS Datensatz um Ausreißer bereinigen. Dabei verwendeten wir eine Heuristik, bei der Datenpunkte, die eine zu hohe Distanz zu vorherigen GPS Messungen aufweisen, sowie in kleinen Gruppen von wenigen Tupeln vorkommen, aussortiert werden. Insgesamt haben wir 21 Datenpunkte aussortiert und den Datensatz dadurch um die signifikantesten

¹²<https://www.mrpt.org/list-of-mrpt-apps/rawlogviewer>

¹³https://www.mrpt.org/tutorials/programming/odometry-and-motion-models/probabilistic_motion_models

¹⁴http://www-personal.acfr.usyd.edu.au/nebot/victoria_park.htm

Ausreißer bereinigt.

2.3 Implementierung

2.3.1 Simulation mehrerer Fahrzeuge

Da es in diesem Projekt auch um die Parallelisierung der Verarbeitung von Daten unterschiedlicher Fahrzeuge geht, haben wir den vorhandenen Datensatz mehrfach repliziert. So kann die Berechnung mehrerer Systeme und deren Zustände simuliert und mit Hilfe von Flink parallelisiert werden. Zur Simulation haben wir unseren Datensatz dupliziert und außerdem einen neuen Datensatz erzeugt, in dem die Reihenfolge der Messwerte umgekehrt wurde. Im einfachsten Fall wurden zwei Datensätze eingelesen.

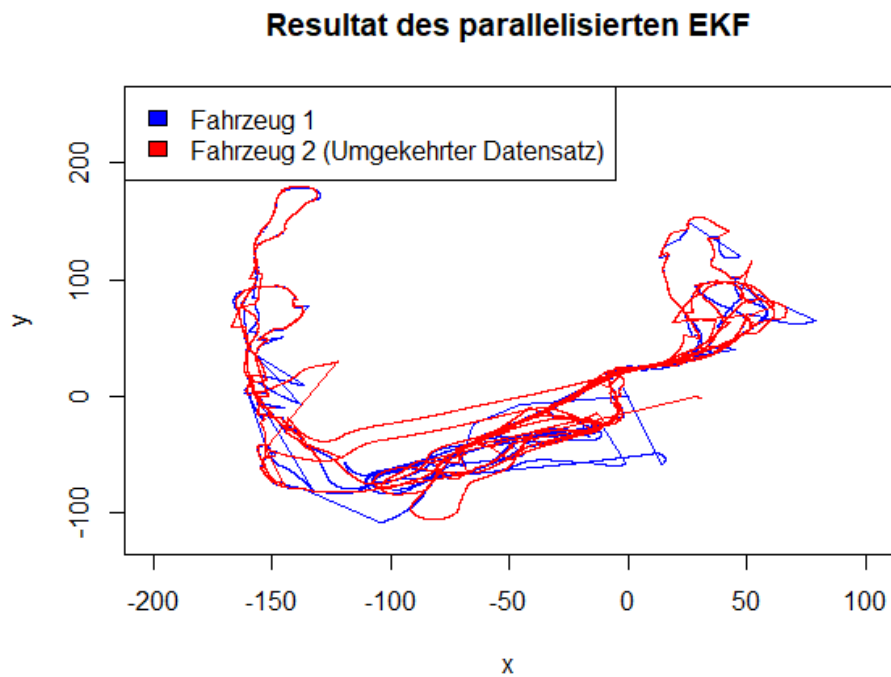


Abbildung 4: Output des parallelisierten EKF mit mehreren simulierten Fahrzeugen.

2.3.2 Umsetzung in Apache Flink

Da die Nutzung des Kalman Filters vor allem vor dem Hintergrund von kontinuierlich, über einen langen Zeitraum eintreffenden Datenströmen sinnvoll ist, fiel die Entscheidung auf einen Streaming-Auftrag.

Abgesehen von der Umsetzung des EKF SLAM Algorithmus bestand die Kernherausforderung beim Implementieren in der Parallelisierung der Berechnungen. Dazu mussten zum die Daten der unterschiedlichen Fahrzeuge bei der Verarbeitung dem entsprechenden Fahrzeug zugeordnet werden. Da jedes Fahrzeug einen eigenen, disjunkten Zustand hat, dürfen die eingehenden Daten nicht vermischt werden. Zur Umsetzung dieser Unterscheidung dienten uns vor allem KeyedDataStreams. Jedes Fahrzeug verfügt über eine eigene ID. Diese ID wird dann zum Schlüssel des KeyedDataStreams, sodass die interne Zuordnung der Eingangsdaten eindeutig und korrekt ist. Die Berechnungen können durch die Schlüssel also für jedes Fahrzeug parallelisiert werden.

In Apache Flink werden benutzerdefinierte Funktionen (*User Defined Functions, UDFs*) mittels einer *map()*-Funktion auf einen Datenstrom angewendet. Das bedeutet, dass die Funktion ein Tupel nach dem anderen übergeben bekommt. Dies birgt die Problematik, dass die Tupel des *Prediction-Step* und die Tupel des *Update-Step* unterschieden werden müssen. Außerdem müssen vorhergegangene Rückgabewerte, die zur Berechnung des neuen Zustandes notwendig sind, persistiert werden. Die erste Problematik wurde gelöst, indem jedem KeyedDataPoint ein Parameter beim Parsing übergeben wird. So erhalten beispielsweise Odometrie Datenpunkte den bezeichner "odo". Das Problem der Persistierung zwischen einzelnen Map-Schritten wird dadurch behandelt, dass ein *ValueState* erzeugt wird, der den vorherigen Zustand des Fahrzeuges als Vektor, sowie die Kovarianzmatrix und den Zeitstempel des vorherigen Tupels beinhaltet, um die nachfolgenden Berechnungen zu ermöglichen.

Weiterhin kann das Einlesen der Daten eines Fahrzeuges nicht parallelisiert werden. Andernfalls würde die Reihenfolge der Datenpunkte vertauscht werden und der Algorithmus würde falsche Ergebnisse liefern. Dieses Problem liegt allerdings auch in der Natur der Simulation. Wir lesen die Daten lediglich aus einer CSV-Datei aus, sodass von Beginn an alle Werte vorliegen. In einem realen Szenario würden die Daten erst zur Laufzeit generiert werden, sodass kein Bedarf an Parallelisierung besteht. Daten von verschiedenen Fahrzeugen werden in verschiedenen Streams parallel eingelesen. Die Ausführung des EKF selbst ist nicht zu parallelisieren, da jeder Schritt des EKF auf dem Resultat eines oder mehrerer vorhergegangenen Schritte ba-

siert.

Allerdings kann die Berechnung der jeweiligen EKF-Schritte von verschiedenen Fahrzeugen parallel ausgeführt werden. Da wir alle Datenpunkte aller Fahrzeuge in einem (Keyed-) Stream verarbeiten, werden die *map()*-Funktionen parallel ausgeführt. Flink verarbeitet Datenpunkte mit verschiedenen Keys automatisch in verschiedenen Threads. Die Positionsbestimmung verschiedener Fahrzeuge erfolgt also parallel und vollkommen unabhängig voneinander. Wir konnten feststellen, dass die Ausführungszeit bei einem und 4 Fahrzeugen konstant blieb.

Abbildung 4 zeigt das Ergebnis der Simulation von zwei parallel Fahren- den Fahrzeugen.

2.3.3 EKF Implementierung

Prediction-Step und Bewegungsmodell Für den reinen EKF, der eine GPS-Messung als Grundlage des *Update-Step* verwendet, initialisierten wir den Filter mit folgenden Werten und Funktionen. Grundlage für die Umsetzung stellten [3], sowie [9] dar.

Der Zustand des Fahrzeugs wird durch die x- und y-Koordinaten sowie den Winkel des Fahrzeuges zur x-Achse des Koordinatensystems angegeben.

$$\hat{x}_t = \begin{pmatrix} x_t \\ y_t \\ \phi_t \end{pmatrix} \quad (1)$$

Da sich der Sensor zur Geschwindigkeitsmessung am linken Hinterrad befindet, muss die Messung noch auf die Fahrzeugmitte zentriert werden.

$$v = \frac{v_{Messung}}{1 - \tan(\alpha) * \frac{H}{L}}$$

mit

$$v = \text{Fahrzeuggeschwindigkeit} \quad (2)$$

$$v_{Messung} = \text{Geschwindigkeit am linken Hinterrad}$$

$$H = \text{Achsabstand Sensor zur Fahrzeugmitte} = 0,76m$$

$$L = \text{Abstand Hinterachse zur Vorderachse} = 2,83m$$

Die Zustandsvorhersage \hat{x}_{t+1}^- ergibt sich aus einem Bewegungsmodell, das anhand der vergangenen Zeit ΔT zwischen dem letzten *Update*- und

dem jetzigen *Prediction-Step* die neue Position des Fahrzeuges schätzt.

$$\begin{aligned}\hat{x}_{t+1}^- &= f(\hat{x}_t, u_t) = \begin{pmatrix} x_{t+1} \\ y_{t+1} \\ \phi_{t+1} \end{pmatrix} \\ &= \begin{pmatrix} x_t + \Delta T(v \cos(\phi_t) - \frac{v}{L} \tan(\alpha)) * (a \sin(\phi_t) + b \cos(\phi_t)) \\ x_t + \Delta T(v \sin(\phi_t) + \frac{v}{L} \tan(\alpha)) * (a \cos(\phi_t) - b \sin(\phi_t)) \\ \phi_t + \Delta(\frac{v}{L} \tan(\alpha)) \end{pmatrix}\end{aligned}\quad (3)$$

Während des EKF wird die Kovarianzmatrix P in jedem Durchlauf verändert. Wir initialisieren sie mit 0.

$$P_0 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}\quad (4)$$

Für die Linearisierung des Bewegungsmodells, d.h. zur Aktualisierung von P , im EKF wird außerdem noch die Jakobi-Matrix des Bewegungsmodells benötigt.

$$\begin{aligned}F_{ekf} &= \frac{\partial f}{\partial \hat{x}} \\ &= \begin{pmatrix} 1 & 0 & -\Delta T(v \sin(\phi_t) + \frac{v}{L} \tan(\alpha) * a \cos(\phi_t) - b \sin(\phi_t)) \\ 0 & 1 & \Delta T(v \cos(\phi_t) - \frac{v}{L} \tan(\alpha) * a \sin(\phi_t) + b \cos(\phi_t)) \\ 0 & 0 & 1 \end{pmatrix}\end{aligned}\quad (5)$$

Update-Step Im *Update-Step* werden die Ergebnisse des *Prediction-Step* mit Hilfe der GPS-Messung aktualisiert. Dazu muss zunächst das Beobachtungsmodell definiert werden. Dessen Aufgabe ist es, die Messungen des GPS-Sensors auf das Format des Zustandsvektors abzubilden. Im vorliegenden Fall werden x- und y-Koordinate im passenden Format gemessen. Der Winkel ϕ wird durch den GPS-Sensor nicht gemessen und daher auch nicht im Beobachtungsmodell abgebildet. Der auf Grundlage des Beobachtungsmodells berechnete Kalman Gain K_t wird aufgrund seiner Dimensionalität dennoch ϕ beeinflussen.

$$z_{ekf} = h(\hat{x}) = \begin{pmatrix} x_{gps} \\ y_{gps} \end{pmatrix}\quad (6)$$

Zwar ist das Beobachtungsmodell linear, da jedoch aufgrund des nicht-linearen Bewegungsmodells der Extended Kalman Filter verwendet wird,

muss auch vom Beobachtungsmodell die Jakobi-Matrix gebildet werden.

$$H(\hat{x}) = \frac{\partial h}{\partial \hat{x}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad (7)$$

$$R = \begin{pmatrix} 0,5 & 0 \\ 0 & 0,5 \end{pmatrix} \quad (8)$$

2.3.4 SLAM Implementierung

Der Unterschied zum EKF ohne SLAM-Anteil liegt vor allem darin, dass der *Update-Step* nun auf Grundlage der in der Umgebung georteten Objekte stattfindet. Der Zustandsvektor \hat{x} des Systems enthält nun zusätzlich zur Position und Fahrtrichtung des Fahrzeuges die Koordinaten jedes gemessenen Objektes - im Falle des Victoria Parks vor allem Bäume.

$$\hat{x} = \begin{pmatrix} x_{fahrzeug} \\ y_{fahrzeug} \\ \phi \\ x_{objekt-1} \\ y_{objekt-1} \\ \dots \\ \dots \\ x_{objekt-n} \\ y_{objekt-n} \end{pmatrix} = \begin{pmatrix} x_i \\ y_i \\ \phi \\ x_L \\ y_L \end{pmatrix} \quad (9)$$

Ableiten der Objektposition aus den Lasermessungen Die Lasermessungen sehen in jedem Datensatz vergleichbar aus und bestehen jeweils aus 361 einzelnen Abstandsmessungen mit einer Reichweite von 80 Metern. Sie müssen gesondert in Bäume umgerechnet werden, wobei ein Baum aus mehreren nebeneinanderliegenden Messungen bestehen kann. Dazu müssen zunächst ungültige Messpunkte, die eine zu geringe oder zu große Messentfernung aufweisen, sowie ungültige Messpunkte, die bei jeder Lasermessung zu Beginn der Messserie auftreten, auf 0 gesetzt werden, damit lediglich gültige Messungen übrig bleiben. Anschließend werden zusammenhängende Messpunkte, die einem individuellen Baum zugesprochen werden, der eine bestimmte Distanz zum nächstgelegenen Baum aufweist, gruppiert. Daraufhin können aus einer Menge von Messpunkten einzelne Bäume mit Koordinaten und Durchmesser gewonnen werden. Hierzu wird aus den beiden äußersten Messungen, die dem selben Baum zugeordnet werden, dessen Mittelpunkt errechnet.

Als Teil des EKF SLAM Algorithmus wird eine Zuordnung von beobachteten Bäumen zu bereits bekannten Bäumen des Fahrzeugs vorgenommen. Hierbei wird ebenfalls entschieden, ob der Zustandsvektor um eine neue Beobachtung erweitert wird. Das ist der Fall, wenn der entsprechende Baum noch nie beobachtet worden ist. Diese Abfrage wurde heuristisch mithilfe einer Methode basierend auf Distanz zwischen beobachteten und existierenden Bäumen implementiert. Da die Performanz des SLAM Algorithmus maßgeblich von dieser Methode abhängt, wird diese in Abschnitt 3.2 diskutiert.

Prediction-Step Das Bewegungsmodell des Fahrzeuges ändert sich im Vergleich zum EKF nicht. Die Funktion $f(\hat{x}_t, u_t)$ wird allerdings um Funktionen für die Abbildung eines Objektes vom Zeitpunkt t zum Zeitpunkt $t+1$ ergänzt. Die Objekte werden zur Vereinfachung als unbeweglich angenommen, daher sind die neue Abbildungsfunktion 10, wie auch die zugehörige Jakobi Matrix (11) trivial.

$$\hat{x}_{t+1}^- = f(\hat{x}_t, u_t) = \begin{pmatrix} x_{t+1} \\ y_{t+1} \\ \phi_{t+1} \\ x_{L_t} \\ y_{L_t} \end{pmatrix} \quad (10)$$

$$F_{slam} = \begin{pmatrix} F_{ekf} & 0 \\ 0 & I \end{pmatrix} \quad (11)$$

Jedoch ändert sich das Beobachtungsmodell z_{SLAM} , welches nun nicht mehr auf GPS Messungen, sondern auf den gemessenen Objekten im Umfeld des Fahrzeuges basiert.

$$z_{slam} = \begin{pmatrix} z_\gamma^i \\ z_\beta^i \end{pmatrix} = h(\hat{x}, x_i, y_i) = \begin{pmatrix} \arctan 2(\sqrt{q} \delta_y, \delta_x) - \phi_t \end{pmatrix}$$

mit

$$q = \delta^\top \delta \quad (12)$$

$$\delta = \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} = \begin{pmatrix} x_{L,t+1} - x_{L,t} \\ y_{L,t+1} - y_{L,t} \end{pmatrix}$$

Im SLAM Fall ist also das Beobachtungsmodell im Gegensatz zum EKF mit GPS nicht linear. Demnach ist auch die Jakobi Matrix H^{voll} (13) komplexer als zuvor.

$$H^{voll}(\hat{x}) = H^{min}(\hat{x})F_{\hat{x},j}$$

mit

$$H^{min}(\hat{x}) = \frac{\partial h}{\partial \hat{x}} = \frac{1}{q} \begin{pmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 & \sqrt{q}\delta_x & \sqrt{q}\delta_y \\ \delta_y & -\delta_x & -q & -\delta_y & -\delta_x \end{pmatrix}$$

$$F_{\hat{x},j} = \begin{pmatrix} 1 & 0 & 0 & 0 \dots 0 & 0 & 0 & 0 \dots 0 \\ 0 & 1 & 0 & 0 \dots 0 & 0 & 0 & 0 \dots 0 \\ 0 & 0 & 1 & 0 \dots 0 & 0 & 0 & 0 \dots 0 \\ 0 & 0 & 0 & 0 \dots 0 & 1 & 0 & 0 \dots 0 \\ 0 & 0 & 0 & \underbrace{0 \dots 0}_{2j-2} & 0 & 1 & \underbrace{0 \dots 0}_{2N-2j} \end{pmatrix} \quad (13)$$

N = aktuelle Größe der Matrix

j = derzeit betrachteter Baum

3 Evaluation

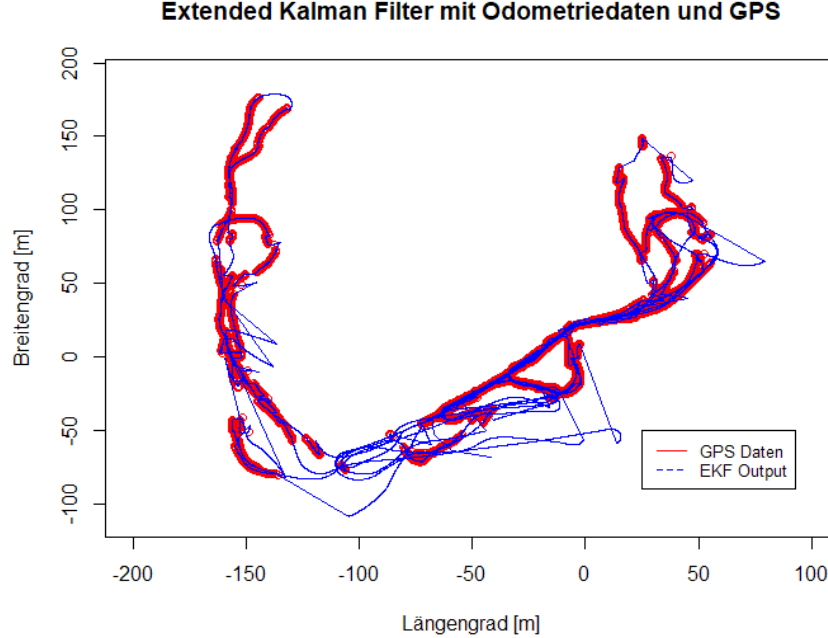


Abbildung 5: Output des EKF mit GPS-Daten als Update.

3.1 Qualität des EKF

Abbildung 5 zeigt das Ergebnis des beschriebenen EKF, implementiert in Flink. Das Resultat unseres EKF's entspricht dem erwarteten.

In Abbildung 6 ist dargestellt, wie sich der Schätzfehler über die Dauer des Experimentes entwickelt. Als Schätzfehler bezeichnen wir hier die Distanz, welche zwischen der durch den *Prediction-Step* bestimmten Position und der vom GPS-Signal gemessenen Position liegt. Die GPS-Position kann unserer Ansicht nach als korrekte Position angesehen werden. Dabei fällt auf, dass es einige größere Schätzfehler gegeben hat. Diese lassen sich auf verschiedene, aber zwei wesentliche, Ursachen zurückführen.

Erstens sind die GPS-Daten fehlerhaft und es gibt Ausreißer in den Messungen. Wir konnten einige wesentliche Ausreißer zwar mit der beschriebenen Heuristik beseitigen, jedoch ist es denkbar, dass es immer noch fehlerhafte GPS-Messungen in unserem Datensatz gibt, welche große Diskrepanzen zwischen geschätzter und gemessener Position hervorrufen.

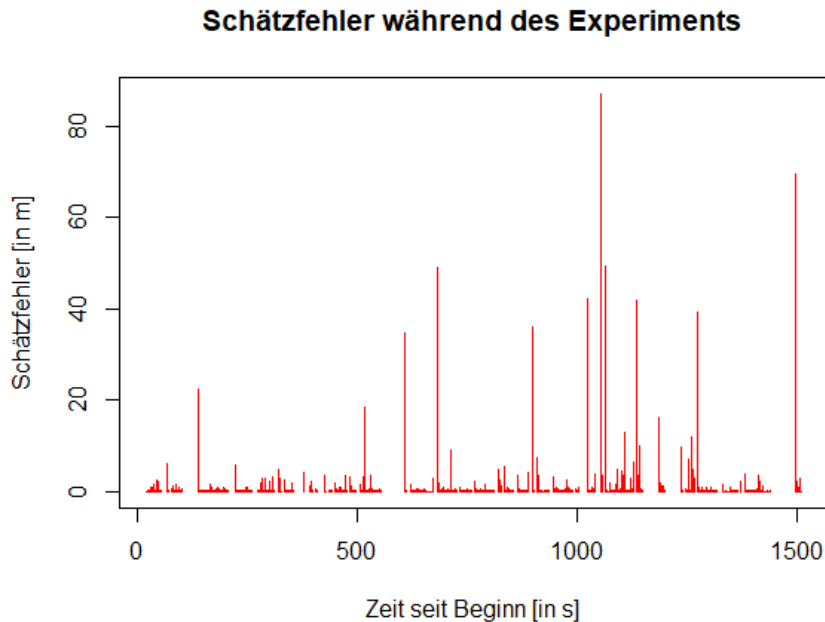


Abbildung 6: Abweichungen zwischen GPS-Messung und EKF-Positionsberechnung

Zweitens ist das GPS-Signal hin und wieder über einen kurzen Zeitraum ausgefallen. Diese Ausfälle erkennt man an den Lücken im obigen Diagramm, denn in diesen Zeiträumen wurde kein *Update-Step* und somit keine vollständige Iteration des EKF durchgeführt. Man sieht, dass am Ende der Lücken jeweils relativ hohe Schätzfehler aufgetreten sind. Das bedeutet, dass die Positionsschätzung nur aufgrund der Odometriedaten relativ ungenau ist. Es wird eine regelmäßige Aktualisierung der Position durch eine externe Messung benötigt, um eine präzise Positionsbestimmung zu gewährleisten.

Im SLAM-Kontext würde diese Aktualisierung durch einen Scan der Umgebung geschehen. Es bleibt auszuwerten, wie präzise die Positionsbestimmung mit dieser Form der externen Messung ist. Zur Evaluierung könnten erneut die GPS Daten verwendet werden.

3.2 Evaluation von SLAM

Da die Implementierung von EKF-SLAM als FullSLAM stattgefunden hat [9], weist die Ausführung des Algorithmus auf einem Computer mit 4 Kern Prozessor und 8 GB RAM bereits nach 1500 von ca. 70000 Tupeln erhebliche Verzögerungen im Sekundenbereich auf. Dies ist einer sehr hoher Komplexität von $O(2N^2)$ bei N Beobachtungen [3] zuzuordnen, da alle Beobachtungen immer komplett aktualisiert werden. Dies macht es unmöglich, den FullSLAM Algorithmus in Echtzeit auszuführen. Außerdem führt es bei den Ausgangseinstellungen der JVM zu einem Heap Error.

Der Heap Error ist unproblematisch zu beheben, aber eine relevante Änderung der Ausführungsumgebung. Dies ist notwendig, um die pro Beobachtung um zwei Zeilen und Spalten wachsende Kovarianzmatrix zu berechnen. Sobald diese etwa eine Größe von 1000x1000 hat, sind signifikante Verzögerungen in der Ausführung zu verzeichnen.

Ein erster Schritt zur Optimierung des Algorithmus war es die *SparseDoubleMatrix2D* von Colt 1.2 zu verwenden, da diese im Gegensatz zur vorher genutzten *DenseDoubleMatrix2D* die Matrizen reihenweise verarbeitet. Das entspricht dem internen Speicherzustand der Matrizen und ist somit schneller. Diese Maßnahme allein hat eine Beschleunigung um ein Vielfaches bewirkt. Ein denkbarer weiterer Schritt zur Optimierung wäre es, die intern verwendeten Gleitkommazahlen im Zustandsvektor und in der Kovarianzmatrix auf einige wenige Nachkommastellen zu begrenzen. Ebenfalls wäre ein minimaler *Prediction-Step*, der lediglich die ersten drei Einträge des Zustandsvektors und die ersten drei mal drei Einträge der Kovarianzmatrix aktualisiert, denkbar. Jedoch sind diese Methoden der Optimierung im Kontext der Komplexität nicht signifikant.

Gleichzeitig basierte die Implementierung unseres Kalman Filters auf der Colt 1.2 Bibliothek, die nach offiziellen Angaben nicht für Parallelisierung geschrieben wurde. Deshalb haben wir die Implementierung zu ParallelColt¹⁵ geändert, was jedoch zu einer signifikanten Verschlechterung der Performanz geführt hat.

Basierend auf der Methode zur Differenzierung von Beobachtungen 2.3.4 werden observierte Bäume unterschiedlich gut wiedererkannt oder neue Bäume unterschiedlich schnell hinzugefügt. Dies wird beispielsweise in der Publikation zu FastSLAM 1.0 und FastSLAM 2.0 [10] behandelt. Hierbei ist es problematisch, wie viele Beobachtungen man im Zustandsvektor und somit auch in der Kovarianzmatrix hat, da die Komplexität zur Berechnung der Matrix $O(2N^2)$ bei N Beobachtungen [3] beträgt. Eine Möglichkeit ist es, Beobachtungen durchgehend eine Relevanz zuzuordnen, die dazu führen kann, dass Beobachtungen wieder entfernt werden, sofern sie als wenig relevant eingestuft werden [10]. Dies würde die Komplexität und somit die Berechnungszeit nachhaltig reduzieren, da nicht immer alle Beobachtungen aktualisiert werden müssten.

4 Zusammenfassung und Diskussion

Ziel des Projektes war es den Extended Kalman Filter mit Simultaneous Localization And Mapping parallelisiert in Apache Flink zu implementieren. Hierbei ergaben sich jedoch Probleme mit den zur Verfügung stehenden Datensätzen und der damit verbundenen Dokumentation. Trotz Problemen mit der Dokumentation der Ausgangsquelle¹⁶ haben wir den EKF auf Basis von GPS in Apache Flink erfolgreich implementiert. Hierzu wurden Ergebnisse des Algorithmus, sowie für die initial für EKF-Slam geplante, jedoch im Kontext des einfachen EKF implementierte Parallelisierung für mehrere Fahrzeuge präsentiert. Dabei wurde besonderes Augenmerk darauf gelegt, die aufgetretenen Probleme ausführlich zu behandeln und zu korrigieren. Die Ergebnisse des EKF entsprechen denen des Victoria Park und die präsentierte Lösung zur Parallelisierung mehrerer Fahrzeuge basiert darauf pro Fahrzeug eine Recheneinheit zu benutzen, somit bleibt die Berechnung eines separaten EKF pro Fahrzeug bestehen.

Im Kontext des einfachen EKF mit GPS Daten wäre eine sinnvolle Ergänzung zum Algorithmus eine Ausreißererkennung für Zeitreihen um, wie im GPS Datensatz vorhandene reale Ausreißer, zu kompensieren, da

¹⁵<https://github.com/rwl/ParallelColt>

¹⁶http://www-personal.acfr.usyd.edu.au/nebot/victoria_park.htm

obwohl der EKF diese relativ abfängt, aber es bei sehr extremen Ausreißern trotzdem nicht zu einer graduellen, sondern plötzlichen, Veränderung kommt, die je nach Zeitabstand zwischen Datenpunkten keiner tatsächlichen Veränderung entsprechen sollte.

Bezüglich des EKF-SLAM wurde eine Implementierung auf Basis des FullSLAM Algorithmus in Flink durchgeführt, wobei eine zu hohe Berechnungskomplexität, sowie ein rudimentär programmierter Erkennungsalgorithmus für Bäume zu unrepräsentativen Ergebnissen geführt hat. Wobei der Stand der Implementierung als Ausgangspunkt für künftige Erweiterungen und Modifikationen genutzt werden kann. Dadurch, dass die Kovarianzmatrix des FullSLAM lediglich wächst ohne zu schrumpfen, kommt es zu Berechnungen sehr großer Matrizen, wodurch der Algorithmus über Echtzeit hinausgeht und somit unbrauchbar wird. Zusätzlich ist die Unterscheidung von neuen Beobachtungen und die Wiedererkennung bereits bekannter Beobachtungen ein integraler Bestandteil des Algorithmus, welcher selber Objekt wissenschaftlicher Arbeit ist. Wobei genau diese Komponente maßgeblich die Genauigkeit des Algorithmus und das Wachstum der Kovarianzmatrix beeinflusst.

Daher wären für SLAM denkbare Ergänzungen zum Projekt andere Algorithmen zur Bestimmung und Differenzierung von Landmarks, sowie Algorithmen zur vereinfachten Berechnung der Matrizen. Für letzteres wäre es denkbar nur lokale Updates durchzuführen oder die Kovarianzmatrix durch Reduzierung von notwendigen Landmarks zu verkleinern um so eine höhere Rechengeschwindigkeit zu gewährleisten.

Literatur

- [1] R. E. Kalman, "A new approach to linear filtering and prediction problems," 1960.
- [2] W. Burgard and K. Arras, "Introduction to Mobile Robotics: SLAM: Simultaneous Localization and Mapping." <http://ais.informatik.uni-freiburg.de/teaching/ss17/robotics/slides/13-slam.pdf>, 2017 (Letzter Zugriff: 27.01.2020).
- [3] G. Jose and E. Nebot, "Simultaneous Localization and Map Building: Test case for Outdoor Applications." http://www-personal.acfr.usyd.edu.au/nebot/publications/slam/IJRR_slam.htm, 2006 (Letzter Zugriff: 31.01.2020).

- [4] K. Juraszek, N. Saini, M. Charfuelan, H. Hemsén, and V. Markl, “Extended Kalman Filter for Large Scale Vessels Trajectory Tracking in Distributed Stream Processing Systems,” in *Advanced Analytics and Learning on Temporal Data* (V. Lemaire, S. Malinowski, A. Bagnall, A. Bondu, T. Guyet, and R. Tavenard, eds.), vol. 11986, pp. 151–166, Cham: Springer International Publishing, 2020.
- [5] G. Bishop, G. Welch, *et al.*, “An introduction to the kalman filter,” *Proc of SIGGRAPH, Course*, vol. 8, no. 27599-23175, p. 41, 2001.
- [6] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, “The Stratosphere platform for big data analytics,” *The VLDB Journal*, vol. 23, pp. 939–964, Dec. 2014.
- [7] “Apache Flink: Stateful Computations over Data Streams.” <https://flink.apache.org>, 2014 (Letzter Zugriff: 28.01.2020).
- [8] “Apache Flink: Stateful Computations over Data Streams.” <https://flink.apache.org/flink-applications.html#layered-apis>, 2014 (Letzter Zugriff: 28.01.2020).
- [9] C. Stachniss, “Robot Mapping: EKF SLAM.” <http://ais.informatik.uni-freiburg.de/teaching/ws13/mapping/pdf/slam05-ekf-slam.pdf>, 2013 (Letzter Zugriff: 27.01.2020).
- [10] M. Montemerlo and S. Thrun, “Simultaneous localization and mapping with unknown data association using fastslam,” in *2003 IEEE International Conference on Robotics and Automation (Cat. No. 03CH37422)*, vol. 2, pp. 1985–1991, IEEE, 2003.