Datenbankprojekt:
Distributed Streaming EKF SLAM

Jestram, Johannes
jestram@posteo.de

Livert, Benedikt
benedikt.livert@gmx.de

Paranskij, Mark
mark.paranskij@gmail.com

2. Februar 2020

Inhaltsverzeichnis

1 Einleitung

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. Um eine möglichst verlässliche Vorhersage treffen zu können, müssen verschiedene Aspekte des betreffenden Systems gemessen werden. Da Messungen jedoch stets - in unterschiedlichem Maß - 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].

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 verortet 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 (i.e. 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. Fahrzeug) 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 Problems" [1] vorgestellt. Er ist ein Ansatz um Zustände zu schätzen. Der Kalman Filter minimiert die geschätzte Fehlervarianz. 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 wird dann im sog. *Update-Step* (auch *Correction-Step*) nach dem Erhalt neuer Messdaten korrigiert und der Algorithmus beginnt von vorne.

Eine attraktive Eigenschaft dieses Algorithmus ist seine rekursive Natur; zur Berechnung des geschätzten neuen Zustands sind nur die Daten

[1] `https://flink.apache.org/`

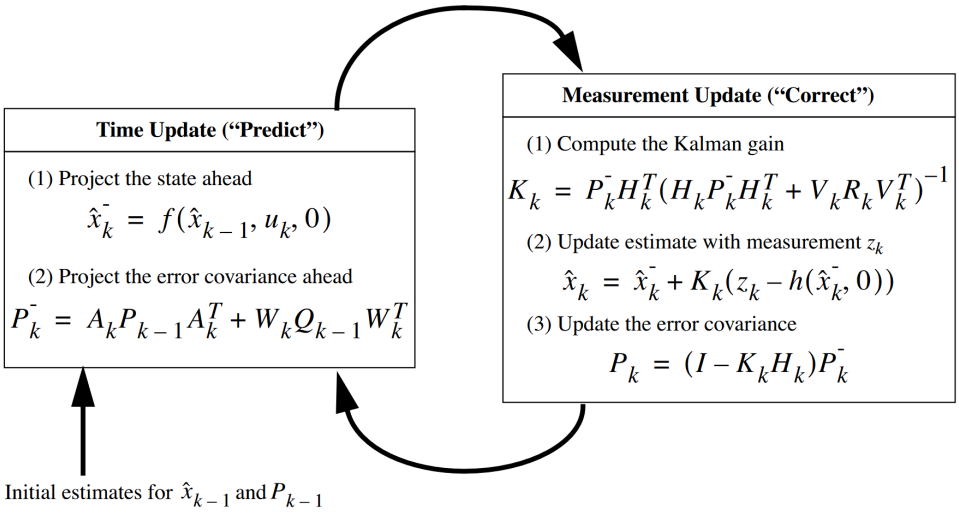

Abbildung 1: Extended Kalman Filter Algorithmus [5].

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 Bewegungsmodells 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 Prediciton-Step geschätzte Zustand wird nun basierend auf neuen, externen Beobachtungen des Zustandes korrigiert. Jedoch ist es nicht so, dass die Schätzung einfach auf die äußere Beobachtung gesetzt wird. Die Korrektur wird gedämpft um den sog. *Kalman Gain*. Dieser berechnet sich sowohl aus Messunsicherheiten, als auch eine 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

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, 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 Projekt[2] [6] der TU-Berlin hervorgegangen und ein Top-Level-Projekt der Apache Foundation [3]. Die Software ist per Open Source Lizenz verfügbar und aufgrund der hohen Performanz [6] ein beliebtes Werkzeug für verteilte und skalierbare Datenverarbeitung.

[2] `http://stratosphere.eu`
[3] `https://apache.org`

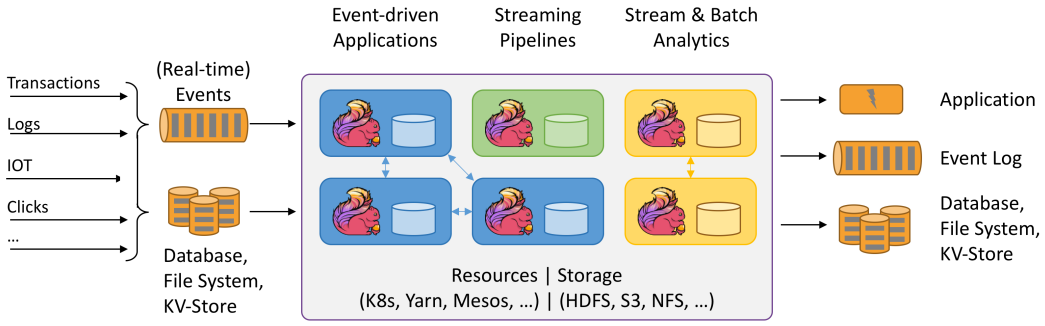

Abbildung 2: Überblick über die Funktionen von Apache Flink [7].

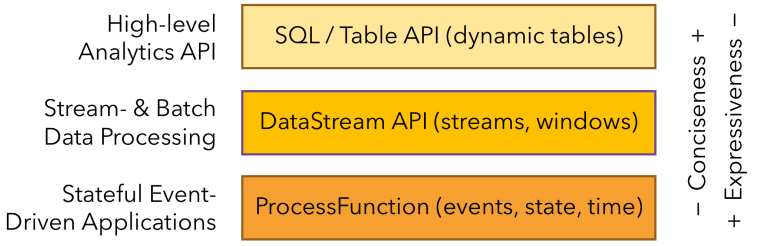

Abbildung 3: Flink's APIs auf verschiedenen Abstraktionsniveaus [8].

Flink basiert auf der Java Virtual Machine (JVM) und ist in Scala und Java implementiert [4]. Der Funktionsumfang umfasst die parallele und verteilte Verarbeitung von Datenströmen, sowie einmaligen (in der Regel sehr 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 automa-

[4] `https://github.com/apache/flink`

tisierte 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

InfluxDB[5] 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 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 in eine externe Bibliothek. Somit haben wir uns für die Colt 1.2 Library [6] entschieden, welche am CERN entwickelt wurde, da alle für notwendigen Matrix- und Vektoroperationen durch Colt zur Verfügung gestellt werden.

[5] `https://www.influxdata.com/products/influxdb-overview`

[6] `https://dst.lbl.gov/ACSSoftware/colt`

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

[7] `https://www.mrpt.org/Dataset_The_Victoria_Park`

2.2.2 Probleme mit den Daten

Im Laufe des Projektes stellte sich der Umgang mit den Daten als das zeitaufwändigste Problem dar. Vor allem die unterschiedlichen Versionen desselben Datensatzes sorgten für regelmäßige 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 war, was zu vielen Unsicherheiten bei der Verarbeitung des Datensatzes geführt hat.

Darüber hinaus befindet sich auf der ursprünglichen Webseite 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 Odometrie Datenpunkten, 4466 GPS-Messungen und 7249 Lasermessungen, insgesamt also 73660 Datenpunkten, ergibt.

Aufgrund eines Mangels an definitiver Information und inkohärenter Datensätze wurde der Projektfortschritt lange durch das auswerten dieser Datensätze hinausgezögert. Hierbei haben wir versucht die Daten durch Visualisierung zu validieren. Dies führte bei dem kleinen Datensatz (4832 Messpunkte) zu keinem Erfolg, da das Format der Inkremente nicht ausreichend dokumentiert ist. Erst mit viel Recherche und durch interne Informationen aus dem Rawlog Viewer [12] sind wir auf das zugrundliegende Bewegungsmodel des Datensatzes [13] gestoßen. Lediglich die GPS-Messungen des MatLab Datensatzes erschienen ähnlich zu der bekannten Route des Fahrzeuges

[8] `https://www.mrpt.org/Dataset_The_Victoria_Park`

[9] `https://www.mrpt.org/list-of-mrpt-apps/rawlogviewer`

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

[11] `https://www.mrpt.org/Dataset_The_Victoria_Park`

[12] `https://www.mrpt.org/list-of-mrpt-apps/rawlogviewer`

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

durch Victoria Park. Daher entschlossen wir uns, den EKF aus Basis der GPS Daten des Matlab Datensatzes anzuwenden.

Insgesamt 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. Beispielsweise weicht der ASCII Datensatz gravierend vom MatLab Datensatz ab. Die GPS-Messungen der beiden Datensätze sind gänzlich unterschiedlich. Die Odometriedaten, durch den selben EKF-Algorithmus verarbeitet, resultierten in komplett unterschiedlichen Ergebnissen. Daher ist der ASCII Datensatz nach unserem Kenntnissstand als falsch anzusehen.

2.2.3 Ausreißer

Die GPS-Messungen des bereitgestellten Datensatzes sind fehlerhaft und unvollständig und wurden daher nicht weiter verwendet [14]. Da es aufgrund der unterschiedlichen Datensätze notwendig wurde den EKF in der einfachsten Form mit GPS zu implementieren, mussten wir dennoch auf den Datensatz zurückgreifen. Damit es trotzdem vergleichbare und repräsentative Ergebnis für den EKF mit GPS geben konnte, entschieden wir uns dazu den GPS Datensatz um Ausreißer basierend auf einer Heuristik zu bereinigen. Hierbei werden Datenpunkte, die eine zu große Distanz zu vorherigen GPS Messungen aufweisen, aussortiert.

2.3 Implementierung

2.3.1 Simulatiaon 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.

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

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

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 einen die Daten der unterschiedlichen Fahrzeuge auch bei der Verarbeitung dem entsprechenden Fahrzeug zugeordnet werden. Da jedes Fahrzeug einen eigenen, distinkten 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 basiert.

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()*-

Funktion 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.

2.3.3 EKF Implementierung

Prediction Schritt 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.

$$\hat{x}_t = \begin{pmatrix} x_t \\ y_t \\ \phi_t \end{pmatrix} \tag{1}$$

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

$$\begin{aligned} v &= \frac{v_{Messung}}{1 - \tan(\alpha) * \frac{H}{L}} \\ &\text{mit} \\ v &= \text{Fahrzeuggeschwindigkeit} \\ v_{Messung} &= \text{Geschwindigkeit am linken Hinterrad} \\ H &= \text{Achsabstand Sensor zur Fahrzeugmitte} = 0,76m \\ L &= \text{Abstand Hinterachse zur Vorderache} = 2,83m \end{aligned} \tag{2}$$

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.

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

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

$$P_0 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{4}$$

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

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

Update Schritt Im Update Schritt werden die Ergebnisse des Prediction Schritt 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 gemssen. 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.

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

Zwar ist das Beobachtungsmodell linear, da jedoch aufgrund des nichtlinearen Bewegungsmodells der Extended Kalman Filter verwendet wird, muss auch vom Beobachtungsmodell die Jakobi-Matrix gebildet werden.

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

$$R = \begin{pmatrix} 0,5 & 0 \\ 0 & 0,5 \end{pmatrix} \tag{8}$$

2.3.4 SLAM Implementierung

Der Unterschied zum EKF ohne SLAM-Anteil liegt vor allem darin, dass der Update-Schritt nun auf Grundlage der in der Umgebung verorteten 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.

$$\hat{x} = \begin{pmatrix} x_{fahrzeug} \\ y_{fahrzeug} \\ \phi \\ x_{objekt-1} \\ y_{objekt-1} \\ ... \\ ... \\ x_{objekt-n} \\ y_{objekt-n} \end{pmatrix} = \begin{pmatrix} x_i \\ y_i \\ \phi \\ x_L \\ y_L \end{pmatrix} \tag{9}$$

Ableiten der Objektposition aus den Lasermessungen Die Lasermessungenn sehen in jedem Datensatz vergleichbar aussehen 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. Anschließend können aus einer Menge von Messpunkten eine einzelner Baum 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 observierten 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.

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

$$F_{slam} = \begin{pmatrix} F_{ekf} & 0 \\ 0 & I \end{pmatrix} \tag{11}$$

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

$$\begin{aligned} &z_{slam} = \begin{pmatrix} z_\gamma^i \\ z_\beta^i \end{pmatrix} = h(\hat{x}, x_i, y_i) = \begin{pmatrix} \sqrt{q} \\ \arctan 2(\delta_y, \delta_x) - \phi_t \end{pmatrix} \\ &\text{mit} \\ &q = \delta^\intercal \delta \\ &\delta = \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} = \begin{pmatrix} x_{L,t+1} - x_{L,t} \\ y_{L,t+1} - y_{L,t} \end{pmatrix} \end{aligned} \tag{12}$$

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

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

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

N = aktuelle Größe der Matrix

j = derzeit betrachteter Baum

3 Evaluation

3.1 Qualität des EKF

Abbildung 4 zeigt das Ergebnis des beschriebenen EKF, implementiert in Flink.

In Abb. 5 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 verschiedenen, aber zwei wesentliche,Ursachen zurückführen.

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

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 vollständiger 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 fehlerhaft ist. Es wird ein regelmäßiges Update der Position

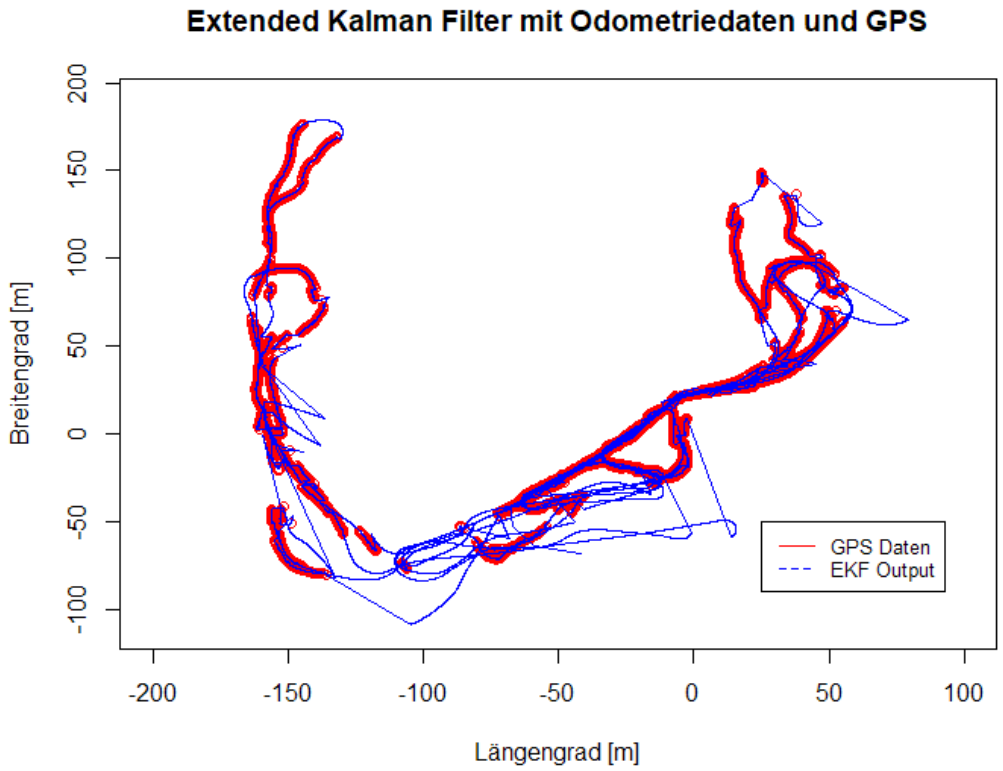

Abbildung 4: Output des EKF mit GPS-Daten als Update.

durch eine externe Messung benötigt, um eine präzise Positionsbestimmung zu gewährleisten.

Im SLAM-Kontext würde dieses Update 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 Komple-

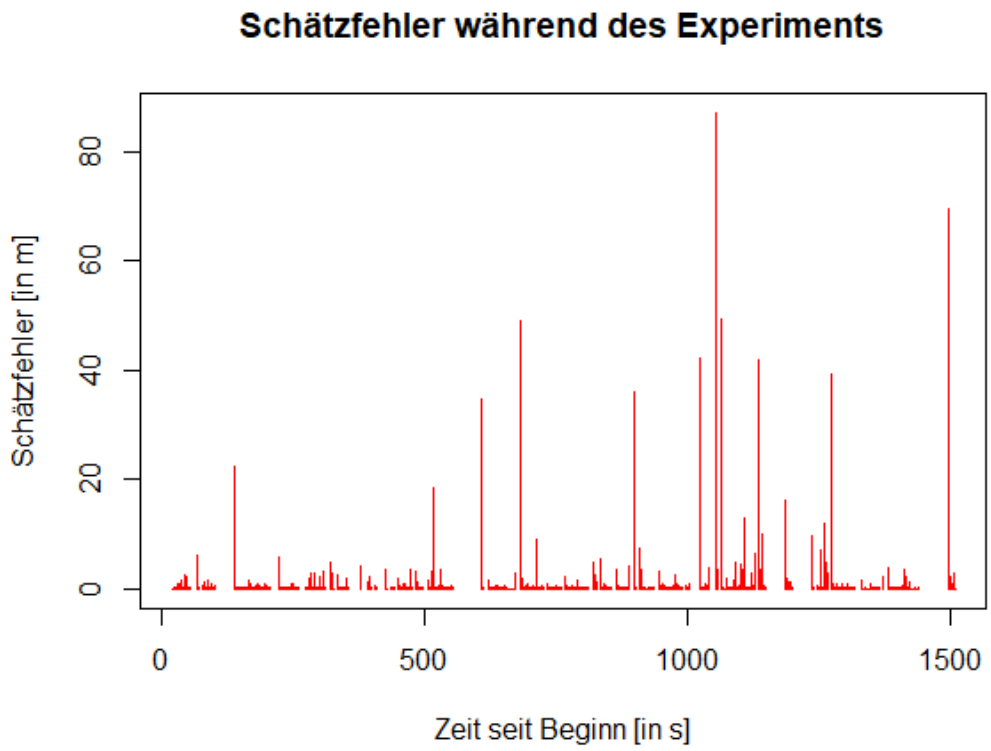

Abbildung 5: Abweichungen zwischen GPS-Messung und EKF-Positionsberechnung

xität 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 verarbei-

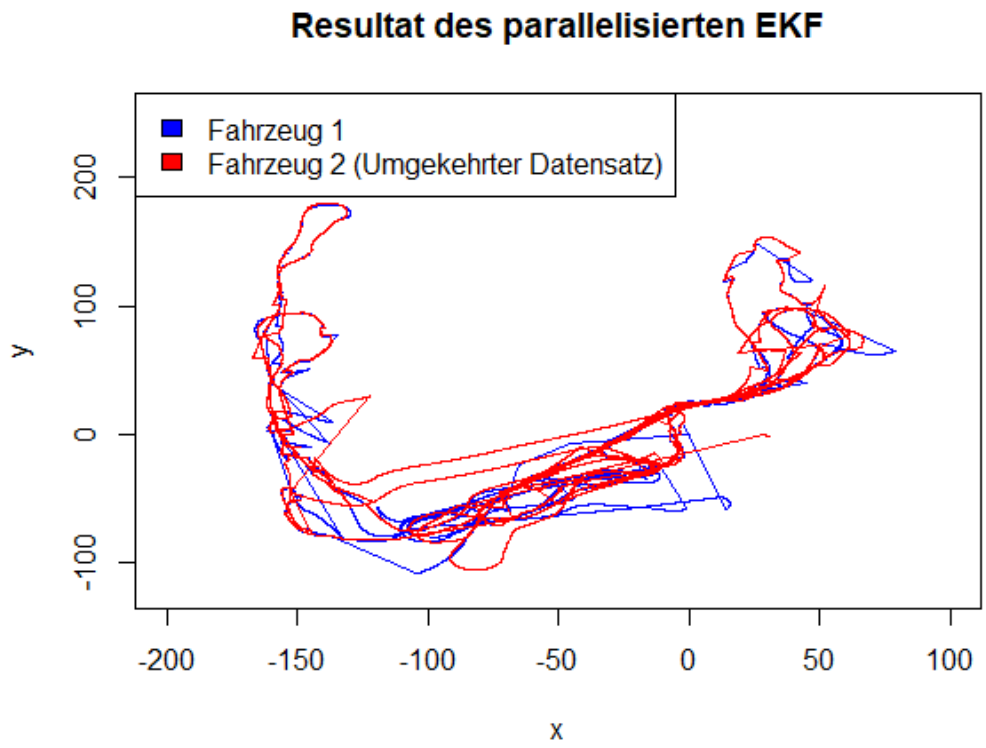

Abbildung 6: Output des parallelisierten EKF mit mehreren simulierten Fahrzeugen.

ten 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 der 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 unserer Kalman Filter auf der Colt 1.2 Bibliothek, die nach offiziellen Angaben nicht für Parallelisierung

geschrieben wurde. Somit haben wir die Implementierung zu ParallelColt [15] 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

5 Diskussion

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," 2017.

[3] G. Jose and E. Nebot, "Simultaneous Localization and Map Building: Test case for Outdoor Applications."

[4] K. Juraszek, N. Saini, M. Charfuelan, H. Hemsen, 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. Welch and G. Bishop, "An Introduction to the Kalman Filter," p. 81.

[15] `https://github.com/rwl/ParallelColt`

[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."

[8] "Apache Flink: Stateful Computations over Data Streams."

[9] C. Stachniss, "Robot Mapping: EKF SLAM," 2013.

[10] S. Thrun, M. Montemerlo, D. Koller, B. Wegbreit, J. Nieto, and E. Nebot, "FastSLAM: An Efficient Solution to the Simultaneous Localization And Mapping Problem with Unknown Data Association," p. 48.