Ein Framework zur automatischen Differenzierung von Extended Kalman Filtern auf Mannigfaltigkeiten

Masterarbeit

Lukas Post

Matrikelnummer: 4000271

2. Dezember 2019

Fachbereich Mathematik / Informatik
Studiengang Informatik

1. Gutachter: Prof. Dr. Udo Frese
2. Gutachter: Prof. Dr. Thomas Schneider
Betreuer: Tom Koller und Dr. Tim Laue

Erklärung

Hiermit versichere ich, die Masterarbeit oder den von mir zu verantwortenden Teil einer Gruppenarbeit ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 2. Dezember 2019

.....

(Lukas Post)

Inhaltsverzeichnis

Kapitel 1

Einleitung

Seit seiner Entwicklung in den sechziger Jahren ist der Extended Kalman Filter (EKF) einer der beliebtesten Filter für nichtlineare Online-Zustandsschätzung in den Ingenieurswissenschaften. Ursprünglich im Rahmen des US-Raumfahrtprogramms zur Positionsschätzung von Raumfähren [18] auf Basis des Linearen Kalman Filters [14] entwickelt, wird der EKF heutzutage zur Lösung einer Vielzahl verschiedener Schätzprobleme verwendet. Diese reichen von der Positionsschätzung eines Kraftfahrzeugs [26] über Orientierungsschätzung von Mobiltelefonen zu Entertainmentzwecken [9] bis hin zur Schätzung von nicht direkt beobachtbaren Parametern in der computergestützten Biologie [17].
Es existieren Weiterentwicklungen des EKF, welche einfacher zu benutzen sind, dafür aber ein Geschwindigkeitsdefizit aufweisen. Ein Beispiel dafür ist der Unscented Kalman Filter (UKF) [12], welcher die Modellfunktionen an verschiedenen Stellen auswertet und aus den Ergebnissen eine sehr genaue Zustandsschätzung erhält. Im Gegensatz dazu verwendet der EKF die partiellen Ableitungen (Jacobimatrizen) von Dynamik- und Messmodellen, welche allerdings vorher vom Benutzer berechnet werden müssen. Besonders für große Modelle ist dies mitunter zeitaufwändig und kann bei ungeübten Benutzern schnell zu Fehlern führen.
In dieser Arbeit soll ein Framework entwickelt werden, welches das Berechnen der Jacobimatrizen vom Benutzer übernimmt und dabei die Performanz eines per Hand abgeleiteten EKF behalten soll.

1.1 Motivation

Da die händische Differenzierung eine Fehlerquelle darstellt, liegt der Gedanke nahe, die Jacobimatrizen durch einen Computer berechnen zu lassen. In der Praxis wird das häufig auch umgesetzt, da Computeralgebrasysteme (CAS) wie Mathematica [36] oder die Symbolic Math Toolbox von MATLAB [29] Funktionalitäten bereitstellen, durch welche ein exaktes symbolisches Bestimmen der Jacobimatrizen ermöglicht wird. Es besteht auch die Möglichkeit einer numerischen Bestimmung der Jacobimatrizen durch den Differentialquotienten. Dieser An-

satz wird vom in MATLAB integrierten EKF [28] als Standardeinstellung verwendet.
Beide Ansätze haben Vor- und Nachteile: So ergibt eine symbolische Differenzierung zwar exakte Ableitungen, aber diese sind bei komplizierten Ausdrücken oft sehr lang wenn auf Optimierung keine Rücksicht genommen wird. Außerdem müssen die Ableitungen nach der Berechnung immer noch per Hand in das Zielsystem übertragen werden, was eine Fehlerquelle birgt. Die numerische Berechnung der Jacobimatrizen ist einfach zu implementieren und schnell, aber das Ergebnis ist nur eine Approximation und ungenügende Genauigkeit kann zu großen Fehlern in der Linearisierung führen. Für die schnelle und exakte Berechnung einer Ableitung direkt im Zielsystem eignet sich die automatische Differenzierung. Durch Aufteilen einer mathematischen Funktion in einzelne Unterausdrücke und rekursives Ausführen der Kettenregel macht es diese möglich, das Ergebnis und die Ableitung an einer Stelle exakt zu bestimmen [20]. Dieses Verfahren ist unkompliziert zu implementieren und die Genauigkeit hängt nur von der Maschinengenauigkeit ab.
Ein EKF auf Basis von automatischer Differenzierung wurde bereits 1999 von Leuck Et al. implementiert und zum Tracking von Fahrzeugen verwendet [16]. Eine Untersuchung des EKF an sich fand nicht statt. Es wurde aber festgestellt, dass der automatisch differenzierte EKF nur etwa 5-10% langsamer war als ein per Hand differenzierter.
Simon Julier und Jeffrey Uhlmann, die Entwickler des UKF, beschreiben zwei Nachteile des EKF gegenüber des UKF [12]. Der Erste betrifft die Eigenschaft des EKF, dass dieser nur korrekt bis zur ersten Ordnung ist, wobei der UKF Korrektheit der dritten Ordnung garantiert [22, S. 454]. Dieser Nachteil betrifft nur die klassische Variante des EKF. Es ist mit einem EKF höherer Ordnung möglich die Übertragungsfunktionen weiter anzunähern und damit ein äquivalentes Ergebnis zum UKF zu erhalten. Der zweite Nachteil ist, dass die Berechnung der Jacobimatrizen nichttrivial ist und zu erheblichen Implementierungsschwierigkeiten führen kann. Da aber durch automatische Differenzierung das händische Berechnen der Jacobimatrizen entfällt, lässt sich ein generischer EKF implementieren, der genau so einfach zu verwenden ist, wie eine Implementierung des UKF. Außerdem ist der EKF für große Zustände immer schneller.
Die Verwendung von Mannigfaltigkeiten als Zustand in einem Kalman Filter erlauben es dem Benutzer, Schätzungen auf komplexen topologischen Räumen durchzuführen, solange diese lokal einem euklidischen Raum ähneln [10]. Ein Beispiel hierfür sind geographische Koordinaten. Diese können lokal durch euklidische Koordinaten angenähert werden, aber global schlägt diese Annahme fehl. Ein Schätzalgorithmus betrachtet deshalb nur die Änderungen zwischen Elementen einer Mannigfaltigkeit, welche wiederum euklidisch angenähert werden können. Die Berechnung von Jacobimatrizen auf Mannigfaltigkeiten ist kompliziert und selbst für niedrigdimensionale Mannigfaltigkeiten resultiert diese in sehr langen Ausdrücken [35]. Durch die automatische Differenzierung können direkt genaue Jacobimatrizen der Mannigfaltigkeiten berechnet werden, ohne dass auf komplexe Ausdrücke Rücksicht genommen werden muss.

1.2 Ziele und Aufbau der Arbeit

Das Ziel dieser Arbeit ist es, ein Framework zu entwickeln, das einen EKF implementiert und die Jacobimatrizen per automatischer Differenzierung berechnet. Dabei soll das Interface es unterstützen auch Mannigfaltigkeiten in die Zustände aufnehmen zu können. Der Fokus des Frameworks soll auf einfacher Benutzbarkeit, Geschwindigkeit und Korrektheit liegen.
Zu Beginn werden Grundlagen erläutert auf denen diese Arbeit basiert. Dazu gehören Differenzierung, Kalman Filter und Mannigfaltigkeiten. Danach folgt ein Überblick über das entwickelte Framework mit einer Interfacespezifikation und Implementierungsdetails. Anschließend wird das Framework unter verschiedenen Gesichtspunkten evaluiert und die Arbeit endet mit einem Fazit und Ausblick.

Kapitel 2

Grundlagen

Die Grundlagen gliedern sich in drei Teile. Zuerst wird ein Überblick über verschiedene computerbasierte Differenzierungsverfahren gegeben. Namentlich, die symbolische, numerische und automatische Differenzierung. Es folgt eine Einführung in die Zustandsschätzung mit Kalman Filtern. Darunter fallen auch der Extended und Unscented Kalman Filter, welche für nichtlineare Schätzung verwendet werden. Das Kapitel endet mit den mathematischen Grundlagen von ⊞-Mannigfaltigkeiten. Diese Art von Mannigfaltigkeit macht es auf sehr elegante Weise möglich Ausgleichsrechnung auf komplexen mathematischen Strukturen durchzuführen.

2.1 Differenzierung

Das Differenzieren einer mathematischen Funktion bildet die Ableitung dieser und beschreibt so die kontinuierliche Veränderung der Funktionswerte. Dieser Vorgang bildet die Grundlage der Differentialrechnung, welche Anwendung in allen Disziplinen der Ingenieurswissenschaft findet.

Das Bestimmen der Ableitung per Hand basiert auf einer Vielzahl von Regeln, welche auf die Funktion angewendet werden können. Jede dieser Regeln entspricht einem eindeutigen Unterausdruck innerhalb dieser Funktion und beschreibt wie verfahren werden muss, um aus diesem Unterausdruck die exakte Ableitung zu erhalten.
Der Umstand, dass in einem einfachen Computerprogramm eine mathematische Funktion als Black Box angesehen wird, macht es unmöglich diese Methode direkt anzuwenden. Die folgenden Kapitel geben einen Überblick über Verfahren, die es möglich machen trotzdem computerbasiert exakt und schnell Ableitungen zu berechnen.

2.1.1 Symbolisch

Symbolische Differenzierung betrachtet mathematische Funktionen in einem Computerprogramm als Datenstruktur und nicht bloß als einfache Eingabe und Ausgabe von Zahlen. Diese Datenstruktur speichert jeden einzelnen Unterausdruck der Funktion ab, sodass es möglich wird Algorithmen zu schreiben, die die Funktion nicht nur Ausführen, sondern auch Analysieren und Optimieren können. Da der genaue Aufbau der Funktion bekannt ist, macht es dieser Ansatz auch möglich exakte unbestimmte Ableitungen dieser Funktionen zu berechnen, indem die Ableitungsregeln auf jeden einzelnen Unterausdruck ausgeführt werden.

Der große Vorteil dieser Methode ist, dass die Ableitungsoperation für jede Funktion nur einmal ausgeführt werden muss. Danach ist diese für jede Eingabe wiederzuverwenden. Allerdings ist dieser Ansatz mit einem erheblichen Implementierungsaufwand verbunden, da für jeden möglichen Unterausdruck nicht nur die Ableitungskorrespondenzen, sondern auch eine Klassenstruktur programmiert werden muss, die jeden möglichen Ausdruck rekursiv darstellen und auch ausführen kann.

Die Methode der symbolischen Differenzierung wird hauptsächlich in Computeralgebrasystemen [29][36] benutzt um dort schnell und unkompliziert Ableitungen berechnen zu lassen und diese dann händisch in das Zielsystem zu übertragen. Zwar führt dies zu der kürzesten Ausführungszeit im Zielsystem, das händische Übertragen kann aber besonders bei langen Ableitungen schnell zu Fehlern führen.

2.1.2 Numerisch

Ein einfacher Weg eine bestimmte Ableitung einer Funktion zu berechnen ist die numerische Differenzierung über den Differentialquotienten [30, S. 325]

$$\left.\frac{df(x)}{dx}\right|_{x=x_0} = \lim_{h\to 0} \frac{f(x_0+h)-f(x_0)}{h} \tag{2.1}$$

Für kleine Werte von h lässt sich so sehr schnell eine Approximation der bestimmten Ableitung an der Stelle x_0 berechnen. Leider ist für jede mögliche Wahl von h die berechnete Steigung nur eine Approximation. Es ist also nicht möglich exakte bestimmte Ableitungen mit dieser Methode zu bestimmen.
Diese Art der Differenzierung ist zwar sehr schnell und sehr einfach zu implementieren, aber im Gegensatz zur symbolischen Differenzierung ist sie nicht exakt und muss für jeden Wert von x_0 neu berechnet werden.

2.1.3 Automatisch

Eine weitere Art computergestützt Ableitungen zu berechnen ist die automatische Differenzierung. Diese wurde von Wengert [34] und Rall [20] popularisiert und seither gab es viele Veröffentlichungen in diesem Feld [11].

Bei der automatischen Differenzierung wird ausgenutzt, dass alle mathematischen Funktionen als eine Hintereinanderausführung von elementaren Funktionen betrachtet werden können. Das Anwenden der Kettenregel macht es möglich für jeden Zwischenschritt die bestimmte Ableitung und das Funktionsergebnis an der gegebenen Stelle parallel zu berechnen.

Sei $f = f_3 \circ f_2 \circ f_1$ eine Verkettung multivariater und multidimensionaler elementarer Funktionen. Die automatische Differenzierung beschreibt für diesen Fall zwei Modi, die beide das Matrixprodukt aus der resultierenden Jacobimatrix und einer Matrix x', y' berechnen [11]. Der Vorwärtsmodus berechnet das Ergebnis des Ausdrucks $f'(x_0) \cdot x'$ wobei $f'(x_0)$ die Jacobimatrix der Funktion f and der Stelle x_0 ist. Der Rückwärtsmodus berechnet den Ausdruck $y' \cdot f'(x_0)$. Die Variablen x' und y' werden vom Benutzer vorgegeben und können jeden gewünschten Wert annehmen. Setzt man x' oder y' auf die Einheitsmatrix entsprechender Größe berechnet man mit dem Vorwärts- und Rückwärtsmodus bloß die Jacobimatrix der Funktion f.
Der Vorwärtsmodus berechnet für jede elementare Funktion in der Verkettung rekursiv zwei Zahlen. Die erste Zahl entspricht dem Ergebnis der Funktion nach jedem einzelnen Ausführungsschritt und die zweite Zahl dem Ergebnis der Kettenregel. Das Ergebnis einer automatischen Differenzierung im Vorwärtsmodus nach jedem Ausführungsschritt am Beispiel von f an der Stelle x_0 ist folgendes:

$$
\begin{gathered}
[x_0 \ , \ x'] \\
[f_1(x_0) \ , \ f_1'(x_0) \cdot x'] \\
[f_2(f_1(x_0)) \ , \ f_2'(f_1(x_0)) \cdot f_1'(x_0) \cdot x'] \\
[f_3(f_2(f_1(x_0))) \ , \ f_3'(f_2(f_1(x_0))) \cdot f_2'(f_1(x_0)) \cdot f_1'(x_0) \cdot x']
\end{gathered}
$$

Das Ergebnis der automatischen Differenzierung findet sich nun in der zweiten Zelle des letzten Schritts. Man beachte, dass in jedem Schritt bloß die nächste Funktion in der Verkettung, dessen Ableitung und das Ergebnis des letzten Ausführungsschrittes verwendet wird.

Bei der Berechnung des Rückwärtsmodus muss zweistufig vorgegangen werden. Da für die Berechnung der Ableitungen mit Hilfe der Kettenregel die Funktionsergebnisse aus späteren Ausführungsschritten benötigt werden. Es werden also zuerst die Ergebnisse der Ausdrücke $f_1(x_0)$, $f_2(f_1(x_0))$ und $f_3(f_2(f_1(x_0)))$ berechnet und abgespeichert. Im zweiten Schritt folgt die Berechnung der Ableitung von hinten nach vorne:

$$
\begin{gathered}
[f_3(f_2(f_1(x_0))) \ , \ y' \cdot f_3'(f_2(f_1(x_0)))] \\
[f_2(f_1(x_0)) \ , \ (y' \cdot f_3'(f_2(f_1(x_0)))) \cdot f_2'(f_1(x_0))] \\
[f_1(x_0) \ , \ (y' \cdot f_3'(f_2(f_1(x_0))) \cdot f_2'(f_1(x_0))) \cdot f_1'(x_0)]
\end{gathered}
$$

Auch hier findet sich das Ergebnis der automatischen Differenzierung in der zweiten Zelle des letzten Schritts.

Diese Art der Differenzierung grenzt sich von der symbolischen Differenzierung ab, da sofort

exakte bestimmte Ableitungen berechnet werden, ohne dass der Umweg über eine symbolische unbestimmte Darstellung der Ableitung genommen wird. Dieser Umstand macht die automatische Differenzierung sehr attraktiv zur Implementierung in einem Computerprogramm, auf welche in den nächsten Kapiteln näher eingegangen wird.

2.1.4 Differenzierung mit dualen Zahlen

Eine sehr elegante Methode zur Berechnung bestimmter Ableitungen per automatischer Differenzierung im Vorwärtsmodus ist die Evaluation einer Funktion mit dualen Zahlen als Funktionsargument. Duale Zahlen wurden 1871 von Clifford [4] eingeführt und definieren eine zweidimensionale Algebra auf den reellen Zahlen. Sie sind in ihrer Art sehr ähnlich zu imaginären Zahlen in dem sie die reellen Zahlen mit einer abstrakten Einheit erweitern. Im Gegensatz zu der imaginären Einheit i für die $i^2 = -1$ gilt, gilt für die duale Einheit $\epsilon^2 = 0$. Sie ist also nilpotent.
Duale Zahlen haben die Eigenschaft, dass sie das Taylorpolynom einer Funktion ab der ersten Entwicklung abschneiden. Da, wenn sie als Funktionsargument verwendet werden, ab der zweiten Entwicklung jeder Term einen Faktor $\epsilon^{\geq 2} = 0$ enthält:

$$\begin{aligned} f(x_0 + \epsilon) &= f(x_0) + f'(x_0)\epsilon + f''(x_0)\frac{\epsilon^2}{2} + f'''(x_0)\frac{\epsilon^3}{6} + \cdots \\ &= f(x_0) + f'(x_0)\epsilon \end{aligned} \tag{2.2}$$

Eine Evaluation einer Funktion mit einer dualen Einheit im Funktionsargument führt also dazu, dass der reelle Anteil das Ergebnis der Funktion an der Stelle x_0 und der duale Anteil die Ableitung der Funktion an der Stelle x_0 ist. Hier ein Beispiel einer automatischen Differenzierung anhand der Funktion $f(x) = x^2 + x$ an der Stelle 10:

$$\begin{aligned} f(10 + \epsilon) &= (10 + \epsilon)^2 + (10 + \epsilon) \\ &= (100 + 20\epsilon + \epsilon^2) + (10 + \epsilon) \\ &= 110 + 21\epsilon + \epsilon^2 \\ &= 110 + 21\epsilon \end{aligned} \tag{2.3}$$

Die per Hand bestimmte Ableitung von $f(x)$ ist $f'(x) = 2x+1$. Der Koeffizient von ϵ entspricht also der korrekten Ableitung an der Stelle 10.
Für multivariate Funktionen existiert ebenfalls ein auf dualen Zahlen basierender Ansatz zur Bestimmung von Jacobimatrizen. Dazu wird für jede Dimension des Funktionsarguments ein eigenes duales Element $\epsilon_i, i \in [1, n]$ definiert, für welche gilt: $\forall i, j \in [1, n] : \epsilon_j \epsilon_k = 0$. Für eine multivariate Funktion $f(x)$ lässt sich die Jacobimatrix $f'(x)$ an der Stelle x_0 berechnen mit:

$$\begin{aligned} g &= f\left(x_0 + (\epsilon_1, \ldots, \epsilon_n)^T\right) \\ f'(x)\big|_{x=x_0} &= ([\epsilon_1](g), \ldots, [\epsilon_n](g)) \end{aligned} \tag{2.4}$$

Wobei $[\epsilon_n](g)$ der Koeffizient der dualen Einheit ϵ_n in g ist.

2.2 Kalman Filter

Der Kalman Filter wird verwendet, um den Erwartungswert eines Zustands $\mu_t \in \mathbb{R}^n$ und die dazugehörige Kovarianz $\Sigma_t \in \mathbb{R}^{n \times n}$ optimal zu schätzen. Er wurde 1960 von Rudolph Emil Kalman entwickelt [14] und in den folgenden Jahren im Rahmen des US Raumfahrtprogramms weiterentwickelt [18].

Wie bei einem Bayes Filter arbeitet der Kalman Filter zweistufig. Zuerst schreitet der Zustand und die Kovarianz durch die Eigendynamik des Systems fort. Diese Dynamik wird modelliert durch eine Zustandsübergangsmatrix $F_t \in \mathbb{R}^{n \times n}$, welche mit dem Zustand multipliziert wird um diesen fortschreiten zu lassen. Dies können beispielsweise Bewegungsgleichungen sein, falls der Zustand eine Position und Geschwindigkeit enthält. Falls der Zustandsübergang von einer Messung $u_t \in \mathbb{R}^p$, wie einem Odometrie Offset, abhängt, kann diese durch eine lineare Abbildung $B_t \in \mathbb{R}^{n \times p}$ in den Zustand einfließen. Der Zustandsübergang von Zustand und Kovarianz ist nun definiert als [31, S. 42]

$$\begin{aligned} \bar{\mu}_t &= F_t \mu_{t-1} + B u_t \\ \bar{\Sigma}_t &= F_t \Sigma_{t-1} F_t^T + Q_t \end{aligned} \tag{2.5}$$

Wobei $Q_t \in \mathbb{R}^{n \times n}$ die Kovarianzmatrix des additiven Rauschens des Zustandsübergangs ist und $\bar{\mu}_t$ und $\bar{\Sigma}_t$ der geschätzte Zustand und die Kovarianz vor Ausführung des Messschritts sind.
Der zweite Schritt des Kalman Filters ist der Mess- oder Updateschritt. Dabei werden aus dem aktuell geschätzten Zustand Voraussagen über eine Messung gemacht und der Zustand entsprechend der gegebenen Kovarianzen optimal geschätzt. Dabei können auch Zustände geschätzt werden, die nicht direkt messbar sind. Zur Modellierung der Messung aus dem Zustand heraus wird eine Messmodellmatrix $H_t \in \mathbb{R}^{m \times n}$ verwendet. Diese berechnet aus dem Erwartungswert des Zustands den Erwartungswert der Messung und der Kalman Filter vergleicht diesen mit einer Beobachtung $z_t \in \mathbb{R}^m$ des Zustands. Der Messschritt ist definiert als:

$$\begin{aligned} K_t &= \bar{\Sigma}_t H_t^T (H_t \bar{\Sigma}_t H_t^T + R_t)^{-1} \\ \mu_t &= \bar{\mu}_t + K_t (z_t - H_t \bar{\mu}_t) \\ \Sigma_t &= \bar{\Sigma}_t - K_t H_t \bar{\Sigma}_t \end{aligned} \tag{2.6}$$

$K_t \in \mathbb{R}^{n \times m}$ ist der sogenannte Kalman-Gain. Dieser ist eine Art Gewichtung, wie stark die Messung in den Zustand eingehen soll und ist abhängig von den Kovarianzen des Zustands $\bar{\Sigma}_t$ und der Messung R_t.

Da Zustandübergangs- und Messmodell lineare Funktionen sind, berechnet der Kalman Fil-

ter für die gegebenen Modelle eine optimale Schätzung von Zustand und Kovarianz. Das bedeutet, dass die Schätzung eine minimale quadratische Abweichung von den tatsächlichen Werten hat.
Der stochastische Prozess eines Kalman Filters gleicht einem Hidden Markov Model [1], da für einen unbeobachtbaren Zustand über Messungen Rückschlüsse gezogen werden. Außerdem hängt auch beim Kalman Filter der Zustand zum Zeitpunkt t nur vom Zustand zum Zeitpunkt $t-1$ und die Messungen vom derzeitigen Zustand ab. Zusätzlich zu den Markov Annahmen werden beim Kalman Filter die Annahmen getroffen, dass die Abhängigkeiten zwischen den Zuständen linearer Natur, die Rauschterme gaußverteilt und die Messungen unabhängig voneinander sind. Die Annahme, dass es sich bei den Modellen um lineare Abhängigkeiten handelt, schlägt für die meisten Schätzprobleme fehl, da Zustandsübergänge und Messabhängigkeiten oft nichtlineare Anteile haben. Für diesen Fall existieren Weiterentwicklungen des Kalman Filters, wie der Extended und Unscented Kalman Filter, die nichtlineare Dynamik- und Messmodelle unterstützen.

2.2.1 Extended Kalman Filter

Der Extended Kalman Filter (EKF) ist eine Weiterentwicklung des Kalman Filters für nichtlineare Systeme. Die Grundidee dieses Filters ist die Linearisierung der nichtlinearen Modelle am momentanen Erwartungswert und die Anwendung der Gleichungen des klassischen Kalman Filters.

Anstelle der linearen Modellmatrizen werden im EKF eine Funktion $f(x,u) : \mathbb{R}^n \times \mathbb{R}^p \mapsto \mathbb{R}^n$ als Dynamikmodell und eine Funktion $h(x) : \mathbb{R}^n \mapsto \mathbb{R}^m$ als Messmodell verwendet. Da durch die Kovarianzmatrix eine mehrdimensionale Gaußverteilung parametrisiert wird, müssen lineare Abbildungen als Übertragungsfunktion verwendet werden. Ansonsten wäre die resultierende Verteilung nicht mehr gaußverteilt. Aus diesem Grund werden die Jacobimatrizen der Modellfunktionen am Erwartungswert berechnet, um die Kovarianz fortschreiten zu lassen [31, S. 42].

$$
\begin{aligned}
&\text{Dynamik:} \\
&\bar{\mu}_t = f(\mu_{t-1}, u_t) \\
&F_t = \left.\frac{\partial f(x,u)}{\partial x}\right|_{x=\mu_{t-1}, u=u_t} \\
&\bar{\Sigma}_t = F_t \Sigma_{t-1} F_t^T + Q_t
\end{aligned}
\tag{2.7}
$$

Messung:

$$
\begin{aligned}
H_t &= \left.\frac{\partial h(x)}{\partial x}\right|_{x=\bar{\mu}_t} \\
K_t &= \bar{\Sigma}_t H_t^T (H_t \bar{\Sigma}_t H_t^T + R_t)^{-1} \\
\mu_t &= \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t)) \\
\Sigma_t &= \bar{\Sigma}_t - K_t H_t \bar{\Sigma}_t
\end{aligned}
\tag{2.8}
$$

Im Gegensatz zum klassischen Kalman Filter ist der EKF durch die Linearisierung kein Optimalfilter. Außerdem können durch die Linearisierung bei nichtlinearen Funktionen hohen Grades Fehler in der Schätzung entstehen. In den meisten Fällen reicht die Performanz eines EKF aber aus und er ist einer der meistverwendeten Algorithmen in der nichtlinearen Zustandsschätzung [12][22, S. 296][31, S. 61].

2.2.1.1 EKF mit nicht-additivem Rauschen

Die ursprüngliche Formulierung des Kalman Filters betrachtet das Dynamikrauschen Q_t und Messrauschen R_t als additiv. Das bedeutet, dass dieses nur über eine Addition in die Schätzung der Kovarianz einfließt. Im Falle der Messung macht dies auch Sinn, da die Wahrscheinlichkeitsverteilungen der modellierten Messung und der Beobachtung summiert werden müssen um eine gesamte Unwahrscheinlichkeitsschätzung zu erhalten. Möchte man aber zum Beispiel in der Dynamik das Rauschen in Abhängigkeit der Zustandsübergangsmessung u_t modellieren, sind die klassischen EKF Gleichungen nicht mehr anwendbar. Für diesen Fall können diese durch eine allgemeinere Formulierung ersetzt werden, welche es erlaubt die Angriffspunkte und das Verhalten des Rauschens in den Modellen selbst zu definieren. Dafür werden Rauschparameter $w_t \sim \mathcal{N}(0, Q_t)$ und $v_t \sim \mathcal{N}(0, R_t)$ in das Dynamik- und Messmodell eingeführt, über welche definiert wird an welchen Stellen das mehrdimensionale Rauschen in den Modellen angreift. Durch Differenzierung über die neuen Parameter kann erreicht werden das Rauschen korrekt in die Schätzung der Kovarianz zu integrieren [22, S. 400]:

Dynamik:

$$
\begin{aligned}
\bar{\mu}_t &= f(\mu_{t-1}, u_t, 0) \\
F_t &= \left.\frac{\partial f(x,u,w)}{\partial x}\right|_{x=\mu_{t-1}, u=u_t, w=0} \\
L_t &= \left.\frac{\partial f(x,u,w)}{\partial w}\right|_{x=\mu_{t-1}, u=u_t, w=0} \\
\bar{\Sigma}_t &= F_t \Sigma_{t-1} F_t^T + L_t Q_t L_t^T
\end{aligned}
\tag{2.9}
$$

Messung:

$$
\begin{aligned}
H_t &= \left.\frac{\partial h(x,v)}{\partial x}\right|_{x=\bar{\mu}_t, v=0} \\
M_t &= \left.\frac{\partial h(x,v)}{\partial v}\right|_{x=\bar{\mu}_t, v=0} \\
K_t &= \bar{\Sigma}_t H_t^T (H_t \bar{\Sigma}_t H_t^T + M_t R_t M_t^T)^{-1} \\
\mu_t &= \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t, 0)) \\
\Sigma_t &= \bar{\Sigma}_t - K_t H_t \bar{\Sigma}_t
\end{aligned}
\tag{2.10}
$$

Indem der Rauschvektor in den Modellen mit dem Ergebnis aufsummiert wird, lässt sich äquivalentes Verhalten zu der klassischen EKF Formulierung erreichen. Da in diesem Fall die Jacobimatrizen L_t und M_t Einheitsmatrizen sind.

Verglichen mit der klassischen benötigt die nicht-additive Formulierung die doppelte Anzahl an Jakobimatrizen und in der Ausführung vier weitere Matrixmultiplikationen. Daher sollte die nicht-additive Formulierung nur verwendet werden, wenn großer Wert auf Präzision und korrekte Rauschmodellierung gelegt wird.

2.2.1.2 Second Order Extended Kalman Filter

Da einfache Linearisierung von den Modellen eines EKF in manchen Fällen zu Fehlern führen kann, kann es nützlich sein diese über eine weitere Entwicklung anzunähern. Dazu wurden Extended Kalman Filter höherer Ordnung entwickelt und der EKF zweiter Ordnung soll an dieser Stelle vorgestellt werden.
Die Kovarianzpropagation ist identisch zu der des klassischen EKF. Für die Berechnung des neuen Erwartungswertes wird die zweite Taylorentwicklung des Modells um den momentanen Erwartungswert berechnet und als neuer Summand eingeführt. Der Dynamikschritt des EKF zweiter Ordnung ist folgendermaßen definiert [22, S. 419]:

$$
\begin{aligned}
C_{t,i} &= \left.\frac{\partial^2 f_i(x,u)}{\partial x^2}\right|_{x=\mu_{t-1}, u=u_t} \\
\bar{\mu}_t &= f(\mu_{t-1}, u_t) + \frac{1}{2}\sum_{i=1}^{n} \vec{e}_i \mathrm{Tr}\left[C_{t,i}\Sigma_{t-1}\right] \\
F_t &= \left.\frac{\partial f(x,u)}{\partial x}\right|_{x=\mu_{t-1}, u=u_t} \\
\bar{\Sigma}_t &= F_t \Sigma_{t-1} F_t^T + Q_t
\end{aligned}
\tag{2.11}
$$

Die Matrix $C_{t,i}$ ist die Hesse-Matrix der i-ten Komponente des Dynamikmodells. In der Berechnung des neuen Erwartungswerts entspricht Tr[X] der Spur der Matrix X und $\vec{e}_i$ dem i-ten kanonischen Einheitsvektor.

Der Messschritt des EKF zweiter Ordnung ist folgendermaßen definiert:

$$\begin{aligned} H_t &= \left.\frac{\partial h(x)}{\partial x}\right|_{x=\bar{\mu}_t} \\ K_t &= \bar{\Sigma}_t H_t^T (H_t \bar{\Sigma}_t H_t^T + R_t)^{-1} \\ D_{t,i} &= \left.\frac{\partial^2 h_i(x)}{\partial x^2}\right|_{x=\bar{\mu}_t} \\ \mu_t &= \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t)) - \frac{1}{2} K_t \sum_{i=1}^{n} \vec{e}_i \mathrm{Tr}\left[D_{t,i}\bar{\Sigma}_t\right] \\ \Sigma_t &= \bar{\Sigma}_t - K_t H_t \bar{\Sigma}_t \end{aligned} \tag{2.12}$$

Wobei die Matrix $D_{t,i}$ die Hesse-Matrix der i-ten Komponente des Messmodells ist. Dieser EKF bietet bessere Performanz als ein klassischer EKF [22, S. 418] benötigt aber die zweiten Ableitungen der Modelle in Form von Hesse-Matrizen, welche aufwändig zu berechnen sind.

2.2.2 Unscented Kalman Filter

Der Unscented Kalman Filter (UKF) ist ein weiterer Ansatz einen Kalman Filter für nichtlineare Modelle zu implementieren. Im Gegensatz zum EKF verspricht der UKF höhere Präzision für hochgradig nichtlineare Modelle und einfachere Benutzbarkeit, da keine Jacobimatrizen der Modelle mehr benötigt werden [12].

Der UKF verwendet zur Fehlerfortpflanzung keine vorgegebenen Jacobimatrizen, sondern die sogenannte Unscented Transform [13]. Diese wertet die Modellfunktion an verschiedenen Punkten aus und berechnet aus der resultierenden Verteilung den neuen Erwartungswert und die neue Kovarianz. Die Auswahl dieser sogenannten Sigmapunkte $\mathcal{X}$ erfolgt dabei nicht zufällig, sondern deterministisch in Abhängigkeit von der Kovarianz Σ und Gewichtungsfaktoren W um den Erwartungswert μ herum. Die Sigmapunkte und Gewichte des Dynamikschritts werden folgendermaßen berechnet:

$$\begin{aligned} \mathcal{X}_{t-1}^{(0)} &= \mu_{t-1} & W^{(0)} &= \frac{\kappa}{n+\kappa} \\ \mathcal{X}_{t-1}^{(i)} &= \mu_{t-1} + \left(\sqrt{(n+\kappa)\Sigma_{t-1}}\right)_i & W^{(i)} &= \frac{1}{2(n+\kappa)} \\ \mathcal{X}_{t-1}^{(n+i)} &= \mu_{t-1} - \left(\sqrt{(n+\kappa)\Sigma_{t-1}}\right)_i & W^{(i+n)} &= \frac{1}{2(n+\kappa)} \end{aligned} \tag{2.13}$$

Dabei ist n die Dimension des Zustands μ und κ ein Skalierungsparameter. Dieser Skalierungsparameter liefert bei $n + \kappa = 3$ für gaußverteilte Wahrscheinlichkeitsdichten ein optimales Ergebnis [12]. Die Quadratwurzel der skalierten Kovarianz kann umgesetzt werden durch die Cholesky-Zerlegung.
Die berechneten Sigmapunkte können nun dazu verwendet werden in dem Dynamikschritt

den neuen Erwartungswert und die neue Kovarianz zu berechnen:

$$\begin{aligned}
\bar{\mathcal{X}}_t^{*(i)} &= f(\mathcal{X}_{t-1}^{(i)}, u_t) \\
\bar{\mu}_t &= \sum_{i=0}^{2n} W^{(i)} \bar{\mathcal{X}}_t^{*(i)} \\
\bar{\Sigma}_t &= \sum_{i=0}^{2n} W^{(i)} \left(\bar{\mathcal{X}}_t^{*(i)} - \bar{\mu}_t\right) \left(\bar{\mathcal{X}}_t^{*(i)} - \bar{\mu}_t\right)^T + Q_t
\end{aligned} \tag{2.14}$$

Ähnlich wie im Dynamikschritt werden im Messschritt wieder Sigmapunkte und Gewichte aus der neuen Kovarianz generiert:

$$\begin{aligned}
\bar{\mathcal{X}}_t^{(0)} &= \bar{\mu}_t & W^{(0)} &= \frac{\kappa}{n+\kappa} \\
\bar{\mathcal{X}}_t^{(i)} &= \bar{\mu}_t + \left(\sqrt{(n+\kappa)\bar{\Sigma}_t}\right)_i & W^{(i)} &= \frac{1}{2(n+\kappa)} \\
\bar{\mathcal{X}}_t^{(n+i)} &= \bar{\mu}_t - \left(\sqrt{(n+\kappa)\bar{\Sigma}_t}\right)_i & W^{(i+n)} &= \frac{1}{2(n+\kappa)}
\end{aligned} \tag{2.15}$$

Nach der Generierung der Sigmapunkte kann dann der Messschritt auf Basis dieser ausgeführt werden:

$$\begin{aligned}
\bar{\mathcal{Z}}_t^{(i)} &= h(\bar{\mathcal{X}}_t^{(i)}) \\
\hat{z}_t &= \sum_{i=0}^{2n} W^{(i)} \bar{\mathcal{Z}}_t^{(i)} \\
\Sigma_t^z &= \sum_{i=0}^{2n} W^{(i)} \left(\bar{\mathcal{Z}}_t^{(i)} - \hat{z}_t\right) \left(\bar{\mathcal{Z}}_t^{(i)} - \hat{z}_t\right)^T + R_t \\
\Sigma_t^{x,z} &= \sum_{i=0}^{2n} W^{(i)} \left(\bar{\mathcal{X}}_t^{(i)} - \bar{\mu}_t\right) \left(\bar{\mathcal{Z}}_t^{(i)} - \hat{z}_t\right)^T \\
K_t &= \Sigma_t^{x,z} \left(\Sigma_t^z\right)^{-1} \\
\mu_t &= \bar{\mu}_t + K_t \left(z_t - \hat{z}_t\right) \\
\Sigma_t &= \bar{\Sigma}_t - K_t \left(\Sigma_t^{x,z}\right)^T
\end{aligned} \tag{2.16}$$

Diese komplexe Berechnung des neuen Zustandes führt dazu, dass der Erwartungswert bis zur dritten Ordnung korrekt bestimmt wird und durch optimieren des Skalierungsparameters κ eine Fehlerminimierung in der vierten Ordnung erreicht werden kann [22, S. 454]. Da aber zur Generierung der Sigmapunkte eine Cholesky-Zerlegung durchgeführt werden muss und die Modelle für jeden Sigmapunkt evaluiert werden müssen, ist dieser Ansatz, besonders für große Zustände, vergleichsweise langsam.

2.3 Mannigfaltigkeiten

Mannigfaltigkeiten beschreiben mathematische Räume, die sich lokal wie ein euklidischer Vektorraum verhalten aber global eine komplexe topologische Struktur haben können [10]. Schätzalgorithmen wie der Kalman Filter arbeiten im Normalfall nur auf Zuständen $x \in \mathbb{R}^n$. Möchte man aber beispielsweise eine Rotation im zweidimensionalen Raum schätzen, könnte man denken, dass es ausreicht einen einfachen Winkel $\alpha \in [-\pi, \pi)$ in den Zustand aufzunehmen. Erreicht dieser Wert die Grenze des Intervalls, reicht eine kleine Veränderung des Zustands aus um eine große Veränderung des Parameters auszulösen. Dieses Verhalten wird als Singularität bezeichnet und kann durch eine Überparametrisierung des Zustands umgangen werden. Eine mögliche Überparametrisierung der zweidimensionalen Rotationen ist die Gruppe der 2D-Rotationsmatrizen $SO(2) := \{Q \in \mathbb{R}^{2\times 2} \mid QQ^T = Q^TQ = I \wedge det(Q) = 1\}$. Es ist aber nicht möglich die Eigenschaften dieser Matrizen im Kalman Filter zu erhalten.

Eine Lösung dieses Problems ist die Variable global überzuparametrisieren und lokal für Veränderungen eine Minimalparametrisierung zu verwenden, welche sich für kleine Werte wie ein euklidischer Raum verhält [10]. Um diese Variable in Algorithmen wie dem Kalman Filter verwenden zu können werden Operatoren eingeführt, mit welchen der Filter auf die Variable zugreifen kann. Der erste Operator ist der $\boxplus$-Operator, mit welchem eine vektorielle Änderung auf eine Mannigfaltigkeit addiert werden kann:

$$\boxplus : \mathcal{S} \times \mathbb{R}^n \mapsto \mathcal{S} \tag{2.17}$$

Der zweite Operator ist der $\boxminus$-Operator, welche die Differenz von zwei Mannigfaltigkeiten als Vektor berechnet:

$$\boxminus : \mathcal{S} \times \mathcal{S} \mapsto \mathbb{R}^n \tag{2.18}$$

Die Definition der Operatoren ist für alle $x \in \mathcal{S}$ an folgende Axiome gebunden:

$$x \boxplus 0 = x \tag{2.19}$$

$$\forall y \in \mathcal{S} : x \boxplus (y \boxminus x) = y \tag{2.20}$$

$$\forall \delta \in \mathbb{R}^n : (x \boxplus \delta) \boxminus x) = \delta \tag{2.21}$$

$$\forall \delta_1, \delta_2 \in \mathbb{R}^n : \|(x \boxplus \delta_1) \boxminus (x \boxplus \delta_2)\| \leq \|\delta_1 - \delta_2\| \tag{2.22}$$

Für eine mathematische Herleitung dieser Operatoren aus der Theorie der Mannigfaltigkeiten siehe [10].

Die Eleganz dieses Ansatzes liegt darin, dass in Schätzalgorithmen im Normalfall nur die $+$ und $-$ Operatoren gegen $\boxplus$ und $\boxminus$ ausgetauscht werden müssen. Der Schätzalgorithmus arbeitet dann nur auf den Veränderungen zwischen zwei Mannigfaltigkeiten, welche als Vektor dargestellt werden. Im Falle des EKF und UKF müssen leichte Veränderungen vorgenommen werden. Die Berechnung der Jacobimatrizen im EKF muss auf eine spezielle Art geschehen (ab Formel 2.23) und der Erwartungswert im UKF wird iterativ an den korrekten Wert

angenähert [10].

Um aus einer Funktion, die eine Mannigfaltigkeit als Funktionsargument oder Funktionswert besitzt, die Jacobimatrix am Punkt x_0 zu berechnen, muss eine der folgenden Formeln verwendet werden [35]:

$$f : \mathcal{S} \mapsto \mathcal{S} \qquad \frac{\partial f(x \boxplus \delta) \boxminus f(x)}{\partial \delta}\bigg|_{x=x_0,\delta=0} \tag{2.23}$$

$$f : \mathcal{S} \mapsto \mathbb{R}^n \qquad \frac{\partial f(x \boxplus \delta)}{\partial \delta}\bigg|_{x=x_0,\delta=0} \tag{2.24}$$

$$f : \mathbb{R}^n \mapsto \mathcal{S} \qquad \frac{\partial f(x + \delta) \boxminus f(x)}{\partial \delta}\bigg|_{x=x_0,\delta=0} \tag{2.25}$$

Aus diesen Gleichungen folgt die Definition des EKF auf Mannigfaltigkeiten [23]:

Dynamik:

$$\begin{aligned}
\bar{\mu}_t &= f(\mu_{t-1}, u_t, 0) \\
F_t &= \frac{\partial f(x \boxplus \delta, u, w) \boxminus f(x, u, w)}{\partial \delta}\bigg|_{x=\mu_{t-1},u=u_t,w=0} \\
L_t &= \frac{\partial f(x, u, w + \delta) \boxminus f(x, u, w)}{\partial \delta}\bigg|_{x=\mu_{t-1},u=u_t,w=0} \\
\bar{\Sigma}_t &= F_t \Sigma_{t-1} F_t^T + L_t Q_t L_t^T
\end{aligned}$$

Messung:

$$\begin{aligned}
H_t &= \frac{\partial h(x \boxplus \delta, v) \boxminus_{\mathcal{M}} h(x, v)}{\partial \delta}\bigg|_{x=\bar{\mu}_t,v=0} \\
M_t &= \frac{\partial h(x, v + \delta) \boxminus_{\mathcal{M}} h(x, v)}{\partial \delta}\bigg|_{x=\bar{\mu}_t,v=0} \\
K_t &= \bar{\Sigma}_t H_t^T (H_t \bar{\Sigma}_t H_t^T + M_t R_t M_t^T)^{-1} \\
y_t &= K_t (z_t \boxminus_{\mathcal{M}} h(\bar{\mu}_t, 0)) \\
D_t &= \frac{\partial ((x \boxplus \delta) \boxplus y) \boxminus (x \boxplus y)}{\partial \delta}\bigg|_{x=\bar{\mu}_t,y=y_t} \\
\mu_t &= \bar{\mu}_t \boxplus y_t \\
\Sigma_t &= D_t (\bar{\Sigma}_t - K_t H_t \bar{\Sigma}_t) D_t^T
\end{aligned} \tag{2.26}$$

Wobei $\boxminus_{\mathcal{M}}$ der $\boxminus$-Operator einer Mannigfaltigkeitsmessung ist. Im Messschritt wird zusätzlich zur Definition des EKF (siehe Kapitel 2.2.1.1) eine weitere Jacobimatrix D_t berechnet. Diese ist notwendig, da die Kovarianzmatrix $\bar{\Sigma}_t$ auf dem lokalen Vektorraum um $\bar{\mu}_t$ definiert ist und im allgemeinen $\mu \boxplus \mathcal{N}(y, \Sigma) \neq (\mu \boxplus y) \boxplus \mathcal{N}(0, \Sigma)$ gilt [23]. Wobei μ der alte Erwartungswert, y die Innovation und Σ eine Kovarianzmatrix ist. Daher muss die Kovarianz des Zustands in das Koordinatensystem um den neuen Erwartungswert $\bar{\mu}_t \boxplus y$ transformiert werden.

Kapitel 3

Framework

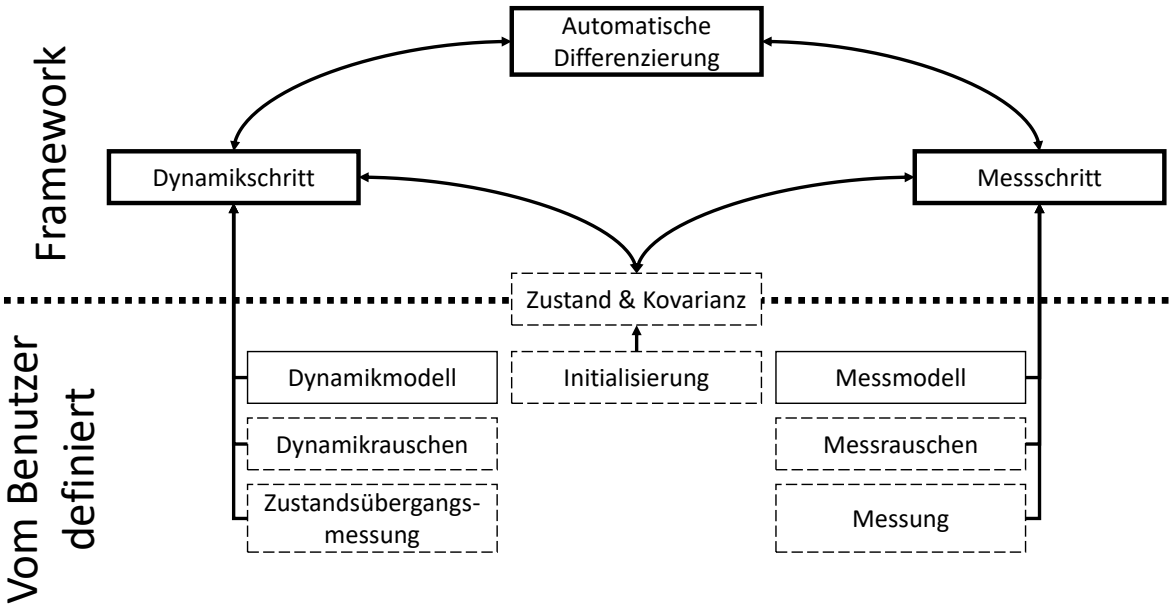

Abbildung 3.1 Übersicht über das entwickelte Framework.
Durchgezogene Rahmen entsprechen Funktionen und gestrichelte Rahmen Variablen

In dieser Arbeit wurde ein Framework entwickelt, welches einen EKF implementiert, ohne dass der Benutzer Jacobimatrizen definieren muss. Dabei unterstützen der EKF und die Funktionen die Aufnahme von Mannigfaltigkeiten als Zustand und Messung. Die Berechnung der Jacobimatrizen geschieht auf Basis von automatischer Differenzierung, da diese schnelle und exakte Ergebnisse liefert. Daher auch der Name des Frameworks: Ein **A**utomatisch **D**ifferenzierter **E**xtended **K**alman **F**ilter – **ADEKF**.

Wie in Abb. 3.1 dargestellt, hat der Benutzer nach Initialisierung von Zustand und Kovarianz die Möglichkeit einen Dynamikschritt oder einen Messschritt auszuführen. Für einen Dynamikschritt muss der Benutzer dabei nur das Dynamikmodell, das Dynamikrauschen in Form einer Kovarianzmatrix und eine Zustandsübergangsmessung übergeben. Die Berechnung der

Jacobimatrizen findet innerhalb des Frameworks über automatische Differenzierung statt. Die Jacobimatrizen für den Messschritt werden ebenfalls innerhalb des Frameworks berechnet und der Benutzer muss nur ein Messmodell, das Messrauschen in Form einer Kovarianzmatrix und eine Messung übergeben.

Technisch gesehen handelt es sich um eine in C++ geschriebene header-only Bibliothek. Da die Bibliothek nur aus Headerdateien besteht, ist es sehr einfach sie in ein bestehendes Projekt zu integrieren. Die einzige obligatorische externe Abhängigkeit ist Eigen [5], eine Bibliothek für lineare Algebra. Intern werden Dateien aus dem Ceres Solver [3] für die Implementierung der dualen Zahlen (siehe Kapitel 3.1) und verschiedenen Hilfsfunktionen verwendet. Eine optionale Abhängigkeit ist der Präprozessor von Boost [2]. Dieser wird nur zur Konstruktion von zusammengesetzten Mannigfaltigkeiten (siehe Kapitel 3.2.2) verwendet.

Die Bibliothek besteht aus sieben Dateien, auf welche hier etwas genauer eingegangen werden soll:

ADEKF.h Dies ist die Hauptklasse des Frameworks. Sie beinhaltet eine Implementierung des Extended Kalman Filters, sowie Funktionalität zur automatischen Bestimmung von Jacobimatrizen. Im Normalfall ist dies die einzige Datei, die vom Benutzer eingebunden werden muss.

ADEKFUtils.h Diese Datei enthält Funktionen zur Dimensionsbestimmung von Matrizen und Mannigfaltigkeiten und interne Hilfsfunktionen für den EKF.

SO2.h Diese Klasse implementiert die Mannigfaltigkeit $SO(2)$, die Gruppe der 2D-Rotationsmatrizen. Zur Schätzung von 2D-Rotationen im ADEKF.

SO3.h Diese Klasse implementiert die Mannigfaltigkeit $SO(3)$, die Gruppe der 3D-Rotationsmatrizen. Zur Schätzung von 3D-Rotationen im ADEKF.

jet.h Eine Implementierung der dualen Zahlen. Diese Datei stammt aus der Ceres Solver Bibliothek.

rotation.h Implementierung von Hilfsfunktionen zur Arbeit mit Quaternionen. Diese Datei stammt aus der Ceres Solver Bibliothek.

ManifoldCreator.h Diese Datei stellt Funktionalitäten zur einfachen Definition von zusammengesetzten Mannigfaltigkeiten bereit.

3.1 Implementierung und Verwendung der dualen Zahlen

Die Implementierung der dualen Zahlen stammt aus dem Ceres Solver [3]. Diese Bibliothek ist ein Open Source Projekt von Google und enthält Werkzeuge zum Lösen von Optimierungsproblemen. Es wurde sich für diese Implementierung entschieden, da der Ceres Solver stetig weiterentwickelt und viel getestet wird.

Damit diese Klasse von Zahlen mit allen in der Standard Bibliothek enthaltenen Funktionen benutzbar ist, werden diese in der Implementierung überladen und berechnen nicht nur das Funktionsergebnis, sondern auch die bestimmte Ableitung durch Anwendung der Kettenregel. Intern wird die Ableitung nicht als Zahl sondern als Vektor gespeichert, damit die Ableitung multivariater Funktionen möglich wird. Die Implementierung dieser dualen Zahlen, welche vom Ceres Solver `Jets` genannt werden, sieht am Beispiel der binären Plus- und Sinusfunktion in etwa folgendermaßen aus:

```
template <typename T, int N>
struct Jet {
  ...
  template <typename Derived>
  Jet(const T& a, const Eigen::DenseBase<Derived>& v)
      : a(a), v(v) {}
  ...
  T a;
  Eigen::Matrix<T, N, 1> v;
};
...
template <typename T, int N>
Jet<T, N> operator+(const Jet<T, N>& f, const Jet<T, N>& g) {
  return Jet<T, N>(f.a + g.a, f.v + g.v);
}
...
using std::sin;
...
template <typename T, int N>
Jet<T, N> sin(const Jet<T, N>& f) {
  return Jet<T, N>(sin(f.a), cos(f.a) * f.v);
}
...
```

Diese Art der Implementierung führt dazu, dass alle Funktionen die auf die Verwendung mit klassischen Gleitkommazahlen ausgelegt sind auch auf dualen Zahlen funktionieren. Um nicht jede Funktion für jeden Typ neu schreiben zu müssen ist es möglich C++-Template Typen anstelle eines festen Datentyps zu verwenden. Solange die Funktionsargumente und Ergebnisse von diesem Template Typ abhängig sind, ist es möglich die Funktion mit dualen Zahlen abzuleiten. Es ist nicht möglich Funktionstemplates ohne Instanzierung an eine andere Klasse zu übergeben. Um den Benutzer von den dualen Zahlen zu entkoppeln, können Funktoren mit getemplatetem Klammeroperator verwendet werden. Diese erlauben es den Funktor an eine Klasse zu übergeben und die Templateinitialisierung erst bei der Ausführung des Funktors vorzunehmen.

```
struct Funktor {
  template <typename T>
  T operator()(T& input) {
    T result;
    ... //Implementation
    return result;
  }
} functor;
```

Um diesen Funktor abzuleiten, muss ein `Jet` entsprechender Größe an diesen übergeben werden. Die Größe richtet sich dabei nach der Anzahl von Funktionsargumenten. Danach wird für jedes Funktionsargument ein `Jet` erstellt, der an der entsprechenden Stelle mit 1 initialisiert ist. Für diese Initialisierung ist im Ceres Solver ein Konstruktor für `Jets` definiert. Führt man nun den Funktor mit `Jets` als Parameter aus, sind die Spalten der Jacobimatrix direkt aus dem resultierenden `Jet` auslesbar. Hier am Beispiel ein `Jet` mit 10 als reellem Anteil:

```
ceres::Jet<double, 1> result = functor(ceres::Jet<double, 1>(10, 0));
```

Für die Berechnung der Jacobimatrix einer multivariaten und multidimensionalen Funktion wird zuerst ein entsprechender Funktor definiert:

```
struct MultiFunctor {
  template <typename T>
  Matrix<T, 3, 1> operator()(Matrix<T, 5, 1>& input) {
    Matrix<T, 3, 1> result;
    ... //Implementation
    return result;
  }
} multiFunctor;
```

Der Funktor erwartet einen Vektor der Größe 5 und gibt einen Vektor der Größe 3 zurück. Die Konstruktion des `Jet`-Vektors geschieht folgendermaßen:

```
Matrix<ceres::Jet<double, 5>, 5, 1> input;

for (unsigned i = 0; i < 5; ++i)
  input[i] = ceres::Jet<ScalarType, Size>(0, i);
```

Es folgt die Ausführung des Funktors auf dem `Jet`-Vektor und die Extraktion der Jacobimatrix aus dem Ergebnis:

```
Matrix<double, 5, 1> x0 = ...; //Calculate the Derivative at the Point x0

Matrix<ceres::Jet<double, 5>, 3, 1> result = multiFunctor(x0 + input);

Matrix<double, 3, 5> jacobian;

for (int i = 0; i < 3; ++i)
  jacobian.row(i) = result[i].v;
```

Jede Zeile der resultierenden Jacobimatrix entspricht dem dualen Anteil des entsprechenden `Jet`-Objekts. Die Jacobimatrix enthält nun die bestimmten partiellen Ableitungen an der Stelle x_0.

3.2 Interface

In diesem Kapitel wird die Verwendung des ADEKF beschrieben. Zuerst die Initialisierung des Filters und die grundlegende Verwendung. In Kapitel 3.2.1 wird die Anwendung von nichtadditivem Rauschen (siehe Kapitel 2.2.1.1) erläutert und in Kapitel 3.2.2 der Aufbau

und die Verwendung von Mannigfaltigkeiten.
Der Konstruktor erwartet eine Initialisierung des Zustands und der Kovarianz. Der Typ des Zustands wird über den Templateparameter der `EKF`-Klasse vom Benutzer festgelegt und der Typ der Kovarianz ist immer eine Eigen-Matrix in der entsprechenden Größe und dem Zahlentyp des Zustands. Eine Initialisierung des Filters mit einem Nullvektor als Zustand und einer Einheitsmatrix als Kovarianz geschieht folgendermaßen:

```
#include "ADEKF.h"

constexpr unsigned StateSize = ...; //The Dimension of the State
using ScalarType = ...; //e.g. float or double

template<typename T>
using State = Eigen::Matrix<T, StateSize, 1>;
using Covariance = Eigen::Matrix<ScalarType, StateSize, StateSize>;

ADEKF::EKF ekf(State<ScalarType>::Zero(), Covariance::Identity());
```

Soll nun ein Dynamikschritt auf diesem Filter ausgeführt werden, kann dies mit der Funktion `predict` erreicht werden:

```
using Control = ...; //The Type of the Control Measurement

struct DynamicModel {
  template<typename T>
  void operator()(State<T>& state, const Control& u) {
    ... //Implementation of the Dynamic Model
  }
} f;

Covariance Q = ...; //Definition of the Covariance Matrix
Control u = ...; //Definition of the Control Measurement

ekf.predict(f, Q, u);
```

Der erste Parameter der `predict`-Funktion ist das Dynamikmodell. Dessen Klammeroperator muss nur einen Parameter annehmen und zwar eine Referenz auf den getemplateten Zustand. Der Rückgabetyp ist `void`. Innerhalb des Dynamikmodells können dann direkte Veränderungen am Zustand vorgenommen werden. Der dritte Parameter der `predict`-Funktion ist variadisch. Alle dort übergebenen Variablen werden ab dem zweiten Parameter in das Dynamikmodell eingesetzt. Dies erlaubt eine beliebige Anzahl von Variablen beliebigen Typs als Zustandsübergangsmessung an das Dynamikmodell zu übergeben. Der zweite Parameter der `predict`-Funktion ist die Kovarianzmatrix des Dynamikrauschens. Es ist wichtig, dass der Template-Typ des Zustands im Dynamikmodell noch nicht initialisiert ist, da dieser in der Implementierung des EKF mit dualen Zahlen initialisiert wird.

Der Messschritt kann mit der Funktion `update` ausgeführt werden:

```
constexpr unsigned MeasurementSize = ...; //Dimension of the Measurement

template<typename T>
using Measurement = Eigen::Matrix<T, MeasurementSize, 1>;
using MeasurementCovariance =
  Eigen::Matrix<ScalarType, MeasurementSize, MeasurementSize>;

struct MeasurementModel {
  template<typename T>
  Measurement<T> operator()(const State<T>& state, const Parameter& p) {
    Measurement<T> result = ...;
    ... //Implementation of the Measurement Model
    return result;
  }
} h;

Measurement<ScalarType> z = ...; //Definition of the Measurement
MeasureementCovariance R = ...; //Definition of the Measurement Covariance Matrix
Parameter p = ...; //Definition of auxiliary paramters

ekf.update(h, R, z, p);
```

Der erste Parameter der `update`-Funktion ist das Messmodell. Wie das Dynamikmodell, wird dieses als Funktor mit getemplatetem Klammeroperator implementiert. Anstelle einer einfachen Referenz auf den Zustand erwartet das Messmodell eine `const`-Referenz und der Rückgabetyp ist der Typ der Messung. Bei der Referenz auf den Zustand und dem Rückgabewert muss darauf geachtet werden, dass beide vom Templateparameter abhängig sein müssen. Wobei der Rückgabetyp auch auf `auto` gesetzt werden kann. Der zweite Parameter der `update`-Funktion ist die Kovarianzmatrix des Messrauschens. Der dritte Parameter ist die Messung, mit welcher der Zustand korrigiert werden soll. Hier ist es wichtig, dass der Typ der Messung mit dem Rückgabetyp des Messmodells übereinstimmt. Ähnlich wie beim Dynamikmodell ist es auch möglich an das Messmodell eine beliebige Anzahl von Parametern von beliebigem Typ zu übergeben. Zu diesem Zweck ist der vierte Parameter des Messmodells variadisch.

3.2.1 Interface für nichtadditives Rauschen

Analog zur Definition des EKF mit nichtadditiven Rauschtermen (siehe Kapitel 2.2.1.1), exisitert auch im `ADEKF`-Framework Funktionalität um diese Definition abzubilden. Der Dynamikschritt wird implementiert über die Funktion `predictWithNonAdditiveNoise`. Diese ist von der Verwendung sehr ähnlich zu der normalen Implementierung des Dynamikschritts. Nur die Dimension der Kovarianz und die Signatur des Dynamikmodells ist verschieden:

```
constexpr unsigned NoiseSize = ...; //Dimension of the Noise Vector

template<typename T>
using NoiseVector = Eigen::Matrix<T, NoiseSize, 1>;
using NoiseCovariance = Eigen::Matrix<ScalarType, NoiseSize, NoiseSize>;

using Control = ...; //The Type of the Control Measurement

struct DynamicModel {
  template<typename T>
  void operator()(State<T>& state, const NoiseVector<T>& noise, const Control& u) {
    ... //Implementation of the Dynamic Model
  }
} f;

NoiseCovariance Q = ...; //Definition of the Covariance Matrix
Control u = ...; //Definition of the Control Measurements

ekf.predictWithNonAdditiveNoise(f, Q, u);
```

Der erste Parameter des Dynamikmodells ist auch hier eine Referenz auf den Zustand. Der zweite muss aber ein Vektor sein, der die gleiche Anzahl von Zeilen hat wie die übergebene Kovarianzmatrix. Aus der Größe der übergebenen Kovarianzmatrix wird intern bestimmt, welche Größe der an das Modell zu übergebende Rauschvektor haben muss. Der Benutzer kann nun im Dynamikmodell angeben an welchen Stellen das Rauschen angreift und die Berechnung der zweiten Jacobimatrix wird intern vom Framework übernommen. Wichtig ist, dass die Größe der Kovarianzmatrix der quadratischen Größe des Rauschvektors entsprechen und der Rauschvektor von dem Templatetyp abhängen muss.

Der Messschritt kann über die Funktion `updateWithNonAdditiveNoise` mit nichtadditivem Rauschen ausgeführt werden. Ähnlich zur Verwendung von `predictWithNonAdditiveNoise` muss auch hier nur ein Rauschvektor in das Messmodell eingeführt werden, über welchen innerhalb des Frameworks differenziert werden kann:

```
constexpr unsigned NoiseSize = ...; //Dimension of the Noise Vector
constexpr unsigned MeasurementSize = ...; //Dimension of the Measurement

template<typename T>
using NoiseVector = Eigen::Matrix<T, NoiseSize, 1>;
using NoiseCovariance = Eigen::Matrix<ScalarType, NoiseSize, NoiseSize>;

template<typename T>
using Measurement = Eigen::Matrix<T, MeasurementSize, 1>;

struct MeasurementModel {
  template<typename T>
  Measurement<T> operator()(const State<T>& state, const NoiseVector<T>& noise, const
    Parameter& p) {
    Measurement<T> result = ...
    ... //Implementation of the Measurement Model
    return result;
  }
} h;

Measurement<ScalarType> z = ...; //Definition of the Measurement
NoiseCovariance R = ...; //Definition of the Covariance Matrix
Parameter p = ...; //Definition of auxiliary paramters

ekf.updateWithNonAdditiveNoise(h, R, z, p);
```

3.2.2 Mannigfaltigkeiten

Das Framework unterstützt Zustände in Form von Mannigfaltigkeiten. Neben den beiden bereits implementierten Mannigfaltigkeiten $SO(3)$ und $SO(2)$ ist es auch möglich eigene Mannigfaltigkeiten zu implementieren und mit dem ADEKF zu verwenden. Die Minimalanforderungen an diese Klassen sind:

- Es muss sich um Klassentemplate mit einem Templatetyp handeln.
- Nach der Instanzierung des Templates muss der Templatetyp in einer Typdefinition `ScalarType` auszulesen sein.
- Die Dimension der Minimalparametrisierung muss aus einer konstanten statischen Variable `DOF` auszulesen sein.
- Einen Plus-Operator, der einen Parameter vom Typ `Eigen::MatrixBase` erwartet und die Basisklasse mit dem korrekten Zahlentyp wieder zurück gibt.
- Einen Minus-Operator, der einen Parameter vom Typ der Basisklasse erwartet und einen Vektor mit dem korrekten Zahlentyp wieder zurück gibt.

Da es möglich sein muss die Plus- und Minus-Operatoren einer mit einem Gleitkommazahlentyp instantiierten Mannigfaltigkeit auch mit `Jets` zu verwenden, bedarf es einer Bestimmung des resultierenden Typs. Dies kann durch die `decltype` und `declval` Operatoren erreicht

werden. `decltype` gibt den Typ eines übergebenen Ausdrucks zurück. Da der resultierende Typ des Ausdrucks aber bekannt sein soll bevor der Ausdruck ausgeführt wird, muss `declval` verwendet werden. `declval` konvertiert einen übergebenen Typ in einen Referenztyp. Dies macht es möglich in einem `decltype`-Aufruf, nur auf Basis von Typen, einen Ausdruck zu schreiben, wovon `decltype` den resultierenden Typ bestimmt. Möchte man also eine Addition von Skalaren implementieren und den resultierenden Datentyp bereits zur Übersetzungszeit bestimmen, kann folgendermaßen vorgegangen werden:

```
template<typename Scalar1,
  typename Scalar2,
  typename ResultScalar =
    decltype(std::declval<Scalar1>() + std::declval<Scalar2>())>>
ResultScalar add(Scalar1 a, Scalar2 b) {
  return a + b;
}
```

Templatetypen können auch direkt an die Funktion übergeben oder aus den Funktionsparametern deduziert werden. Da der Rückgabetyp aber zur Übersetzungszeit bekannt sein muss, macht es aber Sinn, diesen auf dem vorgestellten Weg zu herauszufinden.

In C++14 ist es auch möglich `auto` als resultierenden Datentyp zu verwenden. Da Eigen aber die Ergebnisse von Operatoren und Funktionen nicht direkt ausrechnet, sondern diese in sogenannte Expression Templates kapselt und erst bei Zuweisung ausführt, ist es sicherer mit festen Datentypen zu arbeiten.

Eine mögliche Basisimplementierung einer Mannigfaltigkeit würde folgendermaßen aussehen:

```
template<typename Scalar>
class Manifold {
  private:
    ...
  public:
    constexpr unsigned DOF = ...; //The Degrees of Freedom of the Manifold

    using ScalarType = Scalar;

    template<typename Derived,
      typename OtherScalar = typename internal::traits<Derived>::Scalar,
      typename ResultScalar =
        decltype(std::declval<Scalar>() + std::declval<OtherScalar>())>
    Manifold<ResultScalar> operator+(const MatrixBase<Derived> &delta) const {
      Manifold<ResultScalar> result;
      ... //Implementation of the Boxplus Operator
      return result;
    }

    template<typename OtherScalar, typename ResultScalar =
      decltype(std::declval<Scalar>() - std::declval<OtherScalar>())>
    Matrix<ResultScalar,DOF,1> operator-(const Manifold<OtherScalar> &other) const {
      Matrix<ResultScalar,DOF,1> result;
      ... //Implementation of the Boxminus Operator
      return result;
    }
}
```

Durch `decltype` und `declval` wird zur Übersetzungszeit für jede Verwendung dieser Funktion der resultierende Rückgabetyp bestimmt. So müssen diese Funktionen nicht für `Jets` überladen werden.

Es besteht die Möglichkeit mehrere Mannigfaltigkeiten bzw. Mannigfaltigkeiten und Vektoren in einer zusammengesetzten Mannigfaltigkeit zusammenzufassen. Zu diesem Zweck existiert ein Makro, das für jede integrierte Mannigfaltigkeit und jeden integrierten Vektor die benötigten Operatoren und Variablen automatisch generiert. Um eine zusammengesetzte Mannigfaltigkeit `Pose3D` zu erstellen, welche eine Rotation des Typs `SO3` und eine Position des Typs `Matrix` enthält, wird folgendermaßen vorgegangen:

```
template<typename T>
struct Pose3D {
    ADEKF::SO3<T> rotation;
    Matrix<T,3,1> position;
    ADEKF_CONSTRUCT_MANIFOLD(Pose3D,(rotation, position))
};
```

Das Makro `ADEKF_CONSTRUCT_MANIFOLD` erwartet den Namen der Klasse und ein Tupel der Namen der Attribute. Dieses Makro erstellt folgende weitere Attribute und Funktionen:

- Eine Typdefinition `ScalarType`, die dem Instanziierungstyp des Templates entspricht.
- Eine statische konstante Variable `DOF`, welche die Summe aller `DOF`-Attribute aus gegebenen Mannigfaltigkeiten und Größen gegebener Matrizen enthält.

- Einen Plus-Operator, der einen Vektor der Größe `DOF` erwartet und die Basisklasse als Rückgabetyp besitzt.
- Einen Minus-Operator, der die Basisklasse als Argument erwartet und einen Vektor der Größe `DOF` zurück gibt.

Damit sind alle Anforderungen an Mannigfaltigkeitenklassen vom Anfang des Kapitels erfüllt. Wichtig ist, dass nur Eigen-Matrizen und nach dem oben genannten Schema erstellte Mannigfaltigkeitsklassen in das Makro übergeben werden können. Da die durch das Makro erstellen Klassen ebenfalls diesen Anforderungen entsprechen, ist es auch möglich diese in einem anderen Makro zu verwenden.

3.2.2.1 SO(2)

Eine der dem Framework zugehörigen Mannigfaltigkeitsimplementierungen ist die Gruppe der 2D-Rotationsmatrizen $SO(2)$. Zusammen mit dem Makro zum Erstellen von zusammengesetzten Mannigfaltigkeiten kann schnell eine Kalman Filter Konfiguration zum Tracking von Robotern in einer Ebene implementiert werden.

Die Klasse `SO2` erbt von der Eigen-Implementierung für 2D-Rotationen `Rotation2D`. Diese enthält viele Hilfsmethoden und speichert den Zustand als einfachen Winkel. Der Winkel muss nicht normalisiert abgespeichert werden solange die Differenz zweier Objekte normalisiert ist. Die ungefähre Implementierung der Klasse `SO2` ist folgendermaßen:

```
template<typename Scalar>
class SO2 : public Rotation2D<Scalar> {
  public:
    using ScalarType = Scalar;
    constexpr unsigned DOF = 1;

    SO2(const Scalar &angle = 0) : Rotation2D<Scalar>(angle) {};

    template<typename Derived,
      typename OtherScalar = typename internal::traits<Derived>::Scalar,
      typename ResultScalar =
        decltype(std::declval<Scalar>() + std::declval<OtherScalar>())>
    SO2<ResultScalar> operator+(const MatrixBase<Derived> &delta) const {
      return SO2<ResultScalar>(Rotation2D<Scalar>::angle() + delta[0]);
    }

    template<typename OtherScalar,
      typename ResultScalar =
        decltype(std::declval<Scalar>() - std::declval<OtherScalar>())>
    Matrix<ResultScalar,DOF,1> operator-(const SO2<OtherScalar> &other) const {
      return Matrix<ResultScalar,DOF,1>(
        normalize(Rotation2D<Scalar>::angle() - other.angle()));
    }
};
```

Wobei `normalize` eine Funktion ist, die einen Winkel im Intervall $(-\pi, \pi]$ normalisiert.

Für die Klasse `SO2` existieren die beiden Typdefinitionen `SO2f` und `SO2d`. Dabei handelt es sich um Kurzformen der Typen `SO2<float>` und `SO2<double>`.

3.2.2.2 SO(3)

Neben der Klasse `SO2` ist auch die Klasse `SO3` ein Teil der bereits implementierten Mannigfaltigkeiten. Diese kann zur allgemeinen Orientierungsschätzung im dreidimensionalen Raum eingesetzt werden.

Die Klasse `SO3` speichert die 3D-Orientierung als Quaternion. Die Minimaldarstellung dieser Mannigfaltigkeit ist ein dreidimensionaler Rotationsvektor. Da jede Rotation durch eine Achse und einen Winkel beschrieben werden kann, handelt es sich bei dieser Minimaldarstellung um den Achsenvektor, der um den Winkel skaliert wurde [10].

Der $\boxplus$- und $\boxminus$-Operator implementieren folgende Funktionen:

$$x \boxplus \delta = \exp\left(\frac{\delta}{2}\right) \qquad \exp(\delta) = \begin{pmatrix} \cos(\|\delta\|) \\ \operatorname{sinc}(\|\delta\|) \cdot \delta \end{pmatrix} \tag{3.1}$$

$$x \boxminus y = 2\log\left(y^{-1} \cdot x\right) \qquad log\left(\begin{pmatrix} w \\ v \end{pmatrix}\right) = \begin{cases} 0 & v = 0 \\ \frac{atan(\|v\|/w)}{\|v\|} v & v \neq 0, w \neq 0 \\ \pm\frac{\pi/2}{\|v\|} v & w = 0 \end{cases} \tag{3.2}$$

Der restliche Aufbau der Klasse `SO3` ist sehr ähnlich zu der Klasse `SO2`:

```
template<typename Scalar = float>
class SO3 : public Quaternion<Scalar> {
  public:
    using ScalarType = Scalar;
    constexpr unsigned DOF = 3;

    SO3(const Quaternion<Scalar> &src = Quaternion<Scalar>::Identity())
      : Quaternion<Scalar>(src) {};

    template<typename Derived,
      typename OtherScalar = typename internal::traits<Derived>::Scalar,
      typename ResultScalar =
        decltype(std::declval<Scalar>() + std::declval<OtherScalar>())>
    SO3<ResultScalar> operator+(const MatrixBase<Derived> &delta) const {
      ... //Implementation of Equation 3.1
    }

    template<typename OtherScalar,
    typename ResultScalar =
      decltype(std::declval<Scalar>() - std::declval<OtherScalar>())>
    Matrix<ResultScalar, DOF, 1> operator-(const SO3<OtherScalar> &other) const {
      ... //Implementation of Equation 3.2
    }
};
```

Für die Klasse `SO3` existieren die beiden Typdefinitionen `SO3f` und `SO3d`. Dabei handelt es sich um Kurzformen der Typen `SO3<float>` und `SO3<double>`.

3.3 Softwaretechnische Besonderheiten

Der ADEKF ist abhängig von C++14. Obwohl diese Version schon seit einigen Jahren in Verwendung ist, funktionieren nicht alle Funktionalitäten mit allen Compilern. Zum Zeitpunkt der Bearbeitung dieser Arbeit war es zum Beispiel nicht möglich den ADEKF mit dem *Microsoft Visual C Compiler* zu übersetzen. Der Grund dafür sind die internen Funktionen `predict_impl` und `update_impl`. Diese beiden Funktionen extrahieren die Jacobimatrizen aus den `Jets`, die aus den jeweiligen Modellen resultieren. Da für die Extraktion von Jacobimatrizen aus Vektoren und Mannigfaltigkeiten unterschiedlich vorgegangen werden muss, sind diese Funktionen überladen. Die erste Implementierung für Mannigfaltigkeiten empfängt einen generischen Templatetyp und die zweite für Vektoren nur Eigen-Vektoren von fester Größe. Im Fall, dass ein Vektor an diese Funktionen übergeben werden soll wird im Normalfall die zweite Implementierung ausgewählt, da diese spezialisierter ist als ein generisches Template. Der *MSVC Compiler* wählt aber nicht die spezialisierte Funktion, sondern gibt eine Fehlermeldung aus. Dies scheint ein Fehler im *MSVC Compiler* zu sein, da *gcc* und *clang* den Programmcode korrekt übersetzen.

Lambda Funktionen bieten dem Benutzer die Möglichkeit schnell und einfach lokale Variablen direkt in der Funktionsdefinition zu verwenden. Da es aber in C++14 noch nicht die Möglichkeit gibt Lambda Funktionstemplates zu erstellen, muss für den ADEKF ein Funktor verwendet werden, welcher nicht innerhalb einer Funktion definiert werden kann. Um trotzdem lokale Variablen an den Funktor übergeben zu können, ohne ihn in jedem Durchlauf neu zu konstruieren, wurde sich dafür entschieden variadische Argumente und `std::bind` in den Funktionen zum Dynamikschritt und Messschritt zu integrieren. Dieses Vorgehen soll an dieser Stelle am Beispiel der `predict`-Methode erläutert werden. Das Dynamikmodell, in Form einen Funktors mit einem Funktionstemplate als Klammeroperator, wird über einen generischen Templatetyp an die Funktion übergeben. Dies ist möglich weil der Objekttyp irrelevant ist solange der Klammeroperator implementiert ist. Die Zustandsübergangsmessungen werden über das Function Parameter Pack `u` an die Funktion übergeben. Dabei können verschieden viele Parameter auf einmal übergeben werden. Es ist auch möglich gar keine Parameter an ein Parameter Pack zu übergeben. Per Template Type Deduction werden die Typen aller übergebenen Parameter im Template Parameter Pack `Controls` definiert. Damit der Benutzer möglichst große Freiheit bei der Übergabe von weiteren Parametern hat, diese aber keinen Einfluss auf den unterliegenden Framework-Code haben, wird per `std::bind` ein neues Funktionsobjekt erstellt, welches bereits Referenzen auf die übergebenen Parameter enthält. Da der erste Parameter eines übergebenen Modells immer der Zustand sein muss, muss dieser ungebunden bleiben, damit die Funktion später auf `Jets` ausgeführt werden kann. Zu diesem Zweck kann `std::bind` mit Platzhaltern ausgeführt werden, die den Funktionsargumenten

im generierten Funktionsobjekt entsprechen. Im Beispiel ist dieser Platzhalter definiert als `_1`:

```
template<typename DynamicModel, typename... Controls>
void predict(DynamicModel dynamicModel, const Covariance &Q, const Controls ...u) {
  JacobianOf<State> F;
  auto f = std::bind(dynamicModel, _1, u...);
  auto input = eval(mu + getDerivator<DOF>());
  f(input);
  predict_impl(f, input, F);
  sigma = F * sigma * F.transpose() + Q;
}
```

Im Laufe der Tests hat sich herausgestellt, dass die Berechnung des neuen Zustands und der Kovarianz beschleunigt werden kann, wenn die verwendeten Datenstrukturen an Zweierpotenzen im Speicher ausgerichtet werden und prozessorspezifische Programme erzeugt werden. Dies ist bereits mit früheren Versionen von C++ möglich gewesen, aber die Version C++17 sorgt automatisch dafür, dass Datenstrukturen korrekt ausgerichtet sind. Da dies mit anderen Frameworks nicht funktioniert hat, wurde die Evaluation nicht mit diesen Einstellungen durchgeführt. Aber um den ADEKF zu beschleunigen bietet es sich an diesen mit C++17 zu kompilieren.

Die Implementierung der dualen Zahlen aus dem Ceres Solver verlangt zur Übersetzungszeit nach der Dimension der dualen Zahlen. Aus diesem Grund wurden auch die Matrizen innerhalb des ADEKF mit festen Größen definiert. Während der Evaluation hat sich heraus gestellt, dass Eigen-Matrizen fester Größe immer auf dem Stack alloziert werden. Daher kommt es zu Problemen wenn die Matrizen zu groß sind. Eigen setzt dieses Limit auf 2^{17} Byte pro Matrix. Das entspricht bei doppelter Fließkommagenauigkeit einer quadratischen Matrix der Größe 128^2. Da die Größe der Kovarianzmatrix von der Größe des Zustands abhängig ist, ist es bei doppelter Fließkommagenauigkeit momentan nicht möglich Zustände, die größer als 128 sind, mit dem ADEKF zu verwenden. Bei einfacher Fließkommagenauigkeit liegt dieses Limit bei 181.

Kapitel 4

Evaluation

In diesem Kapitel findet eine Evaluation des entwickelten Frameworks aus verschiedenen Gesichtspunkten statt. Im ersten Teil werden die Ergebnisse des ADEKF mit den Ergebnissen eines per Hand abgeleiteten EKF verglichen. Dies dient der Validierung des Frameworks, bevor im zweiten Teil der Evaluation die Geschwindigkeit des ADEKF mit anderen Kalman Filter Implementierungen verglichen wird. Die Evaluation endet mit einem Vergleich der Codekomplexität zwischen den getesteten Frameworks. Die ersten beiden Teile der Evaluation werden basierend auf drei Schätzproblemen durchgeführt, welche an dieser Stelle erläutert werden sollen.

Das erste Schätzproblem ist ein klassisches *Simultaneous Localization and Mapping* Problem basierend auf dem DLR (*Deutsches Zentrum für Luft- und Raumfahrt*) Spatial Cognition Dataset von Udo Frese [7]. Dieses wird zur Validierung des Frameworks (Kapitel 4.1) und im Rahmen des Laufzeittests ohne Mannigfaltigkeiten (Kapitel 4.2) eingesetzt. In diesem Dataset fährt ein mobiler Roboter durch die Räumlichkeiten des DLR und nimmt mit einer Kamera kreisförmige Landmarken auf. Teil des Datasets sind auch die Assoziationen der Landmarken, um SLAM Algorithmen zu testen.

Das zweite Schätzproblem ist die Orientierungsschätzung einer Handfläche durch eine auf dem Handrücken angebrachte Inertiale Messeinheit (IMU). Dabei soll die Korrektur der Orientierung über ein 3D-Trackingsystem geschehen, indem die vom Trackingsystem gemessene Orientierung direkt als Messung an den Kalman Filter übergeben werden kann. Da die Darstellung der Orientierung im Zustand als Mannigfaltigkeit $SO(3)$ geschehen soll, müssen die verwendeten Filter das Schätzen von Mannigfaltigkeiten und Mannigfaltigkeitsmessungen unterstützen. Die im Rahmen dieser Arbeit aufgenommenen IMU und 3D-Tracking Daten wurden für den Laufzeitvergleich mit Mannigfaltigkeiten (Kapitel 4.3) verwendet.

Das dritte Schätzproblem ist die Orientierungsschätzung eines NAO Roboters [24]. Diese Roboter werden im Rahmen des Studentenprojekts B-Human [21] programmiert, sodass diese jedes Jahr bei internationalen Wettkämpfen im Roboterfußball antreten können. Da auf der Orientierungsschätzung viele weitere Funktionen des Roboters aufsetzen ist eine schnelle und

genaue Berechnung notwendig für ein erfolgreiches Fußballspiel. In Kapitel 4.4 wird ein Laufzeitvergleich zwischen dem ADEKF und der bereits im B-Human Framework vorhandenen Orientierungsschätzung durchgeführt.

Abgesehen von der Validierung des Frameworks wurden keine Präzisionsuntersuchungen des ADEKF durchgeführt. Der ADEKF produziert exakte Ableitungen und liefert damit äquivalente Ergebnisse zu einem klassischen Extended Kalman Filter. Da die Genauigkeit des EKF gegenüber Weiterentwicklungen bereits in vielen Publikationen untersucht wurde ist es redundant dies in dieser Arbeit auch durchzuführen [12][15][19][33].

Die Laufzeittests in Kapitel 4.4 wurden auf einem Intel Atom E3845 mit 1.91 GHz durchgeführt. Der Code wurde übersetzt mit `clang` auf Optimisierungsstufe 3. Alle anderen auf einem Intel i7-7700HQ Prozessor mit 2.8 GHz und Ubuntu 18.04. Bei diesen Tests wurde `gcc` auf Optimisierungsstufe 3 zum übersetzen verwendet. Für jeden Test auf Basis von Datensätzen wurde der Durchschnitt aus zehn Testläufen als die resultierende Laufzeit verwendet. Der Laufzeittest in Kapitel 4.4 verwendet die Durchschnittszeit eines Zeitschritts während einer Ausführungszeit von ungefähr 10 Sekunden.

4.1 Validierung des Frameworks

Bevor Laufzeittests mit dem ADEKF durchgeführt werden, wird der Algorithmus auf Korrektheit überprüft. Dies geschieht durch Ausführung eines EKF SLAM auf Basis vom ADEKF und einem per Hand abgeleiteten EKF. Da die automatische Differenzierung exakte Ableitungen produziert, muss der ADEKF identische Ergebnisse zu einem per Hand abgeleitetem EKF liefern.

Der EKF SLAM wird ausgeführt auf dem DLR Spatial Cognition Datensatz [7]. Der Datensatz enthält Odometrie Offsets und für jeden Zeitschritt eine Menge von relativen Landmarkenmessungen eines mobilen Roboters, der durch die Räumlichkeiten des DLR fährt. Der Roboter verfolgt dabei die folgende Route:

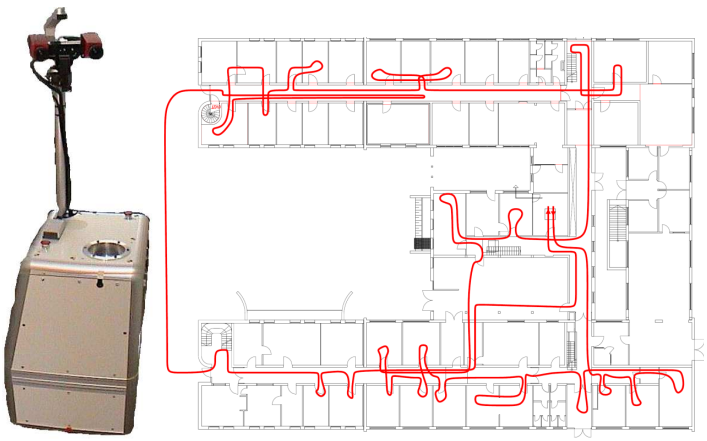

Abbildung 4.1 Links: Der Roboter, mit dem der DLR Datensatz aufgenommen wurde. Rechts: Die Route, die von dem Roboter verfolgt wurde. [7]

Der EKF SLAM schätzt die Pose des Roboters zusammen mit den Positionen von allen Landmarken. Alle Positionen werden dabei in Weltkoordinaten abgespeichert und mit 0 initialisiert.

Jeder Zeitschritt beginnt mit der Addition des Odometrie Offsets auf die geschätzte Position und Orientierung des Roboters. Der Odometrie Offset wird als Zustandsübergangsmessung an das Dynamikmodell übergeben. Während eines Zeitschritts erfasst der Roboter eine Menge von Landmarken. In diesem Fall sind die Landmarken kreisförmige Scheiben, welche auf dem Boden ausgelegt sind (siehe Abb. 4.2). Zu jeder Landmarke sind die Datenkorrespondenzen bekannt. Es kann also die Performanz des EKF SLAM direkt betrachtet werden ohne, dass die Korrespondenzen selbst bestimmt werden müssen.

Abbildung 4.2 Das Blickfeld des Roboters mit den annotierten Landmarken [7]

Falls eine Landmarke zum ersten Mal während des EKF SLAM gesehen wird, wird die Position $l = (l_x, l_y)$ dieser folgendermaßen initialisiert:

$$f_{init}(r, m) = \begin{pmatrix} l_x \\ l_y \end{pmatrix} = \begin{pmatrix} r_x \\ r_y \end{pmatrix} + \begin{pmatrix} \cos(r_\phi) & -\sin(r_\phi) \\ \sin(r_\phi) & \cos(r_\phi) \end{pmatrix} \begin{pmatrix} m_x \\ m_y \end{pmatrix} \tag{4.1}$$

Wobei $r = (r_x, r_y, r_\phi)$ die Pose des Roboters und $m = (m_x, m_y)$ die relative Messung der Position der Landmarke ist. Die Kovarianz R der relativen Position wird vom Datensatz bereitgestellt. Diese muss in die Kovarianz der absoluten Position umgerechnet werden:

$$\begin{aligned} F &= \frac{\partial f_{init}(r, m)}{\partial r} \\ L &= \frac{\partial f_{init}(r, m)}{\partial m} \\ \Sigma_l &= F \Sigma_r F^T + LRL^T \end{aligned} \tag{4.2}$$

Wobei Σ_r die Kovarianzmatrix der Roboterpose und Σ_l die Kovarianz der absoluten Landmarkenposition ist.

Falls eine Landmarke bereits initialisiert wurde, wird die relative Messung der Landmarke an den Messschritt des EKF übergeben. Das Messmodell berechnet dabei die relative Position der Landmarke aus der geschätzten Position des Roboters und der geschätzten Position der Landmarke.

Die Gleichungen des Dynamik- und Messmodells inklusive der Jacobimatrizen können aus dem begleitenden Paper zum DLR Datensatz entnommen werden [7].

In Kapitel 3.3 wurde beschrieben, dass Zustände im ADEKF nur eine maximale Größe von 128 annehmen können. Das bedeutet, dass neben der Position und Orientierung des Roboters nur 62 Landmarkenpositionen geschätzt werden können. Daher wurde sich dazu entschlossen die Evaluation nur auf den ersten 196 Zeitschritten des Datasets durchzuführen, weil zu diesem Zeitpunkt nur 62 Landmarken aufgenommen wurden. Dies beschränkt die Positionen des Roboters ungefähr auf den in Abb. 4.3 eingezeichneten Bereich.

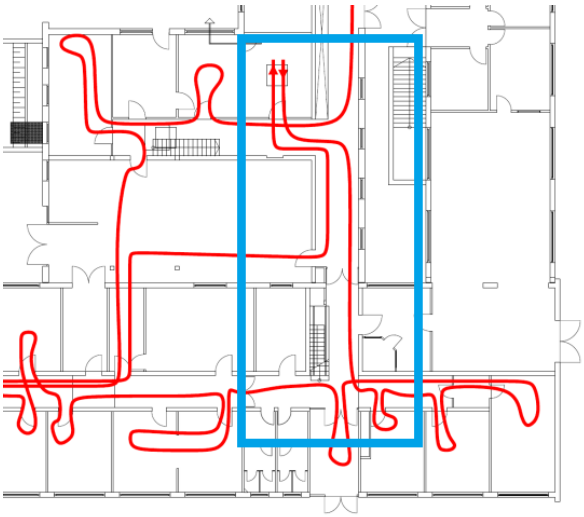

Abbildung 4.3 Im blauen Rahmen: Die ungefähren Positionen des Roboters in den ersten 196 Zeitschritten

Der EKF SLAM wurde mit den beschriebenen Voraussetzungen auf dem ADEKF und einem klassischen EKF ausgeführt. Dabei wurde nach jedem Zeitschritt die geschätzte Position des Roboters und nach Beendigung des Algorithmus die generierte Karte der Landmarken abgespeichert.

Die Ergebnisse zeigen, dass die vom ADEKF und klassischen EKF berechneten Positionen des Roboters und Landmarken bis auf die zwölfte Nachkommastelle übereinstimmen. Diese minimalen Fehler lassen sich erklären durch leichte numerische Abweichungen zwischen den unterschiedlichen Differenzierungsimplementierungen. Da sich aber dieser Fehler durch einen präziseren Zahlentyp verringern lassen würde und nicht abhängig von der Implementierung des ADEKF ist, wird dieser nicht weiter betrachtet. Eine Visualisierung der Ergebnisse ist in

Abb. 4.4 zu sehen.

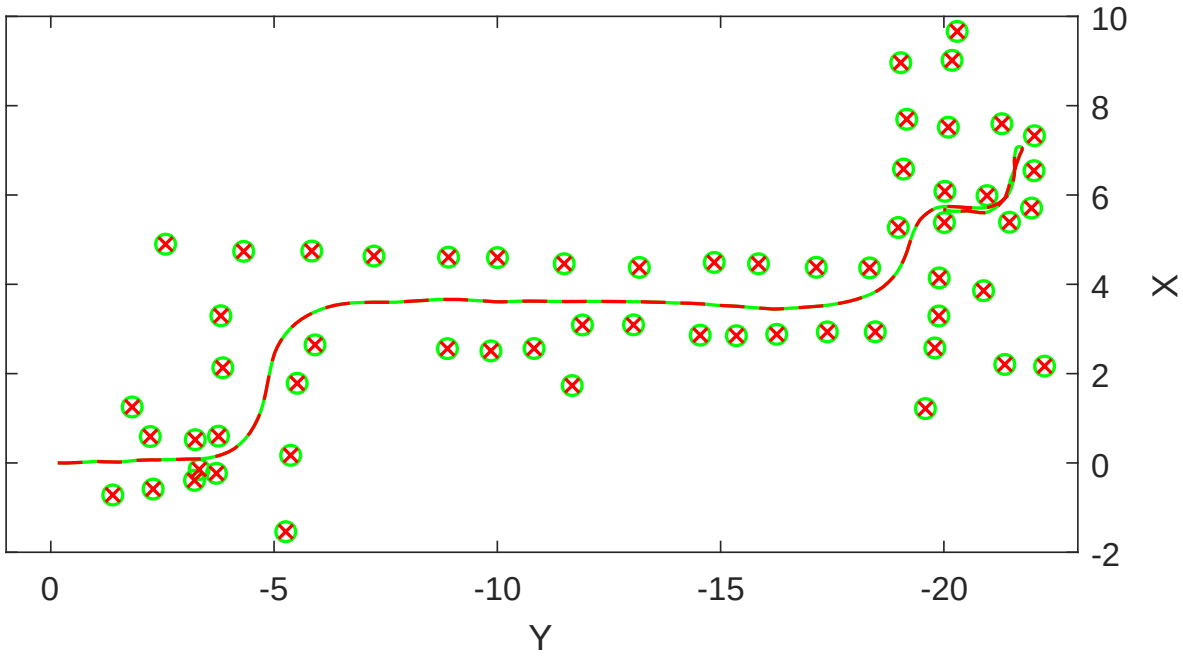

Abbildung 4.4 Die vom Roboter verfolgte Route und die letzte Schätzung der Landmarkenpositionen. In Grün: Die Ergebnisse des ADEKF. In Rot: Die Ergebnisse eines EKF mit per Hand abgeleiteten Jacobimatrizen. Zur besseren Übersicht wurde die Orientierung um 90° gegen den Uhrzeigersinn gedreht.

4.2 Laufzeitvergleich ohne Mannigfaltigkeiten

Um die Laufzeit des ADEKF zu evaluieren, wird dieser in diesem Kapitel mit den Laufzeiten von anderen Kalman Filter Implementierungen verglichen. Alle Implementierungen berechnen den in Kapitel 4.1 beschriebenen EKF SLAM. Zum Vergleich wird die benötigte Gesamtzeit zur Lösung des Algorithmus verwendet.

Zum Vergleich stehen der ADEKF, ein klassischer EKF, ein UKF und ein numerisch abgeleiteter EKF. Die Implementierung des klassischen EKF ist identisch zu der Implementierung des ADEKF ohne die automatische Differenzierung. Der UKF stammt aus dem Manifold Toolkit (MTK) [10]. Die Implementierung des UKF weicht leicht vom klassischen UKF ab, da keine Berechnung von Gewichtungen vorgenommen wird. Es wurde sich trotzdem für den MTK UKF entschieden, da dieser das, für den Laufzeittest in Kapitel 4.3 benötigte, Schätzen von $\boxplus$-Mannigfaltigkeiten unterstützt. Um einen Vergleich zu einem anderen computergestützt abgeleiteten EKF ziehen zu können, wird der Laufzeittest auch auf dem in MATLAB integrierten EKF durchgeführt [28]. Diese EKF Implementierung verwendet numerische Differenzierung (siehe Kapitel 2.1.2), damit der Benutzer keine Jacobimatrizen berechnen muss. Zusätzlich zu der MATLAB internen Implementierung besteht die Möglichkeit den EKF Algorithmus über den in MATLAB integrierten MATLAB Coder [27] als C++-Code zu exportieren. Um zu evaluieren, ob der C++ Export zu einer Geschwindigkeitssteigerung führt,

wurde auch der exportierte MATLAB EKF in die Wertung aufgenommen.

Der Test zeigt, dass der klassische EKF mit 594 ms am schnellsten war. Das entspricht auch den Erwartungen an diesen Test, da die Jacobimatrizen vorher per Hand berechnet wurden. Ebenfalls den Erwartungen entsprochen hat die Laufzeit des UKF, da dieser für große Zustände vergleichsweise langsam ist (siehe Kapitel 2.2.2). Dieser hat mit 2535 ms am längsten für einen Durchlauf des Datensatzes benötigt. Im Vergleich von den computerbasiert differenzierten Extended Kalman Filtern ist der ADEKF mit 693 ms weit schneller als die Implementierungen aus MATLAB mit 1801 und 2273 ms. In Abb. 4.5 sind diese Ergebnisse grafisch dargestellt.

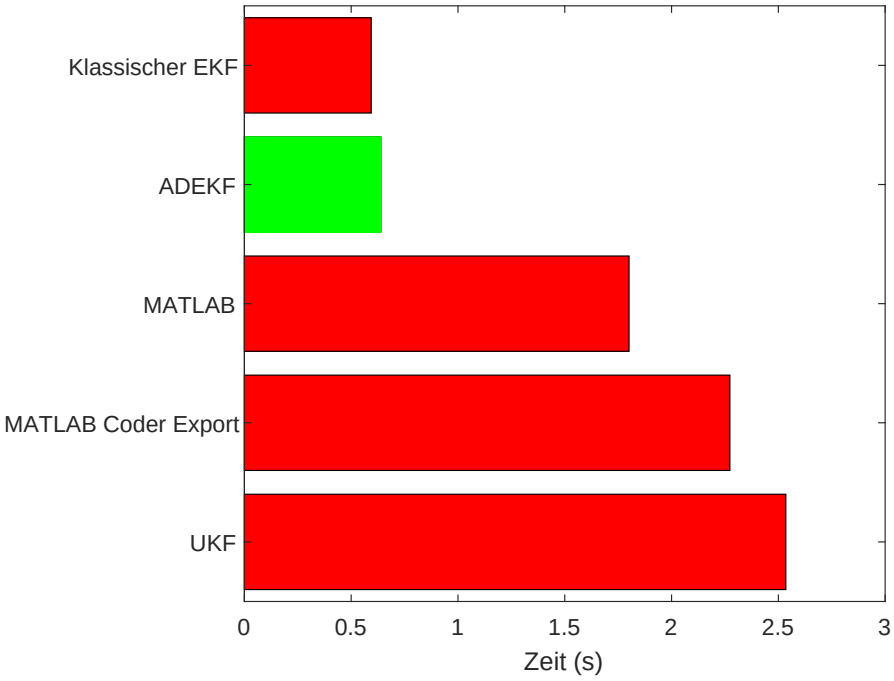

Abbildung 4.5 Laufzeiten eines EKF SLAM auf verschiedenen Kalman Filter Implementierungen

In [16] wurde die Geschwindigkeit eines automatisch differenzierten EKF gegenüber eines klassischen EKF evaluiert. Die Ergebnisse zeigten eine 5-10% höhere Laufzeit des automatisch differenzierten EKF. Dieses Ergebnis entspricht der in dieser Arbeit gemessenen ungefähr 7.5% höheren Laufzeit.

Der ADEKF ist fast drei mal so schnell wie der numerisch differenzierte EKF aus MATLAB. Das ist ein sehr gutes Ergebnis, wenn man bedenkt, dass der ADEKF exakte Ableitungen produziert. Paradoxerweise ist der von MATLAB generierte C++-Code langsamer als die Ausführung des EKF innerhalb von MATLAB.

4.3 Laufzeitvergleich mit Mannigfaltigkeiten

In diesem Kapitel wird evaluiert, wie performant der ADEKF auf Schätzproblemen arbeitet, die Mannigfaltigkeiten enthalten. Da nicht alle Kalman Filter Implementierungen Mannigfaltigkeiten unterstützen kann an dieser Stelle nur der ADEKF und der MTK UKF verglichen werden. Es wurde auch ein MATLAB Interface für das MTK entwickelt [32]. Da dieses aber nicht mit der EKF Implementierung von MATLAB kompatibel ist, sondern nur mit eigens dafür entwickelten Algorithmen, wird dieses nicht mit verglichen.

Der Testdatensatz ist eigens für diese Arbeit aufgenommen worden und enthält 3D-Orientierungen eines 3D-Trackers, Accelerometer- und Gyrometermessungen. Die Accelerometer- und Gyrometerdaten stammen von einer inertialen Messeinheit, welche auf einem Handrücken befestigt ist. Auf der IMU sind Reflektoren eines 3D-Tracking Systems angebracht (siehe Abb. 4.6).

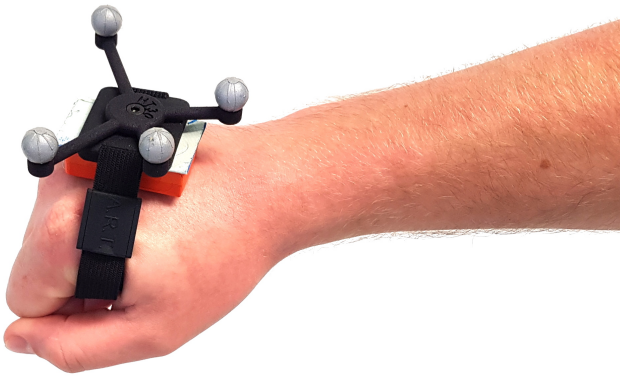

Abbildung 4.6 Die inertiale Messeinheit (orange) und die Reflektoren des 3D-Trackers

Da es möglich sein muss Mannigfaltigkeiten als Zustand aufzunehmen, sowie diese direkt zu messen, soll die Orientierung der Handfläche als Mannigfaltigkeit $SO(3)$ geschätzt werden. Implementiert wurde die Mannigfaltigkeit $SO(3)$ als Quaternion (siehe Kapitel 3.2.2.2). Dabei wird im Dynamikschritt der Zustand über die Gyrometerdaten weitergedreht und alle zwanzig Zeitschritte über eine direkte Zustandsmessung korrigiert. Das Dynamikmodell f und das Messmodell h sind folgendermaßen definiert:

$$f(\mu_{t-1}, u_t) = \mu_{t-1} \boxplus (u_t \cdot \Delta t) \tag{4.3}$$

$$h(\bar{\mu}_t) = \bar{\mu}_t \tag{4.4}$$

Wobei die Zustandsübergangsmessung u_t dem Rotationsvektor der Gyrometerdaten entspricht. Der Faktor Δt ist die Zeit seit dem letzten Zeitschritt.

Es wurde die Durchschnittszeit für einen Durchlauf des Datensatzes aus 10 Durchläufen berechnet. Die Durchschnittszeiten des ADEKF und UKF für einen Durchlauf sind jeweils 9.6 und 17.5 ms (siehe Abb. 4.7).

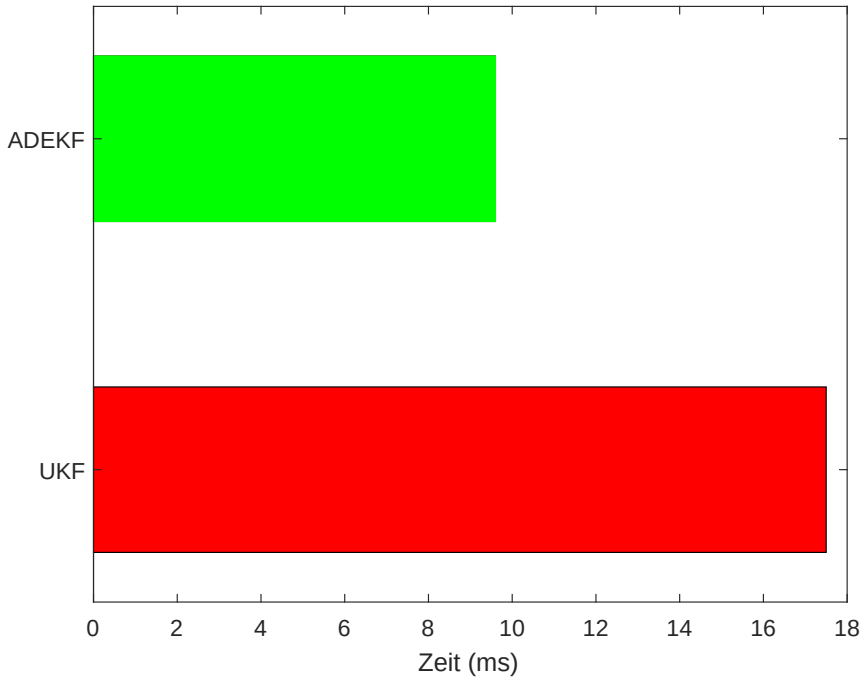

Abbildung 4.7 Laufzeiten einer Orientierungsschätzung auf Mannigfaltigkeiten mit verschiedenen Kalman Filter Implementierungen

Der ADEKF ist mit einen Faktor von 1.8 zwar immer noch schneller als der UKF, aber nicht so dramatisch, wie in Kapitel 4.2 mit einem gemessenen Faktor von 3.6. Dies begründet sich damit, dass der UKF bei niedrigdimensionalen Schätzproblemen weniger Sigmapunkte benötigt. Daher werden die Modelle nicht so oft ausgeführt. Der ADEKF benötigt dagegen für die Berechnung der Jacobimatrizen von Mannigfaltigkeiten mehr Zeit, da jedes Modell zwei mal ausgeführt werden muss (siehe Kapitel 2.3).

Obwohl der ADEKF auf kleinen Mannigfaltigkeiten vergleichsweise langsam arbeitet, ist er ungefähr um einen Faktor 1.8 schneller als der UKF. Und da nach LaViola [15] der EKF und UKF auf Quaternionen ungefähr die selbe Präzision haben, ist der ADEKF in diesem Fall eine gute Alternative zum UKF.

4.4 Einbettung in einem Live Robotiksystem

In diesem Testlauf wird der ADEKF zur Orientierungsschätzung eines NAO Roboters [24] (siehe Abb. 4.8) benutzt. Im Rahmen des Studentenprojekts B-Human [21] werden diese pro-

grammiert, um an internationalen Roboterfußballwettbewerben teilzunehmen. Die aktuelle Orientierungsschätzung basiert auf einem UKF, der sehr ähnlich zu dem bereits getesteten MTK UKF ist und einer im Robotertorso verbauten IMU. Genau wie der MTK UKF unterstützt auch die B-Human Implementierung die Verwendung von Mannigfaltigkeiten. Die Orientierungsschätzung basiert auf einer Mannigfaltigkeit $SO(3)$. Da es sich hier um einen Online Laufzeittest und nicht um die Laufzeit eines Datasets handelt, wurden jeweils nur die für den Dynamik- und Messschritt benötigten Zeiten gemessen. Die Zeiten des Dynamik- und Messschritt wurden addiert und über 1000 Zeitschritte gemittelt.

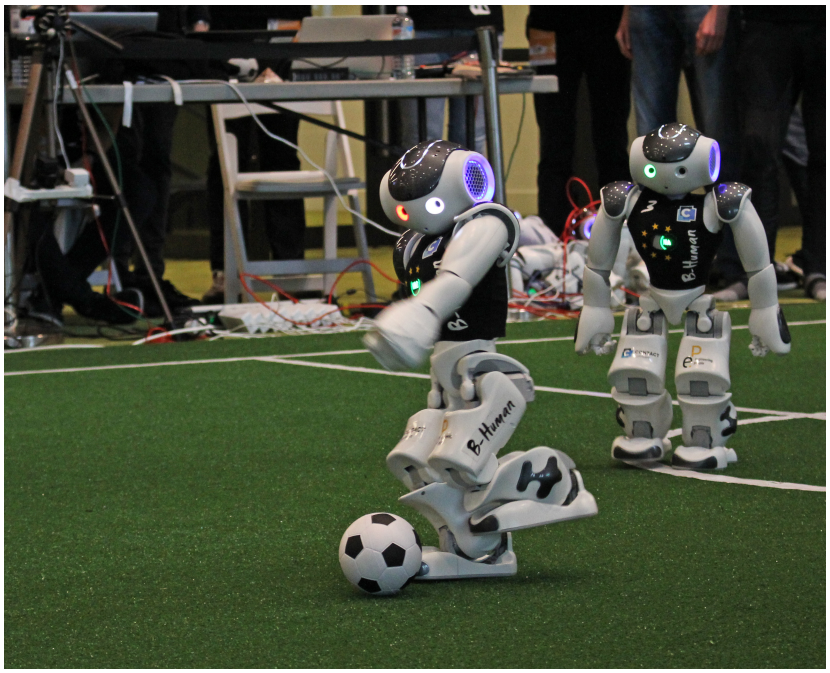

Abbildung 4.8 Ein NAO des Teams B-Human schießt einen Ball.
Foto: Tim Laue, Universität Bremen

Das Dynamikmodell dreht die Schätzung um die, im letzten Zeitschritt, vom Gyrometer gemessenen Winkel und das Messmodell korrigiert die Schätzung mit der Annahme, dass das Accelerometer den um die Schätzung gedrehten Gravitationsvektor misst. Die beiden Gleichungen sind folgendermaßen definiert:

$$f(\mu_{t-1}, u_t) = \mu_{t-1} \boxplus (u_t \cdot \Delta t)$$
$$h(\bar{\mu_t}) = \bar{\mu_t}^{-1} \cdot [0, 0, g]^T$$

Wobei u_t der Rotationsvektor der Gyrometerdaten, Δt die Zeit seit dem letzten Zeitschritt und g der Ortsfaktor ist. Beim Geschätzten Zustand μ handelt es sich um ein Quaternion.

Auf dem Prozessor des NAO Roboters benötigt der ADEKF für einen Zeitschritt 0.0085 ms. Der bisher verwendete UKF benötigt dagegen mit 0.0183 ms mehr als das doppelte an

Laufzeit (siehe Abb. 4.9).

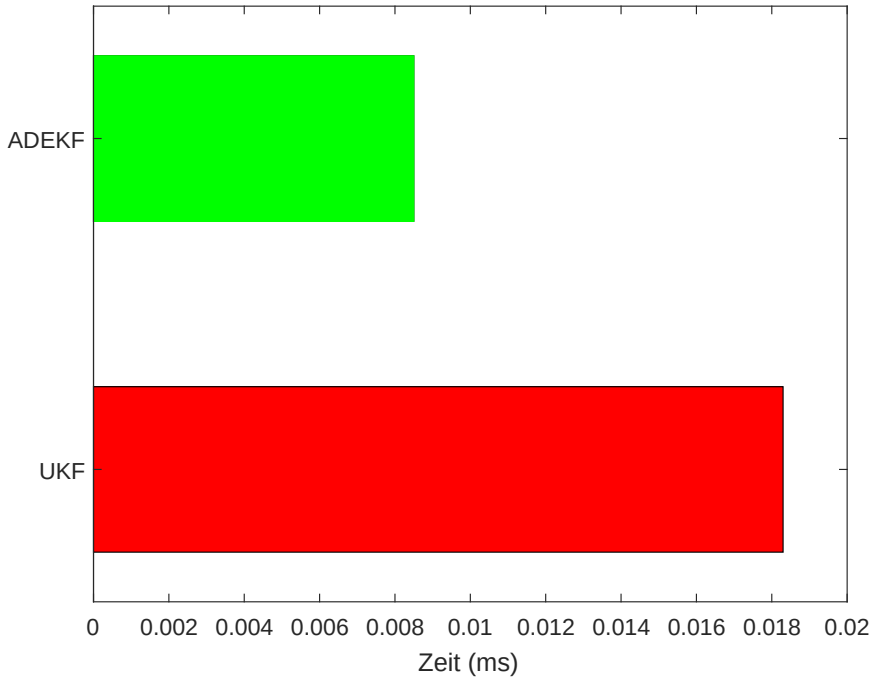

Abbildung 4.9 Laufzeiten einer Orientierungsschätzung auf Mannigfaltigkeiten mit verschiedenen Kalman Filter Implementierungen. Berechnet auf einem NAO Roboter.

4.5 Vergleich der Codekomplexität

In diesem Kapitel wird die Codekomplexität der verschiedenen Frameworks verglichen. Als objektive Kriterien werden die Länge des Codes in Verbindung mit der Anzahl der benötigten Sprachkonstrukte zum Vergleich verwendet. Dabei gliedert sich die Evaluierung in mehrere Teile. Zuerst werden die Dynamik- und Messmodell Definitionen der einzelnen Frameworks verglichen. Danach folgt die Initialisierung des Filters. Und zum Schluss werden die Aufrufe des Dynamik- und Messschritts verglichen. Die zu vergleichenden Frameworks sind der ADEKF, der MTK UKF und der MATLAB EKF. Da der MATLAB EKF die Verwendung von Mannigfaltigkeiten nicht unterstützt, wird dieser Vergleich mit Vektoren als Zustand durchgeführt.

4.5.1 Dynamik- und Messmodell

Der ADEKF benötigt Funktoren mit Funktionstemplates als Klammeroperator für das Dynamik- und Messmodell. Dabei werden für die Syntax des Dynamikmodells 5 Zeilen Code und für das Messmodell 6 Zeilen Code benötigt. Dabei zählt das return-Statement des Messmodells dazu, weil dieses im Dynamikmodell nicht benötigt wird. Bei beiden Modellen werden Verbunddatentypen oder Klassen, Funktionen und Templates verwendet:

```
struct DynamicModel {
  template<typename T>
  void operator()(State<T> &state, const Control &control) {
    ... //Implementation of the Dynamic Model
  }
} dynamicModel;

struct MeasurementModel {
  template<typename T>
  auto operator()(const State<T> &state, const Parameter &parameter) {
    ... //Implementation of the Measurement Model
    return ...; //Return the predicted Measurement
  }
} measurementModel;
```

Die Modelle des MTK UKF werden als einfache Funktionen implementiert. Da bei beiden Modellen Rückgabewerte benötigt werden, werden für die Syntax beider Modelle 3 Zeilen Code benötigt. Das einzige verwendete Sprachkonstrukt sind Funktionen:

```
State dynamicModel(const State &state, const Control &control) {
  ... //Implementation of the Dynamic Model
  return ...; //Return the predicted State
}

Measurement measurementModel(const State &state, const Parameter &parameter) {
  ... //Implementation of the Measurement Model
  return ...; //Return the predicted Measurement
}
```

Ähnlich wie beim MTK UKF werden bei MATLAB die Modelle auch als einfache Funktionen definiert. Dabei müssen die einzelnen Funktionen in eigenen Dateien definiert werden. Auch hier werden für beide Modelle 3 Zeilen für die Syntax benötigt:

```
function result = dynamicModel(state, control)
... %Implementation of the Dynamic Model
result = ...; %Return the predicted State
end

function result = measurementModel(state, parameter)
... %Implementation of the Measurement Model
result = ...; %Return the predicted Measurement
end
```

Die Implementierung der Semantik der Modelle geschieht an den im Code markierten Stellen. Der ADEKF und UKF verwenden dafür von Eigen bereitgestellte Ausdrücke. Für die Implementierung der Modelle in MATLAB können alle in MATLAB integrierten Funktionen verwendet werden.

Im Vergleich zu MATLAB und dem MTK UKF sind die Definitionen der Modelle im ADEKF etwas länger. Der Grund, warum Funktoren mit getemplatetem Klammeroperator in diesem Kontext eingesetzt werden, ist, dass diese vom Ceres Solver für automatische Differenzierung eingesetzt werden. Alternativen wären die in C++20 eingeführten Lambda Templates oder die in C++14 bereits verfügbaren polymorphen Lambdafunktionen. Keiner dieser Ansätze wurde

in dieser Arbeit verwendet, da das Framework von einer möglichst niedrigen C++-Version abhängen sollte. Außerdem erwarten polymorphe Lambdafunktionen Funktionsargumente vom Typ `auto`. Dieses Schlüsselwort kann zu Problemen mit Eigen-Ausdrücken führen, da diese nicht sofort evaluiert werden. Der Benutzer muss in diesem Fall in den Modellen Rücksicht auf den Typ dieser Expression Templates nehmen.

4.5.2 Initialisierung

Beim ADEKF wird für die Initialisierung nur eine Zeile benötigt. Der Konstruktor erwartet Initialisierungswerte für den Zustand und die Kovarianz. Der Template-Typ des EKF wird dabei aus dem Typ des ersten Konstruktorarguments deduziert:

```
ADEKF::EKF ekf(State<double>::Zero(), Cov::Zero());
```

Der UKF wird sehr ähnlich zum ADEKF initialisiert. Der einzige unterschied ist, dass der verwendete Zustand kein Klassentemplate sein muss:

```
ukfom::ukf ukf(State::Zero(), Cov::Zero());
```

Der MATLAB EKF erwartet nicht nur eine Initialisierung des Zustands und der Kovarianz, sondern auch Referenzen auf das Dynamik- und Messmodell. Dabei ist die Zuweisung der Kovarianz etwas komplizierter, da diese über das Schlüsselwort `StateCovariance` eingeleitet werden muss:

```
ekf = extendedKalmanFilter(@dynamicModel,@measurementModel,...
    zeros(...,1),'StateCovariance',zeros(...));
```

Alle Frameworks ließen sich in einer Zeile initialisieren. Auch wird in jedem Beispiel nur ein Konstruktor verwendet. Dabei sind die Aufrufe des ADEKF- und UKF-Konstruktor sehr ähnlich. Der Konstruktor des MATLAB EKF erwartet direkt Referenzen auf das Dynamik- und Messmodell. Das ist etwas unintuitiv, da diese erst bei der Ausführung des Dynamik- und Messschritts verwendet werden. Außerdem muss die Initialisierung der Kovarianz speziell eingeleitet werden. Es wird festgestellt, dass der ADEKF nicht komplizierter zu initialisieren ist als der MTK UKF oder der MATLAB EKF.

4.5.3 Dynamik- und Messschritt

Zur Ausführung eines Dynamikschritts benötigt der ADEKF eine Definition der Kovarianzmatrix und der Zustandsübergangsmessung. Diese können dann zusammen mit dem Dynamikmodell direkt an einen Funktionsaufruf übergeben werden. Es werden mit allen Definitionen 3 Zeilen Code und ein Funktionsaufruf benötigt:

```
Covariance cov = ...; //Definition of the Covariance Matrix

Control u = ...; //Definition of the Control Measurements

ekf.predict(dynamicModel, cov, u);
```

Der Aufruf des Dynamikschritts des MTK UKF ist ähnlich zum ADEKF, abgesehen davon, dass Zustandsübergangsmessungen nicht direkt an den Funktionsaufruf übergeben werden können. Daher müssen die Zustandsübergangsmessungen vor der Zuweisung auf das Dynamikmodell gebunden werden. Die Zeilenanzahl bleibt dadurch gleich, aber es muss neben dem Funktionsaufruf der `predict`-Funktion auch `bind` aufgerufen werden:

```
Covariance cov = ...; //Definition of the Covariance Matrix

Control u = ...; //Definition of the Control Measurements

ukf.predict(std::bind(dynamicModel, _1, u), cov);
```

Beim MATLAB EKF muss vor dem Aufruf der `predict`-Funktion die Kovarianz direkt in das `EKF`-Objekt geschrieben werden. Dies verlängert den Code auf vier Zeilen. Die Zustandsübergangsmessung kann dafür direkt an den Funktionsaufruf übergeben werden:

```
covariance = ...; %Definition of the Covariance Matrix

control = ...; %Definition of the Control Measurements

ekf.ProcessNoise = covariance;
predict(ekf,control);
```

Beim Aufruf des Dynamikschritts vereint der ADEKF die Vorteile des MTK UKF und MATLAB EKF. Die Zustandsübergangsmessungen können dabei direkt an den Funktionsaufruf übergeben werden und es sind keine weiteren Zuweisungen notwendig.

Die Ergebnisse der Komplexitätsuntersuchung des Messschritts sind ähnlich zu den Ergebnissen des Dynamikschritts. Auch beim Messschritt müssen zusätzliche Parameter beim MTK UKF über die `bind`-Funktion auf das Messmodell gebunden werden. Außerdem muss bei der Benutzung des MATLAB EKF das Messrauschen vor der Ausführung des Messschritts in das `EKF`-Objekt geschrieben werden. Der ADEKF empfängt die Kovarianzmatrix des Messrauschens und zusätzliche Parameter direkt im Funktionsaufruf.

4.6 Zusammenfassung

In Kapitel 4.1 wurde gezeigt, dass der ADEKF identische Ergebnisse zu einem klassischen EKF liefert. Die nachfolgenden Laufzeittests in Kapitel 4.2 und Kapitel 4.3 ergaben, dass der ADEKF schneller war als alle anderen getesteten Kalman Filter Implementierungen ohne gegebene Jacobimatrizen. Dazu ist der ADEKF nur ungefähr 7.5% langsamer als ein per Hand abgeleiteter EKF.

Die Codekomplexität des ADEKF ist im Durchschnitt nicht höher als bei den anderen getesteten Frameworks. Beim Aufruf des Dynamik- und Messschritts hat der ADEKF eine niedrigere Komplexität. Im Falle der Definition vom Dynamik- und Messmodell ist die Komplexität in geringem Maße höher als bei den anderen Frameworks. Für diesen Umstand wurden Alternativen vorgestellt, die kürzere Definitionen der Modelle ermöglicht.

Kapitel 5

Fazit

In dieser Arbeit wurde ein Framework entwickelt, das einen Extended Kalman Filter auf Basis von automatischer Differenzierung implementiert. Dies erlaubt es Schätzprobleme mit einem EKF zu lösen, ohne dass vorher die Jacobimatrizen der Modelle berechnet werden müssen.

Die Evaluation hat gezeigt, dass der sogenannte ADEKF identische Ergebnisse zu einem klassischen EKF liefert. Die Geschwindigkeit des ADEKF ist dabei nur ungefähr 7% geringer, als ein per Hand abgeleiteter EKF.

Die Geschwindigkeit des ADEKF wurde weiterhin mit einem Unscented Kalman Filter und dem numerisch differenzierten EKF aus MATLAB vergleichen. Für die Lösung eines SLAM Problems (siehe Kapitel 4.2) war der ADEKF um einen Faktor von 2.5 schneller als die schnellste Implementierung von MATLAB und um einen Faktor von 3.6 schneller als der UKF. In Kapitel 4.3 und 4.4 wurde die Performanz des ADEKF gegenüber dem UKF auf Mannigfaltigkeitsschätzproblemen evaluiert. Bei der Orientierungsschätzung einer Handfläche, mit Mannigfaltigkeiten als Zustand und Messung, konnte eine um einen Faktor 1.8 höhere Geschwindigkeit des ADEKF festgestellt werden. Bei der Orientierungsschätzung eines Robotersystems, mit einer Mannigfaltigkeit im Zustand, war der ADEKF um einen Faktor von 2.1 schneller.

Bei der Evaluation der Codekomplexität in Kapitel 4.5 wurde gezeigt, dass der Implementierungsaufwand des ADEKF ähnlich ist wie bei vergleichbaren Frameworks. Dabei ist die Definition der Modelle zwar etwas komplexer, aber der Aufruf der Dynamik- und Messschritte ist einfacher.

Die automatische Differenzierung auf Basis von dualen Zahlen ist vergleichsweise unbekannt. Diese Arbeit zeigt aber, dass diese sehr nützlich sein können. Besonders die exakte Differenzierung von Mannigfaltigkeiten wurde durch Anwendung von automatischer Differenzierung um ein vielfaches vereinfacht.

Durch den Aufbau des ADEKF als header-only Bibliothek ist es sehr einfach möglich, diesen in bestehende Systeme zu integrieren. Die unterstützte Verwendung von Mannigfaltigkeiten im Zustand und als Messung führt darüber hinaus zu einer deutlich einfacheren Lösbarkeit

von Problemen der Orientierungsschätzung. Allgemein kann der ADEKF, durch seine generische Implementierung, mit allen beschriebenen Vorteilen, überall dort eingesetzt werden, wo bisher andere Kalman Filter Implementierungen verwendet werden. Der ADEKF bietet somit eine Plattform zum lösen von Schätzproblemen jeder Art. Er verbindet die Handhabbarkeit des UKF mit der Geschwindigkeit des EKF in idealer Weise.

Sommer Et al. haben eine erhebliche Beschleunigung von automatischer Differenzierung erreicht, indem für die Mannigfaltigkeitsoperationen spezialisierte Funktionen für duale Zahlen implementiert wurden [25]. Dieser Ansatz ließe sich auch auf den ADEKF übertragen indem die Jacobimatrizen der Funktionen auf den enthaltenen Mannigfaltigkeiten vorher berechnet und optimiert werden. So muss der Benutzer selbst keine Jacobimatrizen implementieren, er profitiert aber von der erhöhten Geschwindigkeit der automatischen Differenzierung.

Die dualen Zahlen wurden 2011 von Fike Et al. erweitert durch die hyper-dualen Zahlen [6]. Diese ermöglichen nicht nur das exakte Berechnen von Ableitungen erster Ordnung, sondern, durch Einführen von weiteren dualen Einheiten, das exakte Berechnen von Ableitungen beliebiger Ordnung. So ließe sich der in Kapitel 2.2.1.2 beschriebene Second Order Extended Kalman Filter oder ein Extended Kalman Filter höherer Ordnung ohne vorheriges Berechnen der partiellen Ableitungen ausführen. Allerdings ist die Laufzeit eines durch hyper-duale Zahlen differenziertem Extended Kalman Filter höherer Ordnung, im Vergleich zum Unscented Kalman Filter noch nicht evaluiert worden.

In Kapitel 3.3 wurden einige Einschränkungen des Frameworks beschrieben. Da die aktuelle Implementierung des ADEKF auf Eigen-Matrizen fester Größe ausgelegt ist, ist die Größe der Zustände Software-seitig beschränkt. Eine Implementierung des ADEKF auf Basis von dynamischen Matrizen würde diese Beschränkung nicht nur aufheben, sondern es auch erlauben, die Zustandsgröße dynamisch zu erweitern. Eine solche Erweiterung wäre sehr attraktiv zum Lösen von Online-SLAM-Problemen, da bei diesen nicht vorhergesehen werden kann, wie groß der Zustand werden muss.

Außerdem müssen Strukturen evaluiert werden, welche das Definieren von Modellen für den ADEKF erlauben. In Kapitel 4.5 wurde gezeigt, dass der ADEKF etwas komplexere Definitionen der Dynamik- und Messmodelle benötigt, als ähnliche Frameworks. Allerdings wurden auch Alternativen vorgeschlagen. Die in C++14 und C++20 eingeführten polymorphen Lambdafunktionen und Lambda Templates ermöglichen eine ähnliche Codekomplexität wie die Funktionsstrukturen der verglichenen Frameworks. Es muss aber evaluiert werden, ob diese Alternativen nahtlos mit Eigen-Ausdrücken arbeiten können, ohne dass der Benutzer Interna des ADEKF in seiner Modelldefinition beachten muss. Wenn dies zutrifft wäre der ADEKF nicht nur schneller sondern auch teilweise einfacher zu benutzen als die verglichenen Frameworks.

Literaturverzeichnis

[1] Baum, Leonard E. ; Petrie, Ted: Statistical Inference for Probabilistic Functions of Finite State Markov Chains. In: *The Annals of Mathematical Statistics* 37 (1966), Nr. 6, S. 1554–1563

[2] *The Boost C++ Libraries*. https://www.boost.org/, . – Nur online: Abgerufen am 01.11.2019

[3] Agarwal, Sameer ; Mierle, Keir ; Others: *Ceres Solver*. `http://ceres-solver.org`, . – Nur online: Abgerufen am 01.11.2019

[4] Clifford, M.A.: Preliminary Sketch of Biquaternions. In: *Proceedings of the London Mathematical Society* s1-4 (1871), Nr. 1, S. 381–395

[5] Guennebaud, Gaël ; Jacob, Benoît u. a.: *Eigen v3*. http://eigen.tuxfamily.org, 2010. – Nur online: Abgerufen am 01.11.2019

[6] Fike, Jeffrey A. ; Alonso, Juan J.: The Development of Hyper-Dual Numbers for Exact Second-Derivative Calculations. In: *AIAA paper 2011-886, 49th AIAA Aerospace Sciences Meeting*, 2011

[7] Frese, Udo: *DLR Spatial Cognition Data Set*. `http://www.informatik.uni-bremen.de/agebv/en/DlrSpatialCognitionDataSet`, 2008. – Nur online: Abgerufen am 01.11.2019

[8] Frese, Udo ; Schröder, Lutz: *Theorie der Sensorfusion*. 2016. – Vorlesungsskript an der Universität Bremen

[9] Gośliński, J. ; Nowicki, M. ; Skrzypczyński, P.: Performance Comparison of EKF-Based Algorithms for Orientation Estimation on Android Platform. In: *IEEE Sensors Journal* 15 (2015), Nr. 7, S. 3781–3792

[10] Hertzberg, Christoph ; Wagner, René ; Frese, Udo ; Schröder, Lutz: Integrating generic sensor fusion algorithms with sound state representations through encapsulation of manifolds. In: *Information Fusion* 14 (2013), Nr. 1, S. 57–77

[11] Hoffmann, Philipp H. W.: A Hitchhiker's Guide to Automatic Differentiation. In: *Numerical Algorithms* 72 (2015), Nr. 3, S. 775–811

[12] Julier, Simon J. ; Uhlmann, Jeffrey K.: *New extension of the Kalman filter to nonlinear systems.* 1997

[13] Julier, Simon J.: The scaled unscented transformation. In: *Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301)* Bd. 6, 2002, S. 4555–4559

[14] Kálmán, Rudolph E.: A New Approach to Linear Filtering and Prediction Problems. In: *Transactions of the ASME–Journal of Basic Engineering* 82 (1960), Nr. Series D, S. 35–45

[15] LaViola, J.J.: A Comparison of Unscented and Extended Kalman Filtering for Estimating Quaternion Motion. In: *Proceedings of the 2003 American Control Conference* Bd. 3, 2003, S. 2435–2440

[16] Leuck, Holger ; Nagel, H.-H: Automatic differentiation facilitates OF-integration into steering-angle-based road vehicle tracking. In: *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* Bd. 2, 1999, S. 360–365

[17] Lillacci, Gabriele ; Khammash, Mustafa: Parameter Estimation and Model Selection in Computational Biology. In: *PLOS Computational Biology* 6 (2010), Nr. 3, S. 1–17

[18] Mcgee, Leonard A. ; Schmidt, Stanley F.: Discovery of the Kalman Filter as a Practical Tool for Aerospace and Industry / National Aeronautics and Space Administration, Ames Research. 1985. – Forschungsbericht

[19] Orderud, Fredrik: Comparison of Kalman Filter Estimation Approaches for State Space Models with Nonlinear Measurements. In: *Proceedings of Scandinavian Conference on Simulation and Modeling - SIMS*, 2005

[20] Rall, L. B.: The Arithmetic of Differentiation. In: *Mathematics Magazine* 59 (1986), Nr. 5, S. 275–282

[21] Röfer, Thomas ; Laue, Tim ; Hasselbring, Arne ; Heyen, Jannik ; Poppinga, Bernd ; Reichenberg, Philip ; Roehrig, Enno ; Thielke, Felix: *B-Human Team Report and Code Release 2018.* 2018. – Only available online: `http://www.b-human.de/downloads/publications/2018/CodeRelease2018.pdf`

[22] Simon, Dan: *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches.* Wiley-Interscience, 2006

[23] Nakath, D. ; Clemens, J. ; Schill, K.: Multi-Sensor Fusion and Active Perception for Autonomous Deep Space Navigation. In: *2018 21st International Conference on Information Fusion (FUSION)*, 2018, S. 2596–2605

[24] Softbank Robotics: *NAO the humanoid robot.* `https://www.softbankrobotics.com/emea/en/nao`, 2019. – Nur online: Abgerufen am 01.11.2019

[25] Sommer, Hannes ; Pradalier, Cédric ; Furgale, Paul: Automatic Differentiation on Differentiable Manifolds as a Tool for Robotics. In: Inaba, Masayuki (Hrsg.) ; Corke, Peter (Hrsg.): *Robotics Research: The 16th International Symposium ISRR*. Cham : Springer, 2016, S. 505–520

[26] Spangenberg, M. ; Calmettes, V. ; Tourneret, J. Y.: Fusion of GPS, INS and odometric data for automotive navigation. In: *2007 15th European Signal Processing Conference*, 2007, S. 886–890

[27] The MathWorks, Inc.: *MATLAB Coder*. `https://de.mathworks.com/products/matlab-coder.html`, 2019. – Nur online: Abgerufen am 01.11.2019

[28] The MathWorks, Inc.: *Create extended Kalman filter object for online state estimation*. `https://de.mathworks.com/help/control/ref/extendedkalmanfilter.html`, 2019. – Nur online: Abgerufen am 01.11.2019

[29] The MathWorks, Inc.: *Symbolic Math Toolbox - MATLAB*. `https://de.mathworks.com/products/symbolic.html`, 2019. – Nur online: Abgerufen am 01.11.2019

[30] Papula, L.: *Mathematik für Ingenieure und Naturwissenschaftler Band 1: Ein Lehr- und Arbeitsbuch für das Grundstudium*. Vieweg+Teubner Verlag, 2011

[31] Thrun, Sebastian ; Burgard, Wolfram ; Fox, Dieter: *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005

[32] Wagner, R. ; Birbach, O. ; Frese, U.: Rapid development of manifold-based graph optimization systems for multi-sensor calibration and SLAM. In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011, S. 3305–3312

[33] Wan, E. A. ; Van Der Merwe, R.: The unscented Kalman filter for nonlinear estimation. In: *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium*, 2000, S. 153–158

[34] Wengert, R. E.: A Simple Automatic Derivative Evaluation Program. In: *Communications of the ACM* 7 (1964), Nr. 8, S. 463–464

[35] Wenk, Felix: *Inertial Motion Capturing : Rigid Body Pose and Posture Estimation with Inertial Sensors*, University of Bremen, Diss., 2017

[36] Wolfram Research, Inc.: *Wolfram Mathematica: Moderne Technische Berechnung*. `https://www.wolfram.com/mathematica/`, 2019. – Nur online: Abgerufen am 01.11.2019