Telematics
Computer Systems

Freie Universität Berlin

Computer Systems and Telematics — Distributed, Embedded Systems

Bachelor Thesis

Design und Implementierung einer Analysesoftware im Kontext eines Referenzsystems zur Indoorlokalisierung

Benjamin Aschenbrenner

Matr. 4292264

Betreuer: Prof. Dr-Ing. Jochen Schiller
Betreuender Assistent: Dipl.Inf. Heiko Will

Institut für Informatik, Freie Universität Berlin, Deutschland

8. Dezember 2011

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, sind als solche gekennzeichnet. Die Zeichnungen oder Abbildungen sind von mir selbst erstellt worden oder mit entsprechenden Quellennachweisen versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner Prüfungsbehörde eingereicht worden.

Berlin, den 8. Dezember 2011

(Benjamin Aschenbrenner)

Zusammenfassung

Zusammenfassung

Diese Arbeit steht im Kontext zum Aufbaus eines Referenzsystems, welches in der Lage ist sich mobil indoor zu bewegen und dabei möglichst genau zu lokalisieren. Der Zweck dieses Referenzsystems ist es, Lokalisierungen als Referenz zur Verfügung zu stellen, um die Genauigkeit von Lokalisierungen mobiler Sensorknoten, die in einem *Wireless Sensor Networks* Wireless Sensor Network (WSN) organisiert sind, zu ermitteln. Das Referenzsystem ist durch einen Roboter realisiert, welcher autonom vorgegebene Wegpunkte in einer Karte abfährt. Bei einer solchen Fahrt zeichnet der Roboter sowie an ihm befestigte Sensorknoten einen Pfad durch regelmäßige Lokalisierung auf. In dieser Arbeit geht es um die Implementierung eines Analysewerkzeugs namens *Pathcompare*, welches ermöglicht, die Pfaddaten zusammenzuführen, für den Tester aufzuwerten und zu visualisieren. Neben mittleren Abstand (Median) zum gewählten Referenzpfad werden Parameter wie Pfadlänge, Anzahl der Pfadpunkte, Stichprobenvarianz und eine Liste der größten Abweichungen angezeigt. Alle Daten können als Comma Separated Values (CSV) exportiert werden. *Pathcompare* ist in das ROS integriert und so entwickelt dass es über Plug-ins erweitert und angepasst werden kann.

Abstract

This thesis is associated with the development of a referece system that has the ability to localize itself. The reason for the development of such a system is to provide localization data. This data is used as reference data for localization data of mobile sensor nodes organized in a *Wireless Sensor Network* WSN inorder to evaluate the precision of their localization measurements. Such a reference system was build in the form of a mobile robot that is able to autonomously navigate to given waypoints. While moving, the robot and sensor nodes mounted on it, generate path data by continously localizing. This work is about the creation of an analysing tool named *Pathcompare* that is used to merge the path data of different sources and visualize them for a tester. The software shows the median distance of a path to the given reference path, the overall pathlength, total number of points per path and also a list of the greates distances to the reference path. All results can be exported as CSV. *Pathcompare* is integrated into the ROS and can be extended via a Plug-in mechanism.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Glossar

Robot Operating System Ein Framework welches zahlreiche Pakete bzgl. Nachrichtenaustausch zwischen verteilten Programmen, Hardwareabstraktion und Robotik bietet. v, 3, 5–11

Akronyme

API Application Programming Interface. 5, 11

CSV Comma Separated Values. v, 12

IMU Inertial Measurement Unit. 2

MEMS Microelectromechanical systems. 2

MOC Meta Object Compiler. 9

UIC User Interface Compiler. 9

WSN Wireless Sensor Network. v

XML-RPC XML - Remote Procedure Call. 6

KAPITEL 1

Einleitung

1.1 Motivation

Systeme zur Positionsbestimmung werden für zahlreiche Zwecke genutzt und deren Bedeutung wächst parallel zur Verbreitung immer neuer sogenannter *location based services* und deren wachsender Nutzung. Für Anwendungen im Freien haben sich Satelliten gestützte Systeme, welche hohe Genauigkeit bieten, etabliert. Als bekanntes Beispiel sei hier das *NAVSTAR-GPS* genannt, welches sich auch im zivil nutzen lässt. Allerdings ergeben sich viele Anwendungsumgebungen, in denen derartige Systeme gar nicht, bzw. nur ungenau funktionieren oder bewusst z.B. aus Kostengründen gemieden werden. Dies sind typischerweise Umgebungen in denen die Satellitensignale zu stark gedämpft werden oder vor allem durch Reflexionen bedingte Laufzeitverschiebungen, sich negativ auf die Genauigkeit auswirken, wie z.B.:

- innerhalb von Gebäuden ("indoor")
- im Untergrund (Tunnel, Höhlen u.ä.)
- im Bereich dicht bebauter urbaner Gebiete (Mehrwegeausbreitung)

Um in solchen Umgebungen dennoch Lokalisierung zu ermöglichen wurden und werden viele theoretische Konzepte und konkrete Systeme entwickelt. Einen Überblick hierzu bietet Quelle anbringen (mobile entity localization and tracking in GPS less enviroments - Buch) Quelle anbringen (mobile entity localization and tracking in GPS less enviroments - Buch) Auch in der Arbeitsgruppe *Computer Systems & Telematics*, an der *FU-Berlin*, wurde dem Problem der indoor Lokalisierung mit der Entwicklung eines *Wireless Sensor Network (WSN)* basiertem Systems im Rahmen des Forschungsprojektes *FeuerWhere*, begegnet. Dieses Projekt entstand u.a. in Kooperation mit der Berliner Feuerwehr. ist das wichtig zu wissen an dieser Stelle? Ziel bei der Entwicklung war ein flexibles indoor Lokalisierungssystem zu schaffen, welches mit low-cost Komponenten bzw. ohne Spezialhardware konstruiert wurde. Im Kern ist das System in der Lage die Entfernung zwischen involvierten Sensorknoten zu bestimmen und dadurch Rückschlüsse auf deren Position zu ermöglichen. In dem WSN unterscheidet man zwei Arten von Knoten, mobile Knoten und Anker Knoten. Diese unterscheiden sich nur dadurch, dass die Position eines Anker Knotens bekannt ist. Bei

einer hinreichenden Zahl von Anker Knoten im WSN kann dann per Trilateration bzw. Multilateration die Position eines mobilen Knotens ermittelt werden. add figure principle of trilateration ? Die Entfernungsmessung zwischen zwei Knoten geschieht hierbei durch Laufzeitmessungen von per Funk gesendeten Round Trip Time (RTT) Paketen, wodurch eine teure sowie aufwendige Zeit-Synchronisierung zwischen den Knoten entfällt, da bei der Messung der RTT nur ein Knoten die Zeit berechnet. Diese Laufzeitmessungen sind jedoch durch in der Hardware auftretenden Jitter und in *non-line of sight (NLOS)* Umgebungen auftretende Mehrwegeausbreitung fehlerbehaftet. Die genaue Funktionsweise und Untersuchung der Auftretenden Fehler ist beschrieben in. hier würde ich natürlich gerne Heiko's paper reffen. Frage: Ist das schon erlaubt? Um diesen Fehler zu untersuchen, ist es sehr nützlich, ein möglichst genaues aber ebenso flexibles Testsystem Referenzsystem? zur Verfügung zu haben, welches mögliche Anpassungen, Konfigurationen und Einsatzszenarien des indoor Lokalisierungssystems, in Hinblick auf dessen Genauigkeit, evaluierbar macht. Der Implementierung und Analyse eines solchen Referenzsystems widmet sich diese Arbeit.

1.2 Aufgabenstellung

Bei der Anwendung kommen

Im folgenden wird zunächst zur Abgrenzung dieser Arbeit innerhalb des Referenzsystems, der grundelegende Aufbau des Referenzsystems beschrieben.

1.2.1 Der Roboter

Auf der Ebene der Hardware steht der mobile Roboter. Dieser hat das Ziel sich genau zu lokalisieren um Referenzwerte für die montierten, mobilen Sensorknoten zu liefern. Da er sich während der Testfahrten indoor bewegt, kann er für seine Lokalisierung keine satellitengestützten Lokalisierungssysteme wie GPS verwenden, aus den in **??** genannten Gründen. Zwei weitere Ansätze, um die Bewegung des Roboters nachzuvollziehen und somit Positionsdaten zu gewinnen sind:

- Inertial Navigation
- Odometrie

Bei der intertial Navigation wird mithilfe von Beschleunigungs- und Gyromessungen auf die ausgeführte Bewegung geschlossen. Diese Messungen lassen sich druch Microelectromechanical systems (MEMS), die in einer Inertial Measurement Unit (IMU) zusammengefasst werden durchführen. MEMS werden in großen Stückzahlen produziert und sind kostengünstig. Typischerweise sind aber die Messungen, auch bei Stillstand, durch Jitter belastet. Dieser kann zwar durch geeignete Filter geglättet werden, lässt sich allerdings nicht ganz ausschließen. Auf längere Strecken entsteht durch die Aufsummierung der Fehler ein Drift, fort von der tatsächlichen Position.

Bei der Odometrie, werden die Antriebsdaten ausgewertet, um auf die Bewegung des Roboters zu schließen. Geht man davon aus, dass der Untergrund, auf dem der Roboter fährt, für dessen Räder geeignet ist und die Räder nicht wegen beispielsweise mangelnder Bodenhaftung stark durchdrehen. So kann sie auf kurzen Strecken sehr genaue Abschätzungen liefern.

Allerdings ist auch eine Odometriemessung stets mit einem Fehler behaftet. Dieser Fehler summiert sich über die Zeit auf und die geschätzte Position weicht immer weiter von der tatsächlichen ab. Man hat also auf längeren Strecken ebenfalls mit einem Drift zu rechnen.

Beide Methoden haben gemeinsam, dass sie unabhängig von Informationen aus der Umgebung des Roboters arbeiten. Somit können sie allerdings den beschriebenen Drift in der Lokalisierung niemals korrigieren, da sie nicht die Positions auf Plausibilität mit der Umgebung abgleichen. Für das Referenzsystem ist Drift aber nicht akzeptabel. Aus diesem Grund erfasst der Roboter Abstände zu Hindernissen seiner Umgebung mithilfe einer *Microsoft Kinect*. Die Kinect erstellt mithilfe eines, im infrarot Bereich gestrahltem, optischen Musters ein Tiefenbild. Die Reichweite liegt dabei bei maximal 10 Meter Entfernung bei einem Blickwinkel von ca 59°. Test haben gezeigt, dass die Genauigkeit der Tiefehmessung mit zunehmender Entfernung abnimmt. Im Nahbereich von zwei Metern aber überraschend Präzise Abstandsauflösung im Zentimeterbreich ermöglicht. Außerdem verfügt der Roboter über eine Karte der Testumgebung. Diese Karte in Kombination mit der *Microsoft Kinect* ermöglicht während einer Testfahrt eine Lokalisierung durch folgende entscheidende Schritte durchzuführen:

1. Abschätzung der derzeitigen Pose durch Odometrie
2. Abgleich mit Karte und Korrektur der Pose

Zum Abtasten der Umgebung hätte alternativ auch ein Laserscanner gewählt werden können, welcher hohe Reichweite mit hoher Genauigkeit kombiniert. Dies wäre aber entgegen den Ziele des Referenzsystems, mit zu hohen Anschaffungskosten verbunden gewesen. Im Sinne günstiger Kosten wurde schlussendlich ein sogenannter *TurtleBot* gebaut. Dies ist ein von *WillowGarage* spezifizierter low-cost Roboter. Im Kern besteht dieser aus einem *Roomba* Staubsaugerroboter von *iRobot*, einer *Microsoft Kinect* und einem Tragegerüst. Das Tragegerüst dient als Abstellfläche für einen Laptop und bietet im Anwendungsfall des Referenzsystems Platz zum Montieren der Sensorknoten.

Die Aspekte der Software, zum Betrieb des Roboters, wurde innerhalb einer anderen Bachelorarbeit, ebenfalls im Rahmen der Entwicklung des Referenzsystems, ausführlich erarbeitet.

Zusammenfassend kann man in dieser Hinsicht festellen, dass zum autonomen Fahren Programme des ROS genutzt werden sowie eine Software implementiert wurde, die den Roboter vorgegebene Wegpunkte abfahren lässt und dabei gesammelte Lokalisierungsdaten innerhalb von ROS bereitstellt. Genauer wird im Teil **??** auf die Funktionsweise von ROS eingegangen.

KAPITEL 2

Pathcompare - Implementierung

2.1 Technischer Rahmen

2.1.1 Robot Operating System - ROS

Innerhalb des Referenzsystems wird ROS im Zusammenhang mit der Steuerung des Roboters genutzt und um die, während einer Testfahrt gewonnenen Pfaddaten, zur Verfügung zu stellen. Im Folgenden wird genauer auf die Fähigkeiten und Ziele von ROS eingegangen.

Obwohl der Name zunächst anderes vermuten lässt ist ROS kein Betriebssystem im klassischem Sinne. Es ist ein Framework, welches auf ein Betriebssystem angewiesen ist um ausgeführt werden zu können. Es bietet aber Funktionalitäten, die abstrahiert betrachtet Betriebsystemfunktionen ähneln. Charakteristisch ist hierbei ROS Fähigkeit lokal- oder nichtlokalausgeführte Programme zu Verbinden und eine strukturierte Kommunikation zwischen diesen zu ermöglichen. Die wesentlichen Elementareigenschaften der Grundphilosophie sind:

- multi-tool Ansatz
- peer-to-peer Kommunikation
- keine feste Bindung an Programmiersprache
- frei und Open-Source

Multi-tool Ansatz bedeutet, dass ROS die Fähigkeiten verschiedener Programme und Libraries zur Verfügung stellt. Diese sind jedoch nicht fest in den Kern von ROS eingebaut sondern modular integriert. Als analoges Beispiel in Hinblick auf Betriebssysteme kann man ROS also als einen Mikrokernel verstehen. Dies bietet den Vorteil, dass ROS selbst vergleichsweise klein ist und nur wirklich gebrauchte tools geladen werden müssen. Die peer-to-peer Kommunikation bezieht sich auf die Kommunikation zwischen diesen, in ROS integrierten, Modulen. Diese wird durch den ROS Kern gesteuert. Der Kern von ROS ist nativ in C++ implementiert, es existieren jedoch bereits Portierungen in andere Sprachen wie Python, Octave und Lisp um die ROS-Application Programming Interface (API) einer größeren Zahl von Entwicklern und Projekten die Nutzung zu ermöglichen. Weitere Portierungen sollen sich in der Implementierung befinden.

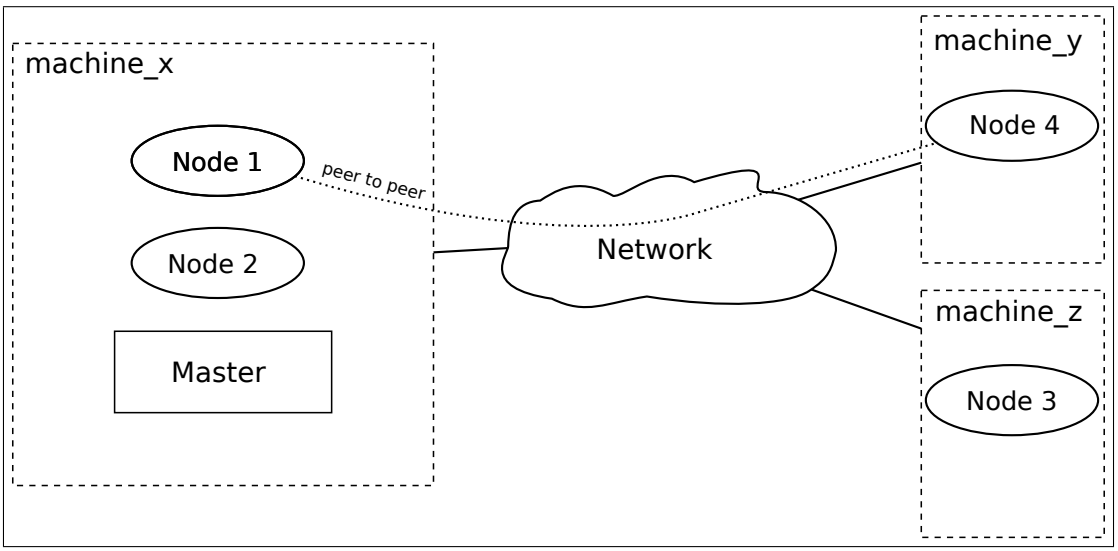

Abbildung 2.1: Beispielhafte Ausführung von ROS auf unterschiedlichen Maschinen

ROS ist darüberhinaus frei Verfügbar und Open-Source. Man kann beliebige Programme, als Module zur Erweiterung und Nutzung von ROS hinzufügen, wie es auch im Rahmen des Referenzsystems geschehen ist. Bei allgemeinem Nutzem und gegebener Pflege, besteht die Möglichkeit, dass diese offiziell zu ROS hinzugefügt werden.

Um die konkreten Abläufe und Komponenten innerhalb von ROS veranschaulichen zu können und damit auch den Bezug zu Pathcompare herstellen zu können, ist es zunächst erforderlich die Begrifflichkeiten innerhalb von ROS zu klären. Im Folgendem werden diese aufgezeigt.

Im Zentrum von ROS steht der sogenannte *master*. Dieser wird als einzelne Instanz gestartet und wartet dann darauf, dass sich tools, die im Kontext von ROS gestartet werden, bei ihm anmelden. Ein gestartetes tool wird innerhalb von ROS als *node* bezeichnet. Ist der *master* nicht gestartet können auch keine *nodes* gestartet werden. Die *nodes* sind also alle zunächst auf Kommunikation mit dem *master* angewiesen. Diese Kommunikation kann lokal oder nichtlokal ausgeführt werden, d.h. der *master* kann sich auch auf einem anderen Rechner wie der *node* befinden solange eine http Verbindung zwischen beiden hergestellt werden kann. Denn das Anmelden des *nodes* beim master erfolgt über einen XML - Remote Procedure Call (XML-RPC), getragen vom http. Für den tool Entwickler auf Anwendungsebene ist diese Kommunikation zur Anmeldung allerding völlig unsichtbar und er muss sich nicht darum kümmern. Das Ausführen von *nodes* auf unterschiedlichen Rechnern ist natürlich ebenso möglich und wie zuvor bereits erwähnt eines der Kernfunktionen von ROS. Dies lässt sich vorteilhaft ausnutzen durch zum Beispiel:

- Verteilung oder Auslagerung rechenintensiver *nodes* auf potente Hardware
- Zusammenführenung von an unterschiedlichen Stellen gewonnener Daten.

So muss beispielsweise ein mobiler Roboter Bilderkennungs Aufgaben nicht selbst ausführen, sondern kann diese an einen *node* weiterleiten der auf einem Rechencluster ausgeführt wird.

Die zweite genannte Möglichkeit ist auch besonders im Bezug auf diese Arbeit wichtig, da Pfaddaten des Roboters und der zu testenden Sensorknoten für *Pathcompare* verfügbar gemacht werden müssen. Die Kommunikation zwischen Nodes erfolgt über sogenannte *messages*, diese enthalten die serialisierte Form der zu übertragenden Daten. ROS bietet in seinen Kernpakten bereits zahlreiche Definitionen für unterschiedliche *message* Typen, aber es ist auch möglich eigene zu generieren und dies wird von zahlreichen Paketen getan um Daten maßgeschneidert übertragen zu können. Einmal definierte *message* Typen können wiederum rekusiv in anderen *message* Typen verwendet werden. Ein Beispiel für eine message ist in Quellcode 2.1 dargestellt; eine Transformation die in ROS geometry_msgs definiert ist. Sie besteht wie erkennbar aus den zwei Typen *Vector3* und *Quaternion*. Letzterer beschreibt die Rotation und der erste die Translation. Zusammengefügt ergibt dies eine Transformationsnachricht.

```
1 geometry_msgs/Vector3 translation
2   float64 x
3   float64 y
4   float64 z
5 geometry_msgs/Quaternion rotation
6   float64 x
7   float64 y
8   float64 z
9   float64 w
```

Quellcode 2.1: ROS transformation message

Soll ein *node* seine *messages* anderen *nodes* senden können, so muss dies zunächst durch festlegen einer sogenannten *topic* beim *master* angemeldet werden. Dann wird diese *topic* für andere *nodes* im ROS über den *master* sichtbar. Eine *topic* besteht im wesentlichen aus zwei Teilen, einer *topic-id* und einem *message-type*. Wobei der *message-type* angibt welcher Typ von *messages* über diese *topic* verschickt wird. Die *topic-id* dient zur eindeutigen Identifikation innerhalb des ROS. *Nodes* welche *messages* einer *topic* empfangen sollen, müssen diese *topic* dann beim *master* abbonieren. In programmatischer Hinsicht wird beim Empfang neuer Nachrichten innerhalb des *nodes* eine festgelegte callback Methode aufgerufen um die *messages* zu bearbeiten. Treffen dabei Nachrichten mit einer höheren Frequenz ein, als abgearbeitet werden können, so kommt es irgendwann zu Verlusten wenn die einstellbare Größe der *message queue* überschritten wird.

Zwei weitere wichtige Begriffe in ROS betreffen die Organisierung der Dateien die zu den einzelnen tools gehören. Dies sind:

- package
- stack

Ein *package* beinhaltet den Code, Libraries sowie die ausführbare Datei eines tools bzw. *nodes*. In ROS sind für packages bestimmte Ordnerstrukturen und Dateien festgelegt sodass mithilfe der von ROS mitgebrachten tools *packages* leicht gebaut, gesucht und gestartet werden können. Beispielsweise basiert das ROS build System auf cmake und so ist eine vorkonfigurierte *CMakeLists.txt* in jedem *package* grundsätzlich enthalten. Eine Zusammenfassung mehrerer *packages* wird als *stack* bezeichnet.

Pathcompare ist also im Bezug auf ROS, während der Ausführung, ein einzelner *node* welcher *topics* abboniert um *messages* zu empfangen. Es wird in einem späteren Teil darauf eingegangen, welche *messages* das genau sind. Alle zum Kompilieren und Ausführen nötige Dateien sind dabei in zwei *packages* namens *pathcompare* und *pathcompareplugins* aufgeteilt.

- Warum brauchen wir ROS?
- Wer entwickelt ROS? (Stanford University & WillowGarage & community)
- Was sind die Grundeigenschaften
- multitool Ansatz
- strukturierte Kommunikation zwischen den Tools
- free and open source
- multilingual
- Begrifflichkeiten des ROS (Stack, Package, Node, Master, Topic, Message)
- Topologie (figure: Roboter, Master, Pathcompare , Sensorknoten)
- plattformen: Ubuntu
- standard Buildsystem cmake

2.1.2 Qt

Pathcompare ist darauf ausgelegt alle Informationen für den Nutzer in einer Benutzeroberfläche (fortan als GUI bezeichnet) zu visualisieren. Da die Anbindung an ROS über C++ erfolgt, lag nahe auch die GUI in C++ umzusetzen. Dazu wurde das Qt Framework gewählt. Qt ist in C++ implementiert, wobei allerdings auch Anbindungen für zahlreiche andere Sprachen wie z.B. Java, C#, Ruby oder Python existieren. Die Entwicklung von Qt begann 1991 und ist zum Zeitpunkt des Schreibens in der Version 4.7.4 verfügbar. Das Framework besteht dabei mittlerweile nicht mehr nur aus reinen GUI Bibliotheken, sondern stellt auch z.B. Netzwerk-, SQL- und andere Anwendungs-Bibliothken zur Verfügung.

Für die Entwicklung von *Pathcompare* waren neben den GUI Bibliotheken, besonders folgende Konzepte und Funktionen beim Entwickeln vorteilhaft:

- Signal-Slot Konzept
- Plattformunabhängigkeit
- Containerklassen mit praktischen Hilfsmethoden
- graphischer GUI Designer

Auf einige dieser Punkte und deren Bezug zu Pathcompare wird nun kurz eingangen.

Das *Signal-Slot* Konzept dient dazu Objekten, bestimmte Veränderungen an Objekten mitzuteilen, welche über diese Änderung informiert werden sollen. Es realisiert das Entwurfsmuster des *Observer patterns*. *Signal-Slot* erspart es dem Programmierer einen Verweis auf das beobachtende Objekt, durch einen registrierenden Methodenaufruf beim aktualisierenden Objekt, zu hinterlegen. Stattdessen emittiert, im Falle einer mitzuteilenden Aktualisierung, das aktualisierende Object einen Methode, welche durch einen Qt Makro als *signal* ausgezeichnet ist. Bei Beobachter Objekten, die auf dieses Signal reagieren sollen, wird dann eine als *slot*

deklarierte Methode selbständig aufgerufen, sofern die Objekte, durch einen Aufruf einer von Qt bereitgestellten, statischen Methode, verbunden wurden. Dieses Konzept vereinfacht die Entwicklung wegen seiner Flexibilität deutlich, allerdings ist der durch Qt erzeugte Code, welcher bei Auflösung der Makros entsteht, aufblähend und dadurch theoretisch langsamer bei der Ausführung als eine klassische callback Implementierung. In der Praxis ist diese Verlangsamung aber unmerklich und wiegt nicht die Vorteile für den Entwickler auf. Deshalb wurde das Konzept auch bei der Entwicklung von *Pathcompare* eingesetzt.

Im Zusammenhang mit dem Signal-Slot Konzepts wurde bereits erwähnt, dass Qt C++ mit verschiedenen Makros erweitert. Diese Makros werden dabei nicht immer direkt in gültigen C++ Code übersetzt, sondern dienen als Annotationen. Dies hat Folgen für den Build Vorgang, denn die mit Annotationen versehenen Klassen müssen dann zuerst, mit dem von Qt bereitgestelltem Meta Object Compiler (MOC) übersetzt werden. Standardmäßig wird in *Qt* das Build Programm qmake verwendet, welches die nötigen Aufrufe des MOC automatisch veranlasst. In Hinblick auf *Pathcompare* war aber eine Integration in das von ROS genutzte Build verfahren cmake notwendig. Hierbei muss beachtet werden, dass cmake die Dateien welche einen MOC Aufruf notwendig machen, in der derzeitigen Version, nicht selbständig erkennt. Diese müssen manuell in der *CmakeLists.txt* deklariert werden. Die sonstige Einbindung *Qt* spezifischer Libraries und deren Verlinkung mit der Applikation ist in cmake leicht möglich. Somit war die Verwenung von Qt in *pathcompare* kein Hindernis für die Verwendung von cmake. Die Nutzung von cmake wird darüberhinaus, für große Projekte, von der Qt Dokumentation selbst empfohlen.

Neben dem MOC existiert noch ein sogenannter User Interface Compiler (UIC). Dieser tritt im Zusammenhang mit dem *QtDesigner* in Erscheinung. Der *QtDesigner* ermöglicht es graphisch Qt GUIs zu erstellen. Dabei wird eine XML Datei vom Designer erstellt, welche den Aufbau der GUI abbildet. Der UIC ist dann dafür verantwortlich diese Datei in C++ Klassen zu übersetzen. Auch hier muss cmake veranlasst werden zunächst für einen UIC Aufruf entsprechender Dateien zu sorgen.

- Warum braucht Pathcompare Qt?
- GUI framework, native in C++ (auch Implementierungen für Java, Ruby,..)
- bietet auch zahlreiche nicht GUI spezifische Funktionalität
- cross platform
- native UI rendering
- UI designing with Qt Designer
- bereichert C++ durch embedded Macros (z.B. signal&slot) -> moc compiler
- standard Buildsystem native qmake
- Unterstützung für cmake gegeben, für große Projekte sogar empfohlen

2.2 Design der Software

Der Hauptfokus beim Design der GUI lag darauf, einfache Benutzung und übersichtliche Darstellung, für den Nutzer zu gewährleisten. Außerdem sollte die Software leicht erweiterbar sein.

2.2.1 Überblick Gesamtsystem

Im Folgendem wird die Software in einer Gesamtansicht dargestellt und anschließend auf die Funktionsweise ihrer Einzelkomponenten genauer eingegangen.

Wie in vorherigen Teilen bereits erwähnt ist *Pathcompare* durch Plug-ins erweiterbar. Dadurch kann man zunächst das Gesamtsystem logisch in zwei Teile untergliedern, nämlich in

1. Rahmen
2. Plug-ins

Diese Aufteilung spiegelt sich am offensichtlichsten auch in der GUI wider. Hierbei bildet das UI des Rahmens die Grundlage für die Visualisierung von Plug-ins. Dabei wird den Plug-ins, nachdem sie geladen wurden, ein Platz innerhalb des Rahmens zugewiesen. Innerhalb dieses zugewiesenen Platzes können sie beliebige eigene UI Elemente laden und haben die volle Kontrolle über diese. Konkret gehören dabei Folgende UI Elemente zum Rahmen:

- Hauptfenster
- Topic Übersicht
- Plug-in Tab-Fenster

Das Hauptfenster dient dabei als Grundlage für die Topic Übersicht und das Plug-in Tab-Fenster. Die Topic Übersicht und das Plug-in Tab-Fenster sind vertikal getrennt. Die Topic-Ansicht ist dabei ganz links angeordnet, siehe . Rechts neben der Topic Übersicht schließt unmittelbar das Plug-in Tab-Fenster an. Wird ein Plug-in geladen, wird in diesem Tab-Fenster ein neuer Tab angelegt. Die Fläche dieses Tabs wird diesem Plug-in dann zur Verfügung gestellt um sein UI darauf aufzubauen. Auf Details bezüglich des Ladens der Plug-ins und deren Funktionsweise wird in einem folgendem Abschnitt eingegangen.

Die Topic Übersicht zeigt die momentan im ROS verfügbaren Topics an. Um die Übersicht auf die Topics zu strukturieren ist sie derart gestaltet, dass Topics desselben *message* Typs gruppiert werden. Diese Gruppen werden dann kompakt im Topic Übersicht Fenster angezeigt und können dort durch den Nutzer aufgeklappt werden um alle *topics* eines Typs anzuzeigen. Dies ermöglicht es, nur solche Topics zu beobachten, die auch tatsächlich relevant sind für den Nutzer.

Die Platzaufteilung zwischen der Topic Übersicht und des Plug-in Tab-Fensters, innerhalb des Hauptfensters, wurde zugunsten des Plug-in Tab-Fensters gewählt, sodass dieses mehr Platz einnimmt. Dies hat auch zur Folge, dass beim Vergrößern des Hauptfensters, die Fläche des Plus-in Tab-Fensters vertikal und horizontal vergrößert wird. Die eingenommene Fläche der Topic Übersicht wächst allerdings nur vertikal wesentlich an. Diese Designentscheidung begründet sich darin, dass die Plug-ins den wesentlichen Inhalt für den Nutzer präsentieren. Die Topic Übersicht wird hingegen vermutlich nur kurzzeitig betrachtet werden und liegt nicht im Zentrum des Interesses. Darüber hinaus ist die Topic Übersicht sehr kompakt gestaltet und der horizontale Platzbedarf fällt gering aus.

2.2.2 Anbindung an ROS

Die Hauptaufgabe des Rahmens ist es eine Anbindung an ROS zu gewährleisten. Diese Anbindung muss es den Plug-ins ermöglichen, benötigte *topics* zu abbonieren. Für diese Anbindung ist in *Pathcompare* die Klasse *ROSManager* verantwortlich. Im Konstruktor dieser Klasse wird durch entsprechende ROS API Aufrufe der *node pathcompare* erstellt. Dies geschieht durch starten eines separaten Threads, da der ROS Code in einer eigenen while-Schleife ausgeführt werden muss, die über die gesamte Lebenszeit des *nodes* bearbeitet wird. Innerhalb dieser Schleife wird beispielsweise, das eintreffen neuer *messages* von abonnierten *topics* abgearbeitet. Zugriff der Plug-ins auf ROS spezifische Funktionalität wird komplett über den *ROSManager* gekapselt. Diese Zentralisierung ist deshalb nötig, umd den Zugriff der verschiedenen Plug-ins, auf ROS, zu koordinieren. Die wichtigste Methode die durch den ROSManager dabei den Plug-ins zur Verfügung steht, ermöglicht es eine als Parameter übergebene *topic* zu abbonieren. Diese Methode ist wie folgt definiert:

Beim Abonnieren einer *topic* wird in ROS typischerweise direkt eine Methode als Callback angegeben, welche eingehende *messages* bearbeitet. Bei einer einfachen Abbonierung ist es dabei zunächst nicht möglich mehrere Callbacks zu registrieren. In *Pathcompare* bestehtehen in dieser Hinsicht allerdings zwei Schwierigkeitn denen bei der Entwicklung begnet werden musste:

1. eventuell abbonieren mehrere Plug-ins eine topic
2. topics können von beliebigem *message* Typ sein

Gelöst wurde das Problem durch die Verwendung von *ros::message_filter::cache*. Objekten dieser Klasse wird bei der Initialisierung eine *topic* zugewiesen. Empfangene *messages* dieser *topic* werden in einem Ringpuffer mit einstellbarer größe Zwischengespeichert. Außerdem erlaubt diese Klasse die Registrierung beliebig vieler Callbacks. Der *ROSManager* sorgt also dafür, dass für jede von den Plug-ins benötigte *topic* genau ein *ros::message_filter::cache* Objekt existiert. Plug-ins erhalten dann einen Verweis auf dieses Objekt und können ihren Callback registrieren.

Weitere Methoden die der ROSMaster den Plug-ins zur Verfügung stellt, werden in einem späteren Abschnitt aufgezeigt.

Die Klasse *ROSManager* sorgt außerdem dafür, dass die Topic Übersicht regelmäßig aufgefrischt wird. Es wird also dem Nutzer ersichtlich, sollten neue *topics* erscheinen oder wenn diese nicht mehr verfügbar sind. Da die ROS API keinen event service anbietet um Beobachter zu benachrichtigen, falls sich der Status von *topics* ändert, stellt der *ROSManager* sekündlich eine Anfrage an den *master* nach dem aktuellen Stand der topics. Dies ist gesteuert über einen QTimer in Verbindung mit dem Signal-Slot Konzept.

2.2.3 Laden von Plugins

Das Einbinden von Plugins in den Rahmen steuert die Klasse *PluginLoader*. Sie sucht dabei in einem Verzeichnis nach als Plug-in infrage kommenden Dateien. Bei dieser Suche werden generell nur shared Library Dateien betrachtet. Handelt es sich um eine für *Pathcompare* gültige Plug-in Datei, fragt der *PluginLoader* zunächst den Namen des Plug-ins über eine im Plug-in Interface spezifizierte Methode *getName()* ab. Anschließend wird im Plug-in Tab

Fenster ein neuer Tab angelegt. Dabei wird im Tab Reiter der erfragte Name des Plug-ins eingetragen. Dadurch sind in der GUI die Plug-ins für den Nutzer leicht zu unterscheiden und können schnell angewählt werden. Der Plug-in Ordner steht während der Laufzeit von *Pathcompare* unter der Beobachtung des *PluginLoaders*. Obwohl *Pathcompare* bereits gestartet wurde, können also Plug-ins in das Plug-in Verzeichnis eingefügt werden und diese werden geladen. Dies bietet mehr Flexibilität für den Nutzer und Entwickler, da kein Neustart der Rahmen Anwendung sowie der ausgeführten *Plug-ins* erforderlich ist.

2.2.4 Main Compare Plug-in

Main Compare implementiert die eigentliche Hauptfunktionalität um Pfaddaten des Referenzsystems und der Sensorknoten zu vergleichen und auszuwerten. Es ist dabei der Philosophie von *Pathcompare* folgend auch als Plug-in implementiert worden, welches beim Starten des Rahmens durch den *PluginLoader* geladen und ausgeführt wird.

In der GUI werden dem Nutzer verschiedene Informationen sowie Einstellungsmöglichkeiten bezüglich der empfangenen Pfaddaten angezeigt. Die Funktionen der einzelnen GUI Elemente werden im Folgenden aufgezeigt. Insgesamt gibt es drei Bereiche in der GUI, welche sich anhand dreier verschiedener Labels abgrenzen lassen. Von oben nach untern gesehen sind dies die Bereiche:

1. Reference path selection
2. Export results
3. Results

Der Bereich der “Reference path selection” beinhaltet eine Combo-Box, welche alle über ROS verfügbaren *topics* des Typs nav_msgs/Path beinhaltet. Der Nutzer kann dann aus dieser Liste einen Pfad auswählen, der als Referenzpfad genutzt werden soll. Das bedeutet, dass der ausgewählte Pfad als Referenz für die übrigen Pfade gilt. So werden dann die Abstandsberechnungen bezüglich dieses Pfades durchgeführt. Die Auswahl des Referenzpfades kann jederzeit geändert werden, und die Abstände werden stets neu berechnet.

Unterhalb des “reference path selection” Bereichs befindet sich der “Export results” Bereich. Er dient dem Nutzer dazu, die berechneten und im “Results” Bereich visualisierten Ergebnisse, permanent zu speichern. Die Speicherung erfolgt dabei als CSV Datei. Der Export wird durchgeführt wenn der Nutzer den in diesem Bereich vorhandenen Button drückt. Das genaue Format der gespeicherten Daten wird später erläutert.

Der “Result” Bereich dient dazu alle ermittelten Informationen bezüglich der Pfade, für den Nutzer anzuzeigen. Die Visualisierung erfolgt hierbei durch eine Tabbellenstruktur. In dieser Tabelle wird für jede Path topic ein Spalte angelegt. In jeder Zeile einer Spalte sind dabei die unterschiedlichen Informationen eingetragen. Main Compare verhält sich außerdem so, dass topics, welche einmal in der Tabelle erfasst wurden auch dort verbleiben. Sollte also eine topic nicht mehr im ROS zur verfügung stehen, bleiben die angezeigten Informationen auf Basis der zuletzt empfangenen Werte dennoch erhalten.

In der Tabelle werden für den Nutzer folgende Informationen: angezeigt:

- Anzahl der Lokalisierungen

- Berechnung der Pfadlänge
- Anzeige des Medians der Abstände
- Anzeige des arithmetischen Mittels der Abstände
- Berechnung der empirische Varianz s^2 der Abstände
- Berechnung der empirische Standardabweichung s der Abstände
- 20 größten Abstände zu Referenzpfad

Die angezeigten Informationen werden bei Empfang neuer Pfad *messages* stets aktualisiert. Generell gilt dabei, dass jede Path message den kompletten Pfad enthalten muss. Pfade können also nicht stückweise übertragen werden. Dieses vorgehen bietet die Möglichkeit, dass Pfaddatenen nachträglich geändert werden können. Also kann beispielsweise die erste Lokalisierung innerhalb eines Pfades zu einem späteren Zeitpunkt abgeändert werden.

Um nachzuvollziehen wie die in Main Compare angezeigten Informationen ermittelt werden, bietet es sich an, zunächst den Typ der *messages*, welche über die Pfad topics empfangen werden, zu analysieren. Denn auf dieser Basis leiten sich alle ermittelten Informationen ab. Wie bereits beschrieben sind die messages vom Typ *nav_msgs/Path*, der wie folgt definiert ist:

```
1  Header header
2    uint32 seq
3    time stamp
4    string frame_id
5  geometry_msgs/PoseStamped[] poses
6    Header header
7      uint32 seq
8      time stamp
9      string frame_id
10   geometry_msgs/Pose pose
11     geometry_msgs/Point position
12       float64 x
13       float64 y
14       float64 z
15     geometry_msgs/Quaternion orientation
16       float64 x
17       float64 y
18       float64 z
19       float64 w
```

Quellcode 2.2: ROS transformation message

Zunächst ist in Quellcode 2.2 zu erkennen, dass sich dieser *message* Typ aus anderen Typen zusammensetzt. Auf der obersten Ebene ist dies ein *Header* und ein Array von *geometry_msgs/PoseStamped*. Der Header dient dazu, die Path message von anderen empfangenen Path messages zu unterscheiden und dazu die messages ordnen zu können. Die Ordnung kann dabei über eine Sequenznummer (seq) oder einen Zeitstempel (stamp) hergestellt werden. Das Feld frame_id definiert einen Bezugsrahmen innerhalb von ROS, es wird aber im Kon-

text von Main Compare nicht ausgewertet und wird hier deswegen nicht genauer erläutert. Die eigentlichen Pfaddaten verbergen sich in dem Array von PoseStamped. Auch PoseStamped ist ein zusammengesetzer Typ, bestehend aus einem Header und einer Pose. Die Pose setzt sich aus einem Point und einer Quaternion zusammen. Der gefahrende Pfad wird nun dadurch in der Path message abgebildet, indem bei jeder durchgeführten Lokalisierung eine PoseStamped angelegt wird. Diese PoseStamped hält im Header die Zeit der Lokalisierung und eine Sequenznummer fest. In der Pose wird dann der bei der Lokalisierung ermittelte Punkt, als Point eingetragen. Die Quaternion hält fest, welche Ausrichtung dabei im Raum vorliegt und wird auch in die Pose eingefügt. Die Ausrichtung ist allerdings optional und wird in der derzeitigen Version von Main Compare nicht beachtet. Außerdem lässt sich in Quellcode 2.2 erkennen, dass die Lokalisierungen dreidimensional erfolgen können. Die von Main Compare ausgeführten Berechnungen sind aber auch mit Daten niederer Dimension möglich, indem die nicht genutze Dimension auf 0 gesetzt wird. Beispielsweise werden, in der derzeitigen Form des Referenzsystems, vom Roboter zweidimensionale Lokalisierungsdaten in der x-y Ebene übermittelt und die z Komponente erhält immer den Wert 0. Wie schon in Bezug auf die orientation erwähnt, werden nicht alle Komponenten der Pfad message in Main Compare verwendet. Für die derzeitige Version sind folgende Felder relevant:

- poses.stamp (Typ ros::time)
- pose.position.x (Typ float64)
- pose.position.y (Typ float64)
- pose.position.z (Typ float64)

Obwohl nicht alle Felder der Path message genutzt werden, ist dieser message Typ für Main Compare die geeignete Wahl, da er in ROS standardmäßig definiert ist, was die Definition und deren Verteilung, eines eigenen message Typs, vermeidbar macht. Außerdem sind die ungenutzen Felder nicht übermäßig groß und werden die Kapazitäten des Übertragungskanals kaum belasten. Vernachlässigt man die ungenutzten Felder des Header, der eine in ROS definierte Standardmessage ist, fallen nur zusätzliche 24 Byte für die ungenutze orientation, pro übertragenem Point, an. Darüberhinaus ist es denkbar, dass die orientation in späteren Versionen von Main Compare doch noch betrachtet wird und somit die durchzuführenden Änderungen klein ausfallen.

Anhand der oben aufgezählten, genutzen Felder, lässt sich ein Pfad abstrakt durch eine Menge M von Tupeln modellieren. Dabei setzen sich die Tupel aus einem Zeitstempel und einem dreidimensionalem Punkt zusammen. In Anlehnung an die verwendeten Bezeichner in Quellcode 2.2, sei M demnach wie folgt definiert:

$$M := \{(p.stamp, p.position) \mid p \in poses\}$$

Die konkrete Repräsentierung dieses Modells erfolgt in Main Compare durch die Klassen *TopicPath* und *Position*. TopicPath repräsentiert dabei die Menge M. Position repräsentiert ein einzelnes Tupel und enthält somit den Zeitstempel und zugehörigen Punkt. Dem Modell entsprechend enthält ein TopicPath Objekt eine Liste von Position Objekten. Neben den Klassen TopicPath und Position gibt es noch eine Klasse *TopicPathManager*. Es wird genau ein Objekt dieser Klasse für jede im ROS verfügbare Path topic angelegt. Diese Klasse bietet eine callback Methode, welche beim, zur topic gehörenden, message_filter::cache reg-

istriert wird. Kommt eine neue message an, wird die callback Methode aufgerufen und aus den Werten der Path message ein neues TopicPath Objekt erstellt. Anschließend werden, anhand des neuen TopicPaths, alle nötigen Berechnungen erneut durchgeführt und die Results Tabelle aktualisiert.

Anhand der abstrakten Modellierung M eines Pfades, werden im Folgenden, die von Main Compare ausgeführten Berechnungen erläutert. In der konkreten Repräsentatio erfolgen alle Berechnungen in der Klasse *TopicPathManager*.

Anzahl der Lokalisierungen

Um die Anzahl der Lokalisierungen zu ermitteln wird in Main Compare lediglich die Mächtigkeit der Menge M bestimmt. Es gilt also:

$$numberofpoints = |M|$$

Im *TopicPathManager* ist dies entsprechend einfach realisiert.

Berechnung der Pfadlänge

Bei der Bestimmung der Pfadlänge werden zunächst alle Abstände, zwischen je zwei direkt aufeinanderfolgenden Punkten im Pfad, bestimmt. Die Summe all dieser Abstände entspricht dann der Gesamtlänge des Pfades. Dazu muss jedoch noch geklärt werden, wann ein Punkt p_1 in einer Path message auf einen anderen Punkt p_2 folgt. Das lässt sich über den Zeitstempel feststellen und ist anschaulich ausgedrückt dann der Fall, wenn keine weitere Lokalisierung in der Zeit zwischen der Lokalisierung p_1 und p_2 stattgefunden hat. Dies kann man auch wie folgend, abstrakt und bezogen auf M, ausdrücken. Ein Punkt p_2 ist direkt folgend auf einen Punkt p_1, genau dann wenn für die zugehörigen Tupel
$t_1 := (z_1, p_1)$ und $t_2 := (z_2, p_2)$ mit $t_1, t_2 \in M$ gilt, dass:

$$z_2 > z_1 \wedge \nexists(z', p') \in M : z_2 > z' > z_1$$

Der Abstand zweier aufeinanderfolgender Punkte lässt sich einfach durch Vektorsubtraktion und anschließende Betragsbildung des resultierenden Vektors bestimmen.

Pfadvergleichsverfahren und Abstandsberechnung

Das Hauptziele von Main Compare ist es, dem Nutzer zu erleichtern, die Genauigkeit eines durch Lokalisierung gewonnenen Pfades, in Bezug auf einen Referenzpfad, abzuschätzen. Um dies zu tun, ist es notwendig, den Abstand von Lokalisierungen des zu untersuchenden Pfads, zu denen des Referenzpfads, zu ermitteln. In einem naiven Ansatz, um diesen Abstand zu bestimmen, könnte man die Lokalisierungen direkt miteinander vergleichen. Unter der Voraussetzung, dass für jeden Punkt des Referenzpfades auch ein Punkt im zu vergleichenden Pfad mit demselben Zeitstempel existiert. Der Vergleich zwischen Referenz und zu untersuchendem Pfad fällt dann denkbar einfach aus, denn man müsste nur den Abstand je zweier Punkte, mit demselben Zeitstempel, bestimmen. Das Problem bei dieser Methode ist

jedoch, dass das Referenzsystem und die einzelnen Sensorknoten unabhängig voneinander sind und dadurch Lokalisierungen nicht immer zum selben Zeitpunkt oder mit einer anderen Frequenz durchführen können. Um dies zu verhindern müsste man wie in Abb. gezeigt einen zentralen Taktgeber in das System integrieren, der durch ausgesendete Impulse, zum Beispiel in der Form einer ROS message, den Komponenten vorschreibt, wann eine Lokalisierung durchgeführt werden soll. Die Einführung eines solchen Taktgebers verkompliziert jedoch die Testausführung und setzt bei allen am Test beteiligten Komponenten voraus, dass deren Software auf diesen Mechanismus reagiert. Zudem ist man während des Tests zwingend auf eine ungestörte Netzwerkkommunikation angewiesen, da sonst der Impuls nicht gleichmäßig oder gar nicht bei den Komponenten ankommt.

Aufgrund dieser schwerwiegenden Nachteile wurde bei der Entwicklung von Main Compare ein Ansatz gewählt, der es den Komponenten erlaubt Lokalisierungen zu beliebigen Zeitpunkten und mit beliebiger Frequenz durchzuführen. Die einzige Voraussetzung ist dabei, dass die Uhren der Komponenten, mit einem gewissen tollerierbaren Fehler, synchronisiert werden. Dies sichert, dass die Zeitstempel der Lokalisierungen zuverlässig vergleichbar sind. Der Abstand wird wie folgt ermittelt, siehe dazu auch Abb. . Für jeden Punkt im zu vergleichenden Pfad A wird der Abstand zum Refernzpfad R bestimmt. Dazu wird der jeweilige Zeitstempel z_a eines Punktes p_a aus A einem passenden Zeitstempelintervall zweier aufeinanderfolgender Punkte p_{r2} , p_{r1} aus R zugeordnet, sodass gilt:

$$z_{r1} <= z_a < z_{r2}$$

Ist dieses Intervall gefunden, wird der Abstand von p_a zum Geradensegment $\overline{p_{r2}p_{r1}}$ bestimmt. Um diesen Abstand zu bestimmen wird das Lot auf die von p_{r1} und p_{r2} bestimmte Gerade ermittelt. Fällt das Lot dabei zwischen p_{r1} und p_{r2} auf die Gerade wird der Abstand vom Lotpunkt zu p_a errechnet. Fällt er jedoch außerhalb dieses Intervalls muss der Abstand, je nach Lage des Lotpunkts, entweder zu p_{r1} oder p_{r2} bestimmt werden. Siehe dazu auch Abb. .

Charakteristisch bei dieser Art der Abstandsbestimmung ist, dass beim Referenzpfad zwischen zwei Lokalisierungen, durch das Geradensegment, die Position interpoliert wird. Es wird also angenommen, dass die tatsächlich ausgeführte Bewegung zwischen den Punkten, einer Gerade entspricht. Daraus folgt aber, dass man die Lokalisierungsfrequenz des Referenzpfades, an die Bewegungsgeschwindigkeit des Referenzsystems anpassen muss. Dabei soll zwischen zwei Lokalisierungen des Referenzpfades die interpolierte Strecke klein bleiben, denn dadurch nähert sie sich stärker der tatsächlich gefahrenen Strecke an. Betrachtet man das derzeitige Referenzsystem, mit dem TurtleBot, so ist die Bewegungsgeschwindigkeit allerdings gering und eine sekündliche Lokalisierung erscheint ausreichend. Dies muss jedoch noch durch weitergehende Tests bestätigt werden.

Bestimmung empirischer Varianz und Standardabweichung

Auf Basis der zuvor beschriebenen Abstandsberechnung wird in Main Compare die empirische Varianz der Abstände bestimmt. Um diese zu berechnen wird zunächst das arithmetische Mittel gebildet, welches ebenso in der Results Tabelle angezeigt wird. Anschließend wird die empirische Varianz s^2 gebildet. Die empirische Standardabweichung ist die Wurzel aus s^s also s.

Bestimmung des Median der Abstände

Der Medians der Abstände, ist robuster gegenüber Ausreißern als das arithmetische Mittel und wird deshalb ermittelt und auch in der Results Tabelle aufgeführt.

2.2.5 Das Plug-in Konzept

Im folgenden Abschnitt wird das Konzept des Pathcompare Plug-in Mechanismus erläutert und aufgezeigt, welche Schritte erforderlich sind, um eigene Plug-ins für Pathcompare zu erstellen. Realisiert sind die Plug-ins in Pathcompare mithilfe der Qt Plugin API. Diese gibt Strukturen vor, um Plug-ins zu definieren. Außerdem beinhaltet sie Klassen, um Plug-ins während der Laufzeit einer Applikation zu laden. Bei Verwendung dieser API müssen, um Plug-ins zu erstellen, im wesentlichen zwei Schritte ausgeführt werde, dies sind:

1. Definition eines Interface
2. Implementierung des Interfaces durch Plug-in

Um eine Anwendung durch Plug-ins erweiterbar zu machen ist also zunächst erforderlich ein Interface zu definieren. Dieses Interface legt fest, welche Methoden durch ein Plug-in zu implementieren sind. Das Interface hat dabei die Form einer Klassendefinition mit nur rein virtuellen Funktionen. Zusätzlich wird diese Klassendefinition durch Qt Makros ergänzt, um es als Interface zu markieren. In Pathcompare ist dieses Interface vorgegeben und trägt den langen Namen *ComparatorPluginFactoryInterface*. Wie die Implementierung dieses Interfaces erfolgt, wird nun anhand eines vorhandenen Plug-ins für Pathcompare erläutert. Dieses Plug-in trägt den Namen Camera View. In der derzeitigen Version von Pathcompare wird es, ebenso wie Main Compare, beim Starten der Anwendung geladen. Es dient dem Nutzer dazu ROS topics des Typs sensor_msgs/Image zu visualisieren. Dieser Typ von Topic wird beispielsweise innerhalb von ROS genutzt um Kamerafeeds der Kinect zur Verfügung zu stellen. Camera View könnte also bei einer Fahrt des TurtleBots eingesetzt werden um dem Tester ein Bild der aktuellen Umgebung des Roboters zu vermitteln. Die Gestaltung der GUI dieses Plug-ins fällt einfach aus. Der Nutzer kann mit einer ComboBox eine topic auswählen. Nach der Auswahl, werden die empfangenden Bilder, der gewählten Image topic, eingeblendet.

Wie zuvor beschrieben muss jedes Pathcompare Plugin das Interface *ComparatorPluginFactoryInterface* implementieren, so auch Camera View.

Dieses Interface beinhaltet zwei Methoden, siehe dazu auch Quellcode 2.3.

```
1
2         virtual ComparatorPluginPtr createComperatorPlugin(ROSManager *
              ros_manager, QWidget *tab_widget) const = 0;
3         virtual QString getPluginName() const = 0;
```

Quellcode 2.3: ROS transformation message

Die Methode createComparatorPlugin() ist eine Factorymethode und erzeugt das eigenliche Objekt, welches die Funktionalität des Plug-ins beinhaltet. Dieser Umweg ist notwendig, da Plug-ins keine typischen Klassenobjekte sind. Denn bei Qt Plug-ins handelt es sich um shared libraries. In einer shared library können zwar Methoden ausführbar hinterlegt werden,

aber kein eigenständiges Objekte mit Attributen existieren. Der Rückgabetyp von createComparatorPlugin() ist ComparatorPluginPtr, definiert durch Quellcode 2.4.

```
1 //typedef for shared_ptr reference to ComparatorPlugin
2 typedef boost::shared_ptr<ComparatorPlugin> ComparatorPluginPtr;
```

Quellcode 2.4: ROS transformation message

Es handelt sich also um einen Verweis auf ein Objekt der Klasse ComparatorPlugin. Diese Klasse ist der Basistyp, für das in der Factory Methode dynamisch erzeugte Objekt. In Camera View ist die Factory Methode wie folgt implementiert worden, siehe Quellcode 2.5:

```
1 ComparatorPluginPtr PathcompareMainFactoryPlugin::createComparatorPlugin(
      ROSManager * ros_manager, QWidget *tab_widget) const
2 {
3         ComparatorPluginPtr comp_plugin(static_cast<ComparatorPlugin *>(
              new CameraView(ros_manager, tab_widget)));
4         return comp_plugin;
5 }
```

Quellcode 2.5: ROS transformation message

Es wird hierin ein Objekt des Typs CameraView angelegt, welches den Basistyp ComparatorPlugin besitzt. Im Konstruktor erhält es Verweise auf den ROSManager, welcher benötigt wird um ROS spezifische Aktionen durchzuführen. Als zweites Konstruktorargument wird die Zeichenfläche für die Camera View GUI übergeben. Am Ende der Methode wird das fertig angelegte Objekt zurückgegeben und steht damit Pathcompare zur Verfügung.

Die zweite Methode im Plug-in Interface, getName() liefert lediglich den Namen des Plug-ins zurück und wird in Pathcompare durch die Klasse PluginLoader genutzt.

Zusammenfassend kann man also sagen, dass die eigenlichen Pathcompare Plug-in Interface Methoden ohne großen Aufwandt zu implementieren sind. Ein vollständiges Referenzbeispiel zur Implementierung neuer Plug-ins ist durch Camera View gegeben. Zu finden sind dabei alle relevanten Code- und Builddateien im Ordner pathcompareplugins/cameraview.

2.3 Anwendung