-Der erste Schritt besteht in der Implementierung einer fähigen Infrastruktur, um sowohl die effiziente Implementierung als auch Erprobung von Lösungsansätzen zu ermöglichen. Nach dem Vorbild von [1] wird eine Kombination aus ROS und Gazebo verwendet, wodurch ein Algorithmus zunächst in der Simulationsumgebung Gazebo getestet und im unmittelbaren Anschluss auf den realen Roboter übertragen werden kann. Bei Gazebo handelt es sich um eine Simulationsumgebung, die einerseits komplexe Dynamik in einer dreidimensionalen Welt zu simulieren vermag, andererseits über ein Plugin mit ROS gekoppelt werden kann, wodurch eine einzelne Implementierung sowohl für die Simulation als auch den realen Roboter verwendet werden kann.

Um den Turtlebot in ROS zu modellieren, wird das sogenannte Universal Robotic Descripction Format (URDF) verwendet [1], welches einer XML-ähnlichen Struktur folgt. In [1]wird die URDF-Datei um zusätzliche Tags erweitert, welche die Kompatibilität zu dem in Gazebo verwendeten Simulation Description Format (SDF) schaffen. Um die Simulationsergebnisse zu visualsieren verwenden Takaya et al. Das RViz Packet.

Die ROS Architektur wird in drei Ebenen: Filesystem, Computation Graph und Community, unterteilt.

Die untereste Ebene, das Dateisystem, setzt sich aus den elementaren Bausteinen von ROS zusammen, wobei als elementarer Baustein so gennante Packages verwendet werden. Dabei handelt es sich Ordner, welche die minimalen Anforderungen an ein ausführbares Programm unter ROS enthalten. Die Dokumentation eines Package erfolgt in Form eines Manifest, welches relevante Information wie Abhängikeiten und Compiler-Einstllungen enthält. Um umfangreiche Funktionalitäten zu erreichen, werden mehrere Packages zu seinem so genannten Stack zusammengefasst. Als Beispiel dient ein Navigationsstack, der sich aus Funktionen für die Auswertung von Sensorik, SLAM-Verfahren, Pfadplannung und Ansteuerung der Aktorik zusammensetzt, welche jeweils als eigenes Package implementiert werden. Die Dokumentation des Stacks erfolgt ebenfalls in Form eines Manifest, welches in diesem Fall als Stack-Manifest bezeichnet wird.

Am Beispiel von Stacks wird bereits deutlich, dass ein Grundprinzip von ROS darin besteht, dass mehrere Prozesse interagieren. Um Daten zwischen Prozessen auszutauschen werdne Nachrichten verwendet, welche im Rahmen von ROS als Messages bezeichnet werden.

Zusätzlich gibt es noch Services, die ähnlich wie Nachrichten allerdings nur kurz beschrieben werden und deshalb auch an dieser Stelle nicht weiter ausgearbeitet werden.

In der zweiten Ebene wird ROS als Netzwerk beschrieben, in welchem die verschiedenen Prozesse verknüpft sind. Die einzelnen Elemente des Netzwerks werden als Nodes bezeichnet, wobei es sich um Prozesse handelt, die über das ROS Netzwerk miteinander verbunden sind. Ein besonderer Bestandteil des Netzwerks stellt der so genannte Master dar, der eine Namen Registrierung und Lookup für die restlichen Nodes bereitstellt, wodurch dieser notwendig für die Kommunikation zwischen den Nodes wird. Eine weitere zentrale Einheit bildet der Paramter Server, welcher die Möglichkeit bietet, Daten in einer zentralen Position mithilfe von Keys zu speichern. Die Kommunikation zwischen den Nodes erfolgt über Nachrichten, welche in Form von Message Types auf der Ebene des Dateisystems festgelegt wurden. Bei der Nachrichtenkommunikation wird ein Publish/Subscribe Schema eingehalten, welches darauf basiert, dass alle Nachrichten mit einem Namen, der als Topic bezeichnet wird, versendet werden. Nodes können Topics subscriben, woraufhin alle Nachrichten dieser Topic an sie weitergeleitet werden. Da es sich bei der Nachrichtenkommunikatoin mithilfe von Topics um eine many-to-many Verbindung handelt, können Topics nicht genutzt werden, um eine direkte Kommunikation zwischen zwei Nodes umzusetzen, wofür die so genannten Services verwendet werden. Wenn eine Node einen Service anbietet, muss dieser einen einzigartigen Namen besitzen, woraufhin andere Nodes den Service über das ROS-Netzwerk ansprechen und nutzen können. Als letzte Einheit der Computation-Graph-Ebene sind die so genannten Bags zu nennen, welche als Datenspeicher fungieren. Beispielsweise können Nachrichten in Bags gespeichert werden, um den gesamten Werteverlauf eines Sensors im Anschluss an ein Experiment zu rekonstruieren.

Erstellen von eigenen Packages erfolgt mithilfe des Befehls

Roscrate-pkg [package\_name] [depend1] [depend2] [depende3]

Beispiel: roscreate-pkg chapter2\_tutorials std\_msgs rospy roscpp

Erstellt das Package chapter2\_tutorials, das von den Packeten std\_msgs, rospy und roscpp abhängt. Wichtig dabei ist, dass das neue Packet bzw. der Ordner, der das Packet darstellt, auf dem ROS\_PACKAGE\_PATH liegt.

Anleitung nach ROS-Wiki für Turtlebot Simulation in Gazebo/Visualisierung mit Rviz [3]:

Intsallation:

sudo apt-get install ros-kinetic-turtlebot ros-kinetic-turtlebot-apps ros-kinetic-turtlebot-interactions ros-kinetic-turtlebot-simulator ros-kinetic-kobuki-ftdi ros-kinetic-rocon-remocon ros-kinetic-rocon-qt-library ros-kinetic-ar-track-alvar-msgs

Fertiges Beispiel, um den Turtlebotsimulation in Gazebo zu starten:

roslaunch turtlebot\_gazebo turtlebot\_world.launch

Durch das festlegen der Umgebungsvariable TURTLEBOT\_GAZEBO\_WORLD\_FILE kann eine .world-Datei bestimmt warden, die als Umgebung für den Trutlebot in Gazebo genutzt wird.

Um die Geschwindigkeit de sRoboters zu setzen, muss die Topic /mobile\_base/commands/velocity gepublished werden, wofür die Nachricht geometry\_msgs/Twist mit dem Format:

rostopic pub /mobile\_base/commands/velocity geometry\_msgs/Twist "linear:

x: 0.1

y: 0.0

z: 0.0

angular:

x: 0.0

y: 0.0

z: 0.0"

ausgegeben werden.

Alternative Installation [https://bitbucket.org/damienjadeduff/456\_kinetic\_turtlebot/src/b8eacebf174bdf2e11cb288dc61cac4bd163fb01/install\_456\_students.sh?at=master&fileviewer=file-view-default]

sudo apt-get -y install ros-kinetic-turtlebot-gazebo ros-kinetic-desktop-full ros-kinetic-turtlebot-rviz-launchers ros-kinetic-warehouse-ros || { echo "Could not install main turtlebot packages."; exit 1;}

sudo apt-get -y install ros-kinetic-gazebo-ros ros-kinetic-gazebo-plugins ros-kinetic-gazebo-ros-control python-rosdep python-wstool ros-kinetic-ros ros-kinetic-warehouse-ros libspnav-dev || { echo "Could not install some extra ROS packages."; exit 1;}

Welt setzen:

TURTLEBOT\_GAZEBO\_WORLD\_FILE=/opt/ros/kinetic/share/turtlebot\_gazebo/worlds/corridor.world

export TURTLEBOT\_GAZEBO\_WORLD\_FILE

roslaunch turtlebot\_gazebo turtlebot\_world.launch

Über Rviz kann man zusätzlich visualisieren:

roslaunch turtlebot\_rviz\_launchers view\_robot.launch

Als großen Vorteil bringt Rviz mit, dass man nachschauen kann wie die topics für Tiefen- und Farbbilder heißen.

Teleoperation:

roslaunch turtlebot\_teleop keyboard\_teleop.launch

Position des Turtlebot in Gazebo anzeigen lassen:

rostopic echo /gazebo/model\_states/pose[9]/position

9 ist in diesem Fall der Index der ‘mobile\_base’ findet man über

rostopic echo /gazebo/model\_states/name

**27.11.2017:**

Morgens habe ich mich in die Partikel-Filter einglesen, die zugrunde liegende Idee ist recht trivial, weshalb ich mich recht schnell davon ab- und der Inbetriebnahme des Navigation-Stacks hingewandt habe. Allerdings bin ich dann auch bei dem Navigations-Stack auf ein paar unschöne Probleme gestoßen.

Als erstes habe ich meine Kenntnisse bezüglich launch-files auf Stand gebracht, was bereits zu ersten brauchbaren Ergebnissen geführt hat. Bei der Problembekämpfung habe ich auch noch beiläufig über Gmapping und hector-SLAM dazugelernt. Nämlich arbeitet gmapping mit einem Partikel-Filter und mit Ray-Casting, was dazu führt, dass gmapping große Schwierigkeiten hat sich in ähnlichen Szenaieren zu lokalisieren. Das wurde bei dem Korridor-Besipiel sehr deutlich, wo der Roboter kontinuierlich in der Gegen umherspringt. Allgemein hat gmapping in den Simulationsszenarien sehr schlecht abgeshcnitten. Der Kommentar des Tutors war, man muss sich Mühe geben eine gute Karte zu zeichnen.

Aber auch hector\_slam hat seine Schwächen und zwar braucht es einen Weitwinkel Laserscanner, um zu funktionieren. In seiner Verteidigung muss man sagen, das es eben dafür konzepiert wurde. Endergebnis ist im Korridor wieder das vollständige Versagen des Ansatzes. Was zu dem Problem geführt hat, dass keiner der beiden Ansätze eine vernünftige Karte des Testszenarios hinbekommt. Außerdem wurde mir klar, dass ich den Sensor eventuell in gazebo implementieren muss, was sowohl zeit- als auch nervenintensiv ist. Besonders beschissen ist, dass das Kartenmalen überhaupt nicht Bestandteil meiner Arbeit ist. Aus diesem Grund habe ich mir überlegt, dass ich die Karte auch händisch malen kann und mit komischen Laserscanner die Navigation simuliere.

Der Vorteil ist, dass ich in der Simulationsumgebung die Kofiguration des Stacks durcharbeiten und dokumenteiren kann. Dann ist der Übergang in die Realität auf Seite der Infrastruktur kein Sprung ins kalte Wasser. Allerdings wird es schwieriger die Ergebnisse des Algorithmus mit der Realität zu vergleichen. An der Stelle muss man aber auch erwähnen, dass die Definition der Ziele noch bevorsteht. Insbesondere die Forderung, dass die Roboter Hindernissen ausweichen sollen, ist gleichermaßen wage wie nicht simulierbar. Insofern ist sowieso in Frage zu stellen, wie aussagekräftig die Simulation sein kann. Vielleicht muss man sich auf Seite der Simulation damit begnügen, dass sie lediglich zur Konfiguration und Illustration der Algorithmen genuttz werden kann. Zusätzlich besthe die Möglichkeit, dass man in der Realität auftretende Probleme in der Simulationsumgebung isoliert rekonstruieren kann, wodurch die Untersuchung vereinfacht wird.Ein weiteres Argument ist, dass die Konfiguration und Implementierung von Sensoren in gazebo pisslangweilig ist, keinen der Beteiligten interessiert udn ich es auch für keienn besonders wertvolle Erfahrung halte.

So betrachtet, denke ich, dass der richtige Ansatz darin besthet, die Karte händisch zu zeichnen und mit Standard-Konfiguration den Navigations-Stack möglichst schnell zum Laufen bekommen. Dabei gilt es die Konfiguration der Navigation möglichst detailiert zu erfassen, um Gefühl und Verständnis für die Stellschrauben zu erlangen. Anhand der Ergebnisse wird dann entschieden, wie es mit der Realität weitergeht, wobei die Anforderungen vom Chief auch nochmal wichtig sind.

Daraus würde ich für die nächsten zwei Wochen das Ziel formuileren, den vereinfachten Simulationsfall zum Laufen zu bekommen, um den Navigations-Stakc zu konfigurieren. Am Ende der zwei Wochen soll eine Dokumentation der Algorithmen plus deren Umsetzung in Ros vorliegen, die ich wiederum mit dem Chief durchgehen kann, um die letztendlichen Ziele für die Realität zu definieren. Das heißt für morgen: Erstmal die Karte zusammenpfuschen, Termin bei Martens über die Aufgabenstellung, mit der Karte Navigationsstack anschmeisen, und dann die Euinzelteile angehen. Navigationsstack soll sowohl mit als auch ohne AMCL laufen.

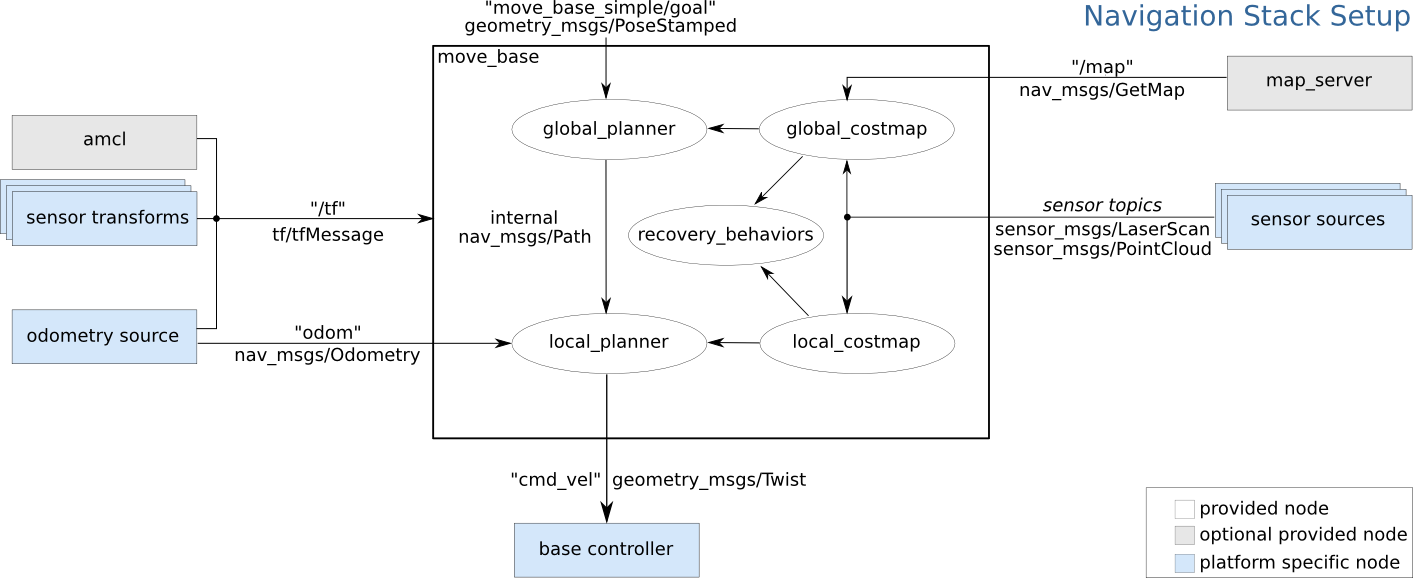
**28.11.2017:**

Die ersten Aufgaben heute bestand darin den Navigation-Stack in der Standard-Konfiguration zum Laufen zu bringen. Als erster Punkt stand die Erstellung der Korridor-Karte auf dem Plan. Die Lösung von dem Problem bringt MATLAB und der map\_server. Die bei der Kartographierung erstelle Karte wird über eine MATLAB-ROS-Node publiziert, von dem map\_server empfangen, der die Karte wiederum abspeichert.

Als nächstes war dann der Navigation-Stack dran, wobei das Ganze sowohl mit als auch ohne AMC-Lokalisierung funktionieren sollte. Außerdem sollen playground und Korridor-Karte verwendet werden können. Nach Tutorial hat der Standard-Fall mit Playground und AMCL recht schnell funktioniert, nach paar Problemchen auch ohne AMCL (hier mus sman manuell die Transformation von Odometrie zu Karten-Frame vorgeben, was ohne Lokalisierung die Identitätsabbildung ist). Um den Korridor zum Laufenzu bringen war der map\_server schwierig, weil in dem gespeicherten yaml-file noch eine NaN-Wert war, der den Server zum Absturz bringt. Den Faller kann man allerdings händisch korrigieren, woraufhin das Ganz funktoiniet.

**29.11.2017:**

Die letzten Tage stand die Inbetriebnahme des ROS-Navigation-Stack auf dem Programm, was bisher auch zu zufriedenstellenden Ergebnissen geführt. Jetzt geht es damit weiter das Ganze mit einem höherem Detailierungsgrad zu verstehen. Es soll die Brücke zwischen der ROS-Implementation und dem bisher dokumentierten Algorithmen geschlagen werden. Dafür starten wir wieder bei dem berühmt berüchtigten Bild:



Anhand der Darstellung werden die Hauptbestandteile ersichtlich, denen man auch jeweils Algorithmen und Funktionsprinzipien zuordenen kann. Diese zu anschauliche Darstellung verleitet einen dazu, zu glauben, das System verstanden zu haben. Allerdings steckt der Teufel im Detail: Sobald man sich näher damit beschäftigt, wie die – zum Teil nur oberflächlich- dokumentierten Algorithmen im ROS-Stack parametrisiert werden, kommen die ersten Probleme auf. An diesen Stellen tretten die bisher vernachlässigten Kopplungen zwischen mathematischer Algorithmik und informatiklastiger ROS-Implementation auf. Um von Verstädnis sprechen zu können, müssen wir diese Verzahnung in ihre kleinsten Bestandteile zerlegen und nachvollziehen. Wir beginne wie so viele; aber mit dem Anspruch tiefer vorzudringen:

Die Steuerung des Roboters wird im Navigation-Stack von dedm Packet move\_base übernommen, indem der globale und lokale Planer realisiert werden. Für die Algorithmen werden eine entsprechend globale und lokale Kostenkarte bereitsgestellt. Als letzte Komponente des Packets ist das recovery\_behavior zu nennen, dessen Beschreibung vorest noch warten muss. Die algorithmischen Prinzipien der beiden Planer wurden bereits ausführlich erläutert. Jetzt wenden wir uns den pragmatischeren Aspekten zu. Zu Beginn gilt die Aufmerksamkeit einmal mehr dem globalen Planner.

**Global-Planer**

Für die Umsetzung der globalen Plannung wird das ROS-Packet global\_planner [4] verwendet. Dieser verwendet eine Kostenkarte, wobei es sich um ein Occupancy-Grid, also eine ortsdiskrete Gitterdarstellung, handelt. Die Kostenkarte erfüllt die Funktion der Aktionsgewichtung. Für die Ermittlung des Pfades kann wahlweise entweder Dijkstra‘s Algorithmus oder A\* verwendet werden. Bei Betrachten der Wiki-Dokumentation fällt einem recht schnell auf, dass man die wissenschaftlichen Erklärungen der Lehrbücher hinter sich gelassen hat und es nun mit Internetdokumentation zu tun hat. Bei der Dokumentation wird das potential von Zellen und Pfaden berechnet, was synonym mit den hier genannten Kosten zu verstehen ist.

Ingesamt können bei dem Planer sieben Paramter konfiguriert werdne [4], wobei lediglich die folgende für den Anwendungsfall interessant sind:

- /allow\_unkown (bool, default:true):

Mit dem Pramter wird bestimmt, ob auch Pfade durch unbekannte Zellen geplant werden soll. Bei einer unbekannten Zelle ist nicht gewisst, ob diese frei oder von einem Hindernis belegt ist.

- /default\_tolerance (double, defualt:0.0):

Toleranzabstand mit dem der geplante Pfad die Zielposition erreichen soll, da es sich um eine diskrete Darstellung handelt stellt die Default-Toleranz von 0 kein Problem dar.

- /use\_dijkstra (book, default: true):

Flag ob dijkstra oder A\* verwendet werden soll. Der Grund für Dijkstra ist, dass die in A\* verwendeten Heuristiken manchmal gegen die Unterschätzungsforderung verstoßen, weshalb suboptimale Pfade erzeugt werden können.

- /use\_quadratic (bool, default: true)

Führt nach [4] dazu, dass eine quadratische Approximation für die Berechnung des Potentials/der Kosten genutzt wird. Im Folgendensatz shcreibt der Autor dennoch „Of what, the maintainer of this package has no idea“ [4]. Etwas tiefer im Internet stößt man auf die seriös wirkende Antwort: Und zwar berechnet das navfn-Packet (global\_planner ist eine neuerer Ersatz der jedoch die gleichen Funktionen bietet) das Potential aus der mit einer Zelle verbunden Kosten und zusätzlich der Distanz zwischen der vorherigen und folgenden Zelle. Die quadratische Approximation wird an dieser Stelle verwendet, um auch Quer-Verbindunge zu ermöglichen, da wie in unseren Beispieln nur oben/unten/links/rechts also vierweg-Verbindungen betrachtet werden.

-/use\_grid\_path(bool, default:false)

Wen auf true gestellt, folgt der Pfad wie in unserem Beispiel den Zellen, was dazu führt, dass der Roboter sich nur in 8 Richtungen bewegen kann (durch use\_qudratic wird die schräge bewegung über zellen möglich?!?!). Stellt man den Paramter auf false wird der Pfad geglättet, woraus rundere Bahnen resultieren.

Morgen steht die Kostenkarte an, als ertes wie das costmap packet mits eienn drei layern funktoiniert, wie konfiguriert werden kann, bzw wie diese Konfiguration für globale und lokale kostenkarte aussieht.

Als steht der lokale Planner auf dem Program, was auch morgen noch drin sein sollte.

**30.11.2017:**

**Globale Kostenkarte**

Die Basis für die Berechnung des globalen Pfades stellt die globale Kostenkarte dar, die mithilfe des ROS-Packetes „costmap\_2d“ [6] erstellt wurde. Wie der Name bereits erahnen lässt, wird das Packet genutzt, um 2D-Kotsenkarten zu erstellen. Allerdings können sowohl 2D als auch 3D-Räume mithilfe des Packets in eine 2D-Kostenkarte überführt werden.

Die Dokumentation des Packets beinhaltet erstmal ein paar Widersprüche. Zuerst wird in Abschnitt 3 behauptet, dass jeder Zelle drei mögliche Werte zugeordnet werdne. Dabei wird zwischen freien, blegente und unbekannten Zellen differenziert. Jedem dieser Zustand wird wiederum ein Kostenwert zugewiesen, die durch die Parameter

costmap\_2d::LETHAL\_OBSTACLE

costmap\_2d:FREE\_SPACE

costmap\_2d:NO\_INFORMATION

präzisiert werden.

Wenig später in Abschnitt 6 wird das Inflation-Prinzip vorgestellt. Aus jeder belegten Zelle werden die Kostenwerte von einer belegten Zelle ausgehend nach außen propagiert. Hier werden den Zellen allerdings diskrete Kostenwerte im Bereich von 255 bis 0 zugeordnet.

Die im Abschnitt 8.2 dargestellte Layerstruktur der Kostenkarte stellt eine mögliche Erklärung des Konfliktes dar. Und zwar kann bei dem Start des Packets konfiguriert werdne, aus welchen Layern die Karte zusammengesetzt wird. Zur Verfügung steht ein „Static Map Layer“[7], ein „Obstacle Map Layer“[8] und ein „Inflation Layer“[9]. Das erste basiert auf einer vorgegebenen Karte und enthält die durch die Karte definierten Hindernisse. In der zweiten Schicht – dem Obstacle Layer – werden Sensorwerte einglesen, um dynamische Hindernisse in der Kostenkarte einzuzeichnen. Zuletzt bleibt als dritte Schicht das Inflation Layer , in welchem die Kosten der Hindernisse propagiert werden.

Insofner wird der mölgiche Kostenraum von 255 bis 0 in den ersten beiden Schichten – dem Static Map Layer und Obstacle Map Layer – auf die drei Werte unbekannt, frei und belegt reduziert. Erst in der dritten Schicht – dem Inflation Layer- wird nach den Inflations-Prinzip eine Kostenverteilung ausgehend von den belegten Zellen berechnet.

Im Navigation-Stack werden zwei Instanzen des Packets costmap\_2d ausgeführt. Die erste erstellt die globale Kostenkarte. Die zweite die lokale. Die beiden Kostenkarte teilen sich eine Menge von gemeinsamen Paramtern die für die Standard-Konfiguration in der Datei

/opt/ros/kinetic/share/turtlebot\_navigation/param/costmap\_common\_params.yaml

festgehalten sind. Die speziellen Einstellungen werden in den Dateien

/opt/ros/kinetic/share/turtlebot\_navigation/param/local\_costmap\_params.yaml

und

/opt/ros/kinetic/share/turtlebot\_navigation/param/global\_costmap\_params.yaml

vorgenommen.

Die globale Kostenkarte wird mitden Paramtern

global\_costmap:

global\_frame: /map

robot\_base\_frame: /base\_footprint

update\_frequency: 1.0

publish\_frequency: 0.5

static\_map: true

transform\_tolerance: 0.5

plugins:

- {name: static\_layer, type: "costmap\_2d::StaticLayer"}

- {name: obstacle\_layer, type: "costmap\_2d::VoxelLayer"}

- {name: inflation\_layer, type: "costmap\_2d::InflationLayer"}

konfiguriert. Relevant in diesem Paramtersatz ist die Layerstruktur: Es werden alle drei Schichten verwendet. Im Gegensatz dazu die lokale Konfiguration

local\_costmap:

global\_frame: odom

robot\_base\_frame: /base\_footprint

update\_frequency: 5.0

publish\_frequency: 2.0

static\_map: false

rolling\_window: true

width: 4.0

height: 4.0

resolution: 0.05

transform\_tolerance: 0.5

plugins:

- {name: obstacle\_layer, type: "costmap\_2d::VoxelLayer"}

- {name: inflation\_layer, type: "costmap\_2d::InflationLayer"}

in der lediglich ein Obstacle und Inflation layer verwendet ewrden. Außerdem wird als globales Koordinatensystem nicht die statische Karte sondern das Odometrie-Bezugssystem verwendet. Das bedeutet wiederum, dass sich die Karte mit dem Roboter bewegt – wie es bei der lokalen Kostenkarte auch gewolt ist.

Die Konfiguration der drei Layer wird in den geteilten Parametersatz durchgeführt:

max\_obstacle\_height: 0.60 # assume something like an arm is mounted on top of the robot

# Obstacle Cost Shaping (http://wiki.ros.org/costmap\_2d/hydro/inflation)

robot\_radius: 0.20 # distance a circular robot should be clear of the obstacle (kobuki: 0.18)

# footprint: [[x0, y0], [x1, y1], ... [xn, yn]] # if the robot is not circular

map\_type: voxel

obstacle\_layer:

enabled: true

max\_obstacle\_height: 0.6

origin\_z: 0.0

z\_resolution: 0.2

z\_voxels: 2

unknown\_threshold: 15

mark\_threshold: 0

combination\_method: 1

track\_unknown\_space: true #true needed for disabling global path planning through unknown space

obstacle\_range: 2.5

raytrace\_range: 3.0

origin\_z: 0.0

z\_resolution: 0.2

z\_voxels: 2

publish\_voxel\_map: false

observation\_sources: scan bump

scan:

data\_type: LaserScan

topic: scan

marking: true

clearing: true

min\_obstacle\_height: 0.25

max\_obstacle\_height: 0.35

bump:

data\_type: PointCloud2

topic: mobile\_base/sensors/bumper\_pointcloud

marking: true

clearing: false

min\_obstacle\_height: 0.0

max\_obstacle\_height: 0.15

# for debugging only, let's you see the entire voxel grid

#cost\_scaling\_factor and inflation\_radius were now moved to the inflation\_layer ns

inflation\_layer:

enabled: true

cost\_scaling\_factor: 5.0 # exponential rate at which the obstacle cost drops off (default: 10)

inflation\_radius: 0.5 # max. distance from an obstacle at which costs are incurred for planning paths.

static\_layer:

enabled: true

Das Static-Map-Layer wird lediglich aktiviert. Ansonsten wird die Standard-Parametrisierung übernommen, in der lediglich der Kostenwert für die freien, unbekannten und belegten Zellen festgelegt wird und die topic defineirt, unter der die Karte veröffentlicht wird.

Die umfangreichste Konfiguration wird für das Obstacle-Layer vorgenommen. Hier insbesondere Einstellung für die Handhabung der z-Achse durchgeführt, also wie hoch die Hindernisse sind un dmit welcher Auuflösung die untersucht werden sollen. Ansonste werdne die Quellen von Sensorinformation festgelegt, das sind in diesem Fall einmal die topic scan, unter der die Daten des LaserScanners veröffentlicht werden. Und die Topic bump, wo die Informationen der Kollisionsdetektion verbreitet werden.

Zuletzt bleibt das Inflation-Layer für das nur zwei Parameter gestezt werden können. Der erste cost\_scaling\_factor beeinflusst den Exponent der e-Funktion, welche die vom Hindernis ausgehenden Abfall der Kosten beshcreibt. Der zweite Parameter inflation\_radius gibt an, bis zu welchem Abstand ein Hindernis die Kosten seiner Umgebung beeinflusst.

**Lokaler Planner**

Die lokale Plannung übernimmt das Packet, dwa\_local\_planner [10], das den Dynamic Window Appraoch verfolgt. Da eine recht große Zahl an Paramtern konfiguriert werden kann, erfoglt ine Unterteilugn in vier Gruppen.

Zuerst werden die Robot-Configuration-Paramters festgelegt. Hier werdne Daten wie minimale, maximale Beschleunigungen, Translations und Rotationsgeschwindigkeiten festgelegt:

~<name>/acc\_lim\_x (double, default: 2.5)

* The x acceleration limit of the robot in meters/sec^2

~<name>/acc\_lim\_y (double, default: 2.5)

* The y acceleration limit of the robot in meters/sec^2

~<name>/acc\_lim\_th (double, default: 3.2)

* The rotational acceleration limit of the robot in radians/sec^2

~<name>/max\_trans\_vel (double, default: 0.55)

* The absolute value of the maximum translational velocity for the robot in m/s

~<name>/min\_trans\_vel (double, default: 0.1)

* The absolute value of the minimum translational velocity for the robot in m/s

~<name>/max\_vel\_x (double, default: 0.55)

* The maximum x velocity for the robot in m/s.

~<name>/min\_vel\_x (double, default: 0.0)

* The minimum x velocity for the robot in m/s, negative for backwards motion.

~<name>/max\_vel\_y (double, default: 0.1)

* The maximum y velocity for the robot in m/s

~<name>/min\_vel\_y (double, default: -0.1)

* The minimum y velocity for the robot in m/s

~<name>/max\_rot\_vel (double, default: 1.0)

* The absolute value of the maximum rotational velocity for the robot in rad/s

~<name>/min\_rot\_vel (double, default: 0.4)

* The absolute value of the minimum rotational velocity for the robot in rad/s

Als nächstes folgen die Toleranzen für das Ziel

~<name>/yaw\_goal\_tolerance (double, default: 0.05)

* The tolerance in radians for the controller in yaw/rotation when achieving its goal

~<name>/xy\_goal\_tolerance (double, default: 0.10)

* The tolerance in meters for the controller in the x & y distance when achieving a goal

~<name>/latch\_xy\_goal\_tolerance (bool, default: false)

* If goal tolerance is latched, if the robot ever reaches the goal xy location it will simply rotate in place, even if it ends up outside the goal tolerance while it is doing so.

Simulations Paramater

~<name>/sim\_time (double, default: 1.7)

* The amount of time to forward-simulate trajectories in seconds

~<name>/sim\_granularity (double, default: 0.025)

* The step size, in meters, to take between points on a given trajectory

~<name>/vx\_samples (integer, default: 3)

* The number of samples to use when exploring the x velocity space

~<name>/vy\_samples (integer, default: 10)

* The number of samples to use when exploring the y velocity space

~<name>/vth\_samples (integer, default: 20)

* The number of samples to use when exploring the theta velocity space

~<name>/controller\_frequency (double, default: 20.0)

* The frequency at which this controller will be called in Hz. Uses searchParam to read the parameter from parent namespaces if not set in the namespace of the controller. For use with [move\_base](http://wiki.ros.org/move_base), this means that you only need to set its "controller\_frequency" parameter and can safely leave this one unset.

Bewertungs-Paramter

cost =

path\_distance\_bias \* (distance to path from the endpoint of the trajectory in meters)

+ goal\_distance\_bias \* (distance to local goal from the endpoint of the trajectory in meters)

+ occdist\_scale \* (maximum obstacle cost along the trajectory in obstacle cost (0-254))

~<name>/path\_distance\_bias (double, default: 32.0)

* The weighting for how much the controller should stay close to the path it was given

~<name>/goal\_distance\_bias (double, default: 24.0)

* The weighting for how much the controller should attempt to reach its local goal, also controls speed

~<name>/occdist\_scale (double, default: 0.01)

* The weighting for how much the controller should attempt to avoid obstacles

~<name>/forward\_point\_distance (double, default: 0.325)

* The distance from the center point of the robot to place an additional scoring point, in meters

~<name>/stop\_time\_buffer (double, default: 0.2)

* The amount of time that the robot must stop before a collision in order for a trajectory to be considered valid in seconds

~<name>/scaling\_speed (double, default: 0.25)

* The absolute value of the velocity at which to start scaling the robot's footprint, in m/s

~<name>/max\_scaling\_factor (double, default: 0.2)

* The maximum factor to scale the robot's footprint by

Die Erklärung von diesem Packet ist wirklich nur sinvoll im Rahmen von dem globalen Planner. Ansonsten bleibt auch noch als offene Frage wie die lokale Kostenkarte sich auf DWA auswirkt. Weil nach dem offiziellen Paper hat das ja nichts miteinander zu tun.

**1.12.2017**

Es ist noch nicht wirklich sinnvoll die Teile hier in Latex zu übertragen, weil sich erst später zeigen wird, was davon relevant sein wird. Insofern ergibt es Sinn, bei amcl weiter zu machen, und dann die Standardkonfiguration mit der gegebenen Situation zu vergleichen. Wir beginnen an der Stelle, wo die größten Diskrepanzen zwischen Realität und Konfiguration auftreten. Ich vermute, dass dieser Fall bei amcl eintreten wird, weil das Paket mit kinetic oder asus Kamera funktioniert.

**3.12.2017**

Ich habe heute erstmal die Dokumentation des Partikelfilters begonnen, mich dabei aber auf eine recht anschauliche Beschreibung beschränkt. Das sollte auch genügen, das mathematische Detail scheint an der Stelle - zumindest vorläufig- zu genügen, um die Funktion nachvollziehen zu können. Bei der Dokumentation fehlen noch die beiden Adaptionen „zufällig“ und „KLD“ die dann letztenendes in AMCL zum Einsatz kommen. Ansonsten muss ich noch das Messmodell durcharbeiten/´dokumentieren.

Jetzt ist der aktuelle Tagespunkt die Diskussion wie die Versuchsreihen auszusehen haben. Was ist das letztendliche Ziel? Welche Rahmenbedigungen und Anforderungen geben wir vor?

Die selbstverständliche Forderung ist, dass man in einer vorgegebenen, mittels hector\_slam erstellten Karte ein Ziel vorgeben kann, das der Roboter dann ansteuert. Hier ist die erste Frage wie relevant ist die Toleranz mit der der Roboter das Ziel erreicht? Ich würde sagen, nicht besonders groß, selbst 20cm sind da ja noch in Ordnung oder?

Hindernissen ausweichen ist ein sehr großer Punkt, primärer Fokus meiner Arbeit. Dabei stellt sich jetzt aber die Frage wie wir das Benchmarken und definieren. Man kann die Hindernisse eigentlich in drei Kategorien einstufen: Einmal die auf der Karte registrierten, die sind statisch und nach wie vor da. Zur zweiten Stufe gehören statische Hindernisse, die nicht in der Karte erfasst wurden. Also der Roboter muss drauf reagieren, aber sie bewegen sich nicht. Die können auch noch sehr schön in einer reproduzierbaren Versuchsumgebung erfasst werden. Schwierig werden Stufe drei Hinerdnisse, welche die sich bewegen und deshalb auch nicht Teil der Karte sind. Das Problem dabei ist es das Ganze in einen reproduzierbaren Rahmen zu formulieren. Am Anfang sind vielleicht Stufe zwei genug, und wenn da ein gutes Ergebnis rauskommt schaut man weiter ob das auch bei Menschen hilft, bzw. was für Probleme das sind.

Der nächste Punkt ist die Lokalisierung, ich möchte dass es reicht den Roboter auszusetzen, vielleicht mittels Rviz eine Schätzung der aktuellen Position mitzugeben und den rest soll der Roboter selber schaffen. Das würde wiederum reine Odometrie ausschließen, weil dabei die exakte Startposition vorliegen muss und da jeder Fehler unmittelbare Auswirkungen auf die Navigation hat.

Man muss ja eigentlich dazu sagen, dass AMCL nicht erforderlich ist um Hindernissen auszuweichen.

Zusammenfassung der Hauptfrage:

* Festlegung von der Umgebung: Bleiben wir bei Labor? Aus pragmatischen Gründen sollten wir eine Raum fix vorgeben, bei dem die erste Untersuchung bleibt? Die Ergebnisse kann man dann mal gerne auch woanders ausprobieren?
* Spezifikation der Hindernisse: Drei Stufen Unterteilung, ich schlage vor als Hauptkriterium vorerst die zweite Stufe zu nehmen, aber mit Ausblick auf drei.
* Anfangslokalisierung, soll der Roboter mit mehr als nur Odometire fahren? Das bedeutet praktisch: AMCL ja oder nein? Ich sage klar ja weil Kreise malen primitiv ist.
* Wie genau soll die Zielposition erreicht werden? In meinen Augen die irrelevanteste Frage, es sollte das mit einigermaßen Genauigkeiten erreichen, was vielleicht 25-50cm sind. Schätze ich aber als eher kleines Problem ein.
* Bleiben wir bei zu erprobenden Möglichkeiten, bei den Sachen die im ROS-Stack drin sind?

**4.12.2017**

Heute steht die Inbetriebnahme der Turtlebots auf dem Programm, prinzipiell haben wir vier Turtlebots und einen Master, wobei es sich nur um einen Mini-PC handelt, der dann mit dem Dign verbunden wird.

Die Jungs haben noch Unbuntu 14.04 verwendet, da es zum damaligen Zeitpunkt die aktuellste Version mit stabiler Untersützung für die TurtleBots war. Mittlerweile sollte das auch mit 16.04 funktionieren. Der Vorteil von 16.04 wäre, dass man meine Desktop-Maschine an das System hängen könnte. Allerdings müste man dann halt updaten, wobei sich die Frage stellt wie groß der Aufwand ist. Eigentlich sollte man auch über den Master Rviz und Co. Für die Visualisierung starten können.

Wir bleiben jetzt erstmal bei der installierten Variante und versuchen das System in der bestehend Form zum Laufen zu bringen. Die erste Aufgabe ist das Aufzeichnen einer Karte mithilfe von hector\_slam.

**5.12.2017**

Karteaufzeichnen hat gestern im ersten Test funktioniert, heute nehmen wir als erstes einen klkeinen Teil des Labors auf, um im Anschluss da rudimentäre Navigation zu testen. Das Ziel ist es einmal mit AMCL einmal nur mit Odometrie, Ziele in circa 1-2 Meter Abstand anzufahren. Zu Beginn nur das Simpelste: Gerade ausfahren.

Für die erste Navigation soll AMCL verwendet werden, es wird die neue karte verwendet, über Rviz visualisiret, wo auch das Ziel vorgegeben wird. Im Vergleich zu dem bisherigen launch-file brauchen wir auch noch ein kobuki bringup.

Bisheriges Rumprobieren und Stochern hat nicht viel gebracht. Es gibt jetzt zwei Möglichkeiten: Entweder die offzielle Dokumentation zum Einrichten der Turtlebots durchgehen oder die Arbeit der Turtleboys. Ich fang bei den Turtleboys an, weil die auch unseren Laserscanner angeschmissen haben.

**Minimal Bringup + Teleop**

Als erstes Einführungsbeispiel soll eine Launch-Datei erstellt werden, die die Turtlebot-HW initialisiert und über die Tastatur den Turtlebot steuern kann.

Nach dem ROS-Wiki [11] genügt auf dem Turtlebot-PC:

roslaunch turtlebot\_bringup minimal.launch --screen

Auf dem Master:

roslaunch turtlebot\_teleop keyboard\_teleop.launch

In der launch-Datei für die Kartenerstellung verwenden auch die Turtleboys das minimal-bringup, um den Turtlebot zu initialisieren. Zusätzlich wird lediglich der laser-scanner initalisiert. Daraus folgt, dass für ein HW-Launch file folgendes genügt:

<launch>

<!-- Basis-Initialisierung -->

<include file=“$(find turtlebot\_bringup)/launch/minimal.launch“/>

<!-- Initialisierung des SICK-Laserscanners -->

< param name=“robot\_description“ command=“$(find xacro)xacro.py ‚$(find sick\_tim)/urdf/example.urdf.xacro‘“/>

< include file=“$(find turtlebot\_chor\_navigation)/launch/include/sick\_tim551\_2050001.launch“/>

</launch>

Wenn dieses Bringup auf dem Slave-PC gestartet wird, sollte einerseits der Roboter über die entsprechenden ROS-Topics gesteuert werden, andererseits der Laserscanner auf der topic /scan publishen. Ansonsten kann ich mir voerst eigentlich nur noch Probleme bei dem tf-tree vorstellen.

Wenn man auf dem Ros-Master die custom-launch Datei für die Odometrie-Navigation startet, beschwert sich Rviz, dass sowohl RobotModel als auch Laser-Scan fehlen. Bei dem Laser-Scan fehle die Transformation. Mithilfe von

rosrun tf view\_frames

kann man sich den aktuellen Transformationsbaum anezeigen lassen, der zeigt, dass eine Verbindung zwischen /map/odom/base\_footprint und base\_link fehlt. In der Simulation wird die Bervingund zwischen base\_footprint und base\_link von /robot\_state\_publisher also gazebo veröffentlicht. In der real Situation gibt es zar die node robot\_state\_publisher, allerdings gibt die nicht den tf aus.

Das navigationproblem scheint an den Transformationen zu leigen. view\_frames zeigt zwar mitterlweile eine vollständige Verbindugn an, allerdings wanrt move\_base, dass die Transformation fehlschlägt, weil die Wartezeit zwischen base\_footprint und map zu lange sei.

Die Wartezeiten entstehen dadruch, dass die Systemuhren der beiden PCs nicht syncrhonisiert sind. Als Abhilfe reicht es auf dem Turtlebot-PC

sudo ntpdate 192.168.0.100

auszuführen.

Auf dem turtle-master sind jetzt zwei launch Dateien, die sowohl AMCL als auch Odometrie Navigation machen, die funktionieren ganz i.O. solange keine unerwarteten Hindernisse rumstehen. Da gibt’s dann überfahrene Mülleimer.

Als nächstes steht jetzt auf dem Plan ordentliche Karte von den Räumen zu machen, eine – nur von dem Labor – ist bereits fertig. Die Zweite sollte dann auch noch den Nebenraum beinhalten.

Die Lokalisierung scheint mit dem Pose-Estimate ganz gut zu gehen, wenn mand en Roboter händisch umstellt. Was gar nicht geht ist das Erkennen von Hindernissen beim lokalen Planer, da gilt es mal die DWA-Config zu checken.

Als erste Möglichkeit fällt auf, dass das Fesnter von möglichen Trajektorien bzw. Geschwindigkeiten sehr klein ist. Vielleicht führt das dazu, dass bei der Planung niemals ein Hindernis erreicht wird, bzw. wenn man davor steht es schon zu spät ist.

**8.12.2017**

Heute gilt es die Hinderniserkennenung zum Laufen zu bringen. Als erstes wird ein eigenes Launch-File für move-base erstellt, indem dann wiederum eine Custom-Konfiguration des lokalen Planners nach DW-Ansatz durchgeführt werden kann.

Erster Versuch bestand darin die Sim-Time zu erhöhen, wodurch bei der lokalen Planung ein größers Zeitfesnter und somit auch größere Distanzen betrachtet ewren, die Änderun gist in Rviz auch ersichtlich, weil die Trajektorien viel länger werden. Ändert allerdings nichts daran, dass Hindernisse überfahren werden.

Für leichtere Debug-Zewcke habe ich in den Custom-Paramtern die maximale Geschwindigkeit heruntergestellt.

Das Hindernis scheint in der lokalen Kostenkarte zu überhaupt keinen Veränderungen zu führen, was auch das Problem bei der Detektion zu sein scheint.

Erste Vermutung: die /scan topic wird nicht an move\_base übergeben. Mithifle von rqt\_graph kann man anzeigen welche topics von welchen Nodes an welche übergeben werden. Dort zeigt sich, dass /scan von dem Sensor an move\_base geht.

Zweite Vermutung: Bei der Planung sagt der, dass der Sensor außerhalb der Kartengrenze leigt, weshalb die kostenkarte dfür nicht raytracen kann. Lösung für das Problem war, dass der Sensor bei 0.55 liegt aber nur 2 Voxels der Höhe 0.2 in der Kostenkarte eignestellt waren, erhöhen auf 4 Voxels macht ddas es geht.

**10.12.2017**

Nachdem am Freitag die Hinderniserkennung auch noch funktioniert hat, beginnt heute die Dokumentationsarbeit. Durch die Gliederung der Dokumentation, was Teil und was nicht Teil sein soll, werden auch die letzten Aufgaben definiert und die Grenzen der Arbeit abgesteckt. Bevor wir uns den einzelnen Teilen widmen mein eigener Anspruch an die Arbeit geklärt werden. Sie dient als Übung meiner Artikulation und sprachlicher Fähigkeiten, wofür ich wiederum den Anspruch erheben muss, dass diese Arbeit einen wahrhaftigen Wert mit sich bringt. Das Alleinstellungsmerkmale der Arbeit kann sein, dass sie die Brücke schlägt zwischen den mathematischen Funktionsprinzipien und der ROS-seitigen Implementierung. In der ROS-Community stößt jede ansatzweise tiefergreifende Frage auf das abrupte Ende: Lies in Buch x nach. Das Internet ist gespickt von anschaulichen Erklräung, aber wir wissen: Da der Teufel im Detail steckt ist anschaulich nur ein schönes Synonym für oberflächlich. Hier muss sich meine Arbeit erheben, revoltieren gegen die bestehenden Wege. Aber die Frage ist wieso? Eigentlich hat mein Experiment erwiesen, dass die fertige Konfiguration genügt. Mein Argument ist, dass in Technik mehr Wert steckt, als lediglich ein funktionierendes Endprodukt. Die Kunst besteht darin, Beispiele und Anwendungsfälle zu finden, die die letzte Gleichung der mathematik illustrieren. Aber umgekehrt will ich die Mathematik demaskieren. Die unnütze, unnötig komplexe Darstellung, die lediglich der Illussion dient, den Schein der unendlichen Schwere erhält, will ich abreisen. Ich will der Wissenschaft ihre Selbstgefälligkeit vorführen.

Das heißt, dass ich für jedes der theoretischen Konzepte versuche, aussagekräftige Anwendugnsbeispiele zu finden. Wir wagen es MATLAB an das Ganze zu hängen, um zu zeigen, dass trotz all des Informatikerhasses MATLAB geil sein kann. Wir probieren die Simulation, um Probleme zu identifizieren. Komplexität eines Hänne-Ei-problems auseinander zu legen.

Vom Aufbau bedeutet das, dass wir uns zunächst auf den ROS-Navigation-Stack konzentrieren, weshalb es damit inhaltich auch losgeht.

**Einleitung**

Gliederung:

* Herleitung der Aufgabe: Chort-Geschichten
* Ziel von Fahrendne Robotern mit Lautsprechern
* Vorarbeit
  + Turtlebot als Hardware
  + Robot-Operating System
  + Erste Navigation mittels Navigation-Stack
* Meine Arbeit: Intensivere Abareitung der Navigation, Abgrenzung der aufgabe sowohl was Funktionen angeht, als auch den Anspruch der Dokumentation/schriftlichen Arbeit
  + Beschränkung auf ROS-Navigation-Stack
  + Hector\_slam Karte liegt vor
  + Zuerst ein einzelner Roboter ,dann mehrere
  + Schwerpunkt 1: Lokalisierung
  + Schwerpunkt 2: Hinderniserkennung bei der Navigation
  + Der Stack soll in seine Einzelteile zerlegt werden, deren Funktionsprinzip erarbeitet wird und anhand von passenden Anwendungsbeispielen und Werkzeugen illustriert wird.
  + Als möglcihe Werkzeuge werden Simulationstools wie gazebo verwendet
  + Als Werkzeuge werden Tools wie MATLAb verwendet
  + Es gilt die oberflächlich eROS-Dokumentation mit der Mathematik der Algorithmen zu verbinden

***Herleitung Aufgabe/Chor-Geschichten***Seit einiger Zeit wird an der Hochschule Karlsruhe die Projektidee eines künstlichen Chors verfolgt, dessen Ziel darin besteht, eine Audiospuren simultan in mehreren Tonhöhen wiederzugeben, um einen mehrstimmigen Chor zu erzeugen. Das zugrundeliegende technische Prinzip lässt sich recht leicht erläutern: Die originale Tonspur wird zunächst mittels FFT in den Frequenzbereich transformiert, wo das Signale in die gewünschte Tonhöhe bzw. Frequenzbereich verschoben wird. Abschließend werden die Signale zurück in den Zeitbereich transformiert und abgespielt. Allerdings gestaltet sich die Umsetzung weitaus anspruchsvoller als das theoretische Konzept. Beispielsweise führen Aussetzer im WLA-Netzwerk, Puffergrößen der Audiotreiber und diskrete Stützstellen des Frequenzspektrums zu unerwarteten Problemen, die sich in Form von knacksenden Lautsprechern Gehör verschaffen. Nichts desto trotz hat der Entwicklungsprozess erreicht, an welchem der Chor kaum noch unerwünschte Töne von sich gibt, weshalb nun die erste große Erweiterung angestrebt wird. Aktuell setzt sich das System aus einem Master-PC und vier BeagleBones zusammen, die jeweils mit einem Lautsprecher ausgestattet in einem Raum verteilt werden. Auf dem Master wird eine Tonspur eingespielt, die er in die gewünschten Tonhöhen pitcht und anschließend über WLAN an die BeagleBones weitergibt, welche jeweils eine der Stimmen über den Lautsprecher ausgeben.

***Ziel von fahrendem Roboter***Im nächsten Schritt soll der Chor mobilisiert werden. Das heißt jedes BeagleBone plus Lautsprecher wird auf einem Roboter montiert, die bei einer Vorführung einen beliebigen Raum betreten können und diesen auf der Suche nach eines passenden Podiums erkunden. Wurde Letzteres gefunden, so sollen sich die Roboter dort im Halbkreis formieren und anschließend vorsingen.

Die neue Aufgabe, die sich mit der autonomen Navigation der Roboter beschäftigt, kann vollkommen entkoppelt von dem Chor betrachtet und bearbeitet werden. Dieser zweite Entwicklungsstrang befindet sich allerdings noch im Anfangsstadium, da sich bisher nur ein einzelnes Entwicklungsprojekt dem Thema gewidmet hat. In dieser Arbeit wurde ein erster Grundstein gelegt, der sich einerseits aus der Auswahl und Inbetriebnahme der Roboter, anderseits aus einem ersten Proof-Of-Concept in Sachen Navigation zusammensetzt.

***Turtlebot als Hardware*** Als mobile Plattform für die Lautsprecher fiel die Wahl auf den Turtlebot 2, der von Willow Garage entwickelt wurde. Der Roboter setzt sich aus der Kobuki-Basis, einem Rechner und einer Asus-Xtion-Pro Live zusammen, wobei Letztere sowohl Kamera als auch Tiefensensor zur Verfügug stellt. Im Rahmen der Vorgängerarbeit wurde der Roboter zusätzlich mit dem Laserscanner TIM-Sick-551 ausgestattet, der als primärer Sensor für die Navigation verwendet wird.

***Robot-Operating-System*** Die Programmierung erfolgt mithilfe des so genannten Robot-Operating-System (ROS), wobei es sich ironischerweise um kein Betriebssystem sondern Middleware handelt, die eine Vielzahl von Tools und Paketen für die autonome Navigation bereitstellt. ROS-Vorteile:

***Erste Navigation mittels Navigation-Stack***

Vorarbeit:

* Turtlebot als Hardware
  + Kobuki Plattform, eigner Rechner ASUS XTION PRO Live
  + Standard-HW in Forschung und Hobby dadurch viel Unterstützung
  + In gazebo als Simulation schon bereit
  + Custom-Anpassung: Eigener Mini-Rechner, Laserscanner
* Robot-Operating-System
  + Ist Middelware mit Paketen etc
  + Man erspart sich eigen Implementierung der Navigations-Algorithmen
  + Enthält fertiges Netzwerk für die Kommunikation zwischen Robotern
  + An das Netz können auch externe Einheiten geschaltet werden, was für Debugging wichtig ist
  + In dem Netz könnne di eRoboter durch Simulationen ersetzt werden
* Erste Navigation mittels Navigation-Stack
  + Als erstes Karte aufzeichnen in der im Anschluss navigiert wird
  + Mehrere Ansätze veroflgt, einerseits Odometrie, andererseits AMCL
  + Letzten endes für Odometrie entschieden, weil bessere Ergebnisse
  + Problem: Keine Lokalisierungsmöglichkeit, Startposition muss exakt sein, zero Robustheit
  + Keine Hinderniserkennung

***Turtlebot als Hardware*** Als mobile Plattform für die Lautsprecher fiel die Wahl auf den Turtlebot 2, der von Willow Garage entwickelt wurde. Der Roboter setzt sich aus der Kobuki-Basis, einem Rechner und einer Asus-Xtion-Pro Live zusammen, wobei Letztere sowohl Kamera als auch Tiefensensor zur Verfügug stellt. Der Turtlebot 2 bringt als Vorteil mit sich, dass der Roboter weite Anwendung im Bereich von Hobby und Forschungsanwendungen finden, weshalb eine breite Community-Unterstützung zu Verfügung steht. Ein in dieser Arbeit eminent wichtiges Beispiel stellt die Simulation des Roboters mithilfe von gazebo dar: Hier besteht bereits eine vollständige Integration des Turtlebot 2, die open source zugänglich ist. Im Rahmen der Vorgängerarbeit wurde der Roboter zusätzlich mit dem Laserscanner TIM-Sick-551 ausgestattet, der als primärer Sensor für die Navigation verwendet wird.

***Robot-Operating-System*** Die Programmierung erfolgt mithilfe des so genannten Robot-Operating-System (ROS), wobei es sich ironischerweise um kein Betriebssystem sondern Middleware handelt, die eine Vielzahl von Tools und Paketen für die autonome Navigation bereitstellt. Dadurch müssen die Algorithmen für die Navigation und Kartographierung nicht eigenständig implementiert werden, sondern können in Form von ROS-Paketen eingesetzt werden. Dadurch beschränkt sich die Inbetriebnahme auf die entsprechende Parametrisierung der ROS-Funktionen. Außerdem beinhaltet ROS ein Netzwerk, über das die Roboter kommunizieren können. Mithilfe der Kommunikationskanäle können auch externe Werkzeuge auf relevanten Daten zugreifen, womit komfortable und effiziente Debugging-Wege geschaffen werden. Umgekehrt können zu Beginn die Roboter auch aus dem Netz entfertn und durch ein Simulationstool wie gazebo ersetzt werden. So können die Algorithmen zunächst anhand einer Simulation erprobt und konfiguriert werden und im Anschluss unverändert auf die Roboter übertragen werden.

***Erste Navigation mittels Navigation-Stack*** Die im Vorprojekt erarbetiete Navigationslösung basiert auf dem ROS-Navigatoin-Stack und setzt sich aus zwei Teilen zusammen. Im ersten Schritt wird mit dem ROS-Packet hector\_slam eine Karte der Umgebung aufgezeichnet. Anschließend navigieren die Roboter anhand dieser Karte zu dem vorgegebenen Ziel, wobei das ROS-Paket move\_base zum Einsatz kommt. Bei dessen Konfiguration wurden zwei verschiedene Ansätze verfolgt: Bei dem Ersten erfolgt die Lokalisierung des Roboters ausschließlich anhand der Odometriedaten. Der zweite Ansatz greift auf die AMC-Lokalisierung zurück. Der Vorteil der zweiten Variante liegt darin, dass die Ausgangsposition des Roboter snciht bekannt sein muss. Bei der Navigation mittels Odometrie übertragen sich sämtliche Fehler bei der Angabe der Anfangsposition unmittelbar auf die Navigation. Im Gegensatz dazu versprecht die AMCL-Variante – zumindest theoretisch - ein höheres Maß an Robustheit. Allerdings konnte diese Hoffnungen in den Experimenten nicht bestätigt werden, weshalb letztenendes die Odometrie-Navigation verwendet wurde. Außerdem war diese Lösung nicht im Stande Hindernisse, die nicth auf der Karte verzeichnet sind, während der Navigation zu erkennen, geschweige denn denen auszuweichen.

Meine Arbeit:

* **Funktionale Anforderungen:**
  + Lokalisierung
  + Hinderniserkennung und Ausweichen
  + Einschränkung der Lösungsmöglichkeiten indem auf ROS-Stack beschränkt
  + Einschränkung: Die Karte ist bereits bekannt, mit hector\_slam gemacht
  + Zu Beginn nur einen Roboter navigieren, im Anschluss mehrere
* Anforderungen an die Vorgehensweise:
  + Es sollen die zu Grunde liegenden Algorithmen aufgearbeitet und erläutert werden
  + Für die Illustration der Beispiel soll auf simulationswerkzeuge zurückgegriffen werden
  + Die praktisch stark gekoppelten Funktionen sollen mithilfe der Simulation getrennt werden
  + Es soll der Zusammenhang zwischen den mathematischen Konzepten und den ROS-Schnittestellen und Paramtersätzen geschaffen werden
  + Konstruktion von Anwendugnsbiespielen, die Charakteristika de Algorithmen illustrieren

**Funktionale Anforderungen:** An dieser Stelle knüpft die vorliegende Arbeit an: Die Navigation soll an den Punkt gebracht werden, wo die Roboter im Stande sind, sich selbst auf der Karte zu lokalisieren, unerwartete Hindernisse zu detektieren und diesen entsprechend auszuweichen. Hierfür soll allerdings auf die bestehenden Möglichkeiten des ROS-Navigation-Stack zurückgegriffen werden, um die Menge von Lösungsmölgichkeiten einzuschränken. Auch die Anforderung, dass die Navigation auf eine mit hector\_slam aufgzeichnete Karte zurückgreifen kann, wird übernommen. Im ersten Schritt wird lediglich ein einzelner Roboter betrachtet. Sobald dieser die Anforderungen im Bereich der autonomen Navigation erfüllt, werden die Ergebnisse auf eine Gruppe von vier Robotern übertragen.

**Anforderungen an die Vorgehensweise:** Neben den rein funktionalen Zielen legt diese Arbeit einen besonderen Wert auf dabei verwndete Vorgehensweise. Die Arbeit beginnt bei den Grundlagen. Der Navigation-Stack soll in seine funktionalen Einzelteile zerlegt werden, deren Funtionsprinzip im Detail erläutert werden sollen. Mithilfe der so gewonnen Erkenntnisse sollen die Komponenten konfiguriert und wieder zusammengesetzt werden. Die Arbeit erhebt den Anspruch nicht nur eine Lösungs zu erzielen, die die Anforderungen erfüllt, sondern auch zu erklären vermag, wieso die Anforderung erfüllt werden können. An der Stelle soll auch auf die Möglichkeiten von Simulations und Analysetools zurückgegriffen werden. Mit der Simulation können die Teilprobleme nicht nur entkoppelt betrachtet werden, sondern auch gezielt untersucht und illustriert werden. Es sollen aussagekräftige Anwendungssiutation konstruiert werden, die die Funktionsprinzipien der Algorithmen entweder in der Simulation oder Realität aufzeigen.

Aufbau der folgendne Arbeit (Reihenfolge muss man noch diskutieren)

* Infrastruktur
  + ROS-Netzwerk (Die frage ist wie viel wir über ROS-sprechen)
  + Simulationswerkzeug: Gazebo
  + Simulationswerkzeug: MATLAB
  + Visualisierungstool: Rviz
* ROS-Navigation-Stack Allgemeiner Aufbau
  + Zweistufiges Planungs konzept, lokal/global
  + Zusatzaspekt Positionsbestimmung
  + Zusatzaspekt vorhandene Karte
  + Gemeinsamer nenner stochastische Modelle
* Stochastische Modelle
  + Allgemeines blabla
  + Bewegungsmodell
  + Sensormodell
* Globaler Planer
  + Diskrete Planung
  + Depth/Breadth-First
  + Dijkstras Algortihmus
  + A\*
* Lokaler Planer
  + DWA
* Karten malen
* Lokalisierung

Bei dem navigation Stack gibt’s einmal die Möglichkeit zuerst eine Übersicht zu geben, was die Grundprinzipien sind etc.. Dann im Anschluss werden den Hauptunkten wie stochastische Modelle, Karte malen, Lokalisierung, globaler Planner, lokaler Planner einzelne Kapitel gewidmet, die jeweils von Mathe bis Parametrisierung oder Beispiel innerhalb ovn ROS/gazebo/MATLAB reichen. Dann wird das ganze wieder in einem Kapitel gesammelt, wo die Ergebnisse zusammengführt werdne. Dann würde ich die Infrastruktur aber auch vorneweg erklären, so ein bisschen ROS, wie die Roboter da rein passen, an welchen Stellen gazebo/MATLAb und co. Eingreifen.

Navigation-Stack obeflächlich, Übergang zu globaler Planer, weil der das einzig nicht stochastische ist, dann einführung in stochastische Modelle+kont Bayes-Filter, dann

Der Aufbau der folgenden Arbeit beginnt der Infrastruktur und Komponenten, deren gemeinsamen Nenner ROS darstellt, deren - für diese Aufgabe relevantesten - Aspekte kurz beleuchtet werden. Hier wird erklärt wie die Roboter über ROS miteinander, dem Master-PC und Diagnosewerkzeugen verbunden sind. Ebenso werden die Anylse und Diagnosewerkzeuge wie gazebo, MATLAB und Rviz erläutert und deren Vorteile diskutiert.

Im Anschluss folgt eine Übersicht des ROS-Navigation-Stack der im Anschluss näher erläutert wird. Zu Beginn wird lediglich auf die Grundprinzipien des Gesamtsystems und das Zusammenspiel der Komponenten eingegangen. Daraufhin folgt die Zerlegung in die einzlenen Bausteine, welche in den folgenden Kapitel separat im Detail untersucht und erklärt werden. Bei der Illustration der Alogrithmen wird auf die bekannten Werkzeuge zurückgegriffen, wodurch simultan die Konfiguration innerhalb des ROS-Framework durchgearbeitet wird. Die zentralen Bausteine sind die Lokalisierung, Planugnsalogrithmen und Kartenerstlelung. Nachdem diese im einzlenen erklärt undkonfiguriert wurden, folgt deren Zusammensetzung und Inbetriebnahme des Gesamtsystems. An dieser Stelle werden die Test-Cases für das Navigationskonzept diskutiert und erprobt.

**11.12.2017**

**Infrastruktur**

* Wofür?
  + Vier Roboter simultan benötigt Kommunikation
  + Navigation eines einzelnen Roboters setzt sich aus mehreren komplexen Alogrithmen zusammen, um das zu Debuggen braucht man eingriffsmöglihckeiten.
  + Deswegen braucht man eine Infrastruktur über die die Roboter kommunizieren können, aber auch exterene Tools relevanten Daten abgreifen können.
  + Diese Aufgabe erfüllt für uns ROS.
* ROS-Netzwerk (Die frage ist wie viel wir über ROS-sprechen)
  + Rechner Aufbau, vier Turtlebots/einer, ein Master-Mini-PC, eventuell weitere Desktop-PCs
  + Verbindung über Ros, roscore wird auf dem Master-PC ausgeführt.
  + ROS-Kommunikation über Nachrichten
  + Nachrichten werden unter topics veröffentlich
  + Mit Analysewerkzeug bestimmte topics subscriben
  + Andersrum topics des Turtlebot durch Simulation ersetzen
* Simulationswerkzeug: Gazebo
  + Idee von Paper mit P3-DX-Roboter
  + Gibt auch Unterstützung für Turtlebot
  + Unser Lasersensor nicth exakt abgebildet
  + Custom-Implementierung möglich, aber zeitaufwendig, verweis auf Quellen
  + Dadurch Nachteil nur begrenz tmit Realität vergleichbar
  + Entkopplungsmöglichkeiten durch Simulation
  + Konfigurationsmöglihckeit des Navigation Stack
* Simulationswerkzeug: MATLAB
  + MATLAB Robotics Toolbox macht MATLAb zur ROS-Node
  + Vorteli von MATLAB ist, dass es für Algorithmen angenehmere Sprache als C++ oder Python ist
  + Bietet Visualisierungsmöglichkeiten für Karten und auch Plots
  + Beispielsweise implementieren wir simple Kartographierung in MATLAB
  + Simple Anwedungsfälle, wie z.B. diskrete Planung können auch vollständig in MATLAb „simuliert“ werden
* Visualisierungstool: Rviz
  + Visualisierungs tool, das in ROS integriert ist
  + Bietet sehr komfortable Schnittstelle zu ROS, weil von Community dafür gemacht
  + Im Bereich von Navigation sind da bereits sämtliche Konfiguration vorhanden un auf den Turtlebot abgestimmt

Die Idee für die Kombination von Gazebo + ROS für die Simulation stammt aus dem einen Paper, wo mit Roboter P3-DX gearbeitet wurde. Vorteile sind

**Wofür?** In der Endvorstellung des Projektes navigieren vier Roboter simultan durch denselben Raum, woran recht leicht ersichtlich wird, dass ein Weg zum Datenaustausch zwischen den Robotern geschaffen werden muss. Auch eine zentrale Steuereinheit soll in der Lage sein, alle Roboter anzusprechend und zu dirigieren. Selbst ein einzelner Roboter stellt bereits einige Anforderungen an die Kommunikationsstruktur: Da eine erfolgreiche navigation sich aus dem Zusammenspiel merherer komplexer Algorithmen ergibt, müssen extern Analysetools in der Lage sein, Daten, die zwischen den Komponenten ausgetauscht werden, abzugreifen um diese visualisieren oder auswerten zu können. Diese Form des Debuggings ist unerlässlich, um das Verhalten der roboter nachvollziehen zu können. In diesem Projekt erfüllt ROS diese Anforderungen, weshalb in den folgenden Abschnitten das Kommunikationskonzept von ROS erläutert wird. Auch die Simulations- und Analysewerkzeuge die in der Arbeit zum Einsatz kommen werden vorgestellt und deren Eingreifen in ROS erläutert.

**ROS-Netzwerk** Jeder Turtlebot ist mit einem Mini-PC ausgestattet, auf denen Ubuntu 14.04 und ROS-Indigo installiert sind. Als zentrale Steuereinheit des Roboterverbundes fungiert ein weiterer Mini-PC, der über WLA-Netzwerk mit den Roboter-PCs verbunden ist. Das private Netzwerk wird mittels eines TP-Link Router verwaltet, in dem sich weitere Entwicklungsrechner einlinken können.

Auf dem Master-PC wird der roscore ausgeführt, über die Kommunikation abläuft. Als Kommunikationsmittel werden in ROS-Nachrichten verwendet, die jeweils unter einer so genannten Topic veröffentlicht werden. Beispiels versendet der Lasersensor zyklisch eine Nachricht des Typs snesor\_msg/laser\_scan unter der Topic /scan. Alle Nodes, die an den Sensordaten interessiert sind, subscriben die Topic. Indem Analysewerkzeuge wie MATLAB relevante Topics mithören, können die zugehörigen Daten aufgezeichnet und visualisiert werden. Umgekehrt ist es auch möglich den Turtlebot vollständig durch eine Simulation zu ersetzen. Gazebo veröffentlicht alle topics, die für gewöhnlich von dem Turtlebot versendet werden, wodruch dieser ersetzt wird. Die restlichen Bestandteile des netzwerkes bleiben dabei erhalten, weshalb der Wechsel zwischen Realität und Simulation problemlos funktoiniert.

**Simulationswerkzeug: Gazebo** Die Idee gazebo in Verbindung mit ROS für die Simulation zu verwenden, entstammt der Arbeit […]. Dort wurde unter einer ähnlichen Konfiguration ein P3-DX-Roboter verwendet. Der primäre Vorteile dieser Simulationsinfrastrutkur besteht darin, dass die Simulation ohne Anpassung mit dem Roboter ausgetauscht werden können. Gazebo bietet auch für die Turtlebots vollständige Unterstützung, das heißt es existieren Integration des Turtlebots in gazebo frei zur Verfügung. Allerdings existiert keine Implementation der hier verwendeten Lasersensoren, SICK-Tim-551, weshalb die Sensorik nicht exakt abgebildet werden kann. Prinzipiell können die Sensoren eingepflegt werden, was jedoch im Rahmen dieser Arbeit aus zeitlcihen Gründen nicht erfolgte. Es folgt als Nachteil, dass die Simulation nur beschränkte Schlüsse auf die Realität zulässt. Nichtsdestotrotz ergeben sich zwei wichtige Vorteile. Einerseits können die verschiedenen Konfiguration des Ros-Navigation-Stack anhand der Simulation erprobt und auf ihre Richtigkeit überprüft werden, bevor sie auf reale Anwendungsszeanrien übertragen werden. Andererseits können im Rahmen der Simulation die Teilkomponenten des Navigation-Stack vollkommen entkoppelt werden. Beispiels hängen die Ergebnisse der Navigations-Algorithmen stark von der Qualität der Lokalisierung ab, weshalb die Planungs- und Lokalisierungsproblematik in einem realen Umfeld nicht getrennt untersucht werden können. In der Simulation können Positionsdaten allerdings unmittelbar abgegriffen und der Navigation zur Verfügung gestellt werden. Insofern stellt gazebo ein mächtiges Werkzeug für die schrittweise Inbetriebnahme des Navigation-Stacks dar.

**Simulationswerkzeug: MATLAB** MATLAB kann im Rahmen der Robotics Toolbox ebenfalls als ROS-Node betrieben werden, wodurch eine Schnittstelle zwischen MATLAB und ROS entsteht. Hier bringt MATLAB den Vorteil mit sich, dass es für die Implementierung von Algorithmen oftmals angenehmer als C++ oder Python ist. Insbesondere die Visualisierungsmöglichkeiten für Karten und Plots erweisen sich als mächtige Werkzeuge bei der Implementierung und Erprobung von Algorithmen. Beispielsweise wird im Rahmen dieser Arbeit ein simpler Kartographierungsalgorithmus mittels MATLAB realsiert und erprobt. Außerdem können simple Anwedungsszenarien, wie sie z.B. im Rahmen der diskreten Planung auftreten, vollständig mit MATLAB simuliert werden.

**Visualisierungstool: Rviz:** Als letztes Werkzeug sei an dieser Stelle Rviz genannt. Hierbei handelt es sich um ein Werkzeug zur Visualisierungen im Bereich von mobilen Roboter. Mithilfe von Rviz können sowohl Karten als auch Robter und Pfade graphisch dargestellt werden. Ein großer Vorteil von Rviz besthet darin, dass es für die kombinierte anwendung mit ROS entwickelt wurde. So stehen vollständige Konfiguration von Rviz für die autonome Navigation zur Verfügung, die Out-of-the-Box genutzt werden können.

**12.12.2017**

Navigation-Stack Übersicht

* Was ist das? Wofür ist es gemacht?
  + Navigation-Stack ist das Ergebnis von mehreren ROS-Paketen, die zu einer vollständigen autonomen Navigation führen
  + Navigation-Stack ist somit Teil von ROS
* Zweischichtiger Aufbau
  + Unterteilung in lokalen und globalen Planer
  + Globaler Planer erstellt mit Karte, diskrete Darstellung der Umgebung, es resultiert ortsdiskreten Pfad
  + Lokaler Planer wird nächster Zielpunkt des globalen Pfades übergeben, soll gschwindigkeiten berechnen, die Stellvektor sind, um den Punkt zu erreichen
  + Bei der lokalen Planung wird die Karte, der Zielpunkt und Sensorwerte beachtet,~
  + Erinnert an menschliches Verhalten, Beispiel Autofahrt:
    - Zuerst wird anhand von einer Karte eine Route geplant, was einem ortsdiskreten Plan entspricht, Auf Autobahn A8 bis Ausfahrt 373
    - Zu dem Zeitpuntk macht man sich keine Gedanken übe rmögliche Spurwechsel, weil ein LKW vor einem fährt
    - Während der Fahrt mach tman sich darüber gedanken, der lokale Planer übernimmt hier
    - Aber treten unerwartete Hindernisse auf, wird auch die globale Planung überdacht. Beispielsweise überlegt man sich doch früher abzufahren, wenn auf einmal Stau gemeldet wird.
    - Deshalb wird die globale Planung in ROS zyklisch wiederholt, mit langsamer Fa als lokal und es werden Sensordaten beachtet.
    - Beispiel wenn ein Hindernis vor dem Roboter steht, das nicht auf der Karte verzeichnet ist, wird das vom Lasersensor erfasst. Der Planer projiziert den Messwert in die Karte, interpretiert ihn als ein Hidnernis und plant deswegen einen Pfad um den Gegenstand herum.
* Unterstüzungsfunktionen
  + Die Planer brauchen recht viele Information, die dem Navigation-Stack von außen zur Verfügung gestellt werden müssen.
  + Ein Beispiel ist die Karte, ROS-paket map\_server, der versendet die als ROS-Nachricht unter der topic /map
  + Die metrische Umgebungskarte wird mittels costmap\_2d in so genannte Kostenkarten transformiert, die jeder Zelle einen Aufwand zuweisen. Bei der Transformation von gewöhnlicher zu Kostenkarte können und werden zusätzlich Sensorwerte beachtet.
  + Sowohl globaler als auch lokaler Planer erhalten ihre eigene Kostenkarte, in denen sie dann optimale Pfade bzw. Trajektorien planen.
  + Außerdem muss die Position des Roboter bekannt sein, wofür Lokalisierungsmechanismen nötig sind. Es kann entweder ausschließlich mittels Odometriedaten oder zusätzlichen Lokalisierungsalgorithmen wie der AMCL gearbeitet werden
* Weiteres Vorgehen:
  + Wir beginne bei dem globalen Planer, weil der erster Schritt ist und auch das einzige deterministische Konzept
  + Dann holen wir erstmal aus und klären die Grundgedanken und –prinzipien der stochastischen Modell in de rRobotik
  + Zur Illustration zeigen wir ein Bewegungs- und Messmodell, die wir später auch für Lokalisierung und Navigation benötigen
  + Dann beginnen wir mit Bayes-Filter, die wir für Kartenmalen und Lokalisierung brauchen
  + Als erstes einfaches Beispiel machen wir Kartenmalen mit bayes for binary state, woran wir auch gleich mal Gazebo und MATLAb illustrieren können
  + Im Anschluss führen wir die Filter weiter wie wir sie für die Lokalisierung benötigen, das heißt einmal diskret/Histogram für Grid-Localization und dann Partikel-Filter für AMCL, hier müssen wir auch noch die beiden extenden Varianten diskutieren
  + Zuletzt kommt dann DWA für den lokalen Planer

**Was ist das? Wofür ist es da?** Der ROS-Navigation-Stack ist das Ergebnis mehrerer zusammengefügter ROS-Pakete für die autonome Navigation von mobilen Robotern. Die Kernaufgabe des Navigation-Stack besteht darin, einen Weg zwischen einer Start- und Zielposition zu planen und abzufahren, wofür auf eine Karte der Umgebung zurückgegriffen wird.

**Zweischichtiger Aufbau:** Das Planungskonzept des Navigation-Stack verfolgt einen zweischichtigen Ansatz, wobei zwischen einem globalen und einem lokalen Planer differenziert wird. Im ersten Schritt nutzt der globale Planer die vorliegende Karte, um einen ortsdiskreten Pfad zwischen Start- und Zielposition zu bestimmen. Der nächste Punkt des Pfades wird dem lokalen Planer übergeben, der wiederum Geschwindigkeiten berechnen soll, die den Roboter zu dem lokalen Ziel führen. Bei der lokalen Planung werden neben dem aktuellen Punkt des globalen Pfades auch die Karte und aktuelle Sensorwerte beachtet. Dieses Vorgehen erinnert an das menschliche Verhalten. Vor einer Autofahrt planen wir anhand einer Karte die Route zum Ziel, was der Aufgabe des globalen Planers entspricht. Hier handelt es sich um eine ortsdiskrete Darstellung, da einzelne Punkte wie Auf- und Abfahrten zu Autobahnen genügen, um den Weg zu spezifizieren. An diesem Zeitpunkt machen wir uns auch noch keine Gedanken, ob wir die Spur wechseln sollten, um den nächsten LKW zu überholen. Derartige Probleme fallen in den Aufgabenbereich des lokalen Planers. Sobald wir uns auf der Reise befinden, und das nächste lokale Ziel, wie zum Beispiel die kommende Autobahnabfahrt, anstreben, planen wir konkrete Überholmanöver. Dennoch können unerwartete Hindernisse, die sich erst während der Fahrt auftun, unsere globalen Pläne über den Haufen werfen. Beispielsweise ziehen wir im Antlitz eines unerwarteten Staus gerne eine Alternativroute in Erwägung. Diese Form der Planänderung wird im Navigation-Stack indem realisiert, dass einerseits die globale Planung zyklisch wiederholt wird, andererseits werden dabei auch aktuelle Sensorwerte beachtet. So werden Distanzmessungen in die Karte projiziert und als zusätzliche Hindernisse interpretiert, die vom Pfad möglichst vermieden werden sollen.

**Unterstützungsfunktionen:** Wie am Beispiel von Karte und Sensordaten ersichtlich wird, benötigt der Navigation-Stack weitere Information, die im von außen zur Verfügung gestellt werden müssen. Diese Aufgaben werden mithilfe von weiteren ROS-Paketen erfüllt. Als erstes sei das Paket map\_server genannt, der die im Voraus aufgezeichnete Karte als ROS-Nachricht des Typs map\_msgsasdf unter der Topic /map veröffentlicht. Die Umgebungskarte wird mithilfe des ROS-Paket costmap\_2d in eine so genannte Kostenkarte, die jeder Zelle einen Aufwand zuordnet. Bei der Transformation können zusätzliche Information wie z.B. Sensorwerte beachtet werden, wodurch auch Hindernisse, die nicht auf der Karte verzeichnet sind, die Planung beeinflussen. Innerhalb des Navigation-Stacks wird sowohl für den lokalen als auch den globalen Planer eine separate Kostenkarte generiert.

Als letzte wichtige Zusatzinformation ist die Position des Roboters zu nennen, die über zwei relevante Wege bestimmt werden kann. Der einfachste Ansatz nutzt die Odometriedaten, um die Position des Roboters zu berechnen. Die AMC-Lokalisierung stellt einen ausgreifteren Ansatz dar, der stochastische Bewegungs- und Messmodelle nutzt, um anhand bekannten Informationen die Position des Roboters schätzt. Diese Lokalisierungsvariante wird in dem ROS-Paket amcl implementiert, das simultan zur Navigation ausgeführt wird.

**Weiteres Vorgehen:** Der nächste Schritt besteht jetzt darin, die einzelnen Teile des Navigation-Stack getrennt zu betrachten und deren Algorithmen nachzuvollziehen. Als erstes wird der globale Planer untersucht, da er einerseits den Ausgangspunkt der Navigation darstellt, andererseits handelt es sich auch um den einzigen deterministischen Algorithmus, wodurch er eine Sonderstellung einnimmt. Die restlichen Algorithmen basieren auf stochastischen Konzepten, weshalb zunächst die Rolle einer stocahstischen Weltanschauung in der Robotik diskutiert wird. Zur Illustration wird sowohl ein stochastisches Bewegungs- als auch Messmodell hergeleitet, die später bei der Lokalisierung und Bewegungsplanung wieder zum Einsatz kommen. Den Ausgangspunkt der meisten Algorithmen stellt das Bayes-Filter dar, das anschließend hergeleitet wird. Als erstes Illustrationsbeispiel dient ein simpler Kartographierungsalgorithmus, um im nächsten Schritt fortgeschrittener Anwendungen im Rahmen der Lokalisierung zu betrachten. Als letzter Bestandteil bleibt der lokale Planer, der nach dem Dynamic Window Approach arbeitet, der zuletzt erläutert wird.

**Weiteres Vorgehen**

Durchsicht von dem alten globalen Planer ist durch, ein paar Anmerkungen zu Formulierunge und das Beispiel sollte man noch ein bisschen ausführlicher erklären, außerdem fehlt noch ein Baumbild.

Jetzt würde ich erstmal mit dem übertragen und durchgehen der bisherigen Dokumentation weiter machen. Und wenn ich einen Gesamteindruck habe, überlegen wie es im Detail weitergeht.

Für Gesamteindruck bin ich gerade jetzt ein wenig zu faul, habe die Ergebnisse mit Wietzke besprochen. Ziel ist jetzt zwei Turtlebots in Betrieb zu nehmen.

**13.12.2017**

Das nächste Ziel ist es zewi Turtlebots parallel zu betrieben, dafür muss die namespace Problematik geklärt werdne. Der erste und einfachste Ansatz ist, das vollständige Programm zweimal auszuführen. Das heißt für jeden Turtlebot läuft auch auf dem Master-PC eine separate RViz-Instanz. Dafür müssen unter ROS namespaces verwendet werden, was bisher auch schon für teleop funktoiniert. Das Problem entsteht bei Rviz, der nicht mehr weiß welche topics zu ihm gehören. Da könnte man jetzt theoretisch eine config händisch anpassen, aber das ist dann nich variable parametrisierbar.

Noch ein Problem werden die Transformationen, die müssen ja auch ihren eigenen Namensraum erhalten. Hierfür können Nodes mit dem argument „tf\_prefix“ parametrisiert werden, das wir auf den Namensraum setzen können.

Auf slave-Seite wird /tf von dem robot\_state\_publisher und mobile\_base\_nodelet\_manager gepublished. Die werden in den launch-files robot.launch (robot\_state\_publisher) und kobuki/mobile\_base.launch (mobile\_base\_nodeleet\_manager) gestartet.

Auf master-Seite wird /tf von base\_link\_footprint\_tf\_broadcaster und amcl veröffentlich. Die hab ich jetzt auch noch gefixed, jetzt ist der tf-tree wegzusammenhängend, allerdings fehlt da jetzt die Karte vollständig. Die Transformation von Karte zu Odometrie wird für gewöhnlich von amcl veröffentlicht.

**14.12.2017**

Heute geht es immer noch darum amcl mit den namespaces zum Laufen zu bringen. Eigentlich geht es ja eher darum, den ganzen Navigatin-Stack im namespace zum Laufen zu bringen, aber gestern konnte das Problem doch auf amcl eingegrenzt werden. Heute habe ich zuerst die Paramter gecheckt: Da kann man die IDs der odometrie, karten und base frames einstellen, wo ich auch den Namespace vorgestellt habe.

Der Vergleich in der namespacelosen Nvaigation zeigt, dass AMCL dort mehr topics in und outputted, die fehlden sind:

Topic: /initialpose, inbound, from rviz

Topic: /particlecloud, outbound, to rviz

Insbeondere die erste könnte das Problem auslösen, da RViz unter falscher topic das published. Vorgehen ist jetzt, alte Navigation auszuführen, die Configuration von Rviz zu ändern, abzuspeiche rund die für die neue Navigation nutzen. Was auch nicht hilft.

Weitere Änderungen:

Ln 326, Fixed Frame: TB/map

Die turtleboys haben auch move base auf den Turtlebot-PCs laufen lassen. Dann läuft auf dem Master nur noch der map server und ein custom konfigurierter RViz. Bei den slave launches mappen sie zusätzlich die map in den namensraum des jeweilgien Slaves. Der Aufbau ist eigentlich besser, dann müssen auf dem Master nicht für jeden Slave ein extra Prozess laufen, weswegen als erstes namespace los dahin geändert wird.

Es gibt auf dem master: master\_navigation\_standalone.launch

Es gibt auf dem slave : slave\_navigation\_standalone.launch

Und später die namespace-Varianten:

Auf master: master\_navigation\_standalone\_namespace.launch

Auf slave: slave\_navigation\_standalone\_namespace.launch

Die move-base Konfiguration war auf dem Master mit vielen custom files für Paramter etc. Die kopiere ich jetzt alle auf den slave, wo ich die includes anpasse. Langfristig ist das aber ziemlich hässlich, weil zweigleisige Lösung. Deswegen sollte man entweder einen ordentliches Versionsmanagement zwischen den PCs einführen oder di ealt evariante über Bord werfen.

Die Variante ohne namespaces ist erledigt und funktioniert auch.

**19.02.2018:** Endspurt. Bei der AMCL-Problematik haben wir uns darauf geeinigt, dass eine Lösung für zwei Roboter hardkodiert wird. Das heißt es werden launch-files angelegt, die jweils für Roboter 1 und Roboter 2 spezialisiert werden. Jeder einzelnen Nachricht und topic wird ein Präffix zugeordnet, sodass eine Unterscheidung zwischen den Robotern erfolgen kann. Nicht gut, aber geht.

Ansonsten steht noch Dokumentation an, hier gilt es vor allem auf die ROS-Themen einzugehen, damit dieses Problem vernünftig dokumentiert ist.

Die ToDos auf Roboterseite sind:

- Karte aufzeichnen

- Funktionierende Navigation in Betrieb nehmen

- Überarbeitete Navigation erstellen und an Roboter 1 in Betrieb nehmen

- Überarbeitete Navigation erstellen und an Roboter 2 in Betrieb nehmen

- Beide Roboter parallel navigieren lassen

Daran mach ich mich morgen, wobei ich die Gelengheit gleich nutze, um eine vernünftige Dokumentation/Anleitung von den Prozessen zu schreiben. Dabei kann ich auch auf die launch-files eingehen.

Die Dokumentation zu den theoretischen Aspekten der Navigation ist eigentlich schon ziemlich fortgeschritten. Insbesondere, weil die schon 40 Seiten lang ist und ich den Kack alleine schreibe.

Heute kann ich die praktischen Teile schon mal angehen und vorbereiten. Für eine ordentliche Ausführung brauche ich die Funktastatur, aber dann kann ich das morgen ziemlich zügig durcharbeiten.

Aufbau/Konfiguration Launch-Files:

Es wird auf dem Master ein Projekt mit dem Namen EML\_Navigation\_Master angelegt, in dem launch files für die Kartenaufzeichnung, Basisnavigation, Roboter1Navigation, Roboter2Navigation liegen. Die Dateien heißen

EML\_Mapping\_Master.launch

EML\_Navigation\_Master.launch

EML\_Navigation\_Robot2\_Master.launch

EML\_Navigation\_Robot4\_Master.launch

Die Slaves besitzen ein – bei beiden Slaves identisches – Projekt, das den Namen EML\_Navigation\_Slave trägt. Hier sind ebenfalls files für mapping, navigation und roboterspezfisische Navigation angelegt.

EML\_Hardware\_Init\_Slave.launch

EML\_Mapping\_Slave.launch

EML\_Navigation\_Slave.launch

EML\_Navigation\_Robot2\_Slave.launch

EML\_Navigation\_Robot4\_Slave.launch

**Beschreibung des Aufbaus:**

In dieser Arbeit wird die Navigation von maximal zwei Robotern zurselben Zeit betrachtet. Jeder der Roboter ist mein einem PC ausgestattet, die über WLAN mit einem TP-Link verbunden sind, der als Acess-Point des Netzwerks konfiguriert ist. Der Roboterverbund ist als Master-Slave-Konzept konzipiert, wobei die Roboter bzw. deren PCs als Slave agieren. Es ist ein weiterer Mini-PC mit dem Router verbunden, der als Master agiert. Der Master-PC ist zusätzlich mit Tastatur, Maus und Bildschirm verbunden und stellt somit die Steuereinheit des Systems dar. Daher stammt auch die Bezeichnung Master-PC, da von diesem die Befehlskette bedient wird. Die Slave-PCs, welche sich auf den Robotern befinden, nehmen die Befehle entgegen, arbeiten also als Slaves. Insofern handelt es sich bei dem Verbund um keinen Master-Slave-Aufbau in dem Sinne eines speziellen Protokolls.

Das Netzwerk läuft in dem IP-Adressbereich 192.168.0.x, wobei die Adressen des Master- und der Slave-PCs statisch zugewiesen wurden. Der Master-PC ist unter der Adresse 192.168.0.100 erreichbar; Roboter R2 unter der Adresse 192.168.0.102; Roboter R4 unter der Adresse 192.168.0.104. Außerdem ist es möglich, dass sich weitere Rechner mit dem Netzwerk verbinden, wodurch auf sämtliche Daten des ROS-Netzes zugegriffen werden kann. Somit können auch Entwicklungstools wie MATLAB verwendet werden, um die Daten abzugreifen und weiterzuverarbeiten.

Das Netzwerkt läuft unter dem Namen „EML\_Turtlebot\_NET“ mit dem Passwort: „turtlebot“. Die Konfiguration der IP-Adressen läuft wie bereits erwähnt statisch ab, was aus der Arbeit [turtleboys, S. 23] übernommen wurde und dort im Detail nachgelesen werden kann.

(Die Konfiguration hat funktioniert, beide Turtlebots verbinden automatisch nach Booten auf das Netzwerk, Master kann mit ihnen kommunizieren und sich mittels ssh verbinden)

Die Nutzer auf den slaves heißen jeweils „turtlebot“ und das Passwort lautet ebenfalls „turtlebot“, womit eine ssh-Verbindung und -Zugriff hergestellt werden kann.

Versuchsdurchführung Kartenaufzeichnung

Verläuft 1:1 nach der Anleitung von der Vorgruppen, wir übernehmen die Dateien, pflegen sie in den neuen WS ein und geben Probs.

**23.2.18:**

Wir beginnen damit das wir die alte Variante ausprobieren, dann die files in das neue System kopieren.

Alte Anleitung:

1. [Auf Master] roscore

2. [Slave] roslaunch turtlebot\_chor\_navigation slave\_createMap.launch

3. [Master] roslaunch turtlebot\_chor\_navigation master\_createMap.launch

4. [Master] roslaunch turtlebot\_teleop keyboard\_teleop.launch

5. [Master] rosrun map\_server map\_saver

Neue Anleitung:

**Remote-Control Turtlebot**

Auf dem Slave besteht als erstes eine Initialisierungs-Launch-Datei, die genutzt sämtliche Hardware an den Punkt bringt, an dem der Roboter über teleop ferngesteuert werden kann.

**EML\_Hardware\_Init\_Slave.launch**

Im aller ersten Schritt werden die beiden Roboter in Betrieb genommen, wofür die Fernsteuerung der beiden Geräte als Demonstration dienen soll. Auf den Slaves wird dafür die Launch-Datei „EML\_Hardware\_Init\_Slave.launch“ angelegt, die die vorhandene Hardware initialisiert, sodass der Roboter über die entsprechenden Steuerbefehle bewegt werden kann.

<!-- 23.02.2018; Michael Meindl; Basisinitialisierung der Hardware -->

<launch>

<!-- Basis-Initialisierung -->

<include file="$(find turtlebot\_bringup)/launch/minimal.launch"/>

<!-- Initialisierung des SICK-TIM551 -->

<param name="robot\_description" command="$(find xacro)/xacro.py '$(find

sick\_tim)/urdf/example.urdf.xacro'"/>

<include file="$(find turtlebot\_chor\_navigation)/launch/include/sick\_tim

551\_2050001\_timefix.launch"/>

</launch>

In der Launch-Datei wird zuerst eine Basis-Initialisierung durchgeführt, wofür die Datei „minimal.launch“ aus dem offziellen Packet *turtlebot\_bringup* [12] inkludiert wird. Als Zweites wird der SICK-Laserscanner initialisiert, wofür eine launch-Datei des Vorgängerprojekts [13] genutzt wird.

Um nun den Turtlebot remote zu steuern wird die Launch-Datei auf dem Slave ausgeführt:

roslaunch EML\_Navigation\_Slave EML\_Hardware\_Init\_Slave.launch

Die Eingabe der Steuerbefehle erfolgt auf dem Master-PC, wofür auf das Packet *turtlebot\_teleop* [14] zurückgegriffen wird. Zum Start wird der Befehl

roslaunch turtlebot\_teleop keyboard\_teleop.launch

auf dem Master-PC ausgeführt. Nun kann der Roboter mittels der Tastatur bewegt werden.

Die Hardware-Init-Datei auf dem Slave wird in den weiteren Launch-Datei wiederverwendet.

**Kartenaufzeichnung**

**Slave**

Für die Kartenaufzeichnung wird wieder auf die vorhandenen Dateien des Vorgängersprojekts [13] zurückgegriffen. Zunächst wird die HW mittels dem zuvor besprochenen Launch-File initialisiert. Im Anschluss werden die Launch-Files für die Kartenaufzeichnung des Packets *hector\_mapping* [15] verwendet.

<!-- 23.02.2018; Michael Meindl; Slave-Setup fuer Kartenaufzeichnung, basiert auf turtlebot\_chor\_navigation/slave\_createMap.launch -->

<launch>

<!-- Hardware-Initialisierung -->

<include file="$(find EML\_Navigation\_Slave)/launch/EML\_Hardware\_Init\_Slave.launch" />

<!-- Hector-Mapping -->

<node name="hector\_mapping" pkg="hector\_mapping" type="hector\_mapping" output="screen">

<!--> <param name="map\_frame" value="map\_link"/> -->

<param name="base\_frame" value="base\_link"/>

<param name="odom\_frame" value="base\_link"/>

<param name="map\_resolution" value="0.05"/>

<param name="map\_size" value="2048"/>

<param name="map\_start\_x" value="0.5"/>

<param name="map\_start\_y" value="0.5"/>

<param name="map\_update\_distance\_thresh" value="0.4"/>

<param name="map\_update\_angle\_thresh" value="0.02"/>

<param name="map\_pub\_period" value="0.05"/>

<param name="map\_multi\_res\_levels" value="2"/>

<param name="update\_factor\_free" value="0.4"/>

<param name="update\_factor\_occupied" value="0.9"/>

<param name="laser\_min\_dist" value="0.4"/>

<param name="laser\_max\_dist" value="10"/>

<param name="laser\_z\_min\_value" value="-1.0"/>

<param name="laser\_z\_max\_value" value="1.0"/>

<param name="pub\_map\_odom\_transform" value="true"/>

<param name="output\_timing" value="false"/>

<param name="scan\_subscriber\_queue\_size" value="1"/>

<param name="pub\_map\_scanmatch\_transform" value="true"/>

<param name="pub\_map\_scanmatch\_transform" value="true"/>

<param name="tf\_map\_scanmatch\_transform\_frame\_name" value="scanmatcher\_frame"/>

</node>

<include file="$(find hector\_geotiff)/launch/geotiff\_mapper.launch"> </include>

<node pkg="tf" type="static\_transform\_publisher" name="map\_nav\_broadcaster" args="0.12 0 0 0 0 0 base\_link laser 100"/>

</launch>

**Master**

Auf dem Master wird die Launch-Datei *EML\_Mapping\_Master.launch* angelegt, die inhatlich wieder aus dem Vorgängerprojekt [13] übernommen ist. In dem File wird lediglich Rviz gestartet, um die Ergebnisse der Kartographierung zu visualisieren.

<!-- 23.02.2018; Michael Meindl; basiert auf turtlebot\_chor\_navigation/master\_createMap.launch -->

<launch>

<!-- Starte Rviz, um Karte anzuzeigen. -->

<param name="robot\_description" command="$(find xacro)/xacro.py '$(find sick\_tim)/urdf/example.urdf.xacro'" />

<node pkg="rviz" type="rviz" name="rviz" args="-d $(find turtlebot\_chor\_navigation)/rviz\_cfg/rviz\_sick.rviz"/>

</launch>

**Anleitung**

1. [master] roscore

2. [slave] roslaunch EML\_Navigation\_Slave EML\_Mapping\_Slave.launch

3. [master] roslaunch EML\_Navigation\_Master EML\_Mapping\_Master.launch

4. [master] roslaunch turtlebot\_teleop keyboard\_teleop.launch

5. [master] rosrun map\_server map\_saver

**Navigation: Einzelner Roboter**

**Slave**

Auf dem Slave wird in der launch-Datei *EML\_Navigation\_Slave.launch* lediglich die HW initialisiert, wofür die Datei *EML\_Hardware\_Init\_Slave.launch* inkludiert wird.

<!-- 23.02.2018; Michael Meindl -->

<launch>

<include file="$(find EML\_Navigation\_Slave)/launch/EML\_Hardware\_Init\_Slave.launch" />

</launch>

**Master**

Auf dem Master wird die Launch-Datei *EML\_Navigation\_Master.launch* angelegt, in der sämtliche Algorithmen der Navigation ausgeführt werden.

<!-- 23.02.2018; Michael Meindl -->

<launch>

<!-- Map-Server -->

<arg name="map\_file" default="$(find EML\_Navigation\_Master)/maps/test\_map.yaml"/>

<node name="map\_server" pkg="map\_server" type="map\_server" args="$(arg map\_file)"/>

<!-- AMC-Localization -->

<arg name="3d\_sensor" default="$(env TURTLEBOT\_3D\_SENSOR)"/>

<arg name="initial\_pose\_x" default="0.0"/>

<arg name="initial\_pose\_y" default="0.0"/>

<arg name="initial\_pose\_a" default="0.0"/>

<arg name="custom\_amcl\_launch\_file" default="$(find turtlebot\_navigation)/launch/includes/amcl/$(arg 3d\_sensor)\_amcl.launch.xml"/>

<include file="$(arg custom\_amcl\_launch\_file)">

<arg name="initial\_pose\_x" value="$(arg initial\_pose\_x)"/>

<arg name="initial\_pose\_y" value="$(arg initial\_pose\_y)"/>

<arg name="initial\_pose\_a" value="$(arg initial\_pose\_a)"/>

<arg name="odom\_frame\_id" value="odom"/>

<arg name="base\_frame\_id" value="base\_footprint"/>

<arg name="global\_frame\_id" value="map"/>

<!-- arg name="scan\_topic" value="$(arg namespace)/scan"/-->

<!-- param name="tf\_prefix" value="$(arg namespace)"/-->

<arg name="use\_map\_topic" value="true"/>

</include>

<!-- Static Transform from base\_footprint to base\_link -->

<node pkg="tf" type="static\_transform\_publisher" name="base\_link\_footprint\_tf\_broadcaster" args="0 0 .5 0 0 0 1 base\_footprint base\_link 10

0"/>

<!-- Rviz -->

<node name="rviz" pkg="rviz" type="rviz" args="-d $(find turtlebot\_rviz\_launchers)/rviz/navigation.rviz"/>

<!-- move\_base -->

<include file="$(find EML\_Navigation\_Master)/launch/include/eml\_move\_base.launch"/>

</launch>

In der Datei wird zunächst die AMC-Lokalisierung gestartet, wofür die Standardparamter des ROS-Packets *amcl* [16] verwendet werden. Des wetieren wird Rviz gestartet, um die Karte, Position des Roboters sowie die Zielposition zu visualisieren. Als Konfiguration wird die Datei *navigation.rviz* des ROS-Packets *turtlebot\_rviz\_launchers* [17]. Zuletzt wird das *move\_base [18]* Packet gestartet, indem die letztendliche Navigation berechnet wird. Als Konfigurationsdatei dient *eml\_move\_base.launch*, die genutzt wird, um move\_base zu parametrisieren.

<!-- Custom-Launch file to start move-base with custom paramters-->

<launch>

<include file="$(find turtlebot\_navigation)/launch/includes/velocity\_smoother.launch.xml"/>

<include file="$(find turtlebot\_navigation)/launch/includes/safety\_controller.launch.xml"/>

<arg name="odom\_frame\_id" default="odom"/>

<arg name="base\_frame\_id" default="base\_footprint"/>

<arg name="global\_frame\_id" default="map"/>

<arg name="odom\_topic" default="odom" />

<arg name="laser\_topic" default="scan" />

<arg name="custom\_param\_file" default="$(find turtlebot\_navigation)/param/dummy.yaml"/>

<node pkg="move\_base" type="move\_base" respawn="false" name="move\_base" output="screen">

<rosparam file="$(find EML\_Navigation\_Master)/param/eml\_costmap\_common\_params.yaml" command="load" ns="global\_costmap" />

<rosparam file="$(find EML\_Navigation\_Master)/param/eml\_costmap\_common\_params.yaml" command="load" ns="local\_costmap" />

<rosparam file="$(find turtlebot\_navigation)/param/local\_costmap\_params.yaml" command="load" />

<rosparam file="$(find turtlebot\_navigation)/param/global\_costmap\_params.yaml" command="load" />

<rosparam file="$(find EML\_Navigation\_Master)/param/eml\_dwa\_local\_planner\_params.yaml" command="load" />

<rosparam file="$(find turtlebot\_navigation)/param/move\_base\_params.yaml" command="load" />

<rosparam file="$(find turtlebot\_navigation)/param/global\_planner\_params.yaml" command="load" />

<rosparam file="$(find turtlebot\_navigation)/param/navfn\_global\_planner\_params.yaml" command="load" />

<!-- external params file that could be loaded into the move\_base namespace -->

<rosparam file="$(arg custom\_param\_file)" command="load" />

<!-- reset frame\_id parameters using user input data -->

<param name="global\_costmap/global\_frame" value="$(arg global\_frame\_id)"/>

<param name="global\_costmap/robot\_base\_frame" value="$(arg base\_frame\_id)"/>

<param name="local\_costmap/global\_frame" value="$(arg odom\_frame\_id)"/>

<param name="local\_costmap/robot\_base\_frame" value="$(arg base\_frame\_id)"/>

<param name="DWAPlannerROS/global\_frame\_id" value="$(arg odom\_frame\_id)"/>

<remap from="cmd\_vel" to="navigation\_velocity\_smoother/raw\_cmd\_vel"/>

<remap from="odom" to="$(arg odom\_topic)"/>

<remap from="scan" to="$(arg laser\_topic)"/>

</node>

</launch>

Allerdings wurden hier lediglich Standardparameter verwendet, die aus einem Demonstrationsbeispiel des move\_base Packet stammen. In der Konfigurationsdatei werden wiederum Parameterdateien geladen, wobei die meisten aus dem Packet *turtlebot\_navigation* übernommen werden. Für Versuchszwecke wurden in dieser Konfigurationsdatei die Paramterfiles für die Kostenkarte und den lokalen Planner durch eigene Implementierung ersetzt. Dadurch können eigene Änderungen an den Paramtern vorgenommen werden.

**Anleitung**

0. [slave] sudo ntpdate 192.168.0.100 //Nur einmalig notwendig, Synchro

1. [master] roscore

2. [slave] roslaunch EML\_Navigation\_Slave EML\_Navigation\_Slave.launch

3. [master] roslaunch EML\_Navigation\_Master EML\_Navigation\_Master.launch

Bevor wir uns an die Hardkodierung machen, wird nochmal eine Group/namespace Implementierung dazwischengeschoben, um möglichst den Fehler zu reproduzieren und dokumentieren.

Dafür wird der group-tag verwendet.

Als erstes Beispiel dient die Hardwareansteuerung

**Simultane Navigation mehrerer Roboter**

Das Endziel dieser Arbeit besteht darin, dass zwei Roboter sich simultan durch einen Raum bewegen. Dabei kommunizieren beide Geräte über dasselbe ROS-Netzwerk, weshalb Namenskonflikte bei der Nachrichtenkommunikation auftreten. Beispielsweise wird die Topic scan, welche die Laserscandaten enthält, von beiden Robotern veröffentlicht, wenn die obigen Lauch-Dateien auf jeder Maschine ausgeführt werden. So wird es unmöglich Sensordaten einem der beiden Robotern zuzuordnen.

Eine mögliche Lösung stellen die group-Tags [19] unter ROS dar, mit denen jeder ROS-Node ein Namensraum zugeordnet werden kann. Als erstes Beispiel dient die Launch-Datei EML\_NS\_Hardware\_Init\_Slave.launch auf dem Slave, welche die Hardware initialisiert.

<!-- 23.02.2018; Michael Meindl; Basisinitialisierung der Hardware -->

<launch>

<arg name="namespace" default="Robot"/>

<group ns="$(arg namespace)">

<!-- Basis-Initialisierung -->

<include file="$(find turtlebot\_bringup)/launch/minimal.launch"/>

<!-- Initialisierung des SICK-TIM551 -->

<param name="robot\_description" command="$(find xacro)/xacro.py '$(find sick\_tim)/urdf/example.urdf.xacro'"/>

<include file="$(find turtlebot\_chor\_navigation)/launch/include/sick\_tim551\_2050001\_timefix.launch"/>

</group>

</launch>

Die Launch-Datei kann mit dem zusätzlichen Argument „namespace“ gestartet werden, wodurch der Namensraum der Knoten festgelegt wird. So führt der Befehl

roslaunch EML\_Navigation\_Slave EML\_NS\_Hardware\_Init\_Slave.launch namespace:=“R4“

dazu, dass jeder Topic der Präfix /R4/ hinzugefügt wird, wodurch eine klare Zuordnung zwischen Nachricht und Roboter erfolgt. Auf dem zweiten Roboter muss lediglich der namespace-Paramter angepasst werden.

Ebenso wird auf dem Master die Launch-Datei EML\_NS\_Teleop\_Master.launch angelegt, womit die Fernsteuerung der Roboter unter Namensräumen erfolgen kann.

<!-- 23.02.2018; Michael Meindl; Namespace Teleoperation -->

<launch>

<arg name="namespace" default="Robot"/>

<group ns="$(arg namespace)">

<include file="$(find turtlebot\_teleop)/launch/keyboard\_teleop.launch"/>

</group>

</launch>

Prinzipiell sollte es möglich sein, die Navigation nach dem selben Ansatz mit einem Namensraum zu versehen und dadurch die beiden Roboter zu entkoppeln. Hierfür wird die Slave-Datei EML\_NS\_Navigation\_Slave.launch angelegt.

<!-- 23.02.2018; Michael Meindl -->

<launch>

<arg name="namespace" default="Robot"/>

<include file="$(find EML\_Navigation\_Slave)/launch/EML\_NS\_Hardware\_Init\_Slave.launch">

<arg name="namespace" value="$(arg namespace)"/>

</include>

</launch>

Analog wird auf dem Master die Datei EML\_NS\_Navigation\_Master.launch angelegt.

<!-- 23.02.2018; Michael Meindl -->

<launch>

<arg name="namespace" default="Robot"/>

<group ns="$(arg namespace)">

<!-- Map-Server -->

<arg name="map\_file" default="$(find EML\_Navigation\_Master)/maps/test\_map.yaml"/>

<node name="map\_server" pkg="map\_server" type="map\_server" args="$(arg map\_file)"/>

<!-- AMC-Localization -->

<arg name="3d\_sensor" default="$(env TURTLEBOT\_3D\_SENSOR)"/>

<arg name="initial\_pose\_x" default="0.0"/>

<arg name="initial\_pose\_y" default="0.0"/>

<arg name="initial\_pose\_a" default="0.0"/>

<arg name="custom\_amcl\_launch\_file" default="$(find turtlebot\_navigation)/launch/includes/amcl/$(arg 3d\_sensor)\_amcl.launch.xml"/>

<include file="$(arg custom\_amcl\_launch\_file)">

<arg name="initial\_pose\_x" value="$(arg initial\_pose\_x)"/>

<arg name="initial\_pose\_y" value="$(arg initial\_pose\_y)"/>

<arg name="initial\_pose\_a" value="$(arg initial\_pose\_a)"/>

<arg name="odom\_frame\_id" value="odom"/>

<arg name="base\_frame\_id" value="base\_footprint"/>

<arg name="global\_frame\_id" value="map"/>

<!-- arg name="scan\_topic" value="$(arg namespace)/scan"/-->

<!-- param name="tf\_prefix" value="$(arg namespace)"/-->

<arg name="use\_map\_topic" value="true"/>

</include>

<!-- Static Transform from base\_footprint to base\_link -->

<node pkg="tf" type="static\_transform\_publisher" name="base\_link\_footprint\_tf\_broadcaster" args="0 0 .5 0 0 0 1 base\_footprint base\_link 100"/>

<!-- Rviz -->

<node name="rviz" pkg="rviz" type="rviz" args="-d $(find turtlebot\_rviz\_launchers)/rviz/navigation.rviz"/>

<!-- move\_base -->

<include file="$(find EML\_Navigation\_Master)/launch/include/eml\_move\_base.launch"/>

</group>

</launch>

An dieser Stelle besteht das erst Problem darin, dass in der Rviz-Konfiguration festgelegt werden, unter welchen Namen die relevanten Nachrichten veröffentlicht werden. Daraus folgt, dass ein Weg gefunden muss, die Rviz-Konfiguration ebenfalls mit dem Namenraum zu parametrisieren.

Als zweites Problem sind die tf Transformationen zu nennen, die jeweils mit einem Prefix versehen werden müssen.

Die andere Alternative besteht darin, die Systeme manuell zu individualisieren. Das heißt sämtliche Launch-Dateien bestehen in einer Roboter2 und einer Roboter4-Variante. Das bedeutet allerdings, das jeder einzelne ROS-Name händisch mit einem Präfix versehen werden muss. Allerdings ist das ein aussichtsloses Vorhaben, wenn die Anzahl der Paramter beachtet wird.

Die Rviz-Problematik kann gelöst werden, indem über die Benutzeroberfläche die fehlenden Präfix nachträglich eingefügt werden. Somit ist es möglich Rviz an beliebige Namensräume anzupassen. Allerdings handelt es sich dabei um eine recht unkomfortable Lösung, weshalb es sich anbietet die bieden Konfigurationen abzuspeichern und nach Programstart gezielt zu laden.

Die Lösung funktioniert auch im praktischen Test, womit als letztes Problem die Transformationen bleiben. ROS verwendet einen so genannten Transformation-Tree bzw. TF-Tree, um die Position von Objekten darzustellen. Bei einer Transformation – kurz tf – handelt es sich um eine Transformation zwischen zwei Bezugssystemen, die im Kontex von ROS als frames bezeichnet werden. Für gewöhnlich fungiert die Karte als Intertialsystem bzw. „fixed frame“. Eine Transformation gibt dann den Zusammenhang zwischen dem Frame „map“ und „robot“ wieder, wodurch die Position des Roboters festgelegt wird. Offensichtlich übernimmt diese Aufgabe die AMC-Lokalisierung. An dieser Stelle treten wieder dieselben Namenskonflikte auf, wie bei den ROS-Nachrichten. Jeder Roboter benötigt einen Transformationsbaum, der von dem des anderen Roboters entkoppelt ist.

An dieser Stelle kann der ROS-Tag tf\_prefix als mögliche Lösung verwendet werden.

Aktuell sieht der TF-Tree so aus, dass die erst Transformation von „map“ zu „odom“ von der AMCL-Node veröffentlicht wird. Anschließend wird von „odom“ zu „base\_footprint“ transformiert, wofür der Knoten „mobile\_base\_nodelet\_manager“ verantwortlich ist. Von „base\_footprint“ wird wiederum in „base\_link“ überführt. Veröffentlicht wird die Transformation von „base\_link\_footprint\_tf\_broadcaster“. Zuletzt erfolgt die Transformation von „base\_link“ zu „laser“ wofür der Knoten „robot\_state\_publisher\_no\_ns“ zuständig ist. In dem Bezugssystem „laser“ werden die Daten des Laserscanners interpretiert. „base\_footprint“ und „base\_link“ sind jeweils körperfeste Bezugssysteme des Roboters, deren relative Ausrichtung zueinander konstant ist.

Die Knoten „mobile\_base\_nodelet\_manager“ und „robot\_state\_publisher\_no\_ns“ werden jeweils über das Slave-Skript gestartet; „mobile\_base\_nodelet\_manager“ und „amcl“ von der Master-Datei.

An der Stelle müssen jetzt sämtliche launch-Dateien, die auf Slave-Seite inkludiert werden, kopiert und so angepasst werden, dass der tf\_prefix bei den relevanten Knoten eingestellt wird.

ToDos Schriftlicher Bericht:

- Kurzbeschreibung von ROS?

- Nodes

- Nachrichten

- Topics

- roscore

- relevante Pakete für die Navigation

- Globaler Planer: Kommentare abarbeiten

- 4.2 Messmodell: Anmerkung durchgehen

- 4.3 Bayes-Filter: Formatierung

- 4.3 AnmerkungeN: Anmerkungen durcghehen

- 5 Kartenerstelung: MATLAB-Beispiel?, Intro wieso wir darauf eingehen, wieso nicht zu tief?

- 6.1 letzte Gleichung argmax G

[1] Kenta Takaya, Toshinori Asaiy, Valeri Kroumov, Florentin Smarandach: „Simulation Environment for Mobile Robots Testing Using ROS and Gazebo“

[2] Aaron Martinez, Enrique Fernandez: “Learning ROS for Robotics Programming”

[3] TurtleBot – ROS-Wiki: [http://wiki.ros.org/Robots/TurtleBot?distro=indigo#turtlebot.2BAC8-Tutorials.2BAC8-indigo.Simulation](http://wiki.ros.org/Robots/TurtleBot?distro=indigo" \l "turtlebot.2BAC8-Tutorials.2BAC8-indigo.Simulation)

[4] Global-Planner <http://wiki.ros.org/global_planner>

[5] ros-blog über navfn/global\_planer quadratic approx [https://answers.ros.org/question/11388/navfn-algorism/?answer=16891#answer-container-16891](https://answers.ros.org/question/11388/navfn-algorism/?answer=16891" \l "answer-container-16891)

[6] costmap\_2d <http://wiki.ros.org/costmap_2d>

[7] staticmap <http://wiki.ros.org/costmap_2d/hydro/staticmap>

[8] obstacles <http://wiki.ros.org/costmap_2d/hydro/obstacles>

[9] inflation <http://wiki.ros.org/costmap_2d/hydro/inflation>

[10] dwa\_local\_planner <http://wiki.ros.org/dwa_local_planner>

[11] Basic Turotirals Turtlebot [http://wiki.ros.org/turtlebot\_bringup/Tutorials/indigo/TurtleBot%20Bringup](http://wiki.ros.org/turtlebot_bringup/Tutorials/indigo/TurtleBot Bringup)

[12] turtlebot\_bringup <http://wiki.ros.org/turtlebot_bringup>

[13] Turtleboys

[14] turtlebot\_teleop <http://wiki.ros.org/turtlebot_teleop>

[15] hector\_mapping <http://wiki.ros.org/hector_mapping>

[16] amcl <http://wiki.ros.org/amcl>

[17] turtlebot\_rviz\_launchers <http://wiki.ros.org/turtlebot_rviz_launchers>

[18] move\_base <http://wiki.ros.org/move_base>

[19] group http://wiki.ros.org/roslaunch/XML/group