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.

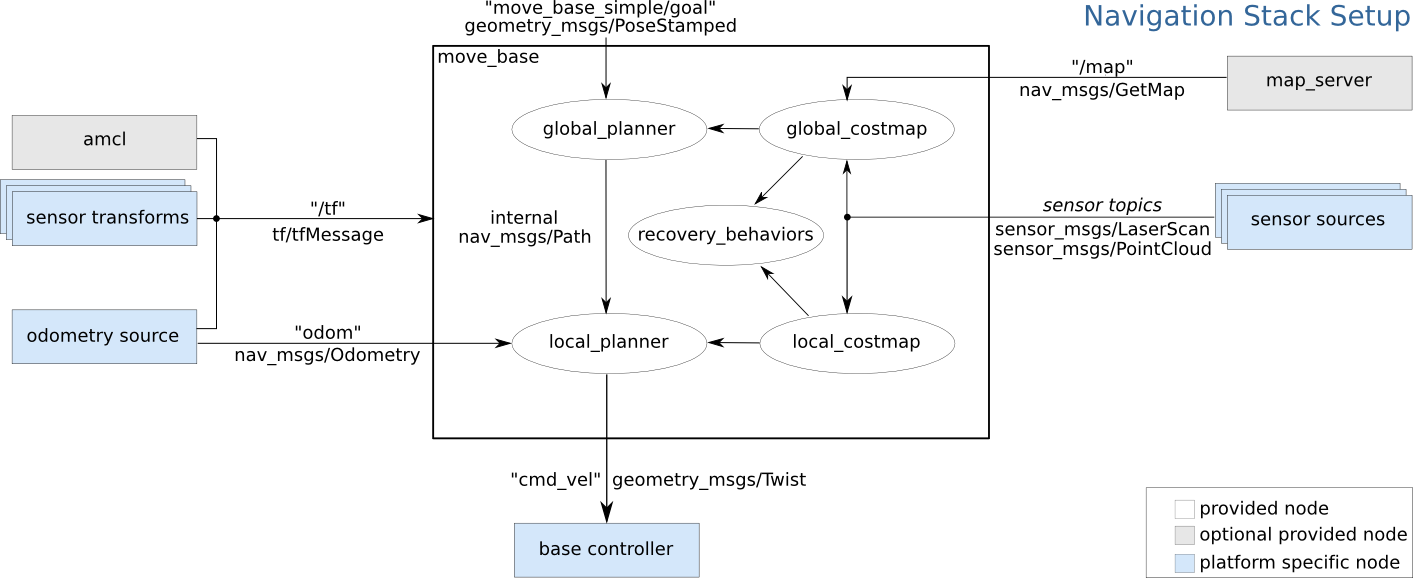
**28.11.2017:**

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.

[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