Hochschule Kaiserslautern
Angewandte Informatik
Studienprojekt im Wintersemester 19/20

# Dokumentation des Studienprojekts Raumerfassung und Sprachsteuerung für einen teilautonomen Roboter

Zweibrücken, 17.02.2020

# Inhaltsverzeichnis

# Einleitung

Dieses Dokument dient als Projektbericht für das Studienprojekt "Raumerfassung und Sprachsteuerung für einen teilautonomen Roboter" im Wintersemester 2019/20 des Studiengangs Angewandte Informatik an der Hochschule Kaiserslautern, Standort Zweibrücken. Unter der Leitung von Prof. Adrian Müller wurder der TurtleBot2 von Markus Dauth, David Kostka, Felix Mayer und Raschied Slet programmiert. Ziel war es, den teilautonomen Roboter TurtleBot2 (Projekt interner Spitzname "Botty McTurtleFace", kurz "Botty") mittels Sprachbefehlen in einem Pick and Place Szenario steuern zu können. Mittels einem ROS-Package kann der TurtleBot durch Sprachbefehle in einem vorgegebenen Raster mit einer 3D-Kamera nach Objekten suchen. Findet der TurtleBot ein Objekt das er kennt, versucht er nach dem Objekt zu greifen. Er kann während des Fahrens Objekte umgehen, indem eine Hinderniserkennung mit einem Lidar stattfindet.

# Dieses Dokument und das Git-Repository

## Dieses Dokument

Dieses Dokument dient als Einstieg für die Dokumentation des Studienprojekt und stellt projektübergreifende Informationen dar.

In diesem Dokument wird im Folgenden auf die folgenden Aspekte eingegangen:

- Zu kennende Konzepte und Begriffe
- Ziele und zeitlicher Ablauf des Projektes und wer daran beteiligt war
- Hard- und Software des TurtleBot
- Vorhandene Demos des Botty-Packages und wie man diese installiert und ausführt
- Einrichten einer Entwicklungsumgebung und wie man den TurtleBot damit programmiert
- Umgesetzte Architektur und Technik des Botty-Packages

## Git-Repository

Das Endergebnis des Projektes ist ein ROS-Package und befindet sich in einem GitHub-Repository, welches unter folgendem Link zu finden ist:
https://github.com/MarkusDauth/botty

Das Git-Repository wurde von Markus Dauth erstellt und wurde von allen Teammitgliedern zur Entwicklung der Software für den TurtleBot benutzt. In dem Repository befindet sich der erstellte Programmcode und dessen Dokumentation. In diesem Projektbericht werden die Artefakte dieses Repository referenziert.
Das Wurzelverzeichnis sowie alle anderen Unterordner enthalten jeweils eine Datei namens README.md, in welcher die Dokumentation der einzelnen Dateien des entsprechenden Ordners (mittels Markup) zu finden ist. Die Readme-Datei im Wurzelverzeichnis dient als Einstieg der Dokumentation und verweist auch auf die übrigen Readme-Dateien. Ausnahme hierbei ist das Verzeichnis “documents”, in dem sich nur Dateien für die Dokumentation (z.B. Bilder und Videos) befinden.

Weitere Informationen zur Benutzung des Git-Repository befinden sich im Kapitel Git.

# Konzepte und Begriffe

Folgende Konzepte und Begriffe sollten vor dem Lesen dieses Dokuments bekannt sein.

Der TurtleBot2 (https://www.turtlebot.com/turtlebot2/) verwendet das Robot Operating System (ROS, http://wiki.ros.org/). Folgende ROS-Konzepte wurden in diesem Projekt angewandt:

- Topics: http://wiki.ros.org/Topics
- Nodes: http://wiki.ros.org/Nodes
- Services: http://wiki.ros.org/Services
- Messages: http://wiki.ros.org/Messages
- Packages: http://wiki.ros.org/Packages
- Launch-Files: https://wiki.ros.org/roslaunch/XML
- Kompilierung des ROS-Packages mit catkin: http://wiki.ros.org/catkin
- ROS-Network: https://edu.gaitech.hk/turtlebot/network-config-doc.html

Zur Hinderniserkennung wurde ein Lidar-Sensor benutzt: https://de.wikipedia.org/wiki/Lidar

Während des Projektes wurde Git als Versionsverwaltung verwendet: https://git-scm.com/

Der TurtleBot2 nutzt als Betriebssystem Ubuntu 16.04. LTS. Um mit dem TurtleBot2 zu arbeiten, sollte man die Basics von Ubuntu beherrschen:
https://wiki.ubuntuusers.de/Startseite/

# Das Botty-Projekt

## Ziele

Ziel ist es, den TurtleBot2 mittels Sprachbefehlen in einem Pick-And-Place-Szenario zu steuern. Der TurtleBot soll in einem Raum nach vortrainierten Objekten suchen. Wird ein Objekt erfolgreich mit einer 3D-Kamera erkannt, hebt der Greifarm das Objekt auf. Über das Lidar soll bei der Navigation eine Hindernisvermeidung stattfinden. Die entwickelten Funktionalitäten werden über ein selbst erstelltes ROS-Package (“Botty”) angeboten.
Nach einer Einarbeitungsphase bezüglich der Prinzipien von ROS sollte eine gemeinsame Entwicklungsumgebung eingerichtet werden, mit der anschließend das ROS-Package agil als vertikaler Prototyp entwickelt wird.

Folgende Funktionalitäten wurden in diesem Projekt realisiert:

- Botty kann durch Sprachbefehle gesteuert werden und Antworten. Botty versteht eine einfache Grammatik und kann mit einem Text-to-Speech-System (TTS) über Lautsprecher antworten.
- Objekterkennung von mehreren vortrainierten Objekten.
- In einem vordefinierten Koordinatensystem (im Folgenden "Grid" genannt) kann Botty an eine Position geschickt werden und an der Position ein vorgegebenes Objekt suchen, wobei er sich im Kreis dreht, bis er das gesuchte Objekte erkannt hat oder sich um 360° gedreht hat, falls er es nicht findet. Er selbst wird auch sein Ergebnis über Sprache mitteilen.
- Bei erfolgreicher Objekterkennung bewegt sich der Arm nach vorne.
- Hinderniserkennung mittels Lidar und Einteilung in Links, Rechts und Vorne.
- Beim Vorwärtsfahren ist Botty in der Lage Objekte zu umfahren, er kann sich drehen und auf Kommando stoppen.

## Team

Dieses Studienprojekt wurde von vier Angewandte-Informatik-Studenten der Hochschule Kaiserslautern unter der Leitung von Prof. Adrian Müller durchgeführt. Aufgrund der Vielseitigkeit des Projektes wurden die einzelnen Komponenten auf die einzelnen Teammitglieder verteilt. Folgende Tabelle stellt die Verteilung dieser Komponenten dar:

| **Name** | **Bearbeitete Komponenten** | **GitHub Nutzername** |
|---|---|---|
| Markus Dauth | Greifarm | https://github.com/MarkusDauth |
| David Kostka | Spracherkennung-/ausgabe und Controller | https://github.com/david-kostka |
| Felix Mayer | Motorsteuerung und Lidar | https://github.com/DeltaSence |
| Raschied Slet | 3D-Kamera | https://github.com/RaschiedSlet |

## Zeitlicher Verlauf

Das Projekt wurde agil in mehreren Phasen durchgeführt. Die einzelnen Phasen können als Sprint angesehen werden. Folgende Tabelle beschreibt den Ablauf des Projektes, wobei für jede Phase das Anfangsdatum angegeben wird.

| **Startdatum** | **Phase** |
|---|---|
| 01.10.2019 | Einarbeitungsphase (Kenntnisse über ROS und den TurtleBot erlangen) |
| 15.10.2019 | Entwicklungs- und Simulationsumgebung einrichten, Hardware-Probleme am TurtleBot beheben |
| 31.10.2019 | Einarbeitung und Testen der Umsetzungsmöglichkeiten der einzelnen Komponenten |
| 30.11.2019 | Realisierung des Pick and Place Szenario als vertikaler Prototyp (aufgeteilt in zwei Sprints) |
| 16.01.2020 | Pause wegen Klausuren |
| 01.02.2020 | Dokumentation und Refactoring |
| 17.02.2020 | Ende |

# Hardware und Software des TurtleBots

Im Folgenden wird die Hardware des TurtleBots 2, welcher im Projekt benutzt wurde, aufgelistet. Der TurtleBot2 wurde von “ROS-Components” bereits fertig aufgebaut geliefert. Für jede Hardware-Komponente wird die verwendete Software zur Steuerung (Driver) aufgelistet. Als “Driver” werden im Folgenden die Schnittstellen zwischen der Hardware und des Botty-Packages bezeichnet, sprich Frameworks von Dritten, um über ROS die Hardware zu steuern.
Die Verwendung der Hardware, sowie die Installation der Driver wird im Kapitel Demos erläutert. Wie die Driver in dem Botty-Package benutzt werden, wird in Kapitel Architektur und Technik erläutert.

Folgendes Bild zeigt den Aufbau des TurtleBots:

Der TurtleBot besteht aus folgenden Hardware-Komponenten:

**Greifarm PhantomX Reactor Arm**
Produkt: https://www.roscomponents.com/en/robotic-arms/100-phantomx-reactor.html#/montaje_widowx-yes/reactor_wrist_rotate-yes
Zur Steuerung des Arms (Driver) wurde dieses Github-Repository benutzt:
https://github.com/RobotnikAutomation/phantomx_reactor_arm

**Lidar Hokuyo URG-04LX-UG01**
Produkt:
https://www.roscomponents.com/en/lidar-laser-scanner/83-urg-04lx-ug01.html
Zur Nutzung des Lidar (Driver) wurde dieses Github-Repository benutzt:
https://github.com/ros-drivers/hokuyo_node

**3D-Kamera Orbbec Astra**
Produkt:
https://www.roscomponents.com/en/cameras/76-orbbec.html
Zur Nutzung der Kamera (Driver) wurde dieses Github-Repository benutzt:
https://github.com/orbbec/ros_astra_camera

**NUC mit Ubuntu 16.04 LTS**
Onboard PC des TurtleBot (im Folgenden NUC genannt), welcher vorinstalliert mit dem

TurtleBot geliefert wurde.
Ausstattung:
Intel Core i5-7260U 7th Gen, 8GB RAM, 240GB SSD M.2, 1TB externe HDD

**Kobuki-Base**
Produkt:
https://www.roscomponents.com/en/mobile-robots/97-kobuki.html

**Lautsprecher (MIFA)**
Bestelllink:
https://www.amazon.de/MIFA-Lautsprecher-Wasserfester-Staubdichter-Audio-Eingang/dp/B074TY4JK4/ref=sr_1_5?__mk_de_DE=%C3%85M%C3%85%C5%BD%C3%95%C3%91&crid=5R1D44F5KSBG&keywords=testsieger+2020+bluetooth&qid=1578574282&sprefix=testsieger+2020+bl%2Caps%2C179&sr=8-5

**Mikrofon (Docooler)**
Bestelllink:
https://www.amazon.de/Docooler-Omnidirektionale-Kondensator-Anschluss-Conference-Type-1/dp/B0757K3P77/ref=sr_1_4?__mk_de_DE=%C3%85M%C3%85%C5%BD%C3%95%C3%91&crid=6S4MWRVHQZNC&keywords=omnidirektionales+mikrofon&qid=1578574640&sprefix=omnidi%2Caps%2C172&sr=8-4

# Demos

Dieses Kapitel dient als Anleitung zur Installation und zum Starten der Komponenten und Demos. Als Demo wird ein eigenständiger, von uns entwickelter Ablauf bezeichnet der über ROS-Nodes des Botty ROS-Packages gestartet werden kann. Beispiele hierfür wäre, dass der Botty sich um 90° im Kreis dreht oder dass der Arm eine vorgegebene Pose annimmt.
Die genauere Funktionsweise und Programmcode der einzelnen Komponenten wird im Kapitel Architektur und Technik erläutert. Dieses Kapitel ist wie folgt gegliedert:

In den Unterkapiteln “Hardware/Software Vorbereitungen” wird die allgemeine Installation und Konfiguration von ROS-Kinetic und Turtlebot2 besprochen.

Danach gibt es für jede Komponente fünf Unterkapitel, in denen Folgendes erklärt wird:

- Vorbereitung
    - Welche Hardware-Komponenten müssen wo angeschlossen sein?
    - Was muss vor dem Starten der Software beachtet werden?
- Installation
    - Es werden alle notwendigen Schritte aufgelistet, um dieses ROS-Package (botty) mit allen erforderlichen Abhängigkeiten zu installieren.
- Starten der Komponenten
    - Es wird erläutert, wie man die Funktionalitäten des Botty-Packages ausführt.
    - Für jede Hardware-Komponente müssen Befehle ausgeführt werden, um deren ROS-Nodes zu starten.
    - Um eine bessere Übersicht über den aktuellen Status der einzelnen Komponenten (und eventuelles Debugging) zu erhalten, wird empfohlen, dass die jeweiligen Skripte in einzelnen Konsolen (Terminals) ausgeführt werden.
- Starten der Demos
    - Hier werden mögliche Demos der einzelnen Komponenten und des Gesamtsystems aufgelistet. Dazu gehört eine Beschreibung der Demo und welche Befehle zum starten der Demo ausgeführt werden müssen.
    - Gegebenenfalls wird bei manchen Demos auch auf entsprechende Videos hingewiesen, die im Git-Repository unter /documents/videos zu finden sind.
- Known Problems
    - Mögliche Probleme, die bei der Ausführung der Demo entstehen können.

# Hardware Vorbereitungen

Um Programme direkt an Botty auszuführen, stehen im Labor ein Bildschirm, eine Tastatur und eine Maus bereit, die an den NUC (den PC von Botty) angeschlossen werden können. Bei den Demos sollte auf Folgendes geachtet werden:

1. Die Sicht des Lidar oder der Kamera sollte nicht blockiert sein. Es sollte beispielsweise darauf geachtet werden, dass keine Kabel des TurtleBot in sichtweite des Lidar sind.
2. Der Arm hat eine große Reichweite und es sollte darauf geachtet werden, dass er einen entsprechenden Freiraum zum Bewegen hat.
3. Vor dem Ausführen von Demos sollte der Akku geladen werden.
4. Die Dockingstation hat einen Wackelkontakt, weshalb es sich empfiehlt das Ladekabel direkt am TurtleBot anzuschließen.
5. Wenn Audioausgabe erwünscht ist:
   Bei dem Bluetooth Speaker den An/Aus Knopf gedrückt halten. Eventuell muss der NUC manuell über Bluetooth mit dem Lautsprecher verbunden werden.

# Software Vorbereitungen

## ROS-Installation

ROS war bereits auf dem Turtlebot vorinstalliert und eingerichtet. Ein USB-Stick mit einer Kopie der bereits installierten Software (und Turtlebot-Manual) wurde mitgeliefert, falls eine Neuinstallation nötig ist (dies wurde aber nicht von uns getestet).
Falls aber eine Neuinstallation ohne USB-Stick nötig ist, siehe das mitgelieferte Turtlebot-Manual und ggf.:
http://wiki.ros.org/kinetic/Installation/Ubuntu
http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment
Wichtig ist hierbei zu beachten, dass bei einer Neuinstallation auch ein neues Catkin-Workspace installiert werden muss.

Das Botty-Repository dient als eigenes ROS-Package. Zur Installation müssen mehrere Packages über apt-get installiert werden und zusätzliche Git-Repositories in den "src" Ordner des Catkin-Workspace geklont werden. Das Catkin-Workspace befindet sich auf dem TurtleBot im Home-Verzeichnis des Nutzers “tb2”.

Vor der Installation der Packages sollte Folgendes direkt auf dem NUC des TurtleBot ausgeführt werden:

```
rosdep update
sudo apt-get update
sudo apt-get dist-upgrade
sudo apt-get install ros-kinetic-catkin python-catkin-tools
```

Klonen des botty-Packages in den Ordner “src” des Workspace:

```
git clone https://github.com/MarkusDauth/botty
```

Danach müssen die Unterordner von “botty” in den “src” Ordner des Workspace (catkin_ws/src) kopiert werden, um die Kompilation mit catkin zu ermöglichen.

Das Einrichten einer Entwicklungsumgebung wird im Kapitel Entwicklung und Wartung beschrieben.

## ROS-Network Konfiguration

Momentan besitzt Botty folgende Zugangsinformationen:

- Ubuntu Benutzer: tb2
- Ubuntu Passwort: ros
- Feste IP-Adresse: 10.0.189.60

Turtlebot und Workstation(s) sollten sich im selben Netzwerk befinden, also in unserem Fall das Hochschulnetz. Der TurtleBot kann sich per WLAN in mit dem Hochschulnetz verbinden. Hierfür muss ein eine WLAN-Konfiguration mit einem Hochschul-Account vorgenommen werden. Dies ermöglicht einen Zugriff per SSH von den Workstations auf den TurtleBot und

eine Auslagerung von Nodes des Botty-Packages auf Workstations mit Hilfe des ROS-Network (siehe unten).

Die IP der Masternode muss auf allen Maschinen (NUC und Workstations) festgelegt werden. Die Masternode kümmert sich um die Kommunikation zwischen den Nodes und darf nur einmal im ROS-Network existieren.
In diesem Projekt ist die Masternode das NUC des Turtlebot und wird gestartet mit:

```
roslaunch kobuki_node minimal.launch
```

Masternode festlegen:
Auf Turtlebot:

1. Folgendes am Ende der Datei ~/.bashrc eintragen:

```
#The IP address for the Master node
export ROS_MASTER_URI=http://10.0.189.60:11311
#The local IP address = IP address for the Master node
export ROS_HOSTNAME=10.0.189.60
```

2. Terminal schließen oder “source ~/.bashrc” ausführen.

Auf Workstations:

1. Folgendes am Ende der Datei ~/.bashrc eintragen, “x.x.x.x” ist dabei die IP der Workstation:

```
#The IP address for the Master node
export ROS_MASTER_URI=http://10.0.189.60:11311
#The IP address for your device/host IP address
export ROS_HOSTNAME=x.x.x.x
```

2. Terminal schließen oder “source ~/.bashrc” ausführen.

Die Masternode muss auf dem Turtlebot laufen, damit die Nodes der Maschinen im Netzwerk miteinander kommunizieren können.
Mehr Infos unter: https://edu.gaitech.hk/turtlebot/network-config-doc.html

## Befehle

Befehle können wie folgt ausgeführt werden:

- Direkt auf dem Turtlebot arbeiten. Tastatur, Maus und Bildschirm müssen hierfür direkt an das NUC angeschlossen werden.
- Per ssh mit dem Turtlebot verbinden. Hierfür müssen sich beide Rechner im selben Netzwerk befinden.
    - ssh tb2@10.0.189.60

- Wenn beide Rechner im ROS-Network sind, also die selbe Masternode-IP besitzen, können sie miteinander durch das ROS-Framework kommunizieren.
    - Das heißt Nodes können auch auf anderen Rechnern laufen und Messages können von jedem Rechner in das Netzwerk versandt werden. Dies ist zum starten der Demos hilfreich (z.B. “rostopic pub” oder “rosservice call”).
    - Nur Driver-Nodes müssen immer auf dem Turtlebot gestartet werden. Alle anderen Nodes sollten auch direkt auf dem Turtlebot laufen, um hohe Latenzen zu vermeiden. Durch die Auslagerung von Nodes könnten aber leistungsintensive Aufgaben auf die Workstations verlagern werden, um den Stromverbrauch des TurtleBots zu verringern.

# Roscore

Für alle Programme muss vorher der roscore in einer eigenen Konsole gestartet werden. Mit dem TurtleBot2 geht das unter:

```
roslaunch kobuki_node minimal.launch
```

# Arm

## Vorbereitung

Der Greifarm muss über sein Stromkabel an der Kobuki-Base und am NUC des TurtleBots angeschlossen sein. Sobald der NUC eingeschaltet ist, ist der Arm mit Strom versorgt und kann über die Demo-Befehle gesteuert werden.

## Installation

Dependencies installieren:

```
sudo apt-get install ros-kinetic-arbotix
sudo apt-get install ros-kinetic-dynamixel-controllers
```

Im Ordner “src” des Workspace dieses Repository klonen und der Installationsanleitung des Repositories für "Arbotix-M" folgen:

```
git clone https://github.com/RobotnikAutomation/phantomx_reactor_arm.git
```

## Starten der Komponenten

Arm-Driver-Node in eigener Konsole starten:

```
roslaunch phantomx_reactor_arm_controller arbotix_phantomx_reactor_arm_wrist.launch
```

Arm-Control-Node starten:

```
rosrun arm arm_control.py
```

## Starten der Demos

**Greifen**

Der Arm führt “Greif-Animation” aus, dabei sollte er den Arm erst nach links ausfahren und sich dann nach rechts drehen.
Ziel ist es das Objekt vor ihm um zu werfen.

```
rosservice call /botty/arm/commands "call: 'push'
param:
- 0"
```

**Home-Position**

Arm soll zurück in Startposition gehen.

```
rosservice call /botty/arm/commands "call: 'home'
param:
- 0"
```

## Known Problems

Beim Ausführen der Demos sind uns keine Probleme bekannt, es sollte jedoch darauf geachtet werden, dass genug Freiraum für die Bewegungen des Armes vorhanden ist.

# 3D-Kamera

## Vorbereitung

Die 3D-Kamera muss am NUC angeschlossen sein.

Folgende Objekte können erkannt werden:

- Cola Flasche
- Roter Pfeil
- Kreideschachtel
- Blume
- Zeitschrift

Diese Objekte stehen im Raum O028 an der Hochschule.

## Installation

Dependencies installieren:

```
sudo apt-get install ros-kinetic-find-object-2d

sudo apt install ros-kinetic-rgbd-launch ros-kinetic-libuvc ros-kinetic-libuvc-camera ros-kinetic-libuvc-ros

sudo apt-get install ros-kinetic-robot-localization
cd ~/catkin_ws/src
git clone https://github.com/orbbec/ros_astra_camera
```

Rosbot_ekf installieren:

```
cd ~/catkin_ws
git clone https://github.com/husarion/rosbot_ekf.git
catkin_make
```

### Starten der Komponenten

Die Kamera kann entweder direkt auf dem TurtleBot ausgeführt werden oder auf einen eigenen Rechner (Workstation) ausgelagert werden. Wenn die Kamera auf einem eigenen Rechner läuft, kann hierdurch Rechenleistung (und somit auch Storm) auf dem TurtleBot gespart werden.

Auf eigenem Rechner starten:

```
roslaunch camera camera.launch
```

Driver-Node auf dem Turtlebot-Rechner starten:

```
roslaunch astra_launch astra.launch
```

### Starten der Demos

Wird ein erkennbares Objekt vor die Kamera gehalten, kann durch den folgenden Befehl der Name des Objektes abgefragt werden.
Ausgabe der Objekte, die gerade von der Kamera erkannt werden:

```
rosservice call /camera_controller/find_object "{}"
```

### Known Problems

Objekte werden nicht immer korrekt erkannt. Dies kann sehr unterschiedliche Gründe haben, beispielsweise wenn Objekte in ungewöhnlichen Positionen/Winkeln zur Kamera stehen, die Lichtverhältnisse sehr dunkel sind oder Objekte zu nah an der Kamera sind.

## Lidar

### Vorbereitung

Das Lidar muss am NUC angeschlossen sein.

### Installation

Im Ordner “src” des Workspace diese Repositories downloaden:

```
git clone https://github.com/ros-drivers/driver_common.git
```

```
cd ..
catkin_make
cd src
git clone https://github.com/ros-drivers/hokuyo_node.git
cd ..
catkin_make
```

Es ist wichtig driver_common VORHER zu kompilieren, bevor hokuyo_node hinzugefügt wird. Sonst scheitert der Prozess!

### Starten der Komponenten

Zunächst muss der Hokuyo in einem eigenen Terminal gestartet werden:

```
rosrun hokuyo_node hokuyo_node
```

Danach der hokuyoInterpreter auch in seinem eigenem Terminal:

```
rosrun lidar hokuyoInterpreter.py
```

Falls das Lidar nicht gefunden wird, muss Folgendes zuvor ausgeführt werden:

```
sudo chmod a+rw /dev/ttyACM0
```

### Starten der Demos

Wenn Botty Hindernisse in seiner Umgebung erkannt hat, dann gibt er diese in Listen (aufgeteilt mit ihrer Gradzahl als Richtung und deren Entfernung) aus . Sie werden in links, rechts und vorne aufgeteilt:

```
rostopic echo /botty/hokuyoInterpreter
```

### Known Problems

Es sollte darauf geachtet werden, dass zum Start sich keine Hindernisse in Bottys Umfeld befinden, da sie bei der Kallibrierung sonst als tote Winkel betrachtet werden.

## Sprache

### Vorbereitung

Für den Audio-Stream wird das Default-Device (für Input und Output) bei Ubuntu verwendet. Es kann das Onboard-Mikrofon oder ein Externes angeschlossen werden. Zur Ausgabe kann der Bluetooth Speaker (MIFA) verwendet werden.

### Installation

Falls 'pip' nicht installiert ist:

```
sudo apt install python-pip
sudo apt install python3-pip
```

```
pip install --upgrade pip
```

Dependencies installieren:

```
sudo apt-get install libasound-dev
sudo apt-get install python-pyaudio
sudo apt-get install swig
```

Pocketsphinx installieren:

```
sudo pip install pocketsphinx
```

Package 'audio_common' für Text-To-Speech (TTS) und Sounds:

```
sudo apt-get install libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev
git clone https://github.com/ros-drivers/audio_common.git
catkin_make
rosdep install sound_play
```

## Starten der Komponenten

Recognizer und Parser gleichzeitig starten:

```
roslaunch speech parser.launch
```

**Oder** den Recognizer und Parser einzeln starten:

```
roslaunch speech recognizer.launch
roslaunch speech parser_only.launch
```

Dazu benötigte Nodes 'send_audio.py' und 'soundplay_node.py' werden automatisch mit gestartet.

## Starten der Demos

**Recognizer**

Nachdem recognizer.launch gestartet wurde kann der Botty ein Sprachbefehl über das Mikrofon entgegennehmen und auswerten.

Ablauf:

1. Dazu erst das Wake-Word sagen (zurzeit “Hey Slave”) und auf ein Piepton warten.
2. Dann kann ein Befehl genannt werden. (z.B. “go to the garage”)
3. Der vom Recognizer interpretierte Befehl wird in Textform auf der Konsole ausgegeben.

Mögliche Befehle:

- “go to the garage”
- "go forward/left/right"

- "go to the docking station"
- “grab the white cup”
- “bring the red ball”
- für weitere Möglichkeiten siehe “speech”-Package

**Parser**
Der Parser gibt die erstellte Datenstruktur (Command-Message) eines gültigen Befehls auf der Konsole aus. Je nach Gültigkeit wird ein Ton ausgegeben. Dies kann in Kombination mit der Recognizer-Demo erfolgen, dafür müssen Parser und Recognizer gleichzeitig ausgeführt (siehe “Start der Komponenten”).

Man kann auch den Text selbst schreiben, also den Recognizer umgehen:
Dafür wird mit Hilfe von “rostopic pub” eine Message mit dem Text an das Topic “/botty/speech/grammar_data” veröffentlicht.

Beispiel: “grab the red ball”

```
rostopic pub /botty/speech/grammar_data std_msgs/String “data: ‘grab the red ball’”
```

Ausgabe:

```
action: 2
obj:
  name: “ball”
  attr:     [red]
```

```
Playing sound nr.: 2
```

### Known Problems

Das Wake-Word muss eventuell oft wiederholt werden, bis es erkannt wird. Außerdem wird fälschlicherweise immer nach dem Piepton versucht ein Befehl zu finden, auch wenn kein möglicher Befehl genannt wurde.

## Motor

### Installation

Der Motor wird über die Kobuki-Base gesteuert und benötigt somit keine zusätzliche Installation.

### Starten der Komponenten

In einer eigenen Konsole folgenden Befehl ausführen:

```
rosrun motor motorService.py
```

## Starten der Demos

Allgemein repräsentiert “call” den gewünschten Befehl für die gewollte Aktion und “param” ist eine Liste der Argumente bzw. Parameter.

**Vorwärts fahren (ohne Hinderniserkennung)**
Botty fährt zwei Meter vorwärts (dabei werden keine Hindernisse umfahren). Der erste Parameter ist die Distanz, die Botty fahren soll (im Beispiel zwei Meter). Der zweite Paramete (im Beispiel 0) lässt sich als Boolean verstehen, das aussagt, dass er nicht auf Hindernisse achten soll:

| rosservice call /motor "call: 'forwardByMeters' param: [2,0]" |
| --- |
| **Video:** /documents/videos/fahre_vorwärts.mp4 |

**Vorwärts fahren (mit Hinderniserkennung)**
Botty fährt zwei Meter Vorwärts und umgeht dabei Hindernisse. Ähnlich wie oben fährt er 2 Meter vorwärts. Sollte aber ein Hindernis seien Weg versperren, so wird er es umfahren. Dabei fährt er seitlich am Hindernis vorbei, merkt sich die Zeit, die er dazu benötigt hat,. Sobald Botty am Hindernis vorbei gefahren ist, fährt er (abhängig von der umgangenen Strecke) weiter, um an dem ursprünglich geplanten Endpunkt zu gelangen.

| rosservice call /motor "call: 'forwardByMeters' param: [2,1]" |
| --- |
| **Video:** /documents/videos/umgehe_objekt.mp4<br>**Video:** /documents/videos/umgehe_objekt_2.mp4<br>In den Videos sieht man, wie Botty im auf dem Boden markierten Grid ein Hindernis umgeht. |

Botty dreht sich nach rechts. Der Parameter gibt die Winkelgröße der Drehung an. Hier 90°:

| rosservice call /motor "call: 'turnRigthByAngle' param: [90]" |
| --- |

**Stop**
Dieses Kommando wird Botty stoppen, egal welche Aktion er gerade ausführt. Da der Befehl keine Parameter benötigt, wird nur eine leere Liste übergeben.

| rosservice call /motor "call: 'stopp' param: []" |
| --- |

## Known Problems

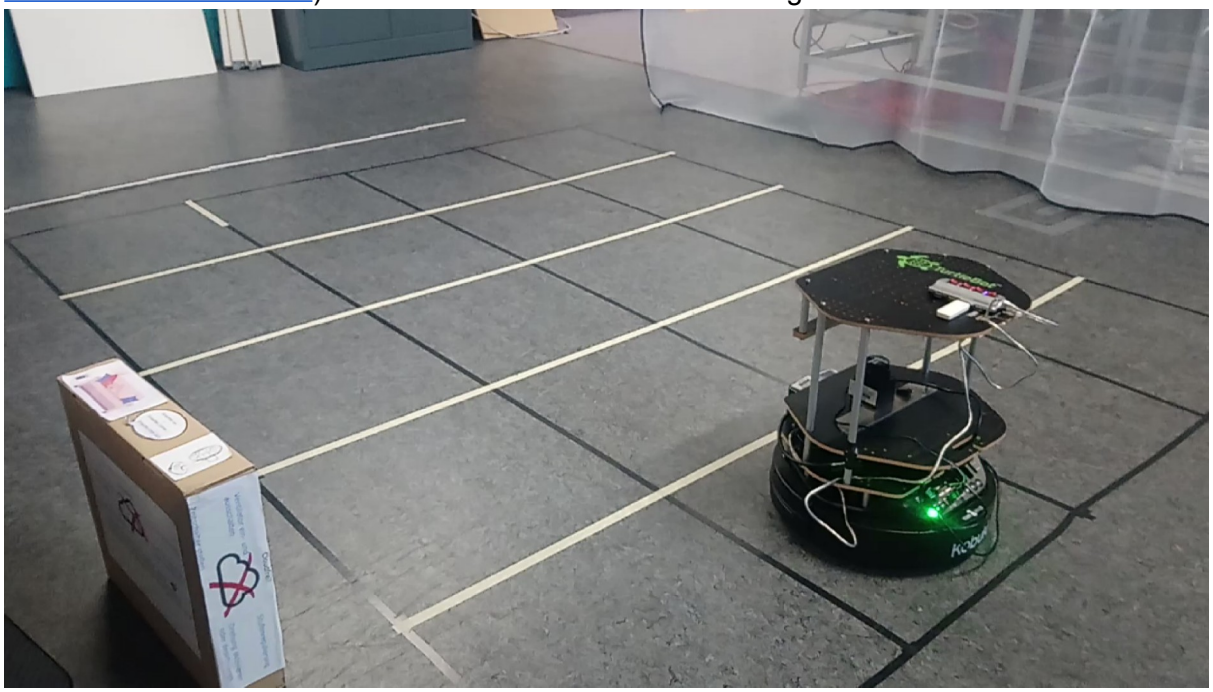
Distanzen und Winkel ändern sich mit verändertem Gewicht des Roboters, was z.B. problematisch ist, da der Greifarm einiges an Gewicht hinzufügt.

# Gesamtsystem / Controller

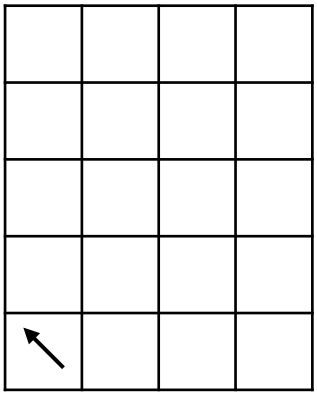
## Vorbereitung

Es müssen alle vorherigen genannten Komponenten gestartet werden, damit der Controller initialisiert werden kann, sprich für jede der anderen Komponenten müssen die entsprechenden Befehle unter “Starten der Komponente” ausgeführt werden.

Die zentrale Komponente des Botty-Packages stellt das Subpackage “Controller” dar. Es wird ein schachbrettartiges Koordinatensystem vorgegeben (im Folgenden “Grid” genannt), in dem sich Botty bewegen kann. Hierfür befindet sich in O028 Markierungen auf dem Boden, welche das Grid darstellen. Für weitere Informationen zum Grid, siehe Kapitel Architektur und Technik). Hier ein Bild zur Veranschaulichung des Grids.



Damit die Positionierung im Grid funktioniert, muss Botty vor dem Start einer Demo an folgende Position und Richtung im Grid gesetzt werden:



## Installation

Keine sonstige Installation nötig.

## Starten der Komponenten

Nur eine Node muss gestartet werden:

```
rosrun controller control.py
```

Sobald alle anderen Komponenten gestartet sind, wird auch der Controller initialisiert. Nach erfolgreichem Start sollte Botty "I am Botty McTurtleFace" sagen/ausgeben.

## Starten der Demos

Man kann Botty mit diesen Demos auf zwei Arten steuern:

1. Sprachbefehl (z.B. man sagt “go forward”)
2. Command-Message (Befehl über die Konsole an eine Node publishen)

Werden Sprachbefehle benutzt, werden diese in die entsprechenden Command-Messages umgewandelt. Es gibt momentan mehr Command-Message-Befehle, die tatsächlich ausführbar sind, als Sprachbefehle. Im Folgenden sind die Kommandos zum Starten der Demos jeweils als Sprachbefehl (falls verfügbar) und Command-Message angegeben.

**Vorwärts/Rechts/Links fahren**

Botty fährt 2 Meter Vorwärts:

```
“go forward”
```

```
rostopic pub /botty/speech/commands controller/Command “action: 1
obj:
  name: ‘forward’
  attr: - ‘”
```

**Botty dreht sich um 90° nach rechts:**

```
“go right”
```

```
rostopic pub /botty/speech/commands controller/Command “action: 1
obj:
  name: ‘right’
  attr: - ‘”
```

**Zu Position fahren**

Botty fährt an die Position [2, 3] im Grid, dabei wird das A*-Algorithmus zur Pfadfindung zum Zielpunkt verwendet:

```
Kein Sprachbefehl verfügbar
```

| |
|---|
| rostopic pub /botty/speech/commands controller/Command “action: 1<br>obj:<br>  name: ‘position’<br>  attr: [‘2’,’3’]” |

**Nach Objekt suchen**

Botty bewegt sich an die Position [1,1] im Grid und sucht an der Position nach dem angegeben Objekt. Bei Suche dreht sich Botty in 90° Intervallen im Uhrzeigersinn im Kreis, stoppt jeweils und überprüft, ob das Objekt gefunden wurde. Hat er das Objekt gefunden, gibt er eine entsprechende Nachricht aus und stoppt seine Drehung, ansonsten dreht er sich bis zu maximal 360°:

| |
|---|
| Kein Sprachbefehl verfügbar |
| rostopic pub /botty/speech/commands controller/Command “action: 5<br>obj:<br>  name: ‘cola’<br>  attr: [‘1’,’1’]” |
| **Video:** /documents/videos/suche_und_greife_flasche.mp4<br>Im Video wird nachdem das Objekt gefunden wurde auch die “Greifen” Demo gestartet. |

**Greifen**

Botty bewegt den Greifarm nach vorne links und schwingt ihn nach rechts. Ziel ist es, ein Objekt vor ihm umzuwerfen.

| |
|---|
| “grab the object” |
| rostopic pub /botty/speech/commands controller/Command “action: 2<br>obj:<br>  name: ‘object’<br>  attr: - ‘”” |
| **Video:** /documents/videos/suche_und_greife_flasche.mp4<br>Im Video führt Botty die Greifen Demo nach der Objektsuche aus. |

**Abbruch**
Während Botty ein Befehl ausführt kann dieser asynchron abgebrochen werden. Da Botty sich danach auf einer undefinierten Position im Grid befinden könnte, werden intern die Koordinaten zurückgesetzt, daher muss er nach diesem Befehl wieder an die initiale Position im Grid gesetzt werden.

| “stop” |
| --- |
| rostopic pub /botty/speech/commands controller/Command “action: 4<br>obj:<br>  name: ‘’<br>  attr: - ‘’” |

### Known Problems

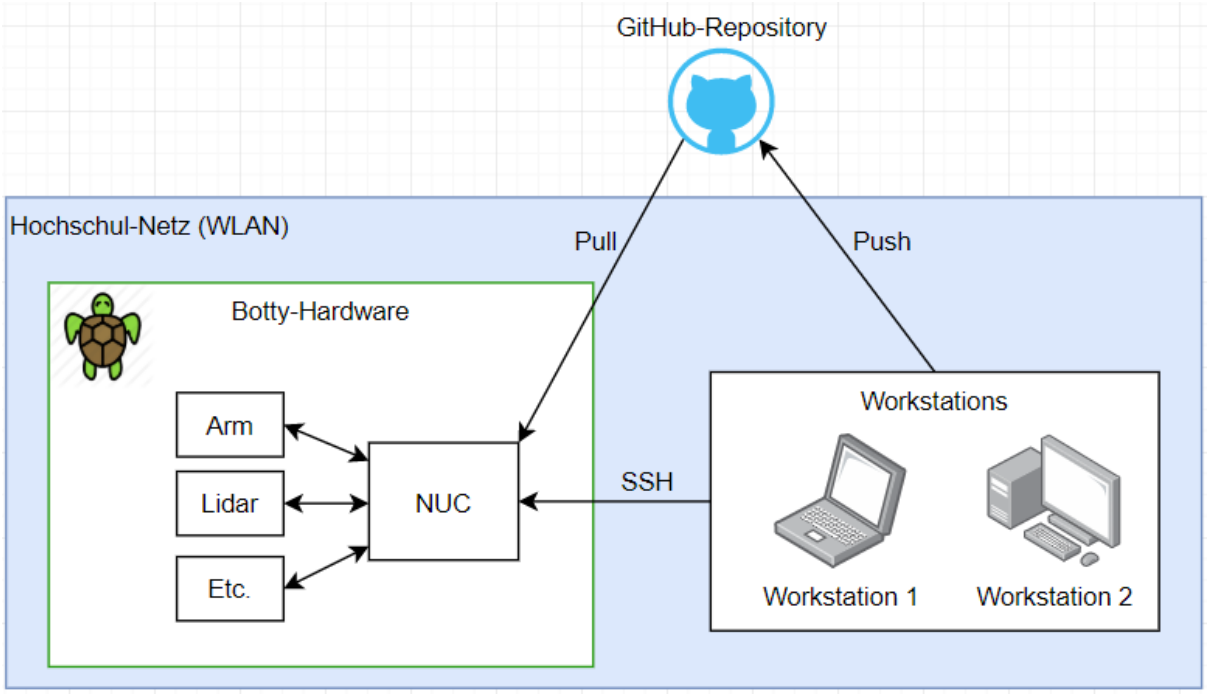
Bei manchen Befehlen funktioniert scheinbar der Abbruch nicht richtig, z.B. beim Suchen eines Objektes.

# Entwicklung und Wartung

In diesem Kapitel wird erläutert, wie man am besten eine Entwicklungsumgebung auf den Workstations einrichtet und damit den TurtleBot programmiert.

## Übersicht

Auf dem TurtleBot befindet sich ein Onboard-Rechner (NUC) mit Ubuntu 16.04 LTS. Über diesen NUC werden die entwickelten Programme des ROS-Packages gestartet und die restliche Hardware des TurtleBots angesteuert. Da der TurtleBot als Produktivumgebung für den erstellen Code dient, empfiehlt es sich nicht direkt auf dem NUC zu programmieren. Es ist besser, wenn jedes Teammitglied einen eigenen Rechner/Laptop (im Folgenden "Workstation" genannt) besitzt und dort den eigenen Code entwickelt. Der TurtleBot und die Workstations sollten sich alle im gleichen Netzwerk befinden (in unserem Fall das Netzwerk der Hochschule), damit auf den TurtleBot via SSH zugegriffen werden kann. Der Code kann dann über GitHub von den Workstations auf den TurtleBot übertragen werden.
Folgendes Schaubild zeigt den Zusammenhang zwischen TurtleBot, Workstations und GitHub:




Die Verwendung des GitHub-Repositories wird im Kapitel "Entwicklungs-Zyklus" weiter unten erläutert.

## Einrichten von Ubuntu auf Workstations

Der Turtlebot2 wird mittels dem ROS-Frameworks gesteuert und programmiert. Das Botty-Package ist mit der ROS-Version Kinetic realisiert. Das empfohlene Betriebssystem für den TurtleBot2 ist Ubuntu 16.04 LTS. Aus diesem Grund empfiehlt es sich auch auf den eigenen Workstations mit der gleichen Ubuntu-Version zu arbeiten.

Näheres zu Ubuntu unter https://ubuntu.com/download.

Es wurden mehrere Einrichtungsmöglichkeiten von Ubuntu ausprobiert, die alle im Folgenden erläutert werden. Dabei wird auch darauf eingegangen, welche Möglichkeiten

sich zur Nutzung eignen und welche nicht.
Am einfachsten wäre es, Ubuntu als alleiniges Betriebssystem eigenen Rechner installiert, jedoch wurde von Teilnehmern dieses Projekts auf den eigenen Rechnern parallel Windows benötigt, wodurch andere Alternativen gesucht wurden.

## Alternative 1 (empfohlen): Einrichten mit Dual Boot

Diese Möglichkeit ist die Performanteste, da sich Ubuntu nativ zusammen mit einem anderen Betriebssystem nach Wahl (z.B. Windows) auf dem Rechner benutzen lässt. Allerdings ist diese Alternative etwas risikoreich, da die Installation Fehleranfällig sein kann und die beiden Betriebssysteme u.U. ihren Speicher gegenseitig überschreiben könnten.

Folgende Installationsanleitung wurde hierfür befolgt: https://itsfoss.com/install-ubuntu-1404-dual-boot-mode-windows-8-81-uefi/

## Alternative 2 (u.U. verwendbar): Einrichten mit externer SSD

Diese Möglichkeit ist sehr praktisch, da der herkömmliche Dual Boot risikoreicher ist, als diese Variante. Hierbei befinden sich das ursprüngliche Betriebssystem und Ubuntu auf zwei verschiedenen, physikalisch getrennten Speichermedien. Dadurch können diese sich nicht gegenseitig den Speicher überschreiben.
Problematisch hierbei ist jedoch, dass diese Möglichkeit nicht auf allen Rechnern funktioniert, da diese die BIOS Einstellung “Legacy Boot” unterstützen müssen. Das Microsoft Surface und ähnliche Systeme tuen dies nicht. Ebenso muss der “Secure Boot” deaktiviert werden.
Für die Installation auf der SSD wurde zuerst eine bootable Ubuntu-Version auf einen USB-Stick geladen (ähnlich wie bei der dritten Variante), um anschließend Ubuntu mit dem USB-Stick auf die SSD zu installieren. Der bootable USB wurde mit “Linux Live USB Creator” erstellt.

Anleitungen hierzu finden sich unter:
Linux Live USB Creator:
https://www.linuxliveusb.com/en/help/guide
Installation von Ubuntu über einen USB:
https://wiki.ubuntuusers.de/Installation_auf_externen_Speichermedien/

## Alternative 3 (nicht empfohlen): Einrichten mit USB-Stick

Die folgende Variante ist nicht zu empfehlen:
Ein USB-Stick mit einer bootbaren Version von Ubuntu wäre eine einfache und leicht transportierbare Variante. Es besteht die Möglichkeit mit einem entsprechenden Programm Ubuntu auf einem USB-Stick zu installieren. Solche Programme, die dafür geeignet wären, sind Rufus (https://rufus.ie/) oder Linux Live USB Creator (https://www.linuxliveusb.com/en/).
Um eine solche portable Version von Ubuntu nutzen zu können, muss an dem jeweiligen Rechner der “secure Boot” in den BIOS Einstellungen deaktiviert werden. Es muss darauf geachtet werden, dass der USB persistent ist, d.h. dass Änderungen nach dem Beenden von Ubuntu auch gespeichert bleiben, dies ist jedoch bei USB-Sticks in der Regel nicht der Fall. In der Praxis hat sich gezeigt, dass die meisten USB-Sticks eine viel zu geringe Performance haben, um damit Software für den TurtleBot zu entwickeln. Der

Installationsvorgang von ROS an sich hat alleine über einen ganzen Tag gedauert.

### Alternative 4 (nicht empfohlen): Einrichten mit lokaler Virtual Machine

Die folgende Variante ist nicht zu empfehlen:
Es wurde getestet, ob eine Virtual Machine (VM) mit Ubuntu als Entwicklungsumgebung geeignet ist. Hierfür muss in den BIOS-Einstellungen des Host-Rechners die Virtualisierung aktiviert sein.
Eine der Virtualisierungsmöglichkeiten von Betriebssystemen ist VirtualBox von Oracle (https://www.virtualbox.org/). Einen Installationsanleitung findet sich unter:
https://www.heise.de/tipps-tricks/Ubuntu-in-VirtualBox-nutzen-so-klappt-s-4203333.html

Problem hierbei ist, dass die virtuelle Umgebung nicht performant genug ist. Grafikkartentreiber werden von den virtuellen Maschinen nicht erkannt, wodurch das Arbeiten mit den grafischen Oberflächen aufgrund von Verzögerungen und mangelnder Rechenleistung suboptimal ist. Des Weiteren kommt es u.U. oft zu Abstürzen der VM.

### Alternative 5 (nicht empfohlen): Einrichten mit Virtual Machine auf einem externen Server

Die folgende Variante ist nicht zu empfehlen:
Es wurde probiert, eine VM auf einem Server der Technischen Universität Kaiserslautern zu installieren. Geplant war, dass auf einem performanten Server sich die Entwickler per Remote-Desktop verbinden und dort ihren Code schreiben und testen.
Aufgrund der hohen Latenzen und Problemen mit Grafikkartentreibern ist auch hier das produktive Arbeiten sehr stark eingeschränkt.

## Einrichten einer Entwicklungsumgebung

Ubuntu bringt nativ nur minimalistische Editoren mit sich. Natürlich können diese zum Entwickeln genutzt werden, jedoch empfiehlt es sich eine richtige Entwicklungsumgebung zu installieren. In diesem Projekt wurde primär der Texteditor ‘Visual Studio Code’ von Microsoft verwendet. Der Vorteil von Visual Studio Code (VS Code) besteht darin, dass es einerseits sehr effizient und benutzerfreundlich ist, aber genauso die in diesem Projekt verwendeten Programmiersprachen (Python und C++) mit den entsprechenden Erweiterungen unterstützt.

### VS Code Installation für Ubuntu

Visual Studio Code kann unter folgender Website heruntergeladen werden:
https://code.visualstudio.com/

Nach dem Download kann es entweder im graphischen Software Center oder mit folgendem Befehl installiert werden:

```
sudo apt install ./[file].deb
```

Genaueres unter: https://code.visualstudio.com/docs/setup/linux

Alternative Lösung:
Die Vergangenheit hat gezeigt, dass bei der Installation Probleme auftreten können, da die heruntergeladene Installationsdatei nicht immer zu dem System passt (Erhalt von 64 bit Version auf 32 Bit Ubuntu Umgebung). Das Problem kann gelöst werden, indem zuvor das Package ‘umake’ installiert wird und die eigentliche Installation von “VS Code” dann mit ‘umake’ ausgeführt wird.

Installation von ‘umake’:

```
sudo add-apt-repository ppa:ubuntu-desktop/ubuntu-make
sudo apt-get update
sudo apt-get install ubuntu-make
```

Installation von “VS Code” mit ‘umake’:

```
umake ide visual-studio-code
```

## VS Code mit Python installieren

Im Folgenden wird beschrieben, wie die jeweilige Python Extension in “VS Code” installiert wird.
Wenn VS Code geöffnet ist, kann mit STRG+P die Suchleiste geöffnet werden und darin mit dem Befehl *'ext install ms-python.python*' die Erweiterung installiert werden. Nach der Installation der Erweiterung empfiehlt es sich, zuerst zu überprüfen, ob die Installation korrekt war. Hierzu reicht es die Version über das Terminal ab zu fragen:

```
python --version
```

Als Nächstes wählen wir den benötigten Interpreter für Python aus. Hierfür muss in VS Code STRG+Shift+P gedrückt werden. In der geöffneten Suchleiste ‘*Python: Select Interpreter*’ eingeben und den aktuellsten Interpreter auswählen.
Zum Erstellen von Python-Skripten, muss eine neue Datei erstellt werden, die die Python Endung ‘.py’ hat (z.B. test.py). Für gewöhnlich erscheint hier die Fehlermeldung ‘no tkinter’, sobald eine Python Datei geöffnet wird. Diese ist einfach zu lösen, indem direkt auf das Feld ‘install’ neber der Fehlermeldung geklickt wird.
VS Code ist nun einsatzbereit, um Skripte mit Python zu entwickeln.

Installationsanleitung:
https://code.visualstudio.com/docs/languages/python

## Entwicklung mit C++

Für die Entwicklung mit C++ wurde in diesem Projekt keine IDE verwendet. Entwickelt wurde mithilfe des in Ubuntu integrierten Text-Editor gedit.
Kompiliert wurde dann mithilfe von catkin_make.
Als Einstieg in die Entwicklung mit C++ von ROS wurde folgendes Tutorial verwendet: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29

Um Services in C++ zu realisieren wurde folgendes Tutorial verwendet:

http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28c%2B%2B%29

# Git-Repository

Als Versionsverwaltung wurde Git verwendet. Das ROS-Package befindet sich im Repository “botty”, welches unter folgendem Link zu finden ist: (https://github.com/MarkusDauth/botty.git).
Folgende Accounts der beteiligten Teammitglieder wurden benutzt:

| **Git-Nutzer** | **Name** |
|---|---|
| https://github.com/MarkusDauth | Markus Dauth |
| https://github.com/david-kostka | David Kostka |
| https://github.com/DeltaSence | Felix Mayer |
| https://github.com/RaschiedSlet | Raschied Slet |

# Entwicklungs-Zyklus

Im Folgenden wird erklärt, wie man den TurtleBot mit dem Inhalt des Botty-Repository programmiert und welche Vorbereitungen getroffen werden müssen. Prinzipiell wurde der Code auf einem eigenen Rechner (meist Laptops) erstellt und später vor Ort am echten TurtleBot ausgeführt.

## Allgemeine Vorbereitungen

Für die Nutzung des Botty-Packages müssen alle Installationsvorbereitungen aus dem Kapitel Kapitel Demos für vorgenommen sein. Zusätzlich müssen für die Entwicklung von ROS-Komponenten folgende Installationen vorgenommen werden:

Installation von Catkin:

```
sudo apt-get install ros-kinetic-catkin
```

Installation der TurtleBot Packages und erstellen des Workspace:

```
mkdir -p catkin_ws
cd catkin_ws
mkdir src
cd src
git clone -b kinetic https://github.com/turtlebot/turtlebot.git
cd ..
catkin_make
```

### Deployment auf dem TurtleBot (NUC)

Geschriebene Programme werden auf einem eigenen Rechner entwickelt. Die Programme werden entweder als Python-Skript mit VS Code erstellt oder in C++ mit gedit. Um die Programme auf Botty auszuführen, muss der Code mit Hilfe des Repository auf den Rechner des Turtlebot übermittelt werden. Hierfür muss zuerst der entwickelte Code in das Repository mittel folgendem Befehl geladen werden:

```
git add .
git commit -m “neuer Code”
git push
```

Befinden sich die Programme im Git-Repository, können diese anschließend auf dem TurtleBot geklont werden. Hierfür kann man entweder direkt auf dem NUC arbeiten oder man verbindet sich von seiner eigenen Workstation mittels ssh auf den NUC (siehe Kapitel Demos). Es folgt eine genauere Übersicht der Befehle, die auf dem NUC ausgeführt werden muss:

Initiales Erstellen des Repository aus GitHub im derzeitigen Pfad (einmalige Ausführung):

```
git clone https://github.com/MarkusDauth/botty.git
```

Die Dateien aus dem Repository müssen zuerst aus dem Botty Package in den Workspace übernommenen werden, bevor sie gestartet werden können. Wenn das getan ist, sollte auch erst ein Kompilierungsvorgang durchgeführt werden, um zu sehen, ob alles seine Richtigkeit hat. Dazu muss im Hauptordner des Workspace (“catkin_ws”) in einem Terminal folgende Befehl ausgeführt werden:

```
catkin_make
```

Wenn alle Pakete richtig eingebunden wurden gibt es 2 Varianten, um ein Programm zu starten unter ROS - ‘roslaunch’ und ‘rosrun’. ‘roslaunch’ erfordert eine zusätzliche Launch File. ‘rosrun’ hingegen nicht, funktioniert aber nur bei einzelnen Python Skripts.

```
roslaunch <package> <file>
rosrun <package> <file>
```

Ein häufiger Fehler, weshalb die Skripte nicht ausgeführt werden können, sind die Berechtigungen. Mit folgender Anweisung können sie freigegeben werden:

```
sudo chmod +x <file>.py
```

## Verwendung von ROS-Simulatoren

Die Simulatoren wie “Gazebo” oder “Stage” ermöglichen es Programme bereits außerhalb des Labors ohne Botty zu testen. Jedoch unterstützen diese nicht ohne weiteres alle Funktionen des Turtlebot2, wie zum Beispiel das Lidar oder der PhantomX Reactor Arm. Zudem spiegeln Simulationen die Realität nicht perfekt wieder, es kann also zu Abweichungen im Vergleich mit realen Situationen kommen.

Auf folgende weisen konnte auch ohne Simulationen mit relativ wenig Aufwand getestet werden:

1. Die Software wurde an Botty im Labor getestet
2. Oder, wenn nur einzelne Komponenten wie Kamera oder Arm gebraucht wurden, wurden sie abmontiert und mitgenommen, um sie außerhalb des Labors zu testen und weiterzuentwickeln.

Letzteres setzt voraus, dass andere Teammitglieder sie nicht benötigen und sollten nur bei vorheriger Absprache mitgenommen werden.

# Architektur und Technik

## Das Botty-Package

Das Package wurde für ROS Kinetic unter Ubuntu LTS 16.04 konzipiert.

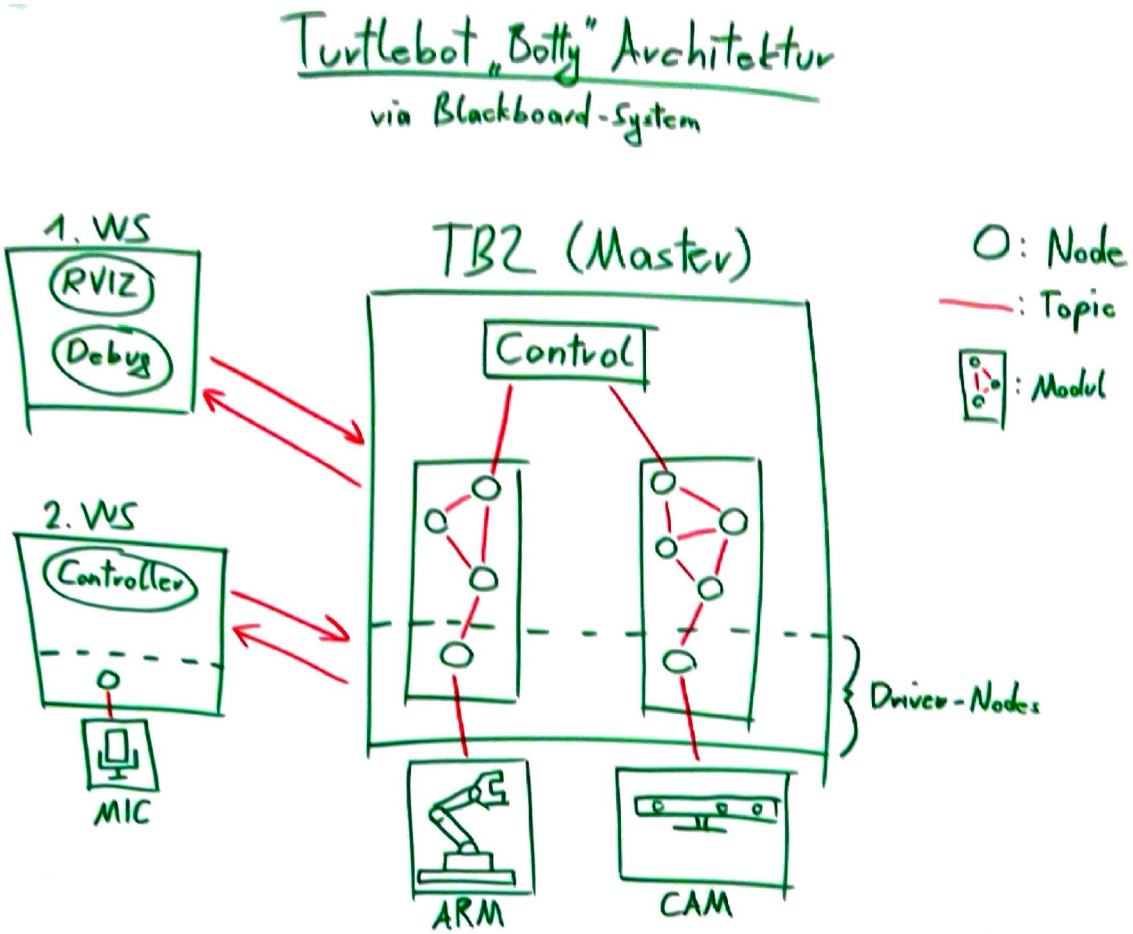
Das Endprodukt des Projektes ist ein ROS-Package namens Botty. Das Botty-Package ist in folgende Komponenten/Module/Subpackages aufgeteilt:

- Arm (Steuerung des Greifarms)
- Camera (Objekterkennung mit der 3D-Kamera)
- Controller (Zentrale Kernkomponente, welche die anderen Packages miteinander verbindet)
- Lidar (Hinderniserkennung)
- Motor (Bewegung und Objekterkennung)
- Speech (Spracherkennung und -ausgabe)

Eine Komponente des Botty-Packages beinhaltet ROS-Nodes. Die Nodes einer Komponente können nicht direkt mit den Nodes einer anderen Komponente kommunizieren. Die einzige Kommunikation zwischen den Komponenten findet immer über den Controller statt. Somit kann die Arbeitsweise von ROS im "Blackboard-Prinzip" strukturierter werden, was die Komplexität reduziert.
Als Driver-Node werden die Nodes bezeichnet, die die Schnittstelle zwischen der Hardware und des Botty-Packages bilden. Die verwendeten Driver-Nodes wurden von Dritten entwickelt und deren Packages befinden sich nicht im Botty-Repository.

Auf der nächsten Seite gibt es dazu ein Schaubild der Architektur mit kurzer Erklärung.

Die folgende Zeichnung zeigt die Kommunikation zwischen den einzelnen Komponenten und Nodes mittels Topics. Der Master(-node) befindet sich auf dem NUC des TurtleBot. Über die entsprechenden Komponenten (Module) kann jeweils der Controller die Hardware ansteuern.




In den folgenden Kapiteln werden die einzelnen Module näher erläutert. Für jede Komponente gibt es folgende Unterkapitel:

- Verwendete Packages
- Dateien
- Konfiguration
- Lessons Learned
- Potenzielle Verbesserungen

# Controller

Der Controller dient als zentrale Einheit für die Kommunikation zwischen den Modulen.

Hier werden einkommende Befehle verwaltet. Das heißt abstrakte Befehle werden:

1. Auf konkrete Funktionen und Threads abgebildet
2. in Teilaufgaben unterteilt
3. an die Module weitergegeben
4. auf das Ergebnis gewartet

Dabei wird auf ein asynchrones Abbruch-Signal reagiert.
Es wird auch die Navigation in einem Grid geplant und ausgeführt.

## Verwendete Packages

Nur das ROS-Package sound_play wird für die Audio-Rückmeldung verwendet.
Siehe: https://github.com/ros-drivers/audio_common/tree/master/sound_play

## Dateien

**Node Botty.py**

Beinhaltet die Klasse Action, welche die Datenstruktur eines Befehls darstellt.

```
class Action:
        GO = 1
        GRAB = 2
        BRING = 3
        STOP = 4
        SEARCH = 5

        SYNONYMS =      {
                        STOP: ['stop', 'halt', 'abort', 'kill', 'panic'],
                        GO: ['go', 'move'],
                        GRAB: ['grab', 'hold', 'take'],
                        BRING: ['bring'],
                        SEARCH: ['search', 'find']
                        }
```

**Node control.py**

Für den Controller gibt es folgende Wrapper-Klassen:

- Arm: Steuerung des Arms
- Camera: Abfrage der Bilderkennung

- Nav: Navigation, Steuerung des Motors

Die Wrapper-Klassen bilden eine Schnittstelle für die Kommunikation mit den Services/Topics der entsprechenden Module. Dabei wird das Senden von Befehlen, für welche der Message-Typ und Service/Topic-Name bekannt sein muss, als Methoden abstrahiert.
Code beispiel für Nav-Wrapper Methode drive():

```
def drive(self, meters, avoid=0):
        param=[]
        param.append(meters)
        if avoid:
                param.append(1)
        command=call()
        command.call="forwardByMeters"
        command.param=param
        return self.job(command.call, command.param)
```

Hier sieht man wie ein Service-Call an Motor in eine Methode gekapselt wird.
Der Controller kann dann die Nav.drive() Methode aufrufen, ohne kenntnisse über das Service haben zu müssen.

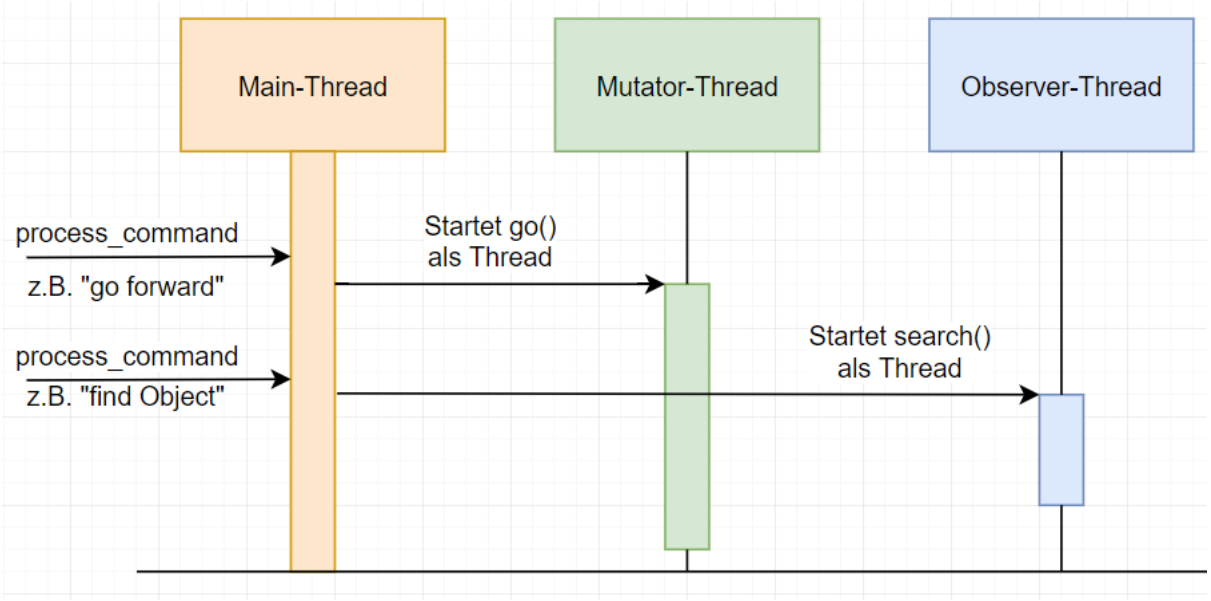
Des weiteren gibt es die Node-Klasse, die für den A*-Algorithmus verwendet wird. A* wird wiederum in Nav für die Navigation verwendet.

Die Klasse Controller fast schließlich alles in einer Steuerungs-Einheit zusammen. Es werden die Wrapper-Objekte und sound_play initialisiert, dabei wird geg. auf die Services gewartet.

Für das Ausführen von Befehlen gibt es zwei Threads:

- **mutator_thread:** Für Befehle die Botty's Zustand verändern. (z.B. Base, Arm bewegungen)
- **observer_thread:** Befehle, die parallel zu Mutator-Befehlen laufen können. (z.B. Antworten, Objekterkennung)

Es kann also nur max. 2 Befehle gleichzeitig laufen. Kommt ein neuer Befehl an den Controller an, wird ausgewertet, um welches Befehl es sich handelt (Action-ID). Je nach Aktionstyp wird entweder ein mutator oder observer Thread gestartet.




Wenn der Thread nicht bereits belegt ist, wird der Befehl mit entsprechenden Parametern asynchron ausgeführt. Wenn er aber belegt ist, wird der neue Befehl verworfen, mit entsprechender Fehlermeldung. Es gibt also keine Warteschlange oder dergleichen.

In der Funktion, die asynchron als Thread ausgeführt werden soll, werden sequentiell Befehle an die anderen Komponenten des Systems (durch die Wrapper-Objekte) gesendet. Dabei wird auf dann auf eine erfolgreiche Rückgabe gewartet. Bei einem Misserfolg wird meistens der gesamte Thread bzw. der Befehl abgebrochen.

Ausnahme: Bei einem Stop-Befehl wird der Signal asynchron an die Wrapper-Objekte weitergegeben, welche die Komponenten darüber informieren. Danach wird auf das Terminieren der Threads gewartet.

**Message Command.msg**
Bildet mit Action-ID und Entity eine Struktur, die ein Befehl darstellt.

**Message Entity.msg**
Besteht aus Name und Liste von Attributen.

## Lessons Learned

Man sollte sich früher auf Schnittstellen der Module einigen, um abgekapselte Struktur zu ermöglichen. Damit kann man mit Dummy-Befehlen arbeiten, die nicht auf eine (fehlende) Implementierung abhängig ist.

## Potenzielle Verbesserungen

control.py in mehere Dateien aufteilen, eine für jede Klasse.
Die Funktionen, die als Befehl im Thread ausgeführt werden, könnten in eine Hilfsklasse ausgelagert werden.
Bessere Nebenläufigkeit, bis jetzt können nur zwei Threads gleichzeitig laufen. Es könnten die Komponenten durch gegenseitigen Ausschluss optimaler parallel genutzt werden und damit die Befehle nicht in Observer/Mutator aufgeteilt werden.

# Arm

Dieses ROS-Package dient zur Steuerung des PhantomX Reactor Arm. Ziel war es, dass der Arm Objekte, die von der 3D-Kamera erkannt werden, gegriffen werden und diese Objekte wieder an anderer Stelle abgelegt werden. Jedoch traten während des Projektes mehrere schwerwiegende Probleme mit dem Greifarm auf, weswegen der Arm nur fest programmierte Posen annehmen kann (siehe Lessons Learned).

## Hardware

Der PhantomX Reactor Arm ist ein ROS-kompatibler Greifarm, welcher bereits aufgebaut mit dem TurtleBot2 geliefert wurde. Der Greifarm wird über einen Arbotix-M gesteuert, welches ein modifiziertes Arduino-Board zur Ansteuerung von Servo-Motoren ist. Auf dem NUC des TurtleBot kann eine ROS-Node gestartet werden, welche Positionierungsbefehle für den Greifarm annimmt und diese Befehle dann an das Arduino-Board weiterleitet. Hierfür befindet sich ein Arduino-Image auf dem Arbotix-M, welches die ROS-Befehle interpretiert und entsprechend die Servo-Motoren ansteuert.

In diesem Projekt wurde der PhantomX Reactor Arm "mit Wrist" verwendet.

**Achtung vor Benutzung:**
Der Greifarm benötigt relativ viel Strom, weswegen das USB-Kabel direkt an den Onboard-Computer des TurtleBots angeschlossen werden muss (nicht über den USB-Hub anschließen)
Wenn der Arm direkt über die ROS-Schnittstellen positioniert werden soll, muss dies mit großer Vorsicht gemacht werden. Es können Befehle gesendet werden, die zu große Winkel für die Gelenke verursachen und somit sich die Motoren verhaken oder sich Kabel lösen.
Je nach vorgesehener Armposition, kann sich der Arm an der Platform des TurtleBots oder sogar mit eigenen Armgelenken verhaken oder kollidieren. Es ist wichtig zu überprüfen, in welcher Reihenfolge die einzelnen Gelenke des Arm angesteuert werden, um diese Probleme zu vermeiden.

## Funktionsweise

Es wird der ROS-Service "/botty/arm/commands" zur Verfügung gestellt, wodurch der Arm zwei vordefinierte Bewegungen durchführen kann:

Bewegung "home": Der Arm begibt sich in seine "Ruheposition" und bleibt in dieser.
Bewegung "push": Der Arm bewegt sich nach vorne und führt eine "Wischbewegung" von Links nach Rechts durch.

## Verwendete Packages

Die Positionierung des Arms erfolgt über das ROS-Package "phantomx_reactor_arm": https://github.com/RobotnikAutomation/phantomx_reactor_arm

Mit diesem Package ist es möglich, die Gradwinkel für die Gelenke des Arm direkt anzugeben (andere Möglichkeiten zur Ansteuerung gibt hiermit leider nicht). In dem Github-Repository befindet sich außerdem eine Installationsanleitung und es wird gezeigt, wie man den Arm steuert.

Achtung: Mit diesem Package werden die Positionierungen der einzelnen Gelenke direkt vorgenommen. Es ist möglich, die Gelenke des Arm zu übersteuern oder Kollisionen mit eigenen Armteilen zu verursachen. Generell sollte vor Testen von Code überlegt werden, welche Armgelenke sich zuerst bewegen müssen, damit keine Kollisionen stattfinden. Um Übersteuerungen zu verhindern, sollten nur Werte für die einzelnen Gelenke zwischen -1,5 und +1,5 verwendet werden.

Troubleshooting: Falls die Ansteuerung mit diesem Package nicht funktioniert, ist es empfehlenswert, die komplette Installationsanleitung des "phantomx_reactor_arm"-Repositories für Arobitx-M nochmal durchzuführen.

## Dateien

**nodes/arm_control.py**

Diese Datei stellt die Node mit den oben genannten Funktionen zur Verfügung. Über das ROS-Package "phantomx_reactor_arm" wird der Arm angesteuert. Wenn der Arm in einer Bewegung mehrere Positionen ansteuern soll, müssen die sleep-Aufrufe benutzt werden, um Kollisionen zu vermeiden. Folgendes Bild zeigt

Zur einfacheren Handhabung der phantomx_reactor_arm-Nodes, wird ein Dictionary Objekt “pub” verwendet, wodurch nicht die kompletten Node-Namen wie (z.B. “/phantomx_reactor_controller/wrist_roll_joint/command”) im sonstigen Programm referenziert werden müssen. Folgendes Bild zeigt den Aufbau dieses Objektes:

```
self.pub = { "elbow_pitch" : rospy.Publisher('/phantomx_reactor_controller/elbow_pitch_joint/command', Float64, queue_size=10),
    "shoulder_pitch" : rospy.Publisher('/phantomx_reactor_controller/shoulder_pitch_joint/command', Float64, queue_size=10),
    "shoulder_yaw" : rospy.Publisher('/phantomx_reactor_controller/shoulder_yaw_joint/command', Float64, queue_size=10),
    "wrist_pitch" : rospy.Publisher('/phantomx_reactor_controller/wrist_pitch_joint/command', Float64, queue_size=10),
    "wrist_roll" : rospy.Publisher('/phantomx_reactor_controller/wrist_roll_joint/command', Float64, queue_size=10)
}
```

```
def push(self):
    self.pub["elbow_pitch"].publish(-0.8)
    self.pub["shoulder_pitch"].publish(1.5)
    self.pub["shoulder_yaw"].publish(1)
    self.pub["wrist_pitch"].publish(0.5)
    self.pub["wrist_roll"].publish(0.0)
    rospy.sleep(4)
    self.pub["wrist_pitch"].publish(-0.8)
    self.pub["elbow_pitch"].publish(-1)
    rospy.sleep(2)
    self.pub["shoulder_yaw"].publish(-1)
    rospy.sleep(3)
```

Das Bild links zeigt den Bewegungsablauf des Armes für die Bewegung zum Umstoßen eines Objektes. Der Arm nimmt hierbei nacheinander drei vordefinierte Positionen an, die durch sleep-Aufrufe getrennt sind.

Sobald die Daten an ein Gelenk gesendet werden, bewegt sich das entsprechende Gelenk. Über die sleep-Methode muss dementsprechend genug Verzögerung eingebaut werden, damit der Arm nicht direkt die finale Position annimmt, sondern die Positionen nacheinander ansteuert. Die sleep-Dauer muss durch manuelles Testen festgelegt werden. Am Ende jeder Bewegung sollte ebenfalls ein sleep-Aufruf erfolgen, damit der Controller entsprechend lange auf Bewegung des Armes wartet.

**named_pose.py**
Diese Datei dient als Beispielvorlage für die potentielle Verwendung des MoveIt-Packages. Es müssen die entsprechenden Posen mit dem MoveIt-Setup-Assistant eingestellt werden. Der Arm lässt sich jedoch nicht mit dem MoveIt-Package steuern, weswegen diese Datei derzeit nicht produktiv verwendet wird. (Siehe Lessons Learned / MoveIt).

## Lessons Learned

Der Umgang mit dem PhantomX Reactor Arm gestaltet sich als sehr schwierig. Während des Projektes gab es zahlreiche Probleme, bei denen etliche Lösungen probiert wurden, jedoch war es zum Schluss des Projektes nur möglich, den Arm mittels hardgecodeten Posen per ROS zu steuern. Online-Dokumentation gibt es für den PhantomX Reactor nur wenige und alle verwendbaren ROS-Packages für den Arm sind sehr schlecht dokumentiert.

**Probleme mit der Installation und Arbotix-M**
Der Arm konnte nach Anlieferung nicht direkt benutzt werden. Sollte der Arm nicht über das "phantomx_reactor_arm"-Package angesteuert werden können, so sollte am besten nochmal die komplette Installationsanleitung für den Arm vorgenommen werden. Siehe: https://github.com/RobotnikAutomation/phantomx_reactor_arm

Es ist darauf zu achten, dass das richtige Arduino-Image auf den Arm gespielt wird und dass die richtigen udev-Regeln auf dem NUC eingestellt sind.

Sollte sich das Arduino-Image nicht auf den Arbotix-M installieren lassen, so könnte es sein, dass das Arduino-Board defekt ist. Dieses kann durch einen neuen Arbotix-M einfach ausgetauscht werden. Zum Testen, ob der Arbotix-M noch richtig funktioniert, dient folgende Anleitung des PhantomX-Herstellers:
https://learn.trossenrobotics.com/arbotix/7-arbotix-quick-start-guide

Das Arbotix-M-Board musste während des Projektes einmal ausgetauscht werden.

**MoveIt**
Zur einfacheren Steuerung der Arms wurde probiert, das MoveIt-Package zu verwenden: http://docs.ros.org/kinetic/api/moveit_tutorials/html/index.html

Das MoveIt-Package benötigt jedoch entsprechende ROS-Controller, die es für diesen Greifarm nicht gibt. Eine manuelle Anpassung des MoveIt-Packages gestaltete sich als zu komplex. Es könnte möglich sein, den Greifarm über MoveIt mittels des Adapters "USB2Dynamixel" zu steuern, jedoch ist dies so gut wie nich nicht dokumentiert.

**Open Source Programm - Interbotix Link Software**

Es gibt ein Open-Source Java Programm, welche Funktionen zur einfacheren Steuerung des Arms anbietet:
https://learn.trossenrobotics.com/36-demo-code/137-interbotix-arm-link-software.html

Es wurde versucht, Funktionalitäten der InterbotiX Arm Link Software in einem eigenen Java Konsolenprogramm zu verwenden. Die InterbotiX Arm Link Software ist jedoch sehr stark an das "Processing" Framework gebunden. Über das Processing Framework wird sowohl die GUI, wie auch die Steuerung des Arms realisiert. Es ist nicht möglich, Teile der Software ohne Processing zu verwenden, wodurch eine Verwendung dieser Software auf dem TurtleBot nicht als hilfreich zu erachten ist.

### Potenzielle Verbesserungen

Für den Arm könnten mehrere Posen in dem Skript nodes/arm_control.py konfiguriert werden. Außerdem könnte der Arm eventuell so programmiert werden, dass er sich je nach erkannten Objekt andere Posen annimmt. Ein Greifen von Objekten ist jedoch mit dem aktuell verwendeten Package schwierig umzusetzen, da die Winkel aller Armgelenke manuell programmiert werden müssen.

Es könnte ein USB2Dynamixel gekauft werden, welcher eventuell eine Steuerung des PhantomX mit MoveIt ermöglichen. Dadurch könnte das tatsächliche Greifen von Objekten, die die 3D-Kamera erkennt, ermöglicht werden.

Eine Alternative für den PhantomX Reactor wäre der WidowX Arm: https://learn.trossenrobotics.com/projects/186-widowx-arm-with-ros-getting-started-guide.html

Für diesen Arm gibt es mehr Dokumentationen und Hilfestellungen online. Er lässt sich außerdem mit dem MoveIt-Package von Haus aus steuern. Siehe: https://github.com/Interbotix/widowx_arm

## Camera

Dieses Modul kommuniziert mit dem Controller und gibt diesem Informationen darüber, ob und welche Objekte erkannt werden.

Dies wurde auf zwei Arten realisiert:

- Der Controller gibt einen Auftrag.

- Wenn Objekte erkannt werden, veröffentlicht das Kamera-Modul auf einem ROS-Topic was gerade gesehen wird.

Im Projekt wurde die Orbbecc Astra 3D Kamera verwendet. Die Spezifikationen der Kamera sind zu finden unter: https://orbbec3d.com/product-astra-pro/

## Verwendete Packages

**astra_camera:**
Hierbei handelt es sich um das Package, welches es erlaubt, die Kamera in ROS zu verwenden. Folgendes Tutorial wurde befolgt:
https://github.com/orbbec/ros_astra_camera

**find_object_2d:**
Dieses Package wurde für die Bilderkennung verwendet. In Kombination mit diesem Package wurde folgendes Tutorial verwendet:
https://husarion.com/tutorials/ros-tutorials/4-visual-object-recognition/

## Dateien

**camera_controller.cpp:**
Hier findet die Auswertung der Bilderkennung statt. Es wurden zwei ROS-Kommunikationen realisiert:

- Anfrage zur Bildauswertung als ROS-Service. Dies wurde für den Controller verwendet.
- Wenn das das Modul ein Objekt erkennt, veröffentlicht es die Objektinformationen als ROS-Topic.

**camera.launch:**
Hier werden alle nötigen Nodes zur Bilderkennung gestartet.

**FindObjects.srv:**
in dieser Datei wird der Rückgabetyp des Services definiert. Es wird eine Liste von Strings mit den erkannten Objekten an den Aufrufenden zurückgegeben.

**Ordner image_rec:**
In diesem Ordner werden die antrainierten Objekte im .png format gespeichert.

Es wurden folgende Objekte antrainiert:

- Cola Flasche
- Roter Pfeil
- Kreideschachtel
- Blume
- Zeitschrift

**Ordner auswertung:**
In diesem Ordner findet sich eine Auswertung in Form einer Konfusionsmatrix, bei der es darum geht, wie gut die Bilderkennung funktioniert. Die Auswertung ist im .pdf und .xlsx Format im Ordner.

## Konfiguration

Um neue Objekte anzutrainieren:

1. Gehe zu "Edit" -> "Add object...",
2. Objekt vor die Kamera halten.
3. Auf "Take Picture" klicken.
4. Das Objekt im Bild markieren und bestätigen.
5. Um die object_id zum Namen des Objektes zu mappen, muss in die Datei camera_controller.cpp über der Funktion set_object_ids eine neue Funktion erstellt werden. Diese Methode soll true zurückliefern, wenn die übergebene object_id, der object_id des gewünschten Objekts entspricht. Den Wert der object_id kann man über den image_rec Ordner herausfinden. Der Dateiname des zu erkennenden Bildes entspricht der object_id.
6. In der Methode object_id_to_string muss ein if-else hinzugefügt werden, indem die vorher erstellte Funktion mit der object_id aufgerufen wird.

Um beispielsweise einen Löffel mit den object_id's 76,77,78 anzutrainieren wird empfohlen bei den Schritten 5 bis 6 auf folgende Weise vorzugehen :

Schritt 5:
Siehe Code-Fragment rechts zur Methode equals_spoon()

```
bool equals_spoon(int object_id){
    if(object_id == 76 ||object_id == 77 || object_id == 78)
        return true;
    return false;
}
```

Schritt 6:
Dies zur Methode object_id_to_string hinzufügen. Modifiziert sieht die aktuelle Methode so aus.

```
std::string object_id_to_string(int object_id){
    if(equals_cocaCola_id(object_id))
        return "Coca Cola Bottle";
    else if(equals_chalk_id(object_id))
        return "Chalk";
    else if(equals_shall_welten_magazine(object_id))
        return "Schall welten Magazin";
    else if(equals_flower(object_id))
        return "Flower";
    else if(equals_red_arrow(object_id))
        return "Red Arrow";
    else if(equals_spoon(object_id))
        return "Spoon";
    else
        return "invalid";
}
```

## Lessons Learned

Während des Projektes wurde mit vielen Frameworks experimentiert, mit dem Ziel das Handling mit der Objekterkennung zu erleichtern. Die Probleme gingen jedoch in den meisten Fällen darauf zurück dass entweder zusätzliche Hardware benötigt wird, oder dass die Nutzung auf Grund der Komplexität der Packages viel zu umständlich ist. Das größte Problem lag daran, dass die Packages zu viel Funktionalität mit sich mitbrachten, welche für das Projekt nicht nötig sind und dadurch viel Overhead verursachten.

Folgende Frameworks wurden getestet:

- Orbbec API:
  - https://orbbec3d.com/develop/
  - Aufgrund der mangelnden Dokument ist es uns nicht gelungen, die Orbbec-Software auf dem TurtleBot zu installieren.
- Joffmann:
  - https://github.com/joffman/ros_object_recognition
  - Bei unseren Tests wurden mehrere Objekte mit diesem Framework antrainiert, jedoch war die Erkennungsquote in allen Fällen sehr niedrig.
- Tensorflow:
  - https://www.tensorflow.org/
  - Tensorflow konnte nicht genutzt werden, da die Trainingsdatensätze sehr groß sind und aufgrund dessen aus Sicht der gegebenen Rechnerleistung nicht anwendbar ist.
- Darknet Yolo:
  - https://pjreddie.com/darknet/yolo/
  - Wäre nutzbar jedoch wird hier eine Cuda konforme Nvidia GPU benötigt, die der TurtleBot jedoch nicht besitzt.

Die weitere Nutzung des find_object_2d packages ist unter Anderem aufgrund der Erkennrate nicht zu empfehlen. Es wurden während des Projektes viele Objekte eingespielt, welche jedoch aufgrund ihrer Größe und oder Komplexität nicht erkannt wurden. Aufgrund dessen wurde entschieden während des Projektes das gleiche Objekt mit mehreren Bildern zu verknüpfen. Dies ist jedoch nicht empfehlenswert da man bei find_object_2d mehrere Objekte nicht auf mehrere Objekte abbilden kann. Dies muss man dann als Programmierer entsprechend beachten.

## Auswertung der Objekterkennung

Zur Auswertung der Objekterkennung mit find_object_2d wurde eine Konfusionsmatrix verwendet. In der folgenden Tabelle wird gezeigt, was bei der Objekterkennung erkannt wurde, abhängig davon ob ein bzw. welches Objekt vor die Kamera gehalten wurde.

| | **Was wurde erkannt?** | | | | | |
|---|---|---|---|---|---|---|
| | Cola Flasche | Roter Pfeil | Blume | Zeitschrift | Kreideschachtel | Nichts |
| Cola Flasche | 1 | | | | | |
| Roter Pfeil | | 1,2,3,4,5 | | | | |
| Blume | | | 1.2,5 | | | |
| Zeitschrift | | | | 1,3,4,5 | | |
| Kreideschachtel | | | | | | |
| Nichts | 2,3,4,5 | | 3,4 | 2 | 1,2,3,4,5 | 1,2,3,4,5 |

| Test | Erläuterung |
|---|---|
| 1 | Ohne äußere Einflüsse |
| 2 | Nur die Hälfte des Objekts ist sichtbar. Hierzu wird das Objekt am Rand des Sichtfeldes der Kamera gestellt (links aus Sicht der Kamera) |
| 3 | Nur die Hälfte des Objekts ist sichtbar. Hierzu wird das Objekt am Rand des Sichtfeldes der Kamera gestellt (rechts aus Sicht der Kamera) |
| 4 | Dunkle Lichtverhältnisse (Licht im Raum O 028 ausgeschaltet) |
| 5 | Helle Lichtverhältnisse (mit dem Blitzlicht meines Handys auf das Objekt geleuchtet |

Interpretation der Daten:

**Cola Flasche:**
Die Cola Flasche ist allgemein schwer zu erkennen. Dies liegt an der Komplexität des Objektes, da je nachdem in welchem Licht bzw. aus welchem Winkel das Objekt gehalten wird, es sein kann, dass das Objekt erkannt wird oder nicht. Ein weiterer Punkt ist, dass die Flasche leer ist. Es wäre möglich die Erkennungsrate durch das Antrainieren einer vollen Cola-Flasche zu erhöhen. Ein weiterer Lösungsversuch war es, das Objekt von verschiedenen Winkeln aus anzutrainieren. Dieser Versuch verlief erfolgreich.

**Roter Pfeil:**
Der rote Pfeil hatte eine sehr hohe Erkennrate. Dies hängt damit zusammen, dass es sich um ein “2D” Objekt handelt. Damit ist gemeint dass der Pfeil keine 3 - dimensionale Form hat, sondern nur eine Zeichnung ist. Ein weiterer Grund warum das Objekt gut erkannt wird, ist die große Anzahl an Kanten, welche im Objekt vorhanden sind. Dadurch entstehen viele leicht identifizierbare Eigenschaften für das Framework (Features).

**Blume**:
Die Blume hat eine gute Erkennrate. Dies hängt mit dem gleichen Grund wie bei dem Pfeil zusammen, also dass es sich hierbei um eine 2D-Zeichnung handelt. Test 3 schlug fehl, weil mehr Features auf der linken Seite sind. Dunkle Lichtverhältnisse machen die Features die zum Identifizieren des Objektes benötigt werden, nicht sichtbar weswegen dieser Test fehlschlug. Es wurde auch probiert, das Objekt explizit in dunklen Lichtverhältnissen anzutrainieren. Dieser Versuch schlug aufgrund mangelnder Features fehl.

**Zeitschrift:**
Die Zeitschrift hat eine gute Erkennrate. Dies hat die gleichen Gründe wie beim roten Pfeil.

**Kreideschachtel:**
Die Kreideschachtel hat eine sehr schlechte Erkennrate. Dies hat mit der Größe des Objektes zu tun. Es wird aufgrund der Größe des Objekts zu wenig Features erkannt. Es wurde probiert, das Objekt anzutrainieren, indem das Objekt möglichst nah an die Kamera gehalten wird. Dies funktioniert jedoch nur, wenn man das Objekt auf sehr kleiner Distanz erkennen will. Da nicht sichergestellt werden kann, dass zu erkennende Objekte unmittelbar vor der Kamera sind, wurde dieser Versuch abgebrochen.

Beim Testen des Frameworks traten keine falsch-positiv Fälle auf.

Empfehlungen:
Für das Antrainieren von Objekten wird empfohlen, dass diese möglichst groß sind. Es ist von Vorteil, Zeichnungen / Bilder anzutrainieren. Wenn jedoch ein 3D Objekt antrainiert wird, ist es empfohlen das Objekt aus mehreren Winkeln mit hilfe von mehreren Bildern anzutrainieren.

### Potenzielle Verbesserungen

Die Weiterentwicklung dieses camera-Packages ist nicht empfohlen. Die tatsächliche Bilderkennung kann man im derzeitigen Stand nicht gut beeinflussen. Was in diesem Package gemacht wird ist, die von der Bilderkennung erhaltenen Daten zu interpretieren.

Aufgrunddessen empfehle ich die Bilderkennung mithilfe von OpenCV(https://opencv.org/) selbst zu realisieren.

Neben dem Mapping von object_id zu Name des Objektes wurde findet sich im Code eine Positionsbestimmung von Objekten, also ob das Objekt aus Sicht der Kamera links oben, rechts oben, links unten oder rechts unten ist. Da diese Funktion im Rahmen des Projektes jedoch nicht mehr verwendet wurde, wurde diese Funktion ausgeschaltet, ist jedoch im Code noch zu finden.

## Lidar

Dieses ROS-Package dient zur Steuerung des Lidar Hokuyo URG-04LX-UG01. Es wird eine Auswertung der Lidar-Werte gemacht, um die Umgebung auf Hindernisse zu überprüfen.

### Hardware

Das Lidar ist ein Hokuyo URG-04LX-UG01. Es handelt sich hierbei um einen Laserscanner zur Berechnung von Distanzen auf einer horizontalen Ebene. Der Wahrnehmungswinkel beträgt 240° und die Wahrnehmungsreichweite ist bis zu ca. 4 Meter.

Genauere Spezifikationen unter: https://www.hokuyo-aut.jp/search/single.php?serial=166

### Funktionsweise

Alle Objekte, die sich in einem vordefinierten Radius des Lidar befinden, werden als Hindernisse erkannt. Der genaue Abstand ist in den Konfigurationen beschrieben. Alle erkannten Hindernisse werden in drei Richtungsgruppen eingeordnet - Links, Rechts und Vorne. Ergebniswerte, welche aus mehreren Listen bestehen, können entweder auf Anfrage (ROS-Service) oder durch ständige Benachrichtigung (ROS-Message) erhalten werden. Für jede Richtungsgruppe gibt es zwei Listen: Eine Liste gefüllt mit Gradzahlen, die die exakte Positionierung der Hindernisse darstellen und eine Liste mit den Distanzen (in cm) zu den jeweiligen Hindernissen. Zu beachten ist, dass die Zugehörigkeit der Gradzahlen zu den Distanzen durch die identische Indexstelle gegeben ist. Wenn beispielsweise an Stelle 7 der Gradzahl-Liste für links ein Hindernis hinterlegt ist, ist die zugehörige Distanz ebenso an

dem 7. Index in der Distanz-Liste für links hinterlegt.

Um die Ergebnisse selbst einzusehen kann man in einem eigenen Terminal folgendes eingeben:

```
rostopic echo /botty/hokuyoInterpreter
```

Nun sollten alle erhaltenen Ergebnisse angezeigt werden. Diese können erprobt werden, indem Objekte in das Sichtfeld des Lidar gehalten werden.

## Verwendete Packages

Daten werden vom Lidar selbst über das 'Hokuyo Node' package ausgelesen und dann in diesem Botty-Package (lidar) weiter verarbeitet. Dazu ist ebenso das Package 'Driver Common' notwendig, da 'Hokuyo Node' allein nicht auf ROS-Kinetic lauffähig ist.

Genauere Infos unter:
https://github.com/ros-drivers/driver_common.git
http://wiki.ros.org/hokuyo_node
https://idorobotics.com/2018/11/02/integrating-the-hokuyo-urg-lidar-with-ros

## Dateien

**hokuyoInterpreter.py**
Nimmt eine Auswertung der Daten des Lidars vor und erfüllt die oben beschriebenen Funktionsweisen.

Beim Start werden die externen Konfigurationsdaten geladen. Danach findet eine Kalibrierungsphase statt, wodurch die Halterungen und andere Bauteile des Turtlebots erkannt und gespeichert werden, damit diese im weiteren Verlauf ignoriert werden und somit nicht als Hindernisse wahrgenommen werden. Alles was während der Kalibrierungsphase in einer vorgegeben Reichweite ist, wird im Folgenden ignoriert. Standardmäßig beträgt die Scanreichweite für die Kalibrierung 25 cm. Die ignorierten Elemente werden als Gradrichtung in eine Liste toter Winkel übernommen, wie im Codeausschnitt zu sehen ist.

```
rospy.loginfo("-Starting Callibration Session with Callibration-Range "+str(callibration)+"meters -")
messurements=len(data.ranges)
for counter in range(0,messurements):
    if data.ranges[counter]<=callibration:
        if counter not in deathAngles:
            deathAngles.append(counter)
        if counter-1 not in deathAngles:
            deathAngles.append(counter-1)
        if counter+1 not in deathAngles:
            deathAngles.append(counter+1)
rospy.loginfo("Death Angles are...")
rospy.loginfo(deathAngles)
```

Genauso werden die benachbarten Winkel als tote Winkel betrachtet, da bei einem unruhigen Fahrstil der eigentliche tote Winkel temporär geringfügig verrutschen könnte. Nach der Kalibrierungsphase werden die gepublishten Daten des Lidars

entgegengenommen. Die Rohdaten, die andauernd von der 'hokuyo node' veröffentlicht werden, werden direkt verarbeitet, sobald diese dem 'hokuyoInterpreter' bekannt werden. Da diese nur eine Liste von Zahlen sind mit ungenauer Listenlänge, wird zunächst eine Umrechnung des Längenverhältnis vorgenommen, um zu ermitteln, welche Indizes der Liste welchen Gradzahlen entsprechen. Die Voraussetzung hierfür ist, dass die Messungen in gleichmäßigen Abständen stattfinden. Nach der Zuordnung der Gradzahlen, wird anschließend eine Aufteilung in die Richtungsgruppen Links, Rechts und Vorne vorgenommen. Anschließend wird geprüft welche der Abstandswerte unter einen vordefinierten Grenzwert, welcher den minimalen Sicherheitsabstand entspricht, fallen. Alle Abstandswerte, die unter diese Grenze fallen, werden mit Distanz, Gradrichtung und Richtungsgruppe anschließend als ROS-Message veröffentlicht. Der Grenzwert selbst ist nicht konstant und ist von den Konfigurationswerten abhängig. Standardmäßig beträgt der Grenzwert mindestens 5 cm und maximal 35 cm. Genaueres in den Konfigurationen. Der folgende Code Ausschnitt zeigt die Berechnung des Grenzwerts, der im Code als “siteDistance” bezeichnet wird.

```
#if the sites of an angle are defined
if siteIncrement!=0:
    #if on the left site of the messurement area, increase the 'siteIncrement'
    if (degree<(messurementsSites+messurementsStart)):
        siteDistance+=siteIncrement
    #if on the rigth site of the messurement area, reduce the 'siteIncrement'
    if ((messurementsEnd-messurementsSites)<=(messurementsStart+(degree-messurementsStart))):
        siteDistance-=siteIncrement
else:
    siteDistance=collisionBlocker
```

Zu beachten ist, dass der Grenzwert verschiedene Längen hat, je nachdem, ob die Seiten einer Richtungsgruppe in der Konfigurationsdatei größer als 0 oder exakt als 0 angegeben wurden.
Das Ergebnis des Lidarzykluses kann per ROS-Service abgefragt werden oder über einen ROS-Message Dienst kontinuierlich überprüft werden.

**config.xml**
Beinhaltet die Konfigurationsdaten, die für hokuyoInterpreter.py benötigt werden.

**msg/Hints.msg und srv/HintsService.srv**
Stellen die für ROS notwendigen Message- bzw. Service-Definitions dar.

## Konfiguration

Jegliche Einstellungsmöglichkeiten werde in "config.xml" vorgenommen. Änderungen hier benötigen keine weiteren Code-Anpassungen.

Prinzipell gibt es drei Richtungsgruppen (Links, Rechst, Vorne), die durch Start- und Endpunkt definiert sind, wobei die Punkte Gradzahlen sind, die in dem Wahrnehmungsbereich des Lidar (240°) liegen. Jede der drei Richtungsgruppen hat auch eine definierte Seitengröße. An den Seiten einer Richtungsgruppe reduziert sich dann der Grenzwert immer weiter, desto näher die Messung am Start- und Endpunkt der Richtungsgruppe ist. Ist die Seitengröße in den Konfigurationen auf 0 gesetzt hat sie keine

Auswirkung und der Grenzwert ist konstant. Ist sie größer 0 wird nur die Länge reduziert, die sich aus dem Konfigurationswert 'collision' ergibt. Der Wert 'constantSecureRange' wird nicht verringert. Der eigentliche Grenzwert ist die Summe aus 'collision' und 'constantSecureRange'.

Folgendes Bild beschreibt die aktuelle Konfiguration des Lidar-Subpackages:

```
<configs>
    <angles>
        <angle start="1" end="60" site="0">rigth</angle>
        <angle start="180" end="240" site="0">left</angle>
        <angle start="30" end="210" site="45">front</angle>
    </angles>
    <callibration range="0.25"/>
    <constantSecureRange range="5"/>
</configs>
```

## Lessons Learned

Wird ein Hindernis erkannt, ist zunächst unklar, in welcher Gradrichtung es sich befindet, weshalb zuerst die Grachrichtung berechnet werden muss. Die Praxis hat gezeigt, dass die Berechnung der entsprechenden Gradzahl nicht einfach ist, da das Lidar mehrere Messwerte für eine Gradzahl veröffentlicht. Auch wenn das Lidar einen Messbereich von 240° hat, werden nicht 240 Ergebnisse geliefert (im Test waren es 512). Der Algorithmus wurde soweit angepasst, dass er unabhängig zu den gemessenen Werten eine Zuordnung zu Gradzahlen vornehmen kann, vorausgesetzt die Messungen finden in gleichmäßigen Abständen statt.

Des weiteren sollte unbedingt beim Start des TurtleBots darauf geachtet werden, dass nicht mehrere Kabel vom Turtlebot die Sicht des Lidar versperren, da sonst während der Kalibrierungsphase sehr viele tote Winkel entstehen.

## Potenzielle Verbesserungen

Von Interesse wäre eine Kartenerstellung mit dem Lidar. Dies könnte auch unabhängig von dem derzeitigen Code erfolgen. ROS stellt hierfür bereits diverse Tools zur Verfügung, aber bei unseren Tests hat sich gezeigt, dass viele dieser Tools nicht ohne großen Aufwand mit dem Turtlebot2 kompatibel sind. Wenn keine passende alternativen Tools gefunden werden, müsste die Kartenerstellung komplett manuell gemacht werden.

# Speech

Dieses Modul dient zur Sprachsteuerung des Roboters. Gedacht ist dabei ein Command-and-Control Prinzip. Im Grunde besteht es aus 3 Teilaufgaben:

**Spracherkennung**

- Interpretation der erkannten Phrasen
- Absenden der Befehle zum Controller
- In diesem Abschnitt wird kurz erläutert, wie das Modul grob funktioniert.
- im Abschnitt "Dateien" wird näher auf die Implementierung eingegangen.

Einige Beispielsätze: "go forward" "go to the kitchen" "find a flower" "grab the cup"

**Interpretation / Parsing**

Mit dem erkannten Text als Eingabe interpretiert der Parser den semantischen Inhalt des Texts.
Ein gültiger Befehl wird als Datenstruktur in Form einer art Prädikat an den Controller übergeben.

## Verwendete Packages

**Pocketsphinx**

Webseite mit Tutorial: https://cmusphinx.github.io/wiki/tutorial/
PocketSphinx bietet ein Framework zur Spracherkennung für "low-resource Platforms" und kann als Package in ROS heruntergeladen werden. Daher bietet es sich gut für dieses Projekt an.
Ein wichtiger Aspekt ist, das die Erkennung in verschiedenen Modi geschehen kann:

- Keyword-Mode
  - Es wird auf eine vorbestimmte Phrase im Audiostream gehört.
  - Der Phrase wird ein Schwellwert für die Trefferwahrscheinlichkeit gesetzt.
- Grammar-Mode
  - Der Recognizer versucht, die Phrase einer validen Grammatik zuzuordnen
  - Die Grammatik für diesen Vorgang ist Benutzerdefiniert.
- Language-Model-Mode
  - Anstatt einer Grammar, wird ein probabilistisches Language-Model genutzt.
  - Das LM ist ebenfalls Benutzerdefiniert

**Festival TTS**

Dieses Package ist ein bei sound_play mit enthalten und ermöglicht eine Rückmeldung von Botty durch Töne und Text-To-Speech.
Nach dem starten der sound_play Node können folgende Befehle ausgeführt werden:

- Sound abspielen (Eigene Sound Datei oder per ID)
- Textausgabe(Synthese)

## Dateien

**Node send_audio.py**
Kapselt die Audio Eingabe und Konfiguration.
Damit muss man für den Audio Stream nur noch dem Topic sphinx_audio subscriben.

**Node recognizer.py**
Mit dem Audio-Stream als Eingabe wird nach Sprachbefehlen gesucht.
Wurde eins gefunden wird dieser in Textform an den Parser weitergegeben.
Der Ablauf besteht aus folgenden Schritten:

1. Im Keyword-Modus wird für das Wake-up-Keyword wie z.B. "Hey Botty" gesucht.
2. Wake-up gehört?
    a. Ja: zu Schritt 3
    b. Nein: weiter hören
3. Schalte in Grammar-Mode
4. Höre auf Phrase.
5. Warte auf das Ende der Phrase.
6. Suche nach möglicher Interpretation der Grammatik
7. Befehl gefunden?
    a. Ja: Sende Text an Parser weiter und gehe zu Schritt 1.
    b. Nein: Gehe zu Schritt 4

```
self.decoder.end_utt()
if self.decoder.hyp() != None:
        rospy.loginfo('OUTPUT: ' + self.decoder.hyp().hypstr)
        self.pub_.publish(self.decoder.hyp().hypstr)
        self.decoder.set_search('kw')
        rospy.loginfo("Listening to Keyphrase")
self.decoder.start_utt()
```

Gezeigt ist das auswerten einer Phrase mit PocketSphinx im Grammar-Modus.
hyp() liefert eine Hypothese des vom Nutzer genannten Befehls mit Hilfe einer Grammatik.
set_search(‘kw’) setzt den Suchmodus wieder auf Keyword-Mode, so dass nach einer erkannten Phrase wieder das Wake-Word gesagt werden muss.

Der Recognizer benötigt zusätzlich zum start bestimmte Konfigurationsdateien:

- Grammatik
- Keyword List
- Dictionary
- Acoustic Model

Diese sind im Ordner config enthalten und werden beim Start des Nodes als Parameter übergeben.

**Node parser.py**
Im Parser wird der Text mit primitiven Low-Level-Parsing auf bestimmte Keywords überprüft. Es wird nur nach Keywords mit semantischen Informationen überprüft (also Wörter wie "go" oder "blue", nicht "the", "please", usw.). Zusätzlich wird überprüft, ob der Befehl semantisch, bzw. syntaktisch gültig ist.

Jede Aktion hat eine ID mit zugehörigen Synonymen, die in der Klasse "Actions" definiert sind. Der (gültige) Sprachbefehl wird in eine Datenstruktur gepackt, welche die Form "Action( Objekt{ Liste von Attributen } )" hat. Die Datenstruktur ist eine ROS-Message namens "Command", welche in controller/msg definiert ist (mehr Infos dazu in Kapitel Controller).

Ablauf der Verarbeitung einer angekommenen Phrase:

1. Suche nach Aktion
2. Je nach Aktionstyp, suche nach zugehörigen Objekten bzw. Orten
3. Suche nach Liste von Attributen
4. Fülle Command Objekt mit entsprechenden Werten
5. Wenn Befehl gültig ist, sende Command-Message an Controller
    a. (1-3) Bei ungültigem Befehl sende Fehlermeldung

Hier ein paar Beispiele für ein Befehl mit entsprechender Repräsentation:

| “go forward” |
| --- |
| action: 1<br>obj:<br>  name: ‘forward’<br>  attr: - ‘’ |

| “grab the white cup” |
| --- |
| action: 2<br>obj:<br>  name: ‘cup’<br>  attr: [‘white’]‘’ |

**Node talker.py**
Wrapper für die sound_play Node.

**Config**

**Grammars:** In .gram Dateien sind JSGF Grammars enthalten, die für die Spracherkennung von PocketSphinx verwendet werden.
**Dictionaries:** .dic Dateien enthalten ein Lexikon mit phonetischen Repräsentationen von Wörtern, auch für PocketSphinx
**Keyword-list:** In .kwlist sind Phrasen mit Schwellwert für Erkennung im Keyword-Modus definiert
**Acoustic Model:** cmusphinx-en-us-5.2 ist ein Ordner, welcher ein “Acoustic Model” für Englisch enthält.

Für den Recognizer werden folgende Dateien verwendet:

- botty.gram
- botty.kwlist
- cmudict.dic
- Cmusphinx-en-us-5.2

JSGF-Grammar in Datei “botty.gram”

```
public <command> = <keyword> | (<action> (<object> | <position>));
<object> = [THE | A] [<attr>] (BALL | CUP | OBJECT | ARROW | ALL);
<position> = [<pp>] ([THE] <place>);

<keyword> = (STOP | ABORT);
<action> = (GO | GRAB | BRING | FIND);
<attr> = (BLUE | RED | GREEN | WHITE | BLACK);
<pp> = (FROM | TO | AT | IN);
<nr> = (ONE | TWO | THREE | FOUR | FIVE | SIX | SEVEN | EIGHT | NINE | TEN);
<place> = (KITCHEN | LIVING ROOM | BEDROOM | GARAGE | DOCKING STATION | FORWARD | BACK | LEFT | RIGHT);
```

## Konfiguration

Wenn mehr/andere Befehle ermöglicht werden sollen, muss

- config/botty.gram
- config/cmudict.dic
- "Action"-Klasse, Token-Listen in parser.py
  eventuell geändert werden.

Bei einem anderen Acoustic Model z.B. Deutsch anstatt Englisch muss die entsprechende Datei heruntergeladen und der Dateipfad als Parameter in der recognizer.launch Datei geändert werden. Dasselbe gilt natürlich auch bei den anderen Dateien, die in recognizer.launch als Parameter übergeben werden.

## Lessons Learned

Suche nach geeignetem Framework:
PocketSphinx wurde gewählt, weil es in ROS als package integriert ist.
Im nachhinein ist die Integration von non-ROS-Frameworks in ROS kein großer Aufwand.

Daher hätte auch ein bekannteres/besseres Framework gewählt werden können. (Einfachere Handhabung, mehr/bessere Tutorials)

Der Parser hat für eine primitive Analyse gar kein Framework gebraucht. Erst war geplant Features von PocketSphinx bezüglich der JSGF-Grammatik für das semantische Parsing zu nutzen. So könnte man dieselbe Grammatik, die für die Spracherkennung verwendet wird, auch für das Parsing benutzen. Diese Features wurden aber leider in neuen Versionen von PocketSphinx entfernt. Es benutzt auch leider kein anderes Framework außer PocketSphinx JSGF-Grammatik.

### Potenzielle Verbesserungen

Es wäre möglich, Spracherkennung und Parsing durch https://snips.ai/ zu ersetzen.
Siehe: https://github.com/CMU-TBD/snips_nlu_ros

Falls dies nicht gemacht wird, dann ...:

- Bis jetzt muss man bei hinzufügen von möglichen Befehlen an vielen stellen Code ändern.
- Wäre gelöst, wenn die Grammatik für die Spracherkennung auch für das Parsing genutzt werden könnte.
- Das enum für Aktion ID's kann in "Command"-Message implementiert werden.
- Passendes Wake-Up-Keyword mit angepassten Schwellwert.
- Für ein paar Kommandos gibt es noch keine möglichkeit, sie als Sprachbefehl einzugeben:
    - Positionsangabe (z.B. go to position 1 3)
    - Nach Objekt an Position suchen
- Lexikon muss auch eventuell mit neuen Objekten erweitert werden, die von der Kamera erkannt werden können.
- Der Parser arbeitet mit regex und ist noch sehr primitiv, hier könnte ein ordentliches Parsing-Algorithmus verwendet werden.

Folgende Befehle dienen als Beispiele, die noch implementiert werden könnten:

- “Komm zu mir”
- “Kannst du dich im Kreis drehen?”
- “Drehe dich im Kreis und fahre dann einen Meter gerade aus”
- “Links rum drehen”
- “Welches Objekt haltest du gerade?”
- “Schneller!”
- “Wie heißt du?”
- “Du siehst gerade eine Taschenlampe” (zum Trainieren)
- “Was kannst du?”

## Motor

Dieses ROS-Package dient zur Steuerung des Motors des TurtleBot2. Ziel war es, dass der Motor von außerhalb aufgefordert werden kann bestimmte Aktionen auszuführen.

## Funktionsweise

Der Motor nutzt einen ROS-Service, durch den Botty diverse Befehle erhalten kann, die er ausführen soll. Botty kann sich eine gewünschte Gradzahl im Kreis drehen oder eine bestimmte Strecke geradeaus fahren, während er dabei mögliche Hindernisse umfährt. Als Service wird nach jeder Aktion auch eine Rückmeldung gegeben bzw. einen Rückgabewert geliefert, der einerseits Aufschluss darüber gibt, welche Stelle Botty erreicht hat und ob seine Aktion erfolgreich ausgeführt werden konnte. Die Rückgabe der Positionierung erfolgt über X und Y Koordinaten in Metern.

## Verwendete Packages

Für den Motor wurden keine externen Packages verwendet.

## Dateien

**motorService.py**

Startet den Motor Service und warte auf Kommandos, die dann ausgeführt werden.

Als ROS-Service wird nach Start zuerst eine Initialisierung vorgenommen. Erst wenn von anderen Komponenten Aufforderungen an den Motor geschickt werden, werden diese verarbeitet und ausgeführt. Für gewöhnlich erhält der Motor die Aufforderungen vom Controller; er kann aber auch unabhängig genutzt werden. Ein Kommando muss in der Form einer ROS-Service-Definition erstellt werden, damit der Motor sie entgegennehmen kann. Sie besteht aus 2 Teilen: 'call': der Name der verwendeten Funktionalität und 'param': eine Floatliste mit Parametern, wie z.B für den Drehwinkel beim Drehen. Für jedes Feature wird dieselbe Service-Definition benutzt, weshalb das erhaltene Kommando auf mögliche Fehler überprüft werden muss. Ein Beispiel wäre der Name eines Features, den es nicht gibt. Infolge dessen würde der Motor Service den Fehler melden: "Error: Call-name is unknown". Anzumerken ist, dass die Fehler nicht zum Programmabsturz führen.
Im folgenden Codeausschnitt ist ein Beispiel für die interne Fehlerüberprüfung und Ausführung des “stopp” Kommandos.

```
def decode(command):
    #check wich command u received and check if it's parameters are correct
    #'call' defines the function name
    #'param' defines the given parameters as integers (boolean are representable by 0 and 1)
    global xCoord
    global yCoord
    done=False
    if command.call=="setSpeed":
        if len(command.param)==1:
            rospy.loginfo("Call: setSpeed("+command.param[0]+")")
            setSpeed(int(command.param[0]))
        else:
            rospy.logdebug("Call: setSpeed; Error: Wrong amount of Parameters! setSpeed takes 1 parameter =>speed")
        done=True
```

Die eigentlichen Funktionalitäten werden im Folgenden einzeln erklärt.

**setSpeed**

Nimmt einen Parameter 'speed' entgegen und ändert die Geschwindigkeit entsprechend. Diese Geschwindigkeit gilt für alle zukünftigen Aktionen, solange sie nicht geändert wird. Standardmäßig beträgt sie 0.75.

Mögliche Fehlermeldung:

"Call: setSpeed; Error: Wrong amount of Parameters! setSpeed takes 1 parameter =>speed"

**forwardByMeters**
Nimmt zwei Parameter entgegen, wobei der Zweite optional ist. Der erste Parameter 'meter' gibt die Distanz an, die der Turtlebot nach vorne fahren soll. Zunächst wird die Zeit berechnet, die er bei der gegebenen Geschwindigkeit benötigt, um die gewünschte Distanz zurück zu legen. Dann wird eine Bussy-Loop genutzt, um ihn immer weiter fahren zu lassen, bis er die gewünschte Zeit gefahren ist und damit auch sein Ziel erreicht hat. Sollte als zweiter Parameter eine 1 angegeben werden, die als Boolean interpretiert wird, wird Botty auf Hindernisse in seinem Weg achten und diese gegebenenfalls umfahren. Dabei geschieht Folgendes:

Wenn Botty ein Hindernis mittels dem Lidar vor sich sieht, hält er an und schaut, ob er rechts vorbei fahren kann. Falls ja, fährt er rechts am Hindernis entlang, bis er das Hindernis nicht mehr erkennt. Botty wird sich die Zeit merken, die er benötigt hat, um am Hindernis entlang zu fahren, damit er später weiß, wie lange er wieder zurück fahren muss, sobald er am Hindernis vorbei gefahren ist. Dann wird er in ursprünglicher Fahrtrichtung am Hindernis weiterfahren und sich zur zur ursprünglich geplante Route zurückdrehen. Die Zeit, die er sich zuvor gemerkt hat, fährt er nun zurück, in der Konsequenz, dass er hinter dem Hindernis steht auf exakt der Stelle, die er wegen des Hindernis nicht erreichen konnte. Danach wird er seinen anfänglichen Pfad weiter folgen. Sollte er aber ganz zu Beginn nicht nach rechts ausweichen können, da dort ebenso ein Hindernis steht, wird er stattdessen links ausweichen. Sollte auch das nicht gehen, da es sich um eine Sackgasse handelt, scheitert die gesamte Aktion mit der Warnung "No space". Sollte die Umgehung von Objekten nicht gewollt sein, kann man auf den zweiten Parameter verzichten oder eine 0 übergeben. In folge dessen wird Botty blind nach vorne fahren.

Mögliche Fehlermeldungen:
"Call: forwardByMeter(distanz, hokuyo); Error: Second Argument only takes 1 or 0 to be converted to boolean!"

"Call: forwardByMeters; Error: Wrong amount of parameters! forwardByMeters takes 1 parameter and 1 optional =>meters,hokuyo=False (take 0 or 1 to represent the boolean)"

**turnRigthByAngle; turnLeftByAngle**
Nimmt zwei Parameter entgegen, wobei der Zweite optional ist. Der erste Parameter 'angle' gibt die Gradzahl an, um die sich Botty drehen soll, während der zweite noch zusätzlich die Geschwindigkeit beim drehen anpassen kann. Zunächst wird die Zeit berechnet, die er bei der gegebenen Geschwindigkeit benötigt, um die gewünschte Drehung durchzuführen. Dann wird eine Busy-Loop genutzt, um ihn immer weiter drehen zu lassen, bis er sich die gewünschte Zeit gedreht hat und damit auch seine Zielausrichtung erreicht hat. Je nachdem welche der beiden Drehungs-Funktionen genutzt wurde, wird er sich entweder im Uhrzeigersinn oder entgegen drehen. Der zweite optionale Parameter wird wie die “setSpeed” Funktionalität die Geschwindigkeit anpassen.

Mögliche Fehlermeldungen:

"Call: turnRight/LeftByAngle; Error: Wrong amount of Parameters! turnRight/LeftByAngle takes 2 parameters; 1 is optional =>angle,speed=None"

**stopp**
Nimmt keinen Parameter entgegen. Bei Aufruf wird jegliche derzeit ausgeführte Aktion abgebrochen. Diese Funktion dient dazu bei ungewolltem Verhalten die Motoren zu stoppen, bevor Botty Schaden nehmen könnte. Auch die Busy-Loops der anderen Funktionalitäten werden dann abgebrochen. Diese Funktion kann manuell, vom Controller oder auch von ROS selbst aufgerufen werden. Sollte es zu einem Systemfehler kommen und das Programm beendet wird, wird ROS selbstständig diese Funktion als letztes ausführen, um sicherzustellen, dass nach Programmabsturz der Motor auch zum Stoppen kommt, da sonst dieser immer weiter Gas geben würde.

Mögliche Fehlermeldungen:
"Call: stopp; Error: Wrong amount of Parameters! stopp takes 0 parameters"

Anmerkung:
Es muss sich keine Sorgen gemacht werden, dass diese Funktion einen Fehler wirft, wenn ROS selbst wegen eines Programmabsturz Botty stoppen will. Dieser Fehler kann nicht auftreten, wenn ROS die Funktionalität nutzt.

**srv/call.srv**
Stellen die für ROS notwendigen Service-Definitions dar.

## Konfiguration

Jede normale Anpassung seines Verhaltens, wie die Distanz, der Drehwinkel oder die Geschwindigkeit, werden über den Service zur Anwendungszeit über entsprechende Aufrufe angepasst. Detail Anpassungen, wie zum Beispiel die Exaktheit der Drehung, müssen im Code vorgenommen werden.

## Lessons Learned

Der Motor ist eine weitaus größere Herausforderung als man ahnt. Die große Schwierigkeit zeigt sich in der Feinjustierung seiner Aktionen. Parameterangaben zum Fahren, als auch zum Drehen, werden durch einen Wert von 0.0 bis 1.0 repräsentiert. Eine uns normal erscheinende Angabe in Metern oder Winkelgrade ist für ihn nativ nicht möglich. Die Frage war also, wie lange muss sich Botty mit einem Wert von 0.75 drehen bis er sich um X Grad gedreht hat? Hierzu waren zahlreiche Tests notwendig bis ein Faktor gefunden wurde, der auf die gewünschte Gradzahl oder Distanz multipliziert wird, sodass das Ist-Ergebnis nach der Ausführung möglichst ähnlich dem Sollergebnis entspricht. Diese Faktoren sind im Code als Abweichungen hinterlegt und entsprechend kommentiert. Ein weiteres Problem ist, dass das Zurücklegen von zwei mal einem Meter und einmal zwei Meter unterschiedliche Ergebnisse bringt. Grund hierfür ist, dass beim Fahren von zweimal einem Meter auch mehrmals beschleunigt werden muss, was entsprechend Zeit kostet. Zusätzlich empfiehlt es

sich bei Tests sich die Möglichkeit des Abbrechen bereit zu halten, da allein kleine Fehler große Auswirkungen in seinem Fahrverhalten haben können, wodurch Botty unberechenbar werden könnte und schnell gegen eine Wand fährt, wenn nicht aufgepasst wird.

### Potenzielle Verbesserungen

Die derzeitigen Beschleunigungen und Bremsvorgänge sind allesamt sehr plötzlich, was dazu führt, dass wenn Botty bei größeren Geschwindigkeiten auf null runter bremst, er noch ein kleines Stück rutscht. Diese Aktionen könnten durch eine ruhigere Beschleunigung und Bremsung optimiert werden. Durch das Verrutschen würde auch seine interne Verwaltung der Positionierung über die Zeit kleinere Abweichungen haben. Ein weitere Verbesserung wäre es, die Werte der Abweichung anzupassen. Diese wurden erstellt und getestet in einer Phase, als der PhantomX Reactor Arm wegen eines Hardware Schadens abmontiert war. Nun wo er wieder aufgesetzt ist bringt er einiges an Zusatzgewicht mit, wodurch Botty langsamer beschleunigt und fährt. Dadurch erreicht er nicht mehr die gewünschte Distanz beim fahren oder dreht sich zu kurz.

# Sonstige Benutzungshinweise

## Stromprobleme

Der TurtleBot lässt sich entweder direkt über das Ladegerät aufladen oder man schließt das Ladegerät an die Ladestation an und positioniert den TurtleBot darauf. Die Positionierung des TurtleBots auf dem Ladegerät muss aber genau zentral sein, um verbunden zu sein. Aus diesem Grund ist es empfehlenswert, den TurtleBot direkt über das Ladekabel zu laden.

Man kann leider nicht die aktuelle Ladung der Akkus auslesen, weswegen der TurtleBot nach Gebrauch geladen werden sollte. Haben die Akkus keinen Saft mehr, schaltet sich das NUC komplett aus. Sollte dies geschehen, sollte vor einschalten des NUCs der TurtleBot für 2 bis 3 Minuten geladen werden, da es sonst möglich ist, dass der TurtleBot sich direkt wieder ausschaltet.

Da die Akkus viele Ladezyklen während des Projektes durchgemacht haben, könnte es eventuell sein, dass neue Akkus bestellt werden müssen.

## USB-Anschlüsse

Der Onboard-Rechner des TurtleBots (NUC) hat nur 4 USB-Ports. Greifarm, Lidar und 3D-Kamera belegen bereits 3 davon. Wenn auf dem NUC direkt mit Maus und Tastatur arbeiten

will, muss man entweder eines der anderen Geräte abtrennen oder einen USB-Hub verwenden. Lautsprecher und Kopfhörer benötigen auch jeweils einen USB-Anschluss. Wenn der USB-Hub benutzt wird, sollte darauf geachtet werden, dass die großen Stromverbraucher(Greifarm, Lidar, 3D-Kamera) direkt am NUC und nicht über den USB-Hub angeschlossen sind, da es sonst Probleme mit der Stromversorgung geben kann und der Akku sich sehr schnell entlädt.

## Lieferung

Bei der Lieferung des TurtleBots sind die Stangen für die Halterungsplattformen beschädigt worden, weswegen neue Stangen und Schrauben nachgeliefert wurden. Die obere Plattform ist aus diesem Grund nicht zu 100% befestigt.
Außerdem ist während dem Projektes der Arduino des Greifers ausgefallen, weswegen hierfür auch ein Ersatz nachgeliefert wurde.

## Umbau der Kamera

Um eine bessere Objekterkennung zu ermöglichen, wurde die Halterung für die 3D-Kamera an der Vorderseite des TurtleBot montiert (ursprünglich befindet sich diese am hinteren Teil). Dadurch sind die Stangen und Halterungsplattformen nicht mehr im Bild der Kamera.

# Zusammenfassung

In diesem Projekt wurde für den TurtleBot2 im Rahmen eines Studienprojekts ein ROS-Package entwickelt. Die Hardware-Komponenten werden mithilfe einer zentralen Kontrolleinheit (Controller) gesteuert, womit ein Pick and Place Szenario realisiert wurde, bei dem der TurtleBot2 zu einer Stelle in einem Grid fährt und dort ein Objekt mithilfe einer 3D-Kamera sucht. Erkennt der TurtleBot ein bekanntes Objekt, kann mit dem Greifarm eine Stoß-Bewegung ausgeführt werden. Über Sprachbefehle kann der Roboter gesteuert werden und über Sprachausgaben werden dem Nutzer Informationen zurückgegeben. Durch ein Lidar kann der Roboter Hindernisse umfahren. Jederzeit kann durch einen asynchronen Stoppbefehl eine Aktion abgebrochen werden.

Während des Projektverlaufs wurden gemeinsame Entwicklungsumgebungen getestet und versucht eine Virtualisierungsumgebung zur Entwicklung mit ROS einzurichten. Dabei wurden auch Hardwareprobleme des TurtleBot2 behoben. Das ROS-Package wurde in mehreren Phasen iterativ als vertikaler Prototyp entwickelt.

In diesem Dokument wurden die Arbeiten des Studienprojektes ausführlich beschrieben. Es wurde gezeigt, welche Funktionen umgesetzt werden und wie die einzelnen Komponenten des hierbei entworfenen ROS-Packages funktionieren.

17.02.2020
Markus Dauth, David Kostka, Felix Mayer, Raschied Slet

# Anhang

## Code-Listing des Git-Repository

Folgendes Listing zeigt alle Ordner des Botty Git-Repository. Alle hier aufgelistet Dateien in allen Verzeichnissen wurden von uns in diesem Projekt selbst manuell erstellt bzw. sind manuell geändert worden. Ausnahmen hierbei sind jeweils die beiden ROS-Package-Dateien “CMakeList.txt” “und package.xml”, welche bei der Erstellung der ROS-Packages automatisch generiert wurden und in mehreren Ordnern zu finden sind. Eine weitere Ausnahme ist der Ordner “cmusphinx-en-us-5.2” unter speech/config, da dies ein vorkonfigurierter Ordner für PocketSphinx ist (fett markiert).

```
├────arm
│   └────nodes
├────camera
│   ├────auswertung
│   ├────image_rec
│   ├────launch
│   ├────src
│   └────srv
├────controller
│   ├────launch
│   ├────msg
│   └────nodes
├────documents
│   └────videos
├────lidar
│   ├────msg
│   ├────scripts
│   └────srv
├────motor
│   ├────scripts
│   └────srv
└────speech
    ├────config
    │   └────cmusphinx-en-us-5.2 (hierbei handelt es sich um ein Paket von Dritten)
    ├────launch
    └────nodes
```

Bearbeitete Datei(en), die sich nicht im GitHub-Repository befinden::

- “.bashrc” im Homeverzeichnis des Ubuntu-Users “tb2”.