General Guide Robotics for AI

Marc Groefsema

November 15, 2022

Contents

Chapter 1

Introduction

For the course Robotics for Artifical Intelligence, at the University of Groningen, a platform has been developed with which students can gain practical experience with respect to multiple topics in AI, as well as gaining experience in software development using the Robot Operating System (ROS) [1]. Topics that are addressed during the course contain autonomous navigation, object detection, object recognition, manipulation and high level behaviour programming.

Over the years, the course has evolved around topics and techniques used while participating in the Robocup@Home [2]. (`https://athome.robocup.org`).

The robotics lab from the University of Groningen has participated multiple times in this competition with the team Borg. For more information about the former participations and publications, please refer to `www.teamborg.nl`.

The tackle kind of challenges that are provided by domestic robotics, at least a basic set of skills is required. This includes autonomous navigation, visual perception, manipulation and auditory perception. During the course these topics, except for auditory perception, are covered. For autonomous navigation the ROS software stack MoveBase is used, which combines odometry, point cloud data and lidar data. Also, object recognition is covered in the course, where point cloud analysis is used for object detection, whereas deep learning techniques are used for object recognition. Finally, manipulation is covered by using the MoveIt™[3] framework, which covered the planning for a safe execution of arm movement, while using point cloud analysis for perception of the world around the robot. These different skills are combined by using a behaviour tree setup, where multiple finite state machines are used to define the exact behaviour of the robot in every situation, including situation where executions fail and defining recoveries.

During earlier iterations of the course relative large platforms have been used, such as Alice and the Tiago platform. While these platforms sufficed in their capabilities to be used for this course, they required a large amount of space to operate in, especially with respect to navigation and manipulation. As the number of students for the course have been in an uptrend, the need for more platforms arose, such that students can have more time to test their programs on the platforms, while the required space to operate these multiple platforms in is preferred to remain the same.

Hence, to satisfy these requirements, new platforms have been developed, which are much smaller than the earlier platforms, hence having less minimal space requirements to operate in, as well as being relatively cheap, compared to of-the-shelf platforms with similar capabilities with respect to the current educational purposes. These platforms allow for a more scalable setup.

1.1 Application requirements

The platform is designed for educational purposes. It is an platform that is designed to be affordable while being able to perform a set of tasks as defined here.

1.1.1 Navigation

The robot should be able to perform autonomous navigation, using the MoveBase package from ROS. This entails that the robot should be able to accurately execute velocity commands, maintain odometry information, having sensors that are suitable for creating a map and potentially other sensors for nearby obstacle detection. During the

course, students should be able to create a map of the environment and define navigation waypoints within this map. Next, the robot should be able to accept navigation commands to the respective waypoints and perform autonomous navigation to reach this waypoint. The MoveBase package will inform the program that sent a navigation request, whether the execution succeeded or failed.

1.1.2 Object detection

Apart from navigation, the robot should be able to interact with object in the environment. The first step of interacting with objects, is to detect objects. Given an object located at the floor, the platform should be able to infer information about the location, orientation and dimensionality of the object, as this information will form the basis of interacting with an object. Obtaining this information has to be done using pointcloud analysis to be compatible with legacy software that has been used in the course Robotics for AI. The exact workings of this procedure will be discussed later on.

1.1.3 Object recognition

Given detection information about an object, the platform should be able to label the object. This is done by applying a convolutional neural network (CNN) for image recognition. Using information from object detection, the platform should be able to extract an region of interest (ROI) from the camera images, which can be used for object recognition. The platform should have enough computational capability of using the CNN for inference at run-time.

1.1.4 Manipulation

Apart from obtaining knowledge about objects in the environment, the platform should be able to pick up an object when required and place it at in a different location in the map. Hence the platform should be able to pick up the object, store it in such a manner that the platform can safely navigate whilst carrying the object, and drop off the object at a specified location. The planning of the manipulator should be done using the MoveIt program in order to be compatible with earlier created software in the course.

1.1.5 Dimensions

As robotic platforms tend to require a relative large amount of space to operate in, such that there is enough room to navigate in, it is desirable to have a relatively small platform, since it would require less space to operate in.

Chapter 2

Sensors and Actuators

2.1 Camera

For vision applications, the platforms makes use of the Intel Realsense d435 camera. This camera is capable of producing 1920×1080 RGB images at 30 fps, with a field of view of 69×42 degrees ($H \times V$). Apart from RGB images, the camera is capable of producing depth images. These images have a resolution of 1280×720 at a field of view of 87×58 degrees. These two types of images together allow for the generation of pointclouds, as will be used by the platform for e.g. obstacle and object detection. This camera allows the platform to do both computer vision using regular 2D images, as well as pointcloud analysis. The pointcloud can e.g. be used to detect objects in the environment in front of the platform. This information has its applications in object detection as well as for obstacle detection during autonomous navigation.

This camera is placed at the top of the neck of the platform in a 45 degrees downward looking position, such that the cameras field of view is minimally obscured by the robotic arm. This way the platform can detect the top of relatively small objects, enabling the platform to estimate the orientation of objects in front of it.

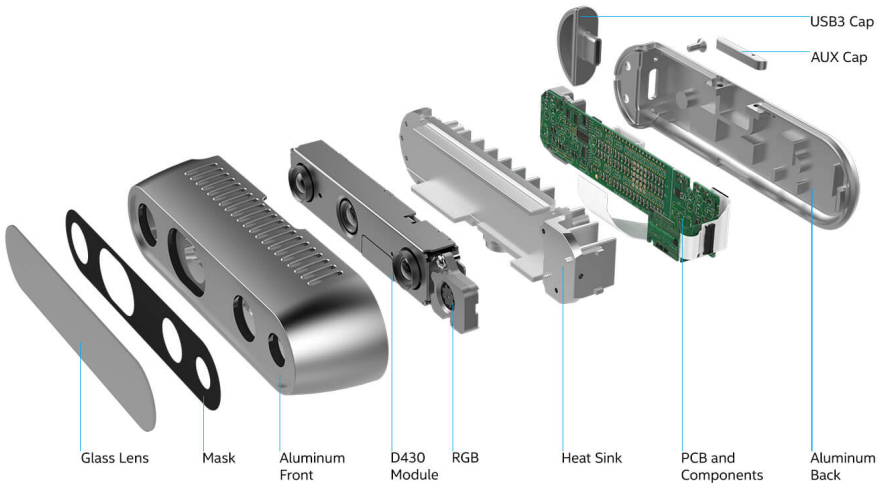
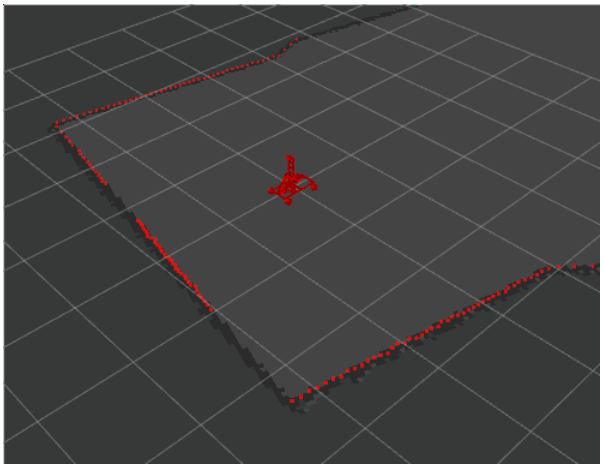


Figure 2.1: The Intel Realsense d435 depth camera

2.2 Lidar

To sense obstacles in the surrounding environment, the platform can use the lidar sensor *RPLIDAR A1M8-R6*, as shown in figure 2.2a. This lidar uses 360 degree scans, with a scanning frequency of 5.5 Hz while measuring up to a distance of 12 meters at an 1 degree angular resolution. The lidar is placed inverted at the bottom of the platform. It is used during autonomous navigation to map an environment, detecting nearby obstacles as well during localisation. An illustration of the lidar detections is shown in figure 2.2b.

(a) RPLIDAR A1M8-R6



(b) An examplke of lidar data

2.3 Encoder motors

The platform uses four Pololu 4755 motors, as shown in figure 2.3. These 12V motors use an encoder with 64 counts per revolution, together with a 102.8 : 1 gearbox, resulting in an resolution of 6533 counts per revolution of the gearbox' output shaft. These motors are used to drive the Mecanum wheels, as describes hereafter. Using the encoder information, as described later-on, the platform is able to calculate its wheel velocities and changes in position. This allows the platform to calculate odometry, as well as enabling a PID control loop to control the platform's driving velocities as desired.

Figure 2.3: Pololu 4755

2.4 Mecanum wheels

The four Mecanum wheels, as shown in figure 2.4, are used to enable the robot to move in three degrees of freedom, namely laterally forwards, backwards and sidewards, as well as rotating left and right. This capability enables the platform to make small changes in position when required by the application. The wheels are attached to the platform in such a manner that the rollers of the wheels all point towards the center of the platform.

2.5 The arm

The platform makes use of a custom 6-DOF arm, which consists mostly of 3D printed parts, actuated by Dynamixel-AX-12A servos. An illustration of the arm is shown in figure 2.5.

Figure 2.4: 10 cm mecanum wheels

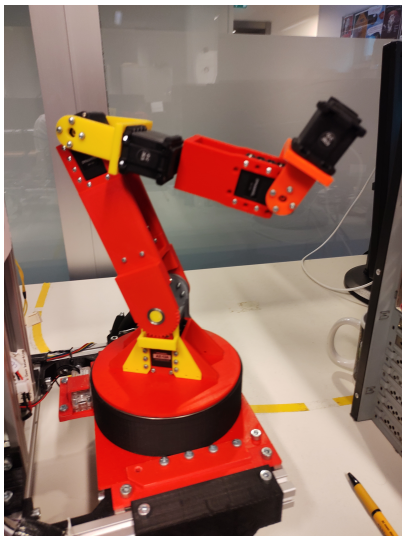


Figure 2.5: The arm

2.5.1 Dynamixel AX-12A

In order to actuate the arm, it makes use of 6 Dynamixel AX-12A servo motors, as shown in figure 2.6. These servo motors can be linked to each other, such that they share the same 12V power supply. Communication is done using a shared communication bus for serial interfacing. These servo motors are able to perform position control, within a 300 degree range, whilst having a maximum torque of $0.2Nm$ at $12V$, drawing $1.4A$. The servo motors are powered using a connection to the power supply unit, while being placed in serial with the emergency button.

Figure 2.6: The Dynamixel AX-12A

Chapter 3

Computational devices

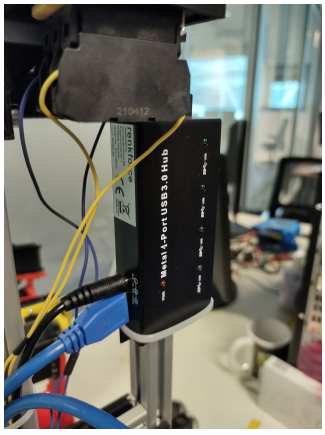
The onboard computational devices in the platform are responsible for running all programs required by the platform. While it is possible to run programs on other computers that are connected to the platform over WiFi, the platform should be able to run most programs on-board.

3.1 Jetson

The Jetson computer is a small computer that is able to run a full Ubuntu 18.04 operating system and serve as the main onboard computer. This computer supports multiple USB connections, which with sensors and actuators can communicate with the programs running on the Jetson. To extend the USB capabilities of the Jetson, a USB3.0 USB hub is used in conjunction with the computer, which is also capable of providing power to the attached sensors and actuators. This enables the usage of peripherals that would require more power than can be delivered by the USB circuitry of the Jetson itself. Next to the USB capabilities of the board, the Jetson is also capable of providing a stable WiFi connection, such that other computers can communicate with the platform in a stable manner. A last note on the Jetson is the capability of using its onboard GPU, containing 128 CUDA cores, enabling proper run-time execution of e.g. neural networks, that would be too computationally demanding when deployed on the CPU processor.

3.2 USB hub

The extend the capabilities of the Jetson computer, an USB hub with external power capabilities is used, as shown in figure 3.1.



(a) The rear side of the USB hub

(b) The front side of the USB-hub

Figure 3.1: The USB hub

3.3 Teensy 4.0

The Teensy 4.0 microprocessor is used in the motor controller, since it is capable to directly interact with the motors and encoders. This microprocessor connects over a USB connection to the Jetson computer, enabling communication between them. The Teensy 4.0 hosts calculation for mappings between velocities of the robot platform to rotational velocities of the individual wheels. Also, it calculates the odometry of the robot, given the rotation information of the wheels. At last, the Teensy 4.0 executes PID control loops for each motor, enabling the platform to execute requested target velocities.

Chapter 4

Development with ROS

4.1 Introduction

This chapter will give a brief overview of design patterns that are commonly used in ROS. By no means is this chapter an extensive ROS tutorial. For this the reader is advised to visit `https://wiki.ros.org/melodic`. In ROS development many different programs are cooperating, where these different programs implement different functionalities for the robot and have seperate responsibilities. These programs in the context of ROS are referred to as *ROS nodes*. ROS provides a network communication infrastructure such that different communication patters are possible between these nodes. The central communication node in ROS is called *roscore*. One of the roles of *roscore* is knowing the adresses of all ROS nodes, such that different nodes are able to talk to each other. When starting programs with *roslaunch*, an instance of *roscore* will automatically be started, of none is running yet. When starting programs with *rosrun* an instance of *roscore* has to be started manually if it isn't running yet.

This chapter will first go into the *publisher - subscriber* pattern, whereafter *services* and *actions* are discussed. At last, we will take a look into behaviour programming, which acts as the top level software which implements the behaviour of the robot, and delegate tasks to existing and custom ROS nodes.

4.2 Folder structure

The folder structure during ros development is as follows. A *catkin_ws* exists, which contains a *src*, *build*, *devel* and *log* folder. All development packages are contained in the *src* folder. The other folders contain auto-generated files, such as compiled programs, compiled header files, log data and system configuration files. In the *src* folder, you will find your repository, containing one or multiple packages that implement nodes or configurations. A package folder can consist of the following subdirectories and files:

- *src*: Here C/C++ source files are stored.
- *include*: Here C/C++ header files are stored.
- *cfg*: Here configuration files are stored.
- *scripts*: Python scripts are stored here.
- *launch*: The directory for storing launch files.
- *msg*: Here custom message definitions are stored
- *srv*: Service definitions are stored here.
- *action*: Action definitions are stored here.
- *CMakeLists.txt*: This file contains build instructions for the respective C/C++ programs, as well as build instruction for autogeneration of code for the custom actions, services and message definitions.
- *package.xml*: This file contains package configurations, e.g. dependencies on other packages.

4.3 Publishers and Subscribers

One commonly used communication pattern in ROS is the *publisher - subscriber* pattern. This pattern is typically used for data streams, e.g. that of sensor data. A program that publishes sends out messages, i.e. a datastructure, under a topic name. Other programs that want to use this data can *subscribe* to this topic name, and receive the sent data in a callback function, such that this data can be used. An example would be a camera. A camera driver publishes every time a new frame is made by the camera. Another program that implements some computer vision task can subscribe to this camera topic, such that it continuously receives the most recent camera frame.

4.3.1 Simple example

Let's first take a look at a simple publisher example in listing 4.1, as provided by the ROS documentation.

Here, first a ROS node is initialized by the function *init_node*. Hereafter, in the function *talker*, a *Publisher* object is created, which is able to send *String* messages over the topic *'chatter'*. Every 0.1 seconds a *String* message is sent to the topic 'chatter', in the while-loop. Once this publisher is running, roscore will know about the existence of this datastream, and *Subscribers* will be able to subscribe to this 'chatter' topic, enabling them to receive the messages that are being sent. Hence note that a single publisher can have multiple subscribers being subscribed to it, which will all receive the messages that are being sent.

```
#!/usr/bin/env python
#source: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

# license removed for brevity
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    rospy.init_node('talker', anonymous=True)
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

Listing 4.1: An illustrative example Publisher node

Taking a look at listing 4.2, we can see an example implementation of a *Subscriber*. Here we see again a *ROS node* being instantiated first, after which a Subscriber object is created. The contructor of the Subscriber requires as arguments first the *topic name* of the datastream. Seconds, the message type, and third a callback function. This callback function will get called every time a message is received by the subscriber. In this example the callback function is called *callback*, which has a single argument, which will be the received message. Once it is received, the content of the *String message* is printed.

```
#!/usr/bin/env python
#source: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)


if __name__ == '__main__':
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
    rospy.spin()
```

Listing 4.2: An illustrative example Subscriber node

4.3.2 Custom messages

To use custom message definitions, multiple things have to be set up. First, the message has to be defined in the related package folder. After this, the *CMakeLists.txt* and *package.xml* have to be set-up such that ROS will generate the messages. This section will act as a simple illustration of message generation. For a more in depth illustration, it is advised to refer to the respective ROS manual: `http://wiki.ros.org/msg`.

Defining a message

To define a message, a *.msg* file has to be created in the *msg* folder of the relevant package. If this folder doesn't exist yet, it has to be created manually. The message is a data structure, containing one or more variables. This example message, as shown in listing 4.3, contains to arrays of strings, named *objects* and *tables*. This refers to the objects that the robot should look for and the tables which it should visit to search for these objects. For a more elaborate overview of message contents, on is advised to refer to the manual on messages as mentioned above.

```
#file: Order.msg
string[] objects
string[] tables
```

Listing 4.3: An example of a custom message

Setting up the CMakeLists.txt

To generate the example message above, the *CMakeLists.txt* file in the package has to be configured. An example is shown in listing 4.4. First, *genmsg* has to be added to the *find_package* section, as the *genmsg* package is responsible for message generation. Second, the *add_message_files* section has to be configured. Here the name of the custom message file has to be noted, such that the *genmsg* package knows which message files have to be created.

```
cmake_minimum_required(VERSION 2.8.3)
project(my_msgs)

## Compile as C++11, supported in ROS Kinetic and newer
# add_compile_options(-std=c++11)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED
        genmsg
)

## Generate messages in the 'msg' folder
add_message_files(
  FILES
  Order.msg
)


catkin_package()
include_directories()
```

Listing 4.4: Illustrative sections for the CMakelists file to generate custom messages

Package.xml configuration

Finally, the *package.xml* file needs to be configured for message generation. An illustrative example is shown in listing 4.5. In addition to the auto-generated *package.xml*, an *exec_depend* and *build_depend* have to be added for *message_generation*. Next, the *buildtool_depend* for *genmsg* needs to be added. The *package.xml* file contains references to which programs are required to the message generation, and not to the specific custom messages themselves.

```
<?xml version="1.0"?>
<package format="2">
  <name>my_msgs</name>
  <version>0.0.0</version>
```

```
  <description>The my_msgs package</description>

  <maintainer email="borg@todo.todo">borg</maintainer>

  <exec_depend>message_generation</exec_depend>
  <build_depend>message_generation</build_depend>
  <buildtool_depend>genmsg</buildtool_depend>
  <buildtool_depend>catkin</buildtool_depend>


  <!-- The export tag contains other, unspecified, tags -->
  <export>
    <!-- Other tools can request additional information be placed here -->

  </export>
</package>
```

Listing 4.5: An illustrative example of the package.xml configuration for message generation.

4.4 Services

Services are another communication pattern in ROS. Here a *remote procedure call* (RPC) is made to another program. I.e. a function call to a function in another program. This function, or service, exists somewhere in the ROS ecosystem of nodes, hence it might even exist on another computer. Typically, these service calls are only used between ROS nodes and not used by behaviours, which are described later on. This is because a service call is a ***blocking*** operation, hence to program making the service request does not continue to run until a result is obtained. The reason for not using service calls from behaviours is exactly this. If used, the entire high level code base will not be able to run until the service call is finished. Hence, typically Actions are used here instead, as described later.

4.4.1 Service definition

The definition of custom Service types are defined by *.srv* files, which are located in the */srv/* folder in the respective package folder. An example definition for the Service example below is shown in listing 4.6. Here we see two sections, seperated by three dashes. In the first section the input arguments of the service call are specified. In the second section the return, or response, variables are specified. The types of these variables can be any of the ROS message types, including custom ones.

```
int64 a
int64 b
---
int64 sum
```

Listing 4.6: AddTwoInts.srv: The Service definition

4.4.2 Service 'server'

Service calls have two parties. First, the program that implements the Service, and second the program calling the service. Let us first take a look at an example Service in listing 4.7 as stated in the ROS tutorials.

```
#!/usr/bin/env python
#source: http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28python%29

from __future__ import print_function

from beginner_tutorials.srv import AddTwoInts,AddTwoIntsResponse
import rospy

def handle_add_two_ints(req):
    print("Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b)))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
```

```
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print("Ready to add two ints.")
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

Listing 4.7: An example Service implementation

Here we can see that again first a ROS node is initialized using the function *init_node*. Hereafter, a Service is declared, containing in its constructor arguments a *service name*, a *type* and a callback function. Using this Service name and type, other programs are able to refer to this service. Every time a request is made to the Service, the callback function will be called, and the Service 'arguments' are passed to this program using the function argument of the callback function, i.e. *handle_add_two_ints*. Here the return value of the Service callback is the datastructure that is passed back to the program that made the Service request.

4.4.3 Service 'client'

An example of a Service client is shown in listing 4.8. Here, in the function *add_two_ints* we see that first we wait for the service to come online. It might be that the client program is started before the service is ready to process a request, hence first we will wait until that service is ready. Next, a *ServiceProxy* is instantiated, with two arguments. First, the service *name* is provided, after which the type is provided. These correspond to what we've seen in the earlier service program. Next, we can see that this *ServiceProxy* object is callable, as if it was a function. The return value of this 'function' call, i.e. the *response*, is stored in the variable *resp1*. Note, that this program will halt during the call on the service, until it received a response, since a service call is a **blocking** operation.

```
#!/usr/bin/env python
#source: http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28python%29

import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException as e:
        print("Service call failed: %s"%e)
if __name__ == "__main__":
    x = 1
    y = 2
    print("Requesting %s+%s"%(x, y))
    print("%s + %s = %s"%(x, y, add_two_ints_client(x, y)))
```

Listing 4.8: An example service client

4.4.4 How to make a custom service definition

When creating a custom service definition, first the package needs to be configured for message generation. This needs to be done both in the *package.xml* and the *CMakeLists.txt* files located in the package folder.

```
  <build_depend>message_generation</build_depend>
```

Listing 4.9: package.xml entries for service generation

```
find_package(catkin REQUIRED COMPONENTS
  message_generation
)

 add_service_files(
   FILES
```

```
    AddTwoInts.srv
)

generate_messages(
  DEPENDENCIES
  std_msgs
  #potential other message dependencies
)
```

Listing 4.10: CMakeLists.txt entries for service generation

4.5 Actions

In ROS an Action is a processes that is implemented by an *Action Server*. Here a task can be delegated to an Action Server, which will then execute this task. The *Action Client*, i.e. the program that makes the request, can ask the Action server about the current state of the execution, and whether the Action Server finished, failed, or whether it is still working on the task. While the Action Server is executing the task, the Client is still able to execute other tasks, since delegating tasks to an Action Server is a non-blocking process. An example of an Action Server is *MoveBase*, which is a program stack that implements autonomous navigation. A client can request the MoveBase Action Server to navigate to a certain point. While MoveBase is executing the navigation, the client can continue to focus on other things, and ask MoveBase so now and then whether the navigation finished or failed. It is not always guaranteed that the Action Server is capable of finishing the task. E.g. for navigation, the robot might get stuck, or no possible path exists for reaching the goal location. In such cases the Action Server can return a Failed status to the client, which on its turn receives the responsibility to handle this failure. E.g. for navigation, the path might have been blocked by a human. If the navigation failed, it might be useful to try again later, or navigate towards another location.

4.5.1 Action Server example

. Listing 4.11 shows a Python implementation of an fake navigation program. Typically, the program for the Action Server is implemented in a class. Here we that the *FakeNavigationServer* class creates an *SimpleActionServer* as a member variable. The *SimpleActionServer constructor* requires a few arguments. First, the name of the Action is specified, as which it will be known by the rest of the ROS ecosystem, where after the Action *type* is specified. Later on this type is discussed in more detail. Third, a callback function is specified. This callback function is typically a member function of the action class that is being implemented. In this case, the member function *goal_callback* is provided as a callback function. Every time that a new goal is given to this action server, this callback function will be called, with a goal message provided as function argument, i.e. *goal_msg* in this example. As a last argument for the *SimpleActionServer constructor* it is instructed that the ActionServer shouldn't start until we explicitly tell it to do so. After the *SimpleActionServer* object is created, the server is started manually. It is often best to start the Action Server manually at the end of the *__init__* function, since more variables have to be initialized after the *SimpleActionServer* is created. This would create the possibility of an call on the callback function, while not all member variables of the class are initialized. Hence, the Action Server is started once all variables are initialized.

The implementation of the callback function in this case directly finishes or fails based on chance. Of course for complex tasks, it is possible to let the callback function only change certain member variables, causing a potential seperate loop execute the task. It is possible to use the *set_succeeded* or *set_failed* functions outside of the callback function.

```
#! /usr/bin/python
import rospy
import actionlib
from fake_navigation.msg import FakeNavigationAction
import random

class FakeNavigationServer(object):

  def __init__(self):
    self.action_server = actionlib.SimpleActionServer("/navigate", FakeNavigationAction, self.
    goal_callback, auto_start = False)
    self.action_server.start()

  def goal_callback(self, goal_msg):
```

```
    sleep_time = random.randint(2, 5) # Sleep for 2 to 5 seconds
    rospy.sleep(sleep_time) # Sleep (pretending the robot is driving)

    success_probability = 0.9 # Decrease the probability to increase failure rate
    success = True if random.random() <= success_probability else False # Navigation will succeed
     most of the time, this can simulate the robot getting stuck.

    if success:
      self.action_server.set_succeeded() # Set the action server to success (robot has arrived at
     its goal)
    else:
      self.action_server.set_aborted() # Set the action server to aborted (pretends the robot got
     stuck)

if __name__ == "__main__":
  rospy.init_node("fake_navigation_server")
  fake_navigation_server = FakeNavigationServer()
  rospy.spin()
```

Listing 4.11: A dummy Action Server that implements a fake navigation program

4.5.2 Action messages

```
# this is a comment

#First, the goal variables are specified.
string goal
---
#Second, the return variables are specified

---
#Third, the current progress feedback variables are specified.
```

Listing 4.12: An example Action declaration

```
#! /usr/bin/python
import rospy
import actionlib
import time
from fake_navigation.msg import FakeNavigationAction, FakeNavigationActionGoal

if __name__ == "__main__":
  rospy.init_node("fake_navigation_client")
  client = actionlib.SimpleActionClient('/navigate', FakeNavigationAction)
  client.wait_for_server()
  goal = FakeNavigationActionGoal()
  goal.goal = "kitchen"
  client.send_goal(goal)


  while True:
    #do other things:
    time.sleep(1)
    #check the Action Server status:
    if client.get_state() == actionlib.GoalStatus.SUCCEEDED:
        break # The Action Server was succesfull.
    elif client.get_state() == actionlib.GoalStatus.ABORTED:
        break # The Action Server failed.
```

Listing 4.13: An Action Client example

4.6 Behaviours

As the usage of many different ROS packages form the basis of the functionality of the platform, these packages require a top level program which command these lower level packages. For this, the platform uses a so-called

behaviour tree. This behaviour structure implements a finite state machine, where a main behaviour is responsible for state transition logic, and multiple subbehaviours implement the communication with lower level packages, depending on the current state the system is in. Where the main behaviour implements a finite state machine with support for multiple custom states, the subbehaviours implement a small state machine also. Typically, a subbehaviour is in a standby state before it is requested to start. Once it receives a *start* command, the subbehaviour will run, until it either *finishes* or *fails*. This section will give some illustrative examples of the usage of this behaviour architecture.

4.6.1 Example: Navigation

When the main behaviour decides that the platform should navigate the an location A, the main behaviour will start a navigation subbehaviour and passes the requested target location. Next, the subbehaviour will look up the respective map coordinates of the requested location, and send these coordinates in the form of an Action goal to the Move-Base node in the system. While Move-Base is executing autonomous navigation, the navigation subbehaviour can track the progress of the navigation and checks whether the navigation *finished*, *failed* or whether MoveBase is still executing the navigation. If MoveBase either finished or failed, the subbehaviour signals the main behaviour about the status update, whereafter the main behaviour decides what subbehaviour the platform should enter next. E.g., if the navigation failed due to an obstruction in the area, the mainbehaviour could decide to navigate to another location. If navigation succeeded, the main behaviour can request a subbehaviour to e.g. return objects at that location if the behaviour implementation contains e.g. object recognition.

4.6.2 Behaviour configuration

When starting to create a behaviour tree, it is required to specify the names of your main and sub behaviours. For this, create a YAML file in the directory *behaviours/config*. An example configuration file is described in listing 4.14.

```
main_behaviour: MyMainBehaviour
sub_behaviours:
 - Sub1
 - Sub2
```

Listing 4.14: my_behaviours.yaml

As can be seen, this configuration defines a mainbehaviour called *MyMainBehaviour* together with two subbehaviours named *Sub1* and *Sub2*. It is possible to add extra subbehaviours later-on, by adding more subbehaviours to the configuration file. The generation of the code structure for the respective configuration is described here after.

4.6.3 Behaviour generation

Once the behaviour configuration is defined, the following command can be used to generate the code template for the behaviours.

roslaunch behaviours create_behaviours.launch file:=my_behaviours

The output of the execution of this command will look similar as shown in listing 4.15.

```
... logging to /home/marc/.ros/log/588d5cc8-fd22-11ec-8885-f832e4bc61e1/roslaunch-marc-System-
    Product-Name-5713.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://marc-System-Product-Name:43111/

SUMMARY
========

PARAMETERS
 * /main_behaviour: MyMainBehaviour
 * /rosdistro: melodic
```

```
 * /rosversion: 1.14.11
 * /sub_behaviours: ['Sub1', 'Sub2']

NODES
  /
    behaviours (behaviours/createBehaviours.py)

auto-starting new master
process[master]: started with pid [5723]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 588d5cc8-fd22-11ec-8885-f832e4bc61e1
process[rosout-1]: started with pid [5734]
started core service [/rosout]
process[behaviours-2]: started with pid [5737]
Creating new behaviour file:  MyMainBehaviour
Creating new behaviour file:  Sub1
Creating new behaviour file:  Sub2
========================================================================================REQUIRED process
    [behaviours-2] has died!
process has finished cleanly
log file: /home/marc/.ros/log/588d5cc8-fd22-11ec-8885-f832e4bc61e1/behaviours-2*.log
Initiating shutdown!
================================================================================
[behaviours-2] killing on exit
[rosout-1] killing on exit
[master] killing on exit
shutting down processing monitor...
... shutting down processing monitor complete
done
```

Listing 4.15: Output of the behaviour generation

This command generates the template for the previously shown configuration file. Note that the *file* argument refers to the name of the YAML file, without using the *.yaml* extension. For your own behaviours, replace the file name with the name of your own behaviour configuration file name.

4.6.4 States

Before going into programming behaviours, first it's important to note that both the main behaviour and subbehaviours implement *finite state machines* (FSM). At all times these behaviours are in a certain state. Although the states of a subbehaviour are relatively simple, as we will see later on, the main behaviour typically implements a more complex FSM, which defines what the robot should do in a given situation. The states that can be used by the behaviours are declared in an *enumeration* in the file *behaviours/scripts/utils/state.py*. The states in this file are shared between the different behaviour trees that can be build in this package. The template of this file is shown in listing 4.16.

```
from enum import Enum, unique

@unique
class State(Enum):
    idle = 0
    start = 1
    finished = 2
    failed = 3
```

Listing 4.16: The default behaviour states

4.6.5 The behaviour class

Before we go into behaviour programming, let's first take a look at the *AbstractBehaviour* class in listing 4.17, which forms the base class for all other behaviours. This class can be found in the file *behaviours/scripts/utils/abstractBheaviour.py*. **NOTE: Do not edit this file.**

```
from state import State
```

```
class AbstractBehaviour(object):

    def __init__(self, all_behaviours):
        self.all_behaviours = all_behaviours
        self.state = State.idle
        self.failureReason = ""

    def get_behaviour(self, name):
        return self.all_behaviours.get_behaviour(name)

    def get_state(self):
        return self.state

    def set_state(self, state):
        self.state = state

    def stopped(self):
        return self.state == State.failed or self.state == State.finished

    def finished(self):
        return self.state == State.finished

    def failed(self):
        return self.state == State.failed

    def get_failure_reason(self):
        return self.failureReason

    def finish(self):
        self.state = State.finished

    def fail(self, reason):
        self.state = State.failed
        self.failureReason = reason

    def start(self):
        self.state = State.start
```

Listing 4.17: The base class for all behaviours

As this class forms the base class for all other behaviours, all functions are inherited by the other behaviour files, hence they are available to these classes. Note that it is possible to overload these functions when required. For illustration, let's take a look at the template main behaviour, as shown in listing 4.18.

```
from utils.state import State
from utils.abstractBehaviour import AbstractBehaviour
import rospy


class MyMainBehaviour(AbstractBehaviour):

    def init(self):
        pass

    def update(self):
        pass
```

Listing 4.18: The template of a main behaviour

As can be seen from the *abstractBehaviour*, the main behaviour has the member variable *state*, which is initially initialized to the state *idle*. This state variables are held by both main behaviour and subbehaviours, which defines which state that respective behaviour is in. State transitioning is done by changing this state variable, preferably by using the function *set_state*.

We can see that the main functions of the main behaviour are the *init* function and the *update* function. The *init* function is called at the start of the behaviour execution and is meant to initialize variables. The *update* function is called every 0.2 seconds. The calling of this update function is done by a separate python script that manages the behaviours which is not discussed here. Hence note that from a behaviour programming perspective there is no need to implement a *main* function, since this already exists in the provided behaviour ecosystem. The

update function is typically implemented as a *if-else* ladder, or a *if-elif* ladder in python lingo. At every update call, the function checks which state the behaviour is in, and executes corresponding code and checks whether state transition conditions are met to set the state variable to another value.

4.6.6 Brain

Before going into an example implementation of a behaviour, we'll take a look first at the behaviour class, as shown in listing 4.19. While there is no need to edit this code at any time, it gives an illustration of how the behaviours are initialized an how they are updated.

```
#!/usr/bin/python
import rospy
from utils.state import State
from utils.AllBehaviours import AllBehaviours

import signal

class Brain(object):

    def __init__(self):
        self.ab = AllBehaviours()
        self.all_behaviours = self.ab.get_all_behaviours()

        self.all_behaviours[0].start()
        for b in self.all_behaviours:
            b.init()

        self.rate = rospy.Rate(10)
        self.running = True

    def stop(self):
        self.running = False

    def run(self):

        while self.running:
            # Check if first/main behaviour is not in stop state
            for b in range(len(self.all_behaviours)):
                behaviour = self.all_behaviours[b]
                should_stop = (behaviour.get_state() == State.finished or behaviour.get_state()
    == State.failed or behaviour.get_state() == State.idle)
                if b == 0 and should_stop:
                    self.running = False

                elif not should_stop:
                    behaviour.update()

            self.rate.sleep()

if __name__ == "__main__":

    rospy.init_node("Brain")
    brain = Brain()

    signal.signal(signal.SIGTERM, brain.stop)
    signal.signal(signal.SIGINT, brain.stop)

    brain.run()
```

Listing 4.19: The 'Brain' class

This brain class is responsible for instantiating the different behaviour files, where they are stored in an array. First, the main behaviour is directly set to a starting state, whereafter all behaviours are initialized using their *init* function. Hereafter the Brain class will check for all behaviours whether they are in a stopping state or not. If not, the update function is called for these behaviours with 10 Hz.

4.6.7 Main behaviour example

To illustrate behaviour programming, let's take a look at a very simple behaviour for a robot, namely walking ten steps forward, turning around and repeat. For this we'll first create two states for this, namely *walking* and *turning*. For the subbehaviours we'll add an extra state, namely *executing*. Hence we'll end up with a set of states as shown in listing 4.20.

```
from enum import Enum, unique

@unique
class State(Enum):
    idle = 0
    start = 1
    finished = 2
    failed = 3
    walking = 4
    turning = 5
```

Listing 4.20: States for the example behaviour

Next, we can take a look at the main behaviour implementation, which is shown in listing 4.21. First, references to the two subbehaviours are obtained using the *get_behaviour* function. As described before, these subbehaviour objects have already been instantiated by the Brain class, hence merely a reference is provided to the main behaviour.

After the initialization, the update function of the main behaviour is called every 0.1 seconds. Here we can see a small finite state machine (FSM), where the FSM first starts in a *start* state, where the state machines directly switches to a walking state. At every state a different subbehaviour is typically active. Hence, when the state is being switched to *walking*, also the *walking* subbehaviour is set to a start state, causing its own update function also to be called every 0.1 seconds. Once the main behaviour is switched to the walking state, the main behaviour will, as implemented by the first if-elif ladder, check what is should do when in this state. Here we can see that at every update the main behaviour merely checks whether the walking subbehaviour either finished or failed. Typically, these conditions are the basis for a state transition in the main behaviour. In this small example only the *finished* case is handled, where in real applications, all failures should be handled by the main behaviour, causing the robot to always behave in a useful manner in every situation. Once the walking subbehaviour is finished, that subbehaviour is first being reset, such that all member variables have a fresh initialization, causing it to be ready for a new call. After this reset, the main behaviour state is set to *turning*, whereafter the turning subbehaviour is started.

```
from utils.state import State
from utils.abstractBehaviour import AbstractBehaviour
import rospy


class MyMainBehaviour(AbstractBehaviour):

    def init(self):
        self.walkingSubbehaviour = self.get_behaviour("Sub1")
        self.turingSubbehaviour = self.get_behaviour("Sub2")
    def update(self):
        print self.state
        if self.state == State.start:
            self.set_state(State.walking)
            self.walkingSubbehaviour.start()
        elif self.state == State.walking:
            if self.walkingSubbehaviour.finished():
                self.walkingSubbehaviour.reset()
                self.set_state(State.turning)
                self.turingSubbehaviour.start()
            elif self.walkingSubbehaviour.failed():
                self.walkingSubbehaviour.reset()
                pass #Do something smart with the knowledge that this behaviour failed.
        elif self.state == State.turning:
            if self.turingSubbehaviour.finished():
                self.turingSubbehaviour.reset()
                self.set_state(State.walking)
                self.walkingSubbehaviour.start()
            elif self.turingSubbehaviour.failed():
```

```
                self.turingSubbehaviour.reset()
                pass #Do something smart with the knowledge that this behaviour failed.
```

Listing 4.21: The example main behaviour

4.6.8 Subbehaviour example

Now that we know how a main behaviour behaves, let's take a look at a simple subbehaviour implementation, as shown in both listing 4.22 and 4.23. Here, these subbehaviours first initialize a member variable in their *init* function, which is this dummy example is only a counter variable. At the start of the behaviour program all subbehaviour states are first initialized to an *idle* state, causing their update function not to be called, until this state is changed by the main behaviour. Once, the subbehaviour state is set to start, the update function will be called, and there is room for a first task execution in the starting state, where after the state transitions to *walking*. In an actual implementation a subbehaviour could e.g. send a goal to an Action Server in the starting state, whereafter it only checks whether that Action Server finished or failed in a different state. Here, while 'walking', the update function merely counts up to 10, whereafter it sets it's state to *finished*.

```
from utils.state import State
from utils.abstractBehaviour import AbstractBehaviour
import rospy


class Sub1(AbstractBehaviour):

    def init(self):
        self.counter = 0

    def update(self):
        if self.state == State.start:
            print "Starting to walk"
            self.set_state(State.walking)
        elif self.state == State.walking:
            print "Walking counter = ", self.counter
            if self.counter == 10:
                self.finish()
            self.counter += 1

    def reset(self):
        self.state = State.idle
        self.init()
```

Listing 4.22: Example of the walking subbehaviour

```
from utils.state import State
from utils.abstractBehaviour import AbstractBehaviour
import rospy


class Sub2(AbstractBehaviour):

    def init(self):
        self.counter = 0

    def update(self):
        if self.state == State.start:
            print "Starting to turn"
            self.set_state(State.turning)
        elif self.state == State.turning:
            print "Turning counter = ", self.counter
            if self.counter == 10:
                self.finish()
            self.counter += 1

    def reset(self):
        self.state = State.idle
        self.init()
```

Listing 4.23: Example of the turning subbehaviour

4.6.9 Example output

The output of the provided example behaviour above is shown here in listing 4.24.

```
... logging to /home/marc/.ros/log/28be4868-fe12-11ec-a4e6-f832e4bc61e1/roslaunch-marc-System-Product-Name-17862.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://marc-System-Product-Name:40151/

SUMMARY
========

PARAMETERS
 * /main_behaviour: MyMainBehaviour
 * /rosdistro: melodic
 * /rosversion: 1.14.11
 * /sub_behaviours: ['Sub1', 'Sub2']

NODES
  /
    brain (behaviours/brain.py)

auto-starting new master
process[master]: started with pid [17872]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 28be4868-fe12-11ec-a4e6-f832e4bc61e1
process[rosout-1]: started with pid [17883]
started core service [/rosout]
process[brain-2]: started with pid [17886]
Starting to walk
Walking counter =  0
Walking counter =  1
Walking counter =  2
Walking counter =  3
Walking counter =  4
Walking counter =  5
Walking counter =  6
Walking counter =  7
Walking counter =  8
Walking counter =  9
Walking counter =  10
Starting to turn
Turning counter =  0
Turning counter =  1
Turning counter =  2
Turning counter =  3
Turning counter =  4
Turning counter =  5
Turning counter =  6
Turning counter =  7
Turning counter =  8
Turning counter =  9
Turning counter =  10
Starting to walk
Walking counter =  0
Walking counter =  1
^CWalking counter =  2
[brain-2] killing on exit
Traceback (most recent call last):
  File "/home/marc/repos/catkin_ws/src/robotics2122/behaviours/scripts/brain.py", line 47, in <module>
    brain.run()
  File "/home/marc/repos/catkin_ws/src/robotics2122/behaviours/scripts/brain.py", line 37, in run
    self.rate.sleep()
  File "/opt/ros/melodic/lib/python2.7/dist-packages/rospy/timer.py", line 103, in sleep
    sleep(self._remaining(curr_time))
  File "/opt/ros/melodic/lib/python2.7/dist-packages/rospy/timer.py", line 134, in sleep
```

```
    rospy.rostime.wallsleep(duration)
  File "/opt/ros/melodic/lib/python2.7/dist-packages/rospy/rostime.py", line 277, in wallsleep
    time.sleep(duration)
TypeError: stop() takes exactly 1 argument (3 given)
[rosout-1] killing on exit
[master] killing on exit
shutting down processing monitor...
... shutting down processing monitor complete
done
```

Listing 4.24: Output of the example behaviour

Chapter 5

Package overview

The software packages used to run the robot applications in in a ROS environment are listed here. These packages each contain one or more programs, such as simulations, specific actions of the platform and different algorithms, together with parameter configurations, such as a model of the platform. The functionality of the packages are generally described. More run instructions are described in section 7 where applicable.

- **approach_objects:**
  This packages implements an algorithm that drives the platform towards an objects, such that this object is positioned in a specific location with respect to the platform. This package is used to orient the platform in a such a manner that the object is in range of the arm, allowing for manipulation, as well as being properly in view of the camera, such that the platform can apply object detection and recognition. More details on the approach are described in section 6.3.

- **arm_description:**
  This package contains the 3D model of the arm, where all parts of the arm are defined separately, allowing for modelling of the arm in different configurations. This model is used by other packages for movement planning and control.

- **arm_gazebo:**
  This package contains configurations of the arm, when used in simulation.

- **arm_moveit_config:**
  This package contains the configurations for the planner of the simulated arm, i.e. MoveIt.

- **body_description:**
  This package contains models of the entire physical platform. These models are used by other packages to e.g. transform between coordinate systems, or visualise the robot in visualisation tools.

- **body_gazebo:**
  This package contains the 3D model of the robot, as used by the simulation environment.

- **image_cropper:**
  The image cropper package contains a program that collects image ROIs for the creation of datasets where object recognition classifiers can be trained on. The program is started for every different object label separately. It requests ROI information from object detection together with the most recent camera image. When the operator presses the space-bar on the keyboard, it will safe this current image frame, together with ROI information in a folder, that is will contain only data examples of the specified image label. Using this program, datasets can be created for different objects, and stored in such a manner that it allows for data augmentation techniques, e.g. using slightly different ROI dimensions and sizes for the object.

- **grasp_plugin:**
  This section contains a plugin for object interaction in Gazebo.

- **keyboard_teleop:**
  The keyboard_teleop package contains a program that allows for sending driving velocity targets to the platform, using a keyboard interface. This allows driving the robot around using a keyboard.

- **manipulation_server:**
  This package contains code to interact with MoveIt.
- **object_recognition:**
  This package contains code for object recognition as described in section 6.4.
- **mecanum_gazebo_plugin:**
  This package contains a gazebo plugin to enable gazebo to simulate the macanum wheel functionality of the platform
- **point_cloud_functions:**
  This package contains a program that implements many functions for pointcloud analysis. These function are made available to other programs running in the ROS environment.
- **bounding_box_server:**
  This package contains the code for the bounding box estimation as described in section 6.5.1.
- **behaviours:**
  This package contains the code for behaviour programming, as described in section 4.6.
- **navigation:**
  This package contains the code for autonomous navigation, as described in section 6.2.

Chapter 6

Theory

6.1 Behaviours

Given the many ROS nodes that implement capabilities of the robot, a top level control architecture is required. For this a behaviour tree is used, which is briefly described here. A detailed description on behaviour programming is provided in section 4.6.

The behaviour tree as used here consists of a main behaviour, with multiple subbehaviour. The main behaviour implements a finite state machine, which described what the robot should be doing at a specific moment. Also, it is responsible for handling succesfull executions and failed executions. E.g. while navigating the robot might get stuck, or no path can be found. In this case the main behaviour has to decide what should be done next, e.g. navigating to a recovery point or something else. Hence, at all times, the main behaviour is in a state, in which a subbehaviour is running. The successes or failures of subbehaviours trigger a state transition in the mainbehaviour. A properly implemented main behaviour catches all possible situations of successes and failures and will never get stuck in a state.

Subbehaviours are started by the main behaviour, typically after a main behaviour state transition. A subbehaviour typically deligates tasks to other ROS nodes, e.g. to an action server. Here the subbehaviour sends a goal to the action server, and checks every iteration what the execution status is of the action.

Given the non-parallel nature of executing code of the main behaviour and subbehaviour, the code should never contain blocking function calls, or extensive looping, as this will block the rest of the behaviour code execution. Whenever a situation arises that such code is needed, it can always be wrapped into an action server in a seperate node that is then responsible for that task, where the behaviour architecture itself merely starts the process and checks the execution state.

6.2 Navigation

The platform is capable of autonomous navigation using the MoveBase package. This package consists of multiple programs that work together to perform autonomous navigation in a safe manner. An overview of these different programs is provided by figure 6.1. Here we can see that MoveBase consists of multiple programs that interact with platform specific programs, hence the core programs of MoveBase are platform independant themselves. First, we will take a look at the platform dependant programs, to provide context about the information that MoveBase works with. After that, we will go more in depth into the MoveBase specific programs. For detailed intructions on how to use the navigation stack, please refer to section 7.4.

6.2.1 Odometry

Odometry is an estimation of where the platform location, based on the wheel movement since the start of the odometry calculation. One of the upsides of odometry is the high refresh rate of information. Every 20 ms the odometry information is updated by the motor controller. Apart from this, odometry has one significant limitation when used on its own. Odometry will have an error is its movement estimation, which builds up over time. Hence, when used 'locally' in time, odometry is very usefull, though when traveling for a longer amount of time, the position estimation error will be too large to be used on its own.

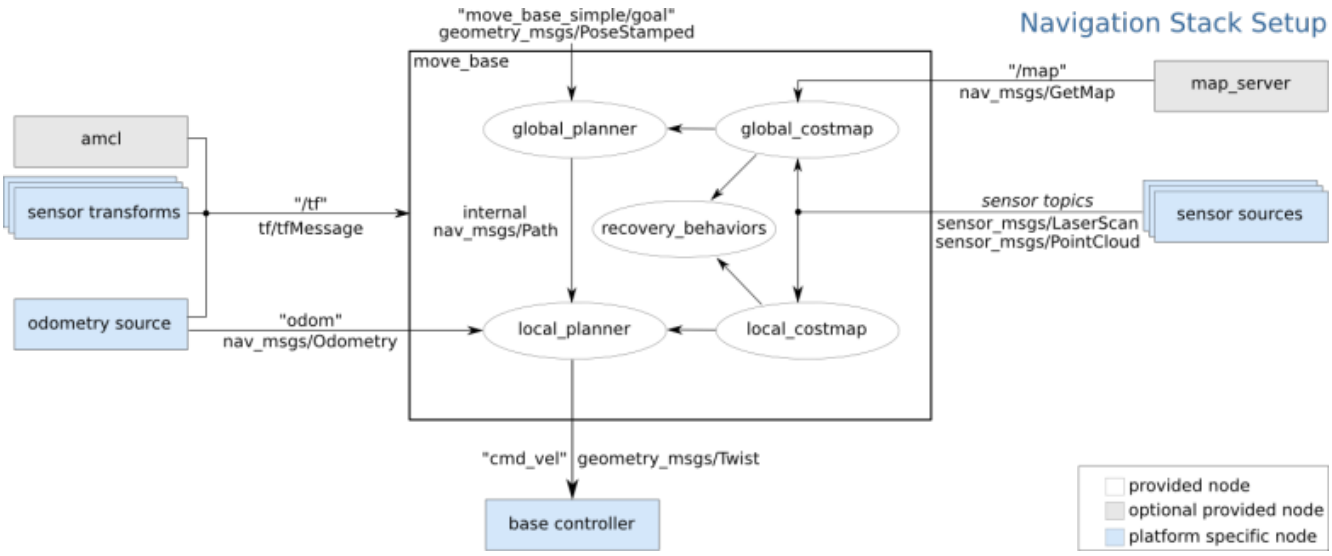
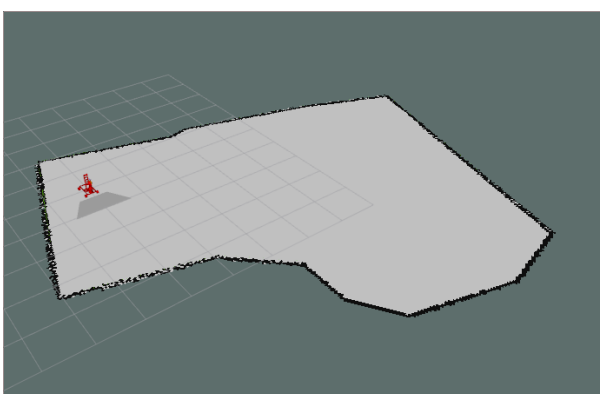


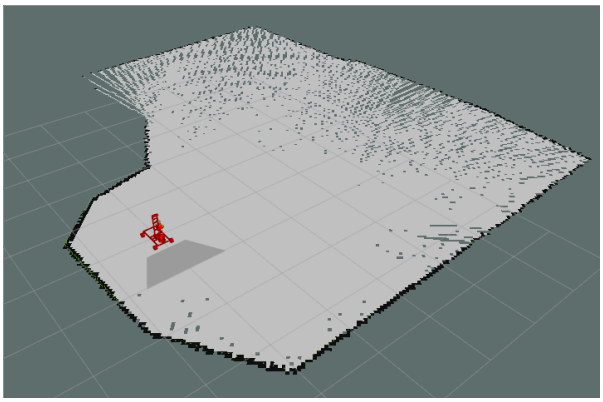
Figure 6.1: An overview of the different programs in MoveBase (source: `http://wiki.ros.org/move_base`)

6.2.2 Simultanious Localisation and Mapping

Simultanious Localisation and Mapping (SLAM) is performed by the program GMapping. When creating a map, a lidar is used, in combination with odometry data. While driving the robot around, lidar data is used to detect obstacles in the surrounding environment. Lidar data on its own can only be used to detect obstacles in the frame of reference of the lidar sensor, but combined with odometry information, GMapping can use the movement estimate to know where the sensor itself is, with respect to earlier scan locations. Of course, as described earlier, odometry has the inherent problem of an error that build up over time. To compensate for this, new lidar data is compared to earlier data that has been collected so far, to estimate the new robot position. For this, a Rao-Blackwellized particle filer is used, as described in [4] and [5]. This technique enables to robot to drive around and to create a map of the surrounding environment. The map consists of three kind of pixels. A pixel is labeled either as *unoccupied*, *occupied*, or as *unknown*. The latter label indicates that no data has been collected on this point is space so far. The driving is typically done manually by an operator, where the operator visually inspects the map that has been created so far. Once a map is created, where all relevant areas have been explored properly, where no *unknown* pixels are left in the area where the robot operates in, the map is usable for the localisation and path planning techniques described hereafter. Examples of a complete and incomplete map are shown in figure 6.2a and 6.2b respectively.



(a) A good map, without unexplored areas



(b) An incomplete map with unexplored areas

Figure 6.2: Two examples of the mapping process

6.2.3 Adaptive Monte-Carlo Localisation

Once a proper map has been created, the robot is able to use Adaptive Monte-Carlo Localisation (AMCL) to localise the robot in the map using the new lidar measurements and odometry information. This technique uses a particle filter approach, where particles are initially scattered in the map, and considered as potential locations. The algorithm uses these particles as follows: First the location of the particles are updated using new odometry data. Next, the lidar data at this new location is projected on the map from the perspective of each particle. If a lidar detection point corresponds with an occupied pixel on the map, this is considered as evidence for the likelihood of the particle location, hence a correlation between the map and the lidar scan is calculated. This correlation forms the basis of the probability that the particle location is correct. After each iteration, the particles are redistributed on the map, based on the probabilities of the particles of the last iteration. These particles are redistributed in such a manner that more particles are placed at likely location, and less particles are positioned at unlikely locations. After multiple iterations, the particles will converge to the location of the robot in the map.

6.2.4 Additional sensors for obstacle detection

In addition the lidar data, the MoveBase package is also capable of integrating other sensors for obstacle detection. An example, as used by this platform, is the usage of a depth camera, which provided pointcloud data to MoveBase. As the Lidar can only detect ostacles in a single 2D plane, it might not detect obstacles that don't obstruct this 2D plane. An example of this could be a table. The legs of the table will be detected by the lidar, although the table surface won't be visible to the lidar. As the robot needs additional information next to the lidar because of this, multiple sensors can be used by MoveBase to extend the perception of obstacles in the surrounding area of the robot.

6.2.5 Global costmap

Based on the initially created map, a global cost map is created. This costmap of the global map is used, as described on the following section to plan paths in. The global planner will search a path that minimises the cost as defined in the global costmap. Each obstacle in the map is assigned a high cost. Also, pixels in the area around obstacles are assigned a high cost, which is referred to as costmap inflation. The intention of costmap inflation is that the global planner doesn't plan paths that are too close to obstacles. E.g. if the robot has to travel around an obstacle without costmap inflation, the path will be as close as possible to that object, which might cause unintended collisions with a the object. By inflating the object in the costmap, the robot will choose a path that avoids the object at a safe distance. An example of a global costmap is shown in figure 6.3a.

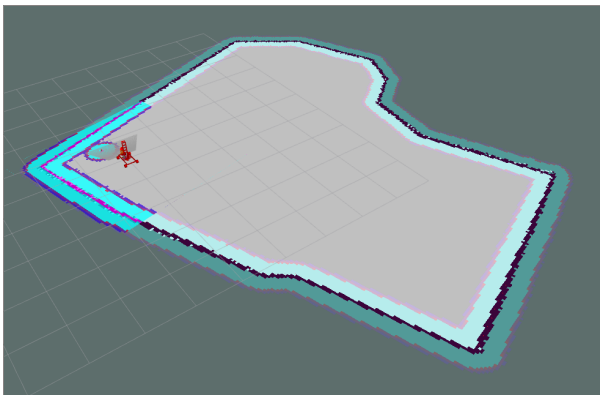
Initially the global costmap is created based on the lidar detections from the map, although while the robot is driving in the environment, obstacle information from additional sensors are also used, specifically the depth camera in front of the robot. If any obstacle is detected by the camera, this is added as a cost to the global costmap. If the camera and the lidar on a later point agree that an area has become clear of obstacles, this will be updated in the global costmap as well.

6.2.6 Global planner

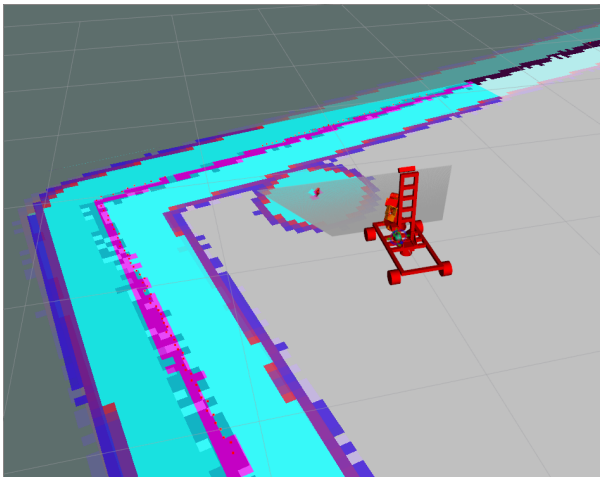
The above mentioned global costmap is used by the global planner to produce a path to a desired goal position. For this Dijkstra's Algorithm [6] is used. During navigation, this path will be recalculated many times, as new sensor data might adjust the global costmap in which the path is planned. E.g, if the initial map containes two table legs, where the area between the two table legs are clear, since no lidar detections where found in the are, it might plan a path through a table. While driving towards a table, the robot might see the table to with the camera, causing the costmap to assign a cost to this area. This would cause the current path plan to fail, hence during execution the path planning is running many times during execution to ensure that the robot is able to follow an optimal path based on the information collected so far.

6.2.7 Local costmap

Where the global costmap is created on past experience, the local costmap only contains the current sensor data. This map contains only a relatively small area around the robot is centered around it. This local costmap is used by the local planner, which is described hereafter. An example of a local costmap is shown in figure 6.3b.



(a) An example of a map, with the global costmap overlayed



(b) An illustration of a local costmap. In here you can clearly see the cost area of an red box that's been placed in the environment.

Figure 6.3: Illustrations of a local and global costmap

6.2.8 Local planner

Given the information from the local costmap, a very local plan is generated. Where the global planner produced a path that is save, in accordance with the global costmap, the local planner only generates a small path, and takes required velocity commands into consideration. The local planner used by this platform is the 'Dynamic Window Approach' (DWA) planner [7]. A brief description of the DWA algorithm, as described in the ROS documentation `http://wiki.ros.org/dwa_local_planner`:

- Discretely sample in the robot's control space (dx,dy,dtheta)
- For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.
- Evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
- Pick the highest-scoring trajectory and send the associated velocity to the mobile base.
- Rinse and repeat.

The obtained desired velocities are provided to the Base controller, as described hereafter, for the execution of these velocities.

6.2.9 Base controller

The velocity commands that are computed by the local planner are send to the Base controller, which is platform specific. For this platform, the motor controllers act as the base controller, and are able to receive velocity commands. These desired velocities are here translated to the required wheel velocities to let the robot drive with the desired velocities. These desired wheel velocities are provided to PID loops for each wheel. The encoders in the wheel provide accurate estimations of the current wheel velocities, hence the PID controllers are able to execute the wheel velocities as desired.

6.2.10 Recovery behaviours

It is possible for both the local planner and the global planner to get in a situation where no valid plans can be generated. E.g. a changing environment could cause the robot to get stuck. MoveBase provides built in recovery behaviour to make an attempt to get the robot 'unstuck'. E.g., a robot with a circular footprint would be able to

rotate around its axis, such that all sensors can observe the direct surroundings of the robot, which might provide new data that could cause the robot to get 'unstuck'. For this specific platform, no recovery behaviours are used. If the robot fails to arrive at the desired location, MoveBase will provide a 'failed' signal to the navigation behaviour, which on it's turn deligates the responsibility of handling this failure to the main behaviour.

6.3 Approaching objects

Once the robot has navigated to a location where one or more objects should be roughly in front of it, it has to be able to approach the object in such a manner that it is able to grasp it if required. Since navigation doesn't allow the robot to navigate close enough to the object because of possible collisions, a separate program is used to approach objects. For this the pointcloud data from the front 3D camera is used. First, the floor is removed from the pointcloud by filtering out the largest horizontal surface in view. Hereafter, the remaining clusters are isolated based on the distance between points, where each cluster of point has to be an object. The coordinates of the nearest object are obtained, for which a set of velocity commands are calculated that form a trajectory towards to object, while remaining at a specified distance. The approach towards an object is made available as a ROS Service towards the rest of the ROS ecosystem.

6.4 Object Recognition

Object detection is performed in a manner similarly to the technique described by the approach service, namely using cluster isolation in pointcloud analysis. Given a known cluster area in the pointcloud, coordinates of an approximate region of interest (ROI) for a 2D image are calculated, which can be obtained by another program.

An illustration of the overall object recognition pipeline is shown in figure 6.4. First, an object recognition subbehaviour sends a goal to an action server that implements the general object recognition program. On it turn, this action server makes a request for ROI information as described earlier. After this, a 2D camera image is requested from the camera data stream, from which the actual ROI is extracted given the provided coordinates. On this ROI preprocessing can be performed, e.g. scaling and pixel normalisation. After preprocessing, the image is provided to a convolutional neural network (CNN), which makes a prediction on which object can be seen in the ROI. Finally, the coordinates of the object, with respect to the robot, together with the image label are provided back to the object recognition sub behaviour as an action result, after which object recognition is completed.

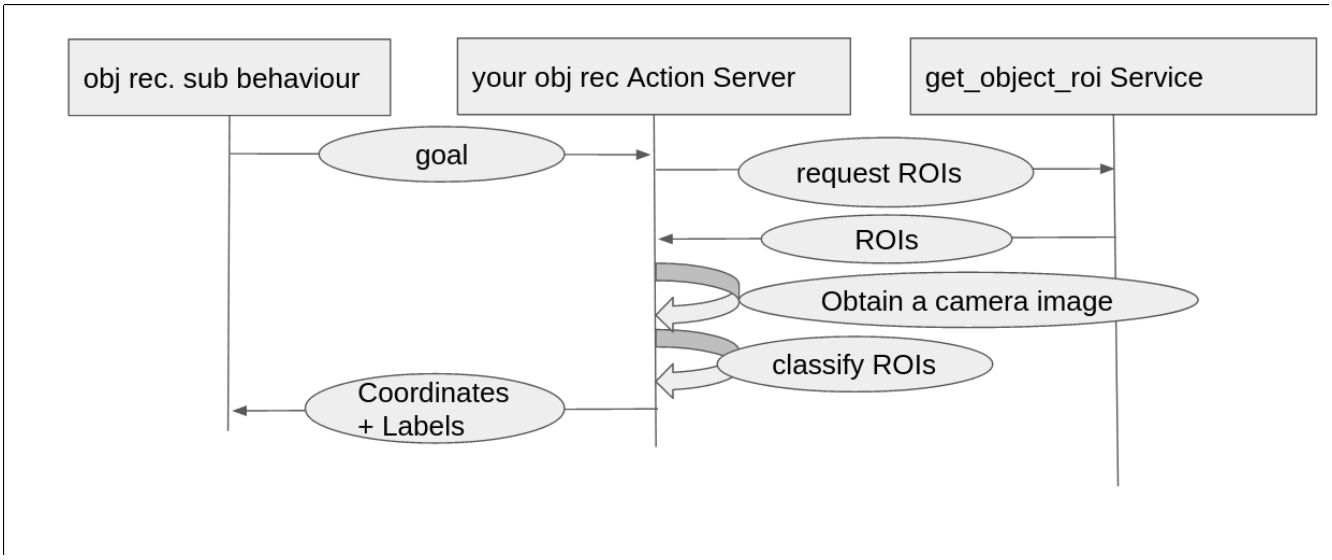


Figure 6.4: The overall pipeline for object recognition

6.5 Grasping

Grasping an object is task that consists of multiple subtasks. The robot needs to have a good estimate on where the object is, on what spot it should grasp the object, how to plan a movement trajectory for the arm, while avoiding collisions, and more..

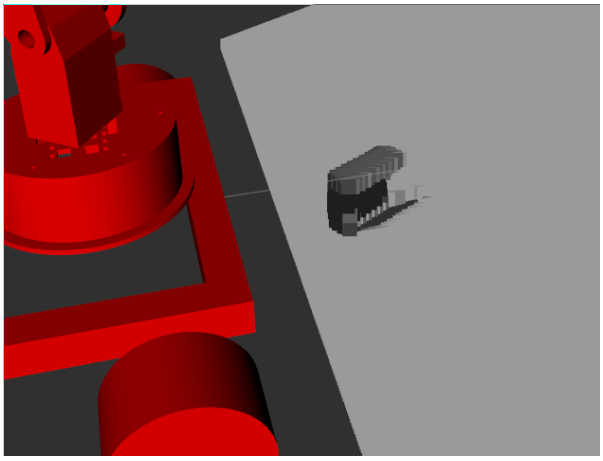
This robot is capable of picking up and transporting small boxes. Hence an important step is to obtain dimensional information about the box it should pick up.

6.5.1 Bounding box estimation

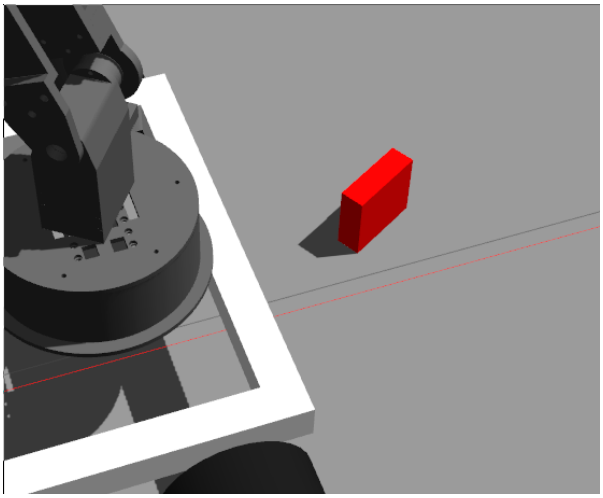
The problem of obtaining the sizes of the box is referred to as bounding box estimation. As the 3D camera in front of the robot can provide 3D pointclouds, the dimensions of the box are obtained using pointcloud analysis. First, we will need to detect the box in the pointcloud. Second, we need to obtain height, width en length information about the box, and finally the orientation of the box needs to be estimated.

Box isolation

An illustrative example of the situation where the robot has to estimate the bounding box information is shown in figure 6.5. Here in figure 6.5b we can see a red box placed in front of the robot. Figure 6.5a shows a pointcloud representation of this box. Note that the robot can only see the front side of the box, as the rear side is obscured by the front. The first step of extracting dimension information from this pointcloud example is to remove the floor from the pointcloud. This is done by a provided function in the PointCloud Library (PCL library). Once the floor is removed, only clusters of points remain, which are objects that are located on the floor. In this case only a single cluster remains, i.e. the red box.



(a) A box in front of the robot, as detected by the 3D camera.



(b) An box in front of the robot in simulation

Figure 6.5: A box in front of the robot

Orientation estimation

Once the red box is isolated in the pointcloud, we can't directly extract dimension information, since the object would need to be aligned with the x-axis, which is pointing forwards with respect to the robot. To do this, we first need to obtain the orientation of the red box. This is done using principle component analysis (PCA). First, the top surface of the object is isolated, of which we project all points on the x,y-plane, i.e. the horizontal plane. This causes us to obtain an set of point in 2D. Next, we perform PCA on the coordinates of these point, which provides us an eigenvector that point in the direction with the most spread in the points, i.e. the long side of the box. To obtain the angle of the box, we compare this eigenvector with the x-axis using the cosine rule, which provides us an angle estimate of the box.

Dimension estimation

Once the angle of the box is obtained, we can rotate the box, such that the long side is aligned with the x-axis. Given this situation we can extract the dimensions of the box, using the minimum and maximum values of the point coordinates in the x, y and z dimensions. The center coordinates of the box are obtained by using the mean values between the minima and maxima in x,y and z. Note, that we can simple average all coordinates in the detected pointcloud, since this average would be highly skewed. This is caused by not having any detection points of the rear side of the object, since only the front and top of the object is visible to the camera.

Once the bounding is estimated, it is published as a *MarkerArray*, allowing for visualisation in RViz. An example visualisation is shown in figure 6.6.

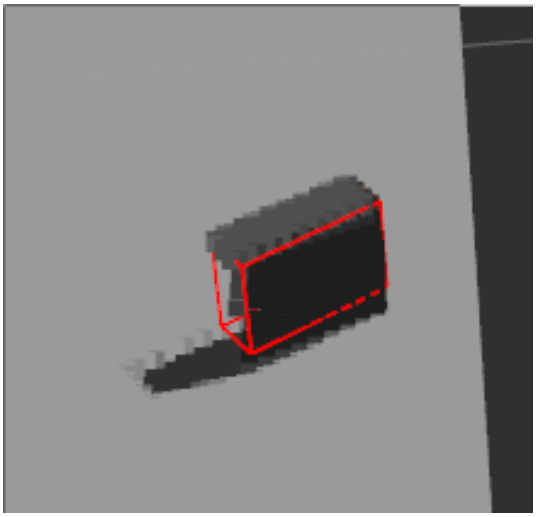


Figure 6.6: A visualisation of the estimated bounding box visualized in rViz

6.5.2 Motion Planning

As a Motion planning framework, MoveIt™is used. This framework is compatible with ROS, and allows for relatively easy motion planning for robotic arms. MoveIt supports inverse kinematics and movement planning of the arm towards a desired location, while avoiding surrounding obstacles. Interaction with MoveIt can be done either through code, by sending a target pose of the end-effector to the framework, or an user interactive approach can be used, where MoveIt can be used through rViz. As shown in figure 6.7, when adding *MotionPlanning* to rViz, it will show an 'ghost' arm, with an interactive end-effector which can be dragged and turned by the used, to indicate a goal position. Next, a plan can be generated, such that the arm will move in a manner where the end effector arrives at the desired location, after which this plan can be executed.

Planning Scene

The planning scene is a 3D representation of the surrounding environment which defines obstacles for arm movement. The planning scene is typically constructed based on pointcloud data from the front camera of the robot. In the case of the setup of this platform, no live pointcloud data is used to construct the occupancy map. It is assumed that the object is located on the floor, where the only obstacles for arm movement are the robot itself, the detected objects, and the floor. As the body of the robot is automatically taken in consideration by MoveIt, the floor is added to the scene, on top of which the detected objects are added. When an object is grasped by the arm, the respective object is attached to the arm for planning further trajectories. This allows the planner to not only avoid collisions between the arm and the environment, but also collisions between the grasped objects and the environment. An illustration of the planning scene is shown in figure 6.8.

6.5.3 Grasping objects

Given a situation where the robot is situated in front of an object, where the arm is in the home position, the first step is to perform the boundingbox analysis as described above. This provides the program with information

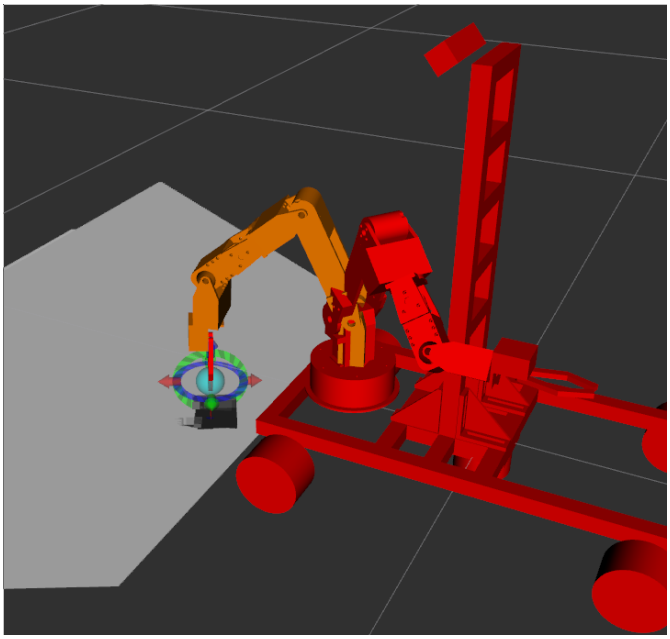


Figure 6.7: An example of the interactive usage of MoveIt in rViz.

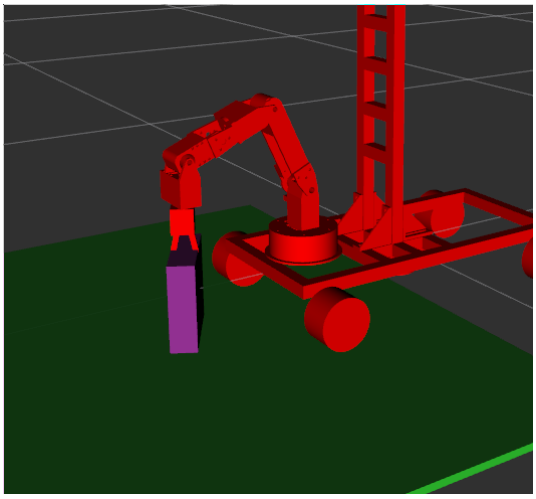


Figure 6.8: An RViz visualisation where the floor and the collision object are visible.

about the location, orientation and dimensions of the box. Given this, a box of the calculated dimensions and pose is added to the planning scene together with the floor. Next, one or more end-effector poses are proposed, above the object, where the end-effector is orientated such that the fingers can grasp the box in the middle, over the smaller x-y dimension. After this the fingers are opened, after which the end-effector is lowered such that it is ready to grasp the box. After this, the fingers are closed, and the collision object, i.e. the box, is attached to the arm in MoveIt, such that MoveIt can plan paths without the box colliding with the surrounding environment during further planning. Finally MoveIt can plan a path such that the arm travels back to the home position, whilst holding the box. This home position is chosen is such a manner that the arm is not visible to the camera, such that there is no camera obstruction during e.g. navigation.

6.5.4 Dropping Objects

If the robot has e.g. navigated to a location, where is should drop the object is it currently holding, to dropping is performed as follows. First, the floor is added to the planning scene, such that collision with the floor can be avoided. After this, a predefined end effector pose in front of the robot is given to MoveIt for planning. Once the

arm is in front of the robot and it is ready to drop the object, the fingers are opened, and the attached collision object, i.e. the box, is removed from the arm in MoveIt. This ensures that MoveIt knows that it doesn't need to consider the box anymore during movement planning. Once the box is dropped, a path is planned for the arm, to move safely back to the home position.

Chapter 7

Run instructions

7.1 Simulation environment

In order to start the simulation environment, use the following command:

roslaunch body_gazebo simulation.launch

This will start the simulator, as shown in figure 7.1. This will start the simulated robot, including all simulated sensors and actuators. To visualise the sensors and actuator status of the simulated robot, one can start rviz with the following command:

rviz

This will open RViz, as can be seen in figure 7.2. The recommended topic to add are listed below. When no map is available, make sure to set the fixed frame in the global options to **base_link**

- Robot model: **RobotModel**
- Point cloud data: **/camera/depth_registered/points** with the type **PointCloud2**
- Camera images: **/camera/color/image_raw** with the type **Image**
- Lidar data: **/scan_filtered** with the type **LaserScan**
- Arm Interaction: **MotionPlanning**
- Map: **/map** with the type **Map**

After this, the simulated robot is ready to be used with additional development packages.

7.2 Real world environment

7.2.1 Starting the platform

When using the platform in the real world, the operator should first power up the platform. First all circuit switches of the robot should be position in a downward off position. A picture of the power supply unit is shown in figure 7.3. Next the battery has to be connected to the power supply unit. Hereafter the power supply can be turned on. Next, the circuit switch for the computer and the USB-hub can be turned on. After a few minutes the computer has started, and an SSH connection becomes available on the WiFI network. The circuits for the motors and the arm should be turned on if the respective hardware has to be used during the session.

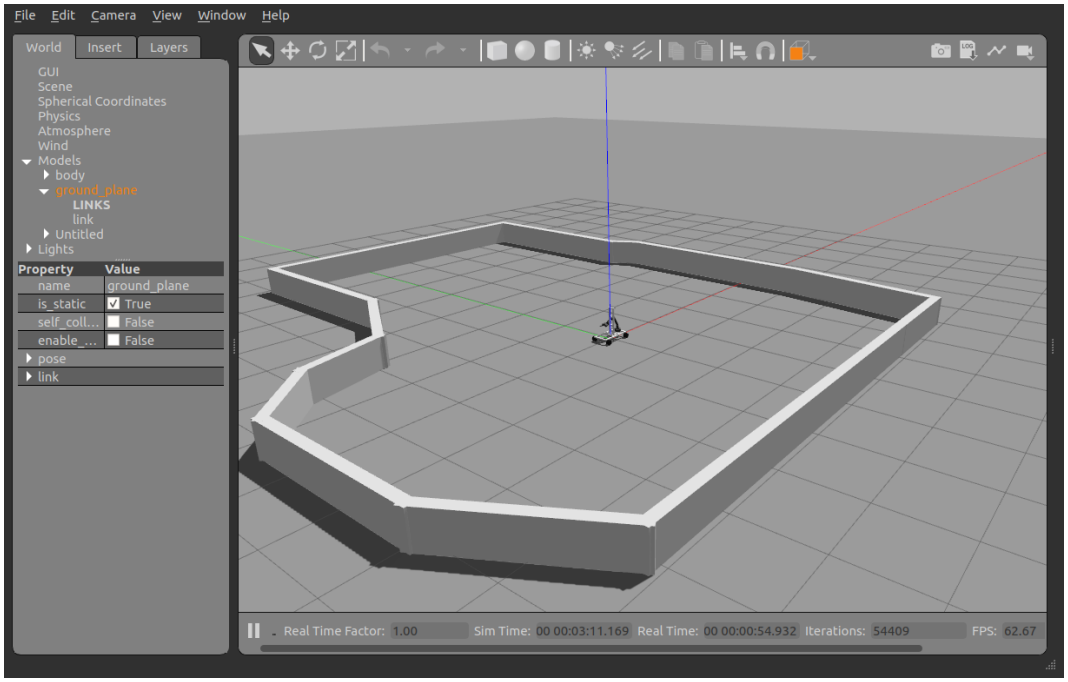


Figure 7.1: The Gazebo simulation environment

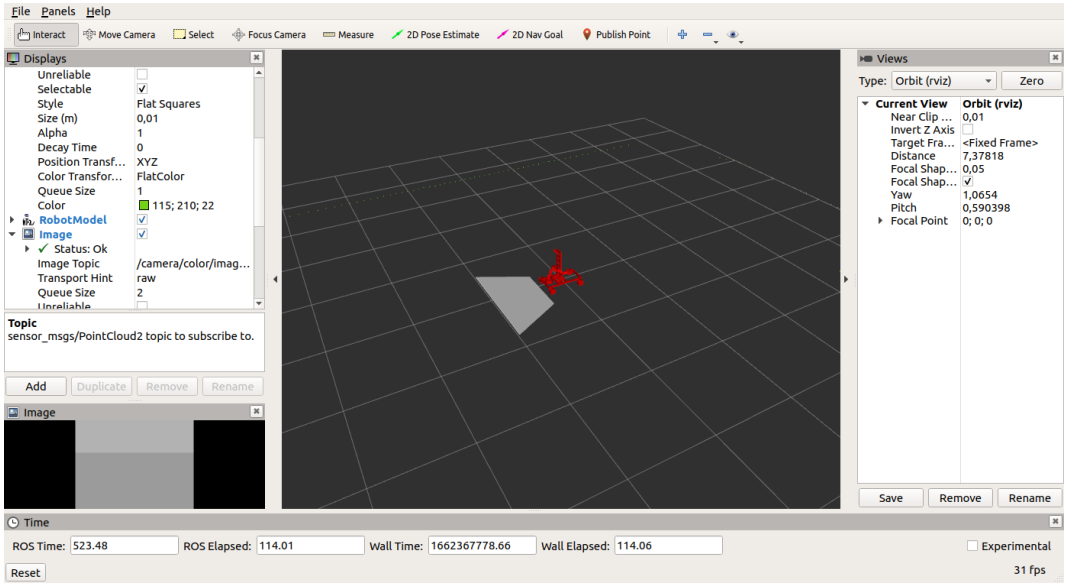


Figure 7.2: RViz while using the simulation environment

7.2.2 Starting the default programs on the platform

The default programs on the robot include publishers of the robot model and state, remote functions for other programs and sensor drivers. To start these programs, the operator should first SSH into the platform. Next, the programs can all be started at once with the following command:

roslaunch robot bringup_robot.launch

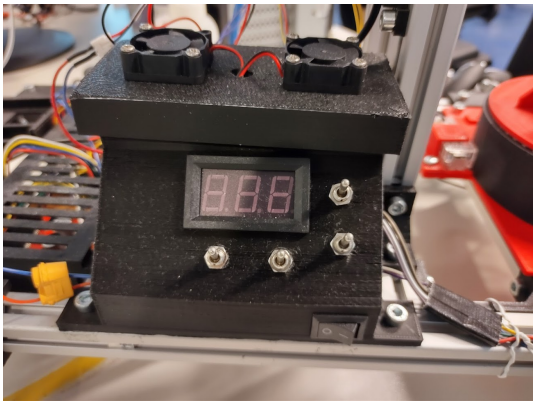


Figure 7.3: The power supply unit

Starting the arm controller

Before starting the arm controller, always verify that the arm is in a safe pose and turn on the power circuit after verifying this. Once the arm is powered, the operator should SSH into the platform, and start the arm driver using the following command:

roslaunch arm_controller arm_controller.launch

7.3 Visualisation

In order to visualise the sensor data and the state of the robot, the program RViz is used. To start RViz, the following terminal command can be used:

rviz

After starting rViz, first make sure that a valid *fixed_frame* is used, e.g. if the *map* frame is selected, while the navigation stack is not running, rViz won't be able to transform the received data to the requested frame of reference. By default, the *base_link* reference frame is suggested. After this, add *displays* to the visualisation using the *add* button.

7.4 Navigation

When preparing for autonomous navigation, the following processes have to be done. First, a map has to be created of the environment using lidar data in combination with odometry. Once a map of the environment has been created, the operator can define waypoints within this map, containing a pose of the platform within this map. These waypoints can be referred back to when sending navigation commands to the navigation programs.

7.4.1 Mapping

When the platform is placed within the environment, with the default programs running, the mapping process can be started using the following command:

rosrun gmapping slam_gmapping scan:=/scan_filtered

This program will listen to odometry data, together with lidar data, to map all lidar detections on a map. The operator has to collect lidar data by driving the platform around manually in the environment, in such a manner that all pixels in the map are classified as either empty, or as an obstacle. To drive the robot around, the keyboard teleop program can be used, which can be started by using the following command:

rosrun keyboard_teleop keyboard_teleop_node

While performing the mapping process, the operator can use RViZ to inspect the map that has been created so far. An example of a map in RViz is shown in figure 7.4. Once the entire environment has been mapped properly, the map can be saved to file using the following command:

rosrun map_server map_saver -f filename

This command will store the map to the current directory the terminal is in. As the map needs to be stored in the navigation package, the terminal directory first needs to be changed to *navigation/maps* This way the navigation stack can find the map when starting the navigation stack.

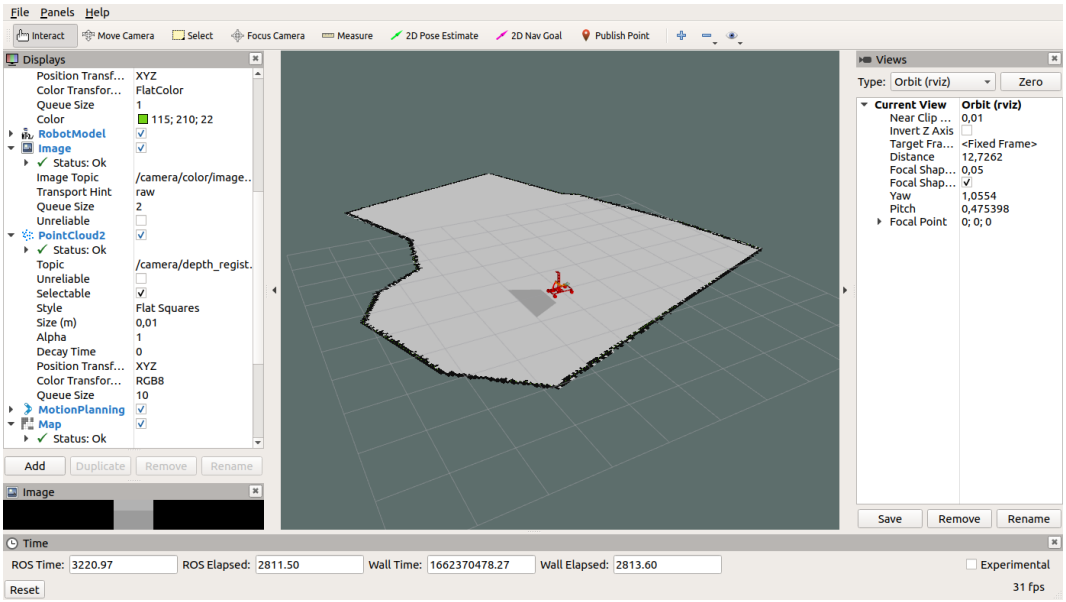


Figure 7.4: An example of a map in RViz

Starting the navigation stack

Before starting the navigation stack, check whether the navigation launch file is currently set to load the correct map. While using the navigation stack, please make sure that the **keyboard_teleop** program is not running, otherwise conflicting velocity commands will be send to the motor controller. Once ready, the navigation stack can be started with the following command

roslaunch navigation navigation.launch

7.4.2 Using RViz to send waypoints to MoveBase

Once that navigation stack is started, it is possible to send navigation waypoints to it. This can be done by either using RViz to send a waypoint goal, or using code. Using the *2D Nav Goal* in RViz, it is possible to click and drag a point and direction in the map, whereafter the coordinates and orientation are send to the navigation stack. This navigation goal will directly be executed by the navigation stack.

Once a navigation goal is send, the coordinates and orientation will also be printed in the terminal in which RViz was started. This can be used as described in the following section.

7.4.3 Defining waypoints

A waypoint is defined as coordinates in the map, together with an orientation and a name for the waypoint. It is recommended to create a *yaml* file to store these waypoints in. As described earlier, if a navigation goal is sent using RViz, the goal coordinates and orientation are printed in the terminal in which RViz was started. These

prints can be used to obtain the coordinates and orientation for desired waypoints. Once obtained, they can be written down in the yaml file, together with a name. After this, a behaviour can read this yaml file. This way, the behaviour code itself is agnostic about the coordinates and orientation itself, but can reason using waypoint names instead. When the actual coordinates and orientations are required to send goals to the navigation stack, the yaml file can be consulted.

7.4.4 Sending waypoints to MoveBase

Sending goal waypoint to the navigation stack, i.e. MoveBase, is done using and Action Server-Client pattern. Here an action client is implemented in a dedicated navigation subbehaviour. MoveBase itself implements an action server and is able to execute a *MoveBaseAction*. For the detailed contents of this action definition, the reader is refered to `http://docs.ros.org/en/fuerte/api/move_base_msgs/html/msg/MoveBaseAction.html`. The *MoveBaseGoal* contains a Pose, in which the coordinates and orientation can be provided. Once the goal is sent to MoveBase, the status of the navigation execution can be obtained as described earlier in the explanation of *Actions*.

Bibliography

[1] Stanford Artificial Intelligence Laboratory et al., "Robotic operating system."

[2] T. van der Zant and T. Wisspeintner, "Robocup@home: Creating and benchmarking tomorrows service robot applications," 2007.

[3] D. Coleman, I. A. Sucan, S. Chitta, and N. Correll, "Reducing the barrier to entry of complex robotic software: a moveit! case study," *CoRR*, vol. abs/1404.3785, 2014.

[4] G. Grisetti, C. Stachniss, and W. Burgard, "Improved techniques for grid mapping with rao-blackwellized particle filters," *IEEE Transactions on Robotics*, vol. 23, no. 1, pp. 34–46, 2007.

[5] G. Grisetti, C. Stachniss, and W. Burgard, "Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pp. 2432–2437, 2005.

[6] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[7] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *Robotics & Automation Magazine, IEEE*, vol. 4, pp. 23 – 33, 04 1997.