Master's Thesis

A flexible autonomous order picking robot

Lechner Dominik, BSc

Institute for Software Technology
Graz University of Technology
Head: Slany, Wolfgang, Univ.-Prof. Dipl.-Ing. Dr.techn.

Reviewer: Wotawa, Franz, Univ.-Prof. Dipl.-Ing. Dr.techn.
Supervisor: Wotawa, Franz, Univ.-Prof. Dipl.-Ing. Dr.techn.
Co-supervisor: Steinbauer, Gerald, Ass.Prof. Dipl.-Ing. Dr.techn.

Salzburg, May 2013

Abstract

An increasing number of industrial processes are being automated. On the one hand to unburden the employees because most of these processes are monotonously, stressful, often also really hard physical work and sometimes quite dangerous. On the other hand these processes are automated to improve efficiency and to reduce costs.

This thesis focuses on the process of order picking in warehouses. Although there is a lot of automation already present in large warehouses for high- and medium-volume items, in smaller warehouses order picking of low-volume goods is barely automated. The reason is that conventional automation concepts are customer-specific special purpose solutions and therefore really cost-intensive so that they are usually unsuitable under such conditions.

The goal of this thesis is a concept for a flexible autonomous mobile order picking robot which can operate in a standard warehouse without major modifications to the warehouse infrastructure. This robot should be able to do the whole order picking from receiving the customers order to the delivery of the order to a drop-off point. Therefore the robot has to be able to move through the warehouse autonomously, to fetch containers from shelves, transfer the required items from the containers to the order bin. To prove the concept a prototype robot, equipped with an omnidirectional drive for a maximum of mobility, different sensor for perceiving its environment, a container manipulation unit and a robotic arm to grasp the goods, is built.

A paper, giving a short overview about the concept and presenting preliminary results, has already been released [5]. This thesis reports the concept details, the implementation details of a prototype implementing parts of the concept as well as the results of experiments to prove the concept.

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Contents

List of Figures

List of Tables

Listings

Chapter 1.

Introduction

1.1. Motivation

What is the motivation for constantly increasing automation of order picking?

Today more and more goods have to be commissioned and delivered in shorter times. In order to achieve this, goods are stored in containers which in turn are stored in increasingly large and automated warehouses. Commissioning or order picking is a basic warehouse process to collect all goods with the correct amount which are needed for a customers order. In large warehouses, especially for goods with high and medium throughput, order picking is more and more automated. Fully automated picking uses automatic dispensers or vision based robot systems implementing the goods-to-robot method like the Schäfer Robo-Pick (SRP). Semi-automated picking implements the so-called goods-to-person method where the picking is done manually by an employee with a throughput up to 1000 pieces/hour. The employee is guided through the whole picking process by an assistance system which also continuously checks for picking errors. Whether picking is done fully automatic or semi-automatic, standard conveyor technique is used to bring the containers to the picking station.

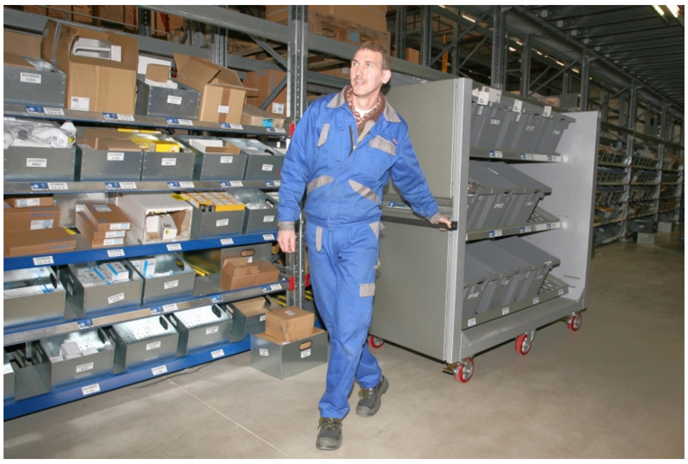


Figure 1.1.: Manual order picking (Photo courtesy of KBS Industrieelektronik GmbH)

If such automation is economically unviable or unfeasible for some reason, picking has to be done manually according to the person-to-goods method (see Figure 1.1). This means an employee usually has to walk through the warehouse and for each article has to fetch the right container from the shelf, take out the correct number of pieces, put the container back and continue with the next article. Similar to the goods-to-person method there are also different assistance and control systems for this kind of order picking like Pick-by-Voice, RF-picking, Pick-by-Light [26] and so on. These assistance and control systems are not only there to unburden the employees. They are necessary to keep picking errors in an acceptable range.

There are two main motivations for automating the process of order picking. Manual order picking is a very monotonous and - especially in the case of the goods-to-person method - a very stressful kind of work. The throughput of the person-to-goods method is much lower than in case of the goods-to-person method but therefore not the less strenuous. For this reason the first motivation for automation is to unburden the employees so that they can focus on more important things like the customer needs and their satisfaction. The second motivation for automation is an economical one. In order to be competitive, it is very important that warehouses work as efficient as possible. This means that a customers order has to be processed and delivered as fast as possible with a minimum of wrong deliveries. It is also important that the costs per order are kept as low as possible. In combination this can only be achieved through an effective fully automated process.

What is the motivation for developing an autonomous mobile order picking robot?

Today a lot of solutions are already available to automate warehouses. These are usually special purpose solutions, individually designed for a given warehouse or consist of a completely new warehouse with all the necessary infrastructure. Such solutions require a lot of money, time and work and especially in case of adapting an existing warehouse will interrupt normal business operation. Similar to the manual person-to-goods method an autonomous mobile order picking robot would implement a kind of robot-to-goods method. The robot could either assist human employees or replace them without major modifications to the warehouse infrastructure. These few necessary changes could be done without interruption in operation why this solution would be a good compromise for warehouses where a fully automation is economically unviable or unfeasible for some reason.

1.2. Goals and challenges

One goal of this thesis is to develop a hard- and software concept for an autonomous mobile robot which is able to do the order picking of small boxlike goods, drug boxes for instance, according to the robot-to-goods method in a standard non automated warehouse. A prototype robot has to be build according to this concept using present and, wherever possible, off-the-shelf hardware. The software concept as well as the implementation have to be based on ROS (Robot Operating System) and use up-to-date software packages and tools to reduce time and effort for software development. It is also part of this thesis to prove concept, prototype design and implementation by performing experiments using the prototype which will identify possible improvements and fields for future work.

Challenges to achieve these goals:

- Manipulation of containers

- 3D object recognition
- Automated grasping of goods
- Autonomous navigation
- Electrical autonomy

All of these points are challenging for a number of reasons. The most important challenge for manipulating containers is to find a simple but still robust method to fetch and respectively place containers. For detecting the boxes in the container, the 3D object recognition part, the problem is similar. The detection method has to be simple to save computing power and time but it still has to be robust with respect to several factors like sensor noise or ambient light, to name just two. Automated grasping as well as autonomous navigation are complex as they combine the problems of path planning and execution, collision detection and avoidance and in case of autonomous navigation also localization. Due to the fact that the picking system is implemented as an autonomous mobile robot the whole system has to be electrical autonomous which means that all the energy needed to power the hardware has to be provided by batteries. Therefore all the systems have to be as energy-saving as possible to increase operating time.

1.3. Outline & contributions

The contributions of this thesis to the logistics problem of automated order picking using an autonomous mobile robot, consist in a hardware and software concept as well as in ideas for its implementation. Divided into different parts the thesis starts with an overview about software modules, basic concepts and algorithms used to solve common robotic problems appearing during this work. This overview is presented in chapter 2.

In the next chapter (chapter 3) some interesting related projects, being part of the background for this work, and also their relevance for this work are presented. The chapter addresses relations to the presented concept as well as to the presented prototype implementation.

The system description presented in chapter 4 is again divided into three sections. The first section (section 4.1) describes more detailed the challenges as well as the developed hardware and software concept for a system being able to solve the problem of order picking using a mobile robot. The hardware and software concept is described separately in the subsections 4.1.1 and 4.1.2. The other two contained sections (section 4.2 and section 4.3) go into the implementation details, hardware and software, of the developed and built prototype robot called *"Kombot"*. An intermediate state of the prototype implementation has already been described and presented in [5].

Chapter 5 is about the experiments performed with the prototype robot. These experiments have the goal to test the built prototype but also allow to evaluate the used systems and methods for their potential to solve particular subproblems within the overall logistics problem mentioned at the beginning.

The mentioned evaluation of systems and methods and the resulting conclusions and suggestions for future work, contained in chapter 6, are also a big part of the thesis' contributions.

Chapter 2.

Basics

For the sake of clarity and a better understanding this chapter shortly introduces the two most important software modules used in this thesis as well as some of their contained components with underlying concepts and algorithms. It also addresses the problem of path planning because it appears twice during this work and is one of the basic problems in robotics. The chapters section about the Robot Operating System is mainly based on [21] and the ROS WIKI documentation (`ros.org`) while the biggest part of the information about the Point Cloud Library and its components is from *Radu Bogdan Rusu*'s dissertation [22] and the projects website (`http://pointclouds.org`). The section about path planning is based on the book *"Principles of Robot Motion"* [11] and the article *"The Open Motion Planning Library"* [4].

2.1. Robot Operating System (ROS)

ROS (Robot Operating System) is a meta-operating system for robotics from *"Willow Garage"* (`http://www.willowgarage.com`) designed for use on Unix-based platforms. The primary goal of ROS is to provide a robotics framework making reuse of code easier for researchers and developers. Therefore ROS provides tools and libraries for developing programs and also to run them distributed on multiple computers. These tools and libraries are usable with different programming languages, especially C++ and Python, which makes developing of robotic software much easier. One of the main concepts of ROS is to divide the whole problem of controlling a robot into smaller problems which are more easy to solve. These smaller problems can be solved in separate processes (executables) called *nodes* which can be distributed over different computers connected via a network in a peer-to-peer topology.

Another important feature of ROS is the integrated package management allowing to group source code and binaries of related *nodes* into packages. In turn related packages can then be grouped into so-called *stacks* containing packages with nodes required to solve a particular problem. The ROS *navigation stack* for instance, contains packages and nodes to solve the problem of 2D navigation for a mobile robot.

For the communication between the distributed nodes the framework offers different communication options. The first one is a centralized data storage and sharing system called *parameter server*. The data is globally viewable and editable by all nodes and usually only used to store configuration parameters because the system is not designed for high performance. The second communication option is an asynchronous streaming of data over so-called *topics* in either common *messages* predefined by ROS or another package or in a complete new defined message type. Defining of new messages is very easy possible via simple structured text files and an integrated tool that automatically generates the required C++ and Python code. The third communication option is a synchronous one called *ROS services* implementing a RPC (Remote Procedure Call) like communication briefly described in the following subsection 2.1.1. The following subsection also describes a library called *Robot Operating System (ROS) actionlib* offering another communication option, used several times during this thesis. For more details about the basic concept of ROS have a look at [21] and the ROS documentation WIKI at `http://www.ros.org/wiki/ROS/Overview` (2012). The WIKI also provides more or less valuable documentation for common packages and stacks.

2.1.1. The Robot Operating System actionlib

For a correct behaviour of the robot some tasks have to be executed continuously such as low level communication with the hardware. Due to limited computing power on autonomous systems this does not make sense for all tasks. One solution for this problem are so-called *ROS services*. One task, called *service client*, can send a *request* to a ROS service provided by another node called *service server*. After processing the request the *service server* node will send a *response* message containing the solution and possibly additional information to the requesting client. In case of ROS services, processing of the request can not be interrupted by the requesting task.

For interruptible tasks *ROS actions* are a possible solution. Similar to ROS services, ROS actions have a server-client-architecture but a more complex communication between server and client node (see Figure 2.1) to keep the client informed about the actual state of running process. This is achieved via so-called *feedback messages* which can be sent from the server to the client to keep it informed. In contrast to *ROS services*, in case of *ROS actions* the request is called *goal* and the response is called *result*. The *actionlib package* contains tools for automatic message generation as well as the C++ and Python versions of the library containing implementations for action client and action server.

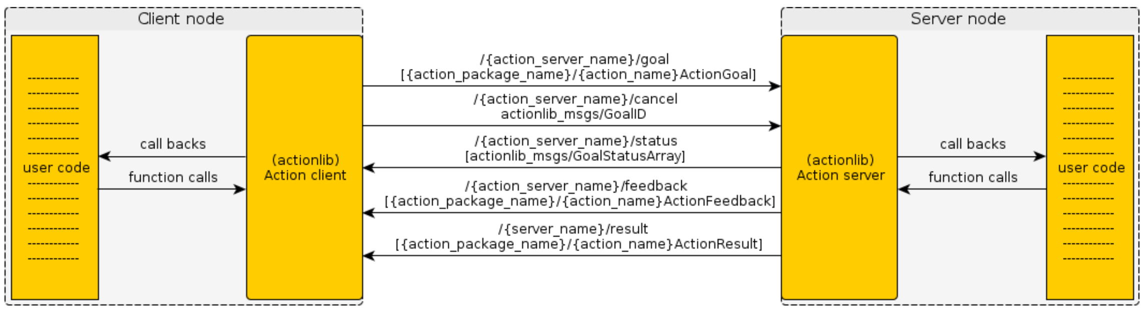

Client node
user code
call backs
function calls
(actionlib)
Action client
/{action_server_name}/goal
[{action_package_name}/{action_name}ActionGoal]
/{action_server_name}/cancel
actionlib_msgs/GoalID
/{action_server_name}/status
[actionlib_msgs/GoalStatusArray]
/{action_server_name}/feedback
[{action_package_name}/{action_name}ActionFeedback]
/{server_name}/result
[{action_package_name}/{action_name}ActionResult]
Server node
(actionlib)
Action server
call backs
function calls
user code

Figure 2.1.: ROS actions client-server structure

The data structure for information passed between client and server task is specified via an *action definition file* (`.action`). This file combines the definition of the three message types (goal, feedback and result) which are generated automatically while the package, containing the action definition file, is built. The first of these three messages is the *goal message*. It contains information necessary for the action server to complete its task.

In case of a 2D navigation goal this would be the robot's goal pose, for instance. The second message defined in the action definition file, is the *feedback message*. This message enables the server task to inform the client task about the actual working progress but is optional and not implemented in all action server nodes. Usually this message type is sent periodically until the process has finished. In case of a 2D navigation goal a good example for a feedback message could be the robot's actual pose. The third message is the *result message* which is sent from the server to the client task when the task has finished. This message can contain possible outcomes of the servers task. In case of an error during processing it could for instance contain additional information about the problem causing the error. In contrast to the feedback message the goal and result messages are non-optional but can be empty.

Due to the fact that more than one action client can connect to a single action server and action goals can be sent asynchronously, the action server follows a strict policy for handling received goals. Only one goal can be processed (active) while a second one is pending. If a third goal is received the pending goal is cancelled. The action server holds a state machine for each goal to track its state until the goal has been finished. The action client also holds a state machine for the goal that was sent to the action server and receives status updates at a rate of about 10 Hz over the *status topic* to keep the two state machines synchronised. To identify the messages corresponding to a specific goal the original goal, feedback and result message are wrapped into a new message before publishing it. The wrapping message contains the original message and adds a unique *goal ID* and a time stamp which is generated from the service client when the goal is sent. All following messages corresponding to this goal contain these additional information. As long as a goal is pending or active the action client can cancel this goal via publishing a message with the goal ID on the *cancel topic*.

The interaction between user code and the library uses function calls and callbacks. Callbacks can be registered to be automatically informed about new goals, goal status changes and so on. Another way to get informed about this things is to poll them via function calls which is not really convenient but can have advantages in some special cases. Usually function calls are used to trigger actions like sending or cancelling a goal (client side), sending a feedback or a result message (server side).

2.2. Point Cloud Library (PCL)

This section will give a short overview of some of the Point Cloud Library's (PCL) components and their underlying methods and algorithms. The PCL is the most important software component for processing and analysing point clouds during this thesis.

2.2.1. Point clouds

In general the term *point cloud* only means a collection P of m points p which have a set of n common properties, also often called features f or more formal:

$$P = \{p_1, p_2, \dots, p_{m-1}, p_m\} \quad \text{with} \quad p_i = \{f_1, f_2, \dots, f_{n-1}, f_n\}$$

Nowadays, especially in the field of robotics, this term is usually used for collections of 3D points which represent a part of the real world. In this case the properties are just the coordinates of the points with the origin of the coordinate frame at the sensing device. So the formal notation changes to:

$$P = \{p_1, p_2, \dots, p_{m-1}, p_m\} \quad \text{with} \quad p_i = \{x_i, y_i, z_i\}$$

Depending on the acquisition technique (stereo cameras, time-of-flight cameras, laser measurement units,...) the points can have additional properties like the distance d from the sensing device, RGB-values, surface remission values r and so on, which leads to the following formal notation:

$$P = \{p_1, p_2, \dots, p_{m-1}, p_m\} \quad \text{with} \quad p_i = \{x_i, y_i, z_i, d_i, R_i, G_i, B_i, r_i, \dots\}$$

Figure 2.2 shows an example of a real scene and its representing point cloud coloured according to different properties. Processing of the point cloud can also produce new properties. So this changes the concept of 3D points to a nD point concept and leads back to the notation at the beginning. Besides the simple way of storing points in a list, different methods of organising the points in tree structures (octree, bd-tree) exist which provide fast access to point locations and their neighbours. (cf. [22, page 17 ff])

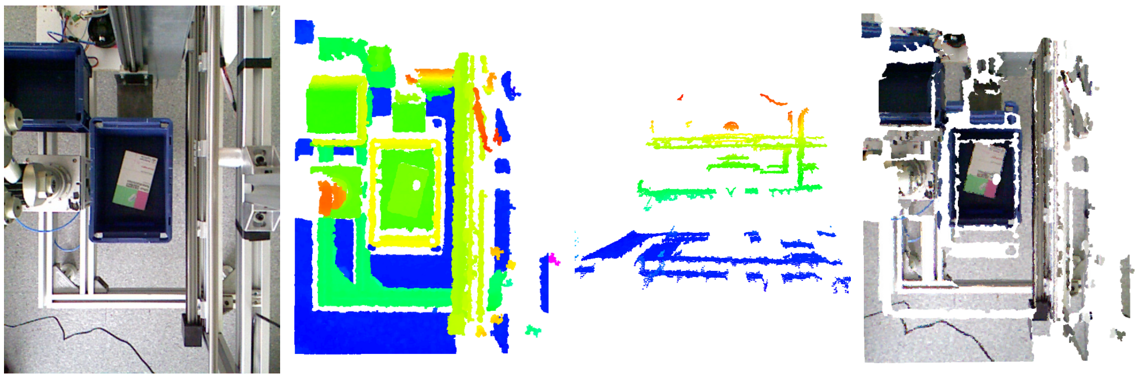


Figure 2.2.: *Left:* Image of a real world scene; *Center:* Point cloud top and side view, colour shows distance on z-axis (red close, blue far away), *Right:* Point cloud top view, coloured with RGB values from the cloud

In the context of the Point Cloud Library (PCL) the term *point cloud* means a special data structure to store these points with their coordinates as well as their additional properties. Therefore the PCL defines a point cloud template class with the point type as template parameter containing a simple vector of points, a header (containing frame, time stamp and a sequence number) and some other information about the point cloud. The PCL also defines different point types with different additional properties. The list below contains some of them.

- PointXYZ, normal 3D point without addiotional properties
- PointXYZI, 3D point plus intensity value
- PointXYZINormal, PointXYZI plus coordinates of a surface normal
- PointXYZL, 3D point plus a class label
- PointXYZRGB, PointXYZHSV, 3D point plus colour information

- and some more combinations of the additional properties

2.2.2. Plane segmentation

This thesis focuses on picking of small boxlike goods. One of the basic steps of the implemented approach for detecting these objects is the plane segmentation, used to find surfaces of these boxes inside a point cloud. This subsection will only give a short introduction to the corresponding PCL module used during this work. For further information on the concept of this module have a look at [22, "Fitting Simplified Geometric Models", page 86 ff].

The PCL module used during this work, providing the feature mentioned above, is called *SACSegmentation*. The output of the module are the model coefficients, defining the fitted plane, and a set of indices indicating points within an user defined maximum distance to the plane. Such points are called *inliers*. The model coefficients a, b, c, d are the coefficients of the plane equation:

$$a \cdot x + b \cdot y + c \cdot z + d = 0$$

In addition to the maximum plane distance, an axis can be defined, forcing the module to search for a plane perpendicular to this axis. It is also possible to define a maximum allowed difference angle between the given axis and the plane normal. Given this limiting parameters, the module tries to fit the plane into the point cloud in such a manner that the number of inliers is maximized. The left part of Figure 2.3 shows the point cloud of a scene with an object on a table. Plane segmentation was used to extract the points corresponding to the table and the floor and visualize them in different colours.

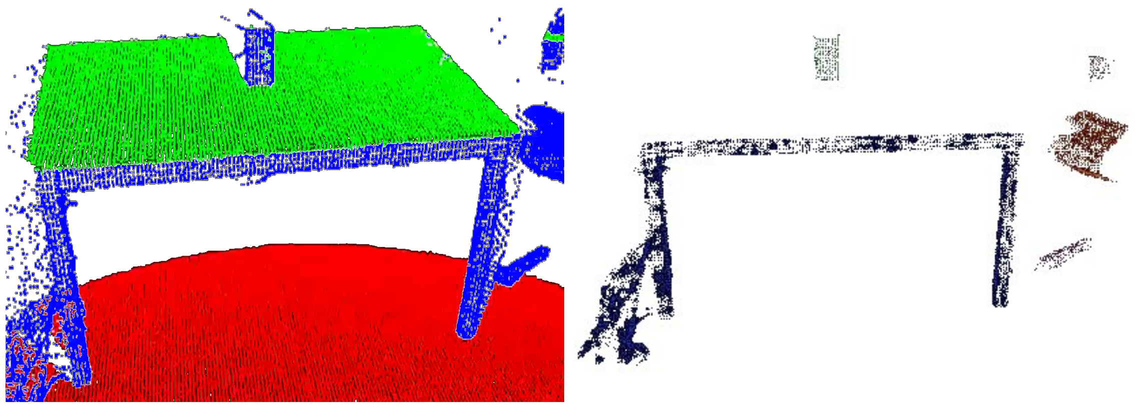


Figure 2.3.: *Left:* Point cloud with table (green) and floor (red) marked using plane segmentation (screen shot from video "Extracting sets of indices", `http://youtu.be/ZTK7NR1Xx4c`, 21.09.2012); *Right:* Same point cloud after extracting table and floor and clustering the remaining points (`http://pointclouds.org/documentation/tutorials/cluster_extraction.php#cluster-extraction`, 21.09.2012); cf. [22]

2.2.3. Euclidean clustering

One important step in detecting objects in a point cloud is to decide which points belong to an object and which do not. Assuming that the objects are spatially separated, a simple method to do this using the PCL is euclidean clustering provided via the *EuclideanClusterExtraction module*. The basic idea is that two points p_i and p_j belong to the same cluster if the euclidean distance is less than an user defined threshold ($||p_i - p_j||_2 < d_{th}$). The module uses a special tree structure (kd-tree) for the point cloud which provides fast queries for either the *k nearest neighbours* or a *radius search* for the neighbours within a defined radius around a search point. Using the radius search, a cluster can be found very easy by executing the following steps which are similar to a flood fill algorithm.

```
1  Create empty cluster list $C$
2  Create queue list $Q$ for points to be checked
3  forall $p_i \in P$ do
4      repeat
5          add $p_i$ to the current queue $Q$
6          forall $p_i \in Q$ do
7              get neighbour set $P_i^k$ of $p_i$ within radius $r < d_{th}$
8              add all neighbours $p_i^k \in P_i^k$ but $p_i^k \notin Q$ to $Q$
9              add $Q$ to the list of clusters $C$
10             reset $Q$
11         end
12     until all $p_i \in P$ are also $\in C$
13 end
```

Algorithm 1: PCL euclidean clustering algorithm (cf. [22])

For more information about the ideas behind this module have a look at [22, "Basic Clustering Techniques", page 88 ff]. Figure 2.3 shows the point cloud of a scene with an object lying on a table. After removing the points corresponding to the tabletop and the floor via plane segmentation, euclidean clustering was used to cluster the remaining points. Different clusters are shown in different colours.

2.2.4. Iterative Closest Point (ICP) algorithm

The Iterative Closest Point (ICP) algorithm is used for geometric alignment or registration of 2D or 3D shapes. In the field of robotics it is often used for registering laser scans or other point clouds. The problem is to find the transformation, rotation (R) and translation (t), for the best alignment between the two sets of points which are usually not fully congruent and only partially overlapping. In the context of the *PCL IterativeClosestPoint module* this transformation rotates and translates the so-called *input point cloud*, sometimes also called *template*, into the pose of best alignment with the so called *target point cloud*. Therefore the algorithm consists of two main steps which are iterated until an error metric reaches its minimum. Starting with an initial guess or user input for the aligning transformation, the first iteration step is to find the corresponding (closest) points (p_i from the input cloud corresponds to q_i in the target cloud, N is the number of corresponding points) of the input and the target cloud using the nearest neighbour search via a kd-tree (already mentioned in subsection 2.2.3). The second step is the error minimization which in case of the PCL module uses Singular Value Decomposition (SVD) to find the transformation parameters minimizing the least square error function.

$$Error = \sum_{i=1}^{N} ||p_j - (Rp_i + t)||^2$$

The biggest problem with the Iterative Closest Point (ICP) algorithm is the fact that it only converges to a local minimum. Therefore a good initial transformation is needed for a successful alignment. If there is no good initial guess a common approach is to repeat the algorithm with different initial transformations and take the alignment with the lowest resulting error as a solution. Another problem, especially when trying to align high resolution point clouds, is the runtime. Therefore it is recommended to use input filters to reduce the number of points. (cf. [19, page 2 f])

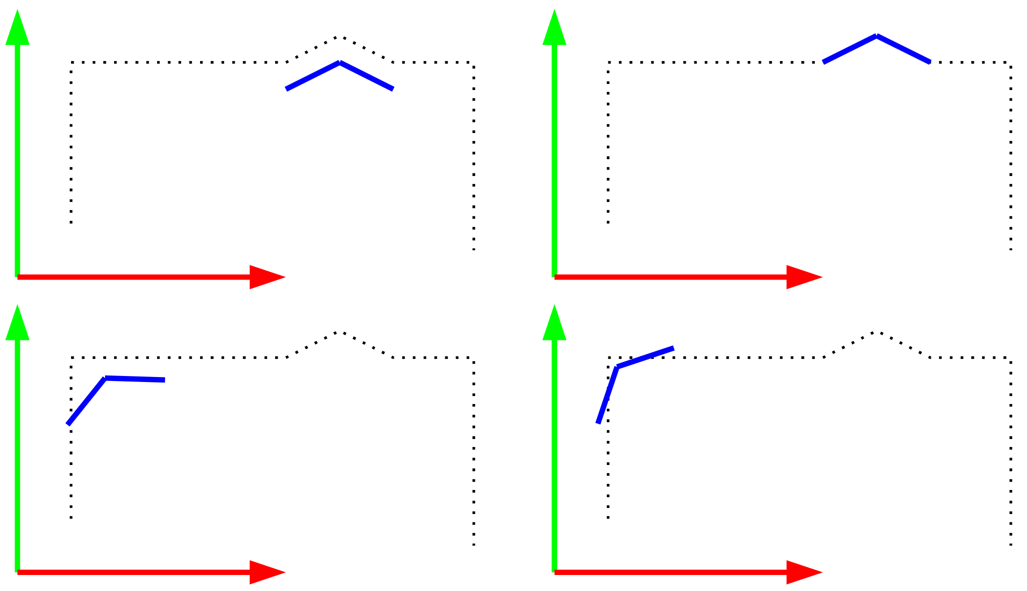


Figure 2.4.: This figure shows aligning of a template to a target point cloud via ICP *Left:* Input point cloud (black dots) and two initial poses for the template (blue wedge) to align via ICP; *Right, top:* Properly aligned; global error minimum; *Right, bottom:* Misaligned; local error minimum

Figure 2.4 shows a schematic example of such a point cloud alignment via ICP. The top left image shows an input point cloud (black dots) and the template (blue wedge) at a good initial position. The bottom left image shows the same point cloud but with a bad initial position for the template. Now ICP is used to find the transformation T for the best alignment, minimizing the error function mentioned above. The resulting alignment shown in the top right image is perfect. This means that the global minimum of the error function is found. In case of a bad initial position, the above mentioned problems with ICP approach lead to a misalignment. This means that the algorithm only converges to a local minimum of the error function.

2.3. Path planning

The problem of path planning appears twice during this work. Therefore a short overview of the problem will be given at this point. This section is mainly base on [11] and [4]. Given a known environment with some obstacles, the task of path planning, whether for a mobile robot (2D), a robot arm (3D) or whatever, is to find an obstacle avoiding path from a start to a goal configuration. For a better understanding of the problem some related terms are defined in the following.

"The *configuration* of a robot system is a complete specification of the position of every point of that system." [11, page 40] A really simple planar scenario assumes a circular mobile robot (radius r) which can only perform translational movements (no rotation). Its position relative to a world coordinate frame is fully specified by the location of its center (x, y). Knowing its radius all the points occupied by the robot can be found easily. For a more complex system, for instance a robotic arm, with a number of N joints a single configuration q is defined by its "N" joint parameters. The most common joints are revolute ones with the joint angle α as parameter. This leads to the following notation for a configuration:

$$q = \{\alpha_1, \alpha_2, \dots, \alpha_{N-1}, \alpha_N\}$$

The definition of a configuration leads to the definition of the *configuration space (C-space)*. The sum of all possible configurations forms the configuration space Q. In case of a system with continuous (stepless) joints the number of possible configurations is in fact infinite. The dimension of this abstract space is the same as the Degrees Of Freedome (DOF) of the robot and each point in the C-space represents a single configuration specified by its "N" parameters. This means that a robot with six degrees of freedom has a six-dimensional C-space.

The configuration space Q must not be mistaken with the *workspace* W of the robot which can be defined as the sum of real world points reachable by the end effector. This is one definition for the workspace but other definitions also exist. For instance the following definition which is more precise: The workspace is the set of end effector poses (position and orientation) in the real world for which at least one valid configuration exists. Using the former definition the C-space is more complex than the workspace. The workspace is usually 2D for mobile robots and 3D for robot arms while the C-space has a higher dimension because most of the points in the workspace are reachable via more than one configuration each represented by a different point in the C-space.

In this connection also appears the term *configuration space obstacle* QO_i which means a point set in the C-space, a set of configurations, at which the robot intersects a real world obstacle WO_i. Unreachable areas (e.g. outside the workspace) and configurations, forbidden due to motion constraints, also appear as C-space obstacles. In the following notation $R(q)$ defines the set of points in the real world occupied by the robot while having the configuration q.

$$QO_I = \{q \in Q | \, (R(q) \bigcap WO_i) \neq \emptyset\}$$

So the *free configuration space* Q_{free} is the part of the C-space Q with no obstacle intersecting configurations.

$$Q_{free} = Q \setminus (\bigcup_i QO_i)$$

A *path* is now a continuous curve in the C-space Q, means an infinite sequence of configurations, connecting the points representing the start and goal configuration. A given path is collision free if it is element of the free C-space Q_{free}.

Especially for systems with many degrees of freedom the C-space with all its C-space obstacles is very complex. Therefore it is not possible to use the entire continuous C-space for finding a path, so sampling-based planning is the solution of choice. Most of the sampling-based planners are based on either the concept of *Probabilistic Roadmap (PRM)* or *tree-based planners* which are briefly described below.

2.3.1. Probabilistic Roadmap planning

The PRM algorithm uses a roadmap like graph of the free C-space to find collision free paths. To generate this graph it starts with the search for a defined number of collision free sample configurations which are the vertices of the graph. Therefore random samples are taken from the C-space and checked for collision until enough samples in the free C-space are found. A local planner tries to connect each configuration to the k nearest configurations by interpolating short paths between them which are checked for collisions with a defined resolution. If the interpolated path is collision free an edge is inserted into the graph. It is assumed that the interpolated path is collision free if none of the interpolated configurations is in collision. A plan from any start to any goal configuration can now be found by connecting them to the roadmap via the local planner and a simple graph search to find the shortest path from the start to the goal. The same roadmap graph can be used for multiple plans with different start and goal configurations as long as the environment (obstacles) has not changed. Figure 2.5 shows an example how the PRM algorithm works in an simple 2D case.

2.3.2. Tree-based planning

Another strategy for sample-based planning is to use a tree structure instead of roadmap graphs. Tree-based planners use the start configuration as a root for a tree. A special expansion heuristic is used to find new valid configuration samples and connect them to the tree via short collision free paths similar to the PRM. It is also checked if the goal configuration can be reached from the new sample which terminates the tree expansion. Figure 2.6 shows the same example as for the PRM but with a tree-based strategy used to find a valid path.

There are a lot of tree-based planners differing in the expansion heuristic which usually gives the algorithm its name. In contrast to PRM, tree-based planners are often single-query planners because they use the start configuration as a root for a tree. One example are Rapidly-Exploring Random Trees (RRT) which try to explore the state space rapidly and uniformly [16]. Another approach for tree-based planners is to use a heuristic that tries to expand the tree towards the goal as fast as possible. In this case the sample configurations are heterogeneously distributed compared to the configurations in a graph coming from a PRM algorithm or a tree using the RRT approach. Furthermore a PRM graph usually contains loops while a tree does not.

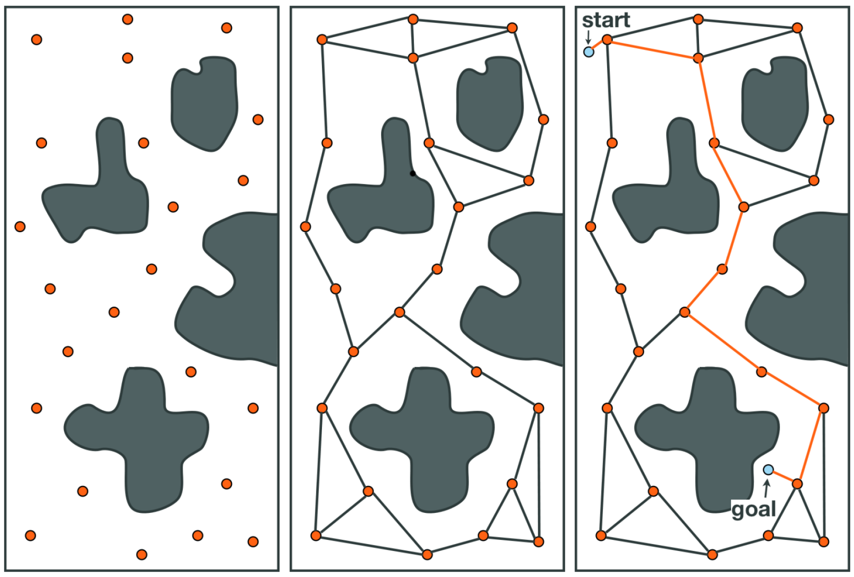


Figure 2.5.: Steps during path planning with PRM; *Left:* Uniformly distributed configuration samples in the free C-space; *Center:* Roadmap after connecting the sample configurations via the local planner; *Right:* Example path from a start to a goal configuration (cf. [4])

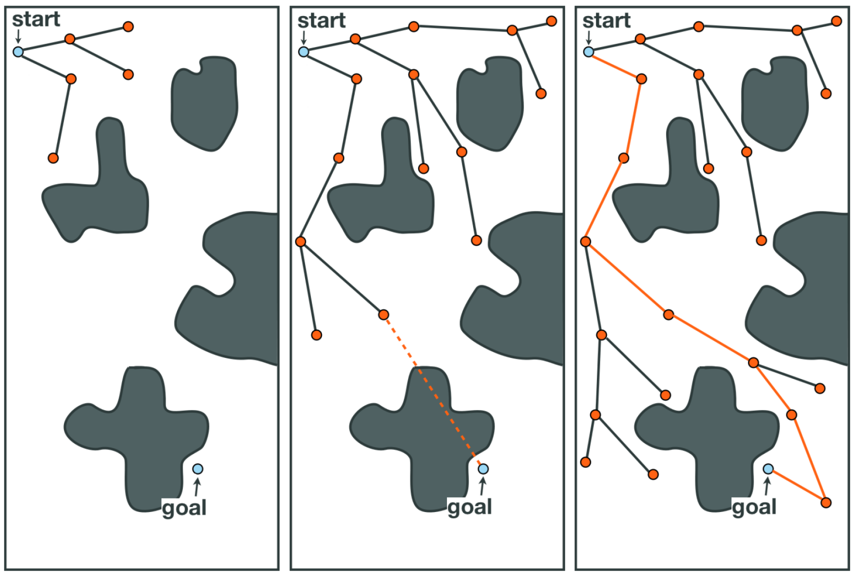


Figure 2.6.: Steps during path planning using a tree-based strategy; *Left:* Tree after connecting the first samples; *Center:* Goal can not be connected - continue tree expansion; *Right:* Goal is connected to the tree; path complete (cf. [4])

Chapter 3.

Related research

This chapter presents some other interesting projects and their relation to this work. The first section is about the *"Little Helper"* system and its underlying concept presented in [18]. This project was taken as a base to identify important capabilities and possible solutions for a mobile logistics system as developed during this work. The second sections is about two interesting mobile manipulators called *"HERB"* and *"EL-E"* while the third sections focuses on the problem of detecting boxlike objects in 3D point cloud data as used during this work.

3.1. "Little Helper" and the AIMM concept

The paper in [18] presents the Autonomous Industrial Mobile Manipulation (AIMM) concept for developing flexible robotic assistants for manufacturing processes and a robotic system called "Little Helper" implementing this concept. The aim is that systems following the AIMM concept can be built and modified in a cheap and easy way using off-the-shelf components and are able to perform various tasks in semi-structured industrial environments. These tasks consist of transportation, pick-and-place operations, classification and so on which are similar to several steps of the automated order picking process discussed in this thesis and therefore the AIMM concept and the presented prototype are taken as a base and adapted according to the special needs of automated order picking.

Under the AIMM concept, the robotic system is fully integrated into the industrial manufacturing process and can carry out different tasks at different workstations or transport objects between them. Therefore the system concept consists of four main modules as shown in Figure 3.1. The platform of the "LittleHelper" prototype is non-holonomic (Neobotix MP-L655) equipped with sensors for safe navigation in an industrial environment alongside people and a Windows computer with all the software for the system components. The environment is slightly modified by placing reflector marks and calibration targets to achieve acceptable localization tolerances especially at the workstations. The manipulator is a 6 DOF robotic arm (Adept Viper s650) equipped with a tool changer system, so depending on the task the end effector can be changed between a vacuum device and a parallel gripper. The vision system consists of a controllable light source for optimal illumination and a monochrome high resolution fire wire camera with adjustable iris and focus to recognize calibration targets or identification patterns on graspable objects. The software has a three layer architecture abstracting the hardware via services provided to a Graphical User Interface (GUI). The GUI provides all functions needed for programming tasks and controlling the "Little Helper" using a specially language developed for AIMM.

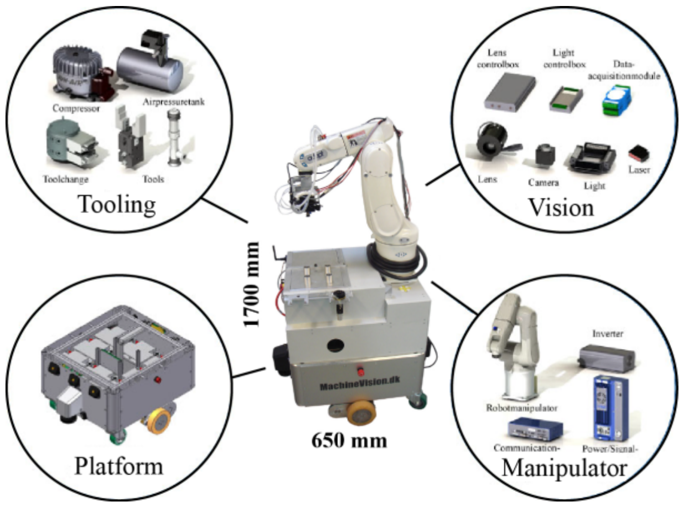


Figure 3.1.: "Little Helper", Aalborg University; prototype implementing the AIMM concept consisting of the four main modules: Platform, Manipulator, Vision and Tooling" (source [18, page2])

3.2. Mobile manipulator systems

Two interesting related research projects with mobile manipulators have been found and will be discussed in brief. They are of special interest for this project as they are using ROS as basic framework which is also intended to be the base for the system developed during this work.

3.2.1. "HERB"

In [28] a mobile manipulator, called "HERB" (see Figure 3.2), is presented which is able to perform manipulation tasks in the home. The project was from interest for this thesis because it uses ROS for controlling the robot and it includes autonomous navigation as well as object recognition and manipulation but has the problematical basic assumption of unlimited computing power which reduces the value for real applications.

The project uses the ROS navigation stack for navigation and localization while moving in the environment. For a more accurate localization while manipulating objects a vision based checkerboard localization is used. This localization method has a positioning error of only $5\,mm$ but takes often 10-30 seconds for a pose estimate and also has a lot of disadvantages in the setup phase. Therefore checkerboard localization it is not suitable for this thesis.

Object recognition via a standard extrinsically calibrated camera uses a training stage to generate 3D models with local feature descriptors extracted from the object. The 3D model is generated from several images using structure from motion in combination with SIFT features. For the recognition and pose estimation a single image is used to extract SIFT features and match them with the known models. This algorithm only works well on textured objects containing enough SIFT features. Due to the fact that each new object type has to be learned separately and the algorithm requires textured objects it also not

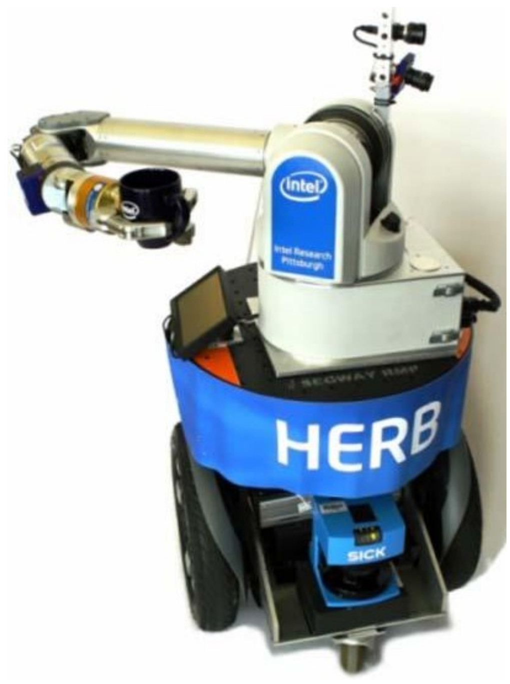


Figure 3.2.: HERB: a platform for personal robotics developed jointly by Intel Research Pittsburgh and Carnegie Mellon University [28]

suitable for this thesis.

The "HERB" project also includes a 6 DOF robotic arm with a very flexible 3 finger gripper for manipulating objects in the environment. The presented framework is not taken into account during this thesis as it seems to be complicated to implement and there is no evidence that it would work with a 5 DOF arm as it is used in this thesis.

3.2.2. "EL-E"

The project presented in [12] is about a mobile manipulator called "EL-E" (see Figure 3.3), developed to assist people with motor impairments because they usually have problems to retrieve objects from different heights. Especially objects lying on the floor or higher shelves are very difficult to fetch. Therefore EL-E is able to pick up from or place objects on flat (horizontal) surfaces at a place highlighted with a laser pointer.

To do this the robot consists of several parts as shown in Figure 3.3. The first part is a differential drive mobile base which carries all the other parts. A tilting laser scanner, used for navigation as well as for object and surface detection during grasping tasks, and a 5 DOF robotic arm with a two finger gripper are mounted on a vertical linear actuator to enable the robot to perform on different height levels. To detect collisions between the arm and the environment the whole arm rests on a force sensor plate. The gripper fingers contain force sensors to detect a successful grasp and contacts with the surface. An omnidirectional camera system with a color filter in combination with a pan- and tiltable stereo camera is mounted on top of the robot. This vision system is used to detect and

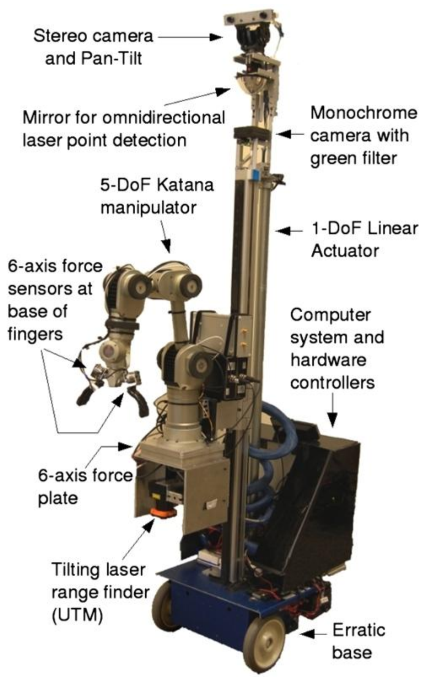

Stereo camera and Pan-Tilt
Mirror for omnidirectional laser point detection
Monochrome camera with green filter
5-DoF Katana manipulator
1-DoF Linear Actuator
6-axis force sensors at base of fingers
Computer system and hardware controllers
6-axis force plate
Tilting laser range finder (UTM)
Erratic base

Figure 3.3.: The mobile manipulator, EL-E [12]

localize laser points, indicating regions of interest.

After a point of interest was selected via the laser pointer and the robot has reached a position near to this point the object detection uses a 3D point cloud from the tilting laser scanner representing the area around the point of interest as an input. The object detection starts by detecting and extracting the surface points and all points beneath. Therefore the z-axis is divided into levels and the level containing the most points is taken as the surface level. After removing the surface points and all points beneath, the point cloud is converted into an occupancy grid and clustered. It is assumed that each cluster represents an object. Now the robot tries to grasp the object near to the point of interest. To decide how to grasp the object a 2D projection of the objects point cloud is used. In the normal case, means the object is small enough for the gripper, the object will be grasped with the gripper above the centroid and the gripper fingers oriented perpendicular to the direction with maximum dimensions in the 2D projection. In a case with an object to big for a standard grasp a special method is used to find a higher grasping point near to the manipulator.

Grasping objects is done by performing an overhead grasp which means that the gripper is oriented downwards, placed above the object and then lowered until the force sensors indicate a contact. Then the gripper is closed and lifted. The grasp is assumed to be successful if the gripper is lifted up by defined distance and both fingers measure a contact pressure above a defined threshold.

The object detection method is not suitable for this thesis because there is usually no detectable surface when detecting objects in a logistics container. Furthermore usually the objects in the container will be close to each other so a simple clustering will not work. The described overhead grasp seams to be a good method to grasp objects lying in a container and therefore a similar approach is used during this thesis. Nevertheless grasping objects in a container with a gripper is difficult and therefore the standard gripper is replaced by a vacuum gripper which makes it much easier to grasp boxlike objects lying close to each other as it is the case in logistic containers.

3.3. Box detection in point clouds

This thesis focuses on grasping boxlike (cuboid) goods lying in a container and therefore a method to recognize these boxes is needed. As it will be mentioned later, 3D point cloud data shell be used for the detection process.

In the last years a lot of new approaches for analysing of, and object recognition in point clouds, coming from various sensing devices have been presented. A lot of these approaches deal with object recognition in table-like scenes, having objects on a flat surface. The approach presented in [24] uses structured light and a stereo camera system to get dense point clouds of the environment which are processed to find graspable objects. The paper [10] also presents an approach for segmenting graspable objects and obstacles from point clouds fetched with a Microsoft Kinect camera as it will be used during this work. Both approaches are not suitable for this work because their basic assumption, that the objects are standing on a supporting plane, does not hold for objects lying in a storage container. Other approaches, like those in [17], try to detect complex objects by segmenting geometric primitives like cylinders, cones, spheres and so on, belonging to these objects and matching them against a known CAD model. Some of these approaches such as the approach mentioned below use also planes as basic geometric primitive.

A possible approach for detecting boxlike objects is to find the bounding surfaces and one possible way to do this in sparse point clouds is presented in [15]. The presented method calculates plane normals for each point by finding two neighbour points assuming that they belong to the same surface plane. Then points with similar normals are grouped together and a least-squares approach is used to fit a plane into this point cluster. It is assumed that three planes belonging to three surfaces of the cuboid can be identified. For these three planes the lines of intersection are calculated and a first vertex is obtained. The size of the cuboid is estimated by finding those points on the detected planes that have a maximum normal distance to the already detected edges. A weak point of this approach is the assumption that two or better three surfaces of the object are fully visible in the point cloud which can not be assumed in this work. It seams also that the presented approach requires a point cloud with very low noise which is not really true for the sensor used in this work. Nevertheless this approach inspired the used approach described later in this work.

Another interesting approach for finding such surfaces in dense point clouds is presented in [22](page 90 ff, chapters 6.3, 6.4 and 6.5). This approach uses curvature analysis ([22, chapter 4.3, page 45 ff]) and a region growing like algorithm to find all points belonging to the same bounded surface. This approach can not be used in this work as the sensor's noise level is to high in relation to the size of the objects to be detected so that the curvature analysis does not work.

Chapter 4.

System description

4.1. Concept

To overcome the challenges and reach the goals presented in section 1.2 the concept presented in this section was developed. The concept is based on the "Little Helper" concept which has already been presented in section 3.1. For a better understanding the whole process of order picking following the robot-to-goods method can be split into smaller subtasks as shown in Figure 4.1. These subtasks, to be executed by the mobile picking robot, can be assigned more or less directly to the mentioned challenges.

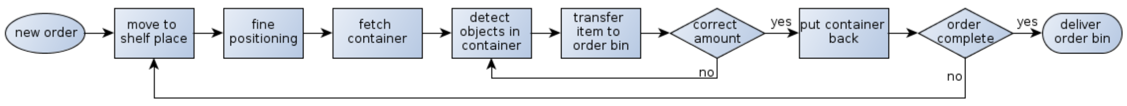


Figure 4.1.: Order picking work cycle overview

The three shown subtasks *"move to shelf place"*, *"fine positioning"* and *"deliver order bin"* for instance correspond with the challenge *"Autonomous navigation"* because the robot has to be able to navigate autonomously through the environment. The environment is assumed to be a standard non automated warehouse with slight modifications like artificial landmarks assisting the robot with precise positioning in front of the shelves or other points requiring precise positioning. A schematic layout of such a warehouse is shown in Figure 4.2.

The subtasks *"fetch container"* and *"put container back"* belong to the *"Container manipulation"* challenge. One problem in this regard is that the container manipulation system has to be as simple and cheap as possible but must also be flexible enough to deal with the positional inaccuracy of the robot relative to the shelf, the tolerances for the position of the container in the shelf and many other things.

The challenge *"3D object recognition"* is represented by the *"detect objects in container"* subtask and very important as without a detection of objects in the fetched container the following grasping step can not take place.

"Automated grasping" is connected with the *"transfer item to order bin"* subtask. The goal here is to grasp one of the items in the fetched container and put it into the order bin. This is done by a robotic arm equipped with a vacuum gripper. Important hereby is collision avoidance because the grasped goods must not be damaged or lost during transfer.

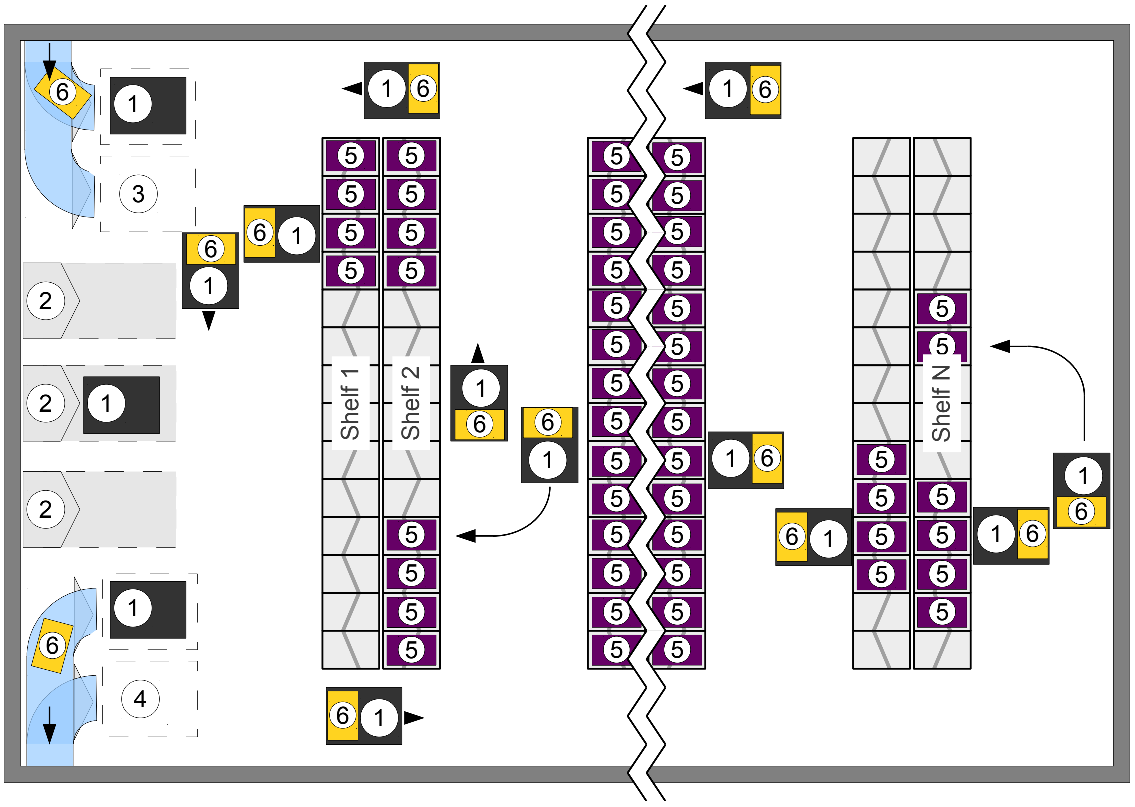


Figure 4.2.: The figure shows a schematic layout of a possible warehouse environment with picking robots (1), charging stations (2), order bin retrieval points (3), order bin drop of points (4), N shelves with containers (5) and order bins (6). Also shown the wedge like landmarks at 2,3,4 and at the shelves

As a last challenge there is *"Electrical autonomy"* which belongs to the whole system because all the needed energy has to be provided by batteries. Therefore both, hard- and software parts, must be developed with regard to a maximum of energy efficiency to meet this challenge.

The mentioned subtasks and challenges have different requirements on hard- and software and therefore the following concept description is split into separate subsections for hardware and software.

4.1.1. Hardware concept

The work cycle subtasks as shown in Figure 4.1 in combination with the goals and challenges presented in section 1.2 lead more ore less directly to the hardware modules shown in Figure 4.3. The modularization in combination with the usage of standard, off the shelf hardware allows a quick, cheap and qualitative development of mobile order picking robots. For the challenge of electrical autonomy one of the most important requirements for all hardware parts is electrical efficiency. Therefore it should be ensured that parts provide a low power stand by mode for times when they are not actively used. Robot arms for example should have joints with mechanical breaks to reduce power consumption while standing still.

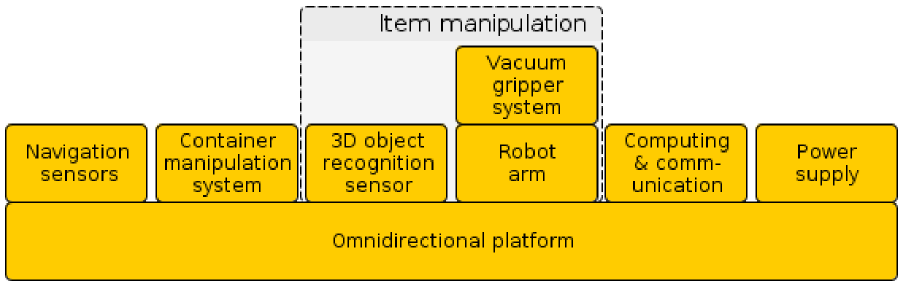


Figure 4.3.: Hardware concept overview

The mechanical and pneumatic concept was developed in cooperation with a project team from the "Institute of Logistics Engineering" at the University of Technology in Graz. During their work, documented in [3], the given problem as well as some possible solutions for the mobile base, container and item manipulation have been analysed and evaluated. Under the given conditions the reported concept was found to be the optimal trade-off between flexibility, complexity and costs.

4.1.1.1. Mobile platform

The task of navigating in a warehouse, maybe alongside human workers, requires a base with a maximum of mobility and flexibility which provides odometry information. Depending on the system used for container manipulation a more ore less precise positioning in front of the shelves is necessary but other places can also require precise positioning like automatic charging stations for instance. To meet these requirements the base module for the hardware concept is a *omnidirectional platform*. From the mechanical point of view the mobile base must have a payload sufficient for carrying all the other hardware modules as well as the order bin but has to be as small as possible because greater dimensions have a negative effect on mobility.

4.1.1.2. Navigation sensors

The *"Navigation sensors"* module contains all sensors required for autonomous navigation, including localization and collision avoidance, in the warehouse environment and landmark based fine positioning in front of the shelves. Typically these sensors are 2D or 3D lasers or sonar sensors. For a maximum of safety these sensors should cover the whole area around the robot to detect possible collisions with the environment or humans. As these sensors are possibly safety critical it is advised to use industry certified scanners. An example environment layout is shown in Figure 4.2.

4.1.1.3. Container manipulation

The *"Container manipulation system"* module consists of a forklift like manipulation unit which is capable of handling containers in standard warehouse shelves and placing it on a defined place within the workspace of the *item manipulation* system for the picking operation. Similar to a forklift the system consists of two linear actuator (horizontal and vertical) for positioning the fork under the container and lifting it up. This solution is considered to be the best trade-off between flexibility, price and complexity.

4.1.1.4. Item manipulation

The *"Item manipulation"* module consists of a 3D sensor for detecting the objects in the container. Different sensing devices are possible but this concept uses the Microsoft Kinect because it is rather cheap, off the shelf hardware and is well integrated into the used software framework (ROS). The actual item manipulation is done by a robot arm equipped with a strong vacuum gripper performing an overhead grasp as described in subsection 3.2.2. This combination allows for a good grasp even if the goods are not lying flat in the container or in case of slight touches between the grasped good and the environment during the transfer to the order bin. The vacuum system includes the gripper, the vacuum generator and a pressure sensor to detect if a grasp was successful or if an item was lost during the transfer operation.

4.1.1.5. Computing & communication

The *"Computing & communication"* module includes all the hardware necessary for running the software as well as communication modules connecting the different hardware modules to the computing devices and a wireless connection to the warehouse management system. The hardware modules offer different interfaces for communication. Common interfaces are Ethernet, CAN, USB, RS485, RS232, to name just a few. Laptops with SSD drives are a good and cheap alternative to industrial PCs as modern notebooks are electrical efficient and come up with built in WLAN hardware which can be used for a cheap and easy connection to the warehouse management system. They also provide a display and keyboard for possible error outputs, diagnostics or maintenance operations.

4.1.1.6. Power supply

The last module is the *"Power supply"* which contains battery packs providing the electrical power and converters necessary to generate the different voltage levels for the several hardware parts. Most of the industrial hardware requires 24-48V with a relative good acceptance of voltage swing while other off the shelf hardware usually requires well stabilized voltage levels of 5, 7, 9 or 12V. It is recommended to use short-circuit proof converters to avoid damages caused by hardware problems. The system should provide information about the actual battery state so that recharging can be scheduled in time. For efficient usage the system should also provide possibilities for either automatic recharging on a charging station or easy changing of the battery packs without the need to shut the robot down.

4.1.2. Software concept

The software concept for the robot is based on ROS (see also section 2.1) and other software tools and packages available for this basic framework. ROS comes up with a lot of packages and tools dealing with common robotic problems like autonomous navigation, localization and many more as well as with development tools for logging and debugging etc. The framework also provides easy and comfortable solutions for Inter-Process Communication (IPC). This easy to use communications paths allow for splitting of the whole software into an on-board and an off-board part (see Figure 4.4).

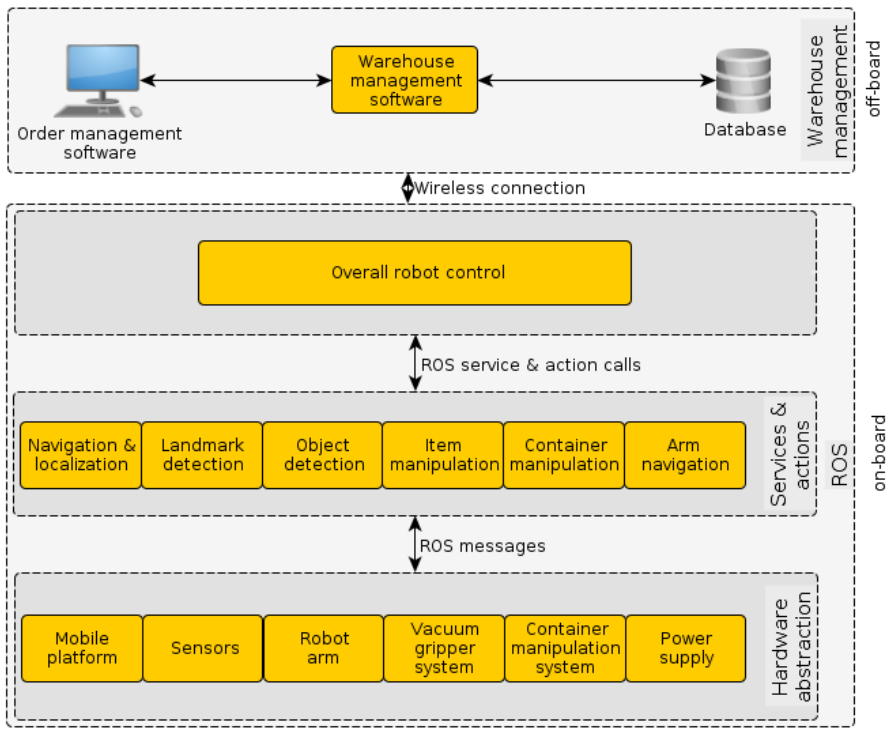


Figure 4.4.: Software concept overview

4.1.2.1. Warehouse management

The concept envisages that all the computation, necessary for performing the task of order picking, will be done on-board so the only off-board part is the warehouse management system. The main parts of this system are a database, a order management software and a warehouse management software. For managing customers orders the order management software has to provide interfaces, a Graphical User Interface (GUI) for instance, to place, modify and cancel orders, to check their state, etc. The database usually holds all the necessary information about orders, customers, inventory etc. For the picking robot additional information about map coordinates for shelf places and dimensions of the item boxes and containers will be needed. The warehouse management software is usually a very complex piece of software with a lot of functionalities. For instance it takes orders and decides when to pick, pack and ship which order, tries to find the most effective picking sequence, etc. In the actual scenario it also has to decide when, which robot should pick a certain order, when a robot should be recharged and in case of a huge robot fleet it has to optimize the picking and recharge sequences to avoid jams in front of shelves, container transfer points and charging stations. Furthermore the warehouse management software needs an interface to send orders to the robot and receive its responses.

4.1.2.2. Hardware abstraction layer

The on-board part of the software consists of three layers as shown in Figure 4.4. On the bottom is the so-called *"Hardware abstraction"* layer. It consists of ROS nodes responsible for the communication with the different hardware devices via the different communication interfaces like Ethernet, CAN, USB, etc. This layer is necessary because data coming from the hardware has to be converted from the hardware specific format into ROS messages and vice versa. If possible, especially in case of standard data like sensor readings, odometry information, etc., existing ROS standard message types should be used.

4.1.2.3. Services & actions layer

In the middle of the three layers is the *"Services & actions"* layer. It consists of modules that process and generate data to perform more complex tasks like autonomous navigation, object detection, etc. as necessary to perform the overall task of order picking. To trigger and control the tasks implemented by these modules, the modules offer ROS service and actions (see section 2.1) that can be called from either the top level robot control or another module in the *"Services & actions"* layer.

Navigation & localization
The *"Navigation & localization"* module provides the ability of map based autonomous navigation and localization. This is achieved through the ROS navigation stack. It is used because it is a widely used, state of the art software module providing all features necessary to solve this task. The stack uses the data from the navigation sensors and drive odometry to localize the robot on a map of the environment. It also generates motion commands for the drive to reach a certain point on the map as defined by the calling task. The *"Navigation & localization"* module also provides the ability of fine positioning, relative to artificial landmarks, in front of the shelves or other places requiring precise positioning like order bin drop of points for instance. Figure 4.2 show a schematic example warehouse layout. This task is assisted by the *"Landmark detection"* module.

Landmark detection
The *"Landmark detection"* module uses laser range data, coming from the main navigation sensor, to detect and track the pose of an artificial landmark. This is used to provide a highly accurate localization relative to the shelves or other important points, necessary for precise positioning at these places. In Figure 4.2 such places are the order bin retrieval an drop off points, charging stations and shelves. The base for this module is the Point Cloud Library (PCL), described in section 2.2, used for converting and analysing the data from the navigation sensor.

Container manipulation
The container manipulation module provides the functionality for fetching a container from the shelf, placing it in front of the robot arm for picking and to put the container back in

the shelf. When triggering the action the only provided information is the operation to be executed and the shelf level. All other necessary information are provided via configuration files.

Object detection

The object detection modules is also based on the functionality of the PCL. It analyses the 3D point clouds from the 3D object recognition sensor, which is a Microsoft Kinect camera, to detect the goods to be fetched lying in the container. It also identifies and reports possible grasp points for the item manipulation.

Arm navigation

The arm navigation module uses the ROS arm navigation stack in combination with other software packages. The task of the module is motion planning for the robot arm. It also uses a 3D model of the robot and data from the 3D recognition sensor for collision avoidance.

Item manipulation

The item manipulation module uses data from the object recognition, calls to the arm navigation module and commands to the vacuum gripper system to perform the task of item manipulation. It uses the robot arm to pick an item from the container and transfer it to the order bin.

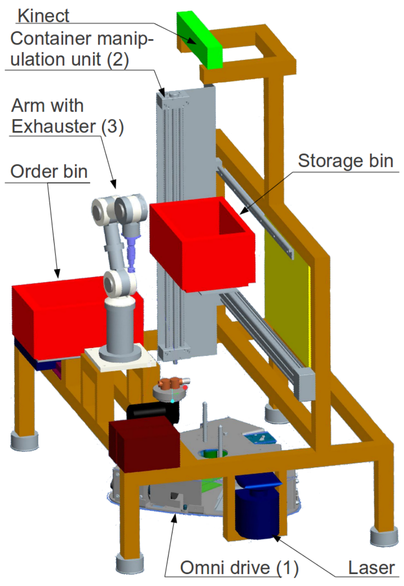
4.1.2.4. Overall robot control

The top level of the on-board software is the *Overall robot control.* It receives order information from the warehouse management system and, as illustrated in Figure 4.1, sequentially calls all the necessary services and actions, provided by the different modules in the *"Services & actions"* layer, to perform the task of order picking. In case of failure recovery actions will be triggered or human assistance will be requested if no recovery action can be performed.

4.2. Hardware implementation

To prove the concept presented in section 4.1 a prototype robot called *"Kombot"* was built. It implements the most important parts of the concept to prove it and to identify possible weaknesses as well as fields for improvements and future work. The left part of Figure 4.5 shows a schematic overview of the prototype hardware components and their planned positions on the main frame while the right part shows a real image of the actual prototype which slightly differs from the schematic. This subsection will report the hardware implementation of the prototype following the concept described in subsection 4.1.1. Wherever possible, hardware parts already present in the lab where used to keep the costs for the prototype as low as possible.

The hardware components listed below were developed and built during a cooperative project at the "Institute of Logistics Engineering" of the University of Technology in Graz documented in [3].



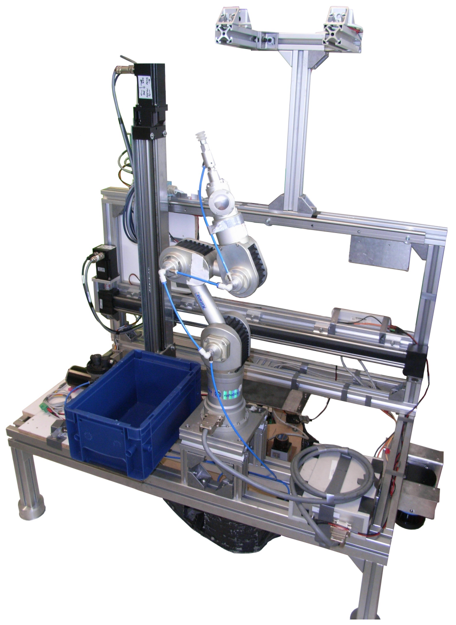


Figure 4.5.: *Left:* Prototype component overview; *Right:* Prototype photo without Kinect

- The main frame including the connection to the Krikkit drive
- The container manipulation unit including the fork and both linear axes
- The vacuum gripper system including the gripper itself, the vacuum pump, a magnetic valve, a pressure sensor and the flexible piping system for the robot arm (all parts shown in Figure 4.10)

4.2.1. Mobile base

The mobile base consists of a self supporting main frame (Figure 4.6, top left) and a drive part. The used drive (Figure 4.6, center) is the base-platform of an old RoboCup Middle Size soccer robot (”Krikkit” generation) as desribed in [29]. It provides the requested omnidirectionality, as it uses omnidirectional mecanum wheels (Figure 4.6, top right), but the small dimensions and the low possible load of the ”Krikkit” drive force the use of a self supporting frame to carry the other components. This frame bears the whole load and guarantees for a high tipping stability. To ensure a minimum of rolling friction the frame rests on four ball casters (Figure 4.6, bottom left).

A special adaptor (Figure 4.6, bottom right) is used to connect the center of the drive to the center of the main frame. The adaptor is able to compensate hight differences between frame and drive but can also transfer shearing and rotatory forces. This adaptor allows a quick change of the drive in case of failure.

The frame is build using standard aluminium profiles and brackets which is cheap and allows for quick mounting, changing and aligning of the other components. The drive uses CAN at a transfer rate of 1 $MBit/s$ to receive motion commands and to periodically send velocity information. For proper operation the drive needs a supply between 26 and 30 V

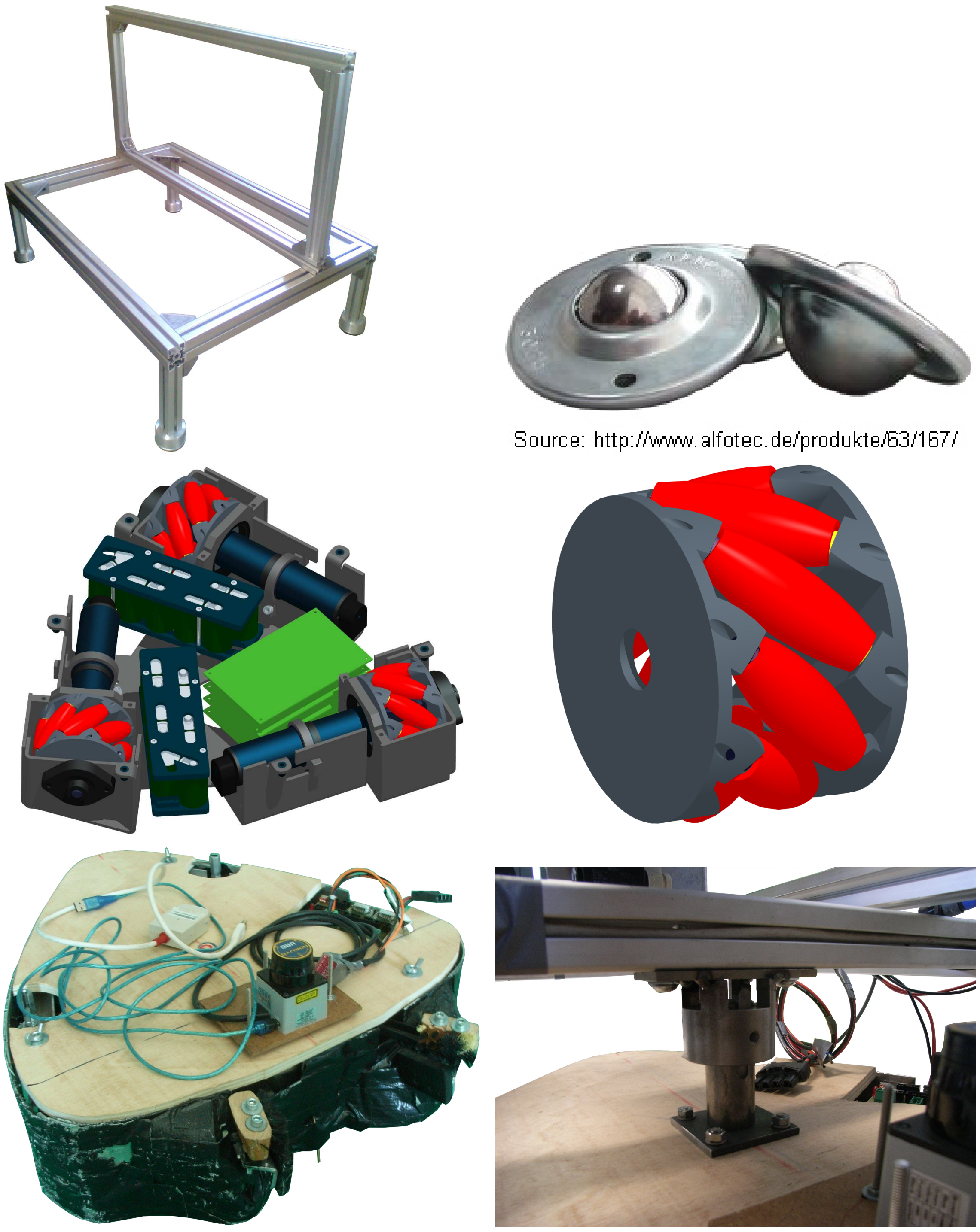
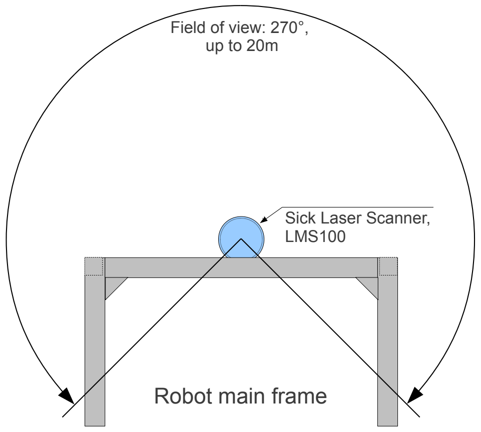


Figure 4.6.: This figure show different components of the mobile base;
1st row: Self-supporting base frame and ball casters it is resting on;
2nd row: Inner life of "Krikkit" omni drive and Mechanum omni wheel [29];
3rd row: "Krikkit" omni drive and connector between drive and base frame

which is provided by two internal battery packs.

4.2.2. Navigation sensors

The sensor for autonomous navigation and localization of the robot is a standard Sick LMS100 with a field of view of 270° and a maximum range of $20\,m$. It has an adjustable angular resolution of either 0.5° with a scanning frequency of $50\,Hz$ or 0.25° with $25\,Hz$ and a $100\,Mbit$ Ethernet interface and can be powered with a voltage between $11\,V$ and $30\,V$. This sensor is also used to find and track artificial landmarks, designed as a wedge with a defined angle of about 134° between the two front surfaces, for exact positioning in front of the shelves.



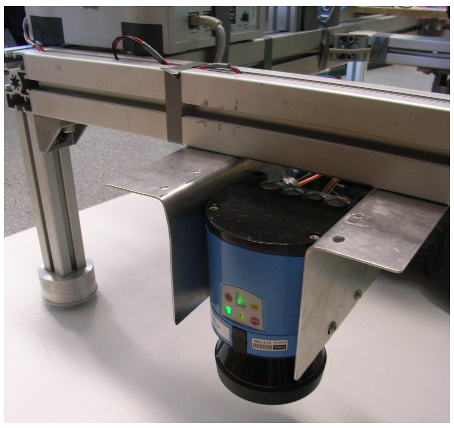


Figure 4.7.: *Left:* Navigation sensor scan area;
Right: Picture of the sensor mounted on the robot

The scanner is mounted upside down at the front of the robot as shown in the left part of Figure 4.7 with the sensing plane about $11\,cm$ above the ground so that it has a free view of 180° and is able to detect smaller obstacles. To use the whole scanning range of 270° the areas containing the feet of the frame (Figure 4.7, right) have to be filtered out so that they are not identified as obstacles. The fact that this single sensor does not cover the whole area around the robot is contrary to the concept but is sufficient for testing the main functions necessary for order picking and can be scaled up easily to cover the whole area.

4.2.3. Container manipulation

The container manipulation unit is a forklift like apparatus, shown in Figure 4.8 consisting of two standard linear actuators of the company Igus (`www.igus.de`) and a simple lifting fork. The lifting fork hast two screws at its end so that the container can be pulled out of the shelf instead of lifting it. This ensures that the container is oriented correctly which means that its broadside is exactly parallel to the horizontal axis because this is one condition for correct object recognition.

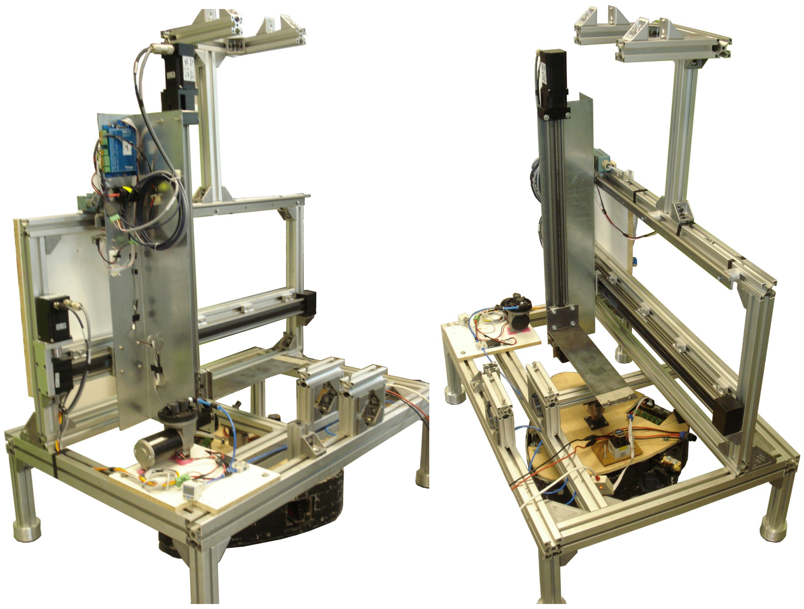


Figure 4.8.: Container manipulation unit with horizontal and vertical linear actuators and lifting fork

The vertical actuator is a spindle axis because of the higher self-locking so that no mechanical break is needed to hold a certain position while reducing the holding current. The horizontal actuator is a toothed belt axis mounted on bottom of the vertical axis and an additional sliding bar mounted on the top of the vertical axis to prevent blocking in case of heavy loads. A toothed belt axis is used for the horizontal axis because it can move faster than a spindle axis but can also be positioned very precise. For exact positioning of the axes, stepper motors in combination with encoders are used for motorization. The motors are controlled via two stepper motor control units of the type SMCI47-S-2 (Figure 4.9) from Nanotec (`www.nanotec.com`).

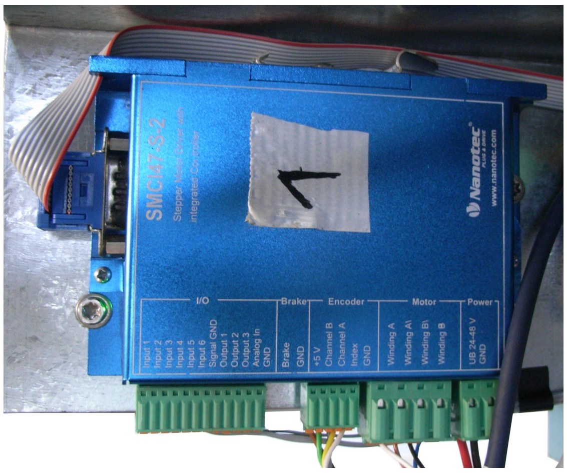


Figure 4.9.: Stepper motor control unit for linear actuators

These control units provide a RS485 interface for connection to a computer and a huge set of control commands to control the motors as well as six input ports with optocouplers and three open-drain outputs. These additional ports can be used to control relays or valves or to read in switch states or other signals. Via the encoders and an integrated position error correction the control units ensure very exact positioning also in case of fast moves and heavy loads and can also detect and report if an axis is in collision. Each axis has a reference position switch indicating the final position where the fetched container has to be placed for the picking process. More information about the container manipulation unit and the used controls can be found in [20].

4.2.4. Item manipulation

The item manipulation concept consists of three hardware modules as shown in Figure 4.3. Figure 4.10 shows the hardware parts used for these three modules.

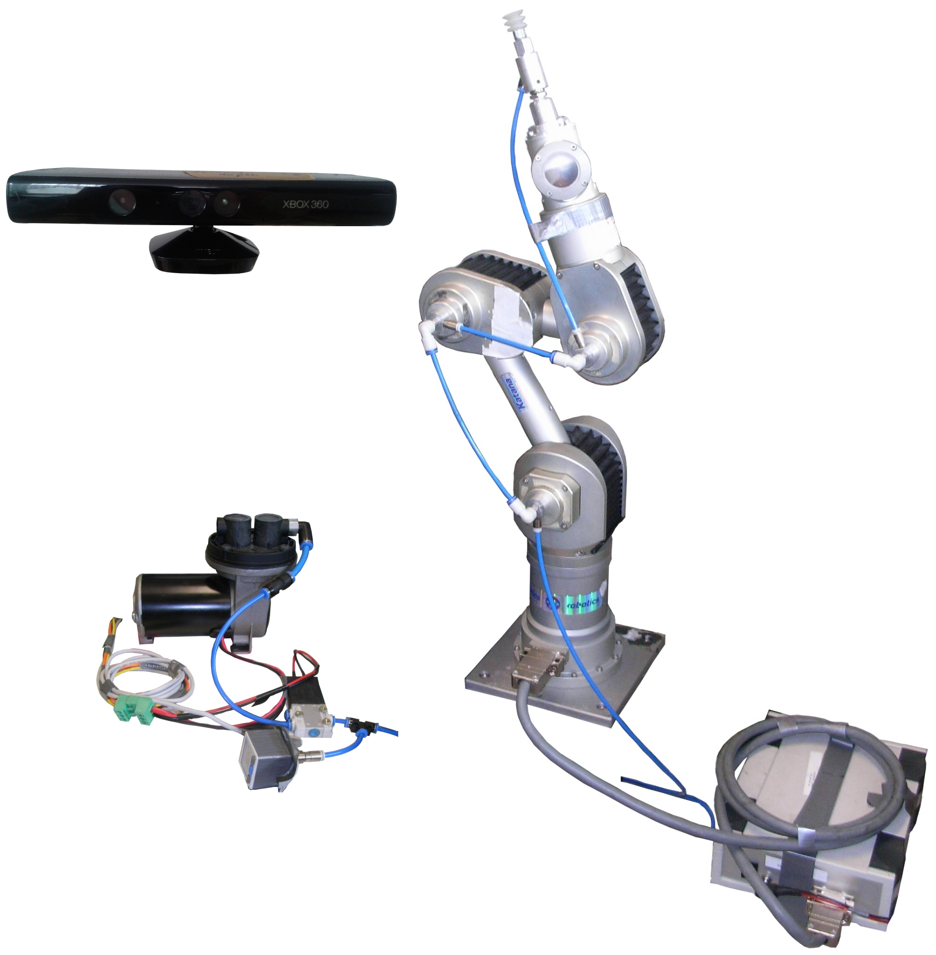


Figure 4.10.: *Left:* Microsoft Kinect, vacuum pump with valve and pressure sensor; *Right:* Robot arm with vacuum gripper and arm control box

The Microsoft Kinect camera is used as 3D object recognition sensor as already mentioned in subsection 4.1.1. It is mounted directly above the place where the container, fetched from the shelf, has to be put for the picking operation (see also Figure 4.5, left part).

The robot arm is a 5 DOF arm from *Neuronics* of the type Katana 450 6M180 (Figure 4.10, center), controlled and powered via its own control box (Figure 4.10, bottom right) which is connected to the network via a 100Mbit Ethernet interface.

The vacuum gripper system consists of the gripper itself, replacing the original two finger gripper of the Katana arm, the vacuum pump (type EVE-TR-M 2.3 24V-DC 24V-DC from Schmalz, `www.schmalz.com`), a pressure sensor (type ZSE30AF-01-B from SMC, `www.smc.at`), a valve and a relay (Figure 4.10, bottom left). The pressure sensor is connected to an input ports of the stepper motor control unit for the horizontal axis of the container manipulation system to detect if a grasp was successful, indicated by a state change of the output signal when the measured pressure (under-pressure) is less than $-50mbar$. Typically the pressure values for a successful grasp are about $-70mbar$. The relay is controlled via an output port of the same motor control unit and switches the valve and the power for the vacuum pump in a way that the vacuum pressure is released when the pump is switched of. This effects that the grasped object is dropped immediately. Figure 4.11 shows a schematic of the vacuum system.

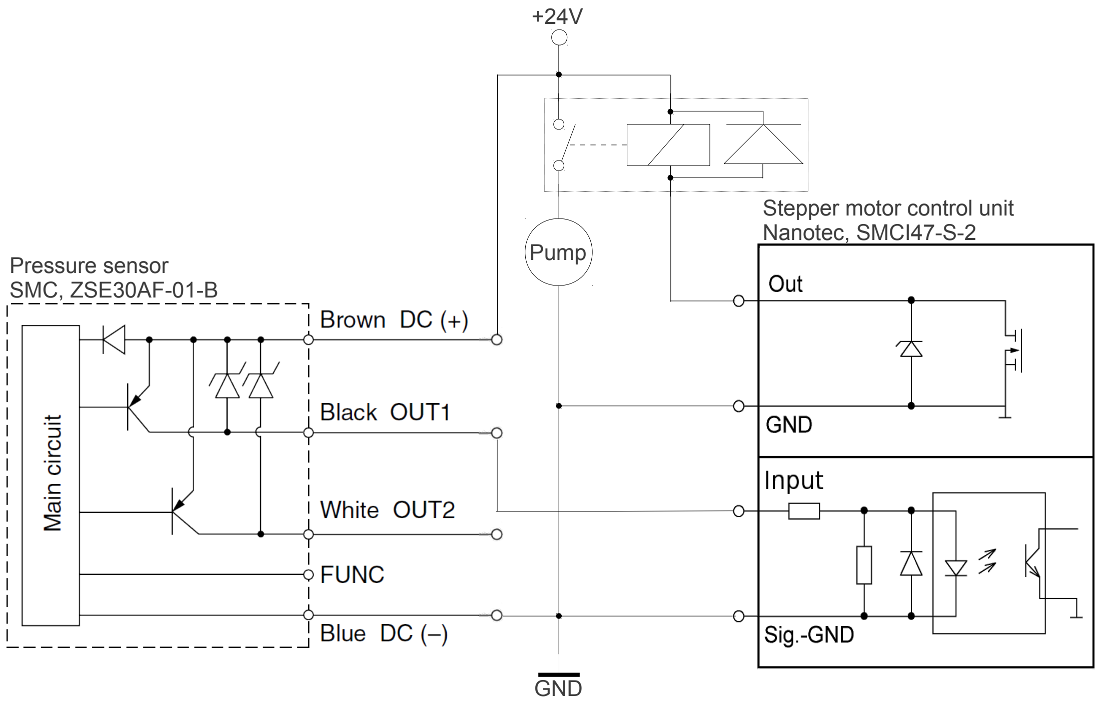


Figure 4.11.: Schematic of the vacuum system, including pump, pressure sensor, relay and control unit inputs and outputs

4.2.5. Computing & communication

Figure 4.12 shows the hardware components and the communication paths between them.

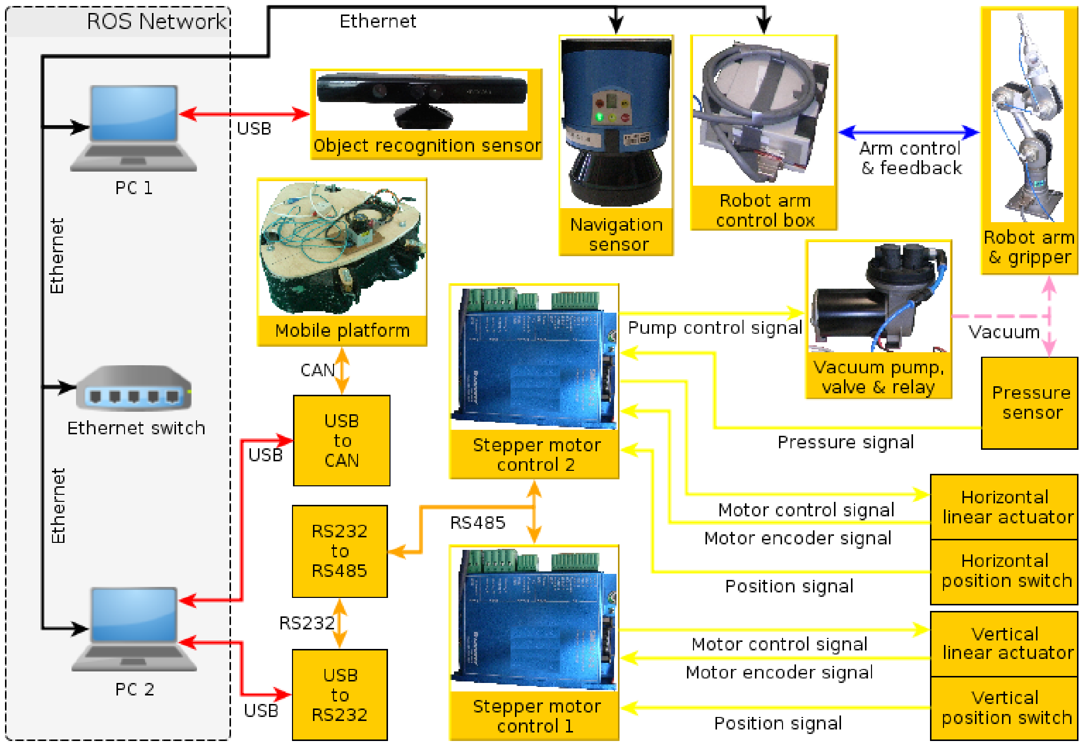


Figure 4.12.: Communication paths between computers and hardware modules

For the tests all the necessary software runs locally on two laptops connect over Ethernet. A Lenovo x60 is used to handle the communication with the navigation sensor, the mobile platform and the two stepper motor control units and is also used to run some computationally less intensive tasks. A more powerful Lenovo T420 is used as ROS master and to handle the Microsoft Kinect, the robot arm and other computationally intensive tasks. To connect the mobile platform a USB-to-CAN converter ot the type PCAN-USB from PEAK-System (`http://www.peak-system.com`) is used because of its excellent Linux support and good experiences in our lab. For the connection of the stepper motor control units a cheap USB-to-RS232 converter present in the lab and a special RS232-to-RS485 convert from Nanotec is used.

4.2.6. Power supply

While the mobile base and the laptops are powered by its own battery packs the whole electrical power for the other hardware parts is provided by two 12 V lead acid batteries and a chain of voltage converters as shown in Figure 4.13.

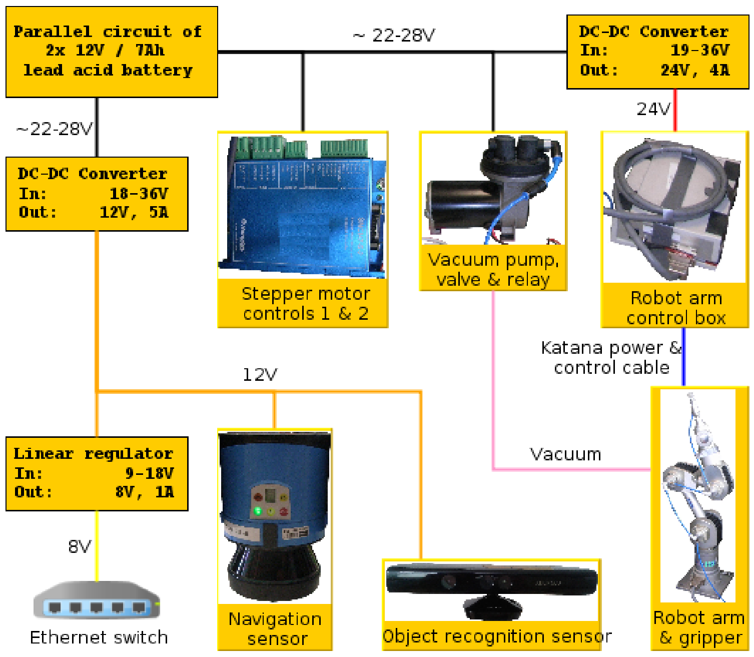

Parallel circuit of 2x 12V / 7Ah lead acid battery
~ 22-28V
DC-DC Converter In: 19-36V Out: 24V, 4A
24V
~22-28V
DC-DC Converter In: 18-36V Out: 12V, 5A
Stepper motor controls 1 & 2
Vacuum pump, valve & relay
Robot arm control box
Katana power & control cable
12V
Vacuum
Linear regulator In: 9-18V Out: 8V, 1A
8V
Ethernet switch
Navigation sensor
Object recognition sensor
Robot arm & gripper

Figure 4.13.: Power supply for hardware modules

The batteries are connected in parallel to get a combined voltage between 22 V and 28 V depending on the batteries state of charge. The stepper motor control units can be powered with a voltage between 21 V and 48 V and are therefore connected directly to the unstabilized battery voltage but have a charging condenser of 10000 μF connect near to their power inputs as it is advised in the user manual. The vacuum pump, valve, pressure sensor and the relay are also directly connected because voltage fluctuations within the expected range have no effect on their operation.

To generate the stabilized voltage of 24 V for the robot arm a DC-DC converter of type "UQQ-24/4-Q12P-C" from *Murata* (`www.murata.com`) is used. This converter has an input voltage range between 10 V and 36 V and an output power of 96 W which is required because the arm has an average power consumption of 50 W.

The Sick laser scanner and the Kinect camera are powered with 12 V generated using a DC-DC converter of the type SDS-060B12 from Sunpower (`http://www.sunpower-uk.com`). To power the network switch these 12 V in turn are converted to 8 V using a simple linear regulator.

All of the used converters are short-circuit-proof and internally protected against overheating to reduce the effects of possible short circuits or electrical overload. In contrast to the concept this power supply does not provide information about the batteries charge state to the control system. Therefore and because of missing charging connectors automatic recharging can not be implemented with this system.

4.3. Software implementation

The software for the prototype robot, called *"Kombot"*, uses the Robot Operating System (ROS) from *"Willow Garage"* as basic software framework. At project start the version *C Turtle* was chosen even though the newer version *Diamondback* has already been released but at this time it did not support the used robot arm. At a later time the version was changed to *Diamondback* as the arm drivers were updated and some important features of the PCL (Point Cloud Library) were not available in the *C Turtle* version of the library.

4.3.1. Mobile platform

In the software the mobile platform is represented via the *krikkit_driver* package which contains two nodes. One node for the connection to the *Krikkit drive* via CAN (Controller Area Network) and one for remote control of the platform. It also defines a new ROS message to send CAN messages over ROS topics (see Listing 4.1). Figure 4.14 shows a simple example using the *teleop_krikkit* node to directly control the Krikkit drive. The figure also shows all possible topic names and message types from and to the *ODO_CAN_krikkit_node* which handles the CAN connection to the drive.

```
# krikkit_driver/CAN_msg
time      time
uint32    id
uint32    msgtype
uint8     len
uint8[8]  data
```

Listing 4.1: ROS message type for CAN messages

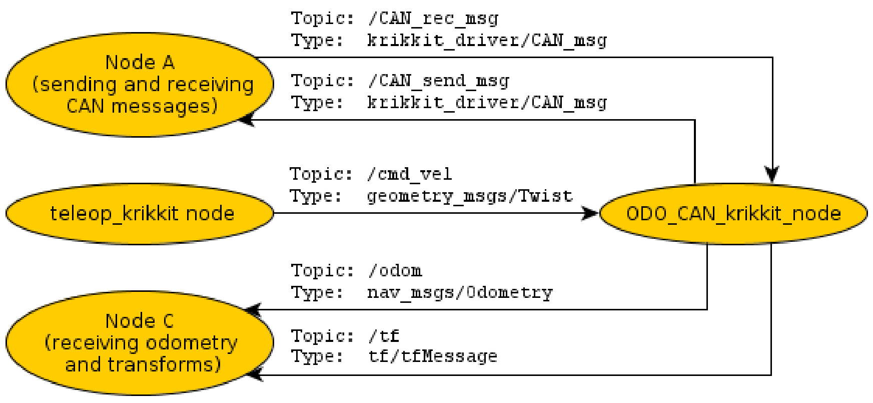


Figure 4.14.: Simple example showing message paths (topic names and message types) from and to the *ODO_CAN_krikkit_node*; the *teleop_krikkit* node is used to control the Krikkit drive

4.3.1.1. ODO_CAN_krikkit_node

This node is responsible for the hardware connecting to the drive via the CAN bus. It is the hardware abstraction for the mobile platform as shown in Figure 4.4 and has several tasks as shown below:

- Receiving motion commands over the *"/cmd_vel"* topic
- Forward the motion commands to the drive via CAN
- Receive velocity information from the drive
- Calculate and publish odometry information on *"/odom"* and *"/tf"* topic
- Forward CAN messages from the *"/CAN_send_msg"* topic over the CAN interface
- Receive messages over the interface and publish it to the *"/CAN_rec_msg"* topic

The base for this node is the PCAN Linux library from PEAK-System (`http://www.peak-system.com`) which provides a software interface for different CAN adapters. A package containing the library, drivers and documentation can be found on the "PEAK-System LINUX Website" under:

`http://www.peak-system.com/fileadmin/media/linux/index.htm`, 2013

To fulfil the tasks mentioned above the *ODO_CAN_krikkit_node* has:

- three threads (*can_read, can_send, odo_publish*) doing most of the work
- two subscribers on the topics *"/cmd_vel"* and *"/CAN_send_msg"* for receiving messages from the ROS framework
- two publishers on the topics *"/odom"* and *"/CAN_rec_msg"* for sending messages to the framework
- one transform broadcaster to send coordinate transformations over the *"/tf"* topic, to be used by the ROS internal transform management system

The node globally stores the arrival time and velocities from the last motion command as well as the latest linear and angular velocity information received from the drive. It also globally stores the latest time-stamp and calculated robot pose. These information are stored globally so that they are accessible for all threads. The pose of the robot at a time t_i is the position ($x_i, y_i \ldots coordinates$) and orientation ($\Theta_i \ldots rotationangle$) of the robot's attached coordinate frame relative to another certain coordinate frame, usually called *world* or *odometry* frame (see Figure 4.15).

$$\text{Position: } \overrightarrow{X_i} = [x_i, y_i]^T, \text{ Orientation: } \Theta_i \ldots rotationangle$$

At program start the connection to the CAN device is established and the publishers, subscribers and the threads are started. The CAN connection is initialized for a transfer rate of 1 $MBit/s$ and to use extended identifiers as it is necessary for communication with the Krikkit platform.

4.3.1.1.1. Receiving new motion commands

Whenever a message of type *"geometry_msgs/Twist"* as shown in Listing 4.2 is received on the *"/cmd_vel"* topic the subscriber invokes a callback function which extracts the needed velocity information, stores it and calls the *can_send_vel_msg* function to immediately send the new velocity command to the drive. The received message contains two vectors for

3D linear and angular velocities. Due to the fact that the Krikkit drive is a planar robot, only the linear velocities along the x- and y-axis, the angular velocity around the z-axis and the arrival time of the message are from interest.

```
# geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Listing 4.2: ROS message type for motion commands

4.3.1.1.2. Sending motion commands to the drive

The *can_send thread* periodically sends the last received motion command to the drive via the *can_send_vel_msg* function at a rate of approximately 100 Hz. This is necessary as the drive expects to receive motion commands at a higher rate as they are typically sent by navigation modules like the ROS navigation stack. If no new motion command is received the drive will stop until a new command arrives and this behaviour would lead to jerky motions.

The *can_send_vel_msg* function fetches the velocity information from the last motion command, constructs a PCAN CAN message and sends it. This PCAN message is a special data struct of type *TPCANMsg*, coming from the PCAN library. The motion command CAN message uses an extended identifier (motion command ID 0x0700004) and 6 Bytes payload, containing the three velocities as 2 Byte integers in the following order:

$$\text{CAN motion command data field: } [v_x, v_y, \omega]$$

The Krikkit drive expects the linear velocities (v_x, v_y) to be in [mm/s] and the angular velocity around the z-axis (ω) in [$mrad/s$]. For security reasons the function will send stop commands if the last received motion command is older than 500 ms.

4.3.1.1.3. Forwarding messages to the CAN bus

When receiving a message of type *"krikkit_driver/CAN_msg"* as shown in Listing 4.1 on the *"/CAN_send_msg"* topic the subscriber invokes a callback function which only copies the contained data into a PCAN CAN message and sends it over the interface.

4.3.1.1.4. Receiving and forwarding of CAN bus messages

The *can_read thread* uses a blocking function with time-out to wait for incoming messages on the CAN bus. The blocking wait preserves system resources and the time-out of one second allows periodically checking if the thread should terminate. If a new message is received over the CAN interface, in every case the message arrival time t_i is saved, the data field is copied into a *"krikkit_driver/CAN_msg"* message and published on the *"/CAN_rec_msg"* topic. In case of a drive velocity message, indicated by the message ID 0x0620001, the actual position, according to the last velocity information, is calculated and saved as new reference position until a new message arrives. After this the message arrival time t_i and the new velocity information $(\overrightarrow{v_i}, \omega_i)$ are stored. Finally the new odometry information is immediately published to the system via the *publish_odom* function which is also periodically called by the *odo_publish thread* at a rate of approximately $100\,Hz$. The drive velocity message data field looks the same as the one of the velocity command message mentioned above but the units are different. In this message the linear velocity unit is $[mm/2s]$ and $[°/2s]$ for the angular velocity.

4.3.1.1.5. Odometry calculations

Odometry is the estimation of a moving systems actual pose from sensor information, like wheel rotations or something similar, relative to its starting pose. The poses are measured in a reference coordinate frame, in this context called *World* frame (see Figure 4.15). This estimation is done by integrating the velocities of the system over time. Therefore the new pose at the moment t_i is always calculated relative to a previous pose at a moment t_{i-1} assuming that the velocities have not changed. In case of the used Krikkit drive the used information are the linear and angular velocity, internally calculated by the drive from wheel rotations.

$$\text{Linear: } \overrightarrow{v} = [v_x, v_y]^T; \text{ Angular: } \omega$$

Figure 4.15 shows a typical scenario for the Krikkit drive, represented as black triangle. The actual pose for the moment t_i is calculated relative to the last pose at t_{i-1}, assuming that the velocities have not changed since t_{i-1}, using the following formulas:

$$\Delta t = t_i - t_{i-1}$$

$$\overrightarrow{\Delta X} = \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} cos(\Theta_{i-1}) & -sin(\Theta_{i-1}) \\ sin(\Theta_{i-1}) & cos(\Theta_{i-1}) \end{bmatrix} \cdot \begin{bmatrix} v_{x_{i-1}} \\ v_{y_{i-1}} \end{bmatrix} \cdot \Delta t$$

$$\Delta\Theta = \omega_{i-1} \cdot \Delta t$$

$$\overrightarrow{X_i} = \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \overrightarrow{X_{i-1}} + \overrightarrow{\Delta X}$$

$$\Theta_i = \Theta_{i-1} + \Delta\Theta$$

4.3.1.1.6. Publishing odometry

This is done via the *publish_odom* function, periodically called by the *odo_publish thread* at a rate of approximately $100\,Hz$. The function is similar to that from the odometry tutorial shown in the ROS wiki:

`http://www.ros.org/wiki/navigation/Tutorials/RobotSetup/Odom`, 2012

It generates and publishes a *nav_msgs/Odometry* message (shown in Listing 4.3) filled with the actual odometry and velocity information. The actual odometry $(t_i, \overrightarrow{X_i}, \Theta_i)$

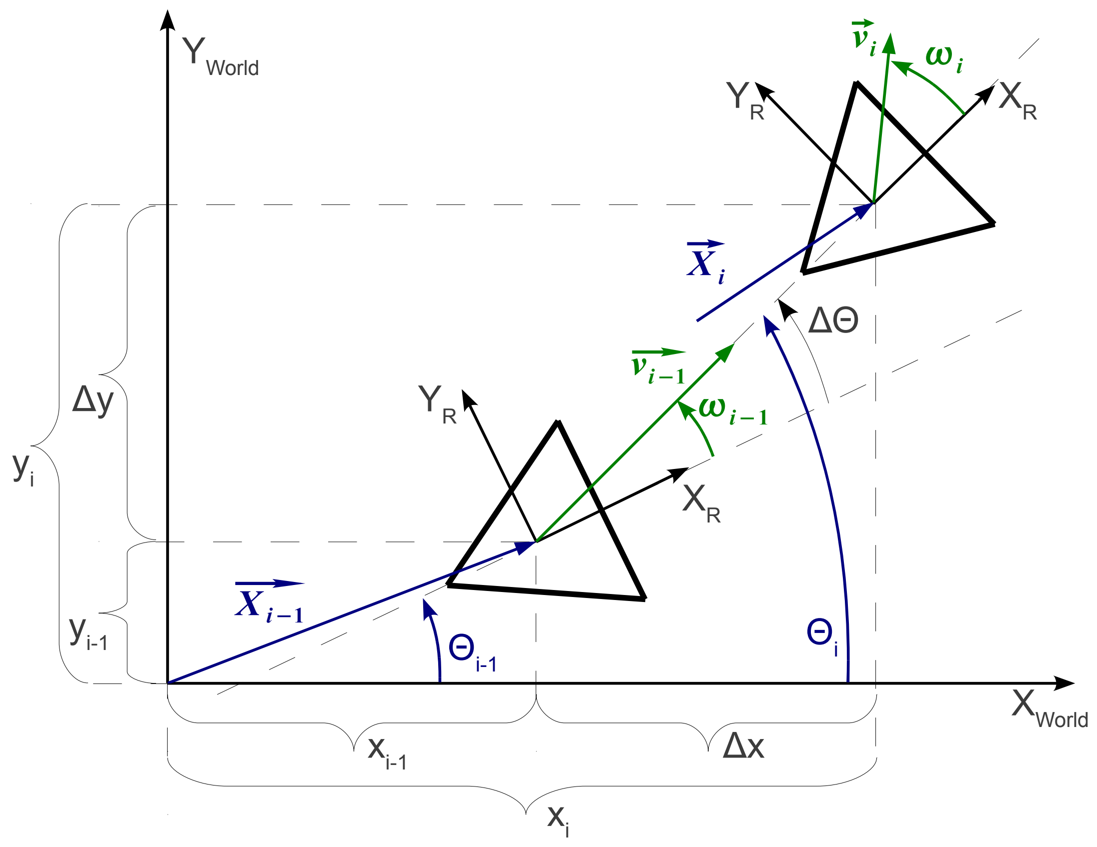


Figure 4.15.: This figure shows two poses of the robot (black rectangle), its attached coordinate frame and actual velocities at two moments t_{i-1} and t_i relative to a world coordinate frame as used for odometry calculations

is calculated via the formulas above (see paragraph 4.3.1.1.5) and the latest odometry and velocity information from time t_{i-1}. The generated odometry message also contains covariances for the pose and velocity data which is set to neutral, known good values as no real covariance values are available. Pose information are also used to publish a transformation to the ROS internal transform management system over the *"/tf"* topic. The ROS transform system handles all the available coordinate transformations and allows for easy coordinate transformations between the different coordinate frames. The frame for the odometry information is called *"/odom"* and the corresponding robot frame is called *"/base_link"* which is located at the robot's center with the x-axis pointing to the robots front and the z-axis pointing upwards which fully defines the coordinate system.

```
# nav_msgs/Odometry
Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

Listing 4.3: ROS message type for odometry information

4.3.1.2. teleop_krikkit node

This is a node for remote controlling the krikkit drive using the keyboard as input device. The linear speed can be varied in 10% steps up to a maximum of $2\,m/s$ while the angular speed can be varied in 5% steps up to $90\,°/s$. Depending on the actual selected speed and the pressed key a *"geometry_msgs/Twist"* message (see Listing 4.2) is filled with the chosen velocities and sent on the *"/cmd_vel"* topic. In contrast to many other remote control nodes this node offers additional commands for lateral movements.

4.3.2. Sensors

4.3.2.1. Navigation sensor

To get data from the navigation sensor which is a laser scanner (Sick LMS100), connected via Ethernet, slightly modified versions of the *LMS1xx* and *libLMS1xx* packages from Konrad Banachowicz are used. The project can be found at the web address below (last checked April 2013).

`https://github.com/konradb3/RCPRG-ros-pkg/tree/master/RCPRG_laser_drivers`

These two packages are the first part in the hardware abstraction for sensors as shown in Figure 4.4. The *libLMS1xx* package contains a library for connecting the scanner and receiving measurements via Ethernet. The laser sends the measurements in plain text which led to problems as the original version of the library uses a buffer with fixed size for storing the incoming data. So the library function for receiving data from the laser was modified to use dynamic memory for the buffer, to resize the buffer if it is full and to give a return value indicating the memory problem.

The *LMS1xx* package contains the node *LMS100* to fetch data from the laser scanner, to convert it to a ROS LaserScan message (see Listing 4.4) and to publish it to the *"/scan"* topic. The IP address for connecting the laser and the frame name used in the published message can be defined via ROS parameters ("host", "frame_id"). By default these values are set to *"192.168.1.2"* for the lasers IP address and *"/laser"* for the laser frame name. The node has been changed to evaluate the return value from the function for receiving scan data and if the return value indicates a problem while fetching the data no message is published. Another change was a bug-fix in the calculation of the number of scan values which depends on the scanning resolution.

4.3.2.2. Object recognition sensor

The 3D object recognition sensor is a Microsoft Kinect camera which is represented in the software via the *kinect_camera* package. This package is included in the ROS *kinect* stack and contains the *kinect_node* for handling the kinect camera as second part of the hardware abstraction for sensors as shown in Figure 4.4. The node fetches RGB, infrared and depth images from the camera and publishes them on different topics. Depth images are published twice as *sensor_msgs/PointCloud* (an older message type) and *sensor_msgs/PointCloud2* (the newer message type) for compatibility reasons. The package documentation can be found here:

`http://www.ros.org/wiki/kinect_camera`, (2012)

```
# sensor_msgs/LaserScan
Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

Listing 4.4: ROS message type for 2D laser data

4.3.3. Container manipulation

The container manipulation part, including the *"Container manipulation"* module from the *"Services & actions"* layer and the *"Container manipulation system"* module from the *"Hardware abstraction"* layer as shown in Figure 4.4, is implemented in the package *mechatronics_contr*. This package is part of the work described in [20] and is here only described briefly as the original work is in German. The package contains two important nodes (*nanotecContr* and *mechatronicsContr*) as well as service and action definitions used by these nodes.

4.3.3.1. nanotecContr node

The *nanotecContr* node uses a serial port for the connection to the Nanotec devices and implements a ROS service server with the name *"NanotecContr"* (definition shown in Listing 4.5) for sending commands to them. It represents the *"Vacuum gripper system"* module as well as the *"Container manipulation system"* module in the *"Hardware abstraction"* layer shown in Figure 4.4.

At start up the node tries to fetch parameters *"/baudrate"* (default 115200) and *"/dev_name"* (default "/dev/ttyUSB0") from the ROS parameter sever. These are then used for the serial connection. The default values represent the values used during this work. The node holds an object of class *NanoContr* shown in Figure 4.16 which opens the serial connection in its constructor. The method *sendCommand(...)* is the callback function for the *"NanotecContr"* service. It constructs the command string which is sent to the Nanotec device, using the *makeString(...)* method, according to the Nanotec command protocol and the received request data, sends it to the device via the *writeString(...)* method and waits for response from the Nanotec device for a maximum period of time as defined by the ROS parameter *"/timeout_time"* (default 2 seconds). If no response is received before time-out, the service call returns *"False"* which indicates an error while processing the call, and that the data contained in the response message are invalid. If

```
# NanotecContr.srv
# request definition
int16    motor_number
string   command
int64    value
bool     with_value
---
# response definition
bool     command_ok
int64    value
```

Listing 4.5: ROS service for sending commands to a Nanotec controller

a response is received from the Nanotec device the response is parsed and it is checked if the command was recognized correctly from the device. The service response is filled and the service call returns with *"True"* to indicate that the response data are valid.

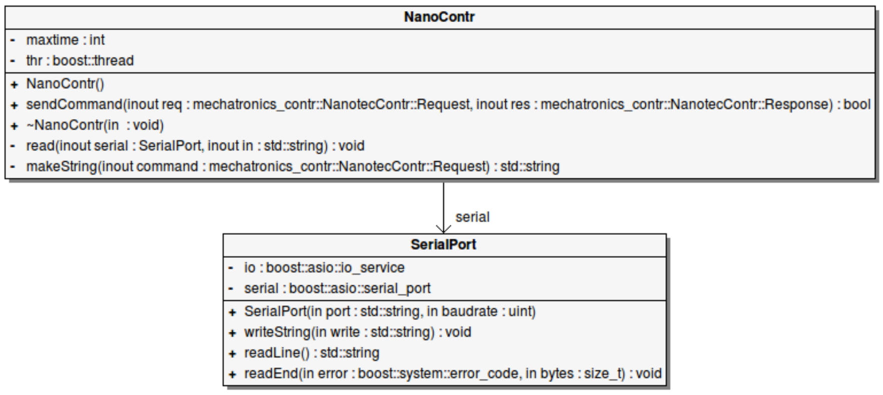

NanoContr
- maxtime : int
- thr : boost::thread
+ NanoContr()
+ sendCommand(inout req : mechatronics_contr::NanotecContr::Request, inout res : mechatronics_contr::NanotecContr::Response) : bool
+ ~NanoContr(in : void)
- read(inout serial : SerialPort, inout in : std::string) : void
- makeString(inout command : mechatronics_contr::NanotecContr::Request) : std::string
serial
SerialPort
- io : boost::asio::io_service
- serial : boost::asio::serial_port
+ SerialPort(in port : std::string, in baudrate : uint)
+ writeString(in write : std::string) : void
+ readLine() : std::string
+ readEnd(in error : boost::system::error_code, in bytes : size_t) : void

Figure 4.16.: Class diagram of the classes used in the *nanotecContr* node (source [20])

4.3.3.2. mechatronicsContr node

The *mechatronicsContr* node is the implementation of the *"Container manipulation"* module in the *"Services & actions"* layer shown in Figure 4.4. It implements a ROS action server of the name *"mechatronics_controller"* with the definition shown in Listing 4.6. Therefore it holds an object of class *MechatronicsContrAction* implementing all the functionality of the action server necessary for container manipulation. This class in turn holds two objects of class *MotorMove*, as shown in Figure 4.18, which control the two linear actuators of the container manipulation system by simply sending commands via the *NanotecContr* service described above.

An object of class *MotorMove* only communicates with the device having the address (motor number) specified in the constructor. The important methods for executing motion commands are *moveRef()* and *movePos(...)*. The method *moveRef()* has to be called at

```
# MechatronicsContr.action
# goal definition
uint8   command
uint8   REFPOS = 0      # move actuator to reference position
uint8   EINLAG = 1      # fetch container from stock
uint8   AUSLAG = 2      # put container to stock

uint8   ebene   # level number
---
# result definition
uint8   command
uint8   REFPOS = 0
uint8   EINLAG = 1
uint8   AUSLAG = 2

uint8   ebene   # level number
uint8   fehler  # error code
---
# feedback definition
```

Listing 4.6: ROS action for container manipulation

the beginning because it executes a calibration run which is necessary as all positions are measured incremental relative to the reference position. First the method sends a command to the device to load the profile for the calibration run followed by the command to start the motor and finally it polls the actual position of the linear actuator until the movement is finished. The method *movePos(const long pos)* loads the profile for absolute positioning relative to the reference position, sets the desired position given in motor steps by the parameter *"pos"*, starts the motor and polls the actual position until the desired position is reached. Possible errors during the process are reported via the return value of these two methods and can have the values shown in Listing 4.9.

In the constructor of the *MechatronicsContrAction* class the necessary ROS parameters (see Listing 4.7) are fetched from the parameter sever, the two *MotorMove* objects are created and the two actuators are moved to their reference positions. Additionally a publisher on the topic *"/kombot_joint_states"* of type *"sensor_msgs/JointState"* (see Listing 4.8) is created which is used by the method *publishPosition()* to publish the actual position of the linear actuators at a rate of 5Hz. This method is executed in a separate thread which is also created in the constructor. Finally the action server is started after it has been initialized with the method *executeCB(...)* as a goal call back and the *preemptCB()* method as preempt callback. The goal call back only calls the method *executeAction(...)* with the right parameters fetched from the goal message The preempt call back only signals the *MotorMove* objects to stop the motors. From now on a received goal triggers a sequence of motion commands to fulfil the requested task. The actions goal definition (see Listing 4.6) defines three possible motion commands. The command *REFPOS* performs a configuration run. *AUSLAG* performs a sequence of movements to

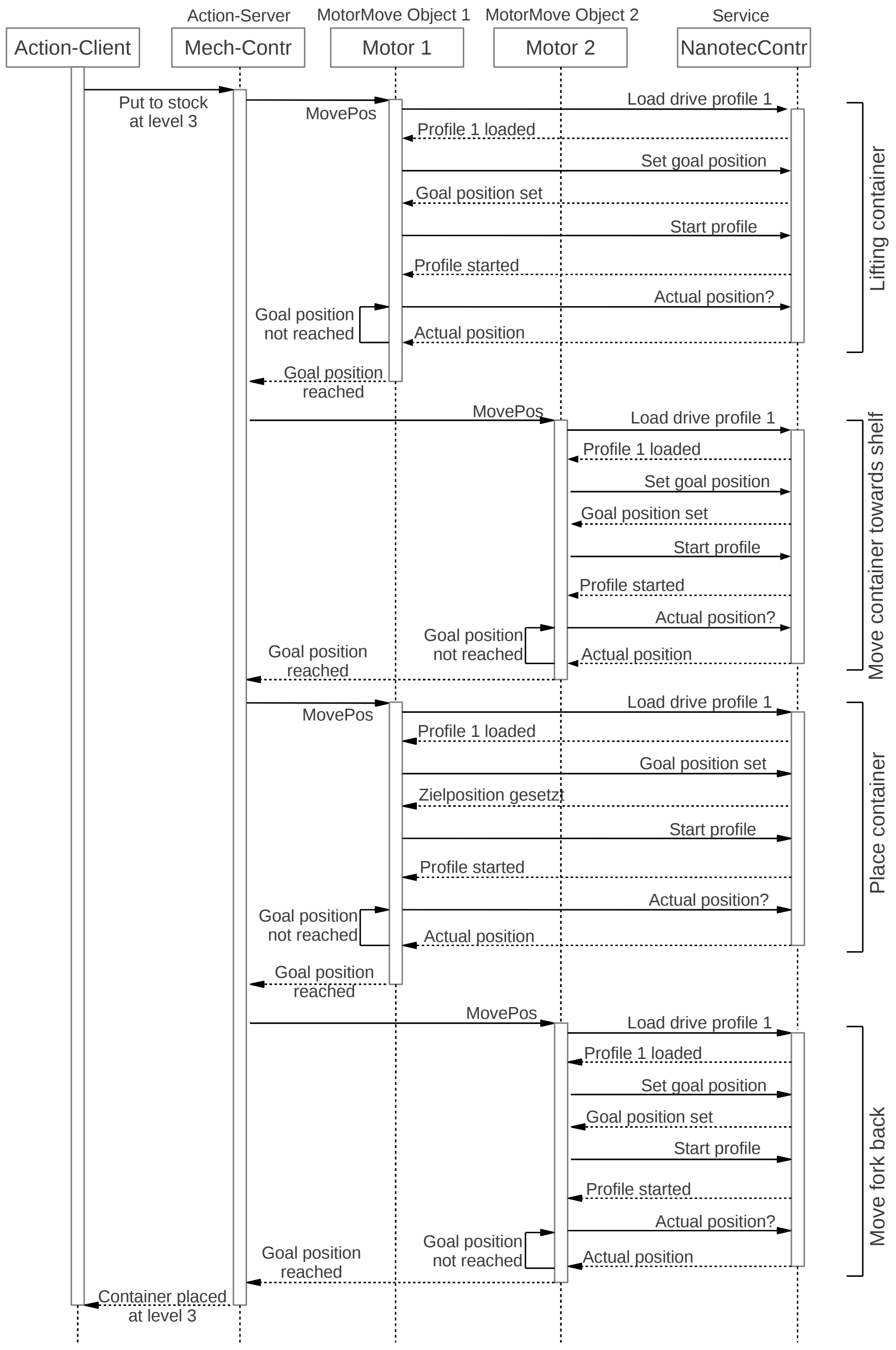


Figure 4.17.: Sequence diagram of the container placing process from the calling action client (left) down to the *NanotecContr* service (right) which is connected to the different Nanotec devices (diagram source [20])

fetch a container from the level given by *ebene* in the goal message while the command *EINLAG* performs a sequence to put the container to the shelf place at the given level. After the execution the action server returns the response containing the received goal information as well as an error code (*fehler*) with possible values as shown in Listing 4.9. Figure 4.17 shows the whole sequence of the container placing process, invoked by sending a goal with the *"EINLAG"* command (see Listing 4.6) to the *"mechatronics_controller"* action server. The action server then sends the necessary movement commands down to the *NanotecContr* service connected to the Nanotec devices.

```
# parameters for the mechatronics_controller action server
/reference_height       ...vertical distance from the floor to
                           the reference position of the
                           vertical axis
/levelheights           ...list containing the heights of the
                           different shelf levels
/endpos_v               ...vertical end position after fetching
                           the container
/endpos_h               ...horizontal end position
/liftheight_auslag      ...height to lift the fork after moving
                           it under the container
/liftheight_einlag      ...height above level height to put the
                           container back into the shelf
/forwardlenght_auslag   ...length to move the fork forward to
                           fetch the container
/forwardlenght_einlag   ...length to move the fork forward to
                           put the container back
/stepsperm_v            ...conversion factor for the vertical
                           linear actuator (steps/meter)
/stepsperm_h            ...conversion factor for the horizontal
                           linear actuator (steps/meter)
```

Listing 4.7: ROS parameters for the mechatronics_controller action server

```
# sensor_msgs/JointState
Header header
  uint32 seq
  time   stamp
  string frame_id
string[] name
float64[] position
float64[] velocity
float64[] effort
```

Listing 4.8: ROS message type for joint state messages

```
# possible error codes of the
# mechatronics_controller action server
 0... no error
 1... invalid command detected by the device
 2... error during communicate with Nanotec controller
 3... motor stopped unexpectedly
 4... action preempted (aborted)
```

Listing 4.9: Error codes of the mechatronics_controller action server

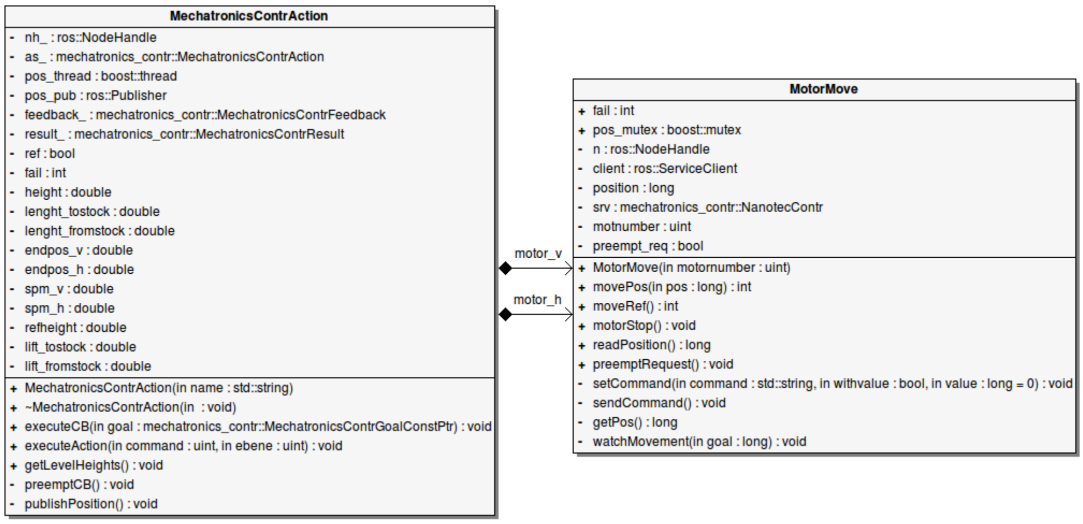

MechatronicsContrAction
- nh_ : ros::NodeHandle
- as_ : mechatronics_contr::MechatronicsContrAction
- pos_thread : boost::thread
- pos_pub : ros::Publisher
- feedback_ : mechatronics_contr::MechatronicsContrFeedback
- result_ : mechatronics_contr::MechatronicsContrResult
- ref : bool
- fail : int
- height : double
- lenght_tostock : double
- lenght_fromstock : double
- endpos_v : double
- endpos_h : double
- spm_v : double
- spm_h : double
- refheight : double
- lift_tostock : double
- lift_fromstock : double
+ MechatronicsContrAction(in name : std::string)
+ ~MechatronicsContrAction(in : void)
+ executeCB(in goal : mechatronics_contr::MechatronicsContrGoalConstPtr) : void
+ executeAction(in command : uint, in ebene : uint) : void
+ getLevelHeights() : void
- preemptCB() : void
- publishPosition() : void
motor_v
motor_h
MotorMove
+ fail : int
+ pos_mutex : boost::mutex
- n : ros::NodeHandle
- client : ros::ServiceClient
- position : long
- srv : mechatronics_contr::NanotecContr
- motnumber : uint
- preempt_req : bool
+ MotorMove(in motornumber : uint)
+ movePos(in pos : long) : int
+ moveRef() : int
+ motorStop() : void
+ readPosition() : long
+ preemptRequest() : void
- setCommand(in command : std::string, in withvalue : bool, in value : long = 0) : void
- sendCommand() : void
- getPos() : long
- watchMovement(in goal : long) : void

Figure 4.18.: Class diagram of the classes used in the *mechatronicsContr* node (source [20])

4.3.4. Navigation & localization

The "navigation & localization" block as shown in Figure 4.4 is split into two parts to solve the two navigation tasks indicated in Figure 4.1. The first task (*"move to shelf place"*) is the autonomous navigation of the robot to a way point on a map using the functionality of the ROS navigation stack. The following second task is the *"fine positioning"* of the robot using the navigation sensors and artificial landmarks. The two software parts, used to fulfil these tasks, will be described in the following.

4.3.4.1. Autonomous navigation

As already mentioned, the ROS navigation stack and its functionality is used for navigation and localization of the robot using a static map, representing the place of action (warehouse), and a laser scanner as navigation sensor. This navigation task also includes the 2D version of the path planning problem described in section 2.3. The basic book addressing common problems of autonomous robotics is [11].

The static map is built in a prior preparation step using the functionality of the *slam_gmapping* node from the ROS *gmapping* package (see also `http://www.ros.org/wiki/gmapping`, 2012). for building a map the robot, equipped with the navigation sensor, has to be driven around remotely and the *slam_gmapping* node, which uses Simultaneous Localization and Mapping (SLAM) techniques to combine odometry data coming from the robot and data from the navigation sensors, builds a occupancy grid map (message type *nav_msgs/OccupancyGrid*). This map is saved to a file for later use, using the *map_saver* node from the *map_server* package. For information about the techniques used in the *slam_gmapping* node have a look at [8].

The navigation stack has a simple concept but has to be configured for each robot type separately. Because this configuration can be very difficult its recommended to follow the tutorial provided under:

`http://www.ros.org/wiki/navigation/Tutorials/RobotSetup`, (2012)

To use the navigation stack some configuration requirements must be met by the robot. A configuration overview is shown in Figure 4.19. The blocks *amcl*, *move_base* and *map_-server* in this figure are part of the *ROS naviation stack*. In case of the Kombot the blocks *base controller* and *odometry source* are implemented by the *ODO_CAN_krikkit_-node* already described in subsubsection 4.3.1.1. The block *sensor sources* is implemented by the *LMS100* node described in subsubsection 4.3.2.1 as it is the only navigation sensor of the Kombot. Finally the coordinate transformation, bringing the laser sensor data from the laser frame to the robot base frame, are published. This is actually done by the *static_transform_publisher* nodes from the *tf* package.

For localization in the map the *amcl* node is used. Its name *amcl* is only the acronym for *Adaptive Monte Carlo Localization* which is the implemented localization approach. It takes a laser based occupancy grid map provided by the *map server* and uses laser scans from the navigation sensor and the transformations between the robot and odometry frame to estimate the robot's pose in the map frame. As result the estimated pose and a transformation between odometry and map frame are published. So the full transformation between robot and map is the combination of the transformation "robot to odometry" and

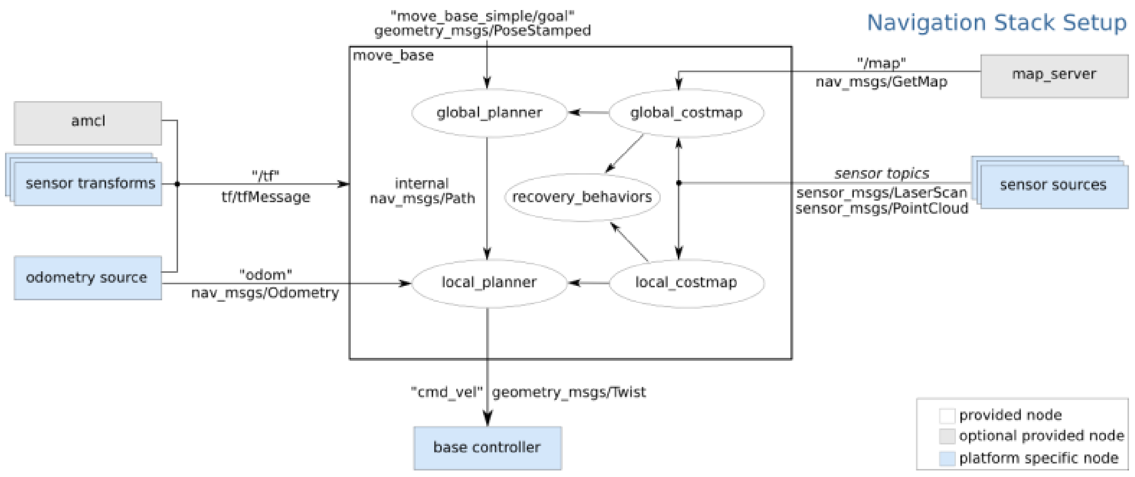


Figure 4.19.: ROS navigation stack configuration overview (source: `http://www.ros.org/wiki/move_base`, 2012)

"odometry to map". For more information about *amcl* and its configuration parameters have a look at `http://www.ros.org/wiki/amcl` (2012).

The actual navigation task is done by the *move_base* which consists of several components (see Figure 4.19). It implements an action server for the *MoveBase* action (definition in package *move_base_msgs*) and uses special interfaces, defined in the *nav_core* package, to link a global and a local planner together. Using these interfaces the default planners can be easily replaced with a custom planner but for the Kombot the default planners from the ROS navigation stack are used. These planners are implemented as plugins (more information about plugins in ROS can be found here: `http://www.ros.org/wiki/pluginlib` (2012)) and use two different cost maps for motion planning which are generated using the map published by the *map server* as a base and modifying it using the data from the *sensor sources*. The global cost map represents the whole map and all known obstacles, while in contrast the local cost map usually only represents a local area around the robot. The default global planner is the *NavfnROS* planner from the *navfn* package which uses the global cost map and the Dijkstra algorithm to find a global path from the actual position to the goal position received via the action server. The basic concept behind this global planner is described in [14]. This global plan is sent to the local planner which utilizes the local cost map to generate motion commands (velocities vx, vy, vth) for the mobile base. The motion commands are published as *"geometry_msgs/Twist"* message on the *"/cmd_vel"* topic and then, in case of the Kombot, sent to the mobile base via the *krikkit_driver* as described in subsubsection 4.3.1.1. When the local planner can not find a path it triggers recovery behaviours to solve the problem and aborts the received goal if no solution is found within some retries. Possible recovery behaviours are in-place rotations or clearing and rebuilding of the cost maps to remove obstacles which are no longer present, usually called dynamic moveable obstacles. The used default local planner is the *"base_local_planner/TrajectoryPlannerROS"* configured to make use of the implemented *Dynamic Window* approach, based on [6].

4.3.4.1.1. Problems with autonomous navigation

As it will be reported later on, the autonomous navigation works fine for the Krikkit drive alone but this is not true for the whole robot. The problem appears directly at start up when the robot tries to localize itself on the map by executing a sequence of standard recovery behaviours which are normally executed when the robot perceives itself as stuck. The most common recovery behaviour of the robot is to perform a few in-place rotations. This usually helps the robot to clear out space by removing no longer present dynamic obstacles from the used cost maps but also helps to relocalise itself. The problem now lies in the small drive in relation to the relatively big and heavy frame carrying all the other components. This combination leads to spinning wheels when the robot starts turning and also prevents the robot from reaching the desired rotational speed. These two things increase the problem because the expected motion and the received odometry is more and more in conflict with the data received from the navigation sensor. Thus, the position estimation is getting worse and worse and in almost all cases leads to fatal collisions during the rotations. So autonomous navigation using the ROS navigation stack does not work for the prototype robot described in this work.

4.3.4.2. Fine positioning

In the given scenario only the container manipulation unit requires highly accurate positioning in front of the shelves but under real circumstances other places with similar strict positioning constraints may exist. Such places could be order bin transfer points or charging stations as shown in Figure 4.2, to only name just two. The linear position tolerance is less than $3cm$ and the angular tolerance is about 3° (see [20]). To achieve such a precise positioning a landmark based positioning system has been developed for the Kombot (*shelf_fine_positioning* package). The system uses data from the navigation sensors, for the prototype a single *Sick LMS100* laser scanner, as input for detecting the landmarks. The used landmarks are wedges with a side length of 255mm and an enclosed angle of $\alpha_{Wg} \approx 134°$ between the front surfaces (see Figure 4.20, left). An image of such a wedge, made of cardboard and used for testing, is shown in the first row of Figure 4.21. They are positioned under the shelf in the middle of two vertical rows of shelf places so that the front surfaces are visible for the robot's navigation sensor and the landmark can be used as position reference for both rows.

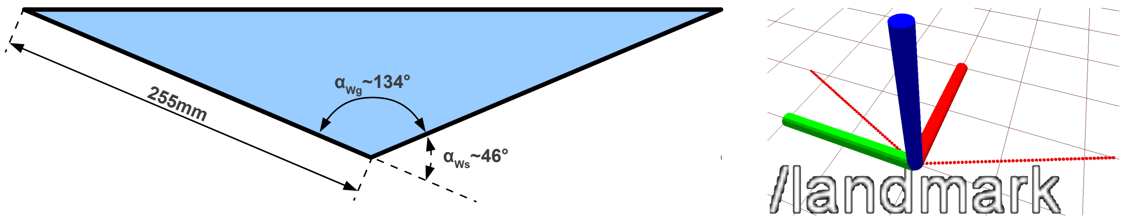


Figure 4.20.: *Left:* Shape and dimensions of the landmarks used for fine positioning; *Right:* Landmark point cloud template (red dotted wedge) and its according coordinate frame ([red, green, blue] → [x, y, z])

As shown in Figure 4.22, two action server nodes are used for fine positioning. One for

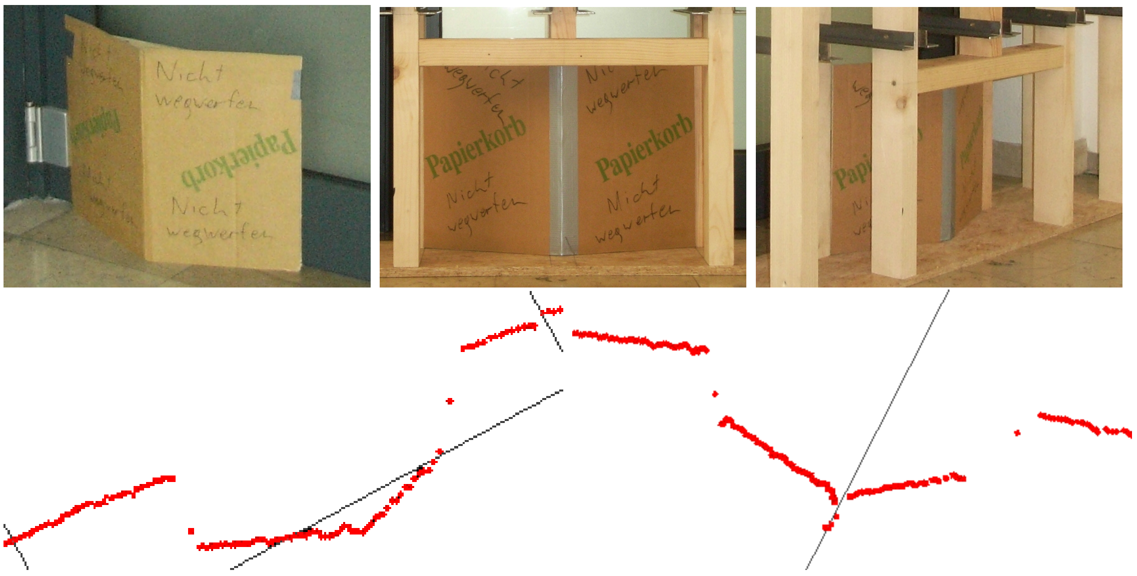


Figure 4.21.: *First row:* Images of the cardboard wedge used as landmark for testing; *Second row:* Laser scans of the landmark wedge during some early tests

tracking the robot's 2D pose relative to the landmark using the navigation sensor and one controlling the robots motion to reach a desired goal pose relative to the landmark within a specified tolerance. The two action servers will be described in the following.

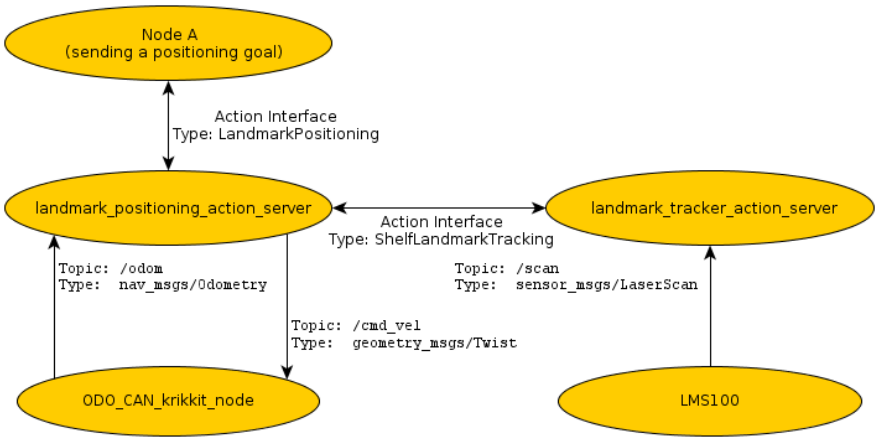

Node A
(sending a positioning goal)
Action Interface
Type: LandmarkPositioning
landmark_positioning_action_server
Action Interface
Type: ShelfLandmarkTracking
landmark_tracker_action_server
Topic: /odom
Type: nav_msgs/Odometry
Topic: /scan
Type: sensor_msgs/LaserScan
Topic: /cmd_vel
Type: geometry_msgs/Twist
ODO_CAN_krikkit_node
LMS100

Figure 4.22.: Overview of the nodes and communication paths for landmark positioning

4.3.4.2.1. Landmark tracker

The node *landmark_tracker_action_server* implements an ROS action server with the action definition shown in Listing 4.10. Its task is to estimate the robot's pose relative to a landmark with the shape shown in Figure 4.20 using the data from the navigation sensor which in case of the Kombot is a Sick laser scanner of type LMS100 (see also subsection 4.2.2 and subsubsection 4.3.2.1). The second row of Figure 4.21 shows laser scans of such landmarks used for testing. The goal defines the topic (*"scan_topic"*) for the sensor data which is expected to be of type *"sensor_msgs/LaserScan"* (see Listing 4.4 for message definition) and the name given to the frame defined by the tracked landmark (*landmark_frame*). While a received goal is processed the action server regularly sends feedback messages containing the actual tracking status. Possible values are *"INITAL_ALIGNMENT"*, which means that no position can be estimated at this time, or *TRACKING*, which means that an estimated position is published as coordinate transformation to the *"tf"* topic and can be used by all nodes via the ROS transform system. The published coordinate transformation is the pose of the *"landmark"* frame measured in the *"laser"* frame which also means that this is the transformation to bring points from the *"landmark"* frame into the *"laser"* frame. As there is no actual goal which can be reached, the task is executed until the goal gets cancelled.

```
# ShelfLandmarkTracking.action
# goal definition
string scan_topic
string landmark_frame
---
#result definition
---
#feedback
uint8 status
uint8 INITAL_ALIGNMENT          = 0
uint8 TRACKING                  = 1
```

Listing 4.10: ROS action for landmark tracking

The node *landmark_tracker_action_server* only defines the execute function and starts the action server which calls this function whenever a new goal is received. The execute function creates an object of class *LandmarkTracker* (Figure 4.23) and uses the corresponding methods to provide it with a pointer to the action server object and the name for the landmark frame. It also opens a subscriber to the *scan_topic* with the *LandmarkTracker*'s *"laserScanCallback(...)"* method as callback which is executed whenever a new laser scan arrives until the goal is cancelled.

When a new laser scan arrives the *"laserScanCallback(...)"* method is called and if there is no valid previous pose estimation for the landmark a method for finding an initial pose (*initialAlignment_LineSegmentation(...)*) is called. If there is a valid previous estimation a simplified method (*track_landmark(...)*) is called to find the relative transformation between the last pose and the actual one. In both cases the methods return if a valid pose was estimated and if this is true the action server feedback is set to *TRACKING* otherwise to *INITAL_ALIGNMENT.*

The whole process of estimating the robot's pose, relative to the landmark, is based on the functionalities of the PCL already described in section 2.2. In both possible states, *INITAL_ALIGNMENT* or *TRACKING*, a received laser scan is converted into a PCL

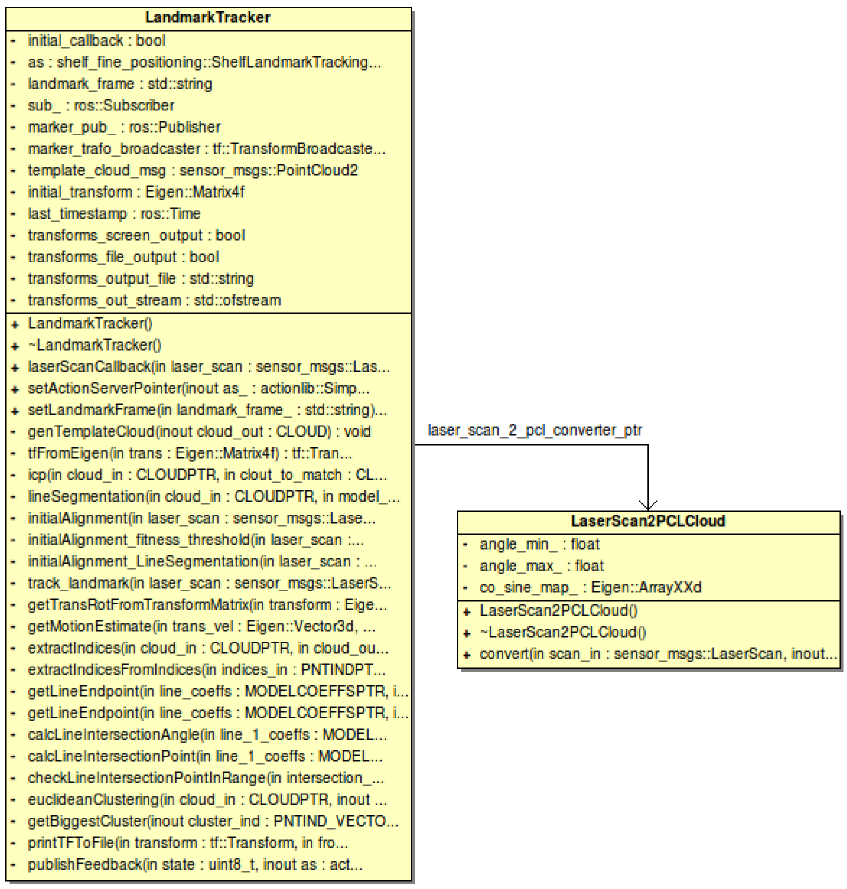

LandmarkTracker
- initial_callback : bool
- as : shelf_fine_positioning::ShelfLandmarkTracking...
- landmark_frame : std::string
- sub_ : ros::Subscriber
- marker_pub_ : ros::Publisher
- marker_trafo_broadcaster : tf::TransformBroadcaste...
- template_cloud_msg : sensor_msgs::PointCloud2
- initial_transform : Eigen::Matrix4f
- last_timestamp : ros::Time
- transforms_screen_output : bool
- transforms_file_output : bool
- transforms_output_file : std::string
- transforms_out_stream : std::ofstream
+ LandmarkTracker()
+ ~LandmarkTracker()
+ laserScanCallback(in laser_scan : sensor_msgs::Las...
+ setActionServerPointer(inout as_ : actionlib::Simp...
+ setLandmarkFrame(in landmark_frame_ : std::string)...
- genTemplateCloud(inout cloud_out : CLOUD) : void
- tfFromEigen(in trans : Eigen::Matrix4f) : tf::Tran...
- icp(in cloud_in : CLOUDPTR, in clout_to_match : CL...
- lineSegmentation(in cloud_in : CLOUDPTR, in model_...
- initialAlignment(in laser_scan : sensor_msgs::Lase...
- initialAlignment_fitness_threshold(in laser_scan :...
- initialAlignment_LineSegmentation(in laser_scan : ...
- track_landmark(in laser_scan : sensor_msgs::LaserS...
- getTransRotFromTransformMatrix(in transform : Eige...
- getMotionEstimate(in trans_vel : Eigen::Vector3d, ...
- extractIndices(in cloud_in : CLOUDPTR, in cloud_ou...
- extractIndicesFromIndices(in indices_in : PNTINDPT...
- getLineEndpoint(in line_coeffs : MODELCOEFFSPTR, i...
- getLineEndpoint(in line_coeffs : MODELCOEFFSPTR, i...
- calcLineIntersectionAngle(in line_1_coeffs : MODEL...
- calcLineIntersectionPoint(in line_1_coeffs : MODEL...
- checkLineIntersectionPointInRange(in intersection_...
- euclideanClustering(in cloud_in : CLOUDPTR, inout ...
- getBiggestCluster(inout cluster_ind : PNTIND_VECTO...
- printTFToFile(in transform : tf::Transform, in fro...
- publishFeedback(in state : uint8_t, inout as : act...
laser_scan_2_pcl_converter_ptr
LaserScan2PCLCloud
- angle_min_ : float
- angle_max_ : float
- co_sine_map_ : Eigen::ArrayXXd
+ LaserScan2PCLCloud()
+ ~LaserScan2PCLCloud()
+ convert(in scan_in : sensor_msgs::LaserScan, inout...

Figure 4.23.: Class diagram of the *LandmarkTracker* class used for the *landmark_tracker_action_server* node

point cloud P using the *convert(...)* method from the *LaserScan2PointCloud* class as shown in Figure 4.23. This class is an adapted and simplified version of the *LaserProjection* class, defined in the *laser_geometry* package, which converts the laser scan messages into messages of type *sensor_msgs/PointCloud* or *sensor_msgs/PointCloud2* instead of PCL point clouds. For each pair of distance value (d_i) and scanning angle (β_i) this convert method calculates the point coordinates x_i and y_i (z_i is always zeros because this is planar laser scan) with the following formula:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = d_i \cdot \begin{bmatrix} cos(\beta_i) \\ sin(\beta_i) \end{bmatrix}$$

The values for $cos(\beta_i)$ and $sin(\beta_i)$ are stored in a lookup table for performance reasons and are only calculated at the first conversion or if the scan parameters (*number of scans, min and max angle, angle increment*) are different, compared to the last conversion. The points are then inserted into the output point cloud if the corresponding distance value is within the valid range as defined in the laser scan message. The method also returns the points clouds extreme values P_{extr} (x_min, y_min, x_max and y_max) because this information will be needed for the initial alignment. The actual process of pose estimation is realized using the ICP algorithm from the PCL described in subsection 2.2.4. Therefore a point cloud template P_{Temp} of the landmarks front surface (see Figure 4.20, right), with a distance of $5mm$ between neighbouring points, is generated via the *genTemplateCloud(...)* method and saved for later use. The angle α_{Ws} is the smaller intersection angle between the lines defined by the landmarks front surfaces (see Figure 4.20, left).

Initial alignment

The initial alignment is done by the method *initialAlignment_LineSegmentation(...)* implementing algorithm 2. The implemented approach uses PCL line segmentation, similar to the plane segmentation described in subsection 2.2.2, to extract all lines L_i having a sufficient number of inlier points ($N_{inlier} > N_{min}$). For all extracted lines the intersection points and angles are calculated. All intersection points within the point clouds border defined by it extreme points P_{extr} having an intersection angle close to the desired wedge angle α_{Ws} are possible candidates for a front edge of a landmark and used for the next steps. All candidate intersection points are now used to generate initial transformations T_{init} for the following ICP step trying to align the landmarks template cloud P_{Temp} to the input cloud from the laser scanner. If the best alignment has a fitness score less than a defined threshold (the smaller the better), it is assumed that the landmark was detected properly and the tracking of the landmark is started otherwise the initial alignment is restarted with the next received laser scan. For the tracking step the resulting transformation T_I is published, velocity data is reset and a time-stamp is saved.

Two other approaches for initial alignment are implemented in the methods *initialAlignment(...)* and *initialAlignment_fitness_threshold(...)*. The first approach generates an user defined number (N_p) of random initial points within the point clouds borders used to create initial transformations T_{init} for the following ICP step. The remaining steps are the same as above. The approach implemented in the method *initialAlignment_fitness_threshold(...)* also generates random initial transformations but in contrast takes the first final transformation T_i with a fitness score below the threshold f_{th} and aborts if no one has been found

```
1  convert the laser scan into a PCL point cloud $P$
2  get extreme points $P_{extr}$ of point cloud $P$
3  get all lines $L_i$ with $N_{inliers} \geq N_{min}$ using PCL line segmentation
4  calculate all line intersection angles $\alpha_i$
5  calculate all line intersection points $p_i$
6  forall $p_i$ within $P_{extr}$ and $\alpha_i \approx \alpha_{Ws}$ ($\pm\Delta_\alpha$, user defined tolerance) do
7      generate initial transform $T_{init}$ with $x = x_{p_i}, y = y_{p_i}, \theta = atan2(x_{p_i}, y_{p_i})$
8      use PCL ICP with $T_{init}$ to align $P_{Temp}$ to $P$
9      get fitness score $f_i$ and final transform $T_{icp}$ from ICP
10     calculate final transform $T_i = T_{icp} \cdot T_{init}$
11 end
12 find alignment with best fitness score $f_I$ having the final transform $T_I$
13 if $f_I > f_{th}$ (user defined threshold) then
14     report a misalignment
15 else
16     publish $T_I$
17     set $vel\_x = vel\_y = vel\_theta = 0$
18     save the laser scans time-stamp and $T_I$ (as $T_{I_{last}}$) for the tracking step
19     start tracking
20 end
```

Algorithm 2: Landmark detection: initial alignment algorithm

until an user defined number (N_t) of tries. During some early tests both approaches performed much worse compared to the line segmentation approach. For the test $N_p = 100$ initial points have been used for the first approach and the second approach had $N_t = 500$ tries to find an alignment with a fitness score below f_{th}.

In these tests a typical initial alignment with the line segmentation approach took less than $100ms$ and almost always found an alignment with $f_i < f_{th}$ at the first arriving laser scan. It also almost always aligned the template at the right place in the point cloud, means at the real position of the landmark. A typical run with the *initialAlignment(...)* approach took about 9 seconds and in about 40% of the test cases it took two laser scans to find an alignment with $f_I < f_{th}$. Similar to the line segmentation approach almost all test cases found the real landmark place. In case of the *initialAlignment_fitness_threshold(...)* approach the average runtime was 1.3 seconds with an average of 16 tries for an alignment with $f_I < f_{th}$, which is not too bad, but in about 30% of the test cases the resulting landmark position was wrong placed.

Landmark tracking

The actual landmark tracking after initial alignment is done by the *track_landmark(...)* method, implementing algorithm 3. After converting the received point cloud into a PCL compatible format the robot's relative motion from the last known position to the actual one is estimated using the saved velocity information and the laser scan time stamp difference. Using the last landmark transformation $T_{I_{last}}$ and the motion transformation a new initial transformation T_{init} is calculated for the ICP step aligning the landmark template cloud P_{Temp} to the received laser scan. If the alignment fits well enough, which means that the resulting fitness score is below the threshold, the new resulting transformation is published and saved as well as the new calculated velocities the new time-stamp. If the alignment does not fit well enough the landmark tracker is reset to initial alignment.

```
1  convert the laser scan into a PCL point cloud $P$
2  estimate robot motion using $vel\_x, vel\_y, vel\_theta$ and $\Delta t$
3                                  // $\Delta t$...laser scans time-stamp difference
4  use motion estimation to calculate a transform $T_{motion}$
5  calculate initial transform $T_{init} = T_{motion} \cdot T_{I_{last}}$
6  use PCL ICP with $T_{init}$ to align $P_{Temp}$ to $P$
7  get the fitness score $f_I$ and transform $T_{icp}$
8  calculate final transform $T_I = T_{icp} \cdot T_{init}$
9  if $f_I > f_{th}$ (user defined threshold) then
10     report a misalignment
11     restart with initial alignment
12 else
13     publish $T_I$
14     calculate and save $vel\_x$, $vel\_y$ and $vel\_theta$ from $T_I$, $T_{I_{last}}$ and $\Delta t$
15     save the the laser scans time-stamp
16     save $T_I$ as $T_{I_{last}}$
17 end
```

Algorithm 3: Landmark detection: landmark tracking algorithm

4.3.4.2.2. Position controller

The *landmark_positioning_action_server* node implements an ROS action server with the action definition shown in Listing 4.11. Via the action goal the user defines a 2D *goal_pose* relative to the *landmark_frame* for a new frame with the name given by *pose_target_frame*. The task for the position controller is to move the robot in a way that the robot's, frame given by *robot_frame_for_target*, reaches the given pose within the tolerances given by *goal_lin_tolerance* and *goal_rot_tolerance*. Therefore the node creates an object of the class *LandmarkPositionController* as shown in Figure 4.24 and executes its *run(...)* method where everything is initialized. The *LandmarkPositionController* object holds a *tf::TransformListener* handling coordinate transformations, a *LandmarkTransformFilter* (also shown in Figure 4.24), an action client connected to the action server for tracking the landmark (see paragraph 4.3.4.2.1), a publisher for sending the motion commands, a subscriber to the odometry topic for checking the actual robot motion and the actual action server as mentioned above.

The *LandmarkTransformFilter* mentioned above is nothing more than a mean filter for the transformation between given target and source frame. It autonomously fetches the transformation from the ROS transform system at the given rate using a separate thread and stores the transformation values (translation and rotation) in a queue of the given length. When called the *getTransform(...)* method calculates the resulting average transformation from the queued values and returns it. The class also offers a method

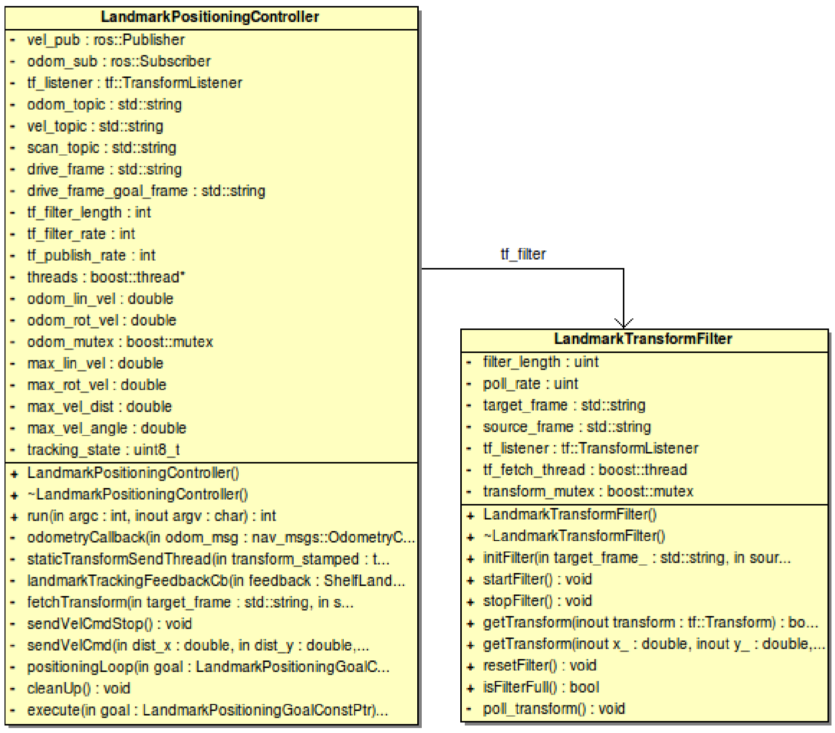

LandmarkPositioningController
- vel_pub : ros::Publisher
- odom_sub : ros::Subscriber
- tf_listener : tf::TransformListener
- odom_topic : std::string
- vel_topic : std::string
- scan_topic : std::string
- drive_frame : std::string
- drive_frame_goal_frame : std::string
- tf_filter_length : int
- tf_filter_rate : int
- tf_publish_rate : int
- threads : boost::thread*
- odom_lin_vel : double
- odom_rot_vel : double
- odom_mutex : boost::mutex
- max_lin_vel : double
- max_rot_vel : double
- max_vel_dist : double
- max_vel_angle : double
- tracking_state : uint8_t
+ LandmarkPositioningController()
+ ~LandmarkPositioningController()
+ run(in argc : int, inout argv : char) : int
- odometryCallback(in odom_msg : nav_msgs::OdometryC...
- staticTransformSendThread(in transform_stamped : t...
- landmarkTrackingFeedbackCb(in feedback : ShelfLand...
- fetchTransform(in target_frame : std::string, in s...
- sendVelCmdStop() : void
- sendVelCmd(in dist_x : double, in dist_y : double,...
- positioningLoop(in goal : LandmarkPositioningGoalC...
- cleanUp() : void
- execute(in goal : LandmarkPositioningGoalConstPtr)...
tf_filter
LandmarkTransformFilter
- filter_length : uint
- poll_rate : uint
- target_frame : std::string
- source_frame : std::string
- tf_listener : tf::TransformListener
- tf_fetch_thread : boost::thread
- transform_mutex : boost::mutex
+ LandmarkTransformFilter()
+ ~LandmarkTransformFilter()
+ initFilter(in target_frame_ : std::string, in sour...
+ startFilter() : void
+ stopFilter() : void
+ getTransform(inout transform : tf::Transform) : bo...
+ getTransform(inout x_ : double, inout y_ : double,...
+ resetFilter() : void
+ isFilterFull() : bool
- poll_transform() : void

Figure 4.24.: Class diagram of the *LandmarkPositioningController* class used for the *landmark_positioning_action_server* node

called *isFilterFull()* for checking if the queue is already filled which can be from interest after a reset (*resetFilter()* method) for instance.

```
# LandmarkPositioning.action
# goal definition
 string                    landmark_frame
 string                    pose_target_frame
 string                    robot_frame_for_target

 geometry_msgs/Pose2D      goal_pose

 float32                   goal_lin_tolerance
 float32                   goal_rot_tolerance
---
#result definition
---
#feedback
```

Listing 4.11: ROS action for positioning relative to a landmark

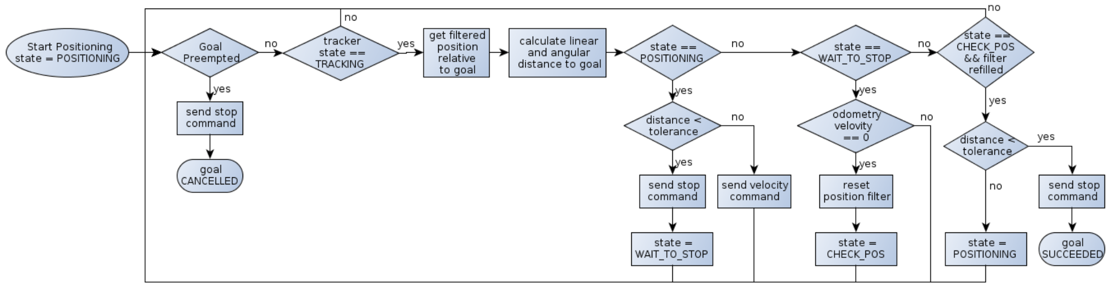


Figure 4.25.: Algorithm sketch of the positioning loop used for fine positioning

When a new goal is received the action server invokes the execute method. This method first sends an action goal to the landmark tracker action server (see paragraph 4.3.4.2.1) to start the tracking process. Then a thread with a static transform publisher is started which periodically publishes a transformation that defines the frame of the positioning goal (*pose_target_frame*) relative to the landmark (*landmark_frame*). The values for this transformation are given by *goal_pose*. The next step is to calculate the goal pose for the robot's drive which is given by the ROS parameter *"DRIVE_FRAME"* and to start a transform publisher thread, publishing this position (*DRIVE_FRAME_goal_frame*) relative to the landmark frame. Then the *LandmarkTransformFilter* is initialized for the transformation *DRIVE_FRAME* to *DRIVE_FRAME_goal_frame*. The task is now to move the robot in a way that the translation and rotation of this transformation become zero. The filter is necessary as the position transformation coming from the landmark tracker is very noisy and therefore needs to be filtered before utilization. Now the positioning loop which is a simple state machine, running at a rate of 10Hz, is started with state *"POSITION-*

ING". Figure 4.25 shows a sequence diagram of this loop. In every cycle it is checked if the positioning goal has been cancelled. If this is the case the loop is aborted and the action server sets the goal to status "cancelled". It is also checked if the sate of the landmark tracker is *"TRACKING"* which means that the transformation, indicating the robot's position relative to the landmark, is valid. Next the position relative to the goal, tracked by the *LandmarkTransformFilter*, is fetched and the linear and angular distance to the goal pose is calculated. The next steps depend on the actual state. In the state *"POSITIONING"* the next step is checking of the distance and if it is within the tolerance a stop command is sent to the drive and the state is changed to *"WAIT_TO_STOP"* otherwise a velocity command is generated and sent via the *sendVelCmd(...)* method. The state *"WAIT_TO_STOP"* checks the received odometry and waits until the robot has stopped. If this is true the *LandmarkTransformFilter* is cleared and the state is changed to *"CHECK_POS"*. If the state is *"CHECK_POS"* and the *LandmarkTransformFilter* is refilled, the distance is checked again and if its within the tolerance the goal is reached (goal *"SUCCEEDED"*) and then the action is finished. If the distance is outside the tolerance the state is changed back to *"POSITIONING"*.

The velocities which are sent to the drive are calculated according to the linear and angular distance to the goal and some ROS parameters. Two ROS parameters *"MAX_LIN_-VEL"* and *"MAX_ROT_VEL"* define the maximum linear and angular velocities which are only applied if the distance to the goal is more than *"MAX_VEL_DIST"* for the linear and *"MAX_VEL_ANGLE"* for angular distance. If the distance is less than this limit the according velocity is linearly reduced until it reaches a lower limit given by the parameters *"MIN_LIN_VEL_FACTOR"* and *"MIN_ROT_VEL_FACTOR"* respectively. The actual velocities are calculated via the scheme described in algorithm 4.

```
1  $f_{lin} = min(1, \frac{|linear\_distance|}{MAX\_VEL\_DIST})$
2  $f_{rot} = min(1, \frac{|angular\_distance|}{MAX\_VEL\_ANGLE})$
3  $t_{lin} = \frac{|linear\_distance|}{f_{lin} \cdot MAX\_LIN\_VEL}$
4  $t_{rot} = \frac{|angular\_distance|}{f_{rot} \cdot MAX\_ROT\_VEL}$
5  if $t_{lin} > t_{rot}$ then
6      $f_{lin} = max(f_{lin}, MIN\_LIN\_VEL\_FACTOR)$
7      $t_{use} = \frac{|linear\_distance|}{f_{lin} \cdot MAX\_LIN\_VEL}$
8  else
9      $f_{rot} = max(f_{rot}, MIN\_ROT\_VEL\_FACTOR)$
10     $t_{use} = \frac{|angular\_distance|}{f_{rot} \cdot MAX\_ROT\_VEL}$
11 end
12 $vel\_linear.x = linear\_distance.x / t_{use}$
13 $vel\_linear.y = linear\_distance.y / t_{use}$
14 $vel\_angular = angular\_distance / t_{use}$
```

Algorithm 4: Landmark positioning algorithm

These calculations ensure that the robot uses a higher velocity for the greater (more important) distance (linear or angular) to reach the goal faster, but keeps the motion smooth.

4.3.5. Arm navigation

The task of arm navigation, represented via the *"Arm navigation"* module in Figure 4.4, is to move the arm from its actual position to a goal pose along a collision free path, also called trajectory. To achieve this, several problems like inverse kinematics, forward kinematics, motion planning and collision checking have to be solved. To solve these problems for the Katana arm the the ROS Diamondback versions of the *ROS arm navigation* stack ([13]) and the *katana_driver* stack ([9]) from the University Osnabrück are used.

The version of the *katana_driver* stack contains, among other things, the package *katana_arm_navigation* with a full configuration of the arm navigation pipeline for three different types of Katana arms but not for the used Katana 450 6M180. So this package and its contained configuration as well as the depending package *katana_description* have been taken as a base and adapted for the needs of the Katana 450 6M180. The adaptations concerned the robot's description in the *katana_description* package which defines the shape of the robot parts and how they are mounted together. This is done by using a URDF (Unified Robot Description Format) configuration file. The original files coming from the *katana_driver* stack only contained the arm and so the rest of the robot parts (main frame, container manipulation units, etc.) had to be added and the original two-finger gripper was replaced with a simple cylinder representing the vacuum gripper. Figure 4.26 shows the resulting robot model used during this work.

Another adaption concerned some configuration and ROS *launch* files containing hardware parameters for the *katana* node which physically connects the arm control box via Ethernet and therefore needs information about the connected hardware like the IP address of the control box or joint limits (angle, speed, acceleration, etc.) of the arm. The mentioned *katana* node represents the *"Robot arm"* module in the *"Hardware abstraction"* layer as shown in Figure 4.4. A simplified diagram of the used arm navigation pipeline configuration, containing the most important nodes and communication paths, is shown in Figure 4.27 and a typical arm movement sequence is described in subsubsection 4.3.5.3.

4.3.5.1. Arm navigation pipeline

In the following paragraphs the most important modules of the arm navigation pipeline are described briefly. They are arranged by their appearance in a typical arm movement sequence as described in the following subsubsection 4.3.5.3.

Collision map
For saving computing power the scene in the robot's workspace, or in this case rather the robot arm's workspace, is assumed to be static as long as the arm is moving. Therefore the node *collision_map_self_occ_node* (Figure 4.27, left) is called, before the arm starts its operation and the created collision map is used for the whole task like moving an

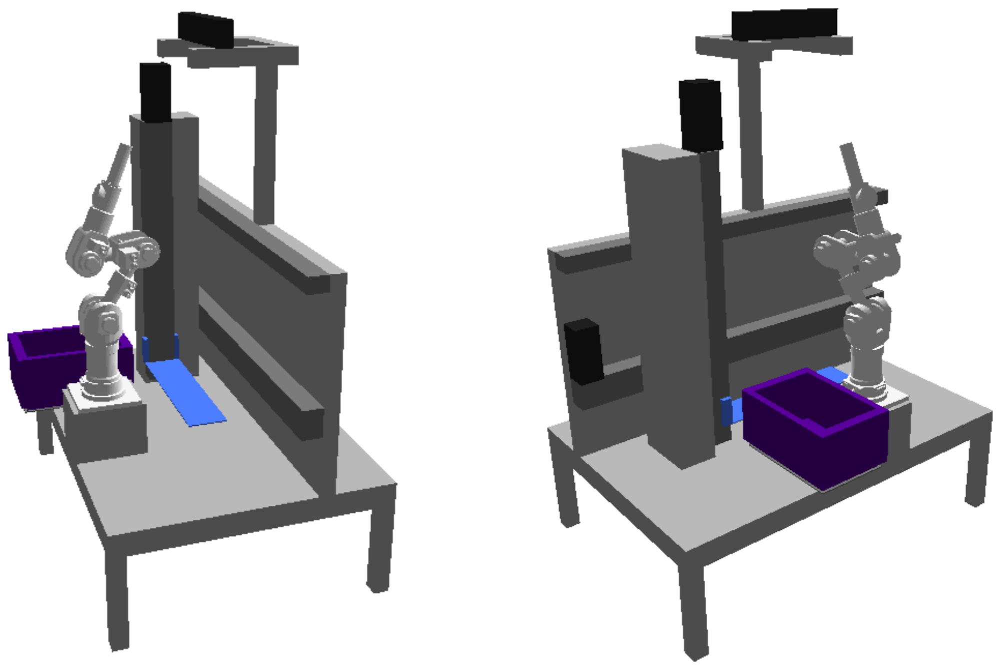


Figure 4.26.: URDF model of the Kombot prototype

item from the container to the order bin. When called via its action interface the *collision_map_self_occ_node* creates and publishes a collision map constructed from point cloud data periodically received over the corresponding ROS topic. The collision map is called static because it is only updated when the node is called to do so in contrast to the dynamic one, that is automatically updated periodically or each time a new point cloud is received.

In the used configuration the used point cloud data comes from the Microsoft Kinect camera and is filtered before provided to the *collision_map_self_occ_node*. In the first step this data is filtered by the *clear_known_objects* node which removes points belonging to *"known"* objects that have been published on the topics *"/collision_object"* and *"/attached_collision_object"* (not displayed). In the second step the *self_filter* node removes points belonging to the robot itself as defined by the URDF description file. The final filtered point cloud is then passed to the *collision_map_self_occ_node* which then creates and publishes the new collision map.

The *collision_map_self_occ_node* groups the remaining points in the point cloud and represents them by a set of oriented bounding boxes. In the collision map message these boxes are represented by its center point, defined in an also given frame and their extents. The message also contains a rotation axis and a rotation angle for each bounding box make them "oriented".

Move arm

The *move_arm_simple_action* node (Figure 4.27, centre) receives an action goal containing an arm goal, defined by many parameters. For instance the goal position for the arm's

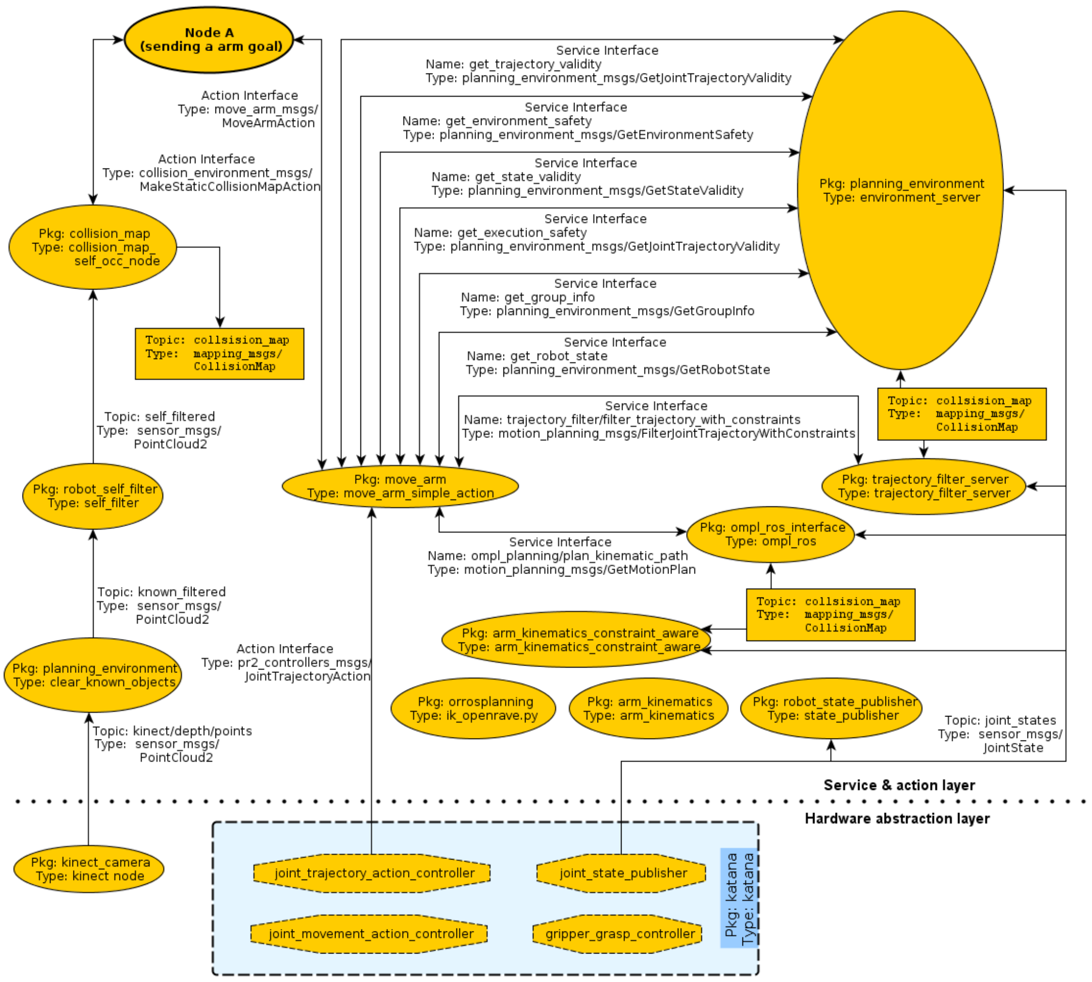

Node A (sending a arm goal)
Action Interface Type: move_arm_msgs/ MoveArmAction
Action Interface Type: collision_environment_msgs/ MakeStaticCollisionMapAction
Pkg: collision_map Type: collision_map_ self_occ_node
Topic: collsision_map Type: mapping_msgs/ CollisionMap
Topic: self_filtered Type: sensor_msgs/ PointCloud2
Pkg: robot_self_filter Type: self_filter
Topic: known_filtered Type: sensor_msgs/ PointCloud2
Pkg: planning_environment Type: clear_known_objects
Topic: kinect/depth/points Type: sensor_msgs/ PointCloud2
Pkg: kinect_camera Type: kinect node
Service Interface Name: get_trajectory_validity Type: planning_environment_msgs/GetJointTrajectoryValidity
Service Interface Name: get_environment_safety Type: planning_environment_msgs/GetEnvironmentSafety
Service Interface Name: get_state_validity Type: planning_environment_msgs/GetStateValidity
Service Interface Name: get_execution_safety Type: planning_environment_msgs/GetJointTrajectoryValidity
Service Interface Name: get_group_info Type: planning_environment_msgs/GetGroupInfo
Service Interface Name: get_robot_state Type: planning_environment_msgs/GetRobotState
Service Interface Name: trajectory_filter/filter_trajectory_with_constraints Type: motion_planning_msgs/FilterJointTrajectoryWithConstraints
Pkg: planning_environment Type: environment_server
Topic: collsision_map Type: mapping_msgs/ CollisionMap
Pkg: trajectory_filter_server Type: trajectory_filter_server
Pkg: move_arm Type: move_arm_simple_action
Service Interface Name: ompl_planning/plan_kinematic_path Type: motion_planning_msgs/GetMotionPlan
Pkg: ompl_ros_interface Type: ompl_ros
Topic: collsision_map Type: mapping_msgs/ CollisionMap
Pkg: arm_kinematics_constraint_aware Type: arm_kinematics_constraint_aware
Action Interface Type: pr2_controllers_msgs/ JointTrajectoryAction
Pkg: orrosplanning Type: ik_openrave.py
Pkg: arm_kinematics Type: arm_kinematics
Pkg: robot_state_publisher Type: state_publisher
Topic: joint_states Type: sensor_msgs/ JointState
Service & action layer
Hardware abstraction layer
joint_trajectory_action_controller
joint_state_publisher
joint_movement_action_controller
gripper_grasp_controller
Pkg: katana Type: katana

Figure 4.27.: Arm navigation configuration (simplified); oval shapes represent ROS nodes, rectangles are placeholders for topics, the *katana* group on the bottom combines different functionalities in one node for technical reasons, arrows represent communication paths between nodes (message topics, service- and action- interfaces)

end effector, which usually will be the most important one, but other constrains can also be given like special orientations for robot links during the motion or at the goal. When receiving a new goal this node handles all required steps for moving the arm to the desired goal by calling the right modules like inverse kinmatics, motion planning and so on in the right order. The modules for the several steps (environment server, trajectory filter, etc.) are "connected" to the node by setting the corresponding ROS parameters defining the according service and action interfaces for communication. The *move_arm_simple_action* node's internal sequence, executed for each received goal, is described more detailed in subsubsection 4.3.5.3.

Environment server

The environment server is the central authority for checking of states and trajectories for collisions as well as constraint or joint limit violations. Therefore it holds the robot model from an URDF description and information about other known objects within the environment, uses joint information to get the robot's actual state and uses range sensor data received in form of collision maps. The functionality of the node is offered via different services as shown in Figure 4.27. Most important are the services for checking robot states (*get_state_validity* service) and trajectories (*get_trajectory_validity* service). These services can be used to check if the contained robot state, or in case of a trajectory all contained intermediate states, mostly defined by their joint angles and velocities, are valid. In this context *"valid"* means that:

- the arm does not collide with the environment or with itself
- the state does not violate joint, velocities or acceleration limits
- the trajectory does not violate path or goal constraints like a special orientation of the end effector during movement, etc.

Such constraints can be set via another service called *set_constraints* which is not shown in Figure 4.27 because it was not used during this work. Some of the other services, important for arm navigation, are mentioned in subsubsection 4.3.5.3.

Inverse kinematics

One of the first steps in planning for arm movements (see also algorithm 5) is the calculation of joint angles for given poses which is called *inverse kinematics"*. Therefore the arm navigation stack uses the *arm_ik* service to this problem. Different modules providing this service are available. Some of them are more general and provide IK solving for any arm by simply using its URDF description, like those in the ROS package *arm_kinematics.* Others are specialized for a specific type of arm, like the Katana robot arm or those from PR2, the research robot from *"Willow Garage"*. IK modules can also be categorised in *constraint aware* and *none constrained aware* IK solvers also having different service interfaces. *Constraint aware* means that additional constraints, like special link and end effector orientations for instance, can be given, that have to be considered when calculating the IK. A ROS module for generic *constraint aware* kinematics based on the "Kinematics and Dynamics Library" (KDL) is provided by the package *arm_kinematics_constraint_aware* contained in the ROS *kinematics* stack. The KDL is part of the ***Orocos*** (Open Robot Control Software) project described in [27]. The KDL is providing a powerful framework for modelling and solving kinematic problems.

As reported later on in subsubsection 4.3.5.2, this part of the arm navigation stack does not work in the given configuration which was one of the major problems during this work.

Motion planner

For finding the actually needed movements to bring the arm from its starting pose to the goal pose, the motion planning problem has to be solved. The problem is to find a collision free path to the goal. This is called path planning, already mentioned in section 2.3. Additionally appropriate motions have to be found, bringing the arm collision free from one intermediate configuration to another.

To achieve this, the motion planner node is used. For the used arm navigation configuration this is the *ompl_ros* node, also shown in Figure 4.27. This node offers a service to receive a motion plan request containing goal and possible path constraints. The request can also contain a start configuration and additional information about the workspace to simulate movements from a future pose for instance. Also contained are information about allowed contacts and allowed collisions which is important if an object shell be touched for grasping. A motion planner node can offer more than one planner by including them like a plugin. Different planners are selectable via the request message. In case of the used motion planner configuration the *ompl_ros* node offers two planners from the Open Motion Planning Library (OMPL) called *SBL* (see [25]) and *LBKPIECE* (see [2]) with *SBL* used by default. For the planning process the node uses the URDF robot model, published joint states and the collision map, for finding the *free configuration space.*

If a solution is found, the response contains a full trajectory consisting of a set of trajectory points. Each trajectory point is defined by a time-stamp, a joint angle, velocity and acceleration for all joints. The given time-stamp defines when the point should be reached relative to the trajectory starting time.

Trajectory filter

As the name already indicates, the *trajectory_filtering_server* node provides a filter chain for filtering the trajectory returned by the motion planner. This is necessary to satisfy the special needs of the actually used arm and to get smoother motions. The filter chain is configurable via the ROS parameter server, defining the filter plugins to be used as well as their order.

For the Katana arm two filters are used in the standard configuration. The first one called *"CubicSplineShortCutterFilterJointTrajectoryWithConstraints"* is used to smooth the trajectory by interpolating extra states if needed and deleting states if a short cut is possible. The approach is similar to those described in [1]. Following the comments in the configuration, it is important that the "discretization" parameter of the filter is not set to high because otherwise the filter will generate to much extra trajectory segments. The second filter is of type
"katana_trajectory_filter/KatanaTrajectoryFilterFilterJointTrajectoryWithConstraints"
having the only purpose of removing the smallest segments of the trajectory until only 16 are left. This special filter is needed because the trajectory controller for the Katana arm can not process trajectories with more than about 16 trajectory segments.

Katana node

The *katana* node, on the bottom of Figure 4.27 visualized as dashed rectangle, is a collection of four functionalities which are implemented together in one node because only one process can connect to the Katana's control box. This node and the *kinect_node* (Figure 4.27, bottom left) are part of the *"Hardware abstraction"* layer while the rest of the nodes belong to the *"Arm navigation"* module in the *"Services & actions"* layer (see Figure 4.4). The *katana* node is specially designed for the Katana robot arms to deal with their special needs. One important thing is the fact that the Katana control interface does not offer a comfortable way to execute trajectories as described later in this paragraph.

The *joint_movement_action controller* can be used to move one or more joints to specific angles. The *gripper_grasp_controller* can be used to open or close the gripper or to move it to an arbitrary state between. Due to the fact that both functionalities are not used during this work, they are not described in more detail.

The *joint_state_publisher* has to fulfil the simple but important task of regularly fetching joint angle information from the arm's control box and publishing it on the according message topic. This data is then used by other nodes, the environment server for instance, to update their internal robot model according to the new joint states. If more than one node is publishing joint states, it can be necessary to use an additional node to merge the different information into one message published on a new topic which is then passed into all nodes requiring a complete set of joint states to update the robot's actual configuration. In the used robot configuration the *mechatronicsContr* node (see subsubsection 4.3.3.2) of the *"Container manipulation"* module is such an additional joint state publisher.

Finally there is the *joint_trajectory_action_controller*, offering an action interface to receive trajectories for execution. When receiving a new trajectory, the *joint_trajectory_action controller* is responsible for sending the right commands at the right time to the control box to exactly follow the planned trajectory. This is a special problem with the Katana arm because it does not offer a simple way to send a complete trajectory. It is also not possible to send a new set of goal joint angles and joint velocities to reach the next intermediate configuration. Instead the joint velocity limits have to be set and only the goal joint angles can be given. This is problematic as it can not be guaranteed that the joints will move on the set velocity limit because the control box does its own planning and smoothing for the motions. Therefore it can not be guaranteed that the arm will exactly follow the trajectory which could cause unforeseeable events. For this reason greater safety distances should be set for planning.

4.3.5.2. Problems with the arm navigation

Due to the poor documentation of the used stacks, packages and nodes some problems have not been resolved. The most important problem is the fact that with the given set up (arm with only 5 DOF) it is not possible to execute *pose goals*. This is a great problem for the item manipulation process as therefore a full pose (position and orientation) for the end effector has to be defined in order to grasp the item safely. During some early tests it appeared that the problem is in the IK (Inverse Kinematic) step (see subsubsection 4.3.5.3, step 2). the received action result always reported that no solution could be found. As to see in Figure 4.27 three different inverse kinematic nodes are launched but it could not be discovered which one, *ik_openrave.py*, *arm_kinematics* or *arm_kinematics_constraint_aware*, is actually called. However, the displayed error messages suggest that it is the *arm_kinematics_constraint_aware* node. After several unsuccessful attempts finding a solution for this problem a workaround was developed. The workaround shifts the inverse kinematic step to the calling node. This means that the node sending the arm navigation goal calculates the IK and only sends a *joint goal* instead of a *pose goal*. The whole workaround will be described in subsection 4.3.7.

Another unsolved problem is collision avoidance for objects attached to the robot arm which should be easy according to a ROS tutorial about this problem (*"Attaching objects to the robot's body"*, `http://www.ros.org/wiki/motion_planning_environment/Tutorials/Attaching%20objects%20to%20the%20robot%27s%20body`, 2011). Tests following the tutorial did not lead to success. Either absolutely nothing happened, or in some few cases the *environment server* received the objects but following movements produced fatal collisions anyway. Therefore other approaches are needed to avoid collisions for attached objects.

4.3.5.3. Arm movement sequence

As shown in Figure 4.27 everything is triggered by the node *"Node A"* displayed top left. At the beginning *"Node A"* calls the node *collision_map_self_occ_node* which then creates and publishes a new static collision map. This collision map represents parts of the environment, which are sensed by the Kinect camera but not included in the robot's URDF model. For the prototype the container fetched from the shelf is such an environment part.

Now the actual movement is triggered by sending a goal via the *MoveArmAction* interface to the *move_arm* node which executes the steps shown in algorithm 5.

The state validity check on line 1 of the presented algorithm is performed via the *get_state_validity* service provided by the *environment_server*. A state is valid if the arm is not in collision with an environment object or with itself. The *arm_ik* service interface, used for the conversion mentioned in line 3, is not displayed in Figure 4.27 because the actual connection could not be identified unambiguously. The underlying problem is described in more detail in subsubsection 4.3.5.2. Environment safety checks (line 6) are performed via a service call to the *environment_server*'s *get_environment_safety* service which checks whether the environment is safe for operation or it is not. Reasons for an unsafe environment could for instance be outdated sensor information. The motion planning service, to be called in line 7, is defined in the goal message via the field *planner_id*. In the used arm navigation configuration the node providing this service is the *ompl_ros* node. For the trajectory check in line 9 the *environment_server*'s *get_trajectory_validity* service is used. The execution safety check (line 19) is done by calling the *environment_server*'s *get_execution_safety* service. If one of the checks or service calls in lines 1, 3 and 6 fail, the goal will be aborted. A failure in lines 6, 7, 8 and 9 in contrast would only lead to a planning restart (line 1) as long as the maximum number of planning attempts is not reached. In this case the goal will be aborted too. The maximum number of planning attempts, also used in line 24, is defined in the goal message field *num_planning_attempts*.

```
1  Check actual robot's state (joint state) validity
2  if received goal is a pose goal then
3      Convert goal into a joint goal via arm_ik service
4  end
5  Check goal validity via the get_state_validity service
6  Check environment safety
7  Call planning service to get a motion plan (trajectory)
8  Check trajectories end position matches goal position via get_state_validity service
9  Check if trajectory is collision free over the full path and satisfies path constraints
10 Call trajectory_filter service to get a filtered version of the trajectory
11 if got filtered trajectory then
12     check filtered trajectory like in lines 8 and 9
13 else
14     continue with unfiltered trajectory
15 end
16 Send trajectory to the joint_trajectory_action_controller for execution
17 // monitor loop
18 while joint_trajectory_action_controller not finished trajectory execution do
19     Check if trajectory execution is safe
20     if True then
21         continue // monitoring loop
22     else
23         stop trajectory by cancelling the joint_trajectory_action_controller's goal
24         if maximum number of planning attempts reached then
25             return goal aborted
26         else
27             GoTo line 1
28         end
29     end
30 end
31 Check if end position is reached via get_state_validity service
32 if True then
33     return goal succeeded
34 else
35     return goal aborted
36 end
```

Algorithm 5: Arm navigation algorithm

4.3.6. Object detection

The intention of this work, already mentioned in section 1.2, is to develop a concept and a prototype robot capable of performing order picking of small boxlike goods which are stored unmixed in containers of known size. One important part in this order picking process, shown in Figure 4.1, is the detection of the items in the container. In the software concept this part is represented by the *object detection* module as shown in Figure 4.4. Due to the used test objects the package is called *medbox_detector*. This package contains the simple *medbox_detector_action_server* node which implements an action server with the action definition given in Listing 4.12. The package also contains the *MedboxDetector* class (see Figure 4.29) which uses a single point cloud as input and a surface detection approach for the actual detection process. One basic assumption for this detection approach is that the dimensions of the storage container as well as the dimensions of the cuboid objects, which shell be detected inside the container, are known and provided to the detector module. A more detailed description about the *MedboxDetector* class is given below in subsubsection 4.3.6.1.

The *medbox_detector_action_server* node is very simple. It only constructs a detector object (class *MedboxDetector*), which does the actual work, starts the action server and waits for incoming goal messages. When a new goal is received the *execute* function is called which triggers a sequence of steps as shown in Figure 4.28.

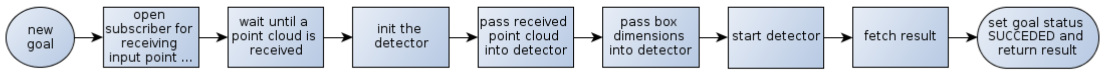

new goal
open subscriber for receiving input point ...
wait until a point cloud is received
init the detector
pass received point cloud into detector
pass box dimensions into detector
start detector
fetch result
set goal status SUCCEDED and return result

Figure 4.28.: Sequence diagram for goal execution of the *medbox_detector_action_server*

As a first step the *execute* function opens a subscriber to the *input_topic*, given by the goal message, and waits until a message arrives. Every 3D sensor, producing a dense point cloud of the container storing the goods, can be used but the message type of the subscribed input topic has to be *sensor_msgs/PointCloud2*. In the next steps the detector is initialized and the input point cloud as well as the given *medbox_dimensions* are passed in. The *medbox_dimensions* array in the goal message has to contain the dimensions of the objects to be detected in the order:

$$\text{[length, width, height] with } length \leq width \leq height$$

The goal message values *container_dimensions* and *container_border_width* are currently not used. Now the detector is launched and after fishing, the detection result is fetched, wrapped into a result message and sent to the calling action client.

4.3.6.1. MedboxDetector class

As already mentioned above, this class (class diagram shown in Figure 4.29) does the actual detection of the objects in the container before grasping.

The used approach has been inspired by the detection and reconstruction method presented in [15] which has already been described briefly in section 3.3. The method in [15] tries to detect a minimum of two or for a better result three surfaces of the cuboid.

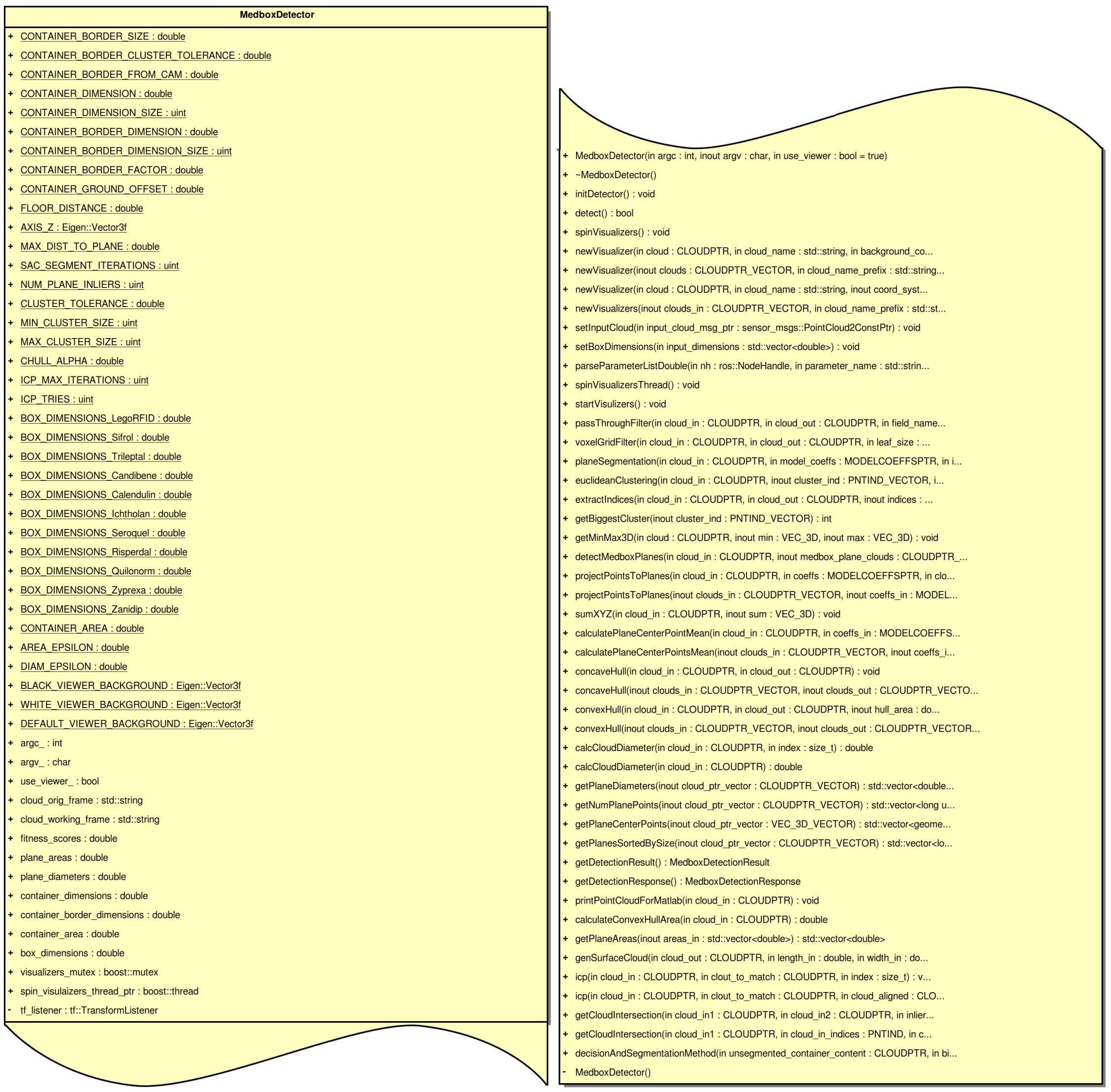

MedboxDetector
+ CONTAINER_BORDER_SIZE : double
+ CONTAINER_BORDER_CLUSTER_TOLERANCE : double
+ CONTAINER_BORDER_FROM_CAM : double
+ CONTAINER_DIMENSION : double
+ CONTAINER_DIMENSION_SIZE : uint
+ CONTAINER_BORDER_DIMENSION : double
+ CONTAINER_BORDER_DIMENSION_SIZE : uint
+ CONTAINER_BORDER_FACTOR : double
+ CONTAINER_GROUND_OFFSET : double
+ FLOOR_DISTANCE : double
+ AXIS_Z : Eigen::Vector3f
+ MAX_DIST_TO_PLANE : double
+ SAC_SEGMENT_ITERATIONS : uint
+ NUM_PLANE_INLIERS : uint
+ CLUSTER_TOLERANCE : double
+ MIN_CLUSTER_SIZE : uint
+ MAX_CLUSTER_SIZE : uint
+ CHULL_ALPHA : double
+ ICP_MAX_ITERATIONS : uint
+ ICP_TRIES : uint
+ BOX_DIMENSIONS_LegoRFID : double
+ BOX_DIMENSIONS_Sifrol : double
+ BOX_DIMENSIONS_Trileptal : double
+ BOX_DIMENSIONS_Candibene : double
+ BOX_DIMENSIONS_Calendulin : double
+ BOX_DIMENSIONS_Ichtholan : double
+ BOX_DIMENSIONS_Seroquel : double
+ BOX_DIMENSIONS_Risperdal : double
+ BOX_DIMENSIONS_Quilonorm : double
+ BOX_DIMENSIONS_Zyprexa : double
+ BOX_DIMENSIONS_Zanidip : double
+ CONTAINER_AREA : double
+ AREA_EPSILON : double
+ DIAM_EPSILON : double
+ BLACK_VIEWER_BACKGROUND : Eigen::Vector3f
+ WHITE_VIEWER_BACKGROUND : Eigen::Vector3f
+ DEFAULT_VIEWER_BACKGROUND : Eigen::Vector3f
+ argc_ : int
+ argv_ : char
+ use_viewer_ : bool
+ cloud_orig_frame : std::string
+ cloud_working_frame : std::string
+ fitness_scores : double
+ plane_areas : double
+ plane_diameters : double
+ container_dimensions : double
+ container_border_dimensions : double
+ container_area : double
+ box_dimensions : double
+ visualizers_mutex : boost::mutex
+ spin_visulaizers_thread_ptr : boost::thread
- tf_listener : tf::TransformListener
+ MedboxDetector(in argc : int, inout argv : char, in use_viewer : bool = true)
+ ~MedboxDetector()
+ initDetector() : void
+ detect() : bool
+ spinVisualizers() : void
+ newVisualizer(in cloud : CLOUDPTR, in cloud_name : std::string, in background_co...
+ newVisualizer(inout clouds : CLOUDPTR_VECTOR, in cloud_name_prefix : std::string...
+ newVisualizer(in cloud : CLOUDPTR, in cloud_name : std::string, inout coord_syst...
+ newVisualizers(inout clouds_in : CLOUDPTR_VECTOR, in cloud_name_prefix : std::st...
+ setInputCloud(in input_cloud_msg_ptr : sensor_msgs::PointCloud2ConstPtr) : void
+ setBoxDimensions(in input_dimensions : std::vector<double>) : void
+ parseParameterListDouble(in nh : ros::NodeHandle, in parameter_name : std::strin...
+ spinVisualizersThread() : void
+ startVisulizers() : void
+ passThroughFilter(in cloud_in : CLOUDPTR, in cloud_out : CLOUDPTR, in field_name...
+ voxelGridFilter(in cloud_in : CLOUDPTR, in cloud_out : CLOUDPTR, in leaf_size : ...
+ planeSegmentation(in cloud_in : CLOUDPTR, in model_coeffs : MODELCOEFFSPTR, in i...
+ euclideanClustering(in cloud_in : CLOUDPTR, inout cluster_ind : PNTIND_VECTOR, i...
+ extractIndices(in cloud_in : CLOUDPTR, in cloud_out : CLOUDPTR, inout indices : ...
+ getBiggestCluster(inout cluster_ind : PNTIND_VECTOR) : int
+ getMinMax3D(in cloud : CLOUDPTR, inout min : VEC_3D, inout max : VEC_3D) : void
+ detectMedboxPlanes(in cloud_in : CLOUDPTR, inout medbox_plane_clouds : CLOUDPTR_...
+ projectPointsToPlanes(in cloud_in : CLOUDPTR, in coeffs : MODELCOEFFSPTR, in clo...
+ projectPointsToPlanes(inout clouds_in : CLOUDPTR_VECTOR, inout coeffs_in : MODEL...
+ sumXYZ(in cloud_in : CLOUDPTR, inout sum : VEC_3D) : void
+ calculatePlaneCenterPointMean(in cloud_in : CLOUDPTR, in coeffs_in : MODELCOEFFS...
+ calculatePlaneCenterPointsMean(inout clouds_in : CLOUDPTR_VECTOR, inout coeffs_i...
+ concaveHull(in cloud_in : CLOUDPTR, in cloud_out : CLOUDPTR) : void
+ concaveHull(inout clouds_in : CLOUDPTR_VECTOR, inout clouds_out : CLOUDPTR_VECTO...
+ convexHull(in cloud_in : CLOUDPTR, in cloud_out : CLOUDPTR, inout hull_area : do...
+ convexHull(inout clouds_in : CLOUDPTR_VECTOR, inout clouds_out : CLOUDPTR_VECTOR...
+ calcCloudDiameter(in cloud_in : CLOUDPTR, in index : size_t) : double
+ calcCloudDiameter(in cloud_in : CLOUDPTR) : double
+ getPlaneDiameters(inout cloud_ptr_vector : CLOUDPTR_VECTOR) : std::vector<double...
+ getNumPlanePoints(inout cloud_ptr_vector : CLOUDPTR_VECTOR) : std::vector<long u...
+ getPlaneCenterPoints(inout cloud_ptr_vector : VEC_3D_VECTOR) : std::vector<geome...
+ getPlanesSortedBySize(inout cloud_ptr_vector : CLOUDPTR_VECTOR) : std::vector<lo...
+ getDetectionResult() : MedboxDetectionResult
+ getDetectionResponse() : MedboxDetectionResponse
+ printPointCloudForMatlab(in cloud_in : CLOUDPTR) : void
+ calculateConvexHullArea(in cloud_in : CLOUDPTR) : double
+ getPlaneAreas(inout areas_in : std::vector<double>) : std::vector<double>
+ genSurfaceCloud(in cloud_out : CLOUDPTR, in length_in : double, in width_in : do...
+ icp(in cloud_in : CLOUDPTR, in clout_to_match : CLOUDPTR, in index : size_t) : v...
+ icp(in cloud_in : CLOUDPTR, in clout_to_match : CLOUDPTR, in cloud_aligned : CLO...
+ getCloudIntersection(in cloud_in1 : CLOUDPTR, in cloud_in2 : CLOUDPTR, in inlier...
+ getCloudIntersection(in cloud_in1 : CLOUDPTR, in cloud_in_indices : PNTIND, in c...
+ decisionAndSegmentationMethod(in unsegmented_container_content : CLOUDPTR, in bi...
- MedboxDetector()

Figure 4.29.: Class diagram of the *MedboxDetector* class used for the *medbox_detector_action_server* node

```
# MedboxDetection.action
# goal definition
string input_topic
float64[] medbox_dimensions
float64[] container_dimensions
float64 container_border_width
---
# result definition
geometry_msgs/PointStamped[] plane_center_points
float64[] plane_diameters
float64[] plane_areas
uint64[] num_plane_points
uint64[] sorted_by_cloudsize
---
# feedback
```

Listing 4.12: ROS action for box detection

Using these surface it then tries to reconstruct the real object including its dimensions. In opposite, the approach in this work is to detect only one surface, or more precisely the largest surface, of an object and to use the additional given information about the object's size to decide if the detection is successful or not. The reason for only detecting a single surface is the fact that only one single sensor is used and mounted straight above the scene. This setting causes that in most cases only one surface is fully visible and represented by a sufficient number of points in the resulting point cloud. The largest surface is used for detection because the greater dimension should make the detection easier and more reliable. Furthermore larger surfaces are a much better contact area for the following grasping step with the vacuum gripper. The implementation uses functionality from the PCL (Point Cloud Library) for all necessary point cloud manipulation and analysis steps. For compatibility reasons the ROS Diamondback built in version of the PCL (version 0.10.0) is used. In the following the used approach and the implemented functions are described in detail.

The fist step necessary for the detection after creating and initializing the detector object is setting the input point cloud via the *setInputCloud(...)* method. Given a ROS point cloud message (type *sensor_msgs/PointCloud2*), this method first converts it into the PCL point cloud format. Then the ROS transform functionality is used to bring the point cloud into the coordinate frame which is given by the ROS parameter *"cloud_working_frame"* and saved for later use by the *detect()* method. All following steps are performed in this coordinate frame defined by the *"cloud_working_frame"* parameter. To allow for a valid detection of the storage container and its content, this frame has to be oriented and placed in a manner that the containers bottom is in the frame's x-y plane and the container is aligned parallel to the its axes. Figure 4.30 shows an example.

The left column shows RGB images of the example scene. The middle column shows different views of the original (unmodified) input point cloud coming from a Microsoft

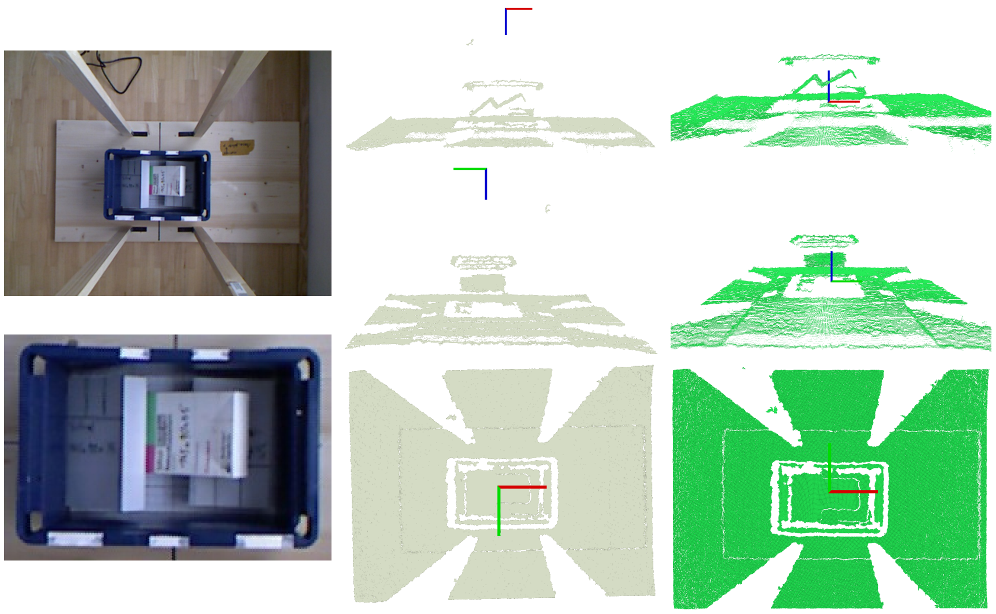


Figure 4.30.: RGB and point cloud images for detection; *Left:* RGB images of the scene; *Centre:* original point cloud; *Right:* point cloud after coordinate transformation; (coordinate system [red, green, blue] → [x, y, z])

Kinect camera as used during this work. The right column shows the same point cloud but transformed from the Kinects *"/kinect_depth"* frame into the frame defined by the *"cloud_working_frame"* parameter.

The next step is setting the dimensions of the box (meaning the cuboid objects inside the container) to be detected. This is done via the *setBoxDimensions(...)* method requiring the box dimensions in the form:

$$\text{[length, width, height] with } length \leq width \leq height$$

Alternatively the ROS parameter *"BOX_DIMENSIONS"* can be used which is read during the detectors initialization via the *initDetector()* method.

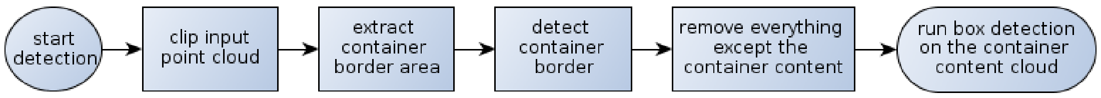

start detection
clip input point cloud
extract container border area
detect container border
remove everything except the container content
run box detection on the container content cloud

Figure 4.31.: Sequence diagram of preparing steps before performing the actual box detection

Now the detection can be started by calling the detectors *detect()* method. This function uses the point cloud previously saved by the *setInputCloud(...)* method (see right column of Figure 4.30) as input for a couple of preparing steps before the actual box detection is performed. Figure 4.31 shows a sequence diagram of these preparation steps.

At the beginning the point cloud is clipped according to the values given by the ROS parameter *"CONTAINER_AREA"* which defines an area in the *"cloud_working_frame"* with its borders at:

$$[x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}]$$

All points outside this area are clipped, using the PCL's *passThroughFilter* functionality, to save computational resources and to facilitate the detection of the container itself which is the next step. The clipped version of the cloud from the example scene mentioned above is shown in the left column of Figure 4.32.

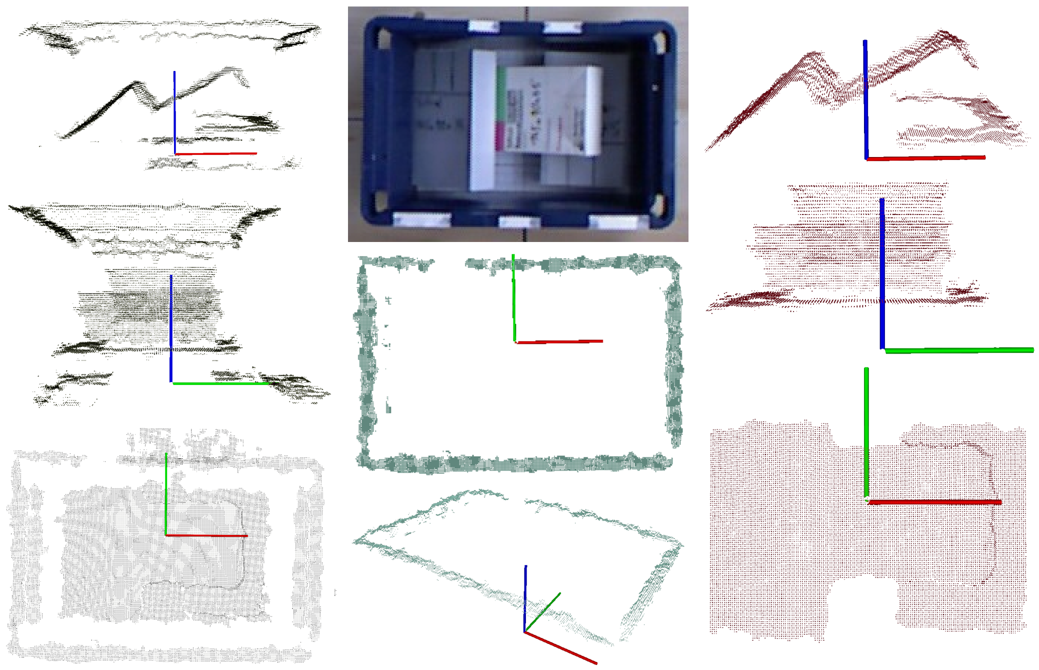


Figure 4.32.: Point cloud images of detection preparation steps
Left: input point cloud after clipping; *Centre:* real world image and the extracted container border point cloud; *Right:* extracted container content; (coordinate system [red, green, blue] → [x, y, z])

The container detection now tries to detect the containers top border. Therefore an area between 0.8 and 1.2 times the containers height on the z-axis is extracted. The height is given via the ROS *"CONTAINER_DIMENSION"* parameter defining the containers dimensions along the axes of the *"cloud_working_frame"*. Then PCL plane segmentation and euclidean clustering, as described in subsection 2.2.2 and subsection 2.2.3, are used to segment the containers top border as shown in the middle column of Figure 4.32. It is assumed that the biggest cluster obtained by the euclidean clustering represents the containers top border. The extreme points of this cluster are used as representation of the containers top border. Using this information and the ROS *"CONTAINER_BORDER_DIMENSIONS"* parameter, defining the border respectively bottom thickness along the x-, y- and z-axis, the points belonging to the container are removed using the PCL's *passThroughFilter* so that the remaining points are only those representing the containers content (see Figure 4.32, right column). This is the input cloud, from now called *container content cloud*, for the actual box detection process which is implemented in the *detectMedboxPlanes(...)* method which is called inside the detectors *detect()* method.

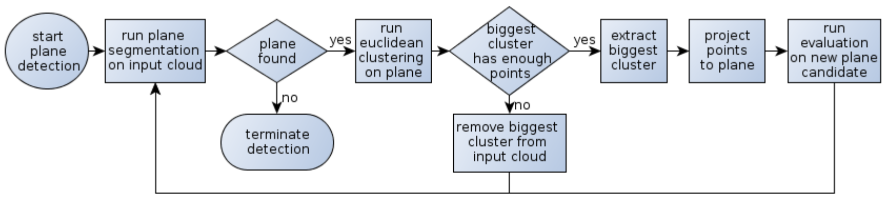

start plane detection
run plane segmentation on input cloud
plane found
yes
no
terminate detection
run euclidean clustering on plane
biggest cluster has enough points
yes
no
remove biggest cluster from input cloud
extract biggest cluster
project points to plane
run evaluation on new plane candidate

Figure 4.33.: Sequence diagram of the box detection loop

In the *detectMedboxPlanes(...)* method the steps shown in Figure 4.33 are executed until no new plane can be segmented from the *container content cloud* which is also the first step in the loop. It is implemented using the PCL plane segmentation (see subsection 2.2.2) which returns the coefficients n_x, n_y, n_z, d of the segmented plane, as defined by the *Hessian Normal form*, and a set of indices referencing the plane inlier points. For more details about the *Hessian Normal form* have a look at [7] or:

`http://mathworld.wolfram.com/HessianNormalForm.html`, 2012

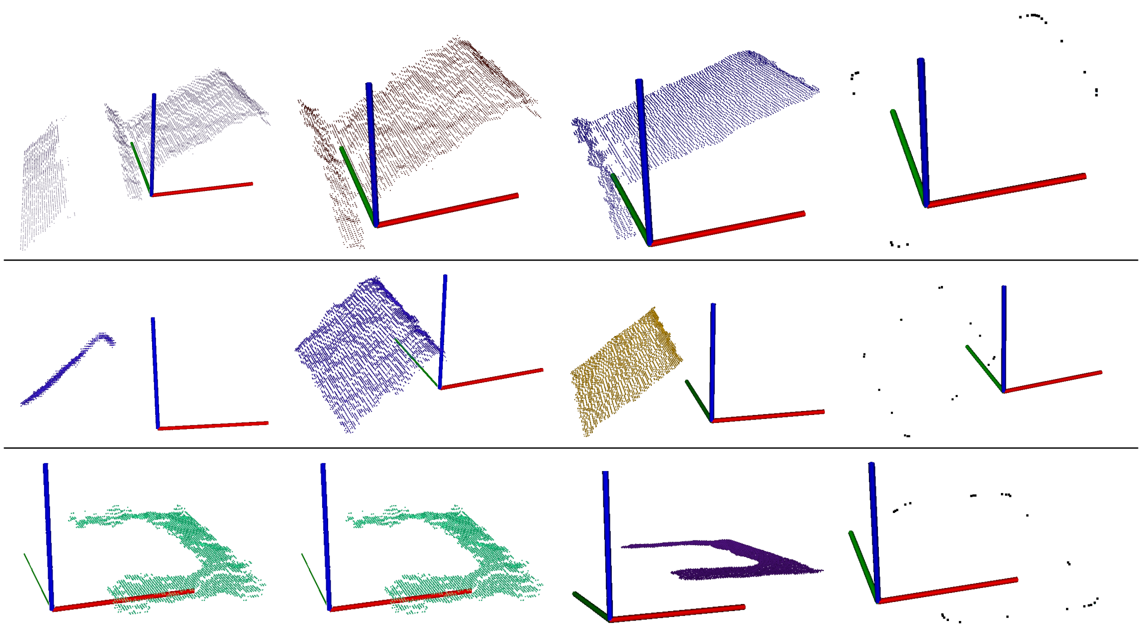


Figure 4.34.: Point cloud images of different surface detection steps; each row shows a different surface belonging to one of three different boxes;
first column: plane segmentation result; *second column:* biggest cluster, used as potential detection candidate; *third column:* cluster projected into the plane; *fourth column:* convex hull of projected cluster; (coordinate system [red, green, blue] → [x, y, z])

For the plane segmentation only points within a given maximum distance to the plane are taken into account and only planes with more inlier points than a given threshold are accepted. The maximum distance is in the range of the Kinects measurement noise which is about $\pm 5\,mm$. For the number of inliers a threshold value of hundred points turned out to be a good choice. The first column in Figure 4.34 shows the segmented planes for the the three box surfaces from the example. If a plane with enough points has been segmented these segment is clustered using the PCL's euclidean clustering (see subsection 2.2.3) to remove points, accidentally lying in the same plane but not belonging to the desired surface. So only more or less fully visible surfaces will be taken into account. As a result the clustering returns all clusters with more than hundred points and the biggest cluster is chosen as a possible detection. The second column in Figure 4.34 shows these clusters corresponding to the segmented planes from the first column in this figure. If this biggest cluster is smaller than the threshold of hundred points its points are removed from the container content cloud and the loop is continued, otherwise its points are extracted (copied) into a new cloud and projected into the segmented plane using the functionality of the PCL's *ProjectInliers* class which is shown in the third column of Figure 4.34. This projected cloud is now called *surface candidate cloud* and is the input cloud for the *decisionAndSegmentationMethod(...)* which evaluates whether the detection is valid or not. A sequence diagram of the steps in the *decisionAndSegmentationMethod(...)* is shown in Figure 4.35.

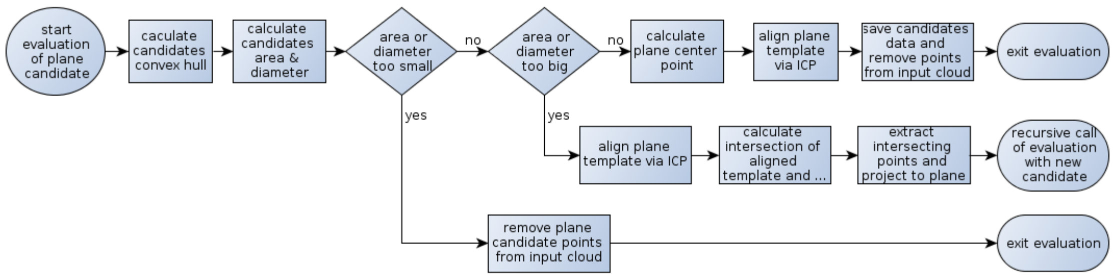

start evaluation of plane candidate
caculate candidates convex hull
calculate candidates area & diameter
area or diameter too small
no
yes
area or diameter too big
no
yes
calculate plane center point
align plane template via ICP
save candidates data and remove points from input cloud
exit evaluation
align plane template via ICP
calculate intersection of aligned template and ...
extract intersecting points and project to plane
recursive call of evaluation with new candidate
remove plane candidate points from input cloud
exit evaluation

Figure 4.35.: Sequence diagram of the box detection evaluation process

The *decisionAndSegmentationMethod(...)* starts with the construction of the convex hull of the given *surface candidate cloud* which is used for evaluating the detection. The convex hull is constructed using the PCL's *ConvexHull* class which returns a point cloud containing the points defining the hull. For the given example these hulls are shown in the fourth column in Figure 4.34.

The convex hull is then used to find the diameter, or rather the diagonal, of the cloud which in this special case is simply defined as the greatest distance between points in the hull. The convex hulls area is also calculated. Therefore the hull is triangulated, which is easy as it was found out that the returned points are already sequentially ordered around the hull. Then the triangle areas are calculated and summed up. The area of a triangle given by its vertices A, B, C can be simply calculated via the following formula:

$$\frac{1}{2}|\vec{AB} \times \vec{AC}|$$

Using the dimension information about the boxes which shell be detected the reference values for diameter and area and their ratios ($hull\,value/reference\,value$) are calculated and used for evaluating the detection.

If the calculated diameter or area ratios are less than $1-\epsilon$, with $\epsilon = 0.15$ as known good value, the candidate is too small. Its points are removed from the *container content cloud* and the whole process is continued with the plane segmentation step shown in Figure 4.33.

In the case that one or both ratios exceed $1+\epsilon$ the candidate is too big which is usually the case if the euclidean clustering step was not able to separate the points belonging to the surface from the rest of the points accidentally lying in the segmented plane. In this case the PCL's ICP module, described in subsection 2.2.4, is used to try a separation.

The ICP module is used to align a surface template to the *surface candidate cloud* and the intersection of the aligned template with the *container content cloud* is calculated using the PCL's nearest neighbour search (see subsection 2.2.3). Hereby for each point of the aligned template cloud the nearest neighbours within a radius of $10mm$ are assumed to belong to the intersection. The intersecting points from the *container content cloud* are extracted (copied) and projected into the plane as before and now used as new candidate for a recursive call to the *decisionAndSegmentationMethod(...)*.

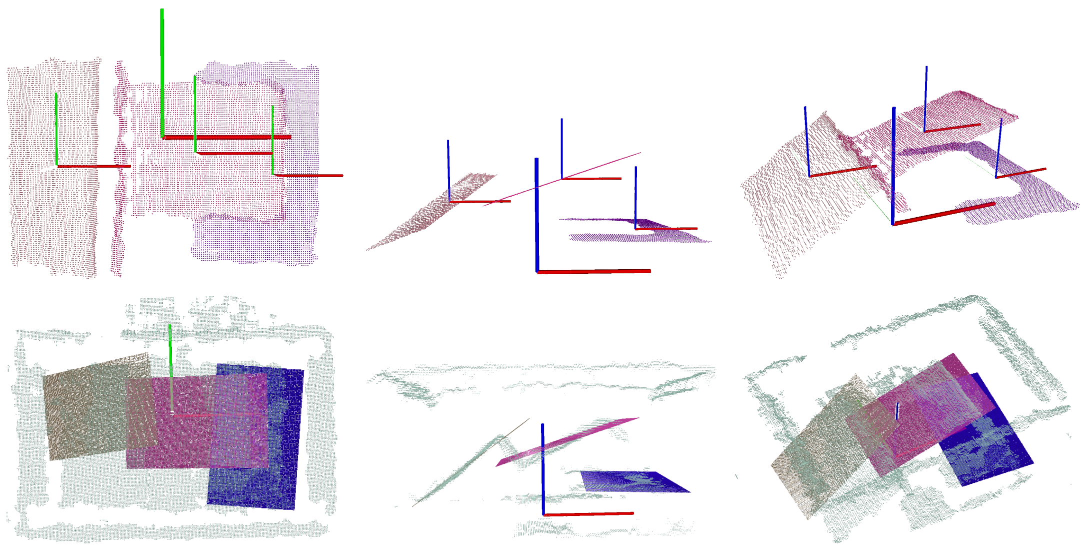


Figure 4.36.: Point cloud images of the detection result; *first row:* different views showing the detected surface points projected into their planes and their center points; *second row:* the clipped input cloud and the aligned surface templates; (coordinate system [red, green, blue] → [x, y, z])

If the candidates diameter and area ratios are in the expected range for a valid detection

$$1 - \epsilon \leq ratio \leq 1 + epsilon$$

its points are removed from the *container content cloud*. The surface center point, which is later used as grasp point, is calculated by simply taking the coordinates mean of all points form the *surface candidate cloud*. Then ICP is used to align a surface template to get the surfaces pose. The idea for this step was to use the surfaces pose for attaching a collision model of the box to the robot arm. This is currently not implemented as this feature does not work for a unknown reason as already mentioned in subsubsection 4.3.5.2. So in the current software this step is actually unnecessary. The first row of Figure 4.36 shows different views of the detected surfaces and their center points while the second row shows the clipped input cloud and the aligned templates.

As a last step, before continuing with the next loop pass, all the interesting data concerning this surface are saved. Among some other less important things, these are:

- the projected version of the *surface candidate cloud*
- the plane coefficients of the segmented plane
- the calculated surface center point
- the resulting transformation returned from the ICP alignment
- the aligned template point cloud
- the convex hull point cloud
- the calculated diameter and area values of the surface

The final step for finishing the goal execution as shown in Figure 4.28 is fetching the result message from the detector and sending it to the action client. For fetching the result message the detector offers the *getDetectionResult()* method which fills and returns the result message which then can be directly sent to the action client. The result message (definition in Listing 4.12) contains arrays with the center points, diameters, areas and the number of points of the detected surfaces. The *sorted_by_cloudsize* array contains the indices of the detected surfaces ordered according to their number of points. The basic assumption is that clouds with more points are detected more reliably and therefore are preferred in the following grasping process.

4.3.7. Item manipulation

The item manipulation step, in the work cycle overview (see Figure 4.1) represented by the block *"transfer item to order bin"*, is implemented in the *"Item manipulation"* module in the *"Services & actions"* layer shown Figure 4.4. It uses the result of the *"Object detection"* module (described in subsection 4.3.6), the *"Arm navigation"* module (see subsection 4.3.5) and the *"Vacuum gripper system"* to grasp one of the items in the storage container and put it into the order bin. An overview of the module with its contained nodes and its connections to other modules is shown in Figure 4.37. It should be mentioned that the shown *"Vacuum gripper system"* module is part of the *"Hardware abstraction"* layer while the shown modules *"Arm Navigation"* and *"Item Manipulation"* are part of the *"Services & actions"* layer.

As it can be seen in Figure 4.37, the implementation of the *"Item manipulation"* module consists of four nodes. The main node is the *katana_medbox_picking_action_server* node, implementing an ROS action server with the action definition shown in Listing 4.13, which triggers all the necessary steps for grasping the item and putting it into the order bin. The three nodes on the right are assisting nodes for the workaround to solve the problem, concerning the inverse kinematics, already mentioned in subsubsection 4.3.5.2. This workaround, described in subsubsection 4.3.7.2, uses the *ik_katana_server.py* node for

```
# MedboxPicking.action
# goal definition
medbox_detector/MedboxDetectionResult detected_boxes
---
# result definition
bool  box_picked
bool  box_placed
bool  back_to_init_pos

uint8 error_code
uint8 NO_ERROR                    =  0
uint8 COL_MAP_ERROR               =  1
uint8 PICKING_INIT_POS_ERROR      =  2
uint8 TOUCH_BOX_ERROR             =  3
uint8 MECH_CONTR_ERROR            =  4
uint8 BOX_NOT_PICKED_ERROR        =  5
uint8 PRE_DROP_POINT_ERROR        =  6
uint8 DROP_POINT_ERROR            =  7
uint8 BOX_LOST_ERROR              =  8
uint8 INIT_POS_ERROR              =  9
uint8 NO_GRASP_POINTS             = 10
uint8 TRANSFORM_ERROR             = 11
---
# feedback
```

Listing 4.13: ROS action for item manipulation

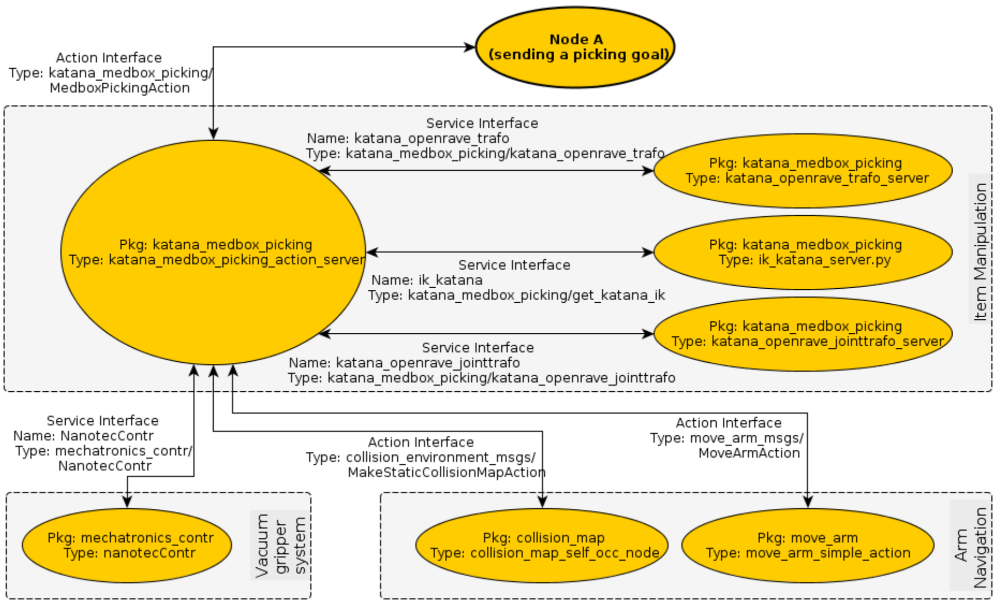

Node A
(sending a picking goal)
Action Interface
Type: katana_medbox_picking/
MedboxPickingAction
Service Interface
Name: katana_openrave_trafo
Type: katana_medbox_picking/katana_openrave_trafo
Pkg: katana_medbox_picking
Type: katana_openrave_trafo_server
Pkg: katana_medbox_picking
Type: katana_medbox_picking_action_server
Pkg: katana_medbox_picking
Type: ik_katana_server.py
Service Interface
Name: ik_katana
Type: katana_medbox_picking/get_katana_ik
Pkg: katana_medbox_picking
Type: katana_openrave_jointtrafo_server
Service Interface
Name: katana_openrave_jointtrafo
Type: katana_medbox_picking/katana_openrave_jointtrafo
Item Manipulation
Service Interface
Name: NanotecContr
Type: mechatronics_contr/
NanotecContr
Action Interface
Type: collision_environment_msgs/
MakeStaticCollisionMapAction
Action Interface
Type: move_arm_msgs/
MoveArmAction
Pkg: mechatronics_contr
Type: nanotecContr
Vacuum gripper system
Pkg: collision_map
Type: collision_map_self_occ_node
Pkg: move_arm
Type: move_arm_simple_action
Arm Navigation

Figure 4.37.: Item manipulation configuration with involved ROS nodes (oval shapes) and their communication paths (service- and action- interfaces); dashed rectangles indicate the correspondence to modules as shown in Figure 4.4

calculating the inverse kinematics via OpenRAVE (Open Robotics Automation Virtual Environment). More information about OpenRAVE can be found at `www.openrave.org`. In the following, the nodes and their implemented approaches are described in detail.

4.3.7.1. katana_medbox_picking_action_server node

This node only creates an object of class *MedboxPicking* and calls its *run()* method which then executes all the steps necessary for item manipulation. As shown in the class diagram (see Figure 4.38) this class in turn holds an object of class *MoveArm* which is created in the constructor and handles the arm movements for the task of item manipulation. In the following these two classes and their methods will be described.

4.3.7.1.1. MoveArm class

As already mentioned, this class handles the arm movements necessary for item manipulation. It also holds the connection to the two action servers from the *"Arm navigation"* module shown in Figure 4.37 which are necessary for executing the actual movements.The connection is established in the classes constructor. It also holds the connection to the three services, in Figure 4.37 shown on the right side inside in the *"Item manipulation"* module, which are part of the inverse kinematics workaround. The class offers, among other things, the methods listed below:

- makeStaticCollisionMap()
- addAllowedContactSpec(...)
- addCollisionOperation(...)

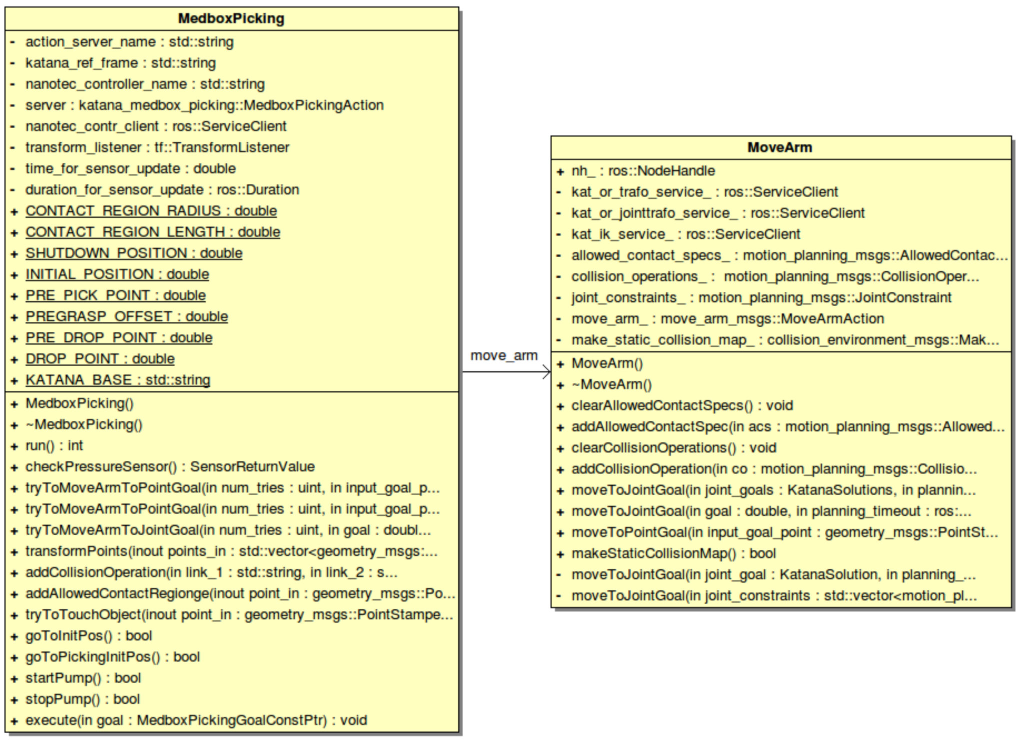

MedboxPicking
- action_server_name : std::string
- katana_ref_frame : std::string
- nanotec_controller_name : std::string
- server : katana_medbox_picking::MedboxPickingAction
- nanotec_contr_client : ros::ServiceClient
- transform_listener : tf::TransformListener
- time_for_sensor_update : double
- duration_for_sensor_update : ros::Duration
+ CONTACT_REGION_RADIUS : double
+ CONTACT_REGION_LENGTH : double
+ SHUTDOWN_POSITION : double
+ INITIAL_POSITION : double
+ PRE_PICK_POINT : double
+ PREGRASP_OFFSET : double
+ PRE_DROP_POINT : double
+ DROP_POINT : double
+ KATANA_BASE : std::string
+ MedboxPicking()
+ ~MedboxPicking()
+ run() : int
+ checkPressureSensor() : SensorReturnValue
+ tryToMoveArmToPointGoal(in num_tries : uint, in input_goal_p...
+ tryToMoveArmToPointGoal(in num_tries : uint, in input_goal_p...
+ tryToMoveArmToJointGoal(in num_tries : uint, in goal : doubl...
+ transformPoints(inout points_in : std::vector<geometry_msgs:...
+ addCollisionOperation(in link_1 : std::string, in link_2 : s...
+ addAllowedContactRegionge(inout point_in : geometry_msgs::Po...
+ tryToTouchObject(inout point_in : geometry_msgs::PointStampe...
+ goToInitPos() : bool
+ goToPickingInitPos() : bool
+ startPump() : bool
+ stopPump() : bool
+ execute(in goal : MedboxPickingGoalConstPtr) : void
move_arm
MoveArm
+ nh_ : ros::NodeHandle
- kat_or_trafo_service_ : ros::ServiceClient
- kat_or_jointtrafo_service_ : ros::ServiceClient
- kat_ik_service_ : ros::ServiceClient
- allowed_contact_specs_ : motion_planning_msgs::AllowedContac...
- collision_operations_ : motion_planning_msgs::CollisionOper...
- joint_constraints_ : motion_planning_msgs::JointConstraint
- move_arm_ : move_arm_msgs::MoveArmAction
- make_static_collision_map_ : collision_environment_msgs::Mak...
+ MoveArm()
+ ~MoveArm()
+ clearAllowedContactSpecs() : void
+ addAllowedContactSpec(in acs : motion_planning_msgs::Allowed...
+ clearCollisionOperations() : void
+ addCollisionOperation(in co : motion_planning_msgs::Collisio...
+ moveToJointGoal(in joint_goals : KatanaSolutions, in plannin...
+ moveToJointGoal(in goal : double, in planning_timeout : ros:...
+ moveToPointGoal(in input_goal_point : geometry_msgs::PointSt...
+ makeStaticCollisionMap() : bool
- moveToJointGoal(in joint_goal : KatanaSolution, in planning_...
- moveToJointGoal(in joint_constraints : std::vector<motion_pl...

Figure 4.38.: Class diagram of the classes *MedboxPicking* and *MoveArm* used for the *katana_medbox_picking_action_server* node

- clearAllowedContactSpecs()
- clearCollisionOperations()
- moveToJointGoal(...)
- moveToPointGoal(...)

The first method is used to trigger an action call to the *collision_map_self_occ_map* node (Figure 4.37, bottom center) to create a new static collision map which then will be used for all following arm planning steps. This is used as the scene for the process of item manipulation is assumed to be static and therefore using this function can ensure that the collision map is constructed when all the important parts of the scene are fully visible by the Kinect camera which is the sensor source for the *collision_map_self_occ_map* node. Important in this context means everything that is not part of the URDF robot model. The most important thing is the container in front of the robot arm, resting on the fork of the container manipulation unit.

The methods *addAllowedContactSpec(...)* and *addCollisionOperation(...)* are used to store *AllowedContactSpecification* and *CollisionOperation* message objects in a list. This list will be added to every goal sent to the *"Arm navigation"* module. As the names of the messages already suggest, they can be used to define regions where collisions between the arm and the environment are allowed. In case of item manipulation this is used to define a region around the grasp point on the box surface where collisions with the gripper are allowed which is necessary as otherwise the final move for picking the box would be detected as invalid during the path planning process.

The following executed methods, *clearAllowedContactSpecs()* and *clearCollisionOperations()*, will clear the two previously mentioned lists so that no collisions will be allowed for the following moves.

With the method *moveToJointGoal(...)* it is possible to send a goal to the *"Arm navigation"* module. This goal message defines the arm's goal pose by a full set of joint angles. This is useful for fixed poses, like the used INIT_POSITION (see paragraph 4.3.7.1.2), as no inverse kinematic has to be solved which reduces the time for the path planning process.

Finally there is the *moveToPointGoal(...)* method. This method is more complex than the others as it takes a single point as input and then makes three additional service calls to solve the inverse kinematics for an overhead grasp pose with the vacuum gripper's tip at this point. The solution from the inverse kinematic step is a joint goal which is then sent to the *"Arm navigation"* module to execute the move. A overhead grasp means that the vacuum gripper is oriented downwards and positioned above the object to be grasped and then lowered until the dedicated grasp point, which in this context is one of the detected surface center points, is reached. An example with typical poses during the grasping process is shown in Figure 4.39. A more detailed description about the used service calls to solve inverse kinematics can be found in subsubsection 4.3.7.2.

All the move methods have the goal as first parameter, followed by a planning time-out and a movement time-out (type *ros::Duration*) which, if exceeded, will lead to a abort of the goal. The last parameter is of type *bool* for disabling the collision monitoring during the movement. This parameter is set to *"False"* by default. This parameter should be used with care because setting to *"True"* involves the risk of collisions between the arm and the environment if the arm does not exactly follow the planned trajectory. In some cases, especially when the arm gets stuck near to some obstacles because the distance has fallen below the threshold, this can help to move the arm away from the obstacle and bring it back to work. The methods return *"True"* if the goal is reached, or rather if the action call to the *"Arm navigation"* module returns *"SUCCEEDED"*, and otherwise they return *"False"*.

4.3.7.1.2. MedboxPicking class

In the *run()* method several ROS parameters are fetched which are necessary for the service connection to the *"Vacuum gripper system"* and one parameter defining the reference frame for *pose goals* during the item manipulation. A list of the parameters with their types and meanings is given in Table 4.1.

Parameter	**Type**	**Description**
PUMP_CONTR_NR	integer	hardware address of the Nanotec device controlling the vacuum pump
PUMP_BIT	integer	command bit referencing the output for the vacuum pump
SENSOR_CONTR_NR	integer	hardware address of the Nanotec device connected to the pressure sensor
SENSOR_BIT	integer	command bit referencing the input for the pressure sensor
SENSOR_UPDATE_TIME	double	defines a number of seconds to wait before checking the pressure sensor of the vacuum gripper system
NANOTEC_CONTR	string	service name for the connection to the *"Vacuum gripper system"*
SERVER_NAME	string	name to use for the action server
KATANA_BASE	string	reference frame for *pose goals*

Table 4.1.: ROS parameters read by the *MedboxPicking* class

The next step in the *run()* method is to establish the service connection to the *nanotecContr* node representing the *"Vacuum gripper system"* (see Figure 4.37) via the *NANOTEC_CONTR* service with the service definition given in Listing 4.5. For more information about this service have a look at subsubsection 4.3.3.1. After that, the robot arm is moved to an initial position (INIT_POSITION, see Figure 4.39, first column) near to its calibration position without collision monitoring. Collision monitoring is disabled to increase the likelihood of reaching the goal but also increasing the danger of collisions with the environment. If this fails notwithstanding five retries the node reports an error and terminates because this initial position is a precondition for the item and container manipulation processes as well as for the box detection process as it guaranties a collision free operation of the container manipulation unit and also that the Object detection sensor has a clear view of the container. By the way it should be mentioned that this position is a stable one. This is good for the case of power loss which otherwise always carries the risk of damage of the robot arm as the Katana arm has no mechanical joint breaks. At the end of the *run()* method the action server, with the name given by the *"SERVER_NAME"* parameter, the action definition given in Listing 4.13 and the classes *execute(...)* method as goal callback, is started.

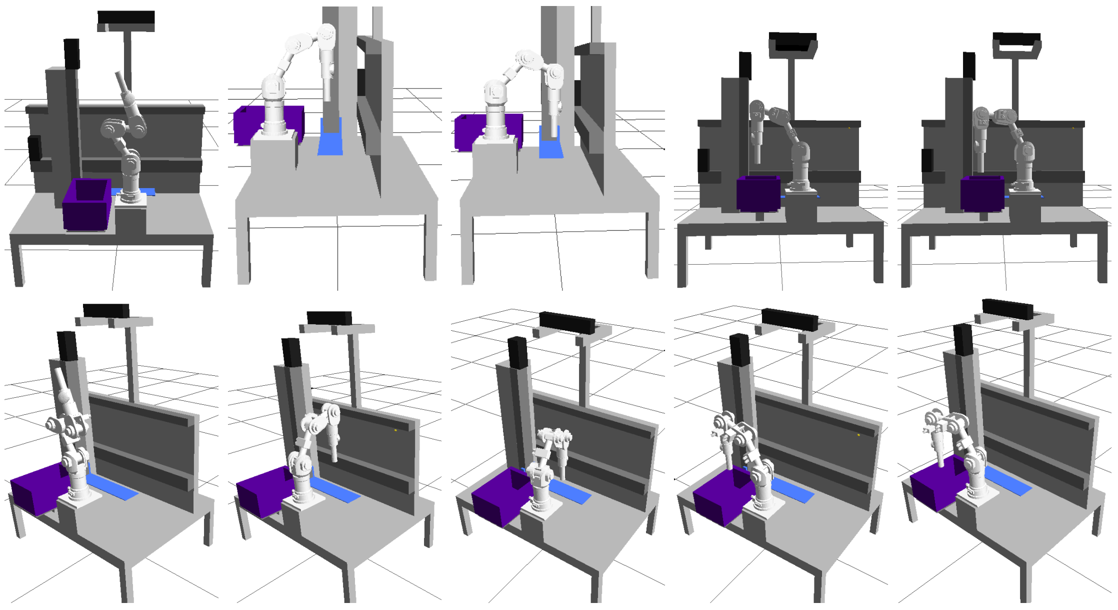


Figure 4.39.: The figures show typical arm poses during the item manipulation process; column by column, left to right: initial position (INIT_POSITION) before grasping; initial point for the overhead grasp (PRE_PICK_POINT); an example for a possible grasp point (GRASP_POINT); pre drop point (PRE_DROP_POINT), drop point (DROP_POINT)

From now on the *execute(...)* method is called whenever a new goal message is received. This method then runs a sequence of actions to pick an item in the container with an overhead grasp and to transfer it to the order bin. As it can be seen in the goal message definition (Listing 4.13) the received goal message contains the result message from the previous detection step. This detection result also includes the detected surface center points which are now used as possible grasp points for the vacuum gripper. The executed sequence contains different steps like arm movements or sending commands to the *"Vacuum gripper system"* and so on. Because all these steps need to be finished successfully for a successful picking operation the goal is aborted if one of the steps fails. In the result message, which is sent to the calling node, the *error_code* is set to the corresponding value (see Listing 4.13). If the arm has already been moved to a different position than the already mentioned *INIT_POSITION* an attempt is being made to move the arm back to this position for safety reasons. The result's *back_to_init_pos* value is used to report the success or fail of the attempt to the calling node. If the attempt fails it is not safe to move the container manipulation unit which has to be handled in the robot control. There are two more fields in the result. The *box_picked* value indicates if a box has been successfully grasped while the value *box_placed* indicates if the box has been successfully placed inside the order bin. This information can be from interest for fault handling implemented in the calling node or at a higher level.

For arm movements the execute method makes use of different class methods:

- goToInitPos()
- goToPickingInitPos()
- tryToTouchObject(...)
- tryToMoveArmToPointGoal(...)

The method *tryToTouchObject(...)* for instance, taking a *geometry_msgs/PointStamped* message as input, will create regions around the goal point where contacts between arm and environment are allowed (*allowed contact regions*), before moving the arm. This is done by using the class methods *addCollisionOperation(...)* and *addAllowedContactRegionge(...)* which in turn are making use of the corresponding methods provided by the *MoveArm* object as described in paragraph 4.3.7.1.2. For moving the arm the methods *goToPickingInitPos()* and *tryToTouchObject(...)*, mentioned in the list above, make use of two similar versions of the *tryToMoveArmToPointGoal(...)* method. Both versions internally use the *MoveArm*'s *moveToPointGoal(...)* method to perform the move and, if the move fails, make a number of retries as given by the first function parameter. Both versions are taking two different time-out parameters which are forwarded to the *MoveArm*'s *moveToPointGoal(...)* method. The only difference is the goal point parameter's data type. One version takes an array of doubles defining the goal point coordinates and a string defining the goal point's coordinate frame and internally generates the *geometry_msgs/PointStamped* message for the *MoveArm*'s *moveToPointGoal(...)* method. The second version directly takes the goal as *geometry_msgs/PointStamped* message and only forwards it to the *MoveArm*'s method.

The *MedboxPicking* class defines three methods for the communication with the *"Vacuum gripper system"*:

- startPump()
- stopPump()
- checkPressureSensor()

These methods use the above mentioned *NANOTEC_CONTR* service to communicate with the *nanotecContr* node (see Figure 4.37) which establishes the connection to the Nanotec control units. The vacuum pump and the pressure sensor are both connected to the I/O ports of control unit two as shown in Figure 4.12. To set the output ports or to read the state of the output and input ports the commands shown Table 4.2 are used. It is only possible to read/write a full 32bit register corresponding to the control units I/Os. Hereby the registers bits 0-5 correspond to the inputs 1-6 and the bits 16 and 17 correspond to output 1 and 2.

Command	**Data type**	**Description**
'Y'	uint, 32bit	Command to set the output register
'ZY'	uint, 32bit	Command to read the I/O register

Table 4.2.: Commads to read/write to the I/Os of the Nanotec control units

To start or stop the vacuum pump a service request, containing the 'Y' command, the value from the *"PUMP_CONTR_NR"* parameter for the *motor_number* field, is made. The contained value is a 32bit unsigned integer with the bit, given by *"PUMP_BIT"* parameter, set to 0/1 to set/reset the corresponding output. The methods will return *"False"* if either the service call returned an error or the received response indicates a problem with the command. To read the pressure sensor the 'ZY' command is sent to the device with the address given by the *"SENSOR_CONTR_NR"* parameter and the response then contains a 32bit value with the *"SENSOR_BIT"* representing the pressure sensors state. If the bit is '1' this means that something has been grasped while '0' means that nothing has been grasped or that the pump is not running (see also subsection 4.2.4). The *checkPressureSensor()* method returns a *SensorReturnValue* which can be *READ_ERROR* if there is a problem with the service call or the command or depending on the pressure sensors state *GRASPED* or *NOT_GRASPED*.

As already mentioned above the *MedboxPicking::execute(...)* method runs a sequence of actions using the described methods to pick an item and to put it into the order bin. At the beginning of the method the goal is checked and in the case of a goal with an empty set of grasp points the goal is aborted, setting the result's *error_code* (see Listing 4.13) to *NO_GRASP_POINTS*. If the goal message is valid the grasp points from the goal message are transformed into the coordinate frame given by the *"KATANA_BASE"* parameter using the *transformPoints(...)* method which uses the ROS transformation system to bring the points into the correct coordinate frame. All the point goals to be used with the *moveToPointGoal(...)* method from the *MoveArm* class are defined in this frame for reasons of simplicity. If the transformation fails, the returned error code is *TRANSFORMATION_ERROR*.

This transformation is followed by an action call to the *collision_map_self_occ_map* node using the *MoveArm* objects *makeStaticCollisionMap()* method to create a new static collision map which then will be used for path planning for the following arm movements. On failure, the goal is set as aborted and the result's *error_code* is set to *COL_MAP_ERROR* which indicates a problem with the collision map.

Now the *goToPickingInitPos()* method is called to move the arm to the *PRE_PICK_-POINT*. The method internally uses the *MoveArm*'s *moveToPointGoal(...)* method which moves the arm in a way that the vacuum gripper is pointing downwards with its end at this given point so that an overhead grasp can be performed. The point is directly in front of the arm, several centimetres above the center of the containers optimal position. Column two in Figure 4.39 shows pictures of this arm pose. On failure the result's *error_code* is set to *PICKING_INIT_POS_ERROR* and the result's *back_to_init_pos* value reports the success or failure of the recovery attempt, implemented in the *goToInitPos()* method. This method uses the *tryToMoveArmToJointGoal(...)* method which is already used in the *run()* method to move the arm back to the *INIT_POSITION*.

The next step is to grasp an item. As grasp point, the surface center point corresponding to the surface with the most point cloud points is used, because more points usually make the detection more reliable. Using this point the *tryToTouchObject(...)* method is called which first adds an allowed contact region around the grasp point and also adds allowed collision operations for the vacuum gripper with objects near this point. Without these regions and allowed collisions, possible collisions between the gripper and for instance the container will be detected during the planning process as the distances possibly will underrun the defined safety distance and this would lead to a planning error. Then the arm will be moved to a pre grasp point a few centimetres above the actual grasp point before the object is actually touched by moving the arm to the grasp point. An example pose for a possible grasp point is shown in the third column of Figure 4.39. On failure, an attempt is made to move the arm back to the *PRE_PICK_POINT*. The corresponding method is called *goToPickingInitPos()*. For a new grasping attempt the next grasp point is chosen from the goal message. If none of the possible grasp points can be reached the goal is aborted, reporting a *TOUCH_BOX_ERROR*.

After starting the pump via the *startPump()* method, possibly reporting a *MECH_-CONTR_ERROR*, the *checkPressureSensor()* method is called. If the method returns a *READ_ERROR* an attempt is made to stop the pump and to bring the arm back to the *INIT_POSITION*. In such cases the result's error code will be *MECH_CONTR_ERROR*, indicating a possible hardware problem with the Nanotec control units. If the *check-PressureSensor()* method returns *NOT_GRASPED*, the same operations will be performed but the result's error code will be *BOX_NOT_PICKED_ERROR*. In contrast to the *MECH_CONTR_ERROR* the *BOX_NOT_PICKED_ERROR* only indicates that no item has been grasped. This may be the case, for example, if the item has moved since the detection, maybe as a consequence of the gripper's touch. In case the *checkPressureSen-sor()* method returns *GRASPED* the result's *box_picked* value is set to *"True"* and the manipulation process will be continued by a new call of the *goToPickingInitPos()* method to bring the arm back to a safe position above the container followed by the deletion of the allowed contact region and the collision operations.

The next steps are movements to a *PRE_DROP_POINT* above the order bin and to the *DROP_POINT* followed by new check of the pressure sensor to be sure that the grasped item was not lost during the movements which would lead to an abort reporting a *BOX_LOST_ERROR*. The *DROP_POINT* is centred, close above the order bin to avoid misplacement or damage of the item when falling into the bin. Smooth placing of the grasped item inside the order bin is actual not possible as there is no information about positions and orientations of already contained items but it is not important for this work because it focuses on non fragile goods. The intermediate movement from the pose above the container to the *PRE_DROP_POINT*, before moving to the actual *DROP_POINT* above the order bin is needed to avoid collisions between the grasped item and the containers because collision avoidance for the carried object does not work. This problem has already been mentioned in subsubsection 4.3.5.2. In case of a successful transfer the vacuum pump is stopped, the result's *box_placed* value is set to *"True"* and the arm is moved back to its *INIT_POSITION*.

If all steps are finished successfully, the action result's *error_code* is set to *NO_ERROR* and the action server's goal state is set to *SUCCEEDED*. This means that an item has been transferred successfully from the storage container to the order bin and that the action server is ready for the next goal.

4.3.7.2. Katana inverse kinematics workaround

As already mentioned repeatedly, the used arm navigation configuration can not plan for *pose goals* due to problems with the computation of the inverse kinematics (IK). For this reason a workaround was developed that shifts the inverse kinematic step to the calling node, which in the case of item manipulation is the *katana_medbox_picking_action_server* node. This node, or more precisely the *MoveArm* class, uses three services to solve the inverse kinematics for the problem of item manipulation so that only a *joint goal* has to be sent to the *"Arm navigation"* module, which then can be executed without the need of inverse kinematic calculations. The workaround consists of three nodes, implementing three services. These nodes are described in the following paragraphs.

4.3.7.2.1. ik_katana_server node

The core of the workaround is the *ik_katana_server.py* node implementing a ROS service named *ik_katana*, with the service definition given in Listing 4.14. It uses the OpenRAVE framework to solve the IK (Inverse Kinematic) for the Katana arm. More precisely, it uses an adaptation of the OpenRAVE *"tutorial_ik5d.py"* example which generates random goal points and orientations for the arm's end-effector and uses a special 5D IK module to find possible configurations for the arm. The first row in Figure 4.40 shows the example environment with some goal poses and corresponding arm configurations. The goals are marked by a line strip between the gripper fingers. Row two of the figure shows the cleared environment with the arm at its origin. This setting is used for the *ik_katana_server.py* node.

```
# get_katana_ik.srv
# request
katana_medbox_picking/KatanaGoal goal_input
---
# response
katana_medbox_picking/KatanaSolutions solutions
```

Listing 4.14: ROS action for box detection

```
# katana_medbox_picking/KatanaGoal
uint64 seq
string frame
float64[6] data
```

Listing 4.15: ROS action for box detection

On start up the node loads the environment file, only containing the Katana's OpenRAVE model, and starts the OpenRAVE planning framework, but in contrast to the example it also starts a ROS service server with the definition from Listing 4.14. The service request contains a *KatanaGoal* message (see Listing 4.15). This message has a data field holding the goal position (pos) and orientation (or) in the following form:

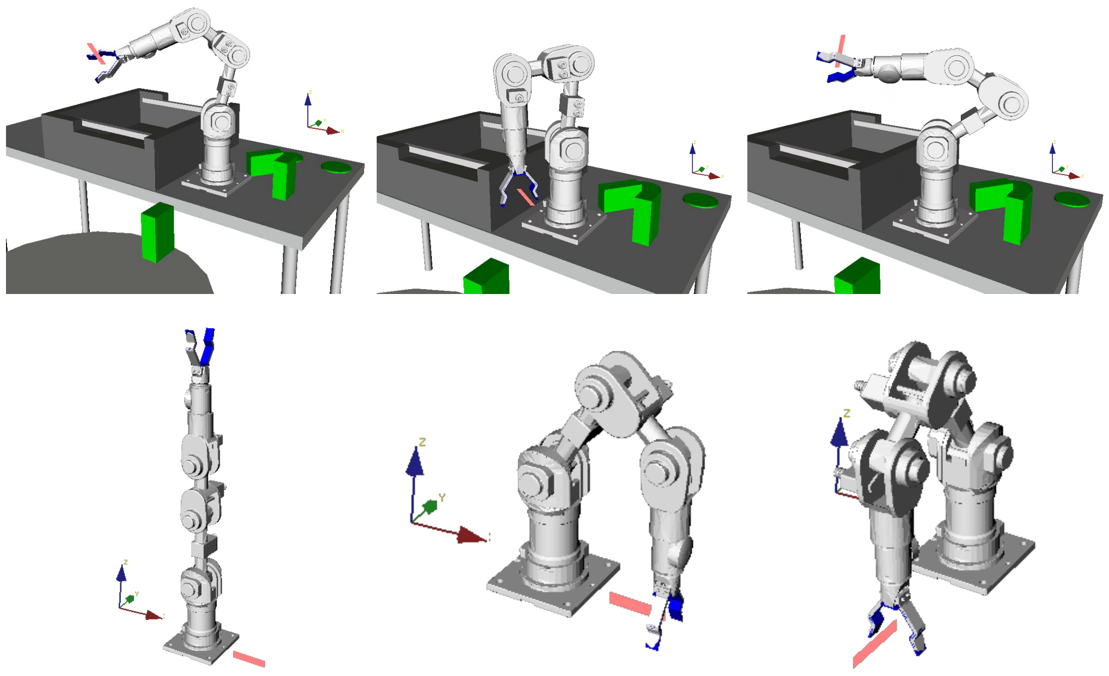


Figure 4.40.: The pictures show different arm goals, marked by a line strip, and configurations resulting from the OpenRAVE IK solutions;
1st line: original OpenRAVE example environment;
2nd line: modified environment as used for the *ik_katana* service;
Bottom left: OpenRAVE initial configuration

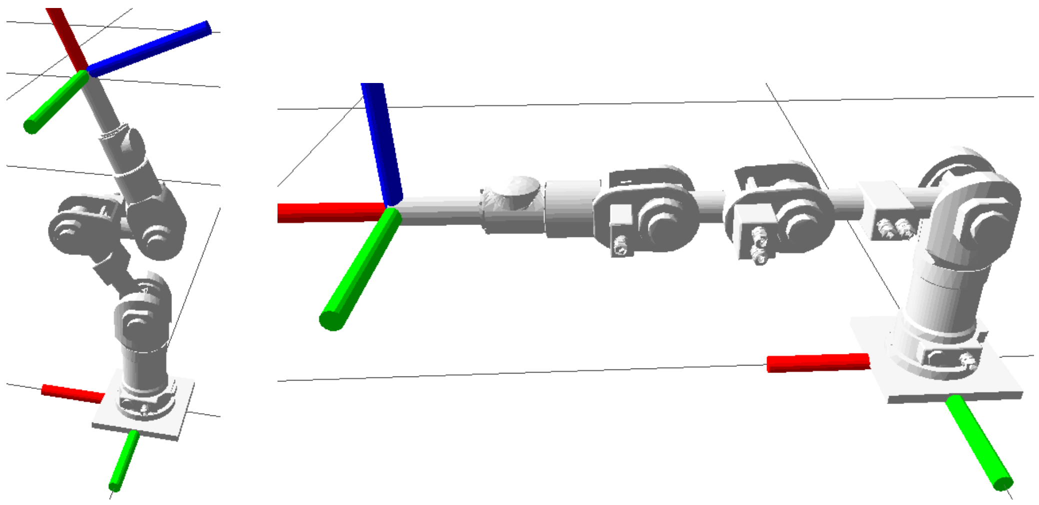


Figure 4.41.: Katanas URDF model and two related coordinate frames used for the IK workaround; attached to the base: *"katana_internal_controlbox_link"*, attached to the end-effector: *"katana_openrave_frame"*;
(coordinate frames [red, green, blue] → [x, y, z])

```
# katana_medbox_picking/KatanaSolutions
uint64 seq
string frame
bool solution
uint64 num_solutions
katana_medbox_picking/KatanaSolution[] solutions
  float64[5] data
```

Listing 4.16: ROS action for box detection

[pos.x, pos.y, pos.z, or.x, or.y, or.z]

This information is simply passed into the OpenRAVE 5D IK solver which then tries to calculate possible arm configurations. Each calculated configuration consists of a full set of five joint angles, one angle for each joint of the Katana arm. The solutions are packed into a response message of type *KatanaSolutions* (see Listing 4.16) and sent to the calling node.

Important to know is that the goal position is defined in the OpenRAVE environment's global frame. In the used configuration this matches the *"katana_internal_controlbox_link"* frame as defined in the Katanas URDF description (see subsection 4.3.5). This frame is located in the bottom center of the Katanas base plate with the x-axis oriented to the arm's front and the z-axis oriented upwards. The back side of the Katana arm's base is characterized by the control box connector. The end-effectors planning point matches the origin of the *"katana_openrave_frame"* from the URDF description. In Figure 4.40 it can be seen that this point approximately lies at the center between the fingers of the original two finger gripper. The given goal orientation is a direction vector and leads to an orientation of the end-effector that the z-axis of the mentioned *"katana_openrave_frame"* from the URDF description is antiparallel to this vector. Figure 4.41 shows pictures of the Katana's URDF model and the coordinate axes of the two mentioned frames.

4.3.7.2.2. katana_openrave_trafo_server node

As described in the previous paragraph, the OpenRAVE IK solver uses another point (coordinate frame) on the end-effector to calculate the IK for, than it is required for the overhead grasp during the item manipulation. This is caused by the fact that the original two finger gripper has been replaced by the vacuum gripper and that the interesting point for grasping with such a gripper and therefore for IK calculation is the gripper's tip. Furthermore the OpenRAVE IK solver expects the goal coordinates to be in relation to the frame attached to the Katana's base plate which in the used URDF description is called *"katana_internal_controlbox_link"* frame (see Figure 4.41).

So the task for the *katana_openrave_trafo_server* node is, to calculate the position of the OpenRAVE planning point relative to the *"katana_internal_controlbox_link"* frame, simulating that the vacuum gripper's tip is already at the desired goal position and in the desired orientation for an overhead grasp. Therefore the node implements a ROS service with the definition given in Listing 4.17. On start up this node fetches three ROS parameters. The *"REFERENCE_FRAME"* parameter defines the URDF description frame matching the OpenRAVE environment frame. In the given case this is the *"katana_internal_controlbox_link"* frame. The *"ENDEFFECTOR_FRAME"* parameter means the URDF description frame for which the IK shell be calculated. As mentioned above, this is the gripper's tip with the attached *"vacuum_endpoint"* frame. The *"KATANA_OR_FRAME"* parameter defines the frame attached to the point for which OpenRAVE calculates the IK. In the used configuration called *"katana_openrave_frame"*. Now the values from the parameters *"ENDEFFECTOR_FRAME"* and *"KATANA_OR_FRAME"* are used to fetch and save the corresponding transform between these frames which is assumed to be fixed and will be used for later calculations. Figure 4.42 shows the *"vacuum_endpoint"* frame and the *"katana_openrave_frame"* as defined in the used Katana arm URDF description.

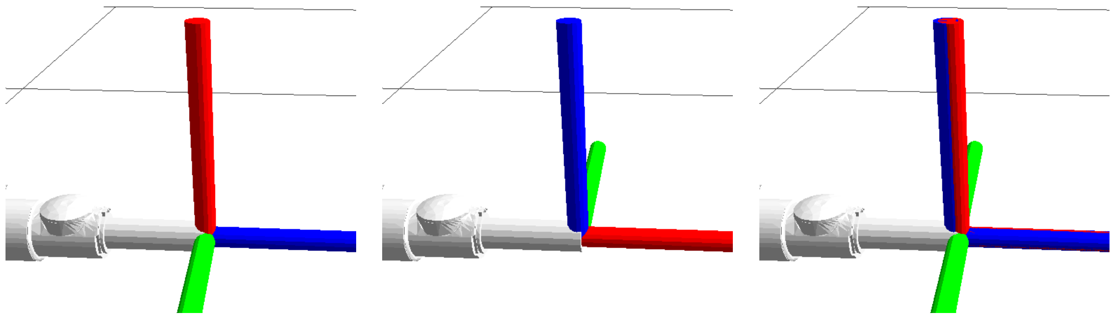


Figure 4.42.: The figure shows the Katanas two different end-effector frames used for IK; *Left: "vacuum_endpoint"* frame attached to the vacuum gripper tip; *Centre: "katana_openrave_frame"* representing the OpenRAVE end-effector frame; *Right:* both frames; (coordinate frames [red, green, blue] → [x, y, z])

```
# katana_openrave_trafo.srv
# request
geometry_msgs/PointStamped target_point
---
# response
geometry_msgs/PointStamped or_target_point
```

Listing 4.17: ROS action for box detection

The first step for every received request point is the transformation into the frame, given by the *"REFERENCE_FRAME"* parameter. This step is skipped if the received point is not already defined in the given *"REFERENCE_FRAME"*. Then a transformation is generated that simulates the overhead grasp pose for the *"vacuum_endpoint"* frame at the desired goal point relative to the *REFERENCE_FRAME*. The translation values of this transformation are the goal point coordinates and the rotation is set in a way that the frame's z-axis is pointing downwards. Using this transformation and the transformation fetched at start up, the position of the *"katana_openrave_frame"* origin is calculated and sent back to the calling node in the service response.

4.3.7.2.3. katana_openrave_jointtrafo_server node

The third part of the IK workaround is the *katana_openrave_jointtrafo_server* node. This node is required because of different joint angle formats. The angle format of the Katana's OpenRAVE description, used for IK calculations inside the *ik_katana_server.py* node, and the angle format of the URDF description, used by the other ROS arm navigation nodes, are completely different. More precisely they differ in the position of the zero-angle and the angle counting direction as to see in Figure 4.43. This means that a given joint angle set leads to different configurations or rather a desired configuration requires different joint angle sets. Figure 4.44 shows some example configurations and the corresponding angle sets for both description formats.

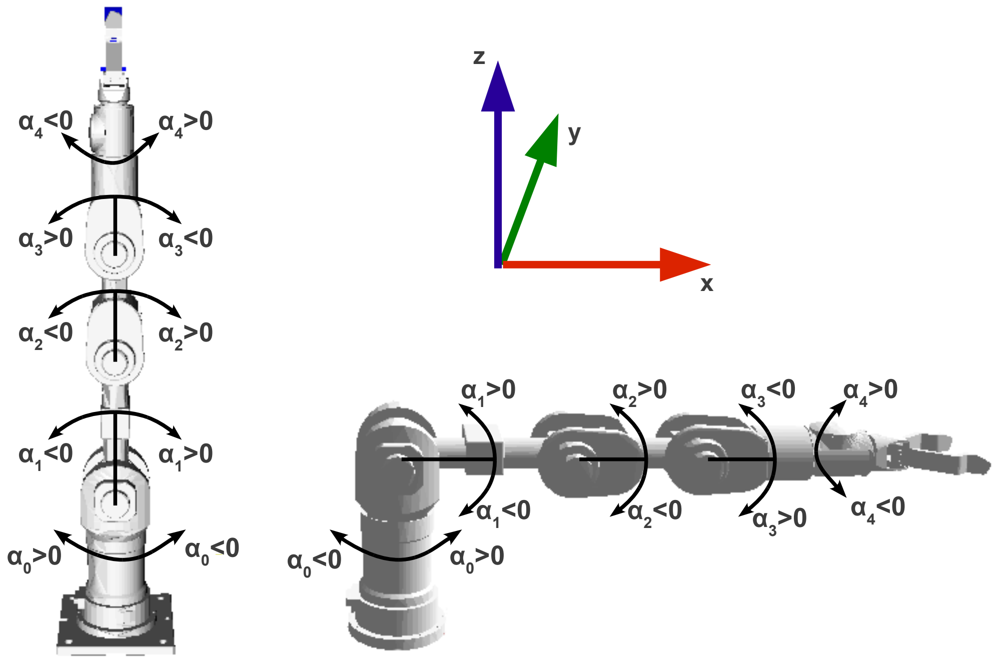


Figure 4.43.: The pictures show arm configurations for a zero angle set and the joint angle counting directions; *Left:* OpenRAVE model; *Right:* URDF model

```
# katana_openrave_jointtrafo.srv
# request
katana_medbox_picking/KatanaSolutions solutions_openrave_format
---
# response
katana_medbox_picking/KatanaSolutions solutions_katana_format
```

Listing 4.18: ROS action for box detection

The node implements a ROS service with the definition in Listing 4.18 which has the same message type for request and response. It is also the same message type as the response from the OpenRAVE IK service described above so that the IK response can be directly used as request for this service. As it is shown in Listing 4.16, the request message can contain several sets of joint angles. All the angle sets, each having five joint values in the range $[-\pi \ldots \pi]$, are simply converted from the OpenRAVE model format (angles α_i) to the URDF model format (angles β_i) using the following formula and sent back to the calling node in the service response.

$$\beta_i = a_i \cdot (\alpha_i + b_i) + c_i$$

$$\begin{aligned} with: \quad a &= [-1, -1, -1, -1, -1] \\ b &= [0, -\frac{\pi}{4}, 0, 0, 0] \\ c &= [0, \frac{\pi}{4}, 0, 0, 0] \end{aligned}$$

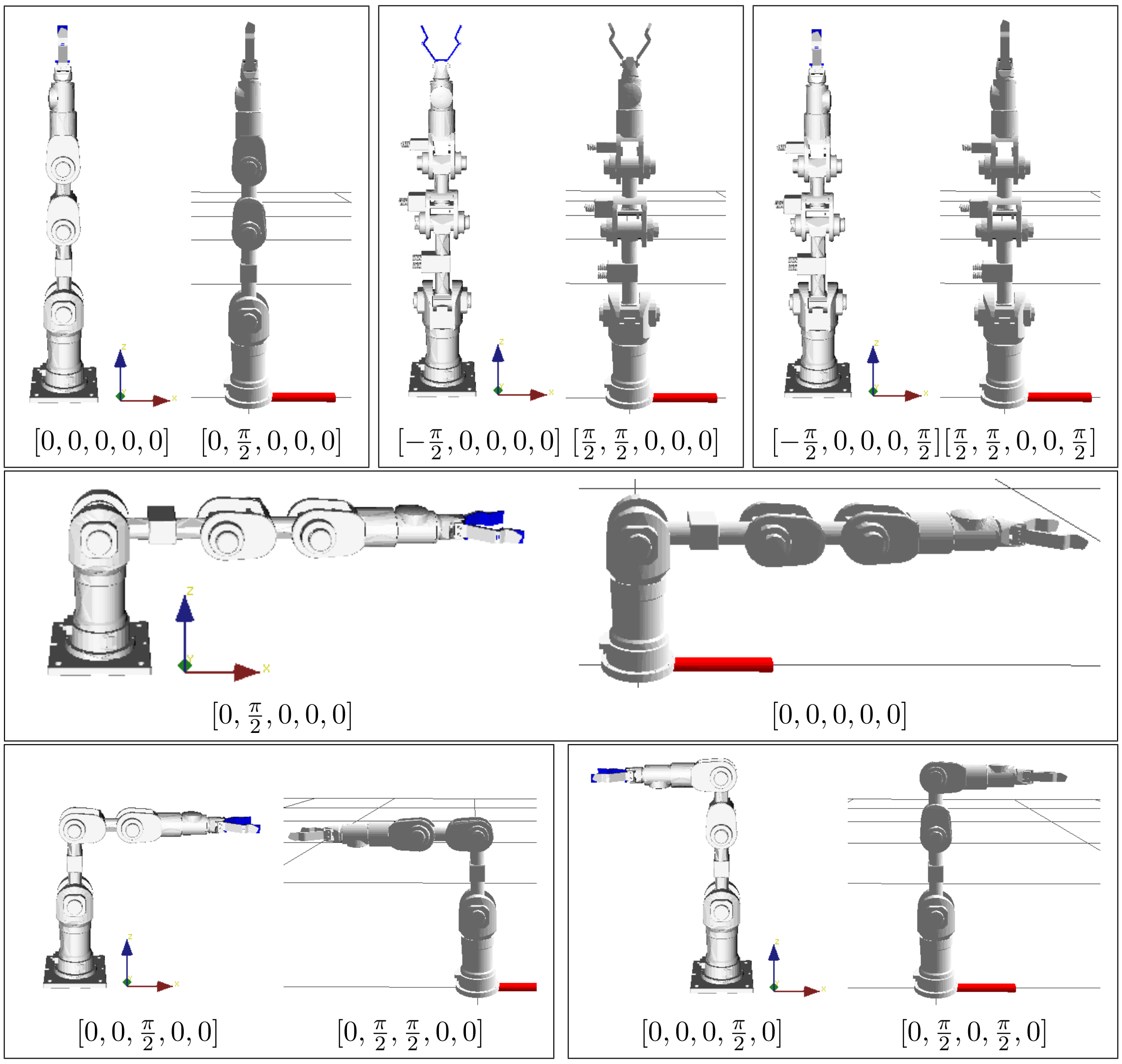

$[0, 0, 0, 0, 0]$
$[0, \frac{\pi}{2}, 0, 0, 0]$
$[-\frac{\pi}{2}, 0, 0, 0, 0]$
$[\frac{\pi}{2}, \frac{\pi}{2}, 0, 0, 0]$
$[-\frac{\pi}{2}, 0, 0, 0, \frac{\pi}{2}]$
$[\frac{\pi}{2}, \frac{\pi}{2}, 0, 0, \frac{\pi}{2}]$
$[0, \frac{\pi}{2}, 0, 0, 0]$
$[0, 0, 0, 0, 0]$
$[0, 0, \frac{\pi}{2}, 0, 0]$
$[0, \frac{\pi}{2}, \frac{\pi}{2}, 0, 0]$
$[0, 0, 0, \frac{\pi}{2}, 0]$
$[0, \frac{\pi}{2}, 0, \frac{\pi}{2}, 0]$

Figure 4.44.: The figure shows Katana arm configurations with the corresponding joint angle sets and configuration differences resulting from different joint angle formats in the OpenRAVE and URDF description; the pairs left parts show the OpenRAVE configuration and the right parts the URDF configuration

4.3.8. Robot control

The several steps, necessary to fulfil the task of order picking, are shown in Figure 4.1. They are implemented as ROS services and actions in the *"Services & actions"* layer (see Figure 4.4). When a new order is sent to the robot, the *"Overall robot control"* receives it and in the following, it is responsible for calling the right services and actions in the right sequence to fulfil the task.

The *"Overall robot control"* is implemented in the *kombot_smach.py* node which, as the name already suggests, uses the functionality of the SMACH (State Machine) python library to build the state machine shown in Figure 4.45. This state machine in turn is wrapped into a ROS action with the definition given in Listing 4.19. A ROS SMACH documentation as well as several tutorials can be found under:

`http://www.ros.org/wiki/smach`, (2012)

```
# goal definition
kombot_smach/KomBotArticle[] order
---
# result definition
uint8 status
uint8 SUCCEEDED       = 0   # Everything done
uint8 PREEMPTED       = 1   # Goal canceled
uint8 ABORTED         = 2   # Aborted, Error occured

string smach_state
kombot_smach/KomBotArticle[] order_result
---
# feedback
string smach_state
kombot_smach/KomBotArticle[] picked_up
```

Listing 4.19: ROS action for sending a new order to the robot

Due to the fact, that this implementation of the state machine is only for testing the basic functionality of the services and actions in the *"Services & actions"* layer, hardly any error and fault handling has been implemented. If a called action finishes not successful the whole order picking is aborted which can also be seen in Figure 4.45. This figure also shows, that the *"move to shelf place"* as well as the *"deliver order bin"* step from the work cycle overview (Figure 4.1) are omitted in the prototype's state machine due to some problems with the autonomous navigation, as already mentioned in subsubsection 4.3.4.1.

A new order is received from the node as action goal and consists of an array of *kombot_smach/KomBotArticle*s (message definition see Listing 4.20). For each article in the order this message contains identifying information about the article, like the article's identification number from the warehouse database and the article's description as well as the article's order size. In this context, order size means the number of pieces to be

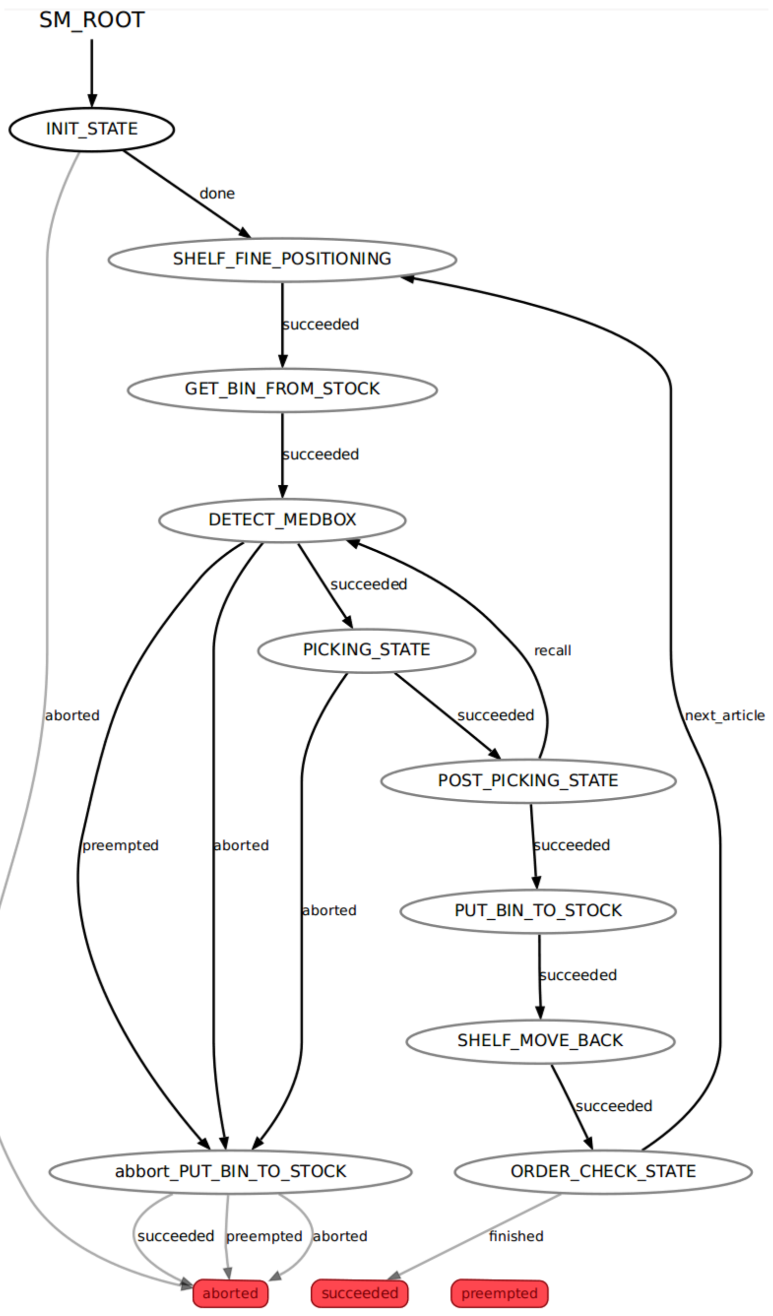


Figure 4.45.: The picture shows the *"Overall robot control"* state machine, implemented with SMACH, which calls the services and actions to fulfil the task of order picking

fetched from the shelf to satisfy the customer's order. The message also contains information where the article can be found (*shelf_...*). The *shelf_row* for instance indicates the shelf level from where the *Container Manipulation* module will fetch the container (see subsection 4.3.3) while the *shelf_number* and *shelf_column* are from interest for the fine positioning process. Another message content is the article's dimension (*box_...*), necessary for the object detection (see subsection 4.3.6). Finally it contains a way point with orientation on the map (*map_...*) directly in front of the article's shelf as it should be used for the autonomous navigation (see subsubsection 4.3.4.1).

```
# kombot_smach/KomBotArticle
uint64     article_number
string     description
uint64     amount
uint64     shelf_number
uint64     shelf_column
uint64     shelf_row
uint64     box_length
uint64     box_width
uint64     box_height
float64    map_x
float64    map_y
float64    map_theta
```

Listing 4.20: ROS message with article information for the robot

From the different possible state implementations provided by the SMACH library only two types are used. The states *INIT_STATE*, *POST_PICKING_STATE* and *ORDER_CHECK_STATE* are implemented as standard states by simply inheriting from the state base class and implementing an execute method which does the actual work. SMACH is relatively easy to use but the state implementation requires the definition of decided state outcomes, inputs and outputs. The outcomes are strings used to define the state transitions depending on the outcome values. Inputs and outputs are used to pass data into the state for processing and to return data from the state respectively. This strict separation of inputs and outputs causes that input data can not be modified directly and have to be copied before editing, which is very inconvenient in some cases but ensures that changes can not have any effect to the data outside the state. The other states, used in the state machine, are of type *SimpleActionState* which is a special state type that simply calls a ROS action and returns the result and the result state. It also provides the feature to register callbacks for generating the goal message, used for the following action call, or for processing the action's result message.

The state machine is stitched together by adding states to it. Each of this states gets at least a name, a state object that has to be inherited from the state base class and a list of pairs defining the following state depending on the states outcome value. It can also get a list of pairs for remapping variables from names used in the state machine definition to variable names used in the state definition and vice versa. Depending on the state's type (standard state, action state, etc.) a state itself will have additional parameters like, in case of an action state, callback functions or the action server name that should be called and so on. Similar to states the state machine itself has outcomes, inputs and outputs, so that a given state machine can be wrapped into a state and used as sub part of an higher state machine. Another possibility is to wrap the state machine into an action server like in the present case. This means that an action server is created that forwards every received goal message to the state machine. Unless otherwise defined, the first added state is used as starting state. The action server also uses three definable lists, mapping state machine outcomes the one of the three possible action result states (SUCCEEDED, ABORTED, PREEMPTED).

When the implemented prototype state machine, shown in Figure 4.45, is started by receiving a new action goal the first step is the *INIT_STATE*. Here the received goal is checked for validity and the first article to be fetched is selected and handed over to the next state which in the actual implementation is the *SHELF_FINE_POSITIONING* because the autonomous navigation part is omitted. In the *SHELF_FINE_POSITIONING* state the *landmark_positioning_action_server* is called with a goal generated in a call back. This call back uses the *shelf_column* information to select the correct position relative to the landmark. The shelf and landmark configuration for the prototype test uses landmarks positioned in the middle between two neighbouring shelf columns so that the robot has to be positioned slightly left from the landmark to fetch containers from columns with uneven numbers and lightly right from the landmark to fetch from columns with even numbers.

After a successful fine positioning the goal callback of the *GET_BIN_FROM_STOCK* state uses the *shelf_row* information to generate the goal message to be sent to the *mechatronics_controller* action server that manages the necessary sub steps to fetch the container from the right shelf level. The container fetching is followed by the *DETECT_MEDBOX* state with a goal containing the detection sensor topic and the dimensions of the desired object. This is the first state that uses a result callback to check the action result as well as the actions result state and to forward the detection result to the next state which is the *PICKING_STATE* calling the *katana_medbox_picking_action_server*. In the *POST_PICKING_STATE* the state machine's internal order progress is updated and the fetched article's stock amount in the warehouse management's database is decremented. If the fetched amount is less than the order's amount, the detection and picking is restarted otherwise the following state is the *PUT_BIN_TO_STOCK* which simply is the complement to the *GET_BIN_FROM_STOCK* state.

A different and maybe a better way to update the stock amount after picking an item from the storage container would be to use the action feedback to signal the calling instance that an article has been taken from the stock and to invoke a stock update. This solution would be better for real world use because only one central part of the robot framework needs access to the warehouse database. The following *SHELF_MOVE_BACK* state is similar to the preceding *SHELF_FINE_POSITIONING* state but uses a position further away from the shelf which should be safe to start the autonomous navigation. The final state in the order picking sequence is the *ORDER_CHECK_STATE* which simply checks, if the order is finished or not. Finished means that all article's have been picked with the requested amount. If this is not the case and there are still some article's missing, it selects the next article from the list and starts over with the shelf fine positioning. If the order is completed the state machine terminates and the action server will send the result data, contained in the according state machine variable, to the calling action client.

It has already been mentioned that this prototype state machine does not really implement any fault or error handling. Therefore the state machine graph, shown in Figure 4.45, is quite simple. Its also simplified by the fact that only explicit state transitions are shown, even though lot of implicit transitions exist too. This is due the fact that all the used action states have three possible outcome values, *succeeded*, *preempted* and *aborted*, but for most of them only the transition for a succeeded, in some case also for an aborted action is explicitly defined. The implicit transitions are automatically connected to the corresponding final state of the state machine which in Figure 4.45 are shown as red ovals. The only more or less fault handling state is the *abbort_PUT_BIN_TO_STOCK* state which is the same as the normal *PUT_BIN_TO_STOCK* state but in every case terminates the wrapping action. The calling node is informed that the action has been aborted. This implementation of "fault handling" is also the best example for the absence of real fault handling because actually this state does not even check if the arm is in a safe position for starting the container manipulation unit after a failed picking action.

4.3.9. Warehouse management

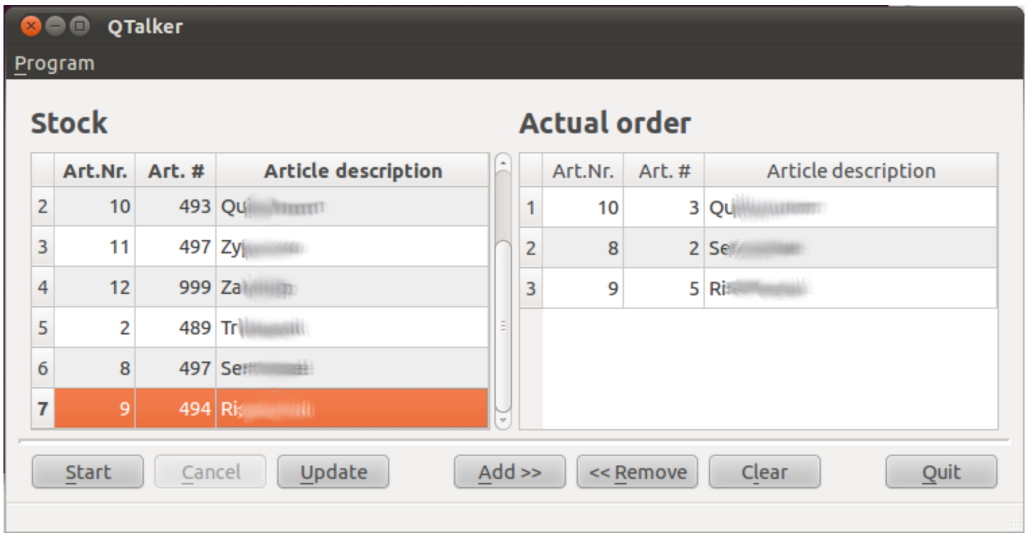
The top layer of the software concept (see Figure 4.4) is the warehouse and order management. As also shown in the figure and mentioned in subsection 4.1.2 this is usually a very complex piece of software with a lot of functionalities. For the prototype testing software the warehouse management layer has been kept quite simple with a MySQL database storing the required information and a simple ROS node combining the order and warehouse management software.

The prototype's database uses a very simple design with only two tables. The first table contains the articles with all additionally needed information as there are:

- article number: a unique number for each article in the system
- article amount: the actual number of articles in stock
- shelf number: the number of the shelf where the article is stored
- shelf column: horizontal position in the shelf and therefore the position for the robot
- shelf row: vertical position in the shelf, used by the container manipulation unit
- package dimensions: necessary for the object detection process

The second table contains a 2D map pose (position and orientation) for each pair of shelf number and shelf column, to be used for the autonomous navigation. These information are sent to the robot, but as repeatedly mentioned, are actually not used because the autonomous navigation part is omitted. The contained map pose has to be chosen in a way that the robot has clear view of the landmark, corresponding to the desired shelf place.

The above mentioned ROS node provides a simple Qt GUI as shown in Figure 4.46 for generating orders and sending them to the robot. On the left side of the GUI the actual



Stock

	Art.Nr.	Art. #	Article description
2	10	493	Qu[redacted]
3	11	497	Zy[redacted]
4	12	999	Za[redacted]
5	2	489	Tr[redacted]
6	8	497	Se[redacted]
7	9	494	Ri[redacted]

Actual order

	Art.Nr.	Art. #	Article description
1	10	3	Qu[redacted]
2	8	2	Se[redacted]
3	9	5	Ri[redacted]

Figure 4.46.: This picture shows an image of the simple warehouse and order management software GUI with the actual stock on the left, the actual order on the right side and control buttons on the bottom to edit the order and to send it to the robot for picking

stock is displayed and the right side shows the actual order. Underneath the two lists are some control buttons for changing the order on the right side or to refresh the actual stock on the left side of the GUI. There are also buttons to start and cancel order picking or to quit the program. The actual stock amount is directly fetched from the MySQL database and displayed in the left half of the GUI. For establishing a connection to the robot, the node uses a ROS action client that connects to the corresponding action server from the *robot control* node, described in the previous section (subsection 4.3.8). To start order picking, the node sends a new action goal to the robot through its action client and for interrupting a running picking process it sends a cancel request.

When the start button is clicked to start the picking process a new goal is generated which contains all the articles contained in the right list of the GUI. To implement a minimum of process optimization, the articles in the new goal are sorted in an ascending order with *shelf number*, *shelf column* and *shelf row* as sort criteria, assuming that the numbering of them is also ascending. This will reduce the robot's ways because all articles stored in the same shelf column will be picked before moving to the next one.

Chapter 5.

Experiments

5.1. Autonomous navigation

The goal of this experiment was to evaluate the positioning accuracy of the single *Krikkit* drive guided by the *ROS navigation stack* (see subsubsection 4.3.4.1). Of particular interest with this experiment was the accuracy of repositioning at a previously saved position which was estimated by the *amcl* node from the navigaiton stack. The *amcl* node is responsible for localization on the map by combining laser and odometry data. This experiment was also the first extensive test for the *Krikkit* driver implemented in the *ODO_CAN_krikkit_node* described in subsection 4.3.1. This is the only experiment concerning the autonomous navigation. A similar experiment with the whole robot failed because the whole robot has serious difficulties with in-place rotations necessary for localization. For a more detailed description of these problems see paragraph 4.3.4.1.1.

5.1.1. Setup

For this experiment the Krikkit drive was equipped with an Hokuyo URG-04LX laser scanner. Markers for measuring the distance to the desired goal pose have been placed at the robot's front and rear centre having a distance of $495\,mm$. The bottom left image in Figure 4.6 shows the used configuration. The navigation stack for the robot was configured according to the navigation stack tutorial already mentioned in subsubsection 4.3.4.1. Interesting to know in this regard are the allowed goal tolerances defining within which maximum linear and angular distances the navigation stack considers the goal as reached:

$$\text{Linear: } 0.05\,m\text{; Angular: } 0.1\,rad\ (5.73°)$$

In a preparation step a map of the laboratory was generated using the ROS *gmapping* package and saved for later use. In a second step the *teleop_krikkit node*, described in subsubsection 4.3.1.2, was used to move the robot to three generic positions on the map. The *amcl* pose estimations at these positions were saved and then used as navigation goals for the later experiment. Markers were placed on the floor directly beyond the markers on the robot. During the experiment these marker were used for measuring the distance to the desired pose. Figure 5.2 indicates these floor markers as big black arrows. Figure 5.1 shows the map section and goal poses used for this experiment. The goal poses relative to the map frame are also shown in the following table.

Position number	x [m]	y [m]	Θ [rad]
1	3.145340	3.226932	-2.739936
2	-0.006413	2.407585	-0.200077
3	6.531130	-2.851779	0.337694

Table 5.1.: Goal poses for the auto navigation experiment

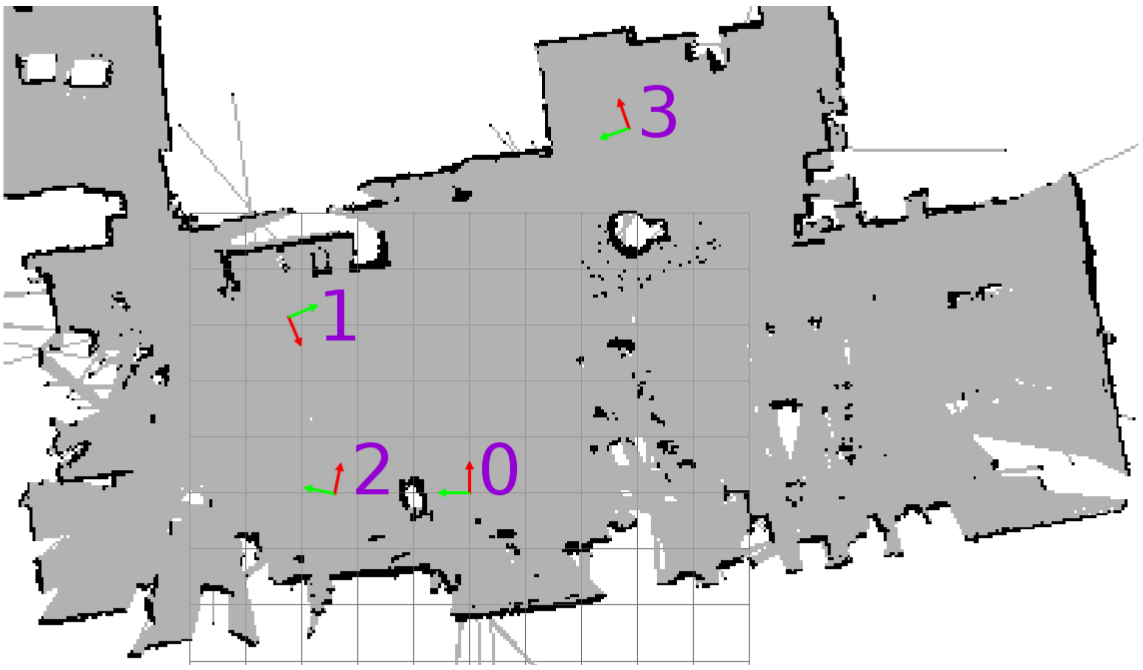


Figure 5.1.: This figure shows the map section of the laboratory used for the auto navigation experiment, the map frame (number 0), three poses (1-3) used for positioning during the experiment and a grid with a cell size of $1\,m$ ([red, green, blue] $\rightarrow$ [x, y, z])

5.1.2. Execution

The experiment started with the robot approximately positioned at the maps origin identified by the number ”0” in Figure 5.1. After starting the navigation stack the *amcl* node was triggered manually by a service call to perform an initial localization. Now one of the goal poses in Table 5.1 was selected randomly and sent to the navigation stack as new auto navigation goal. When the goal was reported as *succeeded* by the navigation stack the current *amcl* pose estimation was saved immediately. The distances between the according markers on the robot and those on the floor were measured as shown in the left part of Figure 5.2. The measured values were noted and used later on for calculating the actual distance to the formerly marked goal pose. This process was repeated 16 times. Due to the fact that this experiment was not intended to be the final auto navigation experiment only a few runs have been performed and caused by the random goal selection position 1 was only selected three times.

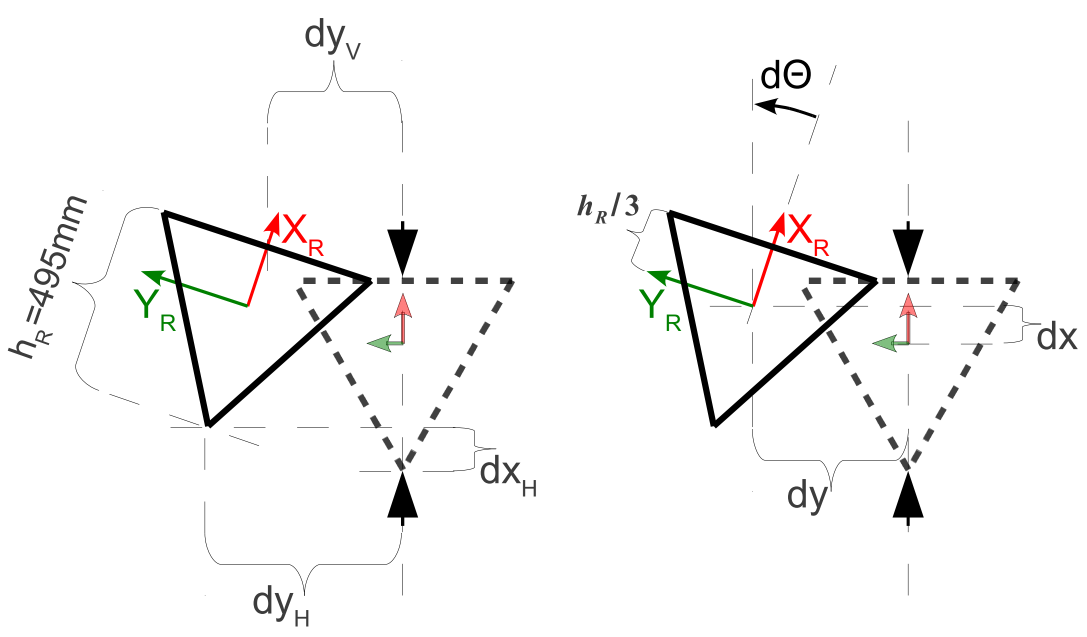


Figure 5.2.: The figure shows the measured and calculated distances of the auto navigation experiment; the actual robot pose is indicated by the black triangle and the desired goal pose is indicated by the dashed triangle; *Left:* Measured distances between the markers on the robot and the markers on the floor indicating the desired goal; *Right:* Calculated distances of the robot's centre between the actual and desired goal pose

5.1.3. Results and evaluation

The direct results of the experiment are reported in Table 5.2. These are the pose estimations of the *amcl* node at the moment when the auto navigation stack reported the goal as succeeded and the distances (dx_H, dy_H, dy_V) between the associated position markers at the robot and on the floor as shown in the left part of Figure 5.2. Using measurement values from Table 5.2 the differences to the original *amcl* goal pose estimation, reported in Table 5.1, and the linear and angular distance to the original pose, indicated by the floor markers, are calculated. The right part of Figure 5.2 shows the distances calculated from the measured marker distances. For these calculations the robot's shape is approximated as an equilateral triangle with a height of $h_R = 495\,mm$ and the robots coordinate frame is located at the incircles centre. The rotational, linear and absolute distances were calculated using the following formulas:

$$\begin{aligned} d\Theta &= arcsin\left(\frac{dy_V - dy_H}{h_R}\right) \\ dx &= \frac{2\,h_R}{3}\,(cos(d\Theta) - 1) + dx_H \\ dy &= \frac{2\,h_R}{3}\,sin(d\Theta) + dy_H \\ d &= \sqrt{dx^2 + dy^2} \end{aligned}$$

For the *amcl* pose difference the goal pose values from Table 5.1 were simply subtracted from the measured values. The absolute distance d is calculated with the same formula as for the marker positions. The calculated values for both, the *amcl* and marker distances, are reported in Table 5.3. This table also shows the distance means and standard deviations separate for each of the three goal poses as well as for all poses together.

In this context it is of particular interest that for only 5/16 calculated *amcl* distances the absolute distance d is above the allowed goal tolerance of $0.05\,m$ while the angular distance is above the tolerance of $0.1\,rad$ in 12/16 runs. This is caused by the system time delays and the robot's typical behaviour when moving towards a goal. When the navigation stack detects that the pose estimation is within the goal tolerances it sends a stop command to the drive via the corresponding topic and reports the goal as *succeeded* via the action interface. The *Krikkit* driver node as well as the action client node receive the messages with a little delay and therefore the robot moved a little further when the noted *amcl* pose is fetched. The fact, that the angular distance is affected more strongly, is caused by the robot's behaviour when moving towards a goal. The robot usually tries to minimize the linear distance and in a last step it rotates in place until the rotational distance is within the tolerance.

In real the positioning is much worse as the *amcl* poses suggest. In 13/16 runs the actual linear distance was above the navigation stack goal tolerance and the angular distance exceeded the tolerance in all runs. On the one hand this is caused by the *amcl*'s pose ***estimation*** which is further worsened by the fact that the used laser scanner has a maximum scanning range of $4\,m$. The small lasers maximum range is a problem because the experiments area width is more than $4\,m$ (see also Figure 5.1) which forces the robot to use odometry for localization which is a lot less accurate than localization via laser data.

Position 1						
Seq. number	**amcl pose** ([m], [rad])			**marker dist.** ([mm])		
	x	y	Θ	dx_H	dy_H	dy_V
4	3.141	3.245	-2.827	40	-160	5
12	3.103	3.215	-3.116	80	-215	-35
15	3.185	3.218	-2.869	-20	-120	10

Position 2						
Seq. number	**amcl pose** ([m], [rad])			**marker dist.** ([mm])		
	x	y	Θ	dx_H	dy_H	dy_V
1	0.022	2.399	-0.258	60	-10	-75
3	0.011	2.376	-0.591	55	-115	-215
5	0.009	2.443	-0.739	10	-45	-140
7	-0.033	2.363	0.082	0	-45	-105
9	0.025	2.421	-0.247	10	-125	-210
11	-0.037	2.449	-0.573	-30	-10	-105
13	0.005	2.369	0.194	-20	-85	-160

Position 3						
Seq. number	**amcl pose** ([m], [rad])			**marker dist.** ([mm])		
	x	y	Θ	dx_H	dy_H	dy_V
2	6.490	-2.849	-0.085	-20	-135	-25
6	6.527	-2.914	0.750	10	-130	0
8	6.506	-2.821	-0.186	-20	-30	85
10	6.514	-2.900	0.279	70	-203	-15
14	6.459	-2.862	0.039	-50	-125	5
16	6.495	-2.851	0.019	-10	-140	10

Table 5.2.: Measurement results of the auto navigation experiment

Position 1								
Seq. number	**amcl distances** ([mm], [rad])				**marker distances** ([mm], [rad])			
	dx	*dy*	*d*	*dΘ*	*dx*	*dy*	*d*	*dΘ*
4	-4.0	18.3	18.7	-0.087	21.1	-50.0	54.3	0.340
12	-42.4	-12.2	44.1	-0.376	57.4	-95.0	111.0	0.372
15	39.6	-9.3	40.7	-0.129	-31.6	-33.3	45.9	0.266
Mean	-2.3	-1.1	34.5	-0.197	15.7	-59.4	70.4	0.326
Std dev	41.0	16.8	13.8	0.156	44.7	31.9	35.4	0.055

Position 2								
Seq. number	**amcl distances** ([mm], [rad])				**marker distances** ([mm], [rad])			
	dx	*dy*	*d*	*dΘ*	*dx*	*dy*	*d*	*dΘ*
1	28.4	-9.0	29.8	-0.058	57.1	-53.3	78.2	-0.132
3	17.6	-31.9	36.4	-0.391	48.2	-181.7	188.0	-0.203
5	15.4	35.7	38.9	-0.539	3.9	-108.3	108.4	-0.193
7	-27.0	-44.9	52.4	0.282	-2.4	-85.0	85.0	-0.122
9	31.3	13.8	34.2	-0.047	5.1	-181.7	181.7	-0.173
11	-31.0	41.0	51.4	-0.373	-36.1	-73.3	81.8	-0.193
13	11.9	-38.4	40.2	0.394	-23.8	-135.0	137.1	-0.152
Mean	6.6	-4.8	40.5	-0.105	7.4	-116.9	122.9	-0.167
Std dev	25.3	35.5	8.5	0.353	34.5	51.2	47.0	0.032

Position 3								
Seq. number	**amcl distances** ([mm], [rad])				**marker distances** ([mm], [rad])			
	dx	*dy*	*d*	*dΘ*	*dx*	*dy*	*d*	*dΘ*
2	-41.1	2.8	41.2	-0.423	-28.3	-61.7	67.8	0.224
6	-4.0	-62.5	62.6	0.412	-1.6	-43.3	43.4	0.266
8	-25.1	30.6	39.5	-0.523	-29.0	46.7	55.0	0.235
10	-17.2	-47.9	50.9	-0.059	45.4	-77.5	89.8	0.389
14	-72.6	-9.9	73.3	-0.299	-61.6	-38.3	72.5	0.266
16	-35.7	0.3	35.7	-0.318	-25.5	-40.0	47.4	0.308
Mean	-32.6	-14.4	50.6	-0.202	-16.8	-35.7	62.7	0.281
Std dev	23.6	34.6	14.8	0.338	36.0	43.1	17.5	0.060

All positions								
	amcl distances ([mm], [rad])				**marker distances** ([mm], [rad])			
	dx	*dy*	*d*	*dΘ*	*dx*	*dy*	*d*	*dΘ*
Mean	-9.8	-7.7	43.1	-0.158	-0.1	-75.7	90.5	0.094
Std dev	31.8	31.2	12.9	0.306	36.9	57.4	45.1	0.242

Table 5.3.: Calculated *amcl* and marker distances as well as their means and standard deviations separate for each goal pose and for all goals together

The loss of localization leads to recovery behaviours of the robot and this significantly increases the runtime. On the other hand the robot does not stop immediately after receiving the stop command but rather needs a while to decelerate. This leads to an additional error compared to the *amcl* distance.

Except the runtime, improvements would require major changes to the whole system. To improve the positioning accuracy a new *move_base* node would be necessary, again checking the pose estimation after the robot has stopped before reporting the goal as succeeded. If the actual pose is outside the goal tolerance this node should continue positioning. A similar behaviour is implemented in the positioning controller of the fine positioning system described in paragraph 4.3.4.2.2. An improvement of the runtime could be achieved more easily by using a laser scanner with a greater scanning radius, like the later used Sick LMS100. This would improve the localization and decrease the amount of recovery actions caused by delocalization of the robot. This in turn would significantly reduce the runtime.

5.2. Fine positioning

The goal of this experiment was to evaluate the positioning accuracy of the whole robot guided by the fine positioning system described in subsubsection 4.3.4.2. The system uses an artificial landmark for positioning relative to it. Of particular interest with this experiment was to find out if the system is accurate enough to position the robot within the acceptable tolerances for the container manipulation system. For the allowed tolerances have a look at section 5.3.

5.2.1. Setup

The setup for this experiment includes the test shelf shown in the left part of Figure 5.3. A cardboard wedge is used as landmark and mounted at the bottom centre of the two center columns. Therefore the landmarks attached frame, indicated by its axes X_M, Y_M in the right part of Figure 5.3, is located at the coordinates

$$x_M^A = 125\,mm \quad \text{and} \quad y_M^A = 530\,mm$$

relative to the auxiliary frame X_A, Y_A without any rotation. The auxiliary frame is located at the shelfs front right edge with its y-axis along the shelfs front. All measured distances are relative to this frame.

Column	**Label**	$\mathbf{x_G^M}$ **[mm]**	$\mathbf{y_G^M}$ **[mm]**	$\mathbf{\Theta}$ [°]
1	a	-175	90	0
2	b	-175	-135	0

Table 5.4.: Goal poses for the fine positioning experiment

In a preparation step two robot poses in front of the shelf columns 1 and 2, representing the optimal poses for container manipulation operations, were identified. This was done by repeated positioning of the robot via the fine positioning system combined with fetching of

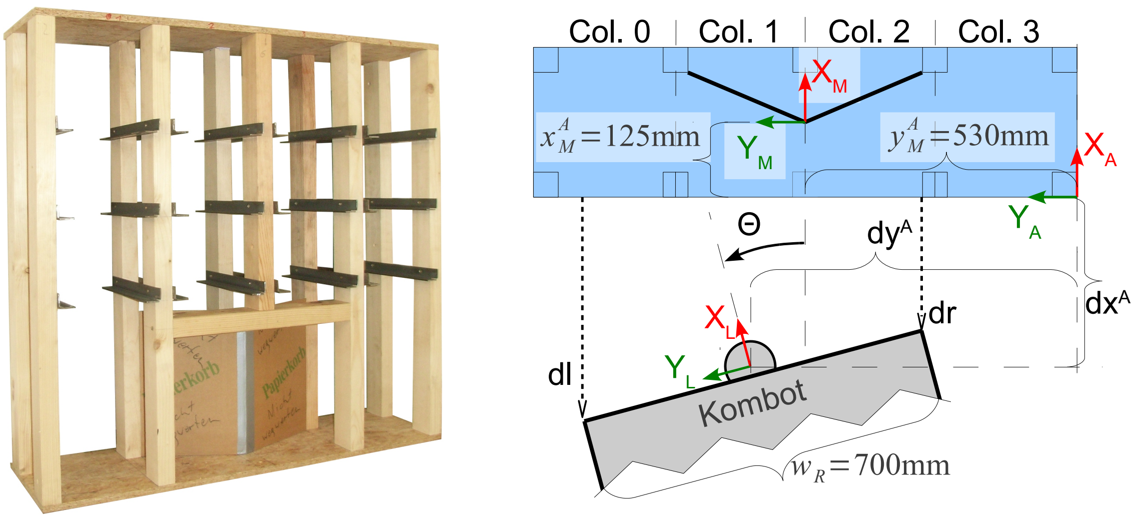


Figure 5.3.: *Left:* Shelf with centred landmark cardboard wedge used for the experiment; *Right:* The figure shows a top view of a typical measurement scenario during the fine positioning experiment including the shelf with its columns 0-3, the landmark (black wedge) centred at the bottom of the columns 1,2, the robot ("Kombot") with the navigation sensor at its front, the measured distances dl, dr, dx, dy and the calculated angle Θ

containers via the container manipulation system. The poses are defined by the location of the laser frame X_L, Y_L relative to the landmark frame X_M, Y_M. Table 5.4 reports these poses. The poses are also shown in Figure 5.4 marked by the crosses "a" and "b". The goal of the positioning system is to bring the robot in a position minimizing the linear and angular distance between the laser frame and goal pose. The goal is reported as succeeded if the distance is still less the goal tolerance after the robot has stopped.

5.2.2. Execution

This experiment consist of 60 runs. For each run the robot was positioned at one of ten start positions which are shown in Figure 5.4, marked with a "X". Each position was repeated three times with three different starting angles:

$$\text{Starting angles: } \Theta_{start} = \{-20, 0, 20\} \; [^\circ]$$

This results in a set of 30 starting poses and each pose was used twice, once for each of the two goal poses reported in Table 5.4. The starting pose defines the pose of the coordinate frame X_L, Y_L attached to the robot's laser scanner. It should be mentioned that the starting angles were only adjusted roughly ($\pm 5^\circ$) while the adjustment of the translational position was quite precise.

The different start poses simulate possible robot poses resulting from a previous auto navigation step to a goal pose in front of the shelf. Positions further away have not been chosen because the auto navigation experiment (see section 5.1) showed that a positioning within a radius of 10 *cm* and a maximum rotation of about 20° can be achieved using the

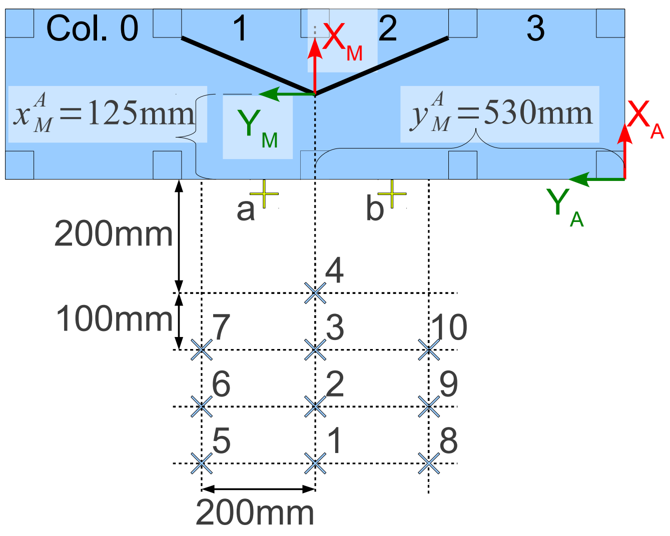


Figure 5.4.: The figure shows the ten start positions marked by a "X" and the two goal positions (a,b) marked by a cross for the fine positioning experiment

navigation stack.

After positioning the robot at the starting pose a testing node was used to send a fine positioning goal to the *landmark_positioning_action_server*. According to the results of the container manipulation experiment described in section 5.3 following values have been chosen for the positioning tolerances:

$$\text{Linear tolerance: } 10\,mm \qquad \text{Angular tolerance: } 0.01\,rad\ (\approx 0.573°)$$

Before starting the run, the distances dl and dr (see Figure 5.3, right part) were noted for calculating the actual starting angle Θ. The distances were measured from the front edges of the robot's frame to the shelf's front. After the run was finished the distances dl and dr were noted again. Also noted were the runtime t, the translation dx^A, dy^A of the laser frame X_L, Y_L relative to the auxiliary frame X_A, Y_A and the debug output of the *landmark_positioning_action_server*, reporting the final estimated distance to the goal pose. The estimated goal distance is the filtered transformation between the robot and the goal pose as estimated by the landmark tracker action server.

5.2.3. Results and evaluation

The translation dx^A, dy^A of the laser frame X_L, Y_L relative to the auxiliary frame X_A, Y_A was noted for calculating the actual resulting position dx^M, dy^M relative to the landmark frame X_M, Y_M and the distances dl and dr are used to calculate the angle Θ via the formulas below. It has to be mentioned that the values for dx^A are negative, referenced to the coordinate system X_A, Y_A. Using these values the linear distances dx, dy between the actual measured position and the desired goal position are calculated. The angle Θ already represents the angular distance to the goal because the rotation for the goals is intended to be zero. Table 5.5 reports the minimum, maximum, mean and standard deviation values for the calculated distanced dx, dx, d, Θ, the runtime and the filtered goal distances finally reported by the *landmark_positioning_action_server*. The values are reported separate for each of the two goal poses as well as combined for both goal poses (a,b).

$$\begin{aligned}\Theta &= arcsin\left(\frac{dl-dr}{w_R}\right)\\ dx^M &= dx^A - x^A_M\\ dy^M &= dy^A - y^A_M\\ dx &= dx^M - x^M_G\\ dy &= dy^M - y^M_G\\ d &= \sqrt{dx^2+dy^2}\end{aligned}$$

Goal "a"								
	Calculated distances				**Tracker distance**			
	Min	**Max**	**Mean**	**Std dev**	**Min**	**Max**	**Mean**	**Std dev**
Θ [°]	−1.64	0.41	−0.75	0.53	−0.53	0.47	−0.03	0.29
dx [mm]	−40	−30	−34.00	4.62	0.07	7.95	4.13	1.77
dy [mm]	30	55	43.33	6.48	−8.89	5.00	−2.22	4.85
d [mm]	46	64	55.40	5.12	2.86	9.96	6.60	2.09
t [s]	23	195	64.17	47.31				

Goal "b"								
	Calculated distances				**Tracker distance**			
	Min	**Max**	**Mean**	**Std dev**	**Min**	**Max**	**Mean**	**Std dev**
Θ [°]	−0.16	2.87	1.28	0.75	−0.51	0.53	0.08	0.33
dx [mm]	−40	−30	−33.33	4.61	−5.01	2.12	0.34	1.92
dy [mm]	50	75	62.67	6.12	−8.63	8.89	−1.21	4.80
d [mm]	61	81	71.20	6.12	0.54	8.92	4.71	2.33
t [s]	20	280	69.22	60.62				

Both goals "a,b"								
	Calculated distances				**Tracker distance**			
	Min	**Max**	**Mean**	**Std dev**	**Min**	**Max**	**Mean**	**Std dev**
Θ [°]	−1.64	2.87	0.26	1.21	−0.53	0.53	0.02	0.31
dx [mm]	−40	−30	−33.67	4.59	−5.01	7.95	2.23	2.65
dy [mm]	30	75	53.00	11.58	−8.89	8.89	−1.72	4.81
d [mm]	46	81	63.30	9.47	0.54	9.96	5.65	2.39
t [s]	20	280	66.69	53.97				

Table 5.5.: Evaluation of the calculated goal distances, the final distances reported by the *landmark_positioning_action_server*s goal tracker and the runtime

In the *"Tracker distance"* columns of the evaluation table (Table 5.5) can be seen that the final estimated linear distance d, which is one criteria for the decision if the goal is reached or not, is less than the tolerance of $10\,mm$ in every run. The second criteria is the angular distance Θ. The tracker's values of Θ are also less the tolerance ($\approx 0.573°$) in all runs. In contrast to the autonomous navigation experiment from the previous section these values are taken when the received odometry ensures that the robot has already stopped. Therefore these values don't change due to robot motion but for the reason of estimation fluctuations.

For the calculated real goal distance evaluation, shown in the left part of Table 5.5, it can be seen that for the separate evaluated goals the mean value of the linear distances d, dx, dy greatly exceeds the tolerance of $10\,mm$. The great mean distance in contrast to the relative small standard deviation, less the tolerance, suggest a great unknown systematic error. The minimum and maximum values show that the linear distance exceeds the tolerance in all runs. The real angular distance mean also exceeds the given tolerance ($\approx 0.573°$) but the standard deviation is in the scale of the angular tolerance. Here the resulting error might be a combination of a slightly misaligned landmark and the limited accuracy of the landmark trackers pose estimation which is the base for the whole positioning procedure. It should be noted that the angular tolerance exceeded the tolerance in 19/30 runs for goal pose "a" and in 24/30 runs for goal pose "b".

It was expected that the runtime would correlate with the starting distance but it turned out that this is not true for the prototype robot. Having a look at the runtimes t in Table 5.5, it can be seen that there are great differences between the minimum and maximum runtime values. The runtime was less a consequence of the starting distance, than rather a random result. This was shown by some runs, starting from the farthest start positions, but having runtimes less than 30 seconds. In opposite, runs starting from the closest starting position had runtimes of more than two minutes. The great runtimes are rather an outcome of the same problems causing the auto navigation to fail (see paragraph 4.3.4.1.1). It is probable that the runtimes could be improved significantly by a better and more powerful locomotion system.

5.3. Container manipulation

This experiment belongs to the work of Bernhard Puchinger (see [20]) and its results are here reported briefly because the original work is in German. The goal of this experiment was to evaluate the workspace of the container manipulation unit.

For this experiment the whole robot was placed in front of the shelf already used in the fine positioning experiment (see left part of Figure 5.3). Starting from the center position, with the fork of the container manipulation unit at the centre of the shelf column, the robot was stepwise moved sideways, forward and backward to find the translational tolerances. It appeared that the maximum lateral offset without rotation, ensuring a proper operation of the container manipulation unit, is $2.75\,cm$. If this maximum is exceeded the fork collides with the lateral profiles carrying the container. The maximum offset along the forks moving direction depends on the parameter settings for the manipulation unit but should not exceed $1.5\,cm$.

For finding the maximum rotational offset, the robot was placed centred in front of the shelf column, with a distance of $10\,cm$ between the shelf and the robot's front and rotated in both directions. With this setting the maximum rotational offset is about $3.3°$. This value shrinks dramatically if a translational offset is added.

For a proper operation of the manipulation unit it is very important to keep the rotational offset as low as possible. This is so important because of the long fork, even small rotations lead to collisions with the shelf when moving the fork forward.

5.4. Object detection

The aim of this experiment was to test the object detection module, described in subsection 4.3.6. It should be tested in different situations and under controlled, comparable conditions. Of special interest are the influences of ambient light, different box surfaces (reflective, matt) as well as the orientation inside the container and the orientation relative to other contained boxes.

5.4.1. Setup & execution

For this experiment the camera mount shown in Figure 5.5 was used. The container was placed on the bottom centre and the Kinect camera, not shown in the figure, was mounted on top. The camera orientation is indicated by its attached coordinate system which has a vertical offset of 880 mm to the base plate. The coordinate system on the bottom is approximately at the containers bottom centre point and used as reference frame for the detection. A light source for simulating different ambient light conditions is indicated by the flash light symbol.

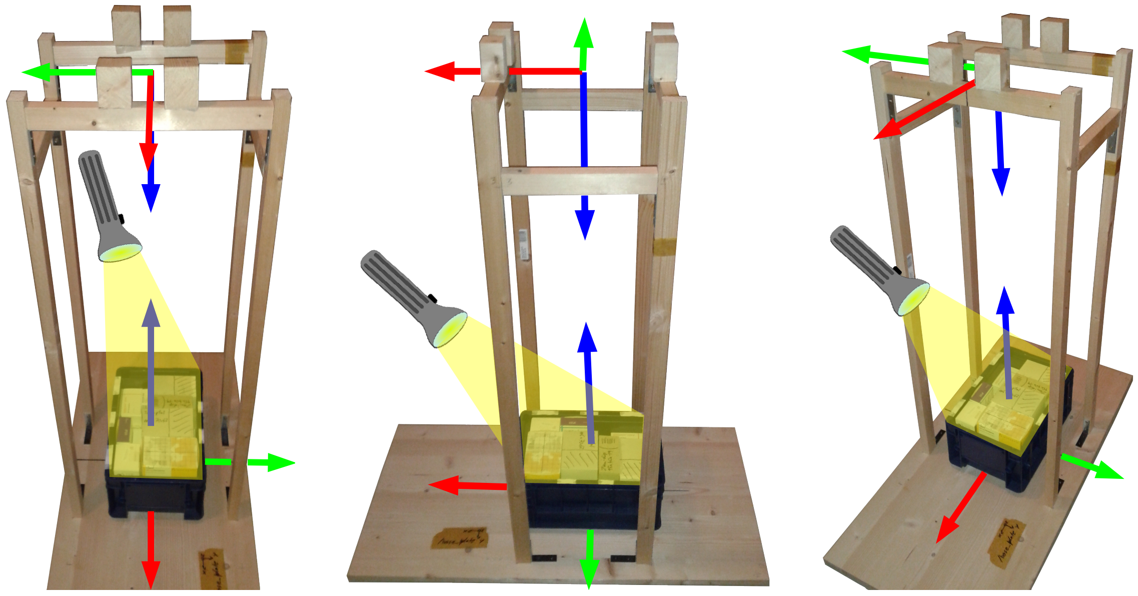


Figure 5.5.: The figure shows the camera mount with base coordinate system and the container used for the object detection experiment. Not shown: Light source, indicated by the flash light and the Kinect camera, indicated by its attached coordinate system (coordinate systems [red, green, blue] → [x, y, z])

Four kinds of drug boxes with different dimensions were used as target objects (see Table 5.6). For each box type there was one original drug box with a glossy surface and

Label	Length	Width	Height	Area	Diameter
A	$145\,mm$	$90\,mm$	$35\,mm$	$13050\,mm^2$	$171\,mm$
B	$105\,mm$	$70\,mm$	$68\,mm$	$7350\,mm^2$	$126\,mm$
C	$97\,mm$	$71\,mm$	$31\,mm$	$6887\,mm^2$	$120\,mm$
D	$95\,mm$	$41\,mm$	$20\,mm$	$3895\,mm^2$	$103\,mm$

Table 5.6.: Box dimensions of the target objects, their largest surfaces area and diameter as used for the detection experiment

two boxes made of matt grey cardboard.

During the experiment 1-3 boxes were placed inside the container. Each test case was repeated for all box types shown in Table 5.6. For testing the detection under variable light conditions a single box was placed inside the container. The detection was executed three times. Once without extra light and two times with two different light intensities. The same procedure was repeated twice. First with the original glossy box and then with the matt one. As a second test case, two or three boxes were placed inside the container. Of special interest was the detection result in case of partially occluded boxed or when they lie close to each other. The basic assumption for these test cases was that the container content is unmixed. For the third test case this assumption was rejected and two boxes of different type were placed inside the container. This should usually not be the case in a well managed warehouse so this test only concerns the detections selectivity.

5.4.2. Results and evaluation

The whole experiment consists of 57 runs with several runs for each of the three test cases described before. The results of the different test cases are described separate in the subsections below.

5.4.2.1. Variable light conditions

Figure 5.6 shows the experiments outcome pictures and point clouds for box type "C" under different light conditions. The result are quite the same for all of the tested box types shown in Table 5.6. The pictures and points clouds (a-c) belong to the original glossy drug box while the pictures (d,e) belong to the matt box type.

It can be seen that the point cloud of the container is dense in case of no extra light (Figure 5.6, (a)) but already has a small gap inside the dashed rectangle which indicates the drug box area. With increasing light intensity these gaps grow. Having a closer look at the point clouds (b) and (d), both taken with light intensity "1", it can be seen that the matt boxes cloud (d) has still no gap while the glossy boxes cloud (b) shows less than the half of the box. Comparing the clouds (c) and (e), taken with light intensity "2" it can be seen that the glossy box is now fully invisible and the matt box is only partially visible.

The detection worked well for both box types, glossy and matt, as long as there was not too much ambient light. The major problem with too much light is the detection of the container border which failed in nearly all cases with extra light and caused the whole

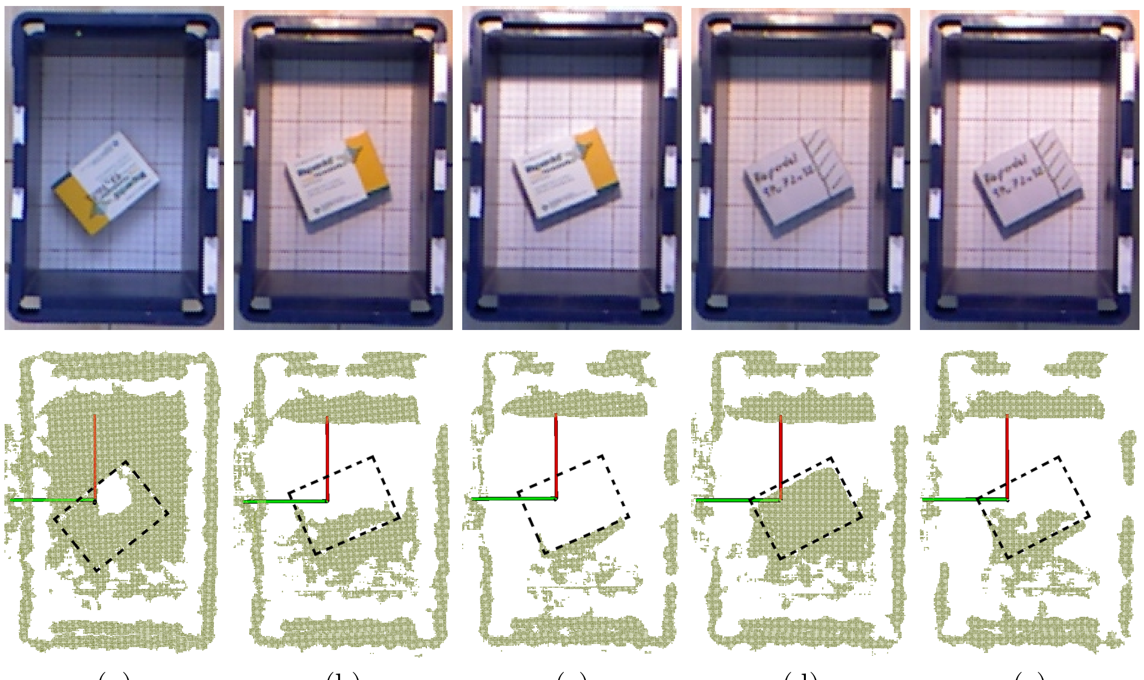


Figure 5.6.: The figure shows images of the container under different light conditions and the corresponding points clouds. *Left to right:* Original glossy drug box with light intensities 0-2 and matt box with intensities 1-2; (coordinate systems [red, green, blue] $\rightarrow$ [x, y, z])

detection to fail. This is caused by the fact that an undetectable container border prevents a proper segmentation of the container's content.

The reason for the ambient light problems with the point clouds is the sensing method of the used Kinect camera. The camera projects and detects infrared patterns which in case of high reflective surfaces or too much ambient light can not be detected properly. It has to be mentioned that the sensing works better on less reflecting surfaces. This can especially be seen in the point cloud of Figure 5.6 (d) which shows a good representation of the box despite the extra light with intensity "1" while the glossy box in (b) is already half invisible.

During this experiment another problem became obvious. The actually used detection method has a great problem with flat boxes like the box type "D" lying direct on the containers floor. Such boxes often can not be separated properly from the container bottom and therefore will not be detected. In some other cases, especially with small boxes, the used plane segmentation does not fit the plane good enough and therefore a big part of the surface is clipped what causes the detection to fail.

5.4.2.2. Multiple boxes

The *multiple boxes* test case is the default scenario for detecting objects in the container. As described in subsection 4.3.6 the used detection module only tries to detect the objects largest surface. The surfaces center points, calculated as mean of the corresponding point coordinates, are used as possible grasp points for item manipulation. This method works well for more or less fully visible surfaces. This means that primarily non-occluded boxes, usually lying on top, or only partially occluded boxes will be detected. Such boxes are easy to grasp what is important for the following manipulation step and therefore such boxes are preferred.

This experiment evaluation will show example scenarios with positive detection results, scenarios that cause the detection to fail or lead to false positive detections. Especially false positives are a great problem because they lead to a more or less blind grasp which carries a great risk to fail. The evaluation will also discuss the implemented but actually not used box pose estimation as well as its problems and weaknesses. Due to the results from the previous described test case no extra lighting of the scene was used to improve detection conditions.

5.4.2.2.1. Partially occluded but detectable boxes

Figure 5.7, (c) shows a scenario with 2 boxes of type "A" (see Table 5.6). One of them is partially occluded by another box. Figure 5.7, (a,b) shows the corresponding input point cloud. The boxes are indicated by two dashed rectangles. The fine dashed rectangle indicates the upper box which partially occludes the other one. This leads to a hole in the point cloud representation of the lower box. This can be seen in Figure 5.7, (b). Nevertheless both boxes are detected properly what can be seen in Figure 5.7, (e,f). These images show the calculated surfaces centre points which are used as grasp points by the item manipulation module. It can be seen that the centre point of the upper surface is more or less exact at its centre while the centre point of the lower surface is shifted along the x-axis in negative direction. This is caused by the hole in the representing point cloud and the calculation method for the centre point which is a simple calculation of the mean along the three coordinate axes. This leads to a shifting of the calculated point towards the majority of points. This can also be seen in Figure 5.7, (d) which shows the aligned templates from the surface pose estimation step. While the template for the upper surface is properly aligned the lower surfaces template is also shifted, similar to the centre point.

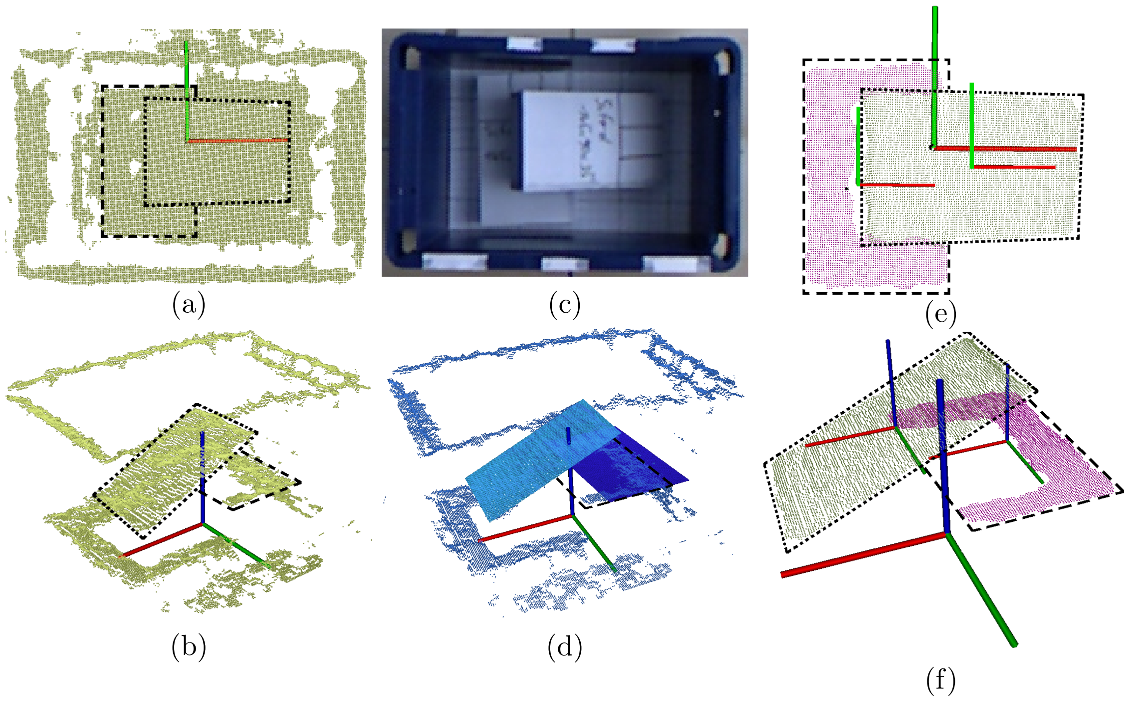


Figure 5.7.: The figure shows images of a standard detection scenario with 2 detectable boxes. The dashed rectangle indicates the bottom box and the fine dashed rectangle the top box surface. *(c)* container RGB image; *(a,b)* corresponding input point cloud of the container and its content; *(e,f)* 2 detected surfaces with its calculated centre points (small coordinate axes); *(d)* the ICP aligned templates; (coordinate systems [red, green, blue] → [x, y, z])

Despite the small shifting error at the lower surface this is a good detection result which can be used as input for the item manipulation without any problems.

5.4.2.2.2. Detection preventing occlusion

Similar to the previous scenarios this one also includes two boxes of type "A" (see Table 5.6), but in contrast the occlusion this time prevents a proper detection of the lower box surface. In Figure 5.8, (a,b) it can be seen that this time the occlusion produces a hole in the point cloud, splitting the representation of the lower box surface into two separate clusters. As shown in Figure 5.8, (d-f), the detection of the upper box is as good as in the previous scenario but the lower box surface it not detected at all. This is caused by the fact, that the detection's plane segmentation step properly segments the lower surfaces plane but the hole in the point cloud causes that the detection's euclidean clustering step separates it into two discrete candidates which will be rejected because their diameter and area are too small. This rejection of such occluded surfaces is deliberated because the visible surface parts are to small for a reliable grasp.

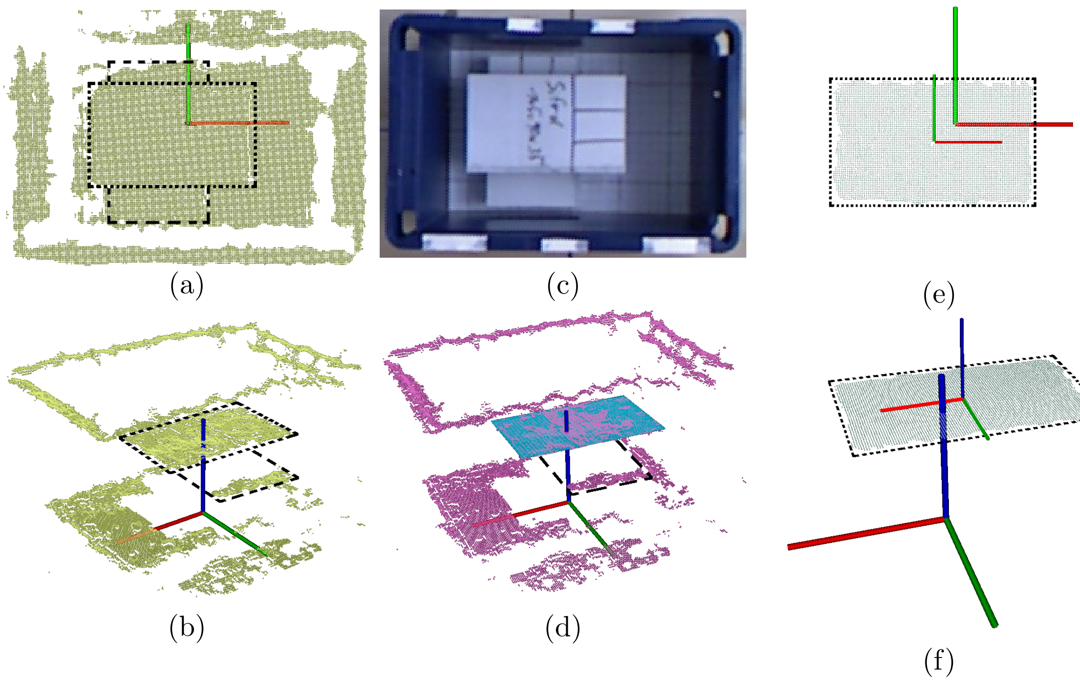


Figure 5.8.: The figure shows images of a standard detection scenario with 1 detectable box. The dashed rectangle indicates the bottom box and the fine dashed rectangle the top box surface. *(c)* container RGB image; *(a,b)* corresponding input point cloud of the container and its content; *(e,f)* 1 detected surface with its calculated centre point (small coordinate axes); *(d)* the ICP aligned template; (coordinate systems [red, green, blue] $\rightarrow$ [x, y, z])

5.4.2.2.3. False positive detection

Figure 5.9 shows an example scenario producing a false positive detection. The container content consist of 3 boxes of type "C" (see Table 5.6) placed upright side by side with their smallest surfaces on top. These small surfaces are indicated by the dashed rectangles in the input point cloud images (Figure 5.9, (a,b)).

Due to the limited resolution and a relative big measurement noise of the Kinect camera the three surfaces can not be identified in the point cloud what is well illustrated in Figure 5.9, (a,b). The result of this special box arrangement is a *combined* surface having similar dimension properties as the *target* surface the detection is actually looking for. Table 5.7 shows these properties as well as the actual area and diameter values estimated by the *detector*. The also shown ratios ($Detector/Target$) clarify the small difference between the detected and target surfaces values. These small differences in combination with the ratio thresholds used by the detector are the reason for the false positive detection of a target surface. Figure 5.9, (e,f) shows the detected surface and the calculated centre point.

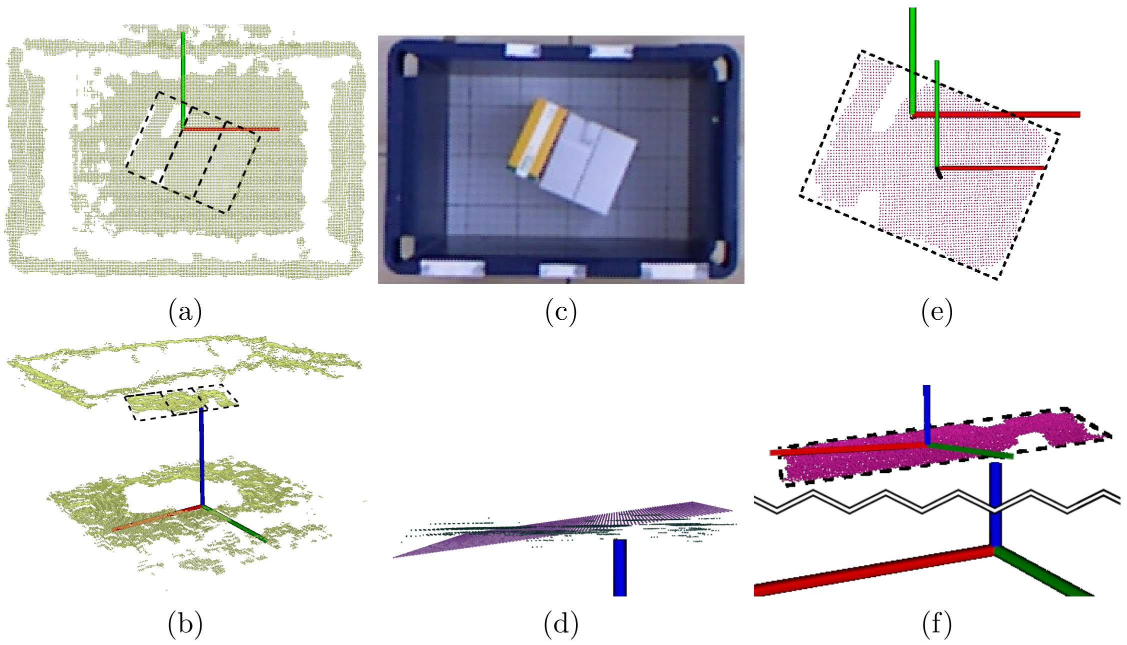


Figure 5.9.: The figure shows images of a detection scenario producing a false positive result. *(c)* container RGB image with 3 boxes upright, side by side; *(a,b)* corresponding input point cloud of the container and its content, top surfaces indicated by dashed rectangles; *(e,f)* 1 wrongly detected surface with its calculated centre point (small coordinate axes); *(d)* the ICP aligned template (slightly tilted); (coordinate systems [red, green, blue] → [x, y, z])

Surface	**Dimensions**	**Diameter**	**Area**
Combined	$93x71\,mm$	$117\,mm$	$6603\,mm^2$
Target	$97x71\,mm$	$120\,mm$	$6887\,mm^2$
Detector		$114\,mm$	$6922\,mm^2$
Ratios		0.950	1.005

Table 5.7.: The table shows surface and detection properties of the false positive detection scenario

Figure 5.9, (d) shows the ICP aligned surface template of the surface pose estimation. It can be seen that it is tilted which is also caused by the obvious noisy surface points.

Figure 5.10 shows another example scenario with a false positive result using boxes of type "A" (see Table 5.6). In contrast to the previous scenario, this time the problem is caused by the ICP separation attempt which comes into play when area and diameter of a surface candidate are to big. In this special arrangement it is not possible to separate the boxes via ICP because no target surface is visible. Figure 5.10, (d,e) makes a great weakness of the ICP algorithm obvious. As already mentioned in subsection 2.2.4, the algorithm requires a good initial transformation which is missing in this context. Therefore the ICP alignment often produces bad results leading to detection failure or more worse, to false positives like in this example .

False positive detected surfaces are more worse than not detected ones. The problem with such false positive detections as input for the following item manipulation is the great risk of a failed grasp. This could for instance happen if the gripper touches the edge between the boxes which would result in a failed, or even worse, unsafe grasp. An unsafe grasp would be more worse because it carries a greater risk of loosing the item. Loosing the item would possibly lead to damage of the item and would also require urgent human intervention.

5.4.2.3. Mixed container content

Figure 5.11 shows a scenario with mixed container content for testing the detections selectivity. Two boxes of type "A" and "C" (see Table 5.6) were placed inside the container and the detection was executed twice. The first time looking for box type "A" and the second time for type "C".

The result for the first run is shown in Figure 5.11, (e). It can be seen that the surface of the box is properly detected with a slight shift along the positive x-axis. In contrast the result when looking for box type "C" is quite bad as shown in Figure 5.11, (f). This is caused by the fact that the detections plane segmentation step returns bigger surfaces first which in this case is the surface of box "A". In the following this surface candidates will lead to an ICP segmentation attempt because they are too big. As already mentioned in paragraph 5.4.2.2.3, this segmentation does not work very well and often leads to false positive detections. Therefore it can be said that the detection is very selective towards surfaces that are smaller than those the detection is actually looking for and more or less not selective towards surfaces that are bigger than the expected one.

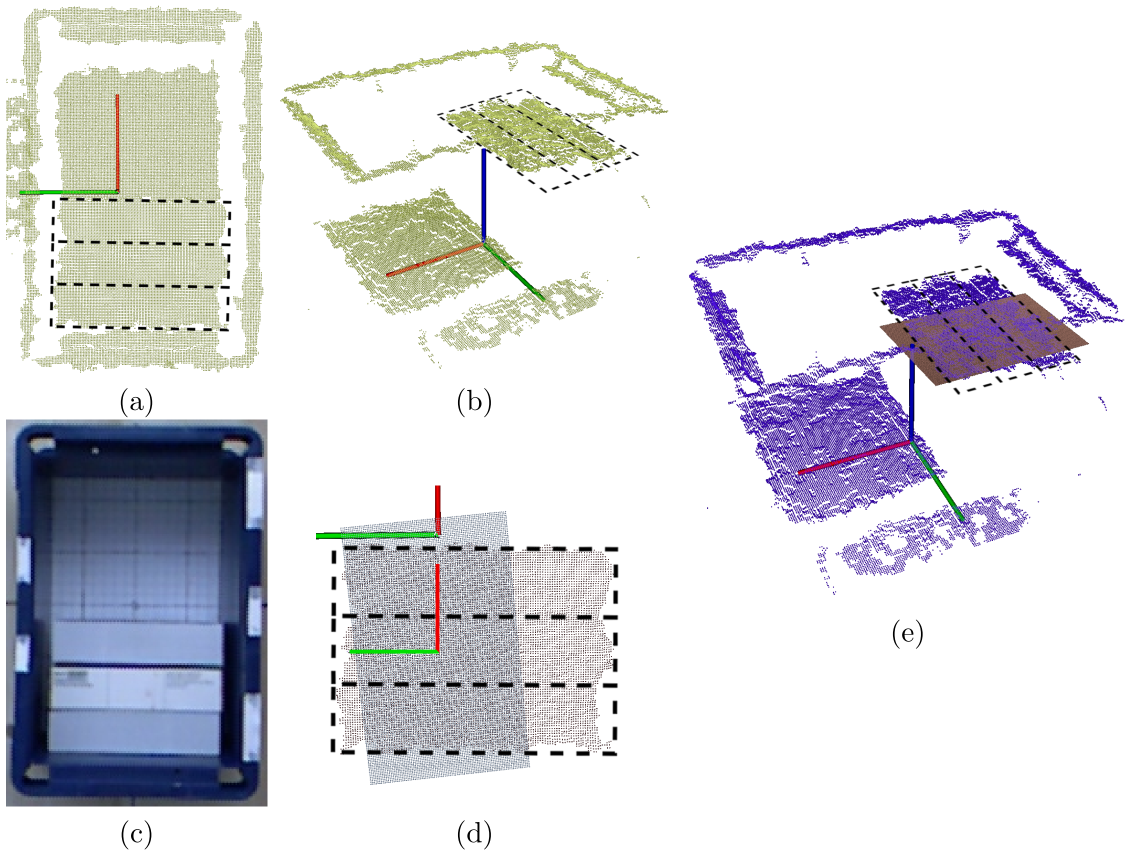


Figure 5.10.: The figure shows images of a detection scenario producing a false positive result caused by the ICP separation. The visible long side box surfaces are indicated by dashed rectangles. *(c)* container RGB image with 3 boxes upright (long side surface on top), side by side; *(a,b)* corresponding input point cloud of the container and its content; *(d)* surface candidate and ICP aligned template used for ICP separation; the intersection is wrongly detected as surface; also shown the calculated centre point; *(e)* container point cloud and ICP aligned template; (coordinate systems [red, green, blue] $\rightarrow$ [x, y, z])

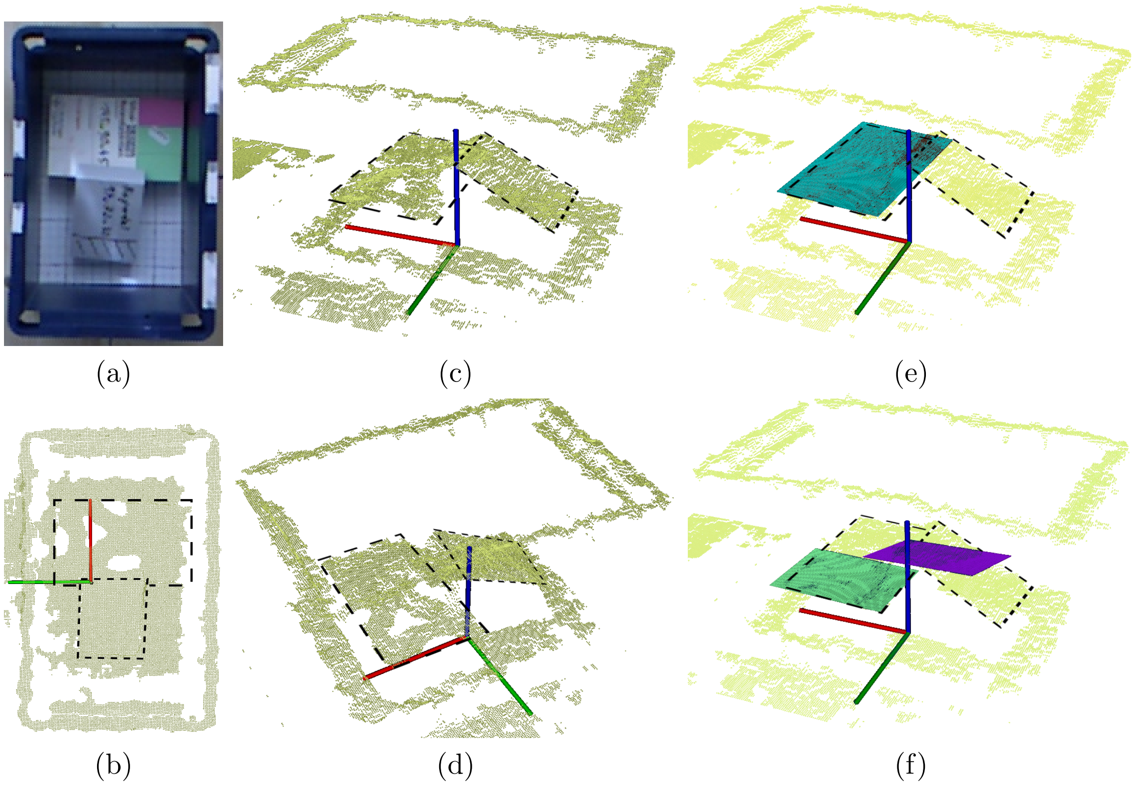


Figure 5.11.: The figure shows images of a scenario for testing the detectors selectivity. Visible box surfaces are indicated by dashed rectangles. Normally dashed: box type "A"; Fine dashed: type "C"; *(a)* container RGB image 2 boxes of type "A" and "C"; type "A" flat on the container bottom; *(b-d)* corresponding input point cloud of the container and its content; *(e)* good detection result for box type "A"; *(f)* wrongly detected surfaces when looking for box type "C"; (coordinate systems [red, green, blue] → [x, y, z])

5.5. Item manipulation

This experiment was designed to test the item manipulation hardware and software modules. Of special interest was the accuracy of the arm navigation while picking an object with an overhead grasp and the reliability of the whole system when moving the item from the storage container to the order bin. It was also part of the experiment to determine the area for grasp points inside the container not leading to collisions between the arm's end effector and the container.

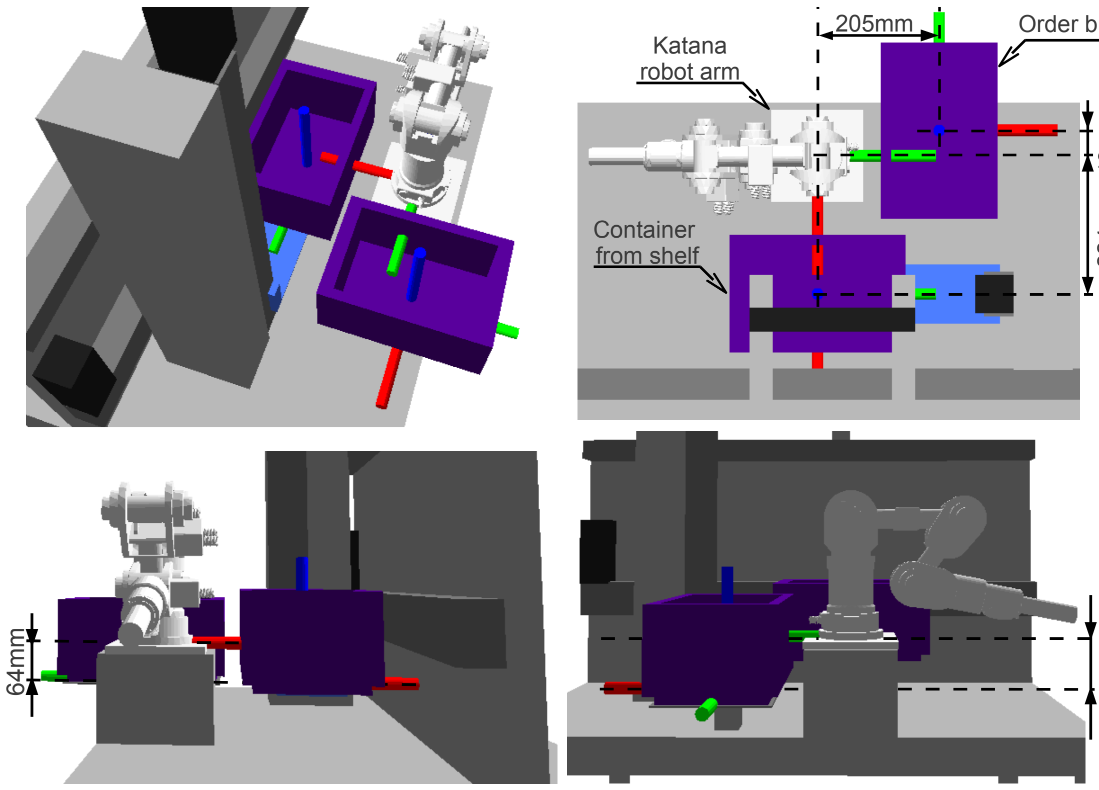


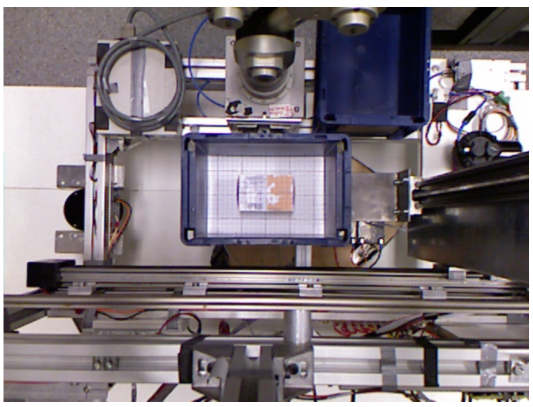
Figure 5.12.: The figure shows images of the important parts of the object manipulation experiment and actual distances between coordinate frames used during the experiment. (coordinate systems [red, green, blue] → [x, y, z])

5.5.1. Setup & execution

For the experiment the setup shown in Figure 5.12 was used. The fork of the container manipulation system was moved to its reference position and a container was placed at the desired optimal position on the fork. During the experiment the goal positions for the vacuum gripper's tip were defined relative to a coordinate frame located at the container's bottom centre hereinafter referred to as *"container frame"*. In the robot's URDF description this *"container frame"* is defined relative to the *"katana_base_link"* frame which is at the top centre of the Katana arm's base plate (see Figure 5.12). The translation according to the used URDF robot description of the *"container frame"* relative to the *"katana_base_link"* frame is:

$$\text{Translation: } [x \quad y \quad z]^T = [224 \quad 0 \quad -64]^T; \text{ Unit: [mm]}$$

This position of the frame is directly in front of the arm and in the same plane as the forks upper side and approximately centred related to the forks longitudinal axle. The container's bottom and some test boxes with different dimensions were provided with a grid for measuring the gripper's contact point on the box and the position of the box relative to the *"container frame"* to get the actual position compared to the given goal position. A comparison is only possible for the x- and y-direction because under pressure the fork yields in z-direction which prevents proper measuring of the position along the z-axis. An image of a typical scene is shown in the left part of Figure 5.13. During the experiment such images were automatically acquired with the Kinect camera mounted straight above the container before starting the manipulation process.



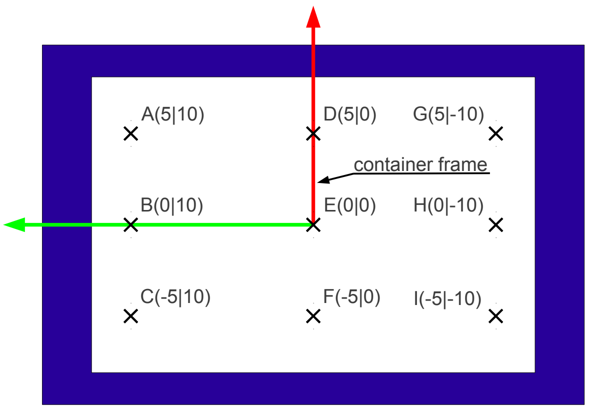


Figure 5.13.: *Left:* Image acquired with Kinect camera looking straight downwards on the scene of the item manipulation experiment; *Right:* Container (border in blue), attached coordinate frame and goal positions (x) for the experiment; Position (x|y) in [mm]; (coordinate system [red, green] → [x, y])

For each run a box, or for higher grasp points a stack of boxes, was placed inside the container. Simulating a perfect detection the top surface centre point was then used as grasp point and sent to the item manipulation module. The module itself has been slightly modified for this experiment by adding breaks after some steps. For instance, when the arm reached the grasping point the module stopped the process until the distance between the desired grasp point and the actual gripper tip position and the arm's joint angle values were noted. At the end of each run the manipulation modules response and additional information about special incidents were also noted for later evaluation. The full action definition of the manipulation module (described in subsection 4.3.7) including the response message is shown in Listing 4.13. The response reports if the module has successfully picked (*box_picked* variable) the box from the container, has successfully placed (*box_placed* variable) the box in the order bin and if the arm successfully returned to its initial position (*back_to_init_pos* variable). It also reports an *error code* giving additional information about possible problems during the process.

5.5.2. Results and evaluation

At the beginning a few runs were executed to find the allowed area inside the container. It was found out that due to the dimensions of the arm's end effector a minimum distance of the used goal points to the containers border of about $3\,cm$ along the x- and about $2\,cm$ along the y-axis has to be adhered to avoid collisions between the arm and the container.

For the second part of the experiment different goal positions along the x- and y-axes, as shown in the right part of Figure 5.13, were used. These positions are covering the whole reachable area inside the container and the same positions were used for four different levels along the z-axis. The level heights are given by the thickness of the three used target box types.

Level heights: 2.0; 4.0; 6.5; 10.5 (Offset: 6.5; Box: 4.0); Unit: [cm]

This combination leads to a total number of 36 gripper tip goal positions. For each position three runs were performed, except if the first run showed up that the goal position is out of the arm's range.

In general, it can be said that the item manipulation works quite well as long as

- the given goal point actually lies on the box surface.
- the given goal point is within the allowed and reachable area.
- the box does not slip away when being touched.
- the box is thin enough, not to collide with the container border during transfer.

In the following some incidents, observations and occurred errors as well as their causes, or rather possible causes, are reported.

Grasp position aberration

In all runs the actual gripper tip position in relation to the desired grasp point position was noted. As already mentioned above it was only possible to measure the distance along the x- and y-axis. Due to the used measurement method the error could only be determined with an accuracy up to $5\,mm$ but it could be seen that the error was in the same range in all runs. One could also see that it was exact the same when repeating the run for a certain goal point. The reason for this position aberration is highly probable a combination of two things. The first part is a slight misalignment of the container relative to the arm, producing a constant translational error as given below.

Translational grasp point position error: $dx = -10\,mm$; $dy = -5\,mmm$

The second part is a small inaccuracy of the Katana arm's joint calibration. During the experiment it could be seen, that the arm's end effector was never looking straight downwards when performing the overhead grasp. The gripper's tip was always slightly pointing towards the arm's base which indicates a problem with the joints "2" and "3". In this case joint "0" being the pan joint at the arm's base (see also Figure 4.43).

Out of range error

If the goal is out of range, is indicated by the error code "3" (*TOUCH_BOX_ERROR*) in the action response (definition given in Listing 4.13). This error occurred only two times during the experiment. It turned out, that the positions "A" and "G" (see Figure 5.13, right part) in combination with a grasp point height of 10.5 *cm* were outside the reachable area of the arm when trying to perform an overhead grasp.

Box not picked

This case is indicated by the error code "5" (*BOX_NOT_PICKED_ERROR*) and the responses field *box_picked* set to *"False"*. During the experiment this case occurred several times when a wrong grasp points were entered and the vacuum gripper missed the box. This case could also arise if the box slides away when being touched by the gripper which can easily happen if the desired box lies inclined on another one.

Box lost

The boxes with 4.0 *cm* and 6.5 *cm* in height produced collisions with the container border when moving the arm towards the order bin. This is the result of keeping the end effector oriented downwards resulting in a relatively small distance, a little bit less than 4 *cm*, between the containers border and the gripper's tip. For the 4.0 *cm* boxes the collision caused a loss of the box in about 50% and for the 6.5 *cm* boxes in about 95% of the runs. The suction power of the used vacuum gripper and vacuum pump is quite high and is usually sufficient to lift a weight of about 500 *g* which is also the reason that not every collision leads to a loss of the box. What actually happens when a box is lost is, that the gripper slides on the boxes surface until it slips of. Before dropping the box into the order bin the item manipulation module checks the pressure sensor to detect if the box was lost during the transfer and if this is the case this is indicated in the action result by the error code "8" (*BOX_LOST_ERROR*) and the field *box_placed* set to *"False"*. As mentioned in subsubsection 4.3.5.2 this problem should be avoided by attaching a collision model of the grasped box to the arm so that the arm navigation can include it in the path planning process and avoid collisions. But as also mentioned all efforts to get this feature to work had been in vain.

Arm collision detected

Whenever an error occurs during the item manipulation process recovery behaviours will be performed but the last step in every case is the try to move the arm back to its initial pose. This is important because this pose is safe and stable. Safe means that the container manipulation unit can operate without the risk of collision. In this context "stable" means that in case of power loss this pose guaranties that the arm will not slump down, possibly leading to damaged. If it is not possible to move the arm back to its initial pose this is indicated by the result's field *back_to_init_pos* set to *"False"* and the error code "9" (*INIT_POS_ERROR*). In this case the whole robot operation should be stopped and human intervention should be requested to avoid damages of the hardware.

During the experiment this error occurred only twice but in fact this is a fatal error because it prevents the robot from accomplishing its function. In both cases the problem appeared after a successful placement of the box in the order bin when moving the arm back to its initial pose. When performing this movement the arm's gripper tip came close to the motor of the container manipulation unit's vertical axis which caused the arm navigation to stop to avoid a possible collision.

This happened because the arm navigation stack continuously checks for possible collisions during arm movements and if the arm undershoots the given minimum distance to some part of the environment it will be stopped. In such a case the item manipulation will start recovery behaviours but in this special case they have to fail and cause the abort of the whole picking process.

In such a case, all implemented recovery behaviours, including movements with disabled collision monitoring (described in paragraph 4.3.7.1.2), will fail because the minimum distance to the environment has already been undershot. This causes the arm navigations path planning process to fail which is not affected by the disabled collision monitoring because this setting only effects the live collision monitoring during movements. This problem could be avoided by finding a way to set a greater distance threshold for the planning process than for the collision monitoring during movements. Another way would be to identify the links with possible collisions and allow collisions between these links for the recovery movement planning process.

Chapter 6.

Conclusion and future work

In this thesis a concept and a prototype system for an autonomous mobile order picking robot is presented. The thesis also reports experimental results concerning the different presented prototype hardware and software modules. The developed prototype hardware system uses off the shelf components except the omnidirectional drive and parts of the vacuum gripper system. For the prototype software a Linux based environment and the popular ROS (Robot Operating System) as well as other up to date libraries, tools and packages like PCL (Point Cloud Library) and OpenRAVE (Open Robotics Automation Virtual Environment) were used. Wherever possible, the used software is free and open source.

In general, it can be said that the developed prototype system, following the presented concept, is in principle capable of performing autonomous order picking even if there is still a lot of work to do before reaching a state ready for productive operation. To only name a few things:

- The hardware, especially the mobile base, has to be re-engineered
- Integration into a real warehouse management system
- Integration of fleet management capabilities
- Improvement of the developed software modules (reliability, runtime, etc.)
- Optimization of all parts with respect to energy efficiency
- etc.

In the following the system main parts will be discussed in detail.

Mobile base

The fundament for the whole system is the mobile base. On the hardware side it is consisting of the omnidirectional Krikkit drive, the coupled self supporting frame and a laser scanner as navigation sensor. During the development and the following experiments several problems where discovered. The first problem is caused by the ball casters of the self supporting frame which have a very small contact surface but carry all the weight. Especially on soft floor coverings this leads to a very high resistance and prevents the robot from moving. The second problem, already reported in paragraph 4.3.4.1.1, comes from the small Krikkit drive in combination with the large and heavy frame which makes a proper operation of the autonomous navigation impossible. Nevertheless it could be shown that the prototypes mobile base with all its issues is still able to reach goals with high accuracy. To solve the mentioned problems the prototypes mobiles base should be replaced with a real omnidirectional drive directly carrying all other required components.

Different systems and projects are already present like the "KUKA omniMove" platform (see `http://www.kuka-omnimove.com`, 2013) or the "OmniRob" platform developed at the University of Applied Sciences and Arts in Dortmund and used for the project presented in [23] to name just two. Another aspect to keep in mind when re-engineering the mobile base is the diameter. A smaller diameter is better for the moveability because the ROS navigation stack uses the robot's circumcircle as convex collision model for motion planning and therefore a smaller diameter will make turns easier. In contrast, a smaller diameter reduces the footprint which increases the risk of tilting.

On the mobile base software side there are the drivers for the Krikkit drive and the laser scanner as well as the ROS navigation stack for autonomous navigation on a known environment map and the landmark based fine positioning system. As reported in section 5.1 and section 5.2 both navigation modules performed quite well under the given experimental conditions. Due to the good performance of the navigation stack with the single Krikkit drive it can be assumed that it will do as well for a bigger drive with better suspension characteristics like one of the two mentioned in the paragraph above. Future research should evaluate the capabilities of the navigation modules under more realistic conditions like in a real warehouse. Especially the performance of the localization in an environment with many similar looking rack aisles. For the fine positioning there should also be a possibility to check if the robot is in front of the correct shelf place which could be achieved easily by a visual tag like a QR-code or something similar. In future research it could also be evaluated, if a new *move_base* node for the navigation stack, combining autonomous navigation and landmark based fine positioning, would have valuable advantages compared to the system presented in this work.

Container manipulation

The container manipulation unit, designed as fork lift like apparatus, is relatively simple but the prototype hardware implementation has several issues. First of all it requires a really precise positioning in front of the shelf, reported in section 5.3, which in turn requires a more precise positioning system and mobile base. Another issue of the used system are the two horizontal slide rails which lead to jerky motions when carrying load on the fork. This becomes worse if the slide rails are not exactly parallel aligned what is almost impossible. The forks implementation itself is also a problem especially in combination with jerky horizontal movements. The prototypes fork is a simple bent steel plate and therefore very flexible. This leads to oscillations with observed amplitudes up to $5\,cm$ at the forks end. Similar to the fork, the connection between the vertical and horizontal axis is also a made of bent sheet and also quite flexible. Due to this problems the movement speed had to be limited dramatically what increases the needed time for container manipulations. Another problem would be collisions of the fork with parts of the environment which could cause deformations especially of the flexible fork and axis connection part.

The container manipulations software, described in subsection 4.3.3, is part of the work presented in [20] and consists of two nodes. In the *"Hardware abstraction" layer* (see Figure 4.4) there is the *nanotecContr* node which is responsible for the communication between the PC and the Nanotec control units. This node is not only associated with the container manipulation module because also the *"Vacuum gripper system"* uses the node to control the vacuum pump and to check the pressure sensor which are both connected to the digital I/O ports of the control unit responsible for the horizontal axis. During all tests, the node performed very well with only a single issue. If the serial connection is lost and the node tries to send some data, the node is terminated due to an unhandled exception. This is not a big problem because during the tests this only occurred when the connection was physically broken, what in every case requires human intervention and a restart of the robot's software. Nevertheless this problem should be solved. Maybe it would also be a good idea to implement an automatic reconnection. This would allow for continuing operation without restarting the robot's software after the connection problem is fixed.

The second node belonging to the container manipulation is the *mechatronicsContr* node from the *"Services & actions" layer* (see Figure 4.4). The node is configurable via several ROS parameters which are fetched at start up. This node contains a state machine, handling the whole container manipulation process. The implemented ROS action server is quite simple and offers three commands for fetching a container from the shelf, placing a container on the shelf and moving the manipulation unit to its reference position. For starting the container manipulation process, a goal, containing a command and a shelf level number, is sent to the action server. The different level heights and other important parameters (see Listing 4.7) are predefined via a configuration file which is loaded at start up. This approach is very unflexible and especially the fixed level heights and distances for horizontal movements are causing problems. Due to the fact that the gap between containers in neighbouring levels is really small, the level heights have to be defined quite precise so that the fork can move into this gap without collisions. This is already challenging under optimal conditions but it is impossible, if the floor is not absolutely even. The fixed distances for horizontal fork movements produce the problem, that it is not possible to balance differing distances to the shelf. The actual distance to the shelf could be easily estimated via the landmark tracker during the fine positioning process. To overcome the level heights problem it would be necessary to detect the gap between vertically neighbouring containers which is the target for the fork. This could be achieved via a camera system at the forks tip that is able to track the target containers bottom edge. So only approximate level heights would be necessary and the actual height to move the fork under the container could be determined exactly for each shelf place. It would also provide a possibility to verify that the right container will be fetched. Therefore the containers could be labelled with bar codes or other camera readable tags.

Another field for future work would be to replace the fork lift like container manipulation unit with a robot arm. On the one hand this would increase the system's complexity, which is already very high, but on the other hand it would also increase its flexibility. The most important advantage would be the ability to balance positioning errors in front of the shelf. In contrast to the fork lift a robot arm would be able to balance especially angular positioning errors in a wider range. This in turn would reduce the requirements for the fine positioning system. For using a robotic arm it would be necessary to detect the container's position and orientation so that it can be grasped. Similar to the approach described before this could for instance be achieved via a camera system able to track a tag on the container front and an iterative grasping approach. Another idea would be to properly detect the container's pose relative to the arm using 3D sensor information and perform a single grasp.

Object detection

The object detection module uses the point cloud data from the Kinect camera and a relatively simple approach to detect the container and the surfaces of its content and to find possible grasp points for the following item manipulation step. As shown in section 5.4 a lot of things are influencing the detection and cause the used detection approach to fail. A special problem during this work was the sensor noise of the used Kinect camera which is relatively high compared to the size of the objects to be detected. Another problem is the fact that the used detection approach does not allow for determining the objects orientation which would be required for collision checking during the arm path planning process. Determining the objects orientation would also be necessary if another gripper system, a hand like gripper for instance, should be used. Because of the problems and limitations of the used detection approach a lot of work has to be done to get a reliable productive detection system.

The detection offers a wide field for future research. Different sensing technologies, like stereo vision or 3D laser scanner, as well as new 3D data analysing methods should be evaluated for their capabilities to solve the detection problems mentioned in the paragraph above. The surface detection approach, using curvature analysis, described in [22](page 90 ff, chapters 6.3, 6.4 and 6.5) and mentioned in section 3.3 is also a good candidate for future research but will require a more precision 3D scanner than the used Kinect camera. Also of interest could be new features of the PCL (Point Cloud Library), using additional information like color values to extend traditional 3D data analysis approaches.

Item manipulation

The item manipulation uses a robot arm with 5 DOF (Degrees Of Freedome), equipped with a vacuum gripper system on the hardware side and the ROS arm navigation stack and other software packages (see subsection 4.3.5 and subsection 4.3.7) on the software side. As already mentioned in section 5.5, the item manipulation worked quite well as long as some basic conditions are met. However, there are still some issues to be resolved.

The first discovered problem is the fact that with the used version of the ROS arm navigation stack, responsible for motion planning, it is not possible to execute pose goals. It seemed to be a problem of the inverse kinematics plugins with the Katana robot arm, or rather arms with less than 6 DOF in general (see subsubsection 4.3.5.2). The developed workaround works for the implemented overhead grasps approach but is not a general solution for the problem. Another problem with the ROS arm navigation stack, also described in subsubsection 4.3.5.2, is the not working collision avoidance for grasped objects. During the experiment, discussed in section 5.5, this provoked collisions between the carried object and the container border but in general collisions of the carried object with the environment can not be excluded. Both problems should hopefully be solved in the newest version of the ROS arm navigation stack but due to some major changes in the architecture of the arm navigation stack and its communication paths plenty of work has to be done to port the developed modules to work with the new version of the arm navigation stack. A third issue in connection with the implemented overhead grasp approach became obvious during the tests. The overhead grasp's condition of keeping the gripper in a straight vertical orientation significantly reduces the reachable area especially along the z-axis. During the experiment this lead to the *"Out of range error"* reported in section 5.5. A possible workaround for this problem without replacing the robot arm would be to lift or lower the fork of the container manipulation unit if it is needed, which would require a corresponding adaptation of the container manipulation software.

One outcome of the experiment and preceding tests is the finding, that a vacuum gripper is a simple but also very efficient and convenient way for grasping boxlike objects as used during this work. Nevertheless it limits the field of possible objects to those with more or less even and gas-tight surfaces. For the future it is the aim to be able to detect and grasp more arbitrary objects. To reach this goal it will be necessary to replace the vacuum gripper with a handlike one. This would increase the flexibility but naturally also the requirements for motion and grasp planning and in turn for object recognition.

Robot control & warehouse management
The top level of the robot's on-board software is the *"Overall robot control"* responsible for triggering the necessary actions in the right sequence to fulfil the task of order picking. On the off-boar side is the warehouse management system responsible for managing customers orders, sending them to the robots and so on.

The prototypes robot control node, described in subsection 4.3.8, only implements the basic work cycle with almost no fault handling and is far away from the required reliability for productive operation. Therefore the node has to be re-engineered with special respect to safety and fault recovery capabilities.

The implemented warehouse management node, described in subsection 4.3.9, was only designed for testing and demonstration purposes. For operative use a full integration into, or at least an interface to a real warehouse and order management system is required. The extension of common warehouse management systems for handling and coordinating a fleet of mobile order picking robots offers great potential for future research and development.

Appendix A.

Abbreviations

AIMM Autonomous Industrial Mobile Manipulation
CAN Controller Area Network
USB Universal Serial Bus
IPC Inter-Process Communication
GUI Graphical User Interface
DOF Degrees Of Freedome
ROS Robot Operating System
ICP Iterative Closest Point
PCL Point Cloud Library
OMPL Open Motion Planning Library
RRT Rapidly-Exploring Random Trees
KDL Kinematics and Dynamics Library
Orocos Open Robot Control Software
URDF Unified Robot Description Format
SRP Schäfer Robo-Pick
SVD Singular Value Decomposition
PRM Probabilistic Roadmap
SIFT Scale Invariant Feature Transform
WLAN Wireless Local Area Network
SLAM Simultaneous Localization and Mapping
SMACH State Machine
OpenRAVE Open Robotics Automation Virtual Environment
IK Inverse Kinematic
PC Personal Computer
RPC Remote Procedure Call
CAD Computer-Aided Design

Bibliography

[1] *Towards Reliable Grasping and Manipulation in Household Environments*, New Delhi, India, 12/2010 2010.

[2] R. Bohlin and E.E. Kavraki. Path planning using lazy prm. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 1, pages 521–528 vol.1, 2000.

[3] Christian Matt and Daniel Wimmer. KomBot - autonomer Kommissionierroboter. Technical report, Institute of Logistics Engineering - Graz University of Technology, 2013.

[4] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 2012. To appear.

[5] Florian Ehrentraut and Christian Landschützer and Dominik Lechner and Christian Matt and Wolfgang Pichler and Bernhard Puchinger and Gerald Steinbauer and Daniel Wimmer. Kombot - An Autonomous Mobile Order Picking Robot. 2012.

[6] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *Robotics Automation Magazine, IEEE*, 4(1):23–33, 1997.

[7] W. Gellert, S. Gottwald, M. Hellwich, H. Kästner, and H. Küstner. Analytic geometry of space. In W. Gellert, S. Gottwald, M. Hellwich, H. Kästner, and H. Küstner, editors, *The VNR Concise Encyclopedia of Mathematics*, pages 530–547. Springer Netherlands, 1990.

[8] G. Grisetti, C. Stachniss, and W. Burgard. Improved techniques for grid mapping with rao-blackwellized particle filters. *Robotics, IEEE Transactions on*, 23(1):34–46, 2007.

[9] Martin Günther and Henning Deeken. Katana driver ros stack documentation. `http://www.ros.org/wiki/katana_driver`, 2011.

[10] Dirk Holz, Stefan Holzer, Radu Bogdan Rusu, and Sven Behnke. Real-time plane segmentation using rgb-d cameras. In *RoboCup Symposium*, 2011 2011.

[11] Howie Choset and Kevin M. Lynch and Seth Hutchinson and George A Kantor and Wolfram Burgard and Lydia E. Kavraki and Sebastian Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations.* MIT Press, Cambridge, MA, June 2005.

[12] Advait Jain and Charles Kemp. El-e: an assistive mobile manipulator that autonomously fetches objects from flat surfaces. *Autonomous Robots*, 28:45–64, 2010. 10.1007/s10514-009-9148-5.

[13] E. Gil Jones. Ros arm navigation stack documentation. `http://www.ros.org/wiki/arm_navigation`, 2012.

[14] Kurt Konolige. A gradient method for realtime robot control. In *International Conference on Intelligent RObots and Systems - IROS*, 2000.

[15] Soon-Wook Kwon, Frederic Bosche, Changwan Kim, Carl T. Haas, and Katherine A. Liapi. Fitting range data to primitives for rapid local 3d modeling using sparse range point clouds. *Automation in Construction*, 13(1):67 – 81, 2004.

[16] Steven M. LaValle, James J. Kuffner, and Jr. Rapidly-exploring random trees: Progress and prospects, 2000.

[17] Sukhan Lee, Jaewoong Kim, Moonju Lee, Kyeongdae Yoo, L.G. Barajas, and R. Menassa. 3d visual perception system for bin picking in automotive sub-assembly automation. In *Automation Science and Engineering (CASE), 2012 IEEE International Conference on*, pages 706–713, 2012.

[18] Mads Hvilshoj and Simon Bogh. "Little Helper" - An Autonomous Industrial Mobile Manipulator Concept. *International Journal of Advanced Robotic Systems*, 2011.

[19] Michael Wild. Recent Development of the Iterative Closest Point (ICP) Algorithm. Technical report, Swiss Federal Institute of Technology Zurich, 2010.

[20] Bernhard Puchinger. Automatisierung eines Ein/Aus-Lagerungsmechanismus für Kisten in der Logistik. Technical report, 2012.

[21] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[22] Radu Bogdan Rusu. *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD thesis, Technische Universität München, 2009.

[23] Röhrig, C. and Hess, D. and Kirsch, C. and Künemund, F. Localization of an omnidirectional transport robot using ieee 802.15.4a ranging and laser range finder. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 3798–3803, Oct.

[24] Radu Bogdan Rusu, Andreas Holzbach, Rosen Diankov, Gary Bradski, and Michael Beetz. Perception for mobile manipulation and grasping using active stereo. In *Humanoids*, Paris, 12/2009 2009.

[25] Gildardo Sánchez and Jean-Claude Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In RaymondAustin Jarvis and Alexander Zelinsky, editors, *Robotics Research*, volume 6 of *Springer Tracts in Advanced Robotics*, pages 403–417. Springer Berlin Heidelberg, 2003.

[26] Schäfer itself. *Automatisierte Systeme in der SSI Schäfer Gruppe*. Schäfer itself.

[27] R. Smits. KDL: Kinematics and Dynamics Library. `http://www.orocos.org/kdl`, 2012.

[28] Siddhartha Srinivasa, Dave Ferguson, Casey Helfrich, Dmitry Berenson, Alvaro Collet, Rosen Diankov, Garratt Gallagher, Geoffrey Hollinger, James Kuffner, and Michael Weghe. Herb: a home exploring robotic butler. *Autonomous Robots*, 28:5–20, 2010. 10.1007/s10514-009-9160-9.

[29] Gerald Steinbauer, Mathias Brandstötter, Martin Buchleitner, Stefan Galler, Simon Jantscher, Martin Mörth, Gerald Krammer, Jörg Weber, and Martin Weiglhofer. Mostly Harmless Team Description 2006 - Robust Control of Mobile Robots. In *International RoboCup Symposium*, Bremen, Germany, 2006.