Master's Thesis

An Integral Mobile Robot Platform for Research and Experiments in the Field of Intelligent Autonomous Systems

Máté Wolfram

Graz, May 2011

Supervisor: Univ.Prof. Dipl.Ing. Dr.techn. Franz Wotawa
Assessor: Univ.Prof. Dipl.Ing. Dr.techn. Franz Wotawa

Institute for Software Technology
Graz University of Technology

Acknowledgements

This master's thesis was authored at the Institute for Software Technology at Graz University of Technology. Special thanks goes to everyone who supported the project and helped me out with words and deeds, in particular to my supervisors Prof. Franz Wotawa and Dr. Gerald Steinbauer.

I'd like to thank my teammates from the RoboCup team KickOffTUG for their support and motivation and for several years of inspiring collaboration, as well as Reinhard Günther, for sharing his knowledge in the field of computer vision and Roland Angerbauer, for helping me out with the once-in-a-while operation of the soldering iron.

I'd especially like to thank my parents, for all the educational groundwork, for backing me, motivating and believing in me. I'd also like to thank all my friends that made an important contribution to who I am today and last but not least my girlfriend Lisa, who inexhaustibly supported me in times of hard work, and helped me find enough time to rest my mind.

Kurzfassung

Autonome mobile Roboter nehmen einen immer größer werdenden Stellenwert ein und gehören einem Forschungsfeld an, welches dank der Verfügbarkeit von kompletten Roboterplattformen und leistbaren Sensoren, sowie einer blühenden Community hinter Open Source Robotersoftware einem immer breiter werdendem Spektrum an Interessenten zugänglich ist.

Der praktische Teil dieser Arbeit beschreibt die Implementierung und Konfiguration eines Lieferroboters basierend auf dem Robot Operating System (ROS), welcher in geschlossenen Räumlichkeiten navigieren und Transportaufgaben lösen kann und dabei in der Lage ist, durch Ausführungs- sowie Sensorfehler, Fehler in der Wissensbasis oder exogene Ereignisse verursachte Inkonsistenzen in seiner Auffassung des Weltzustands zu detektieren und zu korrigieren.

Um die ihm auferlegten Ziele effektiv und verlässlich erreichen zu können, muss der Roboter imstande sein, klassische Problemstellungen der Robotik, wie Lokalisierung, Navigation, Hindernisvermeidung, Wissensrepräsentation und Objekterkennung zu bewältigen. Diese Themen werden im ersten Teil der Arbeit theoretisch aufbereitet, bevor sie im Kontext des Roboters anwendungsbezogen beschrieben werden. Abschließend zeigen zwei Experimente die Funktionstüchtigkeit und Einsatzfähigkeit des Roboters.

Abstract

Autonomous mobile robots are becoming more popular and sought-after every day. Thanks to the availability of complete robot platforms and affordable sensors, as well as a flourishing community developing open source robotics software, this field has become accessible to a wider spectrum of interested people.

The practical part of this thesis describes the implementation and configuration of a robot platform based on the Robot Operating System (ROS), capable of carrying out delivery tasks in indoor environments, as well as detecting and dealing with various kinds of execution and sensing failures, erroneous knowledge and exogenous events.

To achieve its goals in an effective and dependable manner, the robot must be capable of solving traditional problems in robotics, such as localization, navigation, obstacle avoidance, belief representation and object detection. Before discussing the application of these methods to our robot, they are explained in further detail in the theoretical part of this document. Two experiments demonstrating the robot's capabilities conclude this thesis.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Mobile autonomous robotics is a thriving field of research, incorporating a vast amount of subtopics including knowledge representation, sensing and vision, artificial intelligence, path planning, robot dynamics, localization etc.

As of today, it's valid to state that it's an area anyone can participate in, regardless of age, gender or provenance. A glance into the RoboCup[1] domain reveals numerous subleagues that concentrate on different problems of robotics and address different interest groups. This diversity makes it clear that interest is the only requirement for getting in touch with robotics.

The Robot Operating System (ROS) (developed at WillowGarage [2]) offers many modules and mechanisms necessary for implementing autonomous agents, in an easy to understand, yet powerful system. It's an open source framework that's designed to make progress in the robotics field available to the public, thus encouraging collaboration between researchers and avoiding unnecessary re-inventions of the wheel. It eases the first steps in robotics but also allows advanced developers to dive into selected libraries and improve them.

Not only robotics software, but also ready-to-go hardware is available and becoming increasingly affordable. Such platforms comprise mobile robots such as the Pioneer series, quadcopters equipped with GPS sensors, cameras and IMUs, as well as humanoid robots[3], to name only a few. Sensors are also becoming cheaper and more powerful, offering complete, easily integrable solutions. One example is the Microsoft Kinect (described in section 3.1.5 on page 69, an inexpensive visual 3D sensor based on structured light, that's an extension to the XBox 360. Out-of-the-box, it provides 3D spatial information, as well as human skeleton estimation and is therefore highly popular in robotics research.

Robots are more and more becoming part of our lives, not only in industry but also in the private sector. They are becoming increasingly smart, inexpensive and safe. While a simple robotic vacuum cleaner used to drive around randomly a few years long ago, its successor today is capable of fulfilling navi-

[1]http://www.robocup.org/
[2]http://www.willowgarage.com
[3]http://www.aldebaran-robotics.com/en

gation tasks using images of the ceiling as sensory input for localization. Mobile robots can carry out delivery tasks in offices or storage spaces and guide us through museums [TBB$^+$99]. Some of them are prepared to navigate in outdoor settings [IB06], not even necessarily on our planet [TO06]. Not only can robots make our lives easier and more comfortable, but some of them might even save them by locating and rescuing victims from a disaster area. Children have a different view of Lego®today than one or two decades ago. Back then they'd have proven themselves architectural geniuses, today they can additionally find out whether they'll become tomorrow's robotics researchers.

This growing need for and interest in robots in general yields an increasing focus on this topic at numerous educational establishments, including the University of Technology in Graz, that has a strong background in the RoboCup domain, and hosts various other projects in the field of robotics, such as the one presented in the practical part of this document. The theoretical part, on the other hand, is designed to convey an understanding of various subtopics of the robotics domain and to present the Robot Operating System (ROS), that was applied in the project. Before starting, let's have a look at a selected subtopic, namely the robot's belief and knowledge of the real world, and how this challenge is met at the Technical University of Munich.

1.1 KnowRob

Autonomous personal robots require mechanisms for acquiring, managing and reasoning over knowledge, to be able to do *the right things at the right time* [TB09]. KnowRob is a framework developed at the Intelligent Autonomous Systems[4] lab of the Technical University of Munich, that provides autonomous agents with the aforementioned capabilities. It uses encyclopedic knowledge in combination with information about possible actions to decide which tasks the robot is capable to perform at which locations, and to infer subtasks that have to be carried out. The underlying knowledge is represented in an OWL ontology that is inspired by Cyc [TB09] and extended manually to provide richer detail for relevant topics. KnowRob is based on SWI Prolog and therefore allows to query the ontology and to add knowledge using Prolog queries. The parameters to these queries are RDF triples (subject, predicate and object). For easy integration with ROS, there are accessor implementations in C++, Java, Python and Lisp as well [Ten10].

Perceiving and Learning from the Real World. Equipped with a sensor that delivers 3D point cloud information, an agent is capable of segmenting the spatial information into separate objects and classify them into categories of the underlying knowledge base. The robot can then decide how the perceived objects can and should be manipulated. Besides the type of object, its location is also relevant for inference of possible actions. If cups are often put down on

[4]http://ias.cs.tum.edu/

a table, then the tableside might be added to the knowledge base as a "Put-down-objects-place" [TB09]. This and similar information can be retrieved by observing human actions[5] or other robots. The ontology used in KnowRob contains special objects called *Computables* that are mainly responsible for two things:

- Querying relations from and writing them to an SQL database
- Calculating new relations on the fly. This is especially interesting, as concepts such as *"inObject"* and *"onObject"* can be inferred by comparing object positions.

Examples. Consider the following two basic examples for querying the KnowRob knowledge base. The first example shows how to get subclasses of *FoodOrDrink*.

```
?- owl_subclass_of(A, knowrob:'FoodOrDrink').
A='http://ias.cs.tum.edu/kb/knowrob.owl#FoodOrDrink';
A='http://ias.cs.tum.edu/kb/knowrob.owl#Drink';
A='http://ias.cs.tum.edu/kb/knowrob.owl#Coffee-Beverage';
A='http://ias.cs.tum.edu/kb/knowrob.owl#InfusionDrink';
A='http://ias.cs.tum.edu/kb/knowrob.owl#Tea-Beverage';
A='http://ias.cs.tum.edu/kb/knowrob.owl#Tea-Iced'
```

Listing 1.1: KnowRob example 1

The second example shows how to retrieve information on relation instances of a certain object, in this case *Drawer1*.

```
?- owl_has(knowrob:'Drawer1', P, O).
P='http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
O='http://ias.cs.tum.edu/kb/knowrob.owl#Drawer';
P='http://ias.cs.tum.edu/kb/knowrob.owl#widthOfObject',
O=literal(type(xsd:float, '0.58045006'));
P='http://ias.cs.tum.edu/kb/knowrob.owl#
    properPhysicalParts',
O='http://ias.cs.tum.edu/kb/knowrob.owl#Door4';
```

Listing 1.2: KnowRob example 2

The KnowRob tutorial slides [Ten10] offer more examples as well as helpful information on how to link KnowRob with the rest of a ROS ecosystem.

[5]Human motion is analyzed with 51 degrees of freedom and classifiers are trained to convert them to abstract concepts [TB09]

Chapter 2

Theoretical Part

This chapter is aimed to convey background information on selected subtopics of the robotics domain. Most of the topics mentioned here are applied to the project that will be presented in the practical part of this document.

First off, some basics about coordinate transformations in space shall be discussed, followed by typical challenges in mobile autonomous robotics such as map representation and generation, localization and navigation. This will be followed by a section dedicated to the Robot Operating System (ROS), one on Situation Calculus and its applications and finally a discussion about object recognition, focusing on recognition of previously tagged objects.

2.1 Transformations

Before diving into this topic, I'd like to express my appreciation for the excellent work of Thomas Koch (FH Gelsenkirchen), whose Master's thesis [Koc08] served as a basis for most of the information presented in this section.

When representing poses of objects in a three-dimensional world, two components have to be taken into account, namely the object's position and its orientation. The object's position is represented in and handled as a regular three-dimensional vector, as opposed to its orientation, which is trickier to handle. To learn how to deal with orientations or rotations in 3D space, several topics have to be covered in advance.

As we know, 3D vectors can be interpreted as a translation or as a position in space (as the final position can be seen as the sum of the origin and a translation). The same is true for representations of rotation (and orientation, respectively). An orientation is the result of starting in a neutral pose and applying a rotation. Thus, in the following sections it will be sufficient to stick to the terms translation and rotation.

2.1.1 Rotation Matrices

Two fundamental ways of representing rotation in space shall be presented in this document, one of which are Rotation Matrices. These work well in two dimensions, so why not use them in 3D space. The 2D-version of a rotation matrix is shown in the following equation [Koc08]:

$$R(\alpha) = \begin{pmatrix} cos\alpha & -sin\alpha \\ sin\alpha & cos\alpha \end{pmatrix}$$

To perform a rotation, this matrix with the desired rotation angle α set correctly has to be multiplied with the vector that needs to be rotated, as in

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} cos\alpha & -sin\alpha \\ sin\alpha & cos\alpha \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

If we wish to follow the same principle in three dimensions, we have to extend the term rotation matrix. It's defined by a set of sequential (ordered) rotations around arbitrary axes in space, that are required to traverse the coordinate system's origin [Koc08]. In practice, we choose the coordinate system's three axes (X, Y and Z) and define one rotation for each axis, resulting in three rotation matrices, R_x, R_y and R_z and requiring three rotation angles, α, β and γ. The three aforementioned angles are called Euler Angles and the three matrices are Euler's Rotation Matrices. The latter are defined as follows [Koc08]:

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos\alpha & -sin\alpha \\ 0 & sin\alpha & cos\alpha \end{pmatrix}$$

$$R_y(\beta) = \begin{pmatrix} cos\beta & 0 & sin\beta \\ 0 & 1 & 0 \\ -sin\beta & 0 & cos\beta \end{pmatrix}$$

$$R_z(\gamma) = \begin{pmatrix} cos\gamma & -sin\gamma & 0 \\ sin\gamma & cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Given an (ordered) triple of rotation angles (α, β and γ), we can use them to perform the composite rotation:

$$\vec{v'} = R_x(\alpha) \cdot R_y(\beta) \cdot R_z(\gamma) \cdot \vec{v}$$

Please note that the order in which these rotations are performed is vital. These operations are NOT commutative. However, they are associative, so as long as we stick to the same rotation order, we can keep one composite rotation matrix (the product of the three rotation matrices R_x, R_y and R_z). The three rotation axes are often referred to as roll, pitch and yaw, terms used in avionics. Figure 2.1 shows an aeroplane and how rotations around the three aforementioned axes can be used to alter its orientation.

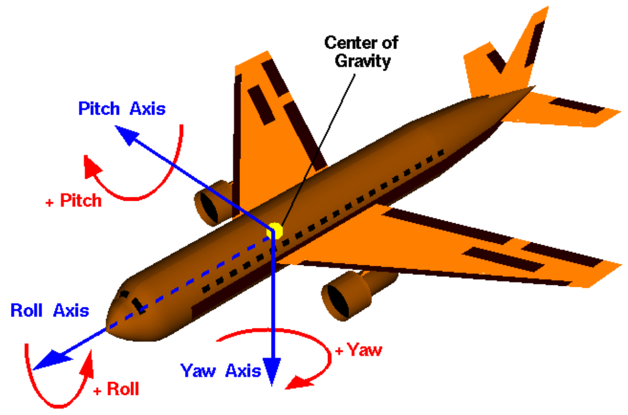
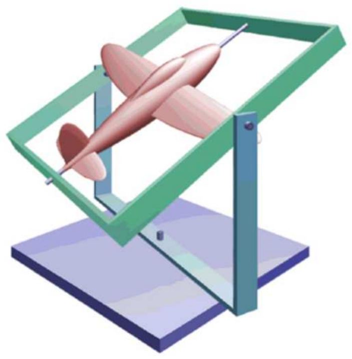


Figure 2.1: Aeroplane with RPY Axes, courtesy of [ROS08]

Gimbal Lock

In the previous section we learned how to represent rotation in 3D space using rotation matrices. This section, however, will let the bubble burst by pointing out a serious drawback of this representation. To start with, consider the construction depicted in Figure 2.2a. The three rings (in this case frames) around the aeroplane in the center correspond to the three possible rotation angles, these are referred to as Cardan[1] rings. They enable the aeroplane to take any orientation in space.


(a) Cardan Suspension

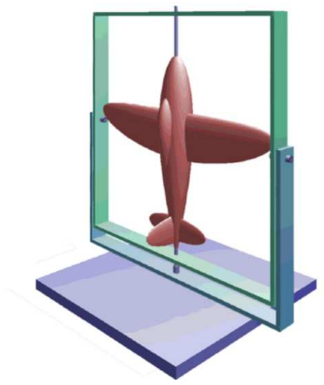

(b) Gimbal Lock

Figure 2.2: Cardan Suspension and Gimbal Lock, courtesy of [Koc08]

In a situation where the outer ring is aligned with the center ring, as depicted in Figure 2.2b, we are suddenly unable to apply yaw rotation (around its Z

[1]http://library.wolfram.com/examples/quintic/people/Cardan.html

axis) to the aeroplane. On the other hand, rotating the outer and inner ring has exactly the same effect. Both actions will apply rotation around the aeroplane's X axis (roll). This situation is referred to as gimbal lock. In systems that rely on cardan suspension, be it mechanical or purely mathematical systems, a gimbal lock can never be prevented, only avoided. Possible solutions are

- Choosing the rotation order that (for a certain application) would be the least likely to yield a gimbal lock
- Early detection of gimbal locks and dynamic change of rotation order (and rotation angles).
- Manually avoiding gimbal locks[2]

For a mathematical representation of gimbal lock, consider the three rotation matrices mentioned before. By combining them to one rotation matrix in an order that corresponds to the aeroplane's example $(R_x(\alpha) \cdot R_z(\gamma) \cdot R_y(\beta))$, as shown below, we receive a matrix that can be used solely for the rotation order α, γ, β. Please note that due to a convention in computer graphics, we have added a fourth, but in this case neutral dimension.

$$R_{(\alpha,\gamma,\beta)} = R_x(\alpha) \cdot R_z(\gamma) \cdot R_y(\beta) =$$

$$\begin{pmatrix} cos\beta \cdot cos\gamma & -sin\gamma & sin\beta \cdot cos\gamma & 0 \\ cos\alpha \cdot cos\beta \cdot sin\gamma + sin\alpha \cdot sin\beta & cos\alpha \cdot cos\gamma & cos\alpha \cdot sin\gamma \cdot sin\beta - sin\alpha \cdot cos\beta & 0 \\ sin\alpha \cdot sin\gamma \cdot cos\beta - cos\alpha \cdot sin\beta & sin\alpha \cdot cos\gamma & sin\alpha \cdot sin\beta \cdot sin\gamma + cos\alpha \cdot cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Now let's set the center rotation (γ) to 90°. Considering the example of the aeroplane, when starting from a neutral position, this would lead exactly to a situation where the outer and inner axes are aligned. As $sin 90° = 1$ and $cos 90° = 0$, our rotation matrix results in ([Koc08])

$$R_{(\alpha,90°,\beta)} = \begin{pmatrix} 0 & -1 & 0 & 0 \\ cos\alpha \cdot cos\beta + sin\alpha \cdot sin\beta & 0 & cos\alpha \cdot sin\beta - sin\alpha \cdot cos\beta & 0 \\ sin\alpha \cdot cos\beta - cos\alpha \cdot sin\beta & 0 & sin\alpha \cdot sin\beta + cos\alpha \cdot cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

and can be further simplified to become

$$R_{(\alpha,90°,\beta)} = \begin{pmatrix} 0 & -1 & 0 & 0 \\ cos(\alpha-\beta) & 0 & -sin(\alpha-\beta) & 0 \\ sin(\alpha-\beta) & 0 & cos(\alpha-\beta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

[2]During the Apollo 11 Mission, maneuvers that would have led to gimbal lock were avoided [Koc08]

Notice how the γ has disappeared (one degree of freedom is lost) and how the resulting rotation depends solely on the difference of α and β (two axis apply the same rotation).

As stated before, when using rotation matrices, gimbal lock is impossible to prevent. In search of a solution for rotation in space that doesn't suffer of this drawback, we therefore have the leave the world of matrices and dive into Quaternion Algebra.

2.1.2 Quaternions

This section describes how Quaternions are defined and treated. The information provided here comes from [Koc08]. Let's start with how Quaternions are defined:

$$q = w + xi + yj + zk | w, x, y, z \in \mathbb{R}$$

In this case, i, j and k are imaginary units that follow Hamilton's rule:

$$i^2 = j^2 = k^2 = i \cdot j \cdot k = -1$$

A Quaternion can be seen as an object with a scalar component (w) and a vectorial component ($xi + yj + zk$). Applied to rotations in space, the vectorial component corresponds to a rotation axis, whereas the scalar component defines the amount of rotation around that axis. Using this approach, there's no need for three predefined axes that can end up in gimbal lock. Instead, we define one "tailor-made" axis for each rotation.

Quaternion Algebra

This section presents essential mathematical operations as they're defined on Quaternions.

Magnitude. A Quaternion's magnitude $|q|$ is defined as ([Koc08], [ROS02])

$$|q| = \sqrt{q \cdot q^*} = \sqrt{w^2 + x^2 + y^2 + z^2}$$

Unit Quaternion. Quaternions that have the value 1 as their magnitude.

$$|q_{unit}| = 1$$

Conjugate. Quaternions are conjugated by inverting the sign of their vectorial component.

$$q^* = w - xi - yj - zk$$

Inverse. A Quaternion's inverse is defined by the equation

$$q^{-1} = \frac{q^*}{N(q)}$$

$N(q)$ is the Quaternion's norm ($N(q) = |q|$). The inverse of a unit quaternion equals its conjugate [ROS02].

Sum and Difference. Adding or subtracting two Quaternions $q_1 = w_1 + x_1 i + y_1 j + z_1 k$ and $q_2 = w_2 + x_2 i + y_2 j + z_2 k$ eventually boils down to adding/-subtracting their scalar and vectorial parts, respectively [Koc08].

$$\begin{aligned}
q_1 \pm q_2 &= w_1 + x_1 i + y_1 j + z_1 k \pm (w_2 + x_2 i + y_2 j + z_2 k) \\
&= w_1 \pm w_2 + (x_1 \pm x_2)i + (y_1 \pm y_2)j + (z_1 \pm z_2)k \\
&= [w_1 \pm w_2, \vec{v_1} \pm \vec{v_2}]
\end{aligned}$$

Multiplication. Multiplication is done by multiplying the Quaternions' components bit by bit, while at the same time respecting Hamilton's law, formally [Koc08]:

$$\begin{aligned}
q_1 \cdot q_2 &= (w_1 + x_1 i + y_1 j + z_1 k) \cdot (w_2 + x_2 i + y_2 j + z_2 k) \\
&= w_1 \cdot w_2 - (x_1 \cdot x_2 + y_1 \cdot y_2 + z_1 \cdot z_2) \\
&\quad + (w_1 \cdot x_2 + w_2 \cdot x_1 + y_1 \cdot z_2 - y_2 \cdot z_1)i \\
&\quad + (w_1 \cdot y_2 + w_2 \cdot y_1 + z_1 \cdot x_2 - z_2 \cdot x_1)j \\
&\quad + (w_1 \cdot z_2 + w_2 \cdot z_1 + x_1 \cdot y_2 - x_2 \cdot y_1)k \\
&= [w_1 \cdot w_2 - \vec{v_1} \cdot \vec{v_2}, \vec{v_1} \times \vec{v_2} + w_1 \cdot \vec{v_2} + w_2 \cdot \vec{v_1}]
\end{aligned}$$

Multiplication of Quaternions is especially important for our purposes, as this operation is used when applying rotation. In practice, Quaternions are often interpreted and written as vectors: $(x, y, z, w)^T$. When sticking to this interpretation, we may as well rewrite the equation above to a form that can be directly applied in program code. In the following equation, the necessary operations for dot and cross products of vectors have been resolved, leaving us with nothing more than additions, subtractions and multiplications of scalars [Koc08]:

$$q_a \cdot q_b = \begin{pmatrix} w_a \cdot x_b - y_b \cdot z_a + y_a \cdot z_b + w_b \cdot x_a \\ z_a \cdot x_b + w_a \cdot y_b - z_b \cdot x_a + w_b \cdot y_a \\ -x_b \cdot y_a + x_a \cdot y_b + w_a \cdot z_b + w_b \cdot z_a \\ -x_a \cdot x_b - y_a \cdot y_b - z_a \cdot z_b + w_a \cdot w_b \end{pmatrix}$$

1. Quaternion multiplication is **not commutative**, thus $q_1 \cdot q_2 \neq q_2 \cdot q_1$. However, it is true that $q \cdot q^{-1} = q^{-1} \cdot q$, as both terms yield the result 1, which is the neutral element in multiplication.

2. Products of Quaternions are associative. Therefore the equation $(q_1 \cdot q_2) \cdot q_3 = q_1 \cdot (q_2 \cdot q_3)$ holds

3. Quaternion multiplication is distributive, leading to the equations $q_1 \cdot (q_2 + q_3) = q_1 \cdot q_2 + q_1 \cdot q_3$ and $(q_2 + q_3) \cdot q_1 = q_2 \cdot q_1 + q_3 \cdot q_1$

Polar Coordinate Form

By setting $w = r \cdot cos\varphi$, $x = r \cdot sin\varphi$, $y = r \cdot sin\varphi$ and $z = r \cdot sin\varphi$, Quaternions can be written in polar coordinate form:

$$\begin{aligned} q &= r \cdot cos\varphi + r \cdot i \cdot sin\varphi + r \cdot j \cdot sin\varphi + r \cdot k \cdot sin\varphi \\ &= r(cos\varphi + i \cdot sin\varphi + j \cdot sin\varphi + k \cdot sin\varphi) \end{aligned}$$

This is the type of notation that we use to convert from axis-angle representation to Quaternions. In the equation above, $r = |q|$ and φ is the Quaternion's angle. Given that q is of unit length and a vector $\vec{n}$ with $|\vec{n}| = 1$ we can write:

$$q = [cos\varphi, \vec{n} \cdot sin\varphi]$$

Application to Rotation in Space

Here comes the interesting part of applying the above knowledge to rotations in three-dimensional space. To alter orientations by a certain rotation defined as a Quaternion, we simply have to multiply the two Quaternions. Given that the original orientation is q_o and the requested rotation is q_r, the new orientation after performing the rotation is defined as

$$q_n = q_r \cdot q_o$$

Rotating 3D vectors is slightly different. First of all we have to write the vector to be rotated, $(x, y, z)^T$, in Quaternion form [Koc08]:

$$[0, \begin{pmatrix} x \\ y \\ z \end{pmatrix}]$$

Next, we need to convert the rotation axis $\vec{v}$ and the angle θ to Quaternion form:

$$[cos\frac{\theta}{2}, \vec{v} \cdot sin\frac{\theta}{2}] = [cos\frac{\theta}{2}, \begin{pmatrix} x \\ y \\ z \end{pmatrix} \cdot sin\frac{\theta}{2}]$$

Alternatively, we can rewrite our Quaternions to $(x, y, z, w)^T$, as mentioned above. Given that p is the vector to be rotated (in Quaternion notation), q the requested rotation and q^{-1} its inverse, we use to following operation to carry out the rotation:

$$p' = q \cdot p \cdot q^{-1}$$

If q is a unit quaternion, $q^{-1} = q^*$, and we can write

$$p' = q \cdot p \cdot q^*$$

The rotated vector is contained in the (x, y, z) part of the resulting quaternion p'.

Multiple Rotations. Consider the following situation in which two rotations, q_1 and q_2 have to be performed in sequence, with P being the vector to be rotated (already in Quaternion rotation). This is done by nesting the rotations (as described in [ROS02]):

$$q_2 \cdot (q_1 \cdot P \cdot q_1^{-1}) \cdot q_2^{-1}$$

Due to Quaternion multiplications being associative, this can be rewritten to

$$(q_2 \cdot q_1) \cdot P \cdot (q_1^{-1} \cdot q_2^{-1}) = (q_2 \cdot q_1) \cdot P \cdot (q_2 \cdot q_1)^{-1}$$

Thus, to combine multiple rotations, all we need to do is multiply all Quaternions and use the result to calculate the total rotation.

Conversion to Rotation Matrix

The operation described above can easily be converted to a Rotation Matrix. This is still required for some applications, such as OpenGL [Koc08]. The Rotation Matrix R corresponding to the rotation described by a quaternion $(x, y, z, w)^T$ is defined as follows [Koc08]:

$$R = \begin{pmatrix} 1 - 2 \cdot (y^2 + z^2) & 2 \cdot (x \cdot y - w \cdot z) & 2 \cdot (x \cdot z + w \cdot y) & 0 \\ 2 \cdot (x \cdot y + w \cdot z) & 1 - 2 \cdot (x^2 + z^2) & 2 \cdot (y \cdot z - w \cdot x) & 0 \\ 2 \cdot (x \cdot z - w \cdot y) & 2 \cdot (y \cdot z + w \cdot x) & 1 - 2 \cdot (x^2 + y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The fourth dimension has already been added due to conventions in computer graphics.

2.1.3 Coordinate Transformations

In a world model, usually multiple coordinate systems exist simultaneously. In ROS (described later on in section 2.5) and in several other applications and documents these systems are called frames. Examples for frames in a virtual world are the map, the robot base, the robot laser, an object that's been recognized by the robot, the camera frame (when rendering the world), each of which represent a different view of the world. Every pose can be represented in any of the available coordinate frames, given the correct transformations between these frames. Thus we need a framework that holds the necessary transformations and is able to apply them to poses. Transformations contain, just like poses, a 3D vector to describe a translation and a Quaternion to describe rotation. To carry out the transformation of pose P given the transformation T, we could convert the rotation described as Quaternion to a rotation matrix, and combine it with the required translation. The formula for the conversion from a Quaternion to a rotation matrix is shown in the following equation [Koc08]:

$$M = \begin{pmatrix} 1 - 2 \cdot (y^2 + z^2) & 2 \cdot (x \cdot y - w \cdot z) & 2 \cdot (x \cdot z + w \cdot y) \\ 2 \cdot (x \cdot y + w \cdot z) & 1 - 2 \cdot (x^2 + z^2) & 2 \cdot (y \cdot z - w \cdot x) \\ 2 \cdot (x \cdot z - w \cdot y) & 2 \cdot (y \cdot z + w \cdot x) & 1 - 2 \cdot (x^2 + y^2) \end{pmatrix}$$

2.2 Mapping

A robot requires some form of representation of the real world (a map), for example to allow for localization. There are various types of map representations, with their advantages and drawbacks.

2.2.1 Map Representation

Clearly, it would be the simplest solution to store any spatial information we retrieve in an exact high-fidelity map. However, this would result in high computational effort for tasks such as localization and path planning. Thus, strategies are required that reduce the necessary storage space and computation time. The environment has to be simplified and abstracted. According to [SN04], map representation strategies can be divided into the following groups:

- Continuous representation
- Map decomposition
  - Exact decomposition
  - Fixed decomposition
- Topological representation

Continuous representation

Typically, office environments will be filled with walls and furniture, thus with simple geometric structures. From a top view, a simplified representation would often manage with nothing more than horizontal and vertical lines. A map of such an environment could be stored as a set of lines that approximate the real world. The danger of this approach is high computational cost in case the representation becomes too exact, thus the key behind its successful application is abstraction [SN04]. An appropriate balance between the amount of information and efficiency has to be determined.

Map decomposition

Another map representation strategy is map decomposition. We differentiate between exact and fixed decompisition, however both approaches have the same underlying mechanism, which is dividing the environment into cells and storing spatial connections between these cells in a connectivity graph.

Exact cell decomposition. This method splits free space into distinct cells, depending on obstacle positions. For example, vertical lines can be drawn through each corner of an object, the combination of the lines and the object borders being the cell border. Unfortunately this approach is complex to implement and the computational complexity depends on the number of objects and their structure [SN04].

Fixed cell decomposition. As opposed to exact cell decomposition, this approach is independent of the environment's density and is easy to implement. Instead of decomposing the environment based on obstacle positions, a grid divides it into cells of equal size. What's left to do is to specify whether cells are occupied or not. This representation allows to apply search methods yielding low computational costs, such as wavefront expansion [SN04]. Fixed cell decomposition is one of the most popular map representation methods. For instance, ROS (described in section 2.5 below) uses cell-based **occupancy grids** to store spatial information. An occupancy grid associates an integer with each grid cell, increasing its value once a range measurement system detects an obstacle within the cell's borders. Above a certain threshold, the cell is considered to be an obstacle [SN04].

Topological representation

Instead of measuring the geometric structure of the environment, topological representations concentrate on those features that are relevant for localization [SN04]. The environment is filled with interconnected nodes, that are associated with features that can be observed at each of these nodes' positions. Two mechanisms are thus required:

- Localization, meaning that the robot must be able to associate its current position with one of the nodes
- Navigation from one node to another

Topological representations can be applied to environments in which geometric structures are not the most salient features [SN04]. Also, they allow for a cost-effective belief storage. A drawback, however, is information loss. For instance, exact localization is only possible when the robot reaches nodes.

2.2.2 Map Creation

While it would be possible to create maps manually, this can obviously be a tedious task for large maps. Instead, we would like to have the robot create a map on its own, either by autonomously exploring its environment or with a human operator guiding it. A popular approach, that can be applied to occupancy grids, is SLAM (Simultaneous Localization and Mapping). SLAM approaches all have the same underlying chicken-and-egg problem that they're designed to solve. On the one hand, the robot's position is required in order to know where to add new sensor readings to the occupancy grid. On the other hand, the information in the occupancy grid is used to calculate the robot's position [Hop10]. In ROS, the *gmapping* package, that relies on sensory input from laser range measurement systems, is responsible for mapping tasks. The underlying method is described in [GSB07]. Vision-based solutions are discussed in [Hop10].

2.3 Localization

Mobile robot localization is the problem of determining a robot's pose from sensor data [TFBD01]. Such sensor data can for example be retrieved using laser scanners (as in the case of our mobile robot described in the practical part of this document). The scan results reflect the current view of the world surrounding the robot, that has to be matched with the robot's belief about the structure of the world, which is usually saved as a map of the environment. This means that based on partial information about the environment, the robot must be able to decide which part of the map is most similar to the snapshot it's facing. Not only the current sensor data will be used to estimate the robot's position, but also its odometry readings. These two sources of information have to be optimally merged to yield good position estimates. [SN04] mentions three essential problems that accompany the localization task, namely sensor noise, sensor aliasing[3] and effector noise (odometric error). Two solutions to this problem shall be mentioned here, namely Kalman filter and Markov localization.

2.3.1 Kalman Filter Localization

A Kalman Filter can be used for a wide range of problems, where Gaussian probability densities with known variances have to be merged. One such problem happens to be robot localization, which usually involves the combination of sensor readings and odometry. To demonstrate this method, let's assume a static robot performing multiple measurements. Given a prior position estimate $\hat{x}_k$ with the corresponding variance σ_k, as well as a new measurement z_{k+1}, again with a corresponding variance σ_z, we can write (according to [SN04])

[3]Not even an absolutely noise-free sensor is sufficient to localize the robot with one snapshot, due to its limited resolution and range

$$\hat{x}_{k+1} = \hat{x}_k + K_{k+1}(z_{k+1} - \hat{x}_k)$$
$$K_{k+1} = \frac{\sigma_k^2}{\sigma_k^2 + \sigma_z^2}$$

This means that each measurement will contribute to the overall estimate to a certain amount depending on its variance. What's left to do is calculate the new estimate's variance, which can be done in the following step:

$$\sigma_{k+1}^2 = \sigma_k^2 - K_{k+1}\sigma_k^2$$

When applied to a dynamic system, e.g. a moving robot, the position will change between sensor readings, so an effector model is required that can propagate the position from one timestep to the other. Of course by doing this, the estimate's uncertainty will be growing, an effect that has to be modeled as well.

An application of a Kalman Filter based localization method is described in [SN04]. This method relies on the extraction of line features and matching them to a known map to estimate the robot's position within the map.

2.3.2 Markov Localization

Another approach to the localization problem, that can work with arbitrary types of probability distributions, as opposed to Kalman Filters, is Markov localization. This method is based on Bayes' formula for calculating the conditional probability of A, given B [SN04]:

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}$$

This can be applied to robot localization by replacing A with the robot's actual location l and B with the sensor input i, resulting in the following equation:

$$p(l|i) = \frac{p(i|l)p(l)}{p(i)}$$

Thus, by modeling the sensor input given a specific location, the sensor model being represented as $p(i|l)$ and multiplying it with the robot's prior position estimate $p(l)$, we can estimate the probability for each discrete position. The denominator $p(i)$ is usually omitted and the resulting probabilities normalized instead. Further details about this method are provided in [SN04].

This and other approaches relying on the Markov assumption are becoming increasingly popular in the field of mobile autonomous robotics [SN04]. The Markov assumption states that a system's state only depends on its previous state and the actions performed since then. This is in general a false assumption [SN04], however it's a good approximation for many systems and dramatically simplifies further calculations, as there's no history to be taken into account.

The following method, that is used for localization tasks in ROS (*amcl* package, described in section 3.4.2 on page 94), also relies on the Markov assumption and applies a particle filter to estimate the robot's position.

2.3.3 Monte Carlo Localization

This approach is based on a particle filter that represents the robot's belief as a set of weighted samples. This belief is repeatedly updated, by performing the following steps at each discrete time step [Fox01]:

1. Samples are randomly fetched from the previous belief according to their importance weights. The probability of a particle being fetched depends on its weight.
2. Each of the particles chosen in the previous step will be used to sample new particles, based on the last control information, which in a robot's case might be derived from odometry.
3. Finally, the particles are weighted using the new measurements. Those that align well with the values received from the measurement are weighted higher.

The particle filter's advantage is that it can represent various types of probability densities, not only Gaussians. A challenge that it presents is to find the optimal amount of samples or particles to get good results without wasting system resources. To illustrate the optimization potential behind this parameter, imagine a situation in which the robot is already well localized and only needs to track its position, as opposed to a situation in which it has no clue about its location (in early stages of operation, with no position estimate provided). In the latter situation a significantly higher amount of particles is required than in the former, as the possible positions are initially distributed all over the map. Obviously a naïve approach would be to stick to a fixed amount of particles that seems to be a good trade-off for the given environment. However, there are more sophisticated solutions.

Likelihood-based sampling

According to [Fox01], the sum of the non-normalized importance weights of particles reveals the position estimate's quality and thus the amount of necessary particles. This becomes clear, considering the two situations mentioned in the previous section. When the robot is well-localized, a moderate amount of particles will already yield a high accumulated importance weight, whereas a delocalized robot will issue many "guesses" that don't align with sensor information, thus yielding low likelihood values for a higher amount of particles. As promising as this approach seems to be, it has difficulties dealing with symmetric environments, e.g. environments that feature several similar spots (rectangular hallways for example). In these cases, based on the measure presented here, the agent might issue a low amount of particles, thinking that it's well-localized, but the particles will be located at various spots in the map that resemble each other.

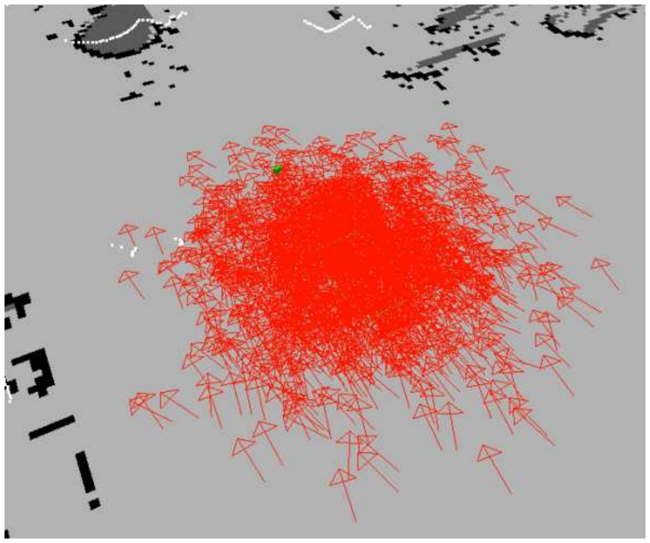


Figure 2.3: Monte Carlo Localization - Particles representing hypotheses

KLD-Sampling

This method for estimating the necessary amount of particles is applied in the *amcl* package that comes with ROS. The basic idea behind this approach is to have the K-L distance (Kullback-Leibler distance) between the true posterior and the maximum likelihood estimate go below a certain threshold. In other words, the error between reality and the agent's belief is measured and the number of particles adapted accordingly. Unfortunately the true posterior isn't directly available, as the particle filter is responsible for approximating it in the first place. However, it suffices to interpret the distribution as a Multinomial (n samples drawn from k bins) and specify the maximum number of samples per bin before it counts as supported (one sample is used in [Fox01]). The number of necessary samples n depends solely on the number of supported bins k, as shown in the following equation:

$$n < \frac{1}{2\epsilon}\chi^2_{k-1,1-\delta}$$

For further details on how this equation is derived, please refer to [Fox01], where you will find an exact description of the approach. The δ and ϵ variables can be seen as parameters to this mechanism, as they allow to configure the required confidence. The essential difference between this approach and Likelihood-based sampling is that it takes the diversity of samples into account. Thus, in the previous example of a rectangular hallway with similar spots, it will notice that not only is the robot's belief well aligned with sensor readings, but that particles are still being placed at various spots in the map, therefore increasing the number of particles.

2.3.4 Challenges in Localization

In [TFBD01], two challenging problems in the field of localization are mentioned, namely the global localization problem and the kidnapped robot problem. To explain these, let's consider a third problem, which is keeping track of the robot's position. In this case, there's already a relatively reliable pose estimate available, which is only progressed to further timesteps and adjusted based on sensor input. So the robot's position is known and the robot is also confident about the estimate's quality. In a situation where a pose estimate is not available and the robot's perfectly aware of the fact that it's delocalized, we face the global localization problem. This is for example the case shortly after startup. On the other hand, if a robot is delocalized, but not aware of this fact, we face the kidnapped robot problem. Imagine the robot being teleported from one spot to another[4]. Consider that given that the robot had been traveling for some time, the measurements before the unanticipated event will be accompanied with high confidence values, so it will be hard to incorporate a dramatic change of position. While a Kalman filter is highly sensitive to the global localization and kidnapped robot problems, these two are solved in a robust way by the Monte Carlo Approach [TFBD01].

2.4 Navigation

An autonomous mobile robot must feature skills that enable it to navigate between specific spots in the world, in order to be able to fulfill its daily tasks, such as surveillance or transportation. Some of the (often conflicting) requirements that accompany these navigation tasks shall be mentioned below ([SN04]).

- **Effectiveness.** It's often required that an agent reach its goal within a certain amount of time, or sticking to the shortest path.
- **Completeness.** If a path to the goal is available, the robot is expected to find this path and traverse it [SN04].
- **Obstacle avoidance.** Obviously the robot is not supposed to hit any obstacles on its way to the goal.
- **Optimization of sensor readings.** For short-ranged sensors, it's often sensible to navigate close to interesting feature points in order to maintain a good localization [SN04].

Navigation is usually divided into two subproblems, namely (global) path planning and obstacle avoidance. The global planner uses a simplified model of the world and the robot to efficiently calculate a way between the robot's position and the goal, whereas the local planner navigates to the goal, while trying to stick to the global plan as strictly as possible (or as requested).

[4]A more credible image is disabling the robot while saving its current state, then moving and finally reactivating it at a new spot

2.4.1 Global Path Planning

A navigation task usually starts with the search of a global plan from the robot's position to the goal. Sticking only to local influences and short-term local plans, the robot might end up in local minima. With the global planner recommending a path, this can be avoided. Also this is the first instance that decides whether a traversable path exists or not. There are several approaches to global path planning, some of which shall be mentioned here. What's common to these approaches is the problem of representing a sensed, continuous environment or a map as a set of discrete spots, lines or cells, thus allowing to model the environment as a connectivity graph. Such a representation is necessary in order to be able to apply traditional path planning algorithms [SN04].

Road Maps

The idea behind this approach is to define a set of paths in the environment, similarly to roads on a map. This can be achieved through a visibility graph or a Voronoi diagram [SN04].

Visibility graph. This method draws direct lines between obstacle edges, wherever possible without intersecting other obstacles. The drawback hereof is that the resulting path will always take the robot close to obstacles. On the other hand, this approach could be advantageous for short-range sensors, providing them with richer input [SN04].

Voronoi diagram. As opposed to the visibility graph, this approach tries to maximize the distance to the obstacles in the environment. One of its advantages is that it's executable: sensor readings can directly be used to correct the robot's position. The major drawback of this method is that the calculated path is often far from optimal [SN04].

Cell Decomposition

Another approach is to discretize the world around the robot by dividing it into cells, either depending on obstacles (exact cell decomposition) or independently (approximate cell decomposition). The cells can then be connected to form a graph that can serve as in input to path planning algorithms. Both methods are discussed in section 2.2.1 above.

Potential Field

The potential field approach creates a virtual field of attracting and repulsing forces, guiding the agent to its goal. The environment can then be seen as a mountainous landscape, with the goal being a valley, obstacles being mountains and the agent a ball rolling towards the valley. The major drawback of this approach is that the robot might get caught in local minima or start oscillating between obstacles.

2.4.2 Local Planning and Obstacle Avoidance

The local planner is responsible for calculating paths around obstacles, while at the same time sticking to the path recommended by the global planner. A simple approach to obstacle avoidance is the so-called Bug algorithm [SN04]. This method issues a path around an obstacle in the robot's way, along the obstacle's contour. In the first version of the algorithm, the robot circles the obstacle once and subsequently "leaves its orbit" at the point that's closest to the goal, while a second version is implemented in a way that it will continue on a direct path to the goal whenever possible. Another, more sophisticated solution is the dynamic window approach, that's part of the ROS navigation stack (section 3.4.3).

Dynamic Window Approach

This method takes the robot's rotational and translational velocities and acceleration capabilities into account. Based on this information, the set of possible velocities (rotational as well as translational) is calculated for the next timestep. As depicted in Figure 2.4, this yields a rectangular field in the 2D velocity search space, called the dynamic window, hence the name of this technique[5] [FBT97] [SN04].

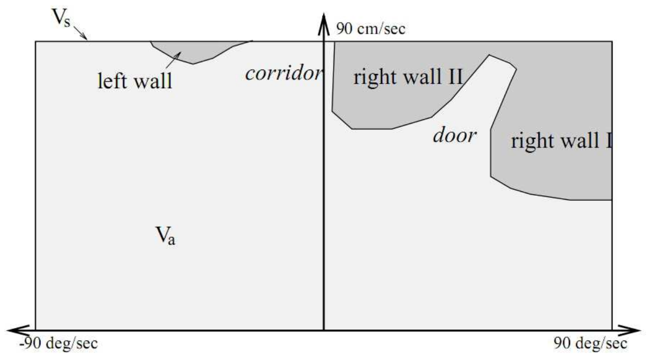


Figure 2.4: Dynamic window in the 2D velocity search space [FBT97]

In a second step, all those velocity combinations that might lead to collisions, are eliminated. The following objective function is then applied to the cropped search space:

$$O = a \cdot heading(v, \omega) + b \cdot velocity(v, \omega) + c \cdot dist(v, \omega)$$

In this equation (from [SN04]), *heading* measures the robot's orientation toward the goal and thus its progress, *dist* tries to maximize the distance to the closest obstacle and *velocity* prioritizes fast movements [SN04].

[5]The original view of the environment is depicted in Figure 2.5

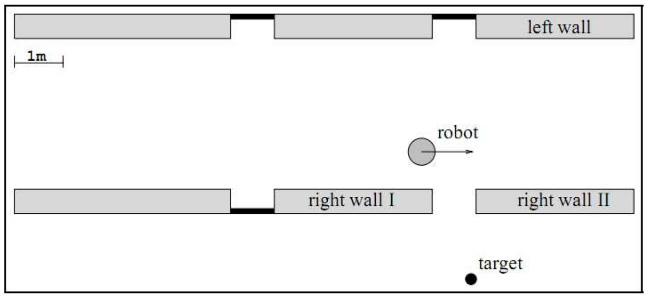


Figure 2.5: Environment corresponding to the DWA example, from [FBT97]

Trajectory Rollout

Another method that is available out-of-the-box in the ROS navigation stack is trajectory rollout, which is quite similar to dynamic window, except that the lookahead time is configurable. Thus trajectory rollout will produce more samples, which makes it computationally more expensive than the dynamic window approach, but might be advantageous for robots with low acceleration limits [ROS11e].

Costmap

Any module that performs local path planning and obstacle avoidance needs some form of information about the current state of the world around the agent, in order to be informed about obstacles that might affect the robot's operation. This information can be retrieved from sensor input, be it laser scan data or point clouds coming from 3D sensors. The data received is stored and managed in a so-called costmap, which is a discrete grid representing the environment. The 3D version of a costmap is called octomap[6], but shall not be treated here. The essential operations that can be performed on this data structure are marking and clearing, as described below. A visualized costmap is displayed in Figure 3.16 on page 96.

Marking cells. Obstacles detected by the robot's sensors are added to the costmap by simply marking grid cells that are at least partly occupied by the obstacle. The maximum distance at which obstacles should be interpreted as such and thus added to the costmap, can be configured.

Clearing cells. Heuristics are required that define how the costmap can be cleared of obstacles that are no longer present or relevant to the robot. A straightforward approach is to remove ostacles that are "far enough" behind the robot. This mechanism is one reason for penalizing reverse motion in navigation tasks. The aforementioned approach is coexistent with a second, more

[6]http://www.ros.org/wiki/octomap

sophisticated one that is absolutely required in dynamic environments. Imagine someone crossing the robot's way, thus obstructing its originally foreseen path for a splitsecond. Consider that this is enough time for a sensor to detect the moving obstacle and, as we've only defined a marking operation so far, virtually flooding the map with obstacles. Thus we have to use sensor information not only to fill, but also to clear the costmap. In the previous example, after the subject has passed by the robot, a laser sensor will now probably sense a wall or another obstacle, for example a chair in the distance. For each reading that provides information on an obstacle (regardless of whether it must be added to the costmap), a ray is projected from the sensor to the obstacle, and all previous obstacles in the costmap that are hit by this ray are removed. This guarantees that both "meanings" of a sensor reading are taken into account, namely the information that on obstacle has been detected and the one that the space between the sensor and an obstacle **must** be unoccupied.

Recovery

With all the well-thought out heuristics and their correct configurations, there might still be situations in which the agent seems to be lost due to being surrounded by obstacles. Sensors don't convey perfect information and might miss important events due to limited operating rates. Also, the world state might change rapidly and dramatically, requiring the robot to completely discard its previous plans and recalculate a solution. For those situations, recovery strategies must be implemented that (often aggressively) try to repair the robot's view of the world (on a quantitative, execution level). Two such methods that are available in the ROS navigation stack shall be mentioned here.

Turning around. In bad times it's a good idea to take a look around. In the end, keen observation might reveal that there's a solution after all.

Forced clearance of costmap. Sarcastically said, problems can be solved by simply ignoring them. This mechanism clears all obstacles from the costmap, regardless of what side effects this may cause. Obviously, this is not a procedure that should be applied in everyday operation, rather for exceptional situations. The ROS navigation stack offers an interface that allows to manually call this procedure (the caller being either the user or other nodes).

2.5 The Robot Operating System (ROS)

To start with, the following quotation taken from ROS.org [ROS11c] should give a rough first impression of what's hiding behind the name ROS:

> *ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more.*

Despite its name, ROS, the Robot Operating System, is not what is usually meant by this term [QCG+09]. Instead of building on top of a hardware architecture and taking care of memory management, scheduling and of other typical OS tasks, it is based on an existing installation of a regular operating system (preferably Ubuntu, according to [ROS11z]) and adds functionality that facilitates development and execution of software for robots. The two most important features of ROS are its communication framework and the rich toolkit it comes with.

A running ROS system usually consists of numerous independent, rather small software modules (called nodes), each of them written in one of the supported languages (C++, Python, Java and others). These nodes can communicate with each other by passing strictly typed messages, either via a publisher/subscriber mechanism or by calling services. The former allows for the definition of communication channels (called topics), that carry one specific type of message.

The basic idea behind ROS was to offer robotics researchers all over the world a platform to facilitate collaboration on a vast set of libraries solving several problems in the field of robotics. Instead of having teams of researchers implement complete robotics systems, they can now focus on selected fields of research, contribute their progress to the ROS community and on the other hand benefit from others' work. Libraries that are made compatible with ROS can, at the same time, remain independent of it, as the only required step is the implementation of a lightweight wrapper module that communicates with the library and forwards requests and results to the ROS communication framework. This thin architecture is one of the main philosophical goals of ROS [QCG+09]. Another was to make ROS absolutely free and open-source.

As many developers and institutes are currently carrying out research in the field of robotics, the ROS community is thriving, offering numerous benefits:

- Rapid implementation and integration of drivers for new hardware
- New useful software libraries are constantly added
- Modules can be checked out from version control, ensuring up-to-dateness
- A rich Wiki with tutorials and documentation for almost any software module
- A mailing list and an answers-portal helping out troubled fellow developers

2.5.1 Languages Supported by ROS

ROS nodes can be implemented in various languages, including C++, Python, Lisp, Java and Octave. Support for others is constantly being worked on. According to ROS philosophy, instead of building on a C-based stub, language support is natively implemented [QCG+09]. Exceptions to this ideology are currently (at least) Java and Octave. Nodes written in *rosjava* for example communicate with the underlying node implementation using JNI. This, however, is only a temporary solution that is subject to change in future releases.

The reason for supporting multiple languages is twofold. On the one hand, we have to face the fact that every programming language has its enthusiasts, as well as its sworn enemies. Thus, in a collaborative environment with possibly thousands of developers at work, hailing from various fields of computer science, there is no one language that will be unanimously accepted. On the other hand, the coexistence of programming languages can be justified by objective argumentation. Some are more efficient than the others, result in cleaner and more structured code or allow for rapid prototyping, just to name a few distinct properties. In the project described in the second part of this document, three of the available programming languages were used. To sum it up in one concise sentence: C++ is fast, Python is flexible and Java is pretty.

2.5.2 ROS File System

A ROS installation is organized in a tree of stacks and packages, where a stack contains a set of interrelated packages. These can be located virtually anywhere in the file system, as long as they are referenced by an environment variable[7]. This variable will point to the folders containing ROS stacks or packages, that will then automatically be made available in build processes, or when browsing code. A sample set of paths pointing to ROS packages, that served our purposes well, is described in our Wiki [ROS11x]. This configuration consists of separate paths for

- the ROS base installation with all its built-in packages
- stacks and packages coming from external sources
- a local copy of the *ist-ros-pkg* folder (checked out from our SVN repository)
- an additional folder serving as "sandbox"

Our institute's repository hosting ROS projects is inspired by the structure used at WillowGarage[8] and is thus considered good practice for organizing ROS code [ROS11y]:

```
* ist-ros-pkg/
      o stacks/
             + stackname1/
                  # branches/
                  # tags/
                  # trunk/
                        * packagename1/
                              o manifest.xml
```

[7]ROS_PACKAGE_PATH
[8]http://www.willowgarage.com/

```
                              o etc.
                        * packagename2/
                        * stack.xml
                        * Makefile
                        * CMakeLists.txt
            + stackname2/
                  # branches/
                  # tags/
                  # trunk/
                        * stack contents...
```

ROS comes with several tools that facilitate manipulation of and navigation in the ROS file system. These are described in further detail in section 2.5.13.

Stacks

A stack is a collection of interrelated software modules, that in combination and as a whole, perform a useful task. From the underlying operating system's point of view, it's nothing more than a folder containing an XML file (the stack manifest, in a file called *stack.xml*) along with a set of packages. The presence of a *stack.xml* file indicates that the folder contains a stack. An example a ROS developer might stumble upon is the *navigation stack*, providing robots with navigational abilities.

According to [ROS11u], stacks serve several purposes, including simplifying the process of code sharing and keeping track of versions and dependencies. When releasing a stack, it's good practice to provide it with a CMakeLists.txt (described in 2.5.2) and a Makefile, located in the stack's root folder. These files facilitate build processes of the stack as a whole.

Helpful tools. The *roscreate-stack* tool described in section 2.5.13 should be used to auto-generate common files in a stack.

The Stack Manifest. As mentioned before, each stack contains a *stack.xml* file serving as the stack's manifest. This file holds various types of information, including licensing information, dependencies to other stacks, information on the author and the release version [ROS11t]. A stub of the stack manifest is auto-generated when using the *roscreate-stack* tool.

Packages

Packages are the basic organizational units of software in ROS. They contain nodes, configurations, message and service definitions, as well as libraries [ROS11h]. Just like stacks, packages contain manifests that specify author, licensing, as well as dependencies, however in this case they are contained in a *manifest.xml* file. This difference in the file names allows for quick discrimination between stacks and packages.

Package Manifest. The package manifest is very similar to the stack manifest (described above). It's contained in an XML file called *manifest.xml*.

Package Makefile. In addition to the package manifest, packages contain a *CMakeLists.txt* file, similarly to a regular Makefile, specifying the way the package should be built. This file includes information on

- nodes contained in the package
- source files corresponding to nodes
- whether messages/services should be generated
- and other information, depending on the package's purpose (action servers and files required for features such as dynamic reconfigure)

Helpful tools. The *roscreate-pkg* tool described in section 2.5.13 offers a functionality similar to the *roscreate-stack* tool, but applied to packages instead of stacks. Some notable differences are that dependencies can be defined on the command line and are consequently added to the manifest, and that the folder containing the package doesn't have to be created by hand before invoking the tool. Navigational tools such as *roscd* and *rosls*, as well as the build tool *rosmake* use an index of package names and their locations and can therefore instantly reference any package in the *ROS_PACKAGE_PATH*, without requiring the developer to specify a path manually.

2.5.3 Nodes

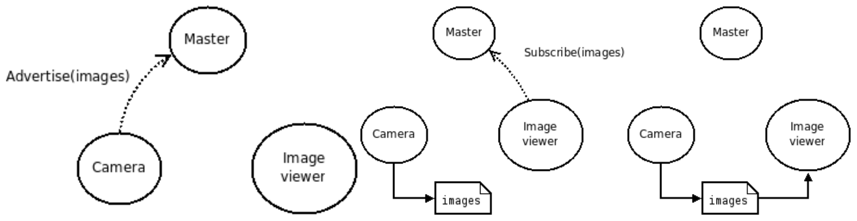
Nodes are processes that perform computation [QCG+09]. In the context of ROS, a node is the equivalent of a software module. It's convenient to visualize a running ROS system as a graph, due to its peer-to-peer structure. In this graph, software modules can be depicted as nodes, whereas topics, the communication channels between these nodes, are, intuitively, displayed as edges. This visualisation style led to the term "node" [QCG+09]. These software modules can be implemented in various languages, as described in section 2.5.1. The communication between nodes is carried out via messages (see section 2.5.5), that are sent and received on certain topics (described in section 2.5.6). Nodes are designed to be lightweight modules, either wrapping around large libs to make them accessible to the rest of the ROS framework or hosting self-implemented logic.

2.5.4 Master

The Master is a central naming and registration service that negotiates communication between nodes using an XML-RPC mechanism, but does not carry out the tedious task of forwarding communication payload [ROS11f]. This is an especially important property, as it corresponds to the peer-to-peer topology

ROS was intended to exhibit [QCG+09]. This mechanism yields the following course of events, when establishing a communication channel between two nodes (example depicted in Figure 2.6 [ROS11f]):

1. Node *Camera* advertises topic *images*. Advertising in this case means notifying the master of the availability of this topic. The topic which we interpret as a communication channel is nothing more than a string.
2. Node *Image Viewer* subscribes to the aforementioned topic *images*. To this end, it queries the master for the set of nodes that publish messages on the requested topic. The master responds to this query by returning *Camera's* address.
3. Finally, *Image Viewer* establishes a **direct connection** to *Camera*, on the *images* topic.



(a) Topic advertisement (b) Topic subscription (c) Connection established

Figure 2.6: ROS Master establishing communication channel, courtesy of [ROS11f]

This publish/subscribe mechanism grants nodes to

1. advertise N topics
2. subscribe to M topics
3. advertise topics that have been previously advertised by other nodes
4. subscribe to topics other nodes are already subscribed to

In case 3, messages from all the nodes publishing on the same topic are merged together, whereas in case 4, the messages are delivered to all nodes that are listening.

Helpful tools

The Master is usually launched by invoking *roscore* [ROS11f], along with a Parameter Server, that's described in section 2.5.9.

2.5.5 Messages

ROS Messages are packets of structured and strictly typed data that are sent between ROS nodes, either on ROS topics or as service requests and responses. They are defined in message definition files[9], written in a language-independent interface definition language [QCG+09]. For each programming language supported by ROS, message handler objects are auto-generated when building packages containing message definitions. From the node developer's point of view, incoming and outgoing messages feel like native objects [QCG+09], with object fields corresponding to message entries. The generated LOC outnumbers the message definition's LOC by far, as demonstrated in Table 2.1. In this case, the world model's message and service structure (discussed in section 3.2.4 on page 74) and the resulting auto-generated code served as an example.

Set	**IDL**	**C++**	**Python**	**Java**
Messages	55	2182	2620	845
Services	61	7126	6706	3095

Table 2.1: LOC for generated messages and services

Please note that in IDL, a shallow view of the message definition was used to calculate the LOC. For example, instead of decomposing a *PoseWithCovarianceStamped* field into its header, covariance etc., it was counted as one line. On the other hand, the world model message files are nicely formatted and verbosely commented, adding a significant amount of lines to the total LOC.

Sample message definition file

A typical message definition file, shown in the following listing, usually contains several entries that hold strictly typed information, message constants and comments:

```
# Message header
Header header

# Object ID
ObjectID objectid

# The header that can be found in pose will be
# used to transport frame_id and stamp
geometry_msgs/PoseWithCovarianceStamped pose

# only for output
string objecttype
```

[9]identified by the *.msg extension

Listing 2.1: WMObject.msg

Message entry

Each message entry contains a type, e.g. *ObjectID* in the example above, and an entry name, *objectid*. When translated to language-native objects, each entry is exposed as a field (cutting out accessor methods). The following example in Java shall demonstrate this functionality:

```
// create a new WMObject
WMObject wmobject = new WMObject();

// create an ObjectID for the WMObject
ObjectID objectid = new ObjectID();
objectid.type = "internal";
objectid.name = "book";

// assign the ObjectID to the WMObject
wmobject.objectid = objectid;

System.out.println(wmobject.objectid.name);
```

Listing 2.2: WMObject Usage in Java

When integrated into an executable Java class, the above code will yield the string *book* as output.

Message constant

Message constants are similar to regular message entries. They are defined by directly specifying the entry's value, as can be seen in the following example:

```
string ENTRYNAME=value
```

Listing 2.3: Example for a message constant

In the example above, **string** describes the entry type, the constant name **ENTRYNAME** is by convention written in capitals and the constant value is defined in-line, using no quotation marks whatsoever.

Comments

Comments are added by prefixing a line with a sharp (#), as in :

```
# This is a comment
```

Listing 2.4: Example for a comment

Primitive	**C++**	**Python**	**Java**
bool	uint8_t	bool	boolean
int8	int8_t	int	byte
uint8	uint8_t	int	short
int16	int16_t	int	short
uint16	uint16_t	int	int
int32	int32_t	int	int
uint32	uint32_t	int	long
int64	int64_t	long	long
uint64	uint64_t	long	long
float32	float	float	float
float64	double	float	double
string	std::string	string	java.lang.String
time	ros::Time	rospy.Time	ros.communication.Time
duration	ros::Duration	rospy.Duration	ros.communication.Duration

Table 2.2: Built-in types

Message Header

The message header contains the coordinate frame id the data is associated with, a time stamp (ROS time) and an automatically generated sequence number.

Message Hierarchy

Message definitions are allowed to contain other message definitions. If they are located in different packages, the containing package has to be specified, as in

```
package_name/MessageName
```

The nesting of message definitions can be arbitrarily deep according to [QCG+09].

Object Orientation

Unfortunately, message definitions don't allow for OO features such as object extension and polymorphism, which is definitely a drawback for designing complex systems. Of course, the mere fact that ROS messages can contain strings enables the developer to use object serialization or marshalling to regain object orientation, however losing the system's out-of-the-box language independence.

Primitives

Table 2.2 (courtesy of ROS.org [ROS11g] and [ROS11k]) shows a set of primitives that can be defined in messages their corresponding types in C++, Python and Java.

2.5.6 Topics

As mentioned before, ROS uses a publisher/subscriber mechanism to transport message packets from one node to another. The channels on which these messages are transported are called topics. Any node can advertise topics by choosing a topic name and a message type it is supposed to carry. On the other hand, nodes can subscribe to these topics and thus receive messages sent on them. Section 2.5.4 explains how topic subscriptions and advertisements are used to establish strictly typed communication channels between nodes. In general publishers and subscribers are not aware of each others' existence [QCG+09]. To avoid collisions between topic names (for example when using multiple cameras, the topic name *camera_rgb* might appear more than once), they can be added to namespaces. In our previous example, we could prefix the topic name by the camera node's name, as in *LeftCam/camera_rgb*.

Helpful tools

There are several tools that help monitoring the communication channels in a ROS system. The topology can be displayed as a graph using the *rxgraph* tool (section 2.5.13), while messages sent on specific topics can be listened to using *rostopic*, with the *echo* parameter supplied. The very same tool is capable of measuring the rate at which messages are sent.

2.5.7 Services

ROS also allows for a service-based communication between nodes, in addition to the publisher/subscriber model that's been discussed so far. Nodes can offer services, that can on the other hand be called by other nodes. Unlike with publishers and subscribers, a service with a certain name can only be offered by one node. Service protocols are described in service definition files (*.srv ending), reusing, however, regular message definitions. A ROS service definition looks very similar to a message definition:

```
ObjectID objectid
---
WMObject wmobject
```

Listing 2.5: GetWMObjectByID.msg

Like message definitions, the content of a service definition is strictly typed. Nesting of service definitions is not possible, however the nesting capabilities of messages can be exploited. ObjectID and WMObject are both defined in the world model (section 3.2 on page 72). The most obvious difference to a message definition are the three horizontal lines that separate the request and response definitions from one another. Accordingly, in the above service definition the service request will contain an ObjectID, to which the response will be a WMObject.

2.5.8 ROS Time

In ROS, time is represented as a tuple of seconds and nanoseconds. This format is contained in message headers and will be displayed by default when printing time information. As these values alone are rather inconvenient to handle, ROS offers time management mechanisms in many languages. Timing values are usually handled in a type-safe manner, with *Time* corresponding to a certain moment, *Duration* to the difference between two points in time and *Rate* to the frequency at which certain events shall occur. Each of these elements exhibit convenience methods for creation and manipulation (e.g. adding time, conversions to a different format). *Rates* are usually used in loops to postpone execution in order to keep up a certain looping frequency. For more information on how ROS handles time in C++, please refer to the corresponding ROS tutorial[10].

2.5.9 Parameter Server

The Parameter Server is a globally accessible data base designed to store and retrieve parameters while a ROS system is up and running. The advantages of such a system are the easier inspectability and manageability of the system configuration. The Parameter Server runs inside the ROS Master (section 2.5.4) and communicates with nodes via an XML-RPC mechanism [ROS11i]. Parameters are defined as tuples of parameter name and parameter value. Just like topic names, parameter names can belong to a certain namespace.

2.5.10 Networking

The ROS communication framework allows to launch nodes on different machines and have them interact in a way that's transparent to the developer. The only requirement is to choose a machine that's assigned the task of running the ROS Master (section 2.5.4). Other machines on the network must know where the Master is running, clearly, to have a unique point of reference when negotiating connections to other nodes. These machines are informed about the Master's URI by setting the *ROS_MASTER_URI* accordingly. A sample command to do so looks as follows:

```
export ROS_MASTER_URI=http://my-ros-laptop.local:11311
```

Optionally, this command can be added to *.bashrc* in order to be loaded when a new console is opened. As can be seen, the ROS Master is located on *my-ros-laptop*[11] and is accessible via port 11311, which is the default port. Consequently, when a node is started on this machine, it will look for the ROS Master on *my-ros-laptop* instead of trying to locate it on localhost. Alternatively, the target machine's IP can be specified, as in

[10]http://www.ros.org/wiki/roscpp/Overview/Time
[11]the local-suffix is required by the *avahi* naming service[12]

```
export ROS_MASTER_URI=http://192.168.5.33:11311
```

ROS was designed for networks with a functioning naming service, however it's also possible to use purely IP-based addressing. If this is the case, **every** machine on the network has to know its **own** *"ROS_IP"*, which is basically the machine's IP. This has to be manually specified using another environment variable, as can be seen in the following command:

```
export ROS_IP=129.27.12.232
```

As ROS was intended to distribute computation tasks over several machines, without a central bottleneck, the communication between nodes follows a peer-to-peer concept. The machine running the ROS Master is not responsible for forwarding all messages traversing the network. Instead, it will manage the negotiation process between two nodes and then have them communicate directly with each other. This mechanism is described in richer detail in [ROS11f]. Consider the following scenario (as described in [QCG+09]): An autonomous, mobile service robot is equipped with a set of compact and not so powerful controller laptops that will perform tasks that go easy on the CPU but are highly I/O-intensive instead (I/O meaning sending and reception of ROS messages). A set of powerful external machines, capable of efficiently performing computationally intensive tasks, is available on the network. To ensure the robot's mobility, it's connected to the offboard computers via a wireless link, whereas offboard as well as onboard computers are among each other connected via LAN. Most of the communication payload circulates among the offboard and among the onboard computers, keeping the required communication on the slow wireless link to a minimum. This is exactly where we can observe the main advantage of the system's peer-to-peer topology. With a central server forwarding the complete communication payload, we'd have to send a high amount of information over the slow wireless link, completely unnecessarily. ROS allows nodes to communicate directly, therefore relieving the wireless link and avoiding the emergence of a bottleneck. The contrast between a star-like topology and the peer to peer topology of ROS is depicted in figures 2.7 and 2.8.

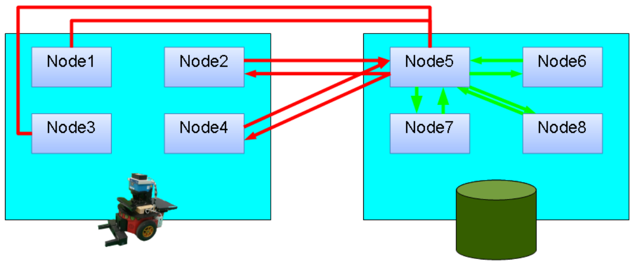


Figure 2.7: Star-Like Communication Topology. The wireless link between the robot and the central server is overloaded with messages.

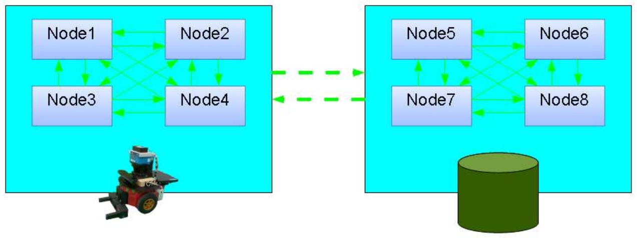


Figure 2.8: Peer to Peer Communication Topology. The wireless link between the robot and the central server is only used for communication negotiation and to transport a limited amount of data. This is the topology used in ROS.

2.5.11 Transformation Framework

One essential concept that must be mentioned in ROS is how it copes with multiple coordinate frames and their transformations. Keeping track of transformations is a tedious task, as well as performing calculations between frames, especially if multiple transformations are involved on the "path" from one frame to another. Thus it's sensible to implement a framework that offers two essential functionalities:

- Providing information on transformations
- Performing coordinate frame conversions

ROS differentiates between Poses and Transforms. The latter are used to convert Poses from one coordinate frame to another, however, they both contain the same type of (essential) information, a 3D coordinate corresponding to a translation and a Quaternion corresponding to a rotation. Poses, as well as Transforms, can thus be converted to a transformation matrix. In the Pose's case, this can be interpreted as the transformation from the coordinate frame's center to the actual pose, whereas in the Transform's case, this is the necessary translation and rotation to convert any Pose from a source frame to a target frame. The Transformation Framework stores and returns Transforms, a message type that contains the source and target frame in addition to the translation and rotation information.

Transformation Framework as a database

The transformation framework stores transformations in a dynamic tree. It can be supplied with and queried for transforms. One of the main advantages of having a dedicated framework handle transformation information is that it's capable to compute composite transformations across several joints. Consider for example a robot that's located somewhere in an indoor office environment, carrying a camera that has just detected a known object. In this setting, the camera has published transformation information between the detected object and its own frame. At the same time, the motorized rotating platform the camera happens to be mounted on reads the motor's current state and thus derives and publishes the transformation between the camera and the robot platform. Thanks to the robot's ability to localize itself in the map, it can publish a transformation between its own position and the center of the map. If the task is to calculate the detected object's global position in the map, it's clearly necessary to calculate three transformations. Fortunately, the transformation framework helps us out here. When requesting the transformation between the object and the map, it combines all the transforms it finds on the "path" between the two coordinate systems to one transformation matrix that it returns as a Transform object. What's left to do is applying this transformation to the object's pose.

Transformation Tree The following block shows an exemplary transformation tree:

```
Tree
   /map
      /odom
         /base_footprint
            /base_link
               /laser
               /camera
                  /marker
```

Both laser and camera are mounted on the robot's base, so there must be a transformation from *base_link* to these two frames. The camera detects a marker and publishes a transformation between its own coordinate system and the marker's frame to the transformation framework. The robot's localization system regularly updates the transformation between the map and the robot's odometry measurement, thus defining the robot's position relative to the map.

Sometimes it's interesting to have transformation information from the past, therefore the transformation framework offers the additional feature of keeping a short history (a couple of seconds) of transformations that can be explicitly requested. This mechanism is described in further detail in [ROS11v].

In ROS, Transforms are usually distributed in a *geometry_msgs/TransformStamped* message. This message reveals the frame of origin, as well as the target frame of the transformation. Furthermore, it contains a 3D-vector corresponding to the translational part and a Quaternion corresponding to the rotational part.

Publishing Transformations

The following C++ code demonstrates how transformations are published to the transformation tree. To do so, a transform broadcaster, an object that communicates with the transformation framework, is required.

```
tf::TransformBroadcaster tfbroadcaster;
tfbroadcaster.sendTransform(
  tf::StampedTransform(
  tf::Transform{tf::Quaternion( rot_qx,
                                rot_qy,
                                rot_qz,
                                rot_qw),
                tf::Vector3(    trans_x,
                                trans_y,
                                trans_z)),
   stamp,
   src_frame,
   tgt_frame));
```

Listing 2.6: Publishing Transformations to the Transformation Framework

Additionally, a timestamp can be set in the *StampedTransform* object, thus specifying at which point in time the transformation was considered valid.

Requesting Transformations

In a similar manner, transformations can be requested from the transformation tree. To this end, a transform listener has to be created (instead of a transform broadcaster, as in the example above). The following listing shows how this object is used to retrieve transforms.

```
tf::TransformListener  tflistener;
tflistener->waitForTransform(
  target_frame,                           // std::string
  source_frame,                           // std::string
  ros::Time::now(),
  ros::Duration(1.0));

if (tflistener->frameExists(target_frame) &&
     tflistener->frameExists(source_frame)
{
    tflistener->lookupTransform(
      target_frame,
      source_frame,
      ros::Time(0),
      returned_transform);      // tf::StampedTransform
}
```

Listing 2.7: Requesting Transformations from the Transformation Framework

This example shows several methods the transform listener class offers. The *waitForTransform* method holds execution (for a maximum duration of one second in this case) until the fresh transform becomes available. This is often necessary as there's usually a brief delay before all nodes publish the requested transformations. Using the *frameExists* method, we can make sure that the frames we've been awaiting are already available. In that case, the *lookupTransform* method can be invoked to retrieve the actual transform, in a *StampedTransform* object.

Applying Transformations

When working with poses and transformations, it's important to distinguish between ROS messages and internal classes that are used for calculations. The naming conventions might be confusing from time to time, for example considering the similarity between *StampedTransform* (the internal representation in the *tf* library) and *TransformStamped* (the ROS message). In C++, static convenience methods that take care of conversions between these types are available, such as

- tf::poseStampedMsgToTF
- tf::poseStampedTFToMsg
- tf::transformStampedMsgToTF
- tf::quaternionTFToMsg

When applying transforms to poses, both have to be available in the internal representation. Consider the following code sample, in which the transformation is represented as a *StampedTransform* object and the pose to be transformed as a ROS message.

```
tf::Stamped<tf::Pose> pose_in, pose_out;
tf::poseStampedMsgToTF(source_pose_msg, pose_in);
pose_out.setData(transform * pose_in);
tf::poseStampedTFToMsg(pose_out, target_pose_msg);
```

Listing 2.8: Transforming Poses using previously fetched Transforms

First of all the source pose is transformed from its message representation to the internal class. Additionally, a variable for the resulting pose (after transformation) is prepared. The actual calculation happens in the third line, where we can observe that coordinate transformation in ROS boils down to applying the *-operator. Finally the class holding the internal representation of the target pose is converted to a ROS message.

Monitoring the Transformation Tree

ROS offers several command-line tools that support developers in monitoring the transformation tree. To get a complete picture of the tree, the *view_frames* node in the *tf* package can be invoked. This node will listen to transformation publications for a while and draft the resulting tree in a PDF file. The *tf_monitor* and *tf_echo* tools provide live text-based information on transformation publications. These tools are described in further detail in section 2.5.13 on page 42.

Visualizing Transformations

The visualization tool RViz features a plugin dedicated to displaying information coming from the transformation framework. Each combination of three lines (red, green and blue) corresponds to the three axes of a coordinate system or frame. In RViz, red corresponds to the X axis (straight forward), green to the Y axis (to the left) and blue to the Z axis (straight up). Figure 2.9 shows a set of transformations as displayed in RViz, when using a mobile robot with a camera mounted (in this case a Microsoft Kinect, described in section 3.1.5 on page 69). The robot base's coordinate frame is shown on ground level, while the robot's laser scanner and vision systems have frames that are located in an elevated position. Arrows between frames denote that they are directly connected in the transformation tree. Clearly, there is a direct connection between the object (bottom left sector) and the robot's object recognition module, as the latter publishes a transformation whenever it recognizes an object.

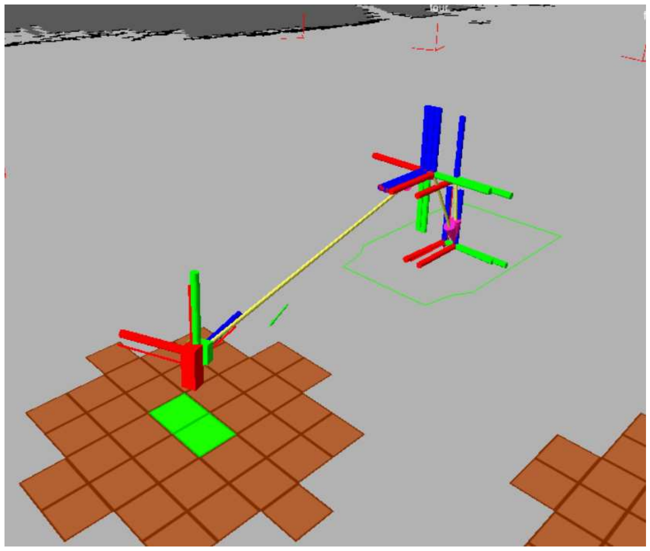


Figure 2.9: Transformation Visualization in RViz

Transformation Framework in Java

At the time when the project described in the practical part of this document was implemented, there was no Java support for the ROS transformation framework available, so Java nodes were unable to communicate directly with the transformation framework. Also, there was no Java library offering convenience methods for applying transformations. Therefore, in the latest implementation of the aforementioned project, a dedicated C++ node handles transformations and offers them to other nodes via service calls. This node is called *tf_adaptor* (section 3.2.7 on page 82) and is located in the *wmstorage* package.

Static Transform Publisher

Nodes might hold transformation information that they regularly publish to the transformation tree, but often this information is static (for example position and orientation of a statically mounted camera) as opposed to dynamic information (for example position and orientation of the robot relatively to the map). In practice, the former case will appear quite frequently, therefore it's useful to have a mechanism that allows to specify the transformation in a concise format and takes care of regularly publishing it to the transformation tree. This mechanism comes with ROS and is called *static_transform_publisher*, a node contained in the *tf* package. It takes the required static transform as a startup parameter and can therefore easily be added to a launch file, as shown in the following listing:

```
<launch>
<node pkg="tf"
       type="static_transform_publisher"
       name="base_link_to_cam"
       args="0.085 0.105 0.30 4.71 0 4.458
              base_link usb_cam 100" />
</launch
```

Listing 2.9: Launch file containing a static transform publisher entry

The transformation is defined in the *args* attribute. The first three floats correspond to a 3D translation, whereas the last three floats denote the rotation, specified as roll, pitch and yaw. Furthermore, source and target frame are defined, as well as the rate at which the transformation should be published.

2.5.12 Point Cloud Library

A point cloud is a convenient and versatile way for representing multi-dimensional data. As the name predicts, a point cloud is a collection of points, theoretically with an arbitrary number of dimensions, practically mostly applied to 3D-space, where they are used to represent spatial information of the environment. The points may contain additional data, such as RGB, intensity values, segmentation results etc. [Ste]

Consider figures 2.10a and 2.10b for example, where an office chair was captured by a 3D measurement device. The data collected is represented in a point cloud and visualized using RViz (described in section 2.5.15).

When stored and processed PCL-internally, point clouds are represented as objects of type *PointCloud*. ROS users, however, will often stumble upon the name *PointCloud2*, which is the name of the corresponding ROS message [Ste].

Sources of Point Cloud Data

There are various ways of collecting spatial information, including Time-of-Flight Cameras, Structured-light 3D scanners, Stereo Vision, tilting laser mea-

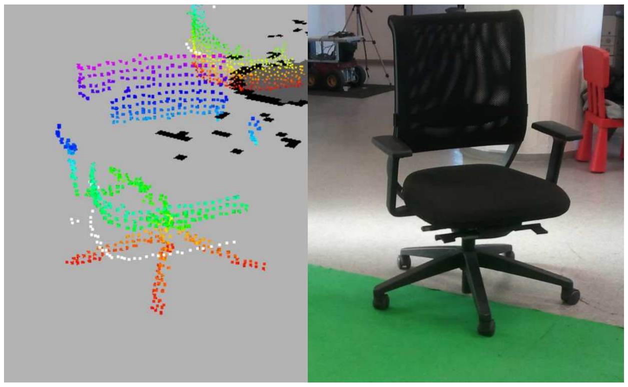

(a) As Point Cloud (b) Real world view

Figure 2.10: Office chair displayed as Point Cloud, next to it the corresponding real world view

surement devices and simulation [Ste]. An example of a Structured-light 3D scanner is the Microsoft Kinect, that was mounted on the robot described in the practical part of this document. Laser measurement devices provide high-quality scans, but are rather expensive and often have a low update rate, due to the mechanical limitations imposed by the tilting device. Stereo Vision is a passive solution, meaning that there's no need for projecting additional light, however this is exactly the reason why this solution is highly dependent on the availability of a texture that offers a sufficient amount of feature points and of proper lighting of course. Time-of-flight cameras are fast, but the resulting resolution is lower (between 64x48 and 176x144 [KBK08]) [Ste].

According to [Ste], the data contained in point clouds is used for three major purposes, namely representation of

1. Scenes
2. Maps
3. Object Models

Scene Representation

3D information of scenes is particularly important for reliable navigation and obstacle avoidance. A mechanism that carries out these tasks in two dimensions is easier to implement, but might be insufficient, especially for large robots or for use in cluttered office environments, where autonomous agents have to exploit any free space they can get in order to achieve their task, including navigating under tables and maneuvering through areas that are filled with obstacles of various heights.

Maps

For robots that don't have an operating space confined to a certain floor of a building, maps containing an additional dimension are sensible. Such robots are unmanned aerial vehicles (usually quadcopters) or ground robots capable of using stairs or an elevator. A three-dimensional map enables these agents to represent the third dimension, which corresponds to height information.

Object Models

Instead of representing an object as a set of areas that correspond to its surface, a finite set of points that are "contained" in the object can be used to model it. The advantage of point clouds is that they can be retrieved directly from sensors providing spatial information (as discussed above). This purely quantitative object information can then be abstracted (object recognition, estimation of object pose, surface normals, etc.) and used for high-level tasks, such as locating, classifying or grasping an object.

Point Cloud Library and ROS

The Point Cloud Library is a collection of modules for 3D point cloud processing. It used to be part of ROS, but was eventually made independent [Ste] [ROS11j], thus following the ROS philosophy of having large libraries thrive independently and integrating them through wrappers [QCG+09].

Processing Point Clouds

There are several algorithms that can be applied to point clouds, including downsampling, filtering, segmentation, removal of outliers, calculation of surface normals etc. [Ste]

Downsampling. Reduces the point cloud's resolution. For many applications, the detail provided by some sensors is higher than necessary, thus wasting resources. Especially visualization tools (such as RViz) often have trouble drawing a large amount of points in space, resulting in slow and laggy performance. By downsampling the point cloud, we can achieve much better performance while not sacrificing too much detail.

Filtering. Some regions of a point cloud might not be relevant for the tasks a robot is given. For these cases, the point cloud library offers mechanisms to filter point clouds (remove points) that are not within certain geometric thresholds, e.g. they are at a height that's not of interest to the robot or they are too far away. An example that demonstrates that less is often more, is when using point clouds for obstacle avoidance. In this case, if the sensor is mounted accordingly, the ground in front of the robot would be added to the point cloud and thus interpreted as an obstacle. This problem can be solved by simply removing all points under a certain height threshold (usually a couple of centimetres).

VoxelGrid Filter. The point cloud library offers a module called *VoxelGrid Filter* that takes care of the two aforementioned tasks. The following launch file demonstrates how it's applied.

```
<launch>
  <node pkg="nodelet"
        type="nodelet"
        name="pcl_manager"
        args="manager"
        output="screen" />

  <!-- Run the VoxelGrid Filter -->
  <node pkg="nodelet"
        type="nodelet"
        name="voxel_grid"
        args="load pcl/VoxelGrid pcl_manager"
        output="screen">
        <remap from="~input"
               to="/camera/depth/points" />
        <rosparam>
            filter_field_name: z
            filter_limit_min: 0.07
            filter_limit_max: 0.75
            filter_limit_negative: False
            leaf_size: 0.03
            input_frame: /base_link
            output_frame: /base_link
        </rosparam>
  </node>
</launch>
```

Listing 2.10: downsample_pointcloud.launch

What's interesting here are the parameters enclosed by the *<rosparam>* tags. The *filter_field_name* defines which field of the point data (in this case the value on the Z axis) to filter. The two lines after that define the thresholds. The *leaf_size* parameter specifies the target resolution in meters. Consequently, the setting above will result in a granularity of 3cm.

2.5.13 Tools

ROS offers a set of convenient tools to aid and speed up development in and usage of a ROS installation. These tools are either ROS nodes themselves, or regular executables. This microkernel design is in accord with the ROS philosophy of offering a thin and tool-based framework [QCG+09].

File System Tools

File system tools aid navigation within and manipulation of the ROS file system, consisting of trees of stacks and packages, usually spread over various folders. To avoid confusion, these tools keep track of package locations and can instantly locate them or use them for build processes. ROS tools usually feel like native Linux commands and therefore offer auto-completion.

rospack. Rospack is the main tool for retrieving information about packages. Its capabilities include locating packages and calculating their dependency trees.

roscd. This is probably one of the most frequently-used tools, when working with ROS. Its purpose is simply to change the working directory to a specific package. It works just like the well-known command *cd*, except that the path leading to the required package does not have to be specified. It's even possible to supply a path after the package's root folder.

rosls. This command lists the directory contents of a ROS package, analogously to the *ls* command. Similarly to *roscd*, the path leading to the package does not have to specified.

roscp. When copying files from one package to another, it's convenient to have a tool that takes care of supplying the path to the remote package. The following line shows the sample usage of *roscp*, as described in [ROS11w]

```
roscp [package_name] [file_to_copy_path] [copy_path]
```

roscreate-stack. This tool generates default versions of files required for ROS stacks, including the stack manifest and a *CMakeLists.txt*, facilitating the stack's distribution process.

roscreate-pkg. Analogously to *roscreate-stack*, this tool generates package contents, along with a folder containing the package. Some command line parameters such as package dependencies can be defined to influence the contents of generated files.

make eclipse-project. For Eclipse developers, this is an invaluable tool that turns any ROS package into an Eclipse project. The only thing left to do after invoking this tool is importing the project into Eclipse.

Installation and Compilation

rosdep. This tool manages external dependencies (system libraries) of ROS. It's capable of calculating dependencies and installs them on demand. The command

```
rosdep install [package]
```

installs all dependencies of the specified package.

rosmake. One of the primarily used tools when working with ROS. The fairly simple usage is demonstrated in the following line:

```
rosmake [package1] [package2] ... [packageN]
```

Cross-package dependencies are resolved automatically, provided the packages are available somewhere on the *ROS_PACKAGE_PATH*. When supplying the *–rosdep-install* parameter, *rosdep* will be used to resolve and install ROS-external dependencies before building the packages.

Execution and Diagnosis.

rosbag. In the field of robotics, it's a common problem that execution results and thus errors are often hard to reproduce, due to the non-determinism caused by various factors. Rosbag[13] supports the debugging process by listening to previously specified topics and dumping the message stream received on these topics to disk. The data is stored in so-called bagfiles. The information gathered can be replayed later, meaning that the messages will be republished with (nearly) the same timing as when they were recorded.

```
rosbag record [topic1] [topic2] ... [topicN]
```

records messages from the specified topics (1 to N), whereas

```
rosbag play bagfile.bag
```

plays back the contents of a previously recorded bagfile.

rxbag. Allows for visualizing a bagfile's contents. The corresponding screenshot (Figure 2.11 is courtesy of ROS.org [ROS11q]).

roscore. As this tool starts the ROS Master, along with the Parameter Server, it should be invoked before starting ROS nodes. Shortly after invocation it will print out the ROS Master's URI which can then be used by remote nodes to access the Master.

rosnode. A tool that monitors the nodes currently running in a ROS environment. Using the **list** parameter, the user will receive a list of running nodes, whereas when typing **info** and the node identifier, detailed information on a specific node will be provided.

[13]http://www.ros.org/wiki/rosbag

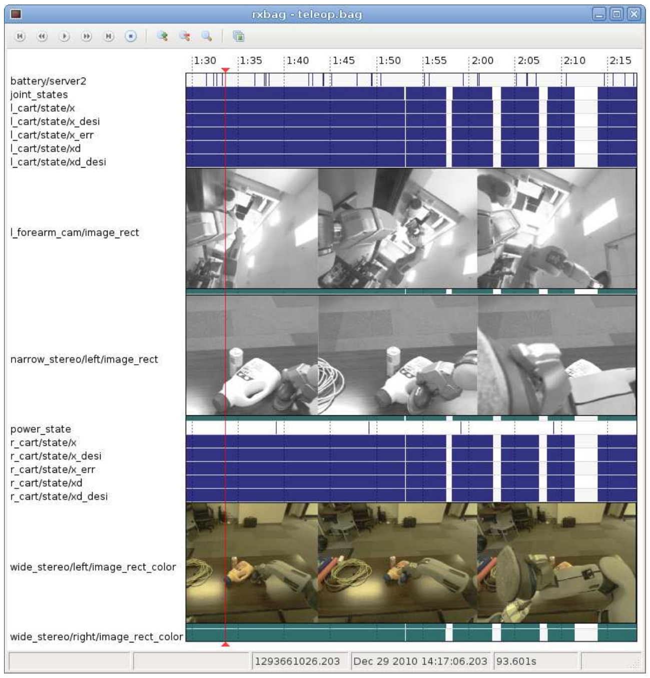


Figure 2.11: Screenshot of the rxbag tool, provided by [ROS11q]

rosservice. Monitors available services in a running ROS system. The **list** parameter yields a list of all services. When using the **type** parameter and adding the identifier of a service, the type of service (namely its service definition name) is returned. The **call** parameter allows to invoke services from the command line. The following line

```
rosservice call /service_name service-args
```

will invoke the service called */service_ name*, passing *service-args* as parameters, and print the service response on the console output. More information on services is available via [ROS11s], whereas documentation of this specific tool can be found at [ROS11n]. The *rosservice* command is not to be confused with *rossrv* (described in the following paragraph).

rossrv. While *rosservice* handles service instances, this tool manages service types. Thus, the relation between these two tools is similar to that between *rostopic* and *rosmsg*. As *rossrv* has exactly the same functionality as *rosmsg*, please refer to the corresponding paragraph for a more detailed description.

rostopic. This is, similarly to **rosservice**, a tool for monitoring active topics in a running ROS environment. The **list** parameter returns a list of advertised topics, whereas the **echo** parameter, with a topic name supplied, will subscribe to this topic and print its contents to the console output. The **hz** mechanism also listens on a specified topic, but instead of printing the message contents measures the rate at which they are published. It's also possible to publish to a topic directly from the command line, by typing the **pub** parameter and specifying the message details. Please refer to the ROS Wiki for more information on this tool[14].

rosmsg. As described in [ROS11l], this tool provides information on ROS message types. The **show** parameter is probably the most frequently used functionality. It displays a specific message's definition. The *rossrv* tool offers exactly the same functionality as *rosmsg*, but applied to services.

rxgraph. Using command-line tools for system monitoring can be a tedious task in large systems, hence ROS comes along with a set of graphical monitoring tools, such as *rxbag*, at which we've already had a glance, and *rxgraph*. Using this tool, the user is provided with a graph depicting the ROS environment, where graph nodes (intuitively) stand for ROS nodes and edges are topics. At one glance, missing connections and thus possible configuration errors can be identified. The graph is automatically updated on the fly. Figure 2.12 shows a simple graph of a running system, with three nodes that are depicted as ellipses. The rectangular elements correspond to topics.

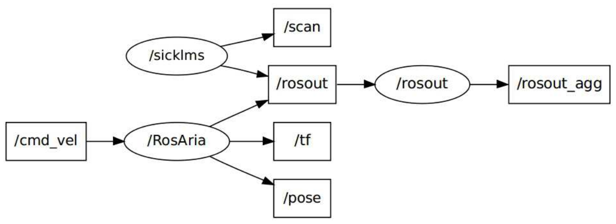


Figure 2.12: Graph of a running ROS system, as visualized in *rxgraph*. The three running nodes are depicted as ellipses, while the rectangular elements correspond to topics.

rxplot. is a virtual oscilloscope that plots information it reads from certain topic.

```
rxplot /topic1/field1 /topic2/field2
```

for example will plot the data of *field1* in *topic1*[15] and the data from *field2* in *topic2* in two separate views. For further documentation, please refer to [ROS11r]. Figure 2.13 shows a screenshot of *rxplot*.

[14]http://www.ros.org/wiki/rostopic

[15]meaning that *topic1* transports messages that contain a field called *field1*

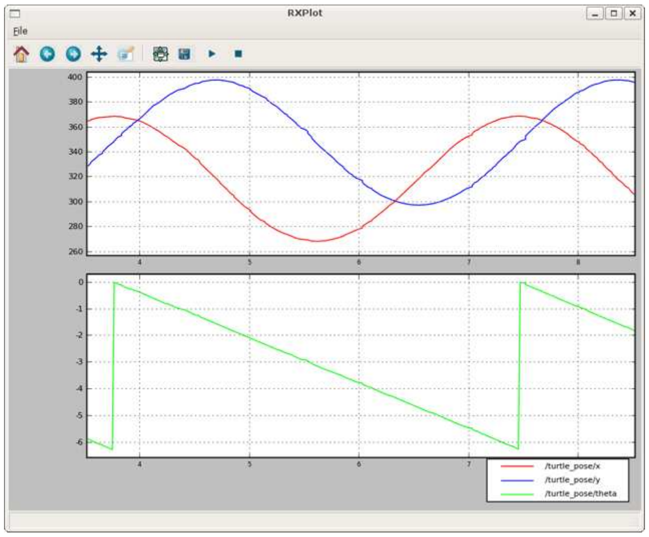


Figure 2.13: Screenshot of the rxplot tool, provided by [ROS11r]

rosrun. Runs a ROS node that was previously built. The ROS node to run is located by passing the package it's contained in and the node name that is unique to the package. Sample call:

```
rosrun [package_name] [node_name]
```

roslaunch. Runs a roslaunch[16] script, that itself will usually launch a list of ROS nodes. The launch script is located by passing the package it's contained in and the file name. Sample call:

```
roslaunch [package] [filename.launch]
```

Unlike with ROS nodes, launch files aren't registered in the package's *CMakeLists.txt* file. However, they are usually located in the *launch* folder. Launch files are coded in an XML format[17]. The following sample shall demonstrate how launch entries are defined.

```
<launch>
  <node pkg="joy" type="joy_node" respawn="false" name="
      joy_node" output="log">
  </node>

  <node pkg="ROSARIA" type="teleop_with_gripper" respawn=
      "false" name="teleop_with_gripper" output="log">
```

[16]http://www.ros.org/wiki/roslaunch
[17]http://www.ros.org/wiki/roslaunch/XML

```
  </node>
</launch>
```

Listing 2.11: teleop.launch - an exemplary launch file

A *node* entry defines a new node to be launched. To be able to locate the corresponding node, the containing package has to be specified using the *pkg* attribute. The *type* attribute corresponds to the node's type (for example the name it was assigned in *CMakeLists.txt*), whereas *name* defines a name for the node instance.

rosparam. Grants command-line control over the Parameter Server (described in section 2.5.9). Amongst the available functionality is getting and setting parameters using the *get* and *set* options, retrieving a list of available parameters using *list* as well as dumping and loading the whole database. Further documentation is available at [ROS11m].

rxloggerlevel. A graphical tool that allows to change the loglevel for each node independently.

rxconsole. A GUI that listens on */rosout* and displays all incoming messages in a structured way. The messages can be filtered by regex, based on their loglevel etc.

roswtf. A tool that looks for inconsistencies in the node graph and elsewhere, for example in launch files. According to [ROS11o], it checks *many, many things, and the list is always growing*.

Transformation Framework Tools

These tools are designed to convey information of the transformation tree's state. They make it possible to monitor which nodes publish transformations, how they are interrelated and what their parameters are.

tf_monitor. Keeps printing a list of nodes that publish transformations and includes information such as source and target frame, as well as the rate at which the transformations are published.

tf_echo. Listens to one or more specific transformations and keeps printing their internal parameters (translation, rotation, source and target frame) whenever they are published.

view_frames. Listens to the state of the transformation tree for a couple of seconds, creates a graph-like representation of the tree and exports this representation in a PDF file.

2.5.14 Debugging with ROS

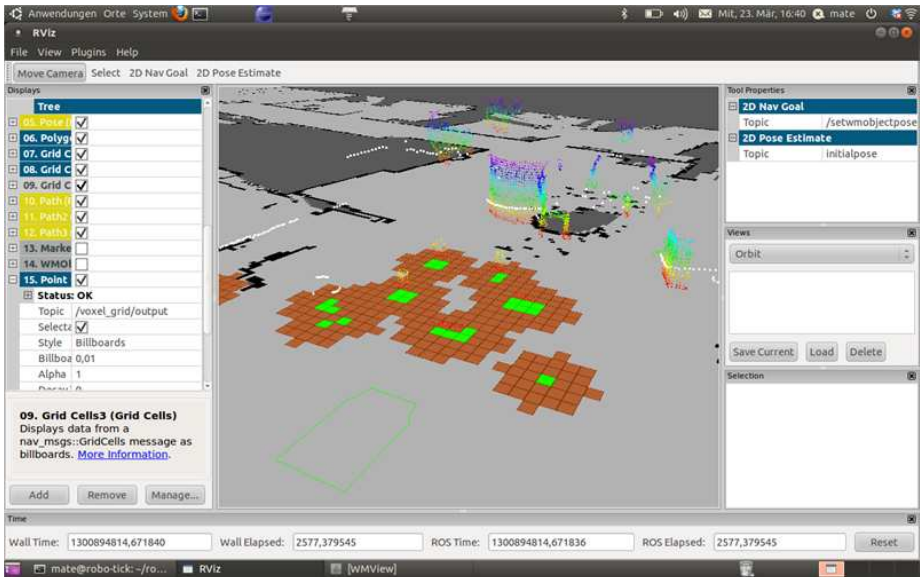
Several tools have been presented, many of which support the debugging process. One of the most powerful features ROS offers when it comes to debugging, is the capability to modify the node graph on the fly, i.e. to stop and restart single nodes while the rest of the software ecosystem is still running [QCG+09]. This is particularly useful in large, complex systems, where frequent restarts would be time-consuming due to long startup times.

According to [QCG+09], the scope of investigation, be it for debugging or enhancement, is often limited to a well-defined area of the system (a couple of software modules or tools). This theory was also confirmed during the project that shall be described in the practical part of this document. Thus, being able to restart single nodes or sets of nodes is a permanently used, invaluable addition that definitely saves development time. It is also important to have nodes under construction run alongside well-tested software modules, as in many cases, they will be dependent on each other's feedback. With ROS, this is also possible. The ecosystem around nodes can even be simulated by recording certain messages and replaying them using rosbag. This tool was already mentioned in section 2.5.13 on page 44.

2.5.15 RViz

It's often convenient to have all information that's circulating in a ROS ecosystem visualized and combined to one comprehensive view. Therefore ROS comes with a visualization tool called RViz, that takes care of collecting information from various sources, such as the robot's footprint, coordinate frames and their transformations, occupancy grids, costmaps, laser scans, point clouds etc. and aligning them to fit in one well-arranged 3D-view of the world. RViz features a plugin-based architecture, meaning that specific modules (called *Display Types*) can be implemented for interpreting message types and displaying them in the application's viewport. So, for instance, to visualize a map and laser scans in RViz, the user would need to add two displays, one for each message type, and connect them to the appropriate topics. Figure 2.14 conveys a screenshot of RViz, with numerous displays activated.

For a complete list of available display types please refer to [ROS11p]. RViz requires up-to-date transformation data to be able to align displays. Consider the use case mentioned above. For the correct alignment of laser scan data and the map, the robot must be localized, hence the localization module must publish transformation data between the robot's base and the map. Additionally, the transformation between the robot's base and the laser scanner must be known. Only with complete transformation data between these two frames can the visualized data be placed in one viewport. What happens if this data is incorrect can for example be observed in cases when the robot is delocalized. RViz requires the user to specify two coordinate frames that serve as reference frames for the 3D viewport. One of them is the fixed frame, which usually corresponds to "world" or "map", the other is the target frame, the one that the



Status: OK
Topic /voxel_grid/output
Style Billboards
Billboa 0,01
Alpha 1

09. Grid Cells3 (Grid Cells)
Displays data from a nav_msgs::GridCells message as billboards. More Information.

2D Nav Goal Topic /setwmobjectpose
2D Pose Estimate Topic initialpose

Wall Time: 1300894814,671840 Wall Elapsed: 2577,379545 ROS Time: 1300894814,671836 ROS Elapsed: 2577,379545

Figure 2.14: Screenshot of RViz in action. The center view shows the aligned view of displays, featuring visualizations of the robot footprint, laser scans, costmap, point clouds coming from a Microsoft Kinect and the map(ground plane). The panel to the left lists displays that have been added to the current configuration.

camera is focused on. For instance, by setting the robot's base as target frame, the camera will "follow" the robot around the map, while setting it to the same as the fixed frame will leave the camera static instead [ROS11p].

2.5.16 Advanced Concepts

This section contains ROS modules that go beyond basic knowledge of the system but are nevertheless frequently used and recommended to have heard of.

Action Library

The ROS Action Library *actionlib* is a system for standardized communication of tasks and their results between certain nodes, based on ROS messages. The basic principle is to have an Action Server running, that offers a certain type of action, which is defined in an action specification file (similarly to service definitions). This server is running within a regular ROS node, that itself has the capability to carry out certain tasks. For example the ROS navigation stack offers Action Servers that listen for navigation commands coming from external nodes. These Action Servers can be triggered using Action Clients. They accept goal-specific parameters and regularly respond to the caller with a progress feedback. After executing the requested action, a message containing the execution result is returned.

ROS *actionlib* is an excellent construct for use in combination with *smach* (described in the next section). For further documentation and tutorials, please refer to [ROS11d] or to the Action Server examples in our project, starting in section 3.3.1 on page 89.

Smach

Smach is a framework that integrates with ROS and allows to define finite **s**tate **mach**ines (hence the name) in Python. Its purpose is to provide a flexible and powerful execution module on an abstract level. The states of the state machine are capable of executing certain tasks, either implemented locally or using an Action Client to connect to an Action Server.

Two types of information are transported across *Smach* state machines, control information and user data, where control information corresponds to the traditional connections between states in a state machine and user data is additional data that can serve as input parameters to states. For example, if a state is responsible for having the robot navigate to a certain spot, it would be impracticable to create another state for another spot and so on. Instead, coordinates can be passed as user data when entering the navigation state. Such information can be collected by other states that, for example, query a ROS service. *Smach* offers several pre-implemented state types, for querying services, calling Action Servers etc. State machines can be nested, and can themselves be wrapped to form Action Servers. The task of the resulting server is then to execute to state machine within.

ROSJava

As mentioned in an earlier section, ROS supports nodes written in Java, however this is not natively implemented but based on a C++ node API that's called via JNI. According to [ROS11k], *rosjava* is still in early alpha state, and the API is subject to change, however it's well-supported.

Getting started with *rosjava* development is not as straightforward as with other languages, but [ROS11k] describes the steps to a successful configuration in sufficient detail. Message handling in *rosjava* used to be an issue in earlier stages of the project, as there were differences in how message definitions were interpreted in Java and other ROS-supported languages. Also, as *rosjava* is not yet ROS-native, basic system messages that come with ROS aren't translated to Java by default. The *rosjava* package remedies this problem by translating not only the package-specific messages, but also messages in related packages. The Java implementation of these messages are then stored in the package under construction. Furthermore, the API is not yet complete and many convenient libraries such as the point cloud library are not translated, but as the *rosjava* package is actively being extended, it can be used without hesitation.

2.6 Situation Calculus

The Situation Calculus, proposed by John McCarthy in 1963 [McC63] and further developed by Raymond Reiter and others, is a methodology for reasoning about actions and change. It allows the definition of possible actions and their consequences in a logic-based language and thus offers elegant ways of proving and testing properties. Some fundamental problems in AI, such as the frame problem, can be solved using Situation Calculus. However, it also suffers several drawbacks, namely the high complexity of domain models and the errors resulting thereof, as well as its computational complexity and the fact that it's based on second order logic [FS10].

Before we continue with details about the Situation Calculus, two problems in AI shall be mentioned here.

2.6.1 The Qualification Problem

For this we'll leave the action domain for a while and dive into the world of biology. We have the task of defining which animals can fly, which we solve in the following term [Rei01]:

$$flies(x) \rightarrow bird(x) \wedge \neg penguin(x) \wedge \neg ostrich(x) \wedge \neg pekingDuck(x).$$

This is absolutely true, but the problem is that we can never infer that an animal flies, just the other way around. So let's flip our term [Rei01]:

$$bird(x) \wedge \neg penguin(x) \wedge \neg ostrich(x) \wedge \neg pekingDuck(x) \wedge \ldots \rightarrow flies(x).$$

Clearly, for this term to be correct, we have to enumerate all factors that are relevant to whether an animal can fly or not. Once we have exhaustively listed all factors, imagine we don't have the information whether the animal we have found is a *pekingDuck* or not, but we know that it's a bird and so on. Then the result will be *false*, although we're "pretty sure" it's *true*. This is why, to solve this problem, we distinguish between *important* and *minor* qualifications. Information on important qualifications must be available and the requirements must be met. Minor qualifications must still meet the required conditions, but if they are unknown, we can ignore them.

2.6.2 The Frame Problem

The Frame Problem concerns the relation between actions and their effects. It's the problem of completely defining which actions can cause which effects, which becomes impracticable with a growing number of actions and effects, if this is done manually. This issue is addressed in section 2.6.4 below, and is described in great detail in [Rei01].

2.6.3 Basic Elements

The basic elements of the Situation Calculus are Actions, Situations and Fluents. These shall be explained here.

Actions

In terms of Situation Calculus, performing Actions is the only way to impose changes on the world state. This becomes logical if we imagine that the expression "an action has been performed" is equivalent to "something has happened". Each Action has its unique name that distinguishes it from other Actions. It is guaranteed that Actions with different names perform different things [FS10]. Situations in terms of Situation Calculus will be mentioned later on, nevertheless it's important to anticipate that carrying out an Action a in a certain Situation s leads to a new Situation s', which is an essential principle of Situation Calculus [FS10]:

$$s' = do(a, s)$$

The *do* function signals that Action a is executed in Situation s. This is only possible if all the preconditions, that can be defined for Actions, hold. As Actions can have an arbitrary number of parameters, we could rewrite the above statement as:

$$s' = do(a(p_1, p_2, \ldots, p_n), s)$$

Situations

Situations in terms of Situation Calculus can be seen as sequences of Actions, or as the history of executed Actions. As described above, Situations can only change as Actions are performed. Each execution starts in situation S_0, the initial situation in which nothing has happened so far. Each Action that is performed is then added to the latest situation. A typical situation can be written as follows [DGLLS09]:

$$do(put(A, B), do(put(B, C), S0))$$

In this example, we can see how calls of the function *do* are nested to build a history of executed Actions, yielding the current situation. Suppose an agent is performing the Actions in the example, then he'd have started in situation S0, then put block B on block C and finally block A on block B. Please note that the situation in terms of Situation Calculus is more than just the arrangement of the blocks after executing these Actions. Consider the example

$$do(turn360, do(put(A, B), do(put(B, C), S0)))$$

and suppose the agent has performed an additional 360 degree turn. Notice how performing a new Action corresponds to its concatenation to the sequence of previous Actions. The block arrangement and even the agent's position and orientation are the same as before, however, this is still a new Situation, because an Action has been performed.

Fluents

Fluents describe properties of the world, such as an object's position or whether a ball is yellow. The former example corresponds to functional Fluents, the latter is an example of a relational Fluent. We could state that the set of Fluents describes the current world state. However, it does *NOT* define the current Situation. Several distinct Situations might hold the same set of Fluents.

Relational Fluents. Simply explained, relational Fluents provide information on world properties in boolean form. These can be relations between objects or properties of objects. The formal notation of relational Fluents is ([FS10])

$$F : (objects \cup actions)^n \times situation \mapsto True, False$$

They are represented by a predicate symbol with arity n+1 (n parameters and a Situation), and can be written as $F(x_1, \ldots, x_n, s)$ [FS10]. To stick to the example above, we could query whether a ball b is yellow in Situation s using the expression $yellow(b, s)$.

Functional Fluents. As opposed to relational Fluents, functional ones map to the set of Objects and Actions ([FS10]):

$$F : (objects \cup actions)^n \times situation \mapsto (objects \cup actions)$$

Thus, they can answer questions such as *"given a certain Situation, what's the distance between the box and the desk?"*. Functional Fluents are represented by a function symbol with n+1 parameters and can be formally written as $f(x_1, \ldots, x_n, s)$ [FS10]. To answer the previously asked question, we could write $distance(b, d, s)$.

2.6.4 Basic Action Theories

Basic Action Theories model available Actions, their consequences to the world and other rules that apply. They hold the following types of axioms [DGLLS09] [FS10]:

- Initial State axioms $\mathcal{D}_{S_0}$
- Precondition axioms $\mathcal{D}_{ap}$ (represented by the special predicate $Poss(a, s)$)
- Successor State axioms $\mathcal{D}_{ssa}$ (described below)
- Unique Name axioms $\mathcal{D}_{una}$
- Foundational axioms for Situations $\sum$

Successor State Axioms

Successor State Axioms define how the world changes when Actions are carried out. In a world without exogenous events, only Actions can change the values of Fluents, thus by correctly specifying all Successor State Axioms, we have completely modeled the effects of our Actions. Unfortunately, due to the frame problem, we have to describe **all** possible effects, leaving us with $2 \times |F| \times |A|$ axioms [FS10]. This number is practically unmanageable in a real-world application. Effect axioms can be noted as follows [FS10]:

$$\begin{aligned} \varepsilon_F^+(x, y, s) &\rightarrow F(x, do(a(y), s)) \\ \varepsilon_F^-(x, y, s) &\rightarrow \neg F(x, do(a(y), s)) \end{aligned}$$

The first type of effect axioms described above are the ones that alter a relational Fluent's value to become true, while the second type has the opposite effect. Sample effect axioms taken from [FS10]:

$$\begin{aligned} fragile(x, s) &\rightarrow broken(x, do(drop(x), s)) \\ false &\rightarrow broken(x, do(paint(x), s)) \end{aligned}$$

The aforementioned notation can be altered to become [FS10]:

$$\begin{aligned} F(x, do(a, s)) &\equiv \gamma_F^+(x, a, s) \vee (F(x, s) \wedge \neg\gamma_F^-(x, a, s)) \\ f(x, do(a, s)) = y &\equiv \gamma_f(x, y, a, s) \vee (f(x, s) = y \wedge \neg(\exists y')\gamma_f(x, y', a, s)) \end{aligned}$$

Consider the following sample axiom that's based on the notation above:

$$ison(x, do(a, s)) \leftrightarrow a = (flipswitch(x) \wedge isoff(x, s)) \vee (ison(x, s) \wedge a \neq flipswitch(x))$$

In natural language, this means that a Fluent will be true after performing an Action a in Situation s, if and only if something has happened to make it true or in case it was already true in Situation s and nothing has happened to make it false. This notation reduces the number of required axioms to $|F|$, but at the same time lets them become more complicated than before. We have to make sure they remain short and simple by assuming that Fluents are altered by a very limited set of Actions [FS10].

2.6.5 Golog

Based on the Situation Calculus a program language for dynamic systems, Golog (al*Gol* for *Log*ic), was created. As opposed to Situation Calculus being a purely theoretical construct, Golog is a practically applicable programming language built on top of Situation Calculus, that can be interpreted using Prolog [FS10].

Reasoning about Action

Before diving into Golog, two essential tasks of reasoning over action have to be mentioned, namely the

- Temporal Projection Task
- Legality Task

The former describes the problem of telling whether a sentence will hold in a certain future situation, while the latter revolves around the question whether a sequence of Actions is executable at all, starting from some initial state. The Legality Task can easily be reduced to a Temporal Projection Task, provided that Action preconditions are defined. All we need to do is prove that the last Action's preconditions are satisfied one Situation before the final one [DGLLS09]. The Projection Task can be written formally as the problem of determining whether the following equation holds ($\mathcal{D}$ being a basic action theory, and a_1 to a_n being Actions) [DGLLS09]:

$$\mathcal{D} \models \phi[do([a_1, \ldots, a_2], S_0)]$$

The Temporal Projection Task can be solved using regression [DGLLS09].

Regression. To prove whether a sentence holds in a future Situation, we can roll the sentence (or query) back in time, until we reach the initial situation and prove that the altered sentence holds there. This procedure is referred to as regression. Given a sentence W, a regression operator $\mathcal{R}$ and a basic action theory $\mathcal{D}$, we can write formally [FS10]:

$$\mathcal{D} \models W \leftrightarrow \mathcal{D}_{S_0} \cup D_{una} \models \mathcal{R}[W]$$

The Clark theorem states that a Prolog interpreter can solve this problem [FAU10].

Semantics of Golog. Golog is based on Situation Calculus and extends it to be applicable to the world of programming. The macro $Do(\delta, s, s')$ states that s' is reachable from s by executing the Golog program δ [FS10].

The following list taken from [FS10] describes the set of mechanisms Golog offers to the developer:

- Primitive Action a
- Test Action ϕ?
- Sequence: $\delta_1; \delta_2$
- Non-deterministic choice of Actions: $\delta_1 | \delta_2$
- Non-deterministic choice of arguments: $(\pi x)\delta(x)$

- Non-deterministic iteration: $\delta*$
- Conditionals: $if\ \phi\ then\ \delta_1\ else\ \delta_2\ endif$
- Loops: $while\ \phi\ do\ \delta\ endwhile$
- Procedures: $proc\ P(x)\ \delta(x)\ endproc$

Unfortunately Golog is not a good solution for agents operating in the real world, as it requires complete knowledge and absolute determinism, which usually isn't available or realistic. Performance becomes an issue, as Golog simulates the whole program trace before executing the first action (off-line semantics). The step to an executable implementation (in Prolog) requires us to leave the pure clean world of Situation Calculus, and we suddenly have to worry about implementation details and run-time issues [FS10].

Successors. Several extensions to Golog have followed, including ConGolog [DLL00], DTGolog [BRST00] and IndiGolog, the latter of which should be discussed in further detail here.

2.6.6 IndiGolog

IndiGolog (Incremental Deterministic Golog) is a successor of Golog and introduces several features that make it applicable to real-time agents. Such features are (as listed in [FS10]):

- on-line execution semantics
- concurrent execution
- sensing and exogenous events
- interrupts

It offers the following set of constructs, in addition to those provided by Golog [FS10]:

- Concurrency (equally prioritized): $\delta_1 \parallel \delta_2$
- Concurrency (δ_1 prioritized higher): $\delta_1 \rangle\rangle \delta_2$
- Concurrent iteration: $\delta^{\parallel}$
- Interrupt: $\langle \Phi \rightarrow \delta \rangle$
- Search Operator: $\sum(\delta)$

IndiGolog's on-line semantics allows it to choose the best-seeming action and execute it, without simulating the whole program trace in advance. However, there are situations that require a certain lookahead, before initiating execution.

As by default, IndiGolog will use on-line semantics, it provides the search operator $\sum$ that allows the developer to explicitly request off-line simulation for certain parts of a program. In these cases, sensing information is derived from successor state axioms and, to guarantee completion, non-deterministic choices are made [FS10].

Thus, an IndiGolog program δ will be executed on-line, except if it's written as $\sum(\delta)$, whereas Golog programs are always executed off-line, as if they were surrounded by a search operator.

As already stated in section 2.6.5, the Projection Task can be solved using regression. However, during long runs, this procedure can become computationally expensive due to the increasing history of Actions. To remedy this problem, we can roll the initial Situation forward using the **progression** operator $\mathcal{P}$ ([FS10]).

$$\mathcal{D} \models \Phi(do(a, S_0)) \leftrightarrow \mathcal{D}_0' \cup \mathcal{D}_{una} \models \Phi[S_0], \mathcal{D}_0' = \mathcal{P}(\mathcal{D}_{S_0}, a)$$

This way we can keep the history short, but on the other hand forget all Actions before the new initial Situation. In practice, the right balance between regression and progression has to be found [FS10].

IndiGolog features two new predicates, namely $Trans$ and $Final$.
$Trans(\delta, s, \delta', s')$ denotes that by executing program δ in Situation s, the resulting Situation will be s' and we'll be left with the rest of the program, δ', whereas $Final(\delta, s)$ states that program δ can terminate in Situation s [FS10].

Sensing

For real world applications, it is important not only to have an effect on the environment but to receive sensing information that can subsequently be related to Fluent states. Sensing is always related to actions, it's the programmer's choice to either define dedicated sensing actions or relate sensing information to primitive actions [FS10].

Sensing information μ is incorporated in the execution history σ, resulting in the following structure [DGLLS09]:

$$\sigma = (a_1, \mu_1) \cdot \ldots \cdot (a_k, \mu_k)$$

Depending on the sensed information, the high-level program can either decide to stop (in a final Situation), return the remainder of the program, or to perform an Action and then return the remainder of the program. The following three expressions correspond to the aforementioned choices [DGLLS09]:

1. Stop: $\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models Final(\delta, end[\sigma])$;[18]
2. Return δ': $\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models Trans(\delta, end[\sigma], \delta', end[\sigma])$
3. Return action a and δ': $\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models Trans(\delta, end[\sigma], \delta', do(a, end[\sigma]))$

[18] $\mathcal{C}$ is the set of axioms that defines Trans and Final [DGLLS09]

An IndiGolog interpreter is available in Prolog. The main cycle is capable of handling exogenous events by a posteriori adding them to the history, performing progression and regression and incorporating sensing results in the history [FS10].

2.7 Object Recognition

To interact with the surrounding world, autonomous agents need mechanisms that allow them to recognize objects they can manipulate. Object recognition can be based on visual methods or rely on other sensory input, such as RFID. This work is focused on the former, an expression comprising several approaches, two of which shall be mentioned:

Untagged Object Recognition. Recognition of untagged objects means finding instances of previously defined models of objects in an input image. This approach is beyond the scope of this work and therefore won't be treated in this document. Having no limitations on the type of objects to recognize and their textures yielded a rather pragmatic approach using AR (Augmented Reality) Tags, promising high robustness in object classification and pose estimation.

Visual Recognition of Tagged Objects. Another mechanism solving object recognition involves application of tags to objects. These tags should have properties that are favorable to computer vision methods, such as sharp, clear edges and high contrasts. An example for such tags are AR (Augmented Reality) tags, an example of which is shown in Figure 2.15.

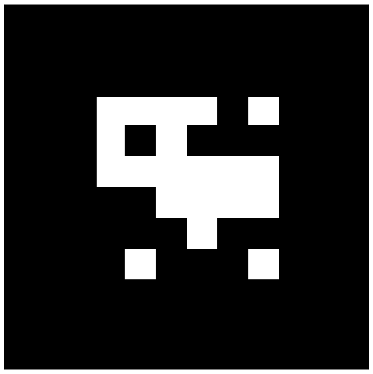


Figure 2.15: An AR (Augmented Reality) Tag

They can be used in combination with computer vision mechanisms, to determine the coordinate transformation between a tagged surface and the camera. As the name predicts, a popular field of application is Augmented Reality, where virtual objects can be overlaid with the view of the real world. An approach to and application of AR Tag Recognition and pose estimation is described in [KB99] and depicted in Figure 2.16.

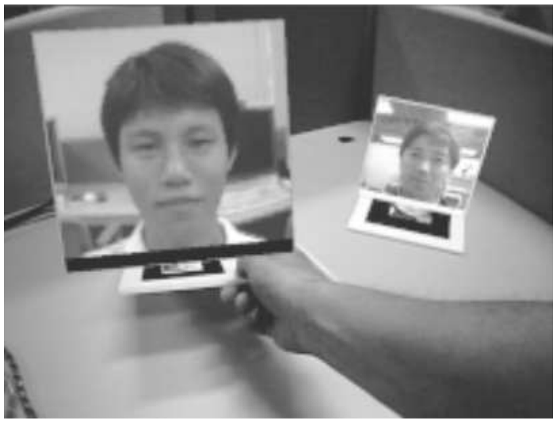


Figure 2.16: A sample application of AR Tag Recognition, as described in [KB99]

Furthermore, tags can be differentiated between by storing information about the pattern contained in each tag and either interpreting the content as a 2D-barcode or assigning the tag to the matching class based on similarity measures. The project described in [KB99] makes use of the latter method, while there are also approaches based on the former [ROS11a].

2.7.1 AR Tag Recognition Basics

The underlying recognition mechanism shall be concisely explained here. In a first step, the incoming image is thresholded, and the four lines that correspond to the AR marker's thick edges (see Figure 2.15), are extracted. As the original size of the AR marker is previously known, these lines can be used to calculate a transformation from the slanted image to the frontal view of the marker. In the case where a similarity-measure-based approch is applied, the sub-image within the thick black edges of the marker is compared with all known patterns to return the best match. This type of pattern recognition is prone to yield false positives, but at the same time any kind of human-readable information can be placed in the tag.

2.7.2 AR Recognition backed by Depth Information

Depth information (retrieved for example using stereo vision) can help to achieve more precise estimation of AR Tag poses. During the ROS 3D Contest[19], a project was presented in which point cloud information coming from a Microsoft Kinect 3.1.5 in combination with the original AR recognition package

[19]http://www.ros.org/wiki/openni/Contests/ROS 3D

ar_pose was used to improve the pose estimation process. Consequently, the estimate became reliable enough to allow for robust localization inside an office environment[20]. This improvement was achieved following a two-step process:

1. Initially, the basic 2D AR recognition method would detect markers in an image and return their respective centers.
2. In a second step, depth information that's aligned with the RGB image was used to calculate a surface normal at the AR tag's position that was collected in the first step. This new information was then used to correct the tag's pose estimate.

When calculating surface normals, an increasing surface size improves the chances of retrieving precise results, thus this method works best if tags are placed on walls or other flat surfaces, this not being a drawback for the researchers at the University of Albany, who came up with this method[21], as their tags were placed on office walls, where the robot would recognize them and localize itself. Of course this mechanism can be turned around to detect manipulable objects. Section 3.4.4 on page 99 shows how we applied this method for object recognition purposes.

[20]http://www.ros.org/wiki/openni/Contests/ROS 3D/Improved AR Markers for Topological Navigation

[21]ROS package: *ar_kinect*

Chapter 3

Practical Part

The theoretical part of this document presented essential knowledge for implementing a dependable[1], autonomous mobile delivery robot that can serve as a basis for further experiments.

The remainder of this document describes the step from theory to practice. The aim of the practical part of this project was twofold. On the one hand a world model with main focus on storage of quantitative information had to be implemented, on the other hand a fully functional delivery robot had to be configured, featuring the newly implemented world model, as well as a connection to an IndiGolog Highlevel. This latter task required coding several ROS nodes and numerous testruns on the live system to come up with a functioning configuration for complex modules such as the navigation stack or the AR recognition tool. The world model, on the other hand, evolved into a system supporting storage of qualitative object information (attributes and relations), thus allowing for reasoning. The aims of this project can be summarized as follows:

- **Implementation of a world model**
  - Focus on quantitative information
  - Extendable to storage of qualitative information
  - Central Point of Reference
  - Long-term memory
- **Build an experiment-ready, autonomous delivery robot, based on the Pioneer P3-DX platform**
  - Equip robot with necessary hardware (laptop, cameras, 2D gripper)
  - Enable robust navigation with obstacle detection beyond the laser scanner's limitations

[1]dependable $\equiv$ failsafe, reliable

  - Object recognition using AR Tags
  - Integration of execution frameworks (smach, IndiGolog)

- Carrying out experiments

- Management (and partly creation) of institute-internal infrastructure (Subversion Repository, Wiki, Hardware management)

3.1 Hardware

As mentioned above, the aim of this project was not only the implementation of software modules but also their application to a real robot. We opted for a Pioneer P3-DX robot platform, as it was already available at our institute, along with valuable know-how concerning communication with the robot and installation of new hardware, and because of its advantageous features, such as a compact footprint and expandability.

The robot platform was equipped with several extensions, including a 2D gripping device, a SICK laser measurement unit and a Microsoft Kinect. Instead of using the robot's onboard computer, we connected it to a controller laptop, small enough to be carried by the robot and robust enough to withstand minor collisions.

3.1.1 Robot Base

The robot platform in use is a Pioneer P3-DX with onboard computer, a sonar array and an optional 2D gripping device. According to ActivMedia Robotics, LLC[2], the robot can already function as a fully autonomous agent without any supplemental equipment, however, we required additional hardware for our purposes. Technical specifications in the following sections have been derived from the Pioneer Manual published by ActivMedia Robotics, LLC [Act03].

The P3-DX offers a serial port (RS-232 compatible 9-PIN DSUB) for connections with external computers. To link the controller laptop with the robot platform, we used a Digiport Serial to USB (4 to 1) converter. The Pioneer P3-DX is communicated with using a package called *ROSARIA*. It's responsible for robot and gripper control, as well as feedback. The original package comes from the University of Zagreb, however we have an altered version in our local repository, as a few changes were required. The *RosAriaWithGripper* is the latest version, including (as the name predicts) gripper support.

Robot Drive

Two high-speed, high-torque reversible-DC motors enable the robot to perform rotational as well as translational movements. The two motorized wheels can be triggered individually. To allow for high-quality odometry measurements

[2]http://www.mobilerobots.com/

(precise position and speed sensing), the robot features high-resolution optical quadrature shaft encoders. The P3-DX performs position integration, saving its current location (x, y, θ) in its internal coordinate system. This value can be manually reset at any time.

In total, the robot platform is standing on three wheels, the aforementioned motorized wheels and a smaller, stabilizing swivel castor wheel on the rear side of the robot body. When mounting additional equipment, it is recommended to keep the center of gravity over the drive wheels. Figure 3.1 shows how heavy instruments, such as the SICK laser measurement system, are mounted on the robot platform.

As mentioned before, the robot can perform translational and rotational movements. This can be done by either controlling each wheel independently or by sending more abstract commands for straight motion or rotation. In the latter case, the robot takes care of precisely maintaining a direction and balancing interferences caused by rough paths, hence this is the recommended way of controlling the robot.

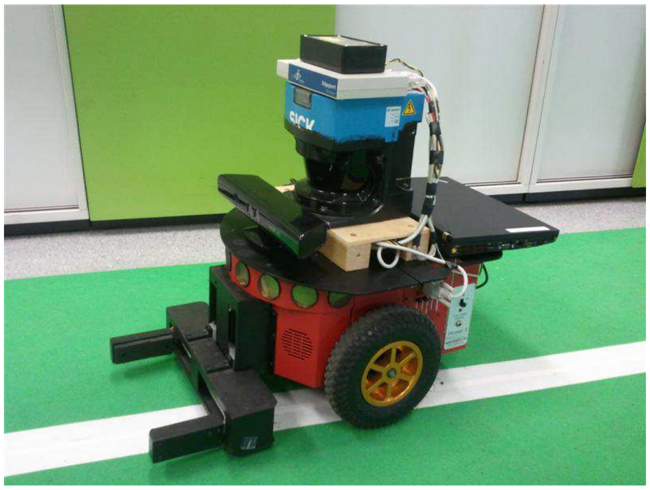


Figure 3.1: Pioneer P3-DX platform with equipment at the Institute for Software Technology, Graz University of Technology

Batteries

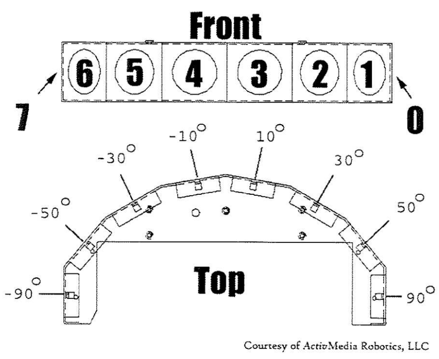
The P3-DX is powered by up to three hotswappable, 7 ampere-hour, 12V DC sealed lead-acid batteries, with the actual output voltage varying from 12,5V (in fully charged state) to 11,5V (charge state considered low) or less. Fresh batteries offer approximately 6 hours of runtime with active motors and no supplemental equipment, and approximately 4 hours when using the onboard computer. With its motors deactivated, the robot lasts several days.

To increase the robot's stability, it's good practice to operate it with three batteries. If less than three batteries are inserted, it is important not to leave the robot unbalanced, thus when using one battery, it should be placed in the middle slot, two batteries on the other hand should be located at the outer slots.

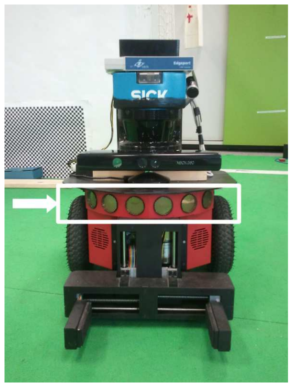
Batteries can be charged either externally using additional lead-acid chargers, with a charger that connects directly to the 12V power plug on the robot's left side, or with a docking/charging station, if the robot is equipped with the required current collector. A docking platform is particularly useful in longterm experiments, as it offers a way for the robot to autonomously recharge itself.

3.1.2 Sonar

The Pioneer P3-DX is equipped with a sonar array on the front, consisting of 8 individual sonars, two on each side and 6 facing outwards, offering a sensing radius of approximately 180 degrees altogether. The range information they provide can be used for feature recognition, localization and navigation. Figure 3.2 shows the arrangement of sonars on the Pioneer P3-DX. Due to the availability of other sensors (such as a Microsoft Kinect and a SICK Laser Measurement Unit), the robot's sonar array wasn't applied in this project.



(a) Schematic view [Act03]



(b) Placement on robot

Figure 3.2: Sonar Array

3.1.3 Gripper

To enable the robot to manipulate objects in its environment, a compatible 2D gripping device was mounted on the front of the robot's body. This device allows for motion in 2 directions, one is the closing or opening motion of gripper paddles, the other is upwards or downwards motion of the gripper lift. The gripper paddles are equipped with two types of sensors:

- pressure sensors on the inside of each paddle

- two light barriers monitoring the gripping area, one on the gripper's nose, one farther behind, closer to the robot

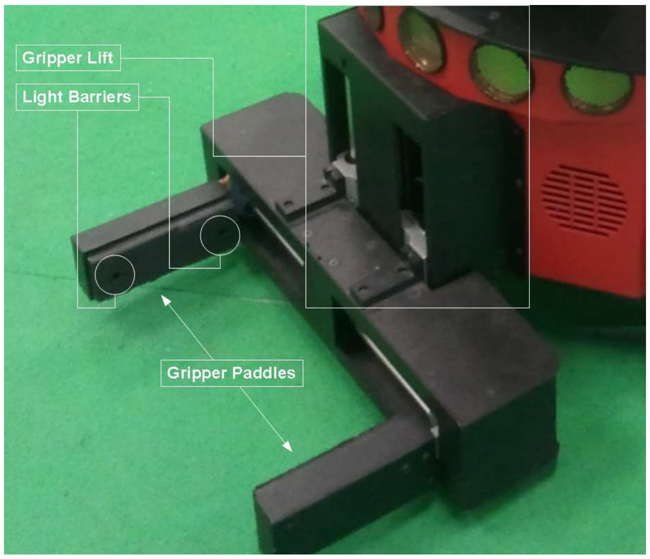


Figure 3.3: Components of the 2D Gripping Device

Int16 Value	**Effect**
1	**Close** gripper paddles
2	**Open** gripper paddles
3	Move lift **up**
4	Move lift **down**
0	**Stop**

Table 3.1: Gripper Commands

When receiving a command requesting a motion, the appropriate gripper motor will remain activated until reception of a new command.

Gripper Feedback

Several modules, such as action servers, require feedback from the gripper sensors, to know when to cease execution of a gripping task for example. This information is provided by the robot platform, but has to be made available to ROS nodes. The *RosAriaWithGripper* node therefore queries the gripper state at a rate of 10hz and publishes the result in a message called *Gripper.msg* on the *gripperinfo* topic. The following listing shows the aforementioned message's definition.

```
Header header

# constants used for field "grip_state"
int32 GRIP_STATE_INBETWEEN = 0
int32 GRIP_STATE_OPEN = 1
int32 GRIP_STATE_CLOSED = 2

# breakbeams (light barriers)
bool outer_breakbeam_broken
bool inner_breakbeam_broken

# paddles
bool left_paddle_triggered
bool right_paddle_triggered

# 0 if gripper paddles between open and closed
# 1 if gripper paddles are open
# 2 if gripper paddles are closed
int32 grip_state

# lift position/motion
bool lift_maxed
bool lift_moving

# gripper motion
bool grip_moving
```

Listing 3.1: Gripper.msg

3.1.4 Laser Measurement Unit

Our Pioneer P3-DX is equipped with a SICK Laser Measurement System (SICK LMS200), a device suiting industrial needs in the fields of

- object measurement
- determining positions
- area monitoring

Information in this section was retrieved from [SICa] and [SICb], both being SICK LMS manuals. The SICK laser scanner provides arrays of range measurements in a field with a radius of 180 degrees and a maximum distance of 80m, at various rates depending on the chosen resolution. The technology applied is optical measurement using an infra-red class 1 laser and an infra-red sensor. As the time of flight of laser light pulses is directly proportional to the distance of objects reflecting the pulses, this value can be used for precise ($\pm$15mm) measurement of object distances. By directing the laser beam onto a rotating mirror[3] inside the device and repeating the measurement at a high rate, a horizontal "slice" of the environment can be obtained. Diagrams of the mechanism are shown in Figure 3.4 [SICa] [SICb].

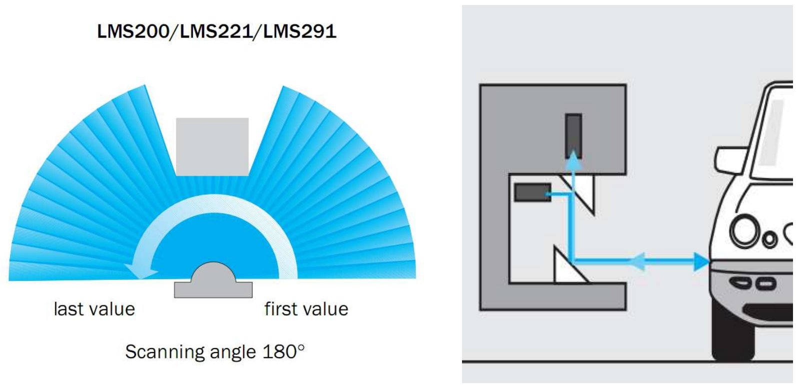


Figure 3.4: Laser Measurement Mechanism

Connecting the Laser Scanner

The SICK LMS200 connects to other devices through an RS-232 compatible serial port. In our case it's connected to a Digiport Serial to USB (4 to 1) converter, just like the robot base. This is particularly convenient, as one USB plug suffices to connect both devices to a laptop computer. The laser scanner can be accessed using the *sicktoolbox* library. The *sicktoolbox_wrapper* package takes care of integrating this library into the rest of our ROS installation, and publishes laser scan results as messages of type *sensor_msgs/LaserScan*. It also allows to specify parameters concerning laser operation such as the baud rate, which should be one of 9600, 19200, 38400 and 500000, as well as the scan resolution. Table 3.2 shows the available scan resolutions and the consequential response times.

Resolution	**Response Time**
0,25°	53,33ms
0,5°	26,66ms
1°	13,33ms

Table 3.2: Laser Scan Resolutions and Response Times

[3]Rotating at a rate of 75hz

RViz is capable of visualizing scan results, and given a correct transformation between the laser's frame and the world frame, align them with other views, as can be seen in Figure 3.5 (with white points corresponding to scan results).

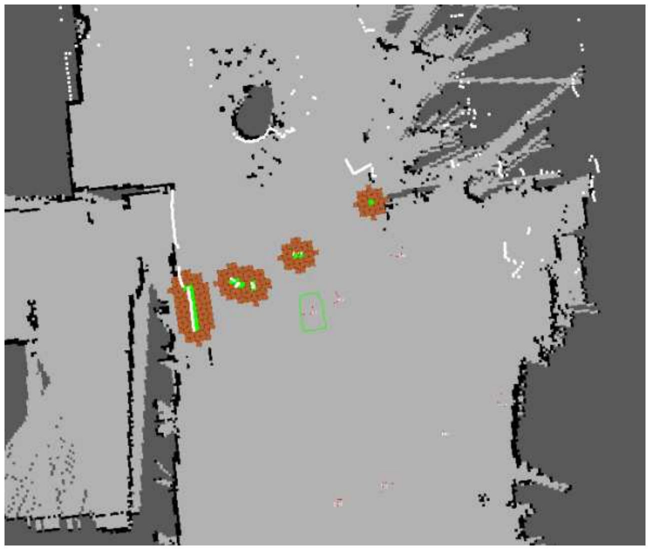


Figure 3.5: Laser Scans displayed in RViz

Power Supply

The SICK LMS200 requires a stable 24V DC power supply and consumes approximately 20W without output load. According to the technical specifications this value rises to approximately 1,8A (hence 43,2W) with output load.

3.1.5 Kinect

The Microsoft Kinect is a device based on computer vision technology from PrimeSense and designed as a controllerless user interface for the Microsoft Xbox 360. It consists of

- A standard 640x480 RGB CMOS image sensor
- A 320x240 monochrome CMOS image sensor
- A class 1 infrared laser projector

Light Coding Mechanism

A pattern of spots (structured light [SS03]) is projected using the device's infrared laser projector. When hitting a surface, the resulting distortions in the pattern are used to reconstruct the surface's 3D structure. This mechanism is called Light Coding. The device having its own light source, it's independent of lighting conditions, which might be unfavorable to traditional stereo vision approaches, for example in dark rooms. As the projected light is within the infrared spectrum, it's invisible to the human eye.

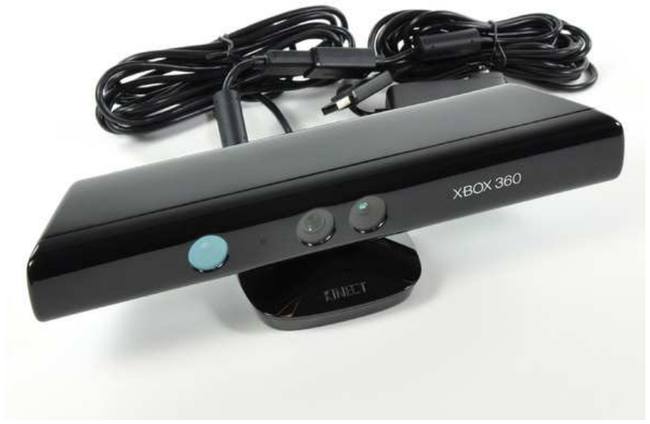


Figure 3.6: Microsoft Kinect, image courtesy of [ROS11b]

Connecting the Kinect

The Microsoft Kinect connects to a PC via USB, but needs an additional power source, as mentioned below. Open-Source drivers are available, as well as ROS packages. The ROS stack *openni_kinect* contains nodes that exploit the Kinect's functionalities, such as

- 3D depth mapping
- Human skeleton tracking (*nite* package)

The Kinect is a tool that's highly attractive for applications in robotics, being a cheap and effective way of retrieving 3D depth maps. The ROS 3D contest[4] has proven the versatility of the Kinect. One of the applications featured in this competition, namely "Improved AR Markers for Topological Navigation"[5], proved especially useful for our purposes, as it helps improving pose estimation of AR tags (as described in section 3.4.4).

Power Supply

The Kinect requires an external, stable 12V DC power supply (additionally to USB power). To connect it to our robot's power circuitry, an additional voltage

[4]http://www.ros.org/wiki/openni/Contests/ROS 3D
[5]http://www.ros.org/wiki/openni/Contests/ROS 3D/Improved AR Markers for Topological Navigation

converter was required to be on the safe side, as the voltage coming from the robot's batteries might rise up to 12,5V.

3.1.6 High-Definition Webcam

In earlier project stages, a Microsoft®LifeCam Cinema™high-definition webcam provided us with the necessary imagery for object recognition. Due to the camera's high resolution (1280x720 at a rate of 15hz)[6] [Mic09], we achieved a highly satisfying recall in marker detection, however pose estimation was less accurate than in applications using a depth-information-backed method. A lower resolution would have yielded a higher frame rate (30hz), but also significantly poorer recall, especially at high distances (approximately 4m and more). The camera is equipped with an autofocus feature, again increasing the performance of object detection, however also having the robot miss markers from time to time, due to difficulties in finding the correct focus. This was the case in narrow corridors, where the robot performed a rotation in search for an object. The camera would focus on the nearby walls, when in the next moment, facing the long corridor, the focus would have to be drastically changed.

Connecting to the Webcam

The webcam is connected via USB and communicates with ROS through the *usb_cam* package. The messages it publishes are of type *sensor_msgs/Image*. When carrying out object recognition using AR tags, the *ar_pose* package subscribes to the camera's image topic to receive data to operate on.

Mounting the Webcam

Originally, the webcam was mounted in a slanted position on the same platform as the laser measurement unit, to the left of it. The required transformation from the camera's frame (*usb_cam*) and the robot (*base_link*) can be interpreted as the combination of two separate transformations.

- The marker information we receive is associated with the camera image's frame, meaning that the X and Y axes define the image plane, while the Z coordinate points away from the camera ($\equiv$ depth, or distance). However, when interpreting the camera as a world object like the robot, the coordinate system should rather be defined as follows: the X axis pointing away from the camera, Y axis pointing to the left and Z axis pointing up.
- Additionally, the camera isn't located exactly at the robot's base link (obviously), but, as mentioned before, next to the laser scanner, in a slanted position. So we have to specify the appropriate translation as well as a rotation on the pitch axis of about -15 degrees.

[6]Diagonal field of view is 73 degrees

Combining these two transformations we came up with the following resulting transformation from the camera image's to the robot's frame:

$$translation^{7} = \begin{pmatrix} 0.085 \\ 0.105 \\ 0.30 \end{pmatrix}$$

$$rotation^{8} = \begin{pmatrix} 4.71 \\ 0 \\ 4.458 \end{pmatrix}$$

3.1.7 Controller Laptop

The Pioneer P3-DX was supplied with a Lenovo X201 portable computer, equipped with a 9-cell Lithium-Ion battery pack to withstand long periods of elevated computational load. The Lenovo X201's 12,1 inch screen allows for a compact design that fits on the robot platform (located behind the laser). There are smaller laptops (for example netbooks) available, however they don't offer the performance that's required for our purposes. The Lenovo X201 features an Intel Core i5-540M 64bit CPU (2,53 GHz, 3MB L3, 1066 FSB), 4GB of system memory (DDR3, 1067Mhz), as well as a half-terabyte hard drive operating at 7200rpm.

Operating System

A 64bit Ubuntu Maverick is installed on the controller laptop, currently hosting a *ROS diamondback* installation.

Power Supply

The controller laptop requires a 12V DC power supply for operation and recharging. The included adaptor is powered by line current. A supplemental adaptor, that connects to a 12V car power plug can optionally be used to connect the laptop to the robot's power supply, this being especially helpful once the robot's able to recharge itself autonomously.

3.2 World Model

This section refers to the ROS package *wmstorage*[9], located in the *worldmodel* stack. The central aim of this project was the implementation of a module to enable the robot to memorize quantitative and qualitative information. The world model is responsible for the following set of tasks.

1. Storage Tasks

[7]in meters
[8]Noted in RPY (roll, pitch, yaw), radians
[9]in the remainder of this document simply referred to as *world model*

- object poses (in various coordinate frames)
- attributes
- relations
- aliases

2. Reasoning Tasks
    - abstractions
    - update objects based on their relations

The world model holds a set of objects (called WMObjects) and their corresponding poses in the real world. World objects may contain attributes and relations and are addressed by one of their registered aliases (section 3.2.1). To interact with other modules, the world model listens on several topics (section 3.2.5 on page 78) for world state updates. On the other hand, it offers several services (section 3.2.6 on page 78) that allow other modules to query the current world state.

3.2.1 Aliases

Aliases are alternate names or IDs for objects in the world model. A mechanism supporting multiple names was necessary, as object information is expected to be retrieved from various sources, all using different naming systems for object identification. A vision-based object recognition mechanism might for example issue numerical IDs, while a human agent will prefer to add and retrieve objects using an intuitively-sounding object name encoded in a string, such as *"calculator"* or *"robot"*. As it's already acting as a central storage system, the world model was chosen to keep track of object aliases. These are registered whenever the world model receives a message of type WMObjectDiscovery (section 3.2.4 on page 78), holding a list of ObjectIDs (section 3.2.4 on page 75). ObjectIDs hold a tuple of strings (alias type and alias name) and are used for addressing objects in the world model. Every object in the world model must have a unique *internal ID* specified, acting as an unquestionable reference of identity. A WMObjectDiscovery message, for example, must hold exactly one ObjectID entry specifying the internal ID.

3.2.2 Attributes

Attributes are tuples of string (attribute name and value) holding any kind of information that can be associated with the object owning the attribute. Object attributes are communicated via messages of type WMObjectAttributes (section 3.2.4 on page 77), holding a list of AttributeEntries (section 3.2.4 on page 76). They can be dynamically set and cleared and are in general not compulsory. Frequently used attribute names are available as constants in the AttributeEntry message definition.

Sample applications in the current system are memorizing an object's frame identifier or the type of marker to use when visualizing the object.

3.2.3 Relations

Relations are tuples of predicate and object, the former specified in a string, extended by optional relation properties, the latter specified using an ObjectID. This corresponds to an RDF[10]-like Subject-Predicate-Object representation, where the subject is the relation's owner.

Relation properties are represented in a string, imposing no limitations whatsoever on its contents. Currently, the self-tailored *stringformat* encoding is used as a simple and straightforward way to encode poses and transformations in a string. This is implemented in Java and would have to be ported to other languages to enable other nodes to benefit from the contents of the relation properties field. A future alternative would be to integrate a wide-spread, preferably language-indepent standard for object encoding, such as XML.

A **sample application** of relations in the current system is storing information on objects being *"attached to"* other objects. This mechanism is necessary, as it enables us to have real-world objects with multiple AR tags attached, instead of directly interpreting an AR tag as a real-world object. For instance, by setting the directed *"attached to"* relation between an AR tag and a real-world object and specifying the transformation between tag and object in the relation properties, the world model's reasoning engine can update the real-world object's pose automatically when receiving information on the AR tag. Further details on this method can be found in (section 3.4.4 on page 99).

The *"attached to"* relation type is available as a constant defined in the RelationEntry message (section 3.2.4 on page 76), along with constants for relation types that are likely to be useful in the future, such as *"in"* and *"close to"*.

3.2.4 Messages

The following messages can be interpreted and generated by the world model. They are used to communicate general information about objects in the world model, as well as their identifiers. Often, messages are used to store constants that are applied across nodes. This is particularly useful when nodes are coded in different languages, as ROS takes care of translating them to all supported languages. The world model's messages contain a Header at top level. Please note that this Header contains information on the transmission of the message, not on the contents itself. Headers in subordinate message elements contain information such as timestamps, that corresponds to the elements in particular. For example if an agent receives a message at timestep 4 holding information on an object pose that was valid at timestep 1, then we're able to store both timestamps. The former will be stored in the top level Header, while the latter will accompany a nested message element.

[10]http://www.w3.org/RDF/

WMObject

World Model Objects (*WMObjects*) convey information about objects' pose and object type. The pose information is encoded in a
geometry_msgs/PoseWithCovarianceStamped
message. Including complex types in message definition files is possible, as nesting of message definitions is supported. The reasons for using the aforementioned type of message for pose information are

- allowing future implementations to include filtering methods that rely on covariance
- having information on the object's age

The Header defined in the WMObject message is only used to hold time and frame information on the transmission of the message, not the contents itself.

```
Header header

# Object ID
ObjectID objectid

# The header that can be found in pose will be used
# to transport frame_id and stamp
geometry_msgs/PoseWithCovarianceStamped pose

# only for output
string objecttype
```

Listing 3.2: WMObject.msg

ObjectID

ObjectIDs are used to reference objects in the world model. Messages conveying them must contain the object ID's type and its name, both encoded in primitive strings. Additionally, ObjectIDs contain constants for frequently-used object ID types and one constant for an object ID name, that is used to identify the agent itself. ObjectIDs will never be transmitted as single entities, but wrapped in message definitions that are designed for transmission.

```
# alias constants (subject to expansion)
string INTERNAL_ALIAS=internal
string ARTAG_ALIAS=artag
string RFIDTAG_ALIAS=rfidtag
string MARKER_ALIAS=marker
string MARKERTEXT_ALIAS=markertext
string SELF_OBJ = self

string type
string name
```

Listing 3.3: ObjectID.msg

AttributeEntry

Each attribute entry conveys exactly one attribute that belongs to a certain object in the world model. To break down timing information to entry-level each attribute entry holds a *time* field, containing a time stamp of the last change to this entry. The semantics of not having any information on a certain attribute can be represented by setting the attribute value to <*not-set*>, which is also available as a constant. This mechanism can be used for several purposes, including clearing attributes from the knowledge base. Similarly to ObjectIDs, AttributeEntries define frequently used attribute names as constants. A single AttributeEntry message will never be communicated between modules. Instead, several instances will be wrapped together in a *WMObjectAttributes* message, as described in section 3.2.4 on page 77.

```
string NOTSET=<not-set>

# standard attribute names as constants
string OWN_FRAME_ATT=ownframe
string MARKER_TYPE=markertype

string name
string value
time lastedited
```

Listing 3.4: AttributeEntry.msg

RelationEntry

RelationEntries convey information about object relations. For this purpose, relevant fields are predicate, target object, relation properties and time of last change. In the World Model, relations are defined as triples of Subject, Predicate and Object, where Predicates are additionally extended by a string holding arbitrary information on the properties of the relation. In the RelationEntry message, the subject field is missing, as the message is (similarly to AttributeEntries) not expected to be transmitted as a single entity, but in a group of RelationEntries wrapped together in a WMObjectRelations message, described in section 3.2.4 on page 77. As can be derived from the WMObjectRelations message's definition, an object that will "own" the passed RelationEntries is specified through an ObjectID. This object is by definition the relation subject.

Every RelationEntry specifies a relation target using an ObjectID. The relationproperties string contains any type of data structure encoded in a string, containing information on the properties of the relation. Please refer to section 3.2.3 for more information on Relations and the encoding of relation properties. Just like AttributeEntries, RelationEntries contain constants for frequently-used names and for the special value <*not-set*>.

```
string NOTSET=<not-set>

# standard relation names as constants
string IN=in
string CLOSETO =closeto
string ATTACHEDTO =attachedto

string predicate
string relationproperties
ObjectID object
time lastedited
```

Listing 3.5: RelationEntry.msg

WMObjectAttributes

WMObjectAttributes messages contain sets of AttributeEntries and provide these with an appropriate header and an ObjectID specifying the message target. These messages are thus compulsory wrappers around AttributeEntries.

```
Header header
ObjectID objectid

AttributeEntry[] attributes
```

Listing 3.6: WMObjectAttributes.msg

WMObjectRelations

Analogously to WMObjectAttributes messages (described above), a WMObjectRelations message contains a set of RelationEntries, wrapping them for communication between modules and adding ObjectID and Header.

```
Header header
ObjectID objectid

RelationEntry[] relations
```

Listing 3.7: WMObjectRelations.msg

WMObjectDiscovery

WMObjectDiscovery messages are used for the following tasks:

- initial registrations of objects in the world model
- adding aliases to existing objects
- changing aliases

WMObjectDiscovery messages must have an internal alias defined. Messages not meeting this condition are rejected by the world model, as the object of reference is unknown.

```
Header header
string objecttype

# Aliases - NOTE: exactly one internal alias
# (ObjectID.INTERNAL_ALIAS) has to be specified
ObjectID[] aliases
```

Listing 3.8: WMObjectDiscovery.msg

3.2.5 Topics

The World Model awaits information about changes of the world state by listening on several topics, briefly summarized in Table 3.3.

Topic Name	**Message Type**	**Purpose**
reportwmobject	WMObject	retrieve object poses
reportwmobjectdiscovery	WMObjectDiscovery	register new wmobject, . . . (3.2.4)
reportwmobjectattributes	WMObjectAttributes	update/add object attributes
reportwmobjectrelations	WMObjectRelations	update/add object relations

Table 3.3: World Model Topics

3.2.6 Services

The World Model offers numerous services to allow users to retrieve information on the current world state. All service calls, except when retrieving the complete set of objects, are object-centered. The object under investigation is referenced by passing any type of object ID[, that the world model will use to locate the corresponding object.

GetWMObjectAliases

```
ObjectID objectid
---
ObjectID[] aliases
```

Listing 3.9: GetWMObjectAliases.srv

Functionality. Returns all aliases (alternate IDs) that belong to the object identified by the passed object ID.

GetWMObjectAttributes

```
ObjectID objectid
---
WMObjectAttributes wmobjectattributes
```

Listing 3.10: GetWMObjectAttributes.srv

Functionality. Returns a list of attributes that have been associated with the object identified by the passed object ID. Please refer to section 3.2.2 on page 73 for further information on object attributes.

GetWMObjectRelations

```
ObjectID objectid
---
WMObjectRelations wmobjectrelations
```

Listing 3.11: GetWMObjectRelations.srv

Functionality. Returns a list of relations that have been associated with the object identified by the passed object ID. Please refer to section 3.2.3 on page 74 for further information on object relations.

GetWMObjectByID

```
ObjectID objectid
---
WMObject wmobject
```

Listing 3.12: GetWMObjectByID.srv

Functionality. Returns the object identified by the passed object ID. The object information provided by this service comprises object pose and object type, as well as information on when the object was last reported.

GetWMObjects

```
---
WMObject[]  wmobjects
```

Listing 3.13: GetWMObjects.srv

Functionality. Returns the complete list of objects available in the current world state, taking no input parameters. The format used is a list of *WMObjects*.

3.2.7 System Architecture

The world model architecture is designed to decorrelate the internal mechanism for object handling and reasoning from the connection to ROS. The classes WMStorage and WMLogic offering the main functionality are wrapped by the WMStorageNode class, which is in turn derived from ROSNode, a class incorporating the implementation of a ROS node in Java. The architecture is depicted in Figure 3.7.

WMStorage

The core of the world model, the WMStorage class, takes care of storing object information, such as aliases, poses, attributes and relations. It additionally holds and queries the WMLogic class (section 3.2.9), which is responsible for reasoning on the current world state. WMStorage implements the interfaces WMObjectListener and WMObjectProvider, thus enabling it to receive and be queried for world state information. The internal architecture around WMStorage is depicted in Figure 3.8.

Internal Storage of Objects

WMStorage holds the set of registered WMObjects in objects of type InternalWMObject, which are placed in a map using the internal object ID as key. When looking up objects using another object alias type, linear search has to be used to find the corresponding object, however, additional lookup maps are planned and can be easily integrated. However, considering the moderate amount of objects, they weren't necessary so far.

An InternalWMObject holds all information on the object that can exist, including its

- Poses (in various frames)

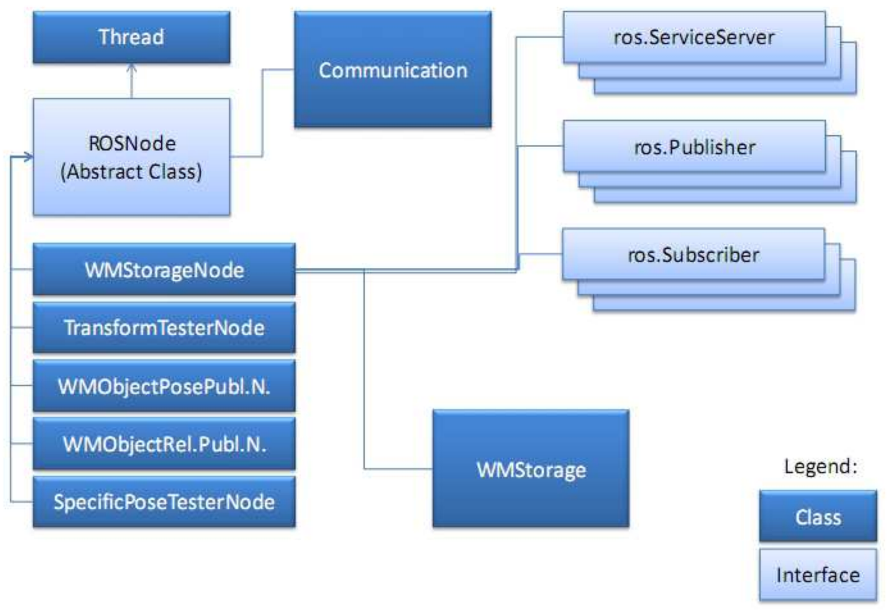


Figure 3.7: World Model: System Architecture

- Aliases
- Attributes
- Relations

MVC Model. WMStorage is designed to be part of a MVC-type architecture, playing the role of the model and for that purpose extending the Observable class. View and Controller are implemented and directly connected to WMStorage, bypassing ROS communication and thus offering faster responses from the GUI. The world model can be launched in headless mode (without a GUI extension), however, a GUI is particularly convenient when it comes to debugging, not necessarily the world model itself, but the system that's using it. Additionally, it offers ways to manually specify object poses in collaboration with RViz (section 2.5.15), as well as a mechanism to issue goals to the navigation stack, which has the robot navigate to user-specified poses.

ROSNode

ROSNode incorporates the implementation of a ROS node in Java. It's designed as an abstract class, that has to be extended to gain access to common ROS mechanisms. Implementing a new ROS node in Java thus boils down to

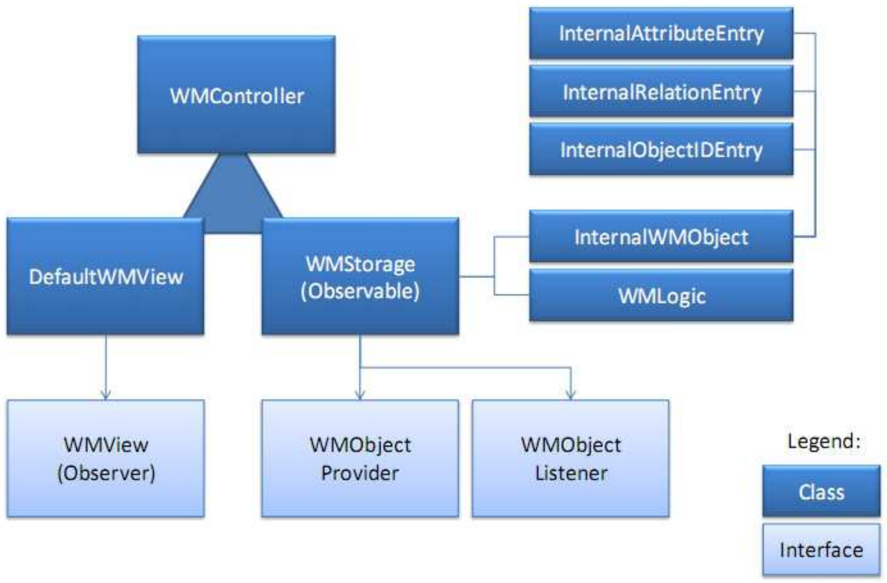


Figure 3.8: WMStorage internal design

1. Extending ROSNode
2. Implementing the *executeLoop()* method
3. Implementing the *beforeShutdownNode()* method

The *executeLoop()* method is called in every iteration of the underlying ROS node and is supposed to contain the node's core functionality. The *beforeShutdownNode()* method will be called after a shutdown has been requested[11].

WMStorageNode is an example of a class derived from ROSNode, acting as a wrapper for WMStorage. It holds all service servers and connectors to topics that might offer or query object information. Requests and object updates are then routed to WMStorage through the WMObjectListener and WMObjectProvider interfaces.

Tranformation Framework in ROSJava

As mentioned before, the world model is implemented in Java, therefore running in ROS on top of a ROSJava node. At the time of implementation, there was no support for the transformation framework in ROSJava, however, certain calculations performed in the world model require coordinate transformations. To offer

[11]due to the *ros.ok()* property changing to *false*

this functionality to the world model, the TF Adaptor node was implemented in C++. This node offers services that perform coordinate transformations, either by passing pre-fetched transforms or by querying the tf tree.

Object Information Translators

To feed the world model with object updates it can interpret, object information must arrive in a format that is comprehensible to the world model. When routing sensory input to the world model, the messages originating from various sensor nodes must first be translated to world model messages using object information translators. For each type of sensor, a module is implemented that intercepts sensory messages and republishes them in a format comprehensible to world model. This module is then placed between the sensor node and the world model. An example of such a translator is the AR Translator, converting AR marker messages to WMObjects. The AR Translator's principle is best explained in a sample use case, described in the following section.

3.2.8 Sample Use Case

The world model is initialized with a set of predefined objects and their positions, attributes and relations. All objects have an internal object ID, a position and an object type. The world model will then await additional object information on one of the topics it's subscribed to. Every object handled by the world model has to be registered using a WMObjectDiscovery (section 3.2.4) message. Object discovery messages are also used to add additional or change existing object aliases. Once an object is registered it can be addressed under any of its aliases defined.

To provide the world model with new object information, the appropriate message, one of

- WMObject (3.2.4)
- WMObjectDiscovery (3.2.4)
- WMObjectAttributes (3.2.4)
- WMObjectRelations (3.2.4)

has to be generated, filled with the necessary information and published on one of the topics the world model is listening on (Table 3.3). The world model will test for each entry whether its timestamp is newer than the latest state of its world model-internal equivalent. This procedure is broken down to entry level, e.g. for a WMObjectAttributes message holding several AttributeEntries, each element will be checked.

Example - Updating and Querying Object Poses. As stated before, the world model can only interpret its own message types, so to integrate other types of messages, we need additional modules. Let's suppose we have a visual sensor conveying object information and an execution module that will request object data from the world model from time to time. The visual sensor will, in case it's already wrapped in a ROS node, publish sensor information on a ROS topic, whenever relevant information on known objects is available (if objects are in its line of sight for example). A good example of such a sensor is described in section 3.4.4 on page 99. The example mentioned above is depicted in Figure 3.9.

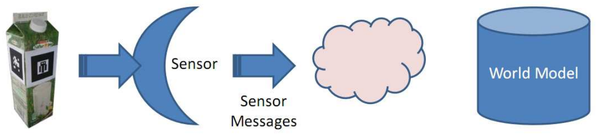


Figure 3.9: Sensor pipeline (a), no translator connected

To forward object information to the world model, a translator (as described in section 3.4.4 on page 99) has to be implemented, that will receive the sensor messages, convert them to WM objects and republish them in a format that's comprehensible to the world model. This setup is shown in Figure 3.10.

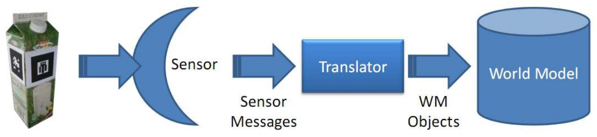


Figure 3.10: Sensor pipeline (b), translator between sensor and world model

At some point, the execution module is requested to issue commands that are necessary to move the robot to a certain spot on the map, depending on an object pose. The execution module will therefore query the GetWMObjectByID service (section 3.2.6) to receive a WMObject message (section 3.2.4) containing the requested object's pose (depicted in Figure 3.11). The world model will provide the latest pose it has received. By default, the center of the default frame (specified in the world model), without any rotation applied, will be returned if no information on the object's pose has ever been received.

The execution module can then calculate motion commands for the robot base or decide that the requested task is unfeasible.

3.2.9 WMLogic

In addition to basic storage mechanisms, the world model offers ways to translate quantitative into qualitative information and to automatically update the world

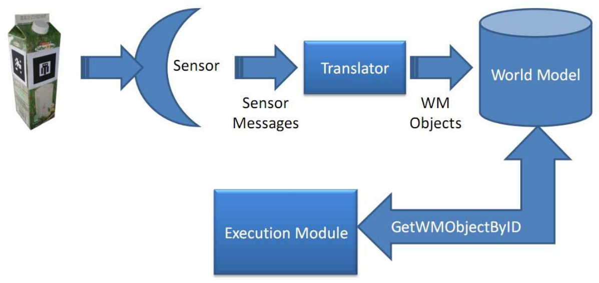


Figure 3.11: Sensor pipeline (c), querying the world model

state based on the latter. This functionality is included in the modules in and around WMLogic. This part of the system is depicted in Figure 3.12, while for further designs, please refer to section 3.2.7 starting on page 80.

Currently, the WMLogic module offers the following set of mechanisms:

- converting quantitative information into abstract statements
- updating object poses depending on object relations
- calculating specific poses depending on the world state

These mechanisms shall be explained in further detail in the following sections.

Abstractions

For the purpose of executing commands with abstract parameters, it is often necessary to translate quantitative information to qualitative statements, such as. . .

- the robot is *"in"* room A
- the box is *"close to"* a room's reference point
- the box is standing / lying on the floor

The last of the four examples mentioned above shall serve as a basis for the following scenario. Consider a robot that is supposed to pick up a box using a 2D-gripper, as described in 3.1.3 on page 65. Before the actual pick-up process, the robot has to choose an approach position somewhere close to the object, preferably facing it, so that only a straight motion is necessary before the object can be grabbed.

This position is calculated by generating a pose, that, when seen from the coordinate system of the target object, is lying on either the X or Z axis, a certain distance away from and facing the target object. Depending on whether the object is lying or not, the correct axis for the approach position has to be

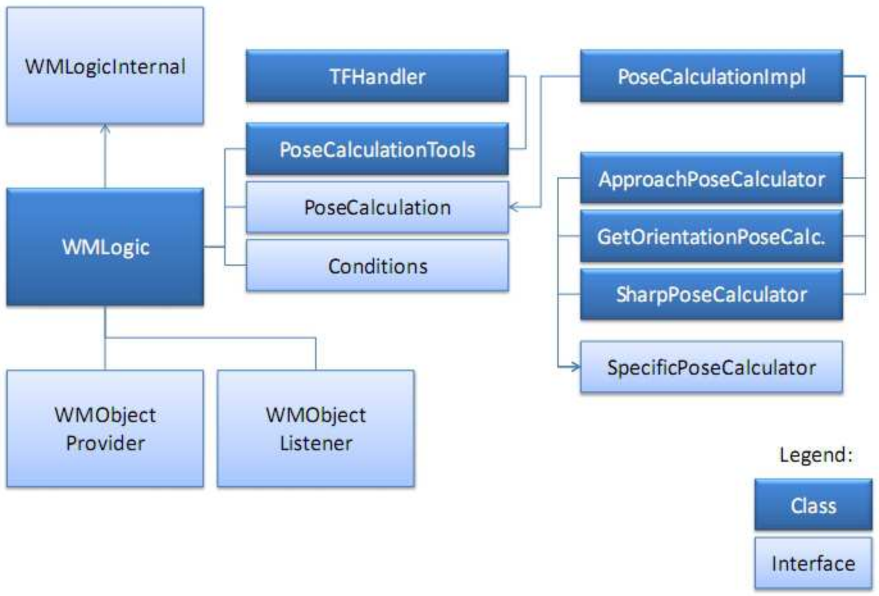


Figure 3.12: WMLogic with surrounding modules

chosen. This will be the Z axis for lying objects and the X axis for objects standing upright. The two situations are depicted in Figure 3.13 on page 87. Further information on the calculation of approach poses can be found on page 87.

Object-Relation-triggered World State Updates

Certain object relations possess underlying semantics that allow the world model to automatically draw conclusions over interdependencies of objects and therefore updating the world state accordingly. An example of such relations are those holding the *"attached to"* predicate, signaling that an object is attached to another and that pose updates of one object must yield updates of the other. This mechanism is also described in section 3.4.4 on page 99.

An important property of such relations, that must be mentioned here, is that they are directed, to avoid endless recursions due to two objects constantly trying to update each other's poses. While the resulting graph of object connections might still contain cycles, these are not checked for. A graph cycle detector might be a future endeavour, especially if relations have to be updated frequently.

Calculation of Specific Poses

Besides returning object poses, as previously received, the world model is capable to calculate additional poses that are useful to other modules. Approach poses, for example, are required for object pick-up procedures.

Approach Pose Calculation. When grabbing an object, the robot has to know where to stand before starting its gripping procedure. This module can calculate the optimum position, by taking into account the object's orientation and whether it's standing upright or lying on the ground. Figure 3.13 shows how the approach position changes depending on the object's orientation. The approach distance is variable.

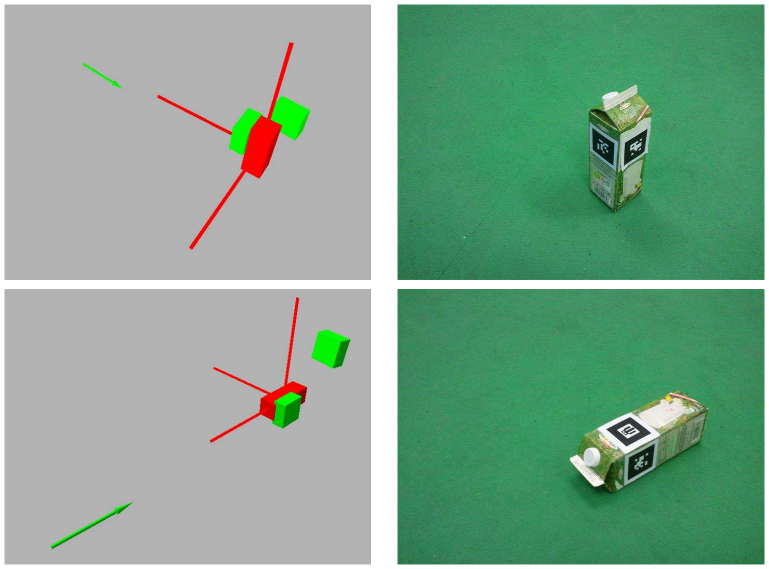


Figure 3.13: Object upright and lying

3.2.10 World Model Visualization

A module has been implemented to offer information about the world state, that is comprehensible to visualisation tools such as RViz (section 2.5.15 on page 49). World objects are translated to visualisation marker messages (*visualisation_msgs/Marker*), that can be visualized by RViz's built-in display type *Markers*. Markers are published in the default world frame and can thus be overlaid with visualisation elements coming from the navigation stack, the map server, the robot base etc. Please note that this feature comes in addition to the World Model's built-in, tabular GUI. In a marker message, the following object types are supported:

```
byte ARROW=0
byte CUBE=1
byte SPHERE=2
byte CYLINDER=3
byte LINE_STRIP=4
byte LINE_LIST=5
byte CUBE_LIST=6
byte SPHERE_LIST=7
byte POINTS=8
byte TEXT_VIEW_FACING=9
byte MESH_RESOURCE=10
byte TRIANGLE_LIST=11
```

Listing 3.14: Marker.msg snippet

Currently, arrows, cubes and labels (*text view facing*) are used to display the world state in RVIZ. A sample of how objects are depicted can be seen in Figure 3.14. Objects the robot can manipulate, in our case milk boxes, correspond to red cubes in the image. Every single object, be it the robot itself, a tangible object or the reference point of a room, is shown by displaying its coordinate system (red lines, one for each axis). Additionally, each object is provided with a label. Green arrows signal optimal approach positions that the robot will navigate to before trying to pick up an object.

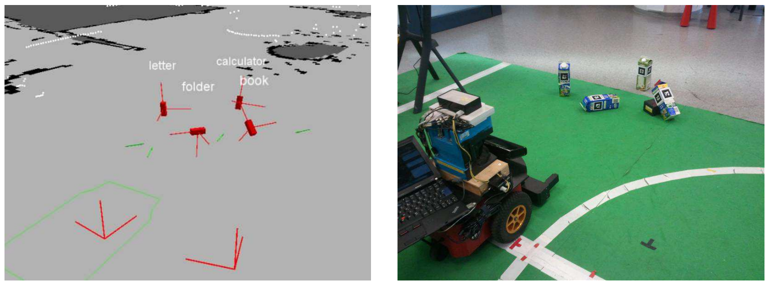


Figure 3.14: World Objects displayed in RVIZ

3.3 Execution Modules

The execution package contains self-implemented action servers, an offline robot control node (*Offline Robot Control*) designed for testing purposes and rapid prototyping, a node that waits for and handles incoming connections from IndiGolog clients (*IndiGolog Connection*) and *smach* state machines, again for testing as well as early demonstration of lowlevel capabilities.

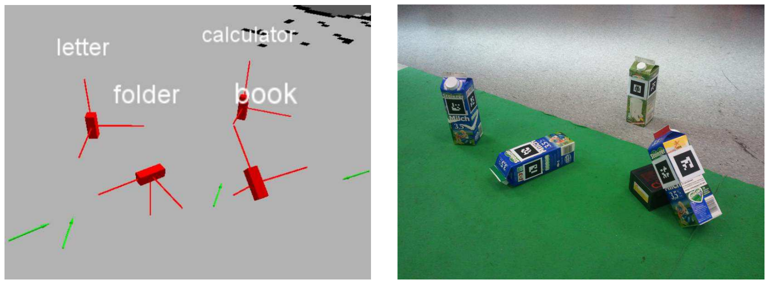


Figure 3.15: World Objects displayed in RVIZ - Closer Look

3.3.1 Action Servers

Action Servers await action goals and subsequently perform the requested actions, constantly offering feedback and a final state after finishing execution. In the current implementation of the execution package, two self-tailored action servers are in use, namely

- Positioning Action Server
- Gripobject Action Server

These shall be described in detail here.

Positioning Action Server

The positioning action server's task is to have the robot perform simple translational and rotational movements. The navigation stack is cut out of this process, movements are thus unchecked against collisions. This means that robot movement is exactly predictable and purely under "our" control, which is convenient, but on the other hand has to be handled with care, as collisions become possible without the navigation stack watching every step.

The commands issued by the Positioning Action server are by default published on the */cmd_vel* topic. The format for actions is described in listing 3.15.

```
#goal definition
int32 NO_CONDITION = 0
int32 TILL_INNER_BREAKBEAM_BROKEN = 1
int32 TILL_OUTER_BREAKBEAM_BROKEN = 2
int32 TILL_INNER_PADDLES_TRIGGERED = 3

int32 cancel_condition
float64 move_straight
float64 yaw
```

```
---
#result definition
int32 NO_CONDITION = 0
int32 TILL_INNER_BREAKBEAM_BROKEN = 1
int32 TILL_OUTER_BREAKBEAM_BROKEN = 2
int32 TILL_INNER_PADDLES_TRIGGERED = 3

int32 cancel_condition
float64 moved_straight
float64 yaw
---
#feedback
float64 moved_straight
float64 yaw
```

Listing 3.15: Positioning.action

As can be seen in the code sample, two types of movement are supported, straight movement and yaw, the former of which is specified in meters, while the latter is given in radians.

Action timing. To make sure that requests can be processed correctly, the commands for starting a movement and for stopping it have to be timed correctly. Thus the action server needs to know the robot's maximum translational and angular speed. We could just define a time parameter for translational and rotational commands, however, in most cases it is much more useful to be able to define distances and angles. The action server issues motion commands at the robot's maximum speed and measures the time consumed at a rate of 10hz. The time necessary for the movement is calculated by the simple formulae:

$$execution\_time = \frac{distance}{maximum\_translational\_speed}$$

$$execution\_time = \frac{angle}{maximum\_rotational\_speed}$$

The iteration rate is an important factor here. We want to find a good trade-off to achieve sufficient precision, while keeping system load to a minimum. A rate of 10hz has proven to convey action results of adequate quality, without an unreasonable waste of system resources.

Cancel conditions. In several cases, an execution has to be stopped before the initially requested goal is reached. This preemption of execution is often foreseen or even expected, rather than the result of erroneous execution. An example would be a robot receiving the task of driving up to a certain object *until* it senses that the object is in its gripper's range. Cancel conditons are responsible for the *"until"* part of this request.

The Positioning Action Server allows the user to define cancel conditions that depend on the state of the robot's gripper. To be able to offer this feature, the server must be subscribed to the *gripperinfo* topic that is published by the *ROSARIA* package (described earlier on in section 3.1.1 on page 63), thus receiving gripper information directly from the node connected to the robot platform. The aforementioned topic transports messages of type *Gripper*, specified in detail in section 3.1.3. The Positioning Action Server accepts changes of both gripper paddle and gripper breakbeam[12] states as cancel conditions. These states are checked and updated in every iteration. Once a cancel condition is met, the current execution is stopped. The available options for cancel conditions are specified as constants in the action message.

As mentioned before, the action server publishes execution feedback, as well as the final result of the requested action. As for processes wrapping the call to the action server (e.g. a state machine) the information whether a cancel condition has been met, might be relevant, this information is included in the result message.

Simultaneous Requests. If multiple requests arrive in the same action goal (both rotational and translational request), they are both executed **simultaneously**. Please note that in this case there is no guarantee for exact positioning, as the movements are interdependent and no sophisticated trajectory planning is used in addition to the simple equations mentioned above.

Gripobject Action Server

The Gripobject Action Server awaits GripObject Goal messages as specified in listing 3.16 and subsequently addresses the robot's gripper by publishing commands (by default) on the */grip_cmd* topic. The *RosAriaWithGripper* node located in the *ROSARIA* package is subscribed to this topic (conveying primitive integer data). The gripper's actual control capabilities are limited to:

- moving lift up
- moving lift down
- opening gripper
- closing gripper
- stopping lift
- stopping gripper

```
#goal definition
int32 NO_CONDITION = 0
int32 TILL_INNER_BREAKBEAM_BROKEN = 1
```

[12]breakbeam ≡ light barrier

```
int32 TILL_OUTER_BREAKBEAM_BROKEN = 2
int32 TILL_INNER_PADDLES_TRIGGERED = 3
int32 TILL_LIFT_MAXED = 4
int32 TILL_GRIPPER_OPEN = 5
int32 TILL_GRIPPER_CLOSED = 6
int32 TILL_GRIPPER_INBETWEEN = 7

int32 required_condition
float64 rel_width
float64 rel_height
---
#result definition
ROSARIA/Gripper gripper_state
float64 rel_width
float64 rel_height
---
#feedback
float64 rel_width
float64 rel_height
```

Listing 3.16: Gripobject.action

Each command yields a permanent movement that will continue even after lift or gripper paddles are maxed out. Therefore gripper control includes initiating a motion and stopping it after a specified amount of time, very similarly to positioning (section 3.3.1). As shown in the Gripobject Action message definition (listing 3.16), gripper paddle width and lift height are specified in *float64* entries. These entries are to be interpreted as relative values, making it unnecessary to keep track of the gripper's state. A value of 1.0 stands for a motion that corresponds to the actuator's maximum range, while -1.0 stands for the same motion in the opposite direction and 0.0 for no motion.

Action timing. Similarly to the solution applied to the Positioning Action Server, the Gripobject Action Server uses a simple equation to calculate the time necessary for an execution.

$$execution\_time = relative\_distance \cdot maximum\_travel\_time$$

In this case *maximum_travel_time* is a constant that's roughly estimated (separately for lift and paddles). If, for example, for traveling from the lowest to the highest position, the lift would consume 5 seconds, a value of 0.5 will yield a motion time of 2.5 seconds, no matter what the starting state is. Of course, as with the Positioning Action Server, the choice of an adequate iteration rate is a relevant issue here.

Required Condition. The concept of a required condition is very similar to that of a cancel condition, as used in the Positioning Action Server. The difference is that a required condition additionally affects the action's result state. It is thus possible to signal that an action has failed if a required condition hasn't been met.

Simultaneous Requests. If requests for both the gripper's lift and paddles arrive in the same action goal, they are executed sequentially.

3.3.2 Simple State Machine

A simple *smach* (section 2.5.16 on page 51) state machine has been created to test execution procedures in the early stages of development. Smach was an attractive option, as it integrates seamlessly with ROS Action Servers, and is easily modifiable. In addition to that, it comes with a visualization tool that allows for intuitive monitoring of machine states.

If in any machine state an error occured, the robot would stop execution. There are other ways to deal with errors, basically any state transition would have been possible, depending on the action result, but this level of error handling was sufficient for testing purposes, especially considering the fact that a more sophisticated control system (*IndiGolog*, described below) was to be applied.

3.3.3 Indigolog Integration

IndiGolog acts as a high-level execution and belief management framework that integrates with the robot's low-level capabilities implemented on top of ROS. To enable a connection between IndiGolog and ROS, a node called IndiGolog Connection has been implemented. This node communicates with an IndiGolog program via a TCP/IP socket. It waits for an incoming command, executes it and notifies IndiGolog when the task is fulfilled. In certain predefined cases, sensing information is sent to IndiGolog.

The robot offers execution and sensing capabilities to the IndiGolog program, as described in the following two sections.

Execution

- **Object Pickup:** A set of subtasks enabling the robot to pick tagged objects up using its 2D gripper device. The pickup procedure comprises exact navigation to the object's approach position, subsequently approaching the object and gripping it.
- **Object Putdown:** Includes navigating to a spot facing the desired putdown location, moving forward, lowering and relasing the object and finally moving back.
- **Robust Navigation:** To any point in the specified map.

Sensing

- **Recognition of Tagged Objects:** Objects supplied with AR tags are recognized and stored in the world model. This information serves as a basis for certain sensing information.
- **Sensor Information from Gripper Device:** Information on the states of light barriers and gripper paddles.

History-Based Diagnosis

The underlying IndiGolog program is capable of detecting conflicts between its internal belief and the actual world state. In these cases, the program does not try to find out what the problem is, but rather what happened [WGRS11]. Hypotheses that might explain the current situation are generated and saved as alternative histories. In further execution steps, those alternatives that prove to be incorrect are discarded.

3.4 System Configuration

Mechanisms that were presented in the theoretical part of the document and are available as libraries in ROS, such as localization, navigation and object recognition, were configured to be applicable to our delivery robot.

3.4.1 Mapping

The robot's belief of the environment is stored in an occupancy grid, that was created using the *gmapping* package. During the mapping procedure, a tele-operation node was applied to guide the robot manually. As RViz is capable of displaying the map while it's being created, we had a way to supervise the emerging map's quality. The SICK laser measurement system in combination with the *sicktoolbox* package served as a source of laser scan data for the mapping process.

3.4.2 Localization

Just like for mapping, laser scans were used as the only information source for localizing the robot. The ROS package *amcl* was responsible for matching laser scans to map data, thus deriving the robot's position relative to the map and publishing transformation data between the map and the robot's odometry frame. While the localization process applied in *amcl* is highly dependable in "normal" operation, it has difficulties recovering from exceptional situations. The underlying problems are well-known in this research area and discussed in [TFBD01].

Global localization problem

With a relatively low amount of salient features, or even worse, with multiple similar areas in a map, the robot had difficulties retrieving an initial pose estimate. In general, however, the initial localization task was fulfilled successfully, sometimes even with a very inaccurate initial estimate (for example with the robot being located in a completely different room).

Kidnapped robot problem

The kidnapped robot problem was experienced mostly when the robot's wheels would stall and cause an unexpected and unnoticed rotational movement. Especially in a sparse office environment, turns of 90, 180 and 270 degrees can yield sensor input similar to previous readings. This of course is fatal, as the robot believes that it's still well-localized and facing in the same direction as before. Manual re-localization is often the only way to recover from this situation, without additional sensory input.

In summary, location tracking tasks are performed robustly at all times, while recovery from the two aforementioned situations is dependent on the robot's location and environment, and the type of error.

3.4.3 Navigation

ROS offers a complete stack dedicated to navigation, however it's still a tedious task to get all necessary modules to work on a particular robot. This section describes how the Pioneer P3-DX was enabled to safely navigate between spots in a mapped office environment, using a SICK laser scanner and a Microsoft Kinect as sensors.

Costmap Configuration

The costmap defines traversible and occupied spots in the robot's environment. In navigation tasks, the local planner will query the costmap to find out about obstacles, in order to avoid collisions. It is vital to have accurate and up-to-date information here, a requirement yielding several difficulties:

1. **Obstacles added too late:** the robot hits an obstacle before it shows up on the costmap
2. **Obstacles missed by the robot's sensors:** due to being outside of the sensor's field of view
3. **Sensors clearing the costmap by mistake:** sensors may signal that cells are free even though they are outside of the sensor's field of view
4. **Overloaded costmaps:** if too many obstacles are kept in the costmap (due to errors or misconfiguration), the robot won't be able to calculate a path to the goal

5. **Incorrect robot footprint:** a footprint that exceeds the robot's size will make it impossible to navigate through tight passages (such as doors), while a too small footprint will lead to collisions
6. **Sensor miscalibration:** leads to confusing results, as the real world and the robot's internal view of it aren't aligned any longer
7. **Erroneous odometry:** stalled wheels or collisions might lead to errors in the robot's odometry, causing it to be misaligned with the costmap

A visualized costmap is depicted in Figure 3.16. Sensory data in this example was retrieved from a laser range measurement system and a Microsoft Kinect.

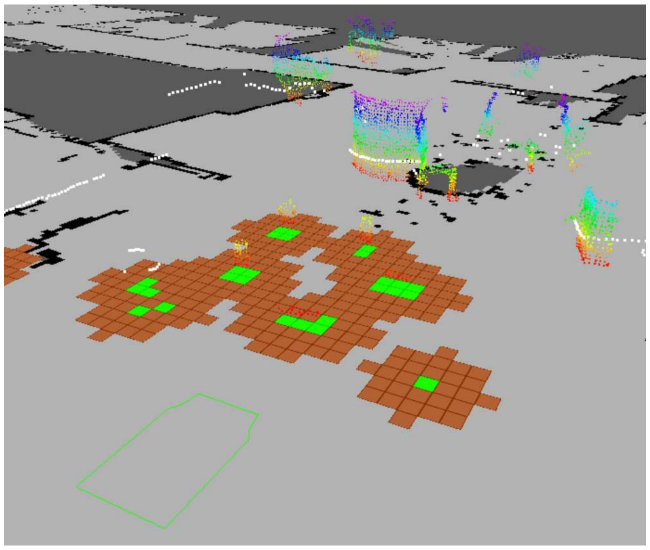


Figure 3.16: Costmap visualized in RViz. Obstacles are signaled by green grid cells. The red cells around the obstacles mark their inflation radii.

Several parameters have to be set before a costmap can be applied effectively, some of which are explained in the following paragraphs.

Robot footprint. To be able to avoid obstacles, the robot has to be aware of its own perimeter. In a 2D application, the robot's body is simplified to a 2D polygon, called the robot footprint, which is defined as a set of points that are used to calculate a convex hull. Alternatively, the robot's radius can be defined, which is useful for robots with a circular footprint. The following code shows how the footprint of our robot (including a gripper and additional space for the controller laptop) is defined:

```
footprint: [[0.21, 0.21],
            [0.27, 0.175],
          [0.37, 0.155],
          [0.37, -0.155],
          [0.27, -0.175],
          [0.21, -0.21],
          [-0.31, -0.21],
          [-0.31, 0.21]]
```

Ranges. As not all sensor readings are relevant for the obstacle avoidance task, the costmap allows to define a maximum object distance, the maximum length of rays (for clearing the costmap), as well as a minimum and maximum height for obstacles:

```
obstacle_range: 2.5
raytrace_range: 3.0
min_obstacle_height: 0.07
max_obstacle_height: 0.75
```

Map type. It's possible to choose between a 3D octomap and a 2D costmap. For our purposes, a 2D representation was sufficient, due to the self-contained shape of our robot, the fact that it's not airborne and that it does not have any moving parts that can reach beyond its predefined perimeters (e.g. robotic arms).

```
map_type: costmap
```

Sensors. The costmap can retrieve input from two different types of sensors, point clouds and laser scans. Obviously, the applied input types have to be specified in the costmap configuration. In our application, we used both point cloud data from a Microsoft Kinect and laser scan data from the SICK laser measurement system. Before integrating the point cloud data into the costmap, it had to be resampled to a lower resolution and filtered to exclude points that are close to ground level (to avoid adding the floor in front of the robot as an obstacle). Hence the point cloud topic the costmap is subscribed to is not the data coming directly from the Kinect, but rather the filtered data that has its source in a voxel grid filter (described in section 2.5.12 on page 41). It's important to note that it's possible to define which costmap tasks certain sensor data may trigger. We achieved the best results by allowing both marking and clearing operations to both point cloud and laser scans.

```
observation_sources: kinect_point_cloud
                     laser_scan_sensor
kinect_point_cloud: {sensor_frame: base_link,
                     data_type: PointCloud2,
                     topic: /voxel_grid/output,
                     marking: true,
                     clearing: true}
```

```
laser_scan_sensor:  {sensor_frame: /laser,
                     data_type: LaserScan,
                     topic: scan,
                     marking: true,
                     clearing: true}
```

The code snippets above can be found in the *costmap_ common_params_ kinect.yaml* configuration file.

Local Planner

For local path planning and obstacle avoidance we applied the trajectory rollout method. The ROS navigation stack offers the dynamic window approach as an alternative that saves computational cost, however trajectory rollout yielded more promising results.

```
dwa: false
```

The following parameters specify the robot's characteristics, as well as weights of the local planner's objective function[13].

```
  # robot characteristics and constraints
  max_vel_x: 0.50
  min_vel_x: 0.10
  max_rotational_vel: 1.0
  min_in_place_rotational_vel: 0.1
  acc_lim_th: 2.00
  acc_lim_x: 1.50
  acc_lim_y: 1.50
  holonomic_robot: false

  # weights of objective function
  # default value 0.01 used for occdist_scale
  goal_distance_bias: 0.8
  path_distance_bias: 0.5
```

Several other parameters can be defined to alter path simulation characteristics, goal tolerance etc. For a full list of parameters please refer to the *base_ local_ planner* package's documentation.

The code snippets above can be found in the *costmap_ common_ params_ kinect.yaml* configuration file.

Accessing the Navigation Stack

The navigation stack offers action servers for accessing functionalities such as navigation to a goal, costmap clearance, canceling navigation tasks or calculating a global path without moving the robot. When action feedback is not

[13]http://www.ros.org/wiki/base_local_planner

required, navigation goals can also be published to the *move_base_simple/goal* that contains messages of type *PoseStamped*.

3.4.4 Object Recognition

To let the robot interact with its environment, it must be able to recognize manipulable objects in the world. In our case, the set of manipulable and recognizable objects is limited to those that have been tagged with at least one AR marker. The *ar_pose* or *ar_kinect* package can be used to detect these tags, estimate their poses and publish this information on a ROS topic. They both rely on the functionality of the *artoolkit* package.

AR Pose Package

The *ar_pose* package requires a device that publishes image information on a ROS topic, such as a USB camera in combination with the appropriate ROS node. In early stages of the project, a Microsoft®LifeCam Cinema™(described in section 3.1.6 on page 71 and operated using the *usb_cam* node) was applied. AR recognition using this camera already yielded good performance, meaning that the robot was able to locate and pick up objects based on the pose information coming from *ar_pose*.

AR Recognition with Kinect

In subsequent stages of the project, a Microsoft Kinect was mounted on the robot, providing us with point cloud information of the environment. Besides navigational purposes, the 3D view of the world was used (in combination with the *ar_kinect* package) to yield more precise object pose estimation results. The mechanism hereof is described in section 2.7.2 on page 60.

Tagging World Objects

In the final configuration, four world objects were used for our experiments. We opted for empty milkboxes, as they can be easily gripped by the gripping device, are light and offer enough flat surfaces for AR tags. To allow for object recognition in any situation, all four sides of an empty milkbox (standing upright) were tagged with different AR tags, as can be seen in Figure 3.17.

Instead of having one tag correspond to one certain object in the world model, they were saved as separate objects of type *artag*. Every *artag* that was applied to a milkbox, had an *attachedto*-relation and the appropriate transformation specified. Figure 3.18 depicts the relation between AR tags and real world objects.

We had to define four distinct transformations, for tags attached to each of the milkbox's four sides. While retrieving the translational transformation is straightforward, the more complicated rotational transformation is shown in the following paragraphs, for each side respectively. The rotations are written as Quaternions $(x, y, z, w)^T$.

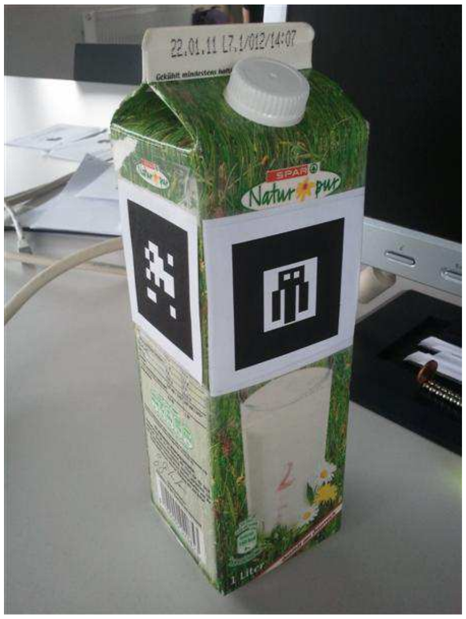


Figure 3.17: Milkbox with AR Tags

Front Tag Rotation.

$$rotation = \begin{pmatrix} -0.5 \\ -0.5 \\ -0.5 \\ 0.5 \end{pmatrix}$$

Left Tag Rotation.

$$rotation = \begin{pmatrix} 0.0 \\ 0.707106781 \\ 0.707106781 \\ 0.0 \end{pmatrix}$$

Right Tag Rotation.

$$rotation = \begin{pmatrix} -0.707106781 \\ 0.0 \\ 0.0 \\ 0.707106781 \end{pmatrix}$$

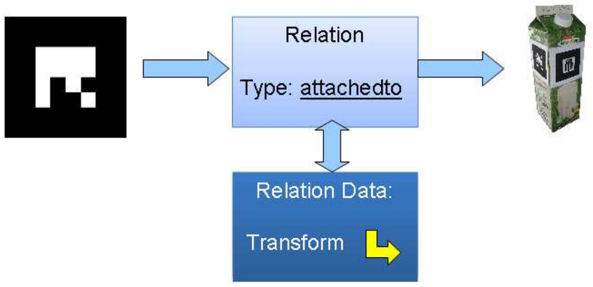


Figure 3.18: The *attachedto* relation type

Rear Tag Rotation.

$$rotation = \begin{pmatrix} -0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{pmatrix}$$

The relation target of an *attachedto* relation is automatically updated together with the relation subject, using the defined transformation between the two. Thus, if any of the four markers attached to a real world object was detected, the object itself was updated as well. Figure 3.19 shows the result, as seen in RViz. The robot's footprint can be seen in the top left corner, while the object to be recognized is located in the lower right sector of the image. Two of the object's four AR tags have been recognized and are depicted as green boxes.

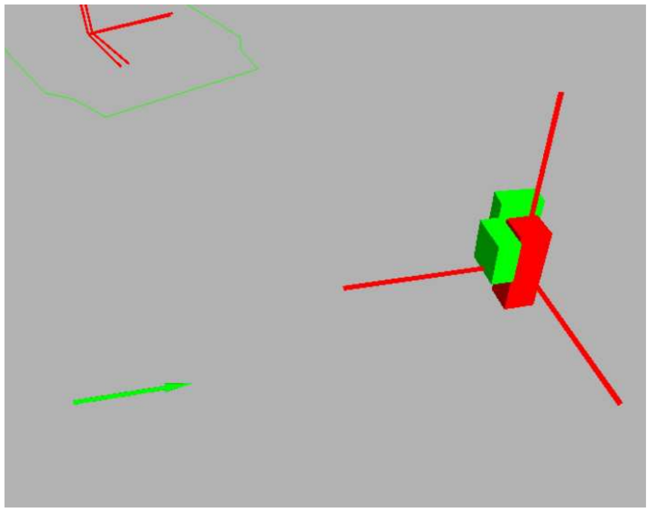


Figure 3.19: RViz View: Multiple tags attached to the same object

3.5 Experiments

The hardware described in section 3.1, in combination with the aforementioned world model and execution modules, was used to carry out two types of experiments (both originally described in [GRSW11]) demonstrating the system's capabilities and overall reliability. The experiments took place in an indoor office environment, under everyday conditions (e.g. with chairs being moved and people walking around). Reference points in rooms were defined to serve as a quantitative abstraction, to be able to translate commands like "go to room C" to actual navigation commands with real world coordinates. These reference coordinates were previously defined in the world model, where they are handled as regular world model objects of type "room". The robot used in the experiments is able to

- grasp objects that fit in its gripper (in our case empty milkboxes, tagged with AR tags as described in section 3.4.4)
- release objects
- carry out navigation tasks

Its belief management module can handle several inconsistencies, namely

- execution failures
- sensing failures
- exogenous events
- incomplete / ambiguous knowledge

3.5.1 Belief Repair Demonstration

This experiment is of relatively short duration (approximately 12 minutes) and is designed as a demonstration of the robot's belief management capabilites. A map of the office environment, with markers pointing to room coordinates can be seen in Figure 3.20 on page 103. In this setting, 4 rooms are defined (kitchen, goal, office and seminarroom), along with 2 objects that can be manipulated by the robot (calculator and letter). The robot has to fulfill 2 delivery tasks:

1. deliver object *calculator* to room *goal*
2. deliver object *letter* to room *office*

Initially, it's supplied with ambiguous information on the *calculator's* position, by being told that its location is either *seminarroom* or *office*. It's unaware of *letter's* position, however, with its starting position located close to *letter*, this object will be seen shortly after starting the experiment and thus added to the knowledge base.

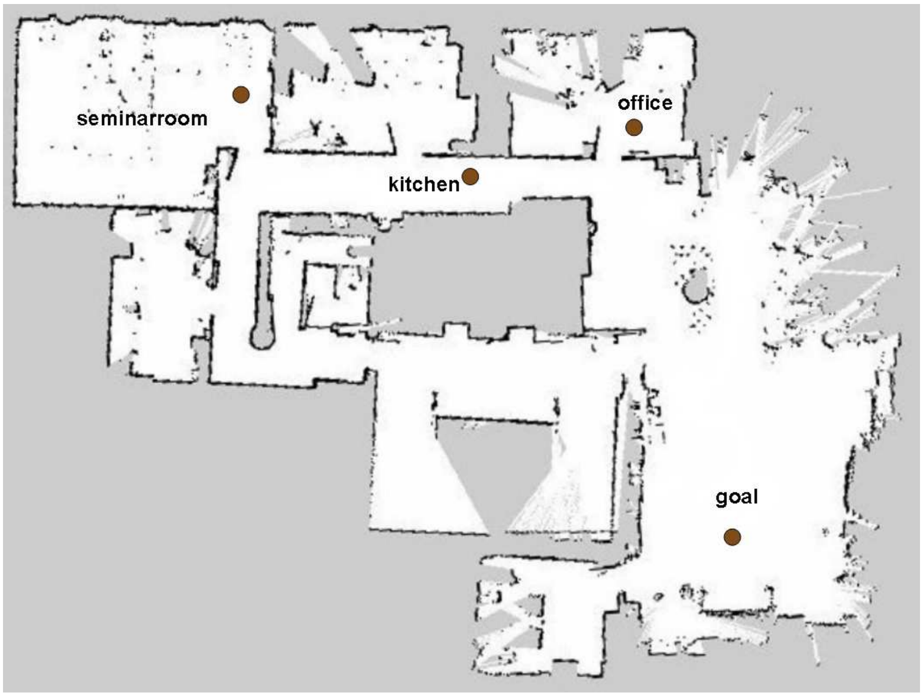


Figure 3.20: Office Environment hosting Experiments

Course of Events

The experiment conducted consisted of the following sequence of events. First of all, let's have a look at the robot's starting knowledge:

Object	**Assumed Location**	**Actual Location**
Robot	Kitchen	Kitchen
Calculator	{Seminarrom; Office}	Office
Letter	<*unknown*>	Kitchen

Looking for Object *Calculator*. Due to ambiguous information on *calculator's* location, the robot has to choose one of the rooms (*seminarroom* or *office*) to investigate first, in search for *calculator*. In our example this is *seminarroom*, to which the robot navigates.

Object	**Assumed Location**	**Actual Location**
Robot	Seminarroom	Seminarroom
Calculator	{Seminarrom; Office}	Office
Letter	Kitchen	Kitchen

***Calculator* is not located in *Seminarroom*.** As the robot's AR recognition module doesn't convey any fresh information on *calculator's* location, the robot believes that it is not located in the current room. Therefore the previously chosen theory is discarded, giving way to the next-best theory, stating that *calculator* is located in *office*. This demonstrates how ambiguous or incomplete information is coped with. The robot navigates from *seminarroom* to *office* to look for the requested object.

Object	**Assumed Location**	**Actual Location**
Robot	Seminarroom	Traveling to Office
Calculator	Office	Office
Letter	Kitchen	Kitchen

An Exogenous Event. While traveling from *seminarroom* to *office*, *letter* is moved from its previous location to *seminarroom*. The robot is not aware of this exogenous event, as it takes place outside of its field of vision.

Object	**Assumed Location**	**Actual Location**
Robot	Office	Office
Calculator	Office	Office
Letter	Kitchen	Seminarroom

Picking Up *Calculator*. After arriving in *office*, the robot performs what we like to call the pickup waltz, which is in fact a series of rotations and pose corrections, followed by a pickup action. Subsequently, *calculator* is transported to its designated destination, which in this case is *goal*.

Object	**Assumed Location**	**Actual Location**
Robot	Goal	Goal
Calculator	Goal	Goal
Letter	Kitchen	Seminarroom

Putdown Failure. Having arrived at *goal*, the robot attempts to put down the object it is holding. To demonstrate the handling of execution failures, the gripper is manually blocked, thus keeping the robot from releasing the object. Resulting from this circumstance, the gripper sensors signal that the robot is still holding something, while at the same time the AR recogntion module doesn't provide any fresh information on the object's location. Both observations suggest that there has been a failure during the putdown procedure, so the robot attempts another putdown, succeeding this time.

Object	**Assumed Location**	**Actual Location**
Robot	Goal	Goal
Calculator	Goal	Goal
Letter	Kitchen	Seminarroom

Navigate to *Letter*. By successfully delivering *calculator* to goal, the robot has successfully carried out the first task and may thus move on to task 2. To start with, it must travel to *kitchen*, where it has last seen *letter*, to pick it up.

Object	**Assumed Location**	**Actual Location**
Robot	Kitchen	Kitchen
Calculator	Goal	Goal
Letter	Kitchen	Seminarroom

***Letter* Has Been Moved.** As mentioned before, *letter* isn't located at *kitchen* any longer, as it has previously been moved to *seminarroom*, without the robot noticing. When the robot arrives at *kitchen* and (to its astonishment) doesn't stumble across *letter*, it assumes an exogenous event. To resolve the inconsistency in its belief, it starts inserting exogenous events, one hypothesis for each room available.

Object	**Assumed Location**	**Actual Location**
Robot	Kitchen	Kitchen
Calculator	Goal	Goal
Letter	{Goal; Seminarroom; Office}	Seminarroom

Looking for Object *Letter*. Again, the robot has to deal with ambiguous information, as in the previous step multiple hypotheses for *letter's* location were added to the knowledge base. In search for *letter*, it will first travel to *goal*, then to *seminarraum*, where it will finally find the object and pick it up.

Object	**Assumed Location**	**Actual Location**
Robot	Seminarroom	Seminarroom
Calculator	Goal	Goal
Letter	Seminarroom	Seminarroom

Delivery to *Office*. The only task left is to deliver *letter* to *office*. To this end, the robot navigates to *office* and attempts a putdown. It succeeds, however, a sensor failure is artificially introduced, by breaking the gripper's light barrier after it has released the object. The robot realizes that object information coming from the AR recognition module is now in conflict with the gripper's sensory data, as the object can be seen standing upright on the floor while the gripper still seems to be holding something. After approximately 80 seconds of calculation time the inconsistency is resolved by assuming a sensor failure. The robot can thus return to its startup position, as both tasks have been successfully accomplished.

Object	**Assumed Location**	**Actual Location**
Robot	Kitchen	Kitchen
Calculator	Goal	Goal
Letter	Office	Office

3.5.2 Longterm Experiment

To demonstrate the system's reliability, an experiment was designed to let the robot perform a (theoretically) endless set of delivery tasks[14]. To this end, the following scenario was created:

- 5 reference points corresponding to rooms
- 4 manipulable objects
- the perpetual task of taking the next object and moving it to the free room

The experiment was carried out in a large open space belonging to an indoor office environment, under everyday conditions. Figure 3.21a shows a map of the experiment, while Figure 3.21b conveys a real world view of the office.

A total of three longterm experiments has been carried out, **two of which lasted 8 hours and one of which even made it to 12 hours**. Statistics collected during the test runs are listed in Table 3.4.

No exogenous events or other artificially caused failures were introduced during these runs.

[14]http://en.wikipedia.org/wiki/Sisyphus

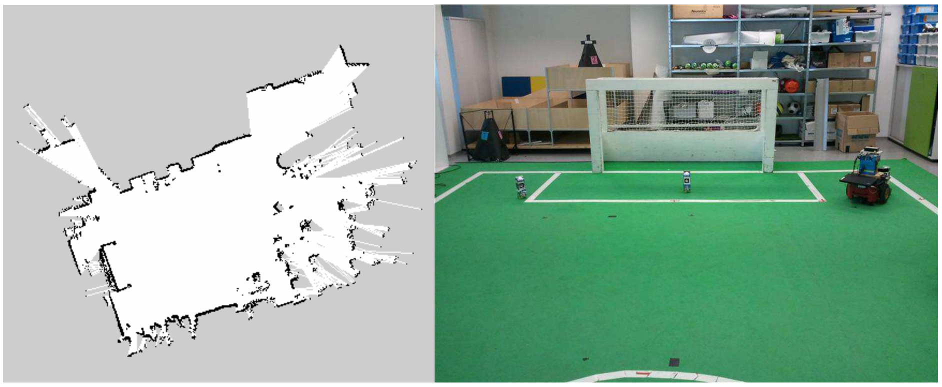

(a) Map (b) Environment

Figure 3.21: A photo and a map of the environment that hosted the longterm experiment. In the background the robot can be seen while picking up one of the milkboxes. The photo has been taken approximately from the center of the map, facing to the left.

Property	**Value**	**Comment**
Average Belief Repairs	5.5	Mostly due to pickup failures
Total Executed Actions	627	
Total Successful Tasks	147	Tasks are assumed to be successful if the situation after executing them is consistent

Table 3.4: Statistics collected during longterm experiment, as described in [GRSW11]

Battery Time

During long runs, battery time obviously becomes an issue. Based on data collected during our experiments, we can state that the P3-DX, in the configuration described in this document runs approximately $3\frac{1}{2}$ hours with a set of three 12V lead-acid batteries, before signaling low battery state. As our robot didn't offer the feature of charging itself at a battery-charging station at the time when the experiments were conducted, the laptop controlling the robot was left running on its built-in battery instead of connecting it to the robot's power supply. The laptop's 9-cell Lithium-Ion battery usually lasted $1\frac{1}{2}$ hours before reaching a charge state we considered critical[15]. Every time the IndiGolog Connection node received a *move* command, it would check the battery state before forwarding the command to the navigation stack. This being a very frequent command, we could state that the battery state was constantly being monitored. In case

[15]An explanation for this relatively short runtime is high computational effort

the battery state fell below a certain threshold[16], the IndiGolog Connection would hold execution and start issuing audible signals. A human agent would then have to manually hotswap the laptop battery and confirm completion of the battery change process by pressing *Enter*. This hotswapping mechanism was necessary in order to keep the system up and running, thus avoiding loss of its internal state.

As the robot's batteries lasted more than twice as long as the laptop's, changing them on every second laptop battery change seemed to be a straightforward solution.

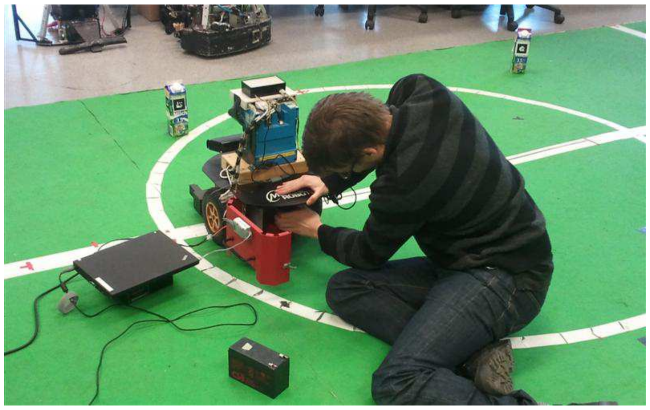


Figure 3.22: Hotswapping the robot's batteries

Autonomous Recharging. A future option is to enable the robot to autonomously recharge itself, by supplying it with a current collector and connecting the controller laptop to the robot's power supply. Based on this setup, the robot's battery state has to be monitored instead of the laptop's. In combination with the *ROSARIA* package, the robot provides information on the battery voltage, but does not estimate the relative charge state out-of-the-box. An experiment has shown that a voltage of 11,5V can be considered low. This value could thus be used as a threshold for holding execution and navigating to the docking station.

[16]During this experiment this was set to 20% battery charge

Chapter 4

Conclusions & Future Work

The aims of this project were to build up know-how in various subtopics of the robotics domain and to configure an autonomous mobile delivery robot that can serve as a basis for experiments. To enable the robot to memorize objects and their relations in the real world, a world model had to be implemented and integrated with the rest of the underlying ROS ecosystem. Several libraries for tasks such as localization, vision, mapping, navigation, teleoperation, point cloud calculations and coordinate transformations had to be combined and configured to form a complete, working system.

A high-level belief diagnosis and repair system was connected to the low-level robot platform and tested in two different settings, as described in section 3.5. These successful experiments show that the outcome of this project meets the initial expectations. The resulting robot platform is capable of

- safely navigating in confined office environments avoiding obstacles and dynamically replanning its path,
- detecting tagged objects,
- picking these objects up and delivering them to any spot in the map and
- dealing with various kinds of errors and unexpected events.

The world model implemented during the project is extendable, as manifold functionality can be added based on the underlying support of managing object attributes and relations. In upcoming versions, the WMLogic module could be enriched with new reasoning capabilities. Also, at some point untagged object recognition would be a useful addition to the system. Besides new modules and approaches, improving the old ones is always a relevant topic. For instance, it would be an interesting challenge to speed navigation up while at the same time maintaining the current execution safety. This might only be possible by covering a larger field of view. The Pioneer P3-DX could be equipped with various other actuators that would enable it to manipulate objects at others than ground level. With more robot platforms available, its performance in multi-agent systems could be tested and enhanced.

The implementation of a world model for a mobile robot requires understanding how positions and orientations are treated in 3D space and how they can be calculated with. Background information on traditional problems such as navigation, localization, mapping etc. is necessary even though the modules are already available, to allow for their correct configuration. This project was thus an ideal way to build up hands-on knowledge in a broad set of robotics-specific topics, and having a functioning robot platform as a result is a motivating and approving factor.

Bibliography

[Act03] ActivMedia Robotics, LLC (Hrsg.): *Pioneer 3 & Pioneer 2 H8-Series Operations Manual.* version 3. ActivMedia Robotics, LLC, August 2003

[BRST00] Boutilier, C. ; Reiter, R. ; Soutchanski, M. ; Thrun, S.: Decision-Theoretic, High-Level Agent Programming in the Situation Calculus. In: *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00) and Twelfth Conference on Innovative Applications of Artificial Intelligence (IAAI-00)*, AAAI Press, 2000, S. 355–362

[DGLLS09] De Giacomo, Giuseppe ; Lespérance, Yves ; Levesque, Hector J. ; Sardina, Sebastian: *IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents.* `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.146.2722`. Version: 2009

[DLL00] De Giacomo, G. ; Lésperance, Y. ; Levesque, H.: ConGolog, A Concurrent Programming Language based on Situation Calculus. In: *Artificial Intelligence* 121 (2000), Nr. 1–2, S. 109–169

[FAU10] *Golog - Situationskalkuel in Logik erster Stufe.* Lecture Slides - Kognitive Systeme 1, 2010. – Lehrstuhl fuer Kuenstliche Intelligenz - Friedrich-Alexander-Universitaet Erlangen-Nuernberg

[FBT97] Fox, D. ; Burgard, W. ; Thrun, S.: The dynamic window approach to collision avoidance. In: *Robotics & Automation Magazine, IEEE* 4 (1997), Nr. 1, 23–33. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=580977`

[Fox01] Fox, Dieter: KLD-Sampling: Adaptive Particle Filters. In: *In Advances in Neural Information Processing Systems 14*, MIT Press, 2001, S. 713–720

[FS10] Felfernig, Alexander ; Steinbauer, Gerald: *Advanced Topics in AI - Logic-Based Programming for Agents.* Lecture Slides, 2010

[GRSW11] GSPANDL, Stephan ; REIP, Michael ; STEINBAUER, Gerald ; WOLFRAM, Máté: *A Dependable Decision-Execution Cycle for Autonomous Robots*. San Francisco, California, USA : Submitted for publication at the IEEE/RSJ International Conference on Intelligent Robots and Systems, 2011

[GSB07] GRISETTI, Giorgio ; STACHNISS, Cyrill ; BURGARD, Wolfram: Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters. In: *IEEE Transactions on Robotics* 23 (2007), Februar, Nr. 1, 34–46. `http://dx.doi.org/10.1109/TRO.2006.889486`. – DOI 10.1109/TRO.2006.889486. – ISSN 1552–3098

[Hop10] HOPPE, Christof: *Large-scale Robotic SLAM through Visual Mapping*. Graz, Austria, Graz University of Technology - Institute for Computer Graphics and Vision, Diplomarbeit, November 2010

[IB06] IAGNEMMA, K. ; BUEHLER, M.: Special Issue on the DARPA Grand Challenge. In: *Journal of Field Robotics* 23 (2006), Nr. 8-9

[KB99] KATO, Hirokazu ; BILLINGHURST, Mark: Marker Tracking and HMD Calibration for a Video-Based Augmented Reality Conferencing System. In: *Augmented Reality, International Workshop on* 0 (1999), 85–94. `http://dx.doi.org/10.1109/IWAR.1999.803809`. – DOI 10.1109/IWAR.1999.803809. ISBN 0–7695–0359–4

[KBK08] KOLB, Andreas ; BARTH, Erhardt ; KOCH, Reinhard: ToF-Sensors: New Dimensions for Realism and Interactivity. In: *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, 2008

[Koc08] KOCH, Thomas: *Rotationen mit Quaternionen in der Computergrafik*. Gelsenkirchen, Germany, Fachhochschule Gelsenkirchen, Diplomarbeit, January 2008

[McC63] MCCARTHY, J.: Situations, Actions and Causal Laws / Stanford University. 1963. – Forschungsbericht

[Mic09] MICROSOFT CORPORATION (Hrsg.): *Microsoft(R) LifeCam Cinema(TM) - Technical Data Sheet*. Rev. 0909A. Microsoft Corporation, 2009. `http://www.microsoft.com`

[QCG+09] QUIGLEY, Morgan ; CONLEY, Ken ; GERKEY, Brian P. ; FAUST, Josh ; FOOTE, Tully ; LEIBS, Jeremy ; WHEELER, Rob ; NG, Andrew Y.: ROS: an open-source Robot Operating System. In: *ICRA Workshop on Open Source Software*, 2009

[Rei01] REITER, Raymond: *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. illustrated edition. The MIT Press, 2001 http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0262182181. – ISBN 0262182181

[ROS02] *genesis3d.com - Using Quaternions to Represent Rotation.* Online. http://www.genesis3d.com/~kdtop/Quaternions-UsingToRepresentRotation.htm. Version: 2002. – last visited 3 May 2011

[ROS08] *NASA.gov - Aircraft Rotations.* Online. http://www.grc.nasa.gov/WWW/K-12/airplane/rotations.html. Version: 2008. – last visited 3 May 2011

[ROS11a] *ARToolkitPlus - Presentation.* Online. http://studierstube.icg.tu-graz.ac.at/handheld_ar/artoolkitplus.php. Version: 2011. – last visited 3 May 2011

[ROS11b] *Microsoft Kinect Teardown.* Online. http://www.ros.org/wiki/tf. Version: 2011. – last visited 5 May 2011

[ROS11c] *ROS.org Wiki.* Online. http://www.ros.org/wiki/. Version: 2011. – last visited 12 April 2011

[ROS11d] *ROS.org Wiki - Action Library.* Online. http://www.ros.org/wiki/actionlib. Version: 2011. – last visited 3 May 2011

[ROS11e] *ROS.org Wiki - base local planner.* Online. http://www.ros.org/wiki/base_local_planner. Version: 2011. – last visited 3 May 2011

[ROS11f] *ROS.org Wiki - Master.* Online. http://www.ros.org/wiki/Master. Version: 2011. – last visited 3 May 2011

[ROS11g] *ROS.org Wiki - Messages.* Online. http://www.ros.org/wiki/msg. Version: 2011. – last visited 3 May 2011

[ROS11h] *ROS.org Wiki - Packages.* Online. http://www.ros.org/wiki/Packages. Version: 2011. – last visited 3 May 2011

[ROS11i] *ROS.org Wiki - Parameter Server.* Online. http://www.ros.org/wiki/ParameterServer. Version: 2011. – last visited 3 May 2011

[ROS11j] *ROS.org Wiki - Point Cloud Library.* Online. http://www.ros.org/wiki/pcl. Version: 2011. – last visited 3 May 2011

[ROS11k] *ROS.org Wiki - ROS nodes in Java.* Online. http://www.ros.org/wiki/rosjava. Version: 2011. – last visited 3 May 2011

[ROS11l] *ROS.org Wiki - rosmsg tool.* Online. http://www.ros.org/wiki/rosmsg. Version: 2011. – last visited 3 May 2011

[ROS11m] *ROS.org Wiki - rosparam tool*. Online. http://www.ros.org/wiki/rosparam. Version: 2011. – last visited 3 May 2011

[ROS11n] *ROS.org Wiki - rosservice tool*. Online. http://www.ros.org/wiki/rosservice. Version: 2011. – last visited 3 May 2011

[ROS11o] *ROS.org Wiki - roswtf tool*. Online. http://www.ros.org/wiki/roswtf. Version: 2011. – last visited 3 May 2011

[ROS11p] *ROS.org Wiki - RViz User Guide*. Online. http://www.ros.org/wiki/rviz/UserGuide. Version: 2011. – last visited 8 May 2011

[ROS11q] *ROS.org Wiki - RXBag*. Online. http://www.ros.org/wiki/rxbag. Version: 2011. – last visited 3 May 2011

[ROS11r] *ROS.org Wiki - RXPlot*. Online. http://www.ros.org/wiki/rxplot. Version: 2011. – last visited 3 May 2011

[ROS11s] *ROS.org Wiki - Services*. Online. http://www.ros.org/wiki/Services. Version: 2011. – last visited 3 May 2011

[ROS11t] *ROS.org Wiki - Stack Manifest*. Online. http://www.ros.org/wiki/StackManifest. Version: 2011. – last visited 3 May 2011

[ROS11u] *ROS.org Wiki - Stacks*. Online. http://www.ros.org/wiki/Stacks. Version: 2011. – last visited 3 May 2011

[ROS11v] *ROS.org Wiki - Transformation Framework*. Online. http://www.ros.org/wiki/tf. Version: 2011. – last visited 3 May 2011

[ROS11w] *ROS.org Wiki - Tutorials - Creating Messages and Services*. Online. http://www.ros.org/wiki/ROS/Tutorials/CreatingMsgAndSrv. Version: 2011. – last visited 3 May 2011

[ROS11x] *TUG IST Wiki - Robotics*. Online. http://www.ist.tugraz.at/robotics/bin/view/Main/WebHome. Version: 2011. – last visited 3 May 2011

[ROS11y] *TUG IST Wiki - SVN and Wiki Guidelines*. Online. http://www.ist.tugraz.at/robotics/bin/view/Main/Guidelines_svn_wiki. Version: 2011. – last visited 3 May 2011

[ROS11z] *ROS.org Wiki - Installation Instructions*. Online. http://www.ros.org/wiki/ROS/Installation. Version: 2011. – last visited 12 April 2011

[SICa] SICK AG (Hrsg.): *Lasermeßsystem LMS 200*. Reute, Germany: SICK AG, http://www.sick.de

[SICb] SICK AG (Hrsg.): *LMS200/211/221/291 Laser Measurement Systems - Technical Description*. Reute, Germany: SICK AG, http://www.sick.de

[SN04] Siegwart, Roland ; Nourbakhsh, Illah R.: *Introduction to Autonomous Mobile Robots*. Bradford Book, 2004 http://portal.acm.org/citation.cfm?id=983690. – ISBN 026219502X

[SS03] Scharstein, D. ; Szeliski, R.: High-accuracy stereo depth maps using structured light. Los Alamitos, CA, USA : IEEE Computer Society, 2003. – ISSN 1063–6919, I-195–I-202

[Ste] Steder, Bastian: *The Point Cloud Library - PCL*. Presentation Slides. http://ais.informatik.uni-freiburg.de/teaching/ws10/robotics2/pdfs/rob2-12-ros-pcl.pdf. – University of Freiburg

[TB09] Tenorth, Moritz ; Beetz, Michael: KNOWROB: knowledge processing for autonomous personal robots. In: *IROS'09: Proceedings of the 2009 IEEE/RSJ international conference on Intelligent robots and systems*. Piscataway, NJ, USA : IEEE Press, 2009. – ISBN 978–1–4244–3803–7, 4261–4266

[TBB$^+$99] Thrun, S. ; Bennewitz, M. ; Burgard, W. ; Cremers, A. B. ; Dellaert, F. ; Fox, D. ; Hähnel, D. ; Rosenberg, C. ; Roy, N. ; Schulte, J. ; Schulz, D.: MINERVA: A Second-Generation Museum Tour-Guide Robot. In: *IEEE International Conference on Robotics and Automation*, 1999

[Ten10] Tenorth, Moritz: *Knowledge Processing for Autono mous Robots*. CoTeSys Presentation Slides, 2010

[TFBD01] Thrun, Sebastian ; Fox, Dieter ; Burgard, Wolfram ; Dellaert, Frank: Robust Monte Carlo localization for mobile robots. In: *Artificial Intelligence* 128 (2001), Nr. 1-2, 99–141. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.8488

[TO06] Trebi-Ollennu, A.: Special Issue on Robots on the Red Planet. In: *IEEE Robotics & Automation Magazine* 13 (2006), Nr. 2

[WGRS11] Wolfram, Mátè ; Gspandl, Stephan ; Reip, Michael ; Steinbauer, Gerald: *Robust Robotics Using History-Based-Diagnosis in IndiGolog*. Hall in Tyrol, Austria : Austrian Robotics Workshop, 2011