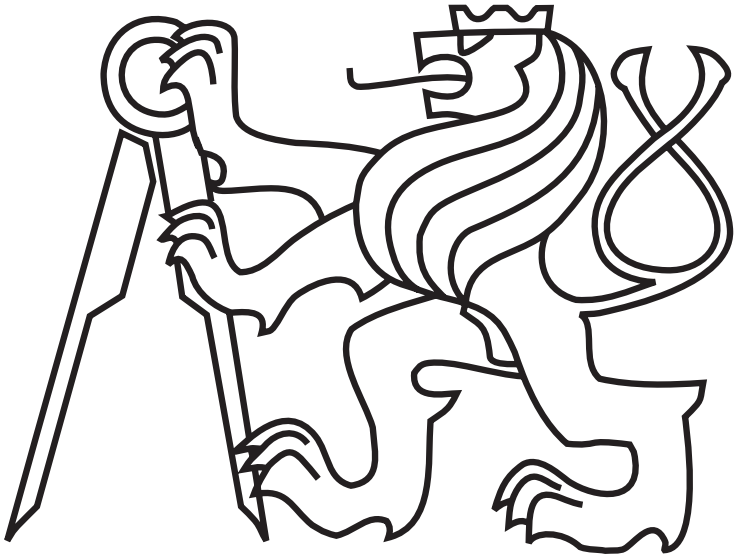
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

DIPLOMA THESIS



Bc. Filip Jareš

Exploration of an Uknown 3D Environment

Department of Cybernetics

Thesis supervisor: **RNDr. Miroslav Kulich, Ph.D.**

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student: Bc. Filip J a r e š

Study programme: Cybernetics and Robotics

Specialisation: Robotics

Title of Diploma Thesis: Exploration of an Uknown 3D Environment

Guidelines:

1. Get acquainted with ROS (Robot Operating System, http://ros.org) and V-REP (http://www.coppeliarobotics.com/) frameworks.
2. Get acquainted with methods for exploration of an unknown 3D environment.
3. Based on discussion with the supervisor, implement a selected exploration method for a robot with several degrees of freedom, e. g. mobile manipulator or humanoid robot.
4. Verify behavior of the implemented method experimentally in the V-REP simulator and discuss the results obtained.

Bibliography/Sources:

[1] C. Dornhege, A. Kleiner: "A frontier-void-based approach for autonomous exploration in 3D," Safety, Security, and Rescue Robotics (SSRR), 2011 IEEE International Symposium on, pp.351-356, 1-5 Nov. 2011.
[2] K. M. Wurm, A. Hornung, M. Bennewitz, C. Stachniss, and W. Burgard: "OctoMap: A Probabilistic, Flexible, and Compact 3D Map Representation for Robotic Systems" in Proc. of the ICRA 2010 Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation, 2010.
[3] R. Shade, P. Newman: "Choosing where to go: Complete 3D exploration with stereo," Robotics and Automation (ICRA), 2011 IEEE International Conference on, pp.2806-2811, 9-13 May 2011.
[4] F. Zacharias, C. Borst, and G. Hirzinger: "Capturing robot workspace structure: representing robot capabilities, " in Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS2007), 2007, pp. 4490–4495.
[5] S. Shen, M. Nathan, V. Kumar: "Autonomous indoor 3D exploration with a micro-aerial vehicle," Robotics and Automation (ICRA), 2012 IEEE International Conference on, pp.9-15, 14-18 May 2012.

Diploma Thesis Supervisor: RNDr. Miroslav Kulich, Ph.D.

Valid until: the end of the summer semester of academic year 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 10, 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Filip J a r e š

Studijní program: Kybernetika a robotika (magisterský)

Obor: Robotika

Název tématu: Prohledávání neznámého 3D prostředí

Pokyny pro vypracování:

1. Seznamte se s prostředím ROS (Robot Operating System, http://ros.org) a V-REP (http://www.coppeliarobotics.com/).
2. Seznamte se s metodami prohledávání neznámého 3D prostředí.
3. Po konzultaci s vedoucím implementujte vybranou metodu prohledávání pro robot s více stupni volnosti, např. mobilní manipulátor či humanoid.
4. Vlastnosti implementované metody experimentálně ověřte v simulátoru V-REP a diskutujte nad získanými výsledky.

Seznam odborné literatury:

[1] C. Dornhege, A. Kleiner: "A frontier-void-based approach for autonomous exploration in 3D," Safety, Security, and Rescue Robotics (SSRR), 2011 IEEE International Symposium on, pp.351-356, 1-5 Nov. 2011.
[2] K. M. Wurm, A. Hornung, M. Bennewitz, C. Stachniss, and W. Burgard: "OctoMap: A Probabilistic, Flexible, and Compact 3D Map Representation for Robotic Systems" in Proc. of the ICRA 2010 Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation, 2010.
[3] R. Shade, P. Newman: "Choosing where to go: Complete 3D exploration with stereo," Robotics and Automation (ICRA), 2011 IEEE International Conference on, pp.2806-2811, 9-13 May 2011.
[4] F. Zacharias, C. Borst, and G. Hirzinger: "Capturing robot workspace structure: representing robot capabilities, " in Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS2007), 2007, pp. 4490–4495.
[5] S. Shen, M. Nathan, V. Kumar: "Autonomous indoor 3D exploration with a micro-aerial vehicle," Robotics and Automation (ICRA), 2012 IEEE International Conference on, pp.9-15, 14-18 May 2012.

Vedoucí diplomové práce: RNDr. Miroslav Kulich, Ph.D.

Platnost zadání: do konce letního semestru 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 10. 1. 2014

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne ....

Acknowledgements

In this place I would like to thank my diploma thesis supervisor RNDr. Miroslav Kulich, Ph.D. for his guidance, patience and understanding. I also want to express my gratitude to my beloved wife Karolína and to our parents for their patience and support. Thanks go to Václav for his interest in the field. I am grateful to all the creators of the wonderful Free software. Thank you. And most importatly, thanks to the Good Creator and Chief Engineer.

Abstrakt

Diplomová práce popisuje algoritmus pro průzkum neznámého 3D prostředí, jehož klíčové části byly implementovány. Jedná se o řešení založené na pojmech „hraničních“ a „prázdných“ oblastí (frontiers and voids), operující nad mřížkou obsazenosti, které bylo původně navrženo v článku Dornhege a Kleinera, a které představuje přirozené zobecnění rozšířené třídy 2D exploračních metod používajících „hraniční“ buňky. Dále práce představuje pojem „mapy schopností“ (capability map), zavedený Franziskou Zacharias. Mapa schopností umožňuje pro daný robotický manipulátor předpočítat charakteristiku pracovního prostoru a její použití při hledání cílu pro plánování pohybu umožňuje vyhnout se mnoha výpočtům inverzní kinematické úlohy. V neposlední řadě práce představuje hlavní prvky simulačního prostředí založeného na ROSu a konfiguraci tohoto prostředí.

Abstract

This thesis describes an algorithm for exploration of an unknown 3D environment that has been partially implemented. It is a frontier-void-based approach working on top of an occupancy grid, originally proposed in the paper by Dornhege and Kleiner, which represents a natural extension of the widespread class of frontier-based exploration algorithms working in 2D. Furthermore, the work presents a concept of capability maps, introduced by Franziska Zacharias, which can be used to precompute capabilities of a robotic manipulator offline and avoid many inverse kinematics computations during the selection of motion planning goals. Finally, key components of a ROS-based simulation environment and its setup are described.

Contents

List of Figures

List of Algorithms

List of Listings

Chapter 1

Introduction

Imagine a robot suddenly finding itself in an unknown room, being given a simple command: explore the room. What should the robot do? Many tasks that look simple and natural to humans – those tasks that usually involve just a little amount of reasoning or no conscious thinking at all – often do not seem that simple when analyzed in more depth. We have already forgotten that we had to learn how to solve them. And while learning these things we possibly did not even notice we are learning anything. Observing small children or other young creatures helps to see the many things we had to learn in order to withstand in the world.

Autonomous exploration, being one of the fundamental problems of mobile robotics, is an autonomous activity of the robot involving search for the purpose of discovery of information about the given territory. The information is being recorded in a map and the process of building the map is called *mapping*. For robots the map has the same purpose as for humans: it represents a model of the environment that can help them with *localization* and *motion planning*. Localization attempts to answer the question on the current position of the robot. Motion planning strives to find a way to reach the goal. Mapping, localization and motion planning are often regarded as being the three essential tasks of mobile robotics. Their relationships are captured in Figure 1.1. Possible applications of autonomous robotic exploration include mapping and so-called urban search and rescue.

Most of the established exploration algorithms consider environments that can be modelled using a 2D map. In some applications two dimensional representation of the environment is not sufficient. This thesis aims at implementation of an autonomous exploration algorithm capable of operating in 3D environment.

The outline of this thesis is as follows. Chapter 2 describes the exploration problem and shortly presents existing exploration algorithms. Chapter 3 describes the choice of algorithm and the selected algorithm itself. A library for representation of capability maps – a concept introduced by Franziska Zacharias et al. in [19] – has been implemented to support the selected exploration algorithm. Capability maps are also described in Chapter 3. This work had been started with a hope that a simulated robotic platform will be available to work with. This was not the case and therefore Chapter 4 presents a simulation environment, based on tools from

the Robot Operating System (ROS) [7], that has been prepared and that was intended for execution of experiments. Results of experiments with artifacts described in previous chapters are presented in Chapter 5. The visualization tools used during the development are mentioned there too. The final evaluation of this work is in Chapter 6.

Appendix A contains an overview of the contents of the attached CD. In Appendix B an overview of the most important ROS packages used in the simulation environment is presented. Appendix C describes the structure of the motion planning configuration package for the HUBO robot used for the simulation.

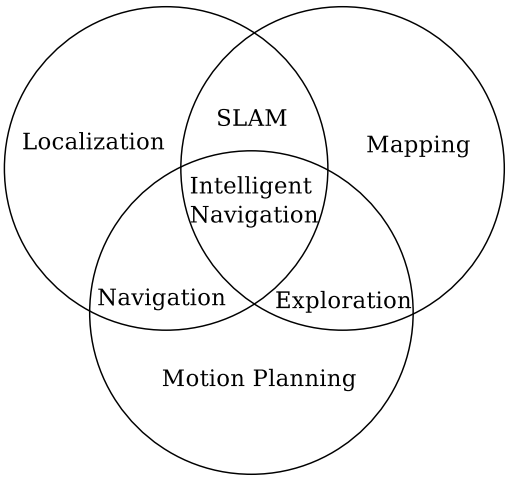


Figure 1.1: Essential tasks of mobile robotics. SLAM stands for Simultaneous Localization and Mapping. Taken from [6].

Chapter 2

Autonomous Exploration

Exploration in mobile robotics is an iterative process that involves navigation. During exploration the robot autonomously moves through an unknown environment with the aim of creating a map of the environment that can be used for subsequent navigation. Exploration is closely related to the other essential tasks of mobile robotics shown in Figure 1.1. It involves mapping of the environment and the sequentially built map serves as a model used for planning of further exploration steps.

Generic Exploration Algorithm Frame

The central question in autonomous exploration is: "given what you know about the world, where should you move to gain as much new information as possible?" [17]. A simplified generic exploration frame can be described in five steps:

1. Scan & Integrate – Capture sensor measurements (e.g. a point cloud) and integrate them into the map.
2. Evaluate – Find and evaluate appropriate goals; select next goal.
3. Plan – Create a motion plan (trajectory) to reach the next goal.
4. Move – Execute the motion plan created in the previous step.
5. Repeat – Unless a termination criterion has been met, continue with 1.

The termination criterion for exploration can be specified as (nearly) complete coverage of the search space.

A problem closely related to exploration is search. A generic search algorithm frame would look very similar to the one just stated, but it differs not only in the termination criterion (object searched for has been found) but also the evaluation step would be different. Whereas

exploration strives to minimize the overall time needed to explore the whole search space, search attempts to minimize the time needed to find the object it is searching for.

A different problem with a link to exploration is the art gallery problem illustrated in Figure 2.1. In the art gallery problem the task is to find the optimal placement of guards or cameras on a polygonal representation of a known map, such that the entire space is covered by their field of view.

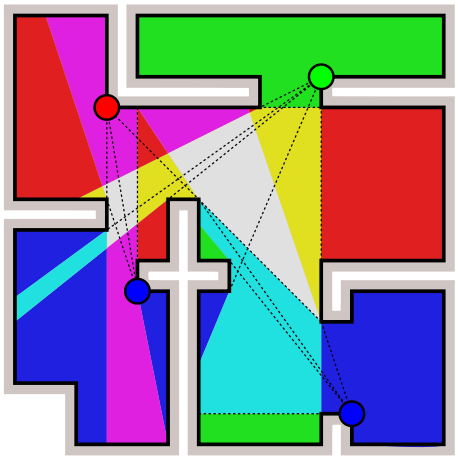


Figure 2.1: Art gallery problem illustration from Wikipedia.

A classical exploration approach in 2D is represented by the *frontier*-based class of exploration algorithms originally proposed by Yamauchi [17]. A frontier can be defined as the boundary between known unoccupied space and the unknown territory. The frontier-based approach is based on the idea that in order to gain the most new information from the environment, the robot (or its sensor) should move to the boundary between free space and unknown territory.

A simple exploration scenario illustrating the concept of frontiers is shown in Figure 2.2.

A problem related to exploration in 3D is the next best view computation which computes a sequence of viewpoints until an entire scene has been observed by a sensor. For exploration in 3D, Dornhege and Kleiner in [4] introduced so-called frontier-void-based exploration approach working on top of an occupancy grid. This algorithm is described in depth in the following chapter. Apart from that, gradient based solutions exist (Shade and Newman, [9]). Shen et al. [10] proposed 3D exploration solution based on stochastic differential equations. Shen et al. used a "multi-volume occupancy grid", introduced in [5], to represent the map, whereas the aforementioned approaches use octrees for occupancy grid storage.

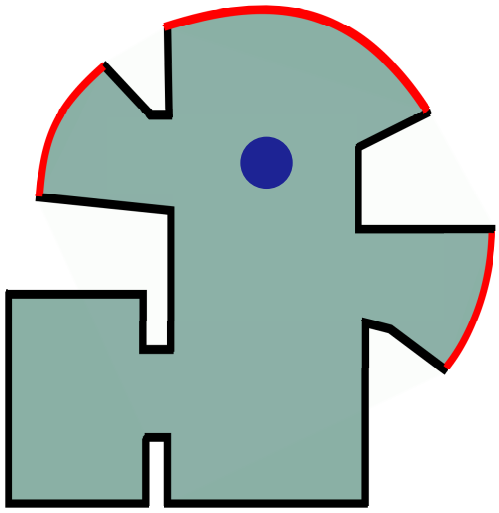


Figure 2.2: The concept of frontiers: walls (obstacles) shown in black, explored free areas in grey and frontiers in red.

Chapter 3

Frontier-Void-Based Exploration

This chapter presents the choice of the exploration algorithm that has been implemented, the core part of the algorithm itself and some information on relevant implementation details.

The task was to implement an exploration algorithm working in three dimensions. The Frontier-Void-Based Exploration Algorithm was selected because it represents a natural extension of the widespread class of 2D frontier-based exploration algorithms into three dimensions and because it deals with kinematic constraints of the robot.

The code has been written in C++, as it is a language that was designed to allow high performance. The ROS framework (described in Chapter 4) supports it as the first class language and many neccessary libraries are implemented in C++ or have C/C++ API. The most important ones for us were OctoMap, Qhull and MoveIt!. Last but not least I also wanted to to take a chance to improve my knowledge of the language.

The core of the Frontier-Void-Based Exploration Algorithm is presented in Section 3.1. In Section 3.2 a capability mapping technique is presented which makes it possible to precompute directional characteristics of a robotic manipulator offline and store it for later use. This can be used in the frontier-void-based exploration algorithm in the evaluation and planning steps and helps to avoid many inverse kinematics computations during the selection of motion planning goals. Section 3.3 briefly describes the OctoMap library, which has been used for efficient storage of the occupancy grid. Some details related to the capability mapping library, which has been implemented to support reasoning about the workspace of the robot's manipulator, are mentioned in the last section too.

3.1 Goal Evaluation and Selection

The Frontier-Void-Based Exploration algorithm, introduced by Dornhege and Kleiner in [4], is described in this section with minor corrections, clarifications and updates. Only the portion of it that corresponds to the "evaluation" part of the algorithm frame presented in the previous chapter on page 3 is discussed here. Input to the evaluation part is the current state of the

map together with newly acquired sensor data (point cloud). Output of the evaluation part is a motion plan that can be executed.

The algorithm is built on top of an occupancy grid – a map that is formed by a regular grid of cube cells (voxels) which can be classified into three disjunct categories: *occupied voxels* are voxels containing points from the point cloud; *free voxels* are voxels which have been covered by the sensor's field of view and do not contain any points (obstacles) and *unknown voxels* are voxels which have never been covered by the sensor's field of view. Visualization of occupied and free voxels can be seen in Fig. 3.1 (c) and (d) respectively. Voxels being neither occupied nor free are unknown and are not shown.

Free and unknown voxels are further differentiated. In accordance with Yamauchi [17], *frontier voxels* are defined as free voxels neighboring any unknown ones. *Void voxels* are unknown voxels which are located within the convex hull[1] of all occupied voxels in the map. Dornhege and Kleiner [4] do not mention what definition of voxel neighborhood they used. We have chosen a definition based on 26-connectivity. Therefore in the following text, two voxels are considered neighbors if and only if they share a common corner.

The idea behind the Frontier-Void-Based exploration approach is to estimate the "exploration potential" (utility) of free voxels in the known map and to use this utility to find "the next best view" – the next exploration goal. In the evaluation part the following steps are performed with the first three items being described in Section 3.1.1 and the last one being described in Section 3.1.2:

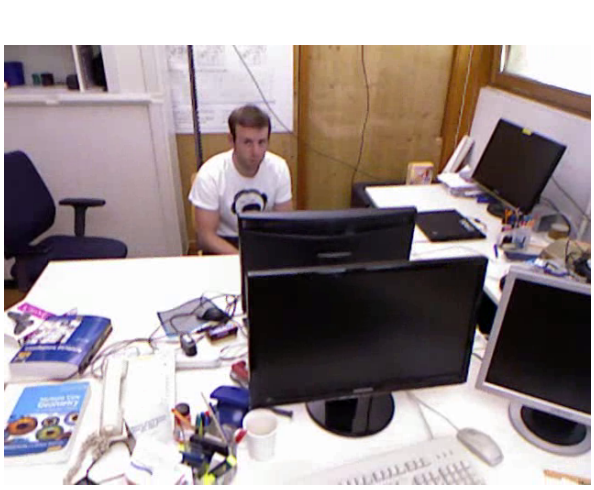
1. void voxels are extracted and ellipsoidal representation of void clusters is built,
2. frontier voxels are extracted and clustered,
3. a set of frontier-voids is created that binds together frontier clusters with void clusters and
4. the set of frontier-voids is used to determine next best view locations.

3.1.1 Extraction of Frontier and Void Voxels and Their Clustering

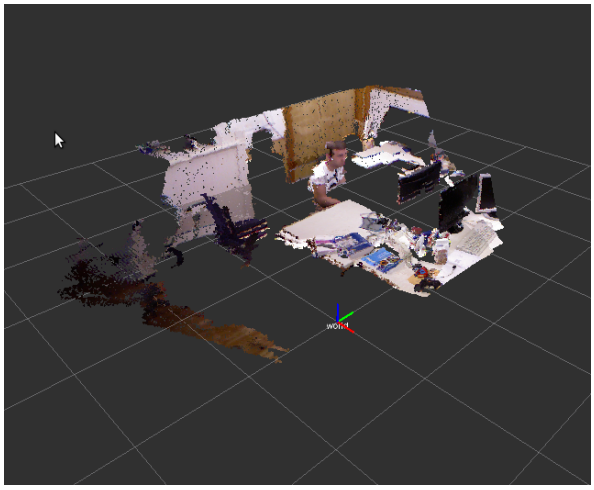
Clustering of Void Voxels – Fitting Ellipsoids to Void Voxels

After the individual void voxels are identified, instead of working with them directly, they are grouped into void clusters, each cluster being represented by an ellipsoid. Ellipsoids have been used in [4] since they naturally model cylindrical and spheroidal distributions, as the authors claim.

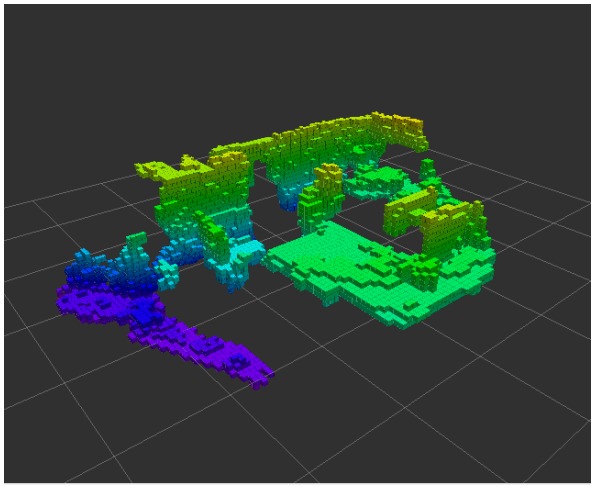
[1] *QHull* library [2] has been used to compute convex hulls.



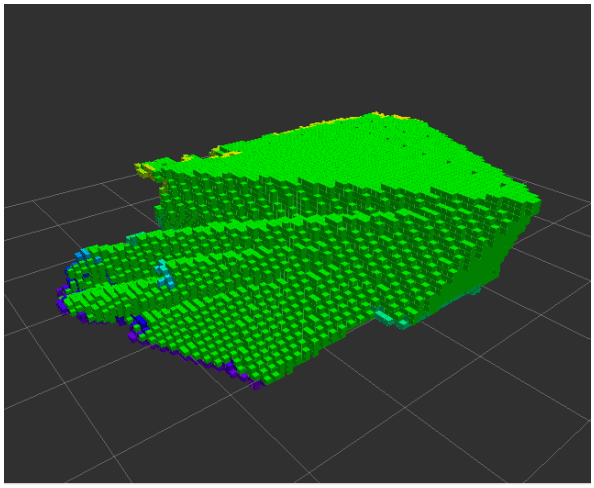
(a) Snapshot of the Scene From the Sensor



(b) Point cloud



(c) Occupied voxels (height color map)



(d) Free voxels (green)

Figure 3.1: Voxel types resulting from the first scan of an environment. Data courtesy of [12].

Principal component analysis is employed to obtain ellipsoidal representation of a given set of n void voxel centers $\mathbf{P} = \{\mathbf{p}_i\}_{i=1}^{n}$, where $\mathbf{p_i} = (x_i, y_i, z_i)^\mathsf{T}$. The symmetric positive definite covariance matrix $\mathbf{\Sigma}$ for a set of points is given by

$$\mathbf{\Sigma} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{p_i} - \boldsymbol{\mu})(\mathbf{p_i} - \boldsymbol{\mu})^\mathsf{T}, \qquad \text{where} \qquad \boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{p_i}. \tag{3.1}$$

Here $\boldsymbol{\mu}$ is the centroid of $\mathbf{P}$. The matrix $\mathbf{\Sigma}$ is then decomposed into principal components by eigenvalue decomposition. An ellipsoid v_i is described by the centroid, eigenvalues and corresponding eigenvectors of its covariance matrix and by the number of void voxels it represents; in other words, it is described by the tuple $v_i = (\boldsymbol{\mu}_i, \mathbf{e}_{i1}, \mathbf{e}_{i2}, \mathbf{e}_{i3}, \lambda_{i1}, \lambda_{i2}, \lambda_{i3}, n_i)$.

In the beginning random sampling is carried out in order to choose "seeds" among void voxels. For each chosen "seed void voxel", its neighbors are examined and an ellipsoid is created that represents the collection of void voxels formed by the selected seed voxel together with any neighboring void voxels that are not attached to any ellipsoid yet. Next, any void voxels left (not attached to any ellipsoid) together with their (unattached) void voxel neighbors are used to create ellipsoids representing them. Now each void voxel is part of some ellipsoid (future void cluster). For each void voxel a reference to the corresponding ellipsoid representing it is maintained.

In a further process, ellipsoids are merged in order to minimize the number of ellipsoids needed for representation of a single void cluster while maximizing the fitting between the ellipsoids and their corresponding void voxels. To quantify this fitting, the following ratio is computed for the merger-ellipsoid and maintained high enough:

$$S = \frac{\text{volume of void voxels}}{\text{volume of the corresponding ellipsoid}} = \frac{nR^3}{\frac{4}{3}\pi\lambda_1\lambda_2\lambda_3}. \tag{3.2}$$

Here, R is the resolution of the map grid (length of the edge of a voxel), n is the number of void voxels the ellipsoid stands for and λ_i are the eigenvalues of the ellipsoid's covariance matrix. Ellipsoids are only merged if the resulting ellipsoid has S above some threshold. The merging procedure merges the nearest ellipsoid pairs first and terminates when there is no merge possible with a result that would satisfy (3.2).

Merging of ellipsoids

Two ellipsoids built from mutually disjunct sets of points can be merged efficiently. Parameters of an ellipsoid C resulting from a merge of ellipsoids A and B can be computed using following formulas for number of void voxels n_C and the centroid $\boldsymbol{\mu}_C$:

$$n_C = n_A + n_B \tag{3.3}$$

$$\boldsymbol{\mu}_C = \frac{n_A\boldsymbol{\mu}_A + n_B\boldsymbol{\mu}_B}{n_C} \tag{3.4}$$

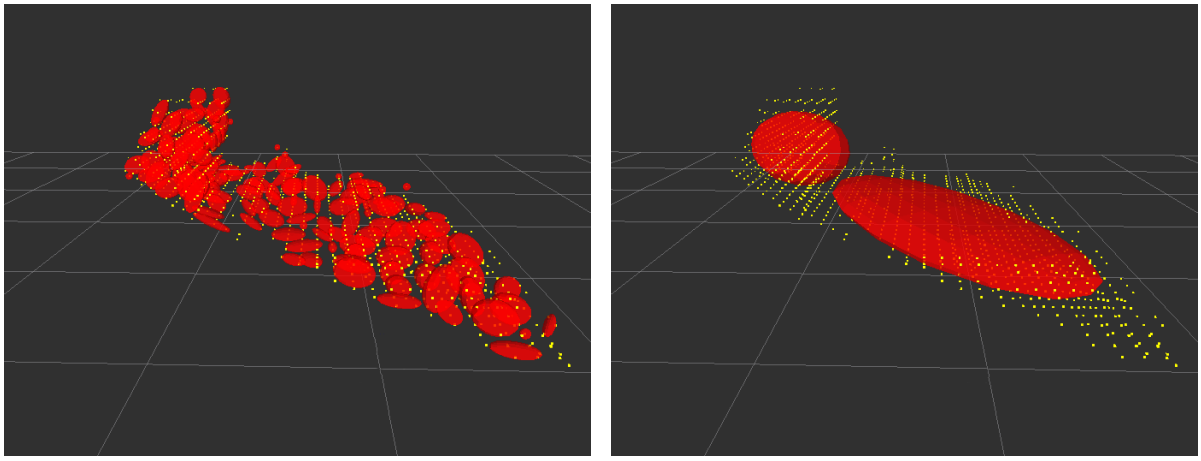
and the following formula for the (i,j)-th element of the covariance matrix $\boldsymbol{\Sigma}_C$ based on [1]:

$$\boldsymbol{\Sigma}_{C(i,j)} = \frac{1}{n_C}\left[n_A\boldsymbol{\Sigma}_{A(i,j)} + n_B\boldsymbol{\Sigma}_{B(i,j)} + [\boldsymbol{\mu}_A(i) - \boldsymbol{\mu}_B(i)][\boldsymbol{\mu}_A(j) - \boldsymbol{\mu}_B(j)]\frac{n_An_B}{n_C}\right]. \tag{3.5}$$

Here $\boldsymbol{\Sigma}_{(i,j)}$ denotes the element in the i-th row and j-th column of the matrix $\boldsymbol{\Sigma}$ and the following notation has been used to denote elements of the centroid vector:

$$\boldsymbol{\mu} = \begin{pmatrix}\boldsymbol{\mu}(1) & \boldsymbol{\mu}(2) & \boldsymbol{\mu}(3)\end{pmatrix}^{\mathsf{T}} \tag{3.6}$$

Initial ellipsoids built from random sampling and results of their merge are shown in Fig. 3.2.



(a) Initial ellipsoids – visualization (b) Merged ellipsoids – visualization

Figure 3.2: Void clustering: merging of ellipsoids representing void clusters. Yellow dots represent centers of void voxels.

Clustering of Frontier Voxels

After frontier voxels are identified, they are clustered using an union-find algorithm.[2] Frontier clusters correspond to connected components of a graph in which nodes are frontier voxels and an edge exists between two nodes if and only if they are neighbors.

The Set of Frontier-Voids

Let's introduce the following notation:

$$\mathcal{FC} - \text{set of frontier clusters,} \qquad \mathcal{VC} - \text{set of void clusters.}$$

Prior to being used for the next best view computation, frontier and void clusters are grouped into the final set of *frontier-voids* $\mathcal{FV}$, each element $fv_i \in \mathcal{FV}$ being defined by a pair formed by an ellipsoid (void cluster) v_i and F_i – a set of frontier clusters neighboring this void cluster.

$$fv_i = (v_i \in \mathcal{VC},\ F_i \subset \mathcal{FC})$$

3.1.2 Next Best View Computation

The goal of the next best view computation is to find configurations of the sensor from which the maximum amount of void space can be observed. Utility value for free voxels is maintained in the map and in free voxels being close to frontiers (and voids), utility is being accumulated.

[2] Union-find algorithms have been analyzed in depth in [15]. We have used boost::disjoint_sets [11] implementation of union-find.

Later, utility of free voxels is used to select the next best view. The approach used in the accumulation step is described in Algorithm 1. The set of frontier-voids is processed, each item $fv = (v, F)$ consisting of a void and a set of frontier voxels. Utility vectors pointing in several directions are created for each frontier voxel $f \in F$ and are used to perform ray tracing into the set of free voxels in the map. Utility of each visited free voxel is increased by the value of $volume(v) = \frac{4}{3}\pi\lambda_1\lambda_2\lambda_3$, which is the volume of the ellipsoid v.

Other functions used in the algorithm 1 are $pos(.)$ that returns 3D position of the center of the given voxel and $corners(v_i)$ that returns a set of seven 3D positions related to the given void cluster – the centroid and set of "extremas" of the ellipsoid:

$$corners(v_i) = \boldsymbol{\mu}_i \cup \bigcup_{j\in\{1,2,3\}} \boldsymbol{\mu}_i \pm \lambda_{ij}\mathbf{e}_{ij}.$$

The function $individualVoxels(F)$ returns a set of frontier voxels that form the set F and $freeVoxelsOnRay(f,\ dir,\ maxRange)$ returns set of free voxels on the ray starting at frontier voxel f in the direction given by dir and with the maximum length of $maxRange$.

```
// Compute utility vectors
UV ← ∅
foreach fv = (v, F) ∈ FV do
    foreach c ∈ corners(v) do
        foreach f ∈ individualVoxels(F) do
            dir ← (pos(f) − c) / ||pos(f) − c||
            utility ← volume(v)
            UV ← UV ∪ (f, dir, utility)
        end
    end
end
// Accumulate utilities in free voxels
foreach uv = (f, dir, u) ∈ UV do
    foreach c ∈ freeVoxelsOnRay(f, dir, s_r) do
        util(c) ← util(c) + u
    end
end
```

Algorithm 1: Next best view computation; s_r denotes the maximum sensor range.

3.2 Capability Maps

Franziska Zacharias et al. realized that robotic manipulator *capabilities* can be precomputed and stored for later use to avoid excessive inverse kinematics computations [19]. Their idea is

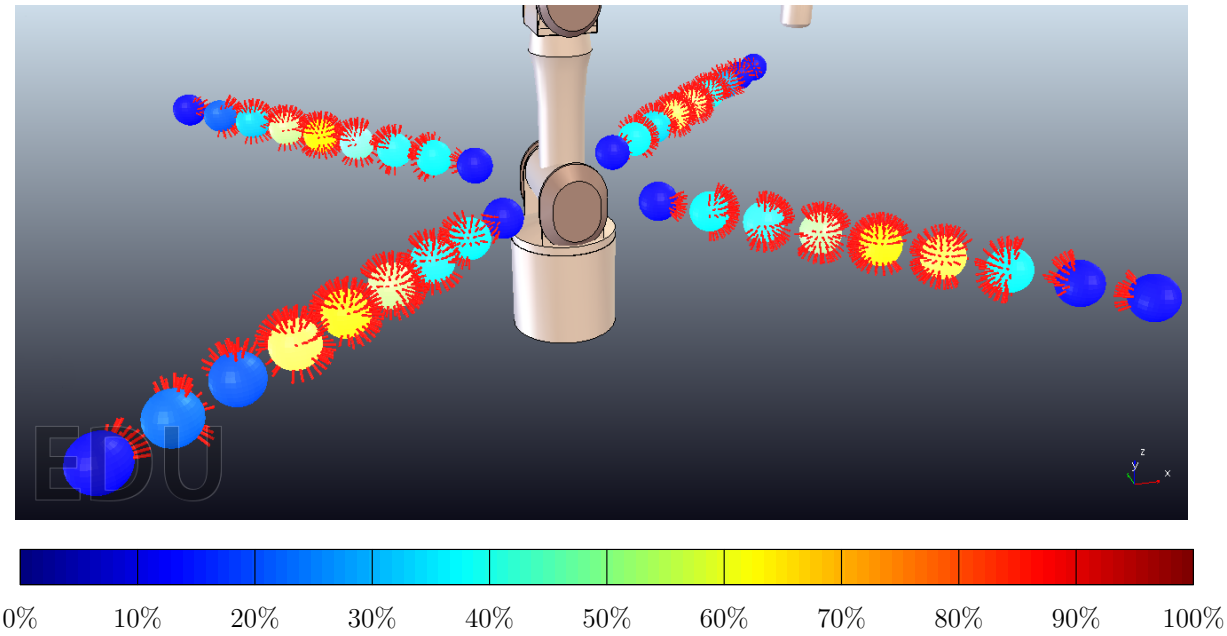


Figure 3.3: Two rows of reachability spheres for a 7DoF manipulator shown in the V-REP simulator; color of reachability sphere corresponds to its reachability index.

based on the fact that we, as humans in our early childhood, learn what *capabilities* our arms have and while looking at objects in our vicinity (or just envisioning them), we are able to estimate whether they are reachable easily and imagine how to best approach them.

A concept of a *capability map* that captures directional preferences of a manipulator has been introduced in [19]. In the manipulator workspace, there are regions that can only be reached from specific directions. Capability map captures this directional information in a data structure which is precomputed offline and helps to choose good approach directions for objects and answer questions like "how to position a mobile platform to have optimal manipulation capabilities with respect to the operating area". A small C++ library that provides representation of capability maps and makes it possible to build and use them has been implemented as part of this work. The rest of this section describes capability maps. A more detailed description, together with the design rationale, can be found in the original article.

This section describes how a capability map is constructed for a manipulator. The process can be divided into two steps: (1) computation of so-called reachability spheres, which involves massive inverse kinematics computations, and (2) replacement of reachability spheres with shape primitives, which condenses the information contained in the reachability spheres and makes it easier to use.

(1) Computation of reachability spheres

The workspace of the robot arm can be encapsulated in a cube, with side length equal to two arm lengths, centered at the robot base. The cube enveloping the workspace is discretized into equally sized smaller cubes and each cube is examined using inverse kinematics computations.

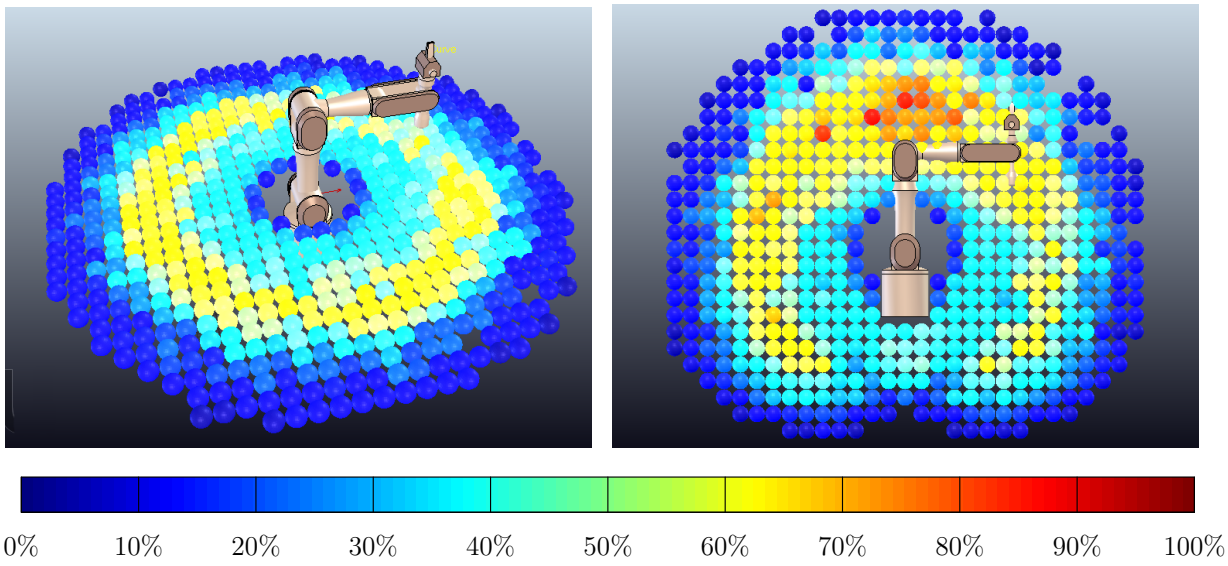


Figure 3.4: Visualisation of reachability index across a 7DoF manipulator workspace.

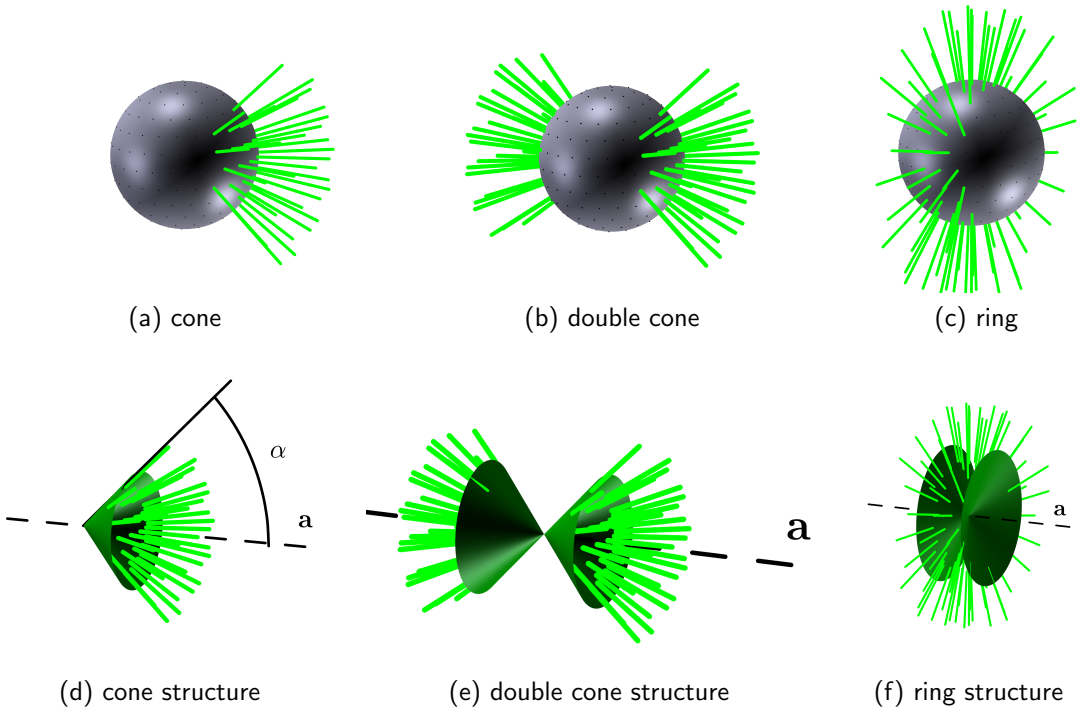
A sphere with diameter equal to the width of the cube is inscribed into each small cube. This sphere captures the reachability of the workspace region contained in the small cube and it is therefore called a *reachability sphere*. N equally distributed points are placed on the surface of the sphere. For each such point a coordinate frame is selected, such that its x and y axes are tangential to the sphere surface and the z-axis points towards the sphere center. The coordinate frame is then rotated around its z-axis with a fixed step size. The coordinate frame represents an end-effector attitude that is used as an input to the inverse kinematics. (The z-axis corresponds to the camera orientation for the purpose of the frontier-void based exploration algorithm.) If for one of the rotated frames at a specific point p on the reachability sphere an inverse kinematics solution exists, the point p is marked as such. As a result, the reachability sphere contains directional information regarding the accessibility of its region. Note, however, that for points having an inverse kinematics (IK) solution, the reachability sphere discards any information regarding viable rotations of the end-effector around the axis pointing to the reachability sphere center (around the z-axis of the end-effector's coordinate frame).

When visualizing reachability spheres, for points with a valid inverse kinematics solution, a line can be drawn pointing in the direction from the sphere's center as shown in Figures 3.3 and 3.5. A directionless measure called *reachability index* D can be assigned to a reachability sphere, that captures the portion of points on the sphere which have an inverse kinematics solution.

$$D = \frac{R}{N} \cdot 100, \qquad R \leq N \tag{3.7}$$

Here, R is the count of valid points on the reachability sphere (points having an IK solution for some rotation around the z-axis) and N is the total number of points on the sphere. The reachability index can be used to visualize some structure inherent to the robot workspace. Two

such pictures are shown in Fig. 3.4.[3]



(a) cone (b) double cone (c) ring

(d) cone structure (e) double cone structure (f) ring structure

Figure 3.5: Reachability spheres and corresponding reachability structures with their axes $\mathbf{a}$ and opening angles α shown. Double cone and ring are referred to as cylinder type 1 and cylinder type 2 respectively too.

(2) Replacement of reachability spheres with shape primitives

The reachability index is however a directionless measure. Reachability spheres can be used to reason about directional preferences of the manipulator in certain workspace regions, but testing a specific direction against all the points on every reachability sphere present in the capability map would be computationally expensive. It turns out that the structure of reachability spheres can be well approximated by three types of geometric structures: a cone, a double cone and a ring. Typical representants of invividual reachability sphere types and corresponding reachability structures are shown in Fig. 3.5. Reachability spheres corresponding to the cone and ring shapes can be seen in Fig. 3.3 showing two rows of reachability spheres in the workspace of a redundant manipulator with 7 degrees of freedom. It feels natural that reachability spheres in distant parts of the manipulator workspace have a low reachability index and can be represented by cones. The same is true for reachability spheres near to the manipulator base. As one moves

[3]Please note that reachability spheres in Fig. 3.3 and reachability index in Fig. 3.4 show some unexpected irregularities. We attribute this to instability of the inverse kinematics solver used. The built-in IK solver of the V-REP simulator has been used. The (possibly alleged) effects of the IK solver used for capability map computation are discussed more extensively in Section 5.6.

into the workspace, the number of valid points on single reachability sphere increases together with the cone's opening angle. As one moves even further, the structure of reachability spheres changes too and cones change to rings. Replacing reachability spheres with shape primitives results in data reduction and faster queries. To determine whether a direction vector constitutes a viable end-effector pose, computation of the angle between this vector and the axis of the shape and its comparison with the shape's opening angle suffices. This makes up a substantial speed up when compared to inverse kinematics computation.

Reachability structure can be described by its type, axis $\mathbf{a}$ and opening angle α. Definition of these values is shown in Fig. 3.5. The axis $\mathbf{a}$ is an unit vector originating at the center of the reachability sphere. For cone, the axis orientation corresponds to the direction of the increasing diameter of the cone. The same is true for the double cone structure, but in this case the axis is not unique and the opposite axis vector represents the same reachability structure. The ring reachability structure in a way constitutes a complement of the double cone reachability structure. Ring having valid points where the double cone has invalid points and vice versa.

The main task of the capability mapping library, besides the representation of reachability structures, is fitting the shape primitives to the data (to reachability spheres). To evaluate the appropriateness of shape primitive type and its parameters, Zacharias et al. introduced shape fit error (SFE) – a measure that evaluates the relative error of the reachability structure representation. SFE is derived from the relative error of the representation of the particular reachability sphere by the reachability structure (primitive shape). The ideal shape covers all R valid points on the sphere. If the particular non-ideal shape wrongly covers u unreachable points (which have no IK solution) and fails to cover r reachable ones, the relative error of the aproximation is $\frac{u+r}{R}$. The shape fit error is bounded, with the limits being 0 and 100 %. We handle reachability spheres with no valid points separately and represent them by "empty" reachability structures which would have zero SFE. For non-empty ($R > 0$) reachability structure with given axis and angle we define:

$$SFE(\mathbf{a}, \alpha) = \begin{cases} \frac{u+r}{R} \cdot 100 & \text{if } u + r \leq R \\ 100 & \text{if } u + r > R \end{cases} \tag{3.8}$$

Fitting shape primitive of given type to the data involves finding the axis and the opening angle such that the resulting shape best represents the reachability sphere. The article [19] does not describe this in more detail. Our approach is as follows. Let us denote the set of valid points on a the reachability sphere as $\mathbf{R}$. To find the axis, each valid point $\mathbf{p} \in \mathbf{R}$ is substituted with a unit vector oriented in the direction from the sphere center and denote it $\mathrm{nvec}(\mathbf{p})$:

$$\mathrm{nvec}(\mathbf{p}) = \frac{\mathbf{p} - \mathbf{c}}{\|\mathbf{p} - \mathbf{c}\|}, \qquad \text{where } \mathbf{c} \text{ is the reachability sphere center.} \tag{3.9}$$

Such vectors are called n-vectors and they uniquely represent the position on the sphere while having no singularities. When fitting a cone, the direction of its axis is established by taking a sum of these n-vectors:

$$\mathbf{a}_{\mathrm{cone}} = \frac{\sum_{\mathbf{p} \in \mathbf{R}} \mathrm{nvec}(\mathbf{p})}{\left\| \sum_{\mathbf{p} \in \mathbf{R}} \mathrm{nvec}(\mathbf{p}) \right\|}. \tag{3.10}$$

For the other structure types, reachability sphere's n-vectors are analyzed using principal component analysis. Covariance matrix $\mathbf{\Sigma}$ is computed and its eigendecomposition is performed.

$$\mathbf{\Sigma} = \frac{1}{|\mathbf{R}|} \sum_{\mathbf{p} \in \mathbf{R}} \text{nvec}(\mathbf{p}) \, \text{nvec}(\mathbf{p})^{\mathsf{T}} \tag{3.11}$$

Obtained eigenvectors are then sorted by the magnitude of their eigenvalues, having $\mathbf{e}_1$, $\mathbf{e}_2$ and $\mathbf{e}_3$ with $\lambda_1 \geq \lambda_2 \geq \lambda_3$. For double cone, the eigenvector with the largest eigenvalue is chosen as the axis. When fitting a ring, the eigenvector with the smallest eigenvalue is chosen:

$$\mathbf{a}_{\text{double cone}} = \mathbf{e}_1, \qquad \mathbf{a}_{\text{ring}} = \mathbf{e}_3. \tag{3.12}$$

The best opening angle α is then identified by evaluating an angle between the selected axis and individual n-vectors on the reachability sphere. For the cone and the double cone with a given axis this is done by finding the minimum α such that the cone/double cone covers all the valid points on the reachability sphere. For the ring reachability structure with a given axis maximum such α is found. For both the double cone and ring structures, α can not exceed $\frac{\pi}{2}$.

When building the capability map, all three types of structures are fitted to every reachability sphere and their SFE is evaluated. The reachability structure type with the smallest SFE is chosen to represent the reachability sphere's region in the capability map.

3.3 Implementation Details

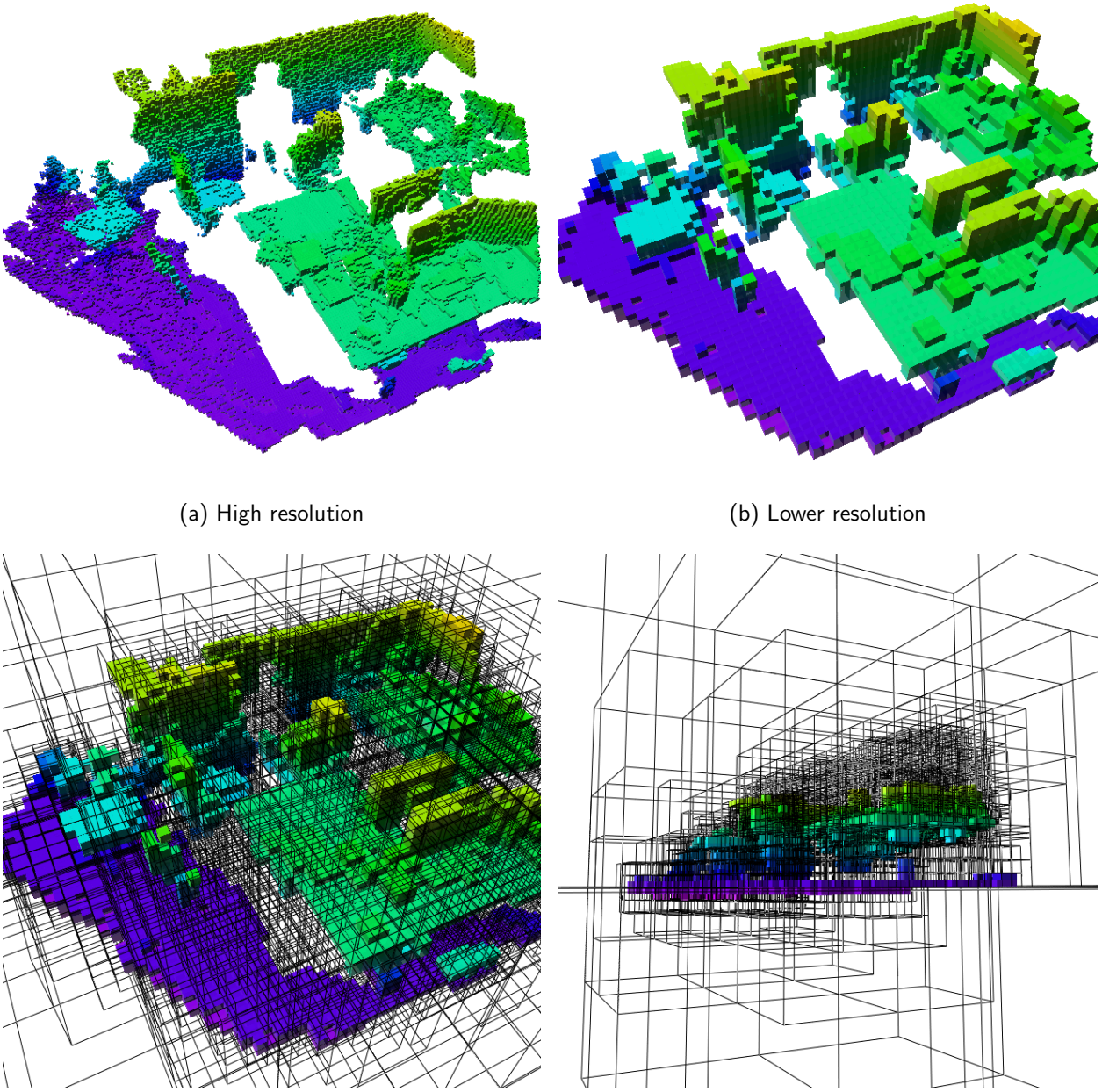
3.3.1 OctoMap

The Frontier-Void-Based algorithm works with a 3D occupancy grid. For an efficient storage of occupancy data, an open-source library named OctoMap has been used. It utilizes an octree data structure to represent the map and encapsulates the map behind a continuous-grid-based interface. This avoids the overhead inherent to plain occupancy grids containing large uniform areas. Moreover, the map can be accessed at different resolutions (see Figure 3.6 which also shows the inner structure of the map in (c) and (d)) The OctoMap has been described by its authors in [16].

The OctoMap library has been extended to make it possible to store utility values of free nodes. This has been done in `UtilityOcTreeNode` and `UtilityOcTree` classes which extend `OccupancyOcTreeBase<UtilityOcTreeNode>` and `OcTreeNode` classes respectively.

3.3.2 Capability Map Parameters and Implementation Details

The capability map construction process has several parameters: the resolution of the capability map (reachability sphere size), the number of points on single reachability sphere N and the rotational step used during examination of single point on a reachability sphere. In this



(a) High resolution

(b) Lower resolution

(c) Octree structure shown

(d) Octree structure shown – another view

Figure 3.6: OctoMap example showing the scene from Fig. 3.1a. Data courtesy of [12].

work we used similar parameter values as the original authors. Figures 3.3 and 3.4 were created with reachability sphere radius approximately 30 times smaller than manipulator arm's length, $N = 200$ and the rotational step used was $\frac{\pi}{6}$.

To distribute N points on a sphere, we used the spiral point algorithm proposed by Rakhmanov

et al. in [8], which gives sufficiently uniform distribution of points on sphere's surface. It needs to be mentioned that either the capability map construction process or the inverse kinematics solver used needs to discard configurations that produce self-collisions. There are some possible optimizations to the capability map construction process (they have limited effect, because capability maps are precomputed offline). If the manipulator has any symmetries, this fact could be used to avoid unneccessary inverse kinematics computations. Further computations can be avoided for spheres lying out of reach of the manipulator (in distant corners of the discretized workspace).

Capability maps capture directional preferences of the manipulator. They do not discriminate regions with singularities and they therefore do not help to find configurations where directionally uniform movement is possible. To address this aspect of manipulator workspace, capability map could be combined with a singularity map based on manipulability measure described by Yoshikawa in [18]. However, for the purpose of Frontier-Void based exploration such extension was not neccessary.

Chapter 4

Simulation Environment

This chapter describes the simulation environment and tools intended to execute experiments with the exploration algorithm described in Chapter 3. A robotic platform was needed to run the exploration and tune the implementation. The work was started with a hope that there will be simulation environment ready to use in the final phase of the work. This was not the case and therefore an attempt has been made to prepare it on our own. A setup had to be prepared, capable of simulating 3D environment and a robot with range imaging sensor. A setup that could be easily modified, if needed, to work with a real sensor and a real robot instead of a simulated one.

From the beginning it was clear that ROS (Robot Operating System, [7]) will play a role in this simulation environment, mainly because it provides a distributed communication middleware and provides an infrastructure that makes it possible to connect components easily and faciliates modular design. It also provides and groups tools and useful libraries (libraries for coordinate frame transformations, point cloud manipulation, visualization and data logging to name a few). In addition to that, ROS is well estabilished in the robotic community and we already had a prior experience with it.

Many components in the simulation environment need an access to the robot model. For example the simulator needs to know what the robot looks like and what its properties are; the motion planner and inverse kinematics solver require kinematic description of the robot. ROS provides a single file format that can be used to describe the robot once and use this description in all the components that need it. This "Unified Robotic Description Format" is described in Section 4.1.

In the the ROS-based simulation environment individual components are represented by ROS nodes. ROS nodes, possible forms of their cooperation and other elementary ROS-related concepts are described in Section 4.2.

ROS now provides generic controllers organized in the ROS Control project. It is mentioned in Section 4.3. An overview of the MoveIt! motion planning framework, which is also a ROS-based project, is given in Section 4.4.

The central part of the simulation environment is a simulator. The main requirement was the ability of the simulator to simulate 3D environment. Two simulator candidates were considered and compared: Gazebo and V-REP. Both are briefly characterized in Section 4.5. Interfacing the Gazebo with the selected MoveIt! motion planning framework was more straightforward and therefore Gazebo has been selected.

ROS packages used are mentioned throughout this chapter and their overview is provided in Appendix B. The ROS ecosystem evolves rapidly. Precompiled packages from the Hydro Medusa ROS release have been used predominantly but the crucial ROS Control, MoveIt! and Gazebo-related packages were compiled from source using the latest versions[1] from their Git[2] repositories. The main reason for this decision was the finding that – as with any fast evolving software – documentation of ROS packages is not always up to date and complete. We concluded that if problems occurr (for instance with configuration or setup), the easiest solution is often to debug or inspect the source code (and find out that the particular configuration XML attribute has been replaced with another one, for example).

When exploring and studying ROS Control and MoveIt! related ROS packages, it helps to be aware of the fact that these projects are "spin-off projects" of previously existing PR2 robot related solutions. This means that there are still the original PR2 specific packages available, which are very similar to the new and generic ones. At the same time, PR2 specific packages are not obsolete and are maintained in paralell. This causes confusion for newcoming users.

4.1 Robot Description

4.1.1 URDF, SDF and SRDF Description Formats

Many tools and components in the ROS ecosystem can use robot description stored in the XML-based "Unified Robotic Description Format" (URDF). URDF is useful file format for kinematics, collision detection and visualization. Xacro XML macro language can be used to make the XML files shorter and more readable.[3]

URDF captures the kinematic structure of the robot (its links and joins). For links, their inertial, visual and collision properties can be specified. Visual and collision properties of a link can be specified using one of three primitive shapes or using a reference to a mesh stored in a file (STL and COLLADA[4] file formats are supported). For joints, their type and geometrical properties can be defined together with eventual limits and optional dynamical properties. Supported joint types are: revolute (limited or continuous), prismatic, floating (6 degrees of

[1] Versions from February 2014 have been used.

[2] Git is open source distributed version control system used for most projects in the ROS ecosystem. It was originally created for Linux Kernel version control.

[3] These formats' and SRDF format's homepages are at `http://wiki.ros.org/{urdf,xacro,srdf}`.

[4] COLLADA (Collaborative Design Activity) is an open standard XML schema defining an interchange format for 3D applications: `https://collada.org/`.

freedom) and planar. Joint can be fixed too. URDF also supports transmission elements that are used to describe the relationship between a joint and its corresponding actuator.

Robot model defined in URDF can be used in several components, either directly or indirectly. The Rviz visualization tool can read URDF description of a robot it should visualize.

The Gazebo simulator described in the next section uses its own XML-based "Simulation Description Format" (SDF) but a tool from the supporting ROS package can convert URDF to SDF transparently. When used for simulation in Gazebo, however, the URDF-based robot description has to be extended with simulation-specific information in order to express relationships and parameters which can not be captured in URDF alone: references to plugins to be used with the model, parameters for these plugins and parameters for sensors supported by the simulator. The V-REP simulator, which uses its own binary format to store robot models, is distributed with an URDF importer plugin (this fact was found out at the last minute). This plugin can be a very useful tool; the imported model, however, needs further interventions before it is ready to use for simulation.

The MoveIt! motion planning framework described in Section 4.4 uses complementary "Semantic Robot Description Format" (SRDF). SRDF is used to define groups of links or joints that form an arm and to define end effectors. It allows to set up named poses for individual groups (arms) and it makes it possible to disable collision checking for selected pairs of links. MoveIt! provides a GUI[5] tool named Setup Assistant that simplifies the creation of SRDF description based on an existing URDF robot description.

A generator of fast analytical inverse kinematics solvers has been developed by Rosen Diankov and it is described in his doctoral dissertation thesis [3]. This "robot kinematics compiler" is named IKFast and it is part of the OpenRAVE suite.[6] IKFast can not use URDF direcly but again, URDF can be converted into the extended COLLADA format that can be processed by IKFast.[7]

4.1.2 HUBO Robot Model

At the beginning of this work, there was a vision of cooperation with Drexel University in Philadelphia, where they are operating HUBO humanoids.[8] We were therefore tempted to use HUBO as the simulated robot model. There was a HUBO robot model available with meshes and a controller setup stub at `https://github.com/wmhilton/hubo-urdf`. This model was, however, not ready to use. At least the controller setup was created as a custom Gazebo plugin and it did not work with the up to date simulator. This model has been taken as a "point of

[5] GUI – Graphical User Interface

[6] IKFast homepage is at `http://openrave.org/docs/latest_stable/openravepy/ikfast/`.

[7] IKFast uses modified COLLADA format with robot-specific extensions defined at `http://openrave.org/docs/latest_stable/collada_robot_extensions/`. Description of the process of generating inverse kinematics solver plugin for MoveIt! based on URDF robot description is available at `http://moveit.ros.org/wiki/Kinematics/IKFast`.

[8] Drexel University is participating in the DARPA Robotics Challenge: `http://www.drc-hubo.com/`.

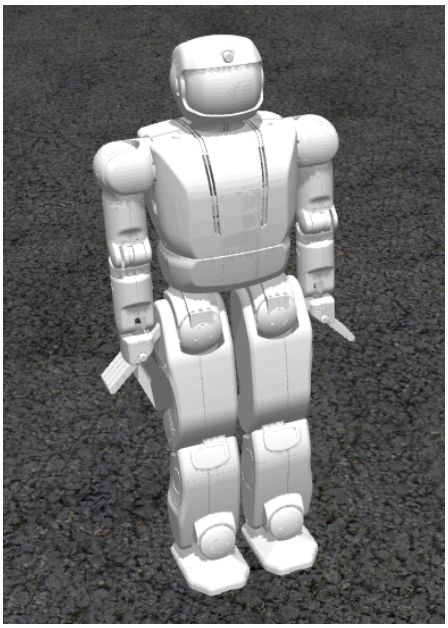


Figure 4.1: HUBO robot model with a sensor in its right hand.

departure" and research has been done to make it work in the Gazebo simulator and with the MoveIt! motion control framework. It has been updated and extended appropriately. A ROS Control plugin (Section 4.3) and a sensor model with an appropriate plugin (Section 4.5) have been added to the model . The HUBO robot model shown in the Gazebo simulator can be seen in Figure 4.1.

4.2 ROS Nodes and Their Interaction

Basic entities in the ROS runtime environment are *nodes*.[9] ROS nodes roughly correspond to processes of the underlying computer operating system and they communicate with each other via *topics* (message streams) and *services* (remote procedure calls). A ROS topic constitutes a named stream of *messages* of a particular type. Messages in ROS are strongly typed. Whereas messages on a topic are being exchanged on publish/subscribe basis, services provide request/reply type of interaction between nodes.[10] Services, like other entity types in the so-called *ROS computation graph*, are named. A service is defined by a pair of messages: one for the request and one for the reply.

For long running tasks which should be preemptable, services alone are not enough. ROS provides a standardized interface for such tasks in the *actionlib* package[11] that is built on top of

[9]Documentation of fundamental ROS related concepts is at `http://wiki.ros.org/ROS/Concepts`.

[10]The role of a "broker" and nameserver plays an unique type of node called ROS Master, which is usually part of the `roscore` executable described below.

[11]The actionlib homepage resides at `http://wiki.ros.org/actionlib`.

ROS messages. This package provides its user with means to create an *action server* and *action client*. Actionlib action specification includes three message types that are specific to a particular action type. The action client first sends a *goal message* to the action server. While fulfilling the request, the action server can emit *feedback messages* that notify the client about the progress towards the goal. Upon goal completion, a *result message* is sent from the action server to the action client. Apart from these three message types, there are two general message types: a *cancel message* can be sent to the action server in order to abandon the goal and a *status message* can be used to notify clients about all the goals tracked by a particular action server. In the simulation environment, that is covered in this chapter, the actionlib interface is used to trigger execution of trajectories created by the motion planner on the simulated robot.

In ROS environment, on a single computer there has to always be a single `roscore` executable running. This executable starts up the ROS Master node which takes care of the communication between other nodes. In addition to that, `roscore` starts up a logging node and a *parameter server*

4.2.1 ROS Nodes Used in the Simulation Environment

The single most important node in the simulation environment is of course the robotic application itself. Apart from it, in our simulation environment there are simulation related nodes and the MoveIt! `move_group` node. In the overview below, the first name corresponds to the ROS node name used. The name in the parenthesis corresponds to ROS package (before the slash) and to the name of the executable (after the slash).

Simulation related

- `gazebo` – Section 4.5,
- `urdf_spawner` (`gazebo_ros/spawn_model`) – Section 4.5,
- `controller_spawner` (`controller_manager/spawner`) – Section 4.3.

Motion Planning related

- `move_group` (`moveit_ros_move_group/move_group`) – Section 4.4.

4.3 ROS Control Controllers

ROS Control project exists that aims to provide a set of standard ROS packages for the actuator control.[12] This section describes the configuration of the simple setup that has been

[12]Packages from the ROS Control project are a rewrite of the PR2-specific `pr2_mechanism` "ROS stack".

prepared. In our setup, ROS Control based controllers are running in a plugin[13] in the Gazebo simulator.

We have used `joint_trajectory_controller::JointTrajectoryController` controller that is capable of executing joint-space trajectories on a group of joints.[14] It is a templated class that is capable of working with both position and effort actuator interfaces and it also provides different spline interpolation strategies for different waypoint specifications.

`JointTrajectoryController` supports both ROS topic based set-point specification and action based interface. The action based interface is defined by the `control_msgs::FollowJointTrajectoryAction` class and is the superior one.

YAML-based[15] configuration for the HUBO robot is shown in Listing 4.1.

```
1  huboplus:
2
3    # Publish all joint states -----------------------------------
4
5    joint_state_controller:
6      type: joint_state_controller/JointStateController
7      publish_rate: 50
8
9    # Trajectory Controllers --------------------------------------
10
11   right_hand_controller:
12     type: effort_controllers/JointTrajectoryController
13     joints: [RSP, RSR, RSY, REP, RWY, RWP]
14     gains:
15       RSP: {p: 2400, i: 800, d: 18}
16     # RSR: ... gains for the other joints are set likewise
17     # RSY: ...
18     # ...
19
20   # left_hand_controller would contain similar configuration;
21   # if a position-based actuator interface is used,
22   # gains are not specified:
23   left_hand_controller:
24     type: position_controllers/JointTrajectoryController
25     joints: [LSP, LSR, LSY, LEP, LWY, LWP]
```

[13] It is the `gazebo_ros_control` plugin from the `gazebo_ros_pkgs` package mentioned in the Simulator Section.

[14] `http://wiki.ros.org/joint_trajectory_controller`

[15] The YAML format is a "human friendly data serialization standard". It is a generalization of JSON. For details, see `http://www.yaml.org/`

Listing 4.1: YAML configuration file of ROS Control controllers

Controller Spawner

Controllers are loaded using a "controller spawner" script from the `controller_manager` subpackage of the `ros_control` ROS package. We realized that when used with the Gazebo simulator, the Gazebo simulator needs to be started in paused state; otherwise controllers do not get loaded properly.

4.4 Motion Planning

For motion planning, we had considered the MoveIt! motion planning framework and the built-in motion planning functionality of the V-REP simulator. We believed that the motion planning API provided by the V-REP simulator is not rich enough. On the other side we considered MoveIt! providing a standalone solution independent of a simulator and – being under active development and being one of prominent (open source) projects in the ROS ecosystem – we believed it had a bigger potential.

MoveIt!, presented in [13], is a framework that strives to incorporate motion planning, manipulation, 3D perception, kinematics, control and navigation functionality. It provides a high-level interface to individual underlying libraries which focus on particular motion planning related tasks and it makes it possible to change these underlying implementations when needed. For kinematic description of the robot and collision avoidance it needs URDF input. For definition of motion planning related units it needs SRDF, which refers to the existing URDF description. URDF and SRDF file formats have been described in Section 4.1.1

Plugin for the Rviz visualization tool mentioned in Section 5.1 exists that makes it possible to test the MoveIt! setup interactively and issue motion planning or motion execution requests from a GUI.

The MoveIt! Configuration Package

The MoveIt! configuration for a particular robot usually resides in a dedicated ROS package. The configuration we created for the HUBO robot is in the `hubo_moveit_config` package.[16] (The `_moveit_config` suffix is a usual convention.) The structure of this package is in the Listing C.1 in Appendix C.

[16] The `hubo_moveit_config` package is available on the attached CD.

The YAML configuration files in the `config` subdirectory are actually used only to set up appropriate parameters on the ROS Parameter server[17]. The launch files[18] in the `launch` subdirectory take care of this parameter setup process.

The `move_group` ROS node

The functionality of the MoveIt! framework is exposed to other ROS nodes via services provided by the `move_group` ROS node. Its interfaces are shown in Figure 4.2.

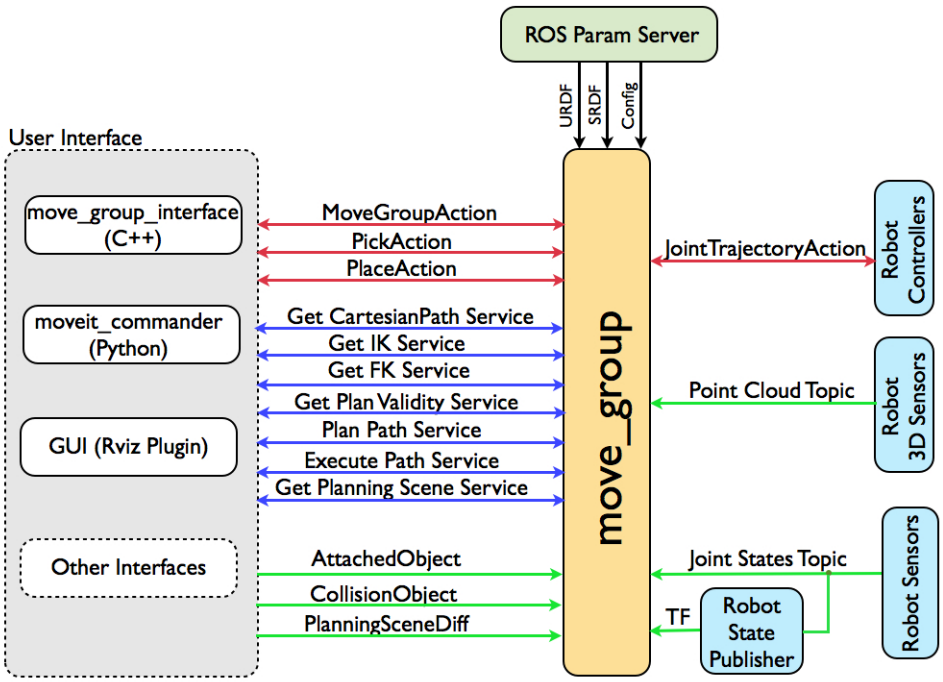


Figure 4.2: MoveIt! `move_group` ROS node interfaces. Note the JointTrajectoryAction actionlib-based interface between the `move_group` ROS node and robot controllers. This interface is used for trajectory execution. Source: `http://moveit.ros.org`.

Trajectory Execution (`MoveItControllerManager`) Configuration

When connected to the controllers on the robot, MoveIt! can take care of the trajectory execution too. This is responsibility of the Trajectory Execution Manager. The connection to the

[17] Another component of the `roscore` executable. It is used for (dynamic) configuration. Described at `http://wiki.ros.org/Parameter%20Server`

[18] Launch files are XML files used for configuration and launching of ROS nodes and ROS Param parameters. For details see `http://wiki.ros.org/roslaunch`

controllers is handled by subclasses of the `moveit_controller_manager::MoveItControllerManager` class defined in the `moveit_core`[19] ROS package.[20]

There are two default `MoveItControllerManager` implementations available. Both are in the `moveit_plugins` ROS package. The `MoveItFakeControllerManager` only publishes on a ROS topic the joint state of the last point of the trajectory requested for execution. This controller manager is useful for testing. The second controller manager available is `MoveItSimpleControllerManager` and it is supposed for real execution of trajectories on a robot that is using controllers providing an action-based interface[21]. (The ROS Control Project provides such controllers in the `ros_controllers` ROS package.)

The configuration of both the controller managers from the `moveit_plugins` ROS package (i.e. the parameters expected on the ROS Parameter server) is best documented in the source code of these controller managers. The controller name and namespace setup needs special care on both the ROS Control and MoveIt! side. Configuration used in our simulation environment is shown in Listing 4.2. The `"follow_joint_trajectory"` string is the name of the action (determined by the controller); `"huboplus"` is the name of the robot; `"huboplus/xyz_controller"` are names of the individual `FollowJointTrajectoryAction`-based controllers; joint names correspond to the joint names defined in the URDF.

```
1  controller_list:
2    - name: huboplus/left_hand_controller
3      action_ns: follow_joint_trajectory
4      type: FollowJointTrajectory
5      joints: [LSP, LSR, LSY, LEP, LWY, LWP]
6    - name: huboplus/right_hand_controller
7      action_ns: follow_joint_trajectory
8      type: FollowJointTrajectory
9      joints: [RSP, RSR, RSY, REP, RWY, RWP]
10   - name: huboplus/legs_controller
11     action_ns: follow_joint_trajectory
12     type: FollowJointTrajectory
13     joints: [LHY, LHR, LHP, LKP, LAP, LAR,
14              RHY, RHR, RHP, RKP, RAP, RAR]
```

Listing 4.2: YAML configuration file for the MoveItSimpleControllerManager

[19] See Appendix B.

[20] Note that there is an entity named "controller manager" on the ROS Control side too. This is a different one.

[21] The actionlib library that implements this interface was mentioned in Section 4.2. `MoveItSimpleControllerManager` supports two kinds of action interfaces: `control_msgs::FollowJointTrajectoryAction` and `control_msgs::GripperCommandAction`.

Inverse Kinematics Solver

The default MoveIt! IK solver is KDL[22] based. A different solver can be used. IKFast solver generator from the OpenRAVE toolset has been used to generate analytical IK solver for the right hand of the HUBO robot, which was used successfully.

Motion Planner

The default motion planning library is the Open Motion Planning Library (OMPL) implemening abstract randomized motion planners. OMPL is described by its authors in [14] and it is nicely documented on its webpage[23].

Collision Detection

FCL, the Flexible Collision Library, is the primary collision detection library used in MoveIt! It supports collision checking for various object types: meshes, primitive shapes and OctoMap.

During the experiments, the OctoMap managed by the MoveIt! framework has been used for collision detection. In the future we need to find out how to use single shared map for both the exploration algorithm and collision detection library.

4.5 Simulator

In this work, two robotic simulators have been considered: V-REP[24] (version 3.0.4) and Gazebo[25] (version 1.9.3).[26] This section attempts to compare them shortly. Emphasis is being put on Gazebo which has been selected for the simulation. Configuration of Gazebo plugin that handles joint control is also described.

V-REP is easy to use. It has a rich graphical user interface. The GUI and the built-in scripting capability provide two ways to modify different aspects of the simulation. V-REP includes many features and computation modules, most of which can be accessed programatically. The API is, however, influenced by the GUI-nature of V-REP and the user needs to manipulate objects in a very similar way he would in the GUI. V-REP includes inverse kinematics solver and motion planning functionality (built on top of the Reflexxes Motion Library) but these computation modules did not meet all our requirements. We did not use the built-in motion planning functionality because it is highly coupled with the simulation and visualization and we

[22] The Orocos Kinematics and Dynamics Library found at `http://www.orocos.org/kdl`.

[23] `http://ompl.kavrakilab.org/`

[24] `http://http://www.coppeliarobotics.com/`

[25] `http://gazebosim.org/`

[26] As of this writing, there is another promising simulator available: the Modular OpenRobotics Simulation Engine (MORSE): `http://www.openrobots.org/wiki/morse/`

were not able to use it for "off-line" computations (to compute a plan without affecting the state of the simulation).

Gazebo has different philosophy and a client-server model. The simulation runs on the server and the client (GUI) only provides means for a basic scene view manipulation. Simulated objects can be moved around using the GUI too, but besides that, all the simulation configuration is done in configuration files. There is no built-in scripting support and the main way to affect the simulation is through plugins using the avaliable C++ API.[27]

We have used Gazebo 1.9.3. Historically, Gazebo started as a 3D simulator for the Player Project[28] (a predecessor of ROS) and when used together with ROS, a specific version of Gazebo had to be used. Today, the Gazebo is a stand-alone package with no dependencies on Player or ROS. As of this writing, the latest version of Gazebo is 3.0.0, but the latest version compatible with ROS Hydro is 1.9. A set of interfaces for using Gazebo in ROS exists in the `gazebo_ros_pkgs` package.[29] This package integrates the simulator with ROS messages and services; provides support for the URDF file format described in Section 4.1.1; and makes it possible to use ROS Control controllers mentioned above together with Gazebo simulated robot.

Worlds, Models and Their Instantiation

Gazebo worlds and models are stored in SDF files. Although it is technically possible to include models into world files, it is not a good solution. It is possible to insert a model into an existing world at runtime. This can be done by the `spawn_model` utlity from the `gazebo_ros_pkgs` package.

Range Imaging Sensor

We have used the `gazebo_ros_prosilica` Gazebo plugin from the `gazebo_ros_pkgs`/`gazebo_plugins` ROS package that produces point cloud messages of the `sensor_msgs`/`PointCloud2` message type.

An useful library for point cloud manipulation and processing is the Point Cloud Library.[30] MoveIt! can be set up to process point cloud messages and it is able to integrate incoming `sensor_msgs/PointCloud2` messages in its built-in OctoMap which is used for collision checking.

[27] `http://gazebosim.org/api/1.9.1/index.html`

[28] Player Project homepage is at `http://playerstage.sourceforge.net/`.

[29] Its homepage is at `http://wiki.ros.org/gazebo_ros_pkgs`. Individual components of the package are described in Appendix B.

[30] `http://pointclouds.org/`

4.6 Conclusion

It turned out that the simulation environment is rather computationally expensive. Running the simulation on a PC with Intel Core2 Duo CPU running at 2.1 GHz with 4 GB of RAM needs some amount of patience.

The motion planning with MoveIt! and the Rviz plugin exhibited some issue. Sometimes it happended that an invalid, possibly infinite, motion plan has been created. We don't know whether this is an issue of the Rviz plugin only or whether this is an issue of the motion planner returning an invalid solution from time to time.

Chapter 5

Experiments

5.1 Overview of Visualization Tools

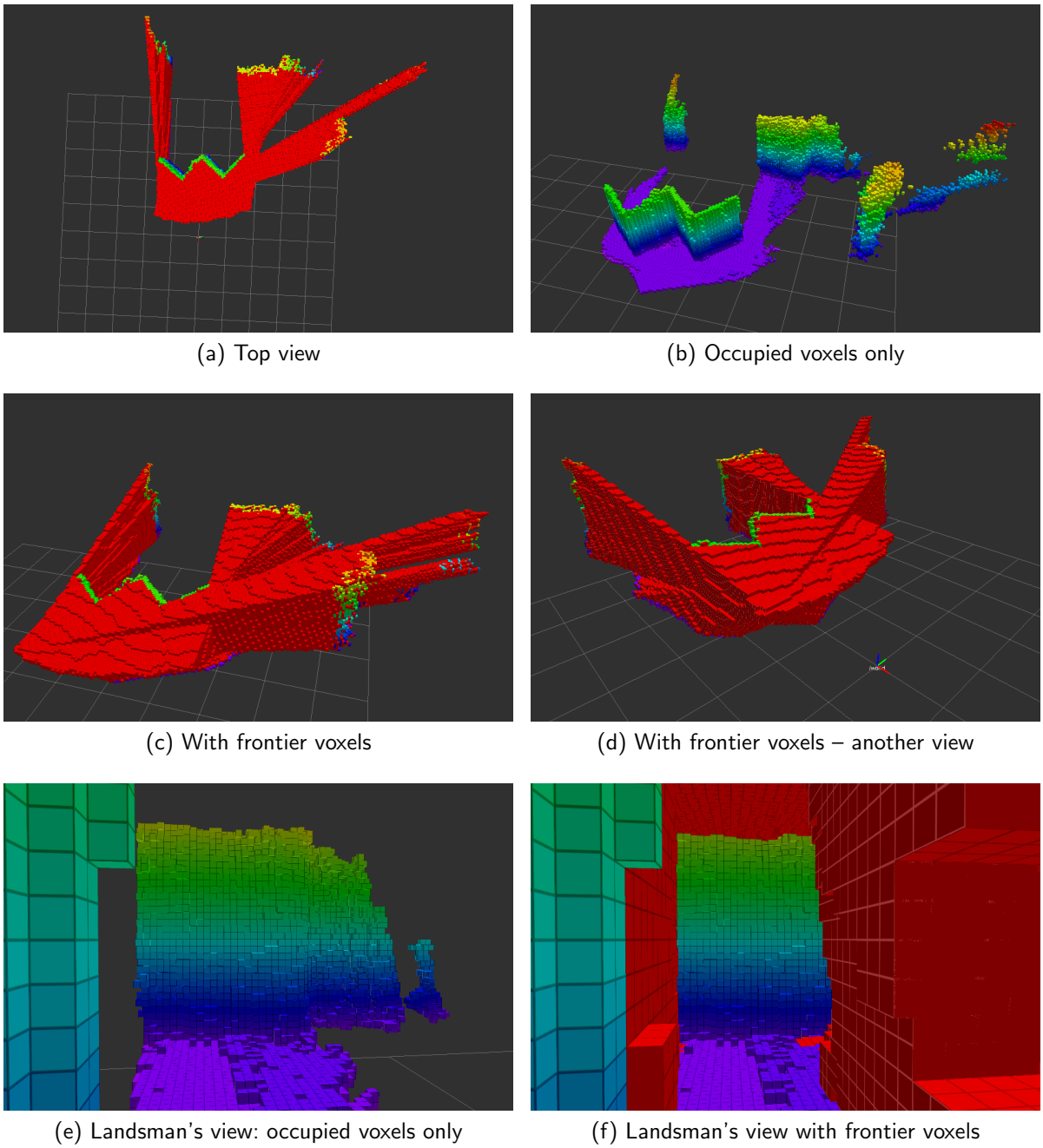
A visual feedback is often needed while debugging algorithms involving geometrical entities. This section lists tools worth inspecting when a need arises to produce some visualizations. Some of these tools were used during the development, some were used to prepare pictures shown in this thesis.

A classic tool is the powerful Gnuplot[1] graphing utility.

Asymptote[2] is a high level, mathematically oriented, vector graphics language for technical drawing inspired by the METAPOST programming language (which had been in turn inspired by Donald Knuth's METAFONT language) but with a cleaner C++-like syntax. Asymptote has been used to produce pictures of reachability structures shown in Figure 3.5, but its potential is better shown on its web page. It uses LaTeX to typeset labels and equations.

Rviz[3] is the visualization tool number one in the ROS environment, capable of an out of the box visualization of many message types available in the environment including coordinate frames, point clouds, laser scans, 2D occupancy grids, polygons, images and other entities. Rviz can also display an URDF-based robot model. ROS message types dedicated primarily for visualization are also available: so-called Markers can be used to visualize basic shapes. Even more powerful options gives the library of Interactive Markers[4], providing means to GUI-based control. Moreover, various packages and libraries provide their own plugins for Rviz. Notable example is the motion planning plugin from the MoveIt! framework, which allows, among other things, an interactive definition of motion planning goals (using the above mentioned Interactive Markers) and visualization of resulting motion plans. Pictures in Figures 3.1, 3.2 and 5.1 have

[1] `http://www.gnuplot.info/`
[2] `http://asymptote.sourceforge.net/`
[3] `http://wiki.ros.org/rviz`
[4] `http://wiki.ros.org/interactive_markers`



(a) Top view

(b) Occupied voxels only

(c) With frontier voxels

(d) With frontier voxels – another view

(e) Landsman's view: occupied voxels only

(f) Landsman's view with frontier voxels

Figure 5.1: Frontier voxels (in red color) before clustering. Data courtesy of [12].

been acquired in Rviz. Visualizations of the HUBO robot in Figure 5.8 have been created in Rviz with the motion planning plugin.

The OctoMap library provides a viewer called Octovis that can be used for basic OctoMap visualization and inspection. Currently, it is only capable of visualization of an OctoMap loaded from a file. Example of pictures produced by Octovis are in Figure 3.6.

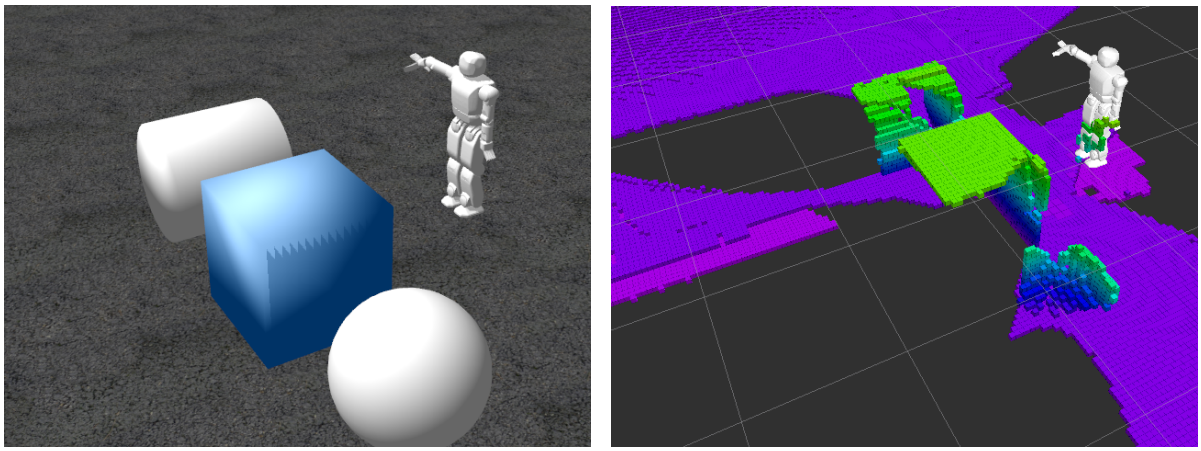
A ROS package providing an `octomap_server::OctomapServer` class exists.[5] This class can publish ROS messages describing the current state of a given OctoMap and these messages can be used for visualization in Rviz. In this work, the `OctomapServer` has been modified in order to allow visualization of OctoMaps given as a pointer to the more general `octomap::OccupancyOcTreeBase<NODE>` class template instead of `octomap::OcTree`. The modified server allows visualization of OctoMaps carrying utility value in free voxels.

A `gzclient` executable distributed with the Gazebo simulator provides a GUI capable of scene visualization. The Figure 4.1, showing the HUBO robot model, presents a screenshot acquired in Gazebo.

The V-REP simulator also has a GUI and it has a rich API, which can be easily used to add objects to a scene. This has been used to visualize reachability spheres in Figure 3.3 and reachability index in Figure 3.4.

5.2 Creating a Map From a Simulated Sensor

Figure 5.2 shows an OctoMap created from the simulated sensor data.



(a) Gazebo simulated scene (b) OctoMap

Figure 5.2: OctoMap world model created from simulated sensor data. Color maps to height.

[5] `http://wiki.ros.org/octomap_server`

5.3 Frontier Visualization

Figure 5.1 shows frontier voxels before clustering. Compare Figures 5.1c and 5.1d with Figure 3.1d. Please note that the set of frontiers is hollow (Figure 5.1f). Only frontier voxels inside the convex hull of occupied voxels in the map are used as "sources of utility" in the "next best view computation" phase of the frontier-void-based exploration algorithm.

5.4 Frontier and Void Clusters in a Map

The Figure 5.3 shows void voxels as yellow dots. It is a visualization of void voxels, initial ellipsoids before merging and resulting void clusters and it was computed from the first scan of the scene in Figure 3.1. Two resulting void clusters can be seen in Figure 5.3d.

Frontier clusters are harder to visualize in a static image. Four different frontier clusters for the same scan from the first scene are shown in Figure 5.4.

5.5 Utilities of Free Voxels

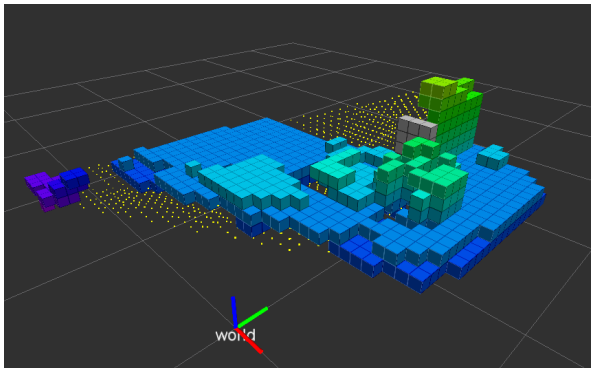
Visualization of free voxels' utilities for the scene from Figure 5.1 is shown in Figure 5.7.

5.6 Capability Maps

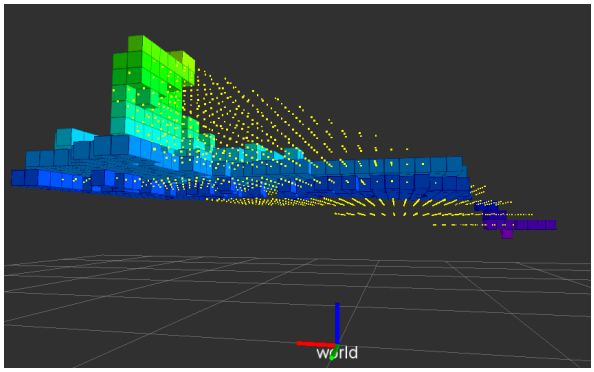
Capability maps were described in Sections 3.2 and 3.3.2. A visualization of reachability spheres for a manipulator[6] with 7 degrees of freedom is shown in Figure 3.3. (The manipulator itself is better seen in Figure 3.4, together with spheres showing reachability index values in two different cross-cuts of the manipulator workspace.)

The reachability spheres shown in Figure 3.3 exhibit "baldnesses" in places where this was not expected. For reachability spheres near to the manipulator base this can be better seen in Figure 5.6: whereas the "spikes", which represent valid points on the reachability sphere, are expected to resemble a cone-like structure, in our pictures there are spikes missing. And although the manipulator itself is symmetric, the reachability spheres are not. (Note, however, that this could be attributed to the fact that the points on reachability spheres were generated using the spiral point algorithm and therefore they are not laterally symmetrical.) Another undesirable effect can be seen in Figure 3.4, especially in the second picture. In this case the reachability index is expected to be symmetrical around the axis of the first joint of the manipulator and in the picture it is not.

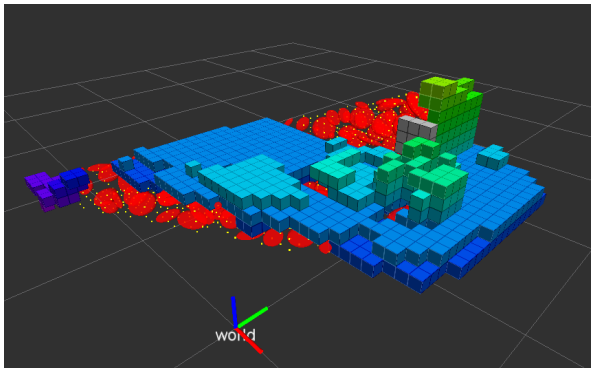
[6]Although the robot model had been available in the library of the V-REP simulator, we had to perform several modifications in order to be able to compute inverse kinematics and make collision checking possible.



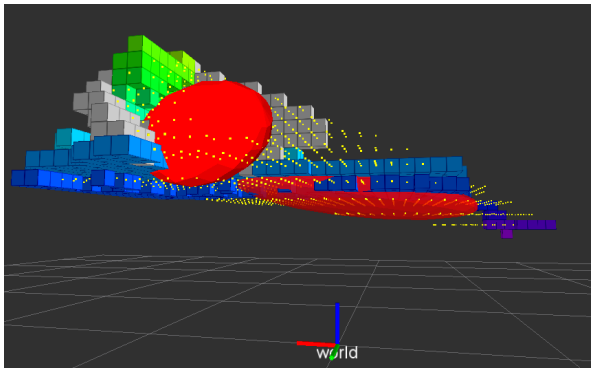
(a) Void voxels – view 1



(b) Void voxels – view 2

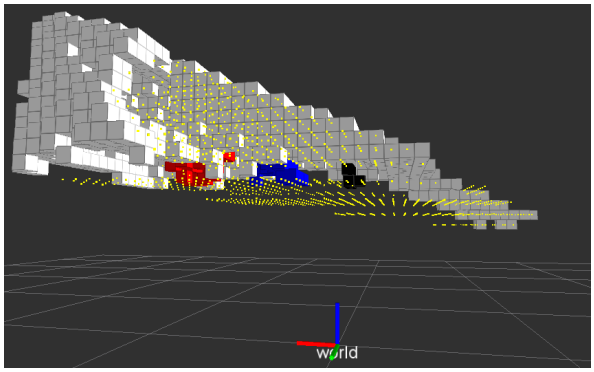


(c) Initial ellipsoids

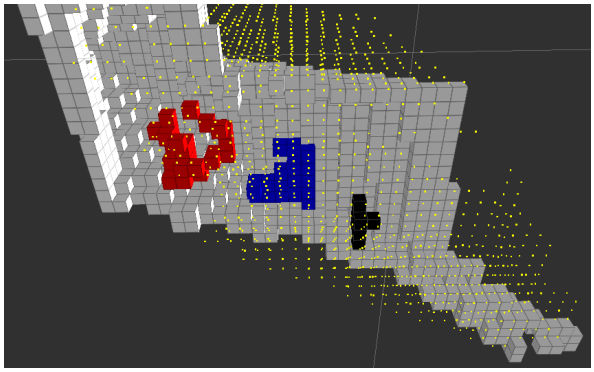


(d) Resulting void clusters

Figure 5.3: Void voxels, inital ellipsoids and resulting void clusters: four views on the scene from Fig. 3.1. The visualization corresponds to the first acquired scan of the scene. Occupied cells shown as color cubes. Void voxels' centers shown as yellow dots.

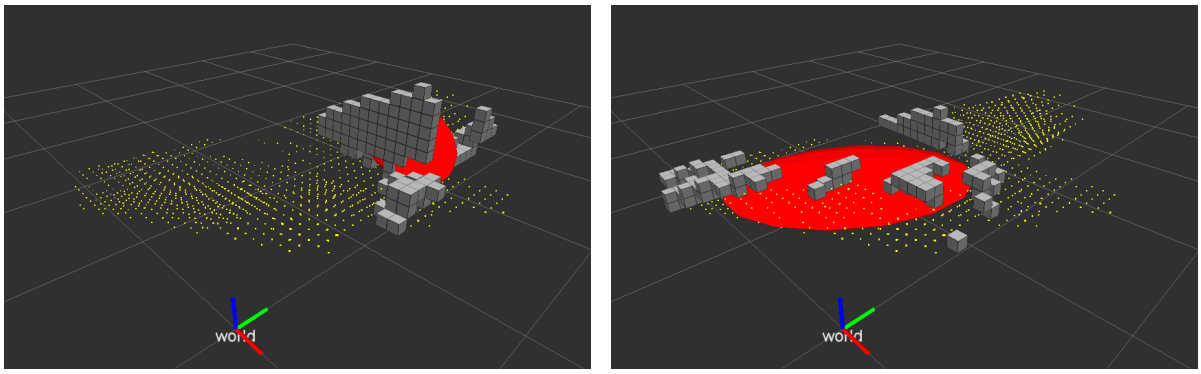


(a) Frontier clusters – view 1



(b) Frontier clusters – view 2

Figure 5.4: Four disjunct frontier clusters in the same scene as is shown in Figure 5.3. Occupied voxels are not shown. Void voxels' centers are shown as yellow dots. The view in (a) is the same as in Subfigures 5.3 (b) and (d).



(a) Frontier-Void set 1 (b) Frontier-Void set 2

Figure 5.5: Visualization of two frontier-void sets created from the void clusters shown in Figure 5.3d and frontier clusters shown in Figure 5.4. The view in both pictures corresponds to the view in Figure 5.3c.

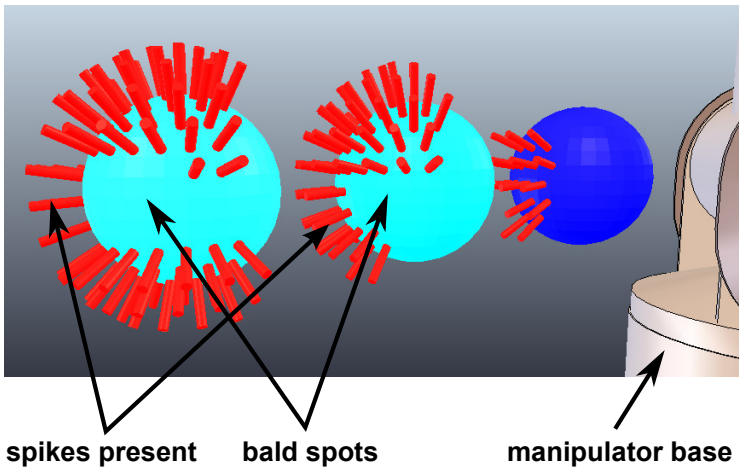
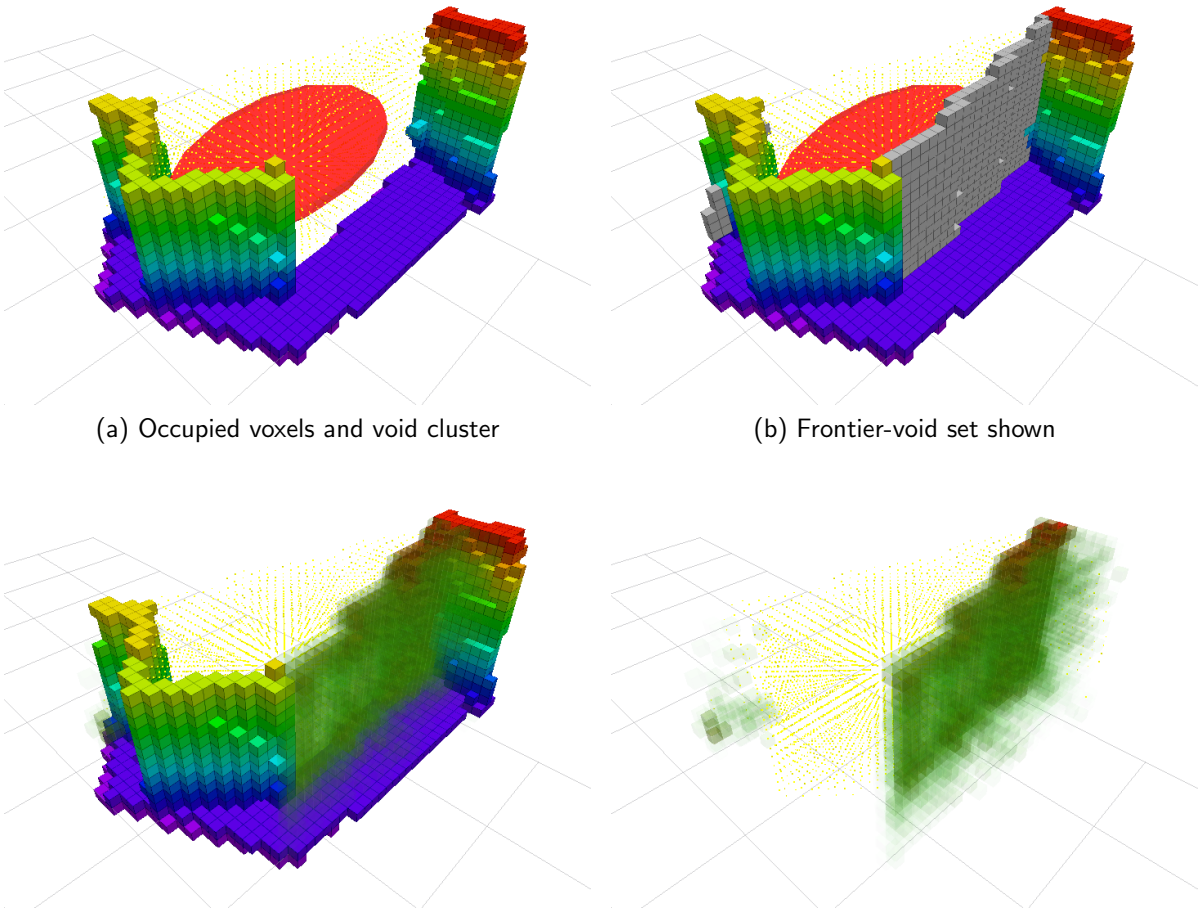


Figure 5.6: "Bald" reachability spheres



(a) Occupied voxels and void cluster

(b) Frontier-void set shown

(c) With utilities of free voxels

(d) Utilities only

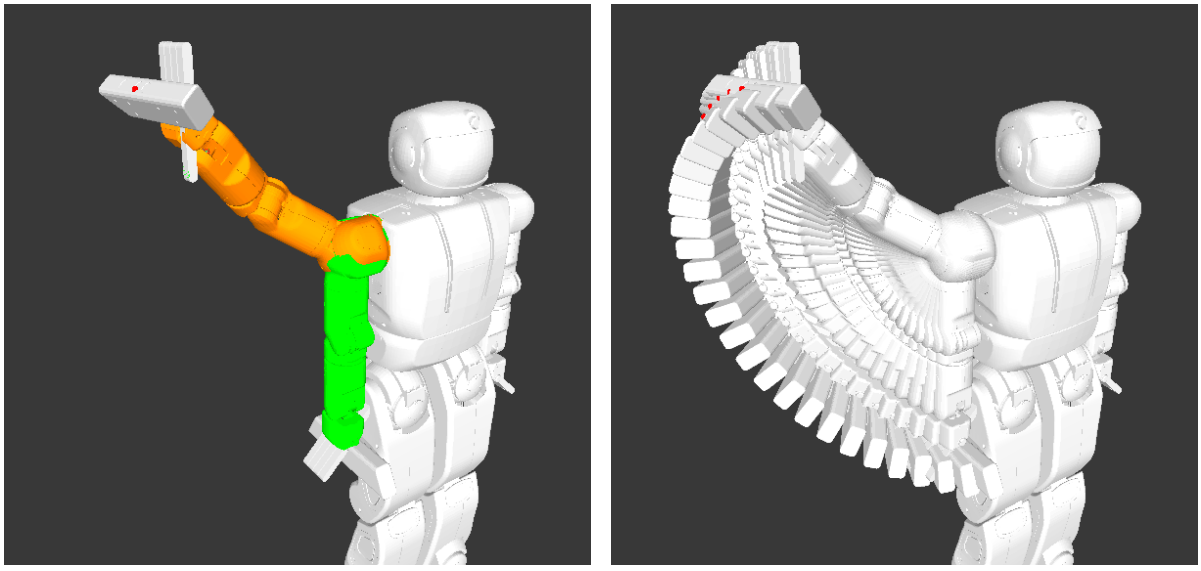
Figure 5.7: Visualization of utilities in free voxels. Data correspond to the first scan of the scene shown in Figure 5.1. In the Figure (b) showing the frontier-void set, frontier voxels are grey. Free voxels are greenish and translucent. Free voxels with higher utilities have reddish color and are more opaque.

We attribute these properties of the obtained reachability spheres to the instability of the numerical inverse kinematics solver used (the built-in IK solver from the V-REP simulator) and explain the behavior of the solver by its dependence on the initial condition. In the future, we would like to employ the solver generated by the IKFast "robot kinematics compiler" by Rosen Diankov, which is supposed to be an analytical one.

While experimenting with the V-REP simulator, an attempt has been made to compute capability map (reachability spheres) for the right arm of the HUBO robot model (having 6 degrees of freedom). We were not able to compute the capability map for HUBO's right hand, because the IK solver exhibited crucial instabilities. Another attempt has been made with the manipulator present on the KUKA youBot model, which has 5 degrees of freedom. For this manipulator, the solver could not find a solution for any of the points on any reachability sphere. We hope this will be resolved by using the IKFast-generated solver.

5.7 Motion Planning

Figure 5.8 shows an example visualization of a motion plan generated for the right arm of the HUBO robot using the MoveIt! motion planning plugin for Rviz.



(a) Start and goal positions (b) Motion plan visualization

Figure 5.8: Visualization of a motion plan for the right arm of the HUBO robot. Start and goal positions are shown in green and orange colors respectively. (Snapshot from Rviz with the MoveIt! plugin.)

Chapter 6

Conclusion

In this thesis we described the Frontier-Void-Based Exporation Algorithm that represents a natural extension of frontier-based algorithms into three dimensions. The exploration problem and alternative approaches to the solution were shortly presented in Chapter 2. The choice of the algorithm and the Frontier-Void-Based Exploration Algorithm itself together with a capability mapping technique allowing precomputation of capabilities of a robotic manipulator were described in Chapter 3. The ROS-based simulation environment that has been prepared for execution of experiments was described in Chapter 4 together with findings that could not be found in the documentation. The results were discussed in Chapter 5 and some examples and illustrations were presented there.

The core components of the Frontier-Void-Based Exploration Algorithm have been implemented in C++ with employment of the OctoMap 3D occupancy grid mapping library. Also, a C++ capability mapping library has been implemented which makes it possible to precompute capabilities of a given robotic manipulator and store them for later use. This makes it possible to avoid many computationally expensive inverse kinematics queries during the phase of motion planning goal selection. A simulation environment setup based on components and tools provided by the ROS "ecosystem" has been prepared which can be used to run the exploration in a simulated environment. The simulation environment includes a HUBO robot model which had been already available but which had to be updated and extended to make it compatible with the up to date simulator and actuator controllers from the ROS Control package. During the work patches to several open source projects have been submitted and other bug fixes are planned to be published later.

The frontier-void-based exploration approach promises a real-world capability which is, however, yet to be proved. The integration of the exploration algorithm with the capability mapping functionality anticipates an exploration method that takes into account the kinematical constraints of the robot.

The article originally describing the Frontier-Void-Based Exploration Algorithm indicates that its authors performed experiments with a still standing mobile robot and carried out motion planning for the attached manipulator only. We did not get any further either, because a con-

siderable effort had to be put into the preparation of the simulation environment and a study of available tools. The simulation environment had been initially expected to be available and ready for use, which was not the case.

Several weak spots in the current implementation have been identified. The bottleneck of the current frontier-void-based algorithm implementation is the naïve ellipsoid merging algorithm used, which could be improved. Also, during the construction of sets of frontier and void voxels and their clustering, iterative scan integration could be employed. The OctoMap library has been extended to allow incorporation of the utility of free voxels into the map. This extension should be revisited and redesigned, because it is not correct under all conditions (pruning of the map would discard the utility). The last known design issue that should be resolved in the future is the integration of the extended OctoMap used in the exploration algorithm with the motion planning framework, which needs a map of the environment for collision checking. This last point is expected to have a straightforward solution, because the MoveIt! motion planning framework used is already capable of working with OctoMaps. The possible design has to be, however, investigated.

During the experiments with the resulting simulation environment, weak spots of some of the software tools have been identified and they have been described in this work. Not surprisingly, the simulation environment turned out to be rather computationally expensive; for convenience, a decent machine should be used to perform the experiments.

Bibliography

[1] *Algorithms for calculating variance on Wikipedia, section on covariance*. Jan. 2014. URL: http://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Covariance.

[2] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. "The Quickhull algorithm for convex hulls". In: *ACM TRANSACTIONS ON MATHEMATICAL SOFTWARE* 22.4 (1996), pp. 469–483.

[3] Rosen Diankov. "Automated Construction of Robotic Manipulation Programs". PhD thesis. Carnegie Mellon University, Robotics Institute, Aug. 2010. URL: http://www.programmingvision.com/rosen_diankov_thesis.pdf.

[4] C. Dornhege and A. Kleiner. "A Frontier-Void-Based Approach for Autonomous Exploration in 3D". In: *Safety, Security, and Rescue Robotics (SSRR), 2011 IEEE International Symposium on*. Nov. 2011, pp. 351–356. DOI: 10.1109/SSRR.2011.6106778.

[5] I. Dryanovski, W. Morris, and Jizhong Xiao. "Multi-volume occupancy grids: An efficient probabilistic 3D mapping model for micro aerial vehicles". In: *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. Oct. 2010, pp. 1553–1559. DOI: 10.1109/IROS.2010.5652494.

[6] T. Krajník. *PhD thesis proposal: Large-scale mobile robot navigation and map building*. 2009.

[7] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*. 2009.

[8] EA Rakhmanov, EB Saff, and YM Zhou. "Minimal discrete energy on the sphere". In: *Math. Res. Lett* 1.6 (1994), pp. 647–662.

[9] R. Shade and P. Newman. "Choosing where to go: Complete 3D exploration with stereo". In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. May 2011, pp. 2806–2811. DOI: 10.1109/ICRA.2011.5980121.

[10] Shaojie Shen, Nathan Michael, and V. Kumar. "Autonomous indoor 3D exploration with a micro-aerial vehicle". In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. May 2012, pp. 9–15. DOI: 10.1109/ICRA.2012.6225146.

[11] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *Disjoint Sets Boost Class*. Jan. 2014. URL: http://www.boost.org/doc/libs/1_55_0/libs/disjoint_sets/disjoint_sets.html.

[12] J. Sturm et al. "A Benchmark for the Evaluation of RGB-D SLAM Systems". In: *Proc. of the International Conference on Intelligent Robot Systems (IROS)*. Oct. 2012.

[13] Ioan A. Şucan and Sachin Chitta. *MoveIt! Project Web Page*. URL: http://moveit.ros.org.

[14] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. "The Open Motion Planning Library". In: *IEEE Robotics & Automation Magazine* 19.4 (Dec. 2012). http://ompl.kavrakilab.org, pp. 72–82. DOI: 10.1109/MRA.2012.2205651.

[15] Robert E. Tarjan and Jan van Leeuwen. "Worst-case Analysis of Set Union Algorithms". In: *J. ACM* 31.2 (Mar. 1984), pp. 245–281. ISSN: 0004-5411. DOI: 10.1145/62.2160. URL: http://doi.acm.org/10.1145/62.2160.

[16] K. M. Wurm et al. "OctoMap: A Probabilistic, Flexible, and Compact 3D Map Representation for Robotic Systems". In: *Proc. of the ICRA 2010 Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation*. Software available at http://octomap.sf.net/. Anchorage, AK, USA, May 2010. URL: http://octomap.sf.net/.

[17] B. Yamauchi. "A Frontier-Based Approach for Autonomous Exploration". In: *Computational Intelligence in Robotics and Automation, 1997. CIRA'97., Proceedings., 1997 IEEE International Symposium on*. 1997, pp. 146–151. DOI: 10.1109/CIRA.1997.613851.

[18] Tsuneo Yoshikawa. *Foundations of Robotics: Analysis and Control*. Cambridge, MA, USA: MIT Press, 1990. ISBN: 0-262-24028-9.

[19] Franziska Zacharias, Christoph Borst, and Gerd Hirzinger. "Capturing robot workspace structure: representing robot capabilities". In: *IROS*. IEEE, 2007, pp. 3229–3236.

Appendix A

Content of the Attached CD

Directory name	Description
`src`	Source code with the implementation.
`sim`	Simulation environment configuration, robot model.
`data`	Other data (V-REP scenes and models, precomputed reachability spheres)
`thesis.pdf`	This Diploma thesis in PDF format.
`README.md`	Text file with detailed description of the CD Content (Markdown text format, UTF-8).

Table A.1: CD Content

Appendix B

Overview of Selected MoveIt!, ROS Control and Gazebo Related Packages

Robot Description Package https://github.com/ros/robot_model

robot_model — Provides packages that work with robot model: e.g. URDF parsers, URDF to COLLADA converter, a ROS node (joint_state_publisher) that makes itpossible to set or publish joint state values of a robot described in a URDF model.

Gazebo ROS Interface https://github.com/ros-simulation/gazebo_ros_pkgs

gazebo_ros_pkgs — Wrappers, tools and additional API's for using ROS with the Gazebo simulator.

- gazebo_ros

  Most importantly contains wrapper start up scripts and ROS launch files that should be used to start Gazebo when used with ROS.

- gazebo_ros_control

  A ROS package for integrating the ROS Control controller architecture with the Gazebo. Contains a Gazebo plugin which instantiates a ROS Control Controller Manager and connects it to a Gazebo model.

gazebo_plugins

A set of Gazebo plugins for sensing (cameras, bumpers, innertial measurement unit, laser sensors) and simple model control (e.g. differential drive). We have used the gazebo_ros_prosilica plugin for sensing.

ROS Control Packages https://github.com/ros-controls

ros_control — A set of packages that include controller interfaces, controller managers and loaders, transmissions, hardware_interfaces and the control_toolbox.

ros_controllers — Concrete controller implementations.

control_toolbox — Control modules used in controllers.

control_msgs — Messages and actions for robot control.

Motion Planning (MoveIt!) Packages https://github.com/ros-planning

moveit_core — Core libraries used by MoveIt!

moveit_planners — Interfaces for the motion planning libraries used in MoveIt!

moveit_plugins — Simple implementations of MoveItControllerManager plugin useful for trajectory execution.

moveit_simple_controller_manager

Can connect to FollowJointTrajectoryAction and GripperCommandAction servers.

moveit_fake_controller_manager

Controller Manager that just publishes the joint state of the last point of the required

trajectory on a ROS topic. Useful for testing.

moveit_ros — MoveIt! components that use ROS built on top of the other MoveIt! packages. This package contains the 'move_group' ROS node, the motion planning plugin for Rviz, and tools for visualizations, benchmarking and planning.

moveit_ikfast — Generator of IK solver plugins for MoveIt! using .cpp files generated by the IKFast 'robot kinematics compiler' from the OpenRAVE suite.

moveit_setup_assistant

MoveIt! Setup Assistant -- GUI tool assisting SRDF creation based on an existing URDF model.

moveit_robots — A library of MoveIt! configuration files for various robotic platforms.

Appendix C

MoveIt! Configuration Package Structure

```
config/
    controllers.yaml            Configuration for the
                                MoveItSimpleControllerManager.
                                (Structure of the configuration tree
                                is specific to this controller manager
                                type.)

    fake_controllers.yaml       Alternative controller manager for the
                                MoveItFakeControllerManager which just
                                discards the trajectories. Useful for
                                testing.

    huboplus.srdf               SRDF description of the robot.

    joint_limits.yaml           Joint velocity and acceleration limits
                                that can be used for trajectory smoothing.

    kinematics.yaml             Specifies IK solver plugins used for
                                individual motion planning groups and
                                parameters for the plugins.

    ompl_planning.yaml          Parameters for the OMPL motion planner.

    sensors.yaml                Specification of plugins used to
```

```
                              incorporate sensor data into the OctoMap
                              maintained internally by MoveIt! for
                              purposes of collision detection.

launch/

    demo.launch               Demonstration launch file that starts
                              up Rviz with the motion planning plugin
                              in addition to the the move_group.launch
                              file.

    move_group.launch         This is the main MoveIt! launch file
                              that starts up the move_group ROS
                              node.

    moveit.rviz               Configuration of the Rviz. (Contains
                              the motion planning plugin setup.)

    moveit_rviz.launch        Starts the Rviz with the moveit.rviz
                              configuration.

    planning_pipeline.launch.xml

                              Launches the selected motion planner.

    ompl_planning_pipeline.launch.xml

                              Takes care of the actual motion
                              planner (OMPL) setup.

    sensor_manager.launch.xml
    huboplus_moveit_sensor_manager.launch.xml

                              Loads parameters from sensor.yaml
                              into the Parameter server.

    planning_context.launch

                              Loads the SRDF description of the
                              robot into the ROS Parameter server,
                              together with settings from
                              kinematics.yaml and joint_limits.yaml
                              files.
```

```
trajectory_execution.launch.xml

                            Launches the selected ControllerManager
                            that can take care of the trajectory
                            execution.

huboplus_moveit_controller_manager.launch.xml
fake_moveit_controller_manager.launch.xml

                            Launch files that set up ROS parameters
                            defining the ControllerManager used
                            and load the ControllerManager related
                            parameters from the appropriate .yaml
                            file into the Parameter server.

setup_assistant.launch      Can be used to launch the Setup
                            Assistant to interactively modify
                            the SRDF configuration file.
```

Listing C.1: The hubo_moveit_config package structure