Single-Query Motion planning for Grasp Execution

J.G.J. Meijer

Master Thesis
Single-Query Motion planning for Grasp Execution

by

J.G.J. Meijer

For the degree of Master of Science in Mechanical Engineering at Delft University of Technology, to be defended publicly on 30th of May, 2017 at 10:00 AM.

Student number: 4326466
Department: BioMechanical Design (BMD)
Project duration: 2016/2017
Thesis committee: Prof. ir. M. Wisse, TU Delft, chairman
Ir. Q. Lei, TU Delft, daily supervisor
Dr.-Ing. J. Kober, TU Delft, external member

An electronic version of this thesis is available at http://repository.tudelft.nl/.
Preface

Ever since my parents gave me a LEGO Mindstorms set, I knew I wanted to design and build robots when I was older. In this spirit, I always steered towards studying mechanical engineering. During my applied sciences bachelor study I gained a lot of hands-on experience in engineering. However, in my opinion I did not gain enough theoretical knowledge in this study, so I changed my goals to gaining a masters degree. An obvious choice for me was to go to the TU Delft. In order for me to get accepted in the master program of mechanical engineering I had to finish a bridging program first. Which was tough coming from applied sciences but I made it through and got accepted.

I decided to follow the track of BioMechanical Design (BMD) in the master study of mechanical engineering. In this track, I chose for the BioRobotics specialization, a study which more-or-less deals with real-life LEGO Mindstorms. This thesis is the result of working in robotic research for a year. The thesis proposal started with Qujiang Lei, of the BioRobotics Lab, who introducing me to robotic grasping, which I find very interesting due to the unexpected complexity of performing grasps. The content of this thesis is intended to help researchers more easily select a motion planner for grasp execution. During the process of this thesis I contributed to the scientific community by presenting or co-presenting 5 scientific (conference) papers. Looking back, I can say I have gained meaningful experience in robotics!

J.G.J. Meijer
Delft, May 2017

Acknowledgements

During the development of this thesis I have used the knowledge and experience of many people. I would like to thank Qujiang Lei and Martijn Wisse for their guidance throughout this thesis and for helping me submit work to the scientific community. This all would not have been happening without the motivation and support of my parents during my whole study time, I am extremely grateful. I would like to thank Debbie for her support and for taking care of me during the long study nights. Special thanks to Benjamin and Beracha for their help in coding and/or writing related problems. To all the friends, thanks for the support and the wonderful times we shared in the studies!
Abstract

The grasping of objects is a highly desired function for service robots. To aid with grasping, researchers have developed grasping approaches. A demerit of existing approaches is that they solely focus on the grasp finding. A more important part in grasping is the grasp execution, which involves the solving of a motion planning problem. Currently, 23 sampling-based motion planners can be chosen from the Open Motion Planning Library (OMPL) within MoveIt!, a ROS framework that provides the tools for motion planning. However, no recommendations for selecting a specific planner for high performance is given. Moreover, difficulties in real-world grasp executions are typically not outlined in existing grasping approaches. In this thesis high-performing planners are selected for grasp executions and difficulties in performing such executions in a real-world setup are outlined.

The performance of the planners was analyzed by means of solved runs, computing time and path length. Three lightweight manipulators with different characteristics have been chosen to collect reliable data on planner performance. Various grasp executions have been defined with individual goals. One grasp execution incorporated a motion constraint that demands a specific orientation of the gripper. To achieve maximum performance, the parameters of the planners have been optimized.

For a grasp execution which starts moving in a confined space towards an open space, high performance was found with mono-directional tree-based planners with goal bias, such as EST, ProjEST, KPIECE and STRIDE. Mono-directional planners can propagate a roadmap out of narrow passages faster. The use of goal bias helps to plan a motion fast in open spaces in which the goal is located. Highest performance was obtained with KPIECE.

For a grasp execution which starts moving from an open space towards a confined space, bi-directional planners with lazy collision-checking (SBL and LBKPIECE) yielded highest performance. These planners also propagate a roadmap out of the goal configuration and only invalid path segments of the candidate solution are altered due to the lazy collision-checking, which makes these planners perform better.

In the investigated grasp execution that incorporates a motion constraint, high performance was obtained with the BiEST planner. This planner takes the density of existing samples into account in the expansion of the roadmap, which starts in the initial and goal configuration.

The experimental setup for real-world grasp execution was realized with modified ROS packages. Difficulties in real-world grasp executions exist due to inaccurate depth data, small errors in extrinsic calibration and the lack of verification of a grasp execution using an automatic command. Motion planning for real-world grasp executions showed a decrease in computing time performance, which is due to the increased demand for computing power by other processes that operate simultaneously in real-world setups.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>An Empirical Study on Single-Query Motion Planning for Grasp Execution</td>
<td>39</td>
</tr>
<tr>
<td>A.1</td>
<td>Abstract</td>
<td>40</td>
</tr>
<tr>
<td>A.2</td>
<td>Introduction</td>
<td>40</td>
</tr>
<tr>
<td>A.3</td>
<td>Background</td>
<td>40</td>
</tr>
<tr>
<td>A.4</td>
<td>Problem formulation</td>
<td>42</td>
</tr>
<tr>
<td>A.5</td>
<td>Defined motion planning problems</td>
<td>42</td>
</tr>
<tr>
<td>A.6</td>
<td>Parameter selection</td>
<td>43</td>
</tr>
<tr>
<td>A.7</td>
<td>Results</td>
<td>43</td>
</tr>
<tr>
<td>A.8</td>
<td>Conclusion</td>
<td>46</td>
</tr>
<tr>
<td>B</td>
<td>Parameter selection</td>
<td>49</td>
</tr>
<tr>
<td>B.1</td>
<td>Problem</td>
<td>49</td>
</tr>
<tr>
<td>B.2</td>
<td>Process</td>
<td>49</td>
</tr>
<tr>
<td>B.3</td>
<td>Example: Process description for the range parameter</td>
<td>49</td>
</tr>
<tr>
<td>B.4</td>
<td>Results</td>
<td>50</td>
</tr>
<tr>
<td>B.5</td>
<td>Discussion</td>
<td>54</td>
</tr>
<tr>
<td>C</td>
<td>Performance saturation of sampling-based motion planners</td>
<td>55</td>
</tr>
<tr>
<td>C.1</td>
<td>Execution</td>
<td>55</td>
</tr>
<tr>
<td>C.2</td>
<td>Results</td>
<td>55</td>
</tr>
<tr>
<td>C.3</td>
<td>Discussion</td>
<td>55</td>
</tr>
<tr>
<td>D</td>
<td>Grasp execution sequence</td>
<td>59</td>
</tr>
<tr>
<td>E</td>
<td>Programming code</td>
<td>61</td>
</tr>
<tr>
<td>E.1</td>
<td>Object segmentation</td>
<td>61</td>
</tr>
<tr>
<td>E.2</td>
<td>AR Track Alvar launch and configuration</td>
<td>65</td>
</tr>
<tr>
<td>E.3</td>
<td>Extrinsic calibration transform</td>
<td>65</td>
</tr>
</tbody>
</table>
## List of Figures

2.1 Solution to a motion planning problem, moving from initial state $q_I$ to goal state $q_G$ [27] . . . . 6
2.2 Workspace of point robot ................................................. 7
2.5 Optimization process of RRTstar ........................................... 11
2.6 ROS topics and nodes [30] .................................................. 12
2.7 Turtle movement after receiving velocity command [30] .................................................. 12
2.8 High level system architecture of MoveIt! (Sucan and Chitta [4]) ......................................... 13
2.9 Planning Scene Monitor architecture for MoveIt! (Sucan and Chitta [4]) ................................. 14

3.1 Pick and place benchmark for UR5 ......................................... 19
3.2 Pick and place benchmark for KUKA LWR 4+ .................................... 19
3.3 Pick and place benchmark for Kinova JACO .................................... 19
3.4 Results for benchmark 1 ...................................................... 20
3.5 Results for benchmark 2 ...................................................... 21
3.6 Results for benchmark 3 ...................................................... 24

4.1 Setup of a grasping pipeline .................................................. 26
4.2 Intrinsic calibration using a checkerboard to provide better depth accuracy [35] ....................... 27
4.3 Execution of extrinsic calibration using AR markers, necessary for grasp execution ................ 28
4.4 Octomap representation of a table with objects and obstacles .............................................. 28
4.5 Difficulty in grasping: visible occlusions in the Octomap representation ............................... 30
4.7 Grasp execution with grasping approach of Lei et al. [36] in the real-world setup of this thesis [36] 31

A.1 Benchmark 1: Grasp between obstacles ..................................... 42
A.2 Benchmark 2: Long motion .................................................. 42
A.3 Benchmark 3 & 4: Place & Pick motion ..................................... 43
A.4 Results for benchmark 1 ...................................................... 44
A.5 Results for benchmark 2 ...................................................... 45
A.6 Results for benchmark 3 ...................................................... 47
A.7 Results for benchmark 4 ...................................................... 47

B.1 Range parameter selection, iteration 1 ..................................... 51
B.2 Range parameter selection, iteration 2 ..................................... 52
B.3 Range parameter selection, iteration 3 ..................................... 53
B.4 Solved runs for default and improved parameters, higher is better ................................... 54

C.1 Place motion planning problem using the UR5. ................................ 56
C.2 Results for different amount of runs. (a) Solved runs; higher is better. (b) Computing time; lower is better, small interquartile range is better. (c) Path length; lower is better, small interquartile range is better. ........................................ 56

D.1 Motion sequence of a grasp execution using LBKPIECE ........................................... 60
## List of Tables

<table>
<thead>
<tr>
<th>Number</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Available OMPL planners in MoveIt!</td>
<td>2</td>
</tr>
<tr>
<td>3.1</td>
<td>Summary of available planners of OMPL in MoveIt!</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Planner parameters for UR5 (U), LWR 4+ (L) and JACO (J)</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>Mean values for benchmark 1</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>Mean values for benchmark 2</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>Mean values for benchmark 3</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>Mean computing times for grasp execution planning in the real-world, for grasp execution shown in figure 4.6</td>
<td>30</td>
</tr>
<tr>
<td>A.1</td>
<td>Summary of available planners of OMPL in MoveIt!</td>
<td>40</td>
</tr>
<tr>
<td>A.2</td>
<td>Specified planner parameters</td>
<td>44</td>
</tr>
<tr>
<td>A.3</td>
<td>Mean values for benchmark 1</td>
<td>45</td>
</tr>
<tr>
<td>A.4</td>
<td>Mean values for benchmark 2</td>
<td>46</td>
</tr>
<tr>
<td>A.5</td>
<td>Mean values for benchmark 3</td>
<td>46</td>
</tr>
<tr>
<td>A.6</td>
<td>Mean values for benchmark 4</td>
<td>46</td>
</tr>
<tr>
<td>B.1</td>
<td>Converged parameter values for UR5 (U), LWR 4+ (L) and JACO (J), used in chapter 3</td>
<td>50</td>
</tr>
<tr>
<td>C.1</td>
<td>Mean values and standard deviations for different amount of runs</td>
<td>57</td>
</tr>
</tbody>
</table>
List of Acronyms

AIM  International Conference on Advanced Intelligent Mechatronics.
BFMT  Bi-directional Forward Marching Tree.
BIEST  Bi-directional Expansive Space Trees.
BITRRT  Bi-directional Transition-based Rapidly Random-exploring Tree.
BKPIECE  Bi-directional Kinodynamic motion Planning by Interior-Exterior Cell Exploration.
CHOMP  Covariant Hamiltonian Optimization for Motion Planning.
EST  Expansive Space Trees.
FMT  Forward Marching Tree.
GUI  Graphical User Interface.
ICAR  International Conference on Advanced Robotics.
IEEE-RAS  Institute of Electrical and Electronics Engineers - Robotics and Automation Society.
KPIECE  Kinodynamic motion Planning by Interior-Exterior Cell Exploration.
LazyPRM  Lazy collision-checking Probabilistic RoadMap.
LBKPIECE  Lazy collision-checking Bi-directional Kinodynamic motion Planning by Interior-Exterior Cell Exploration.
LBT-RRT  Lower Bound Tree-Rapidly Random-exploring Tree.
OMPL  Open Motion Planning Library.
PDST  Path-Directed Subdivision Tree.
PRM  Probabilistic RoadMap.
ProjEST  Projection Expansive Space Trees.
ROS  Robot Operating System.
RRT  Rapidly Random-exploring Tree.
RVIZ  ROS Visualizer.
SBL  Single-query Bi-directional probabilistic roadmap planner with Lazy collision-checking.
SBPL  Search-Based Planning Library.
SPARS  SPArse Roadmap Spanner.
SRDF  Semantic Robot Description Format.
STOMP  Stochastic Trajectory Optimization for Motion Planning.
STRIDE  Search Tree with Resolution Independent Density Estimation.
TRRT  Transition-based Rapidly Random-exploring Tree.
URDF  Unified Robot Description Format.
Introduction

In 2015 the number of professional service robots sold has increased by 25% since 2014. The forecast is that this will increase in the upcoming years [1]. The grasping of objects is a key function for service robots. However, this function is still considered to be a complex problem in robotics due to the many elements that need to be covered. The Robot Operating System (ROS) [2] encourages collaborative robotics software development, allowing users to build upon each other’s work. This enhances and speeds up research in the robotics field [3].

In ROS, the go-to framework for dealing with the manipulation of a robot is MoveIt! [4], which is for instance needed to move a robotic arm from its initial position to another position. To achieve this a motion plan needs to be computed. In MoveIt! several motion planners or motion planner libraries can be configured. The Open Motion Planning Library (OMPL) [5] is the most supported library and the default option in MoveIt!. OMPL contains many state-of-the-art sampling-based motion planners. Currently, 23 of these planners are available in MoveIt! (listed in table 1.1).

Grasping approaches assist the selection of a grasp on an object. A demerit of the existing grasping approaches is that these tend to solely focus on the grasp finding. A more important part in grasping is the grasp execution, which involves the solving of a motion planning problem with a motion planner. Executing a grasp, like most robotic applications, has to be carried out in a timely manner. Without considering robotic hardware, the motion planning time and the path length of a motion plan are the main contributors to the grasp execution time. A motion planner with an optimization step aims to provide shorter paths. The drawback of this is the extra motion planning time. For grasp executions, motion planners can be considered to have high performance when a short path is found with minimal computing time.

Benchmark data of the available planners can aid the selection of the right planner for a given motion problem. However, benchmark data of planners when carrying out grasp executions are scarce, especially for the 12 planners that have been made available in the last MoveIt! release (December 2016). The Planner Arena [6] is a tool that can help selecting the right planner for a motion planning problem. Yet, a motion planning problem that resembles a grasp execution is not available in this tool. MoveIt! and OMPL itself does not provide any recommendations for choosing a motion planner for a defined motion planning problem. Currently, many time-consuming benchmarks need to be conducted and analyzed in order to find a high-performing planner for grasp executions. A study to select these planners could help users pick suitable planners. At the same time the study can provide useful benchmark data for the newer planners available in MoveIt!. This could show recent developments in the sampling-based motion planning regime.

Another shortcoming in existing state-of-the-art grasping approaches [7, 8] is that they do not provide data on how grasp executions can be performed in real-world setups. However, the approaches usually present results of real-world experiments. The difficulties in performing these experiments are not outlined. A demonstration of a real-world grasp execution can uncover these difficulties. Simultaneously, a high-performing sampling-based motion planner can be used to solve the grasp execution problem in a real-world setup and possible performance changes can be noticed.
1. Introduction

Table 1.1: Available OMPL planners in MoveIt!

<table>
<thead>
<tr>
<th>Planner name</th>
<th>Reference</th>
<th>Optimizing planners</th>
<th>In MoveIt! since December 2016</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBL</td>
<td>[9]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EST</td>
<td>Based on [10]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BiEST</td>
<td>[10]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>ProjEST</td>
<td>Based on [10]</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>KPIECE</td>
<td>[11]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BKPIECE</td>
<td>Based on [11]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LBKPIECE</td>
<td>Based on [11][12]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RRT</td>
<td>[13]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RRTConnect</td>
<td>[14]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>PDST</td>
<td>[15]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>STRIDE</td>
<td>[16]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>PRM</td>
<td>[17]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>LazyPRM</td>
<td>[12]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>RRTstar</td>
<td>[18]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>PRMstar</td>
<td>Based on [17][18]</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>LazyPRMstar</td>
<td>Based on [12][18]</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>FMT</td>
<td>[19]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BFMT</td>
<td>[20]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LBT-RRT</td>
<td>[21]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TRRT</td>
<td>[22]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BITRRT</td>
<td>[23]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SPARS</td>
<td>[24]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SPARStwo</td>
<td>[25]</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

1.1. Problem definition

Motion planning seeks to find a sequence of steps of motion (path) between the start and goal configuration such that a continuous movement can be performed, while satisfying motion constraints. Motion planners define a method to perform motion planning. To select high-performing planners a motion planning problem has to be solved, non-optimizing planners need to find feasible paths and optimizing planners need to find optimized feasible paths. A motion planning problem can be defined as follows [26].

Given:

1. A workspace, $W$, in which $W = \mathbb{R}^3$.
2. A manipulator, $M$, defined in $W$ with $n$ joints, $M_{1,2,...,n}$.
3. A configuration state, $x \in C$, where $C$ is the bounded $d$-dimensional configuration space, $C = [0,1]^d$. The obstacle-free space is defined by $C_{\text{free}} = \text{cl}(C \setminus C_{\text{obs}})$, in which $\text{cl}(\cdot)$ denotes the closure of a set and in which $C_{\text{obs}}$ denotes the obstacle space.
4. A path $p$ is a subsequent set of $x$ from an initial state, $x_{\text{init}}$, to a goal state $x_{\text{goal}}$.
5. A path is feasible when a continuous path is collision-free, $p(x) \in C_{\text{free}}$ for all $x \in [0,1]$, and satisfies $p(0) = x_{\text{init}}$ and $p(1) = x_{\text{goal}}$.

For optimized motion planning an extra step is introduced [18]:

6. Considering a motion planning problem $(C_{\text{free}}, x_{\text{init}}, x_{\text{goal}})$ and a cost function $c$, an optimized path is found if a feasible path $p^*$ is found such that $c(p^*) = \min\{c(p) : p \text{ is feasible} \}$.

Currently, 23 sampling-based motion planners of OMPL are available for use in MoveIt!, which are listed in table 1.1. The planners can be used to find feasible or optimized paths for grasp executions. However, no recommendations are given for planner selection nor literature can be found on selecting one of the 23 planners for performing grasp executions.

Existing grasping approaches verify the accuracy of grasps by using a real-world setup, however, it is typically not clearly outlined what the difficulties in real-world grasp executions are and whether if planner performance changes in this setup.
1.2. Overview of planners

This thesis will use the OMPL planners available in MoveIt!, which are listed in table 1.1. The OMPL planners are a mix of non-optimizing planners and optimizing planners. In this section a short description of each planner will be given.

The foundation of sampling-based motion planning is laid by Kavraki et al. [17]. They developed the Probabilistic Roadmap (PRM), this planner constructs a roadmap in the free configuration space by sampling collision-free robot configuration. Valid samples are saved in the roadmap as vertices (nodes). Edges (line segments) are constructed between vertices, which connect them. A graph search can be conducted (query) on the finished roadmap to find a sequence of robot configurations (path). A solution to the motion planning problem exists when a path can be made from the initial robot state to the goal robot state. The roadmap can be used for multiple queries (multi-query planner). The LazyPRM [12] planner initially does not check for valid samples when sampling nodes for roadmap construction. Collision-checking is performed along the vertices and edges of the roadmap once a path has been found. Invalid vertices and edges are then removed, more sampling is done until a new path is found. This process is repeated until a feasible path is found. PRMstar [18] is the asymptotically optimal variant of the PRM planner. It rewire vertices to other near vertices when its beneficial to the cost of path length. An asymptotically optimal path is found when the number of samples approaches infinity. LazyPRMstar [18] is a combination of LazyPRM and PRMstar.

In addition to the PRM planner and its variants, OMPL has two more multi-query planners available in MoveIt!, SPARS and SPARStwo. They are similar to PRMstar but adds a sparse subgraph. This subgraph is an asymptotically optimal roadmap that houses vertices which resemble multiple vertices in a dense graph. Less computing memory is needed to store the asymptotically optimal roadmap. SPARStwo is different since it has an infinite iteration loop.

Single-query planners create a map or graph for one query. The Rapidly Exploring Random Tree (RRT) grows a tree-structure from the initial robot configuration state in the direction of the unexplored areas of the bounded free space. The RRTConnect planner [14] is a bi-directional version of the RRT method. Two processes of RRT are initiated, one in the initial configuration state and one in the goal configuration state. At every new sample it is checked whether the two tree-structures can be connected to each other. The near-optimal variant of RRT, RRTstar [18], checks whether the new sampled vertices can be connected to other near vertices so that the configuration space is more locally refined. RRTstar removes the edges of the new sample that are not beneficial for the cost of the path length, like PRMstar. Lower Bound Tree-RRT (LBTRRT) [21] is an asymptotically optimal planner and uses a so-called lower bound graph, which is an auxiliary graph. To maintain the tree, a method similar to RRTstar is used. Transition-based RRT or TRRT [22] is a combination of the RRT method and a stochastic optimization method for global minima, which performs transition tests to accept new states to the tree. The algorithm computes an optimized path that is not tied to a time limit, unlike RRTstar. The Bi-TRRT [23] is a bi-directional version of this planner.

The EST method [10] stands for Expansive Space Trees. EST tries to determine the direction of the tree-structure by observing neighboring nodes. The tree will grow in the direction of the less explored space. Bi-directional EST (BiEST), based on [10], grows two tree-structures, similar to RRTConnect. Projection EST (ProjEST), also based on [10], detects the less explored area of the configuration space by using a grid, which serves as a projection of the state space.

Single-query Bi-directional probabilistic roadmap planner with Lazy collision checking, also called SBL, grows two tree-structures. The tree-structures expand in the same manner as EST. Due to its lazy collision checking, collision checking is performed along the path instead of at each sample. It deletes vertices and edges of the path that are not valid, similar to LazyPRM.

KPIECE (Kinodynamic motion Planning by Interior-Exterior Cell Exploration) [11] is a tree-based planner that uses layers of discretization to estimate the coverage of the state space. The OMPL implementation only uses one layer. OMPL incorporates a bi-directional variant called BKPIECE and a variant which incorporates lazy collision checking called LBKPIECE.

Fast Marching Tree (FMT) [19] is an asymptotically optimal planner which marches a tree forward in the cost-to-come space on a specified amount of samples. The BFMT [20] planner is a bi-directional variant of this planner.

PDST (Path-Directed Subdivision Tree) [15] represents samples as path segments instead of configuration states. It uses non-uniform subdivisions to explore the configuration space.

Search Tree with Resolution Independent Density Estimation (STRIDE) [16] uses a Geometric Nearneighbor Access Tree (GNAT) to sample the density of the configuration space. This information helps to guide the planner into the less explored area.
1.3. Thesis goals
From the problem definition, goals for this thesis can be established. Underneath the goals are noted, followed by a short explanation on how to reach these goals.

The goal of this thesis is to select high-performing sampling-based motion planners for grasp executions and demonstrate such an execution in a real-world setup, outlining difficulties and changes in planner performance.

To select high-performing sampling-based motion planners, the OMPL planners in MoveIt! will operate in a simulation environment. Performance will be measured in terms of solved runs, computing time and path length. By using multiple manipulators, it can be verified whether the performance of the planners is not bound to the type of manipulator. Also different type of motion planning problems can verify if the performance of the planners is not bound to a specific free configuration space. By analyzing the results, recommendations can be made for choosing a suitable planner in particular cases.

A demonstration of a real-world grasp execution could determine difficulties that arise. To achieve this a grasping pipeline needs to be designed from existing ROS packages. A high-performing planner will be used to solve the grasp execution. Since this part is intended to outline difficulties in executing a grasp in a real-world setup, the aim is not to present the most optimal grasping pipeline.

1.4. Thesis outline
This thesis has been organized in different chapters. Before the main content of the chapter a short explanation is given on why it is relevant.

In the introduction, chapter 1, the problem is introduced and it is explained how the work in this thesis can contribute to the object manipulation regime.

In the first parts of chapter 2 it is explained what motion planning is and how sampling-based motion planners can be used to solve a motion planning problem. The last parts in this chapter will explain how MoveIt! operates inside ROS.

In chapter 3 high-performing planners are selected among three different manipulators. This chapter is written in scientific paper format and was submitted to IEEE RAS International Conference on Advanced Robotics (ICAR).

Chapter 4 presents a grasping pipeline for a real-world setup in order to demonstrate a grasp execution with a high performing planner, which was identified in the previous chapter. The demonstration can identify current difficulties in real-world grasping applications.

The thesis outcome is discussed and concluded by chapters 5 and 6 respectively. The conclusion chapter also discusses proposals for future work.
As explained in the introduction, the goal of this thesis is to identify high performing motion OMPL planners in MoveIt!. This whole chapter serves as background information for users unfamiliar with motion planning, sampling-based motion planning and ROS/MoveIt!. The first sections of this chapter explain what motion planning is and gives various examples of sampling-based motion planners to help understand how such planners operate. The last sections explain how MoveIt! works within ROS.

2.1. Motion planning
In robotics, a solution to a motion planning problem can be defined as a **sequence of motions to drive the robot from its initial state towards the goal state, while satisfying motion constraints**. For example, when aiming for a collision-free solution to a motion planning problem without additional motion constraints, four elements need to be known:

- Start state of the robot
- Goal state of the robot
- Geometric description of the robot
- Geometric description of the world

In this case the problem formulation of motion planning is finding a path that sequentially moves the robot from its initial state to the goal state while never touching any obstacle present in the geometric description of the world.

Motion planning is executed in the **configuration space** \((C)\) of the robot, which is the space that consists of all the possible configurations of the robot. The configuration space consists of **free configuration space** and **obstacle space**, \(C_{free}\) and \(C_{obs}\) respectively. Robot configurations in the free space are collision-free, thus a sequence of linked robot configurations in the free configuration space represents a collision-free motion. A sequence of linked robot configurations starting in the initial robot configuration and ending in the goal robot configuration solves the motion planning problem. This is shown in figure 2.1, in which \(q_I\) is the initial robot configuration and \(q_G\) the goal robot configuration. A mathematical description of the motion planning problem is given in chapter 1.

2.2. Sampling-based motion planning
Motion planners help with motion planning. The first practical planners to solve a motion planning problem were developed in the 80’s by Khatib [28] and Brooks and Lozano-Perez [29]. However, these planners rely on an explicit representation of the obstacle space. This can result in a significant increase in computing effort when dealing with high dimensions and when dealing with a highly occupied configuration space by the obstacle space. Sampling-based motion planners can refrain from such computing effort since they do not require an explicit representation of the obstacle space. Moreover, they can provide **probabilistic completeness**, which means that the probability of the planner not finding a solution to the motion planning problem approaches zero when the number of samples approaches infinity.
Sampling-based motion planners sample the state space to find feasible robot configurations. These configurations are maintained in a map by vertices. By attempting to link the samples in the map to each other a sequence of configuration states can be made, these links are called edges. If a sequence of robot configuration states can be made from the initial state towards the goal state, a solution to the motion planning problem is found.

In the following part, three sampling-based motion planners are described in detail in order to get a good understanding of the concept. The PRM method is chosen due to the multi-query ability, RRT due to its widely used concept, and RRTstar due to its optimization process. To simply explain how the process of how the motion planning works, a motion problem is defined in which a moving point robot has to travel in a 2D plane, avoiding obstacles. In figure 2.2 the workspace for this point robot is shown.

Before going into detail of the three planners, primitive procedures that sampling-based motion planners usually rely on will be discussed. This follows the work of Karaman and Frazzoli [18].

- **Sampling**: Taking a sample can be done in either the full configuration space, \( C \), or the free configuration space, \( C_{\text{free}} \). The function \( \text{Sample} \) samples in \( C \) and \( \text{SampleFree} \), samples in \( C_{\text{free}} \).

- **Near & Nearest**: The map maintained by the sampling-based motion planner that contains vertices \( (V) \) and edges \( (E) \), can be denoted by \( G = (V, E) \). When given the \( \text{Nearest} \) function \( G \) and a random sample \( x_{\text{rand}} \), it will return the closest vertex \( v \in V \). The \( \text{Near} \) function returns a set of vertices \( V' \in V \) that are within a certain radius of \( x_{\text{rand}} \).

- **Steer**: A random sample can not always directly be connected to other vertices in the map, for example if the distance is too great. The \( \text{Steer} \) function creates a new sample, \( x_{\text{new}} \), in direction of the random sample but then being closer to the nearest vertex.

- **Collision**: To test if an edge is not colliding with obstacles, a collision test is performed by the \( \text{CollisionFree} \) function. It returns \( True \) when the line segment \( [x, x'] \in C_{\text{free}} \), otherwise it returns \( False \).

### 2.2.1. PRM

In this part the PRM (Probabilistic RoadMap) [17] planner workings will be explained, the map creation pseudocode of this planner is presented in algorithm 1. PRM starts by sampling states in the free state space (thus using \( \text{SampleFree} \)). For each new sample it is checked whether an edge can be made, by checking for collisions between this sample and other near samples (\( \text{CollisionFree} \), \( \text{Near} \)). After some iterations, a map similar to the map in the workspace of figure 2.3a is made. After a user specified amount of samples \( n \) in the state space the planner returns the map \( G = (V, E) \), shown in figure 2.3b.
Now that the map covers the total free state space, a query can be made. A query specifies the initial state and goal state of the motion planning problem, performing a graph search to find the shortest path as shown in figure 2.3c. Since the roadmap covers the total free state space the last step can be performed again for different queries. Hence, this planner is also referred to as a multi-query planning method. These planners can be useful in fixed environments, since only graph searches need to be executed in order to find motion plans. Though in changing environments single-query performance can be more suitable, since they can create roadmaps faster [18].

Algorithm 1 PRM (graph creation; $G = (V,E)$)

1: $V \leftarrow \emptyset$;
2: $E \leftarrow \emptyset$;
3: for $i = 0, ..., n$ do
4:   $x_{\text{rand}} \leftarrow \text{SampleFree}_i$;
5:   $U \leftarrow \text{Near}(G = (V,E), x_{\text{rand}}, r)$;
6:   $V \leftarrow V \cup \{x_{\text{rand}}\}$;
7:   for each $u \in U$, in order if increasing $||u - x_{\text{rand}}||$, do
8:     if $x_{\text{rand}}$ and $u$ are not in the same connected component of $G = (V,E)$ then
9:       if $\text{CollisionFree}(x_{\text{rand}}, u)$ then
10:          $E \leftarrow E \cup \{(x_{\text{rand}}, u), (u, x_{\text{rand}})\}$;
11:     end if
12:   end if
13: end for
14: end for
15: return $G = (V,E)$;

2.2.2. RRT

A good example of a single-query planning method is a tree-based planner, which creates tree structures in the free state space. A wide variety of tree-based planners exist, the most known tree-based planner is the
RRT planner by Lavalle [13], the map creation pseudocode of this planner is presented in algorithm 2. For simplicity and continuity, the same workspace to explain the principle of the planner will be used, shown in figure 2.2. The planner start creating the tree structure from the start state. It samples random states in the free state space (SampleFree). By using the Nearest function the closest vertex is returned. The planner creates a new state by using the Steer function. This state is in the direction of the random state but within a certain distance of the closest vertex. Random valid samples are chosen by using the planner's specified expansion heuristic. The creation of the tree is shown in figures 2.4a, 2.4b and 2.4c.

Once the tree can be connected to the goal state, a sequence of configuration states can be made that solves the motion planning problem. This is done by searching for the shortest path in the map $G$. The found path is shown in figure 2.4c. As can be seen from the RRT creation figures the chosen path is not the most optimal. Optimizing planners, like RRTstar, can find a more optimal solution to the motion planning problem.

Algorithm 2 RRT (graph creation; $G = (V, E)$)
1: $V \leftarrow \{x_{\text{init}}\}$
2: $E \leftarrow \emptyset$
3: for $i = 0, \ldots, n$ do
4:     $x_{\text{rand}} \leftarrow \text{SampleFree}$;
5:     $x_{\text{nearest}} \leftarrow \text{Nearest}(G = (V, E), x_{\text{rand}})$;
6:     $x_{\text{new}} \leftarrow \text{Steer}(x_{\text{rand}}, x_{\text{nearest}})$;
7:     if ObstacleFree($x_{\text{nearest}}, x_{\text{new}}$) then
8:         $V \leftarrow V \cup \{x_{\text{new}}\}$;
9:         $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$;
10:    end if
11: end for
12: return $G = (V, E)$;
2.2.3. RRTstar

In order to understand the inner workings of the RRTstar planner, four more functions need to be discussed.

- **Obstacle free**: The check that attempts to connect the nearest node to the new sample is called \texttt{ObstacleFree}. If successful the new sample can be added to the vertices $V$.

- **Cost**: The \texttt{Cost} function maps a vertex $v \in V$ to the cost of the path from the root of the tree ($v_0$) to $v$.

- **Parent**: Mapping a vertex $v \in V$ to the unique vertex $u \in V$, such that $(u, v) \in E$, is done by \texttt{Parent}.

- **Line**: The \texttt{Line} function returns the distance between two points, this is usually transformed to a cost $c$.

Optimizing planners can find a more optimal solution for the motion planning problem, this usually comes at a cost for computing effort. The RRTstar [18] planner is a well known planner to optimize paths. The planner start the same way as the RRT method, however, extra steps are performed when adding edges. The pseudocode of the graph creation is presented in algorithm 3. For new sample it is attempted to make a connection to the nearest node (\texttt{ObstacleFree}), this is shown in figure 2.5b. Nearby vertices that lie in a variable radius of the new sample are added to the set $X_{\text{near}}$ by \texttt{Near}, shown in figure 2.5c. For every $x_{\text{near}} \in X_{\text{near}}$ it is checked if a connection can be made to the new sample, as long as the cost towards this near state is less than current cost towards the near state. If a better cost can be found towards $x_{\text{near}}$, a new edge is added to $E$, shown in figure 2.5d. Edges that do not contribute to the cost of the near states are removed from $E$, this is shown in figure 2.5e. When the number of samples $n$ approaches infinity the solution approaches the optimal solution.
2.7. Services can let nodes communicate with each other. This allows nodes to send a request and receive a
will compute and execute the velocity command movement for the turtle on the 2D plane, shown in figure
input. In turn the node computes these inputs and translates it to a velocity command message. This is pub-
filesystem
ROS operates, the three levels of concepts have to be clarified: the
applications. ROS was created to encourage collaborative robotics software development. In order to explain how
The open-source Robot Operating System (ROS) is a suite of software libraries that help create robot appli-
2.3. Motion planning with MoveIt!
The main software to use for mobile manipulation of a robot within ROS is called MoveIt!. In essence it is a
framework that deals with for instance the kinematics, 3D perception, motion planning control and naviga-
tories. Repositories can contain a collection of packages.
for messages in ROS), service descriptions (request and response structures for services in ROS) and reposi-
contain runtime processes (nodes), ROS-dependent libraries, datasets, configuration files, and/or other data
software libraries, or so-called packages, are the main source that make and define ROS. These packages can
Algorithm 3 RRTstar (graph creation; \(G = (V, E)\))

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>(V \leftarrow {x_{\text{init}}});</td>
</tr>
<tr>
<td>2.</td>
<td>(E \leftarrow \emptyset);</td>
</tr>
<tr>
<td>3.</td>
<td>\textbf{for} (i = 0, \ldots, n) \textbf{do}</td>
</tr>
<tr>
<td>4.</td>
<td>(x_{\text{rand}} \leftarrow \text{SampleFree});</td>
</tr>
<tr>
<td>5.</td>
<td>(x_{\text{nearest}} \leftarrow \text{Nearest}(G = (V, E), x_{\text{rand}}));</td>
</tr>
<tr>
<td>6.</td>
<td>(x_{\text{new}} \leftarrow \text{Steer}(x_{\text{rand}}, x_{\text{nearest}}));</td>
</tr>
<tr>
<td>7.</td>
<td>\textbf{if} (\text{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}})) \textbf{then}</td>
</tr>
<tr>
<td>8.</td>
<td>(x_{\text{near}} \leftarrow \text{Near}(G = (V, E), x_{\text{new}}, r_f));</td>
</tr>
<tr>
<td>9.</td>
<td>(V \leftarrow V \cup {x_{\text{new}}});</td>
</tr>
<tr>
<td>10.</td>
<td>(x_{\text{min}} \leftarrow x_{\text{nearest}});</td>
</tr>
<tr>
<td>11.</td>
<td>(c_{\text{min}} \leftarrow \text{Cost}(x_{\text{nearest}}) + c(\text{Line}(x_{\text{nearest}}, x_{\text{new}})));</td>
</tr>
<tr>
<td>12.</td>
<td>\textbf{for each} (x_{\text{near}} \in X_{\text{near}}) \textbf{do}</td>
</tr>
<tr>
<td>13.</td>
<td>\textbf{if} (\text{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \land \text{Cost}(x_{\text{near}}) + c(\text{Line}(x_{\text{near}}, x_{\text{new}})) &lt; c_{\text{min}}) \textbf{then}</td>
</tr>
<tr>
<td>14.</td>
<td>(x_{\text{min}} \leftarrow x_{\text{near}});</td>
</tr>
<tr>
<td>15.</td>
<td>(c_{\text{min}} \leftarrow \text{Cost}(x_{\text{near}}) + c(\text{Line}(x_{\text{near}}, x_{\text{new}})));</td>
</tr>
<tr>
<td>16.</td>
<td>\textbf{end if}</td>
</tr>
<tr>
<td>17.</td>
<td>\textbf{end for}</td>
</tr>
<tr>
<td>18.</td>
<td>(E \leftarrow E \cup {(x_{\text{min}}, x_{\text{new}})});</td>
</tr>
<tr>
<td>19.</td>
<td>\textbf{for each} (x_{\text{near}} \in X_{\text{near}}) \textbf{do}</td>
</tr>
<tr>
<td>20.</td>
<td>\textbf{if} (\text{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \land \text{Cost}(x_{\text{near}}) + c(\text{Line}(x_{\text{near}}, x_{\text{new}})) &lt; \text{Cost}(x_{\text{near}})) \textbf{then}</td>
</tr>
<tr>
<td>21.</td>
<td>(x_{\text{parent}} \leftarrow \text{Parent}(x_{\text{near}}));</td>
</tr>
<tr>
<td>22.</td>
<td>(E \leftarrow (E \setminus {(x_{\text{parent}}, x_{\text{near}})}) \cup {(x_{\text{new}}, x_{\text{near}})});</td>
</tr>
<tr>
<td>23.</td>
<td>\textbf{end if}</td>
</tr>
<tr>
<td>24.</td>
<td>\textbf{end for}</td>
</tr>
<tr>
<td>25.</td>
<td>\textbf{end if}</td>
</tr>
<tr>
<td>26.</td>
<td>\textbf{end for}</td>
</tr>
<tr>
<td>27.</td>
<td>\textbf{return} (G = (V, E));</td>
</tr>
</tbody>
</table>

2.3. Motion planning with MoveIt!
The main software to use for mobile manipulation of a robot within ROS is called MoveIt!. In essence it is a
framework that deals with for instance the kinematics, 3D perception, motion planning control and naviga-
tory. MoveIt! uses instances of ROS to operate, before going into the details of MoveIt! a primitive explanation
of ROS will be given.
2.3.1. ROS
The open-source Robot Operating System (ROS) is a suite of software libraries that help create robot applic-
ations. ROS was created to encourage collaborative robotics software development. In order to explain how
ROS operates, the three levels of concepts have to be clarified: the filesystem level, the computation graph
level, and the community level.
The filesystem level of ROS covers the resources that can be found on the local computing drive. The
software libraries, or so-called packages, are the main source that make and define ROS. These packages can
contain runtime processes (nodes), ROS-dependent libraries, datasets, configuration files, and/or other data
that are useful for a robotic application. The filesystem level also consists of message descriptions (structures
for messages in ROS), service descriptions (request and response structures for services in ROS) and reposi-
tories. Repositories can contain a collection of packages.
A peer-to-peer network of ROS processes, that process data together, is maintained by the computation graph
level. This level mainly consists of nodes, messages, topics and services. Some of these can be explained by using figure 2.6. This figure shows the computation graph of the turtlesim, a package that simulates the movement of a turtle on a 2D plane. The node /teleop_turtle can receive commands from a user input. In turn the node computes these inputs and translates it to a velocity command message. This is published on the /turtle1/command_velocity topic to which the turtlesim node is subscribed. The turtlesim node will compute and execute the velocity command movement for the turtle on the 2D plane, shown in figure 2.7. Services can let nodes communicate with each other. This allows nodes to send a request and receive a
2.3. Motion planning with MoveIt!

(a) Initialization

(b) Adding new state

(c) Nearby neighbor states

(d) Compare cost to goal state

(e) Removal of edges

Figure 2.5: Optimization process of RRTstar
response. The ROS Master is part of the *computation graph*, it helps the nodes to find each other and it can exchange messages and can invoke services.

![Figure 2.6: ROS topics and nodes](image)

The *community* level are ROS resources to help exchange software and knowledge among communities. This level mainly consists of distributions, repositories, ROS wiki and *ROS Answers*. Distributions are collections of versioned stacks that can be installed. To this date, 10 ROS distributions have been released. The latest release is Kinetic Kame which enabled the newer OMPL planners for the MoveIt! package, noted in table 1.1. Repositories are used to share robot software components, *GitHub*\(^1\) is one of the most used tools to share repositories. ROS wiki is a place for the community to find documentations for ROS packages. It also provides tutorials, this can help users learn about the various aspects of ROS. If users cannot find the right answer on the wiki they can ask a question on *ROS Answers*, which is a highly maintained communication channel.

### 2.3.2. MoveIt!

As noted before, MoveIt! deals with a lot of parts necessary for the manipulation of a robot. The high level system architecture of MoveIt! is shown in figure 2.8. The primary node is the *move_group*, to which all the parts connect. The topics in the figure can be categorized into three groups, namely: user interface (left side), robot interface (right side) and the configuration (top).

#### User interface

There are three ways for the user to interact with the *move_group*. Using C++, users can easily configure the primary node by using the *move_group_interface* package. Similarly, this can be done with Python, instead a different package called the *moveit_commander* has to be used. ROS Visualizer (RVIZ) is a Graphical User Interface (GUI) to configure MoveIt! tasks, it uses the *Motion Planning Plugin*.

\(^1\)https://github.com/
2.3. Motion planning with MoveIt!

Configuration
The configuration part of the system architecture deals with the robot model specifics. The MoveIt! framework needs three main file categories that represent the particulars of the robot.

The first is the URDF-file which stands for Unified Robot Description Format, which contains information about links and joints. It also specifies how these elements relate to each other.

The second file is the SRDF-file which stands for Semantic Robot Description Format. In this file the relevant groups of a robot can be specified, for instance a gripper. In this way MoveIt! knows for which joints and links it should make a motion plan. The file also notes whether collisions are allowed between certain links of the robot.

Other configurations for MoveIt! can consist of kinematics, joint limits, controllers and planning related configurations. These configuration files are typically specified in the config folder of the robot unique MoveIt! package.

Robot interface
The move_group communicates with the robot through ROS topics and actions. The joint_states topic includes information about the current state of the robot, the move_group is subscribed to this topic to determine the locations of all the links and joints. MoveIt! itself does not produce a joint_states topic, instead this has to be provided by the robot controller.

To execute a motion with the robot through MoveIt!, the move_group communicates with the robot controllers through the FollowJointTrajectoryAction interface. This is a ROS action interface and has to be provided by the robot.

Other topics for the move_group to communicate with is for instance the Point Cloud topic. This topic can be used by MoveIt! to produce a collision map, like an Octomap. This is part of the planning scene specified in MoveIt!.

Planning scene
MoveIt! uses the planning scene to represent the world in which the robot operates, including information about the robot state itself. This information is handled by the planning scene monitor, shown in figure 2.9. To represent the robot in the planning scene the planning scene needs to listen to the joint_states topic. As
said before a Point Cloud topic can be used to produce a collision map, which represents the geometry of the world. However, other sensor topics can also be used to represent the world geometry. Pre-determined world geometry can be included as input to the planning scene topic.

Motion planners
Performance of motion planning depends on the chosen motion planner. MoveIt! itself does not provide motion planning, but instead it is designed to work with planners or planning libraries. Currently four main planners/planning libraries can be configured to use.

- OMPL (Open Motion Planning Library) [5] is a popular choice to solve a motion problem. It is an open-source motion planning library that houses many state-of-the-art sampling based motion planners. OMPL is configured as the default set of planners for MoveIt!. Currently 23 sampling-based motion planners can be selected for use.

- Stochastic Trajectory Optimization for Motion Planning (STOMP) [31] is an optimization-based motion planner. It is designed to plan smooth trajectories for robotic arms. The planner is currently partially supported in MoveIt!

- Covariant Hamiltonian Optimization for Motion Planning (CHOMP) [32] mainly operates by using two terms. The dynamical quantity term describes the smoothness of the trajectory. The obstacle term is similar to potential fields. The planner is not yet configured in the latest version of MoveIt!

- Search-Based Planning Library (SBPL) [33] consists of a set of planners using search-based planning that discretize the space. The library is not yet configured in the latest version of MoveIt!

2.4. Conclusion
This chapter explained that motion planning is the finding of a sequence of control inputs or configuration states between the initial state and the goal state, while satisfying motion constraints. Sampling-based motion planners are used due to their probabilistic completeness and since they can refrain from high computing effort in high dimensional motion planning. The planners do this by sampling states in the configuration space of the robot by using a planner specific heuristic. Linking the individual states can result in a sequence from start to goal state, thus solving the motion planning problem. Sampling-based motion planners of OMPL are available to MoveIt!, the go-to motion planning framework of ROS. This framework uses a planning scene to represent the world in which the robot operates, including information about the robot state itself. MoveIt! uses elements of ROS to communicate with the robot.
In the introduction, chapter 1, it was explained that finding benchmark data of the planners when performing grasp executions is scarce. Especially for the recent 12 added sampling-based motion planners in OMPL. Performing benchmarks for a motion problem can be time consuming, as well as parameter selection for the planners. A need for planner benchmark data exists, this can guide users in selecting the right planner for a certain grasp execution. This chapter aims to fulfill the goal of identifying high-performing sampling-based motion planners for grasp execution. It uses similar parts that are present in chapter 1.

In an initial study, presented in appendix A, it was shown that for the UR5 manipulator SBL, BKPIECE, LBKPIECE, RRTConnect and BiTRRT are high performing planners when considering solved runs, computing time and path length. The work in this chapter extends this study by performing grasp executions with various manipulators. This could show consistency in planner performance in different configuration spaces due to the differences in manipulators. Moreover, the following work includes a motion constraint grasp execution, which was not investigated in appendix A. To achieve maximum performance of the planners an extensive parameter selection process was conducted, the importance and workings of this process is explained in appendix B. Additionally, it was checked whether the amount of runs has significant impact on selecting high-performing planners for grasp executions, this is analyzed in appendix and discussed.

A version of this work with smaller images was submitted to the IEEE-RAS International Conference on Advanced Robotics (ICAR) which will be held in July, 2017.
Performance Study of Single-Query Motion Planning for Grasp Execution Using Various Manipulators

Jonathan Meijer
TU Delft Robotics Institute
Delft University of Technology
Delft, The Netherlands
j.g.j.meijer@student.tudelft.nl

Qujiang Lei
TU Delft Robotics Institute
Delft University of Technology
Delft, The Netherlands
q.lei@tudelft.nl

Martijn Wisse
TU Delft Robotics Institute
Delft University of Technology
Delft, The Netherlands
m.wisse@tudelft.nl

Abstract—This paper selects high-performing motion planners among three manipulators when carrying out grasp executions. Simultaneously, this paper presents useful benchmarking data. Sampling-based motion planners of OMPL available for use in MoveIt! are compared by performing several grasping-related motion planning problems. The performance of the planners is measured by means of solved runs, computing time and path length. Based on the results, recommendations are made for planner choice that shows high performance for the used manipulators.

I. INTRODUCTION

Currently, 23 sampling-based motion planners of OMPL (Open Motion Planning Library) [1] are available for use in MoveIt! [2], a robot manipulation framework for Robot Operating System (ROS). OMPL is the default and most supported motion planning library in MoveIt!. Support for picking a planner is not provided. In the literature, no research about planner performance can be found when performing grasp executions. Moreover, performance information for 12 planners is scarce, since they have just been released (December 2016). This leaves the user to perform time-consuming benchmarks in order to find the right planner.

This paper aims to select high performing OMPL motion planners available in MoveIt! when carrying out grasp executions. Simultaneously, it aims to present useful benchmarking data for all the 23 planners. By conducting several grasp executions for three different manipulators, we hope to find planners that consistently show high performance. We will use three manipulators that have different geometry but have similar specifications. These are Universal Robots UR5, KUKA LWR 4+ and Kinova JACO. To resemble a real-world grasping problem, the manipulators are fitted with a gripper. In this study, we only consider sampling-based motion planners of OMPL available in MoveIt!, listed in Tab. I. This includes so-called multi-query planning methods. However, only single-query performance is measured. Since most robotic application aim for fast executions, the planners in this paper will operate within a maximum computing time. We have designed a virtual environment that resembles a shelf in which objects can be picked or placed. Planner choice is investigated by considering geometry constraints for two motion planning problems. For the third problem we add a motion constraint.

The performance of the planners is measured in terms of solved runs, computing time and path length. The metric solved runs can be expressed as a percentage of the total motion planning runs that finish correctly. Barplots are used to visualize the difference. For every run, computing time and path length can change due to the randomization in sampling-based motion planners. Boxplots and tables are used to analyze the planners with respect to computing time and path length. Performance depends on the need of the user, high performance for one metric can result in low performance for an other metric. By analyzing each metric separately, we can elaborate on right planners for each metric.

Tab. I shows there are optimizing OMPL planners available in MoveIt! that have a time-invariant goal. This means they stop computing as soon as a path is found that is considered to be more optimal. However, compared to non-optimizing planners, computing effort is increased which can result in an increase in computing time. For optimizing planners is expected that they will produce shorter path lengths. This study can clarify if extra computing time of optimizing planners will considerably decrease the path length compared to non-optimizing planners.

II. BACKGROUND

A. Manipulators

Universal Robots UR5 [3]: Universal Robots aims to provide easily programmable, safe and flexible industrial robots. The UR5 manipulator is designed to be lightweight, flexible and collaborative. The manipulator has 6 degrees of freedom (DOF). Joint limits are max $2\pi$ in both directions. The manipulator has a maximum payload of 5kg and a span of 850mm. A model of the UR5 fitted with a gripper is shown in Fig. 1.

KUKA LWR 4+ [4]: KUKA supplies intelligent automation solutions and is currently one of the top brands in this field. The 7-DOF LWR 4+ is a lightweight collaborative manipulator. Its furthest reach is 1178.5mm and it can lift up to 7kg. The manipulator is made for universal purpose, meaning it can be used for many applications. The upper and lower limits are $\pm 170$ degrees for four joints.
TABLE I: SUMMARY OF AVAILABLE PLANNERS OF OMPL IN MOVEIT!

<table>
<thead>
<tr>
<th>Planner name &amp; Reference</th>
<th>Optimizing planners</th>
<th>Multi-query</th>
<th>Time-invariant goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBL [6]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>EST</td>
<td>✓ Based on [7]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BEST</td>
<td>✓ [7]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ProjEST</td>
<td>✓ Based on [7]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>KPIECE [8]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BKPIECE Based on [8]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LBKPIECE Based on [8][9]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RRT [10]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RRTConnect [11]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PDST [12]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>STRIDE [13]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PRM [14]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LazyPRM [9]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RRTstar [15]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PRMstar Based on [14][15]</td>
<td>✓ ✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LazyPRMstar Based on [9][15]</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>FMT [16]</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>BFMT [17]</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>LBTRRT [18]</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>TRRT [19]</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>BRTRRT [20]</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>SPARS [21]</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>SPARStwo [22]</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
<td>✓ ✓</td>
</tr>
</tbody>
</table>

and ±120 degrees for the remaining joints. A model of the LWR 4+ is shown in Fig. 2.

Kinova JACO\(^1\) and JACO\(^2\) [5]: Kinova has developed two JACO versions, the JACO\(^1\) and the JACO\(^2\). They are lightweight and geometrically identical. The manipulators have a maximum reach of 900mm. Other than the UR5 and LWR 4+, which have straight links, this manipulator has two links which have a curve. Both versions have three joints that can rotate continuously, remaining joints have limits due to the geometry. The maximum payload is 1.5kg for the JACO\(^1\) and 2.6kg for the JACO\(^2\). The manipulator has 6 degrees of freedom. A model is shown in Fig. 3.

B. Software

The open-source ROS is a suite of software libraries that help create robot applications. ROS was created to encourage collaborative robotics software development. MoveIt! is a framework for ROS that helps with the manipulation of robotic hardware. Several motion planner libraries can be configured with MoveIt! to perform motion planning for a specified robot. OMPL is a well-integrated motion planner library and is the default library for MoveIt!. The library includes state of the art sampling-based motion planners. OMPL itself only implements the basic primitives of sampling-based motion planning. MoveIt! configures OMPL and provides the back-end for OMPL to work with motion planning problems.

Using MoveIt!, OMPL creates a path to solve the motion planning problem. By default, OMPL tries to perform path simplification. These are routines that shorten the path. The smoothness of the path may not be affected by this simplification.

C. Overview of planners

Sampling-based motion planners are proven to be probabilistically complete [14], which implicates that the probability of not finding a feasible path in an unbounded setting approaches zero. For this reason, these planners are widely used to find feasible paths in high-dimensional and geometrically constraint environments. Optimizing planners can refrain from potential high-cost paths and rough motions [15]. However, computational effort for finding an optimized path is increased.

Among the 23 motion planners in Tab. I, six can be considered as multi-query planning methods. A common multi-query planning method is the Probabilistic RoadMap (PRM) [14]. The planner attempts to find a path in a constructed roadmap. The construction of the roadmap is executed by sampling valid nodes (configuration states) in the configuration space. These nodes are connected to other nearby nodes by edges (path segments). The maximum nearest neighbors parameter defines the maximum amount of nearest neighbors the node can connect to. More connections will result in a denser map, however, computing effort is increased. In OMPL, the roadmap construction is finished when a certain time limit is reached. Afterwards a simple graph search (query) can be performed on the roadmap to find a path between the start and goal node. Because the algorithm covers the total configuration space with a roadmap, it can be used again to find a path with different start and goal node. This makes it a multi-query planning method.

In MoveIt!, three variants of the PRM planner are available for use. The LazyPRM [9] planner initially does not check for valid states when sampling nodes for roadmap construction. Once a path has been found from between start and goal node, collision checking is performed along the nodes and edges of the roadmap. Invalid nodes and edges are removed and a new graph search is attempted. This process is repeated until a feasible path is found. The range parameter specifies the maximum distance that a tree node is extended towards the sample. This parameter needs to be low enough to cover the configuration space, however, computing effort is increased for lower values. PRMstar [15] is the asymptotically optimal variant of the PRM planner. It rewires nodes to other near nodes if this is beneficial to the cost towards the node. An asymptotically optimal path is found if there is a great number of nodes. LazyPRMstar [15] combines the LazyPRM and PRMstar.

In addition to the PRM planner and its variants, OMPL has two more multi-query planners, SPARS and SPARStwo. They are similar to PRMstar but adds another sparse subgraph. This subgraph is an asymptotically optimal roadmap that houses nodes which resemble multiple nodes in a dense graph. Therefore less computing memory is needed to store the asymptotically optimal roadmap. SPARStwo is different since it has an infinite iteration loop.

The remaining 17 planners in Tab. I are considered as single-query planning methods. These create a roadmap every time a new planning query has to be determined. A common single-query planner is the Rapidly Exploring Random Tree (RRT) method [10]. It grows one tree (mono-directional) from the initial configuration state in the direction of the unexplored areas of the bounded free
space. This is realized by randomly sampling nodes in the free space, sampled nodes that can be are within a certain distance of tree nodes are added to the tree by edges. The process of adding nodes and edges is repeated until the tree reaches the goal node. The goal bias parameter in this planner specifies the probability of choosing the goal configuration as sample rather than a random sample.

The RRTConnect method [11] is a bi-directional version of the RRT method, meaning that two trees are grown. Two processes of RRT are started, one in the start node and one in the goal node. At every iteration or edge addition, it is checked whether the trees can be connected to each other. A path that solves the motion planning problem, is found if these trees can be connected. The near-optimal variant of RRT, RRTstar [15], checks whether the new sampled node can be connected to other near nodes so that the state space is more locally refined. The RRTstar removes the connections of the new sample that are not beneficial towards the cost of the path, like PRMstar. When the number of nodes is big enough, it can result in an asymptotically optimal path from the start-to-goal node. As shown in Tab. I, the RRTstar goal is time-invariant. It keeps trying to optimize the trees by adding new nodes until specified time limit is met.

Lower Bound Tree-RRT (LBT-RRT) [18] is an asymptotically optimal planner and uses a so-called lower bound graph which is an auxiliary graph. To maintain the tree, a similar method as RRTstar is used. Transition-based RRT or TRRT [19] is a combination of the RRT method and a stochastic optimization method for global minima. It performs transition tests to accept new states to the tree. The algorithm computes an optimized path that is not tied to a time limit, unlike RRTstar. The Bi-TRRT [20] is a bi-directional version of this planner.

The EST method [7] stands for Expansive Space Trees. Other than RRT, EST tries to determine the direction of the tree by looking at neighboring nodes. The tree will grow in the direction of the less explored space. Bi-directional EST (BiEST), based on [7], grows two trees like RRTConnect. Projection EST (ProjEST), also based on [7], detects the less explored area of the configuration space by using a grid. This grid serves as a projection of the state space. Single-query Bi-directional probabilistic roadmap planner with Lazy collision checking, also called SBL, grows two trees. The trees expand in the same manner as EST. Due to its lazy collision checking it will determine if a path is valid after the two trees are connected. It deletes nodes and edges of the path that are not valid, similar to LazyPRM.

KPIECE (Kinodynamic motion Planning by Interior-Exterior Cell Exploration) [8] is a tree-based planner that uses layers of discretization to help estimate the coverage of the state space. The OMPL implementation only uses one layer. OMPL incorporates a bi-directional variant called BKPIECE and a variant which incorporates lazy collision checking, this is the LBKPIECE.

Fast Marching Tree (FMT) [16] is an asymptotically optimal planner which marches a tree forward in the cost-to-come space on a specified amount of samples. The BFMT [17] planner is a bi-directional variant of this planner.

PDST (Path-Directed Subdivision Tree) [12] represents samples as path segments instead of configuration states. It uses non-uniform subdivisions to explore the state space. STRIDE (Search Tree with Resolution Independent Density Estimation) [13] uses a Geometric Nearneighbor Access Tree (GNAT) to sample the density of the configuration space. This information helps to guide the planner into the less explored area.

III. PROBLEM FORMULATION

The available planners consist of non-optimizing and optimizing planners. The problem formulation follows the work of Karaman and Frazzoli [15]. Non-optimizing planners attempt to find a feasible path in the bounded d-dimensional configuration space \( C = [0,1]^d \). The free configuration space is defined by \( C_{\text{free}} = cl(C \setminus \text{Obstacle}) \), in which \( cl(\cdot) \) denotes the closure of a set and in which \( \text{Obstacle} \) denotes the obstacle space. A sequence of connected configuration states make up a path \( p \). This path is called feasible when:

\[
\begin{align*}
p(0) &= x_{\text{init}}, p(1) = x_{\text{goal}} \\
p(x) &\in C_{\text{free}} \text{ for all } x \in [0,1]
\end{align*}
\]

Optimizing planners that are given a motion planning problem \((C_{\text{free}}, x_{\text{init}}, x_{\text{goal}})\) and a cost function \( c \), find an optimized path \( p^* \) such that:

\[
c(p^*) = \min \{c(p) : p \text{ is feasible} \}
\]

The performance of sampling-based motion planners is greatly dependant by the free configuration space. Motion planning through narrow passages are known to cause issues for these type of planners. This can be better understood by following the work of Hsu et al. [23]. For any subset \( S \subseteq C, \mu(S) \) denotes the volume. For any \( x \in C_{\text{free}}, \mathcal{V}(x) \) denotes a set seen from \( x \), called the visibility set of \( x \). Two configurations are visible to each other if they can be connected by a straight line in \( C_{\text{free}} \). A large volume of a visibility set makes it easier to find collision-free connections between samples, which results in a better coverage of the free configuration space. In narrow passages the volume of a visibility set is considerably lower, which results in fewer connections with other samples.

Problem implementation. Grasp executions will be simulated in geometry constrained scenes inside the Moverl! framework to retrieve data on planner performance. Non-optimizing planners are instructed to produce feasible paths. The optimizing planners are instructed to produce optimized paths. The motion constraint problem is defined to keep the gripper horizontal. Rotation of the gripper in the horizontal plane is allowed (max \( 2\pi \), other rotations are limited to \( 0.1\pi \)). Path simplification by OMPL for all the motion planning problems is turned on. Multi-query planners are being used as single-query
planners.

**Performance metric.** Solved runs, computing time and path length are used as metrics in our experiments. We analyze the metrics outcome individually to provide the best performing planners in each of the metrics. Solved runs is expressed in terms of the percentage of total runs of a planner resulting in feasible paths. High solved runs is considered as high performance. Computing time is measured for the time it takes for planners to produce feasible paths or optimized paths. Planners with a low median computing time and small interquartile range are considered as high performance. Path length is measured by the length of the sum of motions for a produced path. Planners with short median path length and small interquartile range are considered as high performance. Mean and standard deviation values of computing time and path length can provide extra information on the performance. Lower mean and standard deviations values stand for higher performance.

**Parameters.** In MoveIt! and OMPL, parameters can be set to increase the performance of the planners. We investigate the different parameter settings for each planner and for each manipulator to find suitable parameter values. This is done by increasing and decreasing the default settings by a factor of two. Parameter tuning was conducted again in the direction performance increase was noticed until no improvement was found. These new parameter values will be set before conducting the benchmarks.

IV. DEFINED MOTION PLANNING PROBLEMS

Three grasp execution motions have been defined to measure the performance of the planners. They are defined in the same environment that consists of a simplified shelf and obstacles.

A. Benchmark 1: Place motion obstacles

Benchmark 1 initial end-effector position is located at the end of a shelf, shown as the orange colored robot state in Fig. 1,2,3. The goal state is a stretched arm configuration in front of the shelf (gray robot state). The motion planning problem starts a in a narrow space, the goal is situated in a less constrained space. Meaning that $\mu(V(x_{\text{init}}))$ is considerably smaller compared to $\mu(V(x_{\text{goal}}))$.

B. Benchmark 2: Pick motion

Benchmark 2 is attempting to resemble a picking motion from a narrow shelf, shown as the gray robot state in Fig. 1,2,3. The goal of the problem is to achieve a specific gripper orientation at the end of this shelf (orange robot state). The planner will have to produce a motion plan with high accuracy to reach the end of the shelf. The motion planning problem starts in a less constrained space, the goal is situated in a narrow passage. Meaning that $\mu(V(x_{\text{init}}))$ is considerably bigger compared to $\mu(V(x_{\text{goal}}))$.

C. Benchmark 3: Place motion with motion constraints

For benchmark 3, the same motion planning problem as benchmark 1 is defined. However, for this problem motion constraints are added to keep the gripper horizontally leveled within a small margin. This resembles placing a glass of water from the shelf on the table, without spilling. The motion constraint reduces the free configuration space $C_{\text{free}}$, making it harder to find a feasible path.

V. PARAMETER SELECTION

For 20 of the 23 OMPL planners, parameters have to be set in order for the planner to solve a motion planning problem. Choosing parameter values can improve the performance of the planner. To aim for maximum performance an extensive parameter selection was conducted, since no automatic optimization process is available to this date.

**Process.** An iterative process was used to converge the parameter values to maximum performance, one by one. To have a faster iterative process, three versions of a planner were used. Each version had a different parameter value. For the first iteration, version one had a parameter value of default/2, version two default and version three default *2. If the default value is infinite or zero a substitute value was given. To gather reliable data on the performance, every version was run 50 times. The parameter values for the
### TABLE II: PLANNER PARAMETERS FOR UR5 (U), LWR 4+ (L) AND JACO (J)

<table>
<thead>
<tr>
<th>SBL</th>
<th>U</th>
<th>L</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>0.5 0.6 0.6</td>
<td>range</td>
<td>0.5 0.6 0.6</td>
</tr>
<tr>
<td>EST</td>
<td>0.05 0.05 0.075</td>
<td>goal bias</td>
<td>0.075 0.075 0.075</td>
</tr>
<tr>
<td>BiEST</td>
<td>0.05 0.05 0.075</td>
<td>goal bias</td>
<td>0.075 0.075 0.075</td>
</tr>
<tr>
<td>ProjEST</td>
<td>range</td>
<td>0.5 0.6 0.6</td>
<td>range</td>
</tr>
<tr>
<td>RRT</td>
<td>range</td>
<td>0.5 0.6 0.6</td>
<td>range</td>
</tr>
<tr>
<td>RRTConnect</td>
<td>range</td>
<td>0.5 0.6 0.6</td>
<td>range</td>
</tr>
</tbody>
</table>

**SBL**

<table>
<thead>
<tr>
<th>U</th>
<th>L</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>0.5 0.6 0.6</td>
<td>range</td>
</tr>
<tr>
<td>PRM</td>
<td>0.5 0.6 0.6</td>
<td>goal bias</td>
</tr>
<tr>
<td>LabsPRM</td>
<td>0.5 0.6 0.6</td>
<td>goal bias</td>
</tr>
<tr>
<td>RRT</td>
<td>range</td>
<td>0.5 0.6 0.6</td>
</tr>
<tr>
<td>KPIECE</td>
<td>0.5 0.6 0.6</td>
<td>range</td>
</tr>
<tr>
<td>BKPICE</td>
<td>0.5 0.6 0.6</td>
<td>range</td>
</tr>
<tr>
<td>LBKPIECE</td>
<td>0.5 0.6 0.6</td>
<td>range</td>
</tr>
</tbody>
</table>

**Selection**

The iterative process was conducted for each parameter and for each manipulator independently. The converged parameter values are presented in Tab. II. The global parameter `longest_valid_segment_fraction` was set to 0.005, this value was also retrieved through an iterative process.

**Range**

In the process of the parameter selection, sig-
significant performance changes were noticed when altering the range parameter. As described before this parameter specifies the maximum distance that a tree node is extended towards the sample. The distance has to be low enough to provide dense coverage of the (free) configuration space, resulting in higher solved runs and shorter paths. Computing time increases for lower distances since more samples are needed for total coverage.

VI. RESULTS

A. Methodology

The grasp executions are performed using one thread on a system with an Intel i5 2.70GHz processor and 8Gb of memory. To give reliable data on the solved runs, computing time and path length, each algorithm was run 50 times for the given motion planning problem. The planners were given a maximum computing time of 10s for benchmarks 1 and 2. Due to the more limited configuration space of motion constraint planning, benchmark 3 was given 20s maximum computing time. The time is kept low since most robotics applications need to operate quickly.

To solve the inverse kinematics of the manipulator for benchmark 3, the TRAC-IK solver is used. This inverse kinematics solver shows more success and better average speeds compared to other solvers [24].

B. Simulation results

Results of benchmark 1 are shown in Fig. 4 and Tab. III. This motion planning problem starts planning with low visibility and plans towards a set with greater visibility. Considering all manipulators, solved runs of 80% and higher were found for all single-query planners, except for FMT. Since multi-query planners do not focus on one specific motion planning problem, the roadmap construction needs to cover the whole free. A demerit of this is the extra needed computing effort. Single-query planners do not need to cover the total free configuration space. Heuristics of these planners help propagating a path outwards of a set with low visibility. Lowest computing times were retrieved with EST, ProjEST, KPIECE and STRIDE, which are all mono-directional planners with a goal bias property. Since the goal configuration is located in a large visibility set, the probability of finding a solution
with the goal configuration as sample increases. However, a sample of the roadmap/path needs to be present in the visibility set of the goal configuration \( (p(x) \in V(x_{goal})) \). The fastest planners also use heuristics to quickly cover the configuration space. EST and STRIDE do this by looking at the density of present samples. KPIECE uses a discretization layer which coarsely covers the configuration space. With a goal bias and the presence of the large goal visibility set, short paths can be found with EST, ProEST and KPIECE.

When considering the results of manipulators in benchmark 1 separately, it can be noted that the JACO manipulator was able to generate higher solved runs for multi-query planners. This manipulator does not incorporate any restrictions on joint limits, which helps to find more connections in the configuration space between vertices (greater \( \epsilon \) compared to UR5 and LWR 4+), increasing the solved runs. Moreover, a solution faster is found faster and with a shorter path length. Computing times for the UR5 manipulator were lower with KPIECE, RRT and RRTConnect. Computing times for the LWR 4+ were lower than the UR5 with SBL, EST, KPIECE, BKPIECE, LBKPIECE and STRIDE. In addition to those planners, RRTConnect also had low computing times for the JACO. For the UR5, short path lengths are found with BiEST, ProEST, KPIECE, RRTConnect, TRRT and BiTRRT. For the LWR 4+, these are EST, KPIECE, BKPIECE, LBKPIECE and LazyPRMstar. Planners EST, BiEST, ProEST, KPIECE, LazyPRMstar and BiTRRT found short paths for the JACO manipulator.

Results of benchmark 2 are shown in Fig. 5 and Tab. IV. This motion planning problem starts planning in a set with great visibility and plans towards a set with a low visibility. Considering all manipulators, solved runs of 80% and higher were retrieved with SBL, BKPIECE, LBKPIECE, RRTConnect and BiTRRT. These are all bi-directional tree-based planners. Because of this property path planning is also started in the goal state, which acts similar to the benchmark 1 (from low visibility to a great visibility). Mono-directional planners are not able to find connections with samples in the visibility set of \( x_{goal} \), due to the reason that the planners with goal bias also did not find feasible paths. SBL and LBKPIECE showed the lowest computing times. These planners use lazy collision-checking. Collision-checking is only performed on a candidate path instead on all vertices, which can result lower the computing time. Shortest paths are found with SBL and LBKPIECE, however, standard deviations are higher for the SBL planner. Using a discretization layer to cover the configuration space coarsely helps finding more consistent results (lower standard deviations from the mean).

Individually, the UR5 and LWR 4+ also managed to produce solved runs of 80% and higher for the BiEST planner. This planner is also a bi-directional planner. However, this planner needs more computing time to solve the motion planning problem, which explains why it was not able to produce high solved runs for the JACO manipulator. BiEST looks at the density of samples in its neighborhood to help its expansion, since the configuration space of the JACO manipulator is bigger it will cover the unnecessary configuration space with samples. This increases the needed computing time. The expanded configuration space of the JACO manipulator helps to produce solved runs of 80% and higher for planners PRM, PRMstar and LazyPRMstar. However, computing times are considerably higher since these planners keep optimizing the roadmap until the time-limit is reached. RRTConnect is the fastest planner for the UR5 and LBKPIECE is the fastest for LWR 4+ and JACO. For all three manipulators, LBKPIECE is able to compute feasible paths within 1.5s. Considering high solved runs: BiTRRT finds the shortest paths for the UR5, SBL for the LWR 4+ and FMT for the JACO respectively.

Results of benchmark 3, incorporating motion constraints, are shown in Fig. 6 and Tab. V. This motion planning problem incorporates a motion constraint, which drastically constraints the free configuration space. Considering all manipulators, only the BiEST planner was able to produce solved runs of 80% and higher. The planner looks at the density of samples in its neighbourhood to help its expansion. In the case of planning with motion constraints, this shows to be effective compared to the other 20 planners. The bi-directional property helps finding a solution within the time limit, since the mono-directional EST planner is not able to find a feasible path with a maximum computing time of 20s.

Depending on the manipulator, the highest performing planner differs, which indicates that there is less consistency in planner performance when incorporating motion constraints. SBL, BKPIECE and RRTConnect also had solved runs of 80% and higher for the UR5 manipulator. Orientating the end-effector of the UR5 will result in less self-collisions compared to the other two manipulators, since three degrees of freedom are protruding from the main linkages. This helps to orientate the end-effector independently of the other three degrees of freedom. For the JACO manipulator the KPIECE planner manages to get solved runs of 100%, which is due to the increased configuration space of the JACO manipulator.
TABLE IV: MEAN VALUES FOR BENCHMARK 2

<table>
<thead>
<tr>
<th>Planner name</th>
<th>UR5</th>
<th>LWR 4+</th>
<th>JACO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>Path length</td>
<td>Time (s)</td>
<td>Path length</td>
</tr>
<tr>
<td>EST</td>
<td>8.89 (4.70)</td>
<td>13.03 (4.65)</td>
<td>8.00 (0.00)</td>
</tr>
<tr>
<td>BiEST</td>
<td>2.08 (0.23)</td>
<td>28.18 (0.95)</td>
<td>2.57 (0.00)</td>
</tr>
<tr>
<td>PojST</td>
<td>9.08 (4.15)</td>
<td>26.58 (2.90)</td>
<td>9.09 (0.58)</td>
</tr>
<tr>
<td>PojST</td>
<td>7.59 (0.83)</td>
<td>29.12 (5.76)</td>
<td>9.14 (0.32)</td>
</tr>
<tr>
<td>BKPIECE</td>
<td>11.21 (7.49)</td>
<td>37.00 (2.52)</td>
<td>11.01 (0.00)</td>
</tr>
<tr>
<td>BiTRRT</td>
<td>10.27 (0.35)</td>
<td>17.28 (2.37)</td>
<td>10.19 (0.04)</td>
</tr>
<tr>
<td>BiTRRT</td>
<td>10.21 (2.09)</td>
<td>19.41 (1.71)</td>
<td>10.71 (0.00)</td>
</tr>
<tr>
<td>TRRT</td>
<td>4.47 (0.57)</td>
<td>6.18 (12.85)</td>
<td>8.55 (0.35)</td>
</tr>
<tr>
<td>TRRT</td>
<td>10.15 (0.00)</td>
<td>23.79 (0.00)</td>
<td>10.07 (0.02)</td>
</tr>
<tr>
<td>BKPIECE</td>
<td>3.52 (0.83)</td>
<td>14.90 (21.17)</td>
<td>3.87 (0.83)</td>
</tr>
</tbody>
</table>

Standard deviation in parentheses

Gray cells = time within 2 minau and path length within 1.2 · path length for solved runs > 80%

TABLE V: MEAN VALUES FOR BENCHMARK 3

<table>
<thead>
<tr>
<th>Planner name</th>
<th>URS</th>
<th>LWR 4+</th>
<th>JACO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>Path length</td>
<td>Time (s)</td>
<td>Path length</td>
</tr>
<tr>
<td>EST</td>
<td>5.88 (3.90)</td>
<td>26.88 (5.69)</td>
<td>8.00 (0.00)</td>
</tr>
<tr>
<td>BiEST</td>
<td>2.08 (0.23)</td>
<td>28.18 (0.95)</td>
<td>2.57 (0.00)</td>
</tr>
<tr>
<td>PojST</td>
<td>9.08 (4.15)</td>
<td>26.58 (2.90)</td>
<td>9.09 (0.58)</td>
</tr>
<tr>
<td>PojST</td>
<td>7.59 (0.83)</td>
<td>29.12 (5.76)</td>
<td>9.14 (0.32)</td>
</tr>
<tr>
<td>BKPIECE</td>
<td>11.21 (7.49)</td>
<td>37.00 (2.52)</td>
<td>11.01 (0.00)</td>
</tr>
<tr>
<td>BiTRRT</td>
<td>10.27 (0.35)</td>
<td>17.28 (2.37)</td>
<td>10.19 (0.04)</td>
</tr>
<tr>
<td>BiTRRT</td>
<td>10.21 (2.09)</td>
<td>19.41 (1.71)</td>
<td>10.71 (0.00)</td>
</tr>
<tr>
<td>TRRT</td>
<td>4.47 (0.57)</td>
<td>6.18 (12.85)</td>
<td>8.55 (0.35)</td>
</tr>
<tr>
<td>TRRT</td>
<td>10.15 (0.00)</td>
<td>23.79 (0.00)</td>
<td>10.07 (0.02)</td>
</tr>
<tr>
<td>BKPIECE</td>
<td>3.52 (0.83)</td>
<td>14.90 (21.17)</td>
<td>3.87 (0.83)</td>
</tr>
</tbody>
</table>

Standard deviation in parentheses

Gray cells = time within 2 minau and path length within 1.2 · path length for solved runs > 80%

C. Discussion

The motivation of this work is to help users pick high-performing motion planners for grasp executions. Shortcomings of this work will be discussed.

Computing time. This paper only showed results for motion planning with a time-constraint of 10s or 20s. This was chosen to select high-performing motion planners that find a solution in a timely manner. Selecting a higher time-limit could show improved performance for some planners. However, this is not covered in this work.

Parameter selection. Since parameter values have to be set for a planner to operate, the aim was to achieve maximum performance of the planners. By manually conducting the iterative process explained before, a guarantee of maximum performance cannot be given. We executed the iterative process to the best of our abilities in order to achieve maximum performance. For planners with an exposed range parameter, the distance has to be low enough to provide dense coverage of the configuration space.

Multi-query. For this paper, we only considered single-query motion planning performance. This paper fails to show the potential benefit of lower computing times when using the same roadmap multiple times. For benchmark 1 and 2, we do notice that planners single-query planners are able to compute feasible paths in short amount of time. We therefore argue the need for multi-query planners for online grasp executions similar to benchmarks 1 and 2. The use of multi-query planners can be beneficial when motion constraints have to be used all the time. Computing paths for these motion planning problems can consume more time, as shown in the results. Using the same roadmap again will decrease computing time. However, this map needs to be detailed and the environment needs to be static.

Optimization with time-invariant goal. BiTRRT is the fastest optimizing planner. However, compared to non-optimizing planners, the path length is not consistently shorter. More research needs to be conducted to show the real potential of optimizing planners.

Manipulators. The manipulators studied in this paper have similar specifications. The effect of different manipulator on planner performance cannot be verified with the presented benchmark data. We believe similarly shaped manipulators will yield a similar planner choice. Best overall planner performance, with respect to solved runs, can be obtained with a JACO manipulator.

BFMT and LBTRRT. These planners resulted in errors for the defined motion planning problems. We were unable to provide reliable results to present in this paper. More effort is needed to make these planners work more reliable.

VII. CONCLUSION

This paper presented benchmark data for 21 of the current 23 OMPL planners in MoveIt! for three different manipulators. This data can be useful when performing similar grasp executions. Simultaneously, this paper selected high-performing planners for different motion planning problems, resembling a grasp execution. Planner performance was studied by means of solved runs, computing time and path length. The results showed that the mono-directional BKPIECE planner was highest performing when initiating motion planning from a constrained configuration towards a less constrained space. Bi-directional planners with lazy collision-checking (SBL and LBKPIECE) showed fastest performance when the goal configuration is located within a constrained space. Due to better coverage of the configuration space by using a discretization layer, shorter paths were found with LBKPIECE. For motion planning problems incorporating a motion constraints, consistent performance over the three manipulators was retrieved with BiEST. Considering all the grasp executions presented in this work, RRTConnect was the most reliable planner due to high solved runs. For future work we would like to investigate the performance of a mono-directional tree-based planner with goal bias and lazy collision-checking.

REFERENCES

Fig. 6. Results for benchmark 3. (a) Solved runs; higher is better. (b) Computing time; lower is better, small interquartile range is better. (c) Path length; lower is better, small interquartile range is better.


Grasp execution in real-world setup

Grasping approaches typically do not provide any information about how a grasp execution is realized [34]. An explanation on how to configure/design a grasping pipeline could serve as an example to fill the gap of grasp execution in existing grasping approaches. In this way also difficulties can be discovered to assist with the development of better grasping approaches. At the same time, with the use of the real-world setup, motion planning can be performed with a high-performing planner identified in chapter 3. Possible performance decrease can be determined in this way.

Only a UR5 manipulator can be used to present a full grasping pipeline in the real-world setup due to access. Other limitations to the grasping pipeline will be the use of one RGB-D camera, the ASUS Xtion Pro Live, to provide information about the object and environment.

4.1. Setup

In this section the parts needed to attempt a real-word grasp execution are defined. These parts feed information to the `move_group` node of MoveIt! to plan in a geometrically constrained environment.

In figure 4.1 a high level setup of a full grasping pipeline is shown. This setup only uses data from an RGB-D camera. The device driver OpenNI2\(^1\) exposes the data from the camera by publishing several image and pointcloud topics. The pointcloud can be used by the `move_group` to provide information about the environment to prevent collisions when motion planning. The pointcloud also has to be used by an object segmentation application. This application removes pointcloud points that are not part of the to-be-grasped object, the output can be used by the grasping approach. By using smart heuristics, grasping approaches try to find grasp poses on the object pointcloud. Once a suitable grasp is found, motion planning can be conducted. This can be done by having an application that communicates with the `move_group`, this application is noted as the grasp execution command in figure 4.1. The first part of this application commands `move_group` to find a robot state that satisfies the grasp pose by computing Inverse Kinematics (yellow arrow). In turn, the `move_group` will let the grasp execution command know whether a suitable state is found (green arrow) and continues by giving the `move_group` a command to plan a motion path. This command also specifies which planner should be used and what the maximum computing time is. The `move_group` is in constant communication with the UR5 controller, which mostly provides information about the joints (blue arrow). The UR5 controller retrieves commands from the `move_group` to perform a certain task (red arrow).

4.1.1. Object segmentation

The grasping of objects usually deals with finding stable grasps on the pointcloud of the object. This data can for instance be retrieved through a 3D-camera. However, it is needed to isolate the pointcloud of the object from the scene. It needs to be determined which points in the pointcloud make up the object. A widely used method to isolate the object is by considering that the unknown object is placed on a flat surface. A RANSAC (RANdom SAmple Consensus) search can be conducted on the available pointcloud in order to specify this

\(^1\)https://github.com/OpenNI/OpenNI2
flat surface. Once known, the points outside of this flat surface can be deleted except for the points that are above this flat surface, what is left is the object point cloud. The shortcomings of this is that the object will not be found when it is not located on a flat surface. Moreover, obstacles that are present on the table will be taken as part of the object.

Another approach is to look at the difference between two scenes, one without the unknown object and one with the unknown object. For this approach to work, it needs to be given a static pointcloud of the scene which can be obtained before doing any grasping. The pseudo-code for this algorithm is shown in algorithm 4. For the ROS implementation, a C++ code has been written which can be found in appendix E.1.

**Algorithm 4** Determine if point of real-time pointcloud is part of object

Require: pointcloud\textsuperscript{Real-time}, pointcloud\textsuperscript{Static}

1: point\textsubscript{Real-time}\textsuperscript{i(\textx,\texty,\textz)} \in Pointcloud\textsuperscript{Real-time}
2: point\textsubscript{Static}\textsuperscript{i(\textx,\texty,\textz)} \in Pointcloud\textsuperscript{Static}
3: for \(i = 1\ldots\text{Pointcloud\textsuperscript{real-time} (size)}\) do
4: if point\textsubscript{Real-time}\textsuperscript{i(\textx,\texty,\textz)} \neq point\textsubscript{Static}\textsuperscript{i(\textx,\texty,\textz)} \pm \text{margin} then
5: point\textsubscript{Real-time}\textsuperscript{i(\textx,\texty,\textz)} = point\textsubscript{Unknown object}\textsuperscript{i(\textx,\texty,\textz)}
6: end if
7: end for
8: return pointcloud\textsuperscript{Unknown object} \hspace{1cm} \triangleright \text{Pointcloud only consisting of object points}

**4.1.2. Calibration**

Calibration is an important part of real-world grasp execution. Two groups of calibration need to be performed, intrinsic and extrinsic calibration.

**Intrinsic calibration**

Intrinsic calibration of the camera improves the accuracy of the depth images retrieved from the RGB-D camera internally. This is done by specifying improved intrinsic parameters of the IR (depth) and RGB cameras, including focal length and the distortion model. Intrinsic calibration can be performed with the camera\_calibration\textsuperscript{2} package available in ROS. This package uses OpenCV\textsuperscript{3} to find a specified checkerboard pattern in the image topic, shown in figure 4.2. By holding the checkerboard pattern in several areas of the image.

---

\textsuperscript{2}http://wiki.ros.org/camera\_calibration
\textsuperscript{3}http://wiki.ros.org/opencv3
field the package can find distortions in the image and can provide proper depth information for the IR image. Once enough samples are gathered the calibration file can be saved to the openni2 package, which will load the calibration file automatically.

![Intrinsic calibration using a checkerboard to provide better depth accuracy](image)

**Figure 4.2: Intrinsic calibration using a checkerboard to provide better depth accuracy** [35]

### Extrinsics calibration

Extrinsics calibration is necessary for specifying where the camera lives inside the scene, in other words the transformation of the camera to the robot frame. To find this transformation many methods can be used. A simple method would be to manually measure the translation and rotation of the camera with respect to the base of the robot. However, it is hard to retrieve high accuracy of the measurements without having an accurate build frame that houses the camera and robot. Another possible method is finding known patterns in an infrared (IR) or RGB (Red, Green, Blue) image, for instance when using a checkerboard or AR markers. Found patterns can be used to estimate the transformation, but when using cheaper RGB-D cameras (like the Kinect\(^4\) or Xtion Pro Live\(^5\)), estimations are not accurate enough to provide reliable grasps. To overcome this problem, multiple patterns can be used to decrease the error of the estimation. The ar_track_alvar\(^6\) package can be configured to find bundles of AR markers, when placing a sheet of AR markers at a known distance from the base of the robot a more accurate transformation can be found. This is shown in figure 4.3.

### 4.1.3. Environmental constraints

In order for the move_group to plan collision-free motion plans it should know the environment it is planning in. Environmental information, like a collision map, is stored in the planning_scene topic. Using the GUI interface, RVIZ\(^7\), users can manually add collision objects or a representation of the scene. However this scene needs to be static and effort has to be put in creating an accurate representation of the scene. Another more accurate and easier method uses data depth data from a sensor to represent the scene in an so-called octomap. MoveIt! has two plugins that can create an octomap from either depth images or pointclouds, Depth Image Occupancy Map Updater\(^8\) and PointCloud Occupany Map Updater\(^8\) respectively. The octomap uses hierarchical data structures, octrees, which describe the state of a volume of space. A voxel is a node in the octree that represents the space contained in a cubic volume. The volume of a node can be subdivided into eight (latin: octo) sub volumes, which can repeatedly be subdivided until a minimum voxel size is reached. The accuracy of the map depends on the minimum voxel size, which has to be set at a higher level. Smaller voxel size yields more computational effort. In a dynamic environment the octomap needs to be refreshed, which increases computational effort drastically. Therefore it was decided to update the octomap every 10s in the presented grasp execution. Finding a method to dynamically represent the octomap is an aspect that can be investigated in future work. In figure 4.4, the octomap representation of a scene is shown.

\(^5\)https://www.asus.com/nl/3D-Sensor/Xtion_PRO_LIVE/  
\(^6\)http://wiki.ros.org/ar_track_alvar  
\(^7\)http://wiki.ros.org/rviz  
\(^8\)http://docs.ros.org/indigo/api/pr2_moveit_tutorials/html/planning/src/doc/perception_configuration.html
(a) Extrinsic calibration, as presented in RVIZ

(b) Picture of setup in the real-world

Figure 4.3: Execution of extrinsic calibration using AR markers, necessary for grasp execution

Figure 4.4: Octomap representation of a table with objects and obstacles

4.1.4. Grasp execution command

This section can also be part of the grasping approach, though as explained earlier, most grasping approaches do not consider the grasp execution. In order to execute a grasp, commands have to be given to the move_group. The grasp pose can be given directly to the move_group, which will subsequently perform inverse kinematics in order to find a suitable robot state that fulfills the grasp pose. When successful it will plan and execute a motion path for the robot. However, when a valid robot state cannot be found the grasping pipeline stops. Therefore, a more reliable method would be a two-way communication between the grasp execution command and grasping approach. When no valid robot state can be found, the grasping approach is attempted again to find a different grasp. Once a valid robot state is found for a grasp, the grasp execution command can command the move_group to plan and execute a motion path. This command contains planning related settings, for example the maximum computing time, the motion planner and tolerances.
4.2. Grasp execution in real-world setup

The sections in the previous part together make up the setup for a grasping pipeline in order to perform a grasp execution. In this section the use of a high-performing motion planner in the real-world setup is demonstrated in several user cases. To provide information about computing time, the RVIZ GUI and the terminal window are used. Since the grasping pipeline is not hooked up to the benchmark tool, the path length is not computed.

4.2.1. Discussion on the difficulties in real-world grasp executions

Performing grasp executions in the a real-world setup introduces challenges that were not present in simulation environments. This part will highlight the current difficulties in grasp executions. Possible solutions will be discussed.

RGB-D camera
Using a relatively cheap RGB-D (Red, Green, Blue - Depth) camera, as the ASUS Xtion Pro Live, can give inaccurate pointcloud data due to the low resolution and it being not robust in various lighting situations. The depth is measured by an infrared sensor which initially did not give an accurate representation of the scene. Moreover, the infrared sensor is prone to noise from reflective surfaces which can cause problems like misrepresentations of the environment and wrongly picked unknown objects. By obtaining a high-end 3D camera these difficulties can be decreased.

Calibration
One of the most important parts in a real-world grasping pipeline is the extrinsic calibration of the 3D camera position with respect to the scene/manipulator. A wrongly picked transformation can result in highly inaccurate grasps. Small translation errors can result in an offset for the actual grasp. More important is the orientation error, which is hard to establish. Using multiple AR markers as shown in this chapter helps to provide better accuracy on the camera’s orientation. However, since the length of the arm is 850mm the error of the orientation in one direction by 1 degree can offset the end-effector by 14.8mm. Using multiple AR markers decreases the error as well, however, in the created setup error were in the margin of 10mm. To prevent possible errors, the camera has to be mounted on a fixed structure with accurately known dimensions, with this the translation can accurately be measured.

Representation of environmental constraints
In section 4.1.3 it was explained how a pointcloud is transformed to cubic volumes that represent the environment. However, since only one RGB-D camera there is an occurrence of occlusions, which means no data is missing due to the position of the camera. If a camera was pointed towards an object no information about the back of the object can be given. To effectively represent the environmental constraints the RGB-D camera needs to have a partial or full overview of the manipulator’s configuration space.

In figure 4.5 the occlusions in the obstacles are visible. A problem that can arise is that a motion planner can find a path through the occluded space, causing undesirable collisions. However, the probability of finding a motion plan through the occluded space (being narrow) will be lower than finding a motion plan around the obstacles when choosing a planner that plans in expansive free configuration spaces (EST) or when choosing a planner that coarsely covers the configuration space. (KPIECE-family). Choosing an optimization planner can cause issues due to the possibility of finding optimal paths through the occluded space.

Safe grasp execution command
Problems with an automatic grasp execution command is the lack of motion plan acceptance. Using the RVIZ visualizer a computed motion plan will be simulated, this motion plan can then be accepted and executed after a visual check by the user. This prevents any wrongly calculated motion plans due to missing environmental data caused by noise or occlusions. In human environments extra steps need to be taken in order for automatic motion plan executions, either by a system that keeps track of the humans in the environment or a system that verifies computed motion plans on safety. The grasp executions performed in this thesis were first checked by the visual representation in RVIZ.

4.2.2. Geometrically constraint motion planning
The environment the robotic arm is moving in is comprised of a table with several household objects. In order to have a more complex motion plan the end-effector of the robot needs to be placed among several obstacles. Thus going towards motion planning in a narrow-passage. From the results in chapter 3 it was found
that LBKPIECE planner was the most suitable for this sort of motion planning problem. The bi-directional property will simultaneously plan from the goal configuration state, which will overcome the problem of not finding states inside the visibility set of the goal.

The configured grasp execution in figure 4.6 resembles the pick and place grasp execution the computing times were evaluated. The planners SBL, BKPIECE, LBKPIECE, RRTConnect and BiTRRT have been chosen to execute the grasps, since these planners showed high performance for the UR5 in the work presented in appendix A. Note that processes like the octomap are running in the background as well, which can cause a decrease in performance. The grasp execution is performed 10 times to provide better saturation of the results. The times are noted in table 4.1.

Table 4.1: Mean computing times for grasp execution planning in the real-world, for grasp execution shown in figure 4.6

<table>
<thead>
<tr>
<th>Planner name</th>
<th>Mean computing time (s)</th>
<th>Pick</th>
<th>Place</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBL</td>
<td>3.338</td>
<td>5.495</td>
<td></td>
</tr>
<tr>
<td>BKPIECE</td>
<td>7.243</td>
<td>6.104</td>
<td></td>
</tr>
<tr>
<td>LBKPIECE</td>
<td>2.755</td>
<td>3.765</td>
<td></td>
</tr>
<tr>
<td>RRTConnect</td>
<td>2.752</td>
<td>1.973</td>
<td></td>
</tr>
<tr>
<td>BiTRRT</td>
<td>2.165</td>
<td>2.001</td>
<td></td>
</tr>
</tbody>
</table>

From the results it can be noted that computing times are significantly higher than the computing times of similar grasp execution in chapter 3. Similarly for the grasp execution in appendix A. This is due to the extra processes that are running at the same time as the motion planning. Representing a scene with the use of an octomap is needs high computing effort.

The execution of the grasp, with a feasible path found by LBKPIECE (shown to have highest performance in chapter 3), is shown in figure 4.6, a more detailed sequence of images is shown in figure D.1. It can be seen that a collision-free path was found and no jerky motion are visible.

4.2.3. Grasping objects using C-shape configuration

In the work of Lei et al. [36] the created grasping pipeline was used to provide information of the unknown object data as well as the execution of the grasp after a grasp was found. The grasping approach uses a C-shape configuration, that resembles a gripper, to find stable grasps on the unknown object. This grasp finding approach was inserted at the grasping approach box shown in figure 4.1. The motion planner BiTRRT was picked due to the extra optimization step. Using the grasping pipeline the objects could be picked as shown in figure 4.7.
4.2. Grasp execution in real-world setup

Figure 4.6: Execution of motion plan using LBKPIECE

(a) Initial state  (b) State during motion  (c) Goal state

Figure 4.7: Grasp execution with grasping approach of Lei et al. [36] in the real-world setup of this thesis [36]

<table>
<thead>
<tr>
<th>Object name</th>
<th>Cleaner spray bottle</th>
<th>Electric drill</th>
<th>Spray can</th>
<th>Elephant</th>
<th>Coffee jar</th>
<th>Teddy bear</th>
<th>Milk carton</th>
<th>Wineglass</th>
<th>Shampoo bottle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial setup</td>
<td><img src="image1" alt="Initial setup" /></td>
<td><img src="image2" alt="Initial setup" /></td>
<td><img src="image3" alt="Initial setup" /></td>
<td><img src="image4" alt="Initial setup" /></td>
<td><img src="image5" alt="Initial setup" /></td>
<td><img src="image6" alt="Initial setup" /></td>
<td><img src="image7" alt="Initial setup" /></td>
<td><img src="image8" alt="Initial setup" /></td>
<td><img src="image9" alt="Initial setup" /></td>
</tr>
<tr>
<td>Example grasp load</td>
<td><img src="image10" alt="Example grasp load" /></td>
<td><img src="image11" alt="Example grasp load" /></td>
<td><img src="image12" alt="Example grasp load" /></td>
<td><img src="image13" alt="Example grasp load" /></td>
<td><img src="image14" alt="Example grasp load" /></td>
<td><img src="image15" alt="Example grasp load" /></td>
<td><img src="image16" alt="Example grasp load" /></td>
<td><img src="image17" alt="Example grasp load" /></td>
<td><img src="image18" alt="Example grasp load" /></td>
</tr>
<tr>
<td>Grasp execution</td>
<td><img src="image19" alt="Grasp execution" /></td>
<td><img src="image20" alt="Grasp execution" /></td>
<td><img src="image21" alt="Grasp execution" /></td>
<td><img src="image22" alt="Grasp execution" /></td>
<td><img src="image23" alt="Grasp execution" /></td>
<td><img src="image24" alt="Grasp execution" /></td>
<td><img src="image25" alt="Grasp execution" /></td>
<td><img src="image26" alt="Grasp execution" /></td>
<td><img src="image27" alt="Grasp execution" /></td>
</tr>
<tr>
<td>Objects grasped</td>
<td>Points: 10596, 9929, 7127, 8044, 4345, 4857, 5589, 3503, 5267</td>
<td>Time (s): 1.74, 1.56, 0.91, 1.96, 0.68, 1.82, 0.64, 0.53, 0.67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.3. Conclusion

In this chapter a grasping pipeline was designed to be able to carry out grasp executions in a real-world setup. Important elements that needed to be defined were the calibration, environmental constraints and execution command. Difficulties in real-world grasp executions were found in the accuracy of RGB-D cameras, the extrinsic calibration, representation of environmental constraints and the safety of automatic grasp executions. These difficulties are discussed and potential solutions were given. The real-world experiments showed planners SBL, BKPIECE, LBKPIECE, RRTConnect and BiTRRT were able to produce feasible paths in various cases. These planners showed high performance in the place grasp executions, shown in chapter 3 and A. However, a performance decrease in computing time was noticed due to the decrease in computing power availability as multiple processes need to run simultaneous with the motion planning.
In this chapter the results of the presented work in this thesis will be discussed. The discussion part is divided into several sections, each discussing different aspects of the presented work.

### 5.1. High-performing planners

This section will mainly discuss the work for selecting high-performing planners for grasp execution. Since chapter 3 already has a discussion section most parts will be similar.

**Computing power**

The computing time metric is relative since different results can be obtained using different computing power. To present useful results, the motion planning problems were solved by using the same hardware equipment with identical computing power allocations. With higher computing power the computing time will decrease, in turn this can increase the solved runs metric. However, already high-performing planners will further increase in performance as well.

**Parameter selection**

Parameter selection is needed to provide the best possible results for the planners. In this thesis, the parameter selection has been conducted once for the UR5 manipulator in chapter A and once for every manipulator in chapter 3. Performing parameter selection for every benchmark problem will not show the consistency of the planner performance over various motion planning problems. Moreover, the benchmarking problems are carried out in similar scenes. Performing parameter selection is a highly time-consuming task due to the many different parameters for every planner. Performing parameter selection before every single grasp execution would defeat the purpose of online motion planning.

**Metrics**

In this thesis the planner’s performance was measured using solved runs, computing time and path length. Solved runs shows the performance of finding a solution. In robotic applications, time is frequently a reason for improvement. Moreover, if a motion is long the execution time will also be higher. Therefore, computing time should be as low as possible and path length is desired to be as short as possible. More metrics could have been chosen to investigate the performance. Metrics like path smoothness and path clearance can also be chosen as performance metrics. Path smoothness can be useful to select planners that refrain from jerky motions which can be present in sampling-based motion planning due to the randomness of the tree computed by a non-optimizing planner. Path clearance could help select planners that have higher cost efficiency. Planners that use smaller path clearance will likely produce more optimal paths. These metrics were not chosen due to, what we believe, their insignificant impact on planner choice. Jerky motions are not ideal, but it will not drastically affect the path length or computing time, similarly for path clearance.

**Multi-query**

Only single-query motion planning performance was considered in this work. It fails to show the potential benefit of lower computing times when using the same roadmap multiple times, when using multi-query
motion planners. In the defined grasp executions it is noticed that feasible paths can be found with single-query motion planners in short amounts of time. Moreover, service robots would be of assistance to humans when they can operate in unknown environments, making it hard to obtain a roadmap for grasps. The need for multi-query motion planners can therefore be argued when carrying out grasp executions.

**Robot types**

Planners were initially selected using one manipulator, the UR5. To verify whether the results are also viable for other manipulators, more benchmarks were conducted using different manipulators. However, these manipulators have similar specifications. They can be implemented in human environments without fences, due to their light weight. This thesis does not provide benchmark data for manipulators with radical different shapes. So it can not be verified if the selected planners also show high performance for these manipulators.

**5.2. Real-world setup**

This section will mainly discuss the work about the real-world setup presented in chapter 4.

**Difficulties**

The difficulties discussed that arise in real-world setups depend on the chosen equipment. It was discussed that a cheap RGB-D camera, like the Kinect and ASUS Xtion Pro Live, presents noise in the data. Also these cameras can give inaccurate depth information without proper calibration, using high-end equipment can eliminate inaccuracies in real-world setups. This is similar for extrinsic calibration. When using an accurately build frame for the camera and robot, the extrinsic calibration can be conducted more accurately since all the dimensions can be accurately calculated. However, the outline of these difficulties could still provide new users meaningful insight in performing real-world grasp executions.

**Planner performance**

Since the real-world grasp execution is not configured with a benchmark tool, no information of the path length could be provided. Moreover, the amount of runs was reduced to 10 due to time constraints. This will provide results that are not comparable to the work presented in chapter 3.
Conclusion and future work

6.1. Conclusion
To grasp an object, a robotic arm has to perform movements to get to the goal configuration state that satisfies the grasp. These movements need to be planned. In ROS, motion planning can be performed using MoveIt!, a framework dealing with robot manipulation. This framework is by default configured with OMPL, which houses state-of-the-art sampling-based motion planners. These planners are widely used due to their minimal computing effort in high-dimensional motion planning problems and can provide probabilistic completeness. To execute the motion of a grasp in a real-world setup, elements like execution command and calibration procedure need to be determined as well. Existing grasping approaches fail to enclose the difficulties of doing this. The goal of this thesis was to select high-performing OMPL motion planners available in MoveIt! for grasp execution and demonstrate such an execution in a real-world setup, outlining difficulties and changes in planner performance.

Performance of the motion planners was measured by means of solved runs, computing time and path length. Three manipulators were used for multiple grasp executions to generate results. Therefore the outcome is less dependent on the configuration space of a manipulator. To execute the planners of OMPL, parameters have to be set. An extensive parameter selection protocol has been followed to aim for maximum performance of the planners.

Moving the manipulator from a geometrically confined space towards a less confined space, in other words when the free space of the goal region is bigger than the free space of the start region, best performance was achieved with mono-directional tree-based planners. The KPIECE planner showed highest performance in computing time and path length. The discretization layer of this planner estimates the coverage of the state space which contributes to quickly and efficiently generating a motion-path out of constrained spaces.

Planner performance varies when a manipulator moves into a geometrically confined space from a less confined space, in other words, when the free space of the goal region is smaller than the free space of the start region. Best performance is achieved with a bi-directional tree-based planner with lazy-collision checking, either SBL or LBKPIECE. Mono-directional planners need increased effort in finding suitable configurations to enter the goal region. Bi-directional planners overcome this problem by simultaneously planning from the goal configuration. Incorporating lazy collision-checking helps to quickly find a candidate solution, since it initially accepts invalid configuration states. Only changes have to be made to the invalid parts of the candidate solution, which makes it run faster compared to other planners.

BiTRRT is the only optimizing planner of the five high-performing bi-directional planners. Since LBKPIECE was the highest performing planner, the stochastic optimizing step of BiTRRT did not result in shorter paths. However, computing times of the planner were only significantly higher for the LWR 4+ manipulator. This shows that the optimizing step of BiTRRT does not considerably increase the computing effort.

In addition to the geometrically constraint environment motion planning problems, one motion planning problem added orientation constraints to keep the end-effector horizontally leveled. Such a motion or path constraint narrows the free space ($C_{free}$) in the configuration space, C. Only the BiEST planner was able to produce solved runs higher than 80% for all manipulators. The planner looks at the density of samples in its neighborhood to help its expansion. In the case of planning with path constraints, this shows to be effective compared to the other investigated planners. The bi-directional property helps finding a solution within the
time limit, since EST is unable to find a solution in this amount of time.

Overall highest performance with respect to solved runs for all motion planning problems and considering the three manipulators is achieved with RRTConnect. This planner quickly covers the free configuration space due to its bias to unexplored spaces and due to its bi-directional property. Solution to the motion planning problems is therefore found within the short maximum planning time it was given (10s).

To execute a grasp in a real-world setup a grasping pipeline needed to be designed. This pipeline connects elements like data acquisition from sensors, grasping approaches and MoveIt!. The difficulties in establishing a stable grasp were mainly the noise of RGB-D camera, the extrinsic calibration and the use of an automatic grasp execution command. Solutions to these difficulties were discussed.

The designed grasping pipeline was used to perform grasp executions for several cases. Using planners SBL, BKPIECE, LBKPIECE, RRTConnect and BiTRRT for a picking grasp resulted in a decrease in computing time in the real-world setup. This is due to the increased demand for computing power by other processes that operate simultaneously in real-world setups.

6.2. Future work

During the development of this thesis, interesting topics for future work were exposed. These will be presented below in separate parts.

• **Automatic parameter selection:**

  Future work, that would benefit the motion planning regime in MoveIt! a lot, would be to design a tool that automatically picks the best parameter values for the motion planners. We think this could be achieved by implementing machine learning algorithms when performing benchmarks in the environment provided by the user. This will potentially save the user a lot of time. Currently there does not exist a good method for selecting parameter values.

• **Using potential fields to aid automatic selection of planner and/or parameters:**

  Through describing visibility sets planner can be selected to show high performance for grasps. For example starting in motion planning in small visibility sets towards large visibility sets yields best performance with mono-directional tree-based planners with goal bias. However, visibility sets are not known prior to the motion planning. A way to retrieve such information about visibility potential fields can be used. These fields specify a field around obstacles so they can be avoided. Configurations in tight spaces would have a higher potential field value, this configuration will likely be in a small visibility set. Thus for picking a initial configuration in higher potential field compared to the goal configuration in low value, a mono-directional tree-based planner with goal bias can be picked. Other way around, would yield best performance with a bi-directional tree-based planner with lazy collision-checking.

  The BiTRRT method uses an stochastic method for global minimum, which has similarities with potential fields. However, this planner uses transition test throughout the motion planning to present a more optimal motion plan, which makes it slower. The proposed method only checks the potential field when initializing the planner. This way also specific parameter values can be determined by looking at a potential field.

• **Grasping pipeline framework for ROS:**

  Numerous grasping approaches exist, however, they lack the needed information to execute a grasp in a real-world setup. Moreover, the code of these approaches are mostly not open-source, which means users have to keep designing approaches instead of improving. There is a need for collaborative grasping pipeline development. This could be possibly be achieved with a framework for ROS, like to MoveIt! but then for grasping elements. Researchers could present their grasping approaches or extrinsic calibration processes as plugins for this new framework, similar to OMPL for MoveIt!. This can speed up development, since researchers can focus on one element instead of trying to optimize all the elements.


An Empirical Study on Single-Query Motion Planning for Grasp Execution

In this chapter, the performance of the OMPL planners available in MoveIt! are analyzed when carrying out grasp executions using a UR5 manipulator. Parameter selection was performed before executing the motion planning problems, in appendix B the process of finding these parameter is showcased.

This work is accepted to the IEEE International Conference on Advanced Intelligent Mechatronics (AIM) which will be held in July, 2017.
An Empirical Study of Single-Query Motion Planning for Grasp Execution

Jonathan Meijer, Qujiang Lei, Martijn Wisse

Abstract—This paper selects high-performing Open Motion Planning Library (OMPL) planners for grasp execution and simultaneously presents useful benchmark data. Four grasp executions were defined using a UR5 manipulator. The performance was measured by means of solved runs, computing time and path length. Based on the results, planners are recommended and the reasons are discussed.

I. INTRODUCTION

MoveIt! [1] is widely used for robot manipulation within ROS (Robot Operating System). MoveIt! is by default configured with OMPL [2]. This lets the user easily choose state-of-the-art sampling-based motion planners from the OMPL library. Currently, 23 motion planning algorithms of OMPL are configured to use in MoveIt!. Recommendations for picking a motion planning algorithm is not given. The Planner Arena [3] is created to help users determine which planner suits a given motion planning problem. However, none of the problems resemble the use of a manipulator performing a grasp execution.

This paper aims to provide insight in choosing the right planner(s) for performing grasp executions by conducting an empirical study on the available motion planners of OMPL in MoveIt!. We choose to only investigate the single-query performance of the available sampling-based motion planners, included multi-query planners are being used as single-query planners. The performance of such planners depends on the configuration space of the robot, in particular, motion planning through narrow passages can cause issues [4]. For this paper, four grasp execution motions are defined in the MoveIt! Benchmark environment to discover the behavior of the planners. However, none of the problems resemble the use of a manipulator performing a grasp execution.

The performance of the planners is measured in terms of solved runs, computing time and path length. Solved runs can be noted as a percentage of the total motion planning runs that finish correctly, barplots are used to visualize the difference. For every run, the total computing time and path length can change due to the randomization in sampling-based motion planners. Boxplots and tables are used to analyze the planners with respect to computing time and path length. Performance depends on the users need, right planners for one performance measure can be wrong planners for a different performance measure. By looking at each measure, we can discuss on the preferred planner choice.

Through a quick survey of the available planners of OMPL in MoveIt!, several observations can be made on how they handle motion problems. The comparison (Tab. I) shows that the promise of using FMT, BFMT, LBTRRT, TRRT and BiTRRT. These planners have an optimizing step and stop once an optimized path is found.

II. BACKGROUND

A. Software

The open-source Robot Operation System is a suite of software libraries that help create robot applications. ROS was created to encourage collaborative robotics software development. MoveIt! serves as a framework in ROS to help with the manipulation of robotic hardware. Within MoveIt! several motion planner libraries can be added to perform motion planning for the specified robot. OMPL is a well-integrated motion planner library and is the default library for MoveIt!. It houses state-of-the-art sampling-based motion planners.

*The work leading to these results has received funding from the European Communitys Seventh Framework Programme (FP7/2007-2013) under grant agreement n 609206.

All authors are with the TU Delft Robotics Institute, Delft University of Technology, 2628 CD, Delft, The Netherlands. Email address of Jonathan Meijer is J.G.J.Meijer@student.tudelft.nl. Email addresses of Qujiang Lei and Martijn Wisse are q.lei, m.wisse@tudelft.nl.
planners. OMPL itself only implements the basic primitives of sampling-based motion planning. MoveIt! configures OMPL and provides the back-end for OMPL to work with motion planning problems.

When executed through MoveIt!, OMPL creates a path to solve the motion planning problem. By default, OMPL tries to perform path simplification. These are routines that shorten the path. The smoothness of the path may not be affected by this simplification.

B. Overview planning algorithms

Sampling-based motion planners are proven to be probabilistically complete [13], which implicates that the probability of not finding a feasible path in an unbounded setting approaches zero. For this reason, sampling-based motion planners are widely used to find feasible paths in high-dimensional and geometrically constraint environments. Optimizing planners can refrain from potential high-cost paths and rough motions [14]. However, computational effort for finding an optimized path is increased.

Among the 23 motion planners in Tab. I, six can be considered as multi-query planning methods. A common multi-query planning method is the Probabilistic RoadMap (PRM) [13]. The planner attempts to find a path in a constructed roadmap. The construction of the roadmap is executed by sampling valid nodes (configuration states) in the configuration space. These nodes are connected to other nearby nodes by edges (path segments). In OMPL, the roadmap construction is finished when a certain time limit is reached. Afterward, a simple graph search (query) can be performed on the roadmap to find a path between the start and goal node. Because the algorithm covers the total configuration space with a roadmap, multiple queries can be started to find a path with different start and goal node.

In MoveIt!, three variants of the PRM planner are available for use. The LazyPRM [8] planner initially does not check for valid states when sampling nodes for roadmap construction. Once a path has been found from between start and goal node, collision checking is performed along the nodes and edges of the roadmap. Invalid nodes and edges are removed and a new graph search is attempted. This process is repeated until a feasible path is found. PRMstar [14] is the asymptotically optimal variant of the PRM planner. It rewire nodes to other near nodes if this is beneficial to the cost towards the node. An asymptotically optimal path is found if there is a great number of nodes. LazyPRMstar [14] combines the LazyPRM and PRMstar.

In addition to the PRM planner and its variants, OMPL has two more multi-query planners, SPARS and SPARStwo. They are similar to PRMstar but adds another sparse subgraph. This subgraph is an asymptotically optimal roadmap that houses nodes which resemble multiple nodes in a dense graph. Therefore less computing memory is needed to store the asymptotically optimal roadmap. SPARStwo is different since it has an infinite iteration loop.

The remaining 17 planners in Tab. I are considered as single-query planning methods. These create a roadmap every time a new planning query has to be determined. A common single-query planner is the Rapidly Exploring Random Tree (RRT) method [9]. It grows one tree (monodirectional) from the initial configuration state in the direction of the unexplored areas of the bounded free space. This is realized by randomly sampling nodes in the free space, sampled nodes that can be are within a certain distance of tree nodes are added to the tree by edges. The process of adding nodes and edges is repeated until the tree reaches the goal node. The goal bias parameter in this planner specifies the probability of choosing the goal configuration as a sample rather than a random sample.

The RRTConnect method [10] is a bi-directional version of the RRT method, meaning that two trees are grown. Two processes of RRT are started, one in the start node and one in the goal node. At every iteration or edge addition, it is checked whether the trees can be connected to each other. A path that solves the motion planning problem, is found if these trees can be connected. The near-optimal variant of RRT, RRTstar [14], checks whether the new sampled node can be connected to other near nodes so that the state space is more locally refined. The RRTstar removes the connections of the new sample that are not beneficial towards the cost of the path, like PRMstar. When the number of nodes is big enough, it can result in an asymptotically optimal path from the start-to-goal node. As shown in Tab. I, the RRTstar goal is time-invariant. It keeps trying to optimize the trees by adding new nodes until specified time limit is met.

Lower Bound Tree-RRT (LBTRT) [17] is an asymptotically optimal planner and uses a so-called lower bound graph which is an auxiliary graph. To maintain the tree, a similar method as RRTstar is used. Transition-based RRT or TRRT [18] is a combination of the RRT method and a stochastic optimization method for global minima. It performs transition tests to accept new states to the tree. The algorithm computes an optimized path that is not tied to a time limit, unlike RRTstar. The Bi-TRRT [19] is a bi-directional version of this planner.

The EST method [6] stands for Expansive Space Trees. Other than RRT, EST tries to determine the direction of the tree by looking at neighboring nodes. The tree will grow in the direction of the less explored space. Bi-directional EST (BiEST), based on [6], grows two trees like RRTConnect. Projection EST (ProjEST), also based on [6], detects the less explored area of the configuration space by using a grid. This grid serves as a projection of the state space. Single-query Bi-directional probabilistic roadmap planner with Lazy collision checking, also called SBL, grows two trees. The trees expand in the same manner as EST. Due to its lazy collision checking it will determine if a path is valid after the two trees are connected. It deletes nodes and edges of the path that are not valid, similar to LazyPRM.

KPIECE (Kinodynamic motion Planning by Interior-Exterior Cell Exploration) [7] is a tree-based planner that uses layers of discretization to help estimate the coverage of the state space. The OMPL implementation only uses
one layer. OMPL incorporates a bi-directional variant called BKPIECE and a variant which incorporates lazy collision checking, this is the LBKPIECE.

Fast Marching Tree (FMT) [15] is an asymptotically optimal planner which marches a tree forward in the cost-to-come space on a specified amount of samples. The BFMT [16] planner is a bi-directional variant of this planner.

PDST (Path-Directed Subdivision Tree) [11] represents samples as path segments instead of configuration states. It uses non-uniform subdivisions to explore the state space.

STRIDE (Search Tree with Resolution Independent Density Estimation) [12] uses a Geometric Nearneighbor Access Tree (GNAT) to sample the density of the configuration space. This information helps to guide the planner into the less explored area.

III. PROBLEM FORMULATION

The available planners consist of non-optimizing and optimizing planners. The problem formulation follows the work of Karaman and Frazzoli [14]. Non-optimizing planners attempt to find a feasible path in the bounded \( d \)-dimensional configuration space \( C = [0, 1]^d \). The free configuration space is defined by \( C_{\text{free}} = \text{cl}(C \setminus C_{\text{obs}}) \), in which \( \text{cl}(\cdot) \) denotes the closure of a set and in which \( C_{\text{obs}} \) denotes the obstacle space. A path \( p \) is called feasible when:

\[
\begin{align*}
p(0) &= x_{\text{init}}, \quad p(1) = x_{\text{goal}} \\
p(x) &\in C_{\text{free}} \text{ for all } x \in [0, 1]
\end{align*}
\]  

Optimizing planners that are given a motion planning problem \((C_{\text{free}}, x_{\text{init}}, x_{\text{goal}})\) and a cost function \( c \), find a optimized path \( p^* \) such that:

\[
c(p^*) = \min\{c(p) : p \text{ is feasible } \}
\]  

Problem implementation. Non-optimizing planners are asked to produce feasible paths with a maximum computing time of 3s and 10s. Optimizing planners are asked to produce an optimized path within a maximum computing time of 3s and 10s. Path simplification by OMPL is turned on.

Performance metric. Solved runs, computing time and path length are used as metric in our experiments. We analyze the measures individually to provide the best performing planners in each one of the measures. Solved runs is analyzed in terms of percentage of total runs of the planner resulting in feasible paths, higher performance is considered for higher solved runs. Total computing time is measured for the time it takes for planners to produce feasible or optimized paths with path simplification, a shorter time is considered as higher performance. Moreover, planners with a small standard deviation from the mean computing time and small interquartile range are considered as better performance.

Parameters. In MoveIt! and OMPL parameters can be set to increase the performance of the planners. To choose them, we conducted an iterative process in which we investigated the different parameter settings for each planner by increasing and decreasing the default settings by a factor of two. Parameter tuning was conducted again in the direction performance increase was noticed until no better performance was achieved. These new parameter values were then set before conducting the benchmarks.

IV. DEFINED MOTION PLANNING PROBLEMS

Four grasp execution motions have been defined to measure the performance of the planners.

A. Grasp between obstacles

Benchmark 1 has its end-effector goal placed in such a way that the manipulator has to move through a narrow passage, shown in Fig. 1. This benchmark uses an environment that resembles a table with obstacles. To be able to move through the narrow passage the UR5 robot needs to be in a specific configuration due to its geometry. This decreases the free configuration space near the goal configuration, which makes it harder to find collision-free connections between nodes.

B. Simple motion

Benchmark 2 has its end-effector goal placed at the other side of the scene, shown in Fig. 2. It operates in the same environment as benchmark 1. The end-effector has to be displaced 1.5m in order to reach the goal. This motion planning problem can be solved by mainly actuating the shoulder lift joint. To actuate less joints, using optimizing planners could be beneficial.

C. Place motion

In benchmark 3, initial end-effector position is located at the end of a shelf box, shown in Fig. 3a. This benchmark
uses an environment that houses a simplified shelf box and obstacles placed on a flat surface. The goal position is the initial configuration used in benchmark 1 and 2 (orange state in figure). The motion problem starts in a narrow passage (transparent state in figure), the goal is situated in a less constrained space.

D. Pick motion

Benchmark 4 is attempting to resemble a picking motion from a narrow shelf box, shown in Fig. 3b. The environment is identical to benchmark 3. The goal of the problem is to be in a specific gripper orientation at the end of this shelf. The planner will have to produce a motion plan with high accuracy to reach the end of the shelf. The motion problem starts in a less constrained space (transparent state in figure), the goal is situated in a narrow passage (orange state in figure).

V. PARAMETER SELECTION

Parameters can be set to improve the performance of the planners. In this section, the parameter selection is presented.

While conducting parameter selections for LBTRRT it was found that this planner is behaving unreliable in our setup. We tested all parameter combinations for this planner when conducting various motion planning which resulted in crashes. So we are unable to provide benchmark data for this particular planner.

A. Global planner parameter

There is one parameter that affects all planners. This is the distance parameter \texttt{longest_valid_segment_fraction}. The parameter is called when the planner checks for collisions between two nodes. Collision detection is not checked for the motion if the distance between the nodes is within the parameter value. In narrow passages and corners, this parameter can be critical. The parameter is set in meters and by default has a value of 0.005m. After conducting experiments with lower values, it was found that reducing this parameter did not have an immediate effect on the solved runs for the various benchmark problems.

B. Planner specific parameters

The majority of planners (20 of 23) have their own parameters. For the benchmarks, each parameter was set to values that benefit one or more performance measures, these values are noted in Tab. II.

C. Robot

The UR5 robot that will be used has two joint limit settings for each joint, \( \pi \) and \( 2\pi \). Validating by means of simple motion planning experiments it was found that setting the joint limits to \( \pi \) resulted in favorable performance for all the performance measures.

VI. RESULTS

A. Methodology

The benchmarking experiments are performed using one thread on a system with an Intel i5 2.70GHz processor and 8Gb of memory. To give reliable data on the solved runs, computing time and path length, each algorithm was executed 30 times for the given motion planning problem. The algorithms were given a maximum computing time of 3s and 10s to show the effect of time on the algorithms for which the goal is not time-invariant (shown in Tab. I). The times are kept low since for most robotics applications results are required quickly. Specifically for grasping approaches that try to find grasps in a short amount of time. Planners, where path simplification was not performed by OMPL, have been marked with a * behind the planner name.

B. Plots and Tables

Results of benchmark 1 are shown in Fig. 4 and Tab. III. The motion planning problem affects planners EST, RRT, RRTstar, TRRT and SPARStwo since they were not able to solve all the runs with a percentage higher than 80% with a maximum computing time of 3s and 10s. With exception of SPARStwo, these are mono-directional planners. SBL, BiEST, KPIECE, BKPIECE and LBKPIECE compute valid paths in a computing time shorter than 0.5s. RRTConnect is the fastest planner and BiTRRT is the fastest optimizing planner. SBL has the lowest mean path length with a small standard deviation. For solved runs higher than 80%, planners SBL, KPIECE, and LBKPIECE are able to plan paths of similar lengths. For optimizing planners, BiTRRT has the lowest median path length. TRRT has the lowest path length and standard deviation. Selecting a higher limit for computing time showed shorter paths for planners RRTstar, PRMstar and LazyPRMstar, due to the optimization step. Path simplification contributes to shorter paths.

Results of benchmark 2 are shown in Fig. 5 and Tab. IV. RRTstar, TRRT and SPARStwo have lower solved runs compared to the other planner algorithms. SBL, BiEST, BKPIECE, LBKPIECE, RRTConnect and BiTRRT compute paths in under 0.1s, all being bi-directional planners. BiTRRT is the fastest optimizing planner. SBL and BiTRRT have the shortest paths. The planners that keep sampling the configuration space or optimizing the path until the maximum computing time is reached see improved performance with respect to path length. However, compared to non-optimizing planners

Results of benchmark 3 are shown in Fig. 6 and Tab. V. None of the multi-query planners are able to find feasible paths. Increased computing effort is needed to cover the
TABLE II: Specified planner parameters

<table>
<thead>
<tr>
<th>Planner</th>
<th>Range</th>
<th>Goal Bias</th>
<th>Delay C.C.</th>
<th>KPIECE Range</th>
<th>BKPIECE Range</th>
<th>LBKPIECE Range</th>
<th>STRIDE</th>
<th>FMT</th>
<th>BFMT</th>
<th>TRRT Range</th>
<th>BiTRRT Range</th>
<th>SPARS Range</th>
<th>SPARTwo Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBL</td>
<td>625</td>
<td>0.05</td>
<td>0</td>
<td>625</td>
<td>3125</td>
<td>625</td>
<td>0.3125</td>
<td>0.05</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>EST</td>
<td>625</td>
<td>0.05</td>
<td>0</td>
<td>625</td>
<td>3125</td>
<td>625</td>
<td>0.3125</td>
<td>0.05</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BiEST</td>
<td>625</td>
<td>0.05</td>
<td>0</td>
<td>625</td>
<td>3125</td>
<td>625</td>
<td>0.3125</td>
<td>0.05</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ProjEST</td>
<td>625</td>
<td>0.05</td>
<td>0</td>
<td>625</td>
<td>3125</td>
<td>625</td>
<td>0.3125</td>
<td>0.05</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RRTConnect</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRM</td>
<td>3125</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LazyPRM</td>
<td>3125</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RRTstar</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4: Results for benchmark 1 for 3s and 10s maximum computing time. (a) Solved runs; higher is better. (b) Computing time; lower is better, small interquartile range is better. (c) Path length; lower is better, small interquartile range is better.

total free configuration space with these planners. Of the single-query planners BiEST, RRT, RRTstar and FMT are not able to reach a high level of solved runs. Indicating that these planners are not fast enough to have a proper coverage of the free configuration space. BKPIECE and RRTConnect reach 100% solved runs for 3s maximum computing time. SBL, EST, ProjEST, KPIECE, LBKPIECE, PDST, STRIDE, TRRT and BiTRRT perform better with respect to solved runs when the maximum computing time is set to 10s. RRTConnect is the fastest planner, TRRT is the fastest optimizing planner with solved runs higher than 50%. With exception of RRTConnect, the single-directional planner variants are able to plan a shorter path length compared to the bi-directional planner variant. These planners propagate a path out of a narrow passage and with the use of goal bias shorter paths can be obtained.
Results of benchmark 4 are shown in Fig. 7 and Tab. VI. 5 of the 22 planners were able to compute paths with solved runs higher than 50%. These are all bi-directional planners, motion planning with these planners are also started in the goal configuration. These planners provide high solved runs for max 10s computing time. KPIECE and RRTConnect are the fastest performing planners. BiTRRT has the shortest path length. RRTConnect shows significant performance increase for path length with a higher maximum computing time.

C. Discussion

From the results, observations are made and discussed.

Solved runs. The planners RRTstar, TRRT and SPARStwo show consistent lower solved runs for all the benchmarks, making them less desirable to use for the grasp executions we presented. SBL, LBKPIECE and BiTRRT have high solved runs for a maximum computing time of 10s in all benchmarks. When high solved runs has to be achieved in

<table>
<thead>
<tr>
<th>Planner name</th>
<th>Max. 3s computing time</th>
<th>Path length</th>
<th>Max. 10s computing time</th>
<th>Path length</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBL</td>
<td>0.29 (0.11)</td>
<td>10.07 (0.74)</td>
<td>0.37 (0.18)</td>
<td>9.90 (0.63)</td>
</tr>
<tr>
<td>EST</td>
<td>2.18 (0.31)</td>
<td>10.58 (0.77)</td>
<td>4.65 (2.55)</td>
<td>11.03 (1.01)</td>
</tr>
<tr>
<td>BiEST</td>
<td>0.21 (0.10)</td>
<td>14.81 (5.19)</td>
<td>0.18 (0.07)</td>
<td>13.32 (3.14)</td>
</tr>
<tr>
<td>ProjEST</td>
<td>1.83 (0.86)</td>
<td>11.82 (1.81)</td>
<td>2.37 (1.57)</td>
<td>12.13 (2.19)</td>
</tr>
<tr>
<td>KPIECE</td>
<td>0.20 (0.09)</td>
<td>10.89 (1.80)</td>
<td>0.22 (0.10)</td>
<td>10.55 (1.28)</td>
</tr>
<tr>
<td>BKPIECE</td>
<td>0.42 (0.21)</td>
<td>10.94 (1.86)</td>
<td>0.42 (0.21)</td>
<td>10.56 (1.82)</td>
</tr>
<tr>
<td>LBKPIECE</td>
<td>0.30 (0.08)</td>
<td>10.41 (1.53)</td>
<td>0.26 (0.11)</td>
<td>12.30 (7.16)</td>
</tr>
<tr>
<td>RRT</td>
<td>0.54 (0.54)</td>
<td>11.93 (1.41)</td>
<td>1.48 (2.71)</td>
<td>11.62 (1.14)</td>
</tr>
<tr>
<td>RRTConnect</td>
<td>0.11 (0.08)</td>
<td>12.52 (15.68)</td>
<td>0.09 (0.03)</td>
<td>11.89 (9.10)</td>
</tr>
<tr>
<td>PDST</td>
<td>1.37 (0.87)</td>
<td>11.96 (2.35)</td>
<td>1.68 (1.61)</td>
<td>12.37 (2.15)</td>
</tr>
<tr>
<td>STRIDE</td>
<td>0.59 (0.57)</td>
<td>11.97 (5.35)</td>
<td>1.12 (1.58)</td>
<td>11.20 (2.24)</td>
</tr>
<tr>
<td>PRM*</td>
<td>3.01 (0.01)</td>
<td>15.00 (2.26)</td>
<td>10.01 (0.01)</td>
<td>14.49 (1.65)</td>
</tr>
<tr>
<td>LazyPRM</td>
<td>3.02 (0.00)</td>
<td>12.13 (1.17)</td>
<td>10.02 (0.01)</td>
<td>12.48 (1.96)</td>
</tr>
<tr>
<td>RRTstar*</td>
<td>3.01 (0.01)</td>
<td>12.76 (0.93)</td>
<td>10.02 (0.02)</td>
<td>11.47 (1.03)</td>
</tr>
<tr>
<td>PRMstar*</td>
<td>3.02 (0.01)</td>
<td>14.43 (1.90)</td>
<td>10.02 (0.01)</td>
<td>12.99 (1.67)</td>
</tr>
<tr>
<td>LazyPRMstar</td>
<td>3.02 (0.00)</td>
<td>11.53 (1.23)</td>
<td>10.03 (0.01)</td>
<td>10.95 (1.59)</td>
</tr>
<tr>
<td>FMT</td>
<td>2.07 (0.45)</td>
<td>10.49 (0.99)</td>
<td>1.78 (0.23)</td>
<td>10.33 (0.64)</td>
</tr>
<tr>
<td>BFMT</td>
<td>1.17 (0.36)</td>
<td>11.74 (2.65)</td>
<td>0.89 (0.09)</td>
<td>10.88 (1.06)</td>
</tr>
<tr>
<td>TRRT</td>
<td>0.57 (0.58)</td>
<td>10.21 (1.52)</td>
<td>2.41 (2.82)</td>
<td>10.12 (0.45)</td>
</tr>
<tr>
<td>BiTRRT</td>
<td>0.13 (0.08)</td>
<td>15.56 (16.75)</td>
<td>0.13 (0.10)</td>
<td>11.05 (5.22)</td>
</tr>
<tr>
<td>SPARS*</td>
<td>3.04 (0.04)</td>
<td>23.63 (4.74)</td>
<td>10.07 (0.07)</td>
<td>23.87 (5.57)</td>
</tr>
<tr>
<td>SPARStwo*</td>
<td>3.00 (0.00)</td>
<td>22.98 (3.97)</td>
<td>10.00 (0.00)</td>
<td>26.34 (10.01)</td>
</tr>
</tbody>
</table>

Standard deviation in parentheses
TABLE IV: Mean values for benchmark 2

<table>
<thead>
<tr>
<th>Planner name</th>
<th>Max. 3s computing time (s)</th>
<th>Max. 10s computing time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBL</td>
<td>0.05 (0.01)</td>
<td>0.04 (0.01)</td>
</tr>
<tr>
<td>EST</td>
<td>0.20 (0.13)</td>
<td>0.16 (0.11)</td>
</tr>
<tr>
<td>BIEST</td>
<td>0.09 (0.04)</td>
<td>0.08 (0.03)</td>
</tr>
<tr>
<td>PRM</td>
<td>1.17 (0.11)</td>
<td>1.03 (0.79)</td>
</tr>
<tr>
<td>KPIECE</td>
<td>0.18 (0.09)</td>
<td>0.15 (0.09)</td>
</tr>
<tr>
<td>BKPPIECE</td>
<td>0.11 (0.11)</td>
<td>0.13 (0.16)</td>
</tr>
<tr>
<td>LBKPIECE</td>
<td>0.09 (0.06)</td>
<td>0.08 (0.03)</td>
</tr>
<tr>
<td>RRT</td>
<td>0.52 (0.09)</td>
<td>0.44 (1.10)</td>
</tr>
<tr>
<td>RRTConnect</td>
<td>0.09 (0.04)</td>
<td>0.06 (0.02)</td>
</tr>
<tr>
<td>PDST</td>
<td>0.24 (0.16)</td>
<td>0.24 (0.16)</td>
</tr>
<tr>
<td>STRIDE</td>
<td>0.19 (0.21)</td>
<td>0.14 (0.09)</td>
</tr>
<tr>
<td>PRM*</td>
<td>3.02 (0.01)</td>
<td>1.01 (0.00)</td>
</tr>
<tr>
<td>LazyPRM</td>
<td>3.02 (0.00)</td>
<td>0.88 (1.88)</td>
</tr>
<tr>
<td>RRT*</td>
<td>3.01 (0.02)</td>
<td>0.80 (0.00)</td>
</tr>
<tr>
<td>PRMStar*</td>
<td>3.03 (0.01)</td>
<td>1.20 (1.81)</td>
</tr>
<tr>
<td>LazyPRMStar*</td>
<td>3.02 (0.01)</td>
<td>0.82 (0.00)</td>
</tr>
<tr>
<td>FMT</td>
<td>1.23 (0.16)</td>
<td>1.10 (0.15)</td>
</tr>
<tr>
<td>BFMT</td>
<td>0.79 (0.06)</td>
<td>0.73 (0.06)</td>
</tr>
<tr>
<td>TRRT</td>
<td>0.78 (0.09)</td>
<td>0.72 (0.84)</td>
</tr>
<tr>
<td>BITRT</td>
<td>0.08 (0.02)</td>
<td>0.07 (0.02)</td>
</tr>
<tr>
<td>SPARS</td>
<td>3.05 (0.04)</td>
<td>19.38 (8.05)</td>
</tr>
<tr>
<td>SPARSwo*</td>
<td>3.00 (0.01)</td>
<td>14.98 (5.37)</td>
</tr>
</tbody>
</table>

Standard deviation in parentheses

TABLE V: Mean values for benchmark 3

<table>
<thead>
<tr>
<th>Planner name</th>
<th>Max. 3s computing time (s)</th>
<th>Max. 10s computing time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBL</td>
<td>1.54 (0.69)</td>
<td>2.06 (1.56)</td>
</tr>
<tr>
<td>EST</td>
<td>1.06 (0.64)</td>
<td>1.03 (0.79)</td>
</tr>
<tr>
<td>BIEST</td>
<td>0.11 (0.00)</td>
<td>1.03 (0.79)</td>
</tr>
<tr>
<td>ProEJST</td>
<td>0.98 (0.74)</td>
<td>1.21 (1.03)</td>
</tr>
<tr>
<td>KPIECE</td>
<td>1.15 (0.86)</td>
<td>1.00 (0.90)</td>
</tr>
<tr>
<td>BKPPIECE</td>
<td>1.32 (0.78)</td>
<td>1.19 (0.87)</td>
</tr>
<tr>
<td>LBKPIECE</td>
<td>1.50 (0.88)</td>
<td>1.34 (1.28)</td>
</tr>
<tr>
<td>RRT*</td>
<td>- (-)</td>
<td>3.82 (2.87)</td>
</tr>
<tr>
<td>RRTConnect</td>
<td>0.62 (0.23)</td>
<td>16.13 (13.07)</td>
</tr>
<tr>
<td>PDST</td>
<td>1.27 (0.71)</td>
<td>2.07 (1.77)</td>
</tr>
<tr>
<td>STRIDE</td>
<td>0.89 (0.59)</td>
<td>1.08 (0.82)</td>
</tr>
<tr>
<td>RRT<em>Star</em></td>
<td>3.00 (0.00)</td>
<td>10.01 (0.00)</td>
</tr>
<tr>
<td>FMT</td>
<td>2.91 (0.15)</td>
<td>6.57 (1.23)</td>
</tr>
<tr>
<td>TART</td>
<td>0.90 (0.60)</td>
<td>1.74 (1.92)</td>
</tr>
<tr>
<td>BITRT</td>
<td>1.74 (0.62)</td>
<td>2.65 (1.63)</td>
</tr>
</tbody>
</table>

Standard deviation in parentheses

TABLE VI: Mean values for benchmark 4

<table>
<thead>
<tr>
<th>Planner name</th>
<th>Max. 3s computing time (s)</th>
<th>Max. 10s computing time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBL</td>
<td>1.71 (0.73)</td>
<td>2.40 (1.67)</td>
</tr>
<tr>
<td>BKPPIECE</td>
<td>1.00 (0.63)</td>
<td>1.23 (0.80)</td>
</tr>
<tr>
<td>LBKPIECE</td>
<td>1.21 (0.60)</td>
<td>1.49 (1.10)</td>
</tr>
<tr>
<td>RRTConnect</td>
<td>0.75 (0.32)</td>
<td>0.91 (0.54)</td>
</tr>
<tr>
<td>PRM*</td>
<td>- (-)</td>
<td>10.01 (0.00)</td>
</tr>
<tr>
<td>BiTRRT</td>
<td>1.61 (0.65)</td>
<td>3.05 (2.02)</td>
</tr>
</tbody>
</table>

Standard deviation in parentheses

A shorter time, BKPPIECE and RRTConnect are the best choices when performing varied grasp executions.

Computing time. When feasible paths need to be found in a timely manner, a bi-directional planner is recommended for similar motion planning problems to benchmark 1, 2 and 4. When a path creation starts in a narrow passage as in benchmark 3, a single-directional planner can give improved computing times, with exception of RRTConnect.

Path length. Picking an optimizing planner to find shorter path lengths can not be justified from the benchmark data. Only in benchmark 2, the TRRT and BiTRRT planners for 3s and 10s maximum computing time respectively compute shorter paths compared to non-optimizing planners. Having more computing time for time-variant goals decreases path lengths, however, not drastically. The RRTConnect shows low path length means in all benchmark problems.

Multi-query. For this paper, we only looked at single-query motion planning problems, multi-query planners can also be used as single-query planners. Though this paper fails to show the potential benefit of using the same roadmap multiple times and how this can affect computing time. Though we do notice that RRTConnect and BiTRRT are able to give valid paths in very short amounts of time that we argue the need for multi-query planners for online grasp executions.

Parameter selection. Since parameter values have to be set for a planner to operate, the aim was to achieve maximum performance of the planners. By manually conducting the iterative process explained before, a guarantee of maximum performance can not be given. We executed the iterative process to the best of our ability to achieve maximum performance.

Combined metrics. When looking at all the benchmarks, SBL, BKPPIECE, LBKPIECE, RRTConnect and BiTRRT show high performance in the metrics solved runs, computing time and path length.

Optimization with a time-invariant goal. Planners FMT, BFMT, LBTRRT, TRRT and BiTRRT stop once an optimized path is found. Of these, BiTRRT is the fastest performing planner. However, compared to not non-optimizing planners the path length is not consistently shorter. More research has to be done to see whether other metrics will make the use of this planner more desirable.

Robot. The UR5 is used to compute the motion plans though we have not investigated the change in the performance measures for different types of manipulators. It needs to be determined if the results hold for other types of manipulators. For future work, multiple manipulators can be used to perform the same benchmark to find a better answer on the consistency of the planner’s performance.

Path constraints. The results presented do not give any estimation on how the planners perform when implementing hard path constraints.

VII. CONCLUSION

This paper presented benchmark data of available OMPL planners in MoveIt! for geometrically constrained grasp executions using a 6-DOF manipulator. Planner performance was studied by means of solved runs, computing time and path length for two maximum computing time settings. From the performance analysis remarks and recommendations were made depending on the performance measure. For the defined grasp executions, the bi-directional planners SBL, BKPPIECE, RRTConnect and BiTRRT are high performing planners in terms of all the studied metrics. For future work, we would like to investigate the use of different manipulators to find consistency in planner performance when carrying out grasp executions.

REFERENCES


Fig. 6: Results for benchmark 3 for 3s and 10s maximum computing time. (a) Solved runs; higher is better. (b) Computing time; lower is better, small interquartile range is better. (c) Path length; lower is better, small interquartile range is better.

Fig. 7: Results for benchmark 4 for 3s and 10s maximum computing time. (a) Solved runs; higher is better. (b) Computing time; lower is better, small interquartile range is better. (c) Path length; lower is better, small interquartile range is better.


Parameter selection

In this part the process of selecting parameter values for the UR5 manipulator is presented by providing several examples. The examples are part of the parameter selection conducted for chapter 3.

B.1. Problem

To gain better performance of the sampling-based motion planners in OMPL, exposed planner parameters have to be set. By default the planners have been given a value. There is great confusion in the ROS community on how to set these parameters properly\(^1\)\(^2\), since no clear tutorial exist on how to change them in ROS. For the selection of high-performing planners, these parameters have to be set to achieve highest performance. This is done by presenting a process to do this effectively.

B.2. Process

For this thesis the performance of planners is measured by means of solved runs, computing time and path length. Solved runs can be considered as highest priority, since when this value is higher the probability of the planner finding a solution to the motion planning problem in future runs will be higher. Sometimes a trade-off has to be made between computing time and path length, parameter values that result in the highest performance increase will be chosen. The place motion planning problem in chapter 3 was picked in order to conduct the parameter selection. Parameters are set accordingly in the `ompl_planning.yaml` file inside the `config` of the MoveIt! configured robot or manipulator.

The used process is an iterative process and stopped when no increase in performance is found. For each iteration, three versions of a planner, each having different parameter values for one specific parameter, are executed 50 times in order to give a reliable estimation of the performance. Maximum computing time is set to 10s. For the first parameter selection iteration the parameter values are `default/2`, `default` and `default ∗ 2`. If the default value is infinite or zero a substitute value is given. The parameter values for the second, third, fourth, ..., nth iteration depend on the outcome of the previous (n − 1) iteration.

The process detailed below has been conducted the same way for all the parameters, for every planner and for all manipulators. This meant that a total 192 parameters had to be evaluated for the best value, usually with 2 to 4 iterations each.

B.3. Example: Process description for the range parameter

The range parameter is present in 14 of the 23 OMPL planners available in MoveIt!. As example we take planners EST and BiTRRT to improve the range value parameter for. Since the range parameter is by default 0.0, which actually means no preference, the default value will be set to 0.3. This means that for the first iteration the range parameter is set to 0.15 for planner, 0.3 for planner2 and 0.6 for planner3. The results can be found in figure B.1.

From the results of the first iteration it can be noticed that best solved (runs, in %) for EST and BiTRRT is without a doubt when having a value of 0.6 (EST2, BiTRRT2). Since it seems that higher values for range result

\(^1\)http://answers.ros.org/question/50161/how-tuning-ompl-parameters/
\(^2\)http://answers.ros.org/question/232673/how-to-set-the-parameters-of-ompl-for-better-planning/
in better performance the new parameter values will be set to EST, BiTRRT=0.6 (the same), EST1, BiTRRT1=0.9 and EST2, BiTRRT2=1.2. The results for these parameter values is shown in figure B.2.

In figure B.2 best performance for solved (%) is found for EST for range values 0.6 and 0.9. However better performance for computing time and path length for EST are retrieved with 0.6 range value. Best computing time is retrieved with a value of 0.6 for BiTRRT. To check whether the best performing value is not within 0.6 and 0.9 the iteration 3 will have values: EST=0.6, EST1=0.9 and EST2=0.75. Since 0.6 was the best value for BiTRRT so far the new iteration values are chosen in close proximity to this value, BiTRRT=0.6, BiTRRT1=0.525 and BiTRRT2=0.75. Results for this iteration is shown in figure B.3.

The results show that 0.6 is the best range value for EST and BiTRRT. Both planners give the best result for the three metrics. Since this value was picked in the last two iterations no further iterations have been made and these parameter values are thus final.

### B.4. Results

The results of the parameter selection is shown in table B.1. In this table the specific parameter is noted with its converged value, it is also noted how many iterations were needed to get a proper convergence. In figure B.4 the difference in solved runs is shown for default and improved parameter values in executing the place motion using the UR5 manipulator. This figure shows that higher solved runs were obtained with the converged parameters, found with the process described in this chapter. Except for PDST, STRIDE and FMT no better performance was obtained. Since PDST has no exposed parameters, STRIDE already showed high performance and FMT's parameters did not have effect on the performance in the case of the grasp execution.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>EST</th>
<th>BiEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>range bias</td>
<td>0.75</td>
<td>0.75</td>
</tr>
<tr>
<td>goal bias</td>
<td>0.75</td>
<td>0.75</td>
</tr>
<tr>
<td>Temp c.f.</td>
<td>0.075</td>
<td>0.075</td>
</tr>
<tr>
<td>Goal bias</td>
<td>0.9</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Table B.1: Converged parameter values for UR5 (U), LWR 4+ (L) and JACO (J), used in chapter 3

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>1000</td>
</tr>
<tr>
<td>Goal bias</td>
<td>0.9</td>
</tr>
<tr>
<td>Temp c.f.</td>
<td>0.075</td>
</tr>
<tr>
<td>Goal bias</td>
<td>0.9</td>
</tr>
<tr>
<td>Temp c.f.</td>
<td>0.075</td>
</tr>
</tbody>
</table>

Number of iterations in parentheses, note that each iteration uses three different values
Figure B.1: Range parameter selection, iteration 1
Figure B.2: Range parameter selection, iteration 2
Figure B.3: Range parameter selection, iteration 3
B. Parameter selection

Figure B.4: Solved runs for default and improved parameters, higher is better

B.5. Discussion

The iteration process to get a good convergence of the parameter values was performed to the best of our abilities. The iteration method is conducted manually and therefore no hard claims can be made on best performance. However, good results were obtained as shown in figure B.4.

The parameter that showed most significant change in planner performance was the range parameter. This parameter specifies the maximum distance that a tree node is extended towards the sample. This parameter needs to be low enough to cover the configuration space, which results in higher solved runs. However, computing effort is increased for lower values, since more connections can be made more collision-checks need to be performed.
Performance saturation of sampling-based motion planners

Sampling-based motion planners try to find a sequence of configurations between the initial configuration and goal configuration in order to execute a collision-free motion. Due to the random sampling the probability of finding identical solutions is low, meaning that performance may vary for the same motion planning problem. However, trends in planner performance can be observed when solving the same motion planning problem multiple times (from here on referred to as: runs). This chapter aims to identify the amount of runs needed to find the best saturated results.

C.1. Execution
To generate results the motion planning problem shown in figure C.1 was executed 50, 100 and 500 times. Planner performance is measured in terms of solved runs, computing time and path length. The results are analyzed by using bar plots for solved runs. Tables and boxplots are used for computing time and path length. The tables hold mean and standard deviation values. It is expected that more runs will decrease the standard deviations in computing time and path length since more results are gathered.

C.2. Results
The bar and boxplots are shown in figure C.2 and the table C.1 notes the mean and standard deviation values. From the barplot, showing the solved runs, no significant difference can be noticed for planners with solved runs higher than 80%. In the boxplot of computing time only significant change is noticed for planners BiEST, FMT and TRRT. This change is also noticeable in the mean and standard deviation values in the table. Path length shows similar performance for the three different amount of runs. Change in path length is more apparent in the table, looking at the standard deviation. In all the runs the fastest planner is KPIECE, indicating that the amount of runs did not have an effect on the highest performing planner. This planner is also among the highest performing planners with respect to path length, identical to what was concluded in chapter 3.

C.3. Discussion
In this chapter the difference in planner performance is checked for different amount of runs. Significant change in planner performance can not be noticed from the boxplots. Difference exist between the mean and standard deviations of computing time and path length. However, no correlation between planner performance and amount of runs can be found. An increase in the amount of runs did not decrease the standard deviations, indicating that randomness still exists in high amount of runs. Therefore it can be argued for the need of running the benchmarks more than 50 times, since this does not change the outcome identifying high performing planners.
Figure C.1: Place motion planning problem using the UR5.

Figure C.2: Results for different amount of runs. (a) Solved runs; higher is better. (b) Computing time; lower is better, small interquartile range is better. (c) Path length; lower is better, small interquartile range is better.
### Table C.1: Mean values and standard deviations for different amount of runs

<table>
<thead>
<tr>
<th>Planner name</th>
<th>50 runs</th>
<th></th>
<th>100 runs</th>
<th></th>
<th>500 runs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (s)</td>
<td>Path length</td>
<td>Time (s)</td>
<td>Path length</td>
<td>Time (s)</td>
<td>Path length</td>
</tr>
<tr>
<td>SBL</td>
<td>2.20 (1.47)</td>
<td>14.00 (4.48)</td>
<td>2.13 (1.32)</td>
<td>14.14 (4.50)</td>
<td>2.17 (1.36)</td>
<td>15.10 (9.46)</td>
</tr>
<tr>
<td>EST</td>
<td>1.23 (1.22)</td>
<td>14.71 (5.33)</td>
<td>1.13 (0.99)</td>
<td>13.98 (5.87)</td>
<td>1.22 (0.98)</td>
<td>14.16 (5.34)</td>
</tr>
<tr>
<td>BiEST</td>
<td>5.23 (2.34)</td>
<td>14.59 (5.01)</td>
<td>4.73 (2.31)</td>
<td>15.50 (11.46)</td>
<td>4.91 (2.25)</td>
<td>14.23 (6.69)</td>
</tr>
<tr>
<td>ProjEST</td>
<td>1.16 (0.79)</td>
<td>15.02 (11.66)</td>
<td>1.10 (0.77)</td>
<td>14.09 (9.66)</td>
<td>1.08 (0.84)</td>
<td>13.69 (5.90)</td>
</tr>
<tr>
<td>KPIECE</td>
<td>1.02 (0.77)</td>
<td>13.35 (3.30)</td>
<td>1.02 (0.75)</td>
<td>14.04 (3.91)</td>
<td>0.98 (0.82)</td>
<td>15.32 (13.74)</td>
</tr>
<tr>
<td>BKPIECE</td>
<td>1.91 (1.39)</td>
<td>14.70 (3.91)</td>
<td>2.43 (1.80)</td>
<td>15.50 (6.39)</td>
<td>2.30 (1.72)</td>
<td>16.17 (9.53)</td>
</tr>
<tr>
<td>LBKPIECE</td>
<td>1.41 (1.28)</td>
<td>14.09 (4.03)</td>
<td>1.24 (0.85)</td>
<td>16.09 (9.30)</td>
<td>1.35 (0.97)</td>
<td>16.00 (8.51)</td>
</tr>
<tr>
<td>RRT</td>
<td>2.92 (2.46)</td>
<td>14.72 (2.15)</td>
<td>2.88 (2.54)</td>
<td>16.29 (7.52)</td>
<td>2.97 (2.53)</td>
<td>16.89 (11.43)</td>
</tr>
<tr>
<td>RRTConnect</td>
<td>1.37 (0.54)</td>
<td>28.15 (58.97)</td>
<td>1.35 (0.58)</td>
<td>28.07 (45.23)</td>
<td>1.34 (0.65)</td>
<td>36.25 (78.20)</td>
</tr>
<tr>
<td>PDST</td>
<td>2.16 (1.36)</td>
<td>19.57 (18.29)</td>
<td>1.86 (1.53)</td>
<td>16.97 (7.21)</td>
<td>1.97 (1.50)</td>
<td>16.55 (8.71)</td>
</tr>
<tr>
<td>STRIDE</td>
<td>1.32 (1.16)</td>
<td>16.53 (6.92)</td>
<td>1.37 (1.28)</td>
<td>14.28 (4.47)</td>
<td>1.15 (0.91)</td>
<td>15.31 (13.21)</td>
</tr>
<tr>
<td>RRStar</td>
<td>10.02 (0.02)</td>
<td>16.33 (3.94)</td>
<td>10.02 (0.02)</td>
<td>16.11 (3.65)</td>
<td>10.02 (0.02)</td>
<td>16.43 (3.16)</td>
</tr>
<tr>
<td>FMT</td>
<td>4.66 (1.43)</td>
<td>14.90 (1.63)</td>
<td>6.04 (1.90)</td>
<td>14.73 (1.41)</td>
<td>5.49 (2.98)</td>
<td>15.55 (2.45)</td>
</tr>
<tr>
<td>TRRT</td>
<td>2.84 (2.43)</td>
<td>26.80 (81.98)</td>
<td>3.23 (2.40)</td>
<td>16.87 (11.21)</td>
<td>2.90 (2.10)</td>
<td>16.41 (8.39)</td>
</tr>
<tr>
<td>BiTRRT</td>
<td>1.57 (0.79)</td>
<td>16.69 (5.94)</td>
<td>2.19 (1.55)</td>
<td>19.39 (22.57)</td>
<td>2.12 (1.39)</td>
<td>19.33 (23.76)</td>
</tr>
</tbody>
</table>

Standard deviation in parentheses
Grasp execution sequence

In this chapter the sequence of a grasp execution is visualized by sequential images (from left to right, top to bottom). Initial state is shown top left, end state bottom right. The sequence shown in figure D.1 is retrieved the motion plan computed by LBKPIECE, which showed highest performance in a similar grasp execution in simulation environment.
Figure D.1: Motion sequence of a grasp execution using LBKPIECE
This chapter presents the code used in chapter 4. The code is made in the C++ language.

E.1. Object segmentation

This code represents the object segmentation process explained in chapter 4. It checks the difference between two scenes to find the unknown object.

```cpp
// Code created by Jonathan Meijer, supplementary code by Benjamin Meijer
// This code segments the a point cloud by giving it a fixed point cloud and real-time point cloud
// The result would be the point cloud of the unknown object
// Additionally this code estimates the pose of the unknown object for grasping

#include "ros/ros.h"
#include <pcl/io/pcd_io.h>
#include <pcl/filters/filter.h>
#include <pcl/filters/radius_outlier_removal.h>
#include <pcl/filters/extract_indices.h>
#include <cmath>
#include <sensor_msgs/PointCloud2.h>
#include <pcl_conversions/pcl_conversions.h>
#include <message_filters/subscriber.h>
#include <message_filters/time_synchronizer.h>
#include <message_filters/synchronizer/approximate_time.h>
#include <geometry_msgs/Point.h>
#include <geometry_msgs/PointStamped.h>
#include <tf/transform_listener.h>
#include <pcl/common/common.h>
#include <tf/transform_datatypes.h>
#include <tf/tf.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <tf2/transform_broadcaster.h>
#include <tf2/geometry_msgs.h>

#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <std_msgs/String.h>
#include <sstream>
#include <pcl_ros/transforms.h>
#include <Eigen/Dense>

using namespace message_filters;
using namespace Eigen;
ros::Publisher pub_unknown_obj;
ros::Publisher pub_realscene;
ros::Publisher pub_pose_cyl;
ros::Publisher pub_p_min;
ros::Publisher pub_p_max;
boost::shared_ptr<tf::TransformListener> tf_ptr;
boost::shared_ptr<tf::TransformListener> tf_ptr_cyl;

// This function transposes the pose of the cylinder or object to the robot frame
void transform_pose(const geometry_msgs::Pose geometry_msg) {  
  ROS_INFO("transfo...";
}
```
pub_pose_cyl.publish(pose_cylinder);

// This function transposes the point of the cylinder or object center to the robot frame
void transform_point(const pcl::PointXYZ p_min, const pcl::PointXYZ p_max);

geometry_msgs::PointStamped min_point;
geometry_msgs::PointStamped min_point_tp;
min_point.point.x=p_min.x;
min_point.point.y=p_min.y;
min_point.point.z=p_min.z;

geometry_msgs::PointStamped max_point;
geometry_msgs::PointStamped max_point_tp;
max_point.point.x=p_max.x;
max_point.point.y=p_max.y;
max_point.point.z=p_max.z;

min_point.header.frame_id = "realkinect2frame";
min_point.header.stamp = ros::Time::now();
max_point.header.frame_id = "realkinect2frame";
max_point.header.stamp = ros::Time::now();

ros::Rate rate(10.0);
try {
  tf_ptr->transformPoint("base_link",min_point,min_point_tp); // pose_in is in camera frame
  tf_ptr->transformPoint("base_link",max_point,max_point_tp);
  rate.sleep();
} catch (tf::TransformException ex) {
  ROS_ERROR("transfrom/uni2423exception/uni2423:/uni2423%s", ex.what());
}

cylinder_bottom.header.frame_id = "base_link";
cylinder_bottom.header.stamp = ros::Time::now();
cylinder_bottom.pose = pose_cylinder.pose;
cylinder_bottom.pose.position = min_point_tp.point;

cylinder_top.header.frame_id = "base_link";
cylinder_top.header.stamp = ros::Time::now();
cylinder_top.pose = pose_cylinder.pose;
cylinder_top.pose.position = max_point_tp.point;

// Cylinder estimation, with the use of SACsegmentation
void get_cylinder(const pcl::PointCloud<pcl::PointXYZRGB >::Ptr point_in);

// CYLINDER SEGMENTATION/POSE EXTRACTION
pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> ne;
pcl::SACSegmentationFromNormals<pcl::PointXYZ, pcl::Normal> seg;
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZ> ());
pcl::PointCloud<pcl::Normal>::Ptr cloud_normals (new pcl::PointCloud<pcl::Normal>);
pcl::PointCloud<pcl::Normal>::Ptr cloud_normals2 (new pcl::PointCloud<pcl::Normal>);
pcl::ModelCoefficients::Ptr coefficients_plane (new pcl::ModelCoefficients), coefficients_cylinder (new pcl::ModelCoefficients);
pcl::PointIndices::Ptr inliers_plane (new pcl::PointIndices), inliers_cylinder (new pcl::PointIndices);
pcl::ExtractIndices<pcl::PointXYZ> extract;
pcl::ExtractIndices<pcl::Normal> extract_normals;

//copyPointcloud
pcl::PointCloud<pcl::PointXYZ>::Ptr cylinder_with_plane (new pcl::PointCloud<pcl::PointXYZ>);
pcl::copyPointCloud(*point_in, *cylinder_with_plane);

// Estimate point normals
ne.setSearchMethod (tree);
ne.setInputCloud (cylinder_with_plane);
ne.setKSearch (10);
ne.compute (*cloud_normals);
seg.setOptimizeCoefficients (true);
seg.setModelType (pcl::SACMODEL_NORMAL_PLANE);
seg.setNormalDistanceWeight (0.1);
seg.setMethodType (pcl::SAC_RANSAC);
seg.setMaxIterations (100);
seg.setDistanceThreshold (0.01);
seg.setInputCloud (cylinder_with_plane);
seg.setInputNormals (cloud_normals);

// Obtain the plane inliers and coefficients
seg.segment (*inliers_plane, *coefficients_plane);

// Extract the planar inliers from the input cloud
extract.setInputCloud (cylinder_with_plane);
extract.setIndices (inliers_plane);
extract.setNegative (false);
extract.filter (*cloud_normal);

// Write the plane inliers to disk
pcl::PointCloud<pcl::PointXYZ> cloud_inliers;
eigen::Matrix4f trans;
eigen::Matrix4f trans2;
eigen::Matrix4f trans3;
eigen::Matrix4f trans4;

// Create the segmentation object for cylinder segmentation and set all the parameters
eigen::Matrix4f trans5;
eigen::Matrix4f trans6;
eigen::Matrix4f trans7;
eigen::Matrix4f trans8;

// Obtain the cylinder inliers and coefficients
seg.segment (*inliers_cylinder, *coefficients_cylinder);
E.1. Object segmentation

geometry_msgs::PoseStamped pose_cylinder_camera;  
coefficients_cylinder->values.resize (7);  // We need 7 values  
pose_cylinder_camera.pose.position.x= coefficients_cylinder->values[0];  
pose_cylinder_camera.pose.position.y= coefficients_cylinder->values[1];  
pose_cylinder_camera.pose.position.z= coefficients_cylinder->values[2];  
double roll = coefficients_cylinder->values[3];  
double pitch = coefficients_cylinder->values[4];  
double yaw= coefficients_cylinder->values[5];  
pose_cylinder_camera.pose.orientation = tf::createQuaternionMsgFromRollPitchYaw(roll, pitch, yaw);  
pose_cylinder_camera.header.frame_id ="/realkinect2frame";  
pose_cylinder_camera.header.stamp = ros::Time::now();  
//pub_center_obj.publish(pose_cylinder_camera);  
transform_point(pose_cylinder_camera);  

evaluate.setInputCloud (cylinder_bare);  
evaluate.setIndices (inliers_cylinder);  
evaluate.setNegative (false);  
std::shared_ptr<pcl::PointCloud<pcl::PointXYZ>> cloud_cylinder (new pcl::PointCloud<pcl::PointXYZ>());  
evaluate.filter (*cloud_cylinder);  

pcl::PointXYZ min_p, max_p;  
pcl::getMinMax3D (*cloud_cylinder, min_p, max_p);  
transform_point(min_p,max_p);  
pcl::PCLPointCloud2 unknown;  
unknown.header.frame_id ="realkinect2frame";  
toPCLPointCloud2(*cloud_cylinder, unknown);  
unknown_object_cloud.header.frame_id ="realkinect2frame";  
// Publish the data to ROS  
pres_unknown_obj.publish(unknown_object_cloud);  

// Checks if two points are out of marge, for segmentation  
bool isOutMarge(float pointOne, float pointTwo){  
    float marge = 0.015;  
    // Check if above/below marge  
    if(pointOne >= pointTwo + (marge * pointTwo)) {  
        //std::cout << pointOne << std::endl;  
        //std::cout << pointTwo << std::endl;  
        return true;  
    } else if(pointOne <= pointTwo - (marge * pointTwo)){  
        //std::cout << pointOne << std::endl;  
        //std::cout << pointTwo << std::endl;  
        return true;  
    } else {  
        return false;  
    }  
}  

// Function for saving points from one PointCloud to another  
void saveToCloud(pcl::PointCloud<pcl::PointXYZRGB >::Ptr destination,  
pcl::PointCloud<pcl::PointXYZRGB >::Ptr source,  
int src_index, float rgbi){  
    // Push points from source into back of destination cloud.  
    destination->push_back(source->points[src_index]);  
}

// Main part of the segmentation  
void compute(const sensor_msgs::PointCloud2ConstPtr& realscene, const sensor_msgs::PointCloud2ConstPtr& fixedscene) {  
    std::cout << "Computing/uni2423pointclouds" << std::endl;  
    ros::WallTime start_time = ros::WallTime::now();  
    // Creating a PCL compatible PointCloud2 and import the ROS PointCloud2  
    pcl::PCLPointCloud2 pcl_pc1;  
    pcl_conversions::toPCL(*realscene, pcl_pc1);  
    // Creating PointXYZRGB struct and import the PCL compatible PointCloud2  
    pcl::PointCloud<pcl::PointXYZRGB >::Ptr cloud1(new pcl::PointCloud<pcl::PointXYZRGB >);  
    pcl::fromPCLPointCloud2(pcl_pc1, *cloud1);  
    // Creating a PCL compatible PointCloud2 and import the ROS PointCloud2  
    pcl::PCLPointCloud2 pcl_pc2;  
    pcl_conversions::toPCL(*fixedscene, pcl_pc2);  
    // Creating PointXYZRGB struct and import the PCL compatible PointCloud2  
    pcl::PointCloud<pcl::PointXYZRGB >::Ptr cloud2(new pcl::PointCloud<pcl::PointXYZRGB >);  
    pcl::fromPCLPointCloud2(pcl_pc2, *cloud2);  
    // Creating a PCL PointCloud2 to paste the unknown object in  
    pcl::PCLPointCloud2 pcl_pc3;  
    pcl_conversions::toPCL(*cloud1, pcl_pc3);  
    pcl::fromPCLPointCloud2(pcl_pc3, *cloud1);  
    // Creating a PCL PointCloud2 to paste the unknown object in  
    pcl::PCLPointCloud2 pcl_pc4;  
    pcl_conversions::toPCL(*cloud2, pcl_pc4);  
    pcl::fromPCLPointCloud2(pcl_pc4, *cloud2);  
    float red=0;  
    uint8_t r = 255, g = 0, b = 0;  // Example: Red color  
    rgb = (*reinterpret_cast<float*>(&rgb));  
    // Determineing which points in the Real_Scene pointcloud and from the Fixed_Scene pointcloud for (size_t i=0; i < cloud1->points.size(); i++) {  
        // Seeing the points that are different into the unknown object pointcloud  
        if(isOutMarge(*(cloud1->points)(0).x), *(cloud1->points)(0).y)) {  
            std::cout << "Cloud 1 point at ("<< i << ","<<(cloud1->points)(0).x) + " x " + (cloud1->points)(0).y" + " y"
            //saveToCloud(cloud3, cloud1, i, r, g, b);  
        } else if(isOutMarge(*(cloud1->points)(0).x), *(cloud1->points)(0).z)) {  
            std::cout << "Cloud 1 point at ("<< i << ","<<(cloud1->points)(0).x) + " x " + (cloud1->points)(0).z" + " z"
            //saveToCloud(cloud4, cloud1, i, r, g, b);  
        } else if(isOutMarge(*(cloud1->points)(0).y), *(cloud1->points)(0).z)) {  
            std::cout << "Cloud 1 point at ("<< i << ","<<(cloud1->points)(0).y) + " y " + (cloud1->points)(0).z" + " z"
            //saveToCloud(cloud5, cloud1, i, r, g, b);  
    }  
    // Publish the data to ROS  
    pub_unknown_obj.publish(unknown_object_cloud);  
}
// Delete noise in the data
pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud3_filtered (new pcl::PointCloud<pcl::PointXYZRGB>);
 pcl::RadiusOutlierRemoval<pcl::PointXYZRGB> rad;
 rad.setInputCloud(cloud3);
 rad.setRadiusSearch(0.1);
 rad.setMinNeighborsInRadius(100);
 rad.filter(cloud3_filtered);

// CYLINDER EXTRACTION
get_cylinder(cloud3_filtered);

// Converting realscene cloud to ROS compatible PointCloud2
sensor_msgs::PointCloud2 real;
real.header.frame_id = "realkinect2frame";
pcl::toPCLPointCloud2(*cloud4, real);
sensor_msgs::PointCloud2 real_cloud;
pcl_conversions::fromPCL(real, real_cloud);
real_cloud.header.frame_id = "realkinect2frame";

// Publish cloud
pub_realscene.publish(real_cloud);
ROS_INFO("Solution found in: %f", (ros::WallTime::now()-start_time).toSec());

// Main code, initialising retrieving point clouds
int main (int argc, char** argv)
{
  // Initialize ROS
  ros::init(argc, argv, "cloud_change_publisher");
  ros::AsyncSpinner spinner(1);
  spinner.start();
  ros::NodeHandle nh;
  //ros::NodeHandle nh2;
  tf_ptr.reset(new tf::TransformListener);
  // Get the two point clouds
  message_filters::Subscriber<sensor_msgs::PointCloud2> real_sub(nh, "/xtion_OH/depth/points_OH", 1);
  message_filters::Subscriber<sensor_msgs::PointCloud2> fixed_sub(nh, "/fixedscene", 1);
  // Sync data streams
  typedef sync_policies::ApproximateTime<
    sensor_msgs::PointCloud2, sensor_msgs::PointCloud2>
    MySyncPolicy;
  Synchronizer<MySyncPolicy> sync(MySyncPolicy(10), real_sub, fixed_sub);
  sync.registerCallback(boost::bind(compute, _1, _2));
  // Create a ROS publisher for the unknown object pointcloud
  pub_pose_cyl = nh.advertise<geometry_msgs::PoseStamped>("pose_cylinder", 1);
  pub_pose_cyl_adjust = nh.advertise<geometry_msgs::PoseStamped>("pose_cylinder_adjust", 1);
  pub_pose_cyl_bottom = nh.advertise<geometry_msgs::PoseStamped>("pose_cylinder_bottom", 1);
  pub_pose_cyl_top = nh.advertise<geometry_msgs::PoseStamped>("pose_cylinder_top", 1);

  // Publish pose to tf topic
  ros::Rate rate(10.0);
  tf::TransformBroadcaster pose_cyl;
  tf::Transform tf_pose_cyl;
  tf::TransformBroadcaster pose_cyl_adjust;
  tf::Transform tf_pose_cyl_adjust;
  tf::TransformBroadcaster pose_cyl_bottom;
  tf::Transform tf_pose_cyl_bottom;
  tf::Transform tf_pose_cyl_top;
  tf::TransformBroadcaster pose_cyl_top;

  while (ros::ok())
  {
    while (pose_cylinder.pose.position.x != 0) {
      tf_pose_cyl.setOrigin(tf::Vector3(pose_cylinder.pose.position.x, pose_cylinder.pose.position.y, pose_cylinder.pose.position.z));
      tf_pose_cyl.setRotation(tf::Quaternion(pose_cylinder.pose.orientation.x, pose_cylinder.pose.orientation.y, pose_cylinder.pose.orientation.z, pose_cylinder.pose.orientation.w));
      pose_cyl.sendTransform(
        tf::StampedTransform(tf_pose_cyl, ros::Time::now(), "base_link", "pose_cyl"));
      rate.sleep();
    }
    tf::StampedTransform tf_bottom_top;
    tf::StampedTransform tf_cyl_adjust;
    if (cylinder_bottom.pose.position.x != 0) {
      tf_pose_cyl_bottom.setRotation(tf::Quaternion(cylinder_bottom.pose.orientation.x, cylinder_bottom.pose.orientation.y, cylinder_bottom.pose.orientation.z, cylinder_bottom.pose.orientation.w));
      tf_pose_cyl_bottom.setOrigin(tf::Vector3(cylinder_bottom.pose.position.x, cylinder_bottom.pose.position.y, cylinder_bottom.pose.position.z));
      tf_pose_cyl_top.setRotation(tf::Quaternion(cylinder_top.pose.orientation.x, cylinder_top.pose.orientation.y, cylinder_top.pose.orientation.z, cylinder_top.pose.orientation.w));
      tf_pose_cyl_top.setOrigin(tf::Vector3(cylinder_top.pose.position.x, cylinder_top.pose.position.y, cylinder_top.pose.position.z));
      pose_cyl_bottom.sendTransform(
        tf::StampedTransform(tf_pose_cyl_bottom, ros::Time::now(), "base_link", "pose_cyl_bottom"));
      pose_cyl_top.sendTransform(
        tf::StampedTransform(tf_pose_cyl_top, ros::Time::now(), "base_link", "pose_cyl_top"));
      rate.sleep();
    }
    try {
      //ros::Time past = ros::Time::now() - ros::Duration(0.031);
      tf_ptr->lookupTransform("pose_cyl_bottom", "pose_cyl_top", ros::Time(0), tf_bottom_top);
      tf_ptr->lookupTransform("pose_cyl_bottom", "pose_cyl_top", ros::Time(0), tf_bottom_top);
    }
    catch (tf::TransformException ex) {
      ROS_ERROR("%s", ex.what());
      ros::Duration(1.0).sleep();
    }
    double length = tf_bottom_top.getOrigin().z();
    if (length < 0.01)
      try {
        //...
E.2. AR Track Alvar launch and configuration

This code in this part shows how to launch and configuration of the ar_track_alvar package.

<launch>
  <arg name="marker_size" default="7.00" />
  <arg name="max_new_marker_error" default="0.05" />
  <arg name="max_track_error" default="0.01" />
  <arg name="cam_image_topic" default="/camera/depth_registered/points" />
  <arg name="cam_info_topic" default="/camera/rgb/camera_info" />
  <arg name="output_frame" default="/camera_depth_optical_frame" />
  <arg name="med_filt_size" default="20" />
  <arg name="bundle_files" default="$(find/uni2423my_pcl)/bundles/bundle_frame_5_cross3.xml" />
  <node name="ar_track_alvar" pkg="ar_track_alvar" type="findMarkerBundles" respawn="false" output="screen" args="$(arg.marker_size),$(arg.max_new_marker_error),$(arg.max_track_error),$(arg.cam_image_topic),$(arg.cam_info_topic),"/>
</launch>

<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<multimarker markers="5">
  <marker index="10" status="1">
    <corner x="6.575" y="-3.925" z="0" />
    <corner x="14.425" y="-3.925" z="0" />
    <corner x="14.425" y="3.925" z="0" />
    <corner x="6.575" y="3.925" z="0" />
  </marker>
  <marker index="12" status="1">
    <corner x="-3.925" y="6.575" z="0" />
    <corner x="3.925" y="6.575" z="0" />
    <corner x="3.925" y="14.425" z="0" />
    <corner x="-3.925" y="14.425" z="0" />
  </marker>
  <marker index="13" status="1">
    <corner x="6.575" y="6.575" z="0" />
    <corner x="14.425" y="6.575" z="0" />
    <corner x="14.425" y="14.425" z="0" />
    <corner x="6.575" y="14.425" z="0" />
  </marker>
  <marker index="13" status="1">
    <corner x="6.575" y="-6.575" z="0" />
    <corner x="14.425" y="-6.575" z="0" />
    <corner x="14.425" y="-14.425" z="0" />
    <corner x="6.575" y="-14.425" z="0" />
  </marker>
  <marker index="17" status="1">
    <corner x="17.075" y="17.075" z="0" />
    <corner x="24.925" y="17.075" z="0" />
    <corner x="24.925" y="24.925" z="0" />
    <corner x="17.075" y="24.925" z="0" />
  </marker>
</multimarker>

E.3. Extrinsic calibration transform

This code transposes the camera pose to the robot base link, in this way the octomap represents the geometry in the scene.

// Code created by Jonathan Meijer
// Transforms camera with respect to base of robot with the use of the AR markers

#include "ros/ros.h"
#include "tf/transform_datatypes.h"
#include "tf/tf.h"
#include "tf/transform_broadcaster.h"
#include "ar_track_alvar_msgs/AlvarMarkers.h"
#include "tf/transform_listener.h"
#include "geometry_msgs/Vector3.h"
#include "geometry_msgs/Quaternion.h"
#include "ar_track_alvar_msgs/AlvarMarker.h"
#include "geometry_msgs/PoseStamped.h"

int main (int argc, char** argv)
{
  // Initialize ROS
  ros::init (argc, argv, "tf_camera");
  ros::NodeHandle nh_tf;
  tf::TransformBroadcaster zero_ref;
  tf::Transform tf_markertozero;

  // Initialize tf
  tf_marksertozero.sendTransform(
    tf::StampedTransform(tf_markertozero, ros::Time::now(), "markertosensor", "markertimeout"));

  // Set transform from marker to zero
  tf::Transform tf_pose_cyl_adjust = tf_marksertozero
    .tf::Pose(tf::Origin(), tf::Quaternion());
  tf_pose_cyl_Adjust.setOrigin(tf::Vector3(0, 0, 0));
  tf_pose_cyl_Adjust.setRotation(tf::Quaternion(0, 0, 0, 1));
  pose_cyl.sendTransform(
    tf::StampedTransform(tf_pose_cyl_adjust, ros::Time::now(), "pose_cyl", "pose_cyl_adjust"));

  rate.sleep();
  ros::spin();
  ros::shutdown();
  return 0;
}
```c
int i = 0;
tf::StampedTransform zero_to_cam;
tf::Transform tf_marker;
ros::Rate rate(10.0);
while (nh_tf.ok()) {
    if (i <= 10) {
        // Specify translation between robot and marker
        tf_marker.setOrigin(tf::Vector3(-0.085, 0.007, 0.007));
        tf_marker.setRotation(tf::Quaternion(0,0,0,1));
        tf::TransformListener listener;
        zero_ref.sendTransform(  
            tf::StampedTransform(tf_marker, ros::Time::now(), "ar_marker_10", "world"));
        try {
            ros::Time past = ros::Time::now() - ros::Duration(0.01);
            listener.waitForTransform("camera_rgb_optical_frame", "world", ros::Time(0), ros::Duration(3.0));
            listener.lookupTransform("camera_rgb_optical_frame", "world", ros::Time(0), zero_to_cam);
        } catch (tf::TransformException ex) {
            ROS_ERROR("%s", ex.what());
            ros::Duration(1.0).sleep();
        }
        tf_marker.setOrigin(zero_to_cam.getOrigin());
        tf_marker.setRotation(zero_to_cam.getRotation());
        listener.clear();
        // Make change permanent, such that resources are kept to a minimum
        // After this step AR marker finding can be turned off
        } else if (i >= 11) {
            zero_ref.sendTransform(tf::StampedTransform(tf_marker, ros::Time::now(), "camera_rgb_optical_frame", "world"));
            rate.sleep();
        }
        rate.sleep();
    }
    i += 1;
    } //Spin
    //ros::spin();
    //ros::shutdown();
    return 0;
```