NTNU – Trondheim
Norwegian University of
Science and Technology

Motion Planning Framework for Industrial Manipulators using the Open Motion Planning Library (OMPL)

Martin Barland

Master of Science in Engineering Cybernetics
Submission date: June 2012
Supervisor: Anton Shiriaev, ITK
Co-supervisor: Uwe Mettin, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

Consider a robotic work cell with a 6-DOF industrial robot manipulator, a conveyor, obstacles and objects. The task is to design an intuitive and easy to use motion planning framework for which only a few task primitives are required. The environment around the robot manipulator shall be modified without having to make other big changes to the framework. In particular the following functionality shall be implemented:

- Generation of smooth collision-free paths for the kinematic model of the robot based on task primitives and the current robot state (DH parameters must be provided, obstacle polygons described, Task query and state is input);
- Generation of optimal trajectories based on a path coordinate subject to kinematic or dynamic constraints.

A visualization of the robot in its environment shall be provided for standard scenarios.

Acknowledgements

First I would like to thank my supervisor Professor Anton Shiriaev for letting me work on this project. It has brought many challenges from which I have learned a lot over the time spent working on this project.

Second, I would like to thank my co-supervisor Dr. Uwe Mettin for his valuable inputs and ideas which has been able to guide me in the right direction when I needed a push in the back.

I would also like to thank Torstein Anderssen Myhre for always being available to answer questions and discuss with, as well as letting me use some of his earlier work with robot manipulators for example when it comes to visualization of a found path.

Finally, I would like to thank Kristina Keseler for her patience, support and reviewing during the time spent on this project. And all my co-students, friends and family for making the last five years some of the best in my life so far.

Martin Barland, June 2, 2012

Abstract

Robotic manipulators are used in many different scenarios these days. If one of these manipulators is moved from one location to another it may require a total reprogramming of that manipulator because of the new environment the robot is working in. This is because the path planning and trajectory planning scheme which works in one environment might not be suitable in another. This text takes a look at how an intuitive and easy to use motion planning framework for finding paths in different static environments or scenarios can be made. The use of the Open Motion Planning Library has been used for the path planning and second-order cone programming solved by SeDuMi in Matlab has been used for finding the time-optimal trajectory.

Sammendrag

Robot manipulatorer er brukt i mange forskjellige scenarioer disse dager. Hvis en av disse robotene blir flyttet fra et sted til et annet kan det kreve en total reprogrammering av roboten. Dette er fordi baneplanleggingen og bevegelsesplanlegging som fungerer et sted ikke nødvendigvis fungerer et annet sted. Denne oppgaven ser på hvordan et intuitivt rammeverk, som er enkelt å bruke, for å finne baner i forskjellige miljøer kan lages. For å finne en bane har "Open Motion Planning Library" blitt brukt. For å finne en bevegelse har det blir brukt andre-ordnes kjegle programmering for å sette opp et minimumtid optimaliseringsproblem. Dette har så blitt løst i Matlab ved hjelp av SeDuMi.

Contents

1 Introduction **1**
1.1 Scope 1

2 Problem Formulation **3**
2.1 Importance of Path Planning 3
2.2 Trajectory Generation Problem 4

3 Path Planning **5**
3.1 Path Planning Concepts and Terminology 5
3.2 Methods of Path Planning 6
3.3 Quick Introduction to The Open Motion Planning Library 8
3.4 Path Planning with The Open Motion Planning Library 9
3.4.1 State validation and Collision Detection 12
3.4.2 Collision Models 13
3.4.3 Available Planners 17
3.4.4 Finding a Valid Path 20
3.4.5 Path Post-Processing 21

4 Framework in the Open Motion Planning Library **25**
4.1 Overview of Framework 25
4.2 Benchmarking of Planners in OMPL 27
4.3 Visualization of Paths 30

5 Trajectory Generation for Found Path **33**
5.1 Two Step Trajectory Planning 33
5.2 Second-Order Cone Programming 35
5.3 Optimization by Second-Order Cone Programming . . 36
5.3.1 Object function 36

5.3.2 Reformulation from non-linear optimization problem to second-order cone program 37
5.4 Solving the Optimization Problem Using SeDuMi in Matlab 40
5.5 Finding the Time-Optimal Trajectory 41
5.5.1 Illustration of Obstacle-free point-to-point motion 41
5.5.2 Illustration of obstacle-"clouded" point-to-point motion 45

6 Discussion and Future Work **51**

A **55**
A.1 Simple Examples of Path Planning With and Without `SimpleSetup` 55

B **59**
B.1 An example of a .RAW file for an environment 59

Bibliography **62**

List of Figures

3.1 An example of how the RRT algorithm works by spreading out in a plane using 10, 100 and 500 nodes. Plot is created by running RRT in Matlab and plotting the results. 8
3.2 The Open Motion Planning Library API overview. The blocks in dark blue are blocks the user has to declare to be able to solve a path planning problem. The light blue blocks with lined edges have to be declared in addition if `SimpleSetup` is not used. Light blue blocks can be declared, but have default implementation if the user does not include them. 10
3.3 Flow chart of the order of a path planning problem. The goal for the framework that will be built is that, given a model of the robot and the environment, the only thing the user has to do is give a start and a goal position in the files with the same name. 14
3.4 One of six link models built with triangular polygons in order to be easy to use with the Proximity Query Package 16
3.5 Steps of building an environment in Blender. Objects are created by adding simple meshes around the robot and customizing them, whereas the robot was imported from a CAD file 18
3.6 IRB140 Robot Manipulator, Kinematic structure . . . 20
3.7 Simplification of path shown as number of states 22

4.1 Overview of path planning framework 26
4.2 Path planning framework with inputs and outputs . . . 28
4.3 Environment used for benchmarking of planners 29

4.4 Visualization of a path in Blender 32

5.1 Cone Constraint visualization 36
5.2 Full framework consists of both path planning and trajectory planning. 42
5.3 Visualization of the path used to find a trajectory in a obstacle free environment. 43
5.4 Joint angles on a path in obstacle-free environment. . . 43
5.5 Joint velocities on a path in obstacle-free environment. 44
5.6 Joint accelerations on a path in obstacle-free environment. 44
5.7 Joint torques on a path in obstacle-free environment. The dotted lines are torque constraints. 44
5.8 Path coordinate with respect to time on path in obstacle-free environment. 45
5.9 Velocity profile for a path in an obstacle-free environment. 45
5.10 Visualization of the path used to find a trajectory in an obstacle-clouded environment. 46
5.11 Joint angles on a path in an obstacle-clouded environment. 47
5.12 Joint velocities on a path in an obstacle-clouded environment. 47
5.13 Joint accelerations on path in an obstacle-clouded environment. 47
5.14 Joint torques on a path in an obstacle-clouded environment. The dotted lines are torque constraints. 48
5.15 Path coordinate with respect to time on a path in an obstacle-clouded environment. 48
5.16 Velocity profile for a path in an obstacle-clouded environment. 48

Chapter 1

Introduction

1.1 Scope

The modern era of robotics began around 1959. Since then robot manipulators have been an important factor in industry to be able to relief humans from doing repetitive labor, such as for example assembling, grinding and welding. The need for robot manipulators is still big as they are able to do more and more advanced tasks and thereby increasing productivity and reducing the need for human operators in dangerous environments to reduce damage on personell. One of the challenges with these robot manipulators is the task of finding the path it should be going when solving a task, especially when the robot is relocated from one place to another. Such a relocation may need a complete reprogramming of the robot from a human operator.

In this text we will investigate a method for doing path-planning, and after that trajectory-planning, for a robot where it is easy to tell the robot about the environment it is working in, so no huge reprogramming is needed. If all the user has to give the program is small task primitives such as start and goal position, and perhaps time, it will be easier to use and more intuitive than the traditional way of having a trained application engineer reprogram the robot for every problem to be solved. There are path-planning tools which can help with this problem, but many of them are outdated or rarely updated. We will

in this text therefore take a closer look at how a relatively new tool for path-planning, called the Open Motion Planning Library [1], can be used to help solving this problem.

Chapter 2 is the problem formulation for this project. Some difficulties and challenges with path planning and trajectory generation is introduced.

In Chapter 3 we will take a closer look at path planning for robotic manipulators. The main feature of this chapter is to see how the use of the Open Motion Planning Library can help with solving a classic path-planning problem. It give a short introduction of this library and then some of its useful features.

In Chapter 4 a framework for path planning based around OMPL will be presented. It also shows a benchmarking of some of the available planners in OMPL in order to see how they perform on one specific path planning problem. Finally in this chapter it is shown how a visualization of a path found can be done in a 3D graphics program.

Chapter 5 will discuss a method for calculating optimal trajectories for robot manipulators. It will then attempt to take the path found from the framework described in Chapter 4 and turn it into an optimal trajectory by using the method discussed.

The path-planning is done in C++ while the trajectory planning is done in Matlab. The visualization of the paths found is made in the open source program for 3D graphics called Blender[2]. This program is used both to show how the user easily can create new environment models without having to reprogram the entire robot and for the visualization of a found path. Files related to path-planning, trajectory planning and visualization are attached to this text.

[1]http://ompl.kavrakilab.org/
[2]http://www.blender.org/

Chapter 2

Problem Formulation

2.1 Importance of Path Planning

Path planning for industrial robot manipulator is a very important task that has to be done correct to avoid damage on personell, other equipment in the work area or the robot manipulator itself. These days industrial robot manipulators are working in different environments with different challenges. There might be tight spaces, humans working in same environment or other challenges that the manipulator has to be aware of in order to find a collision free path.

When planning for the ABB IRB140 the problem quickly becomes very complex, because this robot manipulator has six degrees-of-freedom. It requires a lot of computational power when solving a path planning problem because there can be infinitely many paths between the starting position and goal position of the robot. If the robot itself and the environment has complicated collision models, even more computational power is needed.

There are several tools that can be used to help solve path planning problems. In this text a framework for path planning for the ABB IRB140 robotic manipulator has been developed using a relatively new tool, the Open Motion Planning Library [1].

2.2 Trajectory Generation Problem

Trajectory planning for robot manipulators share many of the same difficulties as the path planning. The more degrees-of-freedom the robot has, the more complex the system gets. There is also infinitely many trajectories for the path found, just as there are infinitely many paths between a starting position and a goal position. The goal of the trajectory planning is to take the path which is the solution of the path planning problem and find the time history of the position, velocity and acceleration for each link so this can be provided to the tracking controllers which are embedded in the robot control software. It is important to find the optimal trajectory in order to save both time and possibly energy, or any other factors the user wants to take into consideration. It is also very important to make sure none of the constraints of the robot is broken. Typical constraints on a robot manipulator can for example be how much torque can be applied to each joint.

There are several methods for solving an optimal trajectory problem. Some of them include using polynomials of different degrees to ensure that the constraints on the robot manipulator is not broken. To find an optimal trajectory it is possible to set the problem up as a optimization problem with for example the time or energy as the object function. This depends purely on what the user wants to optimize. In this project the optimal time-trajectory will be found by using second-order cone programming and the second-order cone program solver SeDuMi [1] in Matlab.

[1]http://sedumi.ie.lehigh.edu/

Chapter 3

Path Planning

3.1 Path Planning Concepts and Terminology

The goal of a path planning problem is to find a collision-free path from an initial state to a goal state in the world space, denoted by $\mathcal{W}$. This world space is a 3 dimensional Euclidian space, which is the space where all physical objects are defined. The collection of these objects are gathered in the obstacle space, $\mathcal{O}$.

The robot itself consists of links which also are represented in $\mathcal{W}$. These links are described in the world space by position and orientation. If the position and orientation is known for each link, then the robot is fully described in the world space.

The robot used in this project is an ABB IRB140 which has six degrees-of-freedom, corresponding to one revolute joint for each link. When we collect all of these joint values in order we get a representation of the robot itself as a vector of real numbers in the configuration space $\mathcal{C}$. The configuration space $\mathcal{C}$ is dependent on what type of robot the user is working with. A rotational join adds a dimension to the robot which is homeomorphic to a circle, $\mathcal{S}^1$. A prismatic joint corresponds to a dimension that is homeomorphic to an interval in the real numbers, $[a, b] \in \mathbb{R}$. The ABB IRB140 consists of six rotational joints and

therefore has a configuration space which is a product of six circles, which is the six dimensional torus, $\mathcal{S} \times \cdots \times \mathcal{S} = \mathcal{S}^6$. The set of configurations that avoids collision with obstacles in the world space, $\mathcal{W}$, is called the free space, $\mathcal{C}_{free}$, and a valid path consists purely of configurations located in $\mathcal{C}_{free}$.

3.2 Methods of Path Planning

There are many different methods that can be used to find a valid collision-free path and all of them has their weaknesses and strengths. In this section we will take a quick look at some of these methods before looking at what methods the Open Motion Planning Library provides to the user.

One strategy that has been used to explore $\mathcal{C}_{free}$ is using an artificial potential field, [2, page 168]. When using this method the idea is to treat the robot as a point particle in the configuration space under the influence of an artificial potential field. This field is then constructed such that the robot is attracted to the final configuration and repelled from the obstacles and boundaries on the space. By representing this potential field as a potential function this problem becomes an optimization problem. One big problem with this method is that the robot can easily get trapped in local minimums if the potential field is not constructed properly.

It is also possible in some instances to partition the workspace into descrete cells corresponding to the obstacle free portion of the environment, for example in [3, Chapter 6.2.2]. This can then be turned into a graph where the vertices represent the individual cells and the edges indicate adjancency among the cells. By doing this the problem becomes a classical search problem from the cell with the starting position to the cell with the end position. This does however prove to be difficult in practice if the robot has complex non-linear dynamics, where it is not clear how to move the system from one cell to another. It is also difficult, and impractical, to partition high-dimensional configuration spaces into free cells.

One of the newer methods for path planning which there recently

has been much research on is so called sampling-based path planning. These methods samples the configuration space and remembers the configurations that lie in $\mathcal{C}_{free}$. These configurations are then made into a roadmap by connecting two points if the line segment between them is completely in $\mathcal{C}_{free}$. The algorithms for this kind of path planning usually works well for high-dimensional configuration spaces because their running time is not exponentially dependent on the dimension of $\mathcal{C}$.

One of these methods is called Probabilistic Roadmap Method, PRM[4]. This is one of the most efficient methods today for a robot working in a static workspace. It has two phases, one learning phase and one query phase. In the learning phase it finds collision-free configurations and bind them together. In the query phase the method actually finds the path by connecting the initial and goal configuration by finding a way between them on the roadmap made in the learning phase.

Another fairly popular sampling based path planning method is called Rapidly-exploring Random Tree, or RRT[5]. It builds a graph in the form of a tree from the starting position which expands until it reaches the goal position. One of the reasons that RRT is well-suited for path planning is that it reduces the computational time by expanding faster in a direction with few other points and therefore covers the entire configuration space relatively fast. How RRT spreads in a plane without obstacles can be seen in Figure 3.1. From left to right RRT has been run with 10, 100, 500 nodes to search the plane.

All the planners available in The Open Motion Planning Library are sampling based planners. They are all powerful when it comes to systems with differential constraints or systems with many degrees of freedom. One of the drawbacks of sampling based methods however is that they cannot recognize if a problem does not have a solution. This is why, in OMPL, the user has to define for how long the planner is allowed to search for a solution. More about this use of OMPL will be covered in Section 3.4.

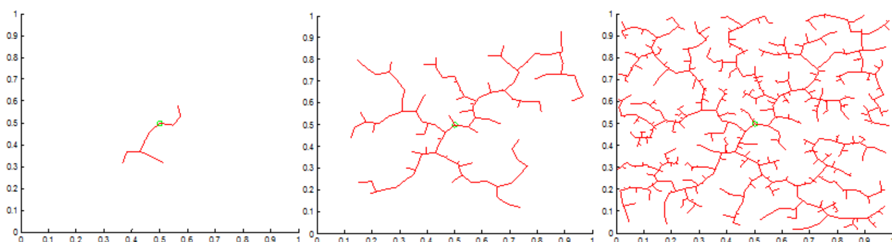

Figure 3.1: An example of how the RRT algorithm works by spreading out in a plane using 10, 100 and 500 nodes. Plot is created by running RRT in Matlab and plotting the results.

3.3 Quick Introduction to The Open Motion Planning Library

The Open Motion Planning Library, OMPL, has been developed by the people at the Kavraki Lab[6] at Rice University under direction by Lydia E. Kavraki. It consists of many state-of-the-art sampling-based motion planning algorithms and also includes other functionality to help users solve path planning problems. The library does not contain any code related to collision detection or visualization so the user is free to use whatever tools for this as he prefers. This also means that OMPL can be used to find paths for all kind of robotic manipulators, be it industrial robot manipulators with six degrees-of-freedom, as the ABB IRB140, robots with fewer or more degrees of freedom, mobile robots or path planning for rigid bodies.

OMPL uses some other terminology than what is used in section 3.1 when it comes to path planning. The configuration space, $\mathcal{C}$, is in OMPL called the state space as each configuration found is a state on a path found. The world space, $\mathcal{W}$, is called the work space, which is the physical space that the robot operates in. The free configuration space, $\mathcal{C}_{free}$ is now called the free state space since the configuration space is called the state space, and a path is a continuous mapping of states in the state space. A path is collision free if each element of the path is an element of the free state space.

Since OMPL does not include any explicit representation of the geometry of the workspace or the robot operating in it the user can, and must, select computational representation for both the robot and the environment.The user must also provide an explicit state validity checker or collision detection method. The Open Motion Planning Library does however provide the user with an abstract representation for all the core concepts in motion planning, including the state space, control state, state validity checking, sampling, goal representations and planners. The application programming interface of OMPL is shown in Figure 3.2.

The Open Motion Planning Library lets the user have the opportunity to customize many of the blocks shown in Figure 3.2. If OMPL does not include a ready to use state space for the problem to be solved in, it is easy to create. In this project the robot manipulator has six degrees-of-freedom and therefore we need a six dimensional state space. This is created by making a six dimensional `RealVectorStateSpace`. It is also possible to add and subtract available configuration spaces in OMPL to make new ones.

The state validity checker is something that has to be implemented by the user so this can be written in any way the user choose to. When these two blocks have been declared their information is gathered in the `SpaceInformation` which contains all the information about the space that the path planning is done in. In this project only planners already available in OMPL have been used. However if the user wants to create or test new planners this can be done by defining the new planner as a class that inherits from the `Planner` class in OMPL. Overall OMPL has a lot of opportunities already implemented, but if the user wants to use something that is not available in OMPL it can easily be customized to serve most needs.

3.4 Path Planning with The Open Motion Planning Library

Path planning in this project is done with the use of the Open Motion Planning Library, OMPL. The Open Motion Planning Library is

only when planning with differential constraints

ControlSampler Implements sampling of controls for a specific ControlSpace

ControlSpace Represents the control space the planner uses to represent inputs to the system being planned for

StatePropagator Returns the state obtained by applying a control to some arbitrary inital state

StateSpace Represents the state space in which planning is performed; implements topology-specific functions: distance, interpolation, state (de)allocation.

StateSampler Implements uniform and Gaussian sampling of states for a specific StateSpace

ProjectionEvaluator Computes projections from states of a specific StateSpace to a low-dimensional Euclidean space.

MotionValidator Provides the ability to check the validity of path segments using the interpolation provided by the StateSpace.

SpaceInformation Provides routines typically used by motion planners; combines the functionality of classes it depends on.

StateValidityChecker Decides whether a given state from a specific StateSpace is valid.

ValidStateSampler Provides the ability to sample valid states.

Path Representation of a path; used to represent a solution to a planning problem.

Planner Solves a motion planning problem.

SimpleSetup Provides a simple way of setting up all needed classes without limiting functionality.

ProblemDefinition Specifies the instance of the planning problem; requires definition of start states and a goal.

Goal Representation of a goal.

User code

User must instantiate this class.
User must instantiate this class unless SimpleSetup is used.
User can instantiate this class, but defaults are provided.
A → B *A* is owned by *B*.

Figure 3.2: The Open Motion Planning Library API overview. The blocks in dark blue are blocks the user has to declare to be able to solve a path planning problem. The light blue blocks with lined edges have to be declared in addition if `SimpleSetup` is not used. Light blue blocks can be declared, but have default implementation if the user does not include them.

a great resource for this task as it has several different planners available and can be used on any problem since state validation has to be implemented by the user. Because the user decides what is a valid state or not it can be used for all kinds of problems such as rigid body path planning, as well as for mobile robots, or in this case industrial robot manipulators.

When setting up for a path planning problem the user has to make sure to create all objects needed for solving a path planning problem. This includes an instance of a `StateSpace` that the user will be planning in, a `SpaceInformation` that contains all information about the space where the planning is done and a `ProblemDefinition` which is the definition of the problem to be solved. The user also have to create a start and goal states and select a `Planner` that is to be used to solve the problem.

The user can however choose to use a class called `SimpleSetup` to simplify the setup procedure. This class makes it so that the user only has to create the state space, start state and goal state. `SimpleSetup` then instantiates the `SpaceInformation` and the `ProblemDefinition`. It also allows for the retrieval of all of these subcomponents for further customization. The use of `SimpleSetup` ensures that all objects are properly created before the planning begins. C++ code for a simple example of how a path planning problem can be implemented with, and without, `SimpleSetup` is shown in Appendix A.1. In this project `SimpleSetup` has been used for all code as there was not found any problems which could not be solved by `SimpleSetup`

This text uses the ABB IRB140 robotic manipulator together with OMPL to be able to find a valid path that then can be used to create a trajectory for the robot manipulator. This is all done offline with no changes in the environment during runtime, but OMPL can also be used for online path planning. This may however cause difficulties if there are hard time constraints as the planner has to be restarted every time the environment model has been updated or a new obstruction has been found.

3.4.1 State validation and Collision Detection

State validation is very important when finding a path in OMPL, as in any path planning problem. If no method of defining what a valid state is made OMPL sets all states to be valid. This is of course not wanted behaviour, so it is up to the user to define what makes a state valid depending on what type of problem is to be solved. In this project a valid state is defined as a state where the robot avoids self-collision and collision with obstacles in the workspace as well as staying inside the boundaries of the state space. To achieve this the Proximity Query Package, PQP [7], has been used to check for collisions.

The Open Motion Planning Library has two abstract classes available in order to allow the user to specify the notion of state validity. The first is the actual state validity checker that checks if a state is valid and can be used on a path. The other part is a motion validator. The motion validator evaluates the validity of motions between two given states. The user can either implement his own motion validator as a class or he can use the default motion validator which is a discrete motion validator. The advantage of using this discrete motion validator is that it uses functionality from the state validity checker that the user has to make. The disadvantage is that if the resolution is set too high the program will have the risk of skipping too many states and therefore it might not detect a collision even though there is one. If it is set to low however it may check so many states it may be impractical when looking for a solution because the time usage will be a lot higher. In this project the default discrete motion validity checker has been used.

The state validity checker the user has to implement can either be implemented as a class that inherits from the abstract class `StateValidityChecker` or as an `isValid` function. If the user decides to create a state validity checker class the only thing that is absolutely necessery to have in this class is a function called `isValid`. This project has created a class to be able to gather all information needed to check for valid states, such as collision detection with the use of PQP, in one class. When either the `isValid` function in a state validity checker class or the function for validity checking has been implemented it is used by telling the defined `spaceinformation` pointer where it can find the function.

Algorithm 1 Set state validity checker without SimpleSetup

```
base::SpaceInformationPtr si(space);
//Call if a state validity checker class has been declared
si->setStateValidityChecker(base::StateValidityCheckerPtr(new
myStateValidityCheckerClass(si));
//Call if only a state validity function has been declared
si->setValidityChecker(boost::bind(&myStateValidityCheckerClass(si)));

si->setStateValidityCheckerResolution(0.03);
si->setup();
```

Algorithm 1 shows how to set the state validity checker if `SimpleSetup` has not been used. If `SimpleSetup`, described in 3.3, has been used when setting up the planning problem the state validity checker is set by telling `SimpleSetup` wwhere the `isValid` function is located and then tell what the `SpaceInformation` is, since `SimpleSetup` automatically creates the `SpaceInformationPtr` for the user. `simpleSetup` sets a default state validity checker resolution, but the user is also free to change this if desired. Figure 3.3 show, in order, all steps needed in order to solve a path planning problem.

3.4.2 Collision Models

To create a state validity checker with collision detection in this project a simplified collision model for the ABB IRB140 robotic manipulator has been made. The CAD, Computer Aided Design, model that is available from ABB's webpage [1] for this robot is too complex to be used for state validity checking as it has too many details and the computational time would therefore be too long for practical use. Therefore a simplified collision model has been made to make the computation of state validity faster. This model models the robot manipulator as six boxes, one box for each link. If the robot is working in an non-empty environment a model for the environment is also needed in order to detect collisions with the environment. This model can for example be

[1]http://www.abb.com/product/seitp327/7c4717912301eb02c1256efc00278a26.aspx

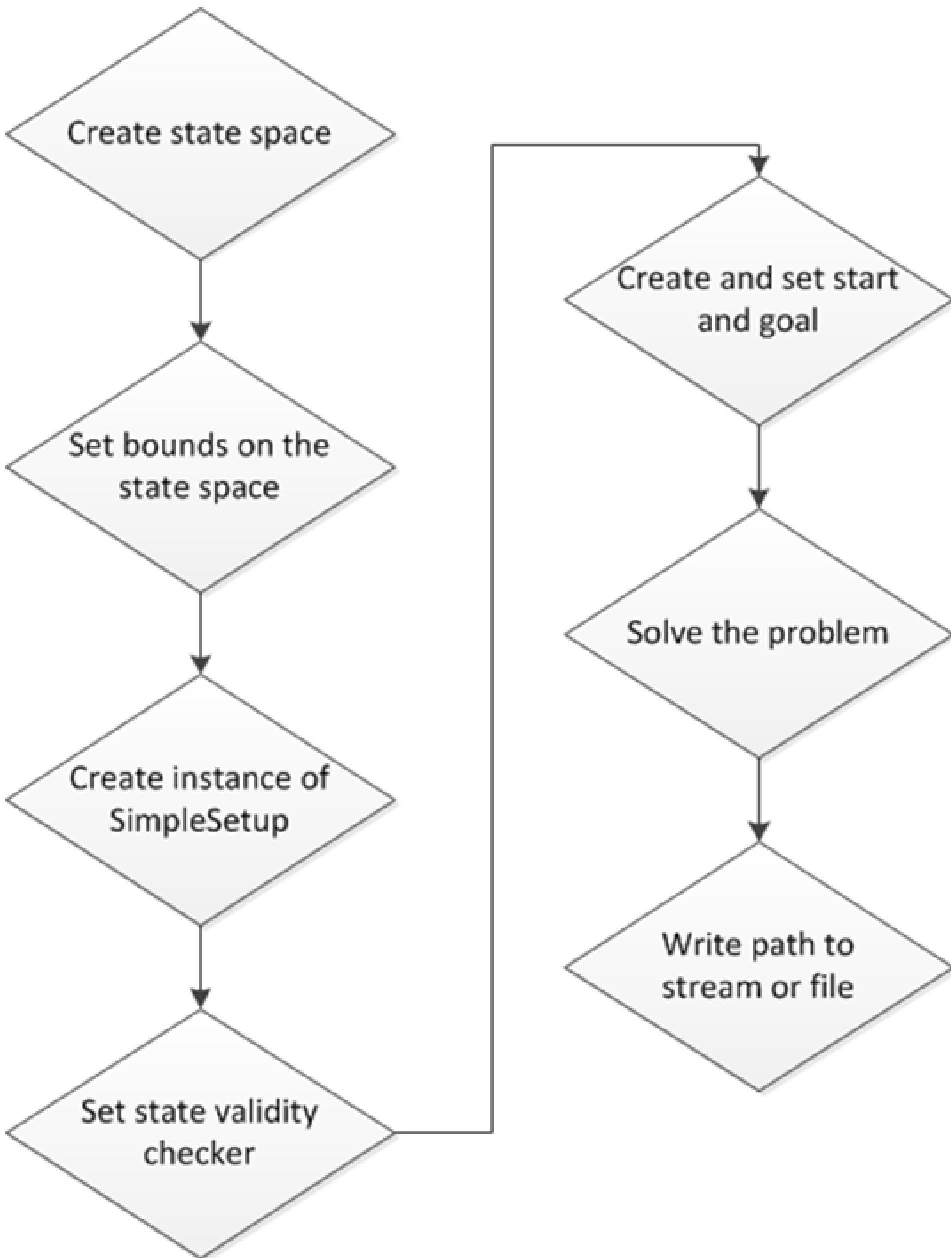

Figure 3.3: Flow chart of the order of a path planning problem. The goal for the framework that will be built is that, given a model of the robot and the environment, the only thing the user has to do is give a start and a goal position in the files with the same name.

built in Blender and then be exported in the .RAW file format which can be read by the path planning framework made in this project. This makes it possible to quickly create many different environments which the robot can operate in.

Both the model of the robot links and the environment is represented by polygonal meshes, which in this case are triangles. This is a deliberate choice as it makes it easy to use the Proximity Query Package for collisions detection. The Proximity Query Package, PQP, creates `PQP_models` and checks for collisions between them. The models are created by making triangles with x, y and z coordinate for each of the three points making the triangle which is then added to a model. PQP has three different ways to inform the user about how these models are related. The first one is noticing if there is a collision between two triangles. When a model is built and the user is adding triangles to the model every triangle has to be given a unique number when it is being added. This makes it so that the user can check which triangles that are colliding and this can be very useful for debugging, for example if two triangles on the same model have collided there is definitively something wrong as this should be impossible. It also gives the user an easy way to see which links are colliding with other links or the environment if a collision is detected. The two other methods to check for proximity includes the option to check the distance between the models or check if two models are closer than some tolerance set by the user.

In this project a collision is detected when two or more triangles collide with each other. How PQP checks for a collision is shown in Algorithm 2. When PQP checks for collisions between two models the user also has to supply a rotation matrix, R, and a translation vector, T, to tell PQP where the models are located. Since the PQP models for the robot are calculated at their current position in the workspace with the current rotation, the position vector T equals the origin and the rotation matrix R equals an identity matrix in this project.

As mentioned in the beginning of Section 3.4.1 it is up to the user to define what a valid state in the Open Motion Planning Library is. In this project a valid state is defined as a state where there are no collisions and the state does not break any of the given boundaries. A

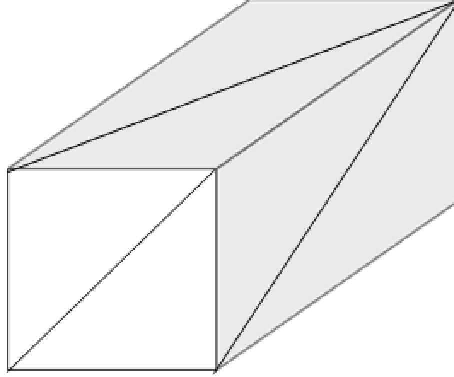

Figure 3.4: One of six link models built with triangular polygons in order to be easy to use with the Proximity Query Package

Algorithm 2 Collision checking in PQP between two links

```
#include "PQP.h"

PQP_Model m1, m2;
m1.BeginModel();
m2.BeginModel();

//Create 12 triangles per link, 24 altogether
PQP_REAL p1[3], p2[3], p3[3];
PQP_REAL q1[3], q2[3], q3[3];
...
//Initialize the points and then add them to the models
m1.AddTri(p1, p2, p3, 0);
m2.AddTri(q1, q2, q3, 1);
...
m1. EndModel();
m2.EndModel();

//Perform collision check between two models
PQP_CollideResult cres;
PQP_Collide(&cres, R1, T1, &m1, R2, T2, &m2);
//The result of the collision check is saved in a result structure
int colliding = cres.Colliding(); // colliding ==1 if a collision accured
```

collision is defined as a self collision if two or more links on the robot collide with each other, and as a collision with the environment if one or more links on the robot collide with the environment it is working in. Each link has its own file where the coordinates of the triangles making up the box-model is and this is read to the program via a user created function called `importLink`. Since each link is a simple box every link model consists of 12 triangles, 2 for each side, as is demonstrated in Figure 3.4.

The environment is built in Blender around a model of the ABB IRB140. The progress of creating an environment is done by opening a file with a model of the robot and adding meshes available in Blender around the model. These meshes can easily be modified to change size and location so the environment can represent many different environments for the robot manipulator to work in. The making of one environment is shown in Figure 3.5. A short film of how to create a new environment is attached and named `howToCreateEnvironment.wmv`.

When an environment is made it can be exported in the .RAW file format from Blender by marking all the objects the user wants to include in the environment. Exporting in this file format makes it so that the environment file consist of triangles, an example of how such a file looks is shown in Appendix B.1. The file then gets read by the user created function `importEnvironment` and since the file consists of triangles it is easy to create a PQP model for the environment. The environment can have as many or as few objects as the user wants it to have.

3.4.3 Available Planners

The Open Motion Planning Library has as already mentioned many planners available for the user to use. Additionally the user has the opportunity to add new planners if that is wanted. All the planners available in OMPL are sampling-based planning algorithms. As of this moment there are 11 different planners available for geometric path planning, all shown in Table 3.1.

In this project however the main focus has been on the planner called

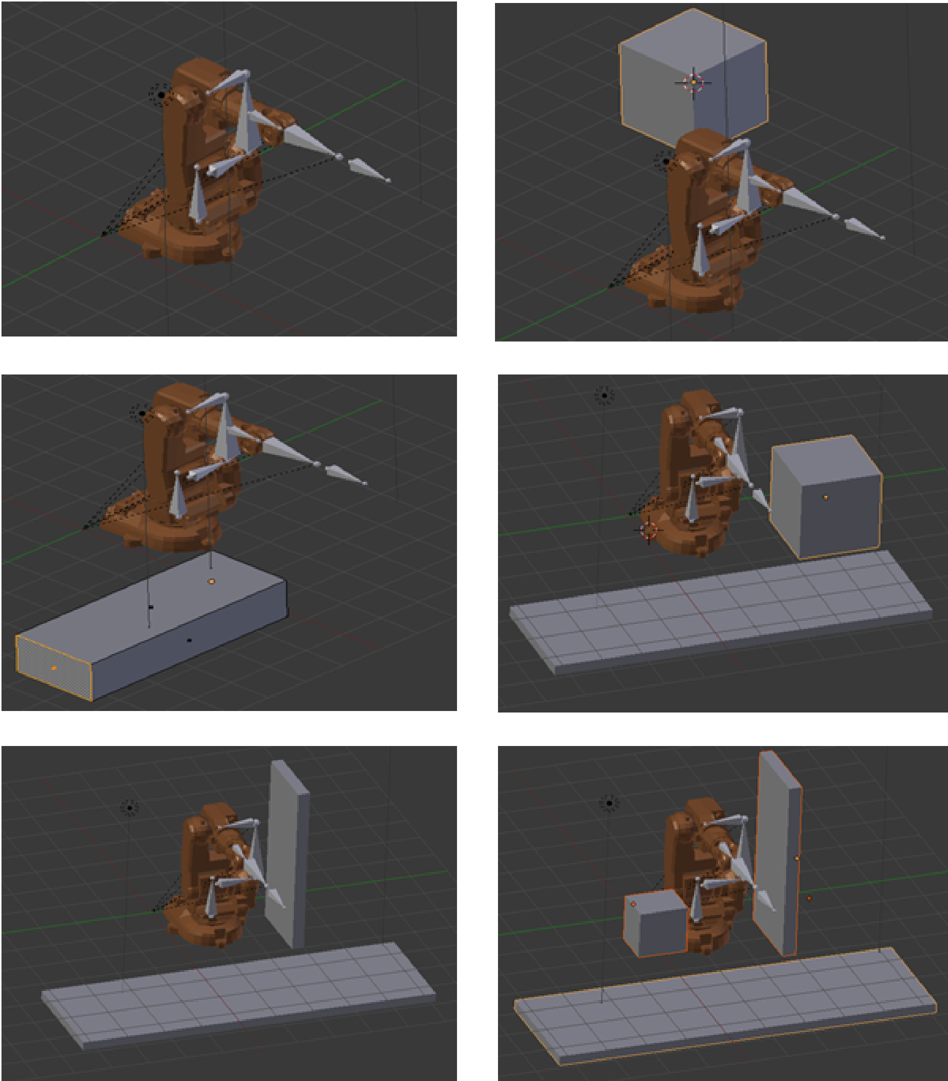

Figure 3.5: Steps of building an environment in Blender. Objects are created by adding simple meshes around the robot and customizing them, whereas the robot was imported from a CAD file

PRM	RRT	EST
SBL	KPIECE	BKPIECE
LBKPIECE	LazyRRT	RRTCONNECT
RRT	BallTreeRRT	SyCLoP (coming soon)

Table 3.1: Planners available for geometric planners in OMPL

Rapidly-exploring Random Tree, RRT, created by Steve LaValle. RRT is, as mentioned in section 3.2, a sampling based planner that creates many states that have to be checked for collisions on its way to find a valid path. At every new state the planner creates in the state space the function `isValid`, in our state validity checker class, is run to check if this state is valid or not.

When a new state is made the first thing `isValid` does is to calculate where each robot link is located and which rotation it has in the workspace at this state. This is done by using forward kinematics and the Denavit-Hartenberg convention, [2, page 76]. A kinematic model for the ABB IRB140 is shown in Figure 3.6 and DH parameters for this manipulator are shown in Table 3.2.

When the position and rotation of each link is found a PQP model is made for each link at its current position. When all links have their own PQP model the collision detection is done by checking for collision between every pair of links and then between every link and the environment. If a collision is detected the state is invalid and if no collision is found the state is valid and can possibly be a state on the path that is found as a solution to the planning problem that is trying to be solved.

d	θ	a	α
352	θ_1^*	70	$-90°$
0	$\theta_2^* - \pi/2$	360	0
0	θ_3^*	0	$-90°$
445	θ_4^*	0	$90°$
0	θ_5^*	0	$-90°$
0	$\theta_6^* + \pi$	0	0

Table 3.2: DH parameters for ABB IRB 140

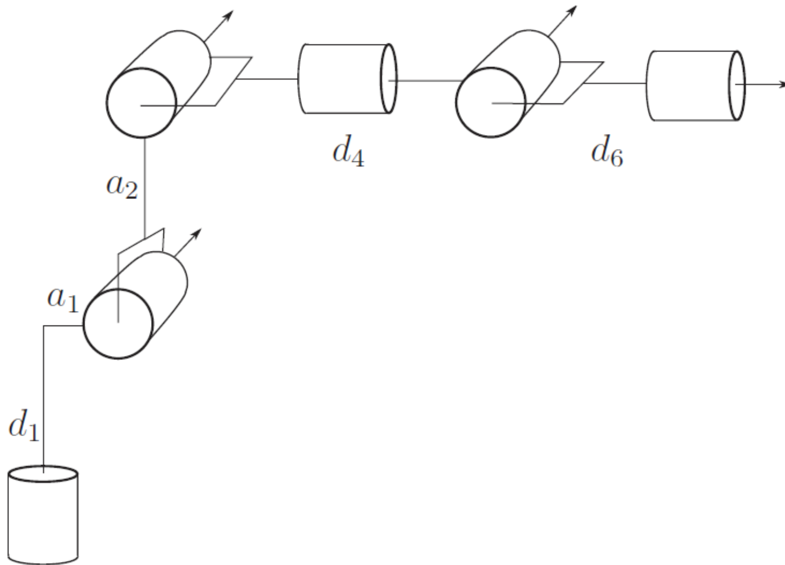

Figure 3.6: IRB140 Robot Manipulator, Kinematic structure

3.4.4 Finding a Valid Path

A planner is created by using the command `ompl::base::PlannerPtr planner(new og::RRT(spaceInformation))`, where RRT can be switched with any planner available if that is wanted. It is then also necessary to tell the planner which problem is to be solved and then run setup on the planner, `planner->setup`. If the user uses `SimpeSetup` a specific planner does not have to be set because OMPL will use a default planner if none is given. However since this project wants to take a closer look at how RRT will be able to solve the problem we have to tell the planner that we want to use RRT as the planner. When a planner is set it is time to let OMPL try to solve the problem given.

Since all the planners are sampling based they will not be able to find out if a solution does not exist, as mentioned in Section 3.2. The user therefore has to decide how long the planner is allowed to search for a solution. `bool solved = ss.solve(int time)` lets the planner search for a solution for the amount of seconds given in the integer time. `solved` is equal to 1 if a solution is found.

If no solution is found within the time period given, OMPL often finds an approximate solution. This solution is, with the experience gained in this project, not advisable to use. It is very often not even close to the solution the user is looking for, since the state space for the robot manipulator used in this project is high dimensional, in this case it is six dimensional. Whether or not an approximate solution is found can be checked by `bool approx = ss.getGoal()->isApproximate()`. In this project an approximate solution has not been approved as a valid solution of the path planning problem. A valid path is therefore only found, and approved, if `solved==1` and `approx != 1`.

3.4.5 Path Post-Processing

When a path is found OMPL has several methods of processing this path or provide more information to the user if that is wanted. One of the most useful path processing mechanisms in OMPL is the function called `simplifySolution` which is a function in `SimpleSetup`. This function attempts to simplify the path found, by running two other functions available in OMPL and then try to smooth it.

The first function to which tries to remove vertices, or states, on the path found while still keeping the path valid is called `reduceVertices`. It does this by so called "short-cutting". Short-cutting is done by checking if it is possible to go from non-consecutive way-points on the path. If this is possible the path is shortened by removing the way-points in between.

The other function that is used in `simplifySolution` is called `collapseCloseVertic` This function tries to remove vertices that are close to each other from the path found while still keeping the path valid. If some vertices for example are on the same straight line only the ones on the ends are really needed as the others do not contribute to avoiding obstacles in the work space. The user created function `isValid` is used for checking for collisions when using `simplifySolution` to make sure all simplifications are valid.

To see how `simplifySolution` works the path planning has been run without any environment. The smoothest path between only the start-

```
Info:    RRT: Starting with 1 states
Info:    RRT: Created 29 states
Info:    SimpleSetup: Solution found in 0.818566 seconds
******** Before simplification ********
Geometric path with 5 states
RealVectorState [6.2831 0 0 0 0 0]
RealVectorState [2.72641 0.829351 -0.406099 0.734841 -1.36052 0.627692]
RealVectorState [1.43727 1.17057 -0.599307 -1.49832 -1.19469 4.00092]
RealVectorState [-1.46858 0.282008 0.887089 -1.15421 0.360981 1.94485]
RealVectorState [0 0 0 0 0 0]

Info:    SimpleSetup: Path simplification took 0.170541 seconds

******** After Simplification ********
Geometric path with 2 states
RealVectorState [6.2831 0 0 0 0 0]
RealVectorState [0 0 0 0 0 0]
```

Figure 3.7: Simplification of path shown as number of states

ing position and the goal position is a straight line so this is what we want to get as our output. Since all planners in OMPL are sampling-based a path will almost always consist of several states before anything has been done with the path. In Figure 3.7 it is shown which states the path consist of before and after `simlifySolution` when the path planning program is run. In this example all redundant states has been removed so we get the result we wanted.

After removing verices `simplifySolution` also tries to smooth the path as good as it can. In the previous example the path ended up being a straight line, and a path can not become any smoother than that. However if the resulting path from the path planning program has curves or other non-smooth properties `simplifySolution` will try to add new states into the path by using a function called `smoothBspline`. This function may add only a few states to make the path more smooth or it may increase the number of states significantly. A smooth path is wanted when trying to create a good trajectory for the robot, so in this case we are not worried about the potential for adding many states to our path as long as the path gets smoother. This can be seen in Table 3.3 where five successfully found paths, with the use of Rapidly-exploring Random Trees, has been processed with `simplifySolution`.

The environment used is the same as the one being used in benchmarking of planners shown in Figure 4.3. It shows how many states the path found initially had, how many it has after it has been simplified and smoothed and how long time the simplification of the path took. A measurement for how long time the planners used to find the initial path is found in section 4.2, Table 4.1, where a benchmarking of some of the planners found in OMPL has been run. There is a big difference in how many states that are needed to smooth out the path.

An increase from 7 states to 81 as shown in run 3 is a big increase in number of states, but in this project we just want the path to get smoothed out so we can use it for trajectory generation. Simplification does take time, usually over a second, so if there is a tight time limit on the path planning and it is not needed with a very smooth path the user can always choose to not smooth it in order to save time.

Run	1	2	3	4	5
Initial number of states	9	10	7	7	12
After simplifySolution	53	45	81	45	45
Time spent (s)	1.1798	1.508807	1.15022	0.991133	1.068794

Table 3.3: Number of states before and after `simplifySolution`

Another useful path processing mechanism is the ability to create more states along the path. Even though the path is long it may only consist of a couple of states. One example is if there is no obstacles in the work space at all, then the path found will come out as just the starting position and the goal position. However if the path is to be used for creating a trajectory we may want more points in order to set the joint velocity and acceleration more places. Therefore when a path is found it may be useful to use the function `interpolate(int states)`. If the path found has less then the number of states given by the integer sat by the user OMPL will create more states in between the states already found so we end up with more via points. `interpolate` can in some way be described as the opposite of `collapseCloseVertices`. If `collapseCloseVertices` is run after `interpolate` all the newly created states will be removed.

Chapter 4

Framework in the Open Motion Planning Library

4.1 Overview of Framework

The framework created during this project is created with the programming language C++ in the Ubuntu operating system. In Figure 4.1 it is shown what the Open Motion Planning Library, which is a big part of this framework, does for the user in this project. It is arguable that collision detection, or state validity in this case, is inside the OMPL block as the user has to create the function that checks for this, the `isValid` function, but when the user has implemented this function OMPL does the rest of the work as in checking every state created by the planner if it is a valid one or not. OMPL also has several state spaces available for the user to choose from, or the user can create its own state spaces. In this project the state space is a six dimensional `RealVectorStateSpace` since the robot has six degrees of freedom. New planners can be made by the user, but in this project only planners already available in OMPL have been used. And the last block, path processing, is a very valuable feature that is included in OMPL.

In Figure 4.2 it is shown what the inputs and outputs of the OMPL framework that is made are. For the program to find a valid path it

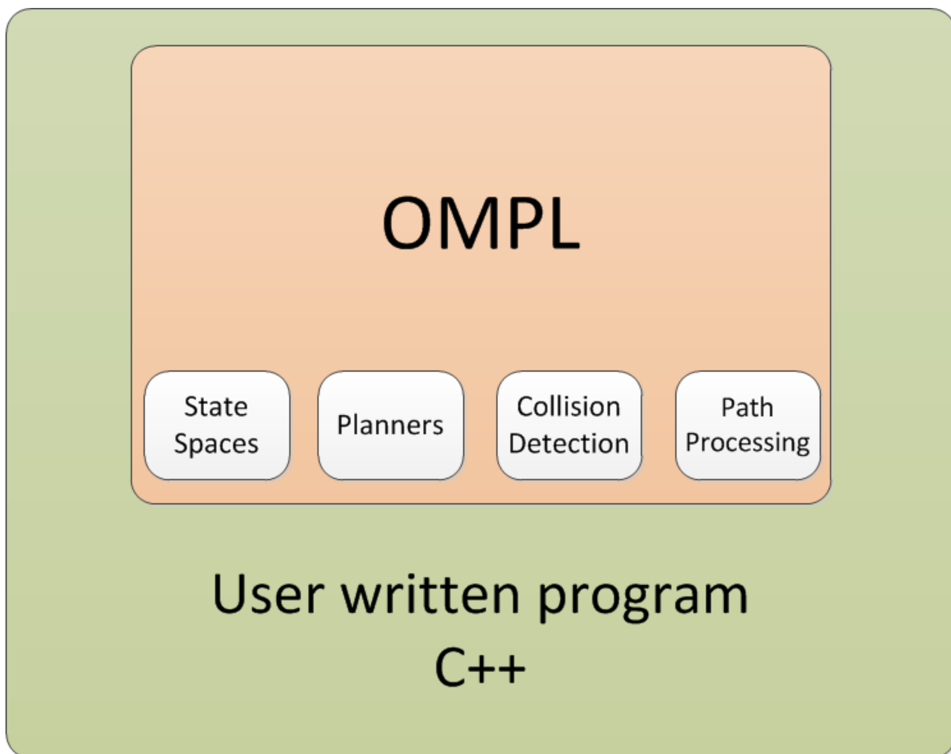

Figure 4.1: Overview of path planning framework

needs a starting position and a goal position. These positions can easily be set in the files called `start` and `goal` and has to include six floats, one for each link. The framework also needs a model of the environment which will be the work space of the robot. The environment can be set in the file called `environment` while the model for the robot is pre-made since this framework is made for use with the ABB IRB140 robot manipulator. There is one file for each of the six links on this robot. The model of the robot and the environment is then made into collision models as described in Section 3.4.2 so PQP can find out if there is a collision or not. If the environment does not change the user can simply change start and goal positions to find a valid point-to-point path. If another robot is to be used the robot files have to be exchanged with new ones, and additionally the part of the framework that calculates the forward kinematics of the robot will have to be rewritten.

The output is a valid, collision-free path with at least 20 states, or via points. It is saved in the file called `validPath` which then can be read by either a visualization program or by a program which will create the trajectory for the robot. Since the trajectory planner uses a simplified model of the robot manipulator the framework also outputs a valid path which only consists of the configurations of the first three links. This decision will be explained more in detail in Chapter 5.

4.2 Benchmarking of Planners in OMPL

The Open Motion Planning Library has a benchmarking class, `ompl::Benchmark`, which opens up for the possibility of solving a motion planning problem several times with different planners, samplers or different version of the same planning algorithm if the user wants to try and improve an existing planner. This feature may therefore show if some planners are better suited for the problem trying to be solved.

The path planning problem to be solved in this benchmark is to have the robot move from $-\pi$ to π on the first joint, a rotation of 360 degrees, in the work space. It is possible to set the number of times each planner shall run and for how long it is allowed to run. This means

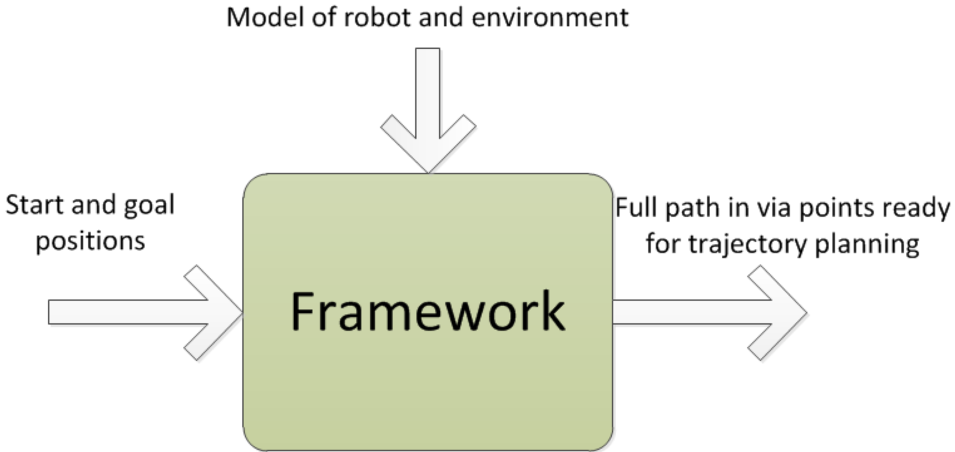

Figure 4.2: Path planning framework with inputs and outputs

that we can run each algorithm several times and then calculate for example the average runtime, or check if some planner is struggling more than the others with the problem it is trying to solve.

In Table 4.1 the planners RRT, LazyRRT, RRTConnect [8], KPIECE (Kinematic Planning by Interior-Exterior Cell Exploration)[9] and LazyK-PIECE have been used to create a benchmark for the planning problem with the ABB IRB140 in the environment shown in Figure 4.3 as the work space. In this benchmarking every planner has been run 100 times with a maximum time of 5.0 seconds. It is also possible to set the maximum amount of memory usage per computation, but in this benchmarking only time is the factor to be investigated.

Planner	RRT	LazyRRT	RRTConnect	KPIECE1	LBKPIECE1
Average runtime (s)	2.48276	1.401135	1.355275	2.24983	3.17213
Slowest runtime (s)	4.99842	4.76461	4.70642	4.99569	4.86868
Fastest runtime (s)	0.744722	0.209158	0.142457	0.187119	0.571448
Solution not found	73	26	14	30	69

Table 4.1: Result from benchmarking of planners in OMPL

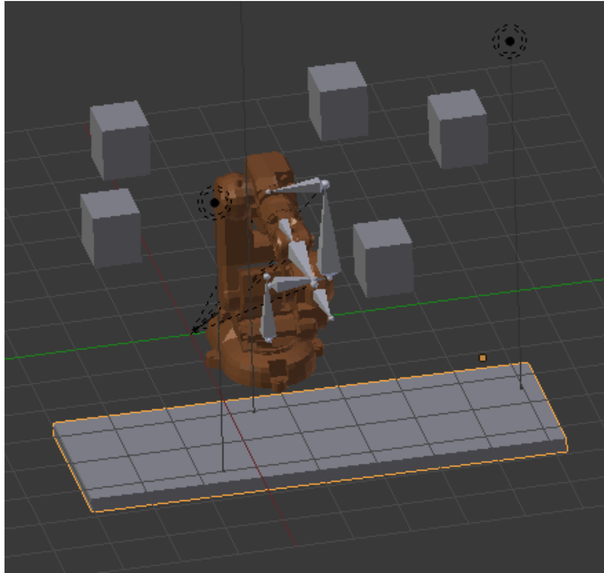

Figure 4.3: Environment used for benchmarking of planners

From Table 4.1 we see that in general the planners based on RRT have a higher solution rate than those based on KPIECE. It is clear that on this problem, during this specific benchmarking the KPIECE planners struggles to find a correct solution in the time given. If the maximum time allowed to search for a solution had been longer we might have seen a better solution rate. However it may seem as the time usage will still be worse for KPIECE planners than the time used with RRT based planners. KPIECE is specifically designed for use with physics-based simulation. The problem in this case has not taken in consideration the specific use of KPIECE planners so that may be why the results are so bad. In [10] it is shown that KPIECE can indeed produce better results than planners based on RRT if the problem is built up to take advantage of this type of planner. It should also be noted that time is the only thing that has been investigated, thus KPIECE might be better when it comes to for example memory usage, but this has not been taken into consideration in this benchmark.

We can also see from Table 4.1 that the planner with the best results over 100 runs in this case proves to be RRT-Connect. Since RRT clearly produces good result it should not come as a big surprise that

RRT-Connect will produce at least as good result, or in this case better results. RRT-Connect builds two trees rooted at the start and goal states which then explore the space around them and also advances towards each other by using a simple greedy heuristic. Since there are two trees exploring the state space at the same time it will use a shorter amount of time finding a valid path. When the two trees connect with each other the shortest path is easily computed by using the graph created. There is quite a spread in the time used for finding a solution, so if time is very important the user should investigate to see if there are areas the code can be optimized. During the work on this project the environment used in this benchmarking was the one environment the planners struggled the most to find a valid path in, so the time limit of 5 seconds may be quite strict, but it still gives a good indication on how different planners performed for this specific problem. Another environment might have given different results.

4.3 Visualization of Paths

In order to see what the path found will look like when run on a robot a way to visualize the path has been made. The environment that gets imported during the path planning phase is made in a program called Blender, which is a free, open-source software for 3D graphics. It can be used for all kinds of 3D graphics including movies, visual effects and animations. In this case it is used for making several environments a robot can possibly work in. The environment models represent for example a table with some objects around it. When a path is found the visualization of the path can also be done in Blender which is very convenient.

In Blender it is possible to attach an armature to a CAD model. It is therefore possible to download the CAD model from the ABB website for the ABB IRB140, load it into Blender and attach an armature by following a procedure made by Herman Bruyninckx [1]. Thereafter a Python script is written to be able to receive paths from an open port. This script is activated by pressing 'P' in Blender. Another script is

[1] http://people.mech.kuleuven.be/~bruyninc/hb46/blender/chapter_robotmodelling.html

then made to send a path to Blender for visualization. This script is called `runVisualization.py` and have to be run from the command line. Every 0.1 second a new via point is sent to the open port and the Blender model of the robot gets updated. The result is that the user gets to see what path the robot will take before turning this path into a trajectory. Since all planners used are sampling based most paths will be different, so this can be a good way to see if the path found from the path planning is a good one.

Several runs have been saved to show different paths found from a starting position to goal position which in this case is from $-\pi$ to π on link one, so the robot is to turn 360 in the work space. The saved runs include different environments with different planners in order to find a collision free path. In Figure 4.4 one of these runs are shown. There is no collision with the environment and no links colliding into each other so the path found is correct. A short film of several paths visualized in different environments is attached and called `visualization.wmv`.

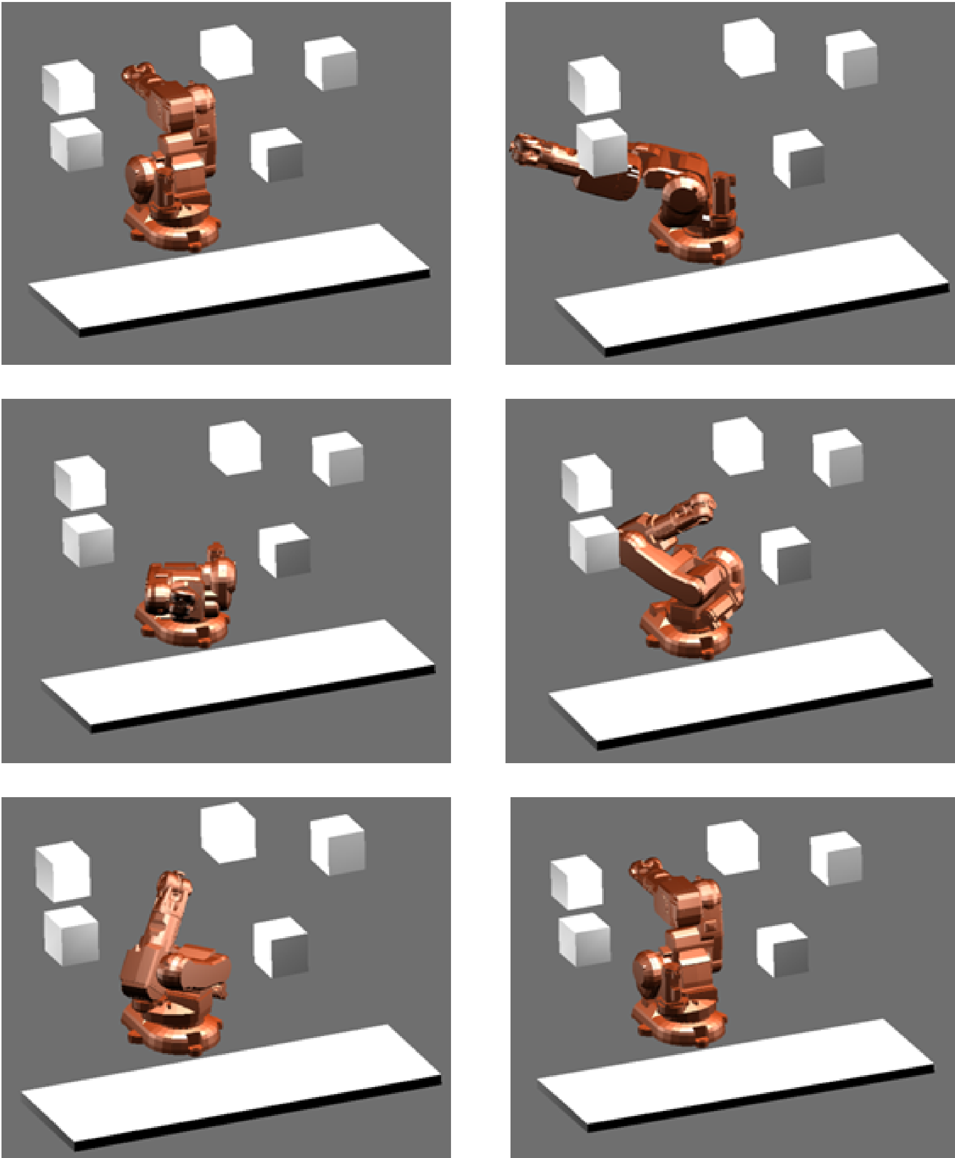

Figure 4.4: Visualization of a path in Blender

Chapter 5

Trajectory Generation for Found Path

5.1 Two Step Trajectory Planning

The problem of finding a good trajectory for a robot manipulator is not a simple task. The more degrees of freedom the robot manipulator has the more demanding and complex the problem of finding a suitable trajectory gets. This has already been mentioned in Chapter 3 and the trajectory generation has many of the same problems as the path generation since there might be infinitely many paths between two points in the configuration space. Therefore it is not straight forward to find the optimal path or trajectory for the robot.

The demands on robot manipulator are big when it for example comes to time spent doing a job or the energy efficiency of the robot while doing some work. Therefore it is generally a good idea to look for not just the first trajectory found, but to try to find the optimal trajectory. Then less time and/or energy can be used to perform the action.

One way of solving the overall trajectory generation is to divide it into two separate sub-problems as done in this project. The first step is to generate a desired path and the second step is to turn this path into a trajectory by assigning velocities along the path. It is possible to do

both steps in one calculation, but by dividing it into two sub-problems the multi-dimensional state space can be reduced to a two-dimensional state space with only the the path coordinate, s, and the derivative of the path coordinate, $\dot{s}$.

When a path, q, is found it is then parametrized as $q(s) = [q_1(s)\ q_2(s)\ \dots\ q_n(s)]^T$ where s is the path coordinate that goes from 0 to 1. The velocity and the acceleration can then be parametrized in the same way to produce $\dot{q}(s) = q'(s)\dot{s} = [q_1'(s)\ q_2'(s)\ \dots\ q_m'(s)]^T\dot{s}$ and $\ddot{q}(s) = q''(s)s^2 + q'(s)\ddot{s}$.

The equation for a robots dynamics is:

$$\sum_{j=1}^{n} m_{ij}(q)\ddot{q}_j + \sum_{j=1}^{n}\sum_{j=1}^{n} c_{ijk}(q)\dot{q}_j\dot{q}_k + h_i(q) = \tau_i \tag{5.1}$$

By using the parametrization above along a found path in the equation for the robot dynamics we end up with

$$\begin{aligned} &\sum_{j=1}^{n} m_{ij}(q(s))q_j'(s)\ddot{s} + g_i(q(s)) \\ &+ \left(\sum_{j=1}^{n} m_{ij}(q(s))q_j''(s) + \sum_{j=1}^{n}\sum_{j=1}^{n} c_{ijk}(q(s))q_j'(s)q_k'(s) \right)\dot{s}^2 = \tau_i \end{aligned} \tag{5.2}$$

In equation 5.2 the dynamics are now on the form

$$\alpha_i(s)\ddot{s} + \beta_i(s)\dot{s}^2 + \gamma_i(s) = \tau_i \tag{5.3}$$

where

$$\alpha_i(s) = \sum_{j=1}^{n} m_{ij}(q(s))q'_j(s)\ddot{s} \tag{5.4}$$

$$\beta_i(s) = \sum_{j=1}^{n} m_{ij}(q(s))q''_j(s) + \sum_{j=1}^{n}\sum_{j=1}^{n} c_{ijk}(q(s))q'_j(s)q'_k(s) \tag{5.5}$$

$$\gamma = g_i(q(s)) \tag{5.6}$$

5.2 Second-Order Cone Programming

In [11] a procedure for optimization is proposed by the use of Second-order cone programming. A second-order cone program is a convex optimization problem on the form

$$\min f^T x \tag{5.7}$$

$$\text{subject to } Fx = g \tag{5.8}$$

$$\|M_j x + n_j\| \leq p_j^T x + q_j \tag{5.9}$$

$$\text{for } j = 1 \ldots m \tag{5.10}$$

A cone constraint used in this method of optimization is on the form

$$\|Ax + b\|_2 \leq c^T x + d \tag{5.11}$$

By constraining the decision variables to lie within such a cone more efficient solvers can be used to solve the optimization problem. As shown in Figure 5.1 the transformation $Ax + b$ transforms a point x in the cone to the cone with its top in the origin. $c^T x + d$ determines the width of the cone and the point x is projected onto the stapled ray given by $c^T x + d$.

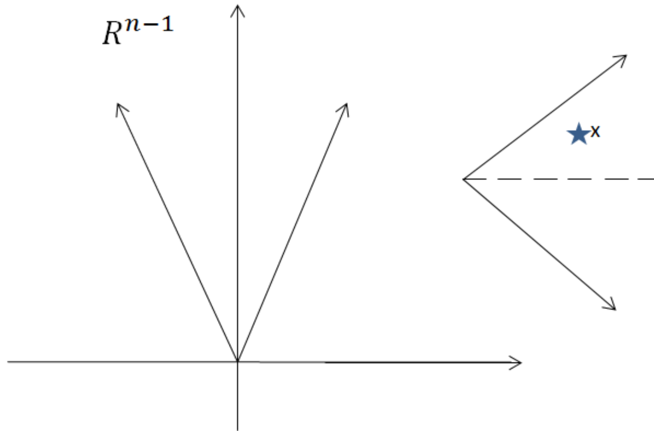

Figure 5.1: Cone Constraint visualization

5.3 Optimization by Second-Order Cone Programming

5.3.1 Object function

When a path is found and parameterized as a function of s, it is possible to calculate the time it will take to execute a motion along this path given a velocity assignment $\dot{s}(s)$. By observing that $dt = \frac{dt}{ds}ds = \frac{1}{\dot{s}(s)}ds$ it is then possible to formulate integrals of how long time the motion takes.

$$T = \int_0^T dt = \int_0^1 \frac{1}{\dot{s}(s)} \mathrm{d}s \tag{5.12}$$

This can then be used to optimize the time which the robot manipulator will use on the motion along the path q that is already found. If the user wants to optimize the energy usage of the robot manipulator

the following energy integral can be used.

$$E_i = \int_0^1 \frac{\tau_i^2(s)}{\dot{s}(s)} \mathrm{d}s \tag{5.13}$$

By combining these two integrals it is possible to find a function that measures a combination of time and energy. To be able to decide what we want to weight the most, speed or energy consumption, a parameter, γ, can be used in front of the energy integral.

$$J = T + \gamma \sum_{i=1}^{n} E_i \tag{5.14}$$

By representing $\dot{s}$ as a function of s the velocity of the system can be specified explicitly. The bounds for each joint are usually given in datasheets and this is also the case for the ABB IRB140 [12].

$$\|\dot{q}(s)\| \leq \|q_i'(s)\dot{s}\| \leq \bar{\dot{q}} \tag{5.15}$$

By using this and that $\bar{\dot{q}}$ is the maximum speed of joint i the maximum value of $\dot{s}$ along a path can be found as

$$max\|\dot{s}\| = \min_i \frac{\bar{\dot{q}}}{\|q_i'(s)\|} \tag{5.16}$$

5.3.2 Reformulation from non-linear optimization problem to second-order cone program

One of the big advantages of using second-order cone programming is that there exists dedicated solvers for these types of problems which are more efficient then trying to solve the optimization problem with any nonlinear solver. To be able to use these types of solvers the problem has to go through a not so straightforward reformulation to get the optimization problem on the form of a second-order cone programming problem.

The method used to reformulate the original optimization problem from a non-linear optimization problem into a convex second-order cone program below is found in [11]. The original time-optimal path tracking problem subject to lower and upper bounds on the torques can be expressed as

$$\min_{T,s(\cdot),\tau(\cdot)} T \tag{5.17}$$

$$\text{subject to} \tag{5.18}$$

$$s(0) = 0, \tag{5.19}$$

$$s(T) = 1, \tag{5.20}$$

$$\dot{s}(0) = \dot{s}_0, \tag{5.21}$$

$$\dot{s}(T) = \dot{s}_T \tag{5.22}$$

$$\dot{s}(t) \geq 0, \tag{5.23}$$

$$\underline{\tau}(s(t)) \leq \tau(t) \leq \bar{\tau}(s(t)) \text{ for } t \in [0, T] \tag{5.24}$$

where $s(0)$, $s(T)$, $\dot{s}(0)$ and $\dot{s}(T)$ is the initial and ending position and velocity and the bounds on the torque may depend on s.

It is then possible to rewrite 5.17 - 5.24 in order to make the entire problem convex. A convex optimization problem is desirable because we know that if we find a solution to the convex problem the solution we find is a global solution. The first step is to use equation 5.12 and 5.13 to make equation 5.14 and use this as our objective function. This objective function is then discretized to create

$$J = \int_0^1 \frac{1 + \gamma \sum_{i=1}^{n} \tau_i(s)^2}{\dot{s}} ds \approx \sum_{k=0}^{K-1} \left[1 + \gamma \sum_{i=1}^{n} (\tau_i^k)^2 \right] \int_{s^k}^{s^{k+1}} \frac{1}{\sqrt{b(s)}} ds \tag{5.25}$$

where K is the number of discretization points. The last integral can then be rewritten by using the parametrization of the squared velocity

$$\dot{s}(s)^2 = b(s) = b^k + \left(\frac{b^{k+1} - b^k}{s^{k+1} - s^k} \right) (s - s^k) \tag{5.26}$$

which is piecewise linear and where b^k is the value of $b(s)$ at s^k, into

$$J = \sum_{k=0}^{K-1} \frac{2\Delta s^k (1 + \gamma \sum_{i=1}^{n} (\tau_i^k)^2)}{\sqrt{b^{k+1}} + \sqrt{b^k}} \tag{5.27}$$

This nonlinear objective function can then be converted into a linear objective function and a hyperbolic constraint by making to substitutions. We introduce the variables c^k and d^k so the objective function becomes linear

$$\sum_{k=0}^{K-1} 2\Delta s^k d^k \tag{5.28}$$

with the two hyperbolic constraints

$$\frac{1 + \gamma \sum_{i=1}^{n} (\tau_i^k)^2}{c^{k+1+c^k}} \le d^k \tag{5.29}$$

$$c^k \le \sqrt{b^k} \tag{5.30}$$

It is then possible to rewrite these two hyperbolic constraints to convert them into a second-order constraint. By using

$$w^2 \le xy,\ x \ge 0,\ y \ge 0 \Leftrightarrow \left\| \begin{bmatrix} 2w \\ x - y \end{bmatrix} \right\| \le x + y \tag{5.31}$$

we get the second-order constraints

$$\left\| \begin{matrix} 2 \\ 2\sqrt{\gamma}\tau_1^k/\bar{\tau}_1 \\ \dots \\ 2\sqrt{\gamma}\tau_n^k/\bar{\tau}_n \\ c^{k+1} + c^k - d^k \end{matrix} \right\| \le c^{k+1} + c^k - d^k \tag{5.32}$$

$$\text{for } k = 0 \dots K - 1 \tag{5.33}$$

and

$$\left\| \begin{matrix} 2c^k \\ b^k - 1 \end{matrix} \right\| \leq b^k + 1 \tag{5.34}$$

$$\text{for } k = 0 \ldots K \tag{5.35}$$

5.4 Solving the Optimization Problem Using SeDuMi in Matlab

When we have the final form of the optimization problem we can use a robust numerical algorithm to solve the second-order cone program. The best solver would have been one based on C++ in order to solve the path planning and trajectory generation in the same program, unfortunately no such good solver is available for free. Because there was not enough time to create a second-order cone program solver in C++ in this project, a solver that runs in Matlab has been used. There are several freely available solvers to do this. In this project the solver chosen is called SeDuMi, which stands for Self-Dual-Minimization, and runs under Matlab on most operating systems. SeDuMi contains functions for solving many types of optimization programs, but the one we will be using to solve our second-order cone program is `sedumi(A, b, c, K)` which tries to solve the optimization problem

$$\begin{aligned} &\min_x c^T x \\ &\text{subject to } Ax = b \\ &x \in K \end{aligned}$$

The A, b and c in `sedumi(A, b, c, K)` is the matrix A and the vectors b and c in the optimization problem. The K is used to define the cone constraints. The three most important properties in K are $K.f$ which is the number of free components, $K.l$ which is the number of non-negative components and $K.q$ which is needed to define the dimensions of the quadratic cones.

Because the dynamics of the ABB IRB140 is not easily available this project has made some simplifications when it comes to the calculation of the optimal trajectory. Only the three first joints are being investigated as they are the ones that do most of the movement in the work space. The three last joints are used for for example rotating a gripper, and does not cover a big area in the work space. The only reason there is movement on the three last links in this project when a path is found, is because of the use of sampling-based planners. This makes it so the configurations the path is made out of often includes movement in all joints. By only looking at the three first joints, and setting no movement on the three last joints, we look at the part that is the most important when trying to avoid collisions.

5.5 Finding the Time-Optimal Trajectory

5.5.1 Illustration of Obstacle-free point-to-point motion

The optimal-time trajectory program first reads a path found from Chapter 4. This path includes only the configuration of the first three links since this was part of the simplification made in order to be able to calculate an optimal trajectory. The program then evaluates $q(s)$, $q'(s)$ and $q''(s)$ at discrete points along the path so that the dynamics can be evaluated along the path. This creates $\alpha_i(s)$, $\beta_i(s)$ and $\gamma_i(s)$. It should be noted that α_i, β_i and γ_i is not totally correct since the real dynamics of the ABB IRB140 are unknown. This is an estimate by using some parameters found in the ABB manual for the robot manipulator. The last thing that is done is to find the matrices needed by SeDuMi to solve the optimization problem by using second-order cone programming. Figure 5.2 shows what the finished framework looks when we add the trajectory planning to the path planning.

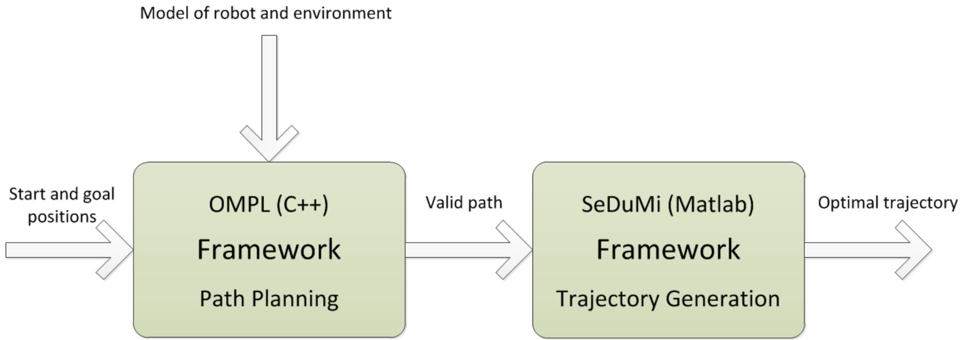

Figure 5.2: Full framework consists of both path planning and trajectory planning.

In figure 5.3 the visualization of a path in an environment without obstacles, except for a table is shown. The start position is set to $-\pi$ and the goal position is π, on the first joint, that so the robot manipulator is to turn 360 degrees. Figure 5.4 - 5.7 shows the joint angles, joint velocities, join accelerations and the joint torques for this path. Figure 5.8 shows the path coordinate s with respect to time while Figure 5.9 shows the relationship between the path coordinate s and the path velocity $\dot{s}$. When observing the results it is noteworthy that the torque is always maxed out on at least one joint, in this case joint one since that is the only one that has to do any movement. For a very short period of time no torque is maxed, when the torque on joint one goes from maximum to minimum, but this is simply because the joint does not change infinitely fast. If there was a period of time where none of the joint where saturated the trajectory found would not have been optimal as it would have been possible to apply more torque to a joint and thereby make the movement go faster. It is also possible to see that the acceleration of the joint behaves quite similar to the torque on the same joint.

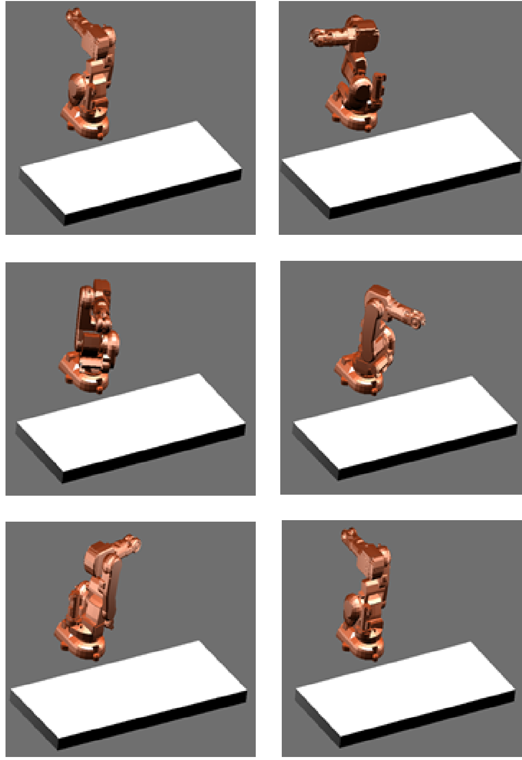

Figure 5.3: Visualization of the path used to find a trajectory in a obstacle free environment.

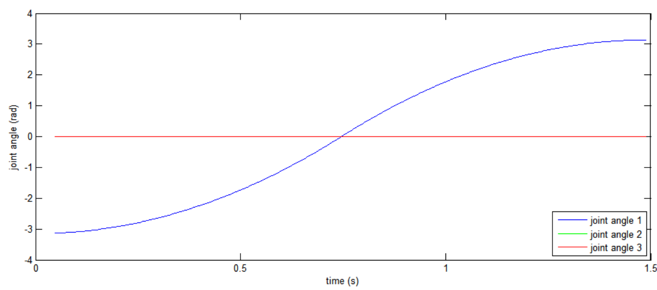

Figure 5.4: Joint angles on a path in obstacle-free environment.

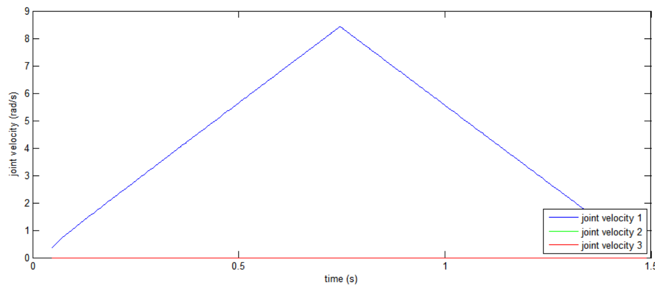

Figure 5.5: Joint velocities on a path in obstacle-free environment.

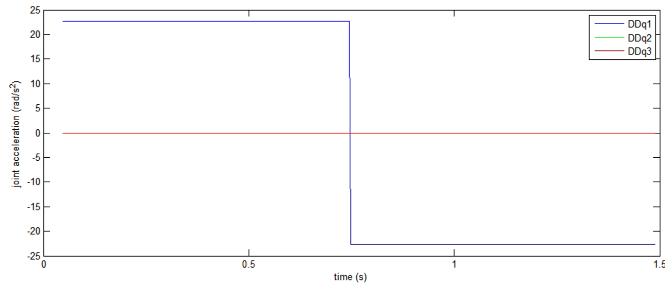

Figure 5.6: Joint accelerations on a path in obstacle-free environment.

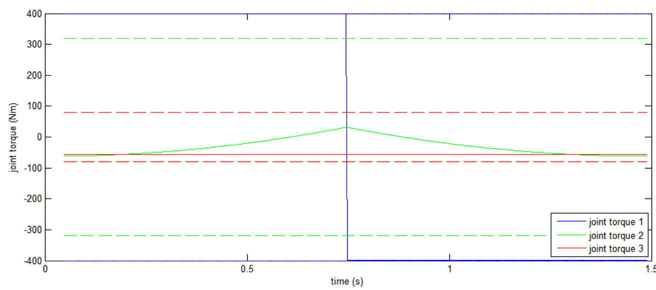

Figure 5.7: Joint torques on a path in obstacle-free environment. The dotted lines are torque constraints.

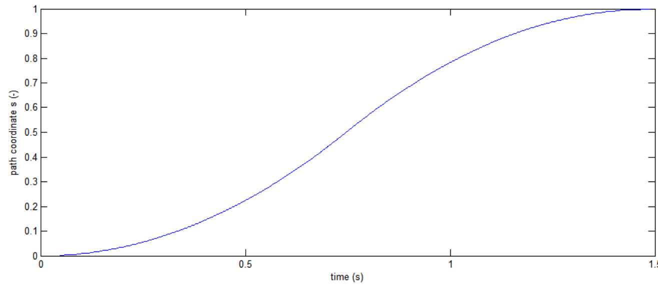

Figure 5.8: Path coordinate with respect to time on path in obstacle-free environment.

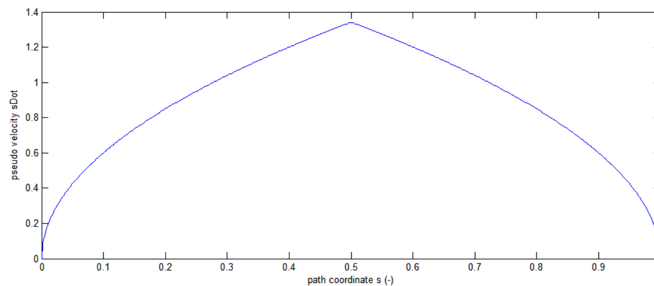

Figure 5.9: Velocity profile for a path in an obstacle-free environment.

5.5.2 Illustration of obstacle-"clouded" point-to-point motion

When the trajectory is calculated for a path where the robot is working within an environment, the result gets more complicated because there is more movement in the joints as they can not go straight from their starting position to their goal position. The start and goal position is the same as in the illustration with a obstacle-free environment, starting position is $-\pi$ and the goal position is π on joint one to turn the manipulator 360 degrees in the work space.

Figure 5.10 shows the visualization of a path in an environment with more objects than just the table, which was in the environment in Section 5.5.1. Figure 5.11 - 5.14 shows joint angles, joint velocity, join

acceleration and joint torque for this path. Figure 5.15 shows the path coordinate with respect to time and Figure 5.16 shows the relationship between the path coordinate s and the path velocity $\dot{s}$. We can see that one of the joint torques in this case also is always in saturation so a time-optimal trajectory has been found. By looking closer at parts of the paths we can also see that the behaviour of acceleration and torque is quite similar in this trajectory as well.

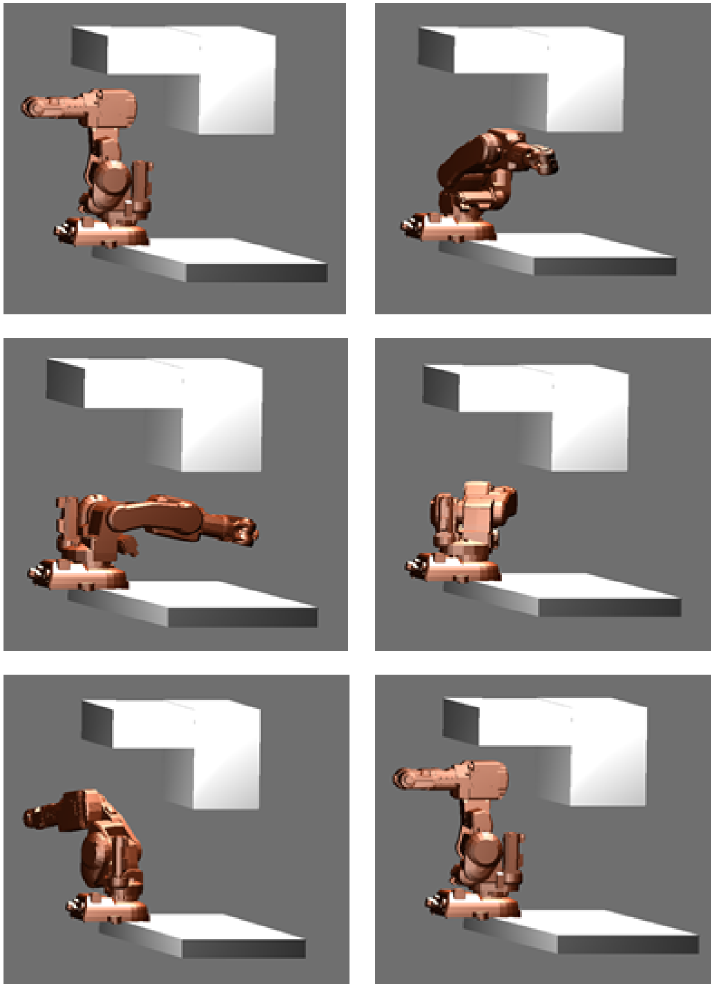

Figure 5.10: Visualization of the path used to find a trajectory in an obstacle-clouded environment.

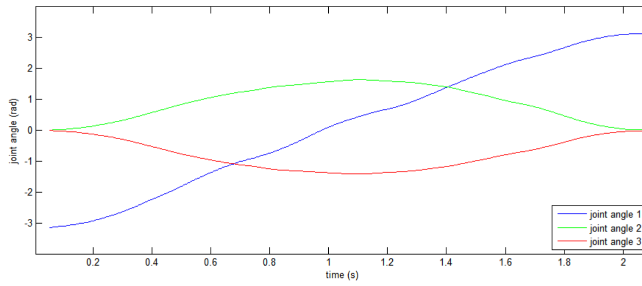

Figure 5.11: Joint angles on a path in an obstacle-clouded environment.

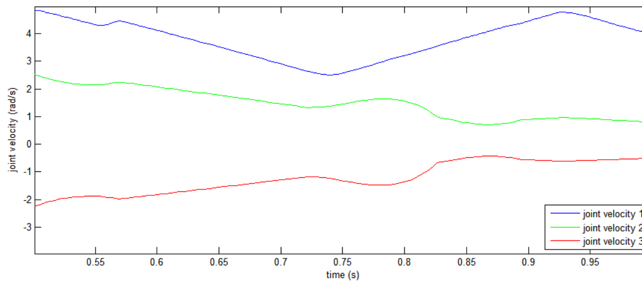

Figure 5.12: Joint velocities on a path in an obstacle-clouded environment.

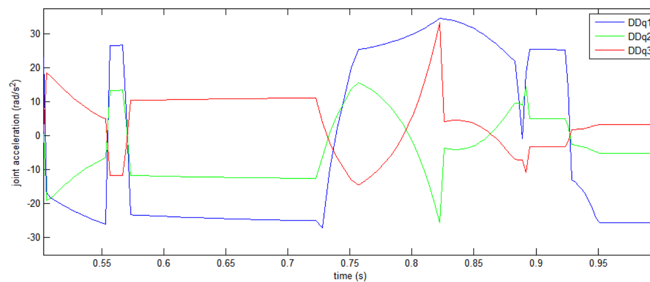

Figure 5.13: Joint accelerations on path in an obstacle-clouded environment.

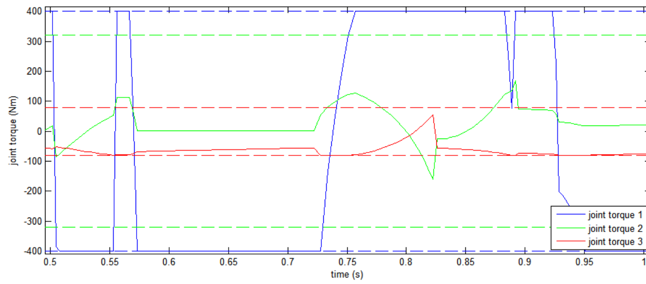

Figure 5.14: Joint torques on a path in an obstacle-clouded environment. The dotted lines are torque constraints.

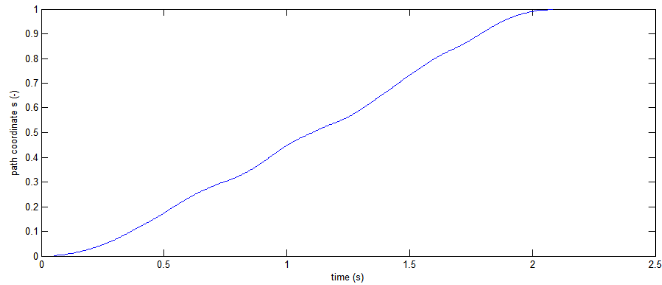

Figure 5.15: Path coordinate with respect to time on a path in an obstacle-clouded environment.

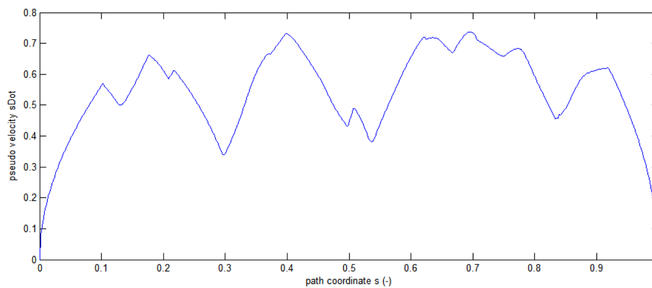

Figure 5.16: Velocity profile for a path in an obstacle-clouded environment.

By looking at both trajectories found it is possible to see that the result from the trajectory planning seems correct for both paths. The

time used for the robot manipulator to go from the start position to the end position in an obstacle-free environment is shorter than in an environment with obstacles. This makes sense since in the environment with obstacles the robot manipulator has to go around the objects and therefore there is more movement on several joints so the overall torque has to be distributed amongst them. This leads to the robot manipulator using longer time on the entire movement.

By looking at Figures 5.16 and 5.9 we can see the maximum velocity profile for both paths. There can be infinitely many trajectories that are slower than the one found, but no trajectory will be able to find a path velocity that is higher than the graphs shown. We can see that the maximum path velocity is higher for the obstacle-free trajectory and also a lot smoother because there are less switches between the active torque constraints. Since we in both cases can observe that the torque is saturated in at least one joint at every point in time, it is advisable to not run the robot manipulator at maximum speed. If this is done the control authority to correct things such as disturbances, is used up. It is possible, and should be done in most cases, to scale down the velocity profile by some percent depending on how critical the user wants to run the system.

Chapter 6

Discussion and Future Work

In Chapter 3 it was investigated how path planning for an industrial robot manipulator can be done. Several different methods to find a valid path was discussed with most focus on sampling-based planners, as all planners in the Open Motion Planning Library are sampling-based. After a quick introduction to OMPL it was shown how path planning is done by using this relatively new path planning tool. One of the most important things about the use of OMPL is how the state validation or collision detection is done. In this project state validity was done by implementing a state validation class. Additionally PQP was used to detect self collisions when two or more links on the robot manipulator collides with each other, or when a collision with environment was detected.

A framework for path planning was created in Chapter 4. What the framework needs and what it does for the user is shown in Figures 4.1 and 4.2. This framework is made so the user has to use few task primitives in order to be able to find a valid path. Given a robot model and an environment model the only thing the user has to tell the program is a start position and a goal position. It is easy to create new environments in for example Blender. The only thing the user has to do to find a path in this new environment is to export this program as a .RAW file and update the file that tells the program where it should read the environment from. The output of this path planning framework is a valid path with several via points that can easily be

used for trajectory planning. A visualization of the path found can be run in Blender. Several runs with different environments and planners have been saved and turned into a short film attached to this project in order to be able to show correct behaviour from the framework.

The last part of Chapter 4 shows a benchmarking of some of the planners available for the user in OMPL. For the problem that was needed to be solved during this benchmark it was clear that the planners based on Rapidly-exploring Random Trees, RRT, where a lot faster and more accurate then the planners based om Kinematic Planning by Interior-Exterior Cell Exploration, KPIECE. RRT is a fast algorithm that quickly covers big parts of the work space and therefore produces good results in this case. KPIECE is still a good algorithm, but it excels in results when the problem has taken into consideration what planners are to be used to solve it. In [10] it is shown an example of a problem where KPIECE produces better results than RRT.

Chapter 5 described a method for solving the problem of finding the optimal trajectory of one of the paths found in Chapter 4. This method used second-order cone programming, SOCP, to solve the optimization problem. To be able to use this method the original objection function and its constraints had to be rewritten quite a bit to create a second-order cone program. When the correct form is found it is possible to use mature solvers which exploit the SOCP structure very efficiently.

It was then shown how this optimization method can be used on a path found by using SeDuMi, which is an efficient solver for solving SOCP in Matlab. When finding the optimal trajectory a simplification of the dynamics of the ABB IRB140 had to be used since the real dynamics are not easily available. The robot manipulator was therefore modeled as a 3-link robot arm when finding the trajectory. An illustration of a trajectory found for one path in an obstacle-free environment as well as a path in an obstacle-clouded environment was shown. When the optimal trajectory was found it was possible to see that one of the torque constraints where always in saturation. If this had not been the case it would have been possible to get a faster trajectory by increasing the torque on one of the joints. It was also observed that the torque and acceleration graphs behave quite similar. Figures 5.9 and 5.16 show that the velocity profile is higher for the trajectory made in

the collision-free environment. This is as expected since there is less movement on the robot and its joints so this is also a good indicator that the trajectory calculation works as it should. It is also important to observe that at least one of the joint torques are saturated at every point in time. Because of this it is advisable to not run the robot manipulator at its maximum speed. If this is done the control authority to correct things such as disturbances is used up. It is possible to scale down the velocity profile by some percent depending on how critical the user wants to run the system, which should be done in most cases.

The framework created in this project is only an example of how a framework can be created for path planning and trajectory generation. It is made to be simple to use, and needs few task primitives from a user in order to be able to find a path and its trajectory. It would have been nice to be able to do both these steps by the use of C++ so there would only be one file to run to find both the path and trajectory. To be able to do this, the need of a solver for a second-order cone program is needed in C++. It should also be noted that the framework is specially designed for the ABB IRB140. It is easy to change environment, but if the user wants to use another robot manipulator most of the framework can be used, but the calculation of the forward kinematics have to be exchanged with new Denavit-Hartenberg parameters as well as new collision models for the new robot.

After using OMPL during this project it seems to have a bright future with its user-friendliness for anyone with a programming background. Additionally its continuous updates will make it even better in the future. The Open Motion Planning Library is already under consideration to be used for path planning problems on robotic systems used by SINTEF.

The trajectories found in Chapter 5 have unfortunately not been tested on a real robot manipulator. For this to be done it is most likely necessary to calculate the real dynamics of the ABB IRB140 and use them to create a trajectory for the full robot with six links instead of the simplifications used in this project.

Appendix A

A.1 Simple Examples of Path Planning With and Without `SimpleSetup`

Algorithm 3 Path Planning With the Use of `SimpleSetup`

```
namespace ob = ompl::base;
namespace og = ompl::geometric;

bool isStateValid(const ob::State *state)

void planWithSimpleSetup(void)
{
    ob::StateSpacePtr space(new ob::SE3StateSpace());

    ob::RealVectorBounds bounds(3);
    bounds.setLow(-1);
    bounds.setHigh(1);

    space->as<ob::SE3StateSpace>()->setBounds(bounds);

    og::SimpleSetup ss(space);

    ss.setStateValidityChecker(boost::bind(&isStateValid, _1));

    ob::ScopedState<> start(space);
    start.random();
    ob::ScopedState<> goal(space);
    goal.random();

    ss.setStartAndGoalStates(start, goal);

    bool solved = ss.solve(1.0);

    if (solved)
    {
        std::cout << "Found solution:" << std::endl;
        ss.simplifySolution();
        ss.getSolutionPath().print(std::cout);
    }
}
```

Algorithm 4 Path Planning Without the Use of `SimpleSetup`

```
namespace ob = ompl::base;
namespace og = ompl::geometric;

bool isStateValid(const ob::State *state)
void plan(void)
{
    ob::StateSpacePtr space(new ob::SE3StateSpace());

    ob::RealVectorBounds bounds(3);
    bounds.setLow(-1);
    bounds.setHigh(1);

    space->as<ob::SE3StateSpace>()->setBounds(bounds);

    ob::SpaceInformationPtr si(new ob::SpaceInformation(space));

    si->setStateValidityChecker(boost::bind(&isStateValid, _1));

    ob::ScopedState<> start(space);
    start.random();
    ob::ScopedState<> goal(space);
    goal.random();

    ob::ProblemDefinitionPtr pdef(new ob::ProblemDefinition(si));

    pdef->setStartAndGoalStates(start, goal);

    ob::PlannerPtr planner(new og::RRTConnect(si));
    planner->setProblemDefinition(pdef);
    planner->setup();

    bool solved = planner->solve(1.0);

    if (solved)
    {
        ob::PathPtr path = pdef->getGoal()->getSolutionPath();
        std::cout << "Found solution:" << std::endl;
        // print the path to screen
        path->print(std::cout);
    }
}
```

Appendix B

B.1 An example of a .RAW file for an environment

Each line has nine coordinates to describe three different points in the space with its x, y and z value. Every line therefore describes one triangle to be used when making a collision model. A box has six faces in the shape of a square. Two triangles makes one square so for a box a model file will have $2 * 6 = 12$ lines, one for each triangle. The file below is for the environment file `airbox` which has three objects, so the file consists of 36 lines, 12 for each object.

3.713757 2.738049 0.414542 3.713757 2.424479 0.414542 1.713757 2.424480 0.414542

1.713757 2.424480 0.414542 1.713758 2.738050 0.414542 3.713757 2.738049 0.414542

3.713758 2.738049 5.324363 1.713757 2.738050 5.324363 1.713757 2.424480 5.324363

1.713757 2.424480 5.324363 3.713757 2.424479 5.324363 3.713758 2.738049 5.324363

3.713757 2.738049 0.414542 3.713758 2.738049 5.324363 3.713757 2.424479 5.324363

3.713757 2.424479 5.324363 3.713757 2.424479 0.414542 3.713757 2.738049 0.414542

3.713757 2.424479 0.414542 3.713757 2.424479 5.324363 1.713757 2.424480 5.324363

1.713757 2.424480 5.324363 1.713757 2.424480 0.414542 3.713757 2.424479 0.414542

1.713757 2.424480 0.414542 1.713757 2.424480 5.324363 1.713757 2.738050 5.324363

1.713757 2.738050 5.324363 1.713758 2.738050 0.414542 1.713757 2.424480 0.414542

3.713758 2.738049 5.324363 3.713757 2.738049 0.414542 1.713758 2.738050 0.414542

1.713758 2.738050 0.414542 1.713757 2.738050 5.324363 3.713758 2.738049 5.324363

4.819256 -0.468421 1.639082 4.819256 -1.664419 1.639082 3.803806 -1.664419 1.639082

3.803806 -1.664419 1.639082 3.803807 -0.468420 1.639082 4.819256 -0.468421 1.639082

4.819257 -0.468421 2.890932 3.803807 -0.468421 2.890932 3.803806 -1.664418 2.890932

3.803806 -1.664418 2.890932 4.819255 -1.664419 2.890932 4.819257 -0.468421 2.890932

4.819256 -0.468421 1.639082 4.819257 -0.468421 2.890932 4.819255 -1.664419 2.890932

4.819255 -1.664419 2.890932 4.819256 -1.664419 1.639082 4.819256 -0.468421 1.639082

4.819256 -1.664419 1.639082 4.819255 -1.664419 2.890932 3.803806 -1.664418 2.890932

3.803806 -1.664418 2.890932 3.803806 -1.664419 1.639082 4.819256 -1.664419 1.639082

3.803806 -1.664419 1.639082 3.803806 -1.664418 2.890932 3.803807 -0.468421 2.890932

3.803807 -0.468421 2.890932 3.803807 -0.468420 1.639082 3.803806 -1.664419 1.639082

4.819257 -0.468421 2.890932 4.819256 -0.468421 1.639082 3.803807 -0.468420 1.639082

3.803807 -0.468420 1.639082 3.803807 -0.468421 2.890932 4.819257 -0.468421 2.890932

6.177412 5.409776 -0.411176 6.177412 -4.093324 -0.411176 3.105231 -4.093324 -0.411176

3.105231 -4.093324 -0.411176 3.105231 5.409777 -0.411176 6.177412 5.409776 -0.411176

6.177413 5.409776 -0.156034 3.105231 5.409776 -0.156034 3.105230 -4.093323 -0.156034

3.105230 -4.093323 -0.156034 6.177411 -4.093325 -0.156034 6.177413 5.409776 -0.156034

6.177412 5.409776 -0.411176 6.177413 5.409776 -0.156034 6.177411 -4.093325 -0.156034

6.177411 -4.093325 -0.156034 6.177412 -4.093324 -0.411176 6.177412 5.409776 -0.411176

6.177412 -4.093324 -0.411176 6.177411 -4.093325 -0.156034 3.105230 -4.093323 -0.156034

3.105230 -4.093323 -0.156034 3.105231 -4.093324 -0.411176 6.177412 -4.093324 -0.411176

3.105231 -4.093324 -0.411176 3.105230 -4.093323 -0.156034 3.105231 5.409776 -0.156034

3.105231 5.409776 -0.156034 3.105231 5.409777 -0.411176 3.105231 -4.093324 -0.411176

6.177413 5.409776 -0.156034 6.177412 5.409776 -0.411176 3.105231 5.409777 -0.411176

3.105231 5.409777 -0.411176 3.105231 5.409776 -0.156034 6.177413 5.409776 -0.156034

Bibliography

[1] Homepage of the open motion planning library. [Online]. Available: http://ompl.kavrakilab.org/

[2] M. W. Spong, S. Hutchinson, and M.Vidyasagar, *Robot Modeling and Control*. John Wiley Sons, Inc, 2006.

[3] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006, available at http://planning.cs.uiuc.edu/.

[4] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. on Robotics and Automation*, 1996.

[5] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," *Department of Computer Science, Iowa State University*, Oct. 1998.

[6] Homepage of the open motion planning library. [Online]. Available: http://kavrakilab.org/

[7] The proximity query package. [Online]. Available: http://gamma.cs.unc.edu/SSV/

[8] J. James J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," *IEE Int'l Conf. on Robotics and Automation*, 2000.

[9] I. A. Sucan and L. E. Kavraki, "Kinodynamic motion planning by interior-exterior cell exploration," in *Algorithmic Foundation*

of Robotics VIII (Proceedings of Workshop on the Algorithmic Foundations of Robotics), vol. 57, STAR. Guanajuato, Mexico: STAR, 2009, pp. 449–464. [Online]. Available: http://www.springerlink.com/content/gm47pt40p0740125/

[10] ——, "On the performance of random linear projections for sampling-based motion planning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, St. Louis, 11/10/2009 2009, pp. 2434–2439. [Online]. Available: http://dx.doi.org/10.1109/IROS.2009.5354403

[11] D. Verscheure, B. B. Demeulenaere, J. Swevers, J. D. Schutter, and M. Diehl, "Time-energy optimal path tracking for robots: a numerically efficient optimization approach," *Automatic Control, IEEE Transactions*, 2009.

[12] Irb140 industrial robot datasheet. [Online]. Available: http://www05.abb.com/global/scot/scot241.nsf/veritydisplay/d7dfcc72e3fd760dc12579c7002ce1e0/$file/PR10031EN%20R14%20LR.pdf