Gezamenlijke opleiding Universiteit Hasselt en KU Leuven

2016•2017
FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
master in de industriële wetenschappen: energie

Masterproef

Collision-free trajectory generation for welding robots: analysis and improvement of the Descartes algorithm

Promotor :
Prof. dr. ir. Eric DEMEESTER

Copromotor :
De heer Maarten VERHEYEN

Bart Moyaers , Giel Vanvelk

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: energie

2016•2017
Faculteit Industriële ingenieurswetenschappen
master in de industriële wetenschappen: energie

Masterproef

Collision-free trajectory generation for welding robots: analysis and improvement of the Descartes algorithm

Promotor :
Prof. dr. ir. Eric DEMEESTER

Copromotor :
De heer Maarten VERHEYEN

Bart Moyaers , Giel Vanvelk
Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: energie

Acknowledgments

We, Bart Moyaers and Giel Vanvelk, opted for this master thesis subject because we where both interested in the application of robot welding. On the one hand, because it is a prime example of automation, which is the specialization we chose in our engineering study. On the other hand, we were attracted to the possibilities of automating industrial processes, using robots.

The master thesis required knowledge which we didn't have at the start of the project. Neither of us had any prior knowledge of used software like Ubuntu, ROS, Descartes, etc. We did have a basic programming knowledge, but not in C++. This made the start a bit rough, but ensured that we learned a lot by the end of the master thesis.

We would like to thank our promoters, prof. dr. ing. Eric Demeester and ing. Maarten Verheyen for the help they have given during the master thesis. They followed the project up close and helped with decision making. A special thanks to Eric Demeester, for the enlightening conversations we had, and to Maarten Verheyen for providing the robot model, which really gave us a kick start. We would also like to thank Ir. Jeroen De Maeyer, whose doctoral research has a similar context. Your insights, ideas, and suggestions were very useful, enlightening and, not less importantly, motivating.

The code used in this thesis is available online in the following repository:
https://github.com/Bart123456/lasrobot_ws
For GIT:
https://github.com/Bart123456/lasrobot_ws.git

Our code has also been used in a paper[33] about the Descartes software package.

Contents

List of Figures 5

List of Tables 7

Abstract in English 9

Abstract in Dutch 9

1 Introduction 13
1.1 Background . . . 13
1.2 Problem definition . . . 13
1.3 Objectives . . . 14
1.4 Methods and materials . . . 14
1.5 Summary . . . 15

2 Literature study 17
2.1 Utilized software . . . 17
2.1.1 ROS . . . 17
2.1.2 Trajectory generation software . . . 20
2.2 Research trajectory generation software . . . 22
2.2.1 Descartes Path Planner (ROS-I) . . . 22
2.2.2 CHOMP . . . 25
2.2.3 TrajOpt . . . 27
2.2.4 Comparison . . . 28
2.3 Chapter summary . . . 29

3 Set-up and installing the Descartes-software package 31
3.1 Workspace . . . 31
3.2 Installing the Descartes-software package . . . 32
3.3 Robot modelling files . . . 34
3.4 Chapter summary . . . 34

4 Robot modelling 35
4.1 URDF . . . 35
4.1.1 Links . . . 35
4.1.2 Joints . . . 36
4.1.3 Meshes . . . 38
4.1.4 Checking the URDF . . . 39
4.2 URDF-launchfile . . . 39
4.3 Visualisation in RViz . . . 39
4.4 MoveIt! set-up assistant . . . 40
4.5 Integrating the robot model in Descartes . . . 46
4.6 Chapter summary . . . 46

5 The Descartes Software Package **47**
5.1 Trajectory points . . . 47
5.2 Transformation matrices and different possible transforms . . . 50
5.3 Inverse Kinematics . . . 51
5.4 Graph building . . . 51
5.5 Trajectory visualization in RViz . . . 54
5.6 Tolerances on local frame vs. Euler angle tolerances . . . 55
5.7 Used convention when defining trajectory points . . . 57
5.8 Custom cost function / edge weights / cost of trajectory point . . . 59
5.9 Saving generated data in .bag files . . . 63
5.10 Defining trajectory points . . . 64

6 Simulations and Examples **67**
6.1 Test case 1: Tube on Plate . . . 67
6.2 Test case 2: L Profile . . . 71
6.3 Test case 6: furniture piece . . . 76

7 Problems and Ideas **81**
7.1 Number of edges and RAM . . . 81
7.2 Iterative tolerances . . . 83
7.3 Glitchy behavior / jumping through obstacles . . . 87
7.4 Smoother trajectories . . . 89

8 Conclusion **91**

References **93**

Appendix A Complete URDF **95**

Appendix B URDF-tree **99**

Appendix C Python script for plotting data from bag-file **101**

Appendix D Testcase 1: tube on plate A **103**

Appendix E Testcase 2: tube on plate B **107**

Appendix F Testcase 3: L-profile with IKfast **111**

Appendix G Testcase 4: L-profile with KDL **115**

Appendix H Testcase 5: L-profile with iterative tolerances **119**

Appendix I Testcase 6: Furniture piece **123**

List of Figures

1 KUKA KR5 arc robot arm, with welding transformer and maintenance station. . 15
2 URDF with the corresponding kinematic and dynamic tree [15]. 18
3 Example of a robot displayed by the tf-package. 19
4 Probabilistic Roadmap example [23]. 21
5 Rapidly Exploring Random Tree (RRT) [25]. 21
6 Dijkstra's algorithm example [24]. 22
7 Rendering of the created workspace. 32
8 Rendering of the workspace with the Descartes-software package. 33
9 The workspace after building. 33
10 The folder which will contain the robot model ('kuka_description'). 34
11 Principle of describing a link in the URDF-file [13]. 35
12 Principle of describing an joint in the URDF-file.[13]. 37
13 All original meshes of the robot model. 38
14 The adjustment of the mesh of the robot base. 38
15 The adjustment of the mesh of the first link of the robot model. 38
16 The created robot model visualised in RViz. 40
17 Step1a: Start window of the MoveIt! Setup Assistant. 41
18 Step1b: Opening and parsing the URDF-model. 41
19 step2: Self-Collison Checking 42
20 step4a: Defining a planning group 43
21 step4b: Adding the planning group 43
22 step5: Adding robot poses 44
23 Resulting robot model displayed in RViz 45
24 Different joint solutions for a single tool pose. 47
25 Left: robot with base frame and tool frame. Right: tool frame moved into the trajectory point frame. 48
26 The welding torch with its frame. (Tool frame.) 48
27 Forward kinematics. 49
28 Three possible poses for the welding torch when using an axial-symmetric point. 50
29 Steps to construct a graph from trajectory points without tolerances. 52
30 Different toleranced frames. 52
31 Steps to construct a graph from trajectory points, including tolerances. 53
32 Visualized frames in RViz, every frame consists of three different markers. 54
33 Visualized example of intrinsic XYZ Euler rotations. 55
34 Conical tolerance zone for the z-axis. 56
35 Pyramid-shaped tolerance zone for the z-axis. 57
36 Different welding angles. 58
37 Bead shapes. 58
38 T-profile example for trajectory point convention. 59
39 Deviation with and without deviation cost. 60
40 Angle between optimal frame and toleranced frame's z-axis. 60
41 Illustrated graph, with chosen path in red. 61
42 Projection of the toleranced z-axis on the x-z- and y-z-planes. 62
43 Angles between z-axis projections and optimal frame's z-axis. 62
44 Example plot of joint angles. 64
45 Trajectory points across a line in space. 65

46 Trajectory points across a circle arc in space. 65
47 Trajectory points forming a helix. 66
48 Tube on plate workpiece. 67
49 Workpiece with trajectory, placed on the welding table. 68
50 TOP 1: Joint angles through time, with deviation cost. 68
51 TOP 1: Robot executing trajectory. 69
52 TOP 1: Joint angles through time, without deviation cost. 70
53 TOP 1: X angle error throughout time. 70
54 TOP 1: Y angle error throughout time. 71
55 Workpiece with trajectory, placed on the welding table. 71
56 L profile: Joint angles through time, with deviation cost. 72
57 L profile: Welding torch evading collision. 73
58 L profile: Joint angles through time, without deviation cost. 74
59 L profile: Y angle error throughout time. 74
60 L profile: Joint angles through time, with deviation cost and KDL solver. 75
61 L profile: Y angle error throughout time, KDL. 75
62 L profile: X angle error throughout time, KDL. 76
63 Furniture piece: workpiece with trajectory. 77
64 Furniture piece: trajectory points. 77
65 Furniture piece: robot executing trajectory. 78
66 Furniture piece: robot executing trajectory, zoom. 78
67 Furniture piece: Joint angles through time. 79
68 Furniture piece: X angle error throughout time. 79
69 Furniture piece: Y angle error throughout time. 80
70 Steps to construct a graph from trajectory points, including tolerances. 81
71 L profile with trajectory. 84
72 Robot executing trajectory. 85
73 Robot executing trajectory. 85
74 L profile, iterative tolerances, joint angles. 86
75 L profile, Y angle errors, iterative tolerances. 86
76 Scenario with tall pole. 87
77 Joint angles through time, jumps are within ellipses. 88

List of Tables

1 Summary table 29

Abstract

This master thesis, performed within the research group ACRO, aims to enhance the production flexibility of robot welding through automatic, collision-free trajectory generation. The first subgoal was to compare and select an existing open-source software to automatically generate collision-free trajectories for welding robots. The second subgoal was to evaluate it on its applicability to robot welding, and subsequently, to improve it.

To select a suitable software package, a literature study has been carried out, in which different software packages were compared. The chosen Descartes software package starts from a sequence of trajectory points, with acceptable tolerances specified by the user. With the help of the inverse robot kinematics, it then generates a graph, which is searched to find the path with the lowest cost.

To evaluate the chosen software package, different test scenarios were developed. Various limitations were revealed and improved. A new cost function based on rotational errors of the welding torch has been added, which keeps the welding torch closer to its optimal orientation. On top of that, it allows to differentiate between specific welding angles, making it possible to prioritize changing one over the other. A new implementation to define rotational tolerances on trajectory points leads to tolerances which are more intuitive and more practical. Finally, the code has been made available online together with a tutorial. They can be found respectively on the GitHub and the ROS Descartes wiki page.

Abstract

Deze masterproef, uitgevoerd bij de onderzoeksgroep ACRO, had als doel de productieflexibiliteit van robotlassen te verhogen door automatische, botsingsvrije trajectgeneratie. Het eerste subdoel was het vergelijken en selecteren van een bestaande open-source software voor het automatisch genereren van botsingsvrije paden. Het tweede subdoel was de gekozen software op zijn toepasbaarheid voor robotlassen evalueren, verbeteren en uitbreiden.

Het selecteren van een geschikt softwarepakket gebeurde op basis van een literatuurstudie die verschillende bestaande softwarepakketten vergeleek. Het gekozen Descartes-softwarepakket vertrekt van een sequentie robotlocaties met aanvaardbare toleranties gespecifieerd door de gebruiker. Daarna genereert het met behulp van de inverse kinematica van de robot een diagram en zoekt hierin het pad met de laagste kost.

Om de gekozen software te evalueren zijn verschillende testscenarios uitgewerkt. Verschillende beperkingen, die zo aan het licht zijn gekomen, zijn verbeterd. Een nieuwe kost op basis van hoekafwijkingen is toegevoegd, waardoor de bewegingstrajecten de optimale oriëntatie van de lastoorts beter benaderen. Bovendien laat het toe om bepaalde lashoeken prioritair aan te passen boven andere. Een nieuwe implementatie voor het definiëren van rotationele toleranties op trajectpunten zorgt voor toleranties die intuïtiever en praktischer zijn. Ten slotte is de code online beschikbaar gemaakt en een tutorial geschreven. Ze zijn respectievelijk te vinden op GitHub en de ROS wikipagina.

1 Introduction

1.1 Background

This master thesis is situated within the research group ACRO, part of the University of Leuven. ACRO focuses mainly on applications using robotics and machine vision. One of the projects of ACRO is the so-called *Smartfactory project*[1]. The goal of this project is to aid the manufacturing industry to achieve sustainable production by helping with the introduction and development of the Smart Factory concept. To reach this goal, 7 tangible technological challenges were conceived. They are also available in [1].

1. Zero ramp-up: trial or test series of products should not be necessary. The start-up phase of production is reduced to a minimum. (Ideally there is no start-up phase.)
2. Safe interaction between humans and robots: robots and humans are able to work safely alongside each other, while the production still remains accessible.
3. From computer aided programming to auto-programming: the robot programming requires no special training, reprogramming the robots for the production of a new series of products can be achieved in less than 10 minutes.
4. Intelligent quality assurance: quality control is automatic and integrated. All products are checked.
5. Benchmark of robot control software: using the correct software in different situations, with a focus on the feasibility of restarting the process when necessary. Complex robot paths can be programmed offline.
6. Remote production monitoring: production generates data that is fed back into a real-time monitoring system, to limit potential damage in the case of unplanned events such as a breakdown.
7. Couple stand-alone resources into networked production cells: smart production cells do not form isolated islets, but communicate with each other to form a *Smart Factory*.

The *smart factory*-concept is relevant to this thesis, specifically regarding challenges 3 and 5.

1.2 Problem definition

Flemish manufacturing companies are experiencing difficulties in competing with companies in low-wage countries. These manufacturing companies often have an abundance of cheap labor to their disposal, allowing them to produce their products with a lower cost. Because of this, many on-shore companies are willing to invest in further automation of their production processes. The goals of this automation are to reduce costly working hours to a minimum and/or enhance the flexibility of the production processes. Processes using autonomous robots offer the possibility to save many hours of manual labor. On the other hand, the programming of these robots requires specific technical knowledge, and extra time to implement. This method of automation is especially attractive when large series of products are produced. In this case the large amounts of time saved in manual labor, far outweighs the extra cost for the

time and knowledge necessary to program the robots. However, this method of automation is not very feasible in regard to the production of small series of products, or many unique products. Every time a new series has to be produced, the robots have to be programmed again, removing the cost reduction of saved manual labor. To allow this method to succeed in these cases, the manual programming of the robots has to be reduced, or removed altogether. In the application of robot welding this comes down to automatically generating the trajectories of a welding robot to place a weld on a workpiece. The software that generates this trajectory needs to take into account the possibility of collisions between the robot and its environment. Otherwise, trajectories could be generated that would be impossible to execute on a real robot. On top of that, the welding speed and different other weld parameters need to be controlled to guarantee a certain weld quality.

1.3 Objectives

The goal of this thesis is to select and use existing open-source software to automatically generate a collision-free trajectory for a welding robot. The software is then evaluated and subsequently improved and expanded, with our specific application (robot welding) in mind. The evaluation of the software needs to happen by generating trajectories on multiple, strategically chosen, test scenarios. A minimum requirement of this thesis is that a trajectory can be generated for a simple workpiece, together with a simple obstacle which has to be avoided. Ideally, this would mean that the information about the weld would be extracted from a CAD file defining the workpiece. However, this is not a goal of this master thesis. The information about the welding paths is generated ourselves. This information is then used as a foundation to generate a correct trajectory. As mentioned in the problem definition, the software needs to take into account possible collisions between the robot and the environment, or the robot with itself. It also needs to check if the generated movement of the robot is executable. In other words, do we adhere to the maximum allowed joint ranges and speeds? Singular positions should also be avoided. If no trajectory can be found, the software must try to find an alternative trajectory by changing the weld parameters between certain predefined limits. This could mean changing certain welding angles, or slightly changing the position of the welding torch when necessary. In the case of changing certain welding angles, different welding angles are preferred to be changed over others, because of their different effects on the weld quality. The software should be able to distinguish between the different welding angles. The user should also have full control over the allowed tolerance limits. After generating the trajectory, the software needs to automatically generate a report with essential information regarding the trajectory. This report contains for example the changed welding parameters and the joint positions during the movement. An important supplementary goal is that the software needs to be made available to third parties. To do this, the code should be published online, and if possible, a tutorial should be written.

1.4 Methods and materials

The master thesis can be divided in following parts. First of all a literature study is carried out. In this literature study, three open-source software packages for trajectory generation are examined and compared. The different software in question are the *Descartes-software package* [2], *Trajopt* [4] and *CHOMP* [5].

After the literature study, the main focus was the evaluation and enhancement of the chosen (Descartes) software. Different tests were performed using scenarios. For this purpose, ACRO provided a KUKA KR5 arc robot arm, equipped with a Fronius 4000 welding transformer, and a maintenance station with torch cleaner. However, this installation has not been used in this thesis. All the tests have been carried out in simulation. In this simulation environment, a robot model based on the KUKA KR5 arc robot arm was used.

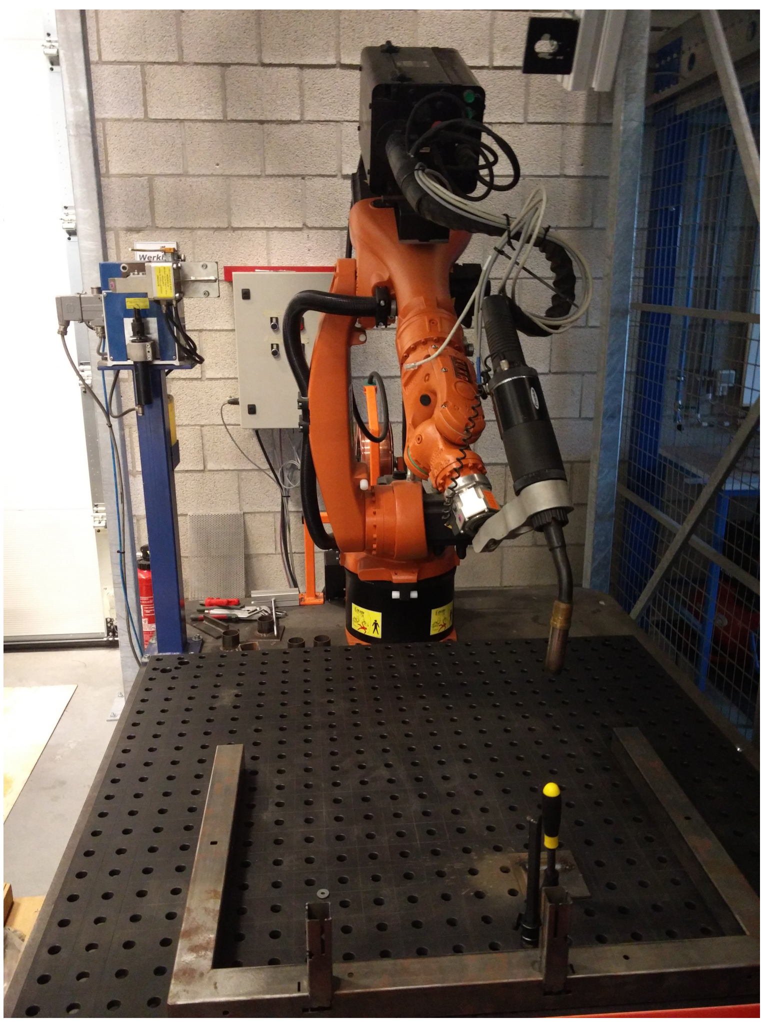


Figure 1: KUKA KR5 arc robot arm, with welding transformer and maintenance station.

1.5 Summary

In this chapter, the setup of the workspace of the master thesis has been described. First of all the framework, in which the master thesis has been conducted, has been described. The problem definition, which gives the research its meaning, was then explained. After that, the goals of the master thesis were outlined. Finally, the used methods and materials where shown.

2 Literature study

To generate trajectories for our robot, an existing software package has to be chosen. At the present time, there are multiple open source software packages available. To make a good choice between different path planning software it is essential to compare different software packages and different path planning methods. This chapter starts with a short introduction of ROS (Robot Operating System) and other popular software packages used within the ROS framework. This is done for readers who are not familiar with ROS, so that they are able to understand certain terms that might be used in the coming chapters of this thesis. After this familiarization follows a short description of certain path planners that may be used for collision-free trajectory generation of robots. First a few so-called *sampling-based* algorithms will be discussed. Then different software packages for trajectory generation and optimization will be looked at and compared. (CHOMP, TrajOpt, and Descartes.) We conclude with a summary table of the advantages and disadvantages of the chosen packages, and the final choice made for this thesis. Although the *Descartes* software package was certainly compared to the other packages in the literature study, this section has been omitted, because the functioning of this package will be explained in more detail in the following chapters.

2.1 Utilized software

2.1.1 ROS

The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behaviour across a wide variety of robotic platforms [6].

Tasks that seem trivial to humans are not always trivial to robots and vice versa. Often, due to a variation or a change of the environment, the necessary implementation of a robot function can completely change. Because handling all of these variations is a difficult task, it cannot be expected that an individual, a laboratory or even an institute can handle all of these variations for each robot function. As a result, ROS was built from the beginning with collaboration in mind.
Consequently, at the base of ROS is its communication system, which allows information to be exchanged between different modules in the form of messages. Messages can be built by defining a new message class, with all the desired data components and their names. However, ROS also provides a set of standard messages ranging from information about a robot's pose, to maps of the environment. By using these standard messages, self-written code can work seamlessly with other software that uses the same messages.

Software in ROS is organized in packages [8]. A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module. The goal of these packages is to provide this useful functionality in an easy-to-consume manner so that software can be easily reused. For a better understanding of the rest of the thesis, the used packages,nodes and ROS-based software will be briefly described.

At the moment, multiple versions of ROS are distributed. The version used in this thesis is

ROS Indigo Igloo [9], which was released on the 22nd of July 2014. More information about ROS can be found at the ROS-Wikipage [10].

Nodes and Packages

A *node* [11] is a process that performs computation. A robot control system usually consists of multiple nodes, communicating with each other through messages. For example, one node might calculate the joint positions of a robot while another node visualizes these joint positions.

URDF URDF (Unified Robot Description Format) is an Extensible Markup Language in XML format for the representation of a robot model. It is used to develop a robot model by defining the different joints and links of a specific robot model, including their physical properties. URDF can only describe robots that have a tree-like structure in their links. This means that the robot must have rigid links, that are connected to each other using joints. The created robot model can then be used to perform simulations. The *URDF-package* [12] [13] contains a built-in C++ parser for URDF.

SRDF The *SRDF-package* [14] is a package intended to contain information about the robot that is not in the URDF file. The intention is to include information that has a semantic aspect to it. Examples of these semantic aspects are:

- The joint and link names of a robot and their assembly into a move-group (a set of links and joints that work together).
- Predefined robot positions (joint values). For example a home position.
- Information about passive joints that are not actuated.
- Information about the disabling of the collision detection between certain links of the robot.

kdl parser The *Kinematic and Dynamics Library (KDL)* [15] is a ROS package that provides a number of parser-utilities. The goal of the package is, starting from the URDF, to create a tree-like structure displaying the kinematic and dynamic parameters of the robot. KDL can be used to publish the joint states, and compute the forward and inverse kinematics of the used model.

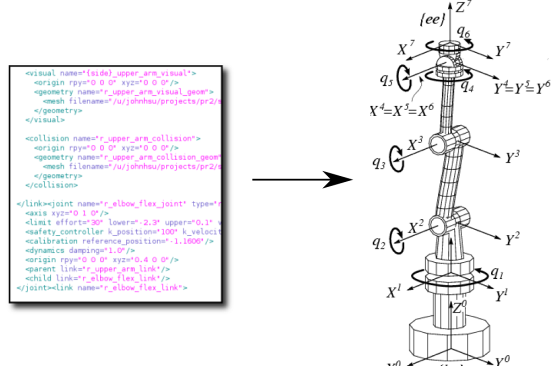


Figure 2: URDF with the corresponding kinematic and dynamic tree [15].

Joint State Publisher The *joint state publisher*[13] [16] is a package containing a node, which reads the robot model, containing the state of the robot's joints. It then publishes these joint values. A GUI with sliders may be used to change the joint values of all non-fixed joints.

Robot State Publisher This package reads the (published) robot joint states and publishes the 3D poses of each robot link, using the kinematics tree built from the URDF [17]. The 3D pose of the robot is published as a *ROS tf (transform)*.

tf *ROS tf* [18] is a package that visualizes the different transformed coordinate frames for every link of the robot. It is possible to see how the robot's position will change if there is a change in joint states.

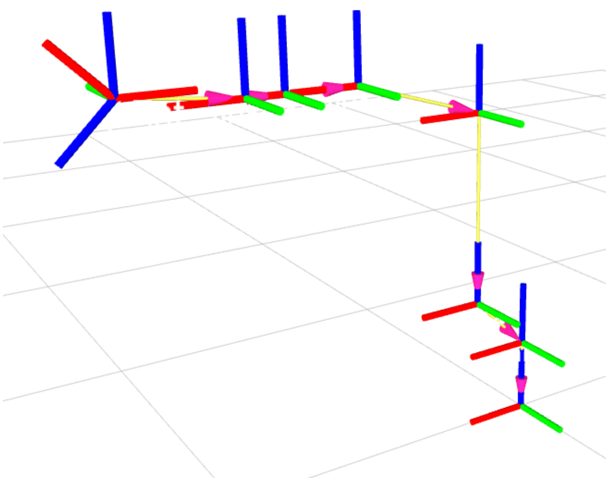


Figure 3: Example of a robot displayed by the tf-package.

Roslaunch (Launchfiles) *Roslaunch* [19] is a tool for easily launching ROS-nodes by writing a so-called .launch-file. The launch file can be executed using following command-line.

```
$roslaunch package_name file.launch
```

ROS-industrial (ROS-I) *ROS-Industrial* is an open-source project that extends the advanced capabilities of ROS to manufacturing automation and robotics. The ROS-Industrial repository includes interfaces for common industrial manipulators, grippers, sensors, and device networks. It also provides software libraries for processing path/motion planning. Unlike the packages described above, ROS-I is a collection of packages [20].

ROS-based software (tools)

RViz *RViz*[31] is a ROS-package for 3D-visualization. After a robot model has been created, it can be visualised in RViz. Rviz transforms the URDF-model in a 3D-model and enables the possibility to observe movements of the robot joints. Next to the robot model another useful application of RViz is the visualisation of the environment around the robot, the different coordinate frames, and user-made visual markers.

MoveIt! *MoveIt!* [21] is a ROS-based software for the manipulation of robots. It provides an easy-to-use platform for developing advanced robotics applications, evaluating new robot designs and building integrated robotics products. MoveIt! makes it possible to generate collision-free trajectories for robots.

2.1.2 Trajectory generation software

We begin this section by going into a popular way of automated collision-free trajectory generation in academia. First, a so-called "Sampling-based Algorithm" tries to find a possible trajectory. Often, depending on the used algorithm, the path is not very smooth. This means that sudden movements may occur in the range, associated with high accelerations. To solve this problem, an optimization step is often used, in which a separate algorithm tries to make the trajectory, generated with the "sampling-based algorithm", smoother. This is called the optimization step.

Then, two different sampling-based algorithms are discussed. These are also implemented in ROS, in a library called *OMPL* [22] (Open Motion Planning Library). After that, three different software packages for path generation are discussed and compared: CHOMP, TrajOpt and Descartes. The comparison is based on the features of the software packages, while keeping the application of robot welding in mind.

Probabilistic Roadmap (PRM)

The *Probabilistic Roadmap Planner (PRM)* is an algorithm for the generation of collision-free trajectories. In this case, a random position in the configuration space is taken and tested whether or not the robot is in collision. If the robot is not in collision, this point is added to a list of points within the configuration space that are collision-free. Afterwards, the algorithm attempts to connect these different points using a local path planner. In the example, these are linear paths between 2D points, but these paths can also be determined differently. Eventually, using a search algorithm (e.g. Dijkstra's algorithm), a trajectory is generated with a set of these points. The PRM method therefore consists of two phases [23]:

1. The learning phase: A list of points is compiled by calculating collision-free points in the configuration space. Using a fast local path planner it is checked if a collision-free path between the different collision-free configurations.

2. The Search phase: A trajectory is searched between start and end points by connecting the trajectories found in the previous phase. The way this path is chosen depends on the algorithm used. For example, in the following 2D example (figure 4), the length of possible paths could be taken as a cost. After that, Dijkstra's algorithm can be used to find the path with the lowest cost.

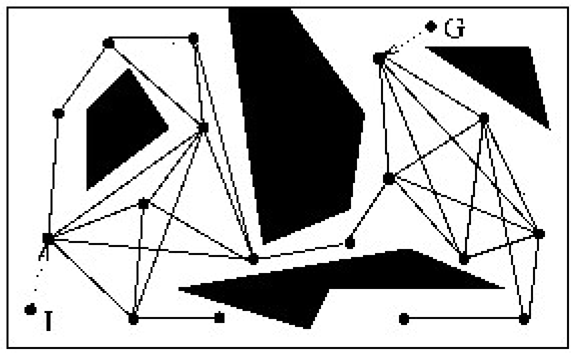


Figure 4: Probabilistic Roadmap example [23].

One of the advantages of PRM is that this method is probabilistically complete. That is, if a collision-free path exists, it can be guaranteed that this global path is found with probability approximating one as the number of sampled configurations goes to infinity. Thus, if a path exists, it will be found. But it may cost a lot of calculations to determine the path. Moreover, this method is well applicable to robots with many degrees of freedom. Trajectory generation software based on PRM is available within *OMPL* [22]. OMPL can also be used within MoveIt! [21].

Rapidly-exploring Random Tree (RRT)

Rapidly Exploring Random Tree (RRT) is an algorithm for the rapid generation of a trajectory with a single start and end point. The Rapidly Exploring Random Tree is especially suitable for problems with many degrees of freedom, trajectory-dependent and kinematic constraints [24]. RRT's are developed by expanding a tree-like structure with any random point in the configuration space. This causes the tree structure to grow into large unexplored areas. The algorithm is very effective in finding a possible trajectory, but it is often not enough to find a proper, optimzed route. This is among other things due to the fact that the algorithm does not limit any additional variables, such as the length and the smoothness of the trajectory. As a result, the algorithm is used as a component that can be integrated into the development of other path generation algorithms.

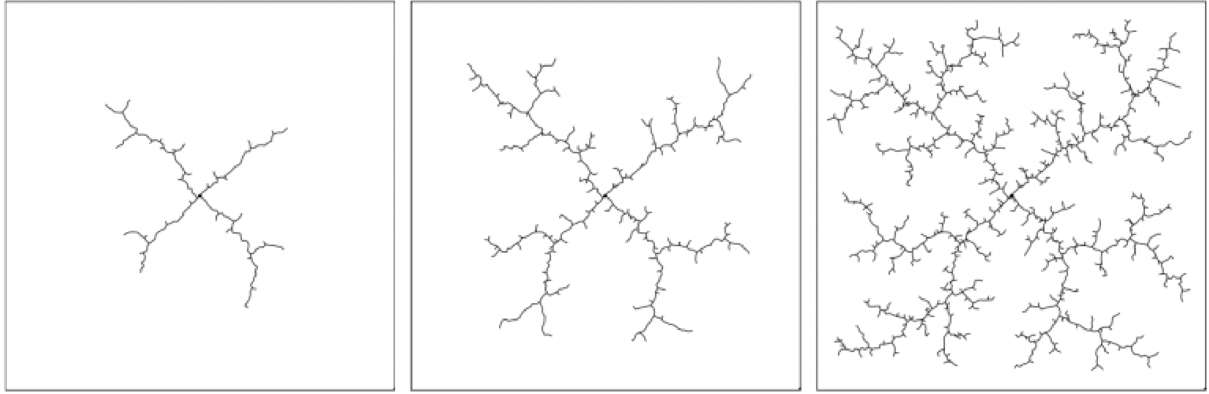


Figure 5: Rapidly Exploring Random Tree (RRT) [25].

Dijkstras algorithm

Dijkstra's algorithm, is an algorithm to seek the shortest path between two nodes of a graph. It can also be used to calculate the shortest distance between a start and end point, connected by a number of intermediate nodes and edges (6). Dijkstra's algorithm can be used for the generation of robot trajectories when a graph is given [26].

DIJKSTRA'S ALGORITHM

Figure 6: Dijkstra's algorithm example [24].

2.2 Research trajectory generation software

2.2.1 Descartes Path Planner (ROS-I)

Descartes, named after French scientist René Descartes, is a software package used to generate trajectories for industrial robots. This planner uses information about the desired trajectory, and the robot geometry, which is then fed into a path planning algorithm.

Trajectory information

This software package uses trajectory points, which are discrete samples from a predefined

path. This trajectory can be built using 3 different kinds of trajectory points:

1. Cartesian points
2. Angular position of the robot joints (joint trajectory points)
3. Axial symmetric points: these axial symmetric points are points in which a rotation of the end effector, around a certain axis, does not have any effect on the process (symmetric end effector) or this difference in rotation does not matter. For example, when rotating the wrist joint, if a symmetric tool is precisely fixed in the center of the wrist, the rotation of this joint does not matter.

Most trajectory points are 6D points, with three translation and rotation components. It is possible to give a certain degree of uncertainty to the spatial coordinates. For example a certain tolerance on the X-, Y- and Z-coordinates or a tolerance on the different rotations of the end effector. It is also necessary that these points are linked to a given timing. The software needs this to get an idea of the speeds with which the path, consisting of different trajectory points in 3D space, should be executed. The functioning of this package corresponds well to the approach of various applications of industrial robots, such as spray painting, welding and milling. In these applications there is always a predefined desired path which the robot has to follow. For example, the welding path that has to be followed. Normally the robot has enough, or even too many, degrees of freedom available to follow this path. Often, the path to be followed is not fully defining the poses of the robot, because the robot has some additional degrees of freedom available. With these additional degrees of freedom, different positions may be adopted that still meet the constraints of the path that has to be followed.

To create a good weld, it is essential to execute this path at a correct speed. As a result, the points that make up the path are bound in time and thus also useful for the Descartes-package. At first glance, the necessary trajectory information, which the software needs to perform proper trajectory planning, seems to be well in line with the available information about the welding paths used in this project.

In the example on the Wiki page about the Descartes package, trajectory items are generated in the following way in the code [2]:

```
for (unsigned int i = 0; i < 10; ++i)
{
  Eigen::Affine3d pose;
  pose = Eigen::Translation3d(0.0, 0.0, 1.0 + 0.05 * i);
  descartes_core::TrajectoryPtPtr pt = makeTolerancedCartesianPoint(pose);
  points.push_back(pt);
}
```

If these points depend on a predefined load path, a solution must be found to make these points available in the code. For experimental purposes, the trajectory points could also simply be defined by us. This means that there is no automatic generation of the trajectory points from a CAD file.

Robot model

The robot model contains information about the geometry and kinematics of the robot. Usually, an industrial robot arm consists of a number of joints, connected through rigid parts (links). The model determines where and how these parts are connected. For example, how these joints can move (range of rotation), the sizes of the links, etc. The robot model is used for:

1. Calculating Inverse Kinematics
2. Calculating Forward Kinematics
3. Collision detection
4. Determine the limits of the robot

As stated above, the robot model is used to determine the forward kinematics. The position of the end effector is determined by means of the known positions of the robot joints. Also for the use of inverse kinematics the robot model is needed to determine the positions of the robot joints from a known position and orientation of the end effector.

When the Descartes planner is executed, it searches the ROS parameter server for the previous robot model loaded within MoveIt! using the MoveIt! set-up assistant [2]. Using this set-up assistant, it is possible to load an URDF-file, which describes the robot. Since we currently have an example of such an URDF, this is an advantage.

Route planner

The path planner is the highest component within the Descartes package. It tries to find the optimal robot movement, using the given trajectory information, and the given robot model. The Descartes-package optimizes the generated trajectory to the so-called joint movements of the robot. This means that for a given path, it will try to move the robot as little as possible.

Currently there are two different path planners available in Descartes. The so-called "Dense Planner" and the "Sparse Planner".

The Dense Planner tries to find a path optimized for the joint movements through the points of a given route. This happens in three steps:

1. For all points of the trajectory, all possible robot positions are calculated using the inverse kinematics.
2. On the generated robot positions, a couple of calculations are performed for each pair of trajectory points. For every possible robot position of the first trajectory point, calculated in the previous step, the motion cost is calculated that is required when moving to any possible robot position of the next point.
3. With this information a graph can be generated, with points representing the kinematic solutions from the first step. Lines between these points, coupled to a certain cost, represent the possible movements between the different robot positions.
4. Finally, using the Dijkstra's algorithm, the trajectory with the lowest cost is determined.

The selected trajectory is now the generated trajectory, in the form of discrete joint positions for every predefined trajectory point.

The Sparse Planner works on the same principle as the Dense Planner, but tries to reduce the number of calculations by sampling a given path. It will calculate the possible robot positions for a portion of the points from the range,using the inverse kinematics. Then, an attempt is made to find a solution to the intermediate points using a linear interpolation within the joint space, and the forward kinematics. For example, if we want to generate a trajectory through points A, B and C. The Sparse Planner will then calculate the possible kinematics solutions for points A and C. It then attempts to reach the intermediate point B by changing the robot pose from A to C. This is controlled by the forward kinematics that requires fewer calculations than the inverse kinematics.

Limitations

None of the Descartes planners checks the occurrence of singularities, or points unreachable by the robot. In these cases, path generation might not be possible [3]. The only factor pushing the robot away from possible singularities, is the cost that is connected to large movements.

Order of constraints

For the application of robot welding it may be interesting to impose constraints to position and orientation of the end effector that are adjusted by the planner in the event of a collision. In this way, when detecting a collision along the trajectory, the position or orientation can be adjusted so that the effect on the quality of the weld is minimized. With the help of a properly defined priority of adjustments to the ideal trajectory, depending on which one has a greater effect on weld quality, the adjustments are only implemented if there is no other solution with better welding quality.

At this time there is no priority of parameters implemented within Descartes. However, Descartes is certainly able to handle tolerances on the 6D-Cartesian points, which are used to partially define the trajectory. With this package, it is thus possible to define a certain tolerance in advance, on certain welding parameters. This means that parameters that have a major effect on the quality of the weld will therefore be given a different tolerance priority than welding parameters with a lesser effect on weld quality.

2.2.2 CHOMP

CHOMP (Covariant Hamiltonian Optimization for Motion Planning) [5], is a method to optimize robot trajectories. This method differentiates itself from previous optimization techniques by allowing collisions in the initial trajectory, which it then optimizes. This allows CHOMP to start off with a very simple representation of a path, out of which it is often able to generate a collision-free and optimized trajectory, which can be executed directly on the robot. An example of such a "simple" starting path can be a movement from point A to B, in a straight line, that might be in collision with the environment.
By directly optimizing the path, CHOMP does not require a real *path-planner* to generate the trajectories. Instead, CHOMP optimizes the starting trajectory in such a way, that it actually

generates the final trajectory on its own.

Another advantage of CHOMP is that it is not necessary to transform obstacles into configuration space, to be able to execute the collision detection. It is often hard to construct these representations in the configuration space, when there are many degrees of freedom.
To execute its optimization, CHOMP uses a technique called *covariant gradient descent*. It uses a cost function consisting of two different terms. The first term calculates a cost based on the vicinity of obstacles to the different links of the robot, or actual collisions between the robot and the environment. The second term generates a cost in function of the joint movements, more precisely the smoothness of the path, and sudden acceleration of the links. Even though part of the generated cost is based on joint acceleration, there is no constraint on the path execution time. CHOMP then calculates the gradient of this cost function. Using this gradient, it will modify the original trajectory by a small step, causing it to have a lower cost calculated by the same cost function. Repeatedly optimizing the new trajectory generated by the previous step causes it to converge to a certain (possibly local) minimum cost. CHOMP guarantees that for each optimization step, the optimized trajectory remains smooth.

Explaining the way that CHOMP takes into account obstacles is also relevant because it explains why CHOMP is able to generate a collision-free trajectory from an input trajectory that already contains collisions. If most obstacles are static objects, it can be advantageous to calculate a distance field in advance. This means every point in space is sampled, and the distance to the nearest collision object is calculated. The distance is negative *inside*, positive outside, and zero on the edge of obstacles. Because the distance field can be calculated inside of objects, CHOMP is able to calculate a valid gradient for every point within the defined space. A trajectory can then be optimized, even if the input trajectory intersects obstacles, or if the robot is in collision.
In CHOMP, the robot is represented by a skeleton which is constructed using a sequence of intersecting spheres, cylinders, and other shapes created by dragging spheres through space. This causes the robot to be approximated by a *union of spheres*. By using this simplified representation, in combination with the distance field, the collision detection does not require a lot of computations. Every center point of a sphere from the robot skeleton can be instantly compared to the value of the distance field in that location. If this distance is greater than the radius of the sphere, the sphere is not in collision.

A disadvantage of this software is the impossibility to add constraints to a path in between defined points. If this software were to be used to generate trajectories for a welding robot, this would mean that the orientation of the welding torch would not be able to be defined during the movement. This software is more suitable for pick&place trajectories, or other trajectories where the followed path itself is no concern, as long as it is free of collisions. Because of this, the software is not very suitable for welding applications.
Above that, as of now there is no CHOMP plugin available for MoveIt! in the indigo version of ROS. This means that the visualization of the trajectory generation becomes more complex.

2.2.3 TrajOpt

TrajOpt [14] is, like CHOMP, a software package to generate trajectories for robots, which is based on optimizing an already existing path. Just like CHOMP, it is able to find collision-free trajectories starting from simple, straight-line trajectories between waypoints. The input trajectories are allowed to be in collision, or intersect the environment. Although the approach is very similar to that of CHOMP, it differs from CHOMP in two ways. First, a different method is used to numerically optimize the trajectory. In CHOMP this happens using a cost-gradient. In TrajOpt, it is formulated as a so-called *sequential convex optimization.* Secondly, the method used to detect collisions, and translating these collisions into a cost is implemented differently.

Sequential Convex Optimization

Path planning problems for robots can be formulated as a non-convex optimization problem. A function $f(x)$ is tried to be minimized, taking different constraints into account, like limited joint angles, and joint speeds of the robot. These constraints can be formulated as inequalities $g_i(x) \leq 0$. Further, different equalities need to be taken into account, like a certain optimal en effector pose. These are formulated as $h_i(x) = 0$. As optimization function $f(x)$, TrajOpt uses the squared sum of the joint movements: $f(x_{1:T}) = \sum_{t=1}^{T-1} ||x_{t+1} - x_t||^2$
This means the path is optimized to minimize total robot movement.
When solving a non-convex optimization problem, there is the possibility of local minima existing in the optimization function $f(x)$. The optimization algorithm will bring the robot to these local minima, and remain stuck in it. If this is not taken into account, the robot will execute a trajectory which could possibly be optimized further. To solve this, TrajOpt uses sequential convex optimization. Around the main non-convex problem, a convex sub-problem is built. This means that the optimization function contains a minimum, and that this minimum is the global minimum within the sub-problem. With the help of this sub-problem, a step Δx is generated. TrajOpt then guarantees that this step Δx within the sub-problem, is also a correct step in the non convex main problem. To be able to construct this sub-problem, the optimization function $f(x)$ and the constraints $g_i(x) \leq 0$ and $h_i(x) = 0$ are converted to a convex approximation.

To be able to use this algorithm well however, two requirements have to be met. First, the step Δx has to remain small enough. This means that the found solution is still within the boundaries where the convex approximation of the different functions is valid. To achieve this, TrajOpt uses a so-called *trust region*, a cube-shaped region around the iteration x_t.

Secondly, a way has to be chosen to convert unfulfilled constraints into a cost, which in the end is supposed to reduce the amount of constraint violations to zero. The costs for the inequalities are defined by $|g_i(x)|^+$, where $|x|^+ = max(x, 0)$. The costs for the equalities are defined by $|h_i(x)|$. These costs are added together and multiplied by a *cost coefficient* μ. Every iteration, this cost coefficient is multiplied by a factor 10, which will cause the costs to rise rapidly with a growing size of iterations. This makes sure that for a possible solution that is very favorable in minimizing $f(x)$, but which does not adhere to the different constraints, the constraints will create a cost that is many times larger than the favorable effect of the optimized $f(x)$. In this way, a solution that follows the constraints is thus preferred.

When using TrajOpt, the simulation of the robot is executed using a software package called

OpenRave [15]. OpenRave uses different robot models (COLLADA file format, or OpenRave XML) than MoveIt!. ROS does however have a package called *collada_urdf*, which allows conversion of a URDF robot model to a COLLADA file format [16]. If TrajOpt is used, the robotmodel will not have to be created from scratch, but it might be easily converted using this package.

Constraint priorities

It is possible to edit the used cost functions within TrajOpt [17]. Also, multiple costs can be used simultaneously. For example, it is possible to add a cost in function of the joint speeds together with a cost in function of robot collisions with the environment. It's also possible to create constraints for the end effector. Using these constraints, it is possible to guarantee that when placing a weld, the welding torch will assume correct poses throughout the complete trajectory. Because changing the cost function in TrajOpt is relatively simple, it seems to be possible to prioritize certain welding angles over others, by combining different costs to them. It might even be possible to take into account other such welding parameters, like end effector speed, or the distance between the welding torch and the workpiece. The costs of these different welding parameters need to be chosen in such a way that their effect on the weld quality is correctly reflected in the total cost of the trajectory. For example, when changing a crucial welding parameter, the change in cost is supposed to be greater. When a less important welding factor is changed, the cost is only supposed to change slightly.

2.2.4 Comparison

In table 1 we use a scoring approach to make a choice between the different software packages discussed in this literature study. In the left column, the different criteria are specified, based on which the packages receive a score. In the three following columns, the scores are noted for every package. A "+" means that the package satisfies the criterion, and thus offers an advantage. A "-" shows a disadvantage. Because not all the criteria are equally important when it comes to robot welding, we use an extra weight between 1 and 5, displayed in the last column. This describes the relevance of the criterion. To calculate the final score, the relevance is added or subtracted for every criterion, depending if the score for that criterion is a "+" or a "-".

Criterion	**Descartes**	**CHOMP**	**TrajOpt**	**Relevance**
Starting trajectory doesn't have to be collision-free	-	+	+	2
Different types of input coordinates, including combination	+	-	-	4
Fast calculations	-	+	+	1
Ability to allow tolerances on input coordinates	+	-	-	5
Ability to add timing constraints to input coordinates	+	-	-	5
Available documentation	+	-	+	3
Can start from a simple trajectory	-	+	+	2
Ability to use meshes to define collision objects	-	-	+	2
MoveIt! plugin	+	-	-	4
Ability to change the cost function and add constraints	+	-	+	4
Trajectory needs to be largely known in advance	-	+	+	1
URDF robotmodel	+	-	-	4
Total	21	-25	-7	

Table 1: Summary table

The *Descartes* software package has a much higher score than the other two, this seems to be a good choice for this application.

2.3 Chapter summary

In this literature study, we started off with a description of the utilized software. This entails a description of the used ROS-packages, and a general description of trajectory generation software. Secondly, three software-packages for the creation of collision-free robot trajectories were reviewed, based on their potential for robot welding. The Descartes-software package was chosen to be the most promising software, and was used in the rest of this master thesis.

3 Set-up and installing the Descartes-software package

This chapter gives an overview of the steps necessary to set up the Descartes software package. First of all a workspace is created, containing all the necessary files. Then the Descartes-software package is installed. Finally, the folder containing the robot model files is created.

In this chapter some of the used code and command-lines will be shown. For clarity, the following convention will be used.

Code:

```
Include <ros/ros.h>
```

Command-line input:

```
$Sudo apt-get install ros-indigo-perception
```

3.1 Workspace

The *workspace* is the place on the hard drive that will contain all files related to this project. It will contain the Descartes-software package, the robot model, the test cases, etc. The creation of a workspace is an easy process, but knowing how it is done explains some other used commands [27].

The first command is to setup the ROS-environment:

```
$source /opt/ros/indigo/setup.bash
```

Now to create the catkin workspace:

```
$mkdir -p  /catkin_ws/src
$cd  /catkin_ws /src
```

This will create an empty workspace. This workspace can now be build using:

```
$cd  /catkin_ws/
$catkin_make
```

Finally, the workspace needs to be *sourced*. This allows ROS to recognize packages in your workspace.

```
$source devel/setup.bash
```

The above procedure is the general procedure. The commands used to create the specific workspace used in this master thesis follow here. The name of the workspace used is 'lasrobot_ws'.

```
$mkdir masterproef_ws
$cd masterproef_ws/
$catkin_init_workspace
$cd  /catkin_ws/
$catkin_make
```

After executing these commands, a folder with the following layout should be created (figure: 7).

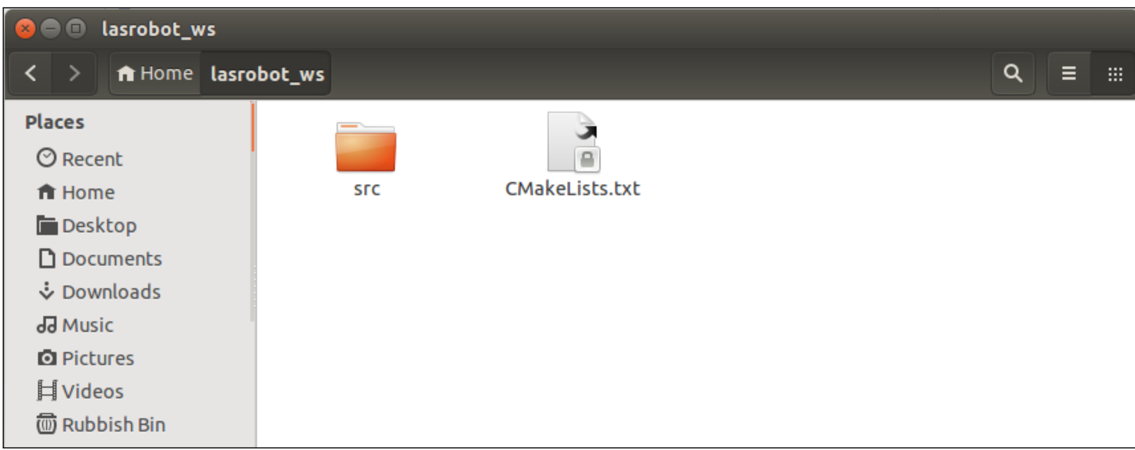


Figure 7: Rendering of the created workspace.

3.2 Installing the Descartes-software package

To install the Descartes-software package, the necessary files are first downloaded from:

https://github.com/ros-industrial-consortium/descartes.git

Extracting the files into the src-folder of the created workspace will provide all the needed packages, libraries and utilities necessary to run Descartes. A basic example executable can be downloaded from:

https://github.com/ros-industrial-consortium/descartes_tutorials.git

In the end the src-folder of the workspace should look like figure 8.

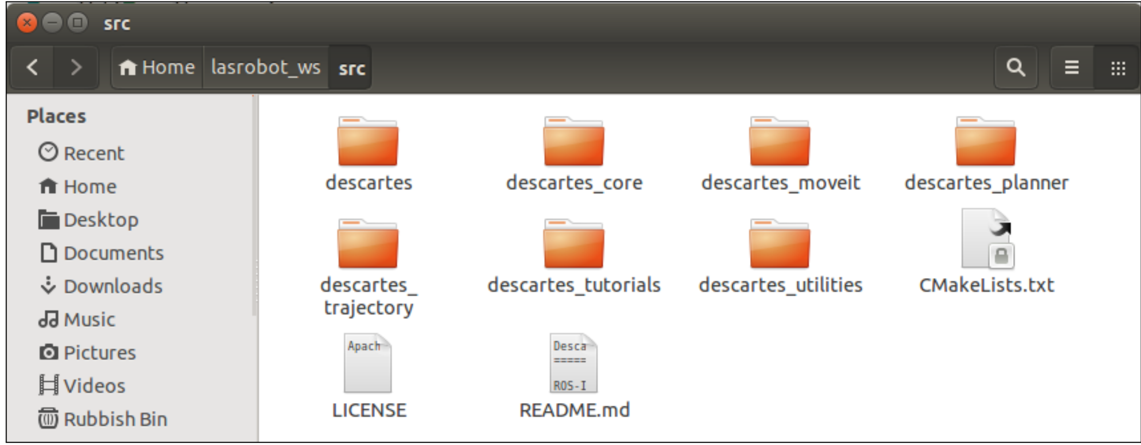


Figure 8: Rendering of the workspace with the Descartes-software package.

Afterwards the workspace should be build using the command:

```
$catkin_make
```

By building the workspace there the folders 'build' and 'devel' should automatically be created in the workspace as illustrated in figure 9.

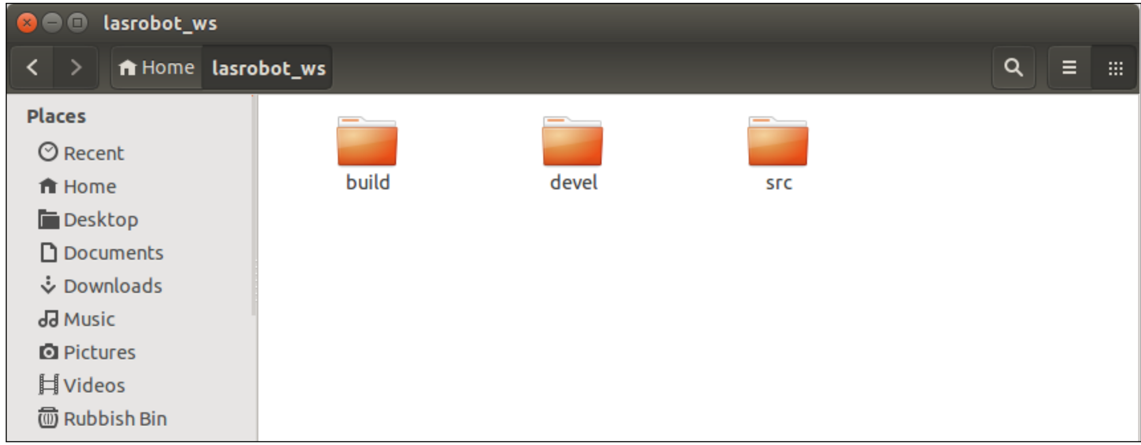


Figure 9: The workspace after building.

The last thing we did was creating a bash-file which makes it easier to source the setup.bash file in the devel folder. This file contains following code:

```
#!/bin/bash
source ~/lasrobot_ws/devel/setup.bash
```

3.3 Robot modelling files

The robot model with all necessary utilities will be created inside a folder named 'kuka_description' which is situated in the src-folder of the workspace (figure: 10).

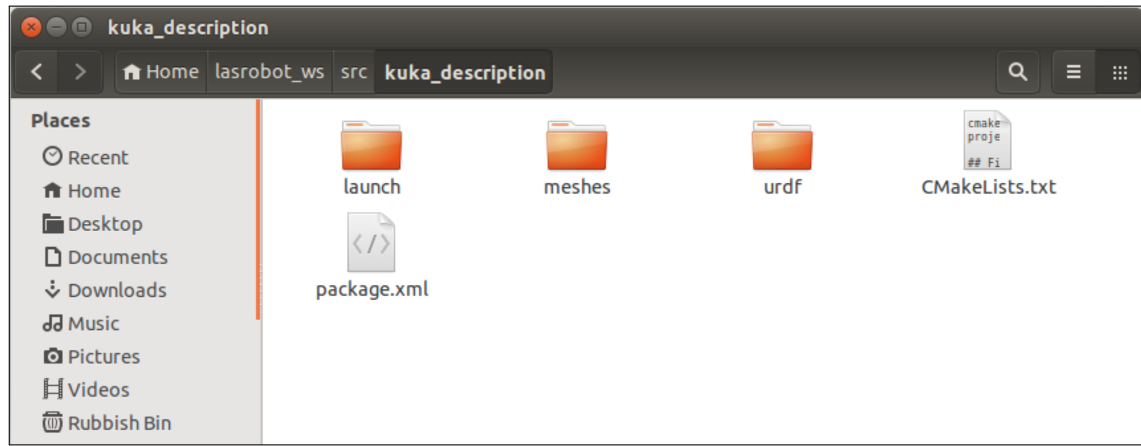


Figure 10: The folder which will contain the robot model ('kuka_description').

This folder contains a folder for the launch-file, the meshes used for the robot model and the URDF-file.

3.4 Chapter summary

In this chapter, the setup of the environment, and installing the Descartes package has been described. It was shown how the necessary executable has to be downloaded, which will run a basic trajectory planning, using a preexisting robot model. Finally, the location of the robot model files was shown.

4 Robot modelling

In this chapter a step-by-step procedure to create the *robot model* will be provided. First of all an overview and explanation of the URDF will be given. Then we look at how the launch-file, needed to launch the URDF-file in RViz, is made. After that, the robot model will be visualized in RViz to get a 3D representation of the model. The robot model is then made ready to use with the Descartes-software package by running the MoveIt! setup assistant. Finally, the robot model is integrated in Descartes.

4.1 URDF

In the URDF-file the visual properties of the links and the physical properties of the joints of the robot need to be defined. Further in this section, an example for defining a link and a joint of the robot is given. The complete URDF can be found in appendix A.
The URDF model is based on the Kuka KR5 ARC robot, available at ACRO.

4.1.1 Links

The description of a link can be split into the following parts [13]:

1. An inertial aspect: this part determines the inertial aspects of the link. This aspect is not necessary to to create a valid robot model an has thus not being incorporated because this information has not being found.

2. A visual aspect: the visual aspect will determine how the link will be depicted. This aspect is also optional but is necessary to gain a visual robot model.

3. A collision aspect: the collisional aspect will determine the dimensions of the link that will be used for collision detection.

In the visual and collision aspect the origin and geometry of the link needs to be defined. To define the geometry, a mesh is used that needs to be scaled to the required dimensions. The following code gives an example of the description of a link. Optionally, the color of the mesh can be defined.

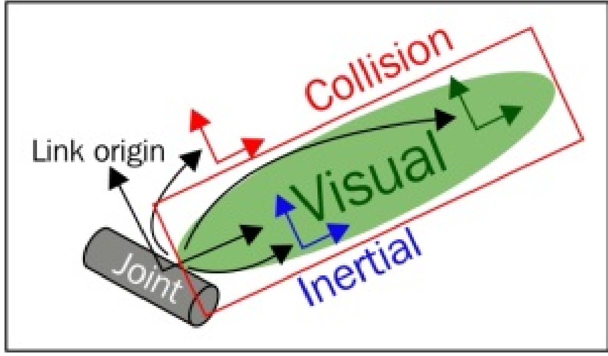


Figure 11: Principle of describing a link in the URDF-file [13].

```
<link name="link1">
  <visual>
      <origin xyz="-0.0014 -0.003 0.018" rpy="0 0 3.141529" />
      <geometry>
         <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Link1c.stl"
      scale=".004 .004 .004"/>
        </geometry>
    <material name ="orange"/>
  </visual>
  <collision>
      <origin xyz="-0.0014 -0.003 0.018" rpy="0 0 3.141529" />
      <geometry>
         <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Link1c.stl"
      scale=".004 .004 .004"/>
        </geometry>
  </collision>
</link>
```

4.1.2 Joints

The first thing to do to define a joint is to choose a joint type. The following types can be selected: (also available in [28].)

- revolute: a hinge joint that rotates along the axis and has a limited range specified by the upper and lower limits.
- continuous: a continuous hinge joint that rotates around the axis and has no upper and lower limits
- prismatic: a sliding joint that slides along the axis, and has a limited range specified by the upper and lower limits.
- fixed - This is not really a joint because it cannot move. All degrees of freedom are locked. This type of joint does not require the axis, calibration, dynamics, limits or safety_controller.
- floating: This joint allows motion for all 6 degrees of freedom.
- planar: This joint allows motion in a plane perpendicular to the axis.

In our robot model only revolute-joints will be used.
After declaring the joint type following elements need to be defined. This is illustrated in figure 12.

- parent link:The name of the link that is the parent of this link in the robot tree structure.
- child link: The name of the link that is the child link.
- origin xyz: This is the transform from the parent link to the child link. The joint is located at the origin of the child link, as shown in the figure above. Represents the x,y,z offset.
- origin rpy: Represents the rotation around fixed axis: first roll around x, then pitch around y and finally yaw around z. All angles are specified in radians.

- axis xyz: The joint axis specified in the joint frame. This is the axis of rotation for revolute joints, the axis of translation for prismatic joints, and the surface normal for planar joints. The axis is specified in the joint frame of reference. Fixed and floating joints do not use the axis field. Represents the x,y,z components of a vector. The vector should be normalized.
- limit lower:An attribute specifying the lower joint limit (radians for revolute joints, meters for prismatic joints). Omit if joint is continuous.
- limit upper: An attribute specifying the upper joint limit (radians for revolute joints, meters for prismatic joints). Omit if joint is continuous.
- limit effort: An attribute for enforcing the maximum joint effort.
- limit velocity: An attribute for enforcing the maximum joint velocity.

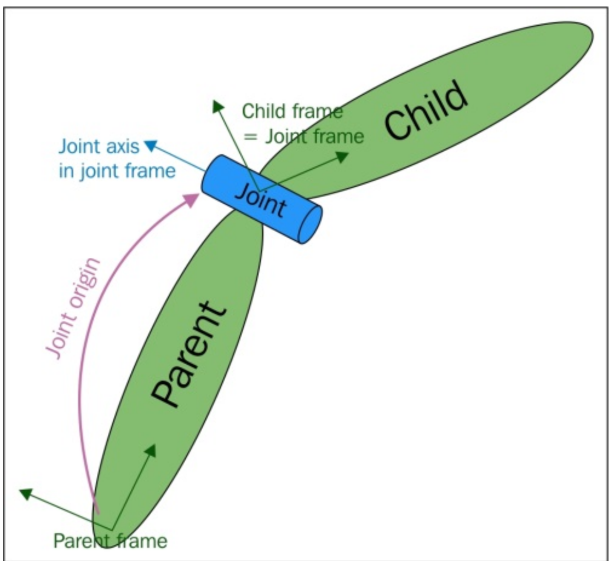


Figure 12: Principle of describing an joint in the URDF-file.[13].

```
<joint name="joint_1" type="revolute">
  <parent link="base_link"/>
  <child link="link1"/>
  <origin xyz="0 0 0.225" rpy="0 0 0" />
    <axis xyz="0 0 -1" />
    <limit lower="-2.70526034" upper="2.70526034" effort="0" velocity="0" />
  <material name ="orange"/>
</joint>
```

4.1.3 Meshes

The meshes of the robot links were provided by ACRO and have a basic STL file extension. In figure 13 all the meshes are shown. Unfortunately, these meshes are not fully complete. Creating an robot model with these meshes results in open spaces between the links of the robot. To solve this problem the base link, and link 1 have been altered. This is visible in figure 14 and figure 15. For this reason, the meshes called "baseb" and "link1b" are included in the URDF.

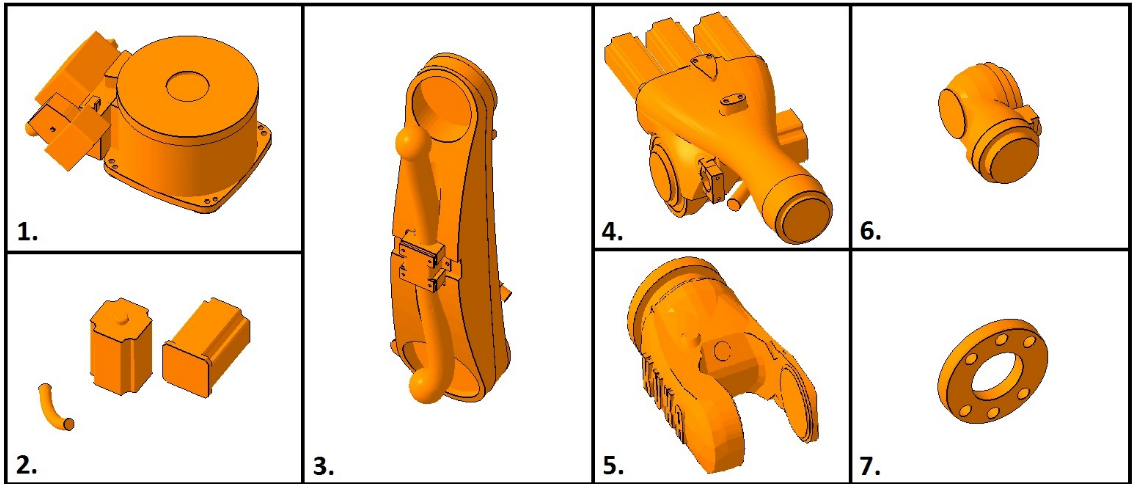


Figure 13: All original meshes of the robot model.

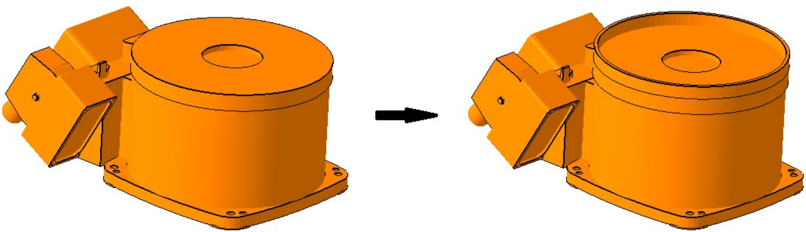


Figure 14: The adjustment of the mesh of the robot base.

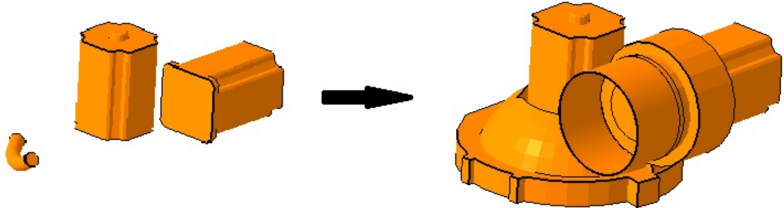


Figure 15: The adjustment of the mesh of the first link of the robot model.

4.1.4 Checking the URDF

After creating the URDF, it can be checked to test whether it is valid or not. This can be done using the following command:

```
$check_urdf robot.urdf
```

Now that the URDF is parsed, it can be visualised in a graph using the "urdf_to_graph" command. The result is written to a pdf file. This pdf can be found in appendix B.

```
$urdf_to_graph robot.urdf
$evince robot.pdf
```

4.2 URDF-launchfile

By running the launch file, multiple nodes can be started at once. The following launch file is used to open the urdf-model in RViz, and activate the joint_state_publisher and the robot_state_publisher. By activating these nodes, and enabling the GUI of the joint_state_publisher, the robot's joint state defined in the GUI is visualized in RViz.

```xml
<?xml version="1.0"?>

<launch>
  <arg name="model" />
  <arg name="gui" default="true" />
  <param name="robot_description" textfile="$(find kuka_description)/urdf/robot.urdf" />
  <param name="use_gui" value="$(arg gui)" />
  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" required="true" />
</launch>
```

4.3 Visualisation in RViz

Using the following command, the robot model can be visualised in RViz:

```
$roslaunch kuka_description display.launch
```

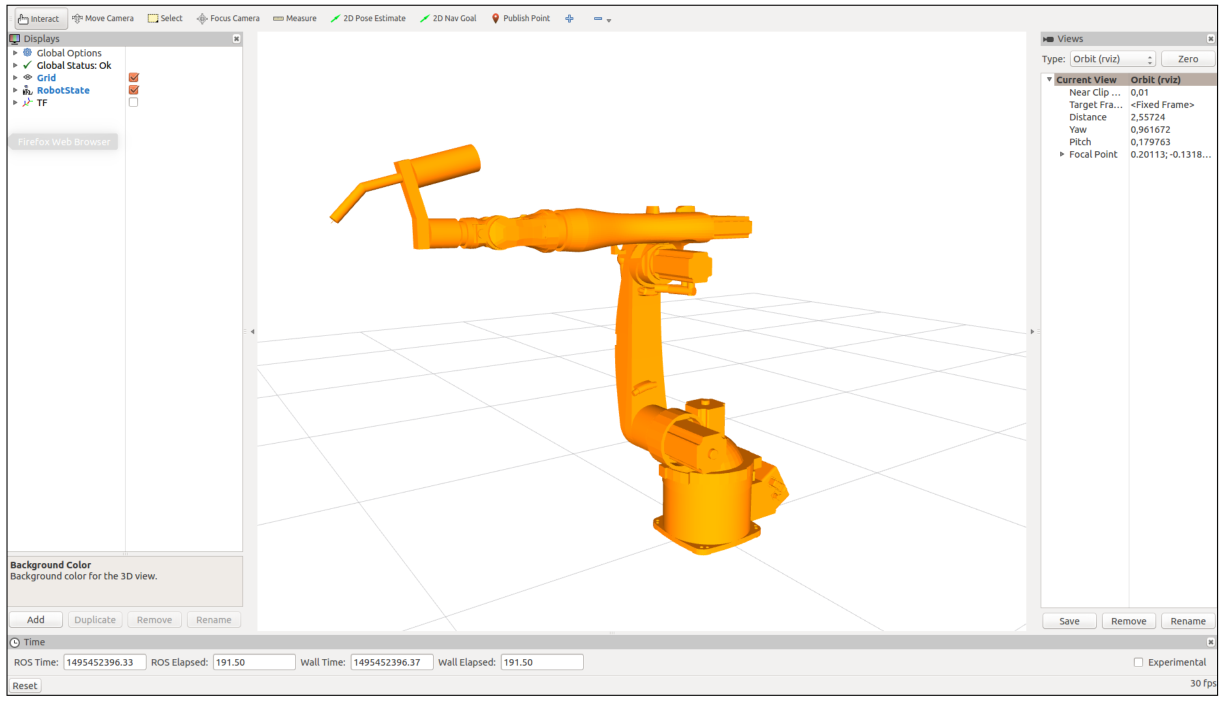


Figure 16: The created robot model visualised in RViz.

4.4 MoveIt! set-up assistant

The MoveIt! setup Assistant is a graphical user interface to configure any robot to be used in MoveIt!. This tool generates SRDF files, configuarion files, launch files and scripts, based on the robot URDF model. These files are required to configure the move_group node integrated in MoveIt!. The SRDF file contains details about the arm joints, end effector, virtual joints, and also the collision link pairs which are configured during the MoveIt! configuration process. The configuration file contains details about the kinematic solvers, joint limits and controllers. These are also configured and saved during the configuration process. Using the generated package of the robot, motion planning in RVIz can be done without the presence of a real robot or other simulation interface[13]. The procedure consists of the following steps:

Step 1: launching the setup assistant tool and loading the URDF-file

This command will start the MoveIt! Setup Assistant tool:

```
$roslaunch moveit_setup_assistant setup_assistant.launch
```

When executed, a GUI as illustrated in figure 17 is launched. After loading the URDF file, the file will automatically be parsed and a 3D representation will be drawn on the right side, as shown in figure 18.

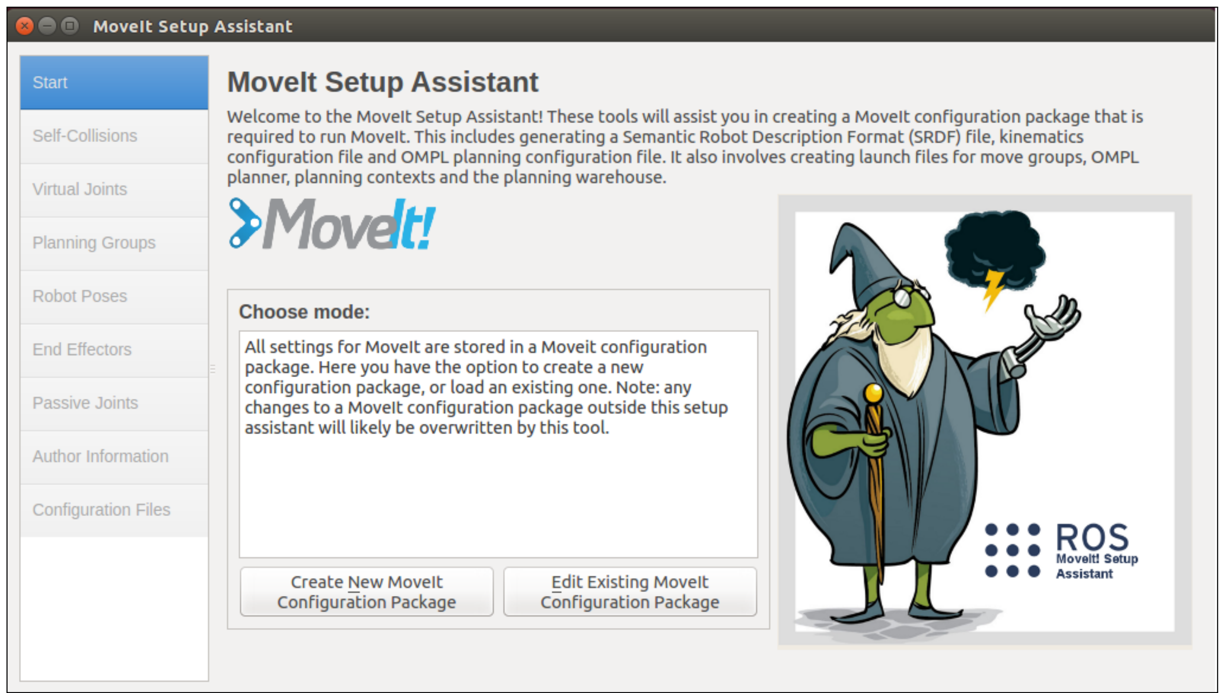


Figure 17: Step1a: Start window of the MoveIt! Setup Assistant.

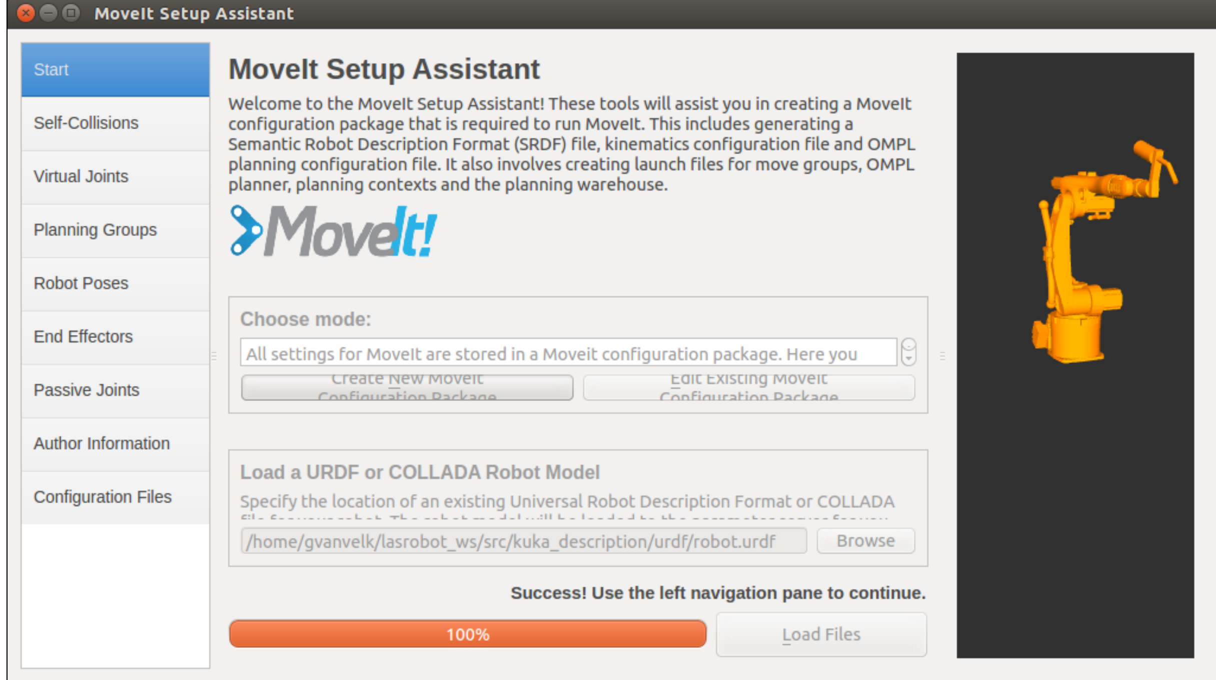


Figure 18: Step1b: Opening and parsing the URDF-model.

Step 2: generating the self-collision matrix

In this step, MoveIt! searches for a pair of links on the robot which can be safely removed from the collision checking, if, for example, it is impossible that they ever touch. These can reduce the collision checking time. This tool analyzes each link pair and categorizes the links as "always in collision", "never in collision", "default in collision", "adjacent links", "disabled",

and "sometimes in collision". It then disables collision checking for any pair of links that make any kind of collision. The sampling density is the number of random positions to check for self-collision. The default value is 10,000. We can see the disabled pairs of links by pressing the Regenerate Default Collision Matrix button, as shown in figure 19.

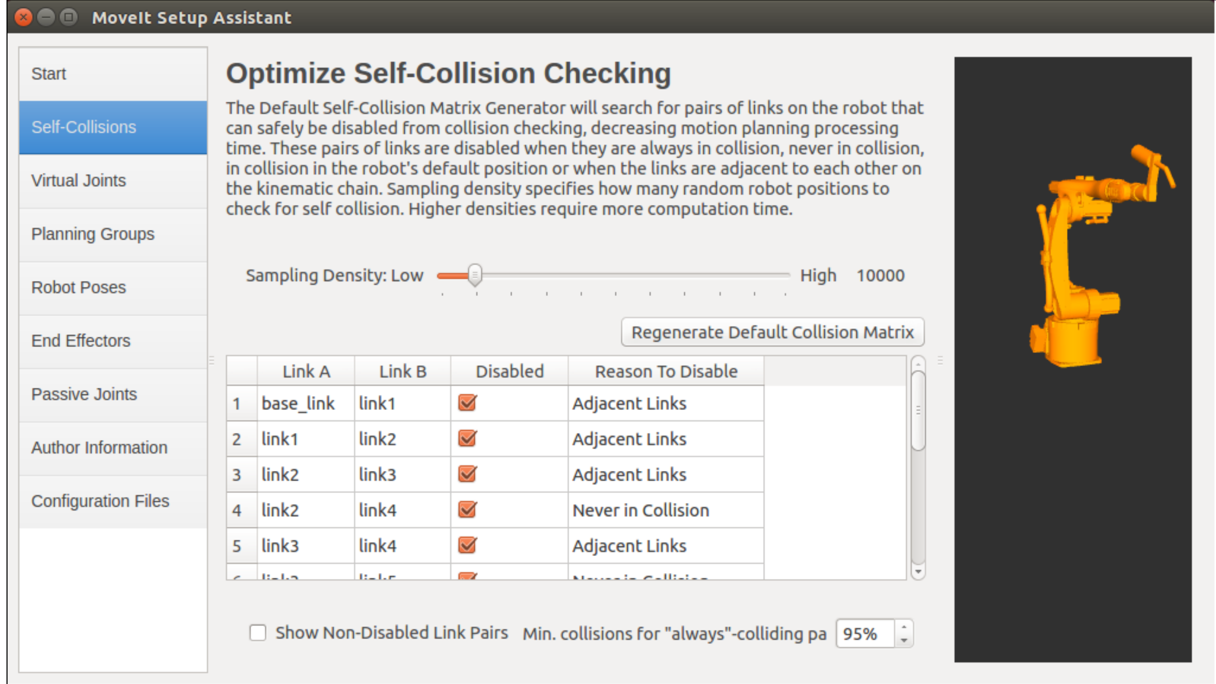


Figure 19: step2: Self-Collison Checking

Step 3: adding virtual joints

Virtual joints can be used to move the complete robot around in the world. It is not necessary for a static robot which doesn't move.

Step 4: defining planning groups

A planning group is basically a group of joints and links in a robotic arm, which is planned in order to achieve a goal position of a link, or the end effector. In this step, the planning group "robotarm" is created, which consists of all the joints and links of the robot. (Shown in figure 20.) Another thing that has to be done in this step is selecting a Kinematic Solver. In this case the "kdl_kinematics_plugin/KDLKinematicsPlugin" was chosen, as shown in figure 21.

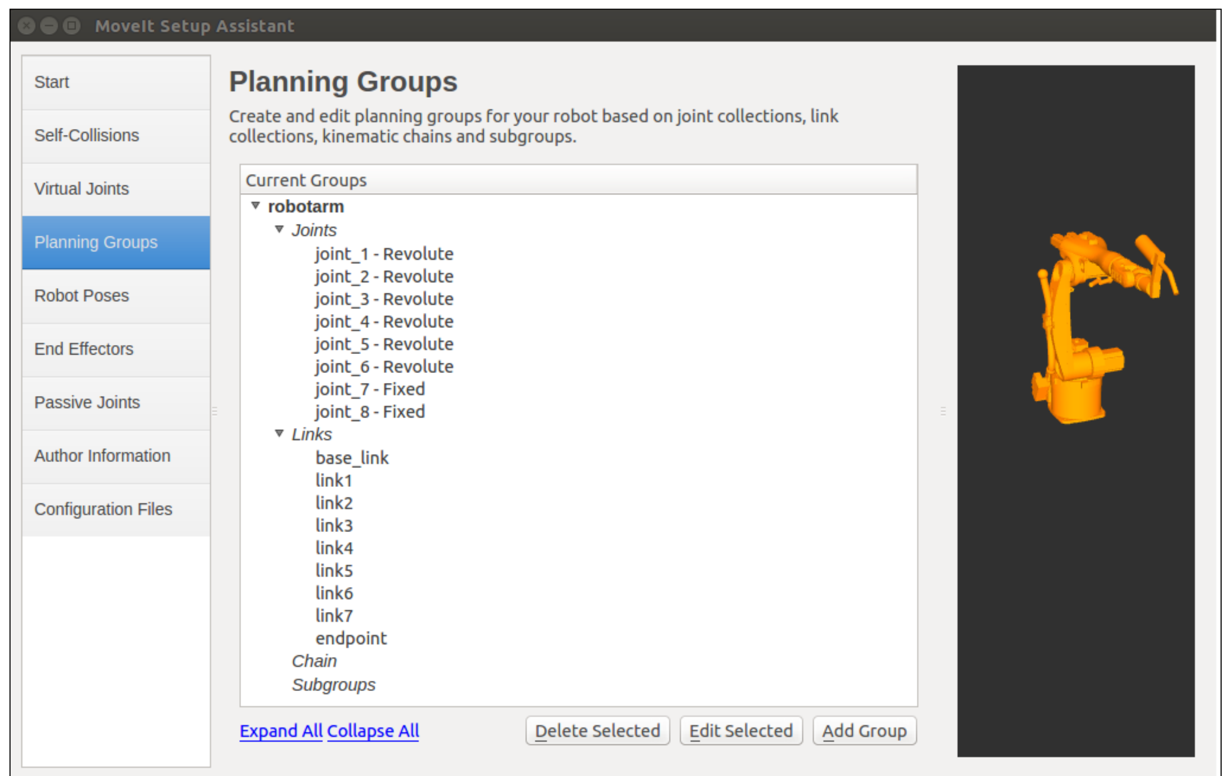


Figure 20: step4a: Defining a planning group

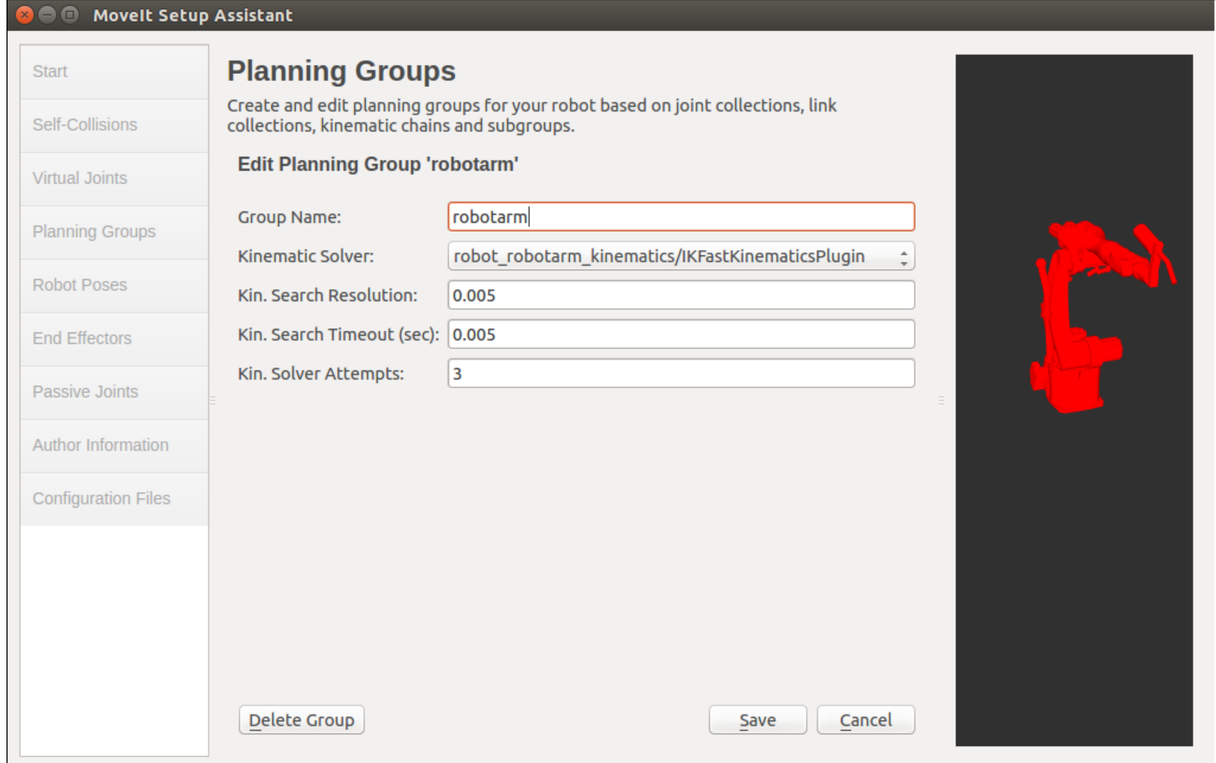


Figure 21: step4b: Adding the planning group

Step 5: adding the robot poses

In this step, certain fixed poses of the robot can be defined. For example, a home position of the robot. (Shown in figure 22.)

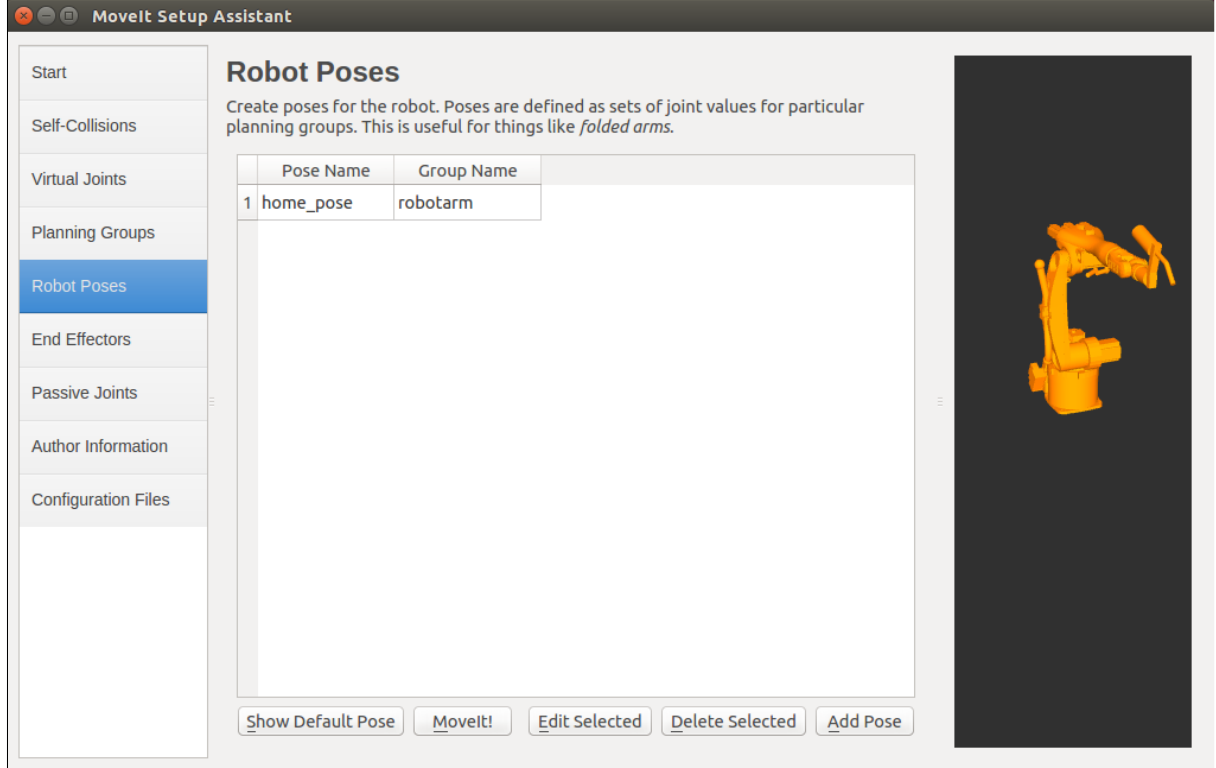


Figure 22: step5: Adding robot poses

Step 6: setup the robot end effector

In this step, we name the robot's end effector, and assign the end effector group, the parent link, and the parent group. This step is not necessary.

Step 7: adding passive joints

In this step, we can specify the passive joints in the robot. Passive joints mean that the joints do not have any actuators. The used robot does not contain any passive joints so this step is skipped.

Step 8: generating configuration files

In this step, the tool generates a package, which contains the files that MoveIt! needs.The name of the package has to be defined. In this case, the package is called "robot_moveit_config" and is placed in the src folder of the workspace.

After completing the setup assistant, the robot file can be launched in RViz using following

command. Figure 23 shows the result.

```
$roslaunch robot_moveit_config descartes_pathplanning.launch
```

We are using a changed version of the launch file, so that the "joint_trajectory_action" server, necessary to execute the generated trajectory, is ran. It contains the following additions:

```
<!- joint_trajectory_action: provides actionlib interface for high-level robot control -->
<node pkg="industrial_robot_client" type="joint_trajectory_action" name="joint_trajectory_action
    "/>
```

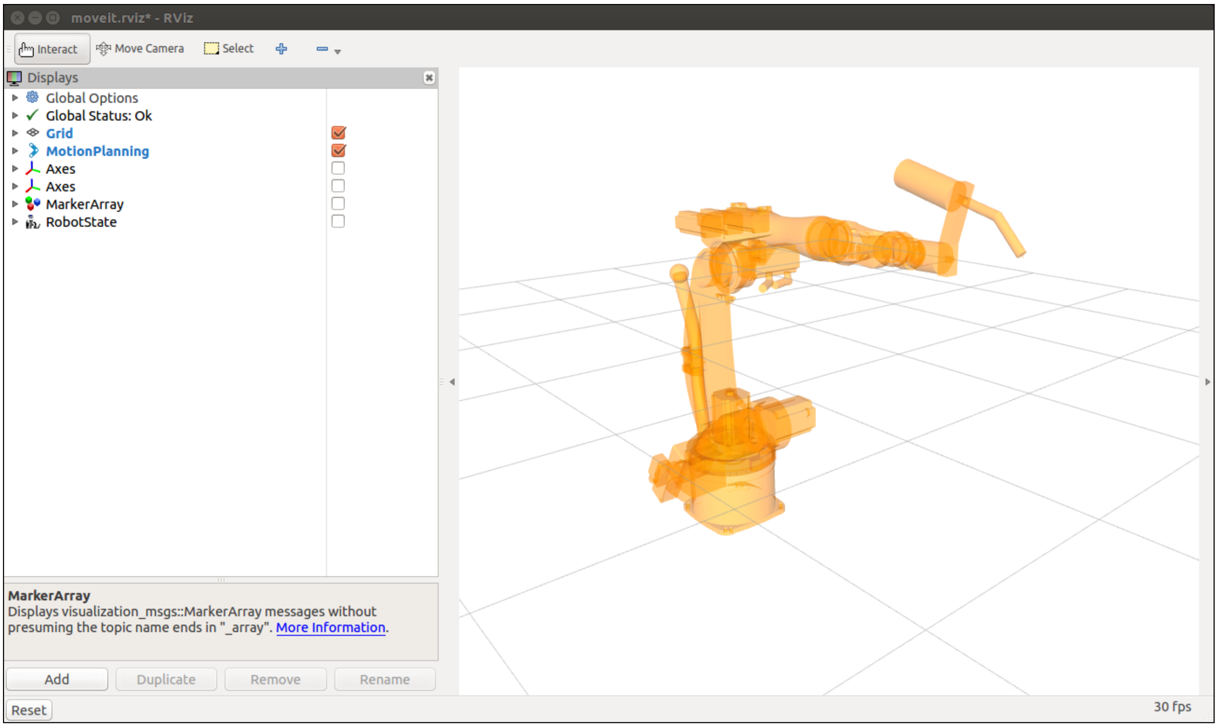


Figure 23: Resulting robot model displayed in RViz

4.5 Integrating the robot model in Descartes

To make sure that Descartes is able to use the new robot model, certain names in the executable (robot.cpp) have to be changed. The lines that need to be checked are lines 305-316 and are displayed below.

```
// Name of description on parameter server. Typically just "robot_description".//
const std::string robot_description = "robot_description";

// name of the kinematic group you defined when running MoveitSetupAssistant.//
const std::string group_name = "robotarm";

// Name of frame in which you are expressing poses. Typically "world_frame" or "base_link".//
const std::string world_frame = "base_link";

// tool center point frame (name of link associated with tool).//
const std::string tcp_frame = "endpoint";
```

4.6 Chapter summary

In this chapter, an overview is given of the process to create a new URDF-model. First, the structure and content of a URDF was described. After that, the necessary launch file was shown, used to visualise the URDF in RViz. Afterwards, a step-by-step walkthrough of the MoveIt! Setup Assistant was provided, in order to configure the robot to be used by MoveIt!. Finally, the necessary adjustments in the Descartes-software were explained, which required to use the robot model for the simulations.

5 The Descartes Software Package

In this chapter an overview of the Descartes software package is given. First, the different types of trajectory points that the software uses, are discussed. Then we delve deeper into the path generation itself. More specifically: how are the different trajectory points used to construct a searchable graph? We discuss how the different trajectories are visualized using RViz. And finally we go into how the generated data were stored in files, which could then be used to visualize the data in graphs. In the coming sections, the "Descartes software package" will be referred to as "Descartes".

5.1 Trajectory points

To generate a trajectory, Descartes expects multiple so-called *trajectory points* as input. Trajectory points are individual points along a path, that is to be followed by the end-effector of the robot. In effect, the Descartes package samples the desired path in discrete steps.

Many industrial processes require an exact positioning of a tool in function of a workpiece, or simply a defined position in 3D space itself. (This would then be a 6 dimensional pose, consisting of 3 translation and 3 rotation components.) Robot welding is certainly one of the processes in which the positions of the tool (in this case a welding torch) can be derived relatively easily from a predetermined welding path. If the tool is mounted on a robot arm however, often a fully defined positioning of the tool does not completely define the position of the robotic arm. There might be multiple differing poses for the robot arm that would put the tool in a certain position. For this reason, these kind of trajectory points are called *partially-constrained* trajectory points, since they only *partially constrain* the positioning of the robot.

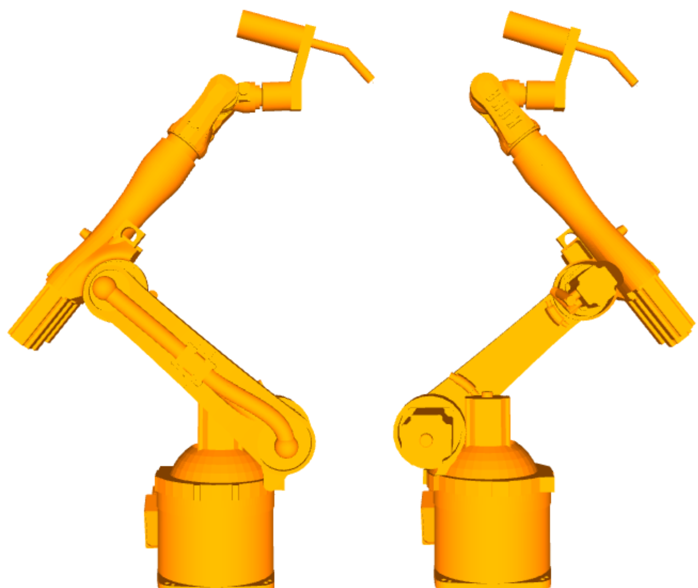


Figure 24: Different joint solutions for a single tool pose.

The end effector of the robot has a local frame, called the *tool frame*, as shown in figure 26. All

the so-called frames can be visualized as 3D coordinate frames. The x-axis is colored red, the y-axis is colored green and the z-axis is colored blue. The location of the tool frame can be calculated using the forward kinematics of the robot. This means that if we know all the joint positions of the robot, we can transform the base frame using the transform of every individual robot link. The end result is the tool frame. This is illustrated in figure 27.
When defining a trajectory point in Descartes, we actually just define a new frame in space. Let's simply call this frame the *goal frame*. Descartes then tries to place the *tool frame* into the *goal frame* by moving the robot into a certain pose, as shown in figure 25. (Multiple or no poses may be possible, depending on the number of IK-solutions.)

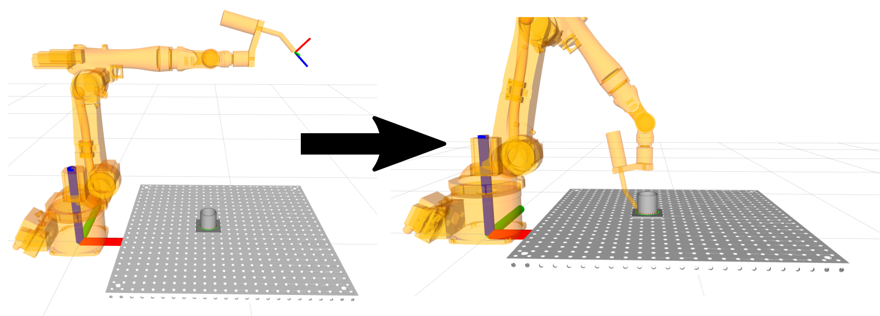


Figure 25: Left: robot with base frame and tool frame. Right: tool frame moved into the trajectory point frame.

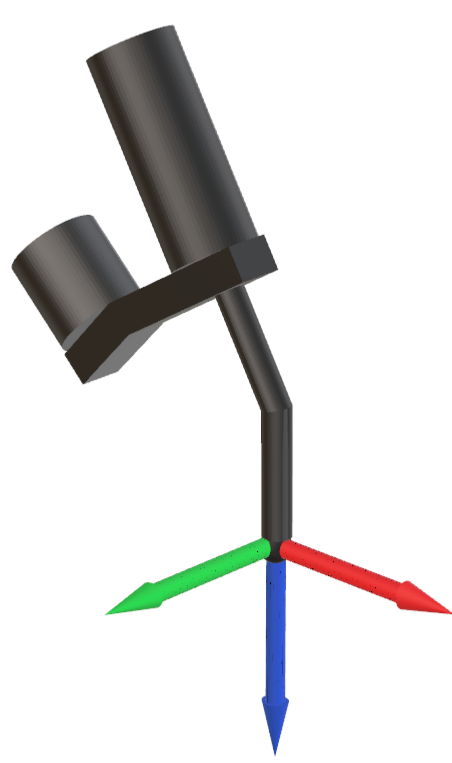


Figure 26: The welding torch with its frame. (Tool frame.)

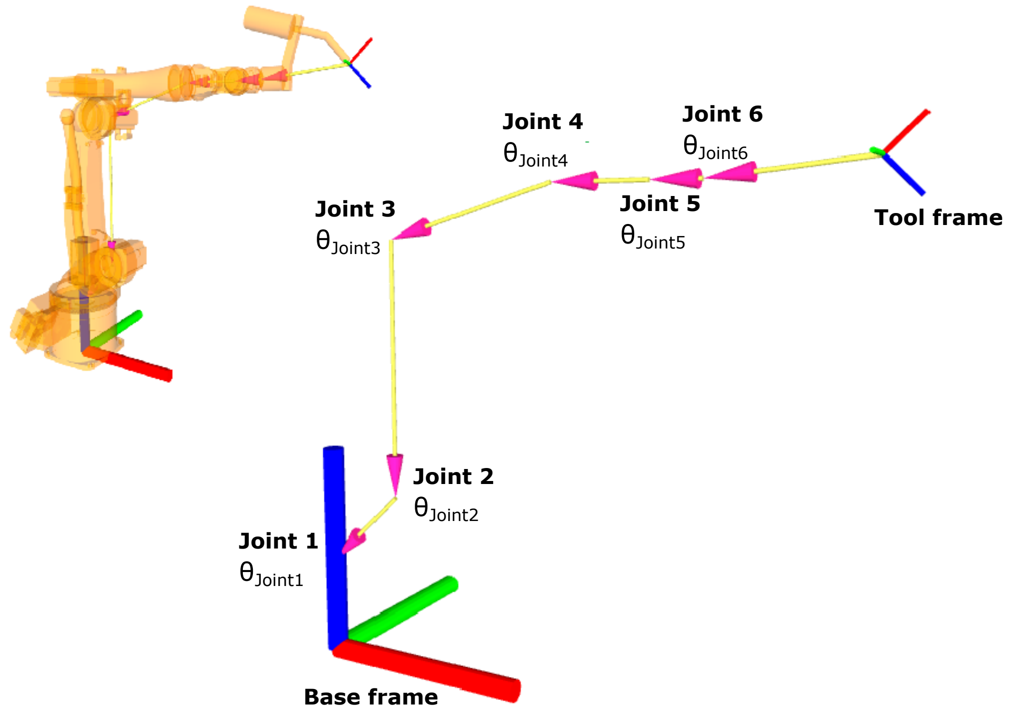


Figure 27: Forward kinematics.

As of now, Descartes allows the user to define three different kinds of trajectory points. First, and most importantly, of all, partially constrained trajectory points can be defined using the so called *Cartesian trajectory point* class. These Cartesian trajectory points represent an arbitrary robot pose, where the tool tip is in a defined six dimensional point. This point is defined by a transformation matrix, which contains the necessary rotations and translations to transform an arbitrary base frame into the goal frame.

Descartes also allows users to define so-called *joint trajectory points*, in which every single joint of the robot is defined. Unlike the *Cartesian* trajectory points, this is a fully constrained robot pose. The tool frame is then fully defined using the robot's forward kinematics.

The last type of trajectory points are called *Axial-symmetric points*. It is actually a sub class of the Cartesian trajectory points class. The only difference lies in the fact that using this type of trajectory point, a rotation tolerance is defined around an arbitrary axis of the transform frame. This is especially handy for processes where the tool has a symmetric shape, or the rotation of the tool around one axis does not have an influence on the process.

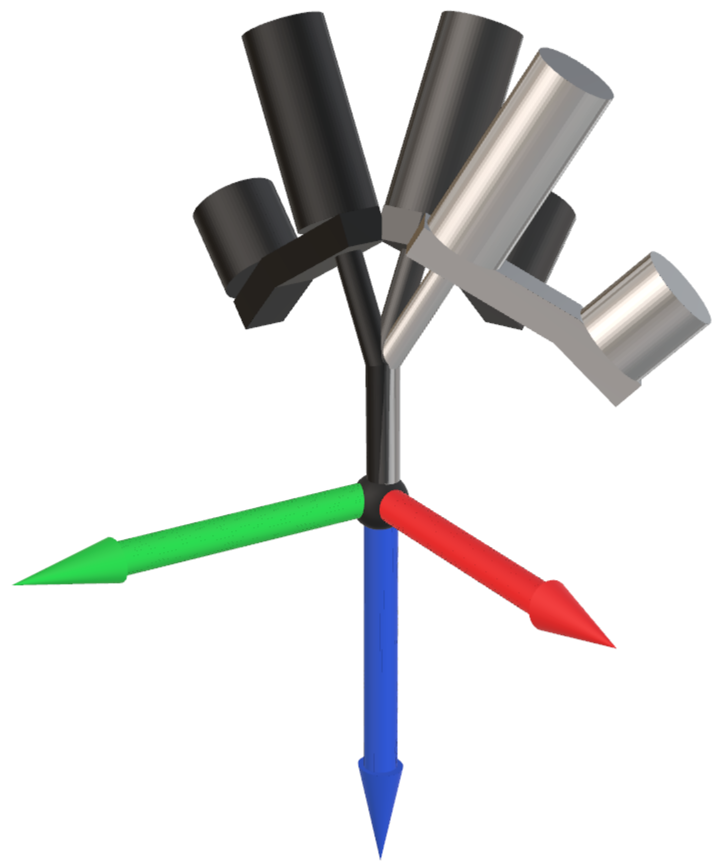


Figure 28: Three possible poses for the welding torch when using an axial-symmetric point.

Descartes also allows to define different tolerances using these types of trajectory points. In the case of the *Cartesian* trajectory points, tolerances can be defined around every coordinate of the points. Since the points are defined by a 6 dimensional transform frame, 6 dimensions can be toleranced: three translations, and three rotations. In the case of *Joint* trajectory points, tolerances can also be defined for every joint angle. This translates into an upper and lower limit for every joint angle. A more detailed explanation of tolerances follows in section 5.6.

The *Axial symmetric* points are actually a special kind of Cartesian point with a predetermined rotational tolerance around one axis. It is impossible to define any extra tolerances when using this type of trajectory points.

5.2 Transformation matrices and different possible transforms

Descartes allows trajectory points to be defined in multiple different local frames. For example, one may define a workpiece, with a certain origin in space. It is then possible to define the trajectory points in function of this workpiece frame. This can be handy, in case the workpiece is rotated, or otherwise moved around. The trajectory points will then move along with it.

Points can also be defined local to the tool frame, and the tool's base frame.

Although this can certainly be handy, in this thesis it was chosen to always define trajectory points in function of the robot base frame, which in our case is also the world frame. This

allows for easier debugging, and helped us start off when we were not yet familiar with the concepts of these different frames.

It also makes the code more compact and easier to understand.

5.3 Inverse Kinematics

Suppose now that we have a sequence of defined tool positions that we need to move through to place a weld on a workpiece. Since the tool positions itself do not necessarily define the complete positioning of the robot, the different pose solutions for the robot first need to be generated by using a certain *Inverse Kinematic solver*. This can result in multiple joint solutions for any unique trajectory point. The used IK-solvers are actually not a part of the Descartes software package, and are thus only mentioned here. In this thesis two IK solvers were used:

- KDL solver
- IKfast solver

The KDL solver uses a numerical approach to generate IK solutions. The IKfast solver searches for IK solutions analytically. The IKfast solver is about 100 times faster than the KDL solver. Furthermore, it generates a greater number of more accurate solutions.

5.4 Graph building

To generate a final trajectory that goes through all of the trajectory points, a choice needs to be made in regard to the different *Inverse Kinematic Solutions*. For every trajectory point one of its solutions needs to be chosen in order to generate a complete path. To be able to make this choice, Descartes generates a graph.
This graph consists of the different joint solutions for every trajectory point (vertices). These joint solutions are connected to every joint solution of the next trajectory point with an edge. The edges have a so-called *weight* that can be calculated in function of the two joint solutions they connect. This weight is a real number calculated by the *cost function*.
The graph is then searched by an algorithm that tries to find a path through the graph with the lowest cost. That is, the trajectory with the smallest sum of the edge weights it traverses.

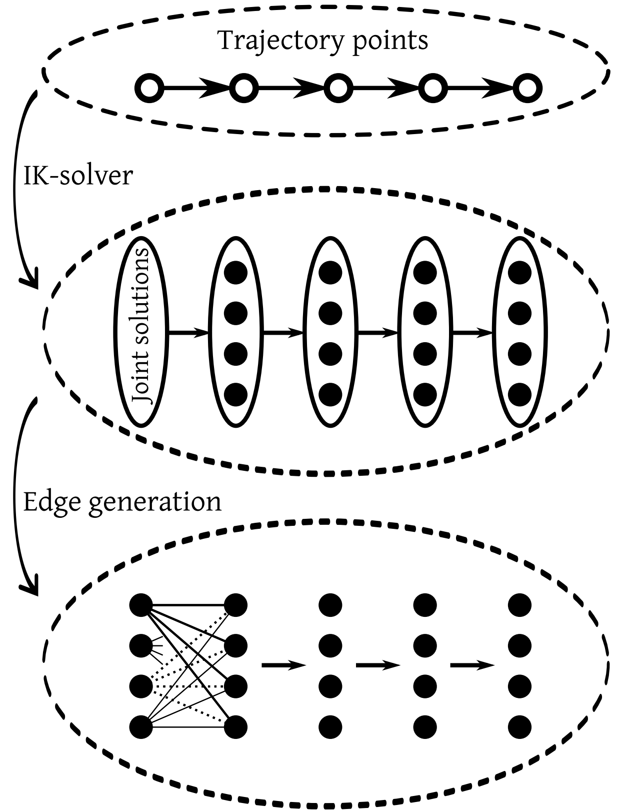


Figure 29: Steps to construct a graph from trajectory points without tolerances.

In figure 29 the main steps of graph building are shown. However, one step is not included in this figure, being the generation of the different *toleranced* trajectory points. As explained in section 5.1, different tolerances can be defined when using certain kinds of trajectory points. Descartes probes these tolerance intervals by stepping through them in discrete steps. The step size used by Descartes is called the *discretization step size*. By stepping through a tolerance window, Descartes generates multiple new trajectory points, which now are completely defined Cartesian trajectory points. To distinguish them from the original trajectory points, used to define the trajectory before path planning started, these points are called *toleranced* trajectory points.

Figure 30 shows the toleranced frames that are generated from a trajectory point with a symmetric tolerance of about 40 degrees around its x-axis, and a discretization step size of 10 degrees.

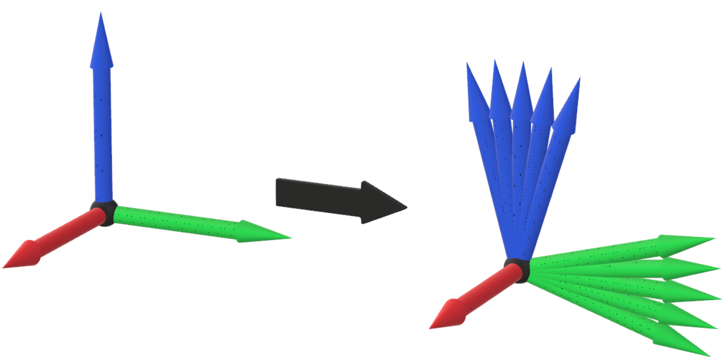


Figure 30: Different toleranced frames.

These toleranced frames (or toleranced trajectory points) all belong to a single trajectory point defined by the user. The IK-solver will then try to find IK-solutions for every toleranced frame that has been generated, just like a regular trajectory point. When the graph is built, each toleranced frame is connected with an edge to every toleranced frame of the following trajectory point. This is illustrated in figure 31.

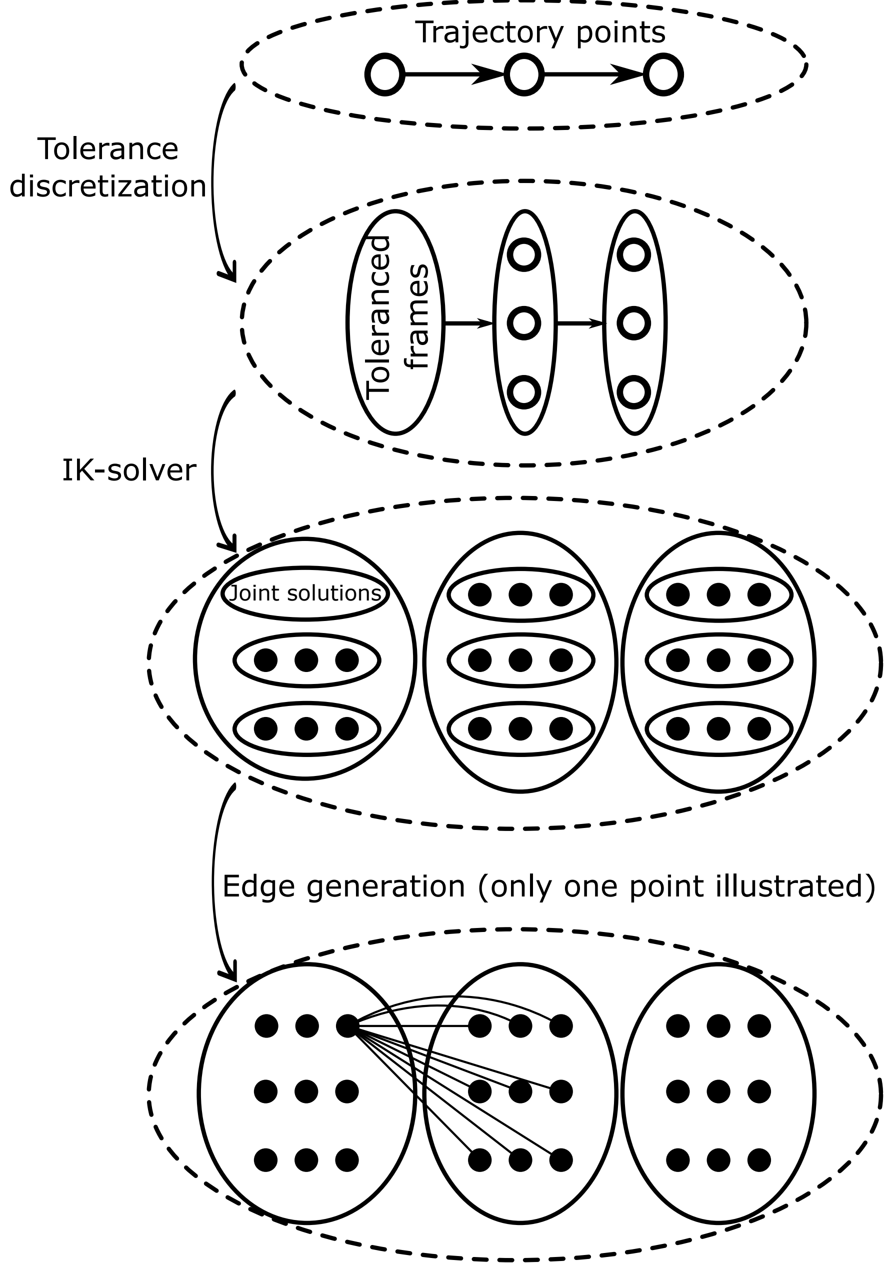


Figure 31: Steps to construct a graph from trajectory points, including tolerances.

So, tolerances on different rotations and translations can be constructed by defining the necessary tolerance intervals. These tolerance intervals contain the upper and lower bounds of the corresponding rotation or translation. What is currently not possible however, is defining unique *discretization step sizes* for different rotations. This means that a single, global, discretization step size is used for all rotations, or all translations in a single Cartesian trajectory point. In our application, this leads to the problem that some some large tolerances which should not be sampled very accurately, are still sampled using the same step size.

5.5 Trajectory visualization in RViz

As mentioned before, the cartesian trajectory points, that Descartes uses to execute its path planning, are internally represented by transformation matrices. Because it is quite hard to visualize all of the trajectory points mentally, based on transformation matrices, a visualization step was required. So one of the first steps in this thesis was writing a small library used to visualize the different trajectory points that are fed into Descartes. Thanks to the ROS framework, and the flexibility of RViz, this was not a hard feat.

The ROS framework allows us to easily send packets of information, called *messages*, between different running programs. In this case, the program running the Descartes software (we will call this the *Descartes node*) had to send information to the RViz node, so that it could visualize the necessary points. RViz already had a message type defined, called *MarkerArray*, which contains data about the markers it has to visualize.

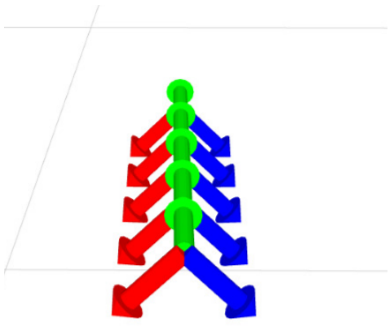


Figure 32: Visualized frames in RViz, every frame consists of three different markers.

The only real "problem" was to extract the necessary information, that was required to build up the *MarkerArray* message, from the transformation matrices used to define the Cartesian trajectory points. By using the *Eigen* library we could transform the transformation matrices into seperate translations and a quaternion defining the rotation, which are required to define the positioning of a marker in RViz.

Now that we could generate visual markers in RViz, it first was decided to only visualize a transform of the base frame's Z-axis. This means, that a single Z-axis was visualized, in the desired position of the tool frame. The reason for this was that we would always use trajectory points with a rotational tolerance around its Z-axis. In the case of robot welding, the rotation of the welding torch has no effect on weld quality, and to allow a more flexible path generation this tolerance is quite essential.

It soon became clear however, that it was much better to visualize the full transform of the base frame. This means that all of the axes were transformed and visualized. This was a much better representation of the transformation matrices used by the trajectory points, because it clearly showed all the different coordinates used to define the matrices. This also helped us to better understand the exact way that Descartes expected us to generate trajectory points, using XYZ-translations and XYZ Euler-rotations. Figure 32 shows a couple of frames visualized in RViz.

5.6 Tolerances on local frame vs. Euler angle tolerances

It was already mentioned that *Descartes* allows *tolerances* to be defined for trajectory points. By *tolerance* it is meant that a certain coordinate of a trajectory point will be tolerated between an upper and lower limit. With the exception of joint trajectory points, the trajectory points in Descartes need 6 coordinates to be fully defined in 3D space: x, y, z translations, and three Euler rotations (α, β, γ) around arbitrary axes. Tolerances may be defined on all of these coordinates.

Suppose now that we want to define a trajectory point with a value of 10 as its x-translation-coordinate. We might define a translational tolerance on this coordinate, say a symmetric tolerance of 10. This would allow the trajectory point to have an x-coordinate between 5 and 15. The same applies for rotational tolerances.

When using Euler rotations, there exist many different conventions to define a complete rotation. Although Descartes allows to use different Euler conventions, most of the time the XYZ-Euler convention is used. This means that a complete 3D-rotation is defined by three rotations, one around the X-axis, one around the Y-axis and finally one around the Z-axis of the local frame, in this order. These rotations always start off with a base frame, according to which different rotations are executed. This base frame can be the absolute center frame of the used space (frame with 0,0,0 translations and 0,0,0 rotations) or a different pre-transformed frame as explained in section 5.2.

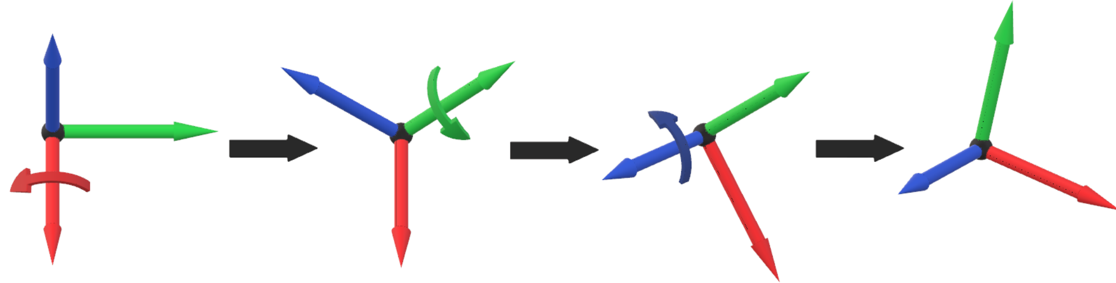


Figure 33: Visualized example of intrinsic XYZ Euler rotations.

The Euler rotations used by Descartes are *intrinsic*. This means that the frame which defines the different axes changes its orientation after every elemental rotation. A visualized example is given in figure 33. The frame on the left is the base frame. The two frames in the middle are the intermediary transforms. The frame on the right is the transformed frame.

Given a certain rotational pose in 3D space, it can be pretty hard to mentally visualize the different intrinsic XYZ Euler rotations necessary to get to this pose from the used base frame. Another disadvantage is that a change in, let's say, the Euler rotation around the y-axis, does not translate into a rotation around the y-axis of either the base frame or the transformed frame. Instead, it is a rotation around an intermediate frame's y-axis. When using the XYZ Euler convention, this intermediate frame would be the base frame, rotated around its X-axis. An advantage of intrinsic Euler angles however, is the fact the the last Euler rotation (rotation around the z-axis in the case of the XYZ convention) is a rotation around the local frame's z-axis. Since most of the time, the robot's tool frame has its z-axis pointing directly out of the

end effector, a tolerance on this last Euler angle can be handy in the case of a symmetric tool, or a process where the rotation of the tool around its z-axis doesn't matter. Since this is the case for robot welding, using intrinsic Euler rotations is, in some sense, a logical choice.

By using these tolerances, we can make the trajectory generation more flexible. In cases where no solutions can be found, tolerances may be used to allow certain deviations from the ideal trajectory. However, when allowing tolerances, the possible effects of the tolerances on the process have to be taken into account. In the case of robot welding, the welding torch needs to point in the general direction of the weld.

The first intuitive tolerance we may want to allow is a certain deviation of the welding torch from the optimal path by a few degrees. If the direction of this tolerance would not matter, this would ideally result in a cone-shaped tolerance, as shown in figure 34. This would mean that the z-axis of the robot's end effector would be allowed to in any position within the cone.

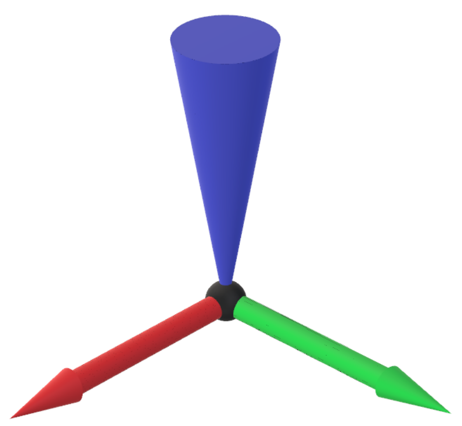


Figure 34: Conical tolerance zone for the z-axis.

Take into account, however, that the only possible tolerances we can define in the Descartes package, are tolerances on the different coordinates. For rotations, these would be tolerances defined on the different Euler rotation angles. The question now arises, if we want to allow such a cone shaped tolerance for the z-axis, how would the tolerances on the Euler angles have to be defined?

This is not an easy problem to solve, because the intuitive tolerance we want to allow is defined in the local (transformed) frame. This while tolerances on the different Euler angles, are defined in the intermediary transform frames. When defining trajectory points, together with their tolerances, the user intuitively wants the rotational tolerances to be defined in the transformed frame. This is because the user is able to see the transformed frame (which can be a trajectory point, and can be visualized) but is unable to see the intermediary transforms generated by using Euler rotations.

Above that, the proposed conical tolerance zone can only be approximated using Euler angle tolerances. Secondly, a given Euler angle solution for a 3D rotation is not necessarily unique. Different possible Euler rotations may exist for one spacial rotation. Descartes stores the Euler angles, given as inputs to define the trajectory points, in rotation matrices. When the toleranced frames are calculated, Descartes uses this rotation matrix to calculate the XYZ

Euler angles for a certain trajectory point. This solution for the Euler angles (which is internally used to apply the tolerances) can differ from the input which the user originally gave to define the trajectory point.

Suppose that we were able to define rotational tolerances around the local (transformed) frame's axes. We could then easily define rotational tolerances around the x- and the y-axis, leading to the tolerance zone shown in figure 35.

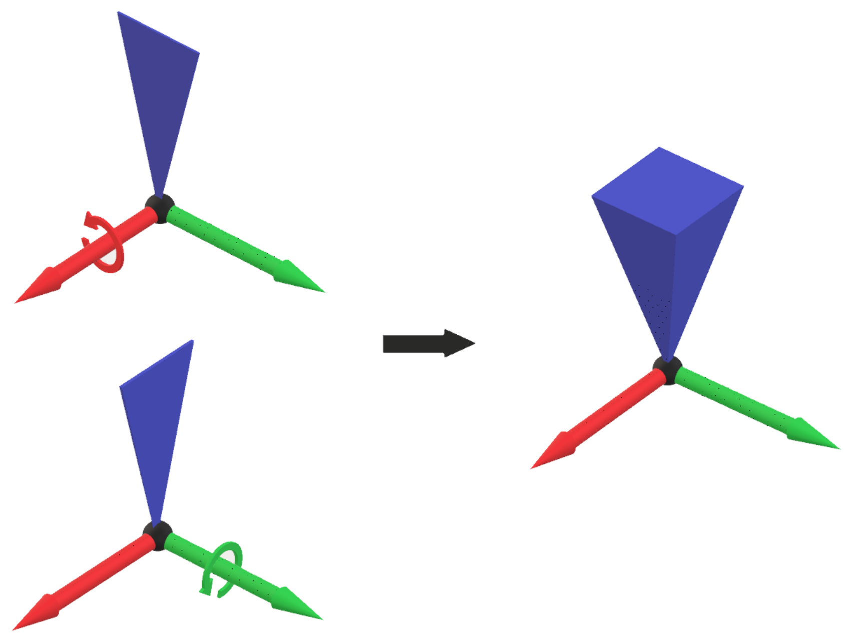


Figure 35: Pyramid-shaped tolerance zone for the z-axis.

This tolerance zone approximates the intuitively proposed conical tolerance zone (figure 34) well. On top of that, it allows to differentiate between different tolerance angles. In this case, the angles around the x- and the y-axis.

5.7 Used convention when defining trajectory points

To be able to allow correct tolerances on trajectory points, a certain convention had to be chosen to place the frames of the trajectory points. A smart placement of the trajectory point frames makes sure that a tolerance around one axis of the frame results in a modification of only one welding angle.
To explain this we first give an overview of the different welding angles used in this application. Figure 36 shows the two different welding angles, the *transverse angle* and the *push/drag angle*.

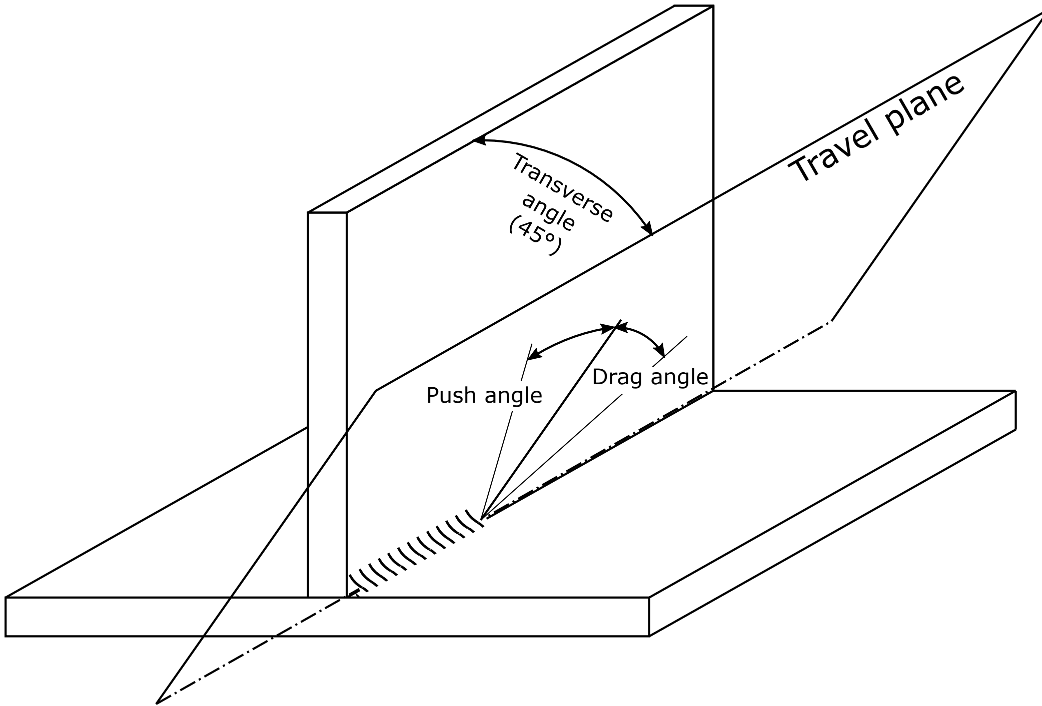


Figure 36: Different welding angles.

The transverse angle has a stronger effect on the weld quality than the push/drag angle. The push/drag angles mostly have an effect on the bead shape, as shown in figure 37.

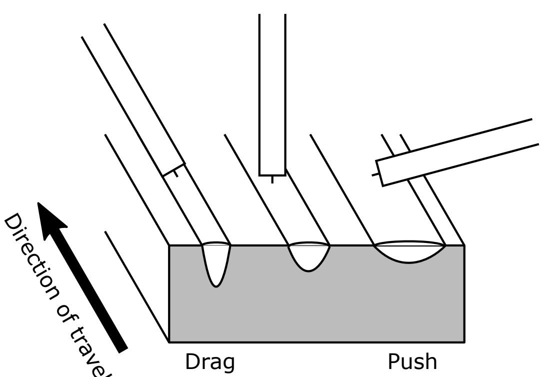


Figure 37: Bead shapes.

When changing the transverse angle too much from the ideal value, the bead will connect less well to one of the plates.

Since we want to allow certain transverse and push/drag angles using tolerances in Descartes, the trajectory point frames need to be placed in a certain pre-defined way. We chose to always aim the frame's y-axis in the direction of travel. While aiming the z-axis straight into the weld. This is illustrated in figure 38.

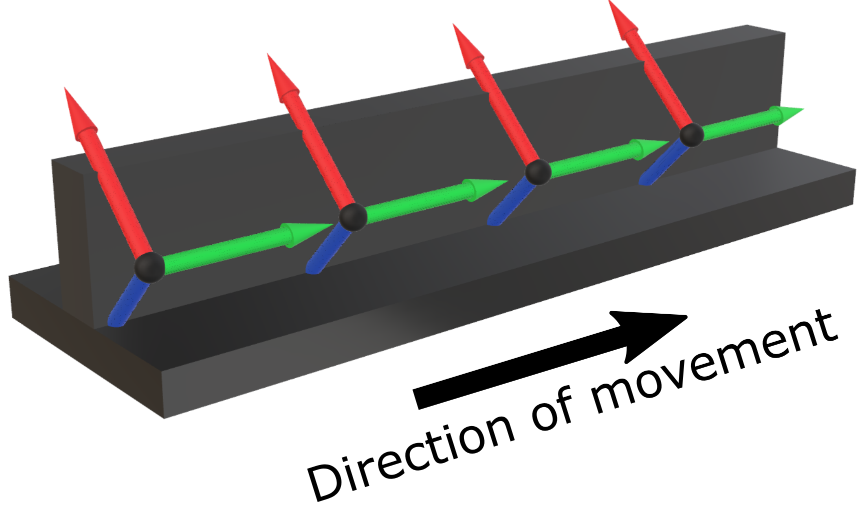


Figure 38: T-profile example for trajectory point convention.

When using this convention, it is possible to define a tolerance around the y-axis of the trajectory point frame, which is the same as a tolerance on the transverse welding angle. While a tolerance around the local x-axis, has the same result as a tolerance on the push/drag angle.

5.8 Custom cost function / edge weights / cost of trajectory point

When generating its graph, Descartes generates edges, with a certain weight. This *edge weight* is a real number, and reflects the *movement cost* to go from joint position A to joint position B. Suppose the robot has i amount of joints, with joint angles written as θ_i, and the edge weight for a jth trajectory point is calculated. We'll write the jth joint position as x_j. The ith joint angle of the jth trajectory point is written as $\theta_i(x_j)$. Then the edge weight w_j is calculated as follows:

$$w_j = \sum_{1}^{i} |\theta_i(x_j) - \theta_i(x_{j+1})| \tag{1}$$

The standard edge weight that Descartes uses is thus a simple sum of the differences in joint

angles between different joint solutions. With a simple change of the code, Descartes allows the possibility to create custom cost functions to generate the edge weights. However, the only variables that can be used in these custom cost functions, is information that can be extracted from the two *joint solutions* given to the edge weight function.

Thanks to the tolerances that Descartes allows, and the convention we chose to place trajectory points, we can now make the trajectory generation more flexible. By allowing a tolerance on the optimal path, the welding torch can deviate from this path, only restrained by the limits of the defined tolerance.

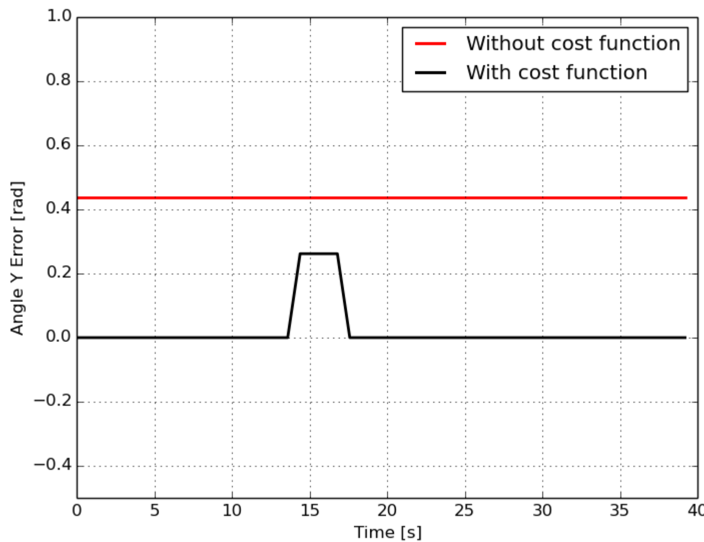


Figure 39: Deviation with and without deviation cost.

This results in paths where the welding torch will deviate from the optimal position across the whole path, simply because it is allowed to do so, and there is no method to limit deviations where they are not necessary. This is illustrated in figure 39, the red line depicts the rotational deviation around the y axis without an extra cost. This plot was generated from one of our test scenarios.

For this reason, an extra cost was added to penalize deviations from the optimal path. The first idea was to penalize toleranced frames, based on the angle between their z-axis and the z-axis of the optimal frame. (In this case the optimal frame, is the frame defined by the trajectory point.) The angle is illustrated in figure 40. The cost would be the size of the angle multiplied by a cost factor.

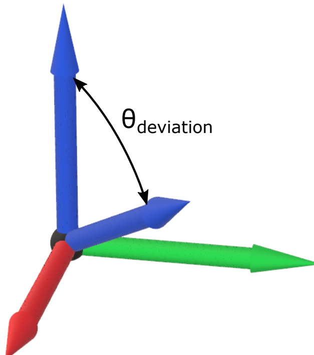


Figure 40: Angle between optimal frame and toleranced frame's z-axis.

However, with the introduction of our convention to place the trajectory points, and to differentiate between the different welding angles, it was then decided to actually split this cost into two separate parts. Both parts would be proportional to the deviation of one of the welding angles. This is illustrated in figure 43.

From now on we will call this extra cost the *welding cost* or the *deviation cost*. Since this

deviation cost is based on the deviation of a toleranced frame in comparison to the optimal frame, the cost is not part of any graph edge. Instead, one could say that the deviation cost belongs to the toleranced frame it was based on. Descartes uses this toleranced frame to calculate its IK-solutions, which are *joint solutions*, and thus in Descartes are saved as *joint trajectory points*. These joint trajectory points now need to include the extra deviation cost from the original toleranced frame they were generated from.

To implement this, all the different trajectory point classes used in Descartes, received an extra *deviation cost* variable, to store this extra cost.

Because the graph searching method can not take into account *node costs* but only edge weights, it was decided that for every generated edge, the deviation cost of the node it connects to is added to its edge weight, this is illustrated in figure 41. The costs from the nodes are added to their corresponding edges, illustrated by the bent arrows. If the deviation cost of the jth trajectory point is written as C_j, then the new edge weight is calculated as follows:

$$w_j = \sum_{1}^{i} |\theta_i(x_j) - \theta_i(x_{j+1})| + C_{j+1} \tag{2}$$

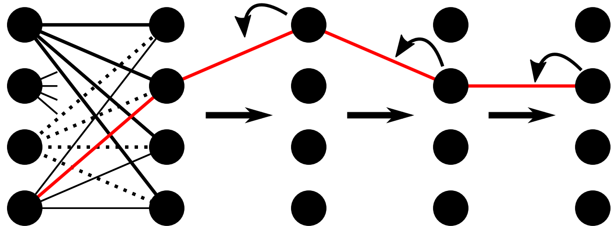


Figure 41: Illustrated graph, with chosen path in red.

To calculate the deviation cost, we need both the optimal frame, and a toleranced frame. (The toleranced frame is one discretization of the allowed tolerances on the optimal frame.) Because the function used in Descartes to calculate the edge weights only uses 2 joint trajectory points as input, it was not possible to calculate the deviation cost within this function. Instead this costwas calculated immediately after generating the list of toleranced frames for a certain trajectory point. The values were then stored inside of their extra cost variable.

When the toleranced frames were converted to joint trajectory points by the IK-solver, it was made sure that the deviation cost from the toleranced frames was copied into the joint solutions.

The calculation of the two different *welding angle deviations* used both the optimal frame, and the z-axis of the toleranced frame, as illustrated in figure 42. By projecting the toleranced frame's z-axis onto the x-z-plane of the optimal frame, the angle between this projection and a unit vector placed along the z-axis of the optimal frame can be calculated using the dot product of these two vectors. This results in the deviation of the *transverse welding angle*. When projecting the toleranced frame's z-axis onto the y-z-plane of the optimal frame, the angle between this projection and the unit-vector along the z-axis of the optimal frame is the deviation of the *push/drag angle*.

If $\vec{a}$ and $\vec{b}$ are vectors, and θ is the smallest angle between them, the dot product $\vec{a} \cdot \vec{b}$ can be defined in the following manner:

$$\vec{a} \cdot \vec{b} = ||\vec{a}||\ ||\vec{b}||\ cos(\theta) \tag{3}$$

If we calculate the dot product between unit vector $\vec{z}_{unit}$ along the optimal frame's z-axis, and the projected vector $\vec{z}_{proj_tol}$ the smallest angle θ between them can be calculated as follows:

$$\theta = acos\left(\frac{\vec{a} \cdot \vec{b}}{||\vec{a}||\ ||\vec{b}||}\right) \tag{4}$$

These angles are then multiplied by unique cost factors and added together to obtain the deviation cost. The unique cost factors allow us to differentiate between the different welding angle costs. This means that we can give a higher cost to an angle that can be more crucial to a given situation, and a lower cost to the angle that is allowed to change more.
Because the graph search tries to minimize the total trajectory cost, this results in a prioritization of certain welding angles over others.

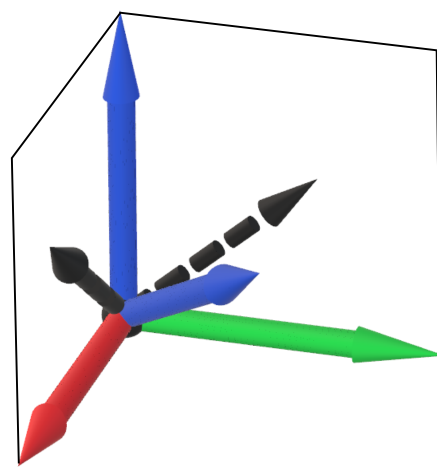


Figure 42: Projection of the toleranced z-axis on the x-z- and y-z-planes.

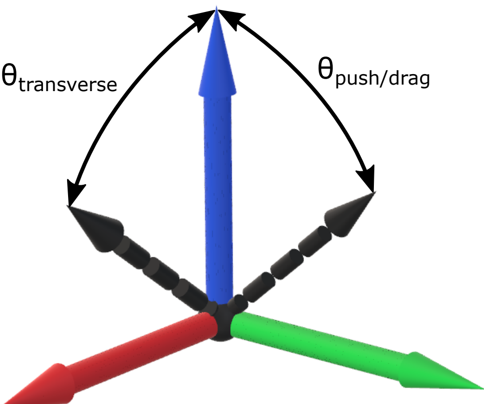


Figure 43: Angles between z-axis projections and optimal frame's z-axis.

5.9 Saving generated data in .bag files

To plot graphs, and extract data from the program, it was necessary to save data to files. Thankfully, ROS has a handy tool for that called *rosbag*[29]. It allows the user to save any ROS message to a so-called BAG file. They are called *bag files* because of their .bag file name extension. Rosbag has a *C++* and a *Python* API, so in both languages bag files can easily be written and read.

The first data we saved was simply the generated list of joint solutions (the trajectory of the robot) generated by Descartes. This could then be used to play back the desired trajectory without having to generate it again. We then continued to add the used collision objects to the bag file, which could then be loaded together with the saved trajectory. (The collision objects were part of a Moveit! *planning scene* which was stored completely into the bag file.) Now we could play back generated trajectories, together with the environment they were created for.

The following other data was stored in the bag file:

- List of trajectory points defined by the user.
- The list of robot poses, chosen by Descartes, in *joint trajectory point* type.
- List of deviation costs for every trajectory point.
- List of the angle errors around the x axis. (Corresponds to the push/pull welding angle error.)
- List of the angle errors around the y axis. (Corresponds to the transverse welding angle error.)

Thanks to rosbag's Python API, in combination with existing open-source plotting libraries for Python, like matplotlib[30], the data can be read from the bag file and plotted in a few lines of code. An example plot is illustrated in figure 44. An example script, used to generate such a plot, can be found in Appendix B.

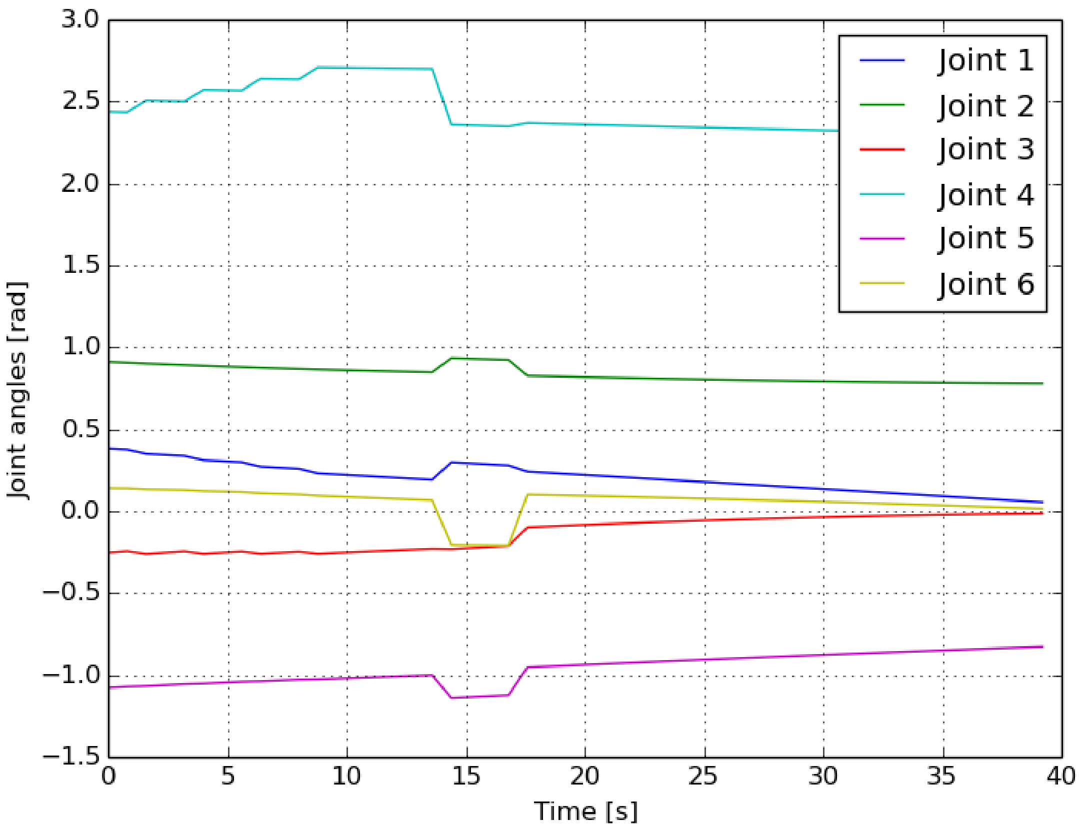


Figure 44: Example plot of joint angles.

5.10 Defining trajectory points

Multiple help functions were written to generate the necessary trajectory points for our test cases. One such function was a simple function to generate multiple poses across any line in space. This helped with defining welding paths that consisted out of straight lines. An example of these generated trajectory points is shown in figure 45.

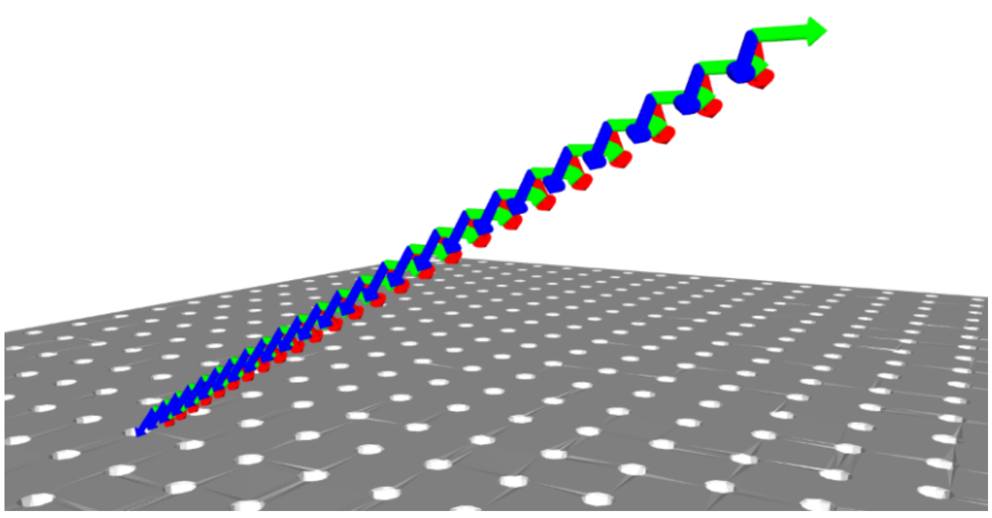


Figure 45: Trajectory points across a line in space.

A second help function we wrote was a function to define circular welding paths in space. This happened by defining the center point of the circular welding ath, and a radius. The circular paths could be turned in any way in space. It was also possible to define smaller circle arcs. An example generation is shown in figure 46.

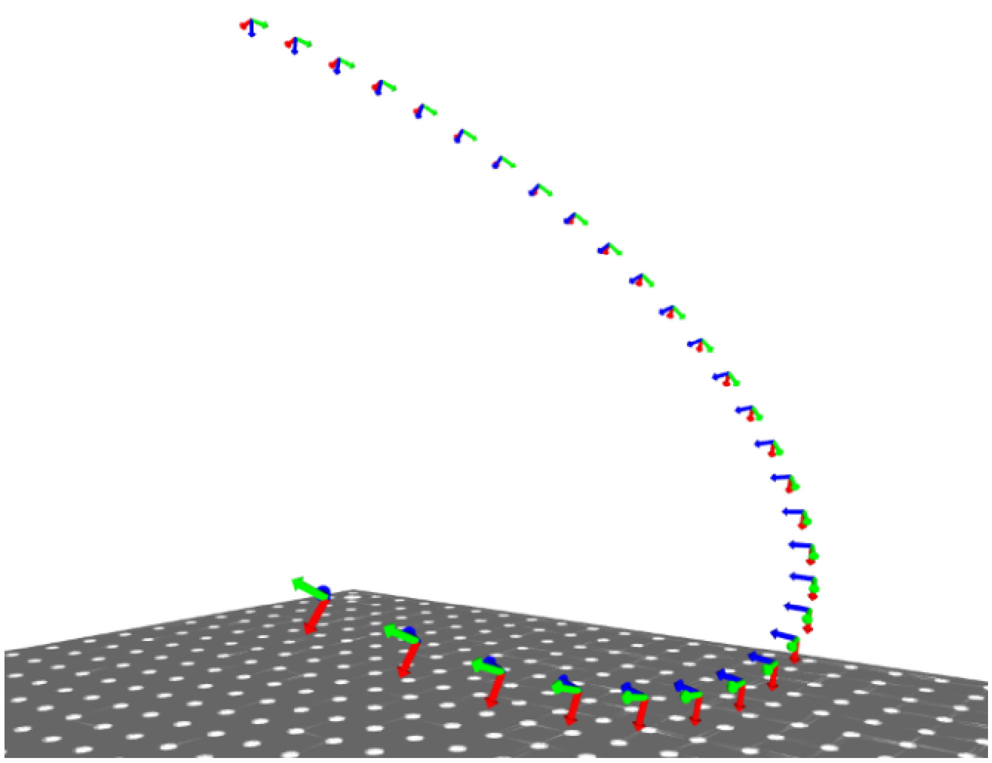


Figure 46: Trajectory points across a circle arc in space.

Finally, thanks to the local frame functions defined to help with the local tolerances, it was also possible to generate more complex trajectories, like ellipses and spiral shapes. An example is shown in figure 47.

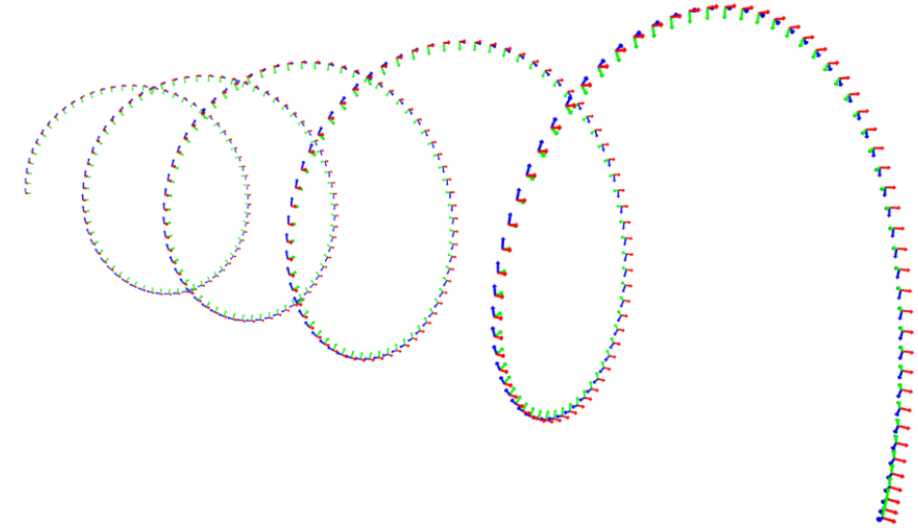


Figure 47: Trajectory points forming a helix.

6 Simulations and Examples

In this chapter, several of our simulations are shown. All the variables of the test cases, together with multiple renderings can also be found in appendix D, E, F, G and H.

In every simulation, the generation of the trajectory can be split up into three phases. In the first phase, the IK-solver calculates all possible joint solutions. In the second phase the graph is generated. In the final third phase, the graph is searched for the trajectory with the lowest cost. For each of these phases, the time needed to perform the phase is tracked.

6.1 Test case 1: Tube on Plate

One of the first workpieces we tested the trajectory generation on, was a hollow cylinder on a plate. Hence called "Tube on plate".

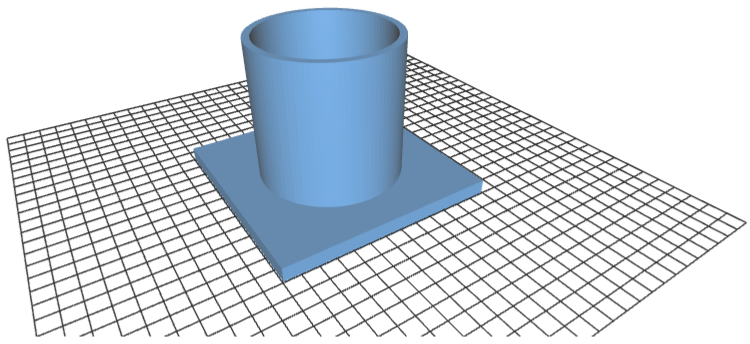


Figure 48: Tube on plate workpiece.

To weld these pieces together, a circular weld was necessary, turned downwards under 45 degrees, as illustrated in figure 49. The trajectory consisted out of 30 trajectory points. We allowed 50° of tolerance around both the x and the y axis. Tolerance around the z axis was 360°. The discretization step size was 5°. The welding speed was set to $0.1 m/s$.

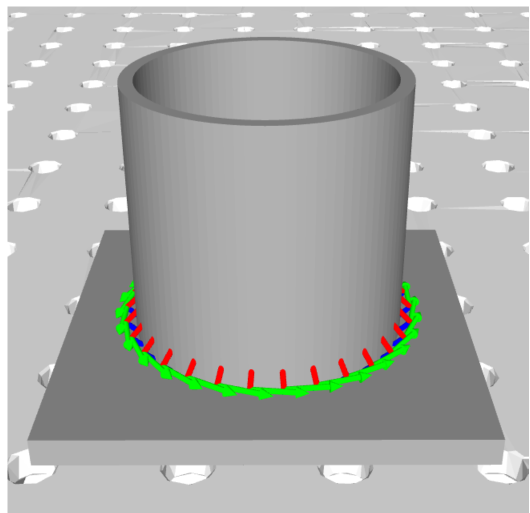


Figure 49: Workpiece with trajectory, placed on the welding table.

Using the IK-fast solver, the generation time was about 27 seconds for the trajectory generation with the welding cost. This resulted in a pretty smooth trajectory, shown in figure 50.

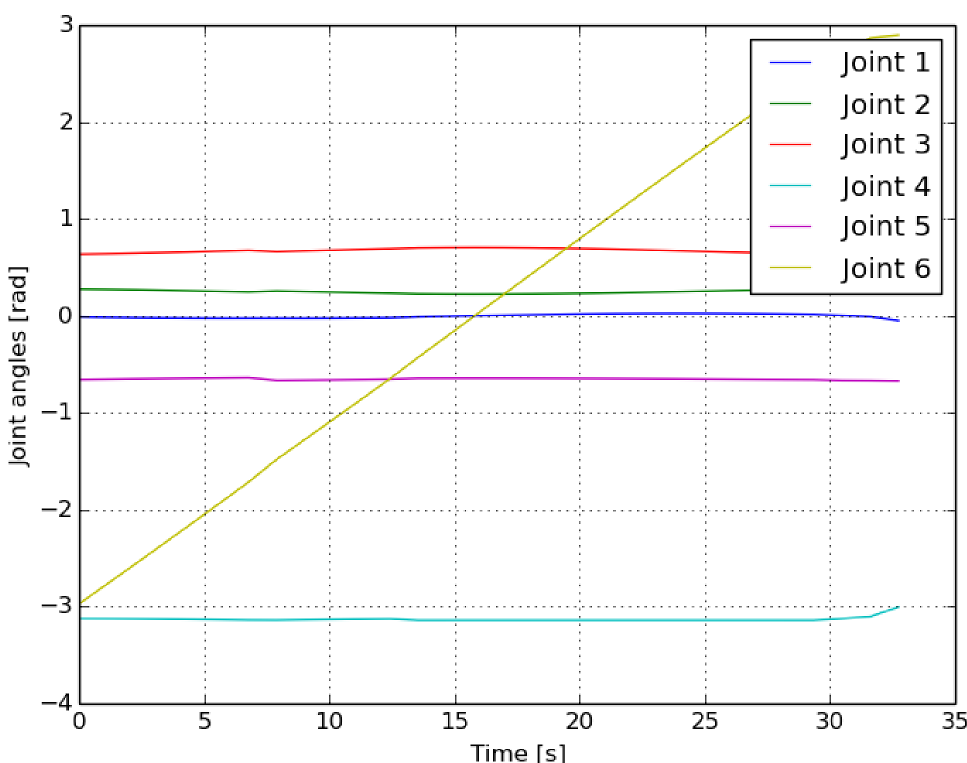


Figure 50: TOP 1: Joint angles through time, with deviation cost.

This is in fact one of the smoothest trajectories of all the test cases. It is interesting to see that joint 6 turns a full circle.

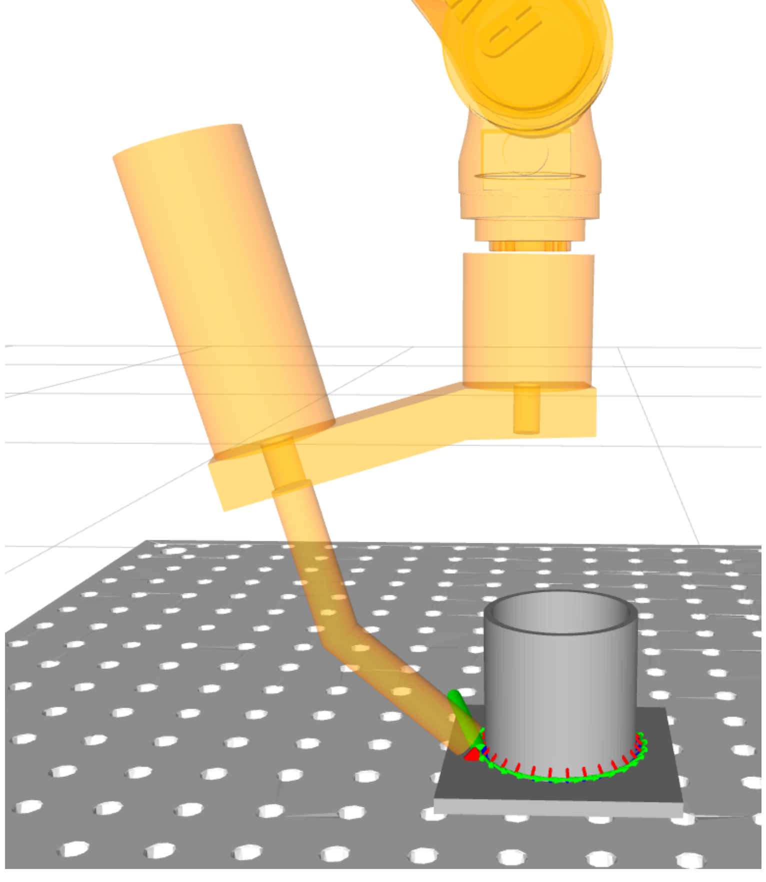


Figure 51: TOP 1: Robot executing trajectory.

The trajectory was then generated again, but without the extra deviation cost. This resulted in a similar trajectory. Watching it move, the trajectory looked smooth, but looking at the joint positions through time (figure 52), it can be seen that joint 4 makes some jumps.

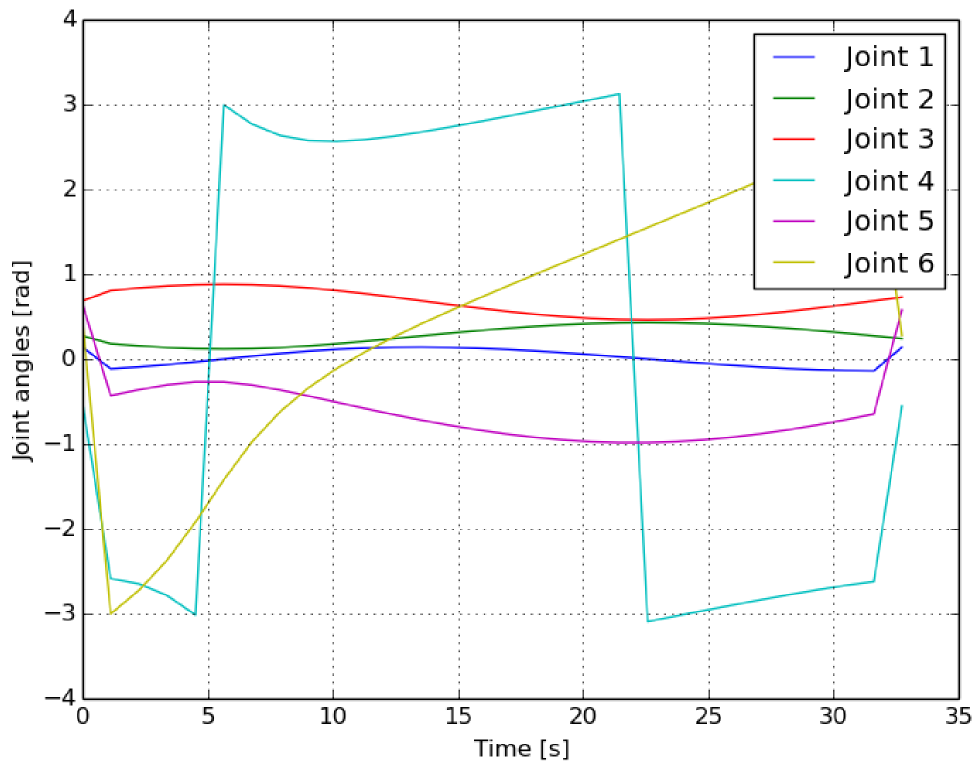


Figure 52: TOP 1: Joint angles through time, without deviation cost.

It can also be noticed that without a deviation cost, the planner deviates from the optimal path by a constant value throughout the whole trajectory. While this error becomes zero when the cost is enabled. This is shown in figures 53 and 54.

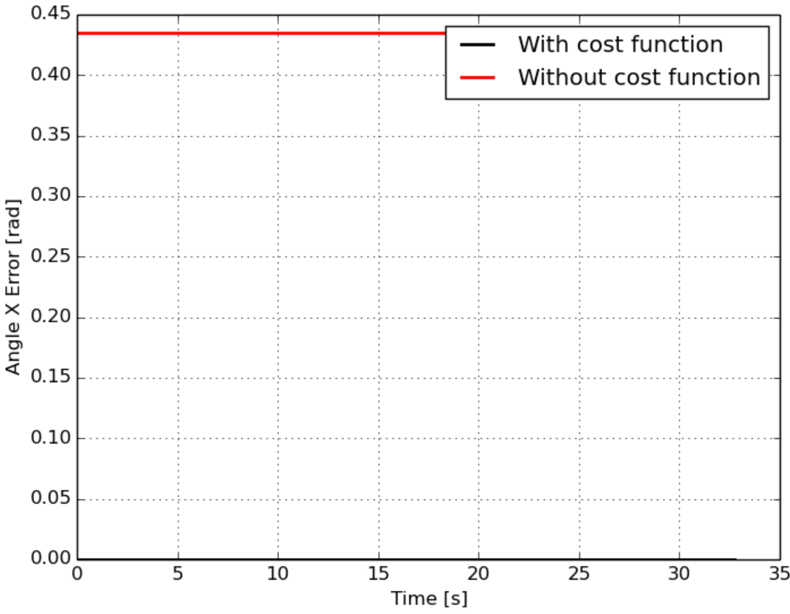


Figure 53: TOP 1: X angle error throughout time.

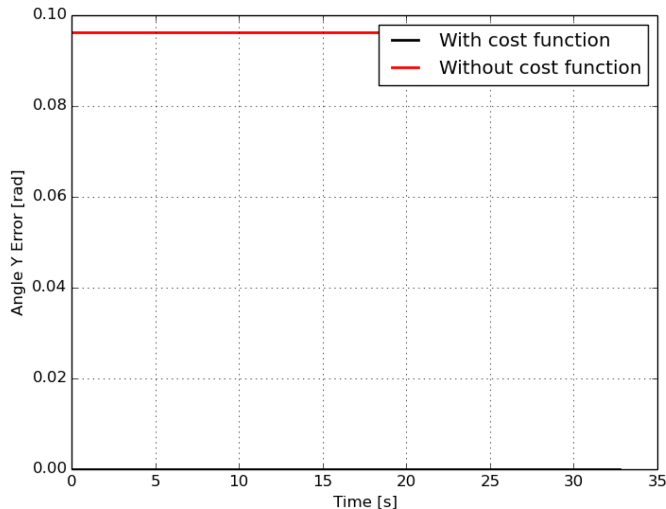


Figure 54: TOP 1: Y angle error throughout time.

6.2 Test case 2: L Profile

This test case was made to illustrate the effect of the deviation cost on the trajectory generation. The workpiece is an L-shaped metal profile, with a small rectangular plate, as shown in figure 55. The purpose of the rectangular plate is to force the welding torch to deviate from the optimal welding path, by forcing a collision.

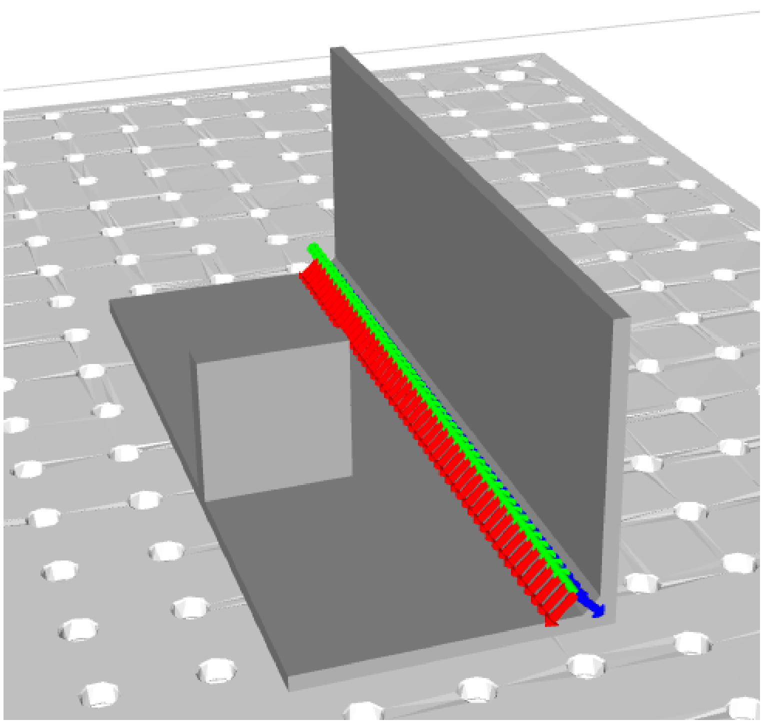


Figure 55: Workpiece with trajectory, placed on the welding table.

The trajectory consists of 50 trajectory points, placed in the corner of the L profile. Two

tolerances are given, one around the y axis, of 50 degrees. The other one is the tolerance around the z axis, of 360 degrees. Discretization step size is 5 degrees. The welding speed is set to $0.1m/s$. The first trajectory generation took about 130 seconds using the IKfast solver, including the deviation cost. In figure 56 you can clearly see at about 14 seconds, that the welding torch starts to deviate from its path to evade the rectangular plate. At about 17 seconds, the robot returns back to its original path, this is exactly what the extra deviation cost is meant to do.

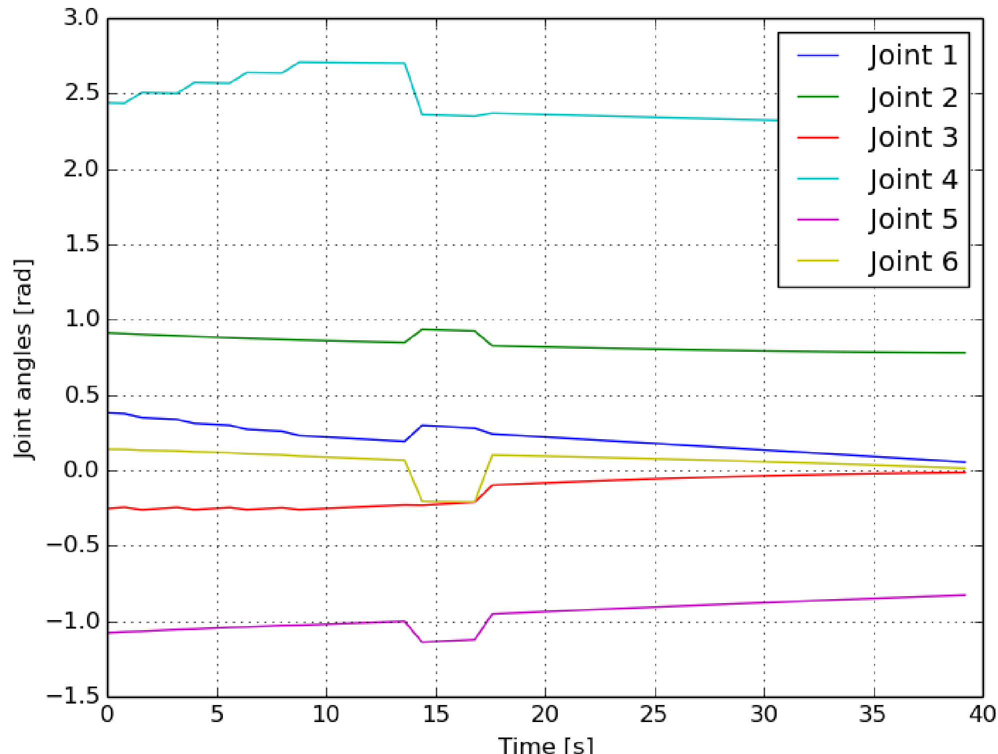


Figure 56: L profile: Joint angles through time, with deviation cost.

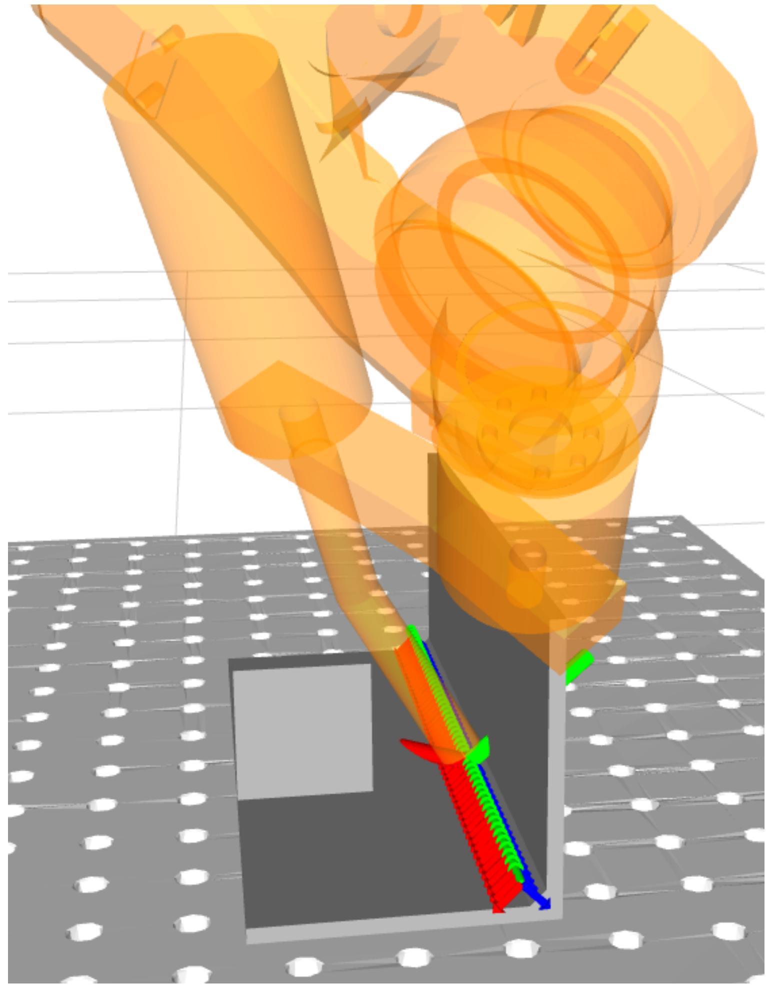


Figure 57: L profile: Welding torch evading collision.

A second trajectory generation with the same settings was done, but without the extra deviation cost. This resulted in the joint angles shown in figure 58. It can already be seen that the robot does not seem to evade any objects during the path. In fact, the robot constantly deviated from the optimal trajectory, causing the welding torch to evade the rectangular plate. This can also be seen in the error plot, figure 59. Without the cost, the torch keeps a constant deviation from the optimal path. With the deviation cost, it only evades the restangular plate, and then reverts back to the optimal trajectory.

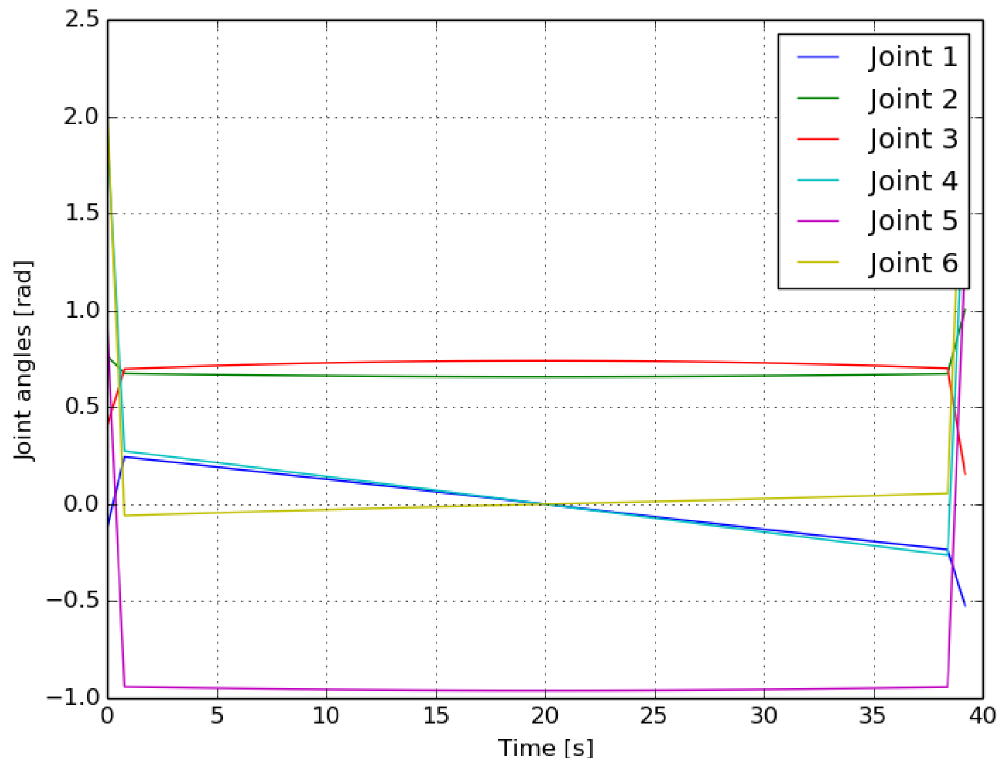


Figure 58: L profile: Joint angles through time, without deviation cost.

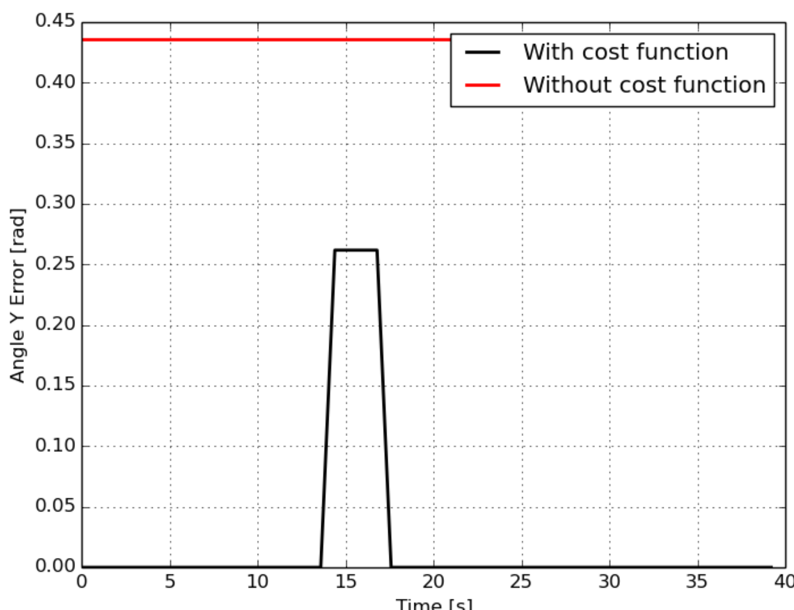


Figure 59: L profile: Y angle error throughout time.

This trajectory was also generated using the slower KDL solver. It used the same settings, but took about 8479 seconds to generate. Using the deviation cost, the welding torch evaded the rectangular plate as expected.

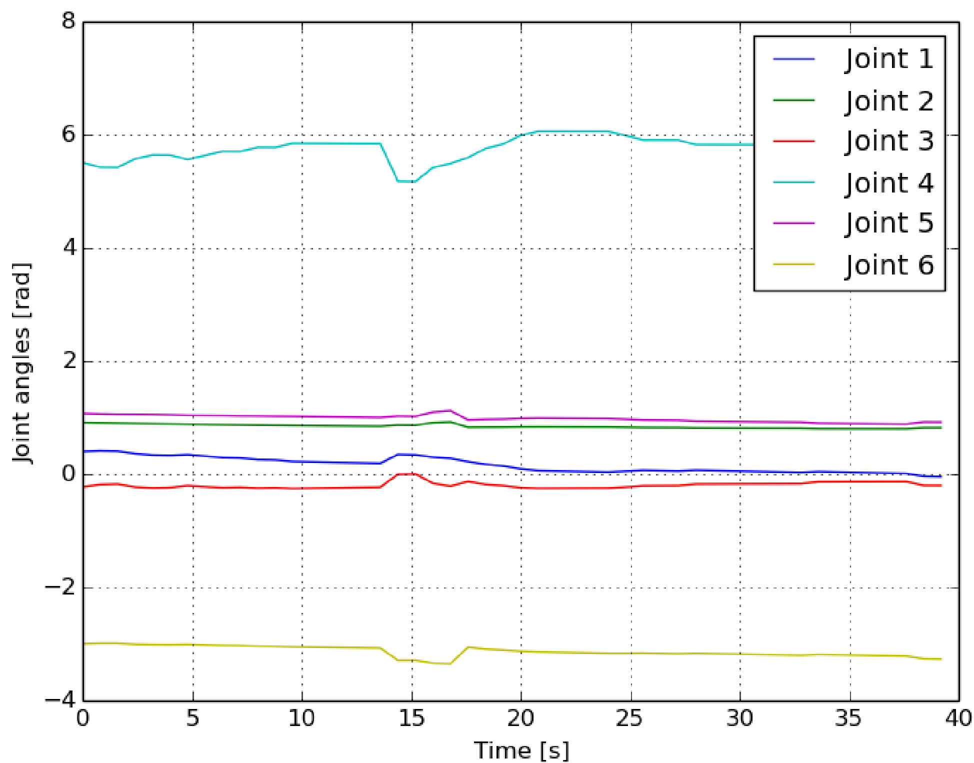


Figure 60: L profile: Joint angles through time, with deviation cost and KDL solver.

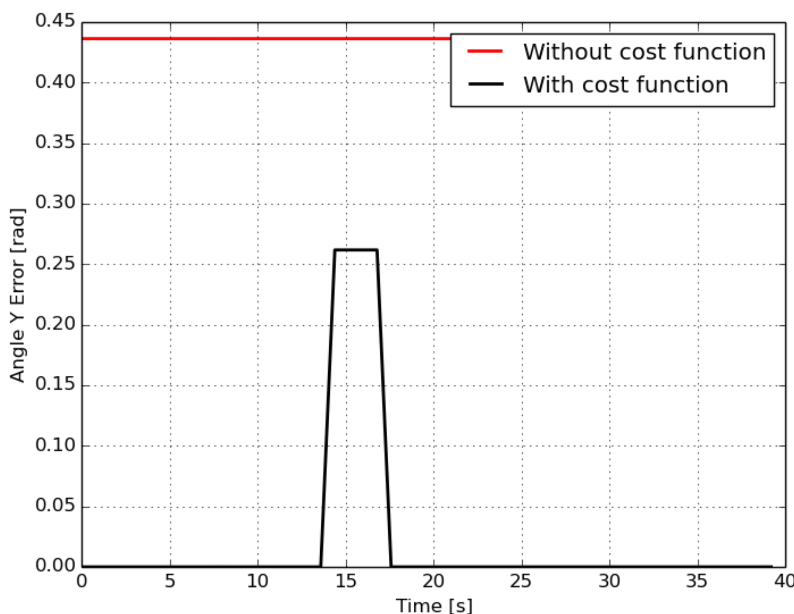


Figure 61: L profile: Y angle error throughout time, KDL.

Another interesting graph is shown in figure 62. This graph shows the angular error around the x axis. Since no tolerance around the x axis was defined, this error should always remain zero. However, this plot shows an effect of the KDL solver's numerical nature. Unlike the IK fast solver, the joint solutions generateed by the KDL solver, for a certain trajectory point, do

not place the tool frame in exactly the same position as the trajectory point. Instead there are some small rounding errors.

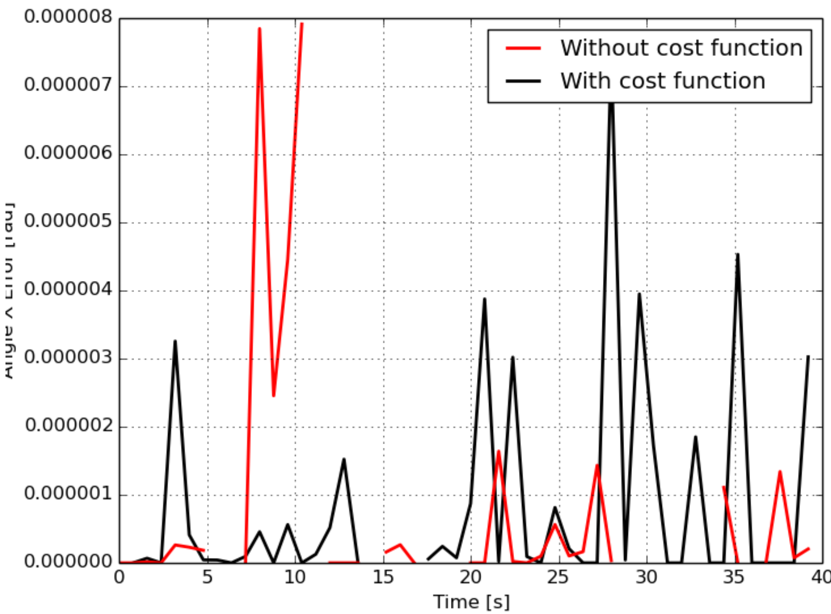


Figure 62: L profile: X angle error throughout time, KDL.

6.3 Test case 6: furniture piece

The final test case shown here is a work piece based on a piece of furniture made by the company Robberechts. As shown in figure 63, the workpiece consists of a large U-shaped profile, with two smaller profiles, perpendicular on the larger frame.

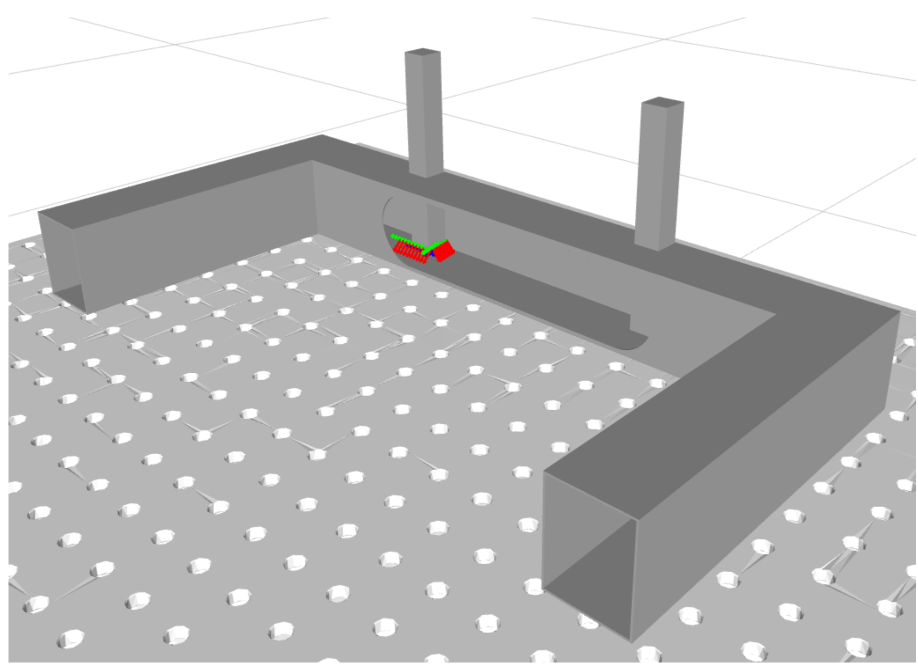


Figure 63: Furniture piece: workpiece with trajectory.

The goal of the weld is mounting the smaller pieces onto the larger frame. A zoom of the trajectory points is shown in figure 64.

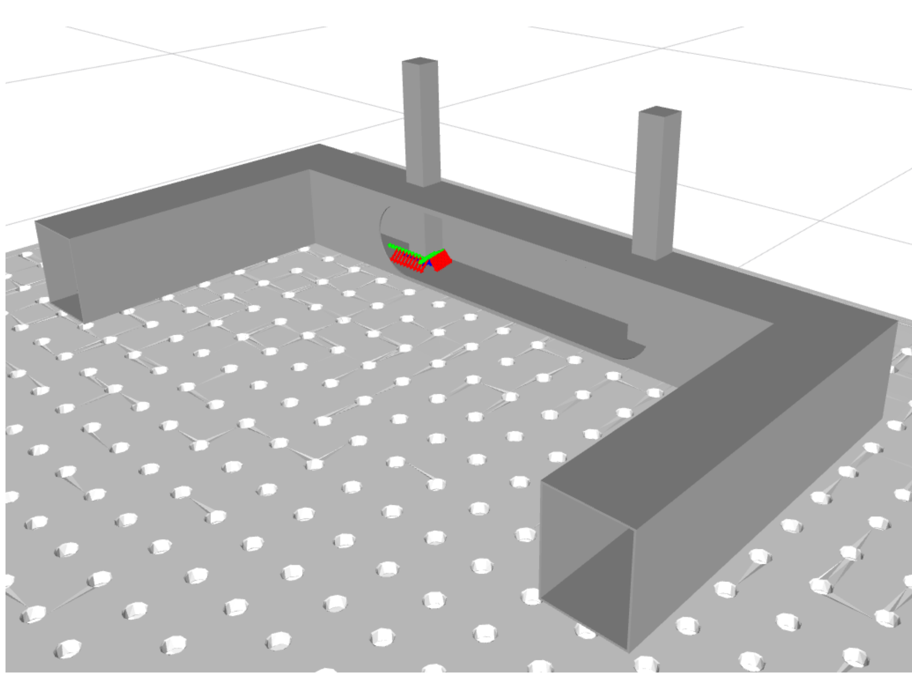


Figure 64: Furniture piece: trajectory points.

The trajectory consists of 20 trajectory points. Tolerance about the x and y axis were both set to 50 degrees. Tolerance around the z axis was 360 degrees. The tolerance discretization step size was set to 5 degrees.

Using the IKfast solver, the generation time was about 368 seconds. In figures 65 and 66, the robot is shown executing the trajectory.

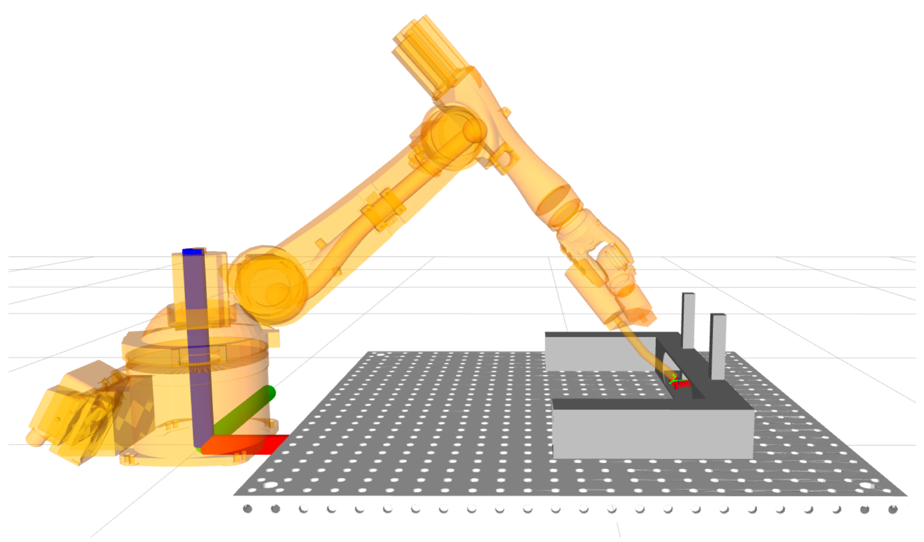


Figure 65: Furniture piece: robot executing trajectory.

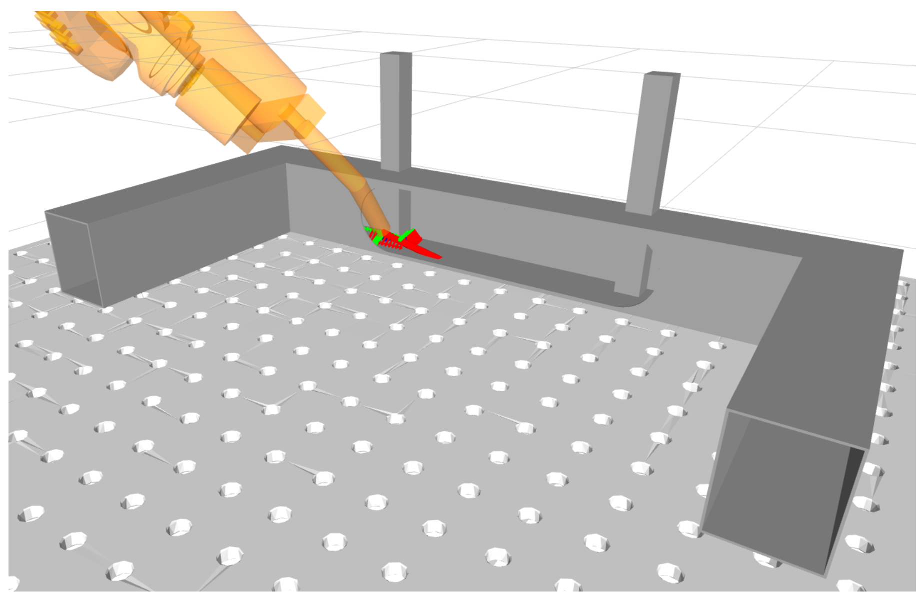


Figure 66: Furniture piece: robot executing trajectory, zoom.

In figure 67, the generated joint angles are shown. It's a pretty smooth trajectory, except for around the 3 second time mark, where the robot has to switch from one weld line to another.

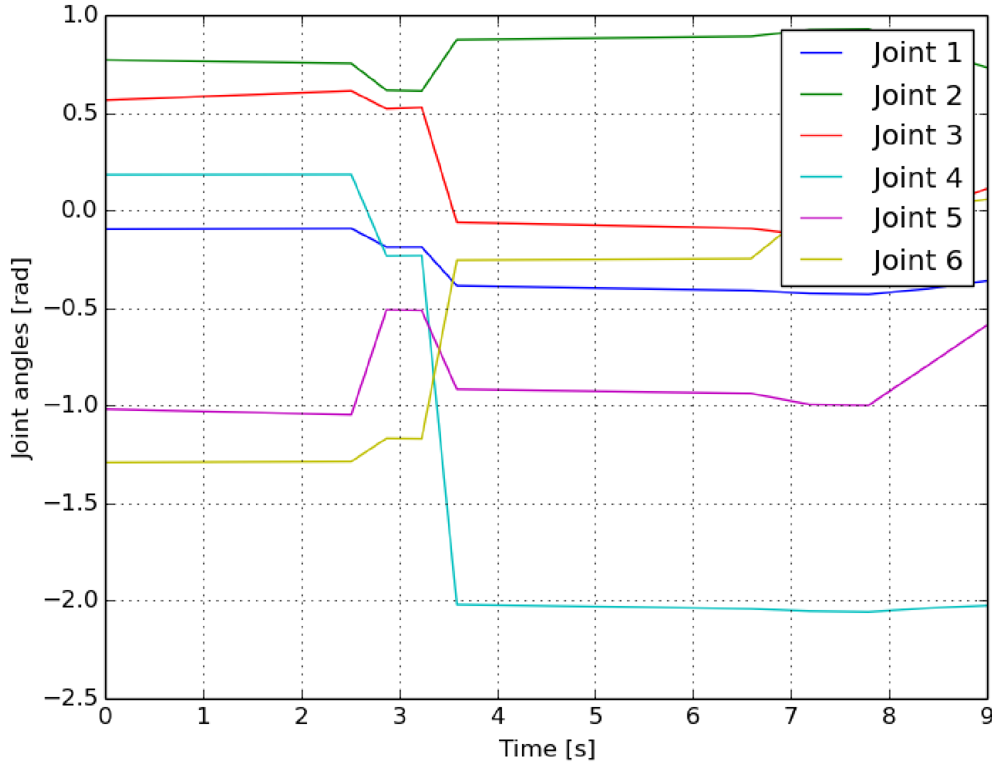


Figure 67: Furniture piece: Joint angles through time.

When looking at the angle errors, shown in figure 68 and 69, it is interesting to note that the angle error around the y axis is almost completely the same, whether we use the extra deviation cost or not. When looking at the angle error around the local x axis however, it can clearly be seen that the error is substantially lower when using the extra cost function.

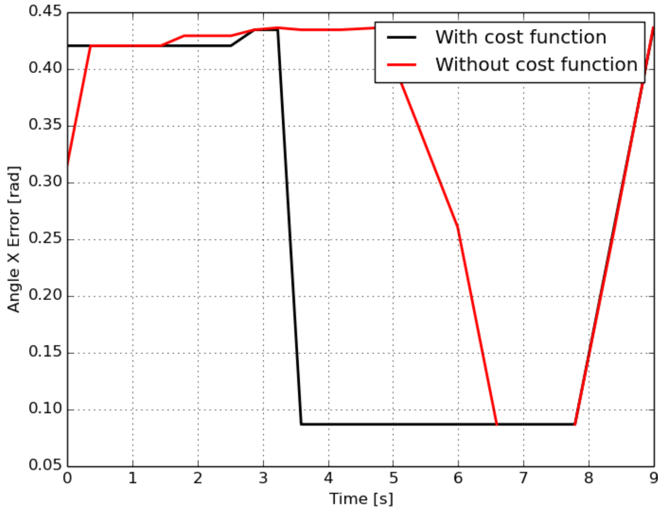


Figure 68: Furniture piece: X angle error throughout time.

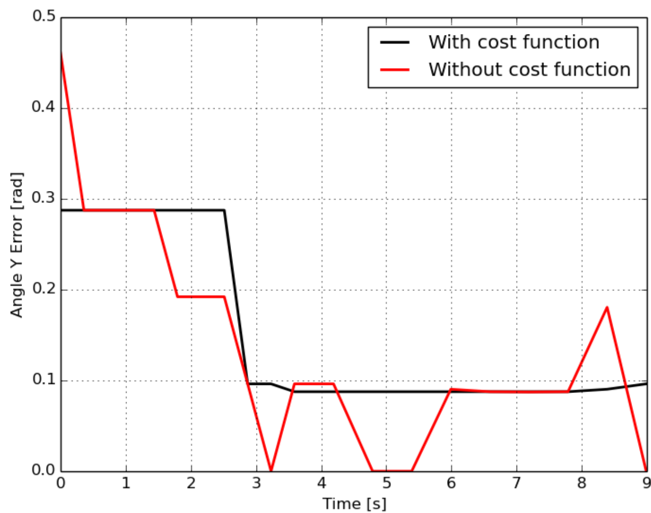


Figure 69: Furniture piece: Y angle error throughout time.

7 Problems and Ideas

7.1 Number of edges and RAM

One of the problems we encountered during our simulations was that the trajectory generation would crash during execution. Inspecting the problem more closely, it became clear that the node crashed because it started to allocate huge amounts of RAM. It happened while generating trajectories with multiple tolerances, and a small discretization step size.

As illustrated in figure 31, Descartes generates edges between every joint solution of a certain trajectory point to each joint solution of the following trajectory point. For clarity, this illustration is repeated here.

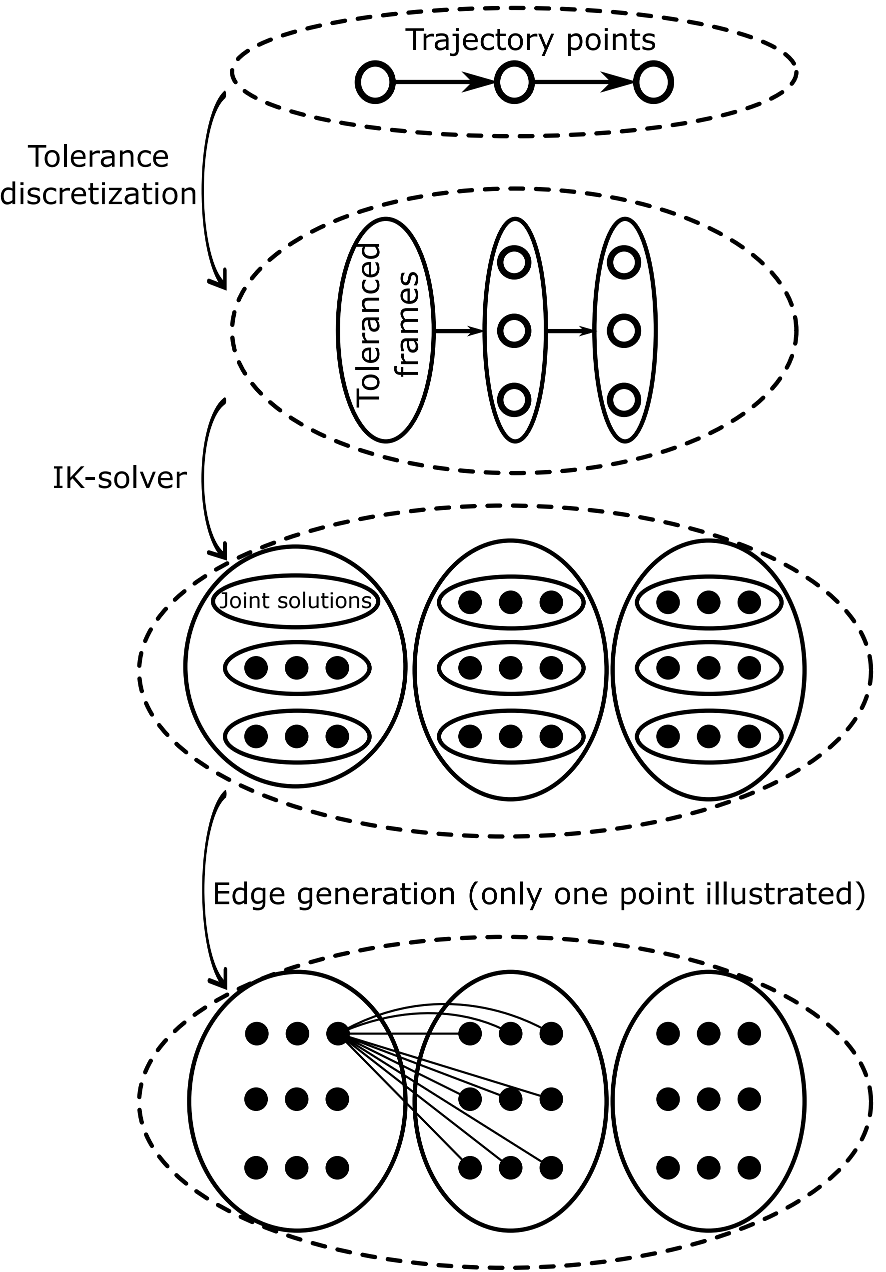


Figure 70: Steps to construct a graph from trajectory points, including tolerances.

To calculate the amount of graph edges that are generated, we will walk through the different steps used to generate the graph. First we start off with the *tolerance discretization.* Suppose that we have a trajectory point with only rotational tolerances. (Translational tolerances were never used during this thesis.) First we have the allowed tolerance around the tool's z axis. Since the tool is allowed to rotate completely around its z axis, the total allowed deviation is 2π. Above this allowed rotational tolerance, two more rotational tolerances may be defined, both around the x and the y axis. For clarity's sake, let's suppose the rotational x and y tolerances are of the same value, written as θ_{tol}. We also have a discretization step size θ_{disc}. The amount of toleranced frames $n_{tolframes}$ generated for this single trajectory point can be calculated as follows:

$$n_{tolframes} = \left(\frac{\theta_{tol}}{\theta_{disc}}\right)^2 \cdot \frac{2\pi}{\theta_{disc}} \tag{5}$$

For a common discretization step size of 1 degree, the second term $\frac{2\pi}{\theta_{disc}}$ already results in 360 toleranced frames.

The second step is the generation of *joint solutions* by the IK-solver. Because the number of inverse kinematic solutions depends strongly on the trajectory point itself, it is impossible to predict this amount. However, through experience we have noted that the amount of IK solutions is usually greater than 1, going up to 10 or more. To illustrate the amount of edges, we will simply guess that every toleranced frame will have 5 IK solutions. The total number of joint solutions $n_{jointsol}$ then is:

$$n_{jointsol} = 5 \cdot n_{tolframes} = 5 \cdot \left(\frac{\theta_{tol}}{\theta_{disc}}\right)^2 \cdot \frac{2\pi}{\theta_{disc}} \tag{6}$$

Suppose that we start with a trajectory consisting of n trajectory points. Except for the last trajectory point, every joint solutions of any trajectory point will be connected to every joint solution of the *next* trajectory point. If we suppose that every trajectory point has the same amount of joint solutions $n_{jointsol}$, the total amount of generated edges n_{edges} can be calculated as follows:

$$n_{edges} = (n-1) \cdot n_{jointsol}^2 = (n-1) \cdot \left(5 \cdot \left(\frac{\theta_{tol}}{\theta_{disc}}\right)^2 \cdot \frac{2\pi}{\theta_{disc}}\right)^2 \tag{7}$$

To get an idea of the amount of generated edges, the following variable values were used:

- Amount of trajectory points: $n = 100$
- Allowed tolerances on x and y axis: $\theta_{tol} = 30°$
- Discretization step size: $\theta_{disc} = 1°$

The total number of edges n_{edges} then equals 10 392 624 000 000 or about $10.4 \cdot 10^{12}$.

Since the edge weight value is stored as a double, which takes 64 bits of data, the amount of RAM necessary for storing only the edge weights, would be about 77 431 gigabytes, in this example. The edge weight object also includes data regarding the joint solutions it belongs to. So we can say that the amount of RAM necessary to store all of the edges will certainly exceed 77 431 gigabytes. To be able to generate trajectories on normal computers, it was thus

necessary to lower the total number of edges, by increasing the discretization step size, or lowering the allowed tolerance intervals, or even removing a certain tolerance altogether.

There are a few other ideas that we had to lower the amount of edges that have to be generated, but they all cause the trajectory generation to lose completion.

The first idea was to only allow tolerances where they are necessary. (Iterative tolerances.) This means that if a trajectory point is encountered, where no IK-solutions can be found, the software automatically replaces this point with a version that includes tolerances. Then the trajectory generation is restarted. This has been tested, and can show promising results, in lowering both the necessary generation time and the number of generated edges. The main disadvantage is that the trajectory generation loses completeness. Using the normal generation, all the possible toleranced frames are calculated. Using one of the toleranced frames in the beginning may open up a totally different interesting trajectory later on. If these tolerances are not calculated, possibilities are lost.

A second idea was to split the trajectory up into smaller sub-trajectories. This was not tested in this thesis.

7.2 Iterative tolerances

The iterative tolerances, mentioned in the previous section, were also tested on the L profile case, which showed very promising results. We start off the trajectory generation with trajectory points with no tolerances on any axis. We let the trajectory generation execute, thus proceeding with the IK-calculations, until a trajectory point is found for which no IK-solutions can be found. The trajectory generation is then stopped. The trajectory point where no IK-solutions can be found for, is then replaced with an identical copy, except this time with allowed tolerances. We restart the trajectory generation, if the same point has problems, we enlarge the allowed tolerances (until a limit is reached), and restart the generation again.

This way, tolerances are only used on the trajectory points where they are absolutely necessary.

We used this approach on the same trajectory of the L-profile case mentioned in the previous chapter. This is shown again in figure 71.

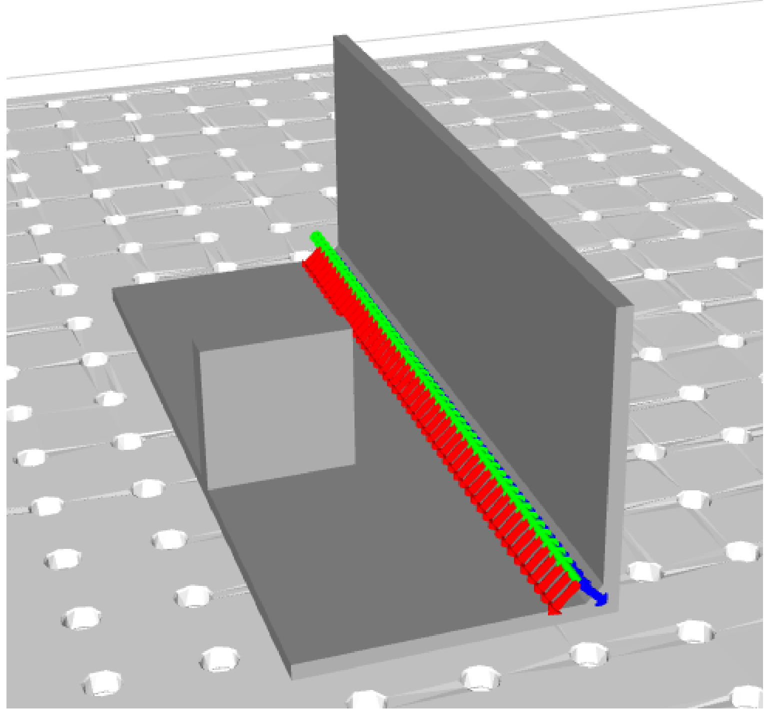


Figure 71: L profile with trajectory.

We used the same 50 trajectory points as the previous case. The discretization step size was set to 1 degree. (More accurate and smooth generations.) As IK solver we used IKfast.

The trajectory was generated in 0.37 seconds. (!) This is very fast, compared to the generation time of 8479 seconds when using KDL and the same scenario. When using IKfast on the same scenario without using iterative tolerances, the generation time was 129 seconds. So this is a thousand fold improvement, in this case.

Figures 72 and 57 show the robot executing the trajectory.

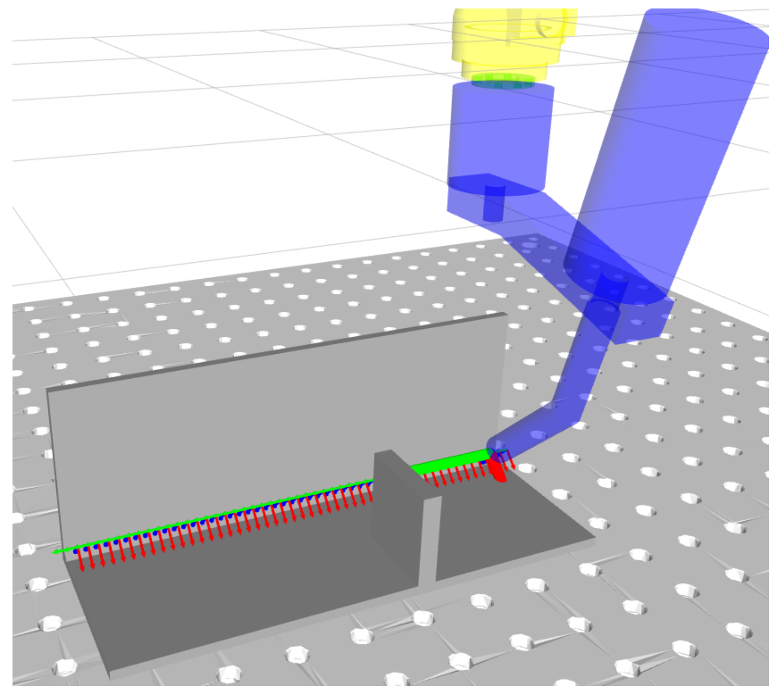


Figure 72: Robot executing trajectory.

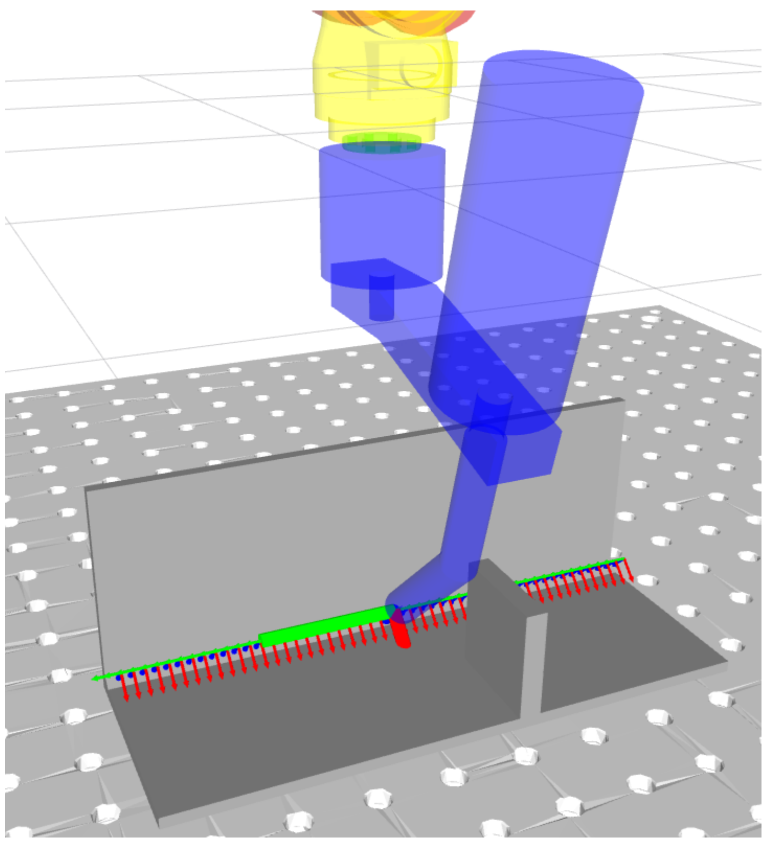


Figure 73: Robot executing trajectory.

In figure 74, the joint angles are shown, which are as smooth (if not smoother) as the original scenario.

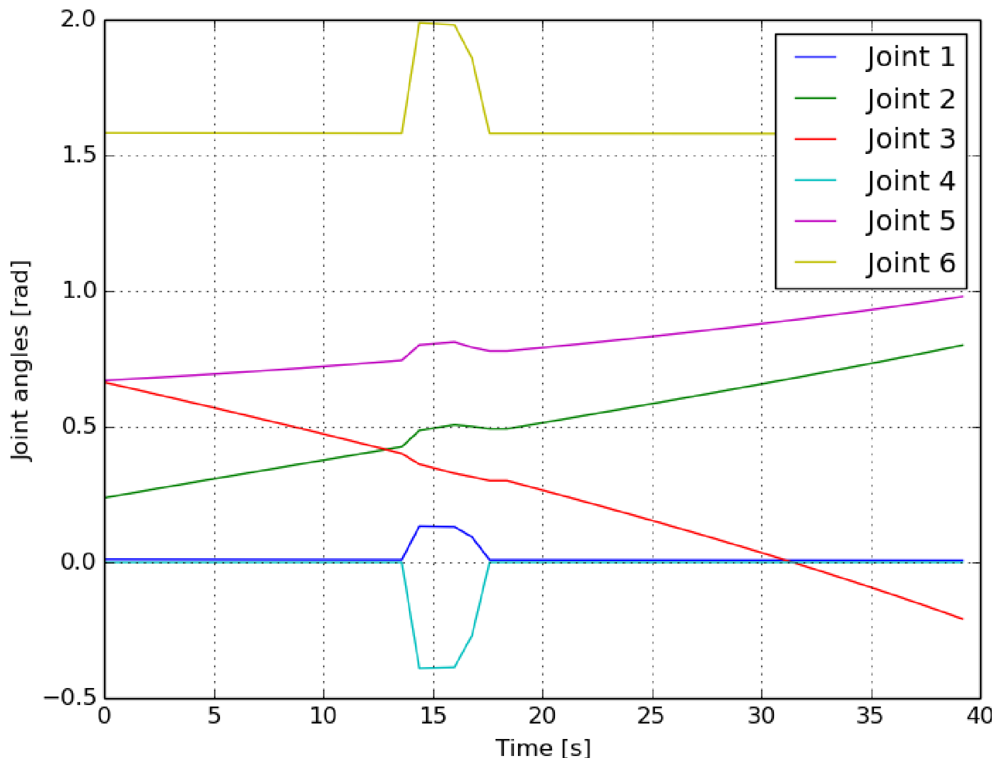


Figure 74: L profile, iterative tolerances, joint angles.

Figure 75 shows another interesting behavior caused by iterative tolerances. You can see that without using an extra deviation cost, the welding torch still return to its optimal trajectory. This is because in the surrounding points, no tolerances are allowed, so the welding torch will automatically return to the optimal trajectory, without requiring any extra functionality like deviation costs.

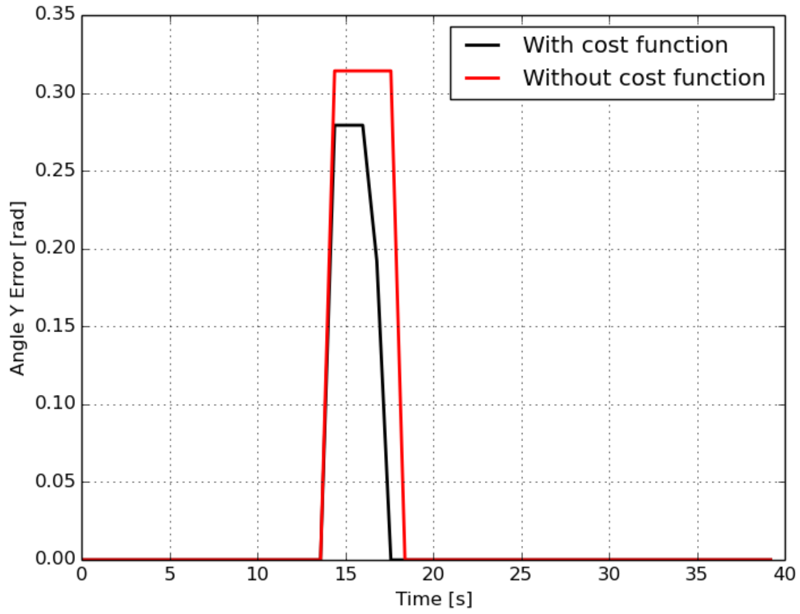


Figure 75: L profile, Y angle errors, iterative tolerances.

7.3 Glitchy behavior / jumping through obstacles

When generating a trajectory for our robot, with environmental collision objects, it was sometimes noticed that the robot ”jumped through” obstacles. In fact, we made up some scenarios that force the robot to pass through an obstacle, and noticed that it did just that. The actual thing happening with the collision detection in Descartes (Descartes uses the collision detection from MoveIt!, while MoveIt! uses a package called FCL[32].) is that every generated joint solution is tested whether it is in collision or not. This can be a self-collision or a collision with the environment. If this joint solution is in collision, it is simply thrown away, and it is not used in the graph, and thus, also not searched in the graph.

Because a trajectory consists of discrete trajectory points, it is guaranteed that the robot will never collide in any of these trajectory points. The problem however, lies in what happens in between these trajectory points. As an example, take a look at figure 76. The robot has to perform a circular weld on a work object, but in front of the work object is a tall pole, which the robot can collide with.

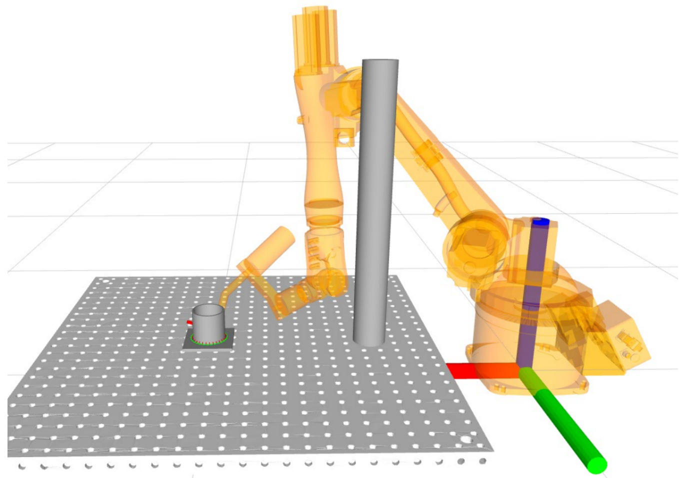


Figure 76: Scenario with tall pole.

In the generated trajectory, the robot will start to execute the weld, until it starts to come closer to the pole. It will then start to ”evade” the pole, while continuing its weld. This works well, until there is a trajectory point that can only be reached if the robot arm is placed on the other side of the pole. The Descartes package will then simply generate the next joint solution, where the robot is positioned on the other side of the pole. In effect, it just jumped right through the obstacle. Because of this ”jump” in joint angles, Descartes will generate a pretty high cost for this jump. But because it is the only possible solution to complete the trajectory, this cost is simply accepted.

These jumps can also be seen in the plot of the joint positions, as shown in figure 77.

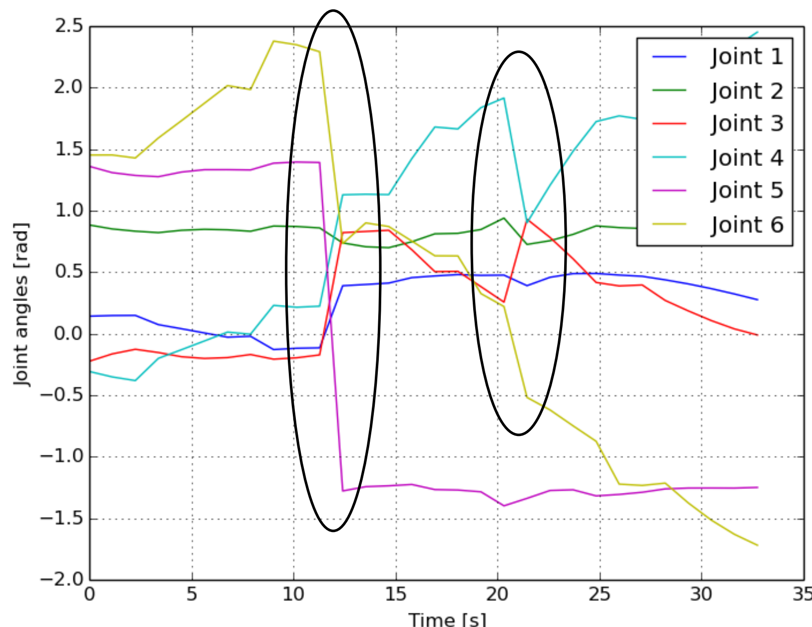


Figure 77: Joint angles through time, jumps are within ellipses.

The "correct" solution would of course be for the robot to stop welding, move around the pole, and then continue welding. Of course, the Descartes package is not made to generate trajectories like this, nor do we expect it to. What should be expected of the package, however, is that it detects impossible trajectories.

To solve this issue, we had the idea of generating intermediary joint positions, in between different joint solutions. Suppose we have joint solutions A and B, consisting of the different joint angles θ_{Ai} and θ_{Bi}, where i is the joint number. Suppose that between joint positions A and B, the robot jumps through the obstacle. We now want to find out, with relative certainty, that the robot collides with the environment in between these joint positions.

To find out, we generate n intermediary joint position steps, for a robot with m joints.

$$With : j = 1...n$$

$$and : i = 1...m$$

The joint angles of the jth intermediary joint solution $\theta_i(j)$ can then be written as:

$$\theta_i(j) = j \cdot \frac{\theta_{Bi} - \theta_{Ai}}{n} \tag{8}$$

Now that we have these intermediary joint position steps, we can do a collision check on all of them. If one of them proves to be in collision, this means that the movement in between joint solutions A and B must also be in collision. Descartes should then give a warning to the user that collisions are detected, and the trajectory can not be executed.

Of course, the same discretization problem arises. The collision detection between trajectory points can only be guaranteed if the number of intermediary joint steps n goes to infinity. But for larger collision objects, one or more intermediary steps are enough to detect the collision.

Although we had the idea to implement this, it was not yet done or tested in this master thesis.

7.4 Smoother trajectories

The results from chapter 6 often show trajectories that have multiple jumps in joint space. These trajectories are not smooth enough to be executed on a real robot, because the ”point to point” movements of the robot to switch between these, suddenly changing, joint positions, could cause collisions.

There were two ideas we had to limit these jumps. Both are not yet tested.

To limit the amount of jumps the robot makes, an extra cost could be added based on the acceleration of the robot joints. This would then make the joint trajectories smoother, so more focus on smoother trajectories, instead of trajectories with the smallest total movement.

Another idea was to limit the maximum allowed change in tolerance value in between adjacent trajectory points. This would force the robot to slowly change into a new position, instead of suddenly jumping into the new position.

8 Conclusion

The Descartes software package proved to be a useful tool to generate trajectories for welding robots. However, due to its experimental nature, a lot of improvements can still be made to the software.
In this thesis a few improvements were proposed and tested:

- Activated collision detection between the robot and its environment.
- Created a library to visualize the trajectory points in RViz.
- Implemented code to calculate the deviations of toleranced frames.
- Added and tested the deviation cost. The deviation cost causes the welding torch to remain closer to its optimal path.
- Changed the rotational tolerances from Euler angle tolerances, to local frame tolerances. Causing the tolerances to be easier to define, and mentally visualize.

On top of that, a few problems arising with the software package were discovered, and solutions were explored:

- Huge amounts of RAM memory necessary to store the graph when multiple accurate tolerances are defined.

  An offered solution to this problem, is the use of iterative tolerances. Iterative tolerances can greatly lower the generation time and the amount of necessary RAM. The drawback is that the algorithm loses its probabilistic completeness.

- Robot model can jump through collision objects.

  A solution to this problem was proposed, but not tested. The solution consists of interpolating the generated joint solutions, and doing more collision checks on these interpolations.

The used robot model of the KUKA KR5 ARC robot was also improved.
When the correct improvements are added, we believe that the Descartes software package can be a useful and practical tool for trajectory generation for industrial robots.

The code used in this thesis is available online in the following repository:
https://github.com/Bart123456/lasrobot_ws
For GIT:
https://github.com/Bart123456/lasrobot_ws.git

Our code has also been used in a paper[33] about the Descartes software package.

References

[1] E. Demeester, M. Verheyen. (2016). *Smartfactory Project ACRO*, [Online]. Available: http://iiw.kuleuven.be/onderzoek/acro/smartfactory

[2] R. Madaan. (2016). *"descartes"*. [Online]. Available: http://wiki.ros.org/descartes.

[3] Jonathan Mey. (2015, April 3) *"descartes_trajectory"*. [Online]. Available: http://wiki.ros.org/descartes_trajectory.

[4] John Schulman. (2013.) *trajopt: Trajectory Optimization for Motion Planning – trajopt 0.1 documentation* [Online]. Available: http://rll.berkeley.edu/trajopt/doc/sphinx_build/html/.

[5] N. Ratliff, M. Zucker *et al.* (2009). *"CHOMP: Gradient Optimization Techniques for Efficient Motion Planning"*, [Online]. Available: http://www.ri.cmu.edu/pub/_files/2009/5/icra09-chomp.pdf

[6] ROS. *"About ROS"*. [Online]. Available: http://www.ros.org/about-ros/.

[7] ROS. *"Core Components"*. [Online]. Available: http://www.ros.org/core-components/.

[8] Karl Hansen. (2015, October 19). *"Packages"*. [Online]. Available: http://wiki.ros.org/Packages.

[9] Mike Purvis. (2014, August 22). *"ROS Indigo Igloo"*. [Online]. Available: http://wiki.ros.org/indigo.

[10] Hoang Giang. (2017, March 31). *"Documentation"*. [Online]. Available: http://wiki.ros.org/.

[11] Ken Conley. (2012, February 02). *"Nodes"*. [Online]. Available: http://wiki.ros.org/Nodes.

[12] Davet Coleman. (2014, October 12). *"urdf"*. [Online]. Available: http://wiki.ros.org/urdf.

[13] L. Joseph, Mastering ROS for Robotics Programming, Livery Place, Birmingham, Packt Publishing Ltd., 2015, ch.2 p.59-113,

[14] Acorn Pooley. (2013, May 20). *"srdf"*. [Online]. Available: http://wiki.ros.org/srdf.

[15] Dave Coleman. (2013, October 18). *"kdl_parser"*. [Online]. Available: http://wiki.ros.org/kdl_parser.

[16] jarvis Schultz. (2015, October 15). *joint_state_publisher_"*. [Online]. Available: http://wiki.ros.org/joint_state_publisher.

[17] Mithun Jacob. (2016, December 04). *robot_state_publisher_"*. [Online]. Available: http://wiki.ros.org/robot_state_publisher.

[18] Isaac Saito. (2015, Julie 19). *tf"*. [Online]. Available: http://wiki.ros.org/tf.

[19] Kamic Colo. (2015, May 12). *"roslaunch"*. [Online]. Available: http://wiki.ros.org/roslaunch.

[20] ROS Industrial *"Description"*. [Online]. Available: http://rosindustrial.org/contributors/.

[21] Ioan A. Sucan, Sachin Chitta. (2011). *"MoveIt!"*. [Online]. Available: http://moveit.ros.org/.

[22] M. M. L. E. K. Ioan A. ucan, *"The Open Motion Planning Library"*, IEEE Robotics and Automation Magazine, nr. 19, pp. 72-82, 2012.

[23] Rice University, *"Probabilistic Roadmap Method (PRM)"*, Kavraki Lab, [Online]. Available: http://www.kavrakilab.org/robotics/prm.html. [Geopend 21 12 2016].

[24] S. Lavalle, *"RRT Page: About RRT's"* [Online]. Available: http://msl.cs.uiuc.edu/rrt/about.html. [Geopend 21 12 2016].

[25] S. Lavalle, (2013, February 14) *"Rapidly Exploring Manifolds: when going from A to B aint easy"* [Online]. Available: https://mappingignorance.org/2013/02/14/rapidly-exploring-manifolds-when-going-from-a-to-b-aint-easy/l.

[26] J. Sibeyn, Chapter 10, 15 Februari 2005. [Online]. Available: http://users.informatik.uni-halle.de/ jopsi/dinf204/chap10.shtml. [Geopend 21 12 2016].

[27] William Woodall. (2017, May 11). *"Creating a workspace for catkin"*. [Online]. Available: http://wiki.ros.org/catkin/Tutorials/create_a_workspace.

[28] GVD Hoorn. (2016, December 20). *"urdf/XML/joint"*. [Online]. Available: http://wiki.ros.org/urdf/XML/joint.

[29] Tim Field, Jeremy Leibs, James Bowman. (2015, June 16). *"rosbag - ROS Wiki"*. [Online]. Available: http://wiki.ros.org/rosbag.

[30] The Matplotlib development team. (2017, May 10). *"Matplotlib: Python Plotting"*. [Online]. Available: https://matplotlib.org/.

[31] David Gossow, William Woodall. (2016, June 15). *"rviz - ROS Wiki"*. [Online]. Available: http://wiki.ros.org/rviz.

[32] Ioan Sucan. (2013, January 27). *"fcl - ROS Wiki"*. [Online]. Available: http://wiki.ros.org/fcl.

[33] J. De Maeyer et al, "Cartesian Path Planning for Arc Welding Robots: Evaluation of the Descartes Algorithm," presented at the Emerging Technologies And Factory Automation, Limassol, ETFA, Cyprus, 2017.

Appendix A Complete URDF

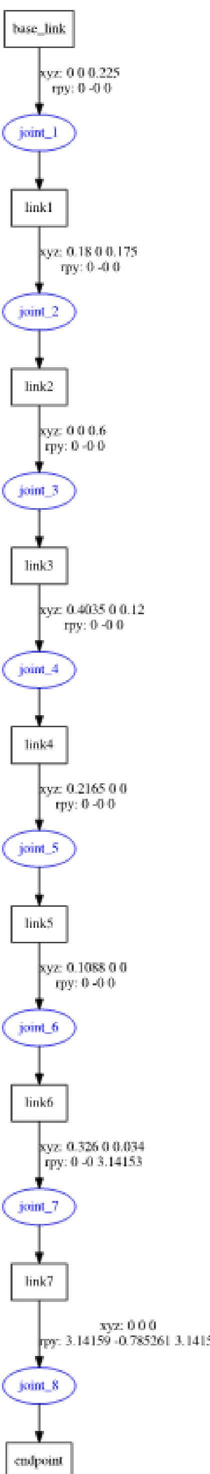
```
<robot name="robot">
<!-___________________________________COLORS____________________________________________-->
<material name="orange">
   <color rgba="1 0.5 0 1"/>
</material>

<!--___________________________________LINKS_____________________________________________-->
  <link name="base_link">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
            <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Baseb.stl"
        scale=".001 .001 .001"/>
          </geometry>
      <material name ="orange"/>
    </visual>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
            <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Baseb.stl"
        scale=".001 .001 .001"/>
          </geometry>
    </collision>
  </link>

  <link name="link1">
    <visual>
        <origin xyz="-0.0014 -0.003 0.018" rpy="0 0 3.141529" />
        <geometry>
            <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Link1b.stl"
        scale=".004 .004 .004"/>
          </geometry>
      <material name ="orange"/>
    </visual>
    <collision>
        <origin xyz="-0.0014 -0.003 0.018" rpy="0 0 3.141529" />
        <geometry>
            <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Link1b.stl"
        scale=".004 .004 .004"/>
          </geometry>
    </collision>
  </link>

  <link name="link2">
    <visual>
        <origin xyz="-0.18742 0 -0.400" rpy="0 0 0" />
        <geometry>
            <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Link2.stl"
        scale=".001 .001 .001"/>
          </geometry>
      <material name ="orange"/>
    </visual>
    <collision>
        <origin xyz="-0.18742 0 -0.400" rpy="0 0 0" />
        <geometry>
            <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Link2.stl"
```

```
      scale=".001 .001 .001"/>
        </geometry>
  </collision>
</link>

<link name="link3">
  <visual>
      <origin xyz="-0.180 0 -1.0" rpy="0 0 0" />
      <geometry>
          <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Link3.stl"
      scale=".001 .001 .001"/>
        </geometry>
    <material name ="orange"/>
  </visual>
  <collision>
      <origin xyz="-0.180 0 -1.0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Link3.stl"
      scale=".001 .001 .001"/>
        </geometry>
  </collision>
</link>

<link name="link4">
  <visual>
      <origin xyz="-0.5835 0 -1.12" rpy="0 0 0" />
      <geometry>
          <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Link4.stl"
      scale=".001 .001 .001"/>
        </geometry>
    <material name ="orange"/>
  </visual>
  <collision>
      <origin xyz="-0.5835 0 -1.12" rpy="0 0 0" />
      <geometry>
          <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Link4.stl"
      scale=".001 .001 .001"/>
        </geometry>
  </collision>
</link>

<link name="link5">
  <visual>
      <origin xyz="-0.800 0 -1.12" rpy="0 0 0" />
      <geometry>
          <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Link5.stl"
      scale=".001 .001 .001"/>
        </geometry>
    <material name ="orange"/>
  </visual>
  <collision>
      <origin xyz="-0.800 0 -1.12" rpy="0 0 0" />
      <geometry>
          <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Link5.stl"
      scale=".001 .001 .001"/>
      </geometry>
  </collision>
</link>

<link name="link6">
```

```
    <visual>
        <origin xyz="-0.9088 0 -1.12" rpy="0 0 0" />
        <geometry>
          <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Link6.stl"
        scale=".001 .001 .001"/>
          </geometry>
      <material name ="orange"/>
      </visual>
    <collision>
        <origin xyz="-0.9088 0 -1.12" rpy="0 0 0" />
        <geometry>
            <mesh filename="package://kuka_description/meshes/kuka_kr5_arc/visual/Link6.stl"
        scale=".001 .001 .001"/>
          </geometry>
    </collision>
  </link>

  <link name="link7">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
            <mesh filename="package://kuka_description/meshes/Welding_Torch/Assembly4_origin_tip3
                .stl"
        scale=".01 .01 .01"/>
          </geometry>
      <material name ="orange"/>
    </visual>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
            <mesh filename="package://kuka_description/meshes/Welding_Torch/Assembly4_origin_tip3
                .stl"
        scale=".01 .01 .01"/>
          </geometry>
    </collision>
  </link>

  <link name="endpoint"/>

<!--____________________________________JOINTS_________________________________________-->

  <joint name="joint_1" type="revolute">
    <parent link="base_link"/>
    <child link="link1"/>
    <origin xyz="0 0 0.225" rpy="0 0 0" />
      <axis xyz="0 0 -1" />
      <limit lower="-2.70526034" upper="2.70526034" effort="0" velocity="0" />
    <material name ="black"/>
  </joint>

  <joint name="joint_2" type="revolute">
    <parent link="link1"/>
    <child link="link2"/>
    <origin xyz="0.180 0 0.175" rpy="0 0 0" />
      <axis xyz="0 1 0" />
    <limit lower="-3.1415927" upper="1.13446401" effort="0" velocity="0" />
  </joint>

  <joint name="joint_3" type="revolute">
    <parent link="link2"/>
```

```
    <child link="link3"/>
    <origin xyz="0 0 0.600" rpy="0 0 0" />
    <axis xyz="0 1 0" />
    <limit lower="-0.2617994" upper="2.75762022" effort="0" velocity="0" />
  </joint>

  <joint name="joint_4" type="revolute">
    <parent link="link3"/>
    <child link="link4"/>
    <origin xyz="0.4035 0 0.120" rpy="0 0 0" />
    <axis xyz="-1 0 0" />
      <limit lower="-6.10865238" upper="6.10865238" effort="0" velocity="0" />
  </joint>

  <joint name="joint_5" type="revolute">
    <parent link="link4"/>
    <child link="link5"/>
    <origin xyz="0.2165 0 0" rpy="0 0 0" />
      <axis xyz="0 1 0" />
      <limit lower="-2.26892803" upper="2.26892803" effort="0" velocity="0" />
  </joint>

  <joint name="joint_6" type="revolute">
    <parent link="link5"/>
    <child link="link6"/>
    <origin xyz="0.1088 0 0" rpy="0 0 0" />
      <axis xyz="-1 0 0" />
      <limit lower="-6.10865238" upper="6.10865238" effort="0" velocity="0" />
  </joint>

  <joint name="joint_7" type="fixed">
    <parent link="link6"/>
    <child link="link7"/>
    <origin xyz="0.326 0 0.034" rpy="0 0 3.141529" />
  </joint>

  <joint name="joint_8" type= "fixed">
    <parent link="link7"/>
    <child link="endpoint"/>
    <origin xyz="0 0 0" rpy="0 3.92685325 0" />
  </joint>
</robot>
```

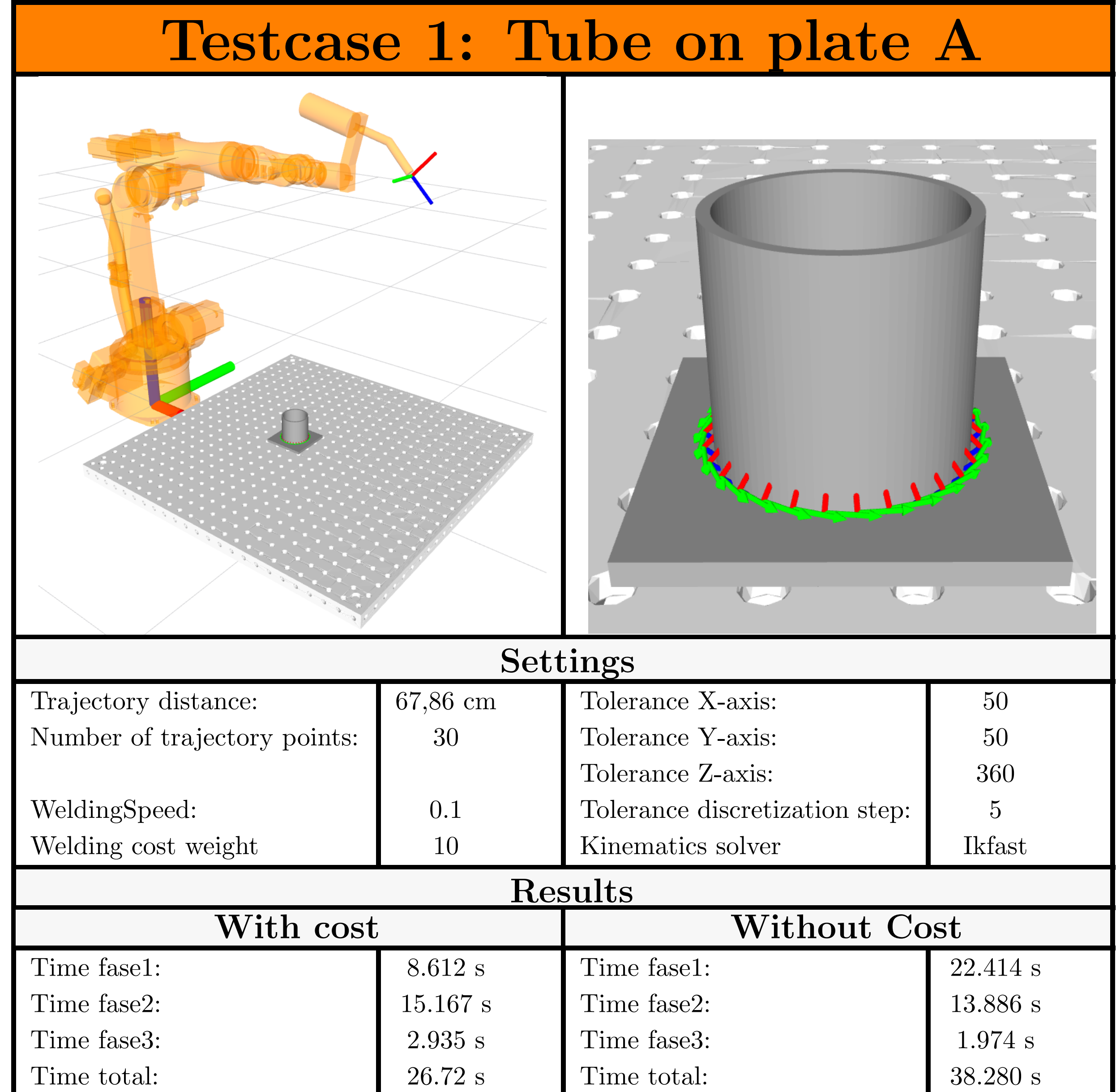
Appendix B URDF-tree



Appendix C   Python script for plotting data from bag-file

```
import rosbag
bag = rosbag.Bag('TOP_Cost.bag')
errorsX = []
errorsY = []
points = []
for topic, msg, t in bag.read_messages(topics=['angleErrorsX','angleErrorsY','trajectory']):
  if topic == 'angleErrorsX':
    errorsX = msg.data

  if topic == 'angleErrorsY':
    errorsY = msg.data

  if topic == 'trajectory':
    points = msg.points
bag.close()

bag2 = rosbag.Bag('TOP_noCost.bag')
errorsXnocost = []
for topic, msg, t in bag2.read_messages(topics=['angleErrorsX']):
  if topic == 'angleErrorsX':
    errorsXnocost = msg.data
bag2.close()

time = []
joint_positions = []
for point in points:
  time.append(point.time_from_start.secs + point.time_from_start.nsecs / 1e9)
  joint_positions.append(point.positions)

#remove 'NaN' values from lists (supposed to be 0)
count = 0
for error in errorsX:
  if error == 'nan':
    errorsX[count] = 0.0
  count += 1
count = 0
for error in errorsY:
  if error == 'nan':
    errorsY[count] = 0.0
  count += 1
count = 0
for error in errorsXnocost:
  if error == 'nan':
    errorsXnocost[count] = 0.0
  count += 1

import matplotlib.pyplot as plt
plt.plot(time, errorsX, 'k', time, errorsXnocost, 'r', linewidth=2.0)
plt.ylabel('Angle X Error [rad]')
plt.xlabel('Time [s]')
plt.grid('on')
plt.legend(['Without cost function', 'With cost function'])
plt.savefig("TOP_angleErrorX_IKfast.png")
```

Appendix D Testcase 1: tube on plate A



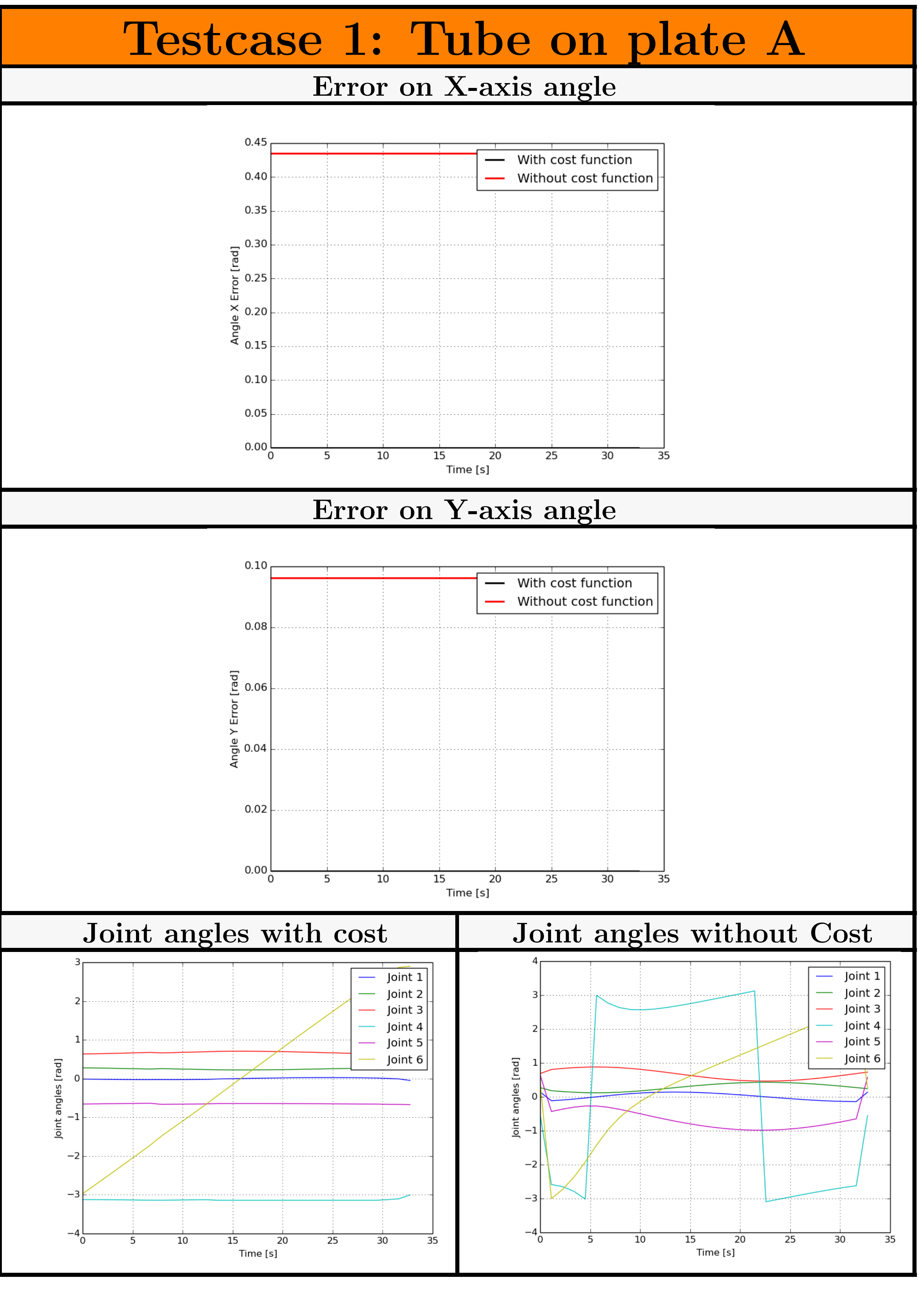
Testcase 1: Tube on plate A

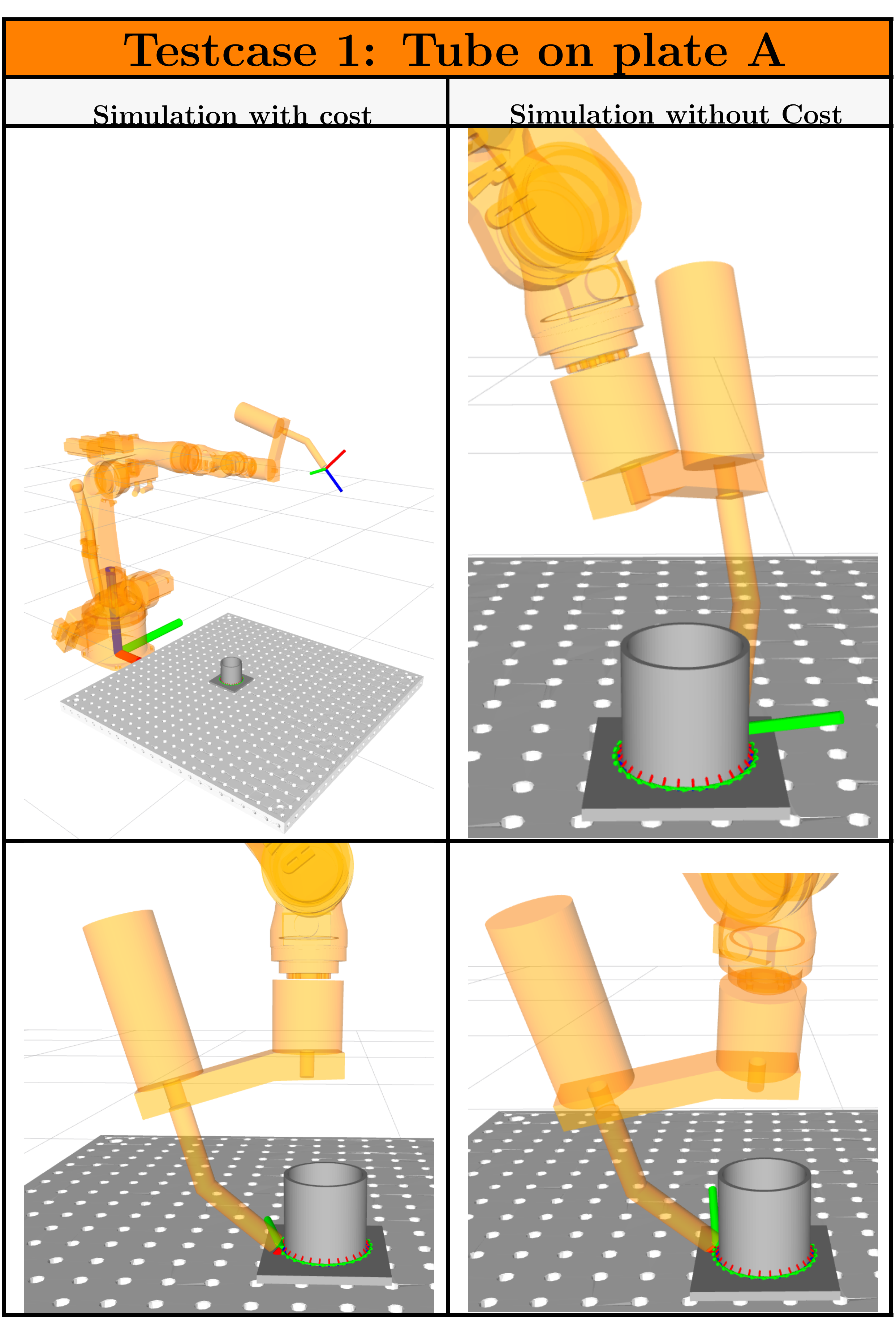
Settings

Trajectory distance:	67,86 cm	Tolerance X-axis:	50
Number of trajectory points:	30	Tolerance Y-axis:	50
		Tolerance Z-axis:	360
WeldingSpeed:	0.1	Tolerance discretization step:	5
Welding cost weight	10	Kinematics solver	Ikfast

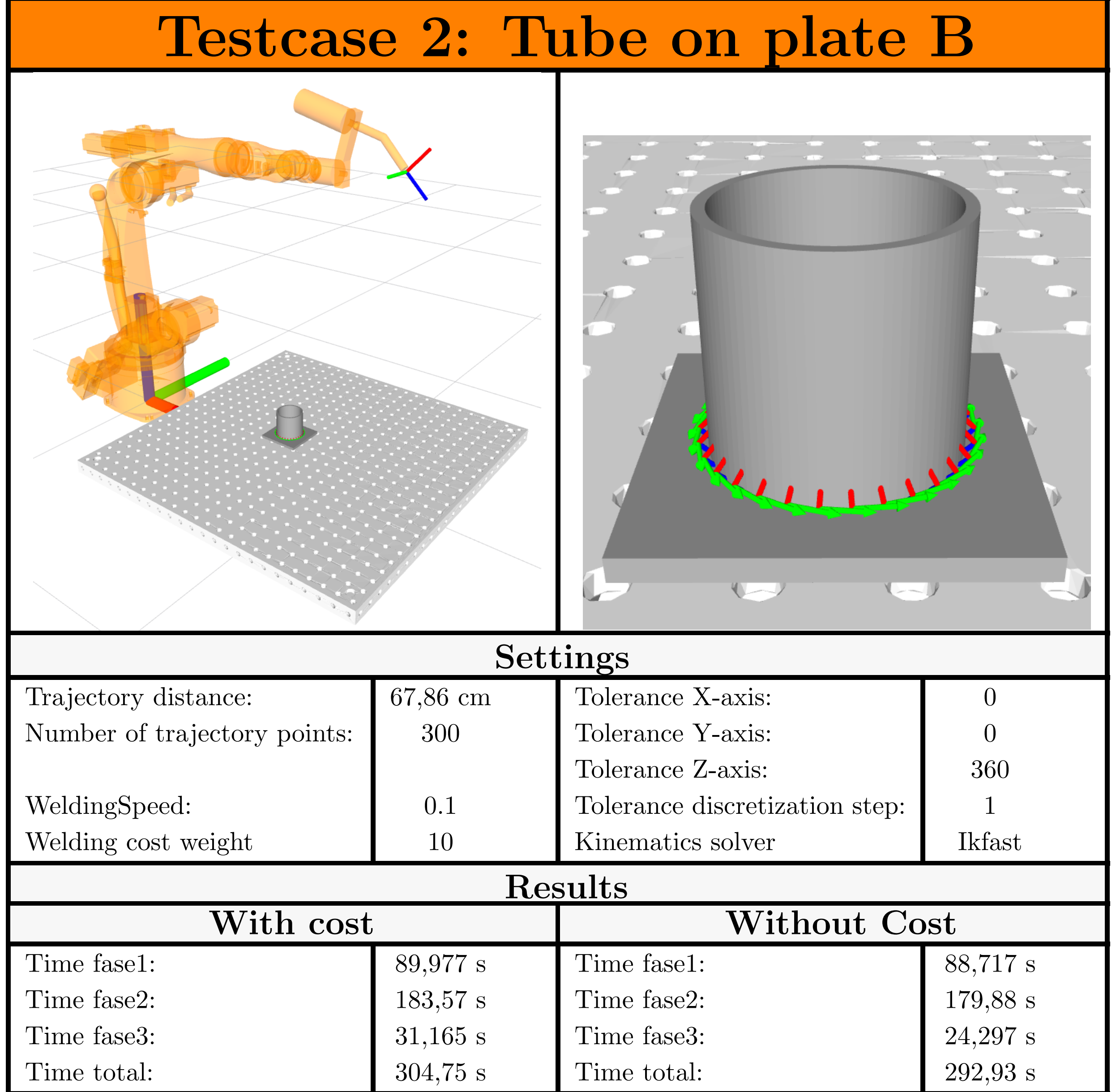
Results

With cost		Without Cost	
Time fase1:	8.612 s	Time fase1:	22.414 s
Time fase2:	15.167 s	Time fase2:	13.886 s
Time fase3:	2.935 s	Time fase3:	1.974 s
Time total:	26.72 s	Time total:	38.280 s


Testcase 1: Tube on plate A
Error on X-axis angle
With cost function
Without cost function
Angle X Error [rad]
Time [s]
Error on Y-axis angle
With cost function
Without cost function
Angle Y Error [rad]
Time [s]
Joint angles with cost
Joint angles without Cost
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6
Joint angles [rad]
Time [s]


Testcase 1: Tube on plate A
Simulation with cost
Simulation without Cost

Appendix E Testcase 2: tube on plate B



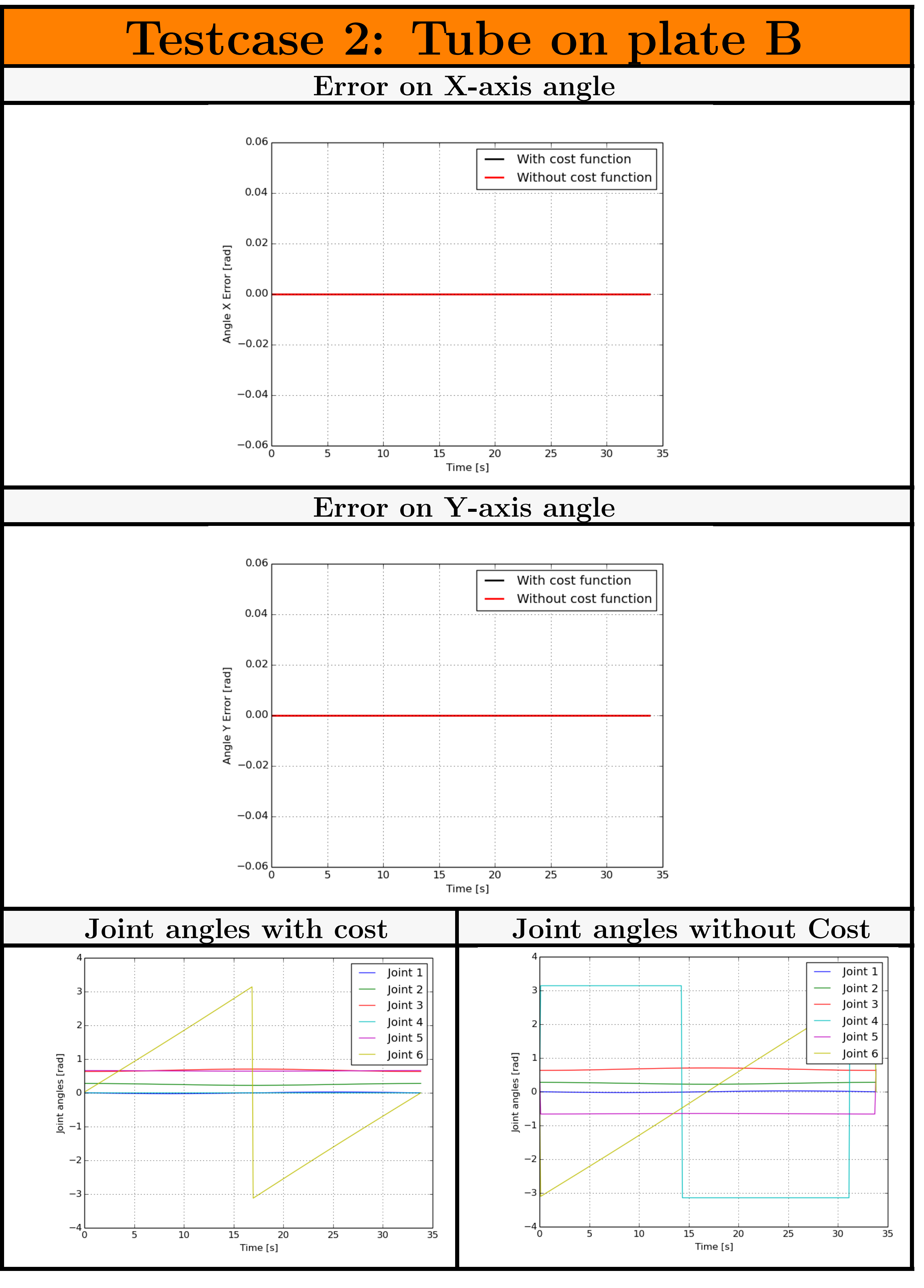
Testcase 2: Tube on plate B

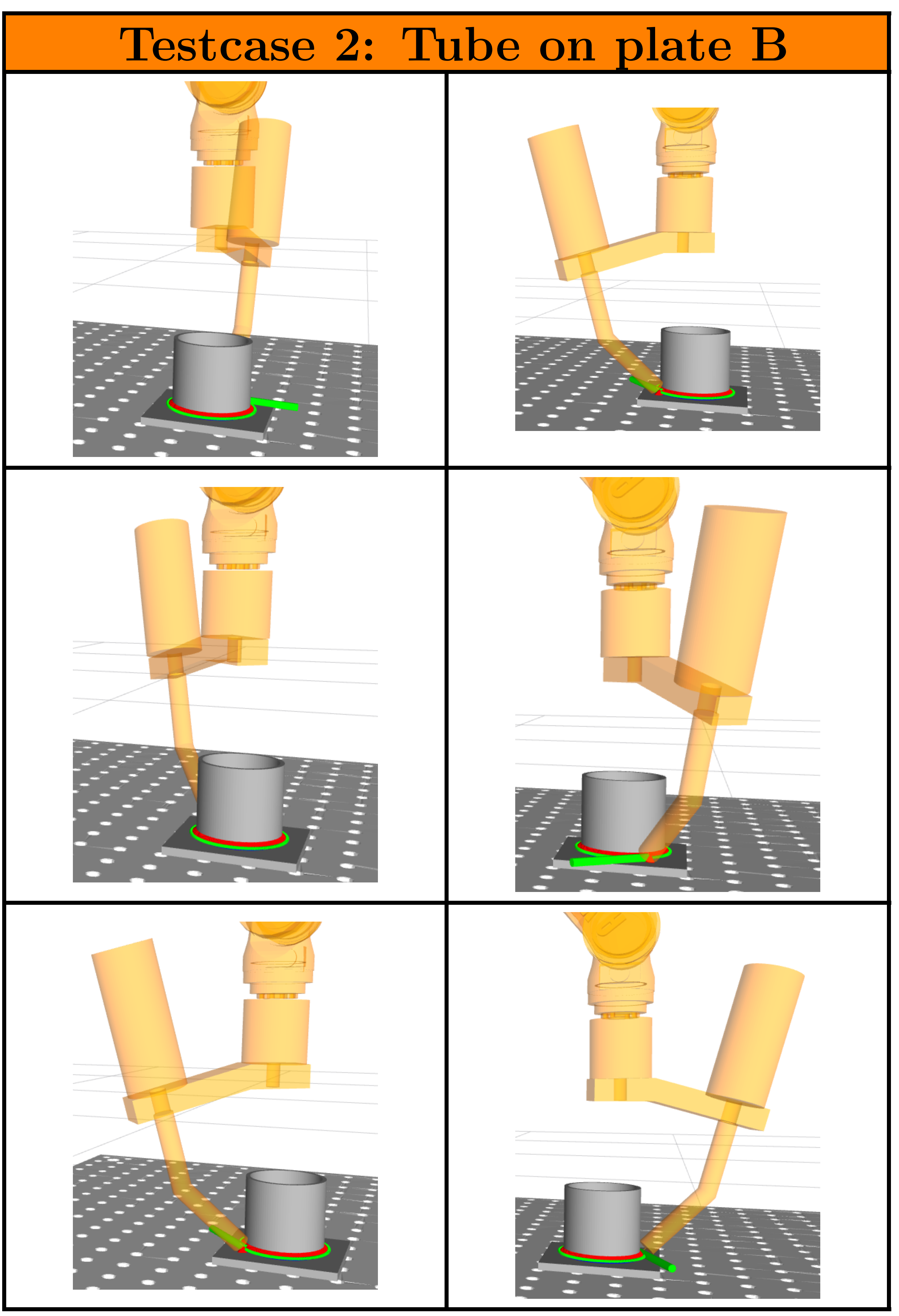
Settings

Setting	Value	Setting	Value
Trajectory distance:	67,86 cm	Tolerance X-axis:	0
Number of trajectory points:	300	Tolerance Y-axis:	0
		Tolerance Z-axis:	360
WeldingSpeed:	0.1	Tolerance discretization step:	1
Welding cost weight	10	Kinematics solver	Ikfast

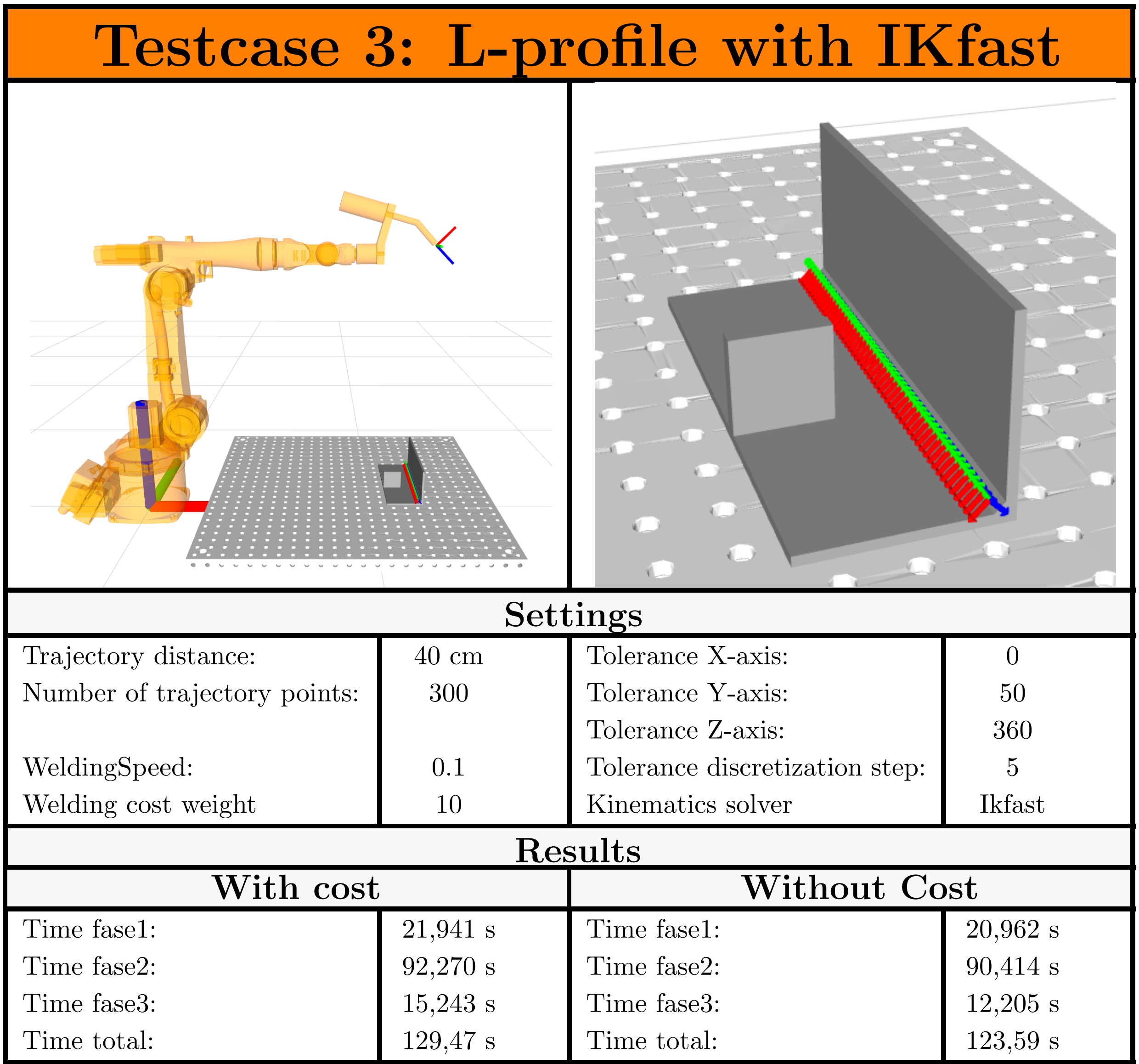
Results

With cost		Without Cost	
Time fase1:	89,977 s	Time fase1:	88,717 s
Time fase2:	183,57 s	Time fase2:	179,88 s
Time fase3:	31,165 s	Time fase3:	24,297 s
Time total:	304,75 s	Time total:	292,93 s


Testcase 2: Tube on plate B
Error on X-axis angle
Angle X Error [rad]
Time [s]
With cost function
Without cost function
Error on Y-axis angle
Angle Y Error [rad]
Time [s]
With cost function
Without cost function
Joint angles with cost
Joint angles without Cost
Joint angles [rad]
Time [s]
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6


Testcase 2: Tube on plate B

Appendix F Testcase 3: L-profile with IKfast



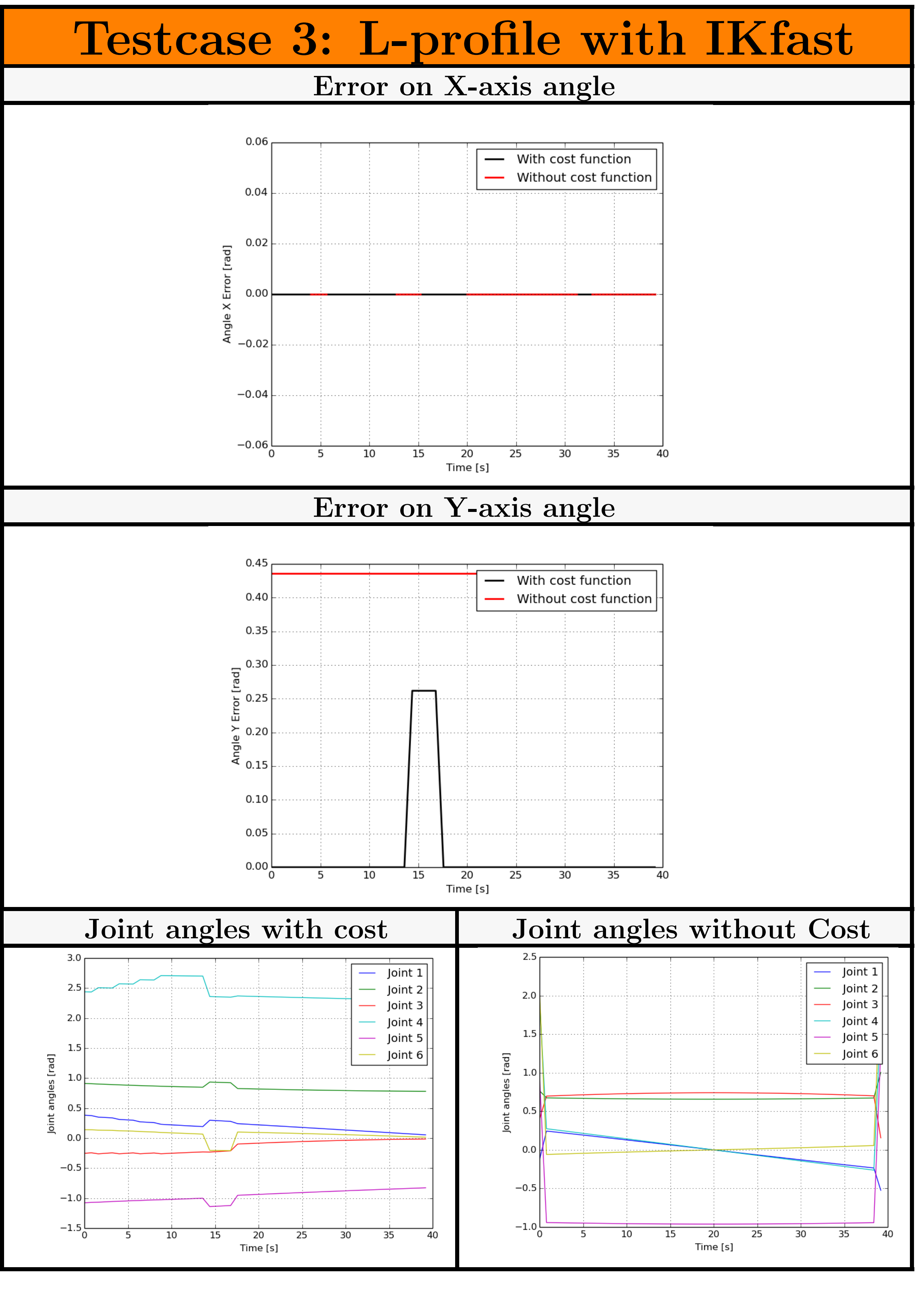
Testcase 3: L-profile with IKfast

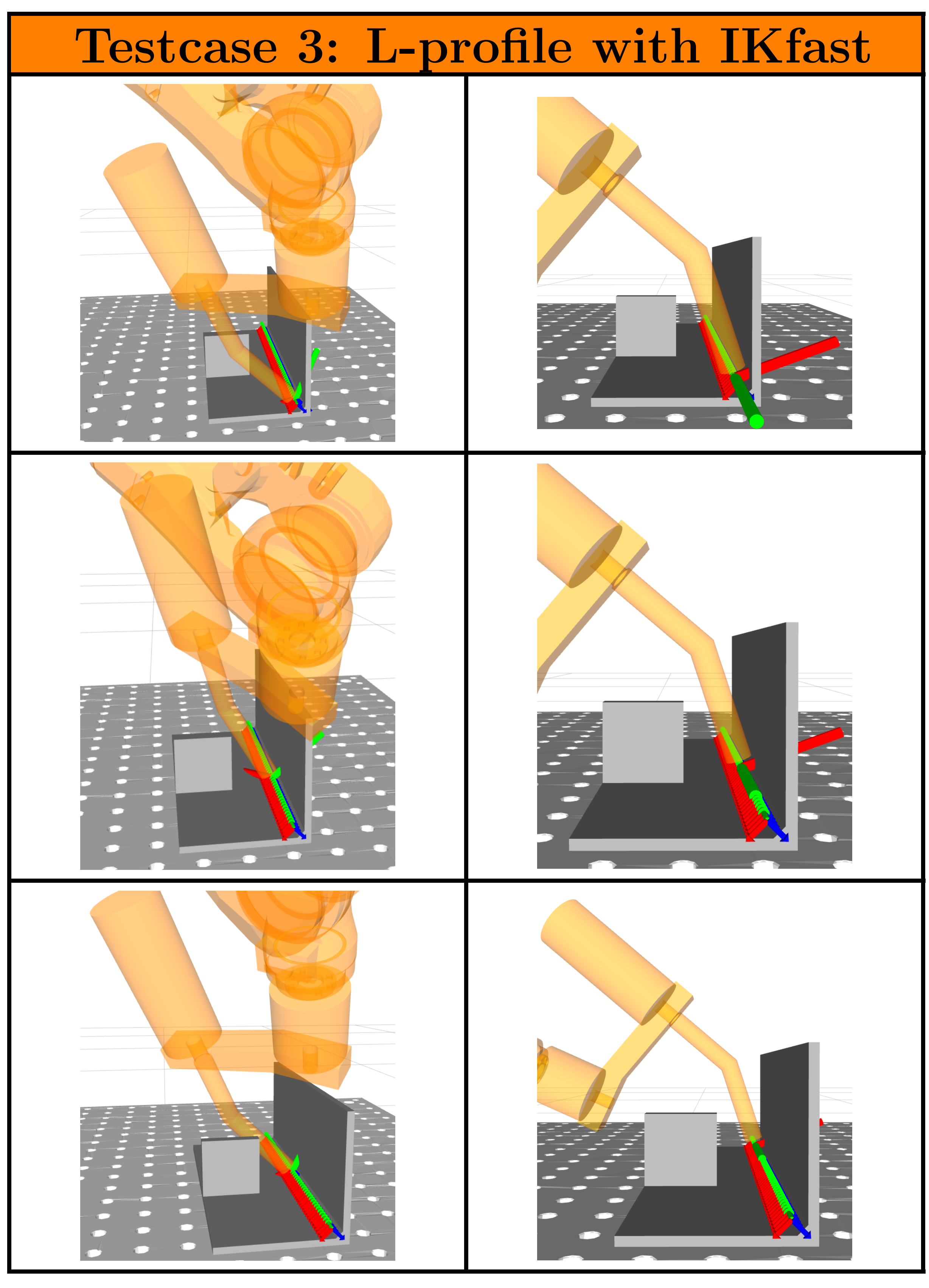
Settings

Trajectory distance:	40 cm	Tolerance X-axis:	0
Number of trajectory points:	300	Tolerance Y-axis:	50
		Tolerance Z-axis:	360
WeldingSpeed:	0.1	Tolerance discretization step:	5
Welding cost weight	10	Kinematics solver	Ikfast

Results

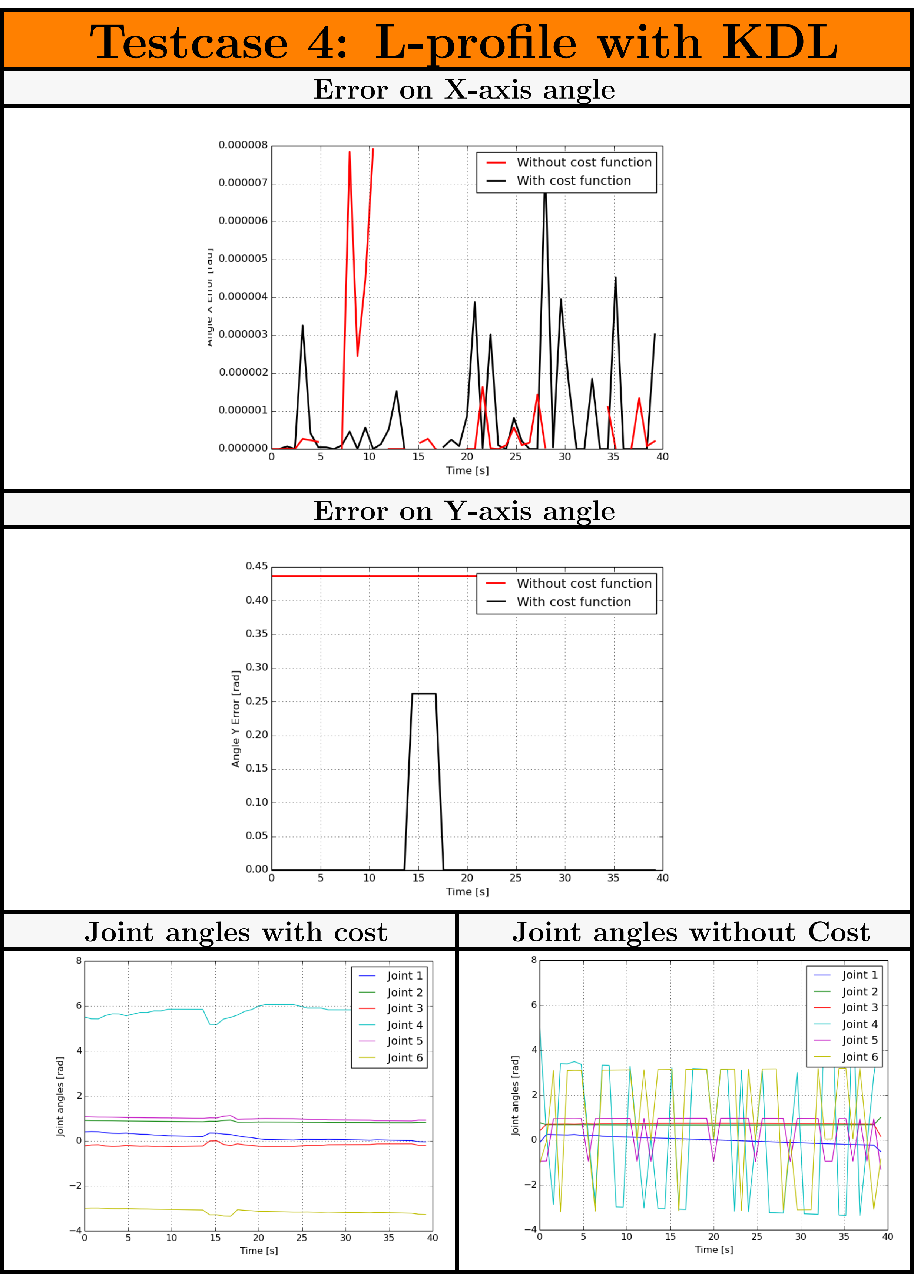
With cost		Without Cost	
Time fase1:	21,941 s	Time fase1:	20,962 s
Time fase2:	92,270 s	Time fase2:	90,414 s
Time fase3:	15,243 s	Time fase3:	12,205 s
Time total:	129,47 s	Time total:	123,59 s


Testcase 3: L-profile with IKfast
Error on X-axis angle
Angle X Error [rad]
Time [s]
With cost function
Without cost function
Error on Y-axis angle
Angle Y Error [rad]
Time [s]
With cost function
Without cost function
Joint angles with cost
Joint angles [rad]
Time [s]
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6
Joint angles without Cost
Joint angles [rad]
Time [s]
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6


Testcase 3: L-profile with IKfast

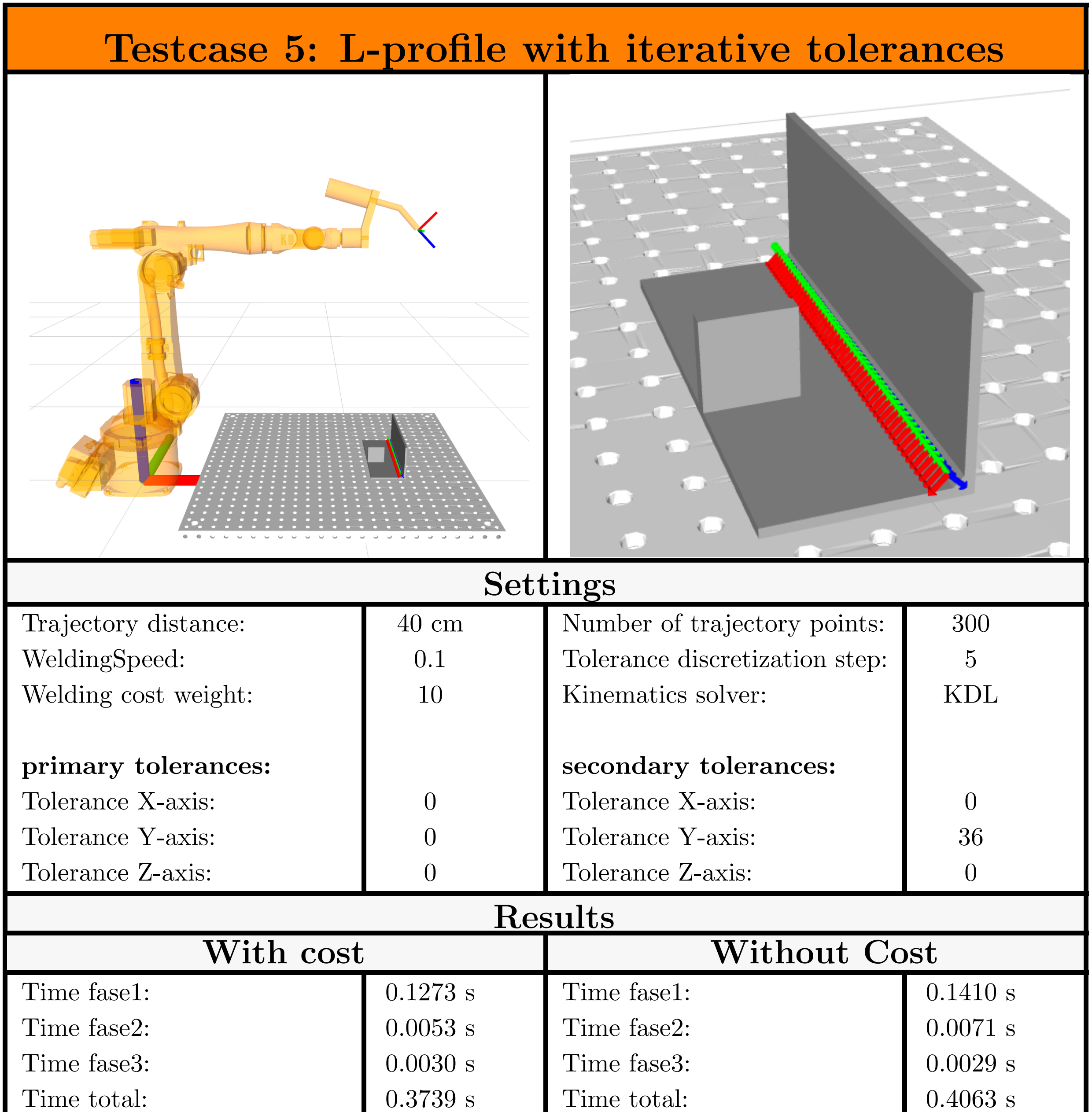
Appendix G Testcase 4: L-profile with KDL

Testcase 4: L-profile with KDL			
Settings			
Trajectory distance:	40 cm	Tolerance X-axis:	0
Number of trajectory points:	300	Tolerance Y-axis:	50
		Tolerance Z-axis:	360
WeldingSpeed:	0.1	Tolerance discretization step:	5
Welding cost weight	10	Kinematics solver	KDL
Results			
With cost		**Without Cost**	
Time fase1:	7955,0 s	Time fase1:	8114,6 s
Time fase2:	378,87 s	Time fase2:	380,71 s
Time fase3:	145,04 s	Time fase3:	97,43 s
Time total:	8479,3 s	Time total:	8593,2 s

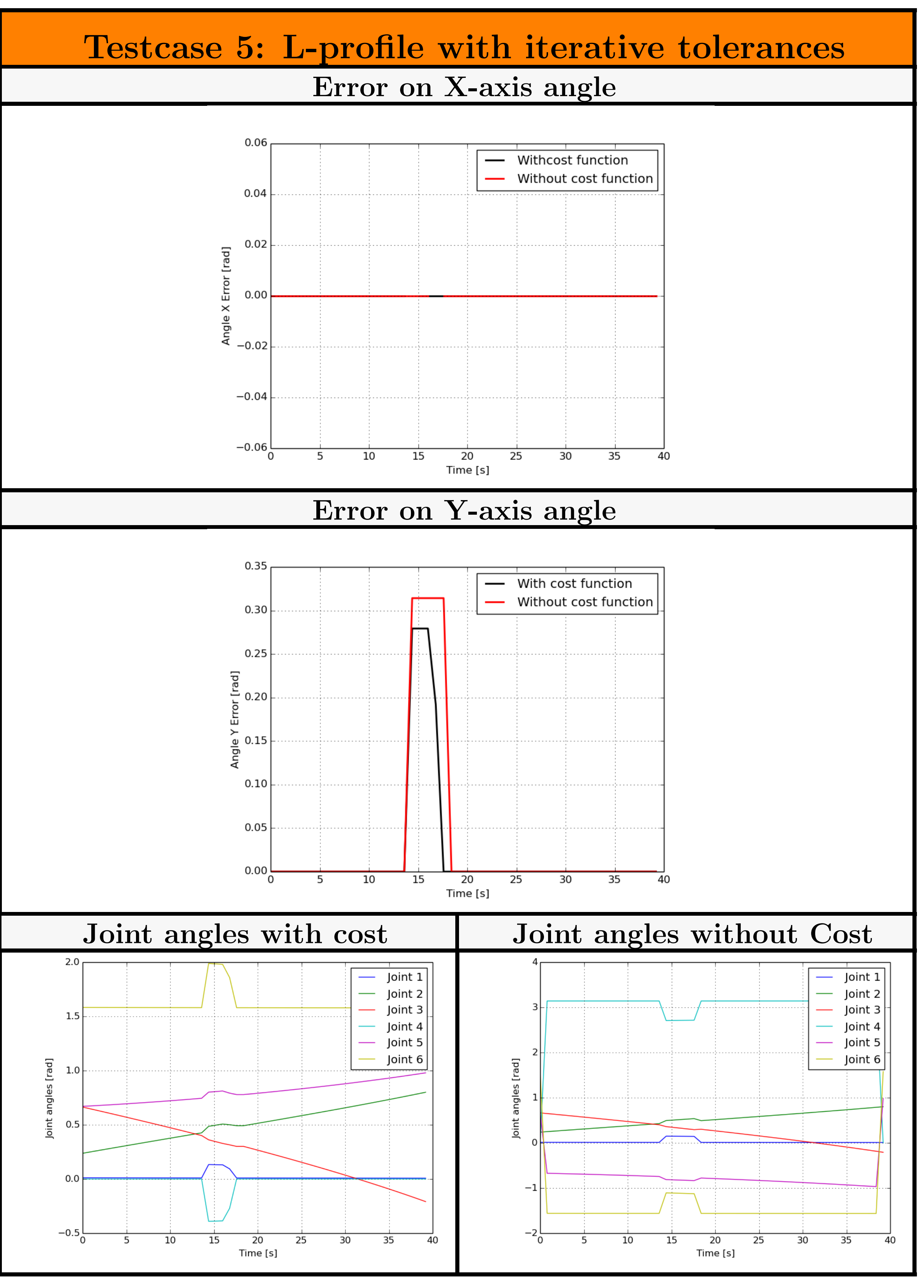

Testcase 4: L-profile with KDL
Error on X-axis angle
Without cost function
With cost function
Time [s]
Error on Y-axis angle
Without cost function
With cost function
Angle Y Error [rad]
Time [s]
Joint angles with cost
Joint angles without Cost
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6
Joint angles [rad]
Time [s]

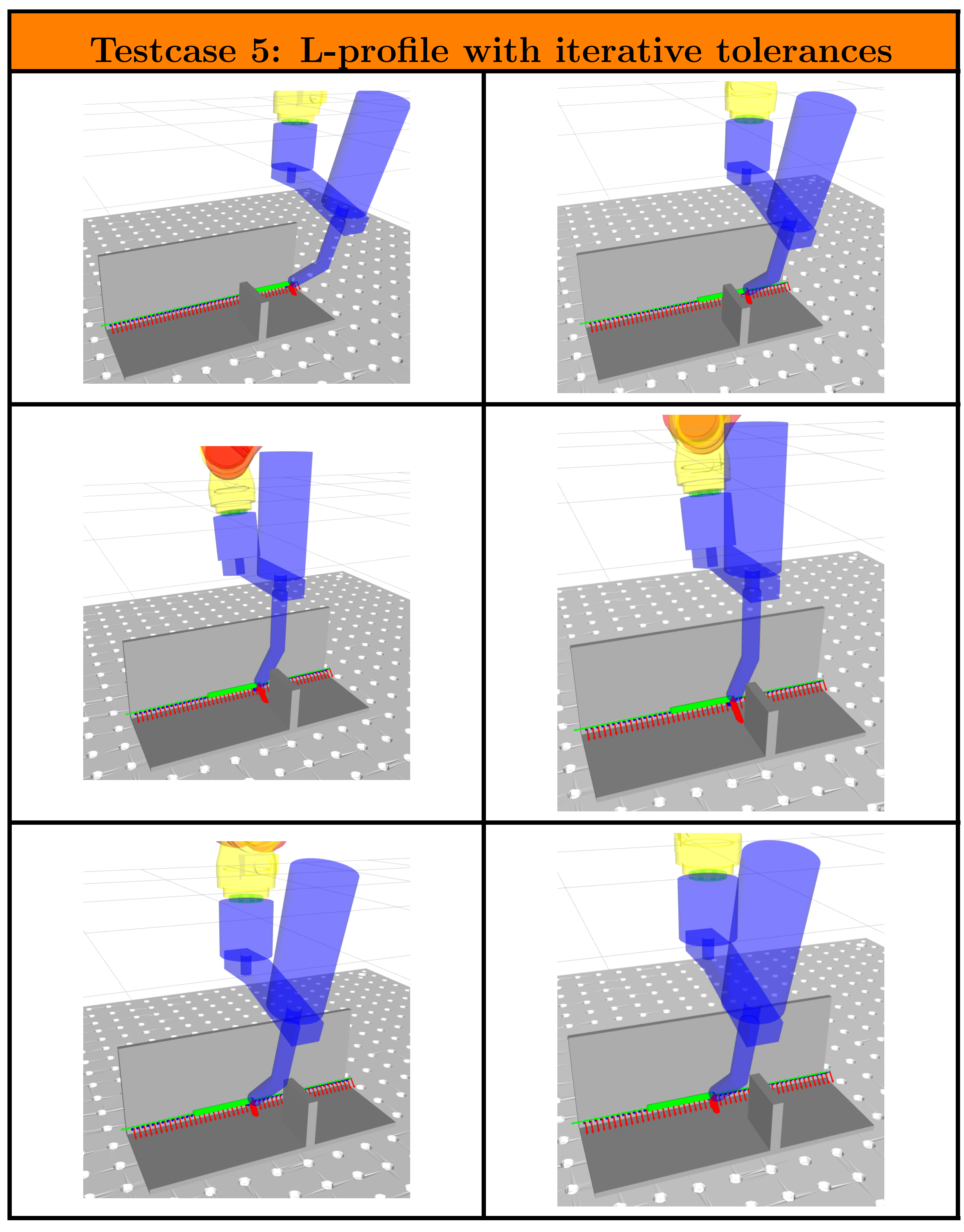
Testcase 4: L-profile with KDL

Appendix H Testcase 5: L-profile with iterative tolerances

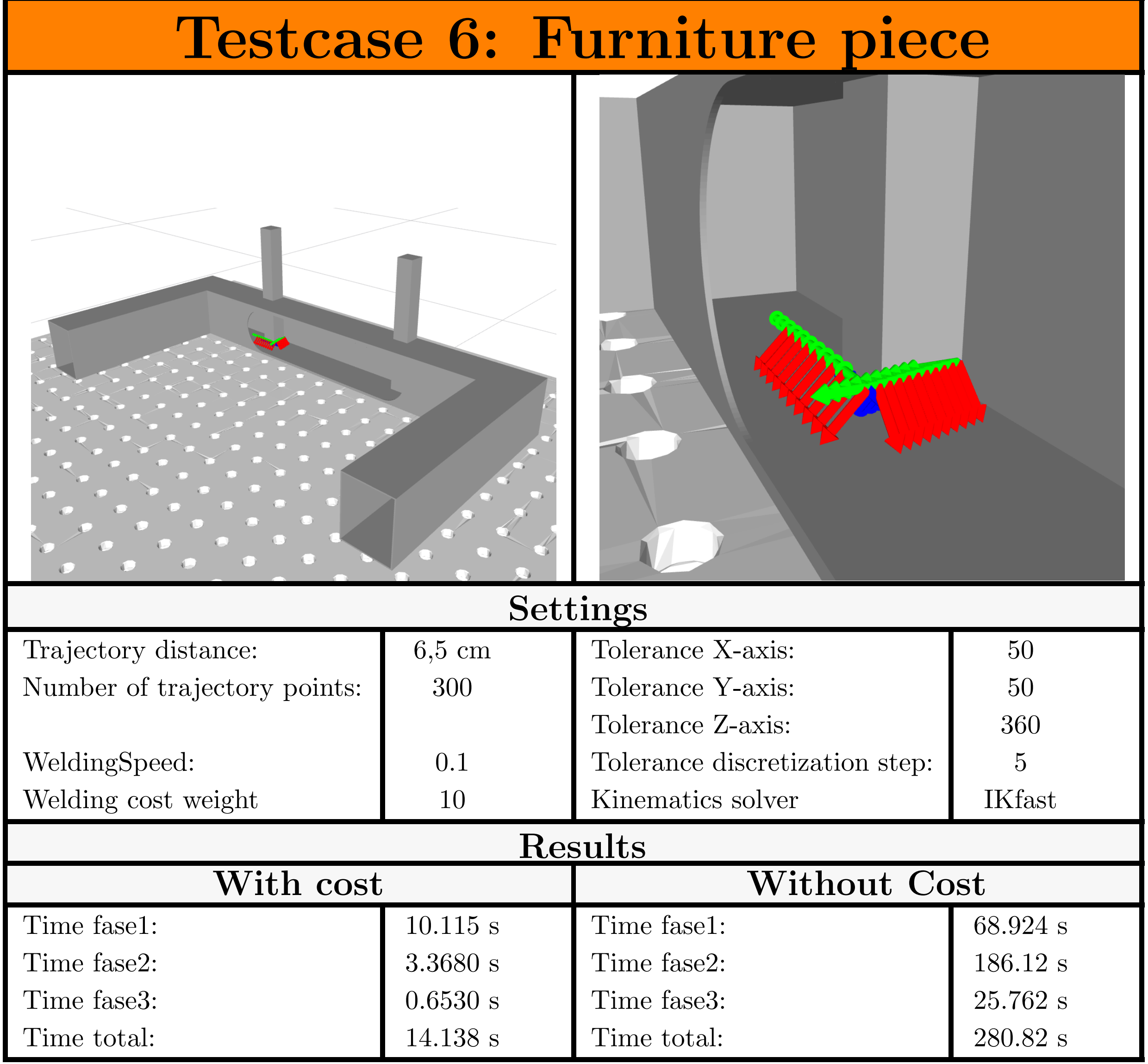


Testcase 5: L-profile with iterative tolerances			
Settings			
Trajectory distance:	40 cm	Number of trajectory points:	300
WeldingSpeed:	0.1	Tolerance discretization step:	5
Welding cost weight:	10	Kinematics solver:	KDL
primary tolerances:		**secondary tolerances:**	
Tolerance X-axis:	0	Tolerance X-axis:	0
Tolerance Y-axis:	0	Tolerance Y-axis:	36
Tolerance Z-axis:	0	Tolerance Z-axis:	0
Results			
With cost		**Without Cost**	
Time fase1:	0.1273 s	Time fase1:	0.1410 s
Time fase2:	0.0053 s	Time fase2:	0.0071 s
Time fase3:	0.0030 s	Time fase3:	0.0029 s
Time total:	0.3739 s	Time total:	0.4063 s


Testcase 5: L-profile with iterative tolerances
Error on X-axis angle
Withcost function
Without cost function
Angle X Error [rad]
Time [s]
Error on Y-axis angle
With cost function
Without cost function
Angle Y Error [rad]
Time [s]
Joint angles with cost
Joint angles without Cost
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6
Joint angles [rad]
Time [s]


Testcase 5: L-profile with iterative tolerances

Appendix I Testcase 6: Furniture piece



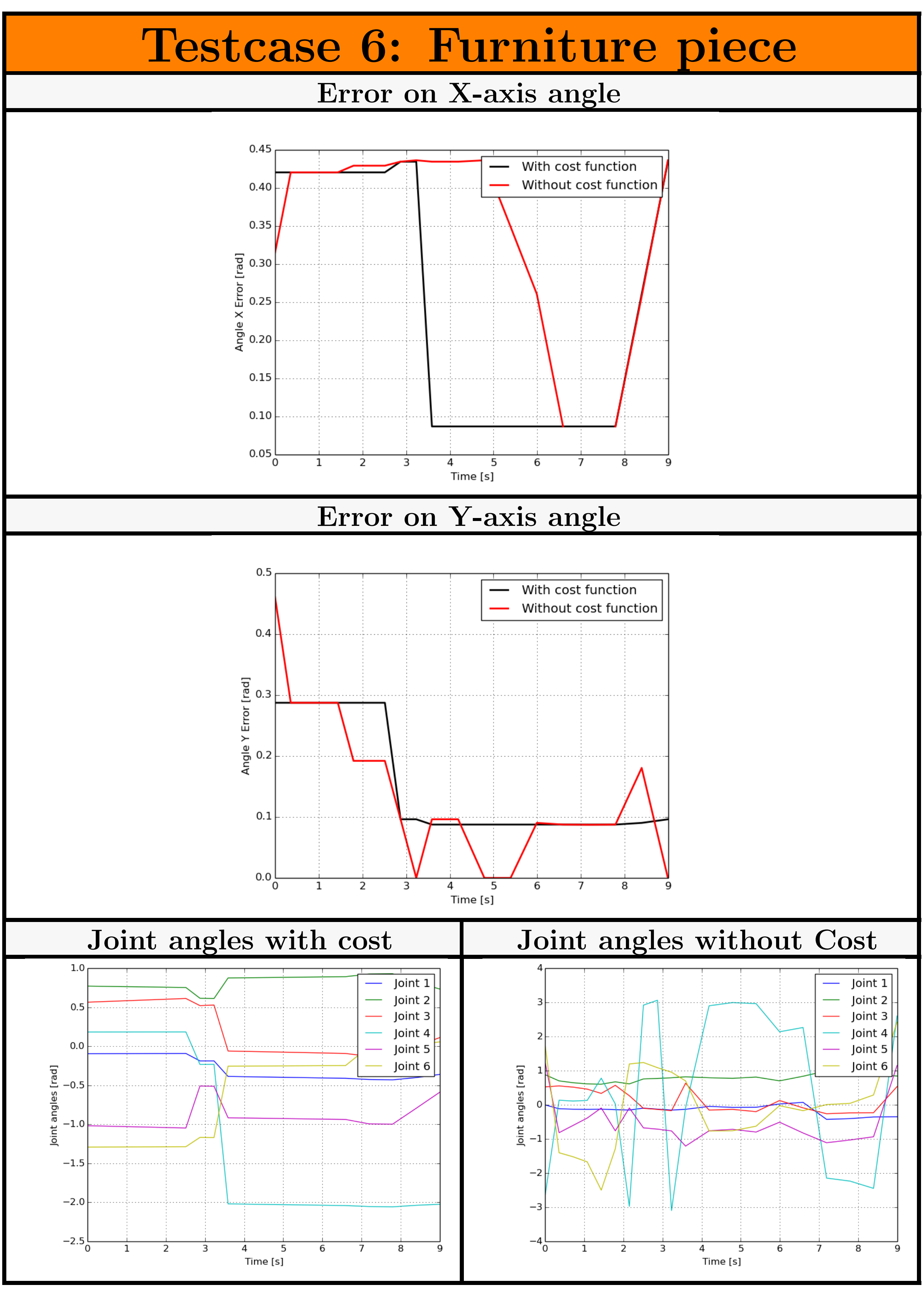
Testcase 6: Furniture piece

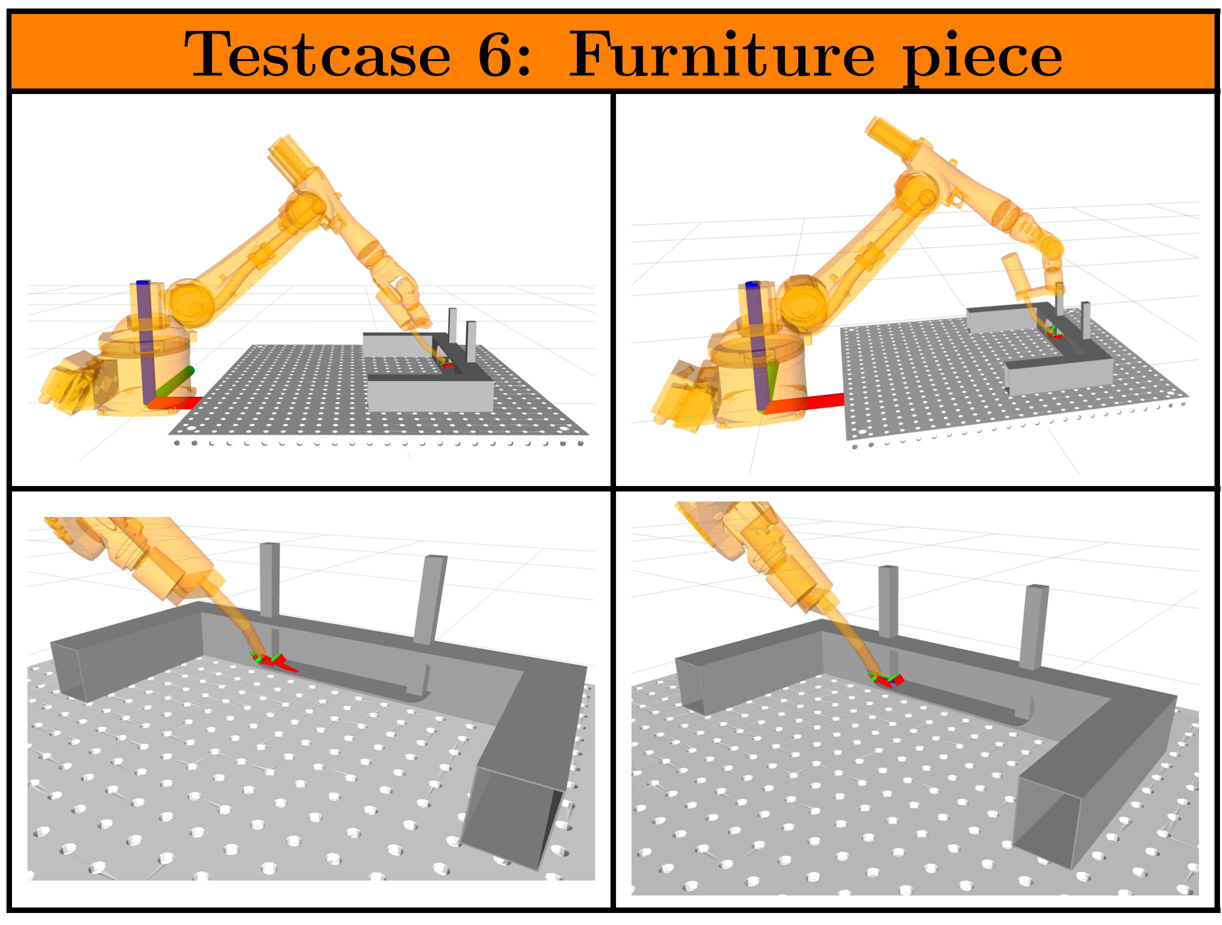
Settings

Trajectory distance:	6,5 cm	Tolerance X-axis:	50
Number of trajectory points:	300	Tolerance Y-axis:	50
		Tolerance Z-axis:	360
WeldingSpeed:	0.1	Tolerance discretization step:	5
Welding cost weight	10	Kinematics solver	IKfast

Results

With cost		Without Cost	
Time fase1:	10.115 s	Time fase1:	68.924 s
Time fase2:	3.3680 s	Time fase2:	186.12 s
Time fase3:	0.6530 s	Time fase3:	25.762 s
Time total:	14.138 s	Time total:	280.82 s


Testcase 6: Furniture piece
Error on X-axis angle
Angle X Error [rad]
Time [s]
With cost function
Without cost function
Error on Y-axis angle
Angle Y Error [rad]
Time [s]
With cost function
Without cost function
Joint angles with cost
Joint angles without Cost
Joint angles [rad]
Time [s]
Joint 1
Joint 2
Joint 3
Joint 4
Joint 5
Joint 6


Testcase 6: Furniture piece

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Collision-free trajectory generation for welding robots: analysis and improvement of the Descartes algorithm

Richting: **master in de industriële wetenschappen: energie-automatisering**
Jaar: **2017**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Moyaers, Bart **Vanvelk, Giel**

Datum: **7/06/2017**