Master Thesis

Czech
Technical
University
in Prague

F2

Faculty of Mechanical Engineering
Department of Mechanics, Biomechanics and Mechatronics

9 DOF Robot Motion Planning for Plastic Tank Welding

Ján Pravda

Supervisor: Ing. Petr Beneš, PhD.
Supervisor–specialist: Ing. Vladimír Smutný, PhD.
Field of study: Applied Sciences in Mechanical Engineering
Subfield: Mechatronics
August 2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Pravda** Jméno: **Ján** Osobní číslo: **475093**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav mechaniky, biomechaniky a mechatroniky**

Studijní program: **Aplikované vědy ve strojním inženýrství**

Specializace: **Mechatronika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Plánování pohybu robotu s 9 stupni volnosti pro svařování plastových nádrží

Název diplomové práce anglicky:

9 DOF robot motion planning for plastic tank welding

Pokyny pro vypracování:

1) Seznamte se s úlohou svařování plastových nádrží. Seznamte se s metodami plánování pohybu průmyslového robotu. Seznamte se se systémem ROS.
2) Implementujte kinematický, vizualizační a kolizní model robotu do prostředí ROS.
3) Navrhněte algoritmus pro plánování pohybu robotu s 9 stupni volnosti pro svařovací úlohu, která má 6 stupňů volnosti. Úlohu formulujte jako optimalizační problém.
4) Implementujte navržený algoritmus do systému ROS.
5) Proveďte experimenty v simulátoru a výsledky zhodnoťte.

Seznam doporučené literatury:

[1] Stejskal, V., Valášek, M.: Kinematics and Dynamics of Machinery, Marcel Dekker, Inc., New York 1996.
[2] Beschi, M. et al.: Optimal Robot Motion Planning of Redundant Robots in Machining and Additive Manufacturing Applications. Electronics 2019, Vol. 8, 1437.
[3] Erdős, G., Kovács, A., Váncza, J.: Optimized joint motion planning for redundant industrial robots, CIRP Annals, Vol. 65, No. 1, 2016, pp. 451-454
[4] https://www.ros.org/

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Petr Beneš, Ph.D. odbor mechaniky a mechatroniky FS

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Ing. Vladimír Smutný, Ph.D. robotické vnímání CIIRC

Datum zadání diplomové práce: **24.04.2023** Termín odevzdání diplomové práce: **14.08.2023**

Platnost zadání diplomové práce: ______________

Ing. Petr Beneš, Ph.D.	prof. Ing. Michael Valášek, DrSc.	doc. Ing. Miroslav Španiel, CSc.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání Podpis studenta

Acknowledgements

I want to thank my supervisor - specialist, Ing. Vladimír Smutný, PhD. from CIIRC CTU for providing me with a plenty of guidance and advice.

Declaration

I hereby declare that all of the work done on this thesis was my own and that I cited all used sources.

In Prague, 14th of August 2023

Abstract

This thesis deals with the development of a motion planning software for automatic welding of plastic tanks. Typically, such tanks are welded by hand, which is a long and arduous process, requiring years of experience. To improve this situation, a welding robot cell has been designed. The cell has 9 Degrees of Freedom, consisting of three external axes and an industrial robot. A welding extruder is mounted on the robot.

The motion planning problem can be described as repeatedly solving inverse kinematics of a robot for a given goal, while avoiding collision and respecting joint limits. The solved joint coordinates have to be smooth and continuous over the path.

The robot cell is redundant due to the 9 Degrees of Freedom. To get an unique solution for the inverse kinematics, we have to employ optimization, providing further constraints, in the form of optimization criteria, on the problem.

In the first chapters the welding problem is introduced, as well as the theoretical means to solve it (planning and optimization algorithms). In further chapters, the robot cell is described in detail and it's kinematic model is introduced. Robot Operating System (ROS), which is used for the planning, is discussed briefly.

The core of the work lies in defining various optimization criteria and implementing them in ROS. Results of planning on selected welds are presented and discussed.

Keywords: optimization, motion planning, robotic welding, OMPL, Ceres

Supervisor: Ing. Petr Beneš, PhD.
Ústav Mechaniky, Biomechaniky a Mechatroniky,
Technická 6,
Praha 160 00

Abstrakt

Tato diplomová práce se zabývá vývojem plánovacího software pro automatické svařování plastových nádrží. Typicky jsou tyhle nádrže svařovány ručně, co je dlouhý a náročný proces, vyžadující léta zkušeností. Pro zlepšení stávajíci situace byla navržená svařovací robotická buňka. Tato buňka má devět stupňů volnosti a skládá se ze tří externích os a průmyslového robota. Na robotu je nainstalován svářecí extrudér.

Problém plánování pohybu můžeme popsat jako opakované řešení úlohy inverzní kinematiky pro danou cílovou polohu, při respektování kloubových limitů a vyhýbání se kolizím. Výsledný průběh kloubových souřadnic musí být spojitý a hladký.

S devíti stupni volnosti je navrhovaná buňka redundantní. Pro nalezení jednoho řešení musíme využít optimalizace a zavedením optimalizačních kriterií vybrat jedno optimální řešení.

V prvních kapitolách práce je představen problém plánovaní pohybu, společně s teoretickými poznatky užitými k jeho řešení. V následujících kapitolách je detailně popsána robotická buňka a její kinematický model. Stručně je také shrnut Robot Operating System (ROS), který je využit pro samotné plánování.

Jádro práce spočíva v návrhu rúznych optimalizačních kriterií a jejich implementace v ROS-u. Prezentovány a diskutovány jsou výsledky plánovaní na několika vybraných svárech.

Klíčová slova: optimalizace, plánování pohybu, robotické svařování, OMPL, Ceres

Překlad názvu: Plánování pohybu s 9 stupni volnosti pro svařování plastových nádrží

Contents

Figures

Tables

Chapter 1
Introduction

Even though industrial robots have become a common occurrence in various fields and many factories are automated to a high degree, there are still areas, where bulk of the work is done by hand. Reasons for this may vary greatly: low production volumes won't use the abilities of an automated production line to it's full extent and the investment most likely won't pay off. Made-to-order products demand a flexible manufacturing process, whose development is rather complex. Luckily, developments in the areas of planning algorithms and machine vision have enabled more sophisticated systems to be applied to a wide variety of problems, supporting the inclusion of robots in processes previously thought to be too complex to handle.

In this work we are describing a 9 Degrees pf freedom (DOFs) robot cell system developed for welding of plastic tanks. We are focusing on the development of a planning algorithm used to generate paths for the welding robot. We will describe the problem in detail, starting from the physical construction, but concentrating mainly on the software: the data flow, inputs and outputs and algorithms used.

Central to the system is an optimization problem, as the mechanism is redundant and no unique solution can be found using pure inverse kinematics. The optimization criterion contains multiple weighted sub-criteria, that will be explained in detail. The actual implementation in ROS (Robot Operating System) will be thoroughly explained, providing a detailed manual for future users of the software. Lastly, experimental results will be discussed and critically assessed on whether the software is able to provide outputs applicable in real world on a production machine.

1.1 Motivation

This thesis is a part of a larger project, whose goal is to introduce automatic welding in an otherwise mostly manual manufacturing process of fabricating tanks (Fig. 1.1) and other equipment from plastic sheets joined by welding. The receiving company *STP Plast s.r.o. [4]* from Stráž pod Ralskem has a long experience in building them, but does most of the work by hand, using skilled labourers, whose training takes multiple years.

To correctly weld the boards together, such that they form a watertight

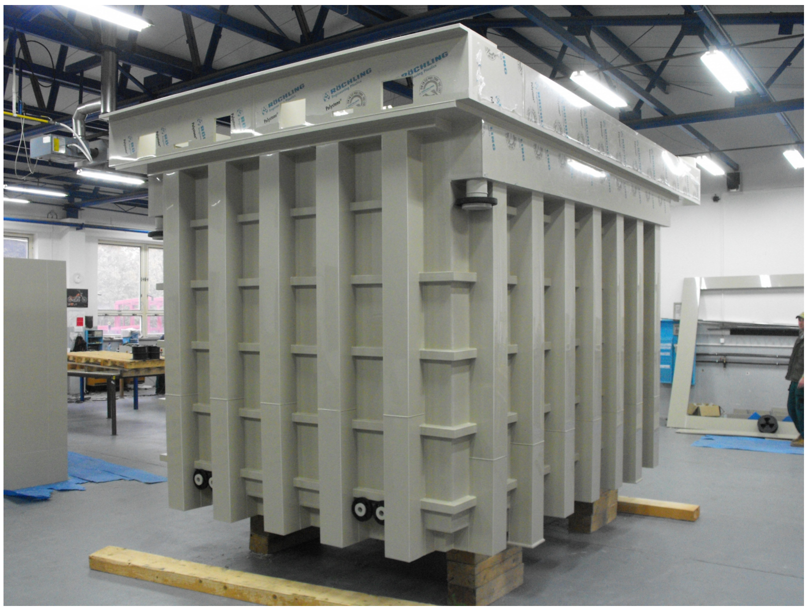

Figure 1.1: An example tank made at STP Plast s.r.o. [4]

seal that is structurally sound, the worker must guide the extruder (Fig. 1.2) at a precise angle and maintain constant speed along the weld. In many cases, the worker has to lie on their back, balance on a ladder or work in a otherwise unergonomic and physically demanding position. The extruder is a drill-shaped tool with a heat blower and a screw mechanism, feeding the hot end. The supplied filament is melted and forced through an opening into the weld. Especially difficult for execution are inside corner welds, where precise manipulation of the extruder is required and it's bulkiness can be a hindrance. Another source of discomfort while welding is heightened temperature. The air blown at the weld seam is heated to around 360° C, which the worker has to endure for extended periods of time.

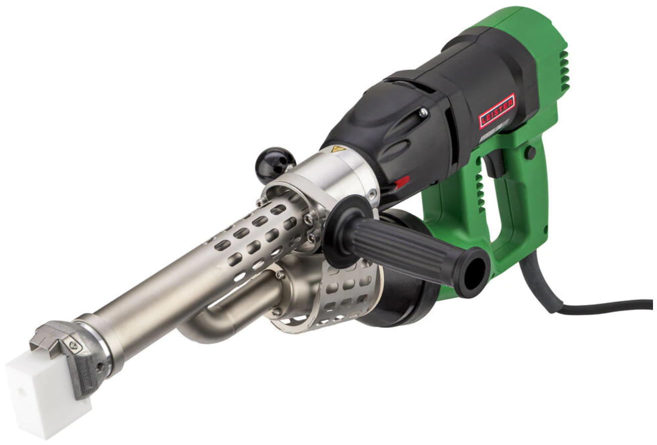

Figure 1.2: Handheld extruder [15]

To elevate the working conditions and possibly increase the effectiveness and automation level of the whole manufacturing process, a project, supported by the Technological Agency of the Czech Republic, was set up. As STP Plast themselves don't have any related experience in developing solutions for automation, further partners were invited: Companies *Alad CZ s.r.o.* and *triotec s.r.o.* provide expertise in developing the physical robot cell, control software and integration, while *CIIRC CTU* (Czech Institute of Informatics, Robotics and Cybernetics of the Czech Technical University) is responsible for the development of the planning software itself. [5]

The goal is to develop a robot cell capable of welding various tanks. The movement of the robot will be planned not according to fixed human-set paths, but rather based on a 3D CAD model of the tank. The planning software will consider technological parameters such as angle, speed, contact force or temperature and will generate optimal collision-free paths.[5]

Two robot cells are to be installed, one slightly larger than the other, but functionally identical. As a first step, simpler manual control software will be developed, where robot trajectories are calculated using analytical geometry based on preset weld paths. This however doesn't allow the full use of the 9 DOF cell and doesn't provide the flexibility in manufacturing various tank shapes and sizes, that is wished for. Nonetheless, it serves as an important test platform for learning about the welding process and testing out various technological parameters.

This work builds up on the foundation laid by a previous Master Thesis by Kamil Horný, who developed a basic version of a planning software for the project. [25]

1.2 Problem description

In it's core the problem centers around finding an optimal path for the robot cell based on input weld positions. The robot has 9 DOFs. That makes the whole mechanism redundant and brings a whole host of challenges with it. But redundancy has also it's advantages, such as extended working space. We will try to use up their full potential.

This thesis concerns itself mainly with the software side of the project. Here, the goal is to build a planning software that can use a 3D CAD model of the tank and a description of the welds on it to generate a continuous collision-free path.

The planner will be solving inverse kinematics of the robot cell: based on a given goal position we want to know the joint coordinates. Because of redundancy, there's an infinite amount of possible solutions this problem. To find a unique solution we need to impose further constaints: optimization criteria and solve the optimization problem.

Another important change coming with this project will be in the area of manufacture planning and workflow. Until now, the designers only design the tank and provide the drawings to the workers. The workers together with the workshop leader must have enough experience to assess how the weld should be made, i.e. what speeds, temperatures, materials and manufacturing processes are to be used. The technological knowledge of the welding process is therefore carried mainly by the workers, not by the designers.

This will change though, as the planning process will be tightly integrated with the design and will occur in the designer's offices. It will place additional requirements on the designers, who will need to learn all the knowledge around the welding process, because the designers will be the ones determining the welding parameters.

Because of this integration initiative, the software will not only contain the planning algorithm. It needs to fit in the entire manufacturing workflow. That means defining interface points and data types for communicating with other software.

Based on the preceding problem description, following goals of this thesis were defined:

- Familiarize yourself with topic of welding plastic tanks. Familiarize yourself with planning methods for industrial robots. Familiarize yourself with ROS.
- Implement kinematic, visualization and collision robot model into ROS
- Propose a planning algorithm for a robot with 9 DOFs for a problem of welding plastic tanks that has 6 DOFs. Formulate the problem as an optimization problem.
- Implement the proposed algorithm in ROS.
- Perform experiments in a simulator and evaluate the results.

Chapter 2

State of the art

2.1 Sampling-based motion planning

The problem in this project corresponds to a classical motion planning problem: Find a collision-free path between the start and end while respecting all of the robot constraints. While seemingly easy and intuitive for a human, these problems pose a significant challenge to a computer, increasing with the number of DOFs [32].

There are various approaches to solving this problem, with one of the most common being sampling-based motion planning. Instead of trying to find a solution in the continuous state space (space of all possible robot configurations), this method depends on a finite amount of random uniformly-sampled states (robot configurations), which it tries to connect these with a collision-free path. Most of these methods offer probabilistic completeness, i.e. the probability of finding a solution converges to one as the number of samples tends to infinity. However, these methods can't recognize a problem with no solution [32].

From the myriad of available libraries and software, we're focusing on *OMPL* (Open Motion Planning Library). It is a popular, well-tested library, which implements many state-of-art planning algorithms.

The goal of *OMPL* is to be as flexible as possible. That's why it doesn't implement problem-specific functionalities, such as collision-checking or the robot model. It's API provides all of the core components required to solving a planning problem:

ProblemDefinition Defines a start and a goal state and an optimization objective (if given).

StateSampler Provides methods for uniform and Gaussian sampling of the state space.

StateValidityChecker A framework for checking the state validity, i.e. that it's collision-free and fulfills all of the constraints. The user has to provide the implementation.

MotionValidator Tasked with determining if the path between two states consists of only valid states. *OMPL* uses interpolation between two states to determine the motion. Only a subset of states on this path are checked for validity.

OMPL provides classes wrapping the most common used routines (e.g. SimpleSetup) and predefined methods for common state spaces. It is also capable of optimal planning, but it depends on the planner used [32].

2.1.1 Algorithm

OMPL implements a whole host of planners, many of them a variation of the two most common ones: *PRM* (Probabilistic Roadmaps) and *RRT* (Rapidly-exploring Random Trees) [32]. Because *PRM* is suited primarily for multi-query applications (a map of the space is constructed and then it can be queried for paths with various start and end states) [26], we've decided to use *RRT*.

As the name suggest, *RRT* attempts to build a tree by sampling random configurations and attempting to join them to the existing tree [26]. Algorithm 1 and Fig. 2.1 show the basic principle.

Algorithm 1 RRT [26]

```
 1: $V \leftarrow \{x_{init}\}; E \leftarrow \emptyset$
 2: for $i = 1, \ldots, n$ do
 3:     $x_{rand} \leftarrow$ SampleFree$_i$;
 4:     $x_{nearest} \leftarrow$ Nearest$(G = (V, E), x_{rand})$;
 5:     $x_{new} \leftarrow$ Steer$(x_{nearest}, x_{rand})$;
 6:     if ObstacleFree$(x_{nearest}, x_{new})$ then
 7:         $V \leftarrow V \cup \{x_{new}\}; E \leftarrow E \cup \{(x_{nearest}, x_{new})\}$;
 8:     end if
 9: end for
10: return $G = (V, E)$;
```

The algorithm starts with an initial configuration x_{init}, which is represented by a vertex in the graph. Then, for n iterations, or until a point in the goal region is found, a simple procedure is repeated: Firstly, a random configuration x_{rand} is sampled. It is then checked, whether it belongs to the free state space, i.e. the configuration is not in collision. Then a nearest vertex already in the graph to the new configuration is found and an attempt to connect them is made. If successful, the algorithm makes a step x_{new} towards the new random configuration (*Steer*) [26]. There are three possible outcomes: the step reaches the configuration, the step doesn't reach x_{rand} but is successful or the step is rejected, because it doesn't lie in the free state space anymore (also called *Trapped*) [27].

There are various modifications to the original *RRT* algorithm. *RRT** doesn't simply take the nearest vertex and connects it with the new vertex, it first searches in a region around x_{new} for other possible vertexes to connect to. From these vertices, the algorithm picks that vertex, whose connection to x_{new} gives a path with minimal cost according to some criterion. The

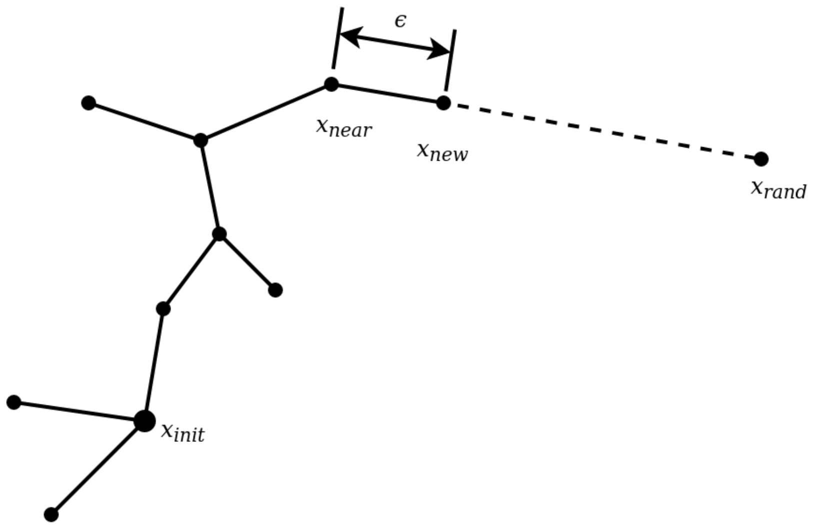

Figure 2.1: RRT expansion

algorithm is also able to rewire the tree, i.e. change the connection of existing vertices, if the path through the new vertex would have a lower cost [26].

An approach with two expanding trees, one from the start and one from the goal is used in the *RRT-Connect* algorithm. The trees are expanding simultaneously while attempting to connect to each other. The disadvantage here is that the goal state has to be known, instead of needing only the workspace coordinates of the end-effector [27].

2.2 Non-linear optimization problem

While finding a minimum of a function can often be very simple, in many practical applications the complexity of the problem doesn't allow the use of classical analytical methods. This happens mainly if the function is strongly non-linear, not smooth or continuous or impossible to write in a closed form. One way to solve such problem is using an optimization algorithm. Optimization has also uses in curve-fitting, kinematic synthesis and many more. Described here are the workings of a optimization solver.

Ceres is a multi-purpose non-linear least-squares solver, which can solve problems in the form [23]

$$\begin{aligned} \min_{\mathrm{x}} \quad & \frac{1}{2}\sum_i \rho_i \left(||f_i(x_{i_1},\ldots,x_{i_k})||^2\right) \\ \text{s.t.} \quad & l_j \le x_j \le u_j \end{aligned} \tag{2.1}$$

where f_i is the cost function depending on the *Parameter Blocks* (groups of parameters) $x_{i_1},\ldots x_{i_k}$. The function ρ_i is the *Loss Function* used to reduce the influence of outlier data. This function is not applicable to our problem and is shown only for completion. Finally, l_j and u_j are the lower and upper bounds on the parameter block x_j [23].

2.2.1 Algorithm

For the purposes of optimization, the problem in 2.1 can be simplified to [23]

$$\begin{aligned} &\arg\min_{\mathrm{x}} \quad \frac{1}{2}||F(x)||^2 \\ &\text{s.t.} \quad L \leq x \leq U \end{aligned} \tag{2.2}$$

where $x \in \mathbb{R}^k$, $F(x) = [f_1(x), \ldots, f_m(x)]^T$ and L and U are the lower and upper bounds on x. Next, we'll define the Jacobian as a $m \times n$ matrix with the elements $J_{ij}(x) = \frac{\partial f_i}{\partial x_j}$ and the gradient as a vector $g(x) = J(x)^T F(x)$ [23].

As is common with numerical methods, the idea behind the solution to non-linear optimization problems lies in repeatedly solving an approximation to the original problem and iteratively improving x by the means of a correction vector Δx. The approximation can be a simple linearization: $F(x+\delta x) = F(x) + J(x)\Delta x$, which transforms the Problem 2.2 to a sequence of minimizations of [23]

$$\min_{\Delta \mathrm{x}} \frac{1}{2}||F(x) + J(x)\Delta x||^2 \tag{2.3}$$

This however, doesn't always lead to a convergence, because the step size depends on the steepness of the function in x and it can easily overstep the local minimum. Thus we want to control the size of the step (*Line Search* methods) or the area, where the approximation can still be considered reliable (*Trust Region* methods). Currently, *Ceres* can only handle *Line Search* methods without constraints, leaving *Trust Region* the only option. The Algorithm 2 shows how *Trust region* algorithm works. [23]

In Algorithm 2, μ is the trust region radius, $D(x)$ defines a metric on the domain of $F(x)$ and ρ determines how well does the approximation represent the change in value of $F(x)$. Depending on ρ (i.e. the quality of the approximation) the radius of the trust region gets increased or decreased [23].

Trust Region methods differ in how they approach the *Solve* step of the algorithm. A very common algorithm, is called *Levenberg-Marquardt* [28, 30]. It transforms the constrained optimization problem 2.2 to an unconstrained one using Lagrange multipliers. The problem then takes the form [23]

$$\arg\min_{\Delta \mathrm{x}} \quad \frac{1}{2}||F(x) + J(x)\delta x||^2 + \lambda||D(x)\Delta x||^2 \tag{2.4}$$

where λ is the Lagrange multiplier, that is inverse to μ. Typically, $D(x)$ is the square root of the diagonal of $J(x)^T J(x)$. Solving the problem is tackled by a linear equations solver, with *Ceres* offering a variety of them suited for various problem sizes (dense, sparse) or specific problem structures [23].

Algorithm 2 Trust region algorithm [23]

```
Given $x_{init}, \mu$
 1: repeat
 2:     Solve
            $\underset{\Delta x}{\arg\min} \quad \frac{1}{2}||F(x) + J(x)\delta x||^2$
            such that $\quad ||D(x\Delta x||^2 \leq \mu$
            $L \leq x + \Delta x \leq U$
 3:     $\rho = \frac{||F(x+\Delta x)||^2 + ||F(x)||^2}{||F(x)+J(x)\Delta x||^2 - ||F(x)||^2}$
 4:     if $\rho > \epsilon$ then
 5:         $x = x + \Delta x$
 6:     end if
 7:     if $\rho > \eta_1$ then
 8:         $\mu = 2\mu$
 9:     else if $\rho < \eta_2$ then
10:         $\mu = \frac{\mu}{2}$
11:     end if
12: until a convergence criterion is met
13: return $x$
```

2.3 Matrix description of a serial chain robot

The movement of a robot in 3D space can be complex and it's sensible to use an unified, compact and algorithmizable notation to describe transformation between coordinate systems. Probably the most used system is based on transformation matrices. Let's assume two coordinate systems, a and b. We're interested in describing the movement of point M in coordinate system b, relative to a. This can formally be written as [31]:

$$\mathbf{r_{aM}} = \mathbf{T_{ab}} \mathbf{r_{bM}} \tag{2.5}$$

$$\left[\begin{array}{c} x_{aM} \\ y_{aM} \\ z_{aM} \\ \hline 1 \end{array}\right] = \left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & a_1 \\ a_{21} & a_{22} & a_{23} & a_2 \\ a_{31} & a_{32} & a_{33} & a_3 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \left[\begin{array}{c} x_{bM} \\ y_{bM} \\ z_{bM} \\ \hline 1 \end{array}\right], \tag{2.6}$$

where $\mathbf{r_{bM}}$ is the radius vector of point M in coordinate system b, $\mathbf{r_{aM}}$ is the radius vector of point M in coordinate system a and $\mathbf{T_{ab}}$ is the transformation matrix between a and b. The 3x3 submatrix is a rotation matrix, whose columns correspond to the unit vectors of b in a. The last column is the translation between origins of a and b. The last row represents identity and is included to get a square matrix [31].

A transformation matrix can describe an arbitrary translation and rotation, but commonly used are matrices describing rotation about or translation in only one axis. Due to the fact that any spatial motion can be composed of basic motions, these matrices can be multiplied to get an arbitrary motion. Herein lies a big strength of this description system: matrices for basic motions are readily available, intuitive to understand and easily algorithmizable. Their use also convenes with the engineering praxis, where joints with single DOF are the most common [31].

Let's take a serial chain robot with n links and assign each link a coordinate system. The transformation between two neighboring links is described by the matrix $\mathbf{T_{i-1,i}}$. We're interested in the motion of the point M, located on the last link. By chaining the transformation matrices we obtain [31]:

$$\mathbf{r_{1M}} = \mathbf{T_{12}}\mathbf{T_{23}} \dots \mathbf{T_{n-1,n}}\mathbf{r_{nM}} = \mathbf{T_{1n}}\mathbf{r_{nM}} \,. \tag{2.7}$$

By handily choosing positions and orientations of coordinate systems (in the joints or other prominent points), we can define the individual matrices using only basic motions. In the end we get one transformation matrix [31]:

$$\mathbf{T_{1n}} = \prod_{i=1}^{n-1} \mathbf{T_{i,i+1}} \,. \tag{2.8}$$

In a serial chain robot, this matrix depends on the joint coordinates $\mathbf{q}$. If $\mathbf{r_{nM}}$ is constant, then by setting q we can find out the position of the end effector of the robot. This is known as the forward kinematics problem [29].

If we instead take a transformation matrix $\mathbf{T_g}$ and set it equal to $\mathbf{T_{1n}}(\mathbf{q})$ we can compute the required joint coordinates $\mathbf{q}$ to reach the desired position. This is called the inverse kinematics problem [29].

2.4 Manipulator dexterity

Even without in-depth knowledge of robots, it's easy to see, that some configurations are inherently better than other. Akin to a human arm, a fully stretched out robot is not able to perform tasks very well. In other cases, if joints align in an unfortunate position a loss of DOF occurs. To describe which configurations are better than others we use dexterity, defined as the inverse of the condition number of Jacobi matrix [34]:

$$D = \frac{1}{\mathrm{cond}(\mathbf{J})} \,. \tag{2.9}$$

The higher the dexterity, the better [34].

A condition number of a matrix is a ratio of the largest and the smallest singular number of a matrix [34]:

$$\mathrm{cond}(\mathbf{J}) = \frac{\sigma_{max}}{\sigma_{min}} \,. \tag{2.10}$$

Singular numbers of a Matrix can be obtained using the SVD decomposition [34]:

$$\mathbf{J} = \mathbf{USV}^{\mathrm{T}} \,, \tag{2.11}$$

where matrix **S** is a diagonal matrix with singular numbers of matrix **J** [34].

Jacobi matrix is a matrix of partial derivatives of the constraint functions with respect to the coordinates and also serves as a operator between the input (joint) and output (end effector) velocities [34]:

$$\dot{\mathbf{v}} = \mathbf{J}(q)\,\dot{\mathbf{q}} \tag{2.12}$$

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\psi} \\ \dot{\vartheta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial q_1} & \frac{\partial f_1}{\partial q_2} & \cdots & \frac{\partial f_1}{\partial q_n} \\ \frac{\partial f_2}{\partial q_1} & \ddots & & \vdots \\ \vdots & & \ddots & \\ \frac{\partial f_m}{\partial q_1} & \cdots & \cdots & \frac{\partial f_m}{\partial q_n} \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \\ \dot{q}_5 \\ \dot{q}_6 \\ \dot{q}_7 \\ \dot{q}_8 \\ \dot{q}_9 \end{bmatrix} . \tag{2.13}$$

Chapter 3

System description

The final goal of the project is having functional robot cells capable of production deployment, which makes the complexity of the whole system fairly high. The hardware has to be reliable and simple to use, with all the necessary safety measures included. Serviceability is another priority: the extruder might need adjustment and the consumed filament resupplied. Ease of loading and unloading the cell has to be considered as well.

The accompanying software is no less complex: it needs to guarantee, that the planned paths are continuous, collision-free and feasible to execute. Safety and reliability are of highest importance here as well. The software needs to provide enough fallbacks and safety checks to prevent any possibility of damage by an incompetent user. For that, a verification of the planned path has to be possible, e.g. in the form of a visualization. The whole system can be divided into a hardware and software section, with each containing multiple components:

- Hardware:
  - Industrial robot
  - External axes
  - Extruder
  - Other equipment
- Software:
  - G-Code Parser
  - Robot model
  - Planner & Optimizer
  - Visualization & Data export

3.0.1 Data flow

The whole manufacturing process begins with a contract. Here, during negotiations, at least one sketch of the tank will be drawn. In the end, the sketch needs to depict all of the customer's requirements and contain all important components and features of the final product. This sketch will then serve as a base for a 3D model of the tank. The job of the designer is to model it in scale and include all features. The model also has to be detailed enough to depict manufacturing features, such as chamfers. This is because the 3D model will be used during planning as a collision model and any missing parts could cause the robot to crash. In the manual manufacturing process, the drawings didn't need to be so detailed, because an experienced welder can recognize problematic areas and adjust themselves. The robot has no perception in this sense though and will blindly follow any given path, regardless whether it will cause it to crash or result in a unsuccessful welding operation.

As previously mentioned, the definition of welding parameters will move from the workers in the workshop to the design office. This will be the responsibility of the manufacturing engineer, but in reality it might be the same person as the designer. They will define the weld positions, order of welding, temperatures, speeds and materials used. Only with this information complete can the input data for the planner be generated. This data will most likely take on the form of modified G-Code generated by a CAM software.

This generated data and the 3D model will serve as an input for the planning software itself. Here all the optimization takes place. It will plan a welding path and visualize it, which the engineer has to verify. If the path fulfills all the criteria and the engineer is satisfied, the data can be exported. They will contain joint coordinates for each weld point and all required technological commands. This exported data can then be input in the robot's control unit, which controls both the movement as well as the technological systems (heaters, extrusion etc.). The whole dataflow is depicted in Fig. 3.1.

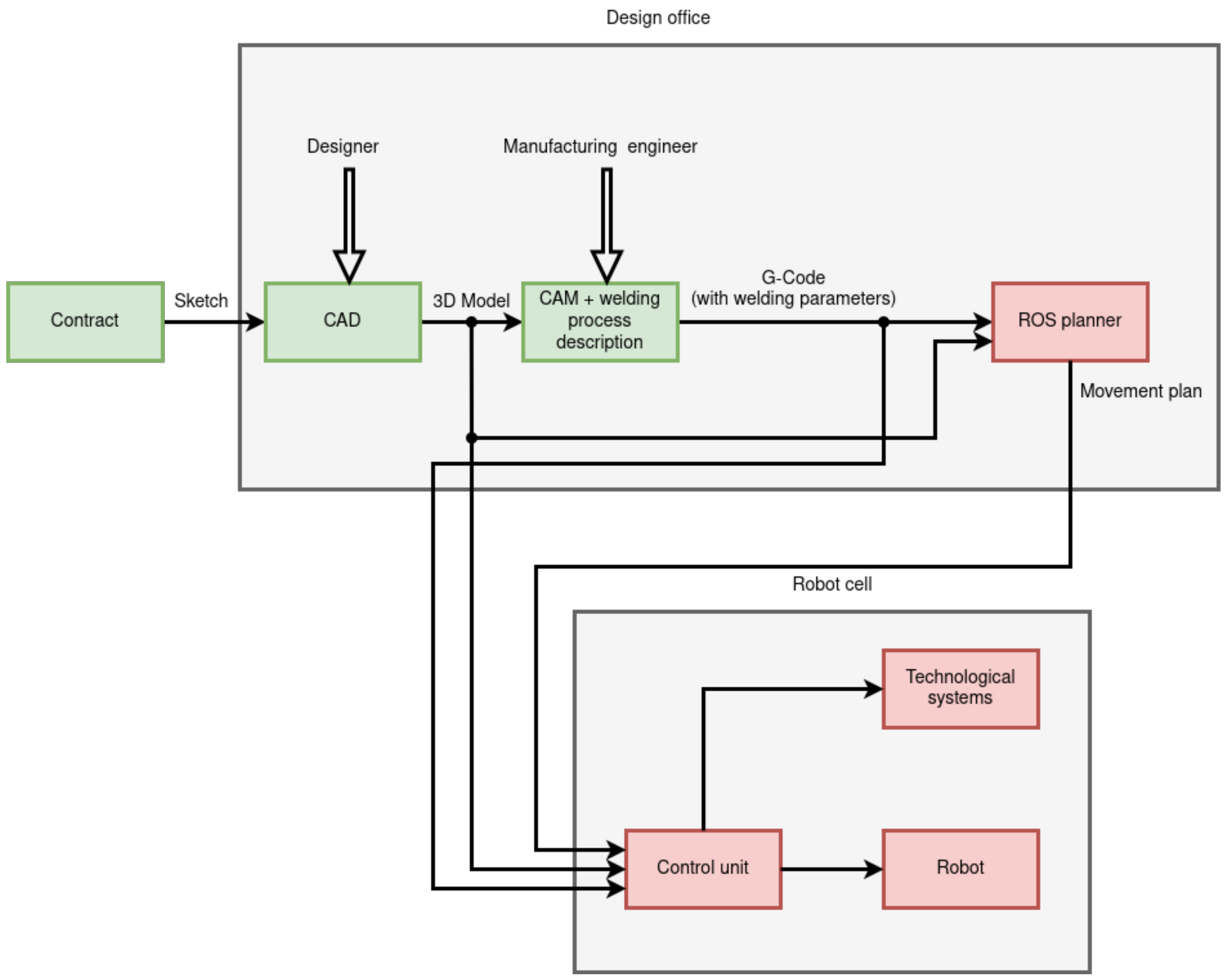

Figure 3.1: Data flow in the planning process - Bulk of the work done on a welding project is done in the design office. Here, after receiving a contract, the designer will model the tank in a CAD software. The 3D model is transferred to the CAM software and, with the help of the manufacturing engineer, the G-Code is generated. This G-Code along with the 3D model is sent to the planner. The planned movement is then executed by the robot cell.

Specification GP88		
Axes	Maximum motion range [°]	Maximum speed [°/s]
S	±180	170
L	+155/-90	140
U	+90/-80	160
R	±360	230
B	±125	230
T	±360	350
Max. payload [kg]		88
Repeatability [mm]		±0.03
Max. working range [mm]		2236

Table 3.1: GP88 Specification [22]

3.1 Hardware

The cell (Fig. 3.2, 3.3) consists of a standard 6-axis industrial robot, mounted upside-down on a beam. This beam is attached to a column and can move vertically, meanwhile the robot base can move horizontally along the beam's longitudinal axis. The third external DOF is supplied by a rotating platform, on which the to-be-welded tank rests. The external DOFs extend the working space considerably and allow for a continuous weld around the tank. The entire robot cell has been designed at *triotec s.r.o.*

3.1.1 Industrial robot

The robot chosen is YASKAWA GP88 (Fig. 3.4). It is a universal high speed robot with 6 rotational axes [22]. Table 3.1 summarizes all of it's important parameters and features.

3.1.2 External axes

The two linear axes are part of the cell. The vertical axis moves the whole beam, which itself is a truss frame. The vertical axis is moved by a ball screw. The horizontal axis, moving the robot base, is moved by a rack and pinion mechanism. Both axes are guided by linear rails. The third, rotational axis rests on the bottom of the frame and rotates a sizable platform on which the tank sits. To fixate the tank during welding, plenty of mounting points are provided on the platform. The table rotation is limitless, thus allowing infinite rotations in either direction. Table 3.2 shows all of the axes' parameters.

3.1.3 Extruder

The robot can't use a standard handheld extruder, but needs a variation suitable for remote control and automatic operation. A Swiss company,

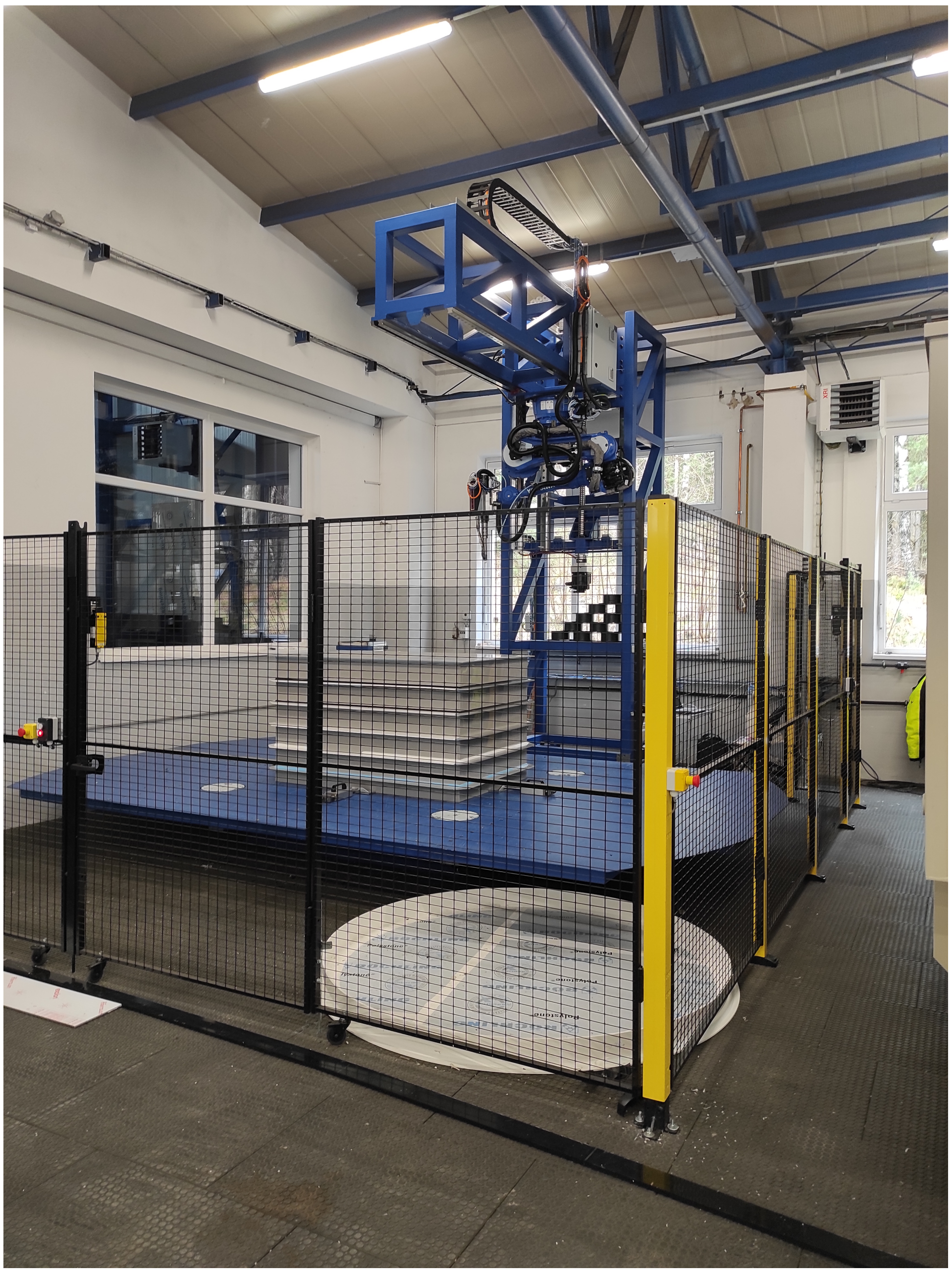

Figure 3.2: The robot cell - Depicted is the entire built robot cell. Wire mesh fence with loading doors separates the cell from the rest of the workshop. Most of the space is taken up by the rotating table, on which a cuboid tank is placed and clamped. The standing column and the moving beam are both truss frames. The robot itself is folded in a parking position in the furthest and highest position of the cell.

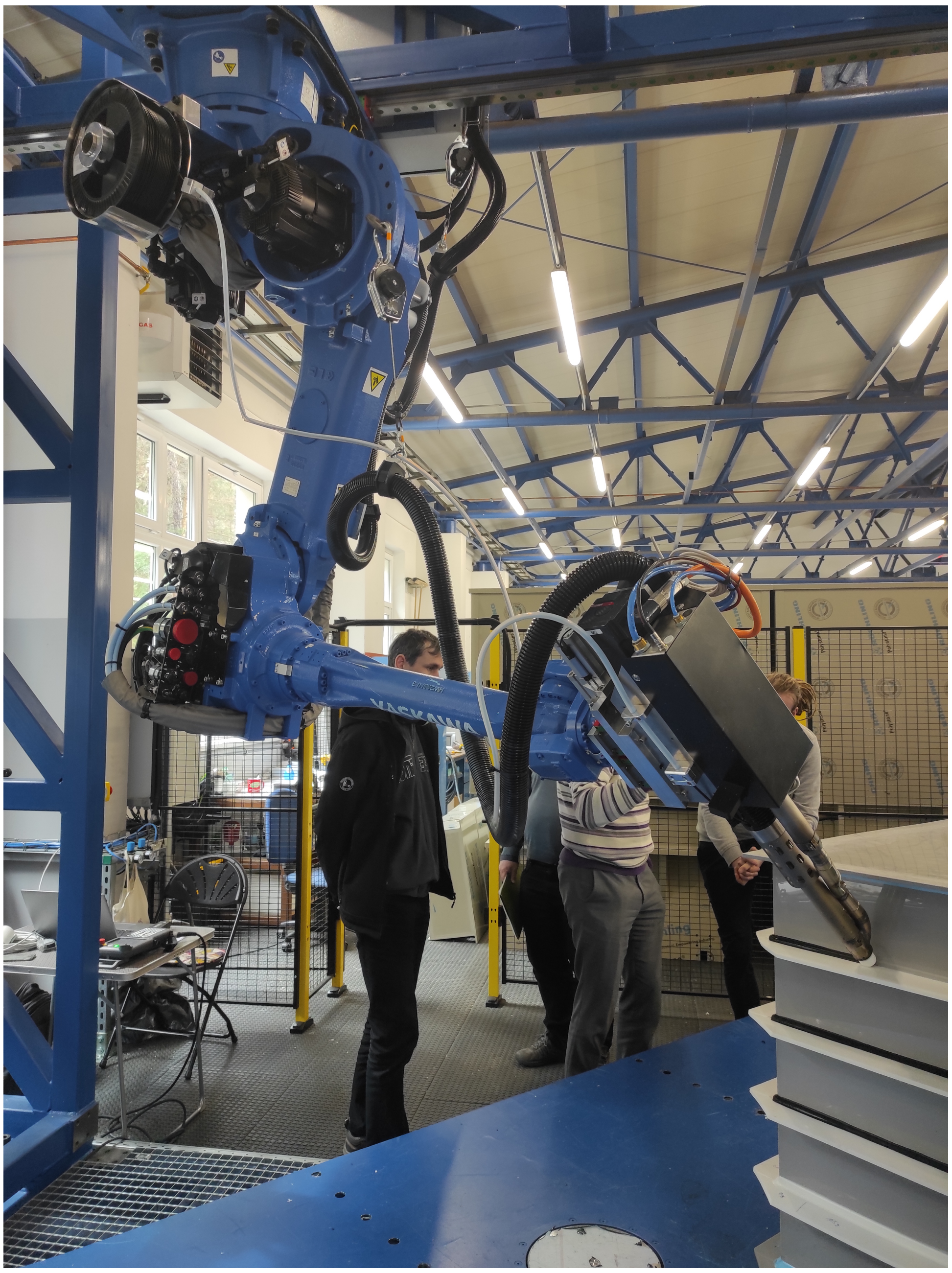

Figure 3.3: The robot cell (detail) - A closeup of the robot cell during welding. The extruder tip is in contact with the tank and the rotation of the extruder is clearly visible. It is also visible, that the tank has been rotated on the table. On the first robot link, a black spool is mounted, containing the welding filament. The filament is brought to the extruder through a flexible tube hanging from the spool.

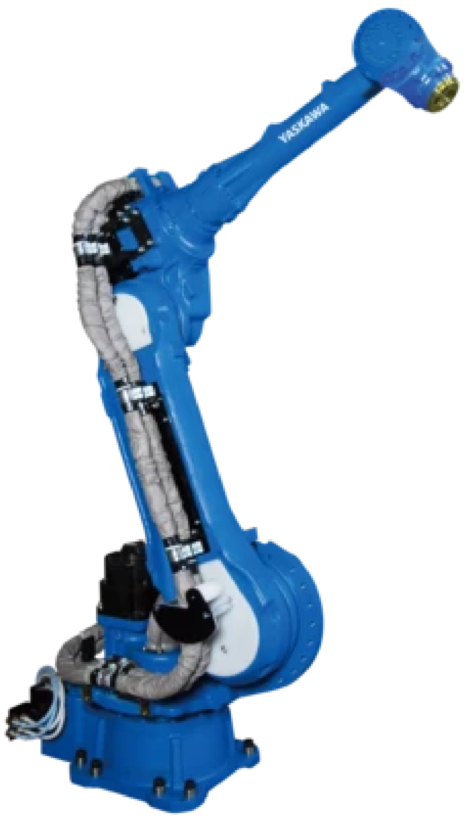

Figure 3.4: Yaskawa Motoman GP88 [22] - GP88 is a standard industrial robot with 6 rotational axes. The last three axes (4, 5 and 6) intersect in one point (wrist). IKT of this joint configuration is solvable analytically.

Leister, offers a model, Weldplast 610-i (Fig. 3.5), which is be used. Table 3.3 shows the most important parameters of the extruder.

The extruder itself can be mounted on the robot's flange, but on it's own, this, due to dimension deviations between the 3D model and a real tank, wouldn't provide the constant force required to keep the extruder tip pushing onto the weld seam. Thus, a mechanism (Fig. 3.6) allowing movement along the extruder's main axis is placed in between the robot and extruder mounting flanges. The constant force is realized using a pneumatic cylinder. This DOF is not considered in the planning problem.

The extruder has been subject to further modifications, to better suit the automated welding process. Normally, the extruders are equipped with a wedge-shaped welding shoe (Fig. 3.7). It helps guide the extruder in the weld and maintain constant angle. This has however shown to be a limiting factor during testing and the shoe has been replaced by a conical one. This helps the extruder to better navigate corners. With it, the extruder can be angled as it approaches a corner, such that it sits at 45° when exactly in the corner.

Another drawback of the original construction is the heater nozzle. The blower opening is next to the extruder tip on one side. This limits, how the

Specification external axes				
Axis	Distance between end-stops	Maximum motion range	Maximum speed	Time to accelerate from 0 to 100% [s]
Table	$\pm\infty$	$\pm\infty$	$3.088°/s$	3
Vertical	0 - 2475 mm	5 - 2470 mm	50 mm/s	0.5
Horizontal	0 - 1320 mm	1 - 1278 mm	412 mm/s	0.5
Max. table payload	2000 kg			

Table 3.2: External axes specification

Specification Weldplast 610-i	
Power [W]	1600
Filament diameter [mm]	⌀4 - 5
Material output at ⌀4 mm [kg/h]	0.1 - 4.0
Material output at ⌀5 mm [kg/h]	0.1 - 8.4
Weight [kg]	22
Length [mm]	876.0
Width [mm]	191.0
Height [mm]	21.0
Welding materials	HDPE, LDPE, LLDPE, PP

Table 3.3: Extruder specification [15]

extruder can move: During welding, the hot air nozzle has to always blow hot on the weld, just before the plastic will be extruded there. The exruder can't therefore be rotated arbitrarily, but the weld seam has to lie in the plane defined by the extruder tip axis and the heater nozzle axis. One rotation DOF of the extruder is constrained by this requirement, which complicates the planning and movement execution, especially in corners, tight areas and inside welds. To overcome this limitation, the nozzle has been modified and now blows hot air concentric all around the hot end (Fig. 3.8). The extruder is thus able to weld in any position rotated around it's longitudinal axis.

3.1.4 Other equipment

The robot cell also contains auxiliary equipment, necessary for safe and continuous operation. One of them is a preheating station (Fig. 3.9). Here the robot will spend time until the extruder reaches operating temperature. A small amount of filament is also extruded here, to normalize the welding conditions.

Important part of the equipment are safety fences and barriers all around the robot cell. Access to the working area is controlled, as no persons should be present during operation.

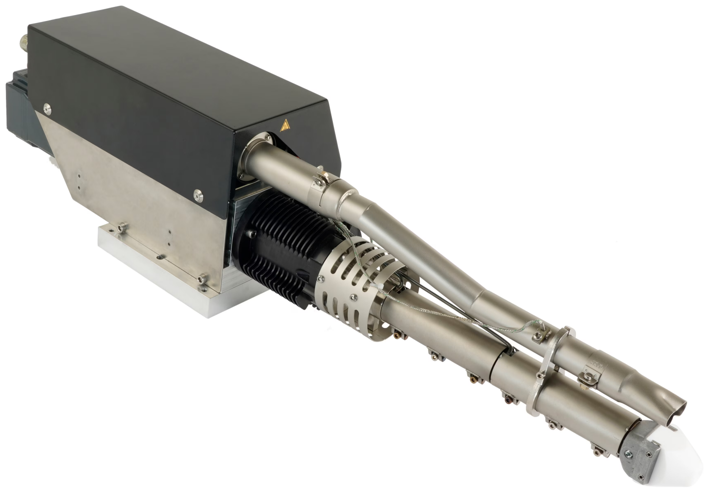

Figure 3.5: Leister Weldplast 610-i [15] - The extruder body (black cover) has two protrusions: the thicker, straight one, with white tip is the extruder nozzle and the slightly curved one is the heater nozzle. The nozzle blows hot air on the weld.

3.2 Software

The development of the planning software is a complex and time-consuming process, due to the software's complexity. The software needs to be able to process the input data, order and save them in a suitable data structure. Then it has to apply planning and optimization algorithms. The created path has to be verified by the designer and only after that it can be exported for use in a control unit. Ideally, the software will also be universal enough to be able to generate paths for a wide array of tank shapes. That requires adjustability of a wide array of parameters, starting from the welding process itself, the tank model and position and going all the way to modifying optimization criteria.

3.2.1 ROS

At the start of the project it was decided to develop the planning part of the software using ROS (Robot Operating System). ROS is a open-source software development environment (sometimes also called a middleware) used for robotic applications. It is used widely in teaching and research, but also in industry. With a strong community support, there's a wide availability of tools and libraries. Many of them contain modern, state-of-art algorithms and help reducing overhead when starting a new project [2].

Over time, ROS has come through various versions, the last being named

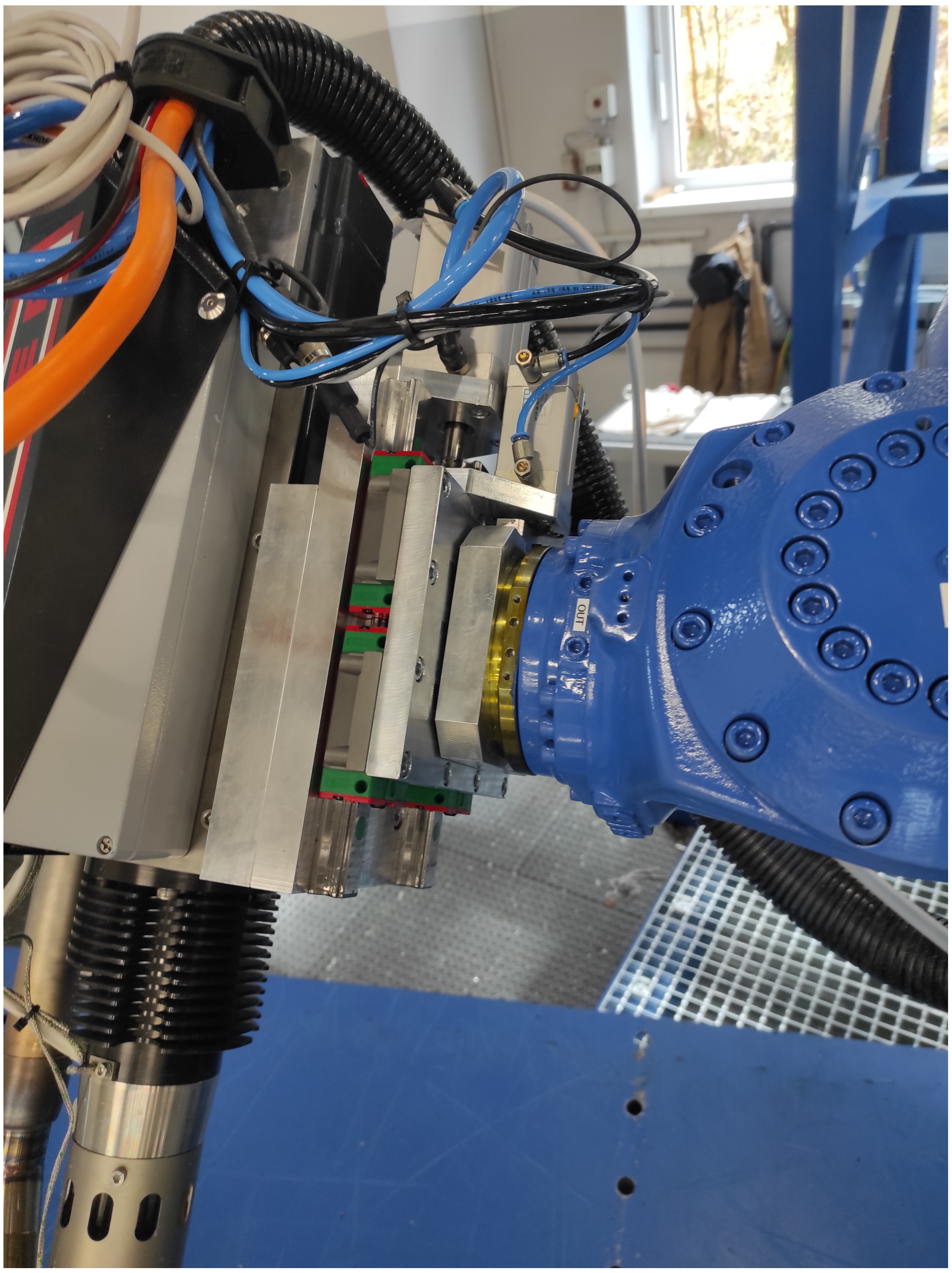

Figure 3.6: Mounting of the extruder on the robot flange - The robot mounting flange is the brass-colored piece at the end of the robot arm (blue). The mechanism pushing the extruder into the weld is clamped between the robot and the extruder mounting flanges: Two linear rails, as well as the carriages (with green and red caps), are visible. The pneumatic cylinder (white) is mounted facing the rear of the extruder.

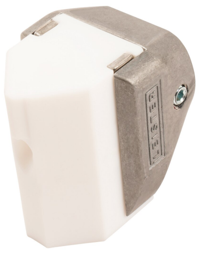

Figure 3.7: Original welding shoe [14] - The original wedge-shaped welding shoe is useful for guiding the extruder during hand welding, but has shown to be problematic during robotic welding. Due to it's size and rigidity, the robot is generally less flexible in manouvering and the new spherical welding shoe helps with that.

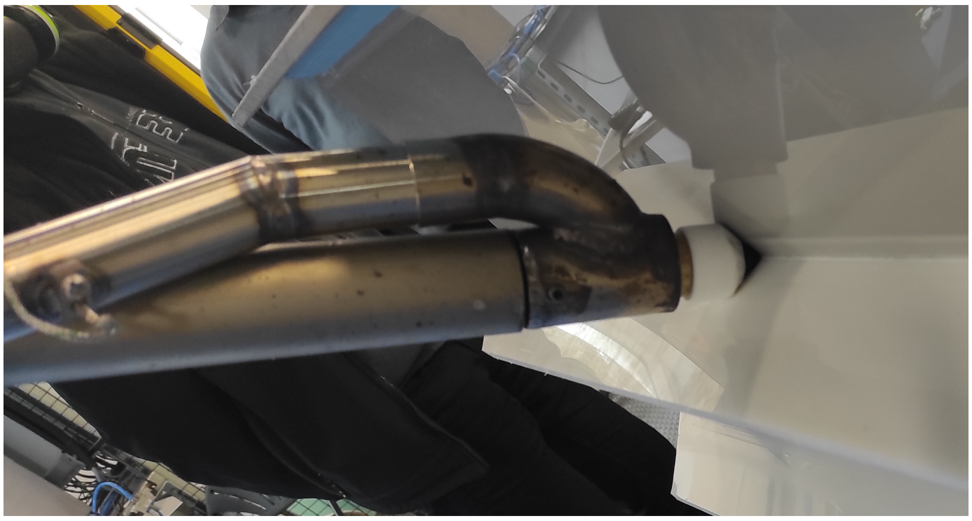

Figure 3.8: Concentric hot air nozzle - A detail of the extruder tip: the straight cylinder is the extruding nozzle itself, capped of by a white welding shoe. This is the modified, spherical welding shoe. The curved tube is supplying hot air from the blower. The hot air nozzle is mounted concetrically and blows air all around the extruder tip.

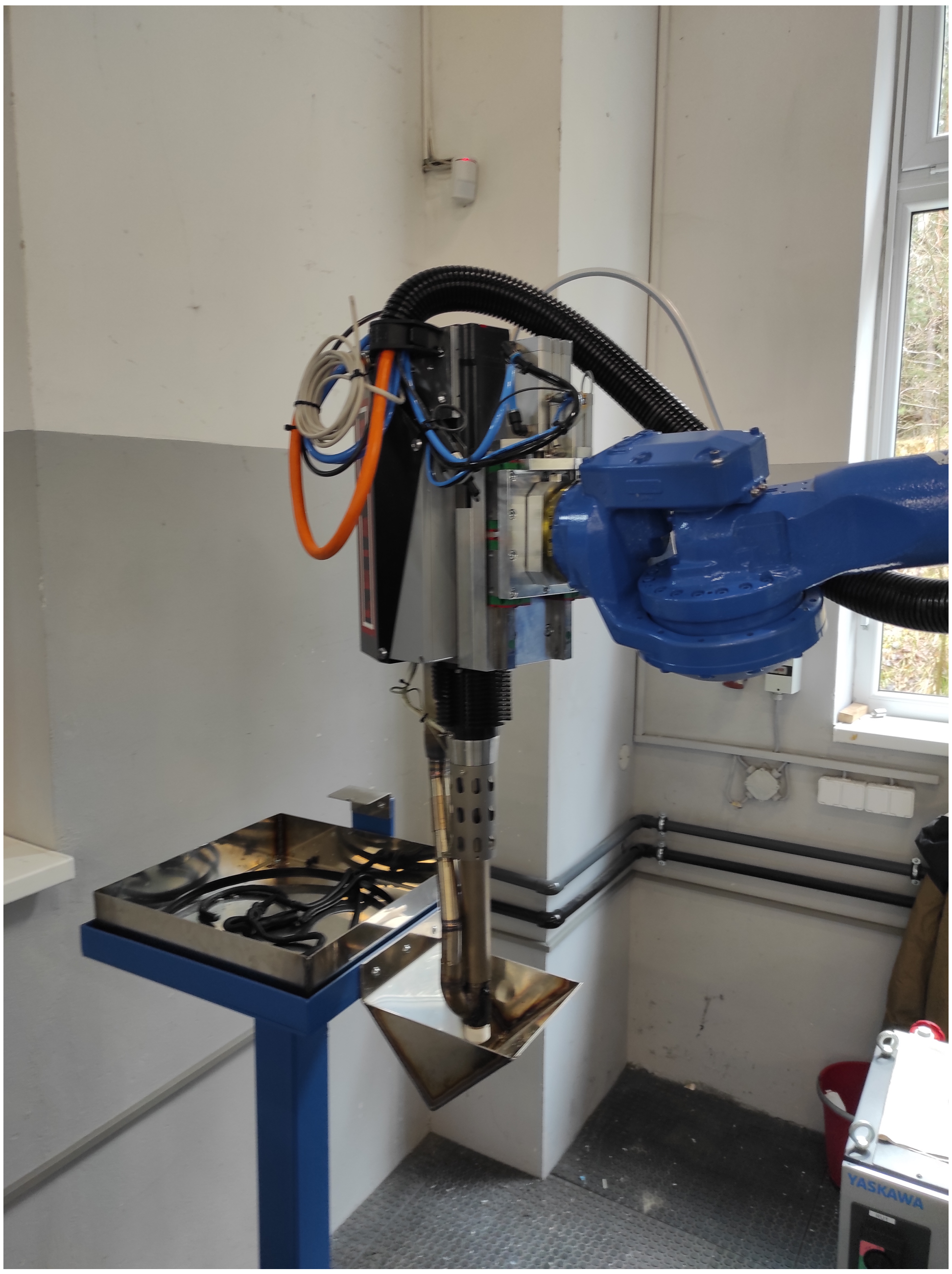

Figure 3.9: Preheating station - During preheating, the extruder is positioned above the preheating station, with the extruder tip facing downward into a catch bowl (triangular prism). After preheating the extruder will, as a test, extrude a small amount of filament in the rectangular box. The hanging extruder plastic will be cut off flush with the small cutting tab on the side of the rectangular box.

Noetic. However, the core has remained the same. To improve ROS further, the development team has decided to create a completely new version, called ROS 2 [6]. The system has been rebuilt from the ground up and has some excellent new features. Most importantly, it is much more suitable for real-time compared to the older ROS 1. While you could control a robot directly from ROS, it often wasn't the first choice and conventional control units were used, with ROS being relegated to offline planning tasks [10].

This project has started with ROS 1. Mr. Horný used version Melodic in his Master Thesis. We have updated the project to the latest ROS 1 version, Noetic. But one of the goals of this project is to use ROS not only for planning, but also for execution. This prompted the switch to ROS 2, which didn't happen during this Master Thesis, but it is planned in the near future.

A basic structure of a ROS project is formed from *nodes*. Each *node* corresponds to a executable that can run independently from other *nodes*. The management of *nodes* and communication between them is facilitated by *ROS Core*. It registers and monitors all running nodes and tracks *publishers* and *subscribers*. They are *nodes*, that have been set-up with communication interface. Communication itself in ROS occurs using *topics* and *services* (Fig. 3.10). A *topic* is a channel, where any amount of *publishers* can publish (send) data and any amount of *subscribers* can receive them. It is a many-to-many system that doesn't require any request calls. The *subscribers* simply receive any and all published data. On the other hand, *services* are a request-and-reply communication system and function one-to-one. The data is published on *topics* and *services* in the form of *messages*. Their structure ranges from primitive data types to complex data structures and can be created to suit the application [8].

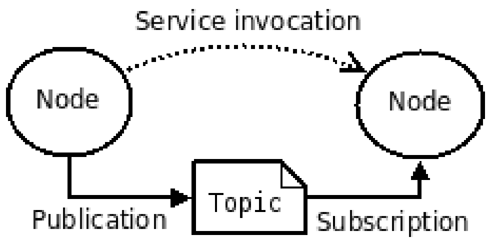

Figure 3.10: ROS Structure [8]

Another important component handled by *ROS Core* is the *Parameter Server*. Instead of using a separate configuration file for each node, the *Parameter Server* centralizes all of the configuration data. The data can be accessed and edited by the nodes during the run of the application. For editing, parameters are stored and edited in *YAML* files [8].

One of the most used libraries within ROS is the *tf* library. *Tf* stands for transformation, because this library handles all of the coordinate systems present within the environment. It tracks transformations between the coordinate systems and can return the actual transformation matrix on a request. [3] This is a very powerful feature, which simplifies the handling of coordinate systems in time. We don't have to write out complex matrix transformations, which is not only a time consuming process, but also a one prone to errors.

On topics of coordinate system, we should mention that internally, ROS uses quaternions for handling of rotations. Conversions to different rotation descriptions, such as Euler angles, are provided [9].

For simpler applications, it might be manageable to build and control the robot model manually, but for anything more complex, this approach is unsuitable. That's why another of the most used libraries in ROS is *MoveIt!*. This library provides a simple, high-level control of the robot with intuitive setup. With *MoveIt!*, the entire robot can be moved using single commands. *MoveIt!* also implements some planning algorithms and collision checking [24].

3.2.2 G-Code Parser

The input data format needs to provide all of the information required about the welding process: the positions of the welds, welding speeds, dwelling, preheat and cooldown times, temperatures, tools and materials used. It also needs to distinguish between moves in free space, happening while the robot moves between welds and work moves, where actual welding takes place, as they have different restrictions placed on them. One possible format for this could be G-Code, widely used in the programming of CNC (Computer Numerically Controlled) machines and 3D printers. This code could be generated by a CAM (Computer Aided Manufacturing) software, that can simulate milling operations on a 3D model [11]. G-Code is a human-readable, extendable programming language. Originally developed for CNC machines, it is now widely used in 3D printing as well. Manufacturers of CNC machines have developed their own versions of G-Code, but the core remains the same. A G-Code file consists of commands, usually defined with a letter+number combination, such as `G01 X10 Y20 Z30 F500`, `G90`, `M06`, etc. The control unit reads them into a buffer and executes sequentially. The most important commands for this project are [11, 12]:

```
G00 X111.1 Y222.2 Z333.3 I444.4 J555.5 K666.6
```

and [12]

```
G01 X111.1 Y222.2 Z333.3 I444.4 J555.5 K666.6 F777.7
```

They both describe a move to the position given by the coordinates `X, Y, Z` and rotation around the axes given by the coordinates `I, J, K`. The main difference is that `G00` is a non-work move and `G01` is a work move. That's why there's an additional parameter, `F`, describing the feed rate, i.e. the velocity. It has to be mentioned, that the form of these commands can differ slightly, depending on the G-Code specification. Also, it is rare to see all three rotations being given, as CNC machines usually have 5 axes at the most.

STP Plast has some experience using CAM software, but only with 3-axis milling machines. This is not enough for welding, as all six DOFs are needed to move the extruder along the weld. The CAM software, *AlphaCAM*, used in the company, can be extended with a plugin for 5-axis milling machines. With this extension, only the rotation of the extruder about it's longitudinal axis has to be calculated additionally. More so, this might not even be required in the future, as the spherical welding shoe and concentric heat nozzle allow welding regardless of this orientation. A few simple G-Code examples have been generated and used for the duration of this master thesis. They describe the required movement of the extruder in a simplified manner, but this was considered to be sufficient.

As the work on the project progressed, it became evident, that a much more detailed G-Code will be needed: due to preheating and the flow of the melted plastic, the robot has to pause in specific points and wait for a few seconds. Also, the handling of the movement in corners was found to be insufficient. Due to this issues, it was decided, that a new version of G-Code will be needed. Ing. Malý, a programmer and engineer from the company *Alad* has developed a G-Code generator with all of the features required. The only significant disadvantage of this generator is that it's command-line only. Because of the very limited experience with G-Code and robotics, this could pose a complication for the engineers at *STP Plast*, who will be preparing the whole welding process. Talks have begun with the supplier of *AlphaCAM*, asking if they would be able to extend their software to fulfill all of the requirements of this project. The decision has not been made yet, as to which path will be taken.

To process the G-Code into a format usable by the planner a parser software has been developed. It reads the G-Code line by line and takes out the important information: position and rotation values as well as the type of move. The data is then saved in a data structure, which uses standard ROS data types for position and rotation. These can be accessed and read by the planner. The parser has been based on the first version of the G-Code created by *AlphaCAM*, but it will be extended to be able to parse the more advanced G-Code developed by *Alad*

3.2.3 Robot kinematic model

To be able to simulate and plan the welding paths a mathematical model of the robot is needed. This model defines coordinate systems and the relations between them. Each robot link has a coordinate system assigned and the relationship is defined through joints and constraints. In this project we are working with a kinematic only model. That means, the links are defined only through their geometry, as are the joints. Mass and inertia are not considered here.

The most primitive way to build a robot model would be directly using transformation matrices. You can then update the individual transformations using the tf library [3]. This, however gets complicated very quickly. That's why robots in ROS are defined using *URDF* (Unified Robot Description

Format). It is a XML format with it's elements representing links and joints. By assigning attributes we can model an arbitrary robot. A link can be assigned a body, either made from primitives (cube, cylinder, sphere, etc.) or from a STL file [20]. This model is then loaded by a `robot_state_publisher` node. This node subscribes to a topic where joint states (coordinates) are being published. It then takes those values and updates all of the transformations. [7]

An ever higher level of abstraction is provided by *MoveIt!*. With it you create a *Robot Model* (the description of all links and joints) and a *Planning Scene* (the environment containing the robot). With simple, one-line commands, you can move the robot (forward kinematics), set a position and orientation (often called together a *pose*) of the end effector (inverse kinematics) or check for collision [17].

To aid *MoveIt!*, another file needs to be defined. It uses the *SRDF* (Semantic Robot Description Format) and specifies various additional information about the robot. The most important are *Groups* (also called *JointGroups*). They are a collection of joints and links on which MoveIt! acts. It will plan for or move an entire group at once: the joints outside the group will be left stationary. Another important use of the *SRDF* file is disabling collision checking between specific links. This helps to reduce computing time, as usually there are multiple links, that can't be in collision with each other and checking for it would be wasteful [21].

3.3 Planner & Optimizer

The planning software needs to distinguish between the two different movement types and apply an appropriate planning algorithm on them. A redundant mechanism has infinite solutions for the inverse kinematic problem, forcing the inclusion of an optimization procedure to find an unique solution. We define a cost function, which we try to minimize. We place various requirements on the movement and configuration of the robot, especially during welding. Each requirement is represented in the form of criterion. Then the overall cost function is a weighted sum of all criteria. The criteria to be considered vary greatly, with the most important being a collision free and a „good" resulting configuration of the robot. „Good" meant in a sense of having high dexterity and avoiding singular or near-singular positions, which lead to high joint velocities or discontinuities in the movement as the robot switches between configurations. They also limit the force applicable on the end effector.

3.3.1 Weld approach move

The weld approach move is occurring when the robot moves in free space, either from it's parking position or between two welds (e.g. from outside to inside of the tank). The extruder is inactive and no welding occurs during this move. This means less restrictions are placed on the path and robot configuration. Most importantly, the robot has to avoid collision and has to end up in a configuration, from which welding itself can begin and continue. That means no additional rotations or changes of configuration should be necessary, only a simple linear move to the point of contact of the extruder tip with the tank. Welding itself is a very slow process, with the maximal welding velocity being around 11 mm/s. In this regard, the time spent executing the approach move is negligible and we're not focusing on finding the shortest or fastest path.

For the weld approach planning we're using *OMPL* with a *RRT*-type algorithm.

3.3.2 Weld move

The part of planning that warrants much more attention is welding itself. Here, the requirements placed on the robot are much higher. The extruder tip has to move exactly along a prescribed path, otherwise the weld will fail. Any discontinuities or sharp changes in the movement will negatively affect the weld as well. The robot also shouldn't switch between configurations and we need to pay attention to the joint limits. If one of the joints hits a limit during a continuous weld, the solver will have a tendency to switch to a different configuration to continue. This we can't allow and need to implement checks to avoid this situation. Other than the already mentioned dexterity and collision avoidance, we also want to limit abrupt acceleration of any of the joints.

Since the robot has to follow an exact path, we need a different approach for the weld move than for the approach move. We need a planner that will solve an inverse kinematic problem for each point of the path, while minimizing the cost function. The Solver we've decided to use is *Ceres*.

The way we decided to approach the optimization problem is as follows: The parameters to optimize are the joint coordinates of the three external axes, while the coordinates of the remaining six axes of GP88 will be solved using analytical inverse kinematics (IKT). While we could optimize any three of the 9 DOFs, it's sensible to pick the three external axes, because analytical kinematics are available for the industrial robot. Another advantage in using analytical IKT is that we can compute all of the possible configurations and pick the most desired one, instead of relying on numerics and risking a sudden switch between configurations over which we have little to no control. Figure 3.11 shows an *UML Activity Diagram* for the planning phase, with the weld move algorithm written out in detail.

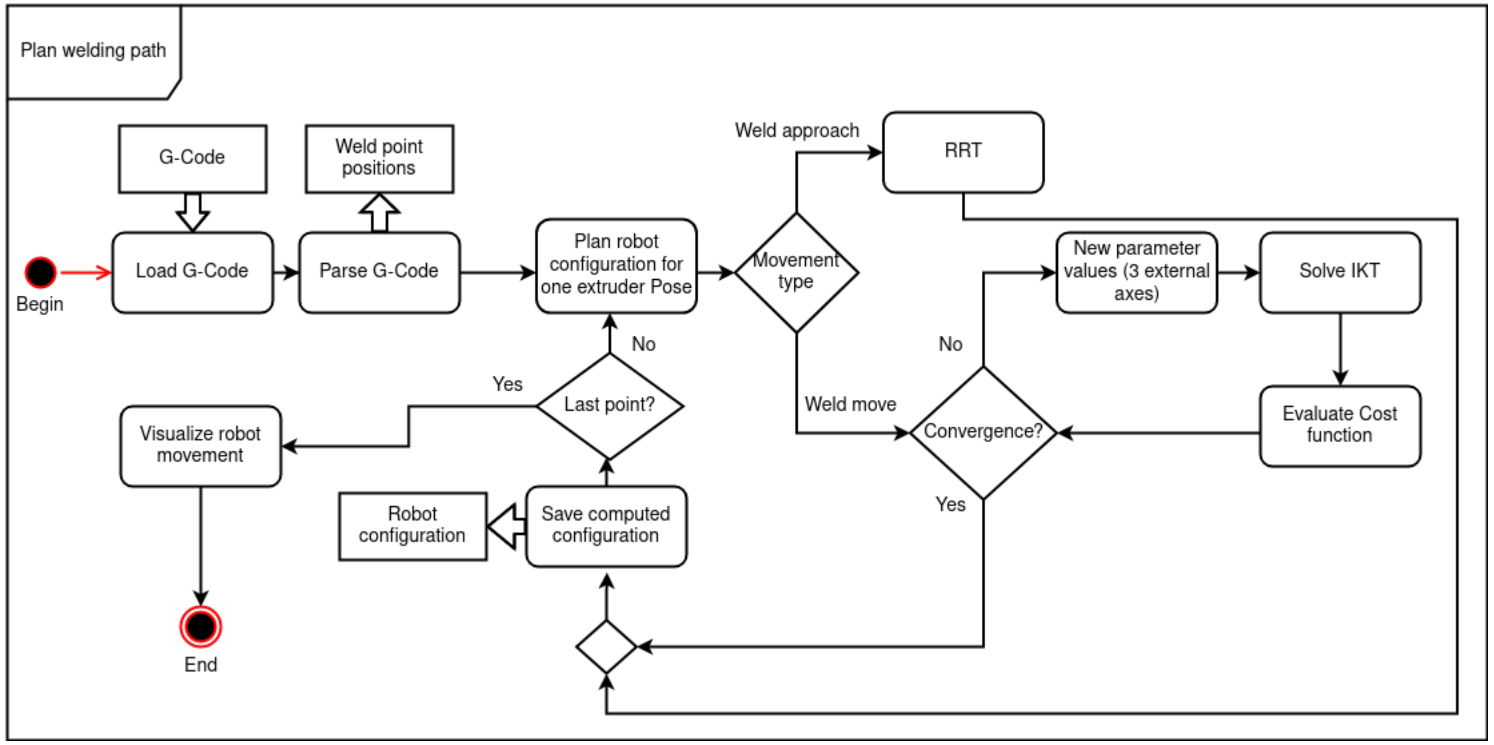

Figure 3.11: UML Activity Diagram for the optimization problem - Simple, black arrows show the control flow, while the thick arrows show the data flow. Rectangles with round corners are processes, rectangles with sharp corners are data storages. Control flow is split or joined in diamonds.

IKT solver

The inverse kinematics library used has been developed by Ing. Libor Wagner at *CIIRC CTU in Prague*. It's a fast standalone C++ library, which can be configured for a variety of 6-axis industrial robots. The analytical kinematics equations have been derived by hand and then implemented. The library is thus able to compute all of the possible solutions for a given inverse kinematic problem.

3.4 Visualization & Data export

The result of the planning should be a continuous collision-free path in robot joint coordinates extended with markers describing technological operations. This will be the input for the real-time robot control system. However, to get to a fully usable product, a wide array of additional functionalities has to be considered, such as input and output data formats, visualization and tests.

3.4.1 RViz

ROS standardly ships with a visualization software called *RViz*. It features a 3D environment, where various objects can be inserted. These objects then listen on specific *topics* and change their state according to the *messages* received. I this way can a movement of a robot be visualized. Figures 4.4 and 4.5 have been generated using RViz.

3.4.2 Graphs

A good data visualization is key to a successful optimization. Displaying the evolution of the cost of the criteria provides a helpful insight on how the solver is working. These graphs are very useful for detecting problems with the solution (i.e. with the planned path), that would be otherwise hard to recognize from the visualization only. Being able to publish any data to a topic, which can be later analysed and plotted, is thus a huge boon of ROS.

Chapter 4

Problem solution

4.1 Computing sixth rotation

An important part of the parser is the computation of the sixth rotation, that is usually missing from G-Code. The unknown rotation is that around the extruder's longitudinal axis. For the solution we have to consider how the extruder is positioned during welding: The preheating nozzle has to always lead before the extruder, in the direction of movement. In other words, the line representing the weld has to lie in the plane defined by the axes of the extruder and the preheating nozzle (Fig. 4.1).

To begin, we'll need the current rotation matrix, with it's columns representing the unit vectors of the coordinate system.

$$\mathbf{S_1^n} = \begin{bmatrix} \mathbf{i_1^n} & \mathbf{j_1^n} & \mathbf{k_1^n} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} i_{11} \\ i_{12} \\ i_{13} \end{bmatrix}^n & \begin{bmatrix} j_{11} \\ j_{12} \\ j_{13} \end{bmatrix}^n & \begin{bmatrix} k_{11} \\ k_{12} \\ k_{13} \end{bmatrix}^n \end{bmatrix} \tag{4.1}$$

Normal vector of the plane can be computed as

$$\mathbf{j_2^n} = \mathbf{k_1^n} \times \mathbf{s} \tag{4.2}$$

where $\mathbf{s}$ is the vector between two neighboring points (n and $n-1$). Then the unit vector lying in this plane, which determines the orientation of the nozzle (the axis x leading out from the extruder tip towards the preheating nozzle) is

$$\mathbf{i_2^n} = -\mathbf{k_1^n} \times \mathbf{j_2^n} \tag{4.3}$$

Since we want to keep the z axis, $\mathbf{k_2} = \mathbf{k_1}$. With that, we get a new rotation matrix:

$$\mathbf{S_2^n} = \begin{bmatrix} \mathbf{i_2^n} & \mathbf{j_2^n} & \mathbf{k_2^n} \end{bmatrix} \tag{4.4}$$

which can be converted to a quaternion and replace the original rotation of the *pose* (Fig. 4.2). Because the path vector $\mathbf{s}$ can't be determined for the last point, the rotation from the second-to-last point is repeated.

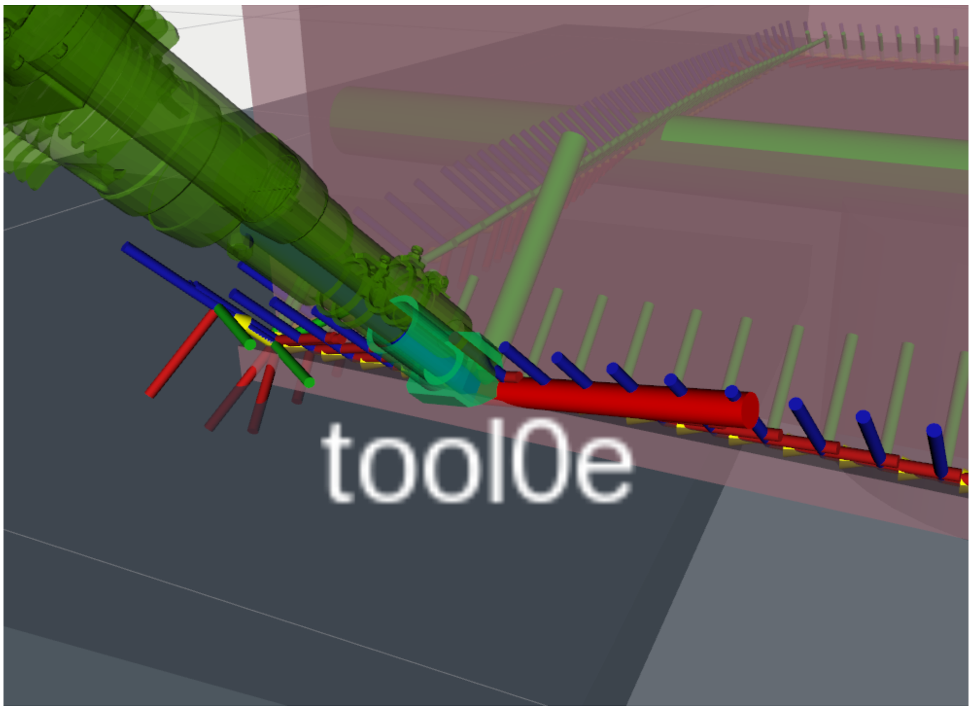

Figure 4.1: Extruder matching the Pose of a weld point - Axes x,y and z are colored red, green and blue respectively. The small coordinate systems represent the individual weld points, while the large coordinate system is the extruder tip (`tool0e`, positive z points into the extruder nozzle). It is visible, that the extruder tip coordinate system matches the weld point coordinate system. The extruder is also correctly aligned, with the preheating nozzle leading the extruder tip (welding direction is from left to right).

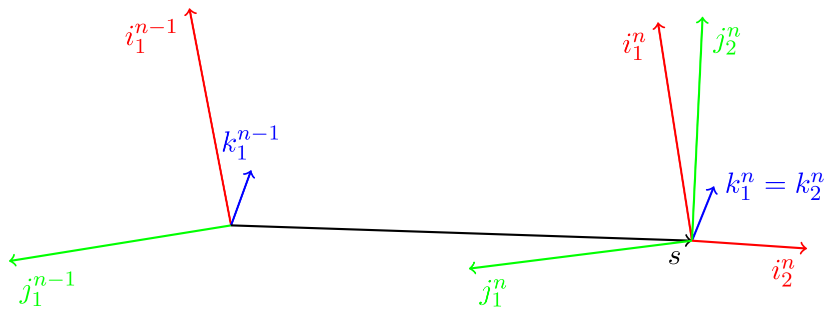

Figure 4.2: Computing sixth rotation - i_1^{n-1}, j_1^{n-1} and k_1^{n-1} are unit vectors of a previous weld point. i_1^n, j_1^n and k_1^n are unit vectors of the current weld point, uncorrected for the 6th rotation. s is the translation vector between these two points. i_2^n, j_2^n and k_2^n are unit vectors of the current weld point, corrected to account for the 6th rotation.

4.2 Robot kinematic model

The robot kinematic model consists of two distinct components: the *URDF* description, defining the relationships between robot links and joints and the collision model, crucial for a correct evaluation of a collision.

4.2.1 URDF

The *URDF* model for the robot contains all of the major links and all moving joints of the cell. The code for the external axes and the extruder have been written by us, while the GP88 robot body has been taken from the *ROS-Industrial* project [13]. ROS is able to export the *URDF* file as a graph, which clearly shows the relationships between all of the links. Due to it's size, the graph is included as an appendix. Figure 4.3 depicts the most important coordinate systems overlaid over the robot model.

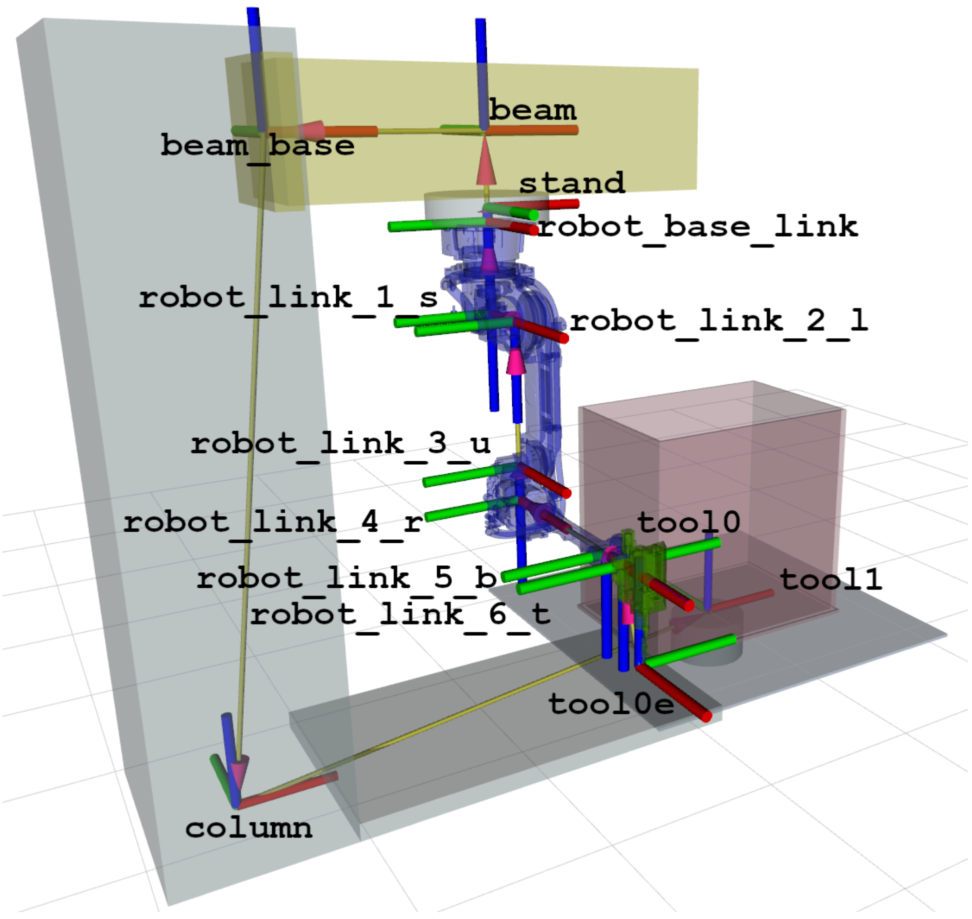

Figure 4.3: Coordinate systems of the robot - Axes x,y and z are colored red, green and blure respectively. The world coordinate system `tool1` is located in the middle of the table. From there a serial chain follows (depicted by the arrows) through the external axes, the robot itself all the way to the extruder tip (`tool0e`). Each robot link has it's own coordinate system.

Matrix description

Using transformation matrices (Eq. 2.8), the kinematic model can be described by the matrix chain:

$$\mathbf{T_{t1,t0e}} = \mathbf{T_{t1,c}}\mathbf{T_{c,bb}}\mathbf{T_{bb,b}}\mathbf{T_{b,s}}\mathbf{T_{s,rbl}}\mathbf{T_{rbl,1s}}\mathbf{T_{1s,2l}} \mathbf{T_{2l,3u}}\mathbf{T_{3u,4r}}\mathbf{T_{4r,5b}}\mathbf{T_{5b,6t}}\mathbf{T_{6t,t0}}\mathbf{T_{t0,t0e}}\,, \qquad (4.5)$$

where:

$$\begin{aligned}
\mathbf{T_{t1,c}} &= \mathbf{T_{\phi z}}(q_1)\mathbf{T_z}(-l_1)\mathbf{T_x}(-l_2) & \mathbf{T_{2l,3u}} &= \mathbf{T_z}(l_9)\mathbf{T_{\phi y}}(q_6)\\
\mathbf{T_{c,bb}} &= \mathbf{T_z}(q_2)\mathbf{T_x}(l_3) & \mathbf{T_{3u,4r}} &= \mathbf{T_z}(l_{10})\mathbf{T_{\phi x}}(q_7)\\
\mathbf{T_{bb,b}} &= \mathbf{T_x}(l_4) & \mathbf{T_{4r,5b}} &= \mathbf{T_x}(l_{11})\mathbf{T_{\phi y}}(q_8)\\
\mathbf{T_{b,s}} &= \mathbf{T_{\phi x}}(\pi)\mathbf{T_x}(q_3)\mathbf{T_z}(l_5) & \mathbf{T_{5b,6t}} &= \mathbf{T_x}(l_{12})\mathbf{T_{\phi x}}(q_9)\\
\mathbf{T_{s,rbl}} &= \mathbf{T_z}(l_6)\mathbf{T_{\phi z}}(\tfrac{\pi}{2}) & \mathbf{T_{6t,t0}} &= \mathbf{T_{\phi x}}(\pi)\mathbf{T_{\phi y}}(\tfrac{\pi}{2})\\
\mathbf{T_{rbl,1s}} &= \mathbf{T_z}(l_7)\mathbf{T_{\phi z}}(q_4) & \mathbf{T_{t0,t0e}} &= \mathbf{T_z}(l_{13})\mathbf{T_x}(l_{14})\mathbf{T_{\phi y}}(\tfrac{-\pi}{2})\\
\mathbf{T_{1s,2l}} &= \mathbf{T_x}(l_8)\mathbf{T_{\phi y}}(q_5)\,. & &
\end{aligned}$$

The parameters l_i are the distances between coordinate systems and can be find out from the robot geometry and construction.

It has to be mentioned, that the model built might seem non-intuitive. An obvious approach would be to fix the base of the cell and make it correspond with the world coordinate system. Then you'd have two branches: one leading to the rotating table and the other to the extruder through all of the remaining links. This configuration is unfortunately not possible, because *MoveIt!* supports only simple serial chains, or branches that move independently from each other. Our robot unfortunately doesn't fulfill this criterion, but by fixing the table and letting the whole cell rotate we can work around it. For practical application we'll simply reverse the rotation of the table calculated during planning.

4.2.2 Collision model

As already mentioned, *MoveIt!* is capable of checking collision. To facilitate this, the links in the *URDF* file must be assigned a collision body. It can be the same shape as the visual body, but usually it's slightly larger and simplified. This is because collision checking is a time-consuming process which greatly increases with the amount of triangles in the mesh. Also, by making the body larger, we are creating a small safety margin around the robot. This is important especially when there are cables and hoses attached to the robot body. Figure 4.4 depicts the visual model used, while 4.5 shows the collision model. We can see that some of the links remain the same, but the robot body has been simplified and wrapped in a coarse mesh with reduced number of triangles.

	tank	table	table_leg	frame_base	column	beam_base	beam	stand	robot_base_link	robot_link_1_s	robot_link_2_l	robot_link_3_u	robot_link_4_r	robot_link_5_b	robot_link_6_t	extruder_body	extruder_tip	botka
tank	⬛																	
table	🟧	⬛																
table_leg	🟥	🟧	⬛															
frame_base	🟥	🟥	🟧	⬛														
column	🟥	🟥	🟥	🟧	⬛													
beam_base	🟥	🟥	🟥	🟥	🟧	⬛												
beam		🟥	🟥	🟥	🟥	🟧	⬛											
stand		🟥	🟥	🟥	🟥	🟥	🟧	⬛										
robot_base_link		🟥	🟥	🟥	🟥	🟥	🟥	🟧	⬛									
robot_link_1_s		🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟧	⬛								
robot_link_2_l		🟥	🟥	🟥				🟥	🟥	🟧	⬛							
robot_link_3_u		🟥	🟥	🟥				🟥	🟥	🟥	🟧	⬛						
robot_link_4_r			🟥							🟥	🟥	🟧	⬛					
robot_link_5_b			🟥							🟥	🟥	🟥	🟧	⬛				
robot_link_6_t			🟥								🟥	🟥	🟥	🟧	⬛			
extruder_body			🟥									🟥		🟥	🟧	⬛		
extruder_tip			🟥									🟥		🟥	🟥	🟧	⬛	
botka			🟥									🟥		🟥	🟥	🟥	🟧	⬛

Table 4.1: Allowed Collision Matrix - Orange entries mark that a collision is not possible, because the links are neighbouring each other. Red entries mark link pairs that can't possibly collide, due to geometrical constraints of the robot. Black marks identity and white entries are all of the link pairs, where collision is possible and, therefore need to be included in collision checking.

Overall, the model is relatively crude and not very faithful to the real construction. This is because at the beginning of the project, no detailed model was available. Nowadays, the model exists, but replacing the current simple meshes with a complex realistic model is a very time-consuming process, which has actually very little impact on the workings of the planning software. The model has been taken from Mr. Horný's thesis [25], but, even though it looks mostly the same, the *URDF* and *SRDF* files have been completely reworked for more clarity and ease of modification.

Table 4.1 shows the *Allowed Collision Matrix*, i.e. the pairs of links, between which collision won't be checked. This matrix has been implemented in the *SRDF* file. Orange cells correspond to adjacent links and red to pairs of links, whose collision is not possible, due to geometric constraints. The diagonal is blacked out, because a collision of a link with itself is not possible (all of the links are rigid). White cells signify that no assumption can be made about that pair of links and they have to be checked for collision.

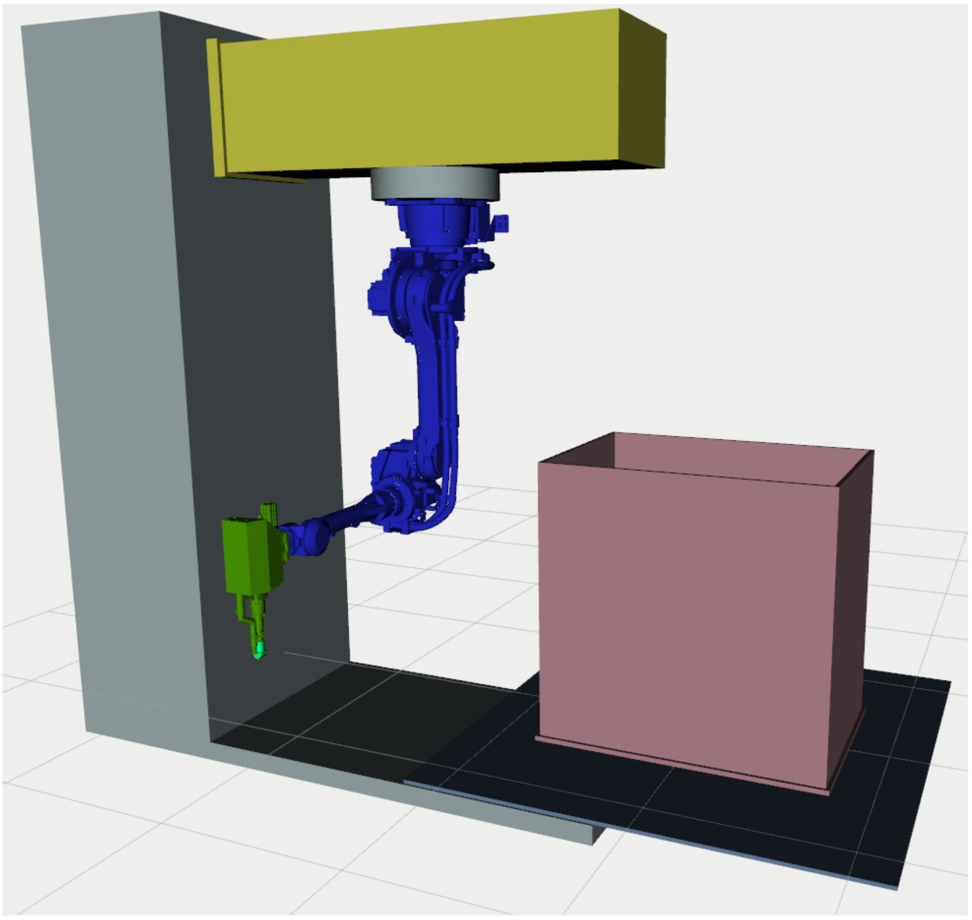

Figure 4.4: Robot visualization model - The robot visualization model depicts the real robot cell, albeit in a slightly simplified manner. The flat grey block is the base, the tall one is the column. The tank (pink) rests on the table. The yellow block is the moving column. The robot body is colored dark blue and is detailed, showing cables and hoses installed on the robot. The extruder (dark green) has a welding shoe (light green) mounted. Both are depicted, again, in detail.

4.3 Optimization criteria and constraints

To construct a cost function we need to define all of the individual criteria, that form the cost function. Main goal when choosing the criteria was to get a collision-free, smooth movement of the robot, that wouldn't switch between configurations, get in positions with low dexterity or close to singular positions and with no sharp changes in the joint velocities, as a result. If possible, we attempted to construct all of the criteria using smooth, continuous functions, with the hope that this would make the numerical derivations simpler and more reliable to calculate. All of them are used during weld planning, but only a subset during weld approach.

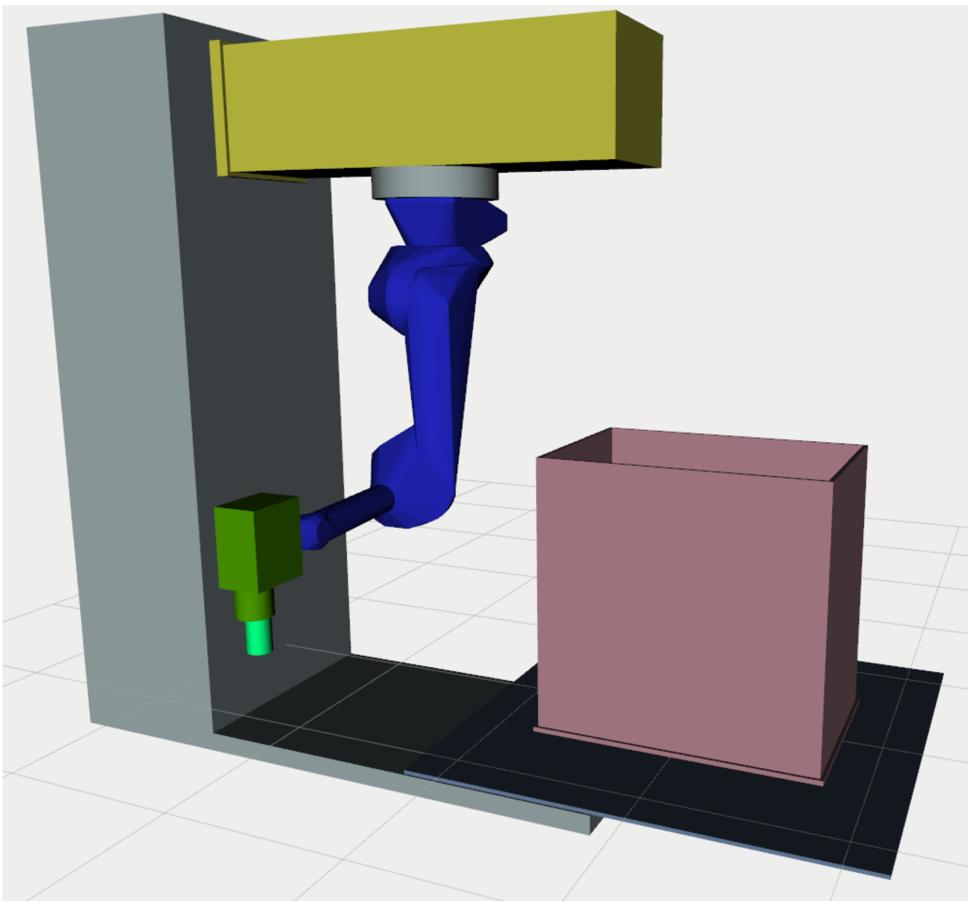

Figure 4.5: Robot collision model - The robot collision model is not nearly as detailed as the visualization model. While the tank, frame, column and beam look the same, the robot body is made up of a coarse mesh, wrapping the detailed visualization model. The extruder as well as the welding shoe are also wrapped in primitives, that are slightly larger, giving us a collision safety margin.

4.3.1 Collision criterion

Even if the robot is not in collision during simulation, it can't be guaranteed that this same will apply during real movements. If the distance to collision is let too small, imperfections in positioning or hanging cables and other appliance could cause a collision. That's why we want to create a safety margin around the robot. Using *MoveIt!*, it's simple to calculate the closest distance to collision, which we can input into a penalization function. This function can have various shapes, but needs to fulfill two requirements: it needs to be continuous and smooth (to avoid problems with non-existing derivations) and it's value has to decrease rapidly the further the robot is from collision.

For this use we've applied two functions. They can be used not only with the collision criterion, but in other criteria as well. The first function is a simple decadic logarithm with additional parameters:

$$c_c(x) = \begin{cases} a^{\frac{\log_{10}(b+x)}{c}}, & \text{if not in collision,} \\ w_{collision}, & \text{if in collision.} \end{cases} \tag{4.6}$$

where a, b and c are configurable parameters, $w_{collision}$ is the penalization value in case of collision and x is the collision distance. The second function is a logistic function (Fig. 4.7) [16]:

$$c_c(x) = \frac{L_l}{1 + \exp(k_l(x - m_u))} + \frac{L_u}{1 + \exp(-k_u(x - m_u))}, \tag{4.7}$$

where L_l and L_u are the saturation values, m_l and m_u the center points of the transition curves and k_l and k_u the slopes of the transition curves.

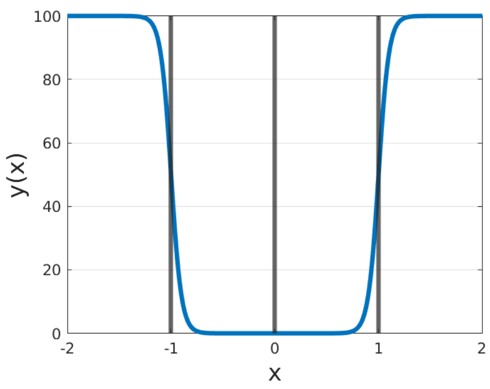

Figure 4.6: Logistic function (Eq. 4.7) - Here, $L_l = L_u = 100$, $m_l = -1$, $m_u = 1$ and $k_l = k_u = 20$. The function doesn't have be symmetrical, but it's the variant that we used the most.

4.3.2 Dexterity criterion

The importance of a good dexterity has been explained in Section 2.4. That's why we've included a dexterity criterion in the optimization problem. In our case, the Jacobi matrix is a 6×9 matrix. While the SVD decomposition doesn't require a square matrix, we need to consider the different units mixed together in the matrix: depending on the type of input and output velocities, $\frac{\text{m}}{\text{m}}$, $\frac{\text{m}}{\text{rad}}$, $\frac{\text{rad}}{\text{rad}}$ and $\frac{\text{rad}}{\text{m}}$ occur, with the matrix composition being:

$$\begin{bmatrix} \frac{m}{s} \\ \frac{m}{s} \\ \frac{m}{s} \\ \frac{rad}{s} \\ \frac{rad}{s} \\ \frac{rad}{s} \end{bmatrix} = \begin{bmatrix} \left[\frac{m}{rad}\right]_{3\times1} & \left[\frac{m}{m}\right]_{3\times2} & \left[\frac{m}{rad}\right]_{3\times6} \\ \left[\frac{rad}{rad}\right]_{3\times1} & \left[\frac{rad}{m}\right]_{3\times2} & \left[\frac{rad}{rad}\right]_{3\times6} \end{bmatrix} \begin{bmatrix} \frac{rad}{s} \\ \frac{m}{s} \\ \frac{m}{s} \\ \frac{rad}{s} \\ \frac{rad}{s} \\ \frac{rad}{s} \\ \frac{rad}{s} \\ \frac{rad}{s} \\ \frac{rad}{s} \end{bmatrix}. \tag{4.8}$$

Soft joint limits									
Joint	q_1 [°]	q_2 [m]	q_3 [m]	q_4 [°]	q_5 [°]	q_6 [°]	q_7 [°]	q_8 [°]	q_9 [°]
Lower limit	$-\infty$	-0.75	-0.75	-90	-90	0	-90	-125	-180
Upper limit	∞	0	1.5	90	155	90	90	0	180

Table 4.2: Soft joint limits

For convenience, the first three columns are shifted to the back, resulting in:

$$\mathbf{J} = \begin{bmatrix} \left[\frac{m}{rad}\right]_{3\times 7} & \left[\frac{m}{m}\right]_{3\times 2} \\ \left[\frac{rad}{rad}\right]_{3\times 7} & \left[\frac{rad}{m}\right]_{3\times 2} \end{bmatrix} = \begin{bmatrix} [\mathbf{J_T}] & [\mathbf{J_P}] \\ [\mathbf{J_R}] & [\varnothing] \end{bmatrix} . \tag{4.9}$$

Entries with $\frac{rad}{m}$ have been omitted, because the movement of the translational external axes has no effect on the rotation of the end effector. The remaining have been concatenated to matrices $\mathbf{J_T}, \mathbf{J_P}$ and $\mathbf{J_R}$.

Using SVD decomposition we compute the three condition numbers, corresponding to $\mathbf{J_T}, \mathbf{J_P}$ and $\mathbf{J_R}$. Using an inverse logarithm, we can penalize very low dexterity values, while the influence of higher values will be negligible. The criterion is included three times (for each of the matrices), in the form:

$$c_{dex}(q) = -\log\left(\frac{1}{\text{cond}(\mathbf{J})}\right) = \log(\text{cond}(\mathbf{J})) . \tag{4.10}$$

This criterion has been introduced already in [25] and has remained largely unchanged.

4.3.3 Joint limits criterion

All of the axes of the robot, except for the turntable, have physical limits (Tab. 3.1 and 3.2). These limits are present in the optimization problem (Eq. 2.2) as bounds on the parameters. But during planning, we want to maintain the joint coordinates within smaller bounds, to discourage the planner from choosing awkward robot configurations and change between them. The change between configurations usually leads through a singularity, which we, too, want to avoid. Therefore, we've defined reduced joint limits, called *soft* joint limits and applied a penalization cost function on them. This is the criterion, where we originally used the Logistic function 4.7. The penalization is soft, meaning the robot is allowed to overstep these limits, if necessary, but we discourage it using this criterion.

The cost function is a sum of all the penalization functions

$$c_j(q) = w_j \sum_{i=1}^{9} \frac{L_l}{1+\exp(k_l(q_i - m_u))} + \frac{L_u}{1+\exp(-k_u(q_i - m_u))} . \tag{4.11}$$

The meaning of the parameters is the same as in Eq. 4.7, only here, q_i are the joint coordinates. This criterion is not present in the weld approach planning using *OMPL*.

4.3.4 Region I criterion

When the robot is welding, it could position itself in two configurations, as visible on Figure 4.7: the „default” depicted configuration (region I) or with its arm „behind its back” (region II). Again, we want to discourage the awkward configuration II, thus we implemented a criterion (c_{rI}), that „pushes” the robot away from the dividing line between regions I and II.

The distance d is calculated using *MoveIt!* functions. If $d > 0$, we consider the robot to be in configuration I. The distance is then input in the logistical function 4.7, to increase the cost of configurations close to the dividing line.

$$c_{rI}(d) = \frac{L_l}{1 + \exp(k_l(d - m_u))}\,, \tag{4.12}$$

with the parameters meaning equal to Eq. 4.7.

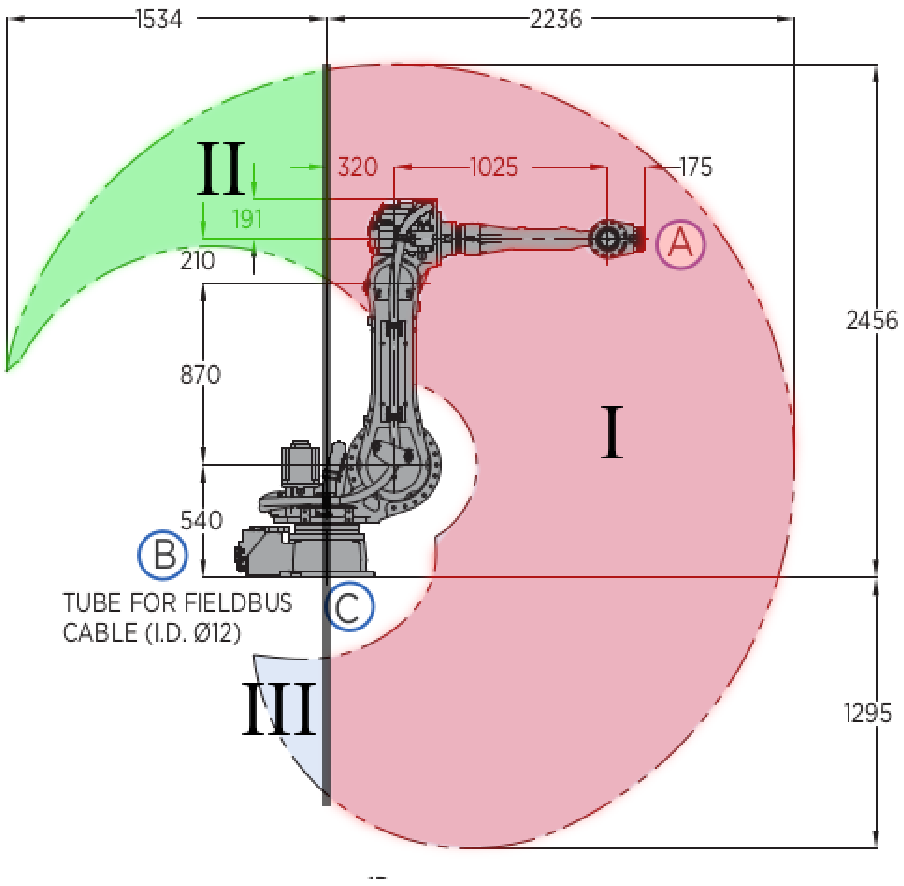

Figure 4.7: Configurations I, II and III (original picture from [22]) - Region I is colored in red, region II in green and region III in grey. The thick dividing line goes through the axis of the first joint and separates the three regions.

This criterion is not present in the weld approach planning using *OMPL*.

4.3.5 Velocity criterion

As of now, the planning doesn't consider time at all. The path is just a sequence of positions without any time stamps. We know, how fast is the extruder tip supposed to be moving, but we lack any frame of reference regarding the joint velocities. This could pose problems during welding, if any of the joints are supposed (based on the planning results) to move faster than their maximal allowed velocity. Therefore we need to implement a criterion, that will keep the joint velocities in check. A simple formulation has been devised in the form:

$$c_v(\mathbf{q_i}, \mathbf{q_{i-1}}) = w_0 \sum_{j=1}^{9} w_j \frac{|\dot{q}_j|}{\dot{q}_{j_{max}}}, \tag{4.13}$$

where $\dot{q}_j$ are joint velocities, w_j the weights and $\dot{q}_{j_{max}}$ the maximal joint velocities. Joint velocities are computed as:

$$\dot{q}_j = q_{j,i} - q_{j,i-1}, \tag{4.14}$$

where q_{ji} and $q_{j,i-1}$ are joint coordinates in the current (i) and previous $(i-1)$ point.

After further experimenting we formulated two alternative versions, how to (approximately) compute the joint velocities. One is based on the knowledge of distance between two weld points and the prescribed velocity of the extruder tip (given in G-Code). With these two, we can calculate the time spent moving between two points as:

$$t = \frac{l}{v}, \tag{4.15}$$

where l is the distance between two points and v is the velocity. From there we can compute the joint velocities as

$$\dot{\mathbf{q}} = \frac{\mathbf{q_i} - \mathbf{q_{i-1}}}{t} \tag{4.16}$$

and proceed as before with Eq. 4.13.

The second alternative involves using the Jacobi matrix. We know from Eq. 2.12 that the Jacobi matrix serves as a operator between joint velocities and the end effector velocities. Because we don't have any information about the angular velocities (yet), we'll use only the translational part of the matrix. Using a pseudoinverse we obtain

$$\dot{\mathbf{q}} = \mathbf{J}^{+}\dot{\mathbf{v}}, \tag{4.17}$$

where $\dot{\mathbf{v}}$ is the translational velocity vector between the points i and $i-1$. We again proceed further with Eq. 4.13.

This criterion is not present in the weld approach planning using *OMPL*.

4.3.6 Previous point distance criterion

In order to eliminate sudden, fast moves, we've included another criterion, similar to the velocity criterion. This one tries to minimize the difference between the current and previous configuration and takes the form

$$c_{pp}(\mathbf{q_i}, \mathbf{q_{i-1}}) = w||\mathbf{q_i} - \mathbf{q_{i-1}}|| \,. \tag{4.18}$$

where $\mathbf{q_i}, \mathbf{q_{i-1}}$ are joint coordinates in the current and previous step. This criterion is not present in the weld approach planning using *OMPL*.

4.3.7 Distance to goal criterion

One of the criteria established in [25] was a distance to goal criterion. It seeks to minimize the translational and rotational error between the extruder and the goal coordinate systems. It consists of two parts: the translational criterion [25]:

$$c_d = \sqrt{(r_{e_x} - r_{g_x})^2 + (r_{e_y} - r_{g_y})^2 + (r_{e_z} - r_{g_z})^2} \,, \tag{4.19}$$

where r_{e_x}, r_{e_y} and r_{e_z} are the extruder tip coordinates, r_{g_x}, r_{g_y} and r_{g_z} are the goal *pose* coordinates. The second is the rotational criterion, which is calculated from the quaternion of relative rotation between the extruder ($\mathbf{q_e}$) and the goal ($\mathbf{q_g}$) [25]:

$$\mathbf{\hat{q}_{rel}} = \mathbf{\hat{q}_g}\mathbf{\hat{q}_e}^{-1} \,. \tag{4.20}$$

Using *Eigen*, the quaternion is transformed to an Angle-Axis rotation description and the angle is extracted.

While this criterion might seem sensible at first, it actually doesn't provide anything useful. Due to the analytical inverse kinematics for the robot arm, if a solution exists, it'll be returned exactly and the extruder and goal Poses will match. That's why we ended up disabling this criterion in weld planning. In weld approach planning, the criterion is still present, but it's debatable, how relevant it is.

4.3.8 No inverse kinematics solution criterion

Looking at the algorithm in Fig. 3.11, the optimizer will choose a set of parameters (positions of the external axes) and then the IKT solver will attempt to solve the for the remaining six joints, to reach the goal Pose. It is therefore possible, that the optimizer might input such positions, that the IKT problem won't have any solutions. In that case, we need to penalize the cost function, as to indicate that such combination of parameters is unacceptable. In the case that no IKT solution is found, we will return a large cost value ($w_{no\,ikt}$) and all other criteria won't be computed.

$$c_{ikt} = \begin{cases} 0, & \text{if IKT solution found,} \\ w_{no\,ikt}, & \text{if no IKT solution found.} \end{cases} \tag{4.21}$$

4.3.9 Weighted sum

Before returning the cost value, all of the criteria (c_i) are summed in a weighted sum. The weights (w_i) allow us to find a balance of importance between the criteria, to scale them to the same magnitude and solve discrepancies in units:

$$c = \sum_i w_i c_i \,. \tag{4.22}$$

4.4 Inverse kinematics

The implementation of the IKT solver considers only solutions in the interval $< -\pi, \pi >$ for the joints *r4* and *r6*. But in reality, the joints are able to rotate full 360° (see Tab. 3.1). This expands the solution set, as we can modify each solution by

$$\begin{aligned} r_4^{II} &= r_4^{I} + \pi \\ r_5^{II} &= -r_5^{I} \\ r_6^{II} &= r_6^{I} + \pi \end{aligned} \tag{4.23}$$

and double the amount of available solutions. The idea to extend the solution set comes from a discussion with Matěj Vetchý, whose Bachelor Thesis also concerned itself with optimal planning of a redundant robot [33].

Chapter 5

Implementation

Until now, we've tried to explain the principles and methods used in this project. Here, we'll focus on the implementation aspect, going over the entire source code, explaining the thought process and the internal workings of the software.

5.1 Workspace composition

Projects in ROS are generally contained in a workspace, which is created by the ROS build system, *catkin*. Here, *packages* are being created by the user and *catkin* is responsible for handling the source code, libraries, compilers and all other dependencies. A *package* can contain multiple nodes, libraries, configuration files, message definitions and more. *Catkin* integrates tightly with *CMake* and each package needs a `CMakeLists.txt` file, declaring all the compilation targets, libraries, etc [1].

Our project consists of four packages: *cell* and *cell_description* contain the robot model description. *gcode_parser* handles the G-Code parsing, visualizing and publishing. It also contains launch files that are used to run the software. The source code for the planner is located in the package *planner*. It also contains the plotting *nodes* used for graphing the progress and result of the optimization.

5.2 Running the planner

In an ideal case, the engineer operating the software shouldn't need to do any major changes before running the planner. It might be only necessary to correctly set the path to the input file containing the G-Code and the 3D model of the tank. After that, the user only has to navigate to the root folder of the workspace and run

Listing 5.1: Run static part of the planner

```
source devel/setup.bash
roslaunch gcode_parser no_debug.launch
```

in one terminal and

Listing 5.2: Run dynamic part of the planner

```
source devel/setup.bash
roslaunch gcode_parser debug.launch
```

in another. The first command is common and is used to load the ROS-specific commands to be recognized by the shell. The second command runs all the commands in the respective *launch* files. The difference is that the command in 5.1 runs those *nodes* and loads that configuration, which is static, i.e. doesn't depend on the configuration nor the status of the planner and can therefore remain running during any number of runs of the planner itself. In contrast, the command in 5.2 runs the *nodes* that need to be re-run every time for the planning and loads the configuration that can change with every run of the planner. This rule of two separate *launch* files applies for debugging as well, which is symbolized in the filenames.

5.3 Robot model description

Most of the files contained in *cell* and *cell_description* have been generated by *MoveIt! Setup Assistant* at the beginning of the project during Ing. Horný's Master Thesis [25]. In *cell_description* we'll find the *URDF* files fo the robot model and the STL meshes. *cell* contains the *SRDF* file and a lot of *YAML* configuration and *launch* files. Some of the files were used in older versions of the project, such as `kinematics.yaml`, which was used with an older version of the inverse kinematics solver. Other files remained unchanged as they were generated by *MoveIt!*. Relevant launch files are being called during startup to run all of the *nodes* and load all the parameters required by *MoveIt*!

5.3.1 URDF

Originally, in [25], the entire model was contained in one file, but for clarity, the it has now been split into multiple parts. The main file, `cell.urdf.xacro` contains the external axes and the extruder and links to the file `gp88_macro.xacro` containing the model of GP88. This file has been taken from the *ROS-Industrial* project [13], to provide a degree of unification in the description.

A supplemental file, `links_joints.urdf.xacro` exists as well. It contains macros written in the *Xacro* language. The macros written in this language allow for reusability and configuration of *URDF* components. It also helps to reduce duplicity and improve readability [20]. In this project the macros are of little relevance and could possibly be removed to streamline the code.

The main *URDF* file is split into four distinct parts: Table, Frame, GP88 and Tools. Each part starts with the links and the relevant joints follow. Joints can be either fixed or movable (continuous, revolute, prismatic). Here, the joint limits have to be set properly, otherwise MoveIt! could be unable to execute the desired movement. Joints always have a parent and a child link, i.e. the model has to begin and end with a link. Here they are `tool1` and `tool0e` respectively. The joint naming convention, for clarity, follows the scheme of `<parent link name>_<child link name>`. All of the links have a visual and collision geometry. The external axes and frame are constructed from primitives (box, cylinder), just the GP88 and the extruder have detailed STL meshes. In the future, the primitives will be replaced by meshes of the real robot cell frame as well, but this hasn't been a priority, since this change only affects the visual part and does little do further the progress on the planner itself.

What's more relevant though is to update the old extruder model with unidirectional heater nozzle to the new model with concentric model. This would be a significant change for the planning process, but unfortunately, hasn't been completed in time for this Thesis. Another improvement that would be beneficial is to remove the tank from the *URDF*. As of right now, the tank is a link in the robot model, which doesn't represent the reality well. More so, to change the tank, the *URDF* has to be edited to load the desired mesh, which is also not desirable from the perspective of the end user, who shouldn't edit any files not explicitly allowed. Instead, the tank should be loaded directly from the code as a collision model in the environment. This has the added benefit that the mesh can be changed without needing to shutdown and rerun the whole software.

5.3.2 SRDF

Correctly setting up the *SRDF* file is of high importance, as without it, *MoveIt!* won't be able to correctly identify joints and links belonging to a group and move them. While this file (`cell.srdf`) too, has been initially generated by *MoveIt!*, it has since then been completely rewritten. The first part of the file contains *JointGroup* definitions. We're using two *groups*: `all`, containing all of the robot cell joints and `robot_arm`, containing only the GP88 joints. This distinction has been made due to the two different methods used during planning of weld approach and weld moves. This way, we can separately move the GP88, whose configuration is being calculated using the IKT solver. A third *group*, `end_effector` is present as well, but it's existence is mostly formal: It groups together the links belonging to the extruder and is not in any way used during planning.

The *SRDF* file also defines the end effector. Here it is the `tool0e` frame, positioned at the tip of the extruder.

Another important part is the disabling of collision checking between links. This part corresponds to the Table 4.1, only in written form.

Lastly, the *SRDF* file defines home positions of the robot (a set of joint coordinates for a particluar group) and virtual joints, connecting the robot to the global, world coordinate frame. These settings are mentioned only for completeness and their relevance in this project is marginal.

5.4 G-Code Parser

The idea and reasons behind the G-Code parser have already been explained in section 3.2.2. Outside of the parser itself, the package also contains launch files for the whole project. After parsing, the G-Code can be visualized in *RViz*, where every *pose* will be displayed as a small coordinate system and a simplified extruder model will move along the path. This has proven to be very useful when verifying that the G-Code has been parsed correctly. We've also added the option of exporting the G-Code to a CSV file An example of a parsed G-Code is shown in Fig. 5.1.

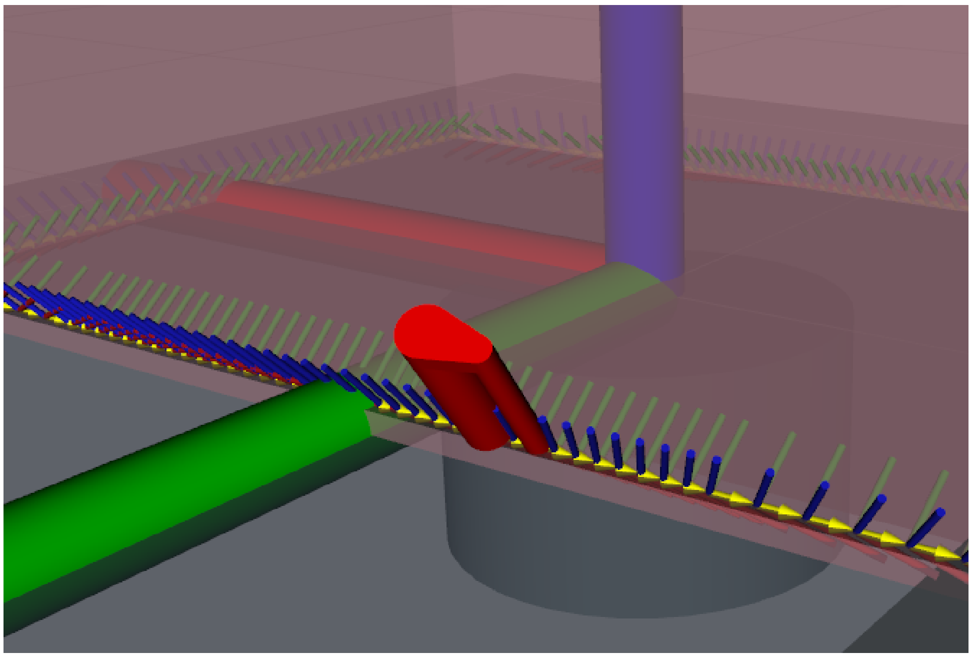

Figure 5.1: Visualizing the G-Code - The two joined parallel cylinders form the simplified extruder. The thinner cylinder is the preheat nozzle, the thicker the extruder nozzle. The weld points, parsed from the G-Code are shown as small coordinate systems (x, y and z axes colored in red, green and blue respectively). The partially opaque pink block is the tank, with the large axes depicting the world coordinate system. Yellow arrows point in the direction of welding.

5.4.1 Usage and configuration

We tried to separate the parser *node* (`gcode_parser.cpp`) from the parser class (`libgcode_parser.cpp`) itself, which is instantiated and its members and methods are called. This leaves us with a fairly simple code for the *node*, which consists mostly of evaluating various configuration parameters and the calling the relevant methods or setting variables. The parameters are loaded from the `gcode_parser.yaml` file:

`world_coord_system` Coordinate system used for visualization of the G-Code.

`visual_markers_topic` Name of the ROS *topic*, where the G-Code visualization will be published.

`interpolate_step` If set to greater than one, the line between two points from the G-Code will be linearly interpolated with the amount of points given.

`interpolate_g00_select` If set to `yes`, also the lines between `G01` points will be interpolated.

`sixth_rot_select` Sets if the computation of the sixth rotation will be performed on the `interpolated` or `original` data.

`end_effector_coord_system` Name of the coordinate system representing the simplified extruder.

`end_effector_topic_rate` The frequency at which is the simplified extruder moved along the parsed G-Code path.

`visualize_welding_traj` Takes `true` or `false` to enable or disable the visualization of parsed G-Code.

`save_select` Selects, which parsed data will be saved to the CSV file. The possible options are: `original_no_rot`, `original_6th_rot`, `interpolated_no_rot` and `interpolated_6th_rot`.

`csv_export_file_path` Path, where the CSV file should be created.

Another *YAML* file is present, `libgcode_parser.yaml`, concerned with the configuration of the parser class itself:

`angle_units` Sets, if the angle units in the G-Code should be interpreted as radians (`rad`) or degrees (`deg`).

`rotation_description` Sets, which type of rotation description should be used when interpreting the G-Code rotations. Currently, only Roll-Pitch-Yaw (`rpy`) is implemented.

`rotation_axes_labels` Takes an array of characters representing the rotations in G-Code. Most common variations are either `A B C` or `I J K` for the three angles. Lowercase variants need to be specified as well.

Other configuration files in the package bear little significance and are mostly remnants of older, abandoned or reworked functions. The parser *node* could be run independently, provided that the relevant *YAML* files are loaded to the *parameter server*. This, however serves little purpose.

After loading all of the parameters, the parser *node* will create an instance of the `GCodeParser` class, calling the method `load()` with the argument being the path to the G-Code file to be parsed. Following is a call to the `search()` method, which will perform the parsing itself. With the parsing complete, it's possible to compute the missing sixth rotation (Section 4.1), using `add6thRotation()` and interpolate the data with `interpolate()`. The order of these two methods depends on the setting of the `sixth_rot_select` parameter.

With these methods, the parsing is finished. What follows are calls to export the parsed data to a CSV file (`saveToCSV()`) and a ROS *service*, named `weld_data` is advertised. When a request arrives to this *service*, the response will contain a structure with the parsed data.

Lastly, the parsed G-Code is visualized: all of the *poses* as small coordinate systems, with arrows between them showing the direction of movement. Then the movement is visualized, with the simplified extruder moving from one point to the next. This is done by repeatedly transforming the coordinate system of the simplified extruder to the *pose* of the next parsed point with a frequency set by the `end_effector_topic_rate` parameter.

Launch files

The parser *node* contains two critical launch files, used in running the whole planning software. The reasoning and their function have been explained in Section 5.2.

The short `no_debug.launch` mainly calls other launch files, created by *MoveIt!*, that set up all of the required nodes and parameters required. The *URDF* robot description is also loaded through them. The *URDF* robot model for the simplified extruder is also loaded here. The only relevant parameter, that could be meaningfully changed, is `use_gui`. Setting it to `true` opens a window with sliders for each robot joint, allowing manual control over the joint position, while the changes are displayed in *RViz*. This setting is useful to verify that the *URDF* model is correct (e.g. after a change) and all joints are moving as supposed. In normal operation it remains set to `false`.

Before the graph plotter *nodes* have been created, this launch file also ran a oscilloscope-style plotting *node*, called *PlotJuggler*. It subscribed to topics with the values of the optimization criteria and displayed their evolution. It has proved to be quite unwieldy, but if a need to use it arises, a *PlotJuggler* configuration has to be loaded after the `debug.launch` has been launched, but before the planning itself started. Otherwise all of the *topics* need to be resubscribed to.

The more relevant launch file is `debug.launch`. It launches the parser, planner and plotter *nodes* themselves, as well as loads all of the *YAML* configuration files. In case the user doesn't want to run the planner, it's possible to set the argument `only_parse` to `true`. A *node*, that's currently not in use, but could possibly be useful, is *rosbag*. It records and saves all of the *messages* published on selected *topics* and allows them to be replayed at a later time. The issue with using *rosbag* is that the files rapidly gain in size (up to lower GBs) and easily fill up the hard drive.

5.4.2 G-Code Parser Class

The development of the `GCodeParser` class was done while attempting to adhere to OOP(Object-oriented programming) principles. If this has succeeded is up for discussion.

Definition of the class and it's members is contained in the file `libgcode_parser.hpp`. Outside of the methods called in the parser *node* itself, exposed to external use are *enums* for various configuration parameters (these are mostly superfluous and could be removed to streamline the class) and a data structure definition for the parsed data. This *struct* consists of three vectors, one for *poses*, one for velocity and one for the weld movement type (named `command`). Elements with the same index in the vectors correspond to the same line in the G-Code.

Under the `private` keyword we will find all of the remaining methods, as well as other member variables. Noteworthy is `parsing_data_buffer`, a vector of strings, where each element (string) contains one line of the unparsed G-Code file. The member variable `global_coordinates_iterator` keeps track of the last line number parsed. Since G-Code is usually written using millimeters as units, but ROS uses meters, the constant `SI_conversion` provides the unit conversion.

When instantiated, the class will load all of the configuration parameters from the *parameter server* and is ready to use. With the method `load()`, the file is read and line after line assigned to `parsing_data_buffer`. Calling `search()` will begin the parsing process. In a `for` cycle, the line is first processed by the method `findGCommand` and after that by `findPointCoordinates`.

All of the parsing in the Class is based on *Regular expressions* (*Regex*). Used by search-and-replace algorithm, a string is searched for sequences matching a given *pattern* [18]. For the G-Code, separated patterns have been created for parsing the G-Code command, the coordinates and the velocity.

In `findGCommand` a pattern is applied to recognize the G-Code command. The string is stripped of the letter G and converted to an integer. For each recognized command, a method has been created, which is selected by a `switch-case` block, based on the integer. There are multiple recognized G-Code commands, but the only ones currently relevant are `G01` and `G00`. Other commands, such as `G68.2` have been used in a specific, older G-Code version, which is not used anymore.

The structure of the methods `G00()` and `G01` is almost identical and very similar to the method `findPointCoordinates`, so they'll be explained together. The latter method parses lines, where no G-Code command is present, only the coordinates. In that case it is assumed, that the command remains the same as on the previous line. The match pattern in all three cases is similar, with six capture groups, which allow us to reference the found coordinates later on. A separate pattern is used for finding the velocity as well. The found coordinates, still in string form, are passed to the `coordinatesFromRegexMatches()` method, to be transformed to a `Pose` data type. The same applies for the velocity as well, but only for the `G01` command. For now, it is assumed, that the movement during weld approach (`G00`) is not limited by a maximal velocity.

The method `coordinatesFromRegexMatches()` simply goes through all of the strings with coordinates and extracts the value based on the first character. Translation is directly assigned to a variable of the type `geometry_msgs::Pose`, but the rotation first needs to be converted from RPY description to a quaternion. This entire `Pose` variable, containing the position and rotation of the coordinate system at a given path point is then returned. In case any coordinate is missing, it is assumed, that it's value is the same as on the previous line.

By arriving at the last line, the parsing is complete and the parsed data can be received by calling `getParsedData()`.

Additionally, with the method `interpolate()`, interpolation can be applied on the data. Translation is interpolated linearly and rotation using *Slerp* (Spherical linear interpolation). The difference is that quaterion rotations are interpolated on an arc, instead of a line, giving more accurate results [19].

5.5 Planner

Central to the planning software is the *planner* package. It consists of a `complete_weld` *node*, multiple libraries (`ikt6.h`) and C++ classes (`OmplPlanner.hpp`, `CeresOpt.hpp`), various configuration files (*YAML* files with optimization parameters, IKT configuration, etc.) as well as of two plotting nodes (`optim_plotter.py` and `joints_plotter.py`).

5.5.1 Usage and configuration

The `complete_weld` node is run from the common launch file, `debug.launch`, as it's closely connected to the other *nodes*, particularly th G-Code parser.

After startup, the node will send a request on the `weld_data` *service* and wait for the response, containing the parsed data. If the response has been successful, *MoveIt!* objects will be initialized, the most important being: *RobotStatePtr* (named `kinematic_state`, contains information about the current state of the robot), two JointGroups (`robot_arm_joint_model_group` and `all_joint_model_group`) and a *PlanningSceneMonitor* (responsible for managing the planning environment, handling collisions, etc. named `psm`).

Because we want to publish the progress of the planing (the evolution of the value of the cost function), we need to advertise various *topics*. There are various *topics*, separate ones for each weld approach and for weld move planning, usually one for each optimization criterion. These are mostly for legacy reasons (*PlotJuggler* compatibility) and a single *topic* (`optim/compact`) has been created, which accumulates all of the criteria values and publishes them at once. A separate *topic* also publishes the resulting joint positions, i.e. the result of the planning. The initialization of the topics is done here, because a named *node* is needed to publish topics. The planning libraries themselves don't have *node* status (they are only called from the planner *node*, not run separately), thus they can't advertise topics. A user prompt has been setup here to allow *PlotJuggler* to subscribe to all of the advertised *topics* before planning starts.

Last set-up needed before planning can begin, is the instantiation of the classes themselves (`CeresOpt` and `OmplPlanner`) with all of the necessary arguments (`kinematic_state`, both *JointGroups* and *PlanningSceneMonitor*). Then, the starting position is set, from the parameter `start_position` and planning can begin by calling the `pointByPointPlanning()` function.

All of the configuration parameters are contained in YAML files `common_setup.yaml, optim_config.yaml, Ceres_optim_config.yaml, OMPL_optim_config.yaml` and `robot_ikt_config.ikt`.

The file containing the most used parameters is `common_setup.yaml`

`gcode_file_path` Absolute path to the G-Code file to be loaded and parsed

`display_select` Takes the same arguments as `save_select` in `gcode_paser.yaml` selecting which parsed data will be displayed in *RViz*.

`export_select` Takes the same arguments as `save_select` in `gcode_paser.yaml` selecting which parsed data will be sent to the planner.

`start_position` A vector of initial joint coordinates.

Configuration parameters relevant to both weld approach move and weld move planning are contained in `optim_config.yaml`

`joint_limits_lower` A vector of lower joint limits.

`joint_limits_upper` A vector of upper joint limits.

`soft_joint_limits_lower` A vector of lower soft joint limits.

`soft_joint_limits_lower` A vector of upper soft joint limits.

`v_limits` A vector of joint velocity limits.

Parameters pertaining to weld move planning (*Ceres*) are gathered in `Ceres_optim_config.yaml`

`Ceres_opt_w` A dictionary of `{key:value}` pairs containing optimization criteria weights.

- **`trans_dex`** Weight for the $\mathbf{J}_T$ part of the dexterity criterion.
- **`rot_dex`** Weight for the $\mathbf{J}_R$ part of the dexterity criterion.
- **`prism_dex`** Weight for the $\mathbf{J}_P$ part of the dexterity criterion.
- **`region_I_boundary`** Weight for the region I criterion.
- **`velocity`** Weight $w_0$for the velocity criterion.
- **`prev_p_dist`** Weight for the previous point distance criterion.
- **`joint_limits`** Weight for the soft joint limits criterion.
- **`collision_k`** Parameter `k` in the logistic function (Eq. 4.7) for the collision distance criterion.

- **`collision_dist_margin`** Parameter `m` and `b` in the logistic function (Eq. 4.7) and the logarithm function (Eq. 4.6) for the collision distance criterion respectively.
- **`collision_dist_div`** Parameter `c` in the logarithm function (Eq. 4.6) for the collision distance criterion.
- **`in_collision`** Penalization value to be returned for when the robot is in collision.
- **`no_ik_solution`** Penalization value to be returned for when no IKT solution has been found.

`Ceres_v_limits_joints_w` A vector of weights w_i in the velocity criterion.

`Ceres_mul` The `mul` constant used for scaling the optimization parameters.

`Ceres_soft_joint_limits_function` By choosing either `sig` or `log` we set the logistic function (Eq. 4.7) or the logarithm function (Eq. 4.6) to be used in the soft joint limist criterion.

`Ceres_joint_vel_comp_method` By choosing either `jac`, `dist_t` or `dist` we choose which method for computing joint velocities will be used.

`Ceres_planner_float_options` A dictionary with settings for the optimizer

- **`max_iter`** Maximal number of iterations allowed for the solver.
- **`step_size`** Initial step size used in the *Trust region* algorithm (Alg. 2).
- **`initial_trust_region_radius`** Initial size of μ in the *Trust region* algorithm (Alg. 2).

`Ceres_soft_joint_limits_param` A dictionary of the parameters `Ll`, `Lu`, `kl` and `ku` for the soft joint limis criterion.

5.6 Planning a weld

As already mentioned, two distinct types of moves occur during planning (weld approach and weld move) with the chosen planning methods being applied. The function `pointByPointPlanning()` iterates through all of the weld points in a `for` cycle and based on the G-Code command calls a corresponding planning method (*OmplPlanner* for `G00` and *CeresOpt* for `G01`). After finishing, the results (joint coordinates) are retreived and stored in a common *RobotTrajectory* structure and the starting joint coordinates are updated. This provides the planner with a initial position, that should ideally be close to the one being solved.

It needs to be said, that much more effort has been put in the weld move planning with *Ceres*, compared to the weld approach move. This resulted in not all functionalities and criteria being implemented within weld approach planning. Some parts of the code also remained as they were written in [25].

5.6.1 Optimization criteria and constraints

There's a slight difference between the approach used during weld approach move planning and weld move. With *OMPL* (using *RRT*), each state is checked for validity. If a collision occurs, the state is not admitted at all. Meanwhile, planning with *Ceres* doesn't have this functionality, so we have to use different means, such as returning an extremely high cost (penalization).

It also has to to be differentiated between checking for constraints, i.e. collision and between a criterion (applicable only when not in collision), that tries to push the robot away from collision.

During weld move planning (*Ceres*), we are using all of the criteria (Section 4.3), except for the Distance to goal criterion. Meanwhile, during weld approach planning (*OMPL*), we're using only the Collision criterion, Dexterity criterion, Soft joint limits criterion and the Distance to goal criterion. It is also possible to plan the weld approach move completely without any criteria, using bare *RRT*.

Collision criterion

For weld move planning, the criterion was tried in both variants. With the logarithm (Eq. 4.6), the parameters `b, c` correspond to the parameters `Ceres_opt_w.collision_dist_margin` and `Ceres_opt_w.collision_dist_div` respectively. The parameter `a` (in code named `collision_coeff`) is calculated as

```
a=-in_collision / (log10(collision_dist_margin) /
    collision_dist_div);
```

with the idea being following: We want the function to return the same value as if the robot was already in collision (`Ceres_opt_w.in_collision`) when the distance drops under a set safety margin (`Ceres_opt_w.collision_dist_margin`). This will give us if not smooth, but at least continuous function.

With the logistic function, we used only the descending half of Eq. 4.7, with the parameters `L1, k1, m1` corresponding to `Ceres_opt_w.in_collision`, `Ceres_opt_w.collision_k` and `Ceres_opt_w.collision_dist_margin` respectively.

For weld approach move, an older, simpler version of Eq. 4.6 was used, with only the parameter `a` being equal to negative of `OMPL_opt_w.collision_dist_mult` and `b, c` equal to zero and one respectively.

Region I distance criterion

The distance to the Region I/II dividing line is computed using *MoveIt!*. We set the robot to the position given by the joint coordinates and query for two transformations: one from the world coordinate system `tool1` to the first link `robot_link_1_s` ($\mathbf{T_{t1,1s}}$) and the second one again from the world coordinate system `tool1` to the fifth link `robot_link_5_b` ($\mathbf{T_{t1,5b}}$). Using inversion we find the transformation between `robot_link_1_s` and `robot_link_5_b`:

$$\mathbf{T_{1s,5b}} = \mathbf{T_{t1,1s}}^{-1}\mathbf{T_{t1,5b}} \tag{5.1}$$

and take the x value as the distance d.

No inverse kinematics solution criterion

As mentioned in Section 4.3.8, it is possible that no IKT solution will be found for a given goal *pose* and a triplet of optimization parameters. We want the optimization solver to avoid stepping into this region altogether. In optimization, penalizing unwanted parameter values is a common approach, especially since *Ceres* doesn't support equality constraints, which would offer a much more elegant approach to this problem. One half-way solution, proposed by *Ceres* developers, is to return `false` from the function evaluating the criteria. Optimizer interprets this as a evaluation failure and assigns infinite cost to it [23]. In our experience, both approaches have been about equally successful.

5.6.2 Weld approach move

Weld approach planning with *OMPL* follows the outline presented in Section 2.1. The method `plan()` is called with the goal *pose* and initial joint coordinates as arguments. First, we create a state space ($\mathbb{R}^9$). To the state space we assign a *StateValidityChecker*, which we implemented in the class `ValidityChecker`. Further, we initialize a *ProblemDefinition* and assign it a optimization objective. The objective is, again, a separate class, named `PathObjective`, where the criteria will be evaluated and returns the cost value. Lastly, we define a start state and a goal state. The goal is a class as well (`PathGoalRegion`), containing a metric defining the distance to the goal.

If we're using *RRT**, we can start planning. Even though the planner might find a solution path very quickly, it's advisable to let the search continue for longer and let the planner explore the state space, because a more optimal path might be found.

For *RRTConnect*, the setup remains mostly the same, with one important difference: because we're building a tree from the goal as well, we need to find a state that corresponds to the goal, i.e. we need to find a IKT solution for the goal Pose. This is accomplished by the means of a sampling function (`findGoalState()`). This function samples the state space in a specific way to find the goal state. In our case we are simply calling the weld move planner (Ceres) with only one point in the path: the goal. With the goal state known, *RRTConnect* can plan a path.

Regarding the aforementioned classes, `ValidityChecker` implements only one method, `isValid()`, which takes a state as an argument and, using *MoveIt!* and the *planningSceneMonitor* returns `true` or `false` based on whether the state is valid (not in collision) or not (in collision).

The class `PathGoalRegion` also implements only one method, `distanceGoal()`. It takes a state as an argument and, using *RobotState* calculates the *pose* of the extruder tip. Using the equations 4.19 and 4.20 it calculates the distance and angle error. Since we need the extruder tip to match not only the goal position, but also rotation, we need a way to combine these two values, which have different units. The metric proposed in [25] is

$$d_g = w_1 c_d + \frac{w_2 c_r}{4}\,, \tag{5.2}$$

where c_r is the angle from the Angle-Axis rotation description and w_1 and w_2 are weights.

Lastly, the class `PathObjective` implements a method, `computeStateCostParts`, where the optimization criteria are calculated. We're using the dexterity criterion, distance criterion, the collision criterion and the soft joint limits criterion, all calculated as explained in Section 4.3. All of the criteria values are published on their respective topics and a weighted sum is returned as the cost.

5.6.3 Weld move

In the weld move planning, the core method is named `planCeresPath2()`. Here a *Ceres Problem* is initialized and a *CostFunction* assigned to it. Even though *Ceres* strongly suggests using Automatic Differentiation, we're forced to use Numeric Differentiation, as the cost function (or function that it calls) are not templateable [23]. A vector of optimization parameters, named `x` is also assigned to the *Problem*. The vector contains initial joint coordinates for the three external axes, scaled by a constant `mul`. This is done to precondition the problem, as the parameters have different units. A value that has worked well for us is 10 000.

After setting the parameter bounds and the optimizer options, we run the solver. The optimizer will call the cost function to evaluate the criteria, which is done in the `CostFunctor2` *struct* by calling the overloaded `operator()()`. Here, the criteria are evaluated, similarly as in weld approach planning. Firstly, the input parameters are scaled by $\frac{w}{mul}$ and with them, an attempt is made to solve the inverse kinematics. If it's successful, all of the criteria are evaluated and summed. Otherwise the criteria are assigned a `NAN` value (except for the no inverse kinematics solution criterion) and `false` is returned by the function.

The criteria used here are the dexterity criterion, velocity criterion, soft joint limits criterion, region I distance criterion, previous point distance criterion and the collision criterion. All of the criteria then are saved and published to their respective *topics*.

Inverse kinematics

The Inverse kinematics solver plays a crucial role during the weld planning. After initialization we pass to it the robot link dimensions, joint zero coordinates offsets, the directions of positive joint rotations and joint limits. This is done to adjust the coordinate systems of the universal internal kinematic robot model of the IKT solver to the robot model used in our problem. Because the IKT kinematic robot model doesn't contain any end effector and solves the IKT simply between the robot base and the mounting flange, we need to pass to the solver the transformation matrix between the extruder tip and the flange. With it, the solver can transform the goal *pose* to the flange coordinate system and solve the IKT.

Before attempting to solve the IKT, we need to transform the goal *pose* from the global coordinate system of the entire robot cell to the coordinate system of the robot arm base.

$$\mathbf{T_{gT}} = \mathbf{T_{t1-rb}}^{-1}\mathbf{T_{g0}} \tag{5.3}$$

Where $\mathbf{T_{t1-rb}}$ is a transformation matrix between the global coordinate system `tool1` and the `robot_base_link` coordinate system. $\mathbf{T_{g0}}$ is the untransformed goal *pose*.

The solver will return a $6 \times n$ matrix, where n is the amount of solutions found. If at least one of them is non-NAN, we continue further. Otherwise a `false` is returned.

If there are any non-NAN solutions, we'll first expand the solution set by computing the solutions outside of the $< -\pi, \pi$ interval for joints r_4 and r_6 as described in Section 4.4.

In Section 4.3.4 we've established that we prefer the configuration of region I. To avoid the possibility of entering the other regions, we check if $d > 0$ in the Region I distance criterion for each solution and leave out those, which don't meet this condition. From the remaining solutions we want to pick that one, which is closest to the solutions of the previous point solved. This is done by finding the minimum of the norm

$$||\mathbf{q_{ij}} - \mathbf{q_{i-1}}|| \tag{5.4}$$

for all j solutions. $\mathbf{q_{i-1}}$ are the joint coordinates of the previous point. The solution with the smallest norm will be returned as the chosen one.

5.6.4 Visualizing and data export

When the planning of the entire weld path is finished, the *complete_weld node* extracts the sequence of joint coordinates and publishes it on two *topics*: one visualizes the path of the tip of the end effector in *RViz* and the other publishes the joint coordinates sequentially for each point with a given frequency. The second *topic* is subscribed to by an *MoveIt!* object in *RViz*, which sets the joints of the visualization model to give the impression of an moving robot. The whole movement can be replayed, paused and stepped by using the *TrajectorySlider*.

The *joints_plotter.py node* will also subscribe to the first *topic* and plot a graph with the joint coordinates over the weld points as well as a text file with the values and save them to a file.

After each planned weld point a *message* is published, containing the evolution of all the criteria and the joint coordinates during a single point optimization. This mesage is read by the *optim_plotter.py node*, which plots a graph for each criterion and saves it as an image. It also saves the criteria values in a text file for possible future use or detailed analysis.

Chapter 6

Experimental results

6.1 Bottom outer weld of a rectangular tank

The planning and optimization were tested on a G-Code for welding the floor to the walls of a rectangular tank (Fig. 6.1). The weld is an outside weld. During the development of the software we preformed tests with various settings, modifications and criteria weights. Here we're presenting only two select variants, differing by the used method for computing the joint velocities. Unfortunately, due to the modifications in code, the weld approach planning, which previously was functional, has stopped working and we were unable to fix this issue until the thesis' deadline. But it also needs to be said that the importance of the weld approach move planning is small compared to the weld move planning. The weld approach planning could possibly be done even without any optimization, using just bare *RRT* and the results would still be acceptable.

6.1.1 Weights values

The sensitivity of the optimization to the various weights varies widely. While some can be changed in a wide spectrum of values, in others even a small change results in a completely unsuccessful planning. These are the weights we used for the final results:

`trans_dex` $= 20$

`rot_dex` $= 20$

`prism_dex` $= 20$

`region_I_boundary` $= 2$

`velocity` $= 0.1$

`prev_p_dist` $= 2$

`joint_limits` $= 1$

`collision_k` $= 20$

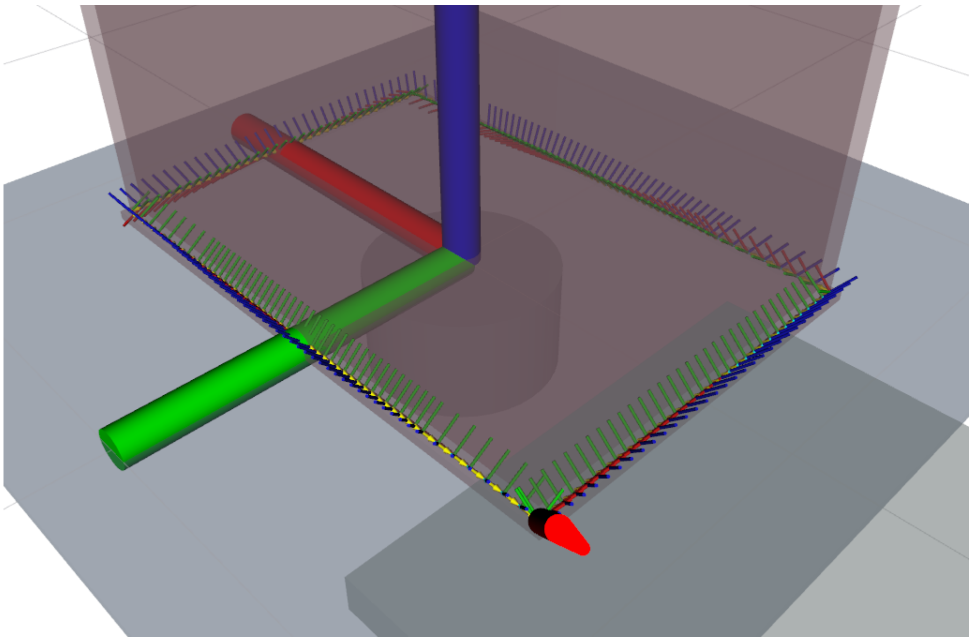

Figure 6.1: Visualization of the G-Code used for planning - The tank is a rectangle tank with a floor. The weld runs around the bottom circumference of the tank and joins the floor panel to the four walls.

`collision_dist_margin` = 0.05

`collision_dist_div` = 10

`in_collision` = 250

`no_ik_solution` = 1000

`Ceres_v_limits_joints_W` = 1 for all joints

`Ceres_mul` = 10 000

`Ceres_soft_joint_limits_function` = `sig` (logistic function, Eq. 4.7)

`Ceres_joint_vel_comp_method` = `jac` or `dist_t`

`max_iter` = 1000

`step_size` = 0.01

`initial_trust_region_radius` = 100

`Ceres_soft_joint_limits_param` :

- **Ll** = 10
- **Lu** = 10
- **kl** = 2
- **ku** = 2

6.1.2 Path planning results

Figures 6.2 and 6.3 show the graphs produced by the planner. A video of the result is available as an appendix. The first thing we notice, that the joint coordinates are continuous, but not smooth. There are jumps occurring in them and they mainly correspond to the corners the robot has to maneuver around. If we look just at the optimized parameters, we see that the table (q_1) has on a few occasions rotated slightly backwards. While this is not necessarily bad, we'd prefer a monotonous trajectory with as little sharp changes as possible.

Looking further at the joint velocities, they are well under the maximal joint velocity limits, but the three sharp changes due to corners are visible here as well.

The pattern with sharp jumps and spikes is visible on most of the criteria as well. A big role here plays the collision checking. We're able to get only the closest collision distance from *MoveIt!* and that's usually between the extruder and the tank. Only when the robot comes very close to the tank, the distance abruptly changes and this leads to „stuttering" of the robot, which moves back and forth as the minimal collision distance changes.

We want to present the results, when we used „time and distance" to calculate the joint velocities as well (Fig. 6.4 and 6.5). Overall, the trend with jumps is similar to the first results, but what's strikingly different are the joint velocities. There are multiple noticeable spikes, where they overstep the maximum allowed velocity. We consider this to be much more faithful to the reality, because some of the jumps in joint coordinates are simply too big to stay within the velocity limits.

6.1.3 Single point planning result

In Figures 6.6 and 6.7 we're presenting the evolution of the costs, parameters and joint coordinates during the optimization of a single point. In this case it's the first point of the weld. The stair-like appearance of the graphs has to do with the way how the optimizer evaluates the gradient of the cost function. We can see discontinuities where no IKT solution was found. Overall, we don't see large improvements in the criteria or big changes in the parameters at the end. That's due to the fact that the points are spaced very closely and only a small change in the coordinates is necessary to reach it.

As with the results of the entire planning, we want to present the results when we used „time and distance" to calculate the joint velocitiesas well (Fig. 6.6 and 6.7). Again, the main difference lies in the joint velocities, but it can be seen, that in this case, the optimizer was able to minimize the criterion and the velocities remained under the limits at the end of the optimization. This, unfortunately wasn't the case for all points.

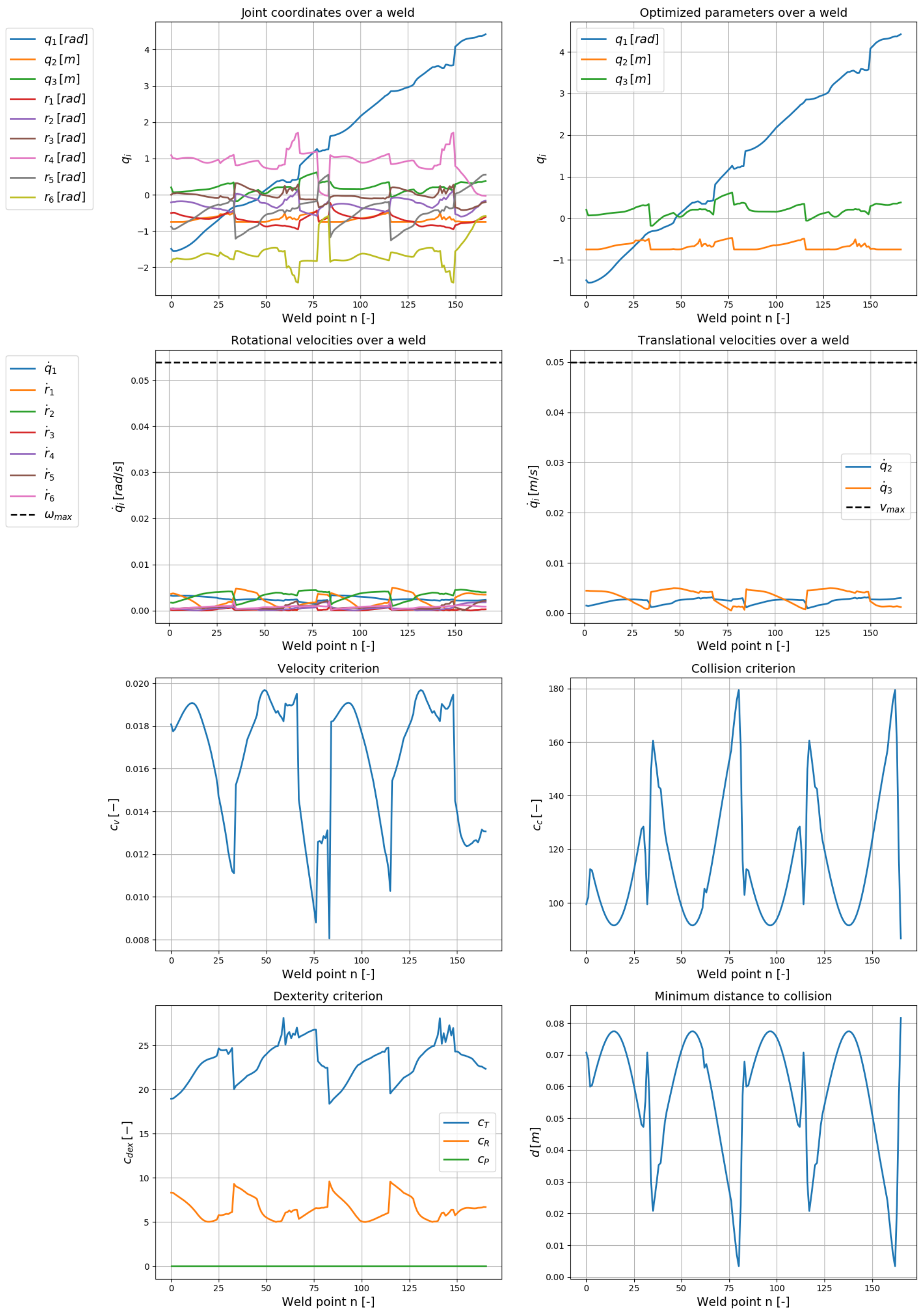

Figure 6.2: Results of an optimization (velocity through Jacobi matrix)

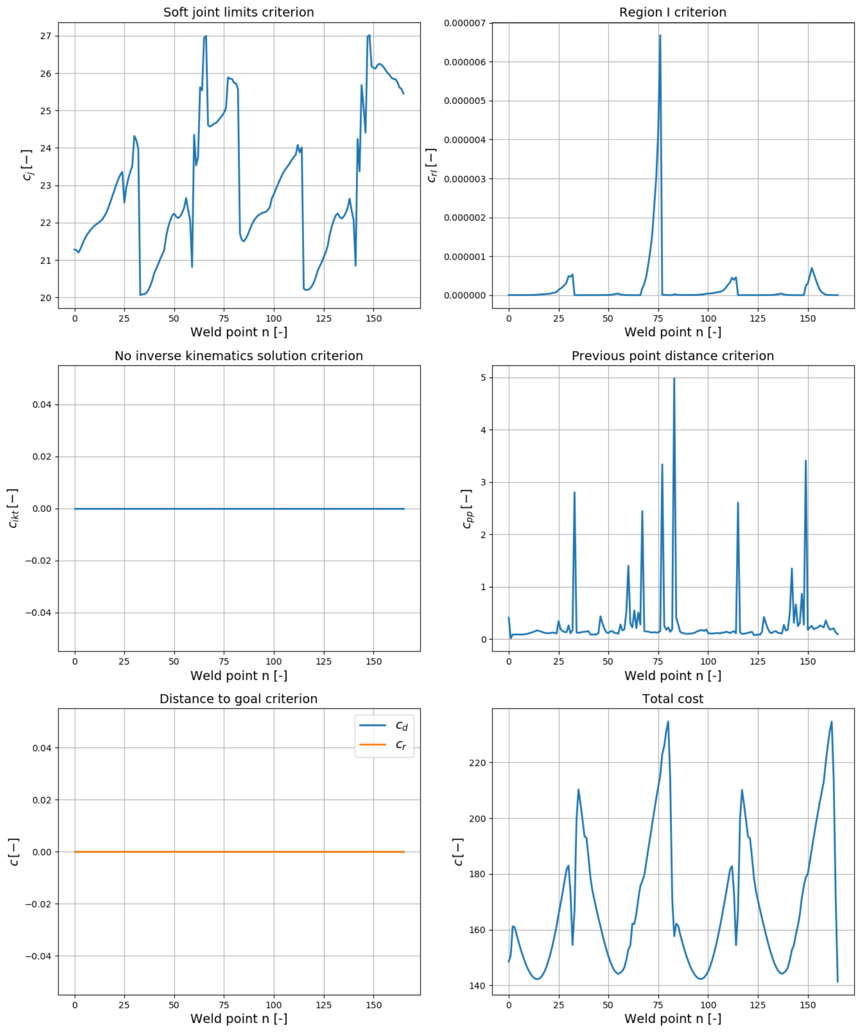

Figure 6.3: Results of an optimization (velocity through Jacobi matrix) (continued)

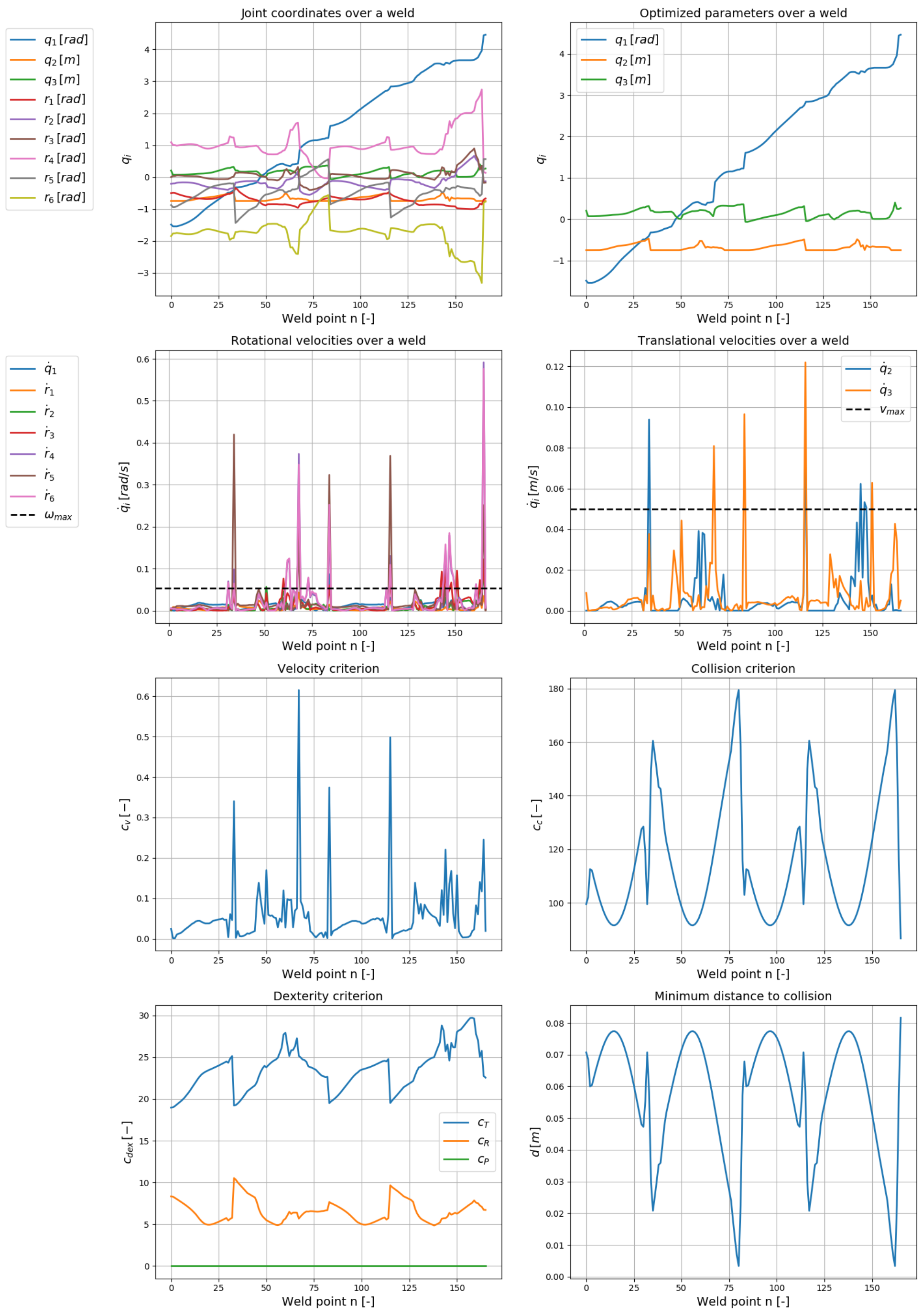

Figure 6.4: Results of an optimization (velocity through time and distance)

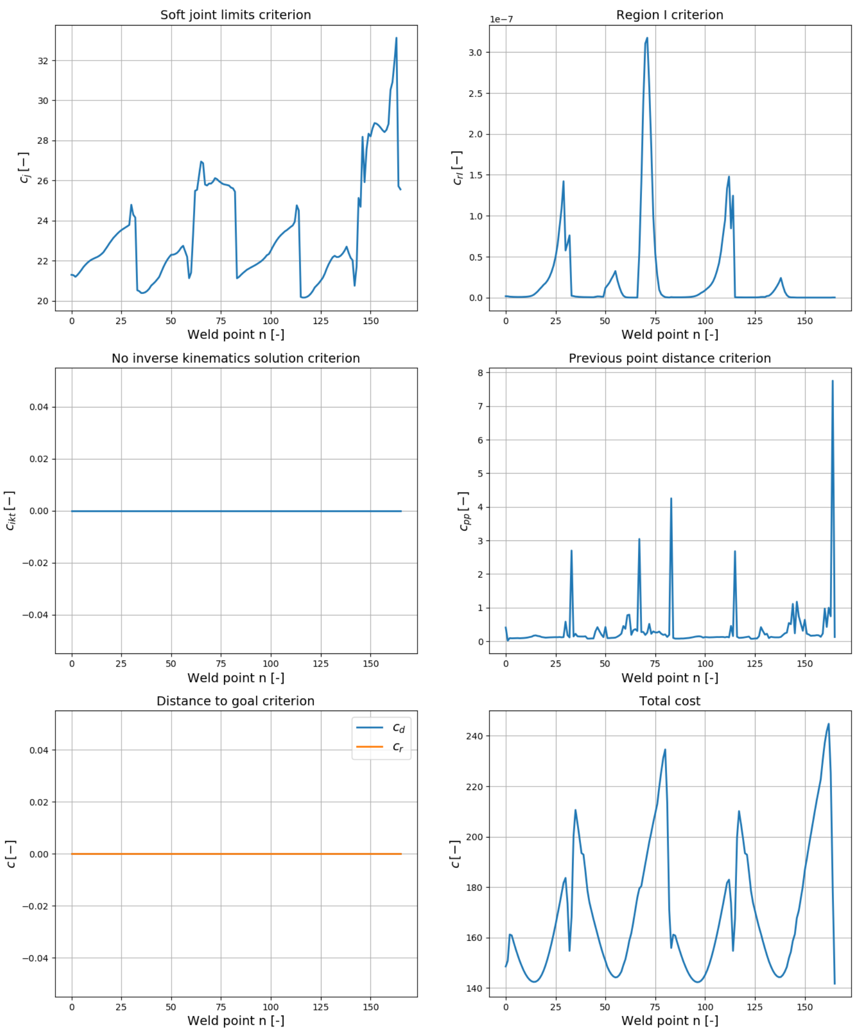

Figure 6.5: Results of an optimization (velocity through time and distance) (continued)

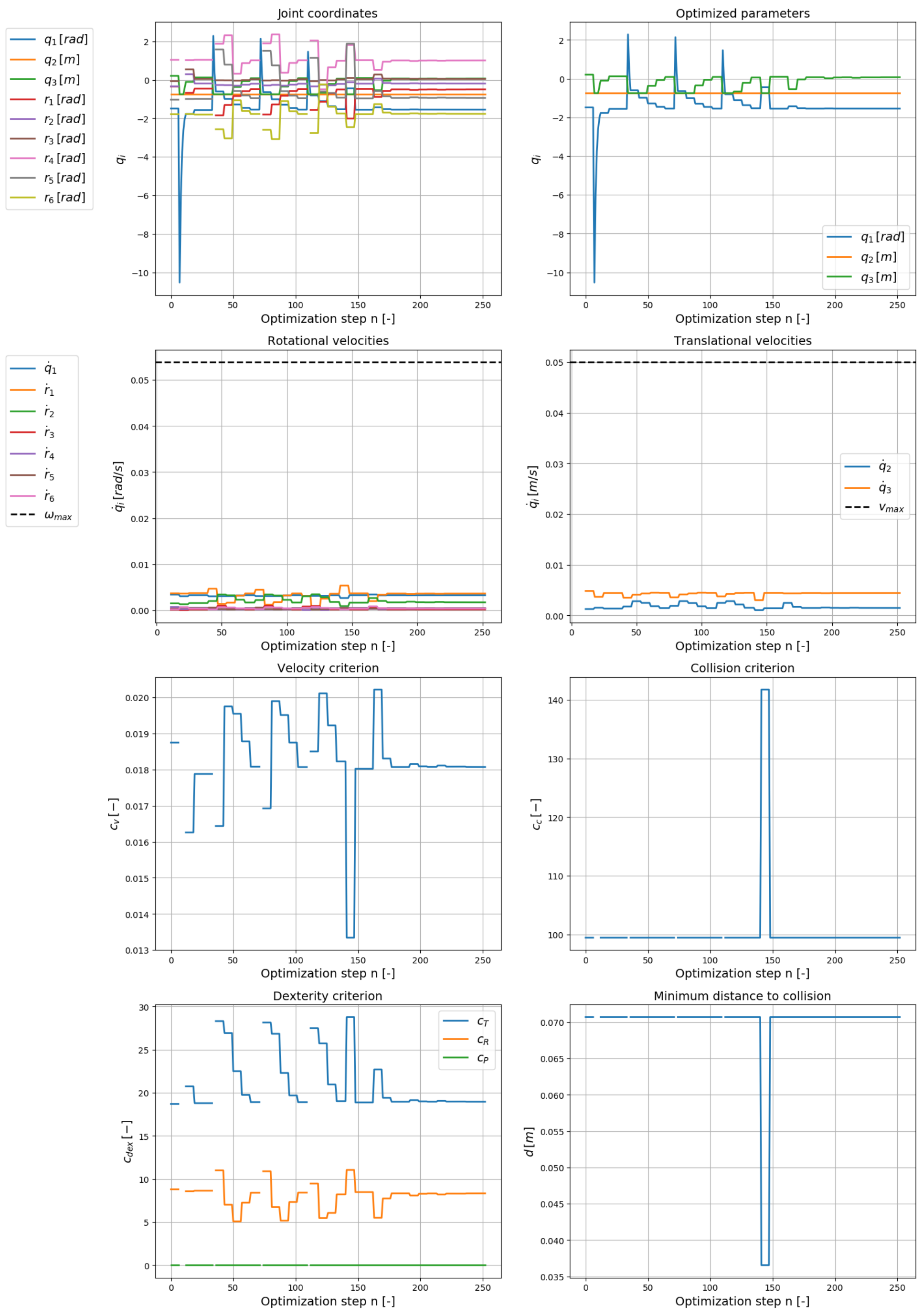

Figure 6.6: Optimization of the first point (velocity through Jacobi matrix)

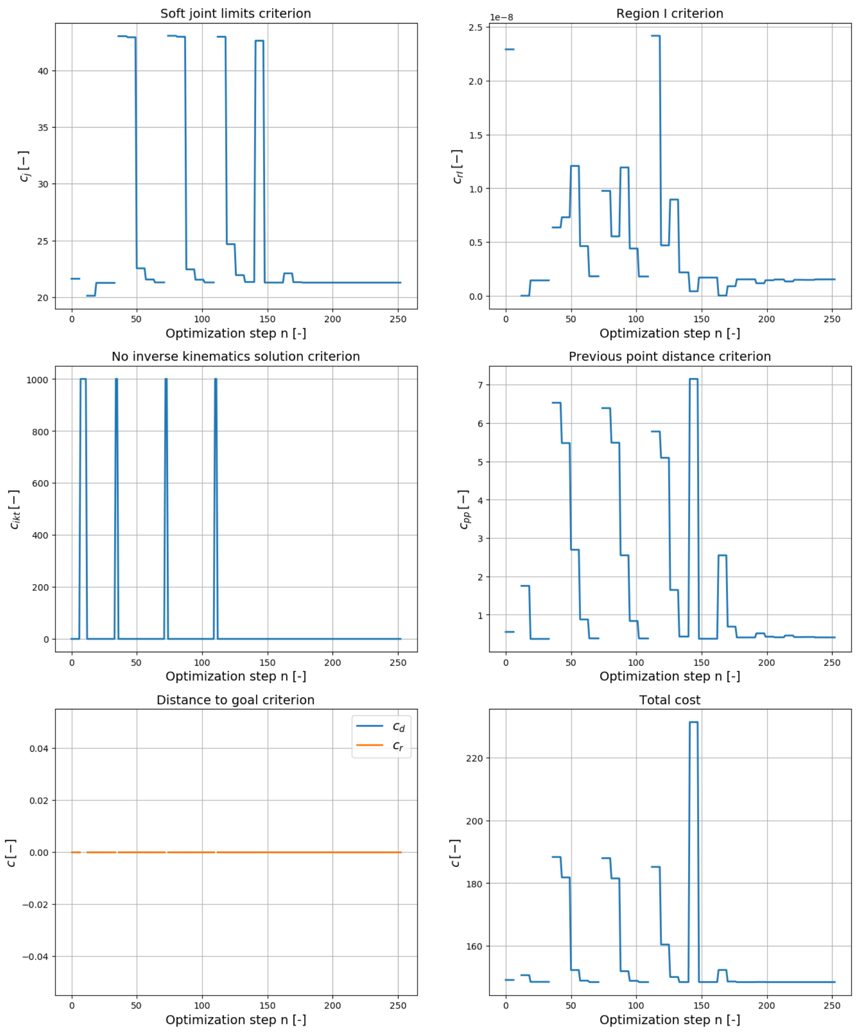

Figure 6.7: Optimization of the first point (velocity through Jacobi matrix) (continued)

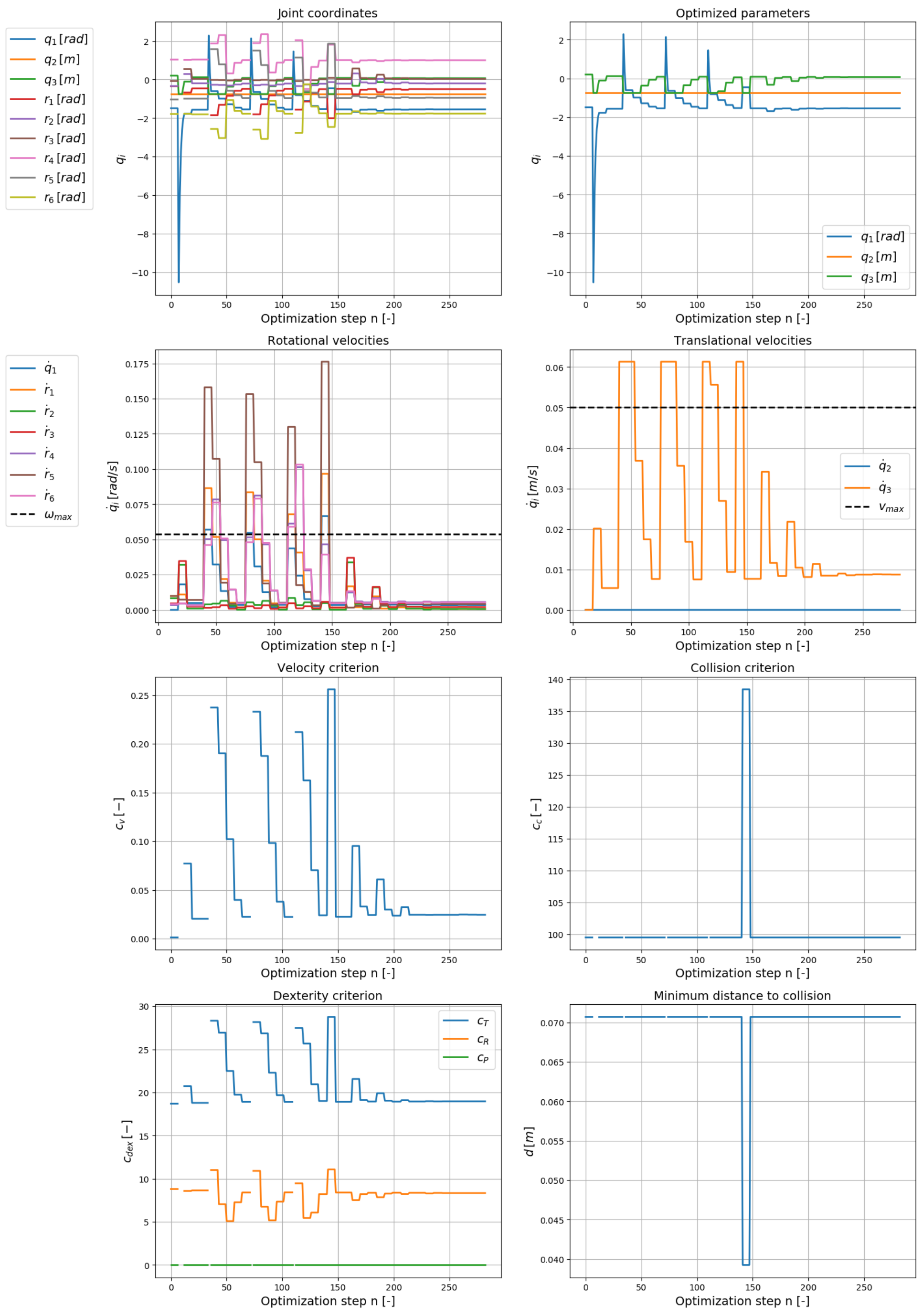

Figure 6.8: Optimization of the first point (velocity through time and distance)

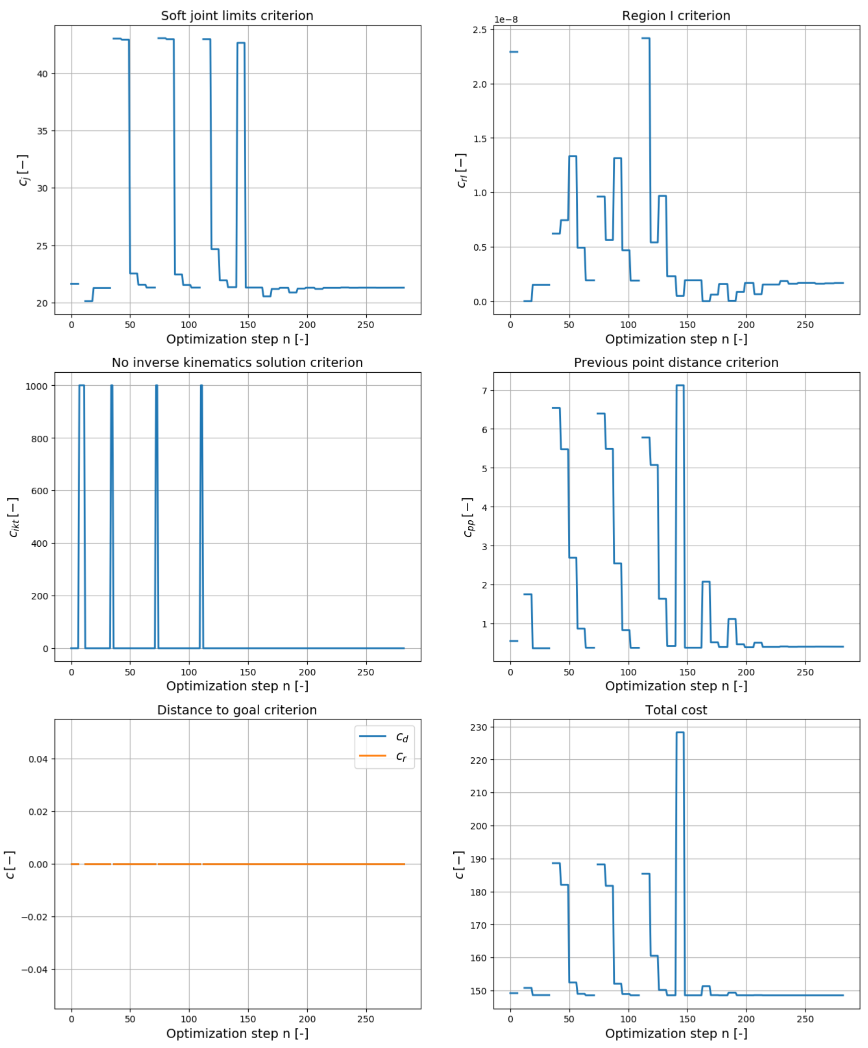

Figure 6.9: Optimization of the first point(velocity through time and distance) (continued)

Chapter 7

Conclusion and future outlook

7.1 Summary of the work

This thesis has concerned itself with the development of a planning software for a 9 DOF robot for welding of plastic tanks. Currently, such tanks are welded mostly by hand, which is an long and arduous process requiring multiple years of training. The proposed robot cell aims to improve this situation and make automatized welding of a subset of tank shapes possible.

The robot cell consists of three external axes and an industrial robot. The tank is mounted on a rotating table and an extruder is attached to the robot's mounting flange.

The robot movement is based on welding trajectories, which will be prescribed by a design/manufacturing engineer. They form the input for the planning software. The software solves the inverse kinematics problem to find the joint coordinates required to reach the desired *pose*. Due to redundancy in the DOFs, infinite solutions exist for the IKT problem. We're employing optimization algorithms to find an unique solution, based on a set of predefined optimization criteria. These criteria are chosen in a such way as to find a robot configuration that suits the welding requirements the best.

In Chapter 1 we've introduced the problem and presented the motivation behind it. Chapter 2 focused on the theoretical foundations of motion planning and optimization used in this thesis. The hardware and software of the robot cell was described in detail in Chapter 3. We've also talked briefly about Robot Operating System, where the planning software was developed.

Core of the work lies in Chapters 4 and 5. We've presented the optimization criteria, the reasonings behind them as well as a mathematical formulation for them. Here, we've recognized the importance of collision avoidance and high dexterity on a successful planning result.

Because the G-Code used for the definition of the weld points lacked one of the spatial rotations, we had to devise a way, how to calculate this sixth missing rotation. This, along with improvements to the inverse kinematics solutions are presented in Chapter 4 as well.

The real implementation in ROS, written in C++ was explained in Chapter 5. We focused on describing all of the configuration parameters that can be adjusted in the software. We also provided a brief explanation of all of the Classes, Methods and Functions written, giving an introspect in how the planning software works.

Lastly, in Chapter 6 we've presented results of the optimization on a outside bottom weld of a rectangular tank, while giving an explanation of the output graphs.

Taking a look at the goals set at the beginning in Section 1.2, we believe that we fulfilled all of them, but we recognize the issues and deficiencies found in the work.

7.2 Critical evaluation

While the planner has been able to plan a welding path that's continuous, there are some glaring issues, that show, that the planner is not yet ready for testing on a real machine.

Firstly, the joint coordinates aren't smooth and sudden changes (or changes in configuration) occur, which the real robot wouldn't be able to handle. Also the occasional stuttering is not welcome, especially for an application such as welding, where smooth movements with small accelerations are desired. This walks in hand with the joint velocities overstepping the velocity limits. If such situation occurs, the robot simply won't be able to perform the move with all of the joints in the required time. This might negatively affect the positioning of the extruder and thus the weld quality.

Further issues have to do with collision checking, where the robot gets very close to collision, before being forced to react and change configuration. This is not only dangerous in real applications, where imperfections due to mounting, tolerances and additional equipment (cables, hoses) affect the real collision distance, but also, again, due to the stuttering and backward rotation of the table it produces.

Lastly, this is noticeable only on the video, but the optimizer tends to prefer moving the external axes as little as possible. This then has to be compensated by the robot, who has to stretch to reach the weld point. Such stretching negatively affects the dexterity of the robot, as it's near singular configuration. In the moment that the robot can't reach the point anymore, the optimizer is forced to take a big step with the external axes, which leads to spikes in velocity. We'd rather prefer the table to move more each step, to keep the robot arm close, in an ideal configuration with high dexterity.

There's certainly the possibility that an ideal set of criteria weights can improve the results greatly. Unfortunately, the optimizer seems to be very sensitive to some of them. Velocity criterion, Previous point distance criterion and Region I criterion are those, who are the most problematic. Even a small change in the weights often leads to a completely failed planning.

The planning of the tested weld takes about 3 minutes on an average laptop. Considering the welding itself can take tens of minutes, if not a few hours, this time is negligible. But if the user has to re-run the planner multiple times, the time quickly adds up. Especially, if various settings and weights want to be tested, the whole process (with the necessary start-ups and shutdowns of the executables) gets quite time-consuming.

Overall, we believe that we have improved the planner a great deal and it shows promising results, but it's still far from a fully usable version. We'd be more pleased with the results, if we were able to implement more features and improve remaining issues.

7.3 Future improvements

The project has come a long way since it's inception, but there's still a lot to do. Here we want to mention unresolved issues and suggest future improvements, that will, hopefully, lead to a better performance of the planner:

Working weld approach planning The weld approach planning doesn't have to fulfill as strict requirements as the weld move planner, but it's still a vital part of the planning. The most important function of the weld approach planning is to plan a path that will lead to a good starting configuration, from where the welding can begin. This configuration has to already fulfill all of the criteria placed on the weld move.

More detailed G-Code A lot of the issues in current results arise due to the insufficient description of movement in corners. They are described only as a single point, where an abrupt 90° change of direction happens. Ing. Malý from *Alad* has been developing an improved G-Code generator specifically for this project. The new G-Code should improve the path planning by a great margin and also unlock the possibility of testing on different welds and tanks.

Multi-step optimization Currently, we optimize the parameters for each point in isolation, with only relationship to the previous point through the Previous point distance criterion. Ideally, we'd include parameters for multiple steps in the future and optimize them together. That way the optimizer can „see the future" and better react to sudden changes. E.g. if the optimizer knows that a corner is approaching, it can start to turn the table in advance. Currently, the optimizer has only one step to react to a change, which also leads to sharp changes in joint coordinates and spikes in velocities. The optimizer could also better predict an upcoming collision and adjust the robot position in advance.

Path verification With the current setup, it can't be guaranteed that no changes between robot configurations will occur. They might happen because one of them gives a better cost value, or because one of the joints has reached it's limit and can't move further. Both situations are unacceptable on a real robot. It's therefore necessary to verify the path after planning and make sure that no such changes in joint coordinates occur. If that's the case, a different starting configuration can be used and the path has to be planned again.

Bibliography

[1] catkin. https://wiki.ros.org/catkin, 2017. Accessed on July 31, 2023.

[2] ROS introduction. https://wiki.ros.org/ROS/Introduction, 2018. Accessed on July 23, 2023.

[3] tf2 package summary. https://wiki.ros.org/tf2, 2019. Accessed on July 24, 2023.

[4] STP Plast s.r.o. http://stpplast.cz, 2020. Accessed on July 12, 2023.

[5] TAČR: Automatické svařování různorodých plastových nádrží. https://old.starfos.tacr.cz/cs/project/FW02020095, 2020. Accessed on July 12, 2023.

[6] ROS. https://www.ros.org/, 2021. Accessed on July 23, 2023.

[7] Using urdf with robot state publisher. https://wiki.ros.org/urdf/Tutorials/Using%20urdf%20with%20robot_state_publisher, 2021. Accessed on July 24, 2023.

[8] ROS concepts. https://wiki.ros.org/ROS/Concepts, 2022. Accessed on July 23, 2023.

[9] tf2 - quaternion basics. http://wiki.ros.org/tf2/Tutorials/Quaternions, 2022. Accessed on July 24, 2023.

[10] Why ROS 2? http://design.ros2.org/articles/why_ros2.html, 2022. Accessed on July 24, 2023.

[11] G-code. https://en.wikipedia.org/wiki/G-code, 2023. Accessed on July 24, 2023.

[12] G-Code Index. https://marlinfw.org/meta/gcode/, 2023. Accessed on July 24, 2023.

[13] GitHub: ros-industrial/motoman. https://github.com/ros-industrial/motoman/, 2023. Accessed on July 24, 2023.

[14] LEISTER. https://www.leister.com/en, 2023. Accessed on July 19, 2023.

[15] LEISTER weldplast 610-i. https://www.leister.com/en/product/Weldplast-610-i/172-580, 2023. Accessed on July 19, 2023.

[16] Logistic function. https://en.wikipedia.org/wiki/Logistic_function, 2023. Accessed on August 2, 2023.

[17] MoveIt! Tutorials. https://ros-planning.github.io/moveit_tutorials/index.html, 2023. Accessed on July 24, 2023.

[18] regular expressions 101. https://regex101.com/, 2023. Accessed on August 2, 2023.

[19] Spherical linear interpolation (slerp). https://splines.readthedocs.io/en/latest/rotation/slerp.html, 2023. Accessed on August 2, 2023.

[20] urdf. https://wiki.ros.org/urdf, 2023. Accessed on July 24, 2023.

[21] URDF and SRDF. https://ros-planning.github.io/moveit_tutorials/doc/urdf_srdf/urdf_srdf_tutorial.html, 2023. Accessed on July 24, 2023.

[22] YASKAWA motoman GP88. https://www.yaskawa.eu.com/products/robots/handling-mounting/productdetail/product/gp88_702, 2023. Accessed on July 19, 2023.

[23] Sameer Agarwal, Keir Mierle, and The Ceres Solver Team. Ceres Solver, 3 2022.

[24] David Coleman, Ioan A. Șucan, Sachin Chitta, and Nikolaus Correll. Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study. *Journal of Software Engineering for Robotics*, 5(1):3–16, May 2014.

[25] Kamil Horný. Automatic Trajectory Planning for Robot Welding. Master's thesis, Czech Technical University, Faculty of Electrical Engineering, Department of Control Engineering, 2021.

[26] Sertac Karaman and Emilio Frazzoli. Sampling-based Algorithms for Optimal M]otion Planning, journal = The International Journal of Robotics Research, volume = 30, year = 2011, number = 4, pages = 846–894.

[27] J.J. Kuffner and S.M. LaValle. RRT-Connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 2, pages 995–1001 vol.2, 2000.

[28] Kenneth Levenberg. A method for the solution of certain nonlinear problems in least squares. *Quarterly of Applied Mathematics*, 2(2):164–168, 1944.

[29] Kevin M. Lynch and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control.* Cambridge University Press, 2017.

[30] Donald W. Marquardt. An algorithm for least squares estimation of nonlinear parameters. *Journal of Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

[31] Vladimír Stejskal and Michael Valášek. *Kinematics and Dynamics of Machinery.* Marcel Dekker Inc., New York, 1996.

[32] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, December 2012. `https://ompl.kavrakilab.org`.

[33] Matěj Vetchý. Kinematic Calibration and Motion Optimization of Industrial Manipulator. Master's thesis, Czech Technical University, Faculty of Electrical Engineering, Department of Cybernetics, 2023.

[34] Jan Zavřel, Martin Jílek, Zbyněk Šika, and Petr Beneš. Dexterity Optimization for Tensegrity Structures Using Local Linear Model Trees. In *2021 9th International Conference on Control, Mechatronics and Automation (ICCMA)*, pages 46–49, 2021.