TUHH
Hamburg
University of
Technology

ICS
Institute
of Control
Systems

Research Project

Evaluating Model Predictive Control for Step Planning on Humanoid Robots in the RoboCup Standard Platform League

Rasmus Mecklenburg

18 January 2025

Supervisors: Jan Kaiser, M.Sc.
Christian Hespe, M.Sc.

Examiner: Prof. Dr-Ing. Annika Eichler

Hereby I declare that I produced the present work myself only with the help of the indicated aids and sources.

Hamburg, 18 January 2025 Rasmus Mecklenburg

Abstract

This project evaluates the use of model predictive control (MPC) for step planning on the NAO robotics platform in the context of robot soccer in the RoboCup Standard Platform League (SPL).

To this end, the step planning problem is formalized and formulated as an MPC problem. This formulation is then implemented in a Rust program for eventual adoption by team HULKs, the SPL team of the TUHH.

This thesis covers the details of the problem formulation, as well as insights gained from the implementation in Rust.

Contents

List of Figures

1 Introduction

Walking is an essential functionality of any humanoid robot. Designing and implementing a bipedal walking routine is a complex, multi-faceted task: An implementation must enable smooth, omnidirectional locomotion of the robot while ensuring dynamic stability and operating within the kinematic constraints of the hardware.

The RoboCup is an annual international robotics research competition, made up of various different leagues. In the RoboCup Standard Platform League (SPL), one of the robot soccer leagues, teams compete with identical hardware. The NAO, the robot used by the teams of the SPL, is a small humanoid robot with relatively low computing power. This work is written in the context of team HULKs, the SPL team of the TUHH.

A crucial part of the walking routine of team HULKs is the *step planner*, which determines where the robot will place its feet. This project evaluates Model Predictive Control (MPC), a versatile scheme for optimal control, for use in the step planner of the soccer robots of team HULKs. MPC is a promising choice for this type of application, as it is able to generate steps with consideration for their impact on the state of the system in the future. The goal is to determine whether MPC is a feasible control scheme for this problem, and to lay the groundwork for eventual adoption of such a step planner in the code of team HULKs.

This work is structured as follows: First, Chapter 2 covers a number of preliminary definitions used throughout the thesis. It also provides further context to the problem by introducing the RoboCup SPL, the NAO, the walking pipeline used by team HULKs, and MPC in greater detail. In Chapter 3, the problem solved by the step planner is formalized. Next, Chapter 4 covers the details of the implementation. Finally, Chapter 5 contains an evaluation of the achieved results, followed by a conclusion in Chapter 6 and a brief outlook in Chapter 7.

2 Preliminaries

This chapter defines a number of key items used throughout this work, starting with an overview of the RoboCup SPL and the robot used in it, followed by a brief introduction into steps, the walking pipeline, the existing step planner, and into MPC.

2.1 RoboCup SPL

The RoboCup [1] is an annual international robotics competition. It is composed of multiple different leagues, each with its own rules and research focus. Many of the leagues are about robot soccer - but there are also leagues about robots for search and rescue missions, and for industrial or residential use. The league considered in this work is the Standard Platform League (SPL), one of five robot soccer leagues of the RoboCup. In the SPL, teams are required to use the same robot platform, which currently is the NAOv6 by Aldebaran [2], a small humanoid robot. Teams are also not permitted to modify the robot hardware. Because of this restriction, research in the SPL is focused on algorithm design and efficient implementations of them.

2.2 NAO

The NAO is a small humanoid robot, designed and manufactured by Aldebaran. It is 58 cm tall, has 25 degrees of freedom, and is equipped with two cameras, four microphones, two speakers, and two pairs of sonar sensors [3]. NAOv6, the latest version and the one used by most teams in the SPL, is powered by an Intel Atom E3845 processor running at 1.91 GHz [4]. Many vital components are housed in the head of the robot, such as the central processing unit (CPU), the cameras, microphones, and speakers. The chest of a NAO robot contains, among other components, a six-axis inertial measurement unit (IMU), the sonar sensors, and the *chestboard*. The chestboard is a microcontroller-powered circuit board, responsible for issuing commands to the motors and reading from the joint position encoders among other tasks [5]. The main CPU in the head communicates with the chestboard over a USB connection running through the neck of the robot. This link has limited bandwidth, which is one reason for the low motor and sensor update rate of the NAO at 83 Hz.

Figure 1: The NAO robot. [6]

The NAO is primarily intended for educational and research use. Compared with purpose-built soccer robots used in other leagues, its capabilities are relatively limited: The processor is slower than those of comparable robots, and although its processor has a small integrated graphics processing unit (GPU), the lack of a dedicated GPU makes inference of large machine learning models and other compute-heavy techniques on the NAO prohibitively slow.

As a consequence of these restrictions, teams in the SPL make heavy use of computationally efficient, hand-crafted algorithms, as opposed to techniques like end-to-end machine learning such as the method shown by Tirumala et al. [7], which are computationally more complex.

2.3 Anatomy of a Step

For robot soccer in the SPL, the world can be reduced to a two-dimensional model for the purpose of step planning, by disregarding the up-down axis of the real world. In the following, we adopt a two-dimensional perspective of the soccer field. Furthermore, we define the following terms:

- *Position*: a point in the two-dimensional soccer field.
- *Orientation*: an orientation in this two-dimensional field; an angle.

- *Pose*: the combination of a position and an orientation.
- *Step*: the transformation from one pose to another, composed of a translation and a rotation.
- *Step Plan*: a sequence of steps. This is the output of the step planner.

During each step, the foot that currently has ground contact is referred to as the *support foot*. The other foot, which is called the *swing foot*, travels along a three-dimensional curve to its next resting position. After the swing foot has gained ground contact, the roles of the feet are switched: The foot which just gained ground contact becomes the new support foot, and the previous support foot is lifted off of the ground to perform the next step as the new swing foot. The step planner considers the position of the support foot as static. In practice, this is not completely true due to slippage.

The state of the step planning system at time step k is descibed by three variables $p_{x,k}$, $p_{y,k}$, and θ_k: $p_{x,k}$ and $p_{y,k}$ describe the two-dimensional position of the robot, and θ_k describes its orientation. This pose does not describe the position and rotation of the feet directly, but rather the position and rotation of the logical position of the robot. This logical position is at a fixed location relative to the current support foot, located half-way between the two soles if the feet of the robot are at their resting position, and this position is kept constant relative to the current support foot. This is visualized in Figure 2.

A step describes the difference between two of these states, it is the combination of a two-dimensional translation and rotation. This kind of transformation is also called a *direct isometry* or *rigid motion*. It comprises three components: f, l, and α. f and l respectively describe the forward and leftward components of the translation, and α is interpreted as an additional rotation around the position reached by the step.

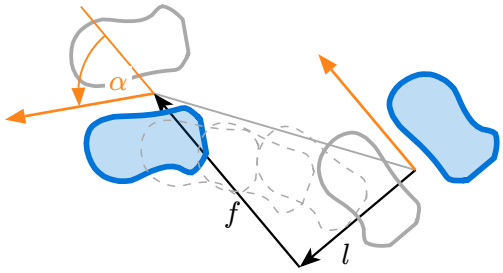

Figure 2: The anatomy of a step. At each step, the current support foot is highlighted in blue. The current swing foot is drawn in light gray, at the position where it would be if the robot were in its resting position there. Note that during normal walking, the swing foot does not actually come to rest at this position, as it travels from its previous position on the ground as the support foot to the its next resting position reached after two steps. The size of the step relative to the dimensions of the robot feet is not drawn to scale.

The definition of a step yields the following state update rule, which defines the state variables after taking one step:

$$
\begin{aligned}
p_{x,k} &= p_{x,k-1} + f_k \cos(\theta_{k-1}) - l_k \sin(\theta_{k-1}) \\
p_{y,k} &= p_{y,k-1} + f_k \sin(\theta_{k-1}) + l_k \cos(\theta_{k-1}) \\
\theta_k &= \theta_{k-1} + \alpha_k
\end{aligned} \tag{1}
$$

2.4 Walking Pipeline

This section describes the walking process currently in use by team HULKs. Partial documentation for this process can be found on the HULKs homepage [8]. In this implementation, the walking problem is broken down into multiple layers, where each layer solves a simpler sub-problem. This is referred to as the walking stack. Each layer of this stack receives input data from the layer above it, and computes the inputs needed by the next layer down the stack.

The walking stack of team HULKs consists mainly of the following layers, from top to bottom:

1. *Target Selection*: The behavior code of the robot determines a target position for it to move to.
2. *Path Planning*: Based on the perceived state of its environment, the robot plans a path to its target.
3. *Step Planning*: A sequence of steps is determined which will take the robot along the planned path.
4. *Walking Engine*: The position of the swing foot in three dimensions is interpolated along a curve from one step position to the next. The shape of this curve determines how the foot will be lifted off of the ground during the step. This step may choose to change the destination of the swing foot from the one requested by the step plan, such as to prevent a backward fall by quickly placing a foot behind the robot.
5. *Inverse Kinematics*: This layer computes the joint angles needed to move the feet of the robot to the requested positions.

In the first step, the robot uses its cameras, odometry, prior knowledge about its environment, and messages received over the network to construct a *world model*. The world model contains information about the current game state, and position estimations of items like the robot itself, other robots, the ball, and potential obstacles.

Robot soccer in the SPL is played by teams of up to seven robots. In order to enable coordinated behavior, each robot on the field assigns itself a *role* which governs its actions. The role is chosen from a small set of pre-defined roles, using information from the world model. Together with the world model, the role is then used to decide what action to perform: A robot with the *keeper* role might decide to defend the goal, while a *striker* robot is programmed to reach the ball and then kick it towards the goal of the opposing team. Most of the actions produced in this step contain a target pose for the robot to walk towards: A keeper robot might target a position just in front of the own goal line, while the target of a striker is placed in front the estimated position of the ball.

In the next step, the path planner uses the current estimated position of the robot, and the positions of any obstacles in the field from the world model, together with the target position produced earlier to determine a viable path connecting the current estimated position of the robot and the target pose, while avoiding any obstacles.

Finally, the *step planner* uses this path to produce a *step plan*, a sequence of steps, describing where the robot should place its feet in order to reach the target pose. This process is illustrated in Figure 3.

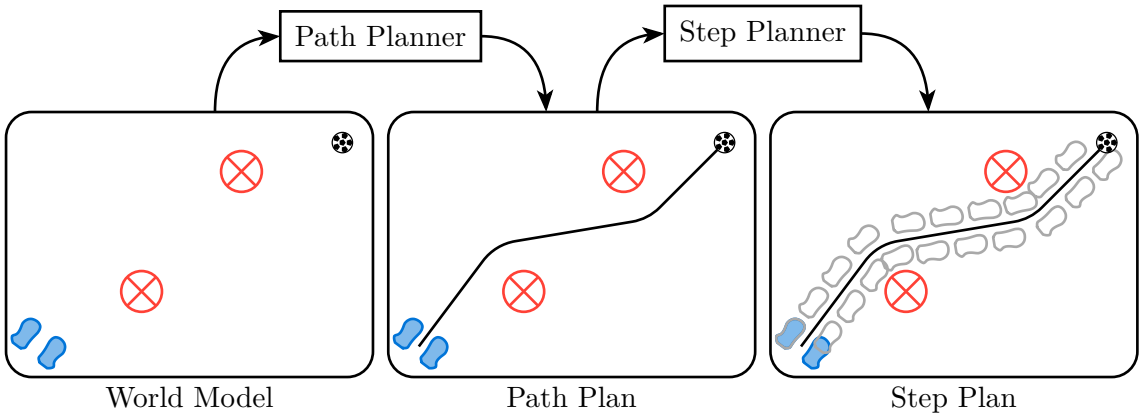

Figure 3: The step planning process. The initial pose of the robot in its world model is represented by the blue pair of robot feet, and the red circles represent obstacles. The target of the path is indicated by the soccer ball in the top right.

The step plan is then executed by the walking engine, which generates three-dimensional trajectories for the feet to follow from one pose in the sequence to the next. Joint angles are then determined through inverse kinematics, sent to the chestboard, and used to generate motor commands.

After having taken a step, the entire step planning process is iteratively repeated with an updated world model.

2.5 Path Plan

The path plan followed by the step planner in this project consists of a sequence of line segments and circle segments (arcs). This definition of the step plan follows from the fact that all detected obstacles are either modeled as a circle or a half-plane in the world model. The path plan is generated by the path planner using the A* pathfinding algorithm [9], based on the current game state and detected ball and obstacle positions.

2.6 Step Planning

Step planning fills a crucial role in the walking stack of team HULKs. Improvements to the step planner may considerably improve the overall quality of the produced walking motion, and MPC is a promising control scheme for such a problem. One major benefit of MPC is

the ability to plan control inputs with consideration for their impact on the state of the system in the future.

We define basic requirements for the steps generated by a step planner as follows:

1. The steps must efficiently take the robot to its target pose.
2. The steps must closely follow the path plan used as an input. Otherwise, the robot risks colliding with an obstacle.
3. The steps must respect the kinematic constraints of the robot, and only contain physically feasible steps.

The step planner currently in use by team HULKs works roughly as follows:

1. First, the step planner determines a suitable target point for the next step to aim for. This point is chosen as the closest end point of a path segment outside of some minimum radius around the current position.
2. Next, it determines a target *orientation* by finding the forward direction of the path at the target position. Some motions require the robot to have a certain orientation while walking, in which case this target orientation is overridden.
3. A target pose is then constructed from the target position and rotation. Then, the step planner finds the step which will move the current pose of the robot onto the target pose.
4. Finally, the step is constrained to the set of physically feasible steps.

This process is visualized in Figure 4.

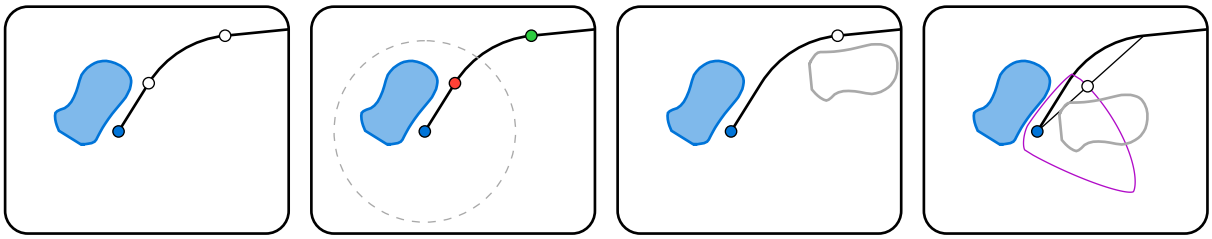

Figure 4: The basic functionality of the current step planner. In this example, the left foot of the robot is the support foot, and the step planner needs to find the next step for the right foot to execute. The blue dot indicates the current logical position of the robot, and the other dots on the line show where the path segments are joined. The purple outline in the last image shows the set of legal step sizes for this fixed turn angle.

This approach fulfills requirements 2 and 3 well, but it leaves some room for improvement with regard for requirement 1:

- The step planner struggles with walking diagonally. This is due to the fact that the NAO cannot execute two consecutive sidesteps in the same direction, as this would require crossing of the legs. This kind of movement is not possible due to the kinematics of the NAO. Thus, when requested to walk with an angle between the walking direction and the facing direction, every other step has its sideward component clamped to almost zero. The result is a walking trajectory with effectively only half the angle between the facing direction and the direction of travel, leading to the robot deviating from its planned path.
- It arrives on the wrong foot. There are some gains to be made in robot soccer by planning steps such that the robot arrives at its destination with a specific foot as the support

foot, chiefly because behavior code decided to perform a *turn kick* upon arrival, a kind of kick which requires the robot to swing with a specific foot.
- It generates suboptimal steps. It is well known that the NAO can perform larger steps to the side than forwards, with the caveat that only every other side step can be large. Therefore, taking purely forward steps, even to walk in a straight line, is probably not optimal. It is an open question how exactly optimal steps would look like, but it may involve a combination of steps with both a forward and sideward component, as well as strategically-placed rotations.

A model predictive step planner may solve these issues.

2.7 Model Predictive Control

Model Predictive Control (MPC) is a versatile scheme for optimal control. Originally developed for the control of chemical plants [10], it is now widely used in many engineering disciplines, such as robotics and self-driving vehicles [11]. The problem formulated in this work specifically is a *discrete-time nonlinear* MPC problem. The notation used follows the book by Rawlings et al. [12].

The main idea of MPC is the use of a model of the *plant*, the system to be controlled, to predict the impact of control inputs in the present on the state of the plant in the future, and then to react accordingly. This model is a function $f(x, u)$, which maps the state of the system x and a control input u to the state of the system after the control input is performed.

The second key component of an MPC formulation is the *cost function* $\mathcal{L}(\boldsymbol{u})$, which decides how "good" a solution is. It assigns each solution a real-valued score, where lower scores are better. An optimal solution is one that minimizes this cost function. At each time step, a model predictive controller finds the optimal control inputs $\boldsymbol{u}$ for the next N time steps.

$$\boldsymbol{u} = (u_1, u_2, ..., u_N) \tag{2}$$

The first element of this sequence, u_1, is then used as the control input at the current time step, and the process is repeated. The rest of the computed control inputs can be used as an initial guess for the next iteration to accelerate the solver convergence. This sliding window is called a *receding horizon*. The size of this horizon determines how far into the future the solver can plan. Using a larger horizon thus yields better results, but is more expensive to compute.

The following is the classical formulation of a discrete-time nonlinear MPC problem, adapted from [12, Eq. 8.1]:

$$\begin{aligned} \underset{\boldsymbol{x},\boldsymbol{u}}{\text{minimize}} \quad & \sum_{k=0}^{N-1} \ell(x_k, u_k) + V_f(x_N) \\ \text{subject to} \quad & x_0 = \hat{x}_0 \\ & x_{k+1} = f(x_k, u_k), \qquad k \in (0, ..., N-1) \end{aligned} \tag{3}$$

The function to be minimized is defined as the sum over N evaluations of the stage cost function $\ell(x_k, u_k)$, which rates the combination of a single system state x_k and a control input u_k at that time. The system states are computed using the model $f(x, u)$, using the initial state of the system is given by $\hat{x}_0$.

For this project, this formutation can be simplified. On the one hand, this formulation includes x_0 in the calculation of the cost, which the optimizer cannot influence. On the other hand, this formulation rates the state at the end of the prediction horizon separately with a different value function $V_f(x_N)$. While this formulation is more expressive, this distinction is not needed for the problem formulation of this project. We therefore perform an index shift, such that the first stage cost no longer operates on x_0, and the final stage cost includes x_N. The modified MPC formulation used in this project is defined like follows:

$$\begin{aligned} \underset{\boldsymbol{u}}{\text{minimize}} \quad & \mathcal{L}(\boldsymbol{u}) := \sum_{k=1}^{N} \ell(x_k, u_k) \\ \text{subject to} \quad & x_0 = \hat{x}_0 \\ & x_k = f(x_{k-1}, u_k), \quad k \in (1, ..., N) \end{aligned} \tag{4}$$

First, this formulation no longer optimizes over $\boldsymbol{x}$, since the system states are uniquely determined by $\hat{x}_0$ and $\boldsymbol{u}$. Next, we remove $V_f(x_N)$ from the cost function, and redefine u_k to have a slightly different meaning: The control input u_k is now the input which brings the system from state x_{k-1} to state x_k. This reformulation greatly simplifies the definition of the stage cost function later.

3 Problem Formulation

This chapter formalizes the problem solved by the step planner. We define the step planning problem as follows: Given an initial pose and a path plan, the step planner shall generate a fixed number of steps that, when executed, fulfill the following criteria:

- The steps should move the robot forwards along the path.
- The steps should follow the path plan closely.
- The steps should be physically feasible for the NAO hardware.

Other criteria are also possible, and might enhance the quality of the generated steps. To limit the scope of this project, we consider exclusively the three aforementioned criteria. The following sections explain the parts of this definition in greater detail.

3.1 Walk Volume

Naturally, steps cannot be arbitrarily large or contain an arbitrary rotation. In addition to the kinematic limits defined by the NAO hardware, there are joint angle limits, illegal joint angle combinations due to self-intersections, and robot poses that are undesirable due to instability.

The maximum step length for pure forwards or pure sidewards steps can be determined with relative ease, as well as the maximum angle by which the robot can turn in one step. Greater complexity arises when trying to define the limits of combinations of these three axes.

A *walk volume* $\mathcal{W}$ is the set of legal steps, defined as a shape in (f, l, α)-space. Due to the dependence on the kinematics of the entire robot, the exact shape of the NAO's walk volume is more difficult to determine exactly. In addition, these limits also depend directly on the environment and the rest of the software used on the robot, as the surface friction, and different walk engines, different movement of the upper body during walking, or different dynamic balancing routines all affect what step sizes are considered stable. Thus, determining exact limits to stable step size for the NAO in general, independent of the software and the environment, is not feasible.

For this project and in practice in the SPL, approximations of the walk volume are used to bound the step size. The simplest of these is a cuboid: Such a walk volume simply limits each of f, s, and α to some interval. This walk volume is simple to implement, but not sufficiently expressive: For example, while a large side step with $l = 10$ cm or a rotation with $\alpha = 0.9$ (radians) are perfectly legal steps on their own, combining them is not possible due to the kinematics of the NAO.

Many teams in the SPL use [13], [14] or have used [15] the walk volume designed by team rUNSWift from the UNSW Sidney as part of their 2014 walk generator [16]. This walk volume $\mathcal{W}_{\text{rUNSWift}}$ is defined piecewise in each octant of (f, l, α)-space by an exponential term.

The maximum step length in f, l, and α depends on the sign of the component. For example, steps can go further forwards than backwards, and sideward steps can only be large if they move away from the support foot. Thus, to calculate $\mathcal{W}_{\text{rUNSWift}}$, we first normalize the components of the step, such that allowed values are mapped to the interval $[0, 1]$:

$$\tilde{f} := \begin{cases} \frac{f}{f_{\min}} \text{ if } f < 0 \\ \frac{f}{f_{\max}} \text{ otherwise} \end{cases} \qquad \tilde{l} := \begin{cases} \frac{l}{l_{\min}} \text{ if } l < 0 \\ \frac{l}{l_{\max}} \text{ otherwise} \end{cases} \qquad \tilde{\alpha} := \begin{cases} \frac{\alpha}{\alpha_{\min}} \text{ if } \alpha < 0 \\ \frac{\alpha}{\alpha_{\max}} \text{ otherwise} \end{cases} \tag{5}$$

Note that due to the alternating roles of support foot and swing foot, $l_{\min}$ and $l_{\max}$ as well as $\alpha_{\min}$ and $\alpha_{\max}$ are exchanged between every step.

Then, $\mathcal{W}_{\text{rUNSWift}}$ is defined by the following exponential term, where the exponents R and T control the exact shape of the resulting volume:

$$\begin{aligned} w_{\text{rUNSWift}}(f, l, \alpha) &:= \left(\tilde{f}^T + \tilde{l}^T\right)^{\frac{R}{T}} + \tilde{\alpha}^R \\ \mathcal{W}_{\text{rUNSWift}} &:= \{(f, l, \alpha) \mid w(f, l, \alpha) \leq 1\} \end{aligned} \tag{6}$$

The resulting shape is visualized in Figure 5.

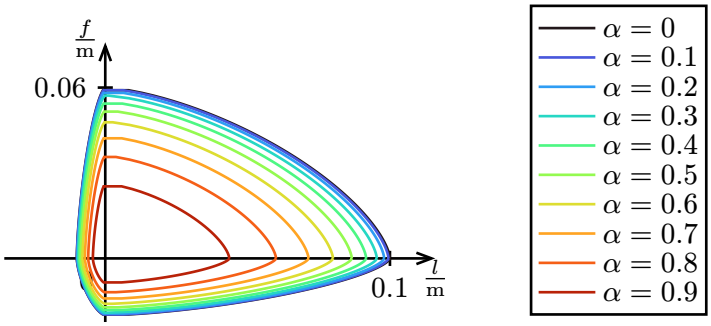

Figure 5: The walk volume defined by team rUNSWift's 2014 walk generator. The real shape of the walk volume is three-dimensional, but it is shown here using two-dimensional slices for different values of α. The parameters used in this example are: $f_{\text{min}} = -0.02\,\text{m}$, $f_{\text{max}} = 0.06\,\text{m}$, $l_{\text{min}} = -0.01\,\text{m}$, $l_{\text{max}} = 0.1\,\text{m}$, $\alpha_{\text{min}} = -1.0$, $\alpha_{\text{max}} = 1.0$, $R = 2.0$, and $T = 1.5$. The support foot in this example is the right foot. Note that the x-axis in the plot shows the l component, which points in the opposite direction compared to the plot.

The parameters chosen for Figure 5 are representative of the ones used in the SPL. The figure shows that the NAO can make substantially larger steps to the side than forwards, which motivates the use of sidewards or diagonal steps.

4 Implementation

The entire codebase of team HULKs is written in the Rust programming language. In order to facilitate easy interoperability with the existing code, this project's implementation of the MPC step planner is written in Rust as well.

Rust is a multi-purpose programming language with a focus on runtime efficiency, safety, and correctness [17]. Compared to other programming languages, Rust is relatively young, having only had the first stable release in 2015 [18]. As a consequence, the ecosystem of Rust libraries ("crates" in Rust parlance) still lags behind those of other languages in terms of number and completeness of crates for a few specific applications.

Writing the step planner in Rust thus comes with a number of advantages, as well as a few challenges: An implementation in Rust benefits from the language's strong memory safety guarantees and excellent runtime performance. The main challenges encountered stem from the fact that the Rust ecosystem still lacks numeric solver crates which are as full-featured as the ones available in other languages like Python, Julia, or C/C++. The implementation of this project uses the L-BFGS solver provided by the `argmin` crate [19], which comes with two major drawbacks. The following sections cover these challenges encountered during the project's implementation. Other numeric solver libraries were considered, but not investigated in detail. These include the `optimization-engine` crate [20], and bindings to C/C++ libraries like Ipopt.

4.1 Solver Setup

This section describes the setup chosen to interface with the numeric solver. All solver crates that were evaluated for this project have a similar interface, making it possible to replace specific solver crate used with a different one in the future. Users of these solvers have to provide the following:

- The number of input variables D, which defines the parameter space $\mathcal{P} = \mathbb{R}^D$
- A cost function $\mathcal{L} : \mathcal{P} \to \mathbb{R}$
- The gradient of the cost function $\nabla\mathcal{L}$

The goal of the solver is to find a parameter $\tilde{\boldsymbol{u}} \in \mathcal{P}$ which minimizes $\mathcal{L}(\tilde{\boldsymbol{u}})$.

In the case of this project, we define N as the number of steps to plan for, which is the size of the prediction horizon in the MPC formulation. D is then defined as $D = 3N$, as each step contains three variables.

In the implementation, $\tilde{\boldsymbol{u}}$ is simply a D-dimensional vector of floating-point numbers, but it is interpreted as $\boldsymbol{u}$, an N-dimensional vector of steps:

$$\begin{aligned} \tilde{\boldsymbol{u}} &= (\tilde{u}_1 \;\; \tilde{u}_2 \;\; ... \;\; \tilde{u}_D)^T \\ &= (f_1 \;\; l_1 \;\; \alpha_1 \;\; f_2 \;\; l_2 \;\; \alpha_2 \;\; ... \;\; f_N \;\; l_N \;\; \alpha_N)^T \\ \boldsymbol{u} &= (u_1 \;\; u_2 \;\; ... \;\; u_N)^T, \quad u_k = (f_k \;\; l_k \;\; \alpha_k)^T \end{aligned} \tag{7}$$

4.2 Solving a Constrained Problem

In many optimization libraries for other languages, solvers also take constraints on the control variables as an additional input. Since `argmin` has no support for constrained optimization, The constraints on the step size are instead implemented as "soft" constraints, which simply penalizes steps outside of the walk volume with a higher cost. This is not a perfect solution: The optimizer will still produce solutions with illegal steps, as long as it gains more through another part of the cost function. In practice, this means the step planner sometimes generates steps which are too large, if a large step enables it to take a shortcut and generate a lower cost.

Aside from using another optimizer which supports constraints out of the box, there are various other approaches which may yield more desirable results: It is possible to implement one of several methods on top of the existing solver to build "hard" constraints, such as the Penalty, Barrier, or Augmented Lagrangian methods. Implementing any of them is a promising avenue for future improvements to the implementation.

The penalty method works by replacing the constrained problem by an unconstrained one with an additional penalty term in the cost function, which increases the cost of a solution based on how much the constraints are violated by it. In practice, the amount by which the constraints are violated are first scaled by some *penalty function* before being added to the total cost. The penalty method then solves this unconstrained problem, this alone would still produce solutions that violate the penalties. The solution from this step is then used as the

starting point of a second iteration, where the penalty function is scaled by some factor. These steps are then repeated until some convergence criterium is met. By incrementally penalizing solutions outside of the legal solution space, the solutions of the successive unconstrained problems converge towards a legal solution of the constrained problem, while avoiding much of the numerical instability of immediately applying a steep penalty function. Since the penalty method may require an arbitrarily large penalty coefficient, it can also become ill-conditioned, as the floating-point numbers used in the calculation have lower precision as they get larger. Other approaches, such as the Augmented Lagrangian method, do not have this problem.

The implementation of this project implements neither of these methods, as a simple quartic penalty on the step size was sufficient to generate satisfactory results. This is discussed in greater detail in Section 4.3.2 and Chapter 5.

4.3 Cost Function

The cost function is a core component of the problem formulation. It rates a set of parameters $\boldsymbol{u} \in \mathcal{P}$, signaling to the optimizer how good of a solution to the problem it is. The better a solution is, the lower the corresponding output of the cost function will be. Thus, the design of the cost function is critical, as it determines which solutions the optimizer will converge to.

In this project, the cost function consists of three basic parts:

- A term that rewards progress along the direction of the path,
- a term that penalizes distance from the path plan,
- and a term that penalizes steps that are outside of the walk volume.

These three terms are the minimum needed to produce viable results, each term reflects one of the basic requirements for step planning outlined at the start of Chapter 3. In a future work, additional terms could be added to produce steps with more desirable properties, such as a term which limits the the change in each of the three components between consecutive steps, one which penalizes step poses that face away from the target, or a term which forces the pose reached after the last step in the sequence to have a specific orientation.

Note that the first two parts of the cost function are not concerned with the steps themselves, but rather the positions reached after executing them. This significantly increases the complexity of the gradient computation, the details of which are discussed in the next section.

Let $x_k = (p_{x,k} \;\; p_{y,k} \;\; \theta_k)^T$ be the state of the system after k steps, and let $u_k = (l_k \;\; f_k \;\; \alpha_k)^T$ be the k-th step. The system state x_k is computed with the state update function f defined by the rule shown in (1). The cost function $\mathcal{L}(\boldsymbol{u})$ used in this project is then defined as the sum of N evaluations of the stage cost function ℓ:

$$\mathcal{L}(\boldsymbol{u}) = \sum_{k=1}^{N} \ell(x_k, u_k) \tag{8}$$

The stage cost function ℓ rates the combination of a single system state x_k and the step u_k that was used to get there. It is defined as the sum of three parts, each implementing one of the three terms outlined at the start of this section:

$$\ell(x, u) = \ell_p(x) + \ell_d(x) + \ell_s(u) \tag{9}$$

The function ℓ_p rewards progress along the path, ℓ_d penalizes distance from the path, and ℓ_s penalizes steps outside the allowed walk volume. Each of these three parts contributes to the overall cost value, and each of them internally scales its output by multiplying it with some coefficient. These coefficients control how important each term is, they can be tuned to control the characteristics of the generated step plan. For example, increasing the coefficient for $\ell_d(x)$ increases the penalty for placing steps further away from the path, resulting in a step plan in which the steps stay closer to the path.

The following sections define each of the three terms in greater detail.

4.3.1 Path Progress and Path Distance

The first two components of the stage cost function, the *path progress cost function* ℓ_p and the *path distance cost function* ℓ_d, both rate the position at which a step lands. They should incentivize the optimizer to place the step positions in desirable locations. Both take the current state $x_k = (p_{x,k} \;\; p_{y,k} \;\; \theta_k)^T$ as input, but only consider the p_x and p_y components. Figure 6 visualizes both of them, as well as their sum.

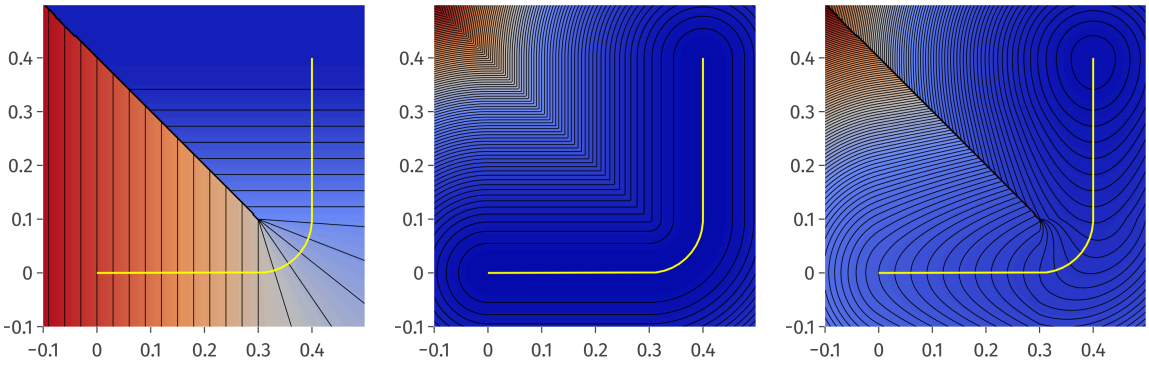

Figure 6: Visualization of the path progress (ℓ_p) and path distance (ℓ_d) components of the cost function and the sum of both, plotted with respect to p_x and p_y, for an L-shaped example path plan. The path plan is marked by the yellow line, starting at $(0, 0)$ and terminating at $(4, 4)$. The color at each point in the image represents the output of the cost function; Colors range from blue for small values to red for large ones. The value is constant along the black contour lines.

Both functions consider the current position $p = (p_x, p_y)$. They first compute the projection $\hat{p}$ of the point onto the path, which is the point on the path closest to p. Since the path is defined as a series of path segments, this point can be determined by projecting p onto each segment individually and finding the projection with the smallest euclidean distance to p.

For the following, we define l as the total length of the path, and $c : [0, l] \to \mathbb{R}^2$ as the parametric C^1-curve which coincides with the path, such that $\|c'(t)\| = 1 \; \forall t \in [0, l]$. It follows that $c(0)$ is the position where the path starts, and $c(l)$ is where it terminates.

The first component is $\ell_p(x)$, the path progress function. It computes the distance on the path between the start of the path and $\hat{p}$, which is the value of t such that $c(t) = \hat{p}$. It then returns the negative of this distance, since the goal is to assign "better" step positions, which are the positions further along the path, a lower score.

The function is implemented in a way to be bounded only in one direction: Walking in the wrong direction at the start of the path will lead to a higher cost. On the other hand, overshooting the end of the path leads to no additional reward, as the cost function is made so it reaches a plateau at the end of the path. This can be seen in the left picture in Figure 6: There are more contour lines to the left of the starting point in the bottom left, but the contour lines stop around the end of the path in the top right. The unbounded penalty before the start of the path is needed to prevent a gradient of zero at the initial position, as this would lead the optimizer to immediately terminate with a trivial solution: standing still.

The second component, $\ell_d(x)$, returns the squared distance from p to $\hat{p}$. This incentivizes the optimizer to place steps close to the planned path - a crucial requirement, as taking shortcuts might cause the robot to collide with obstacles.

4.3.2 Step Size Cost

The final component used in the stage cost function is the *step size cost function* $\ell_s(u)$. Its task is to incentivize the solver to generate feasible steps, which is achieved by penalizing steps based on the size of their components f, l, and α.

For this task, the walk volume introduced in Section 3.1 is used. Ideally, the optimization problem would simply be constrained by the inequality defined in (6):

$$\varpi_{\text{rUNSWift}}(f_k, s_k, \alpha_k) \leq 1 \quad \forall k \in (1, ..., N) \tag{10}$$

Since `argmin` does not support constrained optimization, this hard constraint needed to be relaxed to a soft constraint where illegal steps are discouraged by a higher cost. As a result, the optimizer will produce steps outside of the walk volume if taking such a step pays off with a greater reward from another part of the cost function. In a future work, this could be improved by implementing hard constraints.

The function ϖ, which defines the shape of the walk volume, assigns steps inside the walk volume a value smaller than one, and steps outside it a value greater than one. Note that steps that take full advantage of the kinematic limits of the NAO lie on the border of this walk volume. Importantly, there is no sharp boundary between these two regions in the value returned by ϖ; the function transitions smoothly between them. As a result, steps that are near the border of the walk volume, inside it and outside, would have a similar cost associated with them. There are two problems with this:

- Steps inside the walk volume near the border are penalized, even if they are legal.

- Steps just outside the walk volume are not penalized enough, this leads to the produced steps being relatively far outside of the walk volume.

Therefore, in order to produce steps that are close to the border of the legal region, the value returned by w is transformed by a *penalty function* $g(x)$, which reduces the penalty for steps inside the walk volume and increases it for steps outside of the walk volume. Such a penalty function should be relatively flat for values smaller than one, and have a steep increase for values greater than one.

The following penalty functions were evaluated:

- Integer powers, such a quadratic, $g(x) = x^2$ or a quartic, $g(x) = x^4$
- A modified exponential function, $g(x) = e^{\lambda(x-1)}$
- A logarithmic barrier function, $g(x) = -\lambda \log(1 - x)$

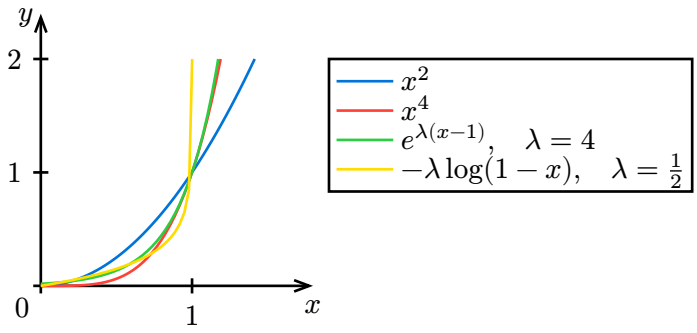

Figure 7: Different penalty functions.

Out of these options, the integer powers yielded the best results, and a quartic term was chosen for the implementation. The other two options drastically slowed down the convergence of the optimizer.

4.4 Cost Function Gradient

Unlike many popular optimization libraries in other languages, `argmin` has no built-in support for automatic differentiation, requiring users to provide the gradient of the problem formulation themselves.

The gradient of the cost funcion, $\nabla \mathcal{L}$, encodes the dependence of the cost value on each input variable. This informs the optimizer in which direction in the parameter space it should search to find a solution with a lower cost. In order to calculate $\nabla \mathcal{L}(\boldsymbol{u})$, it is necessary to find the partial derivatives of the stage cost function $\ell(x, u)$ with respect to all steps $u_i = (f_i \;\; l_i \;\; \alpha_i)^T$:

$$
\begin{aligned}
\nabla \mathcal{L}(\boldsymbol{u}) &= \begin{pmatrix} \frac{\partial \mathcal{L}(\boldsymbol{u})}{\partial u_1} & \dots & \frac{\partial \mathcal{L}(\boldsymbol{u})}{\partial u_N} \end{pmatrix}^T \\
\frac{\partial \mathcal{L}(\boldsymbol{u})}{\partial u_i} &= \begin{pmatrix} \frac{\partial \mathcal{L}(\boldsymbol{u})}{\partial f_i} & \frac{\partial \mathcal{L}(\boldsymbol{u})}{\partial l_i} & \frac{\partial \mathcal{L}(\boldsymbol{u})}{\partial \alpha_i} \end{pmatrix}^T \\
&= \frac{\partial}{\partial u_i} \sum_{k=1}^{N} \ell(x_k, u_k) \\
&= \sum_{k=1}^{N} \frac{\partial \ell(x_k, u_k)}{\partial u_i}
\end{aligned}
\tag{11}
$$

We apply the chain rule to split the partial derivatives of ℓ into their constituant parts:

$$
\frac{\partial \ell(x_k, u_k)}{\partial u_i} = \underbrace{\frac{\partial x_k}{\partial u_i}}_{\text{a)}} \underbrace{\frac{\partial \ell(x_k, u_k)}{\partial x_k}}_{\text{b)}} + \underbrace{\frac{\partial u_k}{\partial u_i}}_{\text{c)}} \underbrace{\frac{\partial \ell(x_k, u_k)}{\partial u_k}}_{\text{d)}}
\tag{12}
$$

The four parts of (12) can be interpreted as follows:

- Term a) states the dependence of the state of the system at time k on the i-th step.
- Term b) reflects the influence of the stage cost function on the current system state.
- Term c) is the dependence of the k-th step on the i-th step.
- Term d) gives the dependence of the stage cost function on the step that moved the system to its current state.

In the following, each of these terms is discussed in greater detail.

Term b) and d) encode how the cost value depends on either the current state or the current step. As defined in (9), the cost function ℓ is defined as the sum of three components ℓ_p, ℓ_d, and ℓ_s, where each component only depends on either the current state or the current step, but not both. This enables us to simplify the definition of both terms:

$$
\begin{aligned}
\frac{\partial \ell(x_k, u_k)}{\partial x_k} &= \frac{\partial \ell_p(x_k)}{\partial x_k} + \frac{\partial \ell_d(x_k)}{\partial x_k} \\
\frac{\partial \ell(x_k, u_k)}{\partial u_k} &= \frac{\partial \ell_s(u_k)}{\partial u_k}
\end{aligned}
\tag{13}
$$

Term a) describes how one step influences the state of the system at some other time. Recall that (4) defines $x_k = f(x_{k-1}, u_k)$. Then, we recursively define this relation as follows, where $\mathbf{0}_3 \in \mathbb{R}^{3\times 3}$ is the zero matrix:

$$
\frac{\partial x(k)}{\partial u_i} = \begin{cases} \mathbf{0}_3 & \text{if } i > k \\ \frac{\partial x_i}{\partial u_i} = \frac{\partial f(x_{i-1}, u_i)}{\partial u_i} & \text{if } i = k \\ \frac{\partial x_{k-1}}{\partial u_i} \frac{\partial x_k}{\partial x_{k-1}} = \frac{\partial x_{k-1}}{\partial u_i} \frac{\partial f(x_{k-1}, u_k)}{\partial x_{k-1}} & \text{otherwise} \end{cases}
\tag{14}
$$

The first case in (14) encodes the fact that steps cannot influence the state of the system in the past. The second case encodes that a step at time i only directly influences the system state at time i, governed by the state update function f. The last case shows how the

system state at time k depends on the previous system state, and thus, transitively, on all previous steps.

Equation (14) can also be written in a closed form:

$$\begin{aligned}\frac{\partial x_k}{\partial u_i} &= \frac{\partial x_i}{\partial u_i} \prod_{j=i}^{k-1} \frac{\partial x_{j+1}}{\partial x_j} \\ &= \frac{\partial f(x_{i-1}, u_i)}{\partial u_i} \prod_{j=i}^{k-1} \frac{\partial f(x_j, u_{j+1})}{\partial x_j}\end{aligned} \tag{15}$$

Since the input variables u_i do not depend on each other, term c) is simply defined as follows, where $\mathbf{I}_3 \in \mathbb{R}^{3\times3}$ is the identity matrix:

$$\frac{\partial u_k}{\partial u_i} = \begin{cases}\mathbf{I}_3 & \text{if } i = k \\ \mathbf{0}_3 & \text{otherwise}\end{cases} \tag{16}$$

As a result of the simplifications shown in (13) and (16), and the following fact:

$$\frac{\partial x_k}{\partial u_i} = \mathbf{0} \quad \text{if } i < k \tag{17}$$

Equation (11) can be rewritten as follows:

$$\frac{\partial \mathcal{L}(\boldsymbol{u})}{\partial u_i} = \frac{\partial \ell_s(u_i)}{\partial u_i} + \sum_{k=i}^{N} \frac{\partial x_k}{\partial u_i} \left(\frac{\partial \ell_p(x_k)}{\partial x_k} + \frac{\partial \ell_d(x_k)}{\partial x_k} \right) \tag{18}$$

This equation shows how the total cost depends on the i-th step through the gradient of the step size cost, as well as the sum over the path progress- and distance cost of all system states from the i-th to the end of the prediction horizon.

Thus, in order to implement the gradient of the cost function of this project, the following components need to be implemented:

- The gradients of the cost function parts ℓ_p, ℓ_d, and ℓ_s. These were implemented manually, and were partially tested by computing a finite-difference approximation of them and comparing the results.
- A method to compute $\frac{\partial x_k}{\partial u_i}$. This was implemented using automatic differentiation, which is explained in the next section.

4.5 Automatic Differentiation with Dual Numbers

The implementation of gradients represents a large part of the work required for the implementation of this project. In particular, the calculation of $\frac{\partial x_k}{\partial u_i}$ posed some unique challenges. This derivative describes the influence of the i-th step on the system state at some other time k. Note that $\frac{\partial x_k}{\partial u_i}$ is a 3×3 Matrix, because steps and system states are both three-dimensional. To calculate the overall cost function gradient $\nabla \mathcal{L}(\boldsymbol{u})$, $\frac{\partial x_k}{\partial u_i}$ needs to be computed for all $i \in (1, ..., N)$. As shown in (15), there is a closed formula to compute $\frac{\partial x_k}{\partial u_i}$, which requires the computation of partial derivatives of the state update function f. While

these single partial derivatives are not difficult to find, implementing the entire concept in code turned out to be more difficult. This is largely due to the growing nature of (15), which linearly increases in the number of terms with N.

Manually implementing the gradient of some function comes with a number of problems:

- Code Complexity: As previously shown, the calculation of some gradients can get relatively complex. As a result, both the initial implementation and the maintenance may require large amounts of effort. Furthermore, a manual implementation of a complex gradient is harder to understand than the function definition itself. Since complexity is perhaps the greatest source of programming errors, it should be avoided as much as possible.
- Correctness: An incorrect gradient calculation slows down the convergence of the solver in the best case, and prevents convergence entirely in the worst case. It is therefore relatively easy to implement the gradient of a function in a subtly wrong manner, such that it only slightly slows down the optimizer. Such an error can easily go unnoticed, requiring extensive testing to find. In addition, because the definitions of the function and its gradient are separate, additional care needs to be taken to ensure both implementations stay synchronized: When changing one of the two, the other needs to be updated accordingly.
- Code Duplication: Manually implemented gradients tend to lead to large amounts of code duplication, as every function whose derivative is required needs to be implemented at least twice: Once for the function itself and one for its gradient. The number of functions to implement is even higher for multi-variate functions where multiple partial derivatives are required.

Automatic Differentiation (AD), also referred to as *autodiff*, is a term used to describe various methods that solve the same problem: Automatically generating derivatives or gradients of arbitrary functions, using just the definition of the function itself. The use of AD enables the programmer to only implement the function itself, leaving the definition of the derivatives to the AD system. This solves all of the three problems mentioned previously:

- Code complexity is drastically reduced, as only the function definition itself is required to be implemented.
- Assuming the AD system used is well-tested, the gradients produced by it can also be expected to be correct. Since the gradient is generated directly from the function itself, it can also never be out of date.
- Code duplication caused by gradient definitions is eliminated entirely.

In addition to AD, there are also two other popular methods to automatically differentiate functions in computer programs, namely symbolic and numeric differentiation. Both of these methods have their own drawbacks, mainly the fact that symbolic differentiation at runtime can get computationally inefficient, and that numeric differentiation can lead to slightly inaccurate results and also requires multiple invocations of the original function. These two methods are not covered in greater detail in this work.

There are two kinds of AD, referred to as *forward* and *reverse accumulation*. Both make use of the chain rule to compute the derivatives of composite functions, which states the following:

$$\begin{aligned} &\text{let} \quad w = f(g(h(x))) = f(g(y)) = f(z). \\ &\text{then,} \quad \frac{\partial w}{\partial x} = \frac{\partial w}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial f(z)}{\partial z}\frac{\partial g(y)}{\partial y}\frac{\partial h(x)}{\partial x}. \end{aligned} \tag{19}$$

The difference between forward and reverse accumulation is the order in which this product of derivatives is computed. In the above example, forward accumulation would multiply the individual terms from right to left, which is forward in the sense that it is also the natural order of computation in the function itself. Reverse accumulation performs the multiplications in the opposite order. Forward accumulation is typically more straightforward to implement, since it can reuse the natural control flow of the function to be differentiated. Reverse accumulation typically involves constructing a directed acyclic graph of the computations in the function, and then traversing this graph from the output nodes back toward the input.

For vector-valued composite functions $f : \mathbb{R}^n \to \mathbb{R}^m$, where the terms in (19) are Jacobian matrices instead of scalars, this choice of order makes a difference in the computational complexity of the calculation: The number of rows in the intermediate products is always m when using reverse accumulation; With forward accumulation, the number of columns in the intermediate products is always n.

Therefore, for functions where $n \gg m$, it is more efficient to calculate the derivatives using reverse accumulation. The prime example of this is deep learning, where a scalar loss value is calculated from a high-dimensional input. As a result, differentiation in deep learning is typically done with *backpropagation*, a special case of reverse accumulation. For functions where $n \ll m$, the opposite is true, and forward accumulation is computationally more efficient.

In the case of this project, the problem solved with AD is the computation of the derivatives of all state variables $p_{x,k}$, $p_{y,k}$, and θ_k with respect to all input variables f_i, l_i, and α_i. Instead of calculating each of these gradients separately, the entire Jacobian matrix containing all of the gradients is calculated once per solver iteration. This way, intermediate results can be reused, increasing the efficiency of the implementation. This Jacobian is made up of $N \times N$ blocks defined by $\frac{\partial x_k}{\partial u_i}$. Since each $\frac{\partial x_k}{\partial u_i}$ is itself a 3×3 matrix, the total size of this Jacobian is $3N \times 3N = D \times D$. As a result of this square shape, neither forward nor reverse accumulation has an inherent advantage in terms of computational complexity. The main difference remaining between the two methods is the fact that forward accumulation is simpler to implement. Thus, a method using forward accumulation was chosen for the implementation of this project.

Specifically, the implementation makes use of *dual numbers* to achieve automatic differentiation. For the purpose of this thesis, we define a dual number (v, d) as the combination of a real number v and its gradient d with respect to all input variables:

$$d \in \mathbb{R}^D, \quad d_i = \frac{\partial v}{\partial \tilde{u}_i} \tag{20}$$

To achieve automatic differentiation, the calculation of the system state x_k is performed with dual numbers instead of reals. At the start of the calculation, all D input variables $\tilde{u}_i$ are replaced by their dual counterpart. The initial gradients of the variables are vectors with one element set to one and the rest being zero. This follows from the following fact:

$$\frac{\partial \tilde{u}_j}{\partial \tilde{u}_i} = \begin{cases} 1 \text{ if } i = j \\ 0 \text{ otherwise} \end{cases} \tag{21}$$

This replacement is shown in the following equation, where $e_i \in \mathbb{R}^D$ is the i-th unit vector.

$$\tilde{\boldsymbol{u}} = \begin{pmatrix} \tilde{u}_1 \\ \vdots \\ \tilde{u}_D \end{pmatrix} \rightsquigarrow \begin{pmatrix} (\tilde{u}_1, e_1) \\ \vdots \\ (\tilde{u}_D, e_D) \end{pmatrix} \tag{22}$$

Then, all operations that make up the state update function f take dual numbers as input and produce dual numbers as output. To achieve this, basic arithmetic operations ($+$, $-$, $*$, $\div$) need to be implemented for dual numbers, as well as basic trigonometric functions (sin, cos, etc.). In the Rust implementation, this means implementing the `RealField` trait of the `nalgebra` crate. The implementation makes use of the `num_dual` crate, which provides types for dual numbers and implements the needed traits for them [21].

These operations are defined such that they correctly combine the derivatives of their in put variables and the derivative of the operation themselves into the derivative term of the produced output. As an example, multiplication and the sin function are implemented roughly like follows:

$$\begin{aligned} (v_1, d_1) * (v_2, d_2) &= (v_1 * v_2, (v_1 * d_2 + v_2 * d_1)) \\ \sin((v, d)) &= (\sin(v), d * \cos(v)) \end{aligned} \tag{23}$$

The state update function f is then easily implemented for dual numbers by utilizing these elementary operations. As a result, the source code for f looks just like it would have for regular reals. The final result of this computation is the system state x_k, with all components $p_{x,k}$, $p_{y,k}$, and θ_k represented as a dual number. The dual component of each part of the system state x_k contains the gradient of it with respect to all input variables.

5 Evaluation

Despite the challenges encountered while implementing this project, the implementation is able to generate step plans which satisfy the three requirements outlined in Section 2.6. As can be seen in Figure 8, the generated steps exhibit many desirable characteristics:

- The optimizer takes advantage of diagonal and sideward steps to generate larger steps. The generated step plans all start with a few steps which orient the robot perpendicular to the path, enabling subsequent sideward steps. Note that the orientation after a step is not directly penalized by any part of the cost function; every rotation is the result of the

optimizer making the system land in a favorable state after it. This demonstrates the ability of MPC to generate control inputs which only pay off in the long run, optimizing over the entirety of steps in the horizon instead of only the next one.

- The steps are close to the limits of legal step sizes, neither much too small or too large.

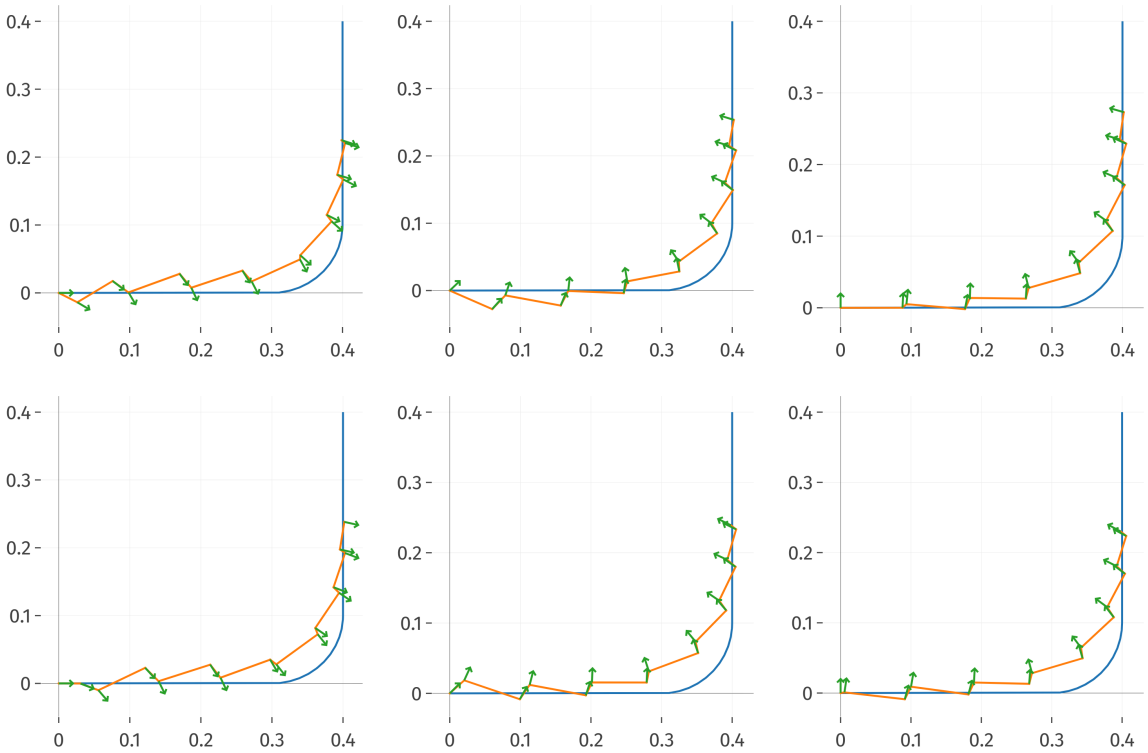

Figure 8: Generated step plans for an L-shaped example path, shown in blue, with different initial conditions. The vertices of the orange line signify the chosen step positions, and the green arrows indicate the target orientation. The pictures in the upper row show paths generated where the left foot is the initial support foot, the pictures on the bottom have the support foot initially set to the right foot. The three columns of pictures show the different paths generated for three differnent initial orientations, indicated by the green arrow at the origin. All of the step plans shown are the result of a single invocation of the optimizer, optimizing a sequence of 15 steps into the future. It is nicely visible how the sideward steps are all roughly 0.1m in length, which is the maximum allowed step length in this direction.

Some steps in the generated step plans are slightly outside the allowed walk volume. This is to be expected, since the optimizer is not given any hard constraints. In future work, the optimizer can be extended to support constrained optimization or switched out for another solver, which would alleviate this problem entirely. In practice, this should not be a serious issue as long as the deviation is small: Similar to the current step planner shown in Figure 4, the steps produced by the step planner have the f and l component clipped in order to remain inside the walk volume. As long as the difference induced by this clipping operation is small, steps are still close to the optimal path plan and the remaining steps still make for a good initial guess for the next iteration of the optimizer.

In the implementation, the individual contribution of each part of the stage cost function to the total is adjustable by changing a parameter. This allows for precise tuning of the generated steps, for example to decrease the deviation from the path at the expense of walking speed, or to further penalize large steps. The maximum allowed step size in each direction (f, l, α) is also adjustable, which can be used to discourage sidewards or backwards steps if desired. Furthermore, the implementation is written such that new cost function parts can be added with relative ease.

One important aspect of the feasibility of an MPC step planner is the runtime performance on the NAO. The implementation has not yet been integrated into the codebase of team HULKs, and thus also not yet executed on the NAO hardware. As a result, it is as of yet unclear how fast such a step planner would run in practice.

6 Conclusion

To summarize, this project has demonstrated that MPC is a promising approach to step planning in the RoboCup SPL. The implementation of this project is able to generate steps which fulfill the basic requirements for step planning, while exhibiting more complex behavior than the current solution. However, more work is required to bring this program into real use in the soccer robots of team HULKs. On the one hand, the code needs to be integrated into the rest of the HULKs codebase. Furthermore, the implementation has yet to reach feature parity with the existing step planner, mainly because it lacks the option to specify a certain target orientation. This could be added by an additional term of the cost function.

Because this approach was not yet tested on a real robot, its true potential remains to be seen. It is yet unclear how an MPC step planner would perform in the environment of a real match of robot soccer, and how fast this approach would run on a real NAO.

7 Future Work

There is a plethora of possibilities for future research into the topic of this project. This chapter outlines a few ideas for future work, but is by no means exhaustive.

To begin, it is unclear how close the walk volume used in the implementation is to the actual limits of the NAO. As outlined in Section 3.1, the exact shape depends in part on the rest of the walking stack. Therefore, it might be interesting to determine the true extents of the walk volume experimentally.

There are also different approaches to step planning in general to evaluate with MPC. For example, the problem could be reformulated without the path planning layer in the walking stack: Instead of penalizing distance from and rewarding progress along a pre-determined path, the cost function could instead penalize collisions with obstacles and reward progress toward the target. This way, the overall complexity of the walking stack would be reduced, since this reformulation does not meaningfully increase the complexity of the MPC formulation while eliminating an entire layer of the walking stack.

Furthermore, the current approach can be improved upon by a number of measures. On the one hand, the cost function can be expanded to include a variety of terms to improve several aspects of the generated steps. Section 4.3 briefly outlined a few possibilities for additional terms. Another interesting avenue is the evaluation of different numeric optimizers, which could greatly influence the runtime performance of the implementation. The start of Chapter 4 contains a brief overview of other available solvers. Using another optimizer with support for constrained optimization yields the additional benefit of enabling the steps to be constrained to the walk volume more effectively. Also, when using an optimizer which supports constraints, it is also possible to reformulate the problem with the *multiple shooting method*, which may exhibit faster convergence [12]. This reformulated problem also does not require the gradient of the system state with respect to all input variables, which could eliminate a large part of the complexity of the current implementation.

Another idea for improvement is to make the number of steps to plan dynamic. Currently, the optimizer always optimizes a fixed number of steps. However, when the target is within a few steps from the current position of the robot, it may be preferable to reduce the number of steps.

Yet another promising area of future research lies in the evaluation of other methods of automatic differentiation. There are other crates available that perform AD, but there is also ongoing work [22] to add automatic differentiation at compile-time to the Rust compiler, using the Enzyme autodiff plugin for LLVM [23]. Enzyme allows the programmer to define a function, and have the compiler automatically generate all derivatives of it. This would drastically reduce the complexity of the current code.

References

[1] "RoboCup." Accessed: Oct. 01, 2024. [Online]. Available: https://robocup.org/

[2] "RoboCup Standard Platform League." Accessed: Oct. 01, 2024. [Online]. Available: https://spl.robocup.org/

[3] "NAO the humanoid and programmable robot." Accessed: Oct. 02, 2024. [Online]. Available: https://www.aldebaran.com/en/nao

[4] "NAO - Technical Specifications." Accessed: Oct. 02, 2024. [Online]. Available: https://support.aldebaran.com/support/solutions/articles/80000959718-nao-technical-specifications

[5] R. Gelin, "NAO," in *Humanoid Robotics: A Reference*, A. Goswami and P. Vadakkepat, Eds., Springer Netherlands, 2018, pp. 1–22. doi: 10.1007/978-94-007-7194-9_14-1.

[6] Aldebaran, "full-nao.png." Accessed: Oct. 26, 2024. [Online]. Available: https://www.aldebaran.com/themes/custom/softbank/images/full-nao.png

[7] D. Tirumala *et al.*, "Learning Robot Soccer from Egocentric Vision with Deep Reinforcement Learning," 2024.

[8] Team HULKs, "Walking - HULKs Documentation." Accessed: Oct. 30, 2024. [Online]. Available: https://hulks.de/hulk/robotics/motion/walking/

[9] "path_planner.rs." [Online]. Available: https://github.com/HULKs/hulk/blob/215fef147987a72bb39c5f3b4be8ff1c73bf299a/crates/control/src/path_planner.rs#L312

[10] J. H. Lee, "Model predictive control: Review of the three decades of development," *International Journal of Control, Automation and Systems*, pp. 415–424, 2011, doi: 10.1007/s12555-011-0300-6.

[11] P. Stano *et al.*, "Model predictive path tracking control for automated road vehicles: A review," *Annual Reviews in Control*, vol. 55, pp. 194–236, 2023, doi: 10.1016/j.arcontrol.2022.11.001.

[12] J. B. Rawlings, D. Q. Mayne, and M. M. Diehl, *Model Predictive Control: Theory, Computation, and Design*. 2017.

[13] rUNSWift, "The 2014 Robocup SPL World Champion Team Code." Accessed: Oct. 18, 2024. [Online]. Available: https://github.com/UNSWComputing/rUNSWift-2024-release/blob/c5e92ff10872c3e8f5036219b282202a28f5b429/robot/motion/generator/Walk2014Generator.cpp#L175

[14] HULKs, "HULK." Accessed: Oct. 18, 2024. [Online]. Available: https://github.com/HULKs/hulk/blob/13a6f823b312f97a41cd1afc3c4db3419fa6efdd/crates/control/src/motion/step_planner.rs

[15] B-Human, "The official B-Human code releases." Accessed: Oct. 18, 2024. [Online]. Available: https://github.com/bhuman/BHumanCodeRelease/blob/6c09f4b8a

060851a49ac997221fef52bbe0aff8e/Src/Modules/MotionControl/WalkingEngine/Walk2014Generator.cpp

[16] B. Hengst, "rUNSWift Walk2014 Report." Accessed: Oct. 18, 2024. [Online]. Available: https://cgi.cse.unsw.edu.au/~robocup/2014ChampionTeamPaperReports/20140930-Bernhard.Hengst-Walk2014Report.pdf

[17] "The Rust Programming Language." Accessed: Oct. 15, 2024. [Online]. Available: https://www.rust-lang.org/

[18] "Announcing Rust 1.0." Accessed: Oct. 24, 2024. [Online]. Available: https://blog.rust-lang.org/2015/05/15/Rust-1.0.html

[19] "argmin: Optimization in pure Rust." Accessed: Oct. 17, 2024. [Online]. Available: https://www.argmin-rs.org/

[20] "OpEn: Fast and Accurate Nonconvex Optimization." Accessed: Oct. 17, 2024. [Online]. Available: https://alphaville.github.io/optimization-engine/

[21] P. Rehner and G. Bauer, "Application of Generalized (Hyper-) Dual Numbers in Equation of State Modeling," *Frontiers in Chemical Engineering*, vol. 3, 2021, doi: 10.3389/fceng.2021.758090.

[22] "Tracking Issue for autodiff." Accessed: Oct. 31, 2024. [Online]. Available: https://github.com/rust-lang/rust/issues/124509

[23] W. Moses and V. Churavy, "Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients," *Advances in Neural Information Processing Systems*, vol. 33, pp. 12472–12485, 2020, [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf