

Fakultät für Physik und Astronomie

Ruprecht-Karls-Universität Heidelberg

Masterarbeit

Im Studiengang Physik

vorgelegt von

Martin Huber

geboren in Frankenthal (Pfalz)

2019

**Implementierung und Fusion von
modellprädiktiver Regelung mit
Verhaltensklonung zur
autonomen Navigation von humanoiden Robotern**

Die Masterarbeit wurde von Martin Huber

ausgeführt am

Institut für Optimierung, Robotik und Biomechanik

unter der Betreuung von

Frau Prof. Katja Mombaur

Department of Physics and Astronomy

University of Heidelberg

Master thesis

in Physics

submitted by

Martin Huber

born in Frankenthal (Pfalz)

2019

**Implementation and Fusion of
Nonlinear Model Predictive Control with
Behavioral Cloning for
Autonomous Navigation of Humanoid Robots**

This Master thesis has been carried out by Martin Huber
at the
Institute of Optimization, Robotics and Biomechanics
under the supervision of
Ms. Prof. Katja Mombaur

Verhaltensklonung zur autonomen Navigation humanoider Roboter mit Nichlinearer Modellprädiktiver Regelung:

In dieser Arbeit erkunden wir die Möglichkeiten der Verhaltensklonung zur autonomen Navigation humanoider Roboter durch bloße Bilder. Hierfür wird eine nichtlineare, Modellprädiktive Regelung, die es ermöglicht, stabile Lauftrajektorien in Echtzeit zu erzeugen, implementiert und evaluiert. Es wird demonstriert, dass minimale Veränderung in der Bildverarbeitung genügen, um vielseitige Bewegungsstrategien in vielfältigen dynamischen und statischen Umgebungen zu erlernen. Diese Einfachheit der Lösung wird als passende Ergänzung zur Meidung von Konvexen Hindernissen identifiziert, welche durch Randbedingungen die Lösungen der nichtlinearen Modellprädiktiven Regelung einschränken. Alle Experimente werden an Heicub, einer Variante des iCub, durchgeführt, welcher speziell für Optimalsteuerung in der Fortbewegung am Istituto Italiano di Tecnologia in Genova entwickelt wurde. Die Auswertung von Stabilitätskriterien zeigt weiterhin, dass ein menschlicher Kontrolleur, einem künstlichen Agenten gegenüber, nicht überlegen ist. Um die präsentierte Methode schließlich auf tauschende Aufgaben zu erweitern, vereinfachen wir die wechselnden Umgebungen auf ein gut gelöstes Klassifizierungsproblem.

Behavioral Cloning for Autonomous Navigation of Humanoid Robots with Nonlinear Model Predictive Control:

In this work we investigate the capabilities of behavioral cloning for autonomous navigation of humanoid robots from raw image input. Therefore, a nonlinear model predictive control that allows for real time generation of stable walking trajectories is implemented and evaluated. It is demonstrated that minor modifications in the vision pipeline are sufficient for the learning of versatile motion strategies in various dynamic and static environments. This simplicity is identified as a well suited addition to the avoidance of convex obstacles, which are represented by constraints to the solution of the implemented nonlinear model predictive control. All of the experiments are carried out on Heicub, a descendant of the iCub, which was especially designed for optimal control in locomotion at the Istituto Italiano di Tecnologia in Genova. The evaluation of stability criteria further reveals that there is no superiority of a human controller over an artificial agent. Finally, to extend the proposed approach to changing tasks, we boil the variation of environments down to a well solved classification problem.

Contents

1	Introduction	8
2	State of the Art	9
3	Background	10
3.1	Humanoid Walking	11
3.1.1	Zero Moment Point	13
3.1.2	Nonlinear Model Predictive Control	17
3.1.3	Interpolating Trajectories	27
3.1.4	Kinematics	30
3.2	Machine Learning	31
3.2.1	Neural Networks	32
3.2.2	Behavioral Cloning	37
3.2.3	Reinforcement Learning	38
3.2.4	Image Processing	40
4	Methods	49
4.1	Software	49
4.2	Implementation	49
5	Experiments	50
5.1	User Controlled Walking	50
5.1.1	Benchmarking of Implementation	50
5.1.2	Performance in Test Environment	52
5.2	Autonomous Walking	54
5.2.1	Camera Calibration	54
5.2.2	Depth Parameter Tuning	56
5.2.3	Data Acquisition and Training	58
5.2.4	Performance in Test Environment	61
5.2.5	Proximal Policy Optimization	65
6	Conclusion	68
6.1	Contributions	68
6.2	Future Work	68

I Appendix	69
A Software Installation	70
A.1 Build the Pattern Generator	70
A.2 Build the Android Joystick App	71
A.3 Build Proximal Policy Optimization	72
A.4 Build Simulation Models	72
A.5 Build Third Party Software	73
A.5.1 Necessary Dependencies	73
A.5.2 Real Robot and Simulation Dependencies	74
A.5.3 Deep Learning Dependencies	76
B Startup and Shutdown Procedures	77
B.1 Real Robot Startup	77
B.1.1 Start the Motors	78
B.1.2 Start the Cameras	80
B.1.3 Connect your own Laptop	81
B.2 Simulated Robot Startup	81
B.3 Start the Pattern Generator	82
B.3.1 Control via the Terminal	82
B.3.2 Control via the Android Joystick App	83
B.4 Start the Behavioral Augmentation Demo	84
B.5 Real Robot Shutdown	85
C Lists	86
C.1 List of Figures	86
C.2 List of Tables	88
D Bibliography	90

1 Introduction

2 State of the Art

3 Background

To generate dynamically balanced walking trajectories for humanoid robots and to let them navigate the environment autonomously, there are several posed challenges that we need to cover. As the logical starting point, in section 3.1 - Humanoid Walking, we want to address the real time generation of walking trajectories for humanoid robots first, and then think of ways to replace the human user by an artificial agent in the control loop (fig. 3.1). The generation of patterns in real time becomes feasible by treating the robot's physics in a simplified way as those of an inverted pendulum (sec. 3.1.1). The zero moment point of the linear inverted pendulum will therefore serve as the balance criteria for the solution of a sequentially quadratic problem (sec. 3.1.2). Resulting positions and orientations for the center of mass and the feet will then be interpolated (sec. 3.1.3) and passed as constraints to the inverse kinematics (sec. 3.1.4) so to transform them into joint angles that can be sent to the humanoid's motor controllers.

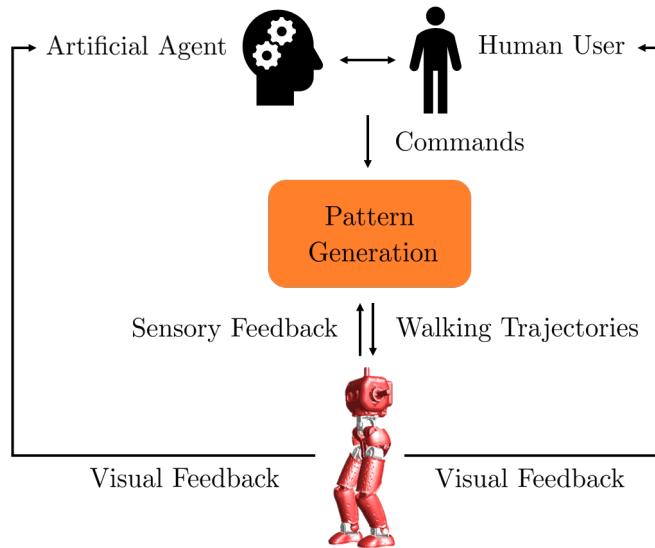


Figure 3.1: Simplified version of the proposed control loop to navigate the robot with either a human user or an artificial agent.

To close the control loop and to steer the robot towards desired goals, whilst avoiding obstacles, requires some sort of high level command that arises from visual feedback. As discussed in section 2 - State of The Art, there are several ways to achieve this, among them human users. Of particular interest to us are novel methods that evolved from the toolbox of machine learning techniques, as they decrease the computational cost into non existence. Let alone this fact enables us to run

them onboard on light weight hardware with low energy usage, which is critical in the domain of humanoid robots. Center to these new methods will be neural nets that we will train on solving the task of autonomous navigation in two different ways. One of which clones the behavior of a human user (sec. 3.2.2), whereas the second presented method (sec. 3.2.3) explores policies and tries to find solutions on its own.

As a side note, within the following chapters there will always be made references to the actual implementation of the presented concepts. This shall enable future readers to bridge the gap between theory and application.

3.1 Humanoid Walking

To get started with and to understand the presented concepts that generate dynamically balanced walking trajectories, we shall have a look at figure 3.1 once more. The pattern generation therein (orange box), consists of four main building blocks: Forward kinematics, nonlinear model predictive control (NMPC), interpolation, and inverse kinematics. The relation between these four building blocks is shown in fig. 3.2. The natural entry point, to this otherwise closed control loop, is given by the

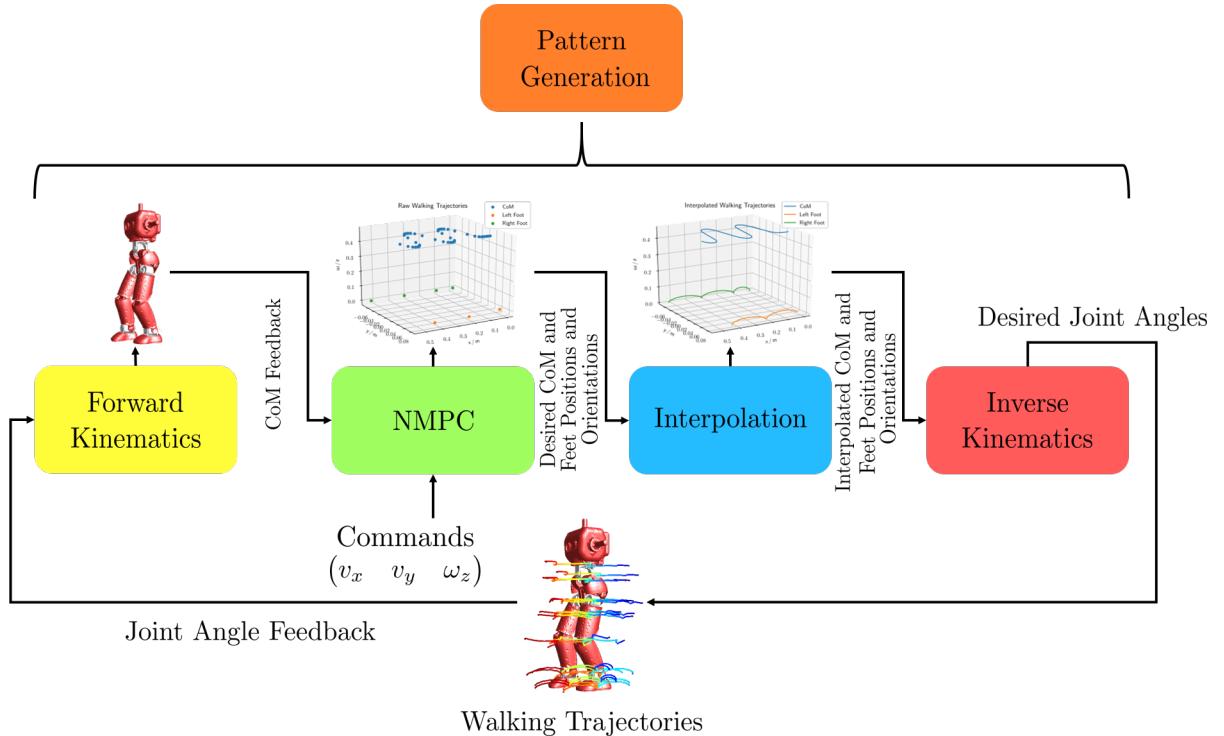


Figure 3.2: Building blocks of the pattern generation. To understand the greater picture, a connection can be drawn to fig. 3.1, where the orange box represents the one shown in this figure.

commands that enter the nonlinear model predictive control. Commands are passed

in the form of a desired velocity \mathbf{v}_{ref} that the robot's center of mass (CoM) shall satisfy optimally according to a cost function that also takes dynamic balance and a smooth motion into account. The future desired positions and orientations for the CoM and the feet then result from the solution to a sequentially quadratic problem that tries to minimize this cost function. The balance criteria within this problem formulation is based upon the zero moment point (ZMP) around which the whole control framework is built. It is only by simplifying the robot's model that we can solve the optimal control problem in real time. Therefore, we assume the robot to be a linear inverted pendulum, for which we have a well defined analytical relation between the CoM and the ZMP. The minimization of the distance between the analytical expression of the ZMP and the foot placement results in the desired dynamic balance. As shown in fig. 3.2, the desired CoM and the feet positions and orientation, as they are obtained from the NMPC, are sparsely distributed in space. Moreover, there is neither information about how the feet shall move along the z-axis, nor along the x-, and y-axis, but only where they should be placed in the x-y-plane. Therefore, as the subsequent step to the NMPC, we need to add an interpolation. The interpolation interpolates the trajectories of the CoM to obtain a finer sampling time. Additionally, the movement of the feet in the x-, y-, and z-direction, as well as their orientation, is computed by polynomials that we require to satisfy the initial and end conditions of the foot placement. Put together, the nonlinear model predictive control and the interpolation between the resulting subsequent solutions for the positions and orientation of both, the CoM and the feet, describe dynamically balanced trajectories, given that the humanoid robot of interest resembles the physics of an inverted pendulum. Now to bridge the gap between dynamically balanced trajectories in Cartesian space, and a humanoid robot that actually satisfies them with its CoM and its feet, the inverse kinematics problem needs to be addressed. The inverse kinematics, which follow immediately after the interpolation step, take the positions and orientations of the CoM and the feet as constraints and find a composite of joint angles that fulfill them. The continuity of subsequent solutions is therein assured by initializing the inverse kinematics with the previous solution. Resulting joint angles, once passed to the humanoid, then result in walking trajectories, as indicated in fig. 3.2 by the colored lines at the joints of the robot. Due to the inherent mismatch of the robot's physics from that of an inverted pendulum, as well as other effect like friction, there is a chance that the desired joint angles differ from the actually achieved ones. To compensate for the discrepancy, the last building block of the pattern generation is the feedback of the measured CoM to the NMPC. The CoM is computed by reading out the achieved joint angles, so that the forward kinematics can be utilized to determine the positions and orientations of the humanoid's links in space, and therefore the CoM.

As already highlighted in the previous paragraph, special attention has to be given to the zero moment point, since it defines the central concept of the presented pattern generation. We therefore will explain its theoretical foundations, as well as

its analytical relationship to the CoM for simplified physical models, and ways to measure it with force torque sensors in the section that lies ahead - Zero Moment Point.

3.1.1 Zero Moment Point

The key metric in this work, for the generation of a dynamically balanced gait, is the zero moment point. The concept was first introduced by Miomir Vukobratović and Davor Juričić in 1968 [1][2] and first utilized 1984 to generate walking trajectories for the WL-10RD robot [3]. The most intuitive understanding for the ZMP arises by thinking about the realization of the simplest arbitrary possible walking motion for which a humanoid robot will not fall. This motion is achieved by ensuring the feet's whole area, and not only the edge, is in contact with the ground [4], or put in other words, we require the robot not to rotate about its feet edges. This constraint can be met by having a reaction force \mathbf{F}_r between the foot and the ground, which compensates for all external moments \mathbf{M}_x , and \mathbf{M}_y around the x-, and y-axis at any time (fig. 3.3). The point \mathbf{r} , at which the reaction force acts, is physically

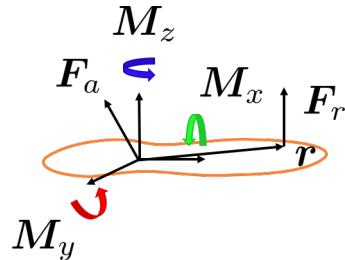


Figure 3.3: Forces acting on the sole.

only meaningful if it lies within the support polygon of the foot. Not only can it not exist outside of the support polygon, since there was no point of interaction between the foot and the ground then, but also was the robot to overturn under these circumstances. Therefore, the ZMP is defined as that point on the ground at which the net moment of the inertial forces has no component along the horizontal axes [5][6]. We now came to appreciate the importance of the support polygon for the definition of the zero moment point. The support polygon is defined as the

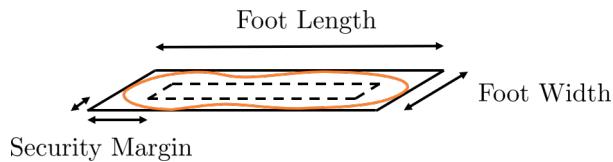


Figure 3.4: Full support polygon, and the resulting support polygon with security margin (dashed lines).

convex hull of all contact points of the feet with the ground, so the minimal number

of points to fully contain all of them. As the most restrictive case for balance, in this work we will only consider the support polygon of one foot at a time. Since the convex hull of a foot is well described by a rectangle, we only rely on the foot width ([link](#)), and foot length ([link](#)) to fully describe it. Also, to ensure that the zero moment point never comes close to the edges of the feet and therefore to provide balance, we define a security margin to their borders ([link](#)). The respective values are robot specific and can be set in the configurations file by following the provided links.

As already pointed out, within this work, we will use a simplified physical model of the humanoid solve the optimal control problem in real time. We will deal with this approximation in the following paragraph - Zero Moment Point of a Linear Inverted Pendulum.

Zero Moment Point of a Linear Inverted Pendulum

Dynamically balanced walking trajectories can be generated by simplifying the dynamics of humanoid robots to those of a linear inverted pendulum [7]. A rigorous derivation for the analytic relation between the center of mass and the zero moment point of a linear inverted pendulum can be found in [8], but for the sake of simplicity we rather explain the physics in terms of cutting forces, for which a short introduction can be found in the summary of the lecture Robotics 1 ([link](#)). The

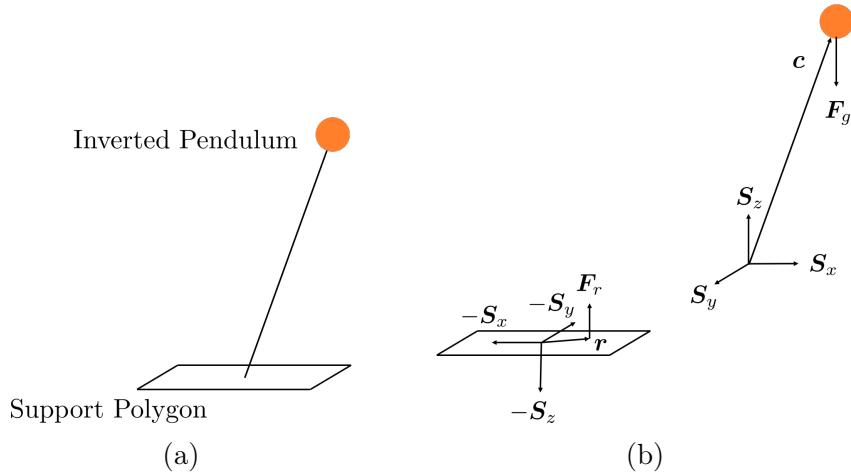


Figure 3.5: Linear inverted pendulum with a support polygon (a), and the corresponding free body diagram with cutting forces $\mathbf{S}_{x/y/z}$ (b).

system of interest is shortly depicted in figure 3.5. We assume the support polygon of the shown linear inverted pendulum to have zero mass. By introducing cutting forces $\mathbf{S}_{x/y/z}$ for each degree of freedom in which the motion of the linear inverted pendulum is restricted, we obtain the free body diagram (fig. 3.5), for which the

acting forces are

$$m\ddot{\mathbf{c}} = \mathbf{S} - \mathbf{F}_g \quad (3.1)$$

$$\mathbf{0} = -\mathbf{S} + \mathbf{F}_r \quad (3.2)$$

where $\mathbf{S} = \mathbf{S}_x + \mathbf{S}_y + \mathbf{S}_z$. The respective moments, since we do not take any inertias into account, are given by

$$\mathbf{0} = (\mathbf{0} - \mathbf{c}) \times \mathbf{S} + \mathbf{M} \quad (3.3)$$

$$\mathbf{0} = (\mathbf{r} - \mathbf{0}) \times \mathbf{F}_r - \mathbf{M} \quad (3.4)$$

where the transfer of the moment \mathbf{M} may for example be induced by friction. If we replace $\mathbf{S} = \mathbf{F}_r$ from eq. 3.2, equations 3.3 and 3.4 yield

$$\mathbf{0} = (\mathbf{r} - \mathbf{c}) \times \mathbf{S} = \begin{pmatrix} (r_y - c_y)S_z - (r_z - c_z)S_y \\ -(r_x - c_x)S_z + (r_z - c_z)S_x \\ (r_x - c_x)S_y - (r_y - c_y)S_x \end{pmatrix} \quad (3.5)$$

Since our goal is to have a robot that does not fall, we want to achieve that the acceleration along the z-axis becomes zero, hence $\ddot{c}_z = 0$. Given this assumption, we can infer from eq. 3.1 that $S_z = mg$, as well as $S_x = \ddot{c}_x m$, and $S_y = \ddot{c}_y m$. Furthermore, our foot shall not lift off the floor, and therefore we have $r_z = 0$. If we take these assumptions and plug them into the first two rows of eq. 3.5, we find

$$r_x = c_x - c_z \frac{\ddot{c}_x}{g} \quad (3.6)$$

$$r_y = c_y - c_z \frac{\ddot{c}_y}{g} \quad (3.7)$$

Therein, r_x , and r_y are the x-, and y-coordinates of the zero moment point, given the assumption of a linear inverted pendulum. We can see that the position is dependent on the height of the point mass, which is in turn dependent on the robot. The specific values can be set in the configurations file ([link](#)).

We have now found a simple analytic expression for the relationship of the zero moment point and the center of mass, which will help us to formulate an optimal control problem that we can solve in real time. This simplification is of course only true to some extend, and we need to find a way to verify its accuracy. The easiest way to do so is to measure the real zero moment point. We will further elaborate on this within the next paragraph - Measurement of the Zero Moment Point, and we will derive a method that only relies on force torque sensors in the ankle.

Measurement of the Zero Moment Point

There are several methods that enable us to measure the position of the zero moment point, among them the utilization of pressure sensitive soles, as outlined in [8].

Furthermore, there exist approximate approaches that involve the knowledge of all acting external forces [9], which can for example be obtained from unconstrained inverse dynamics [10]. Since we can rely on measurements of force torque sensors that are located at the ankles, we will infer the position of the zero moment point from them [8]. If we consider the force torque sensor to be located at position \mathbf{p}_i

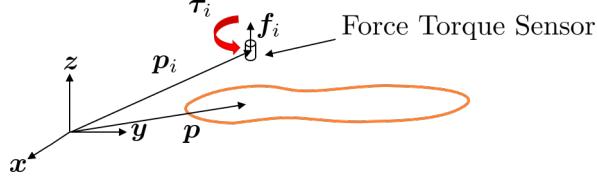


Figure 3.6: Force torque sensors at the foot's ankle.

(fig. 3.6), then we can obtain the moment about any point \mathbf{p} according to eq. 3.8.

$$\boldsymbol{\tau}(\mathbf{p}) = (\mathbf{p}_i - \mathbf{p}) \times \mathbf{f}_i + \boldsymbol{\tau}_i \quad (3.8)$$

by definition, the moment about the zero moment point vanishes along the horizontal axes, therefore we can then set $\tau_x = \tau_y = 0$ in eq. 3.8 and then solve for the position to obtain the zero moment point (eq. 3.9 and 3.10).

$$p_x = \frac{[-\tau_{i,y} - (p_{i,z} - p_z)f_{i,x} + p_{i,x}f_{i,z}]}{f_{i,z}} \quad (3.9)$$

$$p_y = \frac{[-\tau_{i,x} - (p_{i,z} - p_z)f_{i,y} + p_{i,y}f_{i,z}]}{f_{i,z}} \quad (3.10)$$

If we further choose our coordinate system to lie along the z-axis of the force torque sensor (fig. ...), we can simplify equations 3.9 and 3.10 to find

$$p_x = \frac{(-\tau_{i,y} - f_{1,x}d)}{f_{1,z}} \quad (3.11)$$

$$p_y = \frac{(\tau_{i,x} - f_{1,y}d)}{f_{1,z}} \quad (3.12)$$

We can use equations 3.11 and 3.12 to determine the position of the zero moment point for the left and the right foot with respect to coordinates frames that are attached to the respective foot. These circumstances change once not only one, but both feet are in contact with the ground. What still holds true, in the case of a dynamically balanced gait, is the fact that the positions which we just obtained from equations 3.11 and 3.12 represent points where the interaction of the robot with the environment can solely be described by a single force along the z-axis. All other forces or torques cancel out. Therefore, to determine the position of the zero moment point for the double support phase, we need to modify equation 3.8 slightly. This yields

$$\boldsymbol{\tau}(\mathbf{p}) = \sum_{i \in \{L,R\}} (\mathbf{p}_i - \mathbf{p}) \times \mathbf{f}_i \quad (3.13)$$

where the individual torques are now zero and the only forces \mathbf{f}_i that exist between the robot and the environment can be described by the z-component which are measured at the ankles' force torque sensors. Yet again, to obtain the position of the zero moment point, we have to set the x-, and y-components of the torque in equation 3.13 to zero and find

$$p_x = \frac{\sum_{i \in \{L,R\}} p_{i,x} f_{i,z}}{\sum_{i \in \{L,R\}} f_{i,z}} \quad (3.14)$$

$$p_y = \frac{\sum_{i \in \{L,R\}} p_{i,y} f_{i,z}}{\sum_{i \in \{L,R\}} f_{i,z}} \quad (3.15)$$

These expressions of course only hold true in a shared coordinate system and therefore we need to transform the position of the zero moment point which we obtained from equations 3.11 and 3.12 to the world frame. Finally, we can write down the formulation for the zero moment point which holds equally true for the single and double support phase

$$p_x = \frac{p_{R,x} f_{R,z} + p_{L,x} f_{L,z}}{f_{R,z} + f_{L,z}} \quad (3.16)$$

$$p_y = \frac{p_{R,y} f_{R,z} + p_{L,y} f_{L,z}}{f_{R,z} + f_{L,z}} \quad (3.17)$$

At this point we are now equipped with a general understanding for the zero moment point, as well as with the knowledge of simplified models to compute it analytically, and a method to measure it so that we can evaluate the performance of a potential pattern generator which is based upon the zero moment point. Therefore, in the next chapter - Nonlinear Model Predictive Control, we will try to understand a method that allows us to generate dynamically balanced center of mass and feet trajectories, which satisfy the just introduced concepts optimally, given a weighting factor.

3.1.2 Nonlinear Model Predictive Control

At the heart of nonlinear model predictive control stands sequential quadratic programming. Before we come to the actual problem formulation, we need to understand how sequential quadratic programming can be used to solve nonlinear optimization problems. We will then come to recognize that if we can find a canonical formulation of our problem, it will become possible to apply sequential quadratic programming to it. The next paragraph - Sequential Quadratic Programming, will therefore shortly introduce the reader to the desired method that will be used to solve the nonlinear optimization problem, while the subsequent paragraph - Canonical Formulation of Nonlinear Model Predictive Control, will then explain how to fit humanoid walking into this framework.

Sequential Quadratic Programming

Sequential quadratic programming is a powerful concept to solve nonlinearly constrained optimization problems. The nonlinear programming problem to be solved is of the form

$$\min_{\mathbf{x}} \frac{1}{2} f(\mathbf{x}) \quad (3.18)$$

$$\text{subject to: } \mathbf{h}(\mathbf{x}) = \mathbf{0} \quad (3.19)$$

$$\mathbf{g}(\mathbf{x}) \leq \mathbf{0}, \quad (3.20)$$

where $f : \mathbb{R}^N \rightarrow \mathbb{R}$, $\mathbf{h} : \mathbb{R}^N \rightarrow \mathbb{R}^M$, and $\mathbf{g} : \mathbb{R}^N \rightarrow \mathbb{R}^P$ [11]. These problems arise in a variety of applications in science and include quadratic problems as special cases. The great strength of sequential quadratic programming is its ability to solve problems with nonlinear constraints, and its basic idea is to model nonlinear programming at an approximate solution \mathbf{x}_k by a quadratic subproblem, so to find a solution to this subproblem, in order to construct a better approximation \mathbf{x}_{k+1} . Now given an objective function $f(\mathbf{x})$ represents a sum of squares, the problem at hand turns into a nonlinear least squares problem, and the minimization can be expressed in terms of a Gauss-Newton method [12]. That is, given an objective function $f(\mathbf{x}) = \mathbf{F}(\mathbf{x})^T \mathbf{F}(\mathbf{x})$, where $\mathbf{F} = (f_1, \dots, f_l)^T$, we can apply a quasi Gauss-Newton method as follows

$$\nabla^2 f(\mathbf{x}) \Delta \mathbf{x} + \nabla f(\mathbf{x}) = 0, \quad (3.21)$$

where the gradient and the Hessian matrix are given as

$$\nabla f(\mathbf{x}) = \nabla \mathbf{F}(\mathbf{x}) \mathbf{F}(\mathbf{x}) \quad (3.22)$$

$$\nabla^2 f(\mathbf{x}) = \nabla \mathbf{F}(\mathbf{x}) \nabla \mathbf{F}(\mathbf{x})^T + \mathbf{B}(\mathbf{x}). \quad (3.23)$$

Therein, $\mathbf{B}(\mathbf{x}) = \sum_i^l f_i(\mathbf{x}) \nabla^2 f_i(\mathbf{x})$. If we are now sufficiently close to an optimal solution \mathbf{x}^* , such that $\mathbf{F}(\mathbf{x}^*) = (f_1(\mathbf{x}^*), \dots, f_l(\mathbf{x}^*))^T = \mathbf{0}$, we can neglect $\mathbf{B}(\mathbf{x}^*)$, which turns equation 3.21 into the previously stated Gauss-Newton minimization problem

$$\min_{\Delta \mathbf{x}} \|\nabla \mathbf{F}(\mathbf{x}_k)^T \Delta \mathbf{x} + \mathbf{F}(\mathbf{x}_k)\|_2^2, \quad (3.24)$$

where a new iterate is obtained by $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \Delta \mathbf{x}$ with an appropriate step length parameter α_k . This is because equation 3.21 defines the normal equations of equation 3.24. The presented approach assures quadratic convergence, when starting sufficiently close to an optimal solution. Within the next section, we will understand how to apply this concept to control the zero moment point of a linear inverted pendulum in a balanced manner.

Canonical Formulation of Nonlinear Model Predictive Control

Not only do we want to keep a humanoid robot dynamically balanced in terms of the zero moment point, which we derived in equations 3.6 and 3.7, but further do we want to assure this for future time steps that are yet ahead. The underlying model predictive control got first introduced in [7], and is based upon a linear time stepping scheme, which integrates the current jerk of the center of mass iteratively, so to estimate its future position. We will briefly present it in the following paragraph - Linear Time Stepping Scheme.

Linear Time Stepping Scheme

Suppose that the center of mass' jerk $\ddot{\ddot{c}}_k$ at time step t_k is constant, then given the current acceleration \ddot{c}_k , we can obtain the acceleration \ddot{c}_{k+1} at time step t_{k+1} by simple integration. We can do the same for the velocity and position and therefore obtain

$$c_{k+1} = \frac{T^3}{6} \ddot{\ddot{c}}_k + \frac{T^2}{2} \ddot{c}_k + T \dot{c}_k + c_k \quad (3.25)$$

$$\dot{c}_{k+1} = \frac{T^2}{2} \ddot{\ddot{c}}_k + T \ddot{c}_k + \dot{c}_k \quad (3.26)$$

$$\ddot{c}_{k+1} = T \ddot{\ddot{c}}_k + \ddot{c}_k, \quad (3.27)$$

where $T = t_{k+1} - t_k$. We can rewrite this in compact form by

$$\mathbf{c}_{k+1} = \mathbf{A}\mathbf{c}_k + \mathbf{B}\ddot{\ddot{c}}_k \quad (3.28)$$

$$\mathbf{c}_{k+1} = \begin{pmatrix} c_{k+1} \\ \dot{c}_{k+1} \\ \ddot{c}_{k+1} \end{pmatrix}, \quad \mathbf{A} = \begin{pmatrix} 1 & T & \frac{T^2}{2} \\ 0 & 1 & T \\ 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} \frac{T^3}{6} \\ \frac{T^2}{2} \\ T \end{pmatrix} \quad (3.29)$$

Now by recursion, one obtains the positions, velocities, and accelerations for n future time steps via

$$\mathbf{c}_{k+n} = \mathbf{A}^n \mathbf{c}_k \sum_{i=1}^n \mathbf{A}^{i-1} \mathbf{B} \ddot{\ddot{c}}_{k+n-i}, \quad (3.30)$$

where $n \in [1, N]$. Altogether we have

$$\mathbf{A}^n = \begin{pmatrix} 1 & nT & n^2 \frac{T^2}{2} \\ 0 & 1 & nT \\ 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{A}^n \mathbf{B} = \begin{pmatrix} (1 + 3n + 3n^2)T^3/6 \\ (1 + 2n)T^2/2 \\ T \end{pmatrix}. \quad (3.31)$$

The amount of time NT that we predict into the future, is what we call the preview horizon. If we now concatenate the single entries of \mathbf{c}_{k+n} for all $n \in [1, N]$ into one

expression, we can relate the initial states \mathbf{c}_k (link) to the states on the preview horizon in an even more compact form. With the concatenations (link)

$$\mathbf{C}_{k+1} = \begin{pmatrix} c_{k+1} \\ \vdots \\ c_{k+N} \end{pmatrix}, \quad \dot{\mathbf{C}}_{k+1} = \begin{pmatrix} \dot{c}_{k+1} \\ \vdots \\ \dot{c}_{k+N} \end{pmatrix}, \quad \ddot{\mathbf{C}}_{k+1} = \begin{pmatrix} \ddot{c}_{k+1} \\ \vdots \\ \ddot{c}_{k+N} \end{pmatrix}, \quad \ddot{\mathbf{C}}_k = \begin{pmatrix} \ddot{c}_k \\ \vdots \\ \ddot{c}_{k+N-1} \end{pmatrix}, \quad (3.32)$$

we obtain (link)

$$\mathbf{C}_{k+1} = \mathbf{P}_{ps}\mathbf{c}_k + \mathbf{P}_{pu}\ddot{\mathbf{C}}_k \quad (3.33)$$

$$\dot{\mathbf{C}}_{k+1} = \mathbf{P}_{vs}\mathbf{c}_k + \mathbf{P}_{vu}\ddot{\mathbf{C}}_k \quad (3.34)$$

$$\ddot{\mathbf{C}}_{k+1} = \mathbf{P}_{as}\mathbf{c}_k + \mathbf{P}_{au}\ddot{\mathbf{C}}_k, \quad (3.35)$$

where the new matrices are given by (link)

$$\mathbf{P}_{ps} = \begin{pmatrix} 1 & T & T^2/2 \\ \vdots & & \vdots \\ 1 & nT & n^2T^2/2 \end{pmatrix}, \quad \mathbf{P}_{pu} = \begin{pmatrix} T^3/6 & \dots & 0 \\ \vdots & \ddots & \vdots \\ (1+3n+3n^2)T^2/6 & \dots & T^3/6 \end{pmatrix} \quad (3.36)$$

$$\mathbf{P}_{vs} = \begin{pmatrix} 0 & 1 & T \\ \vdots & & \vdots \\ 0 & 1 & nT \end{pmatrix}, \quad \mathbf{P}_{vu} = \begin{pmatrix} T^2/2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ (1+2n)/T^2/2 & \dots & T^2/2 \end{pmatrix} \quad (3.37)$$

$$\mathbf{P}_{as} = \begin{pmatrix} 0 & 0 & 1 \\ \vdots & & \vdots \\ 0 & 0 & 1 \end{pmatrix}, \quad \mathbf{P}_{au} = \begin{pmatrix} T & \dots & 0 \\ \vdots & \ddots & \vdots \\ T & \dots & T \end{pmatrix}. \quad (3.38)$$

If we now additionally consider the relation of the zero moment point and the center of mass, which we obtained earlier in equations 3.6 and 3.7, we can further relate the current center of mass state to the zero moment point on the preview horizon by (link)

$$\mathbf{Z}_{k+1} = \mathbf{C}_{k+1} - \frac{c_z}{g}\ddot{\mathbf{C}}_{k+1} \quad (3.39)$$

$$= \left(\mathbf{P}_{ps} - \frac{c_z}{g}\mathbf{P}_{as} \right) \mathbf{c}_k + \left(\mathbf{P}_{pu} - \frac{c_z}{g}\mathbf{P}_{au} \right) \ddot{\mathbf{C}}_k = \mathbf{P}_{zs}\mathbf{c}_k + \mathbf{P}_{zu}\ddot{\mathbf{C}}_k, \quad (3.40)$$

where the new matrices are given by (link)

$$\mathbf{P}_{zs} = \begin{pmatrix} 1 & T & T^2/2 - c_z/g \\ \vdots & & \vdots \\ 1 & nT & n^2T^2/2 - c_z/g \end{pmatrix} \quad (3.41)$$

$$\mathbf{P}_{zu} = \begin{pmatrix} T^3/6 - Tc_z/g & \dots & 0 \\ \vdots & \ddots & \vdots \\ (1+3n+3n^2)T^3/6 - Tc_z/g & \dots & T^3/6 - Tc_z/g \end{pmatrix}. \quad (3.42)$$

The expressions for the preview horizon now allow us to formulate an objective function that takes the robot's dynamic balance for future time steps into account, which in turn results in actions being taken that already take future predictions of the system's dynamics into account. The cost function will be described in the next section - The Objective Function.

The Objective Function

To create an objective function, as the one already outlined in section 3.1.2, we put together squared L^2 -norm objectives, which account for a desired reference center of mass velocity $\dot{\mathbf{C}}_{k+1}^{\text{ref}}$, a balanced foot step placement \mathbf{F}_{k+1} close to the zero moment point \mathbf{Z}_{k+1} , and a smooth motion for which the center of mass jerk $\ddot{\mathbf{C}}_{k+1}$ enters as a regularization. The objective function itself is similar to the one first introduced in [13], but additionally takes the center of mass' rotation around the z-axis into account, and it can be written down as follows

$$\min_{\mathbf{U}_k} \frac{\alpha}{2} \|\dot{\mathbf{C}}_{k+1}^x - \dot{\mathbf{C}}_{k+1}^{x,\text{ref}}\|_2^2 + \frac{\alpha}{2} \|\dot{\mathbf{C}}_{k+1}^y - \dot{\mathbf{C}}_{k+1}^{y,\text{ref}}\|_2^2 \quad (3.43)$$

$$\frac{\alpha}{2} \|\mathbf{E}_L \dot{\mathbf{F}}_{k+1}^{\theta,L} - \dot{\mathbf{C}}_{k+1}^{\theta,\text{ref}}\|_2^2 + \frac{\alpha}{2} \|\mathbf{E}_R \dot{\mathbf{F}}_{k+1}^{\theta,R} - \dot{\mathbf{C}}_{k+1}^{\theta,\text{ref}}\|_2^2 \quad (3.44)$$

$$\frac{\beta}{2} \|\mathbf{Z}_{k+1}^x - \mathbf{F}_{k+1}^x\|_2^2 + \frac{\beta}{2} \|\mathbf{Z}_{k+1}^y - \mathbf{F}_{k+1}^y\|_2^2 \quad (3.45)$$

$$\frac{\gamma}{2} \|\ddot{\mathbf{C}}_{k+1}^x\|_2^2 + \frac{\gamma}{2} \|\ddot{\mathbf{C}}_{k+1}^y\|_2^2 \quad (3.46)$$

Before we address the foot placement therein in more detail, we shortly want to highlight that the reference velocities are the commands, which enter the control loop from figure 3.2. We set them to be constant over the whole preview horizon and rotate them to the world frame by considering the robot's current orientation ([link](#)). When we consider that the foot cannot move once it is in contact with the ground, it becomes clear that the foot step placement must be very discrete in time. The number of steps N_f that we plan for in advance is simply given by NT/T_{step} , where we just divide the duration of the preview horizon NT by the time it takes to perform a step T_{step} . But as already shown in equation 3.40, and used in equation 3.45, we require balance on a finer timescale. We therefore project the foot placement $\tilde{\mathbf{F}}_k \in \mathbb{R}^{N_f \times 1}$ onto the temporal resolution of the center of mass' control variable by introducing the matrices \mathbf{v}_{k+1} , \mathbf{V}_{k+1} , and the current foot position f_k , which yields

$$\mathbf{F}_{k+1} = \mathbf{v}_{k+1} f_k + \mathbf{V}_{k+1} \tilde{\mathbf{F}}_k, \quad (3.47)$$

where \mathbf{v}_{k+1} is a $N \times 1$ matrix, and \mathbf{V}_{k+1} is a $N \times N_f$ matrix, and they are for example for $N_f = 3$ given by

$$\mathbf{v}_{k+1} = \begin{pmatrix} 1 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \mathbf{V}_{k+1} = \begin{pmatrix} 0 & 0 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \vdots & \vdots & \vdots \\ 0 & 0 & 1 \end{pmatrix} \quad (3.48)$$

As the robot moves, the matrices change, in that the entries of \mathbf{v}_{k+1} are being shifted upwards by one index for every time step T , while the new entries at the bottom are set to be zero. All entries of \mathbf{V}_{k+1} are also shifted upwards, while the bottom right entry is set to one, but further are all entries of \mathbf{V}_{k+1} are being shifted to the left once all entries of \mathbf{v}_{k+1} are zero ([link](#)). If all entries of \mathbf{v}_{k+1} are zero, then \mathbf{v}_{k+1} is replaced by the second column of \mathbf{V}_{k+1} . For example for $N_f = 2$ and $N = 6$ we have

$$(\mathbf{v}_{k+1} \mid \mathbf{V}_{k+1}) = \left(\begin{array}{c|ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{array} \right) \rightarrow \left(\begin{array}{c|ccc} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{array} \right) \rightarrow \left(\begin{array}{c|ccc} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right) \quad (3.49)$$

Now to ensure the rotation of the center of mass, we introduce equation 3.44 to the objective function. The matrices $\mathbf{E}_{L/R}$ therein ensure that only the foot, which is currently not touching the ground, is rotated, the rotational velocity of the center of mass itself is then obtained by averaging over the left and the right foot. Hence $\mathbf{E}_{L/R}$ take the form ([link](#))

$$\mathbf{E}_L = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & 0 & & \\ & & & & \ddots & & \\ & & & & & 0 & 1 \\ & & & & & & \ddots \\ & & & & & & & 1 \end{pmatrix}, \quad \mathbf{E}_R = \mathbf{1} - \mathbf{E}_L \quad (3.50)$$

The diagonal entries therein just take the same form as a single column of \mathbf{V}_{k+1} , all other entries are zero. If we now take equations 3.43-3.46, and replace $\dot{\mathbf{C}}_{k+1}^{x/y}$, as well as $\dot{\mathbf{F}}_{k+1}^{\theta,L/R}$, by equation 3.34, and insert equation 3.40 into the zero moment point on the preview horizon \mathbf{Z}_{k+1} , we obtain following relation ([link](#))

$$\min_{\mathbf{U}_k} \frac{1}{2} \mathbf{U}_k^T \mathbf{Q}_k \mathbf{U}_k + \mathbf{p}_k^T \mathbf{U}_k \quad (3.51)$$

$$\mathbf{U}_k = (\mathbf{U}_k^{xy} \quad \mathbf{U}_k^\theta)^T \quad (3.52)$$

$$\mathbf{U}_k^{xy} = (\ddot{\mathbf{C}}_k^x \quad \tilde{\mathbf{F}}_k^x \quad \ddot{\mathbf{C}}_k^y \quad \tilde{\mathbf{F}}_k^y)^T \quad (3.53)$$

$$\mathbf{U}_k^\theta = (\ddot{\mathbf{F}}_k^{\theta,L} \quad \ddot{\mathbf{F}}_k^{\theta,R})^T \quad (3.54)$$

$$\mathbf{Q}_k = \begin{pmatrix} \mathbf{Q}_k^x & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{Q}_k^y & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Q}_k^L & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{Q}_k^R \end{pmatrix} \quad (3.55)$$

$$\mathbf{Q}_k^x = \mathbf{Q}_k^y = \begin{pmatrix} \alpha \mathbf{P}_{vu}^T \mathbf{P}_{vu} + \beta \mathbf{P}_{zu}^T \mathbf{P}_{zu} + \gamma \mathbf{1} & -\beta \mathbf{P}_{zu}^T \mathbf{V}_{k+1} \\ -\beta \mathbf{V}_{k+1}^T \mathbf{P}_{zu} & \beta \mathbf{V}_{k+1}^T \mathbf{V}_{k+1} \end{pmatrix} \quad (3.56)$$

$$\mathbf{Q}_k^{L/R} = (\alpha \mathbf{P}_{vu}^T \mathbf{E}_{L/R}^T \mathbf{E}_{L/R} \mathbf{P}_{vu}) \quad (3.57)$$

$$\mathbf{p}_k = \begin{pmatrix} \mathbf{p}_k^x \\ \mathbf{p}_k^y \\ \mathbf{p}_k^L \\ \mathbf{p}_k^R \end{pmatrix} \quad (3.58)$$

$$\mathbf{p}_k^{x/y} = \begin{pmatrix} \alpha \mathbf{P}_{vu}^T (\mathbf{P}_{vs} \mathbf{c}_k^{x/y} - \dot{\mathbf{C}}_{k+1}^{x/y,\text{ref}}) + \beta \mathbf{P}_{zu}^T (\mathbf{P}_{zs} \mathbf{c}_k^{x/y} - \mathbf{v}_{k+1} f_k^{x/y}) \\ -\beta \mathbf{V}_{k+1}^T (\mathbf{P}_{zs} \mathbf{c}_k^{x/y} - \mathbf{v}_{k+1} f_k^{x/y}) \end{pmatrix} \quad (3.59)$$

$$\mathbf{p}_k^{L/R} = (\alpha \mathbf{P}_{vu}^T \mathbf{E}_{L/R}^T (\mathbf{E}_{L/R} \mathbf{P}_{vs} \mathbf{f}_k^{q,L/R} - \dot{\mathbf{C}}_{k+1}^{L/R,\text{ref}})) \quad (3.60)$$

where we evaluated all squared L^2 -norms via $\|\mathbf{a} - \mathbf{b}\|_2^2 = (\mathbf{a} - \mathbf{b})^T (\mathbf{a} - \mathbf{b})$, and ordered the terms correspondingly. All terms that do not depend on the control variable \mathbf{U}_k got discarded. Equation 3.51 now represents the canonical formulation of the minimization problem we were looking for in equation 3.18. The formulation itself minimizes our objective of keeping the zero moment point close to the center of the feet, but it does not assure that it never leaves the support polygon. Also does it not consider the kinematic feasibility of the solution. We will therefore have to introduce constraints on the free parameters of the optimal control problem in the next section - The Constraints.

The Constraints

The constraints we are going to deal with first, ensure dynamic balance in that they constrain the zero moment point to the support polygon of the feet. Secondly, we will explain the feasibility constraints, which force the feet positioning to a convex

hull that describes kinematically feasible motions. Finally, the constraints which restrict the feet's relative velocity, and the feet's relative orientation, as well as the ones, which allow obstacle avoidance, are introduced.

Balance Constraints To ensure that the zero moment point stays within the support polygon (figure 3.7), we set up a system of linear equations, of which each describes a line that connects the polygon's edges \mathbf{p}_i ([link](#)). A point \mathbf{x} lies beneath

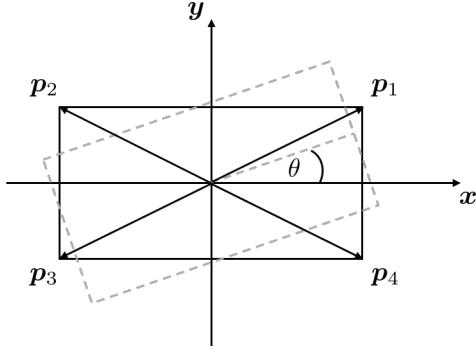


Figure 3.7: The foot's support polygon, which is described by the position vectors \mathbf{p}_i .

the line that connects the edges if

$$\mathbf{A}^{L/R}\mathbf{x} \leq \mathbf{B}^{L/R}, \quad (3.61)$$

where the linear equations are defined by $\mathbf{A}^{L/R} = (\mathbf{A}^{L/R,x} \ \mathbf{A}^{L/R,y}) \in \mathbb{R}^{N_{\text{edges}} \times 2}$, and $\mathbf{B}^{L/R} \in \mathbb{R}^{N_{\text{edges}} \times 1}$

$$\mathbf{A}^{L/R,x}[i] = p_i^{L/R,y} - p_{i+1}^{L/R,y} \quad (3.62)$$

$$\mathbf{A}^{L/R,y}[i] = p_{i+1}^{L/R,x} - p_i^{L/R,x} \quad (3.63)$$

$$\mathbf{B}^{L/R}[i] = (p_i^{L/R,y} - p_{i+1}^{L/R,y})p_{i+1}^{L/R,x} + (p_{i+1}^{L/R,x} - p_i^{L/R,x})p_{i+1}^{L/R,y} \quad (3.64)$$

Since we require the zero moment point to lie inside the support polygon, \mathbf{x} in equation 3.61 is replaced by $\mathbf{R}_z(f_k^\theta)(\mathbf{z}_k - \mathbf{f}_k)$, with $\mathbf{z}_k = (z_k^x \ z_k^y)^T$, and $\mathbf{f}_k = (f_k^x \ f_k^y)^T$. This expression describes the zero moment point with respect to the foot frame, where $\mathbf{R}_z(f_k^\theta) = \begin{pmatrix} \cos f_k^\theta & \sin f_k^\theta \\ -\sin f_k^\theta & \cos f_k^\theta \end{pmatrix}$ is an inverse rotation around the z-axis that adds non-linearities to the constraints. We can now extend the formalism to the whole preview horizon by utilizing equations 3.40 and 3.47. This leads to ([link](#))

$$\mathbf{D}_{k+1}(\mathbf{F}_{k+1}^\theta) \begin{pmatrix} \mathbf{Z}_{k+1}^x - \mathbf{F}_{k+1}^x \\ \mathbf{Z}_{k+1}^y - \mathbf{F}_{k+1}^y \end{pmatrix} \leq \mathbf{B}_{k+1} \quad (3.65)$$

$$\mathbf{D}_{k+1}(\mathbf{F}_{k+1}^\theta) \begin{pmatrix} \mathbf{P}_{zs} \mathbf{c}_k^x + \mathbf{P}_{zu} \dot{\mathbf{C}}_k^x - \mathbf{v}_{k+1} f_k^x - \mathbf{V}_{k+1} \tilde{\mathbf{F}}_{k+1}^x \\ \mathbf{P}_{zs} \mathbf{c}_k^y + \mathbf{P}_{zu} \dot{\mathbf{C}}_k^y - \mathbf{v}_{k+1} f_k^y - \mathbf{V}_{k+1} \tilde{\mathbf{F}}_{k+1}^y \end{pmatrix} \leq \mathbf{B}_{k+1} \quad (3.66)$$

where $\mathbf{D}_{k+1} \in \mathbb{R}^{N_{\text{edges}}N \times 2N}$ depends on $\mathbf{F}_{k+1}^\theta = \mathbf{P}_{ps}f_k^\theta + \mathbf{P}_{pu}\mathbf{F}_{k+1}^\theta$, and holds all the linear equations on the whole preview horizon

$$\mathbf{D}_{k+1} = \begin{pmatrix} \mathbf{A}^{L/R,x} \mathbf{R}_z(f_{k+1}^\theta) & \mathbf{0} & \mathbf{A}^{L/R,y} \mathbf{R}_z(f_{k+1}^\theta) & \mathbf{0} \\ & \ddots & & \ddots \\ \mathbf{0} & \mathbf{A}^{L/R,x} \mathbf{R}_z(f_{k+N}^\theta) & \mathbf{0} & \mathbf{A}^{L/R,y} \mathbf{R}_z(f_{k+N}^\theta) \end{pmatrix}, \quad (3.67)$$

and $\mathbf{B}_{k+1} = (\mathbf{B}^{L/R} \dots \mathbf{B}^{L/R})^T$. Whether the left foot's or the right foot's convex hull is chosen depends on the support foot at the respective preview interval k . Equation 3.66 can now be expressed in terms of the free variables \mathbf{U}_k by

$$\mathbf{D}_{k+1} \begin{pmatrix} \mathbf{P}_{zu} & -\mathbf{V}_{k+1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{P}_{zu} & -\mathbf{V}_{k+1} \end{pmatrix} \mathbf{U}_k^{xy} \leq \mathbf{B}_{k+1} + \mathbf{D}_{k+1} \begin{pmatrix} -\mathbf{P}_{zs} \mathbf{c}_k^x + \mathbf{v}_{k+1} f_k^x \\ -\mathbf{P}_{zs} \mathbf{c}_k^y + \mathbf{v}_{k+1} f_k^y \end{pmatrix} \quad (3.68)$$

$$\mathbf{A}_{\text{zmp},k}(\mathbf{U}_k^\theta) \mathbf{U}_k^{xy} \leq \overline{\mathbf{U}_{\text{zmp},k}}, \quad (3.69)$$

where $\overline{\mathbf{U}_{\text{zmp},k}}$ defines the upper bounds. The above derivation delivers a nice framework, which can be used to express the feasibility constraints as well.

Feasibility Constraints The feasibility constraints constrain the foot positioning to areas that the robot of interest can actually reach. We can therefore just reuse the concept of equation 3.61, and replace \mathbf{x} therein by the the difference of the next foot position $\tilde{\mathbf{F}}_k \in \mathcal{R}^{N_f \times 1}$, and the previous one $\tilde{\mathbf{F}}_{k-1} = \mathbf{S}_0 \tilde{\mathbf{F}}_k + \mathbf{S}_1 f_k$, where $\mathbf{S}_0 =$

$$\begin{pmatrix} 0 & 0 \\ \mathbf{I}_{N_f-1} & 0 \end{pmatrix} \in \mathcal{R}^{N_f \times N_f} \text{ simply shifts } \tilde{\mathbf{F}}_k \text{ down by one row, and } \mathbf{S}_1 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in \mathcal{R}^{N_f \times 1}.$$

Altogether we have ([link](#))

$$(\mathbf{A}_{k+1}^x \quad \mathbf{A}_{k+1}^y) \begin{pmatrix} \tilde{\mathbf{F}}_k^x - \mathbf{S}_0 \tilde{\mathbf{F}}_k^x - \mathbf{S}_1 f_k^x \\ \tilde{\mathbf{F}}_k^y - \mathbf{S}_0 \tilde{\mathbf{F}}_k^y - \mathbf{S}_1 f_k^y \end{pmatrix} \leq \mathbf{B}_{k+1} \quad (3.70)$$

$$(\mathbf{0} \quad \mathbf{A}_{k+1}^x(\mathbf{I}_{N_f} - \mathbf{S}_0) \quad \mathbf{0} \quad \mathbf{A}_{k+1}^y(\mathbf{I}_{N_f} - \mathbf{S}_0)) \mathbf{U}_k^{xy} \leq \mathbf{B}_{k+1} + \begin{pmatrix} \mathbf{S}_1 f_k^x \\ \mathbf{S}_1 f_k^y \end{pmatrix} \quad (3.71)$$

$$\mathbf{A}_{\text{foot},k}(\mathbf{U}_k^\theta) \mathbf{U}_k^{xy} \leq \overline{\mathbf{U}_{\text{foot},k}}, \quad (3.72)$$

where $\mathbf{A}_{k+1}^{x/y} = \begin{pmatrix} \mathbf{A}^{L/R,x/y} \mathbf{R}_z(f_k^\theta) & \mathbf{0} \\ & \ddots \\ \mathbf{0} & \mathbf{A}^{R/L,x/y} \mathbf{R}_z(f_{k+N_f}^\theta) \end{pmatrix} \in \mathbb{R}^{N_{\text{edges}}N \times N_f}$ has

alternating linear inequalities $\mathbf{A}^{L/R,x/y} \rightarrow \mathbf{A}^{R/L,x/y}$ that correspond on the support foot, and so does $\mathbf{B}_{k+1} \in \mathbb{R}^{N_{\text{edges}}N \times 1}$.

Relative Constraints By considering the maximum and minimum angle by which the feet are relatively oriented towards each other, we take hardware limits into account. Furthermore, the restriction of the maximally allowed relative angular velocity decreases that variation of acceleration before the foot landing [14]. The constraints can be formulated as follows ([link](#))

$$-\boldsymbol{\theta}_{\max} \leq \mathbf{F}_{k+1}^{L,\theta} - \mathbf{F}_{k+1}^{R,\theta} \leq \boldsymbol{\theta}_{\max} \quad (3.73)$$

$$-\dot{\boldsymbol{\theta}}_{\max} \leq \dot{\mathbf{F}}_{k+1}^{L,\theta} - \dot{\mathbf{F}}_{k+1}^{R,\theta} \leq \dot{\boldsymbol{\theta}}_{\max}, \quad (3.74)$$

which can be expressed in terms of the free variables with equations 3.33 and 3.34 to find

$$-\boldsymbol{\theta}_{\max} - \mathbf{P}_{ps}(f_k^L - f_k^R) \leq (\mathbf{P}_{pu} - \mathbf{P}_{pu}) \mathbf{U}_k^\theta \leq \boldsymbol{\theta}_{\max} - \mathbf{P}_{ps}(f_k^L - f_k^R) \quad (3.75)$$

$$\underline{\mathbf{A}}_{\text{ori},k} \mathbf{U}_k^\theta \leq \overline{\mathbf{U}}_{\text{ori},k} \quad (3.76)$$

$$-\dot{\boldsymbol{\theta}}_{\max} - \mathbf{P}_{vs}(f_k^L - f_k^R) \leq (\mathbf{P}_{vu} - \mathbf{P}_{vu}) \mathbf{U}_k^\theta \leq \dot{\boldsymbol{\theta}}_{\max} - \mathbf{P}_{vs}(f_k^L - f_k^R) \quad (3.77)$$

$$\underline{\mathbf{A}}_{\text{dori},k} \mathbf{U}_k^\theta \leq \overline{\mathbf{U}}_{\text{dori},k}. \quad (3.78)$$

Obstacle Constraints In contrast to similar methods like [15], we also include the avoidance of convex obstacles by requesting that the feet's positions must lie outside of circles $C = \{(p^x, p^y) \in \mathbb{R}^2 \mid (p^x - x_0)^2 + (p^y - y_0)^2 \leq R^2\}$, which define the obstacles, where x_0 , and y_0 define the obstacle's center in world coordinates. This can be formulated by the free variables by ([link](#))

$$(f_k^x - x_0)^2 + (f_k^y - y_0)^2 \geq (R + m)^2 \quad (3.79)$$

$$\mathbf{U}_k^{xy,T} (\mathbf{0} \quad \mathbf{I}_{N_f} \quad \mathbf{0} \quad \mathbf{I}_{N_f}) \mathbf{U}_k^{xy} - \quad (3.80)$$

$$(\mathbf{0} \quad (2x_0 \quad \dots \quad 2x_0) \quad \mathbf{0} \quad (2y_0 \quad \dots \quad 2y_0)) \mathbf{U}_k^{xy} \geq \begin{pmatrix} (R + m)^2 - x_0^2 - y_0^2 \\ \vdots \\ (R + m)^2 - x_0^2 - y_0^2 \end{pmatrix} \quad (3.81)$$

$$\mathbf{U}_k^{xy,T} \mathbf{H}_{\text{obs}} \mathbf{U}_k^{xy} + \mathbf{A}_{\text{obs}} \mathbf{U}_k^{xy} \geq \underline{\mathbf{U}}_{\text{obs}}, \quad (3.82)$$

which expresses lower bounds for the free variables. Given the constraints, of which some are nonlinear, we can apply a Gauss-Newton method to find the optimal solution. This also implies that we have to linearize all of the constraints that were presented within the previous paragraphs. The linearization is described in the next paragraph - The Gauss-Newton Formulation.

The Gauss-Newton Formulation

As outlined in equation 3.24, we can now linearize the quadratic problem of equation 3.51. This leads to ([link](#))

$$\min_{\Delta \mathbf{U}_k} \frac{1}{2} \Delta \mathbf{U}_k^T \mathbf{Q}_k \Delta \mathbf{U}_k + \tilde{\mathbf{p}}_k^T \Delta \mathbf{U}_k \quad (3.83)$$

$$\text{subject to: } \tilde{\mathbf{U}}_k \leq \tilde{\mathbf{A}}_k \Delta \mathbf{U}_k \leq \overline{\tilde{\mathbf{U}}}_k, \quad (3.84)$$

where

$$\tilde{\mathbf{p}}_k = \begin{pmatrix} \frac{1}{2}\mathbf{U}_{k-1}^{xy,T} \begin{pmatrix} \mathbf{Q}_k^x & \mathbf{0} \\ \mathbf{0} & \mathbf{Q}_k^x \end{pmatrix} + \begin{pmatrix} \mathbf{p}_k^x \\ \mathbf{p}_k^y \end{pmatrix} \\ \frac{1}{2}\mathbf{U}_{k-1}^{\theta,T} \begin{pmatrix} \mathbf{Q}_k^L & \mathbf{0} \\ \mathbf{0} & \mathbf{Q}_k^R \end{pmatrix} + \begin{pmatrix} \mathbf{p}_k^L \\ \mathbf{p}_k^R \end{pmatrix} \end{pmatrix} \quad (3.85)$$

$$\tilde{\mathbf{A}}_k = \begin{pmatrix} \mathbf{A}_{\text{zmp},k}(\mathbf{U}_{k-1}^\theta) & \nabla_{\mathbf{U}^\theta}^T \mathbf{A}_{\text{zmp},k}|_{\mathbf{U}_{k-1}^\theta} \mathbf{U}_{k-1}^{xy} \\ \mathbf{A}_{\text{foot},k}(\mathbf{U}_{k-1}^\theta) & \nabla_{\mathbf{U}^\theta}^T \mathbf{A}_{\text{foot},k}|_{\mathbf{U}_{k-1}^\theta} \mathbf{U}_{k-1}^{xy} \\ \mathbf{H}_{\text{obs}} \mathbf{U}_{k-1}^{xy} + \mathbf{A}_{\text{obs},k} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{\text{ori},k} \\ \mathbf{0} & \mathbf{A}_{\text{dori},k} \end{pmatrix} \quad (3.86)$$

$$\underline{\tilde{\mathbf{U}}}_k = \begin{pmatrix} -\infty \\ -\infty \\ \underline{\mathbf{U}}_{\text{obs}} \\ \underline{\mathbf{U}}_{\text{ori}} \\ \underline{\mathbf{U}}_{\text{dori}} \end{pmatrix} - \mathbf{h}_{k-1}, \quad \overline{\tilde{\mathbf{U}}}_k = \begin{pmatrix} \overline{\mathbf{U}}_{\text{zmp},k} \\ \overline{\mathbf{U}}_{\text{foot},k} \\ \infty \\ \overline{\mathbf{U}}_{\text{ori},k} \\ \overline{\mathbf{U}}_{\text{dori},k} \end{pmatrix} - \mathbf{h}_{k-1} \quad (3.87)$$

$$\mathbf{h}_{k-1} = \begin{pmatrix} \mathbf{A}_{\text{zmp},k}(\mathbf{U}_{k-1}^\theta) \mathbf{U}_{k-1}^{xy} \\ \mathbf{A}_{\text{foot},k}(\mathbf{U}_{k-1}^\theta) \mathbf{U}_{k-1}^{xy} \\ \mathbf{U}_{k-1}^{xy,T} \mathbf{H}_{\text{obs}} \mathbf{U}_{k-1}^{xy} + \mathbf{A}_{\text{obs}} \mathbf{U}_{k-1}^{xy} \\ \mathbf{A}_{\text{ori},k} \mathbf{U}_{k-1}^\theta \\ \mathbf{A}_{\text{dori},k} \mathbf{U}_{k-1}^\theta \end{pmatrix}. \quad (3.88)$$

It follows that we only need to compute the gradient of $\mathbf{A}_{\text{zmp},k}$, and $\mathbf{A}_{\text{foot},k}$, with respect to the free variable \mathbf{U}^θ , which can analytically be done by deriving the rotation matrices' gradient in equations 3.67, and 3.72. The solution to this optimal control problem yields the update $\Delta \mathbf{U}_k$ to the current iterate $\mathbf{U}_k = \mathbf{U}_{k-1} + \Delta \mathbf{U}_k$ ([link](#)) under quadratic convergence, given that we are sufficiently close to a solution, as discussed earlier. The resulting positioning for the feet needs then to be interpolated, which will be explained in the following section - Interpolating Trajectories.

3.1.3 Interpolating Trajectories

As already shortly depicted in figure 3.2, we need to interpolate the trajectories that we obtain from the nonlinear model predictive control. This especially holds true for the feet, since the computed results do only consider the pendulums dynamic balance with respect to the x-, and the y-position, but not with respect to a robot that has to lift its feet along the z-axis. Furthermore, the feet's movement shall be executed such that they stop moving when they are about to touch the ground. This constraint, and others, will be achieved by interpolating the feet trajectories with polynomials. In order to then match the center of mass trajectory's temporal resolution with the feet trajectories', we will upscale it under the already well known assumption of a linear inverted pendulum. The resulting trajectories are shown in figure 3.8, and will further be explained in the following.

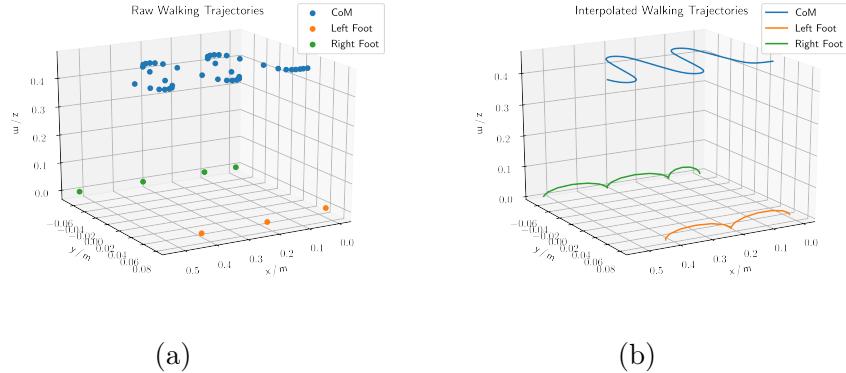


Figure 3.8: Uninterpolated trajectories (a), as obtained from the nonlinear model predictive control, and interpolated trajectories (b) for the feet and the center of mass.

Interpolating the Feet Trajectories

Any trajectory can in principal be approximated by a polynomial function. For our purposes, we want to approximate positions p as they evolve over time t , and further obtain the corresponding velocities \dot{p} and accelerations \ddot{p} (equations 3.89 - 3.91).

$$p(t) = \sum_{i=0}^N a_i t^i \quad (3.89)$$

$$\dot{p}(t) = \sum_{i=1}^N i a_i t^{(i-1)} \quad (3.90)$$

$$\ddot{p}(t) = \sum_{i=2}^N i(i-1) a_i t^{(i-2)} \quad (3.91)$$

The coefficients a_i of the polynomials can be chosen such that certain boundary conditions \mathbf{b} are satisfied. For the lift-off and the drop-down of the robot's feet, these boundary conditions must satisfy a zero initial velocity \dot{z}_{init} and a zero end velocity \dot{z}_{end} , as well as a zero initial height z_{init} and a zero end height z_{end} , and a maximum step height $z_{T/2}$ in between, or else they will hit the ground in an unbalanced way. These conditions are listed below, where each height $z(t)$ and each velocity $\dot{z}(t)$ is written in terms of a polynomial, just as in equations 3.89 and 3.90,

respectively.

$$z(t = 0) = z_{\text{init}} = 0 \quad (3.92)$$

$$\dot{z}(t = 0) = \dot{z}_{\text{init}} = 0 \quad (3.93)$$

$$z(t = \frac{T}{2}) = z_{T/2} \quad (3.94)$$

$$z(t = T) = z_{\text{end}} = 0 \quad (3.95)$$

$$\dot{z}(t = T) = \dot{z}_{\text{end}} = 0 \quad (3.96)$$

To satisfy 5 boundary conditions, it is required to have a polynomial of 4th order with 5 coefficients $a_{z,i}$ in total. In matrix formulation we can express equations 3.92 - 3.96 as follows

$$\mathbf{M}_z \mathbf{a}_z = \mathbf{b}_z \quad (3.97)$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & \left(\frac{T}{2}\right) & \left(\frac{T}{2}\right)^2 & \left(\frac{T}{2}\right)^3 & \left(\frac{T}{2}\right)^4 \\ 1 & T & T^2 & T^3 & T^4 \\ 0 & 1 & 2T & 3T^2 & 4T^3 \end{pmatrix} \begin{pmatrix} a_{z,0} \\ a_{z,1} \\ a_{z,2} \\ a_{z,3} \\ a_{z,4} \end{pmatrix} = \begin{pmatrix} z_{\text{init}} \\ \dot{z}_{\text{init}} \\ z_{T/2} \\ z_{\text{end}} \\ \dot{z}_{\text{end}} \end{pmatrix}. \quad (3.98)$$

Inversion then yields

$$\mathbf{a}_z = \mathbf{M}_z^{-1} \mathbf{b}_z \quad (3.99)$$

$$\begin{pmatrix} a_{z,0} \\ a_{z,1} \\ a_{z,2} \\ a_{z,3} \\ a_{z,4} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ -\frac{11}{T^2} & -\frac{4}{T} & \frac{16}{T^2} & -\frac{5}{T^2} & \frac{1}{T} \\ \frac{18}{T^3} & \frac{5}{T^2} & -\frac{32}{T^3} & \frac{14}{T^3} & -\frac{3}{T^2} \\ -\frac{8}{T^4} & -\frac{2}{T^3} & \frac{16}{T^4} & -\frac{8}{T^4} & \frac{2}{T^3} \end{pmatrix} \begin{pmatrix} z_{\text{init}} \\ \dot{z}_{\text{init}} \\ z_{T/2} \\ z_{\text{end}} \\ \dot{z}_{\text{end}} \end{pmatrix}. \quad (3.100)$$

The obtained coefficients a_i are then used to compute the height of each foot during a single support phase ([link](#)). The maximum step height $z_{T/2}$ ([link](#)), and the single support time T ([link](#)), which is the step time minus the double support time, can be set in the configurations file. For the x-, and the y-positions of the feet, we can define boundary conditions in a similar fashion. In contrary to the computation of the z-position, the x-, and the y-position interpolation of the feet allows for feedback. Therefore, we require additional constraints that satisfy the accelerations as follows

$$x(t = 0) = x_{\text{init}} \quad (3.101)$$

$$\dot{x}(t = 0) = \dot{x}_{\text{init}} \quad (3.102)$$

$$\ddot{x}(t = 0) = \ddot{x}_{\text{init}} \quad (3.103)$$

$$x(t = T) = x_{\text{end}} \quad (3.104)$$

$$\dot{x}(t = T) = \dot{x}_{\text{end}} \quad (3.105)$$

$$\ddot{x}(t = T) = \ddot{x}_{\text{end}}. \quad (3.106)$$

Again, we can rewrite equations 3.101 - 3.106 in matrix formulation

$$\mathbf{M}_x \mathbf{a}_x = \mathbf{b}_x \quad (3.107)$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 1 & T & T^2 & T^3 & T^4 & T^5 \\ 0 & 1 & 2T & 3T^2 & 4T^3 & 5T^4 \\ 0 & 0 & 2 & 6T & 12T^2 & 20T^3 \end{pmatrix} \begin{pmatrix} a_{x,0} \\ a_{x,1} \\ a_{x,2} \\ a_{x,3} \\ a_{x,4} \\ a_{x,5} \end{pmatrix} = \begin{pmatrix} x_{\text{init}} \\ \dot{x}_{\text{init}} \\ \ddot{x}_{\text{init}} \\ x_{\text{end}} \\ \dot{x}_{\text{end}} \\ \ddot{x}_{\text{end}} \end{pmatrix}, \quad (3.108)$$

and inversion yields the polynomial's coefficients $a_{x,i}$

$$\mathbf{a}_x = \mathbf{M}_x^{-1} \mathbf{b}_x \quad (3.109)$$

$$\begin{pmatrix} a_{x,0} \\ a_{x,1} \\ a_{x,2} \\ a_{x,3} \\ a_{x,4} \\ a_{x,5} \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ -\frac{20}{T^3} & -\frac{12}{T^2} & -\frac{3}{T} & \frac{20}{T^3} & -\frac{8}{T^2} & \frac{1}{T} \\ \frac{30}{T^4} & \frac{16}{T^3} & \frac{3}{T^2} & -\frac{30}{T^4} & \frac{14}{T^3} & -\frac{2}{T^2} \\ -\frac{12}{T^5} & -\frac{6}{T^4} & -\frac{1}{T^3} & \frac{12}{T^5} & -\frac{6}{T^4} & \frac{1}{T^3} \end{pmatrix} \begin{pmatrix} x_{\text{init}} \\ \dot{x}_{\text{init}} \\ \ddot{x}_{\text{init}} \\ x_{\text{end}} \\ \dot{x}_{\text{end}} \\ \ddot{x}_{\text{end}} \end{pmatrix}. \quad (3.110)$$

The exact same formalism is used to interpolate the foot's y-position during single support phase ([link](#)). In contrast to the interpolation of the feet's positions, the center of mass positions will be extrapolated under the introduced assumption of a linear inverted pendulum. The method will be shortly explained in the following paragraph - Interpolating the Center of Mass Trajectories.

Interpolating the Center of Mass Trajectories

The center of mass trajectories can now simply be adjusted to the temporal resolution of the feet trajectories by applying the linear time stepping scheme from equation 3.29 with an adjusted temporal resolution T . An iterative application ([link](#)) of these matrices then yields the desired interpolation. The only requirement left to get our robot to walk is now the transformation of trajectories in Cartesian space to trajectories within the joint space, which will be resolved in the following chapter - Kinematics.

3.1.4 Kinematics

As already shortly depicted in figure 3.2, it is required to compute the robot's kinematics in order to switch between the Cartesian space and the joint space. For our purposes we need to find joint angles that satisfy the center of mass and the feet trajectories. While it is rather straight forward to compute the forward kinematics, as it is just a concatenation of spatial transformations, the inverse kinematics require an optimization, since there usually is no unique solution.

Forward Kinematics

The goal in forward kinematics is to transform joint angles \mathbf{q} into Cartesian coordinates \mathbf{x} such that $\mathbf{FK}(\mathbf{q}) = \mathbf{x}$. For this thesis, it enables us to feedback the robot's center of mass position, given the current state of it. In terms of homogeneous coordinates, one can express the forward kinematics as a series of spatial transformations, which include translations \mathbf{t} and rotations \mathbf{R}

$$\mathbf{x}_0 = \prod_{i=N}^0 \mathbf{H}_i^{i-1}(q) \mathbf{x}_N, \quad (3.111)$$

where $\mathbf{x}_i = [x \ y \ z \ 1]^T$, and $\mathbf{H}_i^{i-1} = \begin{pmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix}$ are homogeneous coordinates and transformations from frame i to frame $i - 1$, respectively.

Inverse Kinematics

In inverse kinematics, we aim at finding joint angles \mathbf{q} that satisfy certain positional and orientational constraints, which we set for the kinematic chain of interest. For this work, we require the robot's feet and center of mass to be at a position that we obtain from the nonlinear model predictive control. That is, we minimize the sum of squared differences $S(\mathbf{q}, \Delta\mathbf{q})$ between desired positions and orientations a_i and the linearization of the forward kinematics around the robot's current pose \mathbf{q} (equation 3.112), to find an incremental update $\Delta\mathbf{q}$.

$$S(\mathbf{q}, \Delta\mathbf{q}) = \sum_{i=0}^m \left[a_i - \mathbf{FK}_i(\mathbf{q}) - \frac{\partial \mathbf{FK}_i(\mathbf{q})}{\partial \mathbf{q}} \Delta\mathbf{q} \right]^2 \quad (3.112)$$

A damped version of this minimization yields the Levenberg-Marquardt algorithm [16], with which one can iteratively update the pose \mathbf{q} by $\Delta\mathbf{q}$, so to satisfy the posed requirements (equation 3.113).

$$\Delta\mathbf{q} = (\mathbf{J}^T \mathbf{J} + \lambda \mathbf{I})^{-1} \mathbf{J}^T (\mathbf{y} - \mathbf{FK}(\mathbf{q})) \quad (3.113)$$

Therein, the rows of system's Jacobian \mathbf{J} is governed by $\mathbf{J}_i = \frac{\partial \mathbf{FK}_i(\mathbf{q})}{\partial \mathbf{q}}$.

3.2 Machine Learning

Machine learning methods do play a major role for autonomous navigation of robots, and whilst most recent approaches mainly dealt with tree search methods in 3D point-clouds, we aim at utilizing neural networks for solving the task at hand, since it enables us to combine spatial, semantic, and temporal understanding into one approach. Within this chapter, we will therefore explain the required fundamentals on neural networks in section 3.2.1, then cover two possible methods for training a

neural network, one of which is supervised 3.2.2, whereas the second method is based on reinforcement learning 3.2.3, and finally explain image processing techniques in section 3.2.4, which allow us to extract depth maps from stereo images, so to help the neural networks understand the seen content. The goal here clearly is to introduce a method that is biologically inspired, in that it works directly in the image domain, which is very similar to how humans observe their environment. Therefore, we will shortly explain the biological similarities to a human brain within the next section - Neural Networks.

3.2.1 Neural Networks

Neural networks are inspired by the brain's structure, and as we will see later in this chapter, their building blocks can be thought of as neurons (figure 3.9). And

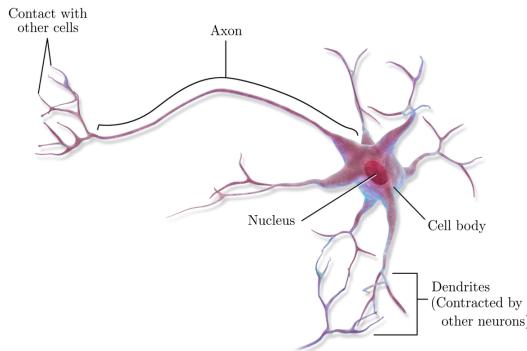


Figure 3.9: Biological neuron, which connects its cell body to dendrites of surrounding neurons via an axon. [17]

although neural networks have gained early attention in research, they have only recently become powerful for their implementation on graphic processing units (GPUs) [18]. This was caused by their mathematical description that is linear and can be parallelized very well. Running neural nets on GPUs on the other hand consumes a lot of energy, which stays in contrast to biological neurons, which communicate by brief energy efficient spikes [19]. And while there are around 100 billion neurons in the human brain, currently huge neural networks have around a factor of 10000 less [20]. Not only is the number of neurons in a neural net comparably small, but also is their complexity way below that of a biological neuron. To tackle this discrepancy, and to learn complicated tasks, it is therefore required to introduce activation functions for neural networks, such as the rectifying linear unit [21]. This enables neural networks to be used on a variety of problems, but they currently lack in transferring knowledge between different domains, and tend to over-fit certain tasks. There exist methods to deal with this tendency, such as max-pooling [22] or dropout [23] layers, which efficiently just reduce the number of neurons within a neural net. To build a good understanding of neural networks, we will introduce different architectures

in the following that will be used throughout this thesis, and we will start with the simplest in the next paragraph - Fully Connected Neural Network.

Fully Connected Neural Network

The biologically inspired perceptron model [24] lay the foundation for neural networks and it got extended pretty soon to the multi-layer perceptron model [25], which is known today as the fully connected neural network. Due to its similarity to the brain's structure, it is often depicted as in figure 3.10, where each orange circle represents a neuron that is connected to its surroundings via simple weights w_{ij} , as neurons inside the brain are by synapses. Mathematically speaking, the feed



Figure 3.10: Fully connected neural network with three inputs and four outputs. Each orange circle represents what is often referred to as neuron, while the black lines indicate the connections between each neuron.

forward process can be described as a simple matrix multiplication with all weights \mathbf{W} , where the input \mathbf{x} gets converted to the output \mathbf{y} via

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \quad (3.114)$$

Therein, the bias \mathbf{b} can be understood as a shift of isolines that are introduced by hyperplanes. These hyperplanes are learned and expressed by the layer's weights w_{ij} (figure 3.11). The output \mathbf{y} , is then further passed through an activation function f , and therefore can be compared to the action potential inside a neuron, as it determines the amount by which the next neuron gets excited. This activation function can be anything from a simple step function for classification to a linear function for regression. In practice there are some activation functions that have shown to be of particular use and we will introduce them later in this chapter.

Convolutional Neural Network

The concept of convolutional neural networks was first inspired by biological structures inside the visual cortex of the human brain. It was introduced as neocognitron [26], and soon after termed convolutional neural network for its mathematical properties, in that it equals convolutions. Figure 3.12 shows how an input \mathbf{x} is fed forward through the network architecture across two layers. The operation is again visualized by utilizing orange circles, which may be referred to as neurons. These neurons are connected by weights $k_{ij}^{l;n}$, which together constitute the kernel $\mathbf{k}^{l;n}$ of

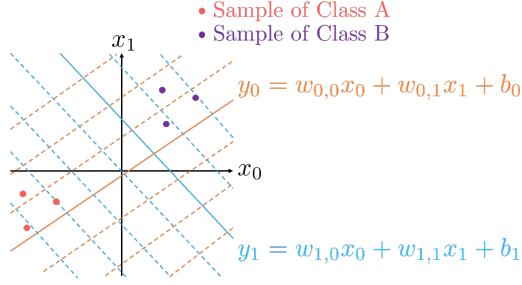


Figure 3.11: Simple interpretation of a fully connected neural network with one layer that takes $\mathbf{x} = (x_0 \ x_1)^T$ as input. The dotted lines are isolines to the hyperplane, which showcase the effect of the bias \mathbf{b} .

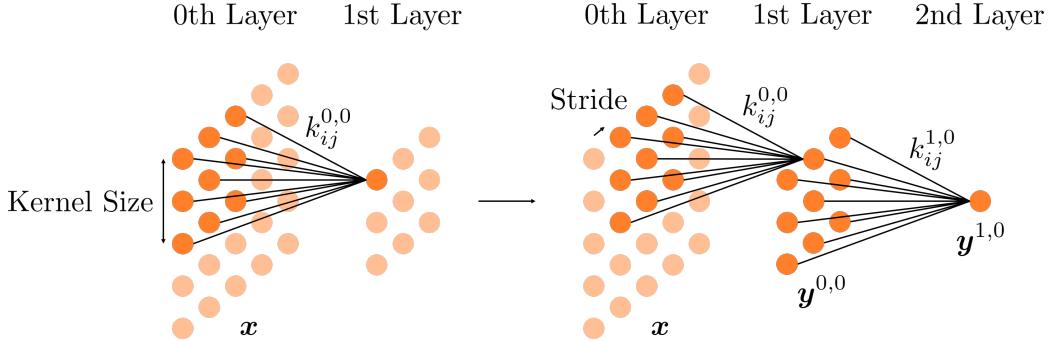


Figure 3.12: Convolutional neural network with a total of three layers of which one is the input layer. For visualization, the kernel size is set to be three, and the stride is set to be one.

each convolution. Therein, l stands for the current layer, and n indexes the kernel within a layer, as there may in principle be many different kernels for a single layer. Mathematically speaking, we can formulate the process as follows

$$\mathbf{y}^{0,0} = f(\mathbf{x} * \mathbf{k}^{0,0}) \quad (3.115)$$

$$\mathbf{y}^{1,0} = f(\mathbf{y}^{0,0} * \mathbf{k}^{1,0}) \quad (3.116)$$

One thing to notice is that the deeper we go, meaning the more layers we have, the more of the initial input contributes to the current activation. This can be seen in figure 3.12, where each neuron within the first layer only sees a kernel size sized snippet of the original input \mathbf{x} , whereas a neuron within the second layer already sees all of it. This intuitive understanding of convolutional neural networks is backed by visualizations of the highest activity neuron's gradient with respect to the input, where the gradient itself equals the transposed convolution [27]. Figure 3.13 shows transposed convolutions for a classification network that was trained on ImageNet [28]. It can be seen that while superficial layers learn to understand edges, deeper layers grab more complex spatial correlations within images, such as car wheels within the third layer. Therefore, convolutional neural networks are particularly well suited for understanding spatial correlations, and while recent advancements

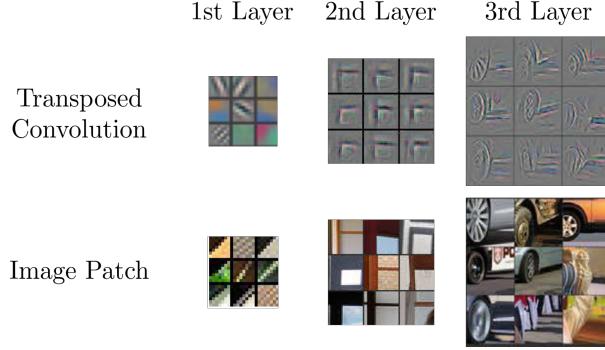


Figure 3.13: Analysis of neurons with highest activation, corresponding to a subset of ImageNet. For the first layer, the kernel itself is shown, and image patches at the kernel’s scale. For the second and third layer, a single neuron with highest activation is upscaled by transposed convolutions to the first layer’s feature map. Images taken from [29].

also propose the promising use of convolutional neural networks for time series analysis [30], a simpler approach are long short-term memory units, which will be explained in the next section - Long Short-Term Memory.

Long Short-Term Memory

Long short-term memory units were introduced to overcome the vanishing and exploding gradient problem for recurrent neural networks in time series analysis [31]. That is the gradient tends to diverge exponentially over the course of the backward pass towards earlier inputs, resulting in intractable updates for the weights of the neural network. This issue got solved by adding a constant recurrent self-connection that allows for constant error propagation through the network, which got further refined by replacing the constant self-connection with a forget gate [32]. The inner workings of a long short-term memory unit is shown in figure 3.14. The underlying

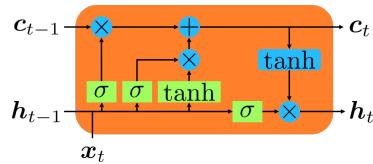


Figure 3.14: Long short-term memory unit with cell states \mathbf{c}_i , hidden states \mathbf{h}_i , input \mathbf{x}_t , and activation functions σ/\tanh , as well as addition + and multiplication \times operators.

mathematical operations can therein be expressed as follows

$$\mathbf{i}_t = \sigma(\mathbf{W}_{ii}\mathbf{x}_t + \mathbf{b}_{ii} + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_{hi}) \quad (3.117)$$

$$\mathbf{f}_t = \sigma(\mathbf{W}_{if}\mathbf{x}_t + \mathbf{b}_{if} + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_{hf}) \quad (3.118)$$

$$\mathbf{g}_t = \tanh(\mathbf{W}_{ig}\mathbf{x}_t + \mathbf{b}_{ig} + \mathbf{W}_{hg}\mathbf{h}_{t-1} + \mathbf{b}_{hg}) \quad (3.119)$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_{io}\mathbf{x}_t + \mathbf{b}_{io} + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_{ho}) \quad (3.120)$$

$$\mathbf{c}_t = \mathbf{f}_t \cdot c_{t-1} + \mathbf{i}_t \cdot \mathbf{g}_t \quad (3.121)$$

$$\mathbf{h}_t = \mathbf{o}_t \cdot \tanh(\mathbf{c}_t), \quad (3.122)$$

where \cdot is an element-wise multiplication. The sigmoid function σ , which ranges from 0 to 1, assures that the input gate \mathbf{i}_t , the forget gate \mathbf{f}_t , and the output gate \mathbf{o}_t let only pass values of interest into, and out of the cell. Multiple long short-term memory units can then be linked for time series analysis as shown in figure 3.15.

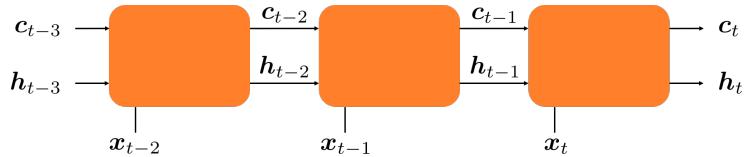


Figure 3.15: Chain of long short-term memory units for temporal understanding of the input sequence \mathbf{x}_i .

Backpropagation

The currently most popular way to train a neural network is backpropagation, which got first introduced in [33]. It has no biological equivalent but poses an effective way to optimize a huge amount of parameters. Newer methods use evolutionary algorithms and treat network parameters as population that develops over time [34], but we wont consider them further. The reason for why backpropagation works so well to optimize neural networks, is the simplicity of the mathematical operations that make them up. Not only do the activation functions have an analytical derivative, but further can we just apply the chain rule multiple times on the loss function, so to find the gradient for every network parameter. Suppose we have the loss L , then the derivative with respect to the weights \mathbf{W}_l of layer l , is just given as

$$\frac{\partial L}{\partial \mathbf{W}_l} = \boldsymbol{\delta}_l \mathbf{x}_{l-1}^T, \quad (3.123)$$

where for the last layer N , and all previous layers l , we have

$$\boldsymbol{\delta}_N = \frac{\partial L}{\partial \mathbf{x}_N} \cdot f'_N(\mathbf{W}_N \mathbf{x}_{N-1}) \quad (3.124)$$

$$\boldsymbol{\delta}_l = \mathbf{W}_{l+1}^T \boldsymbol{\delta}_{l+1} \cdot f'_l(\mathbf{W}_l \mathbf{x}_{l-1}), \quad (3.125)$$

with f' being the derivative of the corresponding activation function. The update for the next time-step $t + 1$ is then performed according to an optimizer specific learning rate $\alpha_{\mathbf{W}_l^t}$ as follows

$$\mathbf{W}_l^{t+1} = \mathbf{W}_l^t - \alpha_{\mathbf{W}_l^t} \cdot \frac{\partial L}{\partial \mathbf{W}_l}, \quad (3.126)$$

where \cdot is an element-wise multiplication.

3.2.2 Behavioral Cloning

Behavioral cloning in itself is not always related to machine learning, but poses one possible way of training a neural net in a supervised manner. The presented concept is easy to understand and got inspired by [35], where it was used for self-driving cars, and since having a car drive along the road is easier to achieve than having a robot walk around an environment, we will deal with the additional details later to focus on the main points for now. The proposed method utilizes the control loop, which was already introduced in figure 3.1. In order to then replace the human user by an artificial agent, we have a human user perform a desired behavior, and copy it. The required extended control loop is shown in figure 3.16. It simply

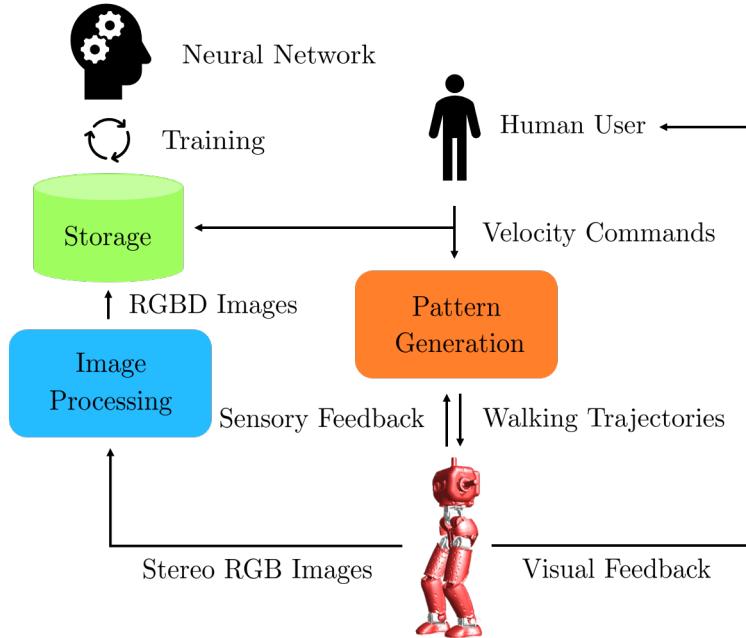


Figure 3.16: Pipeline for behavioral cloning. The neural network is trained on stored RGBD images, and corresponding velocity commands that are correlated by a timestamp

takes the velocity commands from the human user, and stores it alongside RGBD images with a corresponding timestamp to some storage, where the RGBD images

are obtained from stereo RGB images by an image processing step that is explained in section 3.2.4. The timestamp allows to correlate seen images to desired velocities afterwards, which in turn enables an artificial agent to train on the stored data. For our purposes, the artificial agent is a neural network. An appropriately chosen network architecture will then enable us to learn the taught behavior and ultimately lets us replace the human user. This procedure relies on prior knowledge to achieve certain tasks, namely the stored data. It is therefore extremely important to assure that the sampled data, from which we want to learn a task, does not introduce any unwanted bias, that is, we need to take care of the distribution from which we sample in the first place. In principle, it is possible to learn any arbitrary behavior with this technique, but this requires not only good data, but also a vast amount of it. There are other algorithms that explore the state space on their own, and for which we could for example use the taught behavior as prior as well. These algorithms belong to the class of reinforcement learning methods, and we will have a look at a particular one in the next section.

3.2.3 Reinforcement Learning

The goal in reinforcement learning is not only to learn actions a_t at time-step t , given a state s_t , like it is in behavioral cloning, but further to explore actions and states. This is usually performed as shown in figure 3.17, where an agent interacts with an environment to receive a reward r_t , and also changes the state as cause of its action. Therein, the actions a_t are sampled from a policy $a_t \sim \pi_\theta(a_t|s_t)$ that depends on parameters θ , which for our case are simply the weights of a neural network. The difficulty in optimizing the policy π_θ , is to have an agent to discard

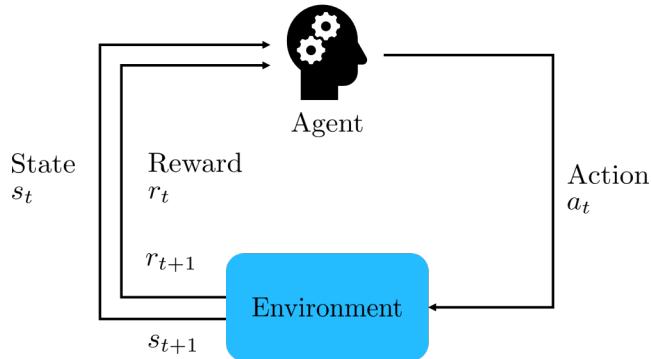


Figure 3.17: Reinforcement learning setup. As the agent interacts with the environment, the state of the environment changes.

immediate rewards over future expected rewards. For discrete action spaces, this got well solved by deep Q-learning [36]. Different approaches for continuous action spaces like trust region policy optimization [37] are rather complicated. The, to this date, most elegant way of solving continuous control problems in a reinforcement learning setup, is proximal policy optimization [38], and we will elaborate on it in

the following. Gradient policy methods, such as proximal policy optimization, try to update the policy π_θ , such that the expected total future reward $\mathbb{E}[\sum_{t=0}^{\infty} r_t]$ is maximized. For the incremental update, it is therefore required to find the gradient of this expression

$$\nabla_\theta \mathbb{E}_{a_t \sim \pi_\theta} \left[\sum_{t=0}^{\infty} r_t \right] = \mathbb{E}_{a_t \sim \pi_\theta} \left[\sum_{t=0}^{\infty} \psi_t \nabla_\theta \log \pi_\theta(a_t | s_t) \right], \quad (3.127)$$

where ψ_t can take many forms. Since the gradient is just an estimate, as it is computed from samples being taken from the reinforcement learning environment, it usually suffers from variance and bias. As shown in [39], we can trade-off variance for bias and the other way around, by replacing ψ_t for the general advantage estimate \hat{A}_t^{GAE} as follows

$$\hat{A}_t^{\text{GAE}(\gamma, \lambda)} = \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l}^V, \quad (3.128)$$

where $\delta_{t,l}^V = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the temporal difference [40], and V the value function, which is given by a critic network. The critic's goal then is to have the gradient estimate steer the acting policy network towards actions that maximize the reward. A huge problem therein is that the policy may diverge and that policies may be discarded on the basis of a gradient estimate with a high variance. This is prohibited in proximal policy optimization by clipping the general advantage estimate, and therefore the gradient, under the following objective

$$L^{\text{CLIP}} = \min(\rho_t(\theta) \hat{A}_t, \text{clip}(\rho_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t), \quad (3.129)$$

where $\rho_t(\theta) = \frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)}$ is the probability ratio of the old and the new policy. The loss is shown in figure 3.18, and it assures for a positive advantage estimate that the gradient does not diverge towards actions that are way more likely under the new policy, than they have been for the old policy. Also, for a negative advantage

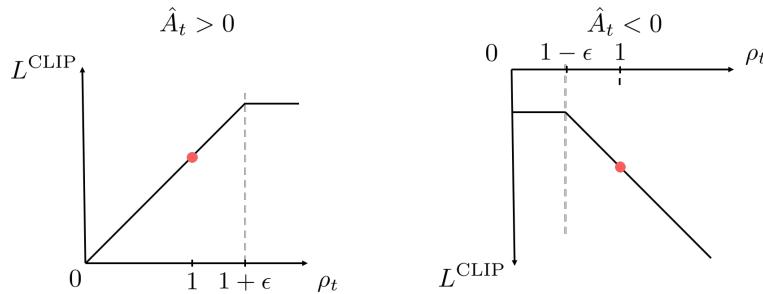


Figure 3.18: Policy gradient clipping in proximal policy optimization for positive and negative advantage estimates \hat{A}_t .

estimate, it assures that the gradient points away from these policies. The total

objective $L_t^{\text{CLIP+VF+S}}$ of proximal policy optimization is further extended by an entropy term S that results in exploration, and the critic's loss L^{VF} , such that it can steer the gradient (equation 3.130).

$$L_t^{\text{CLIP+VF+S}} = \mathbb{E} [L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}}(\theta) + c_2 S[\pi_\theta](s_t)] \quad (3.130)$$

The critic's loss therein is the squared-error of the value function estimate and the explored values $(V_\theta(s_t) - V_t^{\text{target}})^2$. The algorithm then runs as follows

```

for iteration = 1,2,... do
    for runs=1,2,... do
        | Run policy  $\pi_{\theta_{\text{old}}}$  in environment for  $T$  timesteps
        | Compute advantage estimates  $\hat{A}_1^{\text{GAE}}, \dots, \hat{A}_T^{\text{GAE}}$ 
    end
    | Optimize surrogate  $L$  wrt  $\theta$ , with  $K$  epochs and minibatch size  $M \leq NT$ 
    |  $\theta_{\text{old}} \leftarrow \theta$ 
end

```

Algorithm 1: PPO, Actor-Critic Style

3.2.4 Image Processing

In the previous sections we have learned about two different approaches to train neural nets on solving certain tasks. Although we came to understand that the complexity of the task to be solved correlates strongly with the amount of data at hand, there exist domains from which it is undeniably easier to do so. To equip a neural net with some sort of prior knowledge by switching the domain may therefore not only be highly desirable but sometimes also needed if the amount or quality of data is not sufficient. One domain which is of special interest when it comes to interacting in a three dimensional environment is a domain that represents depth information. If there are any, it may sometimes be possible to extract this kind of prior knowledge from a depth camera. As for this work, we need to rely on stereo cameras and powerful algorithms that allow us to compute depth images in real time. The algorithm that helps us to do so, in terms of the extraction of weighted least squares disparity maps [41], will be presented in the following paragraph - Depth Map Extraction.

Depth Map Extraction

As already pointed out, the depth map is generated from stereo camera images by a technique called stereo block matching [42]. This method works best for edge filtered images, as will become clear soon. To obtain edge filtered images \mathbf{E} , the stereo RGB images are first converted into grayscale \mathbf{G} , which are then convolved with the Sobel kernel \mathbf{S}_x along the horizontal axis [43] (equation 3.131, figure 3.19).

$$\mathbf{E} = \mathbf{S}_x * \mathbf{G} \quad (3.131)$$

When having a look at the Sobel kernel \mathbf{S}_x (equation 3.132), it immediately becomes

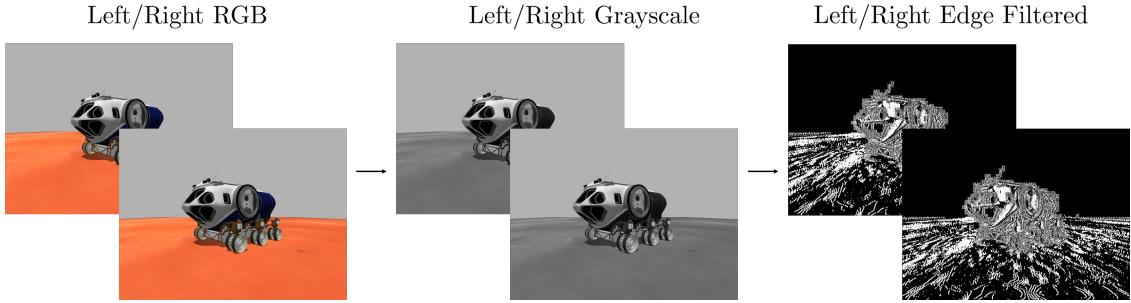


Figure 3.19: Image preprocessing to obtain edge filtered images. The images were taken within the simulation environment Gazebo ([link](#)), and show a space exploration vehicle, for which, with the friendly support of NASA, we generated a Gazebo version ([link](#)).

clear that it approximates the derivative of an image along the horizontal axis. Therefore, at locations of steep change, or simply put, edges, the convolution of the grayscale images with the Sobel kernel results in high values, and thus in the typical appearance of an edge filtered image.

$$\mathbf{S}_x = \begin{pmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{pmatrix} \quad (3.132)$$

To understand the block matching algorithm, we first need to figure out the transformation that images undergo for a change in perspective, which is caused by the two different positions of the cameras within the stereo camera pair. For an ideal setup, we have two identical cameras, and they are neither rotated relatively to each other, nor is there any other translation, but a shift along the x-axis (figure. 3.20). This may of course not always be true, and there are methods to correct for uncertainties, which we will present in the following paragraph, but omit for simplicity right now. The principle goal, for the inference of depth information from two images, is to find points in the right image that correspond to points in the left image. By triangulation, the displacement or disparity of a point in the right image, relative to its corresponding point in the left image, can then be used to extract the depth. The farther a point \mathbf{X} lies away from the cameras, the smaller its displacement will be. In figure 3.20, we can see that a point \mathbf{X} , which is seen by the left camera, could in principle lie anywhere on the epipolar line at \mathbf{x}' , as seen from the right camera, if there is no depth information available. It results that, to find correspondences, one only has to search along the epipolar line. Also, since points in the right image that correspond to points in the left image, will always be displaced to the left, one only has to search in this direction. The procedure is shown in figure 3.21. Instead of looking for single pixel correspondences, it is advised to search for whole block correspondences, since it reduces the noise drastically. Blocks of a defined block size

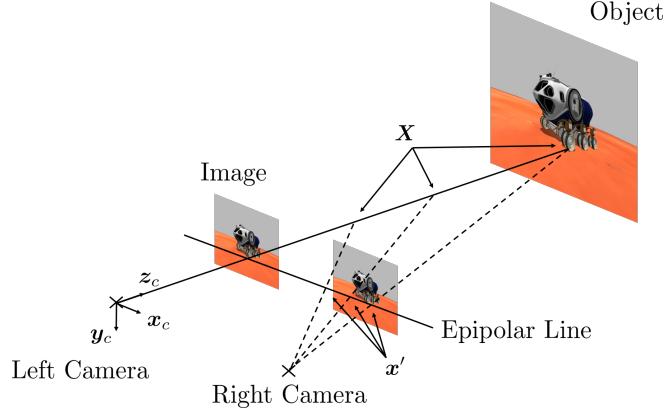


Figure 3.20: The stereo setup with a left and a right camera.

N are taken from the left image, and then the sum of absolute differences SAD is computed for every displacement d in the right image, ranging from zero to number of disparities D (equation 3.133, figure 3.21).

$$\text{SAD}(d) = \sum_{x,y=0}^N |\mathbf{E}_{\text{left}}(x, y) - \mathbf{E}_{\text{right}}(x - d, y)| \quad (3.133)$$

The disparity d that minimizes the sum of absolute differences SAD is taken to serve as the best correspondence and is therefore used in the disparity map. Here we can already see that due to the uniqueness of the edge filtered the images \mathbf{E} , it is easier to find correspondences there, rather than in the grayscale or RGB images. To

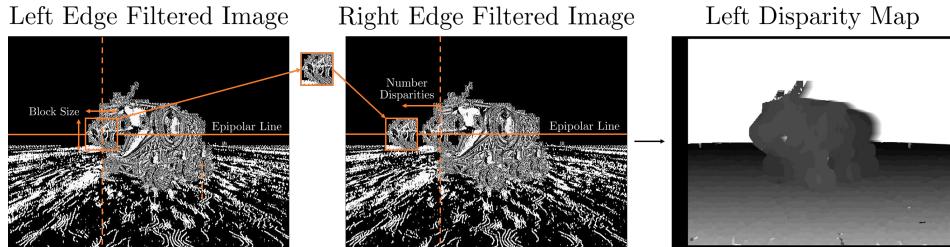


Figure 3.21: Generation of the left disparity map by the block matching algorithm.

further refine the disparity map, and especially to assure good results in textureless regions, we apply a weighted least squares filtering, which is based on the confidence of depth measures. The confidence of depth measures is obtained from the variance within the disparity map \mathbf{D} (equation 3.134, figure 3.22).

$$\text{Var}(\mathbf{D}) = \mathbb{E} [\mathbf{D}^2] - \mathbb{E} [\mathbf{D}]^2 \quad (3.134)$$

Therein, the expectation value for \mathbf{D} is computed by a convolution with the kernel \mathbf{K} from the following equation

$$\mathbf{K} = \alpha \begin{pmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{pmatrix} \quad (3.135)$$

$$\mathbb{E}[\mathbf{D}] = \mathbf{K} * \mathbf{D}, \quad (3.136)$$

where $\alpha = \frac{1}{\text{width} \cdot \text{height}}$ is the normalization factor. The expectation value of the disparity map squared $\mathbb{E}[\mathbf{D}^2]$ is computed in the same way, except for that all elements are squared prior to summing them up. Given the variance, we can introduce a concept which is named confidence map. The confidence map $\text{Con}(\mathbf{D})$ is a measure for the certainty of the computed disparity, and is defined to be linearly dependent on the variance as follows

$$\text{Con}(\mathbf{D}) = \max(1 - r\text{Var}(\mathbf{D}), 0), \quad (3.137)$$

where r is a roll-off factor that defines the change of confidence with growing variance. The resulting disparity confidence is shown in figure 3.22, and is used to outweigh outlying disparity values from the final weighted least squares disparity map. Prior to that, we further introduce an additional measure for the prevention

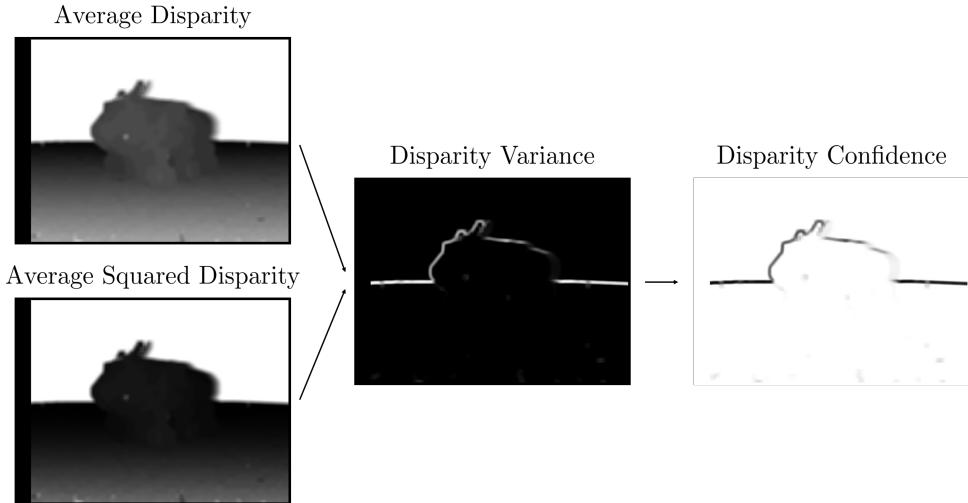


Figure 3.22: Generation of the confidence map from the variance within the disparity map.

of accidentally assigned correspondences in the initial block matching algorithm by using a left right consistency check [44]. Therefore, the block matching algorithm is used on the right image, and we search for correspondences in the left image. In contrast to the computation of the left disparity map \mathbf{D}_{left} , for the right disparity map $\mathbf{D}_{\text{right}}$, we only need to check for correspondences along the epipolar line in the

positive displacement direction. The left right consistency \mathbf{L} is then obtained by

$$\mathbf{L}(x, y) = \begin{cases} \min [\text{Con}(\mathbf{D}_{\text{left}})(x, y), \text{Con}(\mathbf{D}_{\text{right}})(x + d_{\text{left}}, y)] & \text{for } \Delta d < t \\ 0 & \text{else} \end{cases}, \quad (3.138)$$

where d_{left} is the disparity of \mathbf{D}_{left} at position (x, y) , and therefore represents the index shift which results from the block matching algorithm. Furthermore, if $\Delta d = \mathbf{D}_{\text{left}}(x, y) + \mathbf{D}_{\text{right}}(x + d_{\text{left}}, y)$ is smaller than a threshold t , then the left right consistency \mathbf{L} is taken to be the lower bound approximation of the left and right confidences. Otherwise, the consistency is taken to be false, and hence zero (figure 3.23). As already pointed out, the left right consistency, which is nothing but a confidence measure, usually reveals uncertainties in textureless regions. The weighted least squares filtering that we are about to present uses this fact to its advantage. In a first step, a consistency weighted disparity map \mathbf{C} is computed via equation 3.139 (figure 3.23).

$$\mathbf{C} = \mathbf{L} \cdot \mathbf{D}_{\text{left}}, \quad (3.139)$$

where \cdot is an element-wise multiplication. Further, the weighted least squares filter is based on the idea of a bilateral filter [45], and it will try to minimize an energy function $J(\mathbf{U})$, which takes the original grayscale image as guidance to compute a weight $w_{p,q}$ for neighboring pixels p , and q as follows

$$w_{x,y,i,j}(g) = \exp(-|g_{x,y} - g_{i,j}|/\sigma). \quad (3.140)$$

Depending on the range parameter σ , this weight will be high for similar neighboring pixels of the grayscale image g , and therefore lead to huge costs in the following energy function $J(\mathbf{U})$ that we try to minimize

$$J(\mathbf{U}) = \sum_{x,y} \left[(u_{x,y} - c_{x,y})^2 + \lambda \sum_{(i,j) \in \mathcal{N}(x,y)} w_{x,y,i,j}(g) (u_{x,y} - u_{i,j})^2 \right], \quad (3.141)$$

where $c_{x,y}$ are single pixels of the consistency weighted disparity map. The formulation of this energy function results in a solution \mathbf{U} that encourages the propagation of disparity values from high- to low-confidence regions (figure 3.23). Additionally, the weight w , together with the smoothing parameter λ , ensure to have similar disparity values in regions with similar texture. The final disparity map $\mathbf{D}_{\text{final}}$ is then obtained by normalizing the resulting image \mathbf{U} with

$$\mathbf{D}_{\text{final}} = \frac{\mathbf{U}}{\text{WLS}(\mathbf{L})}, \quad (3.142)$$

where $\text{WLS}(\mathbf{U})$ is the weighted least squares filtered version of the left right consistency \mathbf{L} .

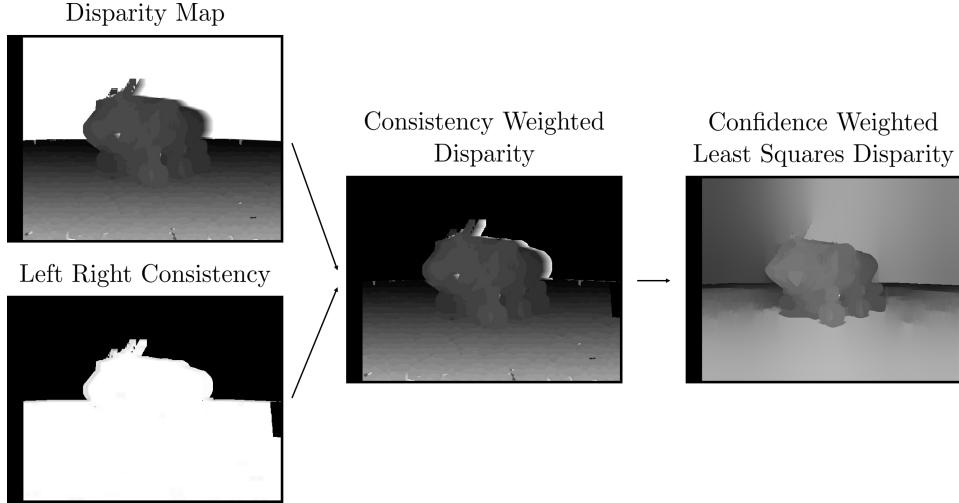


Figure 3.23: Generation of the confidence weighted least squares disparity from the disparity map, and the left right consistency.

As already mentioned for figure 3.20, the assumption of a simply translated stereo camera pair is almost never correct. In addition, there exist camera intrinsics that deform the observed image, and so the epipolar lines. Therefore, as a requirement for the algorithm to work properly, it is important to calibrate the robot's cameras. The next chapter - Mono and Stereo Camera Calibration, will explain in detail how this is done.

Mono and Stereo Camera Calibration

To correct images, as we observe them with a camera, it is required to have a mathematical description of it. A simple one for a camera is the pinhole model, which is shown in figure 3.24. For a pinhole camera model, the image plane lies behind the coordinate frame of the camera, and is turned the other way around, but it is easier to describe the image in a virtual plane, which is located at a distance f along the z_c -axis, where f is the focal length. According to the intercept theorem, a point $\mathbf{X}_c = (X, Y, Z)^T$ is then simply projected to the image plane by the camera matrix \mathbf{K} with

$$\mathbf{x}_c = \mathbf{K} \mathbf{X}_c = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \mathbf{X}_c. \quad (3.143)$$

Therein, \mathbf{K} contains the intrinsic camera parameters, such as the focal lengths and the principal point. For a true pinhole camera $f_x = f_y = f$, but due to errors, usually two different values are chosen. The principal point lies at the position where a light ray connects perpendicularly to the image plane after passing the pinhole, and therefore just defines an offset. For a real setup, it is also required to put a lens at the pinhole's position, which adds some distortion to the image.

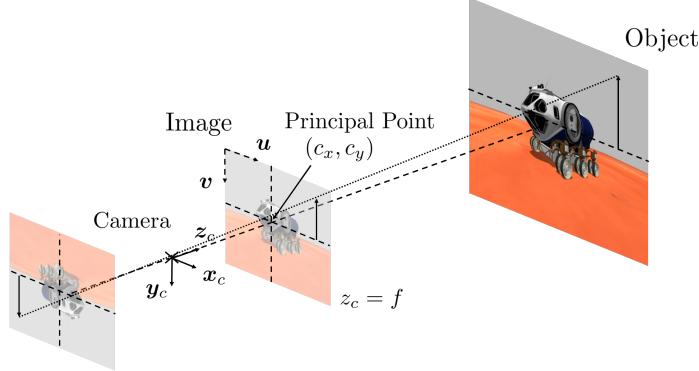


Figure 3.24: Pinhole camera model.

According to [46], we model radial and tangential distortion by

$$x_{c,u} = x_{c,d}(1 + k_1r^2 + k_2r^4 + k_3r^6) + p_1(r^2 + 2x_{c,d}^2) + 2p_2x_{c,d}y_{c,d} \quad (3.144)$$

$$y_{c,u} = x_{c,d}(1 + k_1r^2 + k_2r^4 + k_3r^6) + 2p_1x_{c,d}y_{c,d} + p_2(r^2 + 2y_{c,d}^2), \quad (3.145)$$

where

$(x_{c,d}, y_{c,d})$ = distorted image points within the camera frame c ,
as projected onto the image plane

$(x_{c,u}, y_{c,u})$ = undistorted image points within the camera frame c ,
as projected by an ideal pinhole camera

k_n = n^{th} radial distortion coefficient

p_n = n^{th} tangential distortion coefficient

$$r = \sqrt{x_{c,d}^2 + y_{c,d}^2}.$$

Together, the focal lengths, the principal point, and the distortion coefficients make up the unknowns within our camera model. Goal of the mono camera calibration is now to find these coefficients from images of a well known calibration pattern. Therefore, images of the calibration pattern are taken from different perspectives (figure 3.25). For the mathematical description, the calibration pattern is taken to be at a fixed position and orientation, while it is assumed that the camera was moved. The position of each corner can then be described by the square's size a as follows

$$\mathbf{x}_{nm} = (wa \quad ha \quad 0 \quad 1)^T, \quad (3.146)$$

where we now switched to homogeneous coordinates, and $w \in [0, W]$, $h \in [0, H]$ are whole numbers, corresponding to the width and the height of the pattern. It is then required to find the rotation \mathbf{R} and translation \mathbf{t} , which transforms the object points to the image plane. They are estimated by solving a perspective-n-point problem

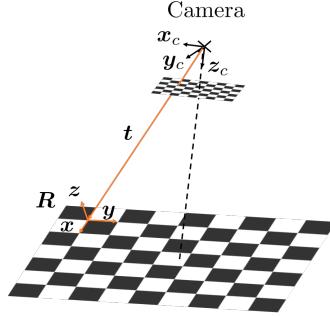


Figure 3.25: Calibration pattern as observed from the camera’s coordinate system C . Within the object’s coordinate system, all chessboard corners lie at a zero z -position.

[47]. Therefore, as shown in figure 3.26 (b), a corner detecting algorithm finds the corners $\mathbf{x}_{c,wh}$ within the image plane. Under the assumption of known intrinsic camera parameters, $\mathbf{x}_{c,wh}$ are then being undistorted according to equations 3.144 and 3.145. Each \mathbf{x}_{nm} can then be transformed to the camera’s frame, and further

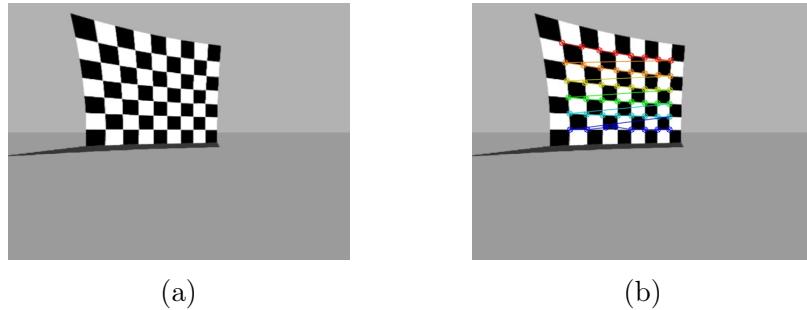


Figure 3.26: Distorted calibration pattern (a), and the image points as found by the algorithm (b).

be projected onto the image plane via

$$\mathbf{x}_{c,wh} = \mathbf{K} (\mathbf{R} \ \mathbf{t}) \mathbf{x}_{wh}, \quad (3.147)$$

where \mathbf{R} describes the rotation, and \mathbf{t} the translation of the camera frame to the object frame. Then, equations 3.144 and 3.145 are applied to obtain the undistorted image points from $\mathbf{x}_{c,wh}$. The undistorted image points $\mathbf{x}_{c,wh,u}$ are then being reprojected by inverting the rotation and translation via

$$\mathbf{x}_{wh,u} = (\mathbf{R} \ \mathbf{t})^{-1} \mathbf{x}_{c,wh,u}, \quad (3.148)$$

from which we compute the re-projection error $\Delta x = \|\mathbf{x}_{wh,u} - \mathbf{x}_{wh}\|_2$. To find the intrinsic parameters, a Levenberg-Marquardt algorithm then optimizes them in an iterative scheme to minimize the re-projection error until convergence [48]. The stereo camera calibration can then be performed by applying the mono camera calibration

to each camera separately, from which the camera intrinsics are obtained. Given the camera intrinsics of both cameras, it is again possible to solve a perspective-n-point problem, which yields the positions and orientations of both cameras with respect to the observed object. This enables us to compute the fundamental matrix \mathbf{F} , which transforms points from the left camera's view $\mathbf{x}_{c_{\text{left}}}$, to points $\mathbf{x}_{c_{\text{right}}}$, as seen by the right camera via

$$\mathbf{x}_{c_{\text{right}}}^T \mathbf{F} \mathbf{x}_{c_{\text{left}}} = \mathbf{0} \quad (3.149)$$

The mapping enables us to rectify the left and the right image [49], using the rectification transforms \mathbf{R}_i , which means that we can compute homography transforms that align epipolar lines within the images (figure 3.27), which were previously defined by the fundamental matrix. These homography transforms map the images, as

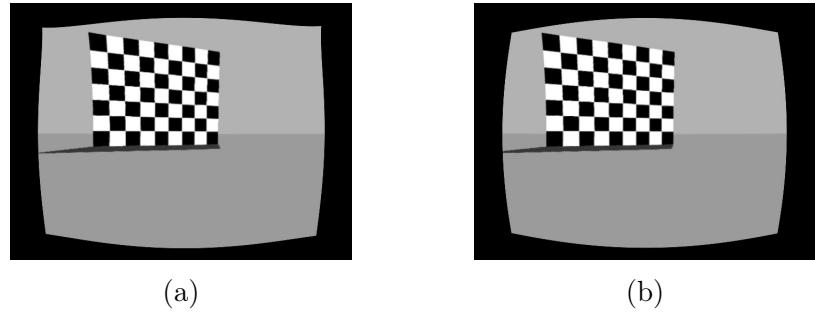


Figure 3.27: Undistorted calibration pattern, as observed by the left camera (a), and by the right camera (b). For comparison, see the distorted calibration pattern in figure 3.26, and note how the horizons within the images align.

we observe them with the cameras, to a shared virtual plane, which is defined by the newly obtained projection matrices \mathbf{P}_i . Therefore, we reached our initial goal, since it enables us to apply the stereo block matching algorithm, which got introduced earlier, to the transformed images.

4 Methods

4.1 Software

[50] [51] [53] [54] [55] [56] [57] [58]

4.2 Implementation

5 Experiments

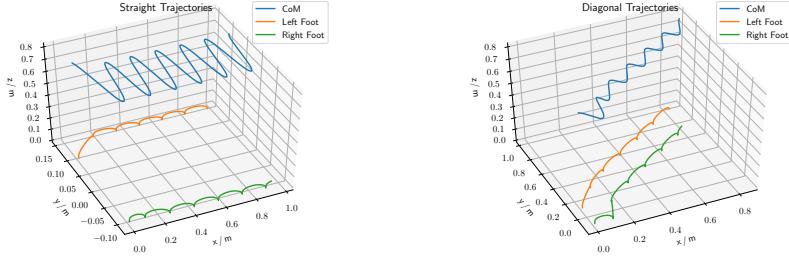
Within the experiments chapter, we will first benchmark the nonlinear model predictive control implementation of ours for purely simulated tasks in section 5.1, and then tune hyperparameters for it to run well on Heicub. This will allow us to define a standardized environment for walking experiments in section 5.1.2, to later compare user controlled performance against autonomously controlled performance in section 5.2. Though, before we can tackle the idea of behavioral cloning, which got described in section 3.2.2, we need to meet the prerequisites, that is we will calibrate the camera, and tune the depth map parameter extraction in sections 5.2.1, and 5.2.2, respectively. All of the above mentioned steps, will then allow us to collect data and to train a newly developed neural network architecture on it in section 5.2.3. Finally, we will compare the humanly controlled robot’s balance with the artificially controlled robot’s balance in section 5.2.4, and investigate on the reinforcement learning approach for autonomous navigation in section 5.2.5.

5.1 User Controlled Walking

As the fundamental building block for the comparison to autonomous walking, we now need to investigate on user controlled walking. This enables us, in contrast to the control by a neural network, to find the best parameters for the pattern generation in a well controllable environment. These parameters will then be kept constant throughout the rest of this thesis, in order to allow for a good comparison between user controlled walking and autonomously controlled walking. Furthermore, we will rely on them to gather data for the behavioral cloning approach in section 5.2.3.

5.1.1 Benchmarking of Implementation

To evaluate the pattern generator implementation of ours, we have benchmarked it against an existing one, which was written in Python. Therefore, we defined velocity commands, as they typically may appear in real world scenarios. The four defined use cases, which are shown in figures 5.1 and 5.2, were parametrized in accordance to the HRP-2 humanoid robot. The parameters for these tests can be found in the YAML file at the provided [link](#). To obtain the pattern generator’s performance, in terms of speed, the straight walk experiment in figure 5.1 (a) got executed ten times on an Intel Core i7-7700HQ CPU at 2.8 GHz, for both, the Python and our implementation. It took 873 ± 23 ms and 147.7 ± 0.5 ms to execute the code for 100 iterations on average, which means that a single iteration took 8.73 ± 0.23 ms and

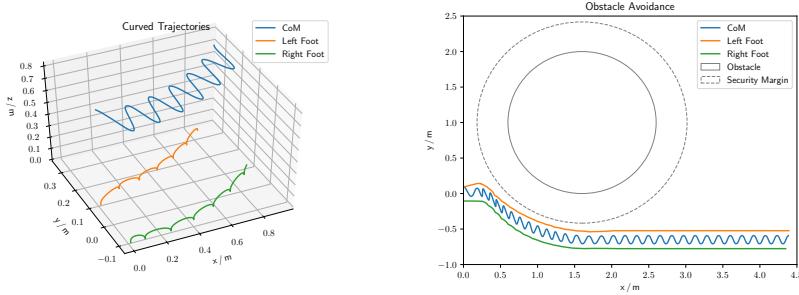


(a) Straight trajectories at
 $\mathbf{v} = (0.1 \ 0.0 \ 0.0)^T$.

(b) Diagonal trajectories at
 $\mathbf{v} = (0.1 \ 0.1 \ 0.0)^T$.

Figure 5.1: Simple trajectories. The velocities are given in units of
 $(\text{m/s } \text{m/s } \text{rad/s})^T$.

1.48 ± 0.01 ms. We therefore could achieve a speedup of around 600 percent with our implementation. We further demonstrate the avoidance of a convex obstacle with a security margin that keeps the robot at a safe distance in figure 5.2 (b). The used obstacle is defined to be located at $x = 1.6$ m and $y = 1.0$ m, with a radius of $R = 1.0$ m, and a security margin of $m = 0.4$ m. To ensure a smooth motion at all



(a) Curved trajectories at
 $\mathbf{v} = (0.1 \ 0.0 \ 0.1)^T$.

(b) Obstacle avoidance at
 $\mathbf{v} = (0.1 \ 0.0 \ 0.0)^T$

Figure 5.2: Advanced trajectories. The velocities are given in units of
 $(\text{m/s } \text{m/s } \text{rad/s})^T$.

times, and to benchmark the interpolation, we plotted the x-, y-, and z-trajectories for the left and the right foot, as shown in figure 5.3. The plots were generated on the curved trajectory of figure 5.2 (a), and they reveal a continuous behavior at all times and for all dimensions. The benchmarked pattern generator then enabled us to run it on the real robot, which we did in a test environment that will be presented in the next section - Performance in Test Environment.

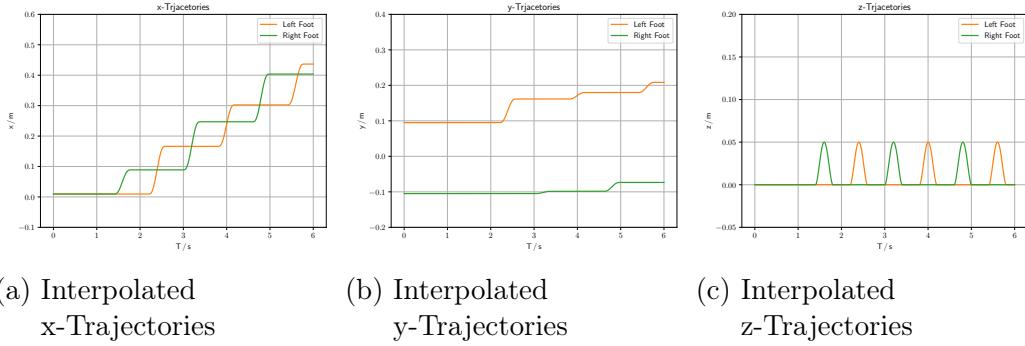


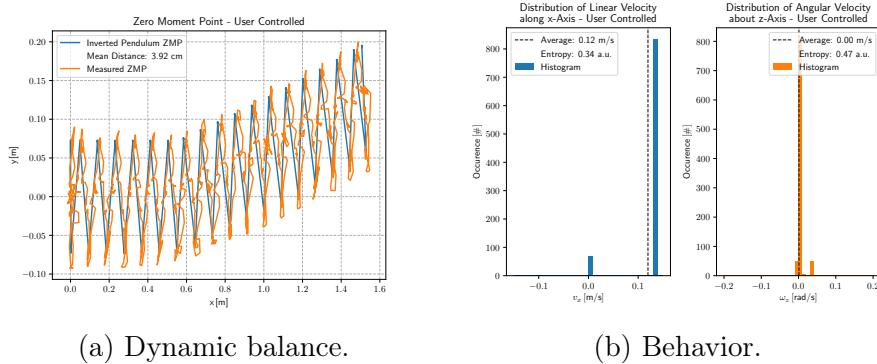
Figure 5.3: Interpolated foot trajectories.

Parameter	Value	Parameter	Value
T_{Step}	3.20 s	N	16 #
$T_{\text{Double Support}}$	1.60 s	α	1.0 a.u.
$T_{\text{Command Period}}$	0.01 s	β	100 a.u.
T_{Preview}	0.40 s	γ	0.01 a.u.

Table 5.1: Parameters which used to work best on Heicub.

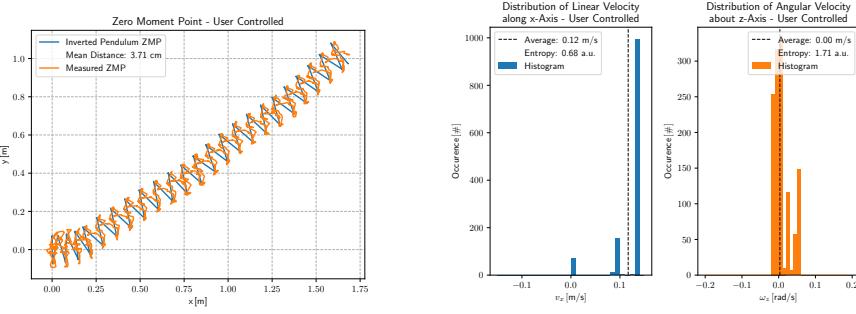
5.1.2 Performance in Test Environment

For the execution on Heicub, we chose a parameter setting with which we could ensure balanced motion for all velocity commands. This could be achieved by choosing the parameters, which are listed in the YAML configurations file at the provided [link](#). The most important ones therein are further shown in table 5.1. Now given these parameters, we designed an environment for Heicub to walk in. A human user had to control the robot in four different scenarios. In each of the scenarios, Heicub started from a reference position, in order to reach a fire extinguisher at a distant location. By design, the setup allows for benchmarking of dynamic balance in all four different scenarios. As will be shown in section 5.2, we require a neural network to execute the same tasks, and therefore we can compare the performances of a human user with that of an autonomous agent. For the dynamic balance evaluation, we extracted the desired ZMP from the nonlinear model predictive controller, and measured the true ZMP, in accordance to equations 3.16 and 3.17, by recording the ankles' force torque sensor readouts. We further kept track of the velocity commands to the pattern generator for all tasks to extract the user's behavior. The first task was simply to go two meters straight forward (figure 5.4). The behavior therein is visualized by a histogram of all velocity commands over the course of the task (figure 5.4 (b)). For the bin size we chose 0.01 m/s and 0.01 rad/s, respectively. The second task was to reach the fire extinguisher, which was located to the left of the robot. In order to reach it, it was therefore required to perform a curved walk (figure 5.5). The third task (figure 5.6) involved the avoidance of an obstacle, namely a chair. For the fourth task (figure 5.7), the robot started with its back



(a) Dynamic balance. (b) Behavior.

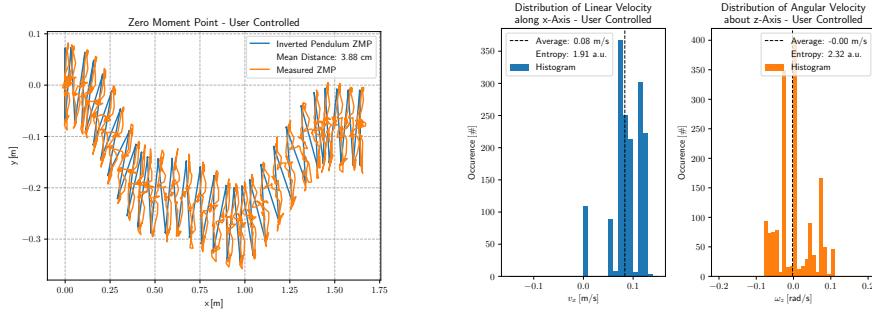
Figure 5.4: User controlled straight walk.



(a) Dynamic balance. (b) Behavior.

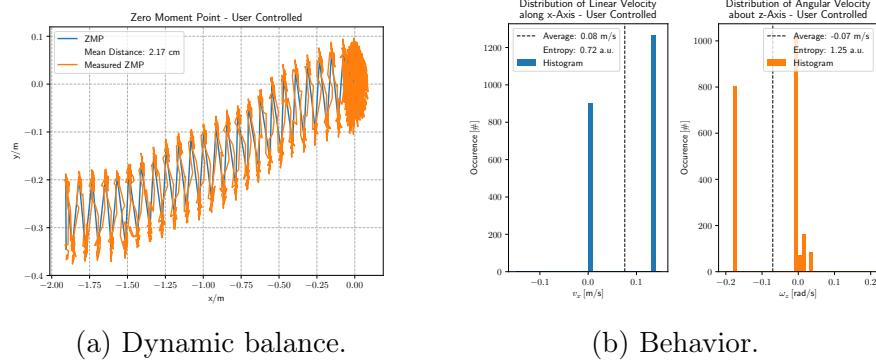
Figure 5.5: User controlled curved walk.

pointing towards the fire extinguisher.



(a) Dynamic balance. (b) Behavior.

Figure 5.6: User controlled obstacle avoidance.



(a) Dynamic balance.

(b) Behavior.

Figure 5.7: User controlled environment scanning.

5.2 Autonomous Walking

The autonomous walking is based upon the performance within the test environment from section 5.1.2. The found parameters are used for comparative reason throughout this section as well. As explained in section 3.2.4, the neural network benefits strongly from an available depth map as input. We will therefore deal with the depth map extraction first.

5.2.1 Camera Calibration

As described in section 3.2.4, in order for the stereo block matching algorithm to work properly (equation 3.133), it is required to calibrate the cameras. We shortly verified this in figure 5.8, where we extracted a depth map from the uncalibrated stereo camera pair. For the calibration we chose to use a chess-board calibration



(a) Left disparity map.



(b) Confidence weighted least squares filtered disparity map.

Figure 5.8: Depth map extraction without calibration. The parameters were set as follows to $N = 13$, $D = 32$, $\sigma = 1$, and $\lambda = 10^4$.

pattern, see figure 5.9. The used calibration pattern has width of $W = 8$, and a height of $H = 6$, where each square has a size of $a = 22.5$ mm (equation 3.146). We took a total of $N = 60$ images of the calibration pattern for varying orientations and translations with respect to the camera, which results in a total of $W \times H \times N = 2880$ points for the calibration. As the resulting mean squared re-projection error

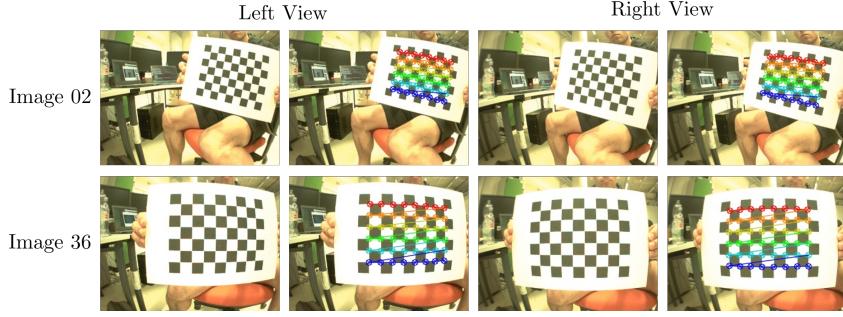


Figure 5.9: Exemplary left and right camera views of the calibration pattern as acquired during the calibration process. The colorful points indicate the detected corners in the image plane. Refer to figure 3.26 for the theory.

Intrinsic Parameter	Left Camera	Right Camera
f_x [pixel/mm]	$2.36 \cdot 10^2$	$2.32 \cdot 10^2$
f_y [pixel/mm]	$2.37 \cdot 10^2$	$2.32 \cdot 10^2$
c_x [pixel]	$1.63 \cdot 10^2$	$1.86 \cdot 10^2$
c_y [pixel]	$1.11 \cdot 10^2$	$1.30 \cdot 10^2$
k_1 [1/pixel ²]	$-4.54 \cdot 10^{-1}$	$-4.58 \cdot 10^{-1}$
k_2 [1/pixel ⁴]	$2.90 \cdot 10^{-1}$	$3.18 \cdot 10^{-1}$
k_3 [1/pixel ⁶]	$-1.21 \cdot 10^{-1}$	$-1.48 \cdot 10^{-1}$
p_1 [1/pixel]	$-2.73 \cdot 10^{-3}$	$3.02 \cdot 10^{-4}$
p_2 [1/pixel]	$2.16 \cdot 10^{-4}$	$7.63 \cdot 10^{-4}$

Table 5.2: Intrinsic parameters of single cameras. These parameters can be found as YAML file on GitHub ([link](#)).

$\Delta\bar{x} = 1/(WHN) \sum_0^{WHN} \Delta x$ (equation 3.148), we obtained $\Delta\bar{x}_l = 0.26$ pixel, and $\Delta\bar{x}_r = 0.25$ pixel, for the left, and the right camera, respectively. According to equations 3.143, 3.144, and 3.145, we therefore determined the camera intrinsic parameters as listed in table 5.2. Then given the calibration of each single camera, we computed the rectification transforms \mathbf{R}_i , and the projection matrices \mathbf{P}_i in the rectified coordinate system for each camera 5.3. Exemplary rectified images, which rely on the matrices of table 5.3, are shown in figure 5.10 (a). Since there is a slight rotation of the calibration pattern, it is not obvious that the images got rectified well. Therefore, the same images are shown in figure 5.10 (b), but slightly rotated such that the calibration pattern aligns horizontally. The blue line therein indicates that in contrast to the original image, straight lines now appear straight across both images, which is crucial for the block matching algorithm in the next section - Depth Map Parameter Tuning.

Camera Extrinsic Parameter

	\mathbf{R} [a.u.]	$\begin{pmatrix} 9.93 \cdot 10^{-1} & -2.65 \cdot 10^{-3} & 1.14 \cdot 10^{-1} \\ 5.41 \cdot 10^{-1} & 1.00 \cdot 10^0 & -2.39 \cdot 10^{-2} \\ -1.14 \cdot 10^{-1} & 2.43 \cdot 10^{-2} & 9.93 \cdot 10^{-1} \end{pmatrix}$
Left	\mathbf{P} [pixel/mm]	$\begin{pmatrix} 2.34 \cdot 10^2 & 0.00 & 1.88 \cdot 10^2 & 0.00 \text{ mm} \\ 0.00 & 2.34 \cdot 10^2 & 4.87 \cdot 10^1 & 0.00 \text{ mm} \\ 0.00 & 0.00 & 1.00 & 0.00 \text{ mm} \end{pmatrix}$
	\mathbf{R} [a.u.]	$\begin{pmatrix} 9.95 \cdot 10^{-1} & -2.30 \cdot 10^{-2} & 9.93 \cdot 10^{-2} \\ 2.07 \cdot 10^{-2} & 1.00 \cdot 10^0 & 2.38 \cdot 10^{-2} \\ -9.98 \cdot 10^{-2} & -2.16 \cdot 10^{-2} & 9.95 \cdot 10^{-1} \end{pmatrix}$
Right	\mathbf{P} [pixel/mm]	$\begin{pmatrix} 2.34 \cdot 10^2 & 0.00 & 1.88 \cdot 10^2 & -1.60 \cdot 10^1 \text{ mm} \\ 0.00 & 2.34 \cdot 10^2 & 4.88 \cdot 10^1 & 0.00 \text{ mm} \\ 0.00 & 0.00 & 1.00 & 0.00 \text{ mm} \end{pmatrix}$

Table 5.3: Rectification transforms \mathbf{R}_i , and projection matrices \mathbf{P}_i , for the left, and the right camera, respectively. These parameters can be found as YAML file on GitHub ([link](#)).

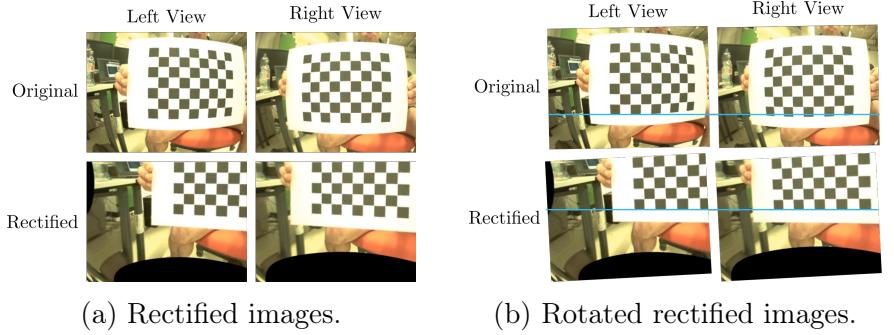


Figure 5.10: Rectified and original view of the stereo camera. Refer to figure 3.27 for the theory.

5.2.2 Depth Parameter Tuning

Within this section, we shortly explore the effects of all tunable parameters on the depth map generation. Therefore, we utilize a simple experimental setup. Within the setup, Heicub points its stereo camera towards three chairs that are located at a distance of 1 m towards each other, and towards the cameras, so to cover close, medium, and far distances. The consecutive chairs are slightly shifted, in order to enable the simultaneous observation of all of them. The rectified view of the environment is shown in figure 5.11. The depth map extraction, which utilizes the rectified images, depends on a stereo block matching algorithm that got explained in

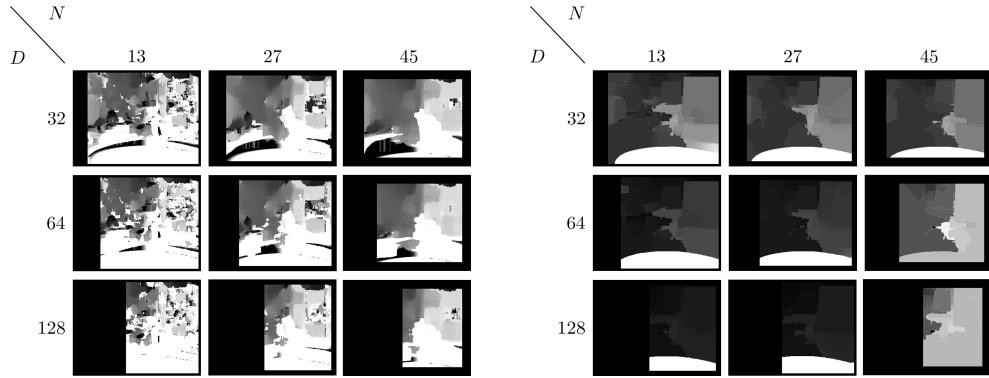


(a) Left camera's view.

(b) Right camera's view.

Figure 5.11: Heicub's perspective of the scene for the depth map parameter tuning.

section 3.2.4. It mainly depends on the window size and the number of disparities for the sum of absolute difference computation. We evaluate the influence of those two parameters in figure 5.12 (a) in a grid search fashion. It is apparent that the change in the number of disparities has close to no influence onto the depth map quality, while it removes plenty of useful information from the left hand side of images. The same holds true for the window size, except that it removes some noise from the depth maps. In combination with the confidence weighted least squares filtering, we



(a) Left disparity. Please refer to figure 3.21 and equation 3.133 for the theory.

(b) Confidence weighted least squares disparity map. Please refer to figure 3.23 and equation 3.142 for the theory.

Figure 5.12: Left disparity map and confidence weighted least squares disparity for changing SAD window sizes N and number of disparities D .

can observe that most of the noise is already removed (figure 5.12 (b)), for which it is more import to keep the information close to the images' borders. We therefore chose to set the number of disparities $D = 32$, and the windows size $N = 13$ in the following. Within these depth maps, the global energy weighting λ was set to 10^4 , and the local bilateral filter decay σ to 1, since we observed the best performance for them. The influence of those two parameters is visualized in figure 5.13. We can see that, in good accordance with the theory, σ contributes to the smoothing of the depth map, and that λ enforces a change in depth across edges within the RGB images.

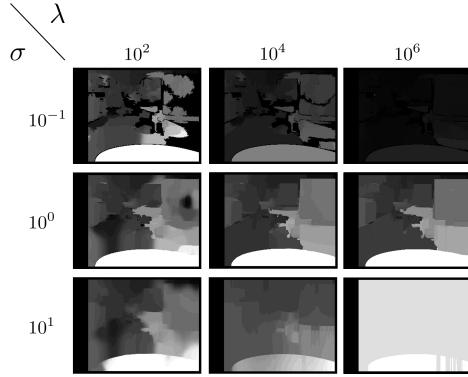


Figure 5.13: Confidence weighted least squares disparity for changing energy weightings σ and λ . Please refer to equations 3.140 and 3.141 for the theory.

5.2.3 Data Acquisition and Training

As already pointed out, the task we wanted our neural network to solve was to find a fire extinguisher within a room and then to move towards it. For us to apply behavioral cloning to the task, it was necessary to act according to this policy first, and then to train a neural network on the acquired data. For the data, it is essential to sample homogeneously over the task’s distribution. We therefore recorded velocity commands and the corresponding images for 66 distinct epochs. Each of the epochs was designed to reflect different scenarios, among them obstacle avoidance, interaction with humans, walking towards the fire extinguisher, and searching for the fire extinguisher when it was not possible to directly see it. As a result of the observations that we gained from the user controlled walking in section 5.1, we restricted the user’s policy to only use linear velocities along the x-axis, and angular velocities about the z-axis, since linear velocities along the y-axis revealed to pose an inefficient way of moving the robot. Exemplary samples of the recorded dataset are presented in figure 5.14. It contains 134401 samples in total, which were recorded at a rate of 5 frames per second, and therefore they account for roughly 7.5 hours of data. For each sample, we stored the left camera’s RGB view, as well as the confidence weighted least squares disparity map. This lead to a total of 268802 recorded images at a resolution of 240×320 pixels, which together make up for 4.7 GB of data. As a preprocessing step, we cropped regions of the recorded images that do not contain useful information. These regions are firstly caused by deformations that are introduced at the rectification step, as can for example be seen in figure 5.11, and secondly by the depth map’s extraction, which results in unknown regions at the image’s boarders (figure 5.12). The exemplary samples from the dataset (figure 5.14), are already preprocessed, and we can observe how only useful information is kept, in order not to confuse the neural network. To reduce the required amount of GPU memory, we further downscaled the preprocessed images to a size of 60×80 pixels, a size at which most of the information is still being kept, and stacked the RBD images with the confidence weighted least squares disparity map to obtain



Figure 5.14: Samples of the recorded dataset. The dataset shows a very diverse number of situations for the neural network to learn from.

RGBD images. For the training of the neural network, we split the acquired dataset into a train, and a validation set. The training set held a randomly sampled fraction of 90% of all recorded images, while the validation split held the other 10%. We only trained the neural network on the training set, and stored the weights that performed best on the validation split, in order to avoid overfitting. Since it has shown very good convergence on our dataset, we chose to use a UNet [59] as the network architecture, which can be seen in figure 5.15. The UNet promotes image abstraction capabilities of an auto-encoder that is caused by its bottleneck design, and furthermore shows faster convergence due to its residual connections, which allow the gradient flow to reach deeper layers earlier. Each image that is being forwarded, goes through a number of convolutional layers with rectifying linear unit activation functions, and is subsequently downsampled by max-pooling operations. We repeated this process for two times. Once the most downsampled layer is reached, the weights are being upsampled by simple interpolations again, which simply results in the same resolution that the layers had during the downscaling process. The skip connections then concatenate the shallow with the deep layers to a new layer at each scale, which is then being forwarded further through a number of convolutional layers with rectifying linear units. The nature of the robot's motion, which inher-

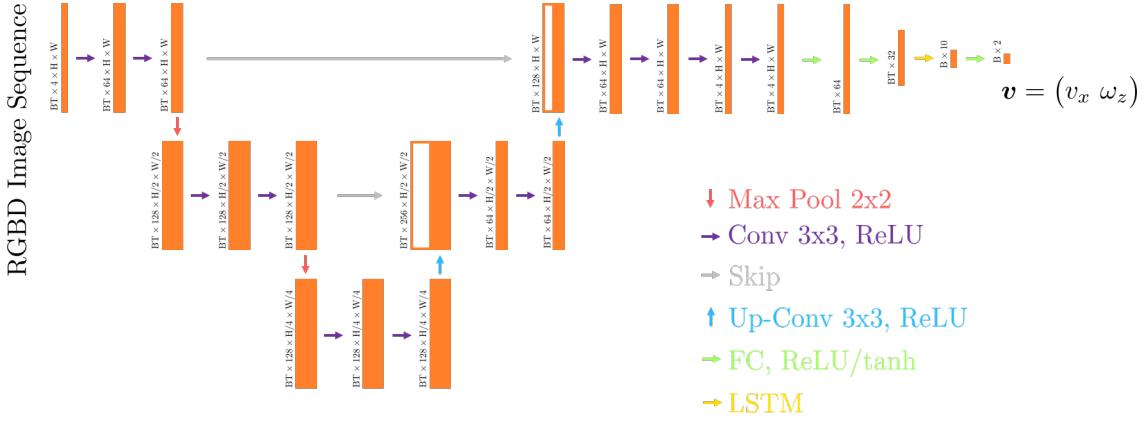


Figure 5.15: UNet-LSTM network architecture.

ently causes the cameras to periodically move from the left to the right, required us to equip the network architecture by a temporal understanding. We therefore extended the UNet architecture to a novel UNet-LSTM structure. The developed architecture takes up a sequence of consecutive RGBD images, and forwards them through the UNet until it reaches a fully connected regression layer that shall output velocity values in the end. Each image therein creates a signal, from which the LSTM is supposed to only keep the most relevant information. This design helped the network to understand that a fire extinguisher to the left of the image may only be caused by the cameras that are temporally displaced to the right. The final fully connected layer then returns what we use for the loss function, and therefore a velocity. In contrast to the preceding layers, the last fully connected layer uses a hyperbolic tangent function, which restricts the output to a range of $[-1, 1]$, which is then being scaled by the velocity that the pattern generator maximally allows. The architecture can be found at the provided [link](#). Due to memory limitations, we trained the network on a sequence length of 5 RGBD images, and a batch size of 32. For the loss, we chose a mean squared error with respect to the most recent velocity command. We used the Adam optimization [60] at a learning rate of 0.001, and trained for 100 epochs on a Nvidia GTX 1080 with 8 GB RAM, to which we were granted access to. This took us around 48 hours. The loss history is shown in figure 5.16, which reveals a good convergence after around 40 epochs. After training, we generated velocity histograms, as we already did in section 5.1, but this time over the whole validation split for both, the ground truth and the predicted behavior, as shown in figure 5.17. To generate the predictions on the validation split took 4 ms with a GeForce GTX 1050 for each of the 13341 images on average. By pure sight, it can already be seen that the network performs well on the validation split, and the Kullback-Leibler divergence D_{KL} , which measures the distance of probability distributions, enforces this observation further. The Kullback-Leibler divergence for

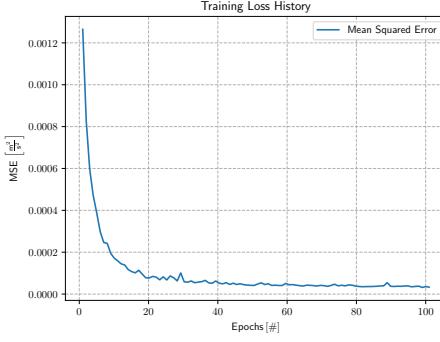
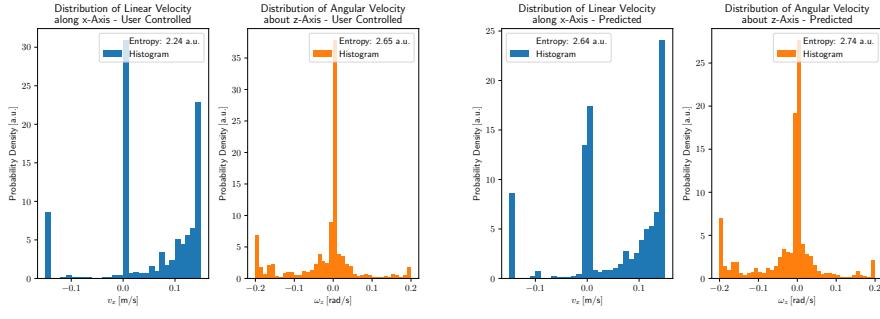


Figure 5.16: Mean squared error training loss history.



(a) Ground truth velocity commands. (b) Predicted velocity commands.

Figure 5.17: Normalized velocity histograms over the validation split.

two discrete probability distributions $p(v)$ and $q(v)$, is computed as follows

$$D_{\text{KL}} = \sum_{v \in V} p(v) \log \frac{p(v)}{q(v)}, \quad (5.1)$$

where for our case, $p(v)$ is the ground truth velocity distribution, and $q(v)$ is the predicted velocity distribution. For the distribution of linear velocities along the x-axis v_x , we computed the Kullback-Leibler divergence to be $D_{\text{KL}}^x = 0.33$ a.u., and for the angular velocity about the z-axis ω_z , we obtained $D_{\text{KL}}^z = 0.06$ a.u.. Now the beauty within the task at hand lies in the fact that we were not solely dependent on a validation split for performance evaluations, but instead that we simply could run Heicub in a previously unseen test environment. It is the same test environment, which we already introduced in section 5.1, and we will evaluate the trained neural network's behavior within it in the next section - Performance in Test Environment.

5.2.4 Performance in Test Environment

For the performance benchmarking, we relied on the well defined experimental setup from section 5.1. Once more the tasks were to move straight towards a fire extin-

guisher, to turn and to move towards it, to avoid an obstacle on the way, and to find the fire extinguisher. In order to ensure reproducibility, we let Heicub solve each of these tasks twice, as we did for the user controlled case. The setup is shown in figure 5.18, which shows Heicub's movement over the course of each task along with its sight of the scene. While Heicub did manage to solve the straight walk,

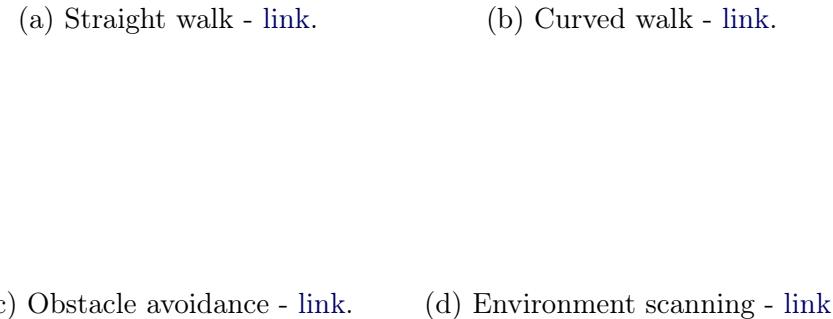


Figure 5.18: Heicub's behavior in the test environment for benchmarking tasks.

the curved walk, and the environment scanning, it had trouble to go towards the fire extinguisher, once it avoided the obstacle. Again, we tracked the zero moment point from the pattern generator, as well as the true zero moment point from the force torque readouts. The results of these measurements are shown in figures 5.19 - 5.22. It can clearly be seen that the neural network's behavior for within the test environment is much noisier than it was for the validation split in figure 5.17. This especially holds true for the angular velocity distributions. In contrast to

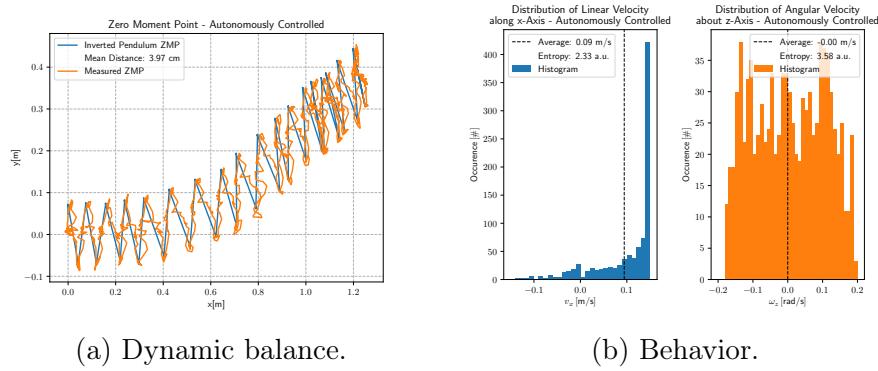
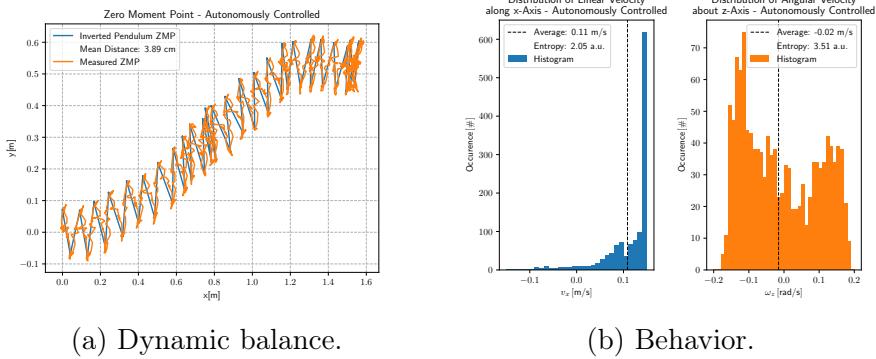
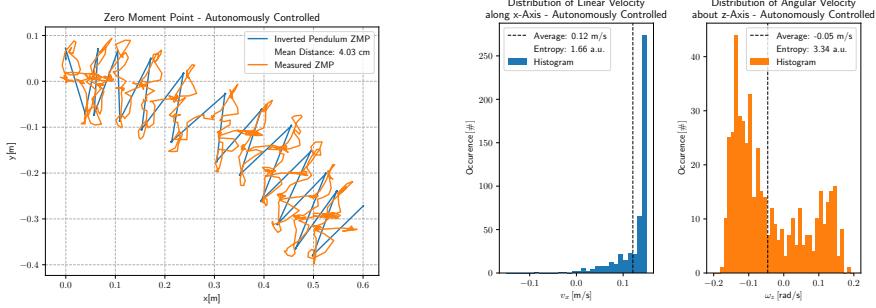


Figure 5.19: Autonomously controlled straight walk.



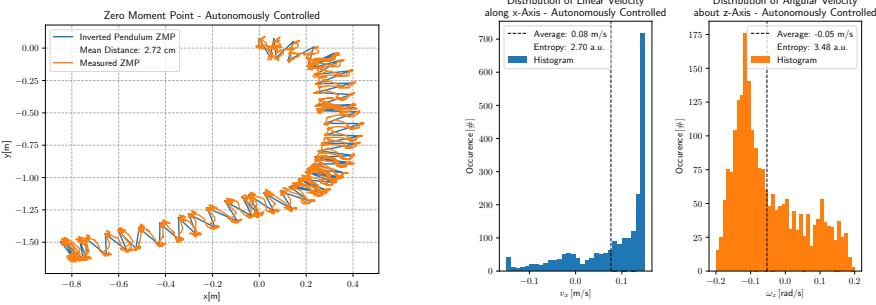
(a) Dynamic balance. (b) Behavior.

Figure 5.20: Autonomous controlled curved walk.



(a) Dynamic balance. (b) Behavior.

Figure 5.21: Autonomous controlled obstacle avoidance.



(a) Dynamic balance. (b) Behavior.

Figure 5.22: Autonomous controlled environment scanning.

the validation split, there is no simple method anymore to compare the results of user controlled walking and autonomously controlled walking. This has two main reasons. First of all, a human, which controls the robot, does so from a third person perspective, while the neural network interacts with the environment from a first person perspective. Secondly, different actions that are taken, result in different states being in. That said, once the human agent and the artificial agent only take

slightly different decisions, the two behaviors will be driven from completely different states, which causes yet another different action. We can therefore only compare whether or not the task of interest got solved, and how the different behaviors influenced the primal goal of dynamic balance, which brings us back to the observation of the neural network’s noisy policy. To assess the level of noise, we computed the entropy $S(p(v))$ within the velocity distributions as follows

$$S(p(v)) = \sum_{v \in V} p(v) \log p(v) \quad (5.2)$$

Furthermore, to rate the dynamic balance, we computed the mean distance of the inverted pendulum zero moment point from the pattern generator, and the measured zero moment point. We did so for every behavior and dynamic balance plot of our benchmarking setup. We then plotted the mean distance against the entropy in figure 5.23, in order to probe the influence of noisy decisions onto the balance. The average mean distance with the standard deviation for user controlled

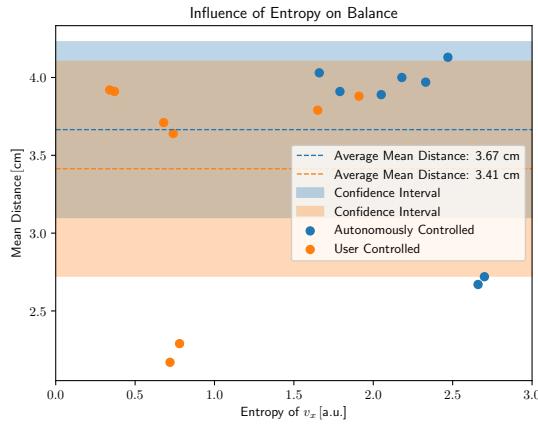
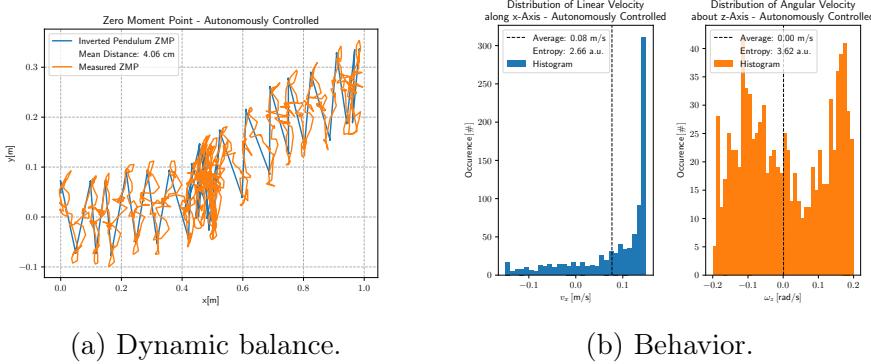


Figure 5.23: Influence of entropic commands onto Heicub’s dynamic balance.

walking therein is 3.41 ± 0.69 cm, while that of autonomously controlled walking is 3.67 ± 0.56 cm. Within the 1σ -range, we could therefore demonstrate that the control signal’s entropy does not have an affect on the balance. In addition to the benchmarking tasks, we further wanted to demonstrate the robot’s behavior in two more scenarios. The first additional scenario involves a dynamic environment, in which the robot interacts with a human and a moving fire extinguisher. The second additional scenario simply demonstrates semantic understanding of the neural network, in that it poses the challenge of having similarly colored objects to distinguish from. Both additional tests are shown in figure 5.24, and they were solved successfully. Especially in the behavior plot of figure 5.26, we can see that the robot successfully managed to walk backwards to avoid a collision with the human. Given the successful results of our first approach to train a neural network on autonomous navigation, we then continued to evaluate the proximal policy optimization algorithm, which got described in section 3.2.3.

(a) Dynamic environment - link. (b) Semantic understanding - link.

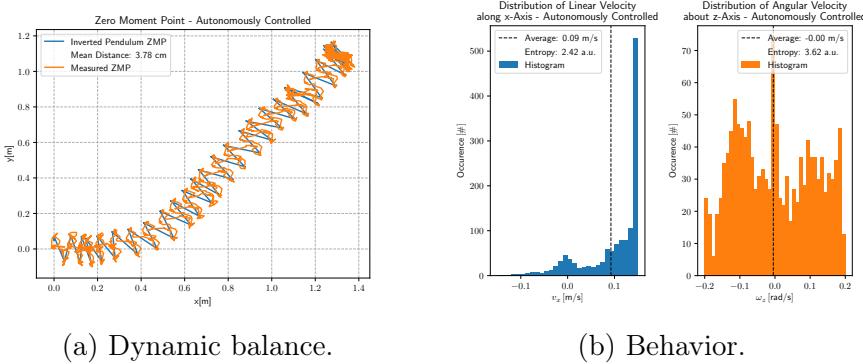
Figure 5.24: Heicub's behavior in the test environment for additional tasks.



(a) Dynamic balance.

(b) Behavior.

Figure 5.25: Autonomously controlled in a dynamic environment.



(a) Dynamic balance.

(b) Behavior.

Figure 5.26: Autonomously controlled for demonstration of semantic understanding.

5.2.5 Proximal Policy Optimization

Since there is to this date no feasible way of training an agent on autonomous navigation in real time with reinforcement learning, we decided to implement a benchmarking environment that is shown in figure 5.27. The agent's goal within this setup is to move towards the red dot, while keeping a maximum distance of 10 a.u. towards it. The environment's state is simply described by a concatenation of the agent's position $\mathbf{a} = (a_x \ a_y)^T$ with that of the goal $\mathbf{g} = (g_x \ g_y)^T$. The reward r_t , at time step t , is designed to encourage motion towards the goal, by taking the

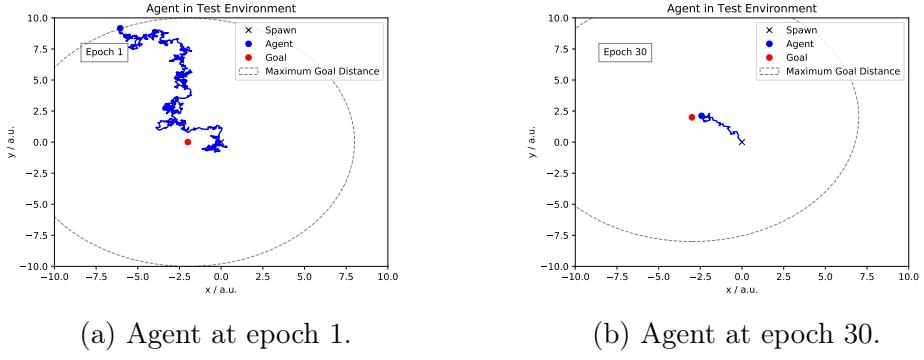


Figure 5.27: Artificial agent in proximal policy optimization test environment.

difference of the previous and the current goal distance $r_t = \|\mathbf{a}_{t-1} - \mathbf{g}_{t-1}\|_2 - \|\mathbf{a}_t - \mathbf{g}_t\|_2$. Furthermore, a reward of ten was gained for successful completion, while a reward of negative ten was granted for whenever the agent left the maximally allowed distance towards the goal, see for example figure 5.27 (a). In each of the cases, the environment was reset, and the goal got spawned at a random location. For both, the agent and the critic network, we used a fully connected neural network with 2 hidden layers of size 16, and 32, respectively. The output layer provided 2 units, which reflect our agent's degrees of freedom in the environment. For the hidden units, we again relied on rectifying linear units as our activation function, while we used a hyperbolic tangent for the output. We were able to produce the best results with a gradient clipping at $\epsilon = 0.2$ (see equation 3.129), and the cost function hyper-parameters $c_1 = 0.5$ and $c_2 = 0.1/\bar{r}_t$, where \bar{r}_t denotes the average reward, and which can be found in equation 3.130. We ran the environment for 10000

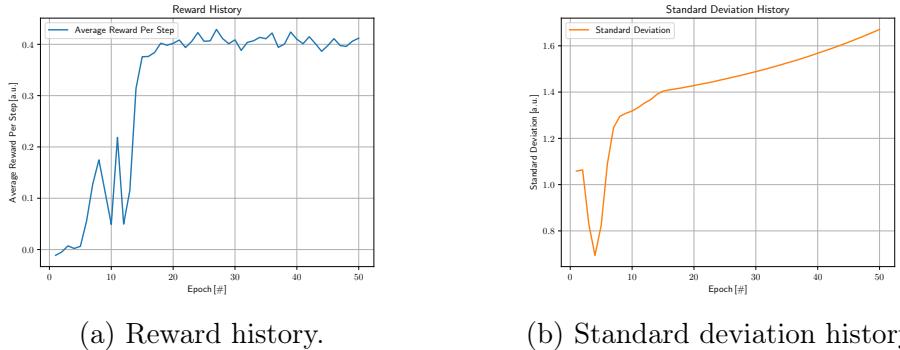


Figure 5.28: Proximal policy optimization in test environment over 50 epochs.

steps per epoch, and updated the networks every 4096 actions, with a minibatch size of $M = 512$ for 8 proximal policy optimization epochs (see algorithm 1). The Adam optimizer then led to convergence at a learning rate of 0.01 after about thirty epochs (see figure 5.28). A main reason for the fast convergence was caused by the chosen entropy hyperparameter c_2 , which encouraged exploration on low rewards,

and damped exploration on high rewards. We computed the entropy $S[\pi_\theta]$ from the differential entropy of our Gaussian policy π_θ via ([link](#))

$$S[\pi_\theta] = 0.5 + 0.5 \log(2\pi) + \log(\sigma), \quad (5.3)$$

where σ is the standard deviation. We can then see the standard deviation's influence on the reward, as it starts to increase strongly in figure 5.28 (b). After having trained the agent successfully for 50 epochs, we ran the policy π_θ without noise contribution, but rather took the average μ , as proposed by the actor network. An example of the agent's behavior can be seen in figure 5.29, which now appears really smooth. We

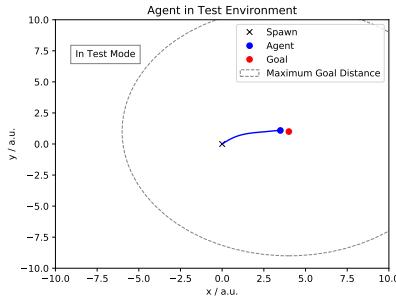


Figure 5.29: Proximal policy optimization in test mode, that is without noisy policy π_θ .

ran the agent ten times for 10000 steps without noise contribution, and observed 311 ± 3 wins on average, and no lost game at all, which indicates that the neural network learned to generalize the task well.

6 Conclusion

6.1 Contributions

6.2 Future Work

Part I

Appendix

A Software Installation

Since all of the software is freely available on GitHub, you can always find a build section within the provided readme file there. The links to the respective GitHub folders are provided within each following section. For the case of an error that may occur during the build procedure, do no hesitate to open up an issue there. I will then help you to get the software running on your device.

A.1 Build the Pattern Generator

The pattern generator can be found on GitHub at the following [link](#). Proceed as described below.

1. Make sure all dependencies are installed. The pattern generator only requires the necessary dependencies (section A.5.1). For communication with the real robot, one further needs the real robot and simulation dependencies (section A.5.2, and the simulation models A.4). For the deep learning support, it is further required to have the deep learning dependencies installed (section A.5.3).
2. Clone the pattern generator from GitHub with
`https://github.com/mhubii/nmpc_pattern_generator.git`
3. Then do one of the following steps
 - a) To just build the pattern generator, do
`mkdir build && cd build`
`cmake ..`
`make`
 - b) To build the pattern generator with communication to the real robot or the simulation do
`mkdir build && cd build`
`cmake -DCMAKE_BUILD_WITH_YARP=ON`
`make`
 - c) To build the pattern generator with deep learning support do
`mkdir build && cd build`
`cmake -DBUILD_WITH_LEARNING=ON \`
`-DCMAKE_PREFIX_PATH=/absolute/path/to/libtorch ..`
`make`

4. This is not necessary, but you can then install or uninstall the pattern generator with
`make install`
`make uninstall`

The pattern generator comes with some tests and examples. They can be executed as follows

1. To run the tests do
`cd build/bin`
`./pattern_generator_tests`
2. To run an example do
`cd build/bin`
`./nmpc_pattern_generator_example`
3. The results may be visualized with
`cd plot`
`python plot_pattern.py`

A.2 Build the Android Joystick App

The Joystick app can be found on GitHub at the following [link](#). Proceed as described below.

1. Clone the Android app from GitHub with
`git clone https://github.com/mhubii/ijoy.git`
2. Copy the `.apk` file that you cloned from GitHub to your Android smartphone.
3. Make sure your device allows installation of apps from unknown sources. Under Android 7:
 - a) Go to settings and open lock `screen&security`.
 - b) Find entry `unkown sources` and enable it.
4. Find the `.apk` using the file browser of your choice and execute it (figure A.1 (a) and (b)).
5. Follow the on screen instructions and wait for the installation to finish (figure A.1 (c) and (d)).

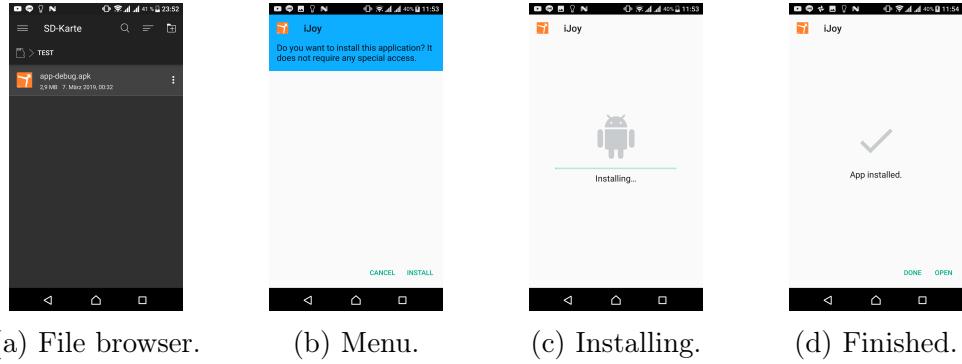


Figure A.1: Installation process.

A.3 Build Proximal Policy Optimization

The proximal policy optimization can be found on GitHub at the following [link](#). Proceed as described below.

1. Make sure the C++ API of PyTorch got install as described in section A.5.
2. Clone proximal policy optimization from GitHub with

```
git clone https://github.com/mhubii/ppo_libtorch.git
```
3. Then do

```
mkdir build && cd build
cmake -DCMAKE_PREFIX_PATH=/absolute/path/to/libtorch ..
make
```

You can then train and evaluate the neural network as described below.

1. Train the neural network with

```
cd build
./train_ppo
```
2. Test the trained neural network with

```
cd build
./test_ppo
```
3. Visualize the results with

```
python plot.py
```

A.4 Build Simulation Models

The simulation models can be found on GitHub at the following [link](#). Proceed as described below.

1. Clone the models from GitHub with

```
git clone https://github.com/mhubii/gazebo_models.git
```
2. You can either install the models to a location that Gazebo knows, or update the path, where Gazebo is looking for models.
3. To update the path add following line to your `bashrc`

```
export GAZEBO_MODEL_PATH=<>/gazebo_models:$GAZEBO_MODEL_PATH
```

Therein, replace `<>` by the location to which you cloned the repository.
4. To install the models do

```
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=~/.gazebo ..
make install
```
5. To uninstall the models do

```
cd build
make uninstall
```

A.5 Build Third Party Software

The software that was developed as part of this thesis has some third party dependencies, of which not all are necessary but required for some additional features.

A.5.1 Necessary Dependencies

These dependencies need to be installed.

Eigen

1. The pattern generator is based on the blazingly fast Eigen library. To install it do

```
sudo apt install libeigen3-dev
```
2. You may need to create a symbolic link

```
sudo ln -s /usr/include/eigen3/Eigen/ /usr/include/
sudo ln -s /usr/include/eigen3/unsupported/ /usr/include/
```

qpOASES

To solve the sequential quadratic program, we need to install qpOASES. Please follow the [install instructions](#), or head on as described below

1. Download qpOASES


```
wget https://www.coin-or.org/download/source/qpOASES/qpOASES-3.2.1.zip
unzip qpOASES-3.2.1.zip
cd qpOASES-3.2.1
```
2. Now since we want a shared library, in the CMakeLists.txt change ADD_LIBRARY(qpOASES STATIC \${SRC}) to ADD_LIBRARY(qpOASES SHARED \${SRC})
 Then proceed with


```
mkdir build && cd build
cmake ..
make
sudo make install
```

YAML

The configurations are read in using the YAML file format. Run the command

1.

```
sudo apt install libyaml-cpp-dev
```

A.5.2 Real Robot and Simulation Dependencies

To run the NMPC generator on a real robot or the simulation, we will need to install some more dependencies.

Gazebo

The simulation environment can be installed according to the install instructions. The main steps are

1.

```
sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable"\`"lsb_release -cs" main' > /etc/apt/sources.list.d/gazebo-stable.list'
wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
sudo apt-get update
sudo apt-get install gazebo9
sudo apt-get install libgazebo9-dev
```

RBDL

The rigid body kinematics are solved with RBDL. To install RBDL, do

1.

```
hg clone https://bitbucket.org/rbdl/rbdl
cd rbdl
```

```
hg checkout dev  
mkdir build && cd build  
cmake -DCMAKE_BUILD_TYPE=Release \  
-DRBDL_BUILD_ADDON_URDFREADER=ON ..
```

YARP

Additionally, for communicating with the real robot, or the simulation, we need YARP. To install YARP, follow the [installation instructions](#), or head on as described below

1.

```
git clone https://github.com/robotology/yarp.git  
cd yarp && mkdir build && cd build
```
2. If you have previously installed Anaconda, YARP may complain here. Go and install OpenCV within your Anaconda distribution

```
# activate your anaconda environment, if you followed the  
# instructions before in PyTorch, do conda activate py37_torch  
conda install opencv
```
3. Then do

```
cmake -DOpenCV_DIR=$HOME/anaconda3/envs/py37_torch/share/OpenCV ..  
make  
sudo make install
```

Gazebo YARP Plugins

Plugins for Gazebo are used to clone the behavior of the real robot into the simulation environment. Proceed as below

1.

```
git clone https://github.com/robotology/gazebo-yarp-plugins.git  
cd gazebo-yarp-plugins  
mkdir build && cd build  
cmake -DCMAKE_INSTALL_PREFIX=$HOME/gazebo-yarp-plugins ..  
make  
make install
```
2. Next, you need to tell Gazebo where to find the plugins, therefore add following to the `bashrc`

```
export GAZEBO_PLUGIN_PATH=${GAZEBO_PLUGIN_PATH}  
:$HOME/gazebo-yarp-plugins/lib
```

Gazebo Models

The models are used within the simulation environment Gazebo. See section A.4.

NCurses

For the visualization of the control pannel, we need to install ncurses, do

```
1. sudo apt install libncurses5-dev
```

A.5.3 Deep Learning Dependencies

To learn Nonlinear Model Predictive Control or simple navigation on top of Nonlinear Model Predictive Control, we will need to install PyTorch. For PyTorch to work in combination with RBDL, we need a source installation. Please checkout this gist to figure out how to perform a clean setup.

B Startup and Shutdown Procedures

In this chapter, we will briefly document how to run the software that was developed within this thesis. In section B.1, we will introduce how the Heicub robot is supposed to be started, while in the subsequent section B.2, we will explain how to run the robot within the simulation environment Gazebo. Furthermore, once the robot has been started, either in real or in simulation, it is possible to use the provided pattern generator for control in realtime with the terminal interface or the Android joysitck app. Both possibilities will be explained in section B.3. Finally, a short demonstration for the behavioral augmentation is given in section B.4.

B.1 Real Robot Startup

1. Turn on the icubsrv (figure B.1), username is `icub`, password is `icub`.



Figure B.1: The icubsrv is the Dell laptop. Below it the switch.

2. Turn on the power suppliers (figure B.2), and keep the red button pressed (figure B.7 (a)). The suppliers should initially provide 13V and 0V, if not, do no proceed.



(a) Turn on these buttons. (b) Expected initial voltage.

Figure B.2: Power suppliers.

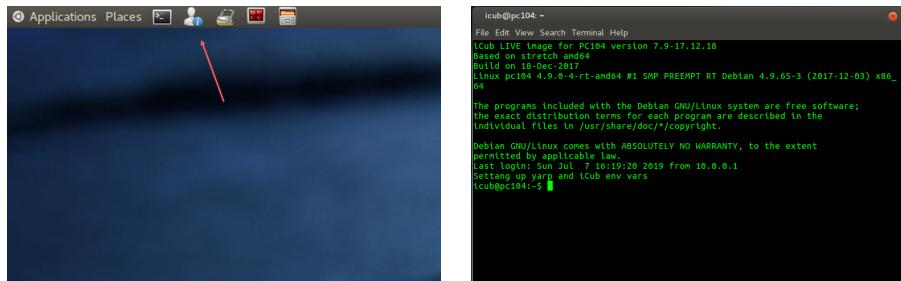
3. Turn on the pc104 (figure B.3 (a)). At this step you may want to wait up to 5 minutes, to give the board computer enough time to start.



(a) Turn on the CPU of pc104. (b) Turn on the motors of Heicub.

Figure B.3: Heicub's switches.

4. On the icubsrv, connect via ssh to pc104. Therefore, click on the highlighted symbol within figure B.4 (a). If it fails to connect, turn off the CPU, and go back to the previous step.



(a) Run ssh to connect to pc104. (b) Terminal to pc104.

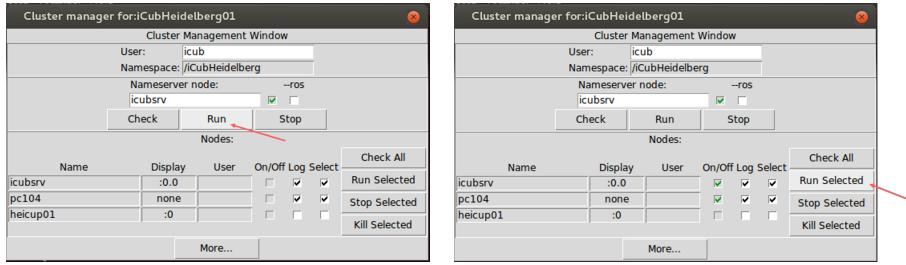
Figure B.4: Connect to pc104.

5. Run the cluster manager from a new terminal on the icubsrv, therefore do

```
cd /usr/local/src/robot/icub-main/build-pc104/bin
python icub-cluster.py
```
6. Within the cluster manager, run the yarp name server, and then yarp on all other devices (figure B.5 (a), then (b)).
7. We can now turn on the motors and the cameras, as well as to connect our own laptop, which is explained in the following sections.

B.1.1 Start the Motors

If the real robot is up and running (section B.1), you can now start Heicub's motors. Therefore proceed as described below.



(a) Run the yarp nameserver.

(b) Run yarp on other devices.

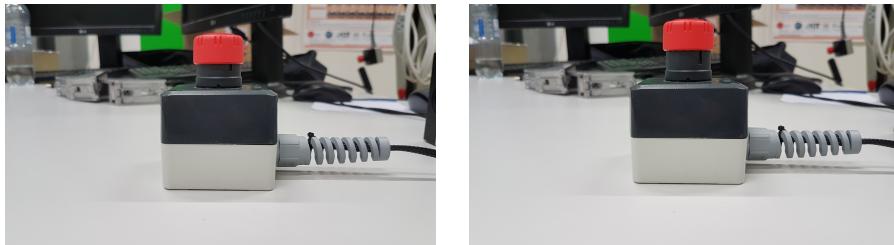
Figure B.5: Run yarp.

1. Turn on the motors (figure B.3 (b)). The power suppliers should now show 13V and 40V. Wait until the blue lights at Heicub stopped blinking (figure B.6).



Figure B.6: Blue motor lights.

2. Release the safety button (figure B.7 (b)).



(a) Pressed safety button.

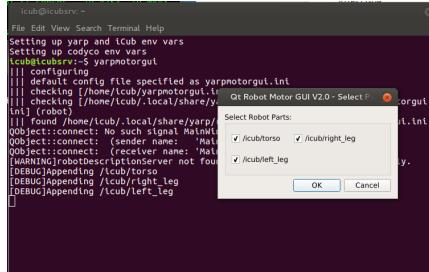
(b) Released safety button.

Figure B.7: Safety button.

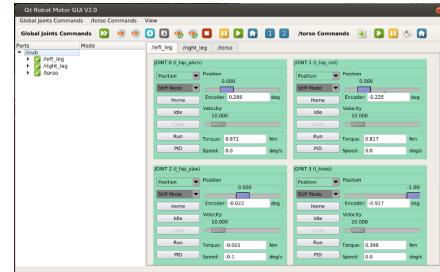
3. Within your terminal to pc104 (figure B.4 (b)) run the command `yarprobotinterface`
This will run all the motor drivers and connect them to the yarp network.
4. If no errors occurred, we can now run the `yarpmotorgui` to play around with the motors. This step is not necessary. To run the `yarpmotorgui`, open a new terminal on the icubsrv, and run the command

yarpmotorgui (figure B.8 (a))

Then, press **OK**. The motors can now be manipulated from within the GUI (figure B.8 (b)), e.g. by clicking on the house, which will bring all motors to the home position.



(a) Run the yarpmotorgui.



(b) The yarpmotorgui.

Figure B.8: Accessing the motors.

- The motors may need to be calibrated when the robot runs for a long time. Therefore, lift up the robot, and bring it to the home position via the yarp-motorgui. Then, start a new terminal on pc104 and run the command
`yarp rpc /wholeBodyDynamics/rpc`
In the interface that will open up, type
`calib all 300`.

B.1.2 Start the Cameras

If the real robot is up and running (section B.1), you can now start Heicub's cameras. Therefore proceed as described below.

1. Within a new terminal to pc104 (figure B.4 (a)), do

```
cd ./local/share/yarp/robots/iCubHeidelberg01/camera
```

Then, start the camera via

```
yarpdev --from dragonfly2_config_left.ini
```

This will run the left camera and connect it to the yarp network. Repeat the above steps for the right camera, but replace left by right within the `.ini` file.

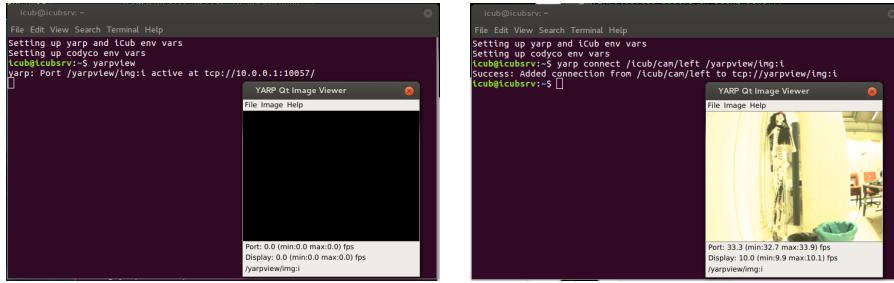
2. If no errors occurred, we can now run a yarpview to see what Heicub sees. This step is not necessary. To run a yarpview, open a new terminal on the icubsrv, and run the command

yarpview (figure B.9 (a))

Then, connect a camera to the yarpview. Therefore, open a new terminal on the icubsrv and run the command

```
yarp connect /icub/cam/left /yarpview/img:i (figure B.9 (b))
```

You should now see an image.



(a) Run a yarpview.

(b) Connect a camera to the yarpview.

Figure B.9: Accessing the cameras.

B.1.3 Connect your own Laptop

Make sure you installed YARP, as described in section A.5. You can then connect your own laptop via ethernet to the same network as Heicub. Therefore, proceed as described below.

1. Connect a LAN cable to the switch to which the icubsrv is connected as well (figure B.1). You will then get assigned an IP address within the same domain as the icubserver, such as `10.0.0.x`. Check this by running `ifconfig`. The IP address of the icubsrv is `10.0.0.1`, while that of the pc104 is `10.0.0.2`.
2. Check the connection by pinging the icubsrv via `ping 10.0.0.1`. If this works, skip this point, otherwise you can create a manual connection. Therefore, search for `Network Connections` among your applications and open it. Then, click `Add`. If it is not set already, check the hardware address by running `ifconfig`, it should show something similar to `HWaddr 9C:EB:E8:B2:AB:27`, choose this as your device. Then go to IPv4 settings and set the IP address to `10.0.0.3`, the netmask to `24`, and the gateway to `10.0.0.255`. Then press save.
3. If you connected successfully, open a terminal on your device and run `yarp detect --write`
If it does not find the running yarpserver, manually configure the connection via the following commands from a terminal
`yarp conf 10.0.0.1 1000`
`yarp namespace /iCubHeidelberg`
`yarp detect`

B.2 Simulated Robot Startup

Make sure you installed Gazebo, YARP, and the Gazebo YARP plugins as described in section A.5. Also, you need to have the simulation model installed, which is

described in section A.4. When these requirements are satisfied, then proceed as described below.

1. Open a terminal and start YARP via
`yarpserver --write`
2. Open another terminal and start Gazebo via
`gazebo -s libgazebo_yarp_clock.so`
The clock library will thereby synchronize the YARP clock, and the Gazebo clock.
3. In Gazebo, go to the `Insert` bar, and insert `heicub_without_weights`.

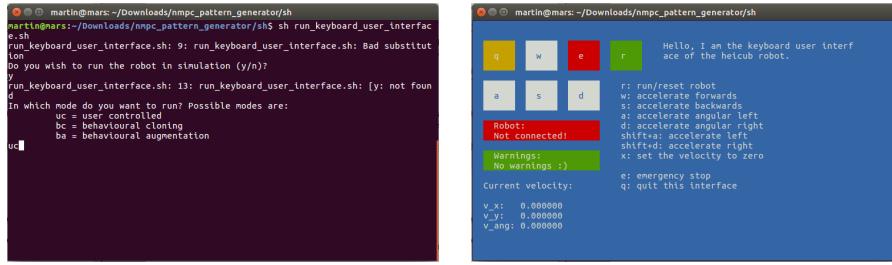
B.3 Start the Pattern Generator

Make sure you installed the pattern generator library as described in section A.1. The robot needs to be running, either in real (section B.1, and section B.1.1) or in simulation (section B.2). By construction, the startup procedure for the simulation and the real robot is the same. You can choose to control the robot via the terminal, or the provided Android app. Both possibilities are described below in section B.3.1, and section B.3.2, respectively.

B.3.1 Control via the Terminal

The terminal user interface comes with the pattern generator, so there is no additional software that needs to be installed. Proceed as described below.

1. Open a new terminal and go to the shell scripts within the pattern generator folder via
`cd nmpc_pattern_generator/sh`
On the icubsrv, this folder is located at `/usr/local/src/robot`. Then, run the keyboard user interface via
`sh run_keyboard_user_interface.sh`
2. The shell script will then ask you whether to run the robot in real or in simulation, write `y` or `n` and press enter (figure B.10 (a)).
3. The shell script will then ask you for the mode to run in. Write `uc` and press enter (figure B.10 (a)).
4. The user interface should now show up and explain how to control the robot (figure B.10 (b)).



(a) Select the mode to run the pattern generator in.

(b) Keyboard user interface.

Figure B.10: Run the keyboard user interface.

B.3.2 Control via the Android Joystick App

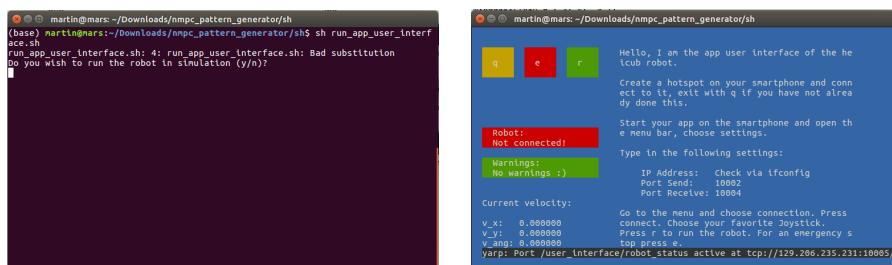
Make sure you installed the Android joystick app as described in section A.2. Proceed as described below.

1. Open a hotspot from your phone and connect the icubsrv or your laptop to it.
2. Open a new terminal and go to the shell scripts within the pattern generator folder via

```
cd nmpc_pattern_generator/sh
```

On the icubsrv, this folder is located at `/usr/local/src/robot`. Then, run the app user interface via

```
sh run_app_user_interface.sh
```
3. The shell script will then ask you whether to run the robot in real or in simulation, write `y` or `n` and press enter (figure B.11 (a)).
4. The user interface should now show up and explain how to control the robot (figure B.11 (b)).

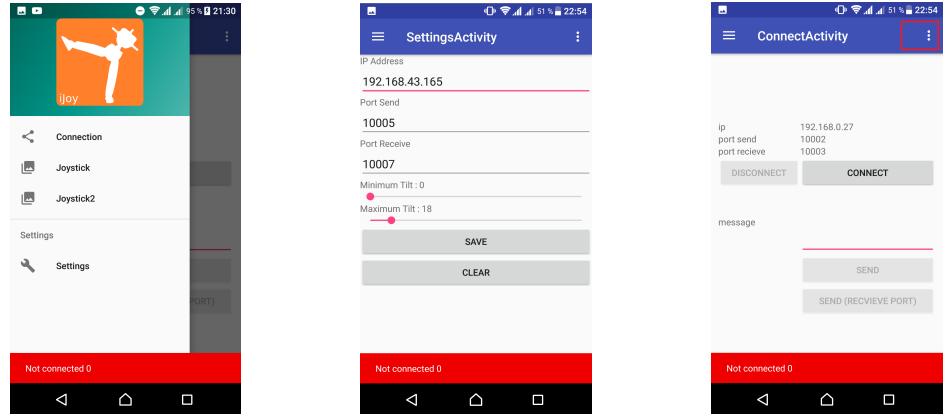


(a) Select the mode to run the pattern generator in.

(b) App user interface.

Figure B.11: Run the app user interface.

5. Open the Android app on your smartphone. Within the app, choose settings from the navigation drawer (figure B.12 (a)). In the settings activity (figure B.12 (b)), choose the IP address and the ports as proposed by the app user interface (figure B.11 (b)).



(a) Navigation drawer. (b) Settings activity. (c) Connect activity.

Figure B.12: Connect the app to the computer.

6. Now go to the connection activity (figure B.12 (c)), and press connect. You should now be connected.
7. Go to the app user interface on the laptop (figure B.11 (b)), and press `r` to run the pattern generator.
8. Within the navigation drawer, choose from one of two joysticks to control the robot.

B.4 Start the Behavioral Augmentation Demo

There is a trained neural network available on heicub01 to run a demonstration of the behavioral augmentation as trained in section 5.2. The network will navigate Heicub towards a fire extinguisher, and look around if it does not see one. Make sure the robot and its cameras are running (section B.1, section B.1.1, and section B.1.2). Proceed as described below.

1. Login to heicub01, username is `icub`, password is `icubheidelberg01`.
2. Make sure that YARP is running on heicub01 (figure B.5 (b)). Therefore, select heicub01 and click `Run Selected`.
3. On heicub01, open a new terminal and go to the shell scripts within the pattern generator folder via
`cd /home/icub/Documents/nmpc_pattern_generator/sh`

Then run the keyboard user interface with

```
sh run_keyboard_user_interface.sh
```

4. The shell script will then ask you whether to run the robot in real or in simulation, write `n` and press enter (figure B.10 (a)).
5. The shell script will then ask you for the mode to run in. Write `ba` and press enter.
6. The user interface should now show up (figure B.10 (b)). Press `r` to run the pattern generator. The robot will now be controlled by the trained neural network.

B.5 Real Robot Shutdown

Before you shutdown the robot, make sure it is in a safe position, that is, lift it up. Proceed as described below.

1. Lift the robot from the floor.
2. Close all running applications. `Ctrl+C` the yarrobotinterface and the camera interfaces.
3. Type `sudo poweroff` in a terminal that is connected to pc104.
4. Turn off the motors.
5. Turn off the CPU of pc104.
6. Turn off the power suppliers.
7. Just to be sure, press the red button.

C Lists

C.1 List of Figures

3.1	Simplified version of the proposed control loop to navigate the robot with either a human user or an artificial agent.	10
3.2	Building blocks of the pattern generation. To understand the greater picture, a connection can be drawn to fig. 3.1, where the orange box represents the one shown in this figure.	11
3.3	Forces acting on the sole.	13
3.4	Full support polygon, and the resulting support polygon with security margin (dashed lines).	13
3.5	Linear inverted pendulum with a support polygon (a), and the corresponding free body diagram with cutting forces $\mathbf{S}_{x/y/z}$ (b).	14
3.6	Force torque sensors at the foot's ankle.	16
3.7	The foot's support polygon, which is described by the position vectors \mathbf{p}_i	24
3.8	Uninterpolated trajectories (a), as obtained from the nonlinear model predictive control, and interpolated trajectories (b) for the feet and the center of mass.	28
3.9	Biological neuron, which connects its cell body to dendrites of surrounding neurons via an axon. [17]	32
3.10	Fully connected neural network with three inputs and four outputs. Each orange circle represents what is often referred to as neuron, while the black lines indicate the connections between each neuron.	33
3.11	Simple interpretation of a fully connected neural network with one layer that takes $\mathbf{x} = (x_0 \ x_1)^T$ as input. The dotted lines are isolines to the hyperplane, which showcase the effect of the bias \mathbf{b}	34
3.12	Convolutional neural network with a total of three layers of which one is the input layer. For visualization, the kernel size is set to be three, and the stride is set to be one.	34
3.13	Analysis of neurons with highest activation, corresponding to a subset of ImageNet. For the first layer, the kernel itself is shown, and image patches at the kernel's scale. For the second and third layer, a single neuron with highest activation is upscaled by transposed convolutions to the first layer's feature map. Images taken from [29].	35
3.14	Long short-term memory unit with cell states \mathbf{c}_i , hidden states \mathbf{h}_i , input \mathbf{x}_t , and activation functions σ/\tanh , as well as addition + and multiplication \times operators.	35

3.15	Chain of long short-term memory units for temporal understanding of the input sequence \mathbf{x}_i	36
3.16	Pipeline for behavioral cloning. The neural network is trained on stored RGBD images, and corresponding velocity commands that are correlated by a timestamp	37
3.17	Reinforcement learning setup. As the agent interacts with the environment, the state of the environment changes.	38
3.18	Policy gradient clipping in proximal policy optimization for positive and negative advantage estimates \hat{A}_t	39
3.19	Image preprocessing to obtain edge filtered images. The images were taken within the simulation environment Gazebo (link), and show a space exploration vehicle, for which, with the friendly support of NASA, we generated a Gazebo version (link).	41
3.20	The stereo setup with a left and a right camera.	42
3.21	Generation of the left disparity map by the block matching algorithm.	42
3.22	Generation of the confidence map from the variance within the disparity map.	43
3.23	Generation of the confidence weighted least squares disparity from the disparity map, and the left right consistency.	45
3.24	Pinhole camera model.	46
3.25	Calibration pattern as observed from the camera's coordinate system C . Within the object's coordinate system, all chessboard corners lie at a zero z -position.	47
3.26	Distorted calibration pattern (a), and the image points as found by the algorithm (b).	47
3.27	Undistorted calibration pattern, as observed by the left camera (a), and by the right camera (b). For comparison, see the distorted calibration pattern in figure 3.26, and note how the horizons within the images align.	48
5.1	Simple trajectories. The velocities are given in units of $(\text{m/s } \text{m/s } \text{rad/s})^T$	51
5.2	Advanced trajectories. The velocities are given in units of $(\text{m/s } \text{m/s } \text{rad/s})^T$	51
5.3	Interpolated foot trajectories.	52
5.4	User controlled straight walk.	53
5.5	User controlled curved walk.	53
5.6	User controlled obstacle avoidance.	53
5.7	User controlled environment scanning.	54
5.8	Depth map extraction without calibration. The parameters were set as follows to $N = 13$, $D = 32$, $\sigma = 1$, and $\lambda = 10^4$	54
5.9	Exemplary left and right camera views of the calibration pattern as acquired during the calibration process. The colorful points indicate the detected corners in the image plane. Refer to figure 3.26 for the theory.	55

5.10 Rectified and original view of the stereo camera. Refer to figure 3.27 for the theory.	56
5.11 Heicub's perspective of the scene for the depth map parameter tuning.	57
5.12 Left disparity map and confidence weighted least squares disparity for changing SAD window sizes N and number of disparities D	57
5.13 Confidence weighted least squares disparity for changing energy weight- ings σ and λ . Please refer to equations 3.140 and 3.141 for the theory.	58
5.14 Samples of the recorded dataset. The dataset shows a very diverse number of situation for the neural network to learn from.	59
5.15 UNet-LSTM network architecture.	60
5.16 Mean squared error training loss history.	61
5.17 Normalized velocity histograms over the validation split.	61
5.18 Heicub's behavior in the test environment for benchmarking tasks.	62
5.19 Autonomously controlled straight walk.	62
5.20 Autonomously controlled curved walk.	63
5.21 Autonomously controlled obstacle avoidance.	63
5.22 Autonomously controlled environment scanning.	63
5.23 Influence of entropic commands onto Heicub's dynamic balance.	64
5.24 Heicub's behavior in the test environment for additional tasks.	65
5.25 Autonomously controlled in a dynamic environment.	65
5.26 Autonomously controlled for demonstration of semantic understanding.	65
5.27 Artificial agent in proximal policy optimization test environment.	66
5.28 Proximal policy optimization in test environment over 50 epochs.	66
5.29 Proximal policy optimization in test mode, that is without noisy pol- icy π_θ	67
A.1 Installation process.	72
B.1 The icubsrv is the Dell laptop. Below it the switch.	77
B.2 Power suppliers.	77
B.3 Heicub's switches.	78
B.4 Connect to pc104.	78
B.5 Run yarp.	79
B.6 Blue motor lights.	79
B.7 Safety button.	79
B.8 Accessing the motors.	80
B.9 Accessing the cameras.	81
B.10 Run the keyboard user interface.	83
B.11 Run the app user interface.	83
B.12 Connect the app to the computer.	84

C.2 List of Tables

5.1 Parameters which used to work best on Heicub.	52
---	----

5.2	Intrinsic parameters of single cameras. These parameters can be found as YAML file on GitHub (link)	55
5.3	Rectification transforms \mathbf{R}_i , and projection matrices \mathbf{P}_i , for the left, and the right camera, respectively. These parameters can be found as YAML file on GitHub (link).	56

D Bibliography

- [1] Miomir Vukobratovic and Davor Juricic. Contribution to the synthesis of biped gait. *IFAC Proceedings Volumes*, 2:469–478, 1968.
- [2] Miomir Vukobratovic and Davor Juricic. Contribution to the synthesis of biped gait. *IEEE Transactions on Biomedical Engineering*, pages 1–6, 1969.
- [3] Jin-ichi Yamaguchi, Atsuo Takanishi, and Ichiro Kato. Development of a biped walking robot compensating for three-axis moment by trunk motion. *Journal of the Robotics Society of Japan*, 11(4):581–586, 1993.
- [4] Miomir Vukobratović and Branislav Borovac. Zero-moment point—thirty five years of its life. *International journal of humanoid robotics*, 1(01):157–173, 2004.
- [5] Kazuo Hirai, Masato Hirose, Yuji Haikawa, and Toru Takenaka. The development of Honda humanoid robot. *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No. 98CH36146)*, 2:1321–1326, 1998.
- [6] Anirvan Dasgupta and Yoshihiko Nakamura. Making feasible walking motion of humanoid robots from human motion capture data. *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C)*, 2:1044–1049, 1999.
- [7] Shuuji Kajita, Fumio Kanehiro, Kenji Kaneko, Kiyoshi Fujiwara, Kensuke Harada, Kazuhito Yokoi, and Hirohisa Hirukawa. Biped walking pattern generation by using preview control of zero-moment point. *2003 IEEE International Conference on Robotics and Automation (Cat. No. 03CH37422)*, 2:1620–1626, 2003.
- [8] Shuuji Kajita, Hirohisa Hirukawa, Kensuke Harada, and Kazuhito Yokoi. *Introduction to humanoid robotics*. Springer, 2014.
- [9] Qiang Huang, Kazuhito Yokoi, Shuuji Kajita, Kenji Kaneko, Hirohiko Arai, Noriho Koyachi, and Kazuo Tanie. Stability compensation of a mobile manipulator by manipulator motion: Feasibility and planning. *IEEE Transactions on robotics and automation*, 17(3):280–289, 2001.
- [10] Raphael Michel. *Dynamic filter for walking motion corrections*. Bachelor’s thesis, Heidelberg University, 2017.

- [11] Paul T. Boggs and Jon W. Tolle. Sequential quadratic programming. *Acta numerica*, 4:1–51, 1995.
- [12] K. Schittkowski. Solving constrained nonlinear least squares problems by a general purpose SQP-method. In *Trends in Mathematical Optimization*, volume 83, pages 295–309. Springer, 1988.
- [13] Andrei Herdt, Holger Diedam, Pierre-Brice Wieber, Dimitar Dimitrov, Katja Mombaur, and Moritz Diehl. Online walking motion generation with automatic footstep placement. *Advanced Robotics*, 24(5):719–737, 2010.
- [14] Maximilien Naveau, Manuel Kudruss, Olivier Stasse, Christian Kirches, Katja Mombaur, and Philippe Soueres. A reactive walking pattern generator based on nonlinear model predictive control. *IEEE Robotics and Automation Letters*, 2(1):10–17, 2016.
- [15] Andrei Herdt, Nicolas Perrin, and Pierre Brice Wieber. Walking without thinking about it. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 190–195. IEEE, 2010.
- [16] Jorge J More. The Levenberg-Marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116, 1978.
- [17] Mikael Haggstrom and Others. Medical gallery of Mikael Haggstrom 2014. *WikiJournal of Medicine*, 1(2), 2014.
- [18] Kyoung-Su Oh and Keechul Jung. GPU implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.
- [19] Alan L Hodgkin and Andrew F Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500–544, 1952.
- [20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.
- [21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [22] Juyang Weng, Narendra Ahuja, and Thomas S Huang. Cresceptron: a self-organizing neural network which grows adaptively. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, volume 1, pages 576–581, 1992.
- [23] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

- [24] SS Viglione. 4 Applications of pattern recognition technology. In *Mathematics in Science and Engineering*, pages 115–162. 1970.
- [25] A G. Ivakhnenko. Polynomial theory of complex systems. *IEEE transactions on Systems, Man, and Cybernetics*, (4):364–378, 1971.
- [26] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.
- [27] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.
- [28] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255, 2009.
- [29] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.
- [30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.
- [31] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [32] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with LSTM. *IET*, 1999.
- [33] Seppo Linnainmaa. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. *Master’s Thesis (in Finnish)*, Univ. Helsinki, pages 6–7, 1970.
- [34] David J Montana and Lawrence Davis. Training Feedforward Neural Networks Using Genetic Algorithms. In *IJCAI*, pages 762–767, 1989.
- [35] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, and Others. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.
- [36] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, and Others. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

- [37] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.
- [38] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [39] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.
- [40] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*. 1998.
- [41] Dongbo Min, Sunghwan Choi, Jiangbo Lu, Bumsub Ham, Kwanghoon Sohn, and Minh N. Do. Fast global image smoothing based on weighted least squares. *IEEE Transactions on Image Processing*, 23(12):5638–5653, 2014.
- [42] Rostam Affendi Hamzah, Rosman Abd Rahim, and Zarina Mohd Noh. Sum of absolute differences algorithm in stereo correspondence problem for stereo matching in computer vision application. In *2010 3rd International Conference on Computer Science and Information Technology*, volume 1, pages 652–657, 2010.
- [43] Irwin Sobel. An Isotropic 3x3 Image Gradient Operator. *Presentation at Stanford A.I. Project 1968*, 2014.
- [44] Geoffrey Egnal, Max Mintz, and Richard P. Wildes. A stereo confidence metric using single view imagery with comparison to five alternative approaches. *Image and vision computing*, 22(12):943–957, 2004.
- [45] Carlo Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. In *Iccv*, volume 98, page 2, 1998.
- [46] C Brown Duane. Close-range camera calibration. *Photogramm. Eng*, 37(8):855–866, 1971.
- [47] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.
- [48] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Transactions on pattern analysis and machine intelligence*, 22, 2000.
- [49] Charles Loop and Zhengyou Zhang. Computing rectifying homographies for stereo vision. In *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*, volume 1, pages 125–131, 1999.

- [50] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, pages 2149–2154. IEEE, 2004.
- [51] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. YARP: yet another robot platform. *International Journal of Advanced Robotic Systems*, 3(1):8, 2006.
- [52] G. Bradski. The OpenCV Library, 2000.
- [53] Gael Guennebaud and Jacob Benoit. Eigen v3, 2010.
- [54] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.
- [55] Hans Joachim Ferreau, Christian Kirches, Andreas Potschka, Hans Georg Bock, and Moritz Diehl. qpOASES: A parametric active-set algorithm for quadratic programming. *Mathematical Programming Computation*, 6(4):327–363, 2014.
- [56] Martin L Felis. RBDL: an efficient rigid-body dynamics library using recursive algorithms. *Autonomous Robots*, 41(2):495–511, 2017.
- [57] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain’t markup language (yamlTM) version 1.1. *yaml.org, Tech. Rep*, page 23, 2005.
- [58] Greg Miller, Gennadiy Civil, and Carlos O’Ryan. Google Test, 2013.
- [59] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.
- [60] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Erklärung:

Ich versichere, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den (Datum)