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 neuronalen Netzwerken 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 Neural Networks 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

Implementierung und Fusion von modellprädiktiver Regelung mit neuronalen Netzwerken zur autonomen Navigation von humanoiden Robotern:

In dieser Arbeit erkunden wir die Fähigkeit von neuronalen Netzwerken für die autonome Navigation von humanoiden Robotern. Hierfür wird eine nichtlineare, Modellprädiktive Regelung, die es ermöglicht, stabile Lauftrajektorien in Echtzeit zu erzeugen, implementiert und evaluiert. Darauf folgend werden die neuronalen Netzwerke auf zwei verschiedene Methoden trainiert, nämlich per Verhaltensklonung und per selbstverstärkendem Lernen. Die erste von beiden beruht auf stereo RGB Bildern und es wird demonstriert, dass die Einführung von Tiefenbildern in die Bildverarbeitung genügt, 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. Weiterhin wird gezeigt, wie sich das selbstverstärkende Lernen in einer Simulation trainieren lässt und dabei die Entstehung von selbst erlernten Strategien beobachtet, welche sich äquivalent auf den echten Roboter übertragen lassen. Alle Experimente werden an Heicub, einer Variente des iCub, durchgeführt, welcher speziell für Optimalsteuerung in der Fortbewegung am Istituto Italiano di Tecnologiia in Genove entwickelt wurde. Die Auswertung von Balancekriterien zeigt schließlich, dass ein menschlicher Kontrolleur, einem künstlichen Agenten gegenüber, nicht überlegen ist.

Implementation and Fusion of Nonlinear Model Predictive Control with Neural Networks for Autonomous Navigation of Humanoid Robots:

In this work, the capability of neural networks for autonomous navigation of humanoid robots is investigated. Therefore, a nonlinear model predictive control that allows for real-time generation of balanced walking trajectories is implemented and evaluated. Following that, the neural networks are trained on autonomous navigation in two different ways, namely via behavioral cloning, and via reinforcement learning. The first of which relies on stereo RGB images as input, and it is demonstrated that the introduction of depth maps to the vision pipeline is 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. The reinforcement learning approach is then shown to be trainable in a simulation environment, and the emergence of rich self-taught policies is observed, which are equally applicable to the real robot. 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 balance criteria finally reveals that there is no superiority of a human controller over an artificial agent.

Contents

1 Introduction

The development of humanoid robots has ever since posed an issue of special interest to researchers. Not only is this ambition driven by human nature, which tries to understand its surroundings, including itself, but further by two additional arguments. One of which relies on the Darwinian view of biology, which regards evolution in itself as an optimization that ultimately led to two-legged motion as a global optimum. While this is true to some extent, we also need to consider that human imagination is very much bound to experience, and that there may even be more optimal solutions, which are completely suppressed by the dominance of humankind. The second argument mostly relates to the potential use of humanoid robots that is caused by artificial environmental features, which got developed by and moreover for humans. As thus, future research on humanoid robots will mainly aim at replacing human workforce from humanly designed environments including but not limited to package deliveries [1], airplane construction [2], applications in space, military, and for rescue missions. Ultimately, this will unlock the potential to keep humans away from hazardous encounters to avoid situations as they occurred in Chernobyl, where about 600000 [3] so-called liquidators, or simply put men, were exhibited dangerous radiation to remove radioactive material from the rooftops. Robots could have similarly been used during the Fukushima Daiichi nuclear disaster, for why the Defense Advanced Research Projects Agency (DARPA) pushed forward the development of humanoid robots. Furthermore, will they help to maintain our current living standards, which are threatened by the relative loss of workforce as a result of the demographic change.

All the above stated possible applications for humanoid robots have two things in common. To achieve them, one must investigate methods that let the robots walk through the environments without falling, and additionally one has to figure how this ability can be used to achieve certain goals while ensuring safe interaction with humans. The achievement and the combination of these aims may equivalently be regarded as the achievement of motion intelligence. Given a suitable walking pattern generator, there have been several attempts to let humanoid robots navigate the environment in a goal-driven and intelligent way. First approaches relied on methods, which were well established for wheeled robots, among them heuristic search algorithms or artificial potential fields [4]. Other methods introduced simultaneous localization and mapping to estimate the robot's camera position under the assumption of a linear velocity model [5], which led to visual servoing that took the robot's dynamics into account [6], and that got soon extended into the walking pattern generation itself to walk towards relative positions to defined objects [7]. More

recent methods demonstrated impressive results, based on expensive Lidar sensors [8]. However, none of these methods distinguishes within the nature of objects, only the introduction of rule-based fuzzy logic [9] may then trigger a variety of behaviors for different objects. All these approaches have been furthermore tailored to rather specific tasks, in that they only provide smart navigation but for example, no object manipulation. Meanwhile, the introduction of neural networks may provide a suitable solution to tackle all these unsolved issues at once, yet there has not been research on it. The contribution of this work is, therefore, to explore the use of neural networks for the achievement of motion intelligence, given inexpensive camera sensors.

To contribute to the just stated state of the art research, a twofold objective is to be achieved within the scope of this thesis. The first sub-objective is the implementation of a nonlinear model predictive control for the generation of walking patterns, which is mainly based on the works of [23], but uses a slightly modified cost function. The second objective is the design and training of a neural network architecture, which can control the pattern generation at a high level, just as a human user could. Within this thesis, the designed neural networks are trained via two different approaches. The first approach is based on behavioral cloning [45], while the second utilizes reinforcement learning [48]. To reach the objectives, a broad understanding of the underlying concepts is required, which is explained in depth in the background section 2. Methods with which to implement the background, and a short note on Heicub, the humanoid robot which we used, are consequentially presented in the subsequent section 3. Finally, experiments are carried out to evaluate the proposed methods in section 4, which is followed by the conclusion in section 5. The appendix in section I then provides all details on how to install the implemented software, and how to use Heicub.

Note: Best view this document in electronic form. Find a digital copy under the following link https://github.com/mhubii/masters_thesis.

2 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 2.1 - 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. 2.1). 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. 2.1.1). 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. 2.1.2). Resulting positions and orientations for the center of mass and the feet will then be interpolated (sec. 2.1.3) and be passed as constraints to the inverse kinematics (sec. 2.1.4) so to transform them into joint angles that can be sent to the humanoid's motor controllers.

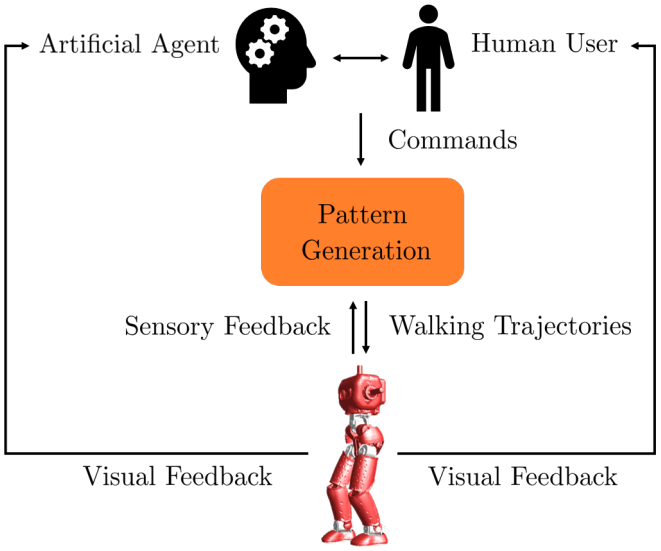


Figure 2.1: Simplified version of the proposed control loop to navigate the robot with either a human user or an artificial agent. The commands will be given in the form of linear velocities v_x, and v_y, along the x-, and the y-axis, as well as an angular velocity ω_z about the z-axis of the robot's coordinates system.

To close the control loop and to steer the robot towards desired goals, whilst avoiding obstacles, requires some high-level command that arises from visual feedback. As discussed in section 1 - Introduction, 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 nonexistence. Let alone this fact enables us to run them onboard on lightweight 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. 2.2.2), whereas the second presented method (sec. 2.2.3) explores policies and tries to find solutions on its own.

Note: Within the following chapters, there will always be made references to the actual implementation of the presented concepts, which shall enable future readers to bridge the gap between theory and application.

2.1 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 2.1 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. 2.2. The natural entry point, to this otherwise closed control loop, is given by the commands that enter the nonlinear model predictive control. Commands are passed in the form of a desired velocity $\mathbf{v}_{\text{ref}}$ 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 bases upon the zero moment point (ZMP) around which the whole control framework builds. 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. 2.2, 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 to place them 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

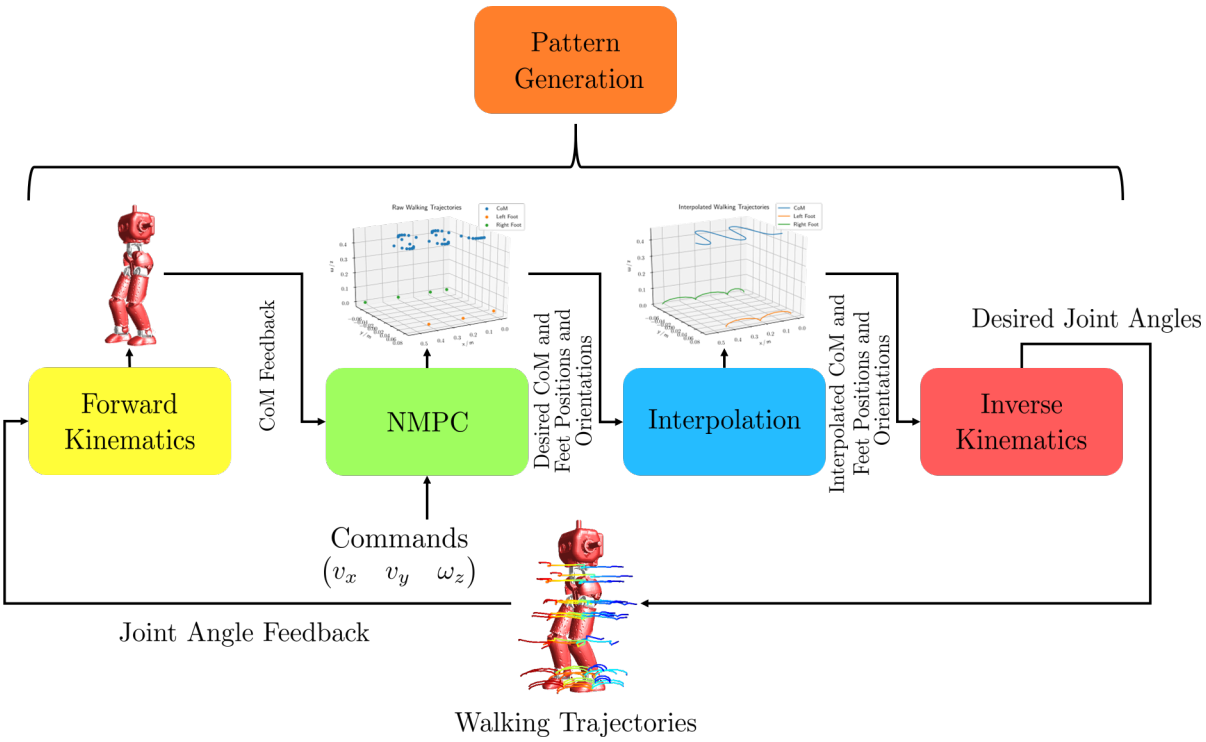


Figure 2.2: Building blocks of the pattern generation. To understand the greater picture, a connection can be drawn to fig. 2.1, where the orange box represents the one shown in this figure.

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 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. 2.2 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 effects like friction, there is a chance that the desired joint angles differ from the 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.

2.1.1 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 [10][11] and first utilized in 1984 to generate walking trajectories for the WL-10RD robot [12]. 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 [13], 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 $\boldsymbol{F}_r$ between the foot and the ground, which compensates for all external moments $\boldsymbol{M}_x$, and $\boldsymbol{M}_y$ around the x-, and y-axis at any time (fig. 2.3). The point $\boldsymbol{r}$, at which the reaction force acts, is

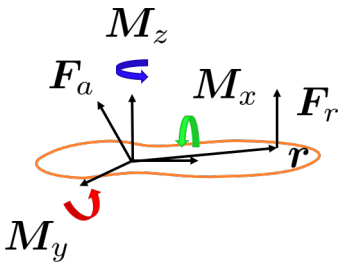


Figure 2.3: Forces acting on the sole.

physically only meaningful if it lies within the foot's support polygon. 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 [14][15]. 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 convex hull of all contact points of the feet with the ground, so the minimum 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 a rectangle well describes a foot's convex hull, we only rely on the foot width (link), and the 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

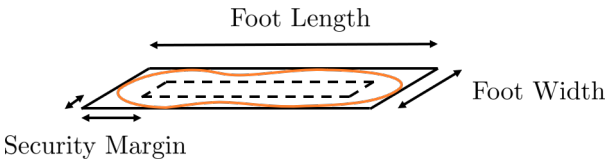


Figure 2.4: Full support polygon, and the resulting support polygon with security margin (dashed lines).

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 to 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 [16]. A rigorous derivation for the analytic relation between the center of mass and the zero moment point of a linear inverted pendulum exists in [17], but for the sake of simplicity we rather explain the physics in terms of cutting forces, for which a short introduction is in the summary of the lecture Robotics 1 (link). The system of interest is shorty

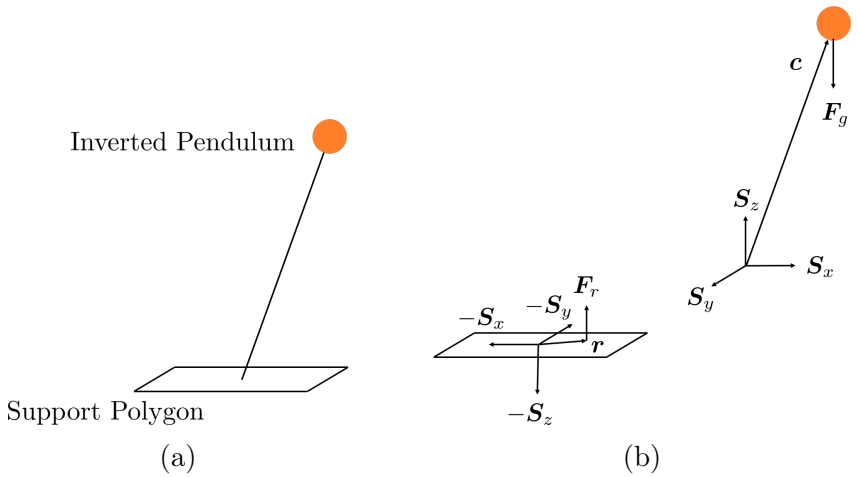


Figure 2.5: Linear inverted pendulum with a support polygon (a), and the corresponding free body diagram with cutting forces $\boldsymbol{S}_{x/y/z}$ (b).

depicted in figure 2.5. We assume the support polygon of the shown linear inverted pendulum to have zero mass. By introducing cutting forces $\boldsymbol{S}_{x/y/z}$ for each degree of freedom in which the motion of the linear inverted pendulum is restricted, we

obtain the free body diagram (fig. 2.5), for which the acting forces are

$$m\ddot{\boldsymbol{c}} = \quad \boldsymbol{S} - \boldsymbol{F}_g \tag{2.1}$$

$$\mathbf{0} = -\boldsymbol{S} + \boldsymbol{F}_r \tag{2.2}$$

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

$$\mathbf{0} = (\mathbf{0} - \boldsymbol{c}) \times \boldsymbol{S} + \boldsymbol{M} \tag{2.3}$$

$$\mathbf{0} = (\boldsymbol{r} - \mathbf{0}) \times \boldsymbol{F}_r - \boldsymbol{M} \tag{2.4}$$

where the transfer of the moment $\boldsymbol{M}$ may for example be induced by friction. If we replace $\boldsymbol{S} = \boldsymbol{F}_r$ from eq. 2.2, equations 2.3 and 2.4 yield

$$\mathbf{0} = (\boldsymbol{r} - \boldsymbol{c}) \times \boldsymbol{S} = \begin{pmatrix} (r_y - c_y)S_z - (r_z - c_z)S_y \\ -(r_x - c_x)S_z + (r_z - c_z)S_x \\ (r_x - c_x)S_y - (r_y - c_y)S_x \end{pmatrix} \tag{2.5}$$

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. 2.1 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 to rows of eq. 2.5, we find

$$r_x = c_x - c_z \frac{\ddot{c}_x}{g} \tag{2.6}$$

$$r_y = c_y - c_z \frac{\ddot{c}_y}{g} \tag{2.7}$$

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 for humanoid walking that we can solve in real-time (section 2.1.2). This simplification is, of course, only true to some extent, 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

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 [17]. Furthermore, there exist approximate approaches that involve the knowledge of all acting

external forces [18], which can, for example, be obtained from unconstrained inverse dynamics [19]. 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 [17]. If we consider the force-torque sensor to be located at position $\boldsymbol{p}_i$ (fig. 2.6),

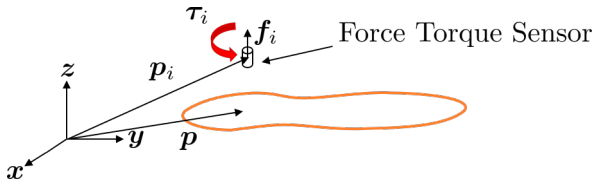


Figure 2.6: Force-torque sensors at the foot's ankle.

then we can obtain the moment about any point $\boldsymbol{p}$ according to eq. 2.8.

$$\boldsymbol{\tau}(\boldsymbol{p}) = (\boldsymbol{p}_i - \boldsymbol{p}) \times \boldsymbol{f}_i + \boldsymbol{\tau}_i \tag{2.8}$$

by definition, the moment about the zero moment point vanishes along the horizontal axes, therefore we can set $\tau_x = \tau_y = 0$ in eq. 2.8 and then solve for the position to obtain the zero moment point (eq. 2.9 and 2.10).

$$p_x = \frac{[-\tau_{i,y} - (p_{i,z} - p_z) f_{i,x} + p_{i,x} f_{i,z}]}{f_{i,z}} \tag{2.9}$$

$$p_y = \frac{[-\tau_{i,x} - (p_{i,z} - p_z) f_{i,y} + p_{i,y} f_{i,z}]}{f_{i,z}} \tag{2.10}$$

If we further choose our coordinate system to lie along the z-axis of the force-torque sensor, we can simplify equations 2.9 and 2.10 to find

$$p_x = \frac{(-\tau_{i,y} - f_{1,x} d)}{f_{1,z}} \tag{2.11}$$

$$p_y = \frac{(\tau_{i,x} - f_{1,y} d)}{f_{1,z}} \tag{2.12}$$

We can use equations 2.11 and 2.12 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 2.11 and 2.12 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 2.8 slightly. This yields

$$\boldsymbol{\tau}(\boldsymbol{p}) = \sum_{i \in \{L,R\}} (\boldsymbol{p}_i - \boldsymbol{p}) \times \boldsymbol{f}_i \tag{2.13}$$

where the individual torques are now zero and the only forces $\boldsymbol{f}_i$ that exist between the robot and the environment can be described by the z-components 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 2.13 to zero and find

$$p_x = \frac{\sum_{i\in\{L,R\}} p_{i,x} f_{i,z}}{\sum_{i\in\{L,R\}} f_{i,z}} \tag{2.14}$$

$$p_y = \frac{\sum_{i\in\{L,R\}} p_{i,y} f_{i,z}}{\sum_{i\in\{L,R\}} f_{i,z}} \tag{2.15}$$

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 2.11 and 2.12, 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

$$p_x = \frac{p_{R,x} f_{R,z} + p_{L,x} f_{L,z}}{f_{R,z} + f_{L,z}} \tag{2.16}$$

$$p_y = \frac{p_{R,y} f_{R,z} + p_{L,y} f_{L,z}}{f_{R,z} + f_{L,z}} \tag{2.17}$$

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.

2.1.2 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

$$\min_{\boldsymbol{x}} \frac{1}{2} f(\boldsymbol{x}) \tag{2.18}$$

$$\text{subject to: } \boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{0} \tag{2.19}$$

$$\boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{0}, \tag{2.20}$$

where $f : \mathbb{R}^N \to \mathbb{R}$, $\boldsymbol{h} : \mathbb{R}^N \to \mathbb{R}^M$, and $\boldsymbol{g} : \mathbb{R}^N \to \mathbb{R}^P$ [20]. 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 $\boldsymbol{x}_k$ by a quadratic subproblem, so to find a solution to this subproblem, in order to construct a better approximation $\boldsymbol{x}_{k+1}$. Now given an objective function $f(\boldsymbol{x})$ 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 [21]. That is, given an objective function $f(\boldsymbol{x}) = \boldsymbol{F}(\boldsymbol{x})^T \boldsymbol{F}(\boldsymbol{x})$, where $\boldsymbol{F} = (f_1, ..., f_l)^T$, we can apply a quasi Gauss-Newton method as follows

$$\nabla^2 f(\boldsymbol{x}) \Delta \boldsymbol{x} + \nabla f(\boldsymbol{x}) = 0, \tag{2.21}$$

where the gradient and the Hessian matrix are given as

$$\nabla f(\boldsymbol{x}) = \nabla \boldsymbol{F}(\boldsymbol{x}) \boldsymbol{F}(\boldsymbol{x}) \tag{2.22}$$

$$\nabla^2 f(\boldsymbol{x}) = \nabla \boldsymbol{F}(\boldsymbol{x}) \nabla \boldsymbol{F}(\boldsymbol{x})^T + \boldsymbol{B}(\boldsymbol{x}). \tag{2.23}$$

Therein, $\boldsymbol{B}(\boldsymbol{x}) = \sum_1^l f_i(\boldsymbol{x}) \nabla^2 f_i(\boldsymbol{x})$. If we are now sufficiently close to an optimal solution $\boldsymbol{x}^*$, such that $\boldsymbol{F}(\boldsymbol{x}^*) = (f_1(\boldsymbol{x}^*), ..., f_l(\boldsymbol{x}^*))^T = \boldsymbol{0}$, we can neglect $\boldsymbol{B}(\boldsymbol{x^*})$, which turns equation 2.21 into the previously stated Gauss-Newton minimization problem

$$\min_{\Delta \boldsymbol{x}} ||\nabla \boldsymbol{F}(\boldsymbol{x_k})^T \Delta \boldsymbol{x} + \boldsymbol{F}(\boldsymbol{x}_k)||_2^2, \tag{2.24}$$

where a new iterate is obtained by $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \Delta \boldsymbol{x}$ with an appropriate step length parameter α_k. This is because equation 2.21 defines the normal equations of equation 2.24. 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 2.6 and 2.7, 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 [16], 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 $\dddot{c}_k$ 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

$$c_{k+1} = \frac{T^3}{6}\dddot{c}_k + \frac{T^2}{2}\ddot{c}_k + T\dot{c}_k + c_k \tag{2.25}$$

$$\dot{c}_{k+1} = \frac{T^2}{2}\dddot{c}_k + T\ddot{c}_k + \dot{c}_k \tag{2.26}$$

$$\ddot{c}_{k+1} = T\dddot{c}_k + \ddot{c}_k, \tag{2.27}$$

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

$$\boldsymbol{c}_{k+1} = \boldsymbol{A}\boldsymbol{c}_k + \boldsymbol{B}\dddot{c}_k \tag{2.28}$$

$$\boldsymbol{c}_{k+1} = \begin{pmatrix} c_{k+1} \\ \dot{c}_{k+1} \\ \ddot{c}_{k+1} \end{pmatrix}, \ \boldsymbol{A} = \begin{pmatrix} 1 & T & \frac{T^2}{2} \\ 0 & 1 & T \\ 0 & 0 & 1 \end{pmatrix}, \ \boldsymbol{B} = \begin{pmatrix} \frac{T^3}{6} \\ \frac{T^2}{2} \\ T \end{pmatrix} \tag{2.29}$$

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

$$\boldsymbol{c}_{k+n} = \boldsymbol{A}^n\boldsymbol{c}_k \sum_{i=1}^{n} \boldsymbol{A}^{i-1}\boldsymbol{B}\dddot{c}_{k+n-i}, \tag{2.30}$$

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

$$\boldsymbol{A}^n = \begin{pmatrix} 1 & nT & n^2\frac{T^2}{2} \\ 0 & 1 & nT \\ 0 & 0 & 1 \end{pmatrix}, \ \boldsymbol{A}^n\boldsymbol{B} = \begin{pmatrix} (1+3n+3n^2)T^3/6 \\ (1+2n)T^2/2 \\ T \end{pmatrix}. \tag{2.31}$$

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 $\boldsymbol{c}_{k+n}$ for all $n \in [1, N]$ into one

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

$$\boldsymbol{C}_{k+1} = \begin{pmatrix} c_{k+1} \\ \vdots \\ c_{k+N} \end{pmatrix}, \ \dot{\boldsymbol{C}}_{k+1} = \begin{pmatrix} \dot{c}_{k+1} \\ \vdots \\ \dot{c}_{k+N} \end{pmatrix}, \ \ddot{\boldsymbol{C}}_{k+1} = \begin{pmatrix} \ddot{c}_{k+1} \\ \vdots \\ \ddot{c}_{k+N} \end{pmatrix}, \ \dddot{\boldsymbol{C}}_{k} = \begin{pmatrix} \dddot{c}_{k} \\ \vdots \\ \dddot{c}_{k+N-1} \end{pmatrix}, \tag{2.32}$$

we obtain (link)

$$\boldsymbol{C}_{k+1} = \boldsymbol{P}_{ps}\boldsymbol{c}_k + \boldsymbol{P}_{pu}\dddot{\boldsymbol{C}}_k \tag{2.33}$$

$$\dot{\boldsymbol{C}}_{k+1} = \boldsymbol{P}_{vs}\boldsymbol{c}_k + \boldsymbol{P}_{vu}\dddot{\boldsymbol{C}}_k \tag{2.34}$$

$$\ddot{\boldsymbol{C}}_{k+1} = \boldsymbol{P}_{as}\boldsymbol{c}_k + \boldsymbol{P}_{au}\dddot{\boldsymbol{C}}_k, \tag{2.35}$$

where the new matrices are given by (link)

$$\boldsymbol{P}_{ps} = \begin{pmatrix} 1 & T & T^2/2 \\ \vdots & & \vdots \\ 1 & nT & n^2T^2/2 \end{pmatrix}, \ \boldsymbol{P}_{pu} = \begin{pmatrix} T^3/6 & \dots & 0 \\ \vdots & \ddots & \vdots \\ (1+3n+3n^2)T^2/6 & \dots & T^3/6 \end{pmatrix} \tag{2.36}$$

$$\boldsymbol{P}_{vs} = \begin{pmatrix} 0 & 1 & T \\ \vdots & & \vdots \\ 0 & 1 & nT \end{pmatrix}, \ \boldsymbol{P}_{vu} = \begin{pmatrix} T^2/2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ (1+2n)/T^2/2 & \dots & T^2/2 \end{pmatrix} \tag{2.37}$$

$$\boldsymbol{P}_{as} = \begin{pmatrix} 0 & 0 & 1 \\ \vdots & & \vdots \\ 0 & 0 & 1 \end{pmatrix}, \ \boldsymbol{P}_{au} \begin{pmatrix} T & \dots & 0 \\ \vdots & \ddots & \vdots \\ T & \dots & T \end{pmatrix}. \tag{2.38}$$

If we now additionally consider the relation of the zero moment point and the center of mass, which we obtained earlier in equations 2.6 and 2.7, we can further relate the current center of mass state to the zero moment point on the preview horizon by (link)

$$\boldsymbol{Z}_{k+1} = \boldsymbol{C}_{k+1} - \frac{c_z}{g}\ddot{\boldsymbol{C}}_{k+1} \tag{2.39}$$

$$= \left(\boldsymbol{P}_{ps} - \frac{c_z}{g}\boldsymbol{P}_{as}\right)\boldsymbol{c}_k + \left(\boldsymbol{P}_{pu} - \frac{c_z}{g}\boldsymbol{P}_{au}\right)\dddot{\boldsymbol{C}}_k = \boldsymbol{P}_{zs}\boldsymbol{c}_k + \boldsymbol{P}_{zu}\dddot{\boldsymbol{C}}_k, \tag{2.40}$$

where the new matrices are given by (link)

$$\boldsymbol{P}_{zs} = \begin{pmatrix} 1 & T & T^2/2 - c_z/g \\ \vdots & & \vdots \\ 1 & nT & n^2T^2/2 - c_z/g \end{pmatrix} \tag{2.41}$$

$$\boldsymbol{P}_{zu} = \begin{pmatrix} T^3/6 - Tc_z/g & \dots & 0 \\ \vdots & \ddots & \vdots \\ (1+3n+3n^2)T^3/6 - Tc_z/g & \dots & T^3/6 - Tc_z/g \end{pmatrix}. \tag{2.42}$$

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 2.1.2, we put together squared L^2-norm objectives, which account for a desired reference center of mass velocity $\dot{\boldsymbol{C}}^{\text{ref}}_{k+1}$, a balanced footstep placement $\mathbf{F}_{k+1}$ close to the zero moment point $\boldsymbol{Z}_{k+1}$, and a smooth motion for which the center of mass jerk $\dddot{\boldsymbol{C}}_{k+1}$ enters as a regularization. The objective function itself is similar to the one first introduced in [22], but additionally takes the center of mass' rotation around the z-axis into account, and it can be written down as follows

$$\min_{\boldsymbol{U}_k} \frac{\alpha}{2}||\dot{\boldsymbol{C}}^{x}_{k+1} - \dot{\boldsymbol{C}}^{x,\text{ref}}_{k+1}||_2^2 + \frac{\alpha}{2}||\dot{\boldsymbol{C}}^{y}_{k+1} - \dot{\boldsymbol{C}}^{y,\text{ref}}_{k+1}||_2^2 \tag{2.43}$$

$$\frac{\alpha}{2}||\boldsymbol{E}_L\dot{\boldsymbol{F}}^{\theta,L}_{k+1} - \dot{\boldsymbol{C}}^{\theta,\text{ref}}_{k+1}||_2^2 + \frac{\alpha}{2}||\boldsymbol{E}_R\dot{\boldsymbol{F}}^{\theta,R}_{k+1} - \dot{\boldsymbol{C}}^{\theta,\text{ref}}_{k+1}||_2^2 \tag{2.44}$$

$$\frac{\beta}{2}||\boldsymbol{Z}^{x}_{k+1} - \boldsymbol{F}^{x}_{k+1}||_2^2 + \frac{\beta}{2}||\boldsymbol{Z}^{y}_{k+1} - \boldsymbol{F}^{y}_{k+1}||_2^2 \tag{2.45}$$

$$\frac{\gamma}{2}||\ddot{\boldsymbol{C}}^{x}_{k+1}||_2^2 + \frac{\gamma}{2}||\ddot{\boldsymbol{C}}^{y}_{k+1}||_2^2 \tag{2.46}$$

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 2.2. 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 footstep 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 2.40, and used in equation 2.45, we require balance on a finer timescale. We therefore project the foot placement $\tilde{\boldsymbol{F}}_k \in \mathbb{R}^{N_f \times 1}$ onto the temporal resolution of the center of mass' control variable by introducing the matrices $\boldsymbol{v}_{k+1}$, $\boldsymbol{V}_{k+1}$, and the current foot position f_k, which yields

$$\boldsymbol{F}_{k+1} = \boldsymbol{v}_{k+1}f_k + \boldsymbol{V}_{k+1}\tilde{\boldsymbol{F}}_k, \tag{2.47}$$

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

$$\boldsymbol{v}_{k+1} = \begin{pmatrix} 1 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \boldsymbol{V}_{k+1} = \begin{pmatrix} 0 & 0 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \vdots & \vdots & \vdots \\ 0 & 0 & 1 \end{pmatrix} \tag{2.48}$$

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

$$\left(\begin{array}{c|cc} \boldsymbol{v}_{k+1} & \boldsymbol{V}_{k+1} \end{array} \right) = \left(\begin{array}{c|cc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{array} \right) \rightarrow \left(\begin{array}{c|cc} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{array} \right) \rightarrow \left(\begin{array}{c|cc} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right) \tag{2.49}$$

Now to ensure the rotation of the center of mass, we introduce equation 2.44 to the objective function. The matrices $\boldsymbol{E}_{L/R}$ 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 $\boldsymbol{E}_{L/R}$ take the form (link)

$$\boldsymbol{E}_L = \begin{pmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & 0 & & & & & \\ & & & & \ddots & & & & \\ & & & & & 0 & & & \\ & & & & & & 1 & & \\ & & & & & & & \ddots & \\ & & & & & & & & 1 \end{pmatrix}, \quad \boldsymbol{E}_R = \mathbf{1} - \boldsymbol{E}_L \tag{2.50}$$

The diagonal entries therein just take the same form as a single column of $\boldsymbol{V}_{k+1}$, all other entries are zero. If we now take equations 2.43-2.46, and replace $\dot{\boldsymbol{C}}_{k+1}^{x/y}$, as well as $\dot{\boldsymbol{F}}_{k+1}^{\theta,L/R}$, by equation 2.34, and insert equation 2.40 into the zero moment point on the preview horizon $\boldsymbol{Z}_{k+1}$, we obtain following relation (link)

$$\min_{\boldsymbol{U}_k} \frac{1}{2}\boldsymbol{U}_{\boldsymbol{k}}^T \boldsymbol{Q}_k \boldsymbol{U}_k + \boldsymbol{p}_k^T \boldsymbol{U}_k \tag{2.51}$$

$$\boldsymbol{U}_k = \begin{pmatrix} \boldsymbol{U}_k^{xy} & \boldsymbol{U}_k^{\theta} \end{pmatrix}^T \tag{2.52}$$

$$\boldsymbol{U}_k^{xy} = \begin{pmatrix} \dddot{\boldsymbol{C}}_k^x & \tilde{\boldsymbol{F}}_k^x & \dddot{\boldsymbol{C}}_k^y & \tilde{\boldsymbol{F}}_k^y \end{pmatrix}^T \tag{2.53}$$

$$\boldsymbol{U}_k^{\theta} = \begin{pmatrix} \dddot{\boldsymbol{F}}_k^{\theta,L} & \dddot{\boldsymbol{F}}_k^{\theta,R} \end{pmatrix}^T \tag{2.54}$$

$$\boldsymbol{Q}_k = \begin{pmatrix} \boldsymbol{Q}_k^x & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Q}_k^y & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{Q}_k^L & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{Q}_k^R \end{pmatrix} \tag{2.55}$$

$$\boldsymbol{Q}_k^x = \boldsymbol{Q}_k^y = \begin{pmatrix} \alpha \boldsymbol{P}_{vu}^T \boldsymbol{P}_{vu} + \beta \boldsymbol{P}_{zu}^T \boldsymbol{P}_{zu} + \gamma \boldsymbol{1} & -\beta \boldsymbol{P}_{zu}^T \boldsymbol{V}_{k+1} \\ -\beta \boldsymbol{V}_{k+1}^T \boldsymbol{P}_{zu} & \beta \boldsymbol{V}_{k+1}^T \boldsymbol{V}_{k+1} \end{pmatrix} \tag{2.56}$$

$$\boldsymbol{Q}_k^{L/R} = \begin{pmatrix} \alpha \boldsymbol{P}_{vu}^T \boldsymbol{E}_{L/R}^T \boldsymbol{E}_{L/R} \boldsymbol{P}_{vu} \end{pmatrix} \tag{2.57}$$

$$\boldsymbol{p}_k = \begin{pmatrix} \boldsymbol{p}_k^x \\ \boldsymbol{p}_k^y \\ \boldsymbol{p}_k^L \\ \boldsymbol{p}_k^R \end{pmatrix} \tag{2.58}$$

$$\boldsymbol{p}_k^{x/y} = \begin{pmatrix} \alpha \boldsymbol{P}_{vu}^T (\boldsymbol{P}_{vs} \boldsymbol{c}_k^{x/y} - \dot{\boldsymbol{C}}_{k+1}^{x/y,\mathrm{ref}}) + \beta \boldsymbol{P}_{zu}^T (\boldsymbol{P}_{zs} \boldsymbol{c}_k^{x/y} - \boldsymbol{v}_{k+1} f_k^{x/y}) \\ -\beta \boldsymbol{V}_{k+1}^T (\boldsymbol{P}_{zs} \boldsymbol{c}_k^{x/y} - \boldsymbol{v}_{k+1} f_k^{x/y}) \end{pmatrix} \tag{2.59}$$

$$\boldsymbol{p}_k^{L/R} = \begin{pmatrix} \alpha \boldsymbol{P}_{vu}^T \boldsymbol{E}_{L/R}^T (\boldsymbol{E}_{L/R} \boldsymbol{P}_{vs} \boldsymbol{f}_k^{q,L/R} - \dot{\boldsymbol{C}}_{k+1}^{L/R,\mathrm{ref}}) \end{pmatrix} \tag{2.60}$$

where we evaluated all squared L^2-norms via $||\boldsymbol{a} - \boldsymbol{b}||_2^2 = (\boldsymbol{a} - \boldsymbol{b})^T (\boldsymbol{a} - \boldsymbol{b})$, and ordered the terms correspondingly. All terms that do not depend on the control variable $\boldsymbol{U}_k$ got discarded. Equation 2.51 now represents the canonical formulation of the minimization problem we were looking for in equation 2.18. 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 2.7), we set up a system of linear equations, of which each describes a line that connects the polygon's edges $\boldsymbol{p}_i$ (link). A point $\boldsymbol{x}$ lies beneath

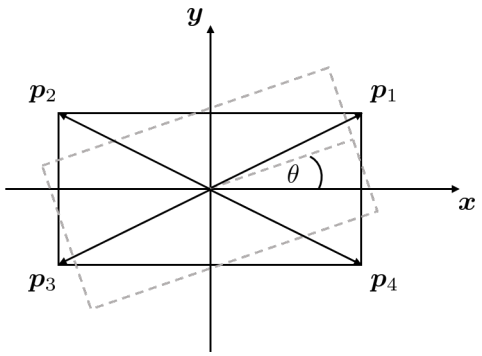


Figure 2.7: The foot's support polygon, which is described by the position vectors $\boldsymbol{p}_i$.

the line that connects the edges if

$$\boldsymbol{A}^{L/R}\boldsymbol{x} \leq \boldsymbol{B}^{L/R}, \tag{2.61}$$

where the linear equations are defined by $\boldsymbol{A}^{L/R} = \begin{pmatrix}\boldsymbol{A}^{L/R,x} & \boldsymbol{A}^{L/R,y}\end{pmatrix} \in \mathbb{R}^{N_{\text{edges}}\times 2}$, and $\boldsymbol{B}^{L/R} \in \mathbb{R}^{N_{\text{edges}}\times 1}$

$$\boldsymbol{A}^{L/R,x}[i] = p_i^{L/R,y} - p_{i+1}^{L/R,y} \tag{2.62}$$

$$\boldsymbol{A}^{L/R,y}[i] = p_{i+1}^{L/R,x} - p_i^{L/R,x} \tag{2.63}$$

$$\boldsymbol{B}^{L/R}[i] = (p_i^{L/R,y} - p_{i+1}^{L/R,y})p_{i+1}^{L/R,x} + (p_{i+1}^{L/R,x} - p_i^{L/R,x})p_{i+1}^{L/R,y} \tag{2.64}$$

Since we require the zero moment point to lie inside the support polygon, $\boldsymbol{x}$ in equation 2.61 is replaced by $\boldsymbol{R}_z(f_k^\theta)(\boldsymbol{z}_k - \boldsymbol{f}_k)$, with $\boldsymbol{z}_k = \begin{pmatrix}z_k^x & z_k^y\end{pmatrix}^T$, and $\boldsymbol{f}_k = \begin{pmatrix}f_k^x & f_k^y\end{pmatrix}^T$. This expression describes the zero moment point with respect to the foot frame, where $\boldsymbol{R}_z(f_k^\theta) = \begin{pmatrix}\cos f_k^\theta & \sin f_k^\theta \\ -\sin f_k^\theta & \cos f_k^\theta\end{pmatrix}$ is an inverse rotation around the z-axis that adds non-linearities to the constraints. We can now extent the formalism to the whole preview horizon by utilizing equations 2.40 and 2.47. This leads to (link)

$$\boldsymbol{D}_{k+1}(\boldsymbol{F}_{k+1}^\theta)\begin{pmatrix}\boldsymbol{Z}_{k+1}^x - \boldsymbol{F}_{k+1}^x \\ \boldsymbol{Z}_{k+1}^y - \boldsymbol{F}_{k+1}^y\end{pmatrix} \leq \boldsymbol{B}_{k+1} \tag{2.65}$$

$$\boldsymbol{D}_{k+1}(\boldsymbol{F}_{k+1}^\theta)\begin{pmatrix}\boldsymbol{P}_{zs}\boldsymbol{c}_k^x + \boldsymbol{P}_{zu}\dddot{\boldsymbol{C}}_k^x - \boldsymbol{v}_{k+1}f_k^x - \boldsymbol{V}_{k+1}\tilde{\boldsymbol{F}}_{k+1}^x \\ \boldsymbol{P}_{zs}\boldsymbol{c}_k^y + \boldsymbol{P}_{zu}\dddot{\boldsymbol{C}}_k^y - \boldsymbol{v}_{k+1}f_k^y - \boldsymbol{V}_{k+1}\tilde{\boldsymbol{F}}_{k+1}^y\end{pmatrix} \leq \boldsymbol{B}_{k+1} \tag{2.66}$$

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

$$\boldsymbol{D}_{k+1} = \begin{pmatrix} \boldsymbol{A}^{L/R,x} \boldsymbol{R}_z(f^{\theta}_{k+1}) & & \boldsymbol{0} & \boldsymbol{A}^{L/R,y} \boldsymbol{R}_z(f^{\theta}_{k+1}) & & \boldsymbol{0} \\ & \ddots & & & \ddots & \\ \boldsymbol{0} & & \boldsymbol{A}^{L/R,x} \boldsymbol{R}_z(f^{\theta}_{k+N}) & \boldsymbol{0} & & \boldsymbol{A}^{L/R,y} \boldsymbol{R}_z(f^{\theta}_{k+N}) \end{pmatrix}, \tag{2.67}$$

and $\boldsymbol{B}_{k+1} = \begin{pmatrix} \boldsymbol{B}^{L/R} & \dots & \boldsymbol{B}^{L/R} \end{pmatrix}^T$. 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 2.66 can now be expressed in terms of the free variables $\boldsymbol{U}_k$ by

$$\boldsymbol{D}_{k+1} \begin{pmatrix} \boldsymbol{P}_{zu} & -\boldsymbol{V}_{k+1} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{P}_{zu} & -\boldsymbol{V}_{k+1} \end{pmatrix} \boldsymbol{U}^{xy}_{k} \leq \boldsymbol{B}_{k+1} + \boldsymbol{D}_{k+1} \begin{pmatrix} -\boldsymbol{P}_{zs} \boldsymbol{c}^{x}_{k} + \boldsymbol{v}_{k+1} f^{x}_{k} \\ -\boldsymbol{P}_{zs} \boldsymbol{c}^{y}_{k} + \boldsymbol{v}_{k+1} f^{y}_{k} \end{pmatrix} \tag{2.68}$$

$$\boldsymbol{A}_{\text{zmp},k}(\boldsymbol{U}^{\theta}_{k}) \boldsymbol{U}^{xy}_{k} \leq \overline{\boldsymbol{U}_{\text{zmp},k}}, \tag{2.69}$$

where $\overline{\boldsymbol{U}_{\text{zmp},k}}$ 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, which can again be defined as a set of linear inequalities, with the position vectors $\boldsymbol{p}_i$ that are shown in figure 2.8. We can therefore just reuse the concept of equation 2.61, and replace $\boldsymbol{x}$ therein by

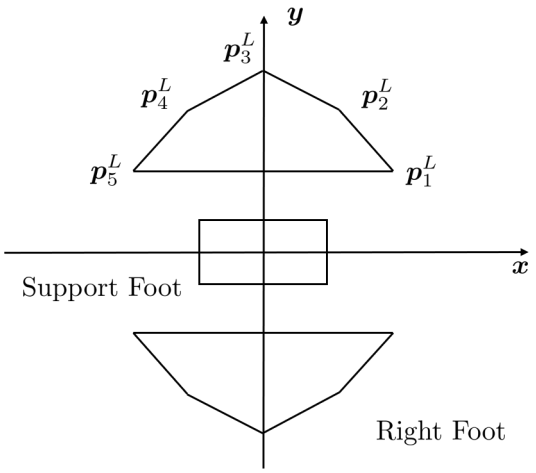


Figure 2.8: The foot's feasibility polygon, which is described by the position vectors $\boldsymbol{p}_i$. The center is always defined by the current support foot's position, within the picture the support foot can therefore be the left as well as the right foot.

the difference of the next foot position $\tilde{\boldsymbol{F}}_k \in \mathcal{R}^{N_f \times 1}$, and the previous one $\tilde{\boldsymbol{F}}_{k-1} =$

$\boldsymbol{S}_0\tilde{\boldsymbol{F}}_k + \boldsymbol{S}_1 f_k$, where $\boldsymbol{S}_0 = \begin{pmatrix} 0 & 0 \\ \mathbf{I}_{N_f-1} & 0 \end{pmatrix} \in \mathcal{R}^{N_f \times N_f}$ simply shifts $\tilde{\boldsymbol{F}}_k$ down by one row, and $\boldsymbol{S}_1 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in \mathcal{R}^{N_f \times 1}$. Altogether we have (link)

$$\begin{pmatrix} \boldsymbol{A}^x_{k+1} & \boldsymbol{A}^y_{k+1} \end{pmatrix} \begin{pmatrix} \tilde{\boldsymbol{F}}^x_k - \boldsymbol{S}_0\tilde{\boldsymbol{F}}^x_k - \boldsymbol{S}_1 f^x_k \\ \tilde{\boldsymbol{F}}^y_k - \boldsymbol{S}_0\tilde{\boldsymbol{F}}^y_k - \boldsymbol{S}_1 f^y_k \end{pmatrix} \leq \boldsymbol{B}_{k+1} \tag{2.70}$$

$$\begin{pmatrix} \mathbf{0} & \boldsymbol{A}^x_{k+1}(\mathbf{I}_{N_f} - \boldsymbol{S}_0) & \mathbf{0} & \boldsymbol{A}^y_{k+1}(\mathbf{I}_{N_f} - \boldsymbol{S}_0) \end{pmatrix} \boldsymbol{U}^{xy}_k \leq \boldsymbol{B}_{k+1} + \begin{pmatrix} \boldsymbol{S}_1 f^x_k \\ \boldsymbol{S}_1 f^x_k \end{pmatrix} \tag{2.71}$$

$$\boldsymbol{A}_{\text{foot},k}(\boldsymbol{U}^\theta_k)\boldsymbol{U}^{xy}_k \leq \overline{\boldsymbol{U}_{\text{foot},k}}, \tag{2.72}$$

where $\boldsymbol{A}^{x/y}_{k+1} = \begin{pmatrix} \boldsymbol{A}^{L/R,x/y}\boldsymbol{R}_z(f^\theta_k) & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \boldsymbol{A}^{R/L,x/y}\boldsymbol{R}_z(f^\theta_{k+N_f N}) \end{pmatrix} \in \mathbb{R}^{N_{\text{edges}} N \times N_f}$ has alternating linear inequalities $\boldsymbol{A}^{L/R,x/y} \rightarrow \boldsymbol{A}^{R/L,x/y}$ that correspond on the support foot, and so does $\boldsymbol{B_{k+1}} \in \mathbb{R}^{N_{\text{edges}} N \times 1}$.

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 [23]. The constraints can be formulated as follows (link)

$$-\boldsymbol{\theta}_{\max} \leq \boldsymbol{F}^{L,\theta}_{k+1} - \boldsymbol{F}^{R,\theta}_{k+1} \leq \boldsymbol{\theta}_{\max} \tag{2.73}$$

$$-\dot{\boldsymbol{\theta}}_{\max} \leq \dot{\boldsymbol{F}}^{L,\theta}_{k+1} - \dot{\boldsymbol{F}}^{R,\theta}_{k+1} \leq \dot{\boldsymbol{\theta}}_{\max}, \tag{2.74}$$

which can be expressed in terms of the free variables with equations 2.33 and 2.34 to find

$$-\boldsymbol{\theta}_{\max} - \boldsymbol{P}_{ps}(f^L_k - f^R_k) \leq \begin{pmatrix} \boldsymbol{P}_{pu} & -\boldsymbol{P}_{pu} \end{pmatrix} \boldsymbol{U}^\theta_k \leq \boldsymbol{\theta}_{\max} - \boldsymbol{P}_{ps}(f^L_k - f^R_k) \tag{2.75}$$

$$\underline{\boldsymbol{U}_{\text{ori},k}} \leq \boldsymbol{A}_{\text{ori},k}\boldsymbol{U}^\theta_k \leq \overline{\boldsymbol{U}_{\text{ori},k}} \tag{2.76}$$

$$-\dot{\boldsymbol{\theta}}_{\max} - \boldsymbol{P}_{vs}(f^L_k - f^R_k) \leq \begin{pmatrix} \boldsymbol{P}_{vu} & -\boldsymbol{P}_{vu} \end{pmatrix} \boldsymbol{U}^\theta_k \leq \dot{\boldsymbol{\theta}}_{\max} - \boldsymbol{P}_{vs}(f^L_k - f^R_k) \tag{2.77}$$

$$\underline{\boldsymbol{U}_{\text{dori},k}} \leq \boldsymbol{A}_{\text{dori},k}\boldsymbol{U}^\theta_k \leq \overline{\boldsymbol{U}_{\text{dori},k}}. \tag{2.78}$$

Obstacle Constraints In contrast to similar methods like [24], we also include the avoidance of convex obstacles by requesting that the feet's positions must lie outside of circles $C = \{(p^x, p^y) \in \mathbb{R}^2 \,|\, (p^x - x_0)^2 + (p^y - y_0)^2 \leq R^2\}$, 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)

$$(f_k^x - x_0)^2 + (f_k^y - y_0)^2 \geq (R+m)^2 \tag{2.79}$$

$$\boldsymbol{U}_k^{xy,T} \begin{pmatrix} \mathbf{0} & \mathbf{I}_{N_f} & \mathbf{0} & \mathbf{I}_{N_f} \end{pmatrix} \boldsymbol{U}_k^{xy} - \tag{2.80}$$

$$\begin{pmatrix} \mathbf{0} & \begin{pmatrix} 2x_0 & \dots & 2x_0 \end{pmatrix} & \mathbf{0} & \begin{pmatrix} 2y_0 & \dots & 2y_0 \end{pmatrix} \end{pmatrix} \boldsymbol{U}_k^{xy} \geq \begin{pmatrix} (R+m)^2 - x_0^2 - y_0^2 \\ \vdots \\ (R+m)^2 - x_0^2 - y_0^2 \end{pmatrix} \tag{2.81}$$

$$\boldsymbol{U}_k^{xy,T} \boldsymbol{H}_{\text{obs}} \boldsymbol{U}_k^{xy} + \boldsymbol{A}_{\text{obs}} \boldsymbol{U}_k^{xy} \geq \underline{\boldsymbol{U}_{\text{obs}}}, \tag{2.82}$$

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 must 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 2.24, we can now linearize the quadratic problem of equation 2.51. This leads to (link)

$$\min_{\Delta \boldsymbol{U}_k} \frac{1}{2} \Delta \boldsymbol{U}_k^T \boldsymbol{Q}_k \Delta \boldsymbol{U}_k + \tilde{\boldsymbol{p}}_k^T \Delta \boldsymbol{U}_k \tag{2.83}$$

$$\text{subject to: } \underline{\tilde{\boldsymbol{U}}_k} \leq \tilde{\boldsymbol{A}}_k \Delta \boldsymbol{U}_k \leq \overline{\tilde{\boldsymbol{U}}_k}, \tag{2.84}$$

where

$$\tilde{\boldsymbol{p}}_k = \begin{pmatrix} \frac{1}{2}\boldsymbol{U}_{k-1}^{xy,T} \begin{pmatrix} \boldsymbol{Q}_k^x & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Q}_k^x \end{pmatrix} + \begin{pmatrix} \boldsymbol{p}_k^x \\ \boldsymbol{p}_k^y \end{pmatrix} \\ \frac{1}{2}\boldsymbol{U}_{k-1}^{\theta,T} \begin{pmatrix} \boldsymbol{Q}_k^L & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Q}_k^R \end{pmatrix} + \begin{pmatrix} \boldsymbol{p}_k^L \\ \boldsymbol{p}_k^R \end{pmatrix} \end{pmatrix} \tag{2.85}$$

$$\tilde{\boldsymbol{A}}_k = \begin{pmatrix} \boldsymbol{A}_{\text{zmp},k}(\boldsymbol{U}_{k-1}^{\theta}) & \nabla_{\boldsymbol{U}^\theta}^T \boldsymbol{A}_{\text{zmp},k}|_{\boldsymbol{U}_{k-1}^{\theta}} \boldsymbol{U}_{k-1}^{xy} \\ \boldsymbol{A}_{\text{foot},k}(\boldsymbol{U}_{k-1}^{\theta}) & \nabla_{\boldsymbol{U}^\theta}^T \boldsymbol{A}_{\text{foot},k}|_{\boldsymbol{U}_{k-1}^{\theta}} \boldsymbol{U}_{k-1}^{xy} \\ \boldsymbol{H}_{\text{obs}} \boldsymbol{U}_{k-1}^{xy} + \boldsymbol{A}_{\text{obs},k} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{A}_{\text{ori},k} \\ \boldsymbol{0} & \boldsymbol{A}_{\text{dori},k} \end{pmatrix} \tag{2.86}$$

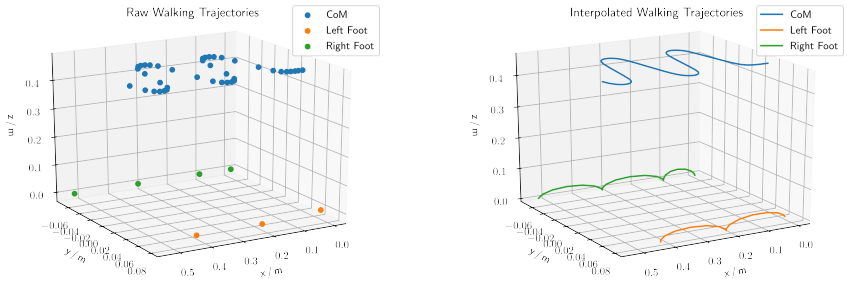
$$\underline{\tilde{\boldsymbol{U}}_k} = \begin{pmatrix} -\infty \\ -\infty \\ \underline{\boldsymbol{U}_{\text{obs}}} \\ \underline{\boldsymbol{U}_{\text{ori}}} \\ \underline{\boldsymbol{U}_{\text{dori}}} \end{pmatrix} - \boldsymbol{h}_{k-1},\ \overline{\tilde{\boldsymbol{U}}_k} = \begin{pmatrix} \overline{\boldsymbol{U}_{\text{zmp},k}} \\ \overline{\boldsymbol{U}_{\text{foot},k}} \\ \infty \\ \overline{\boldsymbol{U}_{\text{ori},k}} \\ \overline{\boldsymbol{U}_{\text{dori},k}} \end{pmatrix} - \boldsymbol{h}_{k-1} \tag{2.87}$$

$$\boldsymbol{h}_{k-1} = \begin{pmatrix} \boldsymbol{A}_{\text{zmp},k}(\boldsymbol{U}_{k-1}^{\theta}) \boldsymbol{U}_{k-1}^{xy} \\ \boldsymbol{A}_{\text{foot},k}(\boldsymbol{U}_{k-1}^{\theta}) \boldsymbol{U}_{k-1}^{xy} \\ \boldsymbol{U}_{k-1}^{xy,T} \boldsymbol{H}_{\text{obs}} \boldsymbol{U}_{k-1}^{xy} + \boldsymbol{A}_{\text{obs}} \boldsymbol{U}_{k-1}^{xy} \\ \boldsymbol{A}_{\text{ori},k} \boldsymbol{U}_{k-1}^{\theta} \\ \boldsymbol{A}_{\text{dori},k} \boldsymbol{U}_{k-1}^{\theta} \end{pmatrix}. \tag{2.88}$$

It follows that we only need to compute the gradient of $\boldsymbol{A}_{\text{zmp},k}$, and $\boldsymbol{A}_{\text{foot},k}$, with respect to the free variable $\boldsymbol{U}^\theta$, which can analytically be done by deriving the rotation matrices' gradient in equations 2.67, and 2.72. The solution to this optimal control problem yields the update $\Delta\boldsymbol{U}_k$ to the current iterate $\boldsymbol{U}_k = \boldsymbol{U}_{k-1} + \Delta\boldsymbol{U}_k$ (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.

2.1.3 Interpolating Trajectories

As already shortly depicted in figure 2.2, we need to interpolate the trajectories that we obtain from the nonlinear model predictive control. This especially holds 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 for a robot that must 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 2.9 and will further be explained in the following.



(a) (b)

Figure 2.9: 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 2.89 - 2.91).

$$p(t) = \sum_{i=0}^{N} a_i t^i \tag{2.89}$$

$$\dot{p}(t) = \sum_{i=1}^{N} i a_i t^{(i-1)} \tag{2.90}$$

$$\ddot{p}(t) = \sum_{i=2}^{N} i(i-1) a_i t^{(i-2)} \tag{2.91}$$

The coefficients a_i of the polynomials can be chosen such that certain boundary conditions $\boldsymbol{b}$ are satisfied. For the lift-off and the drop-down of the robot's feet, these boundary conditions must satisfy a zero initial velocity $\dot{z}_{\text{init}}$ and a zero end velocity $\dot{z}_{\text{end}}$, 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 2.89 and 2.90,

respectively.

$$z(t=0) = z_{\text{init}} = 0 \tag{2.92}$$
$$\dot{z}(t=0) = \dot{z}_{\text{init}} = 0 \tag{2.93}$$
$$z(t=\frac{T}{2}) = z_{T/2} \tag{2.94}$$
$$z(t=T) = z_{\text{end}} = 0 \tag{2.95}$$
$$\dot{z}(t=T) = \dot{z}_{\text{end}} = 0 \tag{2.96}$$

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 2.92 - 2.96 as follows

$$\boldsymbol{M}_z \boldsymbol{a}_z = \boldsymbol{b}_z \tag{2.97}$$
$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & \left(\frac{T}{2}\right) & \left(\frac{T}{2}\right)^2 & \left(\frac{T}{2}\right)^3 & \left(\frac{T}{2}\right)^4 \\ 1 & T & T^2 & T^3 & T^4 \\ 0 & 1 & 2T & 3T^2 & 4T^3 \end{pmatrix} \begin{pmatrix} a_{z,0} \\ a_{z,1} \\ a_{z,2} \\ a_{z,3} \\ a_{z,4} \end{pmatrix} = \begin{pmatrix} z_{\text{init}} \\ \dot{z}_{\text{init}} \\ z_{T/2} \\ z_{\text{end}} \\ \dot{z}_{\text{end}} \end{pmatrix}. \tag{2.98}$$

Inversion then yields

$$\boldsymbol{a}_z = \boldsymbol{M}_z^{-1} \boldsymbol{b}_z \tag{2.99}$$
$$\begin{pmatrix} a_{z,0} \\ a_{z,1} \\ a_{z,2} \\ a_{z,3} \\ a_{z,4} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ -\frac{11}{T^2} & -\frac{4}{T} & \frac{16}{T^2} & -\frac{5}{T^2} & \frac{1}{T} \\ \frac{18}{T^3} & \frac{5}{T^2} & -\frac{32}{T^3} & \frac{14}{T^3} & -\frac{3}{T^2} \\ -\frac{8}{T^4} & -\frac{2}{T^3} & \frac{16}{T^4} & -\frac{8}{T^4} & \frac{2}{T^3} \end{pmatrix} \begin{pmatrix} z_{\text{init}} \\ \dot{z}_{\text{init}} \\ z_{T/2} \\ z_{\text{end}} \\ \dot{z}_{\text{end}} \end{pmatrix}. \tag{2.100}$$

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

$$x(t=0) = x_{\text{init}} \tag{2.101}$$
$$\dot{x}(t=0) = \dot{x}_{\text{init}} \tag{2.102}$$
$$\ddot{x}(t=0) = \ddot{x}_{\text{init}} \tag{2.103}$$
$$x(t=T) = x_{\text{end}} \tag{2.104}$$
$$\dot{x}(t=T) = \dot{x}_{\text{end}} \tag{2.105}$$
$$\ddot{x}(t=T) = \ddot{x}_{\text{end}}. \tag{2.106}$$

Again, we can rewrite equations 2.101 - 2.106 in matrix formulation

$$\boldsymbol{M}_x \boldsymbol{a}_x = \boldsymbol{b}_x \tag{2.107}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 1 & T & T^2 & T^3 & T^4 & T^5 \\ 0 & 1 & 2T & 3T^2 & 4T^3 & 5T^4 \\ 0 & 0 & 2 & 6T & 12T^2 & 20T^3 \end{pmatrix} \begin{pmatrix} a_{x,0} \\ a_{x,1} \\ a_{x,2} \\ a_{x,3} \\ a_{x,4} \\ a_{x,5} \end{pmatrix} = \begin{pmatrix} x_{\text{init}} \\ \dot{x}_{\text{init}} \\ \ddot{x}_{\text{init}} \\ x_{\text{end}} \\ \dot{x}_{\text{end}} \\ \ddot{x}_{\text{end}} \end{pmatrix}, \tag{2.108}$$

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

$$\boldsymbol{a}_x = \boldsymbol{M}_x^{-1} \boldsymbol{b}_x \tag{2.109}$$

$$\begin{pmatrix} a_{x,0} \\ a_{x,1} \\ a_{x,2} \\ a_{x,3} \\ a_{x,4} \\ a_{x,5} \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ -\frac{20}{T^3} & -\frac{12}{T^2} & -\frac{3}{T} & \frac{20}{T^3} & -\frac{8}{T^2} & \frac{1}{T} \\ \frac{30}{T^4} & \frac{16}{T^3} & \frac{3}{T^2} & -\frac{30}{T^4} & \frac{14}{T^3} & -\frac{2}{T^2} \\ -\frac{12}{T^5} & -\frac{6}{T^4} & -\frac{1}{T^3} & \frac{12}{T^5} & -\frac{6}{T^4} & \frac{1}{T^3} \end{pmatrix} \begin{pmatrix} x_{\text{init}} \\ \dot{x}_{\text{init}} \\ \ddot{x}_{\text{init}} \\ x_{\text{end}} \\ \dot{x}_{\text{end}} \\ \ddot{x}_{\text{end}} \end{pmatrix}. \tag{2.110}$$

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 2.29 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.

2.1.4 Kinematics

As already shortly depicted in figure 2.2, 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 $\boldsymbol{q}$ into Cartesian coordinates $\boldsymbol{x}$ such that $\mathbf{FK}(\boldsymbol{q}) = \boldsymbol{x}$. 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 $\boldsymbol{t}$ and rotations $\boldsymbol{R}$

$$\boldsymbol{x}_0 = \prod_{i=N}^{0} \boldsymbol{H}_i^{i-1}(q)\boldsymbol{x}_\mathrm{N}, \tag{2.111}$$

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

Inverse Kinematics

In inverse kinematics, we aim at finding joint angles $\boldsymbol{q}$ 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 $S(\boldsymbol{q}, \Delta\boldsymbol{q})$ between desired positions and orientations a_i and the linearization of the forward kinematics around the robot's current pose $\boldsymbol{q}$ (equation 2.112), to find an incremental update $\Delta\boldsymbol{q}$.

$$S(\boldsymbol{q}, \Delta\boldsymbol{q}) = \sum_{i=0}^{m} \left[a_i - \mathbf{FK}_i(\boldsymbol{q}) - \frac{\partial \mathbf{FK}_i(\boldsymbol{q})}{\partial \boldsymbol{q}} \Delta\boldsymbol{q} \right]^2 \tag{2.112}$$

A damped version of this minimization yields the Levenberg-Marquardt algorithm [25][26], with which one can iteratively update the pose $\boldsymbol{q}$ by $\Delta\boldsymbol{q}$, so to satisfy the posed requirements (equation 2.113).

$$\Delta\boldsymbol{q} = (\boldsymbol{J}^T\boldsymbol{J} + \lambda\boldsymbol{I})^{-1}\boldsymbol{J}^T(\boldsymbol{y} - \mathbf{FK}(\boldsymbol{q})) \tag{2.113}$$

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

2.2 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 2.2.1, then cover two possible methods for training a

neural network, one of which is supervised 2.2.2, whereas the second method bases on reinforcement learning 2.2.3, and finally explain image processing techniques in section 2.2.4, 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.

2.2.1 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 2.10). Although

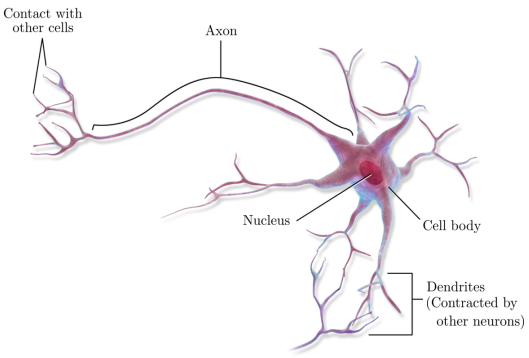


Figure 2.10: Biological neuron, which connects its cell body to dendrites of surrounding neurons via an axon. [27]

neural networks have gained early attention in research, they have only recently become powerful for their implementation on graphic processing units (GPUs) [28]. This followed from their mathematical description that is linear and very well parallelizable. Running neural nets on GPUs, on the other hand, consumes much energy, which stays in contrast to biological neurons, which communicate by brief energy-efficient spikes [29]. Moreover, while there are around 100 billion neurons in the human brain, huge neural networks currently have around a factor of 10000 less [30]. Not only is the number of neurons in a neural network 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 [31]. 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 [32] or dropout [33] layers, which reduce the number of neurons within a neural network. 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 [34] lay the foundation for neural networks, and it got extended soon to the multi-layer perceptron model [35], 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 2.11, where each orange circle represents a neuron that connects to its surroundings via simple weights w_{ij}, as neurons inside the brain are by synapses. Mathematically speaking, the feed forward process

Figure 2.11: 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.

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

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \tag{2.114}$$

Therein, the bias $\boldsymbol{b}$ is as a shift of isolines that are introduced by hyperplanes. These hyperplanes are learned and expressed by the layer's weights w_{ij} (figure 2.12). The output $\boldsymbol{y}$, 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, some activation functions have shown to be of particular use, such as the rectifying linear unit, or the hyperbolic tangent.

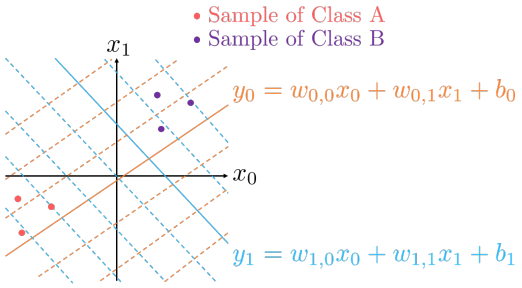


Figure 2.12: Simple interpretation of a fully connected neural network with one layer that takes $\boldsymbol{x} = (x_0\ x_1)^T$ as input. The dotted lines are isolines to the hyperplane, which showcase the effect of the bias $\boldsymbol{b}$.

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 [36], and soon after termed convolutional neural network for its mathematical properties, in that it equals convolutions. Figure 2.13 shows how an input $\boldsymbol{x}$ is fed forward through the network architecture across two layers. The operation is again

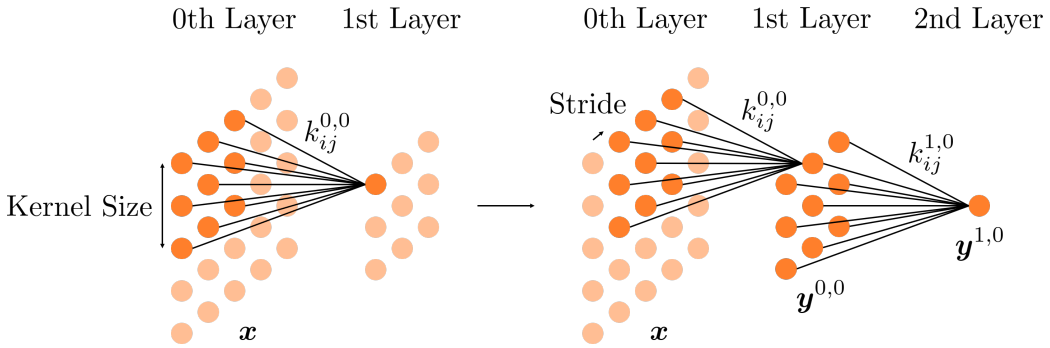


Figure 2.13: Convolutional neural network with a total of three layers, of which one is the input layer. For visualization, the kernel size is set to be three, and the stride is set to be one.

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 $\boldsymbol{k}^{l,n}$ of 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

$$\boldsymbol{y}^{0,0} = f(\boldsymbol{x} * \boldsymbol{k}^{0,0}) \tag{2.115}$$

$$\boldsymbol{y}^{1,0} = f(\boldsymbol{y}^{0,0} * \boldsymbol{k}^{1,0}) \tag{2.116}$$

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 2.13, where each neuron within the first layer only sees a kernel size sized snipped of the original input $\boldsymbol{x}$, 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 [37]. Figure 2.14 shows transposed convolutions for a classification network that was trained on ImageNet [38]. 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

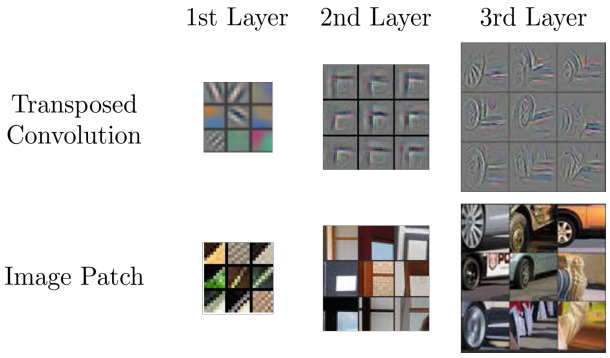


Figure 2.14: Analysis of neurons with the highest activation, corresponding to a subset of ImageNet. For the first layer, the kernel itself is shown, and image patches at the kernel's scale. For the second and the third layer, a single neuron with the highest activation is upscaled by transposed convolutions to the first layer's feature map. Images taken from [39].

for understanding spatial correlations, and while recent advancements also propose the promising use of convolutional neural networks for time series analysis [40], 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 [41]. That is the gradient tends to diverge exponentially throughout 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 [42]. The inner workings of a long short-term memory unit is shown in figure 2.15. The underlying

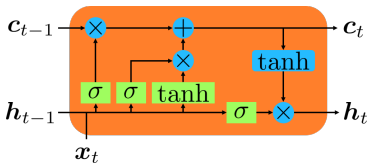


Figure 2.15: Long short-term memory unit with cell states $\boldsymbol{c}_i$, hidden states $\boldsymbol{h}_i$, input $\boldsymbol{x}_t$, and activation functions $\sigma/\tanh$, as well as addition $+$ and multiplication $\times$ operators.

mathematical operations can therein be expressed as follows

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_{ii}\boldsymbol{x}_t + \boldsymbol{b}_{ii} + \boldsymbol{W}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{hi}) \tag{2.117}$$
$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}_{if}\boldsymbol{x}_t + \boldsymbol{b}_{if} + \boldsymbol{W}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{hf}) \tag{2.118}$$
$$\boldsymbol{g}_t = \tanh(\boldsymbol{W}_{ig}\boldsymbol{x}_t + \boldsymbol{b}_{ig} + \boldsymbol{W}_{hg}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{hg}) \tag{2.119}$$
$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_{io}\boldsymbol{x}_t + \boldsymbol{b}_{io} + \boldsymbol{W}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{ho}) \tag{2.120}$$
$$\boldsymbol{c}_t = \boldsymbol{f}_t \cdot c_{t-1} + \boldsymbol{i}_t \cdot \boldsymbol{g}_t \tag{2.121}$$
$$\boldsymbol{h}_t = \boldsymbol{o}_t \cdot \tanh(\boldsymbol{c}_t), \tag{2.122}$$

where $\cdot$ is an element-wise multiplication. The sigmoid function σ, which ranges from 0 to 1, assures that the input gate $\boldsymbol{i}_t$, the forget gate $\boldsymbol{f}_t$, and the output gate $\boldsymbol{o}_t$ 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 2.16.

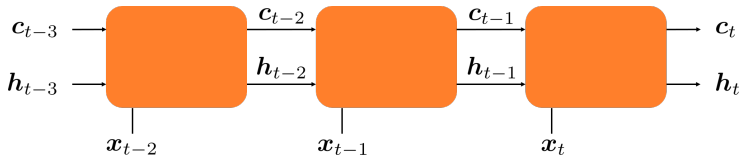


Figure 2.16: Chain of long short-term memory units for temporal understanding of the input sequence $\boldsymbol{x}_i$.

Backpropagation

The currently most popular way to train a neural network is backpropagation, which got first introduced in [43]. It has no biological equivalent but poses an effective way to optimize a huge number of parameters. Newer methods use evolutionary algorithms and treat network parameters as population that develops over time [44], but we will not consider them further. The reason 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 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 $\boldsymbol{W}_l$ of layer l, is just given as

$$\frac{\partial L}{\partial \boldsymbol{W}_l} = \boldsymbol{\delta}_l \boldsymbol{x}_{l-1}^T, \tag{2.123}$$

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

$$\boldsymbol{\delta}_N = \frac{\partial L}{\partial \boldsymbol{x}_N} \cdot f_N'(\boldsymbol{W}_N \boldsymbol{x}_{N-1}) \tag{2.124}$$

$$\boldsymbol{\delta}_l = \boldsymbol{W}_{l+1}^T \boldsymbol{\delta}_{l+1} \cdot f_l'(\boldsymbol{W}_l \boldsymbol{x}_{l-1}), \tag{2.125}$$

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 $\boldsymbol{\alpha}_{\boldsymbol{W}_l^t}$ as follows

$$\boldsymbol{W}_l^{t+1} = \boldsymbol{W}_l^t - \boldsymbol{\alpha}_{\boldsymbol{W}_l^t} \cdot \frac{\partial L}{\partial \boldsymbol{W}^l}, \tag{2.126}$$

where $\cdot$ is an element-wise multiplication.

2.2.2 Behavioral Cloning

Behavioral cloning in itself is not always related to machine learning but poses one possible way of training a neural network in a supervised manner. The presented concept is easy to understand and got inspired by [45], where they used it 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 2.1. 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 2.17. It simply 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 2.2.4. The timestamp allows to correlate seen images to desired velocities afterward, 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 essential 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. Other algorithms 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.

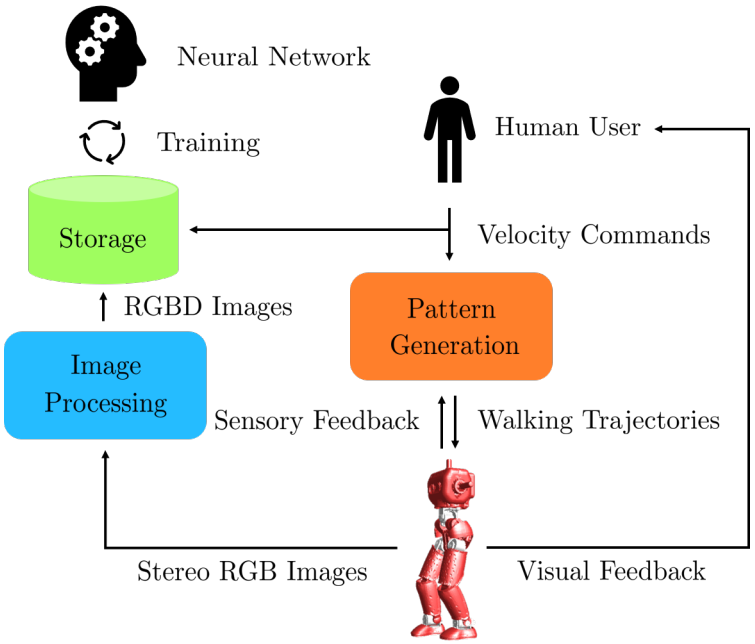


Figure 2.17: Pipeline for behavioral cloning. The neural network is trained on stored RGBD images, and corresponding velocity commands that are correlated by a timestamp

2.2.3 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 2.18, 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

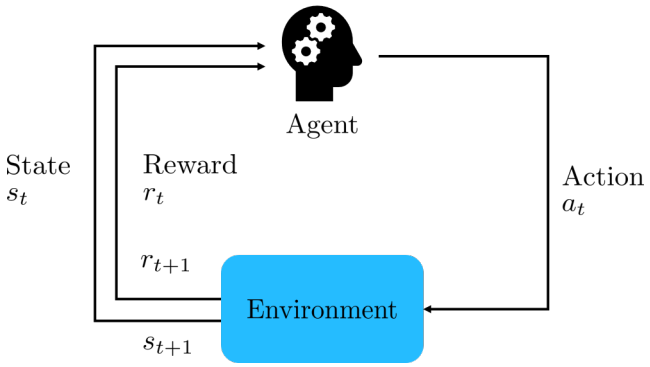


Figure 2.18: Reinforcement learning setup. As the agent interacts with the environment, the state of the environment changes.

difficulty in optimizing the policy π_θ, is to have an agent to discard immediate

rewards over future expected rewards. For discrete action spaces, this got well solved by deep Q-learning [46]. Different approaches for continuous action spaces like trust region policy optimization [47] are rather complicated. The currently most elegant way of solving continuous control problems in a reinforcement learning setup, is proximal policy optimization [48], 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 $\mathbb{E}\left[\sum_{t=0}^{\infty} r_t\right]$ is maximized. For the incremental update, it is therefore required to find the gradient of this expression

$$\nabla_\theta \mathbb{E}_{a_t \sim \pi_\theta}\left[\sum_{t=0}^{\infty} r_t\right] = \mathbb{E}_{a_t \sim \pi_\theta}\left[\sum_{t=0}^{\infty} \psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)\right], \tag{2.127}$$

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 [49], we can trade-off variance for bias and the other way around, by replacing ψ_t for the general advantage estimate $\hat{A}_t^{\mathrm{GAE}}$ as follows

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V, \tag{2.128}$$

where $\delta_{t,l}^V = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the temporal difference [50], 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 based on 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

$$L^{\mathrm{CLIP}} = \min(\rho_t(\theta)\hat{A}_t, \mathrm{clip}(\rho_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t), \tag{2.129}$$

where $\rho_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\mathrm{old}}}(a_t|s_t)}$ is the probability ratio of the old and the new policy. The loss is shown in figure 2.19, 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 estimate, it assures that the gradient points away from these policies. The total objective $L_t^{\mathrm{CLIP+VF+S}}$ 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 2.130).

$$L_t^{\mathrm{CLIP+VF+S}} = \mathbb{E}\left[L_t^{\mathrm{CLIP}}(\theta) - c_1 L_t^{\mathrm{VF}}(\theta) + c_2 S\left[\pi_\theta\right](s_t)\right] \tag{2.130}$$

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

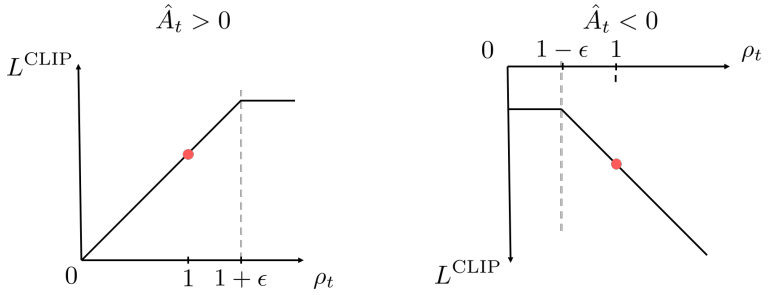


Figure 2.19: Policy gradient clipping in proximal policy optimization for positive and negative advantage estimates $\hat{A}_t$.

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

Algorithm 1: PPO, Actor-Critic Style

2.2.4 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 network with some 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 [51], 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 [52]. This method works best for edge filtered images, as will become clear soon. To obtain edge filtered images $\boldsymbol{E}$, the stereo RGB images are first converted into grayscale $\boldsymbol{G}$, which are then convolved

with the Sobel kernel $\boldsymbol{S}_x$ along the horizontal axis [53] (equation 2.131, figure 2.20).

$$\boldsymbol{E} = \boldsymbol{S}_x * \boldsymbol{G} \tag{2.131}$$

When having a look at the Sobel kernel $\boldsymbol{S}_x$ (equation 2.132), it immediately becomes

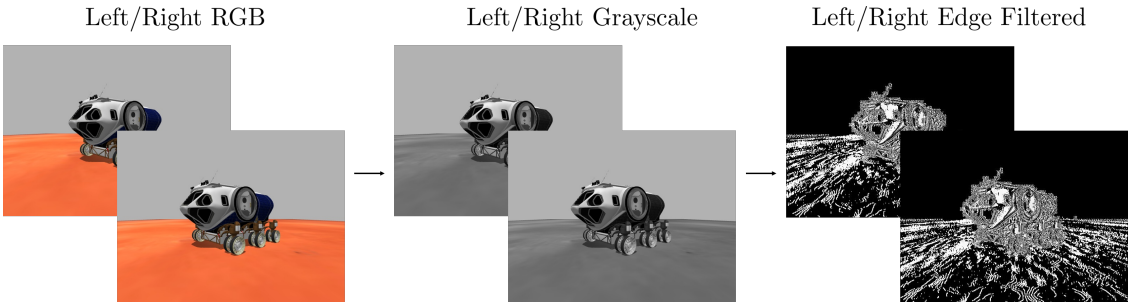


Figure 2.20: Image pre-processing to obtain edge filtered images. The images were taken within the simulation environment Gazebo (link), and show a space exploration vehicle, for which, with the friendly support of NASA, we generated a Gazebo version (link).

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.

$$\boldsymbol{S}_x = \begin{pmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{pmatrix} \tag{2.132}$$

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. 2.21). 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 $\boldsymbol{X}$ lies away from the cameras, the smaller its displacement will be. In figure 2.21, we can see that a point $\boldsymbol{X}$, which is seen by the left camera, could in principle lie anywhere on the epipolar line at $\boldsymbol{x}'$, as seen from the right camera, if there is no depth information available. It results that, to find correspondences, one only must 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

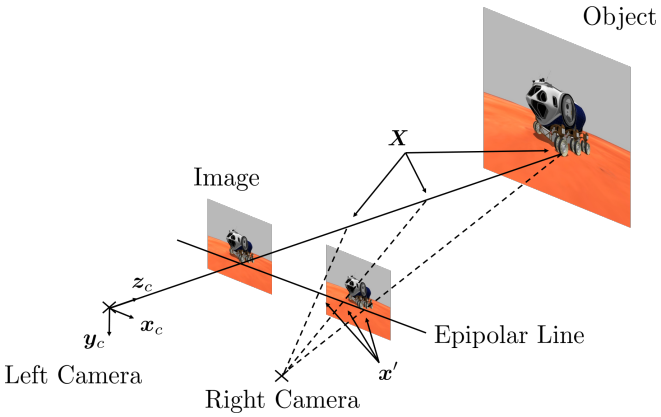


Figure 2.21: The stereo setup with a left and a right camera.

only must search in this direction. The procedure is shown in figure 2.22. 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 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 2.133, figure 2.22).

$$\mathrm{SAD}(d) = \sum_{x,y=0}^{N} |\boldsymbol{E}_{\mathrm{left}}(x,y) - \boldsymbol{E}_{\mathrm{right}}(x-d,y)| \tag{2.133}$$

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 $\boldsymbol{E}$, it is easier to find correspondences there, rather than in the grayscale or RGB images. To

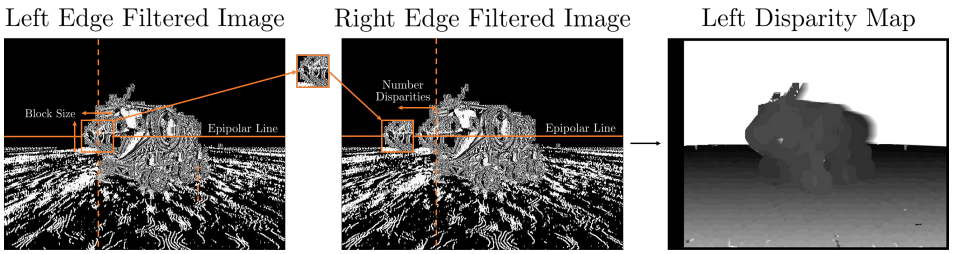


Figure 2.22: Generation of the left disparity map by the block matching algorithm.

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 $\boldsymbol{D}$ (equation 2.134, figure 2.23).

$$\mathrm{Var}(\boldsymbol{D}) = \mathbb{E}\left[\boldsymbol{D}^2\right] - \mathbb{E}\left[\boldsymbol{D}\right]^2 \tag{2.134}$$

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

$$\boldsymbol{K} = \alpha \begin{pmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{pmatrix} \tag{2.135}$$

$$\mathbb{E}\left[\boldsymbol{D}\right] = \boldsymbol{K} * \boldsymbol{D}, \tag{2.136}$$

where $\alpha = \frac{1}{\text{width} \cdot \text{height}}$ is the normalization factor. The expectation value of the disparity map squared $\mathbb{E}\left[\boldsymbol{D}^2\right]$ 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 $\text{Con}(\boldsymbol{D})$ is a measure for the certainty of the computed disparity, and is defined to be linearly dependent on the variance as follows

$$\text{Con}(\boldsymbol{D}) = \max\left(1 - r\text{Var}(\boldsymbol{D}), 0\right), \tag{2.137}$$

where r is a roll-off factor that defines the change of confidence with growing variance. The resulting disparity confidence is shown in figure 2.23 and is used to outweigh outlying disparity values from the final weighted least squares disparity map. Before that, we further introduce an additional measure for the prevention

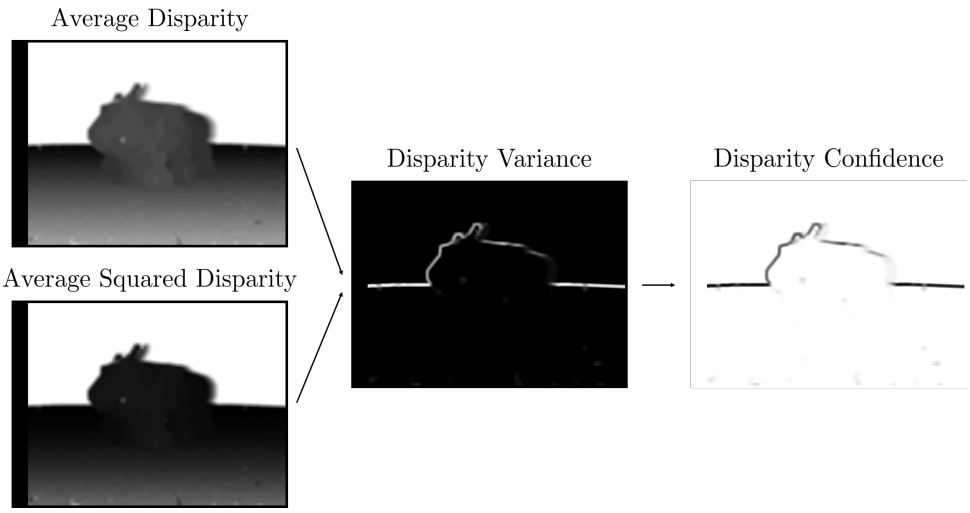


Figure 2.23: Generation of the confidence map from the variance within the disparity map.

of accidentally assigned correspondences in the initial block matching algorithm by using a left-right consistency check [54]. Therefore, the block matching algorithm is used in the right image, and we search for correspondences in the left image. In contrast to the computation of the left disparity map $\boldsymbol{D}_{\text{left}}$, for the right disparity map $\boldsymbol{D}_{\text{right}}$, we only need to check for correspondences along the epipolar line in the

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

$$\boldsymbol{L}(x,y) = \begin{cases} \min\left[\mathrm{Con}(\boldsymbol{D}_{\text{left}})(x,y), \mathrm{Con}(\boldsymbol{D}_{\text{right}})(x+d_{\text{left}},y)\right] & \text{for } \Delta d < t \\ 0 & \text{else} \end{cases}, \tag{2.138}$$

where d_{left} is the disparity of $\boldsymbol{D}_{\text{left}}$ at position (x,y), and therefore represents the index shift which results from the block matching algorithm. Furthermore, if $\Delta d = \boldsymbol{D}_{\text{left}}(x,y) + \boldsymbol{D}_{\text{right}}(x+d_{\text{left}},y)$ is smaller than a threshold t, then the left-right consistency $\boldsymbol{L}$ 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 2.24). 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 $\boldsymbol{C}$ is computed via equation 2.139 (figure 2.24).

$$\boldsymbol{C} = \boldsymbol{L} \cdot \boldsymbol{D}_{\text{left}}, \tag{2.139}$$

where $\cdot$ is an element-wise multiplication. Further, the weighted least squares filter is based on the idea of a bilateral filter [55], and it will try to minimize an energy function $J(\boldsymbol{U})$, which takes the original grayscale image as guidance to compute a weight $w_{p,q}$ for neighboring pixels p, and q as follows

$$w_{x,y,i,j}(g) = exp(-|g_{x,y} - g_{i,j}|/\sigma). \tag{2.140}$$

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 $J(\boldsymbol{U})$ that we try to minimize

$$J(\boldsymbol{U}) = \sum_{x,y} \left[(u_{x,y} - c_{x,y})^2 + \lambda \sum_{(i,j)\in\mathcal{N}(x,y)} w_{x,y,i,j}(g)(u_{x,y} - u_{i,j})^2 \right], \tag{2.141}$$

where $c_{x,y}$ are single pixels of the consistency weighted disparity map. The formulation of this energy function results in a solution $\boldsymbol{U}$ that encourages the propagation of disparity values from high- to low-confidence regions (figure 2.24). 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 $\boldsymbol{D}_{\text{final}}$ is then obtained by normalizing the resulting image $\boldsymbol{U}$ with

$$\boldsymbol{D}_{\text{final}} = \frac{\boldsymbol{U}}{\mathrm{WLS}(\boldsymbol{L})}, \tag{2.142}$$

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

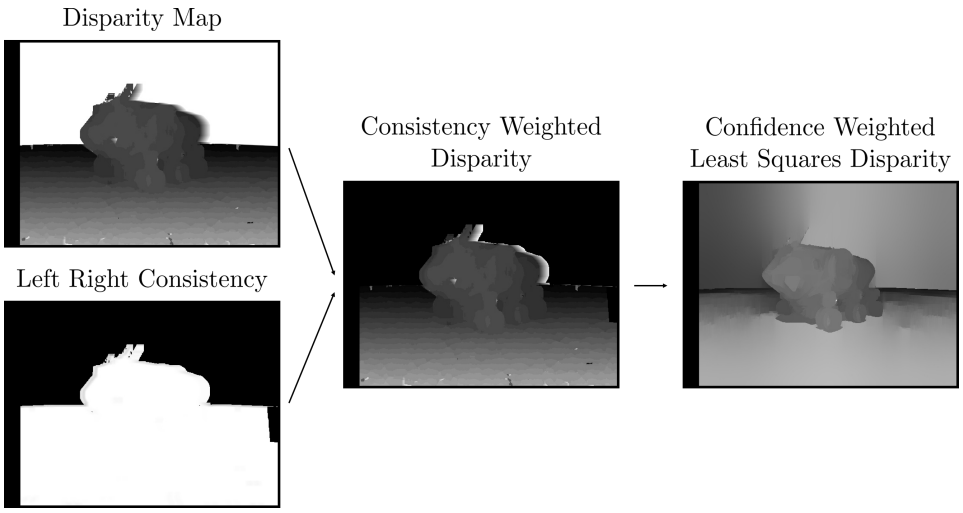


Figure 2.24: Generation of the confidence weighted least squares disparity from the disparity map, and the left-right consistency.

As already mentioned in figure 2.21, the assumption of an only translated stereo camera pair is rarely correct. Besides, 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 2.25. 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 $\boldsymbol{X}_c = (X, Y, Z)^T$ is then simply projected to the image plane by the camera matrix $\boldsymbol{K}$ with

$$\boldsymbol{x}_c = \boldsymbol{K}\boldsymbol{X}_c = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \boldsymbol{X}_c. \tag{2.143}$$

Therein, $\boldsymbol{K}$ 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.

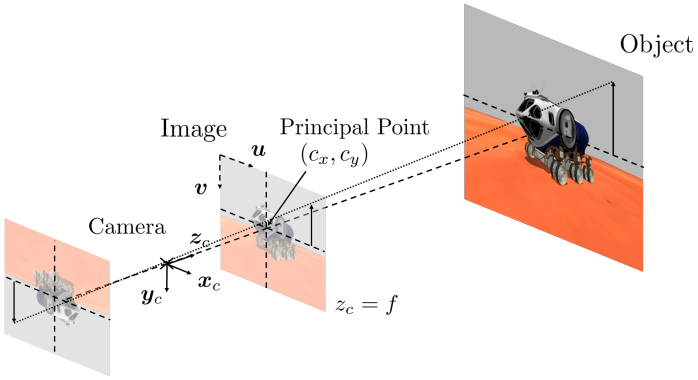


Figure 2.25: Pinhole camera model.

According to [56], we model radial and tangential distortion by

$$x_{c,u} = x_{c,d}(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2x_{c,d}^2) + 2p_2 x_{c,d} y_{c,d} \tag{2.144}$$

$$y_{c,u} = x_{c,d}(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x_{c,d} y_{c,d} + p_2(r^2 + 2y_{c,d}^2), \tag{2.145}$$

where

$$\begin{aligned}
(x_{c,d}, y_{c,d}) &= \text{distorted image points within the camera frame } c, \\
&\quad \text{as projected onto the image plane} \\
(x_{c,u}, y_{c,u}) &= \text{undistorted image points within the camera frame } c, \\
&\quad \text{as projected by an ideal pinhole camera} \\
k_n &= \text{n}^{\text{th}} \text{ radial distortion coefficient} \\
p_n &= \text{n}^{\text{th}} \text{ tangential distortion coefficient} \\
r &= \sqrt{x_{c,d}^2 + y_{c,d}^2}.
\end{aligned}$$

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 2.26). 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

$$\boldsymbol{x}_{nm} = \begin{pmatrix} wa & ha & 0 & 1 \end{pmatrix}^T, \tag{2.146}$$

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 $\boldsymbol{R}$ and translation $\boldsymbol{t}$, which transforms the object points to the image plane. They are estimated by solving a perspective-n-point problem

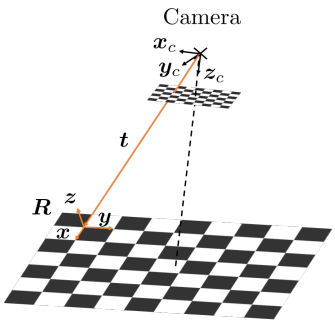
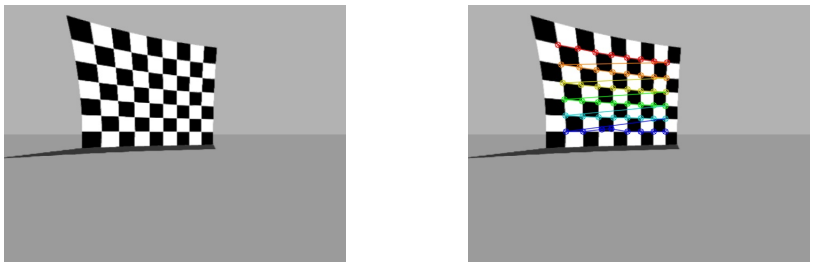


Figure 2.26: Calibration pattern, as observed from the camera's coordinate system C. Within the object's coordinate system, all chessboard corners lie at a zero z-position.

[57]. Therefore, as shown in figure 2.27 (b), a corner detecting algorithm finds the corners $\boldsymbol{x}_{c,wh}$ within the image plane. Under the assumption of known intrinsic camera parameters, $\boldsymbol{x}_{c,wh}$ are then being undistorted according to equations 2.144 and 2.145. Each $\boldsymbol{x}_{nm}$ can then be transformed to the camera's frame, and further



(a) (b)

Figure 2.27: Distorted calibration pattern (a), and the image points as found by the algorithm (b).

be projected onto the image plane via

$$\boldsymbol{x}_{c,wh} = \boldsymbol{K} \begin{pmatrix} \boldsymbol{R} & \boldsymbol{t} \end{pmatrix} \boldsymbol{x}_{wh}, \tag{2.147}$$

where $\boldsymbol{R}$ describes the rotation, and $\boldsymbol{t}$ the translation of the camera frame to the object frame. Then, equations 2.144 and 2.145 are applied to obtain the undistorted image points from $\boldsymbol{x}_{c,wh}$. The undistorted image points $\boldsymbol{x}_{c,wh,u}$ are then being reprojected by inverting the rotation and translation via

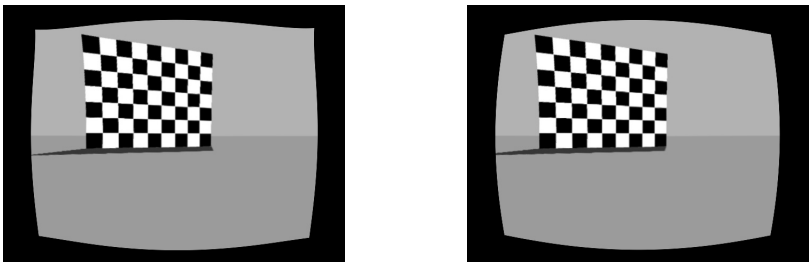
$$\boldsymbol{x}_{wh,u} = \begin{pmatrix} \boldsymbol{R} & \boldsymbol{t} \end{pmatrix}^{-1} \boldsymbol{x}_{c,wh,u}, \tag{2.148}$$

from which we compute the re-projection error $\Delta x = ||\boldsymbol{x}_{wh,u} - \boldsymbol{x}_{wh}||_2$. To find the intrinsic parameters, a Levenberg-Marquardt algorithm then optimizes them in an iterative scheme to minimize the re-projection error until convergence [58]. 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 $\boldsymbol{F}$, which transforms points from the left camera's view $\boldsymbol{x}_{c_{\text{left}}}$, to points $\boldsymbol{x}_{c_{\text{right}}}$, as seen by the right camera via

$$\boldsymbol{x}_{c_{\text{right}}}^{T} \boldsymbol{F} \boldsymbol{x}_{c_{\text{left}}} = \boldsymbol{0} \tag{2.149}$$

The mapping enables us to rectify the left and the right image [59], using the rectification transforms $\boldsymbol{R}_i$, which means that we can compute homography transforms that align epipolar lines within the images (figure 2.28), which were previously defined by the fundamental matrix. These homography transforms map the images, as



(a) (b)

Figure 2.28: 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 2.27, 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 $\boldsymbol{P}_i$. 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.

3 Methods

While there is an extensive guide on the installation of the implemented software within the appendix A, the purpose of this chapter is to give the future reader a good understanding of it. The code itself is generic, such that it is not dependent on the used robot at all, for why it is very well-suited to perform future research on arbitrary humanoid robots. Therefore, section 3.1 will give the reader a broad overview of the overall code's structure, in order to allow for a quick assessment of possibly reusable implementations. The completely decoupled pattern generation and deep learning parts of the software are hence explained first in section 3.2, and section 3.3, respectively. Following that, Heicub, our humanoid robot, and the communication with it will be explained in section 3.4.

3.1 Code Structure

As shown in figure 3.1, there are four main folders within the implemented code. Those include the libraries inside the libs folder, the models, among them Heicub's URDF model for computing kinematics, and another one for pure visualization with MeshUp [60], the sh folder, which contains shell scripts for easy execution, and furthermore the source folder, which relies on the libraries to build executables from. Moreover, while these executables are very much dependent on the robot at hand, as they are coupled to the robot's communication system, they pose some nice examples on how the pattern generator can be embedded into a robot's operating system. Now in order to reuse the code for a different robot, one could either install the libraries and include the headers, as explained in section A.1, or replace the input-output module (io_module) by a new one and keep the folder's structure. Nonetheless, we will only focus on the libraries' usage by introducing simple examples in the following sections and keep the details of implementation on a different operating system up to the future reader. We will further bridge the gap between the background of section 2 by explaining the different methods, which we implemented, and we will further highlight its correlation with the YAML [61] configuration files that are a building block of each library.

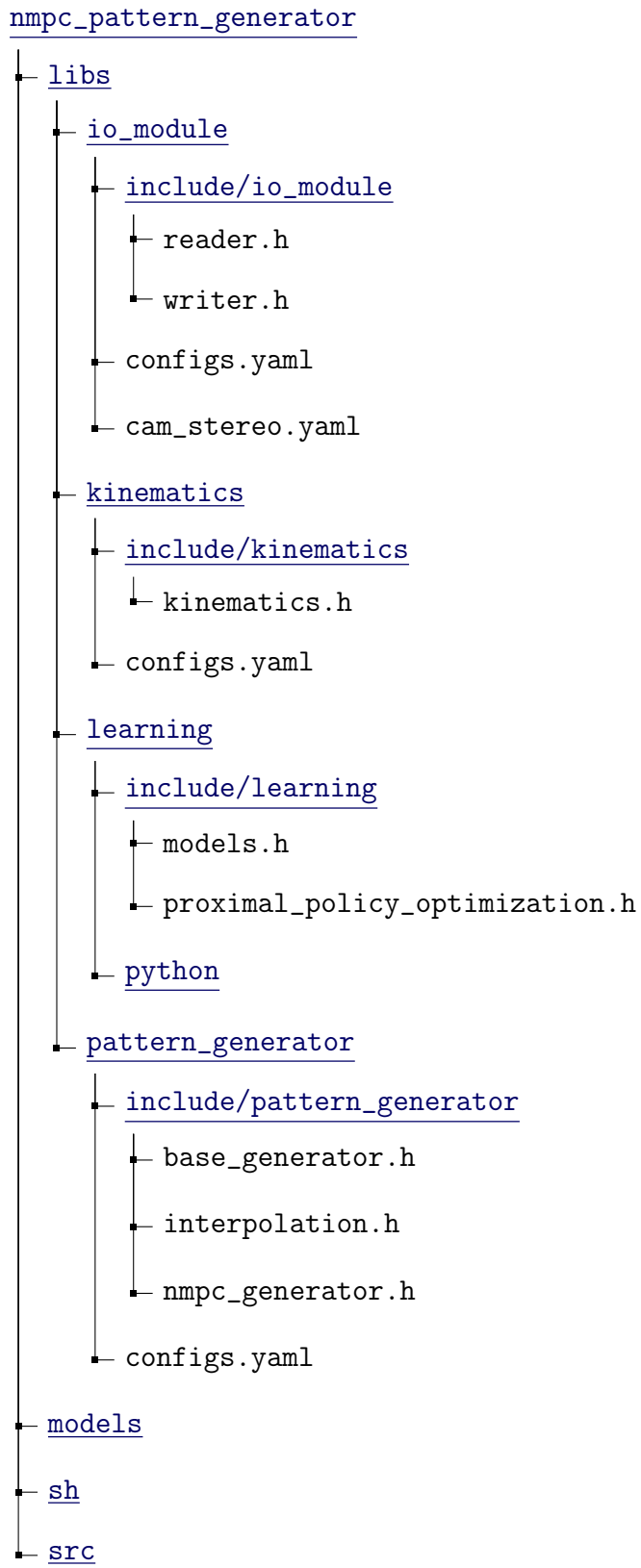


Figure 3.1: Folder structure of the code, which got implemented for this thesis. The code is freely available on GitHub at the provided link. Install instruction can be found in the appendix A.

3.2 Implementation of the Pattern Generation Library

The pattern generation library is built upon the fast linear algebra library Eigen [62], which makes it perfectly compatible with the rigid body dynamics library RBDL [63] that is used for the kinematics. Its main duty is to build the formulation of the optimal control problem that was introduced with much detail in section 2.1.2. Starting from the linear time-stepping scheme (equations 2.33-2.35), the `BaseGenerator` class sets up all of the necessary matrices for the nonlinear model predictive control. What the `NMPCGenerator` class then mainly does is to inherit from `BaseGenerator` in order to sort the matrices for the sequential quadratic programming formulation of equation 2.83. All of the ordering is then being done by `NMPCGenerator::CalculateCommonExpressions`, while `NMPCGenerator::CalculateDerivatives` computes the linearization of the optimal control problem, which got explained in detail in section 2.1.2. Then to solve the sequential quadratic programming task, the pattern generation library utilizes qpOASES [64], and the results are being interpolated by the `Interpolation` class, which relies on the theory of section 2.1.3. However, to use the pattern generation library, the user does not have to deal with the private methods and can only use the public interface. A typical use case would involve the initialization of a `NMPCGenerator` object like so

```
// Initialize pattern generator.
const std::string config_file_loc = "/path/to/configs.yaml";
NMPCGenerator nmpc(config_file_loc);
```

where the YAML configurations file loads the robot specific settings at run time, that is they can be changed without having to recompile the code. An example of the pattern generator's configurations file is shown in figure 3.2. One then one must set the initial values as follows

```
// Pattern generator preparation.
nmpc.SetSecurityMargin(nmpc.SecurityMarginX(),
                       nmpc.SecurityMarginY());

// Set initial values.
PatternGeneratorState pg_state = {nmpc.Ckx0(),
                                  nmpc.Cky0(),
                                  nmpc.Hcom(),
                                  nmpc.Fkx0(),
                                  nmpc.Fky0(),
                                  nmpc.Fkq0(),
                                  nmpc.CurrentSupport().foot,
                                  nmpc.Ckq0()};

nmpc.SetInitialValues(pg_state);
```

The interpolation is then done based on the pattern generator's state, and therefore it takes a reference to it as follows

```
Interpolation ip_nmpc(nmpc);
ip_nmpc.StoreTrajectories(true);
```

which brings us to the main loop

```
Eigen::Vector3d velocity_reference(0.1, 0., 0.);

// Pattern generator event loop.
for (int i = 0; i < 100; i++) {
    std::cout << "Iteration: " << i << std::endl;

    // Set reference velocities.
    nmpc.SetVelocityReference(velocity_reference);

    // Solve QP.
    nmpc.Solve();
    nmpc.Simulate();
    ip_nmpc.InterpolateStep();

    // Initial value embedding by internal states and simulation.
    pg_state = nmpc.Update();
    nmpc.SetInitialValues(pg_state);
}
```

These are the main methods a user would want to interact with. `Simulate()` computes the states on the preview horizon for the optimization, according to equations 2.33-2.35, given the current state, which is then being updated by `Update()`. It is possible to elaborate feedback by changing the CoM, and, or the feet members of the `PatternGeneratorState` struct. Afterwards, the trajectories can be stored with

```
// Save interpolated results.
Eigen::MatrixXd trajectories = ip_nmpc.GetTrajectories().transpose();
WriteCsv("trajectories.csv", trajectories);
```

The trajectories, which are being written out with this method, are ordered according to c_k^x, $\dot{c}_k^x$, $\ddot{c}_k^x$, c_k^y, $\dot{c}_k^y$, $\ddot{c}_k^y$, c_z, c_k^θ, $\dot{c}_k^\theta$, $\ddot{c}_k^\theta$, z_k^x, ^y_k, z_k^z, $f_k^{L,x}$, $f_k^{L,y}$, $f_k^{L,z}$, $f_k^{L,\theta}$, $f_k^{R,x}$, $f_k^{R,y}$, $f_k^{R,z}$, $f_k^{R,\theta}$. An exemplary plotting routine in python is provided at the following link.

Specifications	`# Specifications.`
Double + single support time	**t_step**: 3.2
Used to shrink convex hull $\boldsymbol{p}_i$ in fig. 2.7, 2.4	**security_margin**: [0.02, 0.02]
The convex hull $\boldsymbol{p}_i^L$ in fig. 2.8	**left_foot_convex_hull**: [−0.11, 0.14, ...]
The convex hull $\boldsymbol{p}_i^R$ in fig. 2.8	**right_foot_convex_hull**: [−0.11, −0.14, ...]
Used to compute convex hull $\boldsymbol{p}_i$ in fig. 2.7, 2.4	**foot_width**: 0.1
Used to compute convex hull $\boldsymbol{p}_i$ in fig. 2.7, 2.4	**foot_length**: 0.2
Distance between feet frames	**foot_distance**: 0.14
Interpolation	`# Interpolation.`
Evaluate eq. 2.89-2.91 at $t = nt_{\text{command\_period}}$	**command_period**: 0.01
Initially stand still for n_{still} preview horizons	**n_still**: 0
Double support time	**t_ds**: 1.6
Maximal step height $z_{T/2}$ in eq. 2.94	**step_height**: 0.03
Initial Values	`# Initial values.`
Initial x-CoM $\boldsymbol{c}_k^x$ in eq. 2.33-2.35	**com_x**: [0.0, 0.0, 0.0]
Initial y-CoM $\boldsymbol{c}_k^y$ in eq. 2.33-2.35	**com_y**: [0.05, 0.0, 0.0]
Initial z-CoM c_z in eq. 2.40	**com_z**: 0.45
Initial θ-CoM $\boldsymbol{c}_k^\theta$ in eq. 2.33-2.35	**com_q**: [0.0, 0.0, 0.0]
Initial support foot	**support_foot**: left
Initial support foot f_k^x in eq. 2.47	**foot_x**: 0.0
Initial support foot f_k^y in eq. 2.47	**foot_y**: 0.07
Initial support foot $\boldsymbol{f}_k^\theta$ in eq. 2.33-2.35	**foot_q**: 0.0
Environment	`# Environment.`
Gravity in eq. 2.40	**gravity**: 9.81
Obstacle	`# Obstacle.`
False if no obstacle is used	**obstacle**: false
Obstacle position x_0 in eq. 2.79	**x_pos**: 10
Obstacle position y_0 in eq. 2.79	**y_pos**: 10
Obstacle radius R in eq. 2.79	**radius**: 0.5
Optimization	`# Optimization.`
Preview intervals N in eq. 2.30	**n**: 16
Preview time T in eq. 2.31	**t**: 0.4
CoM and feet position feedback time	**t_feedback**: 0.4
Weight in eq. 2.43-2.46	**alpha**: 1
Weight in eq. 2.43-2.46	**beta**: 1e2
Weight in eq. 2.43-2.46	**gamma**: 1e−2
Maximally allowed CPU time for qpOASES	**cpu_time**: 0.1
Maximally allowed number of calculations for qpOASES	**nwsr**: 1000

Table 3.1: The configurations YAML file for the pattern generator, and its relation to the background of section 2. It can be found at the provided link.

To run the code on a real robot, one must use inverse kinematics, which turn the obtained CoM, and feet positions, into joint angles $\boldsymbol{q}$. Within this work, this is done with RBDL, and implemented in the `Kinematics` class, which also provides methods to generate feedback via forward kinematics. It is therefore required to initialize a `Kinematics` object as follows

```
// Kinematics.
const std::string config_file_loc = "/path/to/configs.yaml";
Kinematics ki(config_file_loc);
```

which loads the configurations from a YAML file that can be found in figure 3.2. One of the most important settings in there is the location of the robot's URDF model. For Heicub, this model is located in the models folder from figure 3.1. It

is then required to initialize the inverse kinematics with an appropriate guess, such that it does not converge to an unwanted local minimum. We can do so by setting the initial joint angles with

```
// Initialize position with a good guess.
Eigen::VectorXd q_init(21 /*heicub's degrees of freedom*/);
q_init.setZero();

q_init = ...;

ki.SetQInit(q_init);
```

and finally compute the desired joint angles by

```
// Get the desired initial state of the robot.
Eigen::MatrixXd traj = ip_nmpc.GetTrajectories();

// Initialize inverse kinematics.
Eigen::MatrixXd com_traj(4, traj.cols());
Eigen::MatrixXd lf_traj(4, traj.cols());
Eigen::MatrixXd rf_traj(4, traj.cols());

ki.Inverse(com_traj, lf_traj, rf_traj);
Eigen::MatrixXd q_traj = ki.GetQTraj();
```

How to use these joint angles, in order to get it running on Heicub, is described in section 3.4.2. We have now seen that it is only required to set the reference velocity to control the robot, and the next section will explain how this can be done with the use of a neural network.

URDF Model	`# Location of the urdf model.`
URDF model for kinematics with RBDL	**urdf_loc**: ../../models/icub_heidelberg01_no_weights.urdf
Initial Inverse Kinematic Runs	`# Number of pre-initializations for inverse kinematics.`
Run inverse kinematics initially n_{init} times	**n_init**: 10
Inverse Kinematic Parameters	`# Parameters for the inverse kinematics.`
Tolerance used for convergence detection	**step_tol**: 1e-12
Damping λ in eq. 2.113	**lambda**: 1e-2
Maximally allowed optimization steps	**num_steps**: 1000
Body Points	`# Body points.`
Computed automatically by RBDL	**com_body_point**: [0.0, 0.0, 0.0]
Left foot inverse kinematics constraint	**lf_body_point**: [0.0, 0.0, 0.0]
Right foot inverse kinematics constraint	**rf_body_point**: [0.0, 0.0, 0.0]

Table 3.2: The configurations YAML file for the kinematics, and its relation to the background of section 2. It can be found at the provided link.

3.3 Deep Learning Integration

Again, the first step within the deep learning pipeline is the image processing, as already introduced in section 2.2.4. The camera calibration within this work was

done using OpenCV [65]. Exemplary code with which we did the depth map parameter tuning in section 4.2.2, can be found at the provided link. It relies on the rectification matrices $\boldsymbol{R}_i$, and the projection matrices $\boldsymbol{P}_i$ that got explained in section 2.2.4. YAML files store them inside of the io_module folder of figure 3.1. They are highly dependent on the robot but will not change for Heicub over time, for why the future reader will be able to reuse them. As already explained in section 2.2.2, Heicub's recorded images and the corresponding velocities got stored inside of a folder. Locations to the image files, as well as the velocities, were then stored in a text file. For a faster prototyping, we then decided to implement the deep learning training pipeline within Python with PyTorch [66], for which the routine can be found at the provided link. The trained model got then converted to a just in time (JIT) script that loads into the executables in the src folder of figure 3.1 via

```
auto module = torch::jit::load(net_location);
```

The code to convert a model, which got trained in Python, to C++, is part of the python folder and can be found at the following link. For Heicub, there was an additional step to be done, which is explained in section 3.4.2, but given the camera images and depth maps are provided as `cv::Mat`, it is possible to convert them to a `torch::Tensor`, which can then be concatenated into a single RGBD tensor. We temporarily saved a sequence of these tensors, which enabled us to forward them through our long short-term memory based neural network, for which the architecture is given in figure 4.15. For the behavioral augmentation, this can, for example, be seen at the provided link. Once the neural network predicts a desired velocity for the pattern generation, one only has to convert the desired velocity back to an `Eigen::Vector3d`, which is the datatype that the pattern generation library works with. This velocity is then forwarded, just as described in section 3.2, via the `NMPCPatternGenerator::SetVelocityReference` method, and a new iteration is executed. In the following section, we will explain how the developed pipelines can be used together with Heicub. It is therefore mainly required to implement the control loop of figure 2.1.

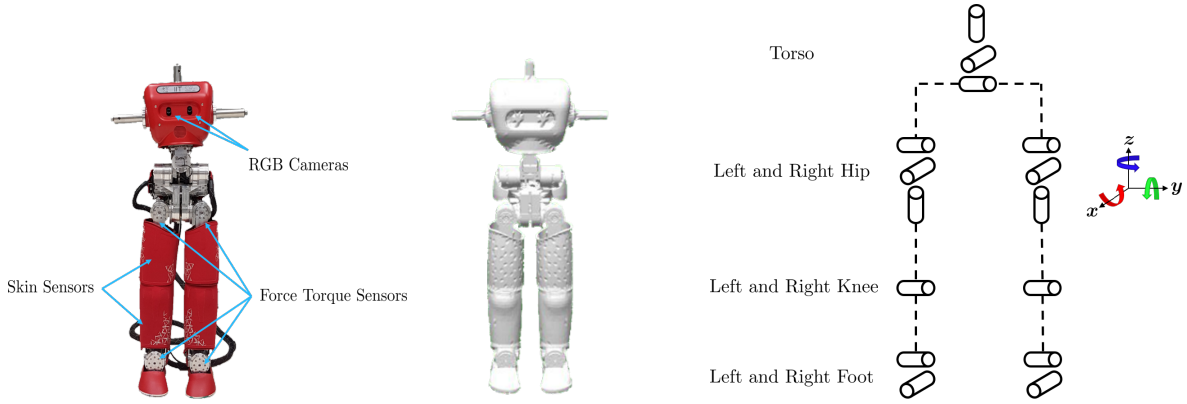
3.4 Heicub

Heicub is a descendant of the iCub, which was specially designed for optimal control in locomotion at the Istituto Italiano di Tecnologia in Genova. It is used within this thesis to test the implemented algorithms, for which it provides all the required actuators and sensors, which are shortly introduced in the following section.

3.4.1 Heicub's Sensors

As shown in figure 3.2 (c), Heicub has a total of 21 degrees of freedom, of which six correspond to the floating base, and 15 to the rotational joints. Its two RGB cameras have a resolution of 240×320 pixels at a framerate of 60 fps, and are

located within the chest, as shown in figure 3.2 (a). The force-torque sensors, with which we compute the zero moment point, are located within the ankles and the hip. According to equations 2.11 and 2.12, we only require the force-torque sensors from the ankles, where $d = 0.03\,\text{m}$. Heicub also supports skin sensors, and an inertial measurement unit, but both are not used within this thesis. The use of the robot, and all its sensors, is well described within the appendix, starting from section B. Furthermore, Heicub has a Gazebo simulation model, which is shown in figure 3.2, and for which the installation instruction can be found in section A.4. It provides the exact same functionality as its real equivalent does. This is achieved via Gazebo plugins, which communicate to the YARP network, just as Heicub does. The plugins can be installed by following the instructions in section A.5.2. The Gazebo model offers a perfect environment for software development, as we can make sure that implemented algorithms work properly, prior to their usage on the real robot, which could take harm from wrongly designed software. YARP stands short for Yet Another Robot Platform, and its installation is explained in section A.5.2. Most important for us is to understand the YARP network in more detail, as it allows us to communicate with the robot. It is therefore explained within the next section - Communication with Heicub.



(a) Heicub. (b) Gazebo model. (c) Kinematic chain.

Figure 3.2: Heicub, the robot, which we used for the evaluation of the implemented software. There exists the real robot (a), as well as a simulated version of it, which offer equivalent functionality (b). Heicub's degrees of freedom, which can be actuated, are shown in (c). All the joints are rotational joints.

3.4.2 Communication with Heicub

With YARP [67], it is possible to directly interface the robot's motors, the cameras, and the force-torque sensors. Moreover, it enables the user to run multiple programs in parallel, which can then communicate with each other, which is of special importance for the control loop that was implemented within the scope of this

thesis (see figure 3.3). As we know from the previous section, one of these threads

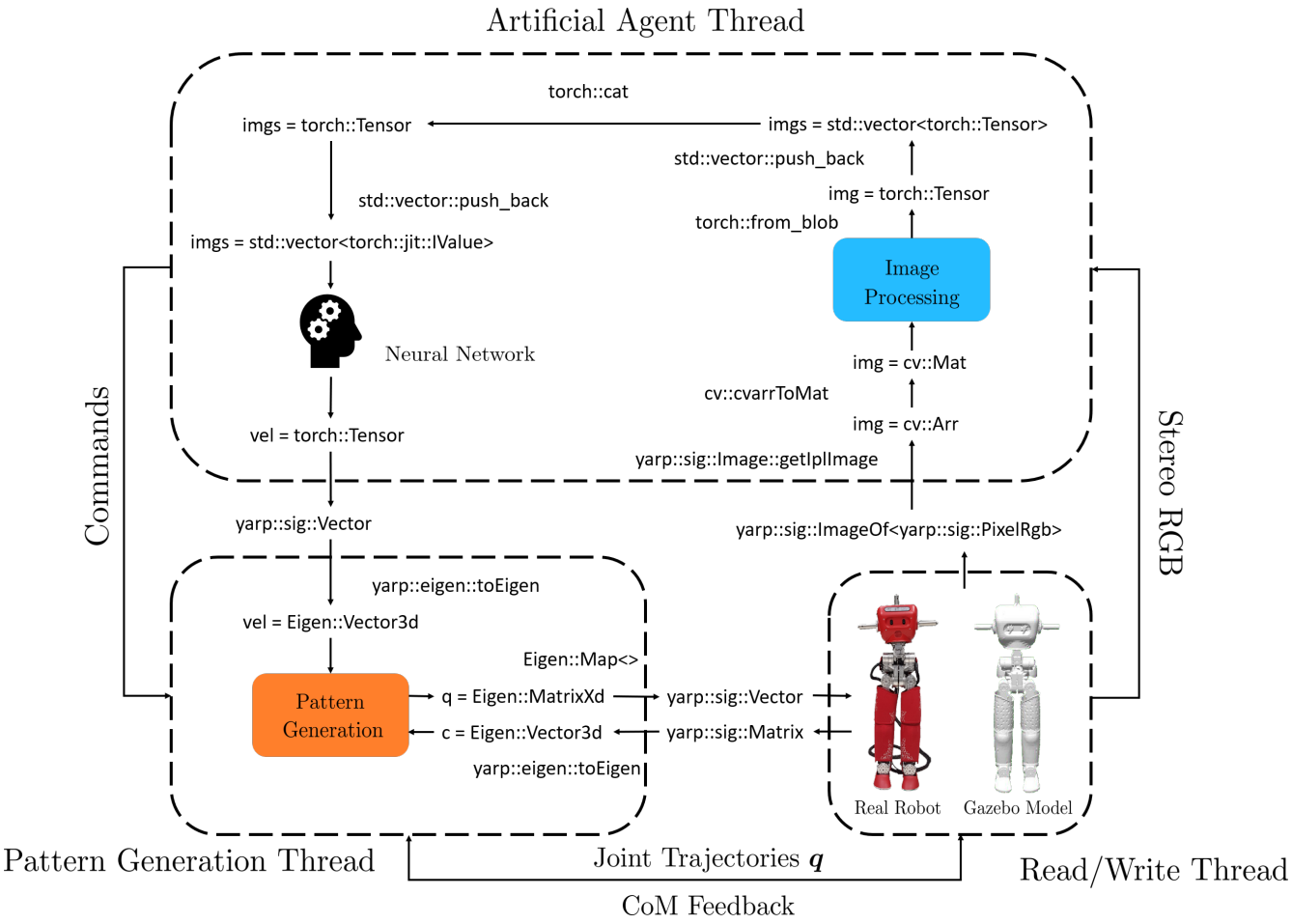


Figure 3.3: YARP is used to run multiple threads in parallel, each of which is indicated by the dashed boxes. It further enables the individual threads to communicate with each other via ports, which exchange YARP objects. It enables communication to the real robot as well as to a simulated version of it. The diagram demonstrates the types of data, which are being used, and the functions that convert them. Notice that this is an extended version of figure 2.1.

is committed to make decisions with a neural network, given RGBD images. This thread is depicted as the artificial agent thread in figure 3.3. The pattern generation thread performs the nonlinear model predictive control, given the velocity command of the artificial agent thread. Furthermore, there are two additional running threads that communicate with Heicub's motors. These threads are the read, and the write thread in figure 3.3. The classes, which implement the communication to the robot, are located within the io_module folder of figure 3.1. The `WriteJoints` class implements a `yarp::os::RateThread`, which is periodically being called, to accesses the motors, which are defined within the YAML configuration file, and changes the

motors' settings to position direct mode. Therefore, whatever is being written to the port that `WriteJoints` uses to communicate with the YARP network, and which is defined in the YAML configuration file, directly gets executed on the robot's motors. This communication to Heicub's motors corresponds to the very lowest part of figure 3.3, where, as extensively explain in section 3.2, the pattern generation uses the forward kinematics to generate joint angles $\boldsymbol{q}$, which are being written as `yarp::sig::Vector` to the `WriteJoints` rate thread. The reading of the robot's sensors is also implemented as part of the io_module folder from figure 3.1. There are several classes, which implement `yarp::os::RateThread`s for different reading tasks. Among them are the `ReadJoints` class, which reads out the motor encoders to obtain the joint angles, the `ReadCameras` class, which reads out the cameras and pushes them as `yarp::sig::ImageOf<yarp::sig::PixelRgb>` onto the network (see figure 3.3), as well as the `AppReader` class, and the `KeyReader` class, which handle the communication to the joystick app, and the terminal, respectively. Both, the `AppReader` class, and the `KeyReader` class, utilize NCurses to generate a user interface on the terminal, which is internally being trapped in a while loop until exit. They read out the input, which may originate from the joystick app, or the keyboard, and push them as the velocity commands onto the YARP network (see figure 3.3 left). The velocity commands, which are converted into an `Eigen::Vector3d` for the `NMPCGenerator::SetVelocityReference` method from section 3.2, may alternatively also originate from a neural network, which is being presented in figure 3.3. The `GenerateVelocityCommands` rate thread, which enables this feature, is implemented as part of the src folder in 3.1, as it only utilizes the provided libraries. It can be found at the provided link. Its main task is to read in the images, which are constantly being pushed to the YARP network by `ReadCameras`, and to convert them into `cv::Mat` matrices, for us to perform the image processing on them, which includes the rectification, and the depth map extraction that are explained in section 2.2.4. The `GenerateVelocityCommands` class additionally stores a sequence of the processed images in the form of a `std::vector<torch::Tensor>`. Whenever a new image is read from the YARP network, the oldest image within the `std::vector<torch::Tensor>` is being deleted, and all other images are shifted up by one index, such that the newest image is available as the first entry. This vector of tensors is then being converted into a single tensor, by concatenating the individual tensors along the first dimension, which is, by definition of the long short-term memory units, required in PyTorch. The concatenated tensor is further being converted into a `std::vector<torch::jit::IValue>`, such that the JIT script, which defines the neural network that got trained in Python (see 3.3), can forward it. The output is then obtained as a `torch::Tensor` yet again, which is being written in the form of a `yarp::sig::Vector` to the YARP network, such that, as explained above, the pattern generation can use it as input. This pipeline works equivalently on the real robot, as well as on a simulated version of it in Gazebo [68], for which install instructions are provided in the appendix A.4. Although it allowed us to prototype within the simulation, we will use the pipeline to train the real robot Heicub, as

an example for our method, on finding a fire extinguisher within the next section - Experiments.

4 Experiments

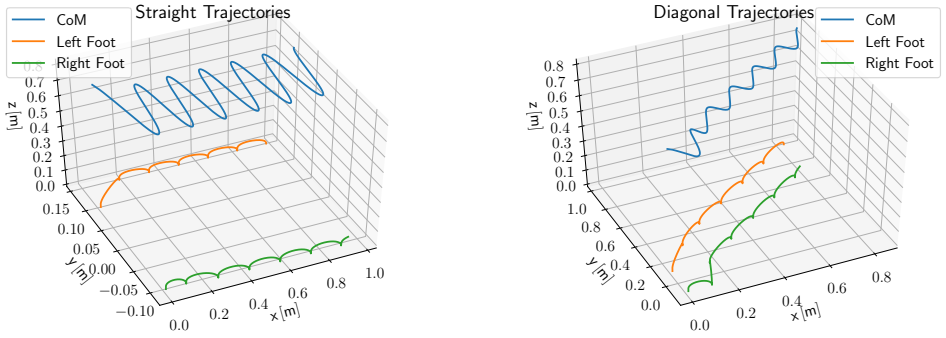
Within the experiments chapter, we will first benchmark the nonlinear model predictive control implementation of ours for purely simulated tasks in section 4.1, and then tune hyperparameters for it to run well on Heicub. This will allow us to define a standardized environment for walking experiments in section 4.1.2, to later compare user-controlled performance against autonomously controlled performance in section 4.2. Though, before we can tackle the idea of behavioral cloning, which got described in section 2.2.2, we need to meet the prerequisites, that is we will calibrate the camera, and tune the depth map parameter extraction in sections 4.2.1, and 4.2.2, 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 4.2.3. Finally, we will compare the humanly controlled robot's balance with the artificially controlled robot's balance in section 4.2.4 and investigate on the reinforcement learning approach for autonomous navigation in section 4.3.

4.1 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 4.2.3.

4.1.1 Benchmarking of Nonlinear Model Predictive Control

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 4.1 and 4.2, 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 4.1 (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 1.48 ± 0.01 ms. It was therefore possible to achieve a speed-up of around 600

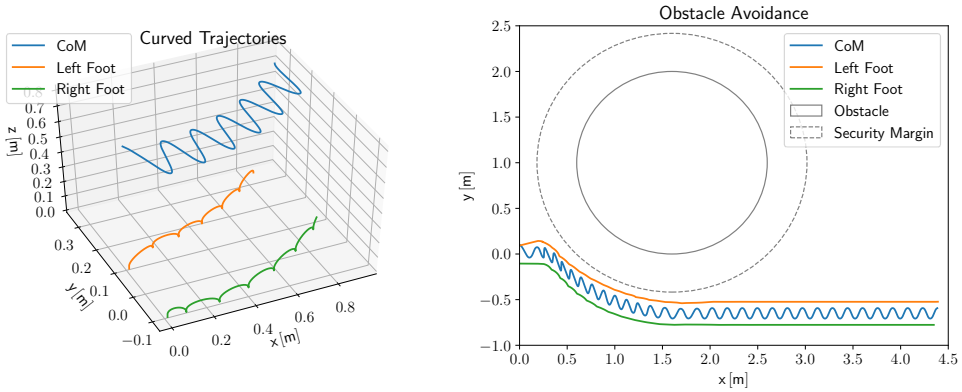


(a) Straight trajectories at $\boldsymbol{v} = \begin{pmatrix} 0.1 & 0.0 & 0.0 \end{pmatrix}^T$.

(b) Diagonal trajectories at $\boldsymbol{v} = \begin{pmatrix} 0.1 & 0.1 & 0.0 \end{pmatrix}^T$.

Figure 4.1: Simple trajectories. The velocities are given in units of $(\text{m/s} \quad \text{m/s} \quad \text{rad/s})^T$, where the first two entries describe the robot's velocity in the x-, and the y-direction, and the last entry describes the robot's angular velocity about the z-axis, form a frame that is attached to the robot. The trajectories start on the left-hand side.

percent with the presented implementation. We further demonstrate the avoidance of a convex obstacle with a security margin that keeps the robot at a safe distance in figure 4.2 (b). The used obstacle is defined to be located at $x = 1.6\,\text{m}$ and $y = 1.0\,\text{m}$, with a radius of $R = 1.0\,\text{m}$, and a security margin of $m = 0.4\,\text{m}$. To always ensure



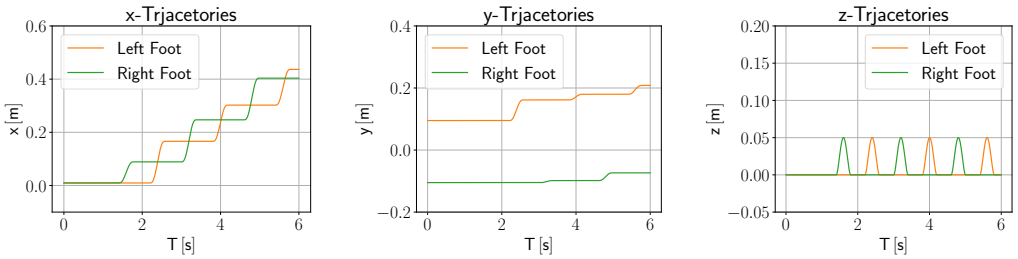
(a) Curved trajectories at $\boldsymbol{v} = \begin{pmatrix} 0.1 & 0.0 & 0.1 \end{pmatrix}^T$.

(b) Obstacle avoidance at $\boldsymbol{v} = \begin{pmatrix} 0.1 & 0.0 & 0.0 \end{pmatrix}^T$

Figure 4.2: Advanced trajectories. The velocities are given in units of $(\text{m/s} \quad \text{m/s} \quad \text{rad/s})^T$, see figure 4.1.

a smooth motion, and to benchmark the interpolation, we plotted the x-, y-, and z-trajectories for the left and the right foot, as shown in figure 4.3. The plots were

generated on the curved trajectory of figure 4.2 (a), and they reveal a continuous behavior at every time step for all dimensions. The benchmarked pattern generator



(a) Interpolated x-Trajectories (b) Interpolated y-Trajectories (c) Interpolated z-Trajectories

Figure 4.3: Interpolated foot trajectories. As explained in figure 2.9, the interpolation interpolates the feet's movement, given an initial, and a final foot position. The continuity shows that the interpolation got implemented correctly.

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.

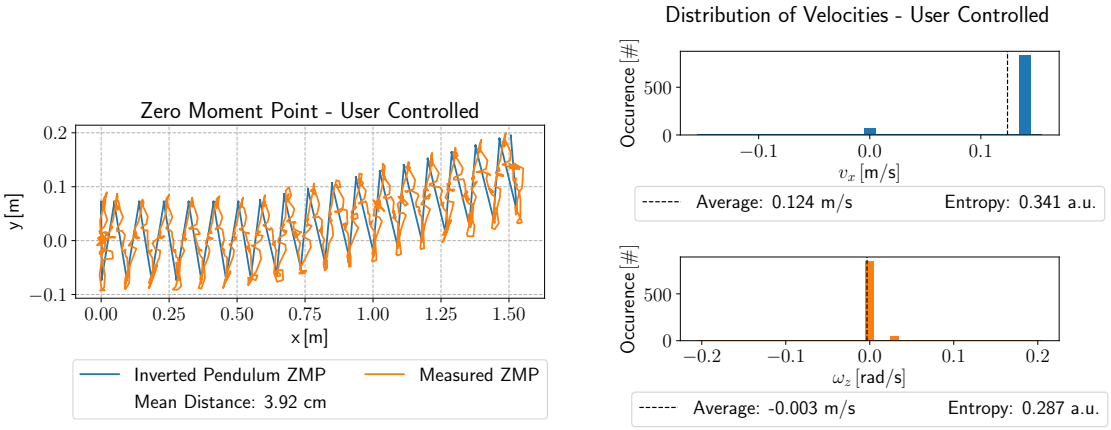
4.1.2 Performance in Test Environment

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 4.1. 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 4.2, 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,

Table 4.1: Parameters which used to work best on Heicub.

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.

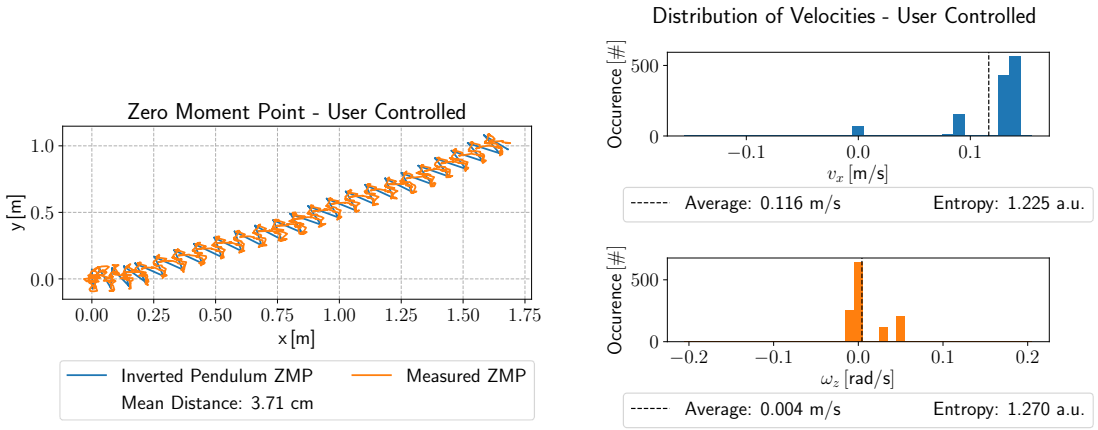
and measured the true ZMP, in accordance to equations 2.16 and 2.17, by recording the ankles' force-torque sensor readouts. We further kept track of the velocity commands to the pattern generator for all tasks to extract the user's behavior. The first task was simply to go two meters straight forward (figure 4.4). The behavior



(a) Dynamic balance. (b) Behavior.

Figure 4.4: User-controlled straight walk. The robot started to the plot's left-hand side (a), and moved forward until it reached the fire extinguisher.

therein is visualized by a histogram of all velocity commands over the course of the task (figure 4.4 (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

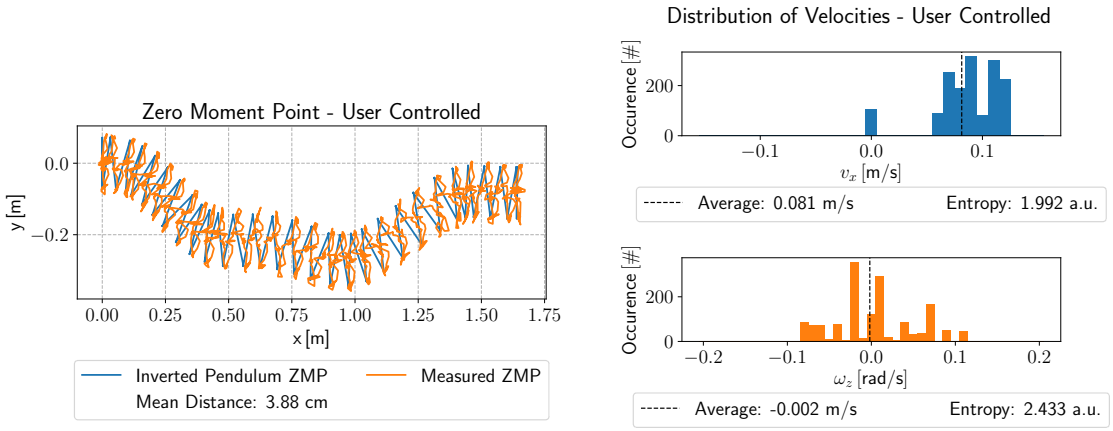


(a) Dynamic balance. (b) Behavior.

Figure 4.5: User-controlled curved walk. The robot started to the plot's left-hand side (a), and performed a left turn on its way to the fire extinguisher, where it stopped.

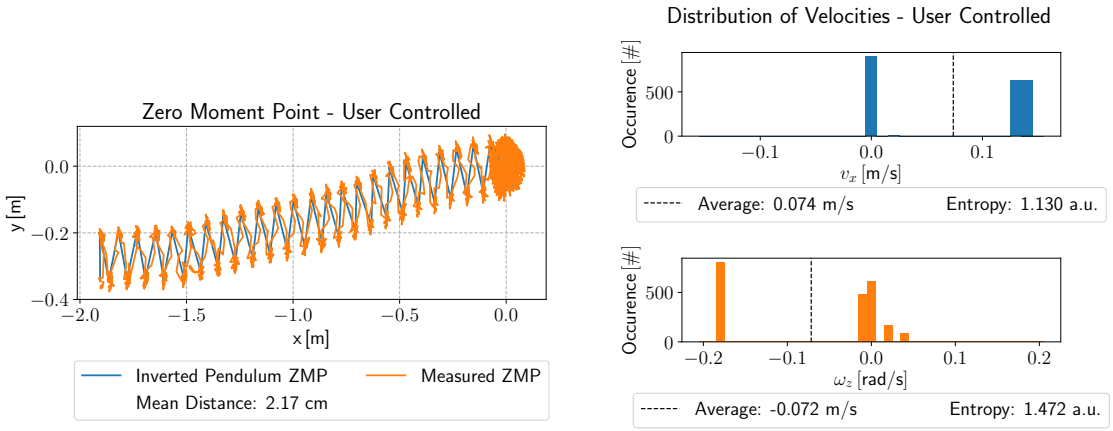
of the robot. In order to reach it, it was therefore required to perform a curved walk (figure 4.5). The third task (figure 4.6) involved the avoidance of an obstacle,

namely a chair. For the fourth task (figure 4.7), the robot started with its back pointing towards the fire extinguisher.



(a) Dynamic balance. (b) Behavior.

Figure 4.6: User-controlled obstacle avoidance. The robot started to the plot's left-hand side (a), and moved towards a fire extinguisher. On about half of the distance it avoided a chair by turning to the right, and then to the left again, until it reached the fire extinguisher and stopped.



(a) Dynamic balance. (b) Behavior.

Figure 4.7: User-controlled environmental scanning. The robot started to the plot's right-hand side (a), facing to the right, and performed a full 180° turn, before walking almost straight to the fire extinguisher, which is here located to the plot's left-hand side.

4.2 Autonomous Walking via Behavioral Cloning

The autonomous walking is based upon the performance within the test environment from section 4.1.2. The found parameters are used for comparative reason throughout this section as well. As explained in section 2.2.4, the neural network benefits strongly from an available depth map as input. We will therefore deal with the depth map extraction first.

4.2.1 Camera Calibration

As described in section 2.2.4, for the stereo block matching algorithm to work properly (equation 2.133), it is required to calibrate the cameras. We shortly verified this in figure 4.8, where we extracted a depth map from the uncalibrated stereo camera pair. For the calibration we chose to use a chess-board calibra-

(a) Left disparity map.

(b) Confidence weighted least squares filtered disparity map.

Figure 4.8: Depth map extraction without calibration. The parameters were set as follows to $N = 13$, $D = 32$, $\sigma = 1$, and $\lambda = 10^4$.

tion pattern, see figure 4.9. The used calibration pattern has width of $W = 8$, and a height of $H = 6$, where each square has a size of $a = 22.5\,\text{mm}$ (equation 2.146). 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

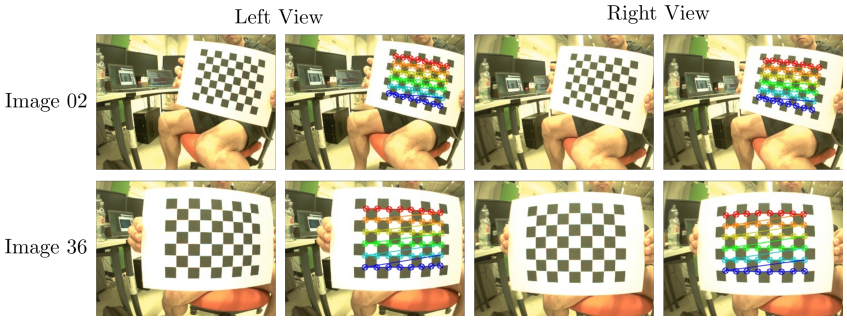
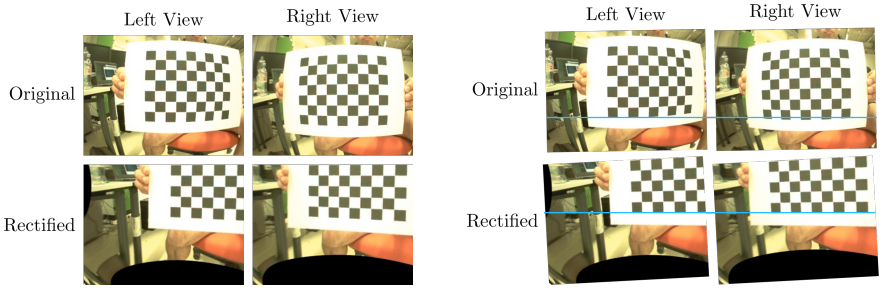


Figure 4.9: Exemplary left and right camera views of the calibration pattern as acquired during the calibration process. The colorful points indicate the detected corners in the image plane. Refer to figure 2.27 for the theory.

Table 4.2: Intrinsic parameters of single cameras. These parameters can be found as YAML file on GitHub (link).

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]	$2.16 \cdot 10^{-4}$	$7.63 \cdot 10^{-4}$

squared re-projection error $\Delta\bar{x} = 1/(WHN)\sum_0^{WHN}\Delta x$ (equation 2.148), 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 2.143, 2.144, and 2.145, we therefore determined the camera's intrinsic parameters as listed in table 4.2. Then given the calibration of each single camera, for each camera we computed the rectification transforms $\boldsymbol{R}_i$, and the projection matrices $\boldsymbol{P}_i$ in the rectified coordinate system, all of which can be found in table 4.3. Exemplary rectified images, which rely on the matrices of table 4.3, are shown in figure 4.10 (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 4.10 (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.



(a) Rectified images. (b) Rotated rectified images.

Figure 4.10: Rectified and original view of the stereo camera. Refer to figure 2.28 for the theory.

Table 4.3: Rectification transforms $\boldsymbol{R}_i$, and projection matrices $\boldsymbol{P}_i$, for the left, and the right camera, respectively. These parameters can be found as YAML file on GitHub (link).

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

4.2.2 Depth Map Parameter Tuning

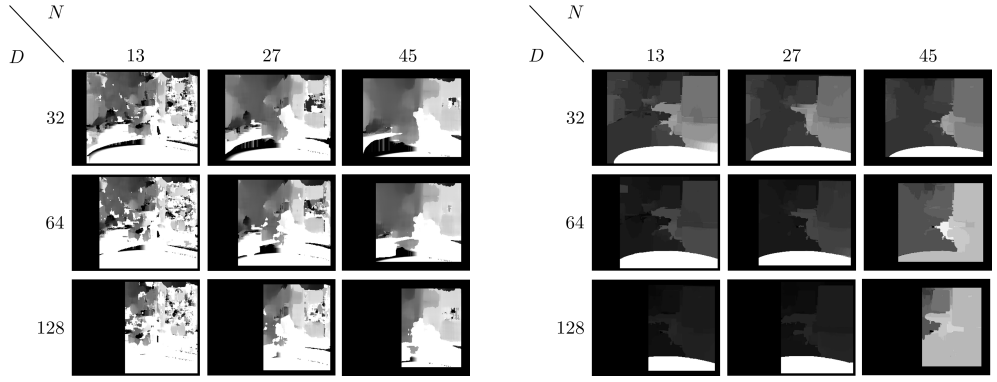
Within this section, we shortly explore the effects of all tuneable parameters on the depth map generation. Therefore, we utilize a simple experimental setup. Within the setup, Heicub points its stereo camera towards three chairs that are located at a distance of 1 m towards each other, and towards the 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 4.11. The depth map extraction, which utilizes the

(a) Left camera's view. (b) Right camera's view.

Figure 4.11: Heicub's perspective of the scene for the depth map parameter tuning.

rectified images, depends on a stereo block matching algorithm that was explained in section 2.2.4. 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 4.12 (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 for the window size, except that it removes some noise from the depth maps. In combination with the confidence weighted least squares filtering, we can



(a) Left disparity. Please refer to figure 2.22 and equation 2.133 for the theory.

(b) Confidence weighted least squares disparity map. Please refer to figure 2.24 and equation 2.142 for the theory.

Figure 4.12: Left disparity map and confidence weighted least squares disparity for changing SAD window sizes N and number of disparities D.

observe that most of the noise is already removed (figure 4.12 (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 4.13. 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. Now given these parameters, we evaluated the performance of our image processing pipeline. We ran it for 1000 times and observed an average required time of 7.4 ± 0.5 ms on an Intel Core i7-7700HQ CPU at 2.8 GHz.

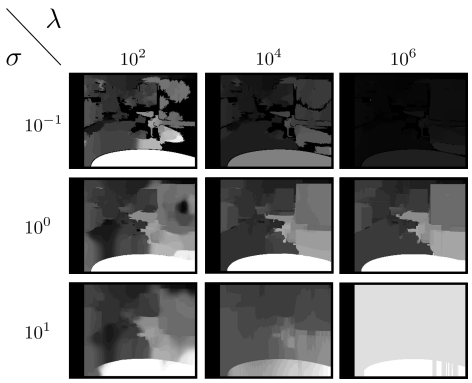


Figure 4.13: Confidence weighted least squares disparity for changing energy weightings σ and λ. Please refer to equations 2.140 and 2.141 for the theory.

4.2.3 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 4.1, 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 4.14. 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 led 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 pre-processing 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 4.11, and secondly by the depth map's extraction, which results in unknown regions at the image's borders (figure 4.12). The representative samples from the dataset (figure 4.14), are already pre-processed, 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 pre-processed 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

Figure 4.14: Samples of the recorded dataset. The dataset shows a very diverse number of situations for the neural network to learn from. It consists of a total of 2×134401 recorded images at a resolution of 240×320 pixels, which together make up for 4.7 GB, and roughly 7.5 h of data.

obtain RBGD images. For the training of the neural network, we split the acquired dataset into a training, 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 good convergence on our dataset, we chose to use a U-Net [69] as the network architecture, which can be seen in figure 4.15. The U-Net 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 several convolutional layers with rectifying linear unit activation functions, and is subsequently downscaled by max-pooling operations. We repeated this process two times. Once the most downscaled layer is reached, the weights are being upscaled by simple interpolations again, which 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

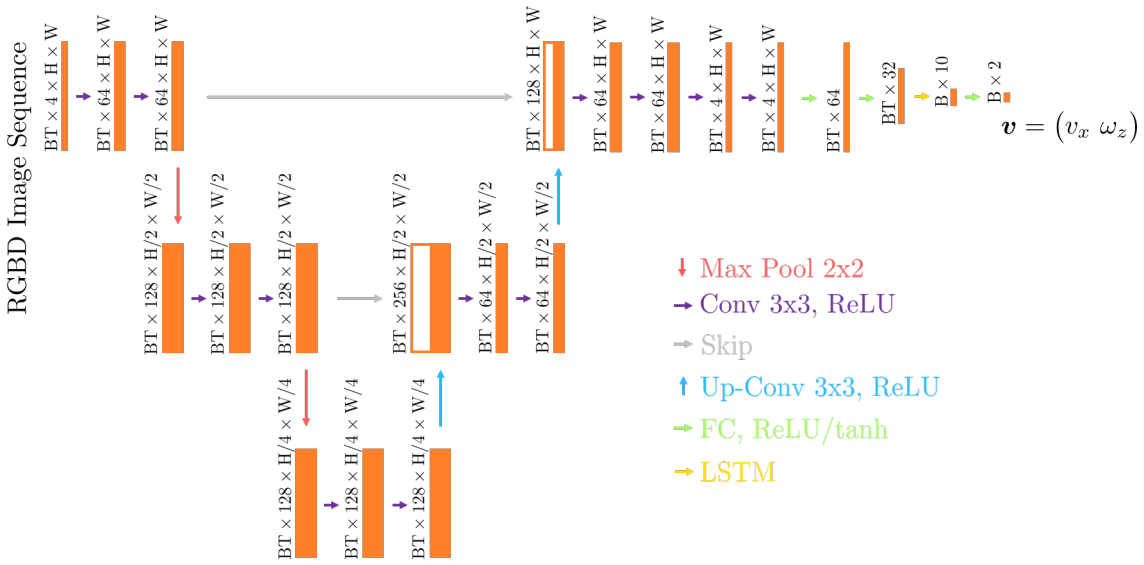


Figure 4.15: U-Net-LSTM network architecture. B stands for the batch size, and T for the number of images within a time sequence. The arrows indicate the layers that are being used, where we use ReLU activation functions except for the output layer, where we use a hyperbolic tangent and where we expect the output to represent the velocity command. The skip connections take the output of a shallow layer and concatenate it with its deep counterpart. The kernel size of each layer is consistently used, as defined within the legend.

is then being forwarded further through several convolutional layers with rectifying linear units. The nature of the robot's motion, which inherently causes the cameras to move from the left to the right periodically, required us to equip the network architecture by a temporal understanding. We, therefore, extended the U-Net architecture to a novel U-Net-LSTM structure. The developed architecture takes up a sequence of consecutive RGBD images and forwards them through the U-Net 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 only to 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 last 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 [70] at a learning rate of 0.001 and trained for 100 epochs

on an 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 4.16, which reveals a good convergence after around 40 epochs. After training, we generated velocity

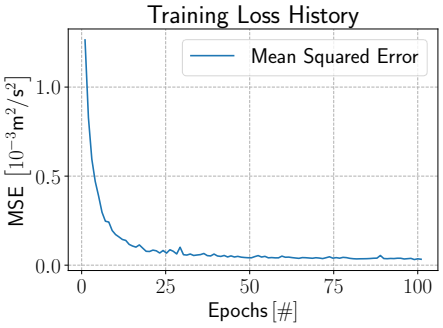
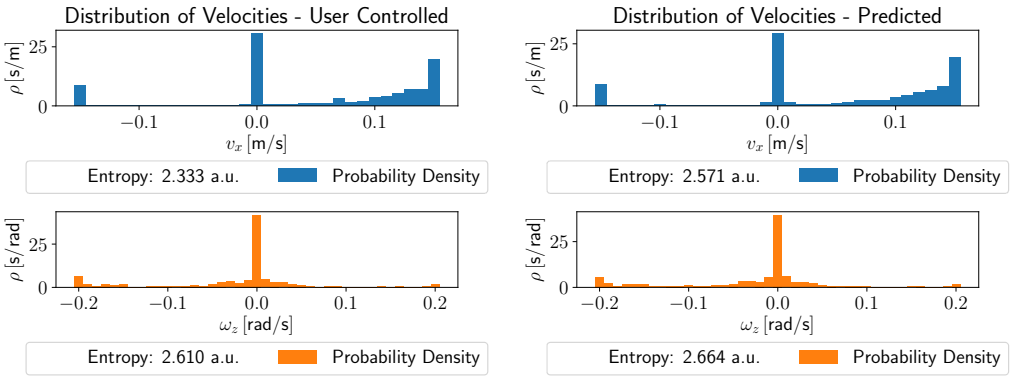


Figure 4.16: Mean squared error training loss history of the U-Net on the validation split of the dataset, shown in figure 4.15. The training took about 48 h on an Nvidia GTX 1080.

histograms, as we already did in section 4.1, but this time over the whole validation split for both, the ground truth and the predicted behavior, as shown in figure 4.17. 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



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

Figure 4.17: Normalized velocity histograms over the validation split. The ground truth (a), and the predicted velocity commands (b), appear to be very similar, which indicates a successful training.

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 two discrete

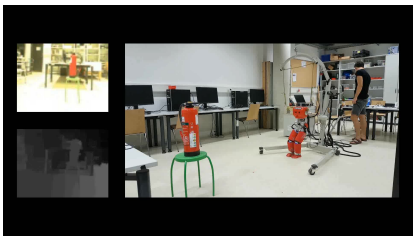
probability distributions $p(v)$ and $q(v)$, is computed as follows

$$D_{\mathrm{KL}} = \sum_{v \in \boldsymbol{V}} p(v) \log \frac{p(v)}{q(v)}, \tag{4.1}$$

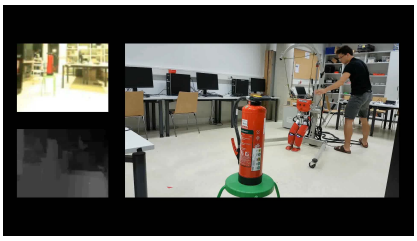
where for our case, $p(v)$ is the ground truth velocity distribution, and $q(v)$ is the predicted velocity distribution. For the distribution of linear velocities along the x-axis v_x, we computed the Kullback-Leibler divergence to be $D^x_{\mathrm{KL}} = 0.20\,\mathrm{a.u.}$, and for the angular velocity about the z-axis ω_z, we obtained $D^z_{\mathrm{KL}} = 0.01\,\mathrm{a.u.}$. 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 could run Heicub in a previously unseen test environment. It is the same test environment, which we already introduced in section 4.1, and we will evaluate the trained neural network's behavior within it in the next section - Performance in Test Environment.

4.2.4 Performance in Test Environment

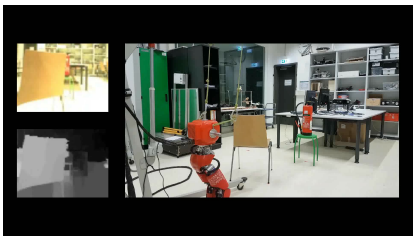
For the performance benchmarking, we relied on the well-defined experimental setup from section 4.1. Once more, the tasks were to move straight towards a fire extinguisher, to turn and to move towards it, to avoid an obstacle on the way, and to find the fire extinguisher. In order to ensure reproducibility, we let Heicub solve each of these tasks twice, as we did for the user-controlled case. The setup is shown in figure 4.18, 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, the curved



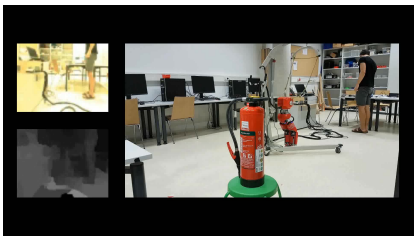
(a) Straight walk - link.



(b) Curved walk - link.



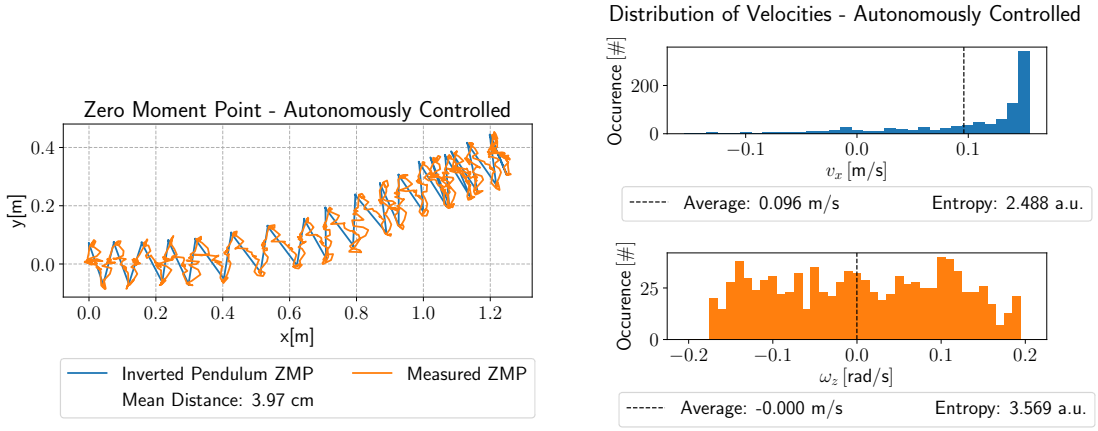
(c) Obstacle avoidance - link.



(d) Environmental scanning - link.

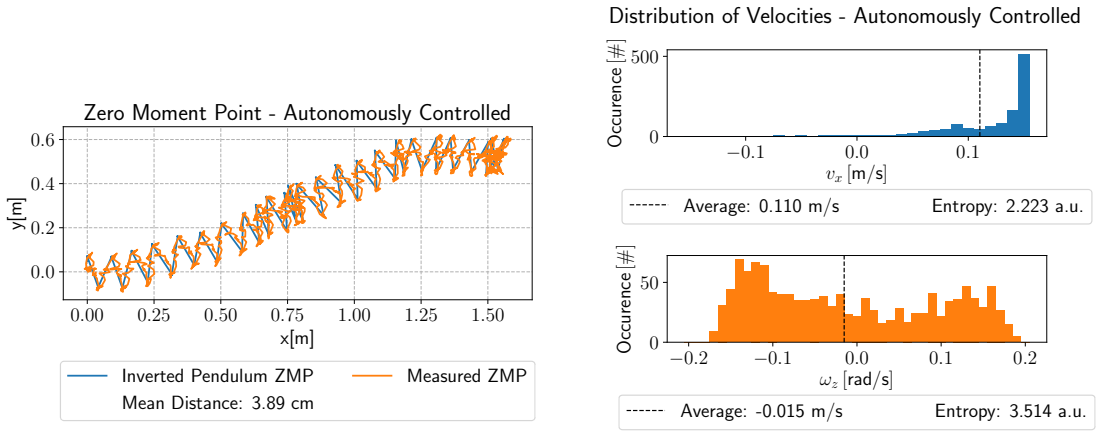
Figure 4.18: Heicub's behavior in the test environment for the benchmarking tasks. The robot within these trials was controlled by the U-Net model, shown in figure 4.15.

walk, and the environmental 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 4.19 - 4.22. 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 4.17. This especially holds true for the angular velocity distributions. In contrast to the validation split, there



(a) Dynamic balance. (b) Behavior.

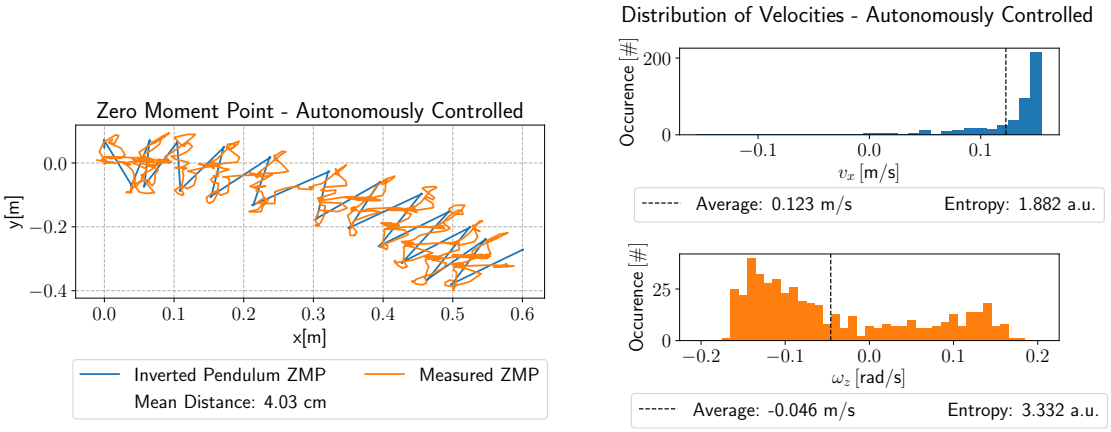
Figure 4.19: Autonomously controlled straight walk. The robot started to the plot's left-hand side (a), and then moved straight towards the fire extinguisher until it stopped in front of it.



(a) Dynamic balance. (b) Behavior.

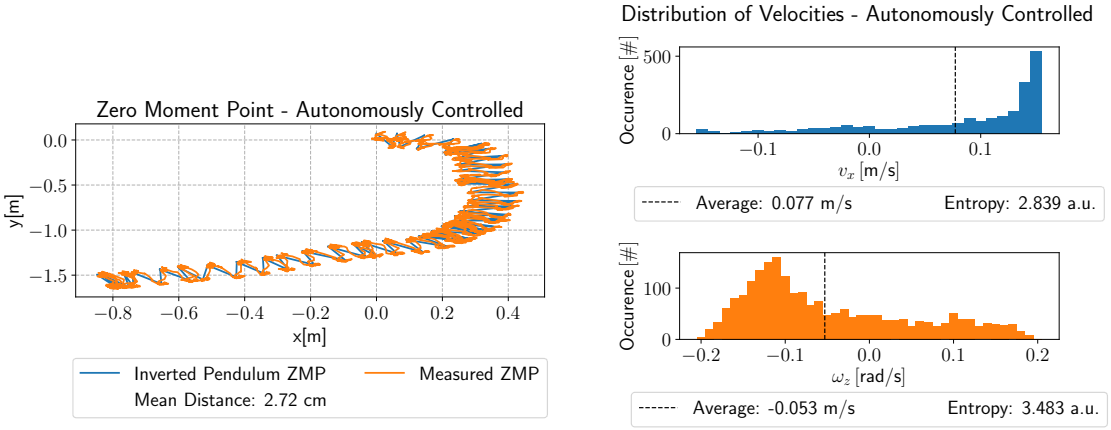
Figure 4.20: Autonomously controlled curved walk. The robot started to the plot's left-hand side (a), and moved on a curved line towards the fire extinguisher, which was located to its left.

is no simple method anymore to compare the results of user-controlled walking and



(a) Dynamic balance. (b) Behavior.

Figure 4.21: Autonomously controlled obstacle avoidance. The robot started to the plot's left-hand side (a), and avoided an obstacle by turning to the right, but then lost track of it.



(a) Dynamic balance. (b) Behavior.

Figure 4.22: Autonomously controlled environmental scanning. The robot started to the plot's right-hand side (a), facing to the right, and performed a turn to the right, until it saw the fire extinguisher and walked straight towards it.

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 taken actions result in different states. 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 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

$$S(p(v)) = \sum_{v \in \boldsymbol{V}} p(v) \log p(v) \tag{4.2}$$

Furthermore, to rate the dynamic balance, we computed the distance Δz 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. An exemplary distribution of the distances Δz is shown in figure 4.23, and it can be seen that there are mainly to contributions to the distribution, of which we figured that the distribution at small distances originates from the rotational degree of freedom, see figure 4.24. Since it can be seen in figure

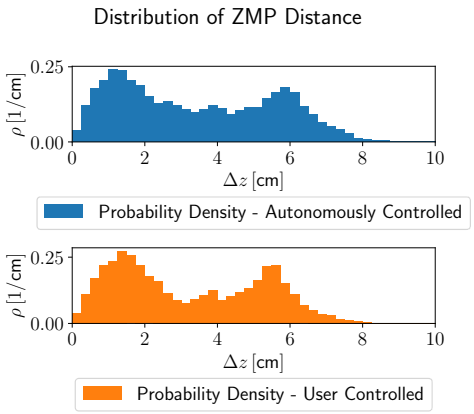
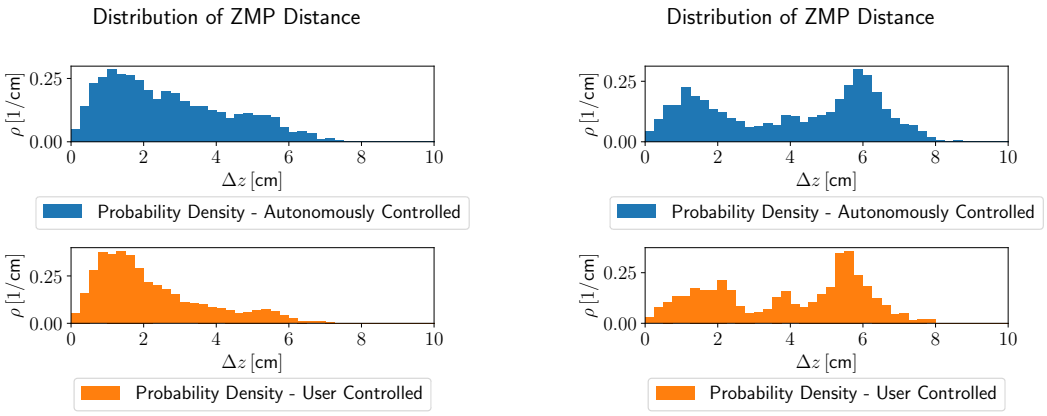


Figure 4.23: The plot shows the distribution of the distances between the measured zero moment point and the zero moment point as it originates from the nonlinear model predictive control. We can observe two main contributions within it.

4.24 that the translational velocity v_x contributes to the greater deviations from the desired zero moment point, we used the linear velocity's entropy as reference for the balance evaluation. The mean distance is therefore plotted against the entropy in figure 4.25, in order to probe the influence of noisy decisions onto the balance. The mean distance with the standard deviation for user-controlled walking therein is $3.24 \pm 1.99\,\text{cm}$, while that of autonomously controlled walking is $3.43 \pm 2.13\,\text{cm}$. Within the 1σ-range, we could, therefore, demonstrate that the control signal's entropy does not have an effect 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 demonstrates a 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 4.26, and they were solved successfully. Especially in the



(a) ZMP distance distribution for the environmental scanning.

(b) ZMP distance distribution for the straight walk.

Figure 4.24: Within the ZMP distance distributions, we can see that the contribution at lower distances mainly originates from the robot's rotation. We can say so, since (a) represents the distribution for the environmental scanning benchmarking test, for which a vast amount of rotation is required. Whereas (b) corresponds to the straight walk, and where we can see that the second contribution to the zero moment point distance is much higher.

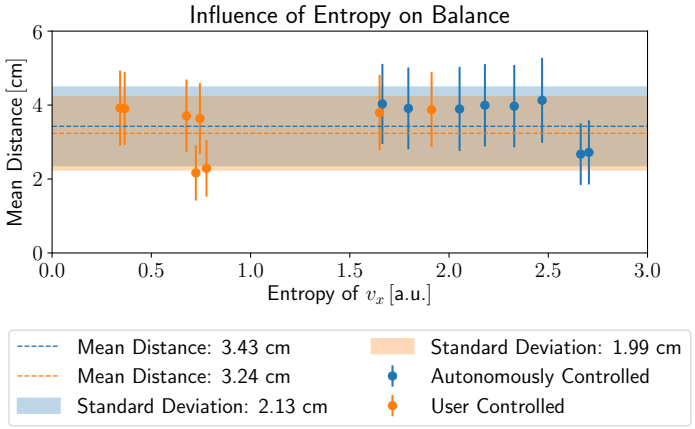
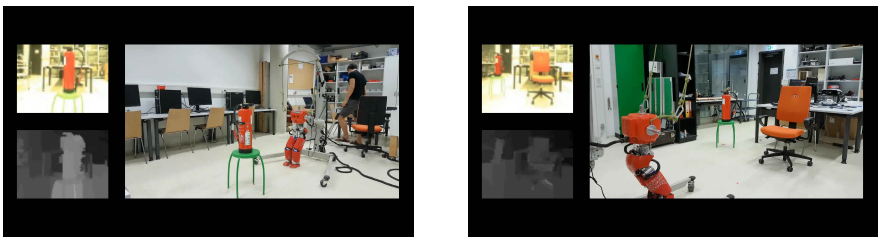


Figure 4.25: Influence of entropic commands onto Heicub's dynamic balance. Within the standard deviation there is no effect of the command signal's entropy on the robot's balance.

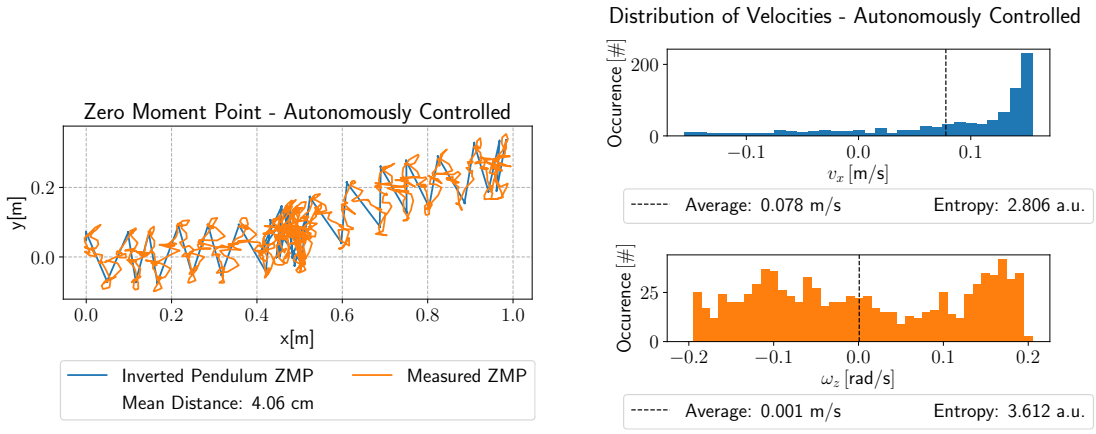
behavior plot of figure 4.28, we can see that the robot successfully managed to walk backwards to avoid a collision with the human. Given the successful results of our first approach to train a neural network on autonomous navigation, we then continued to evaluate the proximal policy optimization algorithm, which got described in



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

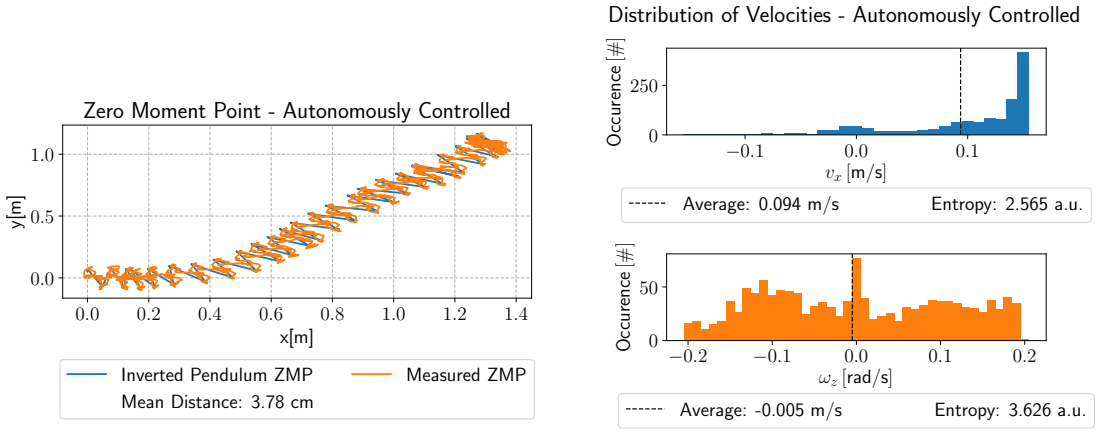
Figure 4.26: Heicub's behavior in the test environment for additional tasks. The robot within these trials was controlled by the U-Net model, shown in figure 4.15.

section 2.2.3.



(a) Dynamic balance. (b) Behavior.

Figure 4.27: Autonomously controlled in a dynamic environment. The robot started to the plot's left-hand side, and moved forward towards the fire extinguisher, where it stopped on half way to avoid a human, until the pathway was free again.



(a) Dynamic balance. (b) Behavior.

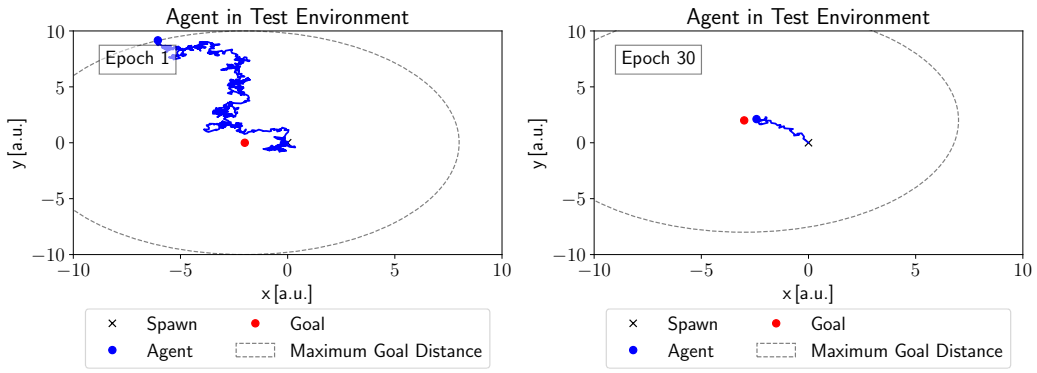
Figure 4.28: Autonomously controlled for demonstration of a semantic understanding. The robot started to the plot's left-hand side (a), and moved forward to the fire extinguisher to its left. Heicub had to distinguish between the fire extinguisher and another orange object right in front of it.

4.3 Autonomous Walking via Reinforcement Learning

Since there is to this date no feasible way of training an agent on autonomous navigation in real time with reinforcement learning, we first decided to implement a benchmarking environment, which is introduced in section 4.3.1, and then to use the benchmarking environment to test the fusion of nonlinear model predictive control with proximal policy optimization in, which is demonstrated in section 4.3.2.

4.3.1 Benchmarking Proximal Policy Optimization

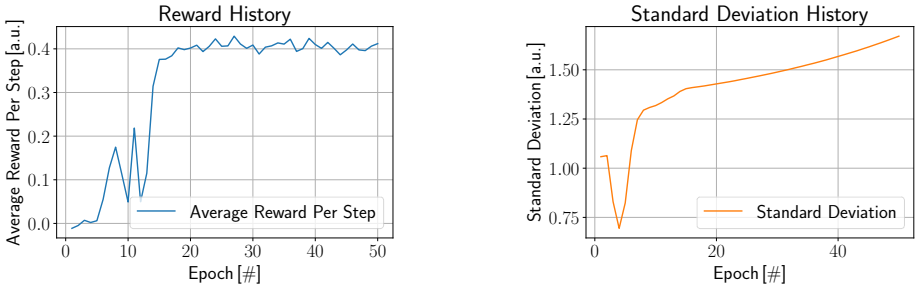
To validate the implementation of proximal policy optimization, we used a little benchmarking environment that is shown in figure 4.29. The agent's goal within this setup is to move towards the red dot, while keeping a maximum distance of 10 a.u. towards it. The environment's state is simply described by a concatenation of the agent's position $\boldsymbol{a} = \begin{pmatrix} a_x & a_y \end{pmatrix}^T$ with that of the goal $\boldsymbol{g} = \begin{pmatrix} g_x & g_y \end{pmatrix}^T$. The reward r_t, at time step t, is designed to encourage motion towards the goal, by taking the difference of the previous and the current goal distance $r_t = ||\boldsymbol{a}_{t-1} - \boldsymbol{g}_{t-1}||_2 - ||\boldsymbol{a}_t - \boldsymbol{g}_t||_2$. 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 4.29 (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



(a) Agent at epoch 1. (b) Agent at epoch 30.

Figure 4.29: Artificial agent in proximal policy optimization test environment. While the agent acts randomly in epoch 1 (a), the goal is reached with high confidence at epoch 30 (b).

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 2.129), and the cost function hyper-parameters $c_1 = 0.5$ and $c_2 = 0.1/\overline{r_t}$, where $\overline{r_t}$ denotes the average reward, and which can be found in equation 2.130. We ran the environment for 10000



(a) Reward history. (b) Standard deviation history.

Figure 4.30: Proximal policy optimization in test environment over 50 epochs. The agent learned to maximize the reward after around 20 epochs, by increasing its exploration with a higher standard deviation within the policy π_θ (b).

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 4.30). 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)

$$S[\pi_\theta] = 0.5 + 0.5\log(2\pi) + \log(\sigma), \tag{4.3}$$

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 4.30 (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 4.31, which now appears smooth. We ran the

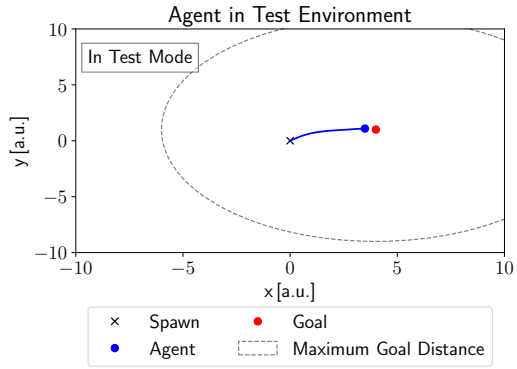


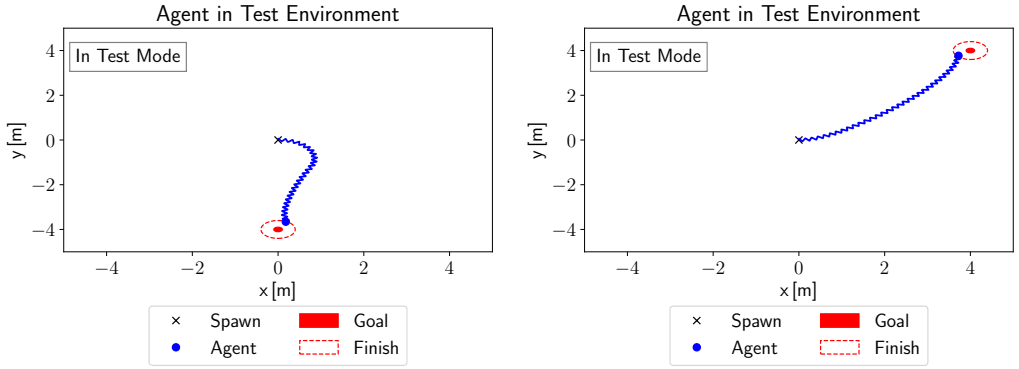
Figure 4.31: Proximal policy optimization in test mode that is without noisy policy π_θ. The agent almost learned to move the shortest path towards the goal.

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.

4.3.2 Fusion of Nonlinear Model Predictive Control with Proximal Policy Optimization

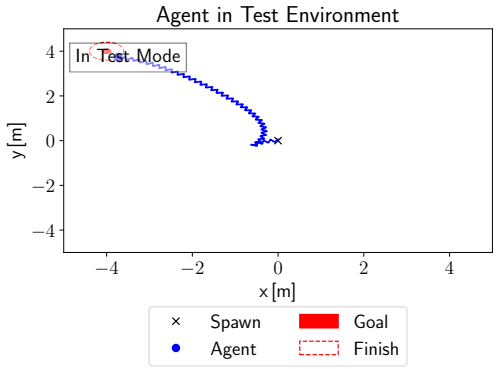
For the autonomous navigation, it is then possible to combine proximal policy optimization with nonlinear model predictive control (link). Just as in the previous section, a neural network has to solve goal navigation, but this time the agent's trajectories are computed by using the nonlinear model predictive control. The environment's state is simply described by the goal's position $\boldsymbol{g}_a = \begin{pmatrix} g_x & g_y \end{pmatrix}^T$, as seen from the agent's coordinate frame, which can be expressed by world coordinates via $\boldsymbol{g}_a = \boldsymbol{R}_z^{-1}(\boldsymbol{g}_w - \boldsymbol{a}_w)$, where $\boldsymbol{R}_z$ is the agent's current rotation, which is given by $\boldsymbol{c}_k^\theta[0]$, $\boldsymbol{g}_w$ is the goal's position in the world frame, and $\boldsymbol{a}_w$ is the agent's position in the world frame. The reward r_t for this task got designed to ensure fast frontal motion towards the goal. It therefore consists of three terms, one of

which accounts for motion towards the goal via $r_t^{\text{goal}} = ||\boldsymbol{a}_{t-1} - \boldsymbol{g}_{t-1}||_2 - ||\boldsymbol{a}_t - \boldsymbol{g}_t||_2$, whereas the second term enforces the robot coordinate system's x-axis to point towards the goal via $\boldsymbol{r}_t^{\text{frontal}} = (\boldsymbol{g}_{a,t-1} - \boldsymbol{g}_{a,t-1})[0]$, and the third term punishes slow paths via $r_t^{\text{time}} = t$. In total, we found the following weights to work best $r_t = 2 \cdot 10^3 r_t^{\text{frontal}} + 6 \cdot 10^3 r_t^{\text{goal}} - 10^{-1} r_t^{\text{time}}$. An additional reward of 100 was granted for a successfully finished task. The agent then got trained for 50 epochs, with the Adam optimizer at a learning rate of 0.003. The gradient got clipped at $\epsilon = 0.2$, the cost function hyper-parameters were set to $c_1 = 0.5$, and $c_2 = -50$, to ensure a decreasing entropy, as the pattern generator does not allow for arbitrary high commands v_x, and ω_z. We ran a single agent $N = 1$ for 2000 preview horizon time-steps, with a mini batch size of $M = 200$ for 5 proximal policy optimization epochs (see algorithm 1). For the network architecture we again chose to go with a fully connected neural network with an input layer of size 2×64, two hidden layers at a size of 64×64 each, and an output layer of size 64×2. The activation functions were set to be hyperbolic tangents. The particular model can be found at the provided link. After having fully trained the agent, it was shown that the agent solved the goal navigation task for randomly spawned goals in 100 out of 100 cases. Therefore, we can argue that the agent learned to generalize the task well, and as shown for four exemplary cases in figure 4.32, the agent also learned to move backwards and turn.

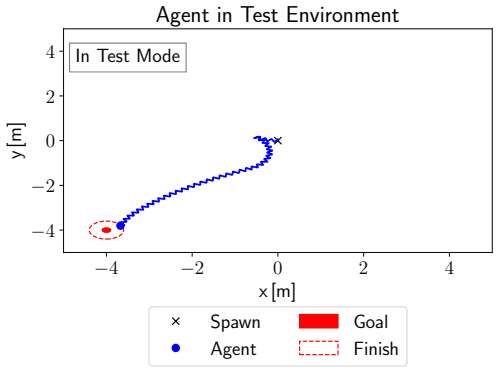


(a) Goal located at $x = 0$, $x = -4$.

(b) Goal located at $x = 4$, $x = 4$.



(c) Goal located at $x = 4$, $x = -4$.



(d) Goal located at $x = -4$, $x = -4$.

Figure 4.32: Nonlinear model predictive controlled agent in the test environment. The used fully connected neural network successfully learned to steer the NMPC agent in all cases towards the goal, by taking the goal's position as input. Just as in the behavioral cloning setup, the agent was restricted to only send the velocity commands v_x, and ω_z to the NMPC. The trajectories are similar to the ones shown in figure 4.1, but only viewed from above. The blue lines, therefore, represent the robot's center of mass trajectories and it initially faces towards the positive x-direction in all of the four cases. Note how for the cases (c), and (d), the agent learned to walk backwards, and to turn.

5 Conclusion

5.1 Contributions

Within the scope of this thesis, two main sub-objectives were achieved. The first was to implement a nonlinear model predictive control, based on the works of [71][23], while the second was to utilize neural networks to achieve a high-level control of it. Therefore, neural networks were trained by two different approaches. The first approach relied on behavioral cloning, which got mainly inspired by [45], where it was used for autonomous navigation of self-driving cars, while the second approach used proximal policy optimization [48], a form of reinforcement learning. Within this work, to the best knowledge of the authors, it was the first time that iCub navigated an environment while being controlled by a neural network. It was also the first time that Heicub's cameras where actually being used, and an exhaustive guide on how this was done is included in the appendix.

This work began as a spin-off from previous work, which was carried out at our group [71], and where two main future steps were proposed. The first was to re-implement the nonlinear model predictive controller together with the inverse kinematics into an enclosed library, in order to compensate for delays that were induced by the YARP network, which had to manage these two building blocks separately. This aim was successfully met, as the presented implementation ran within a single thread of the YARP network (see figure 3.3). Furthermore, a speed-up of 600 % (section 4.1.1) was achieved, when compared to the previous Python implementation of the nonlinear model predictive control from within our group, which served as a template. The second proposal of the previous work was the use of additional constraints for the inverse kinematics, as hip dislocations were observed for long trajectories. Within this work, it was then found that rotational constraints to the hip solved the hip's dislocation, which for the first time allowed Heicub to move for arbitrarily long times (section 4.1.2). The achievement of the two proposed future steps then enabled the implementation of a real-time control, for which a terminal user interface got created (section B.3.1). Moreover, the same real-time control was synchronized with an existing android joystick app (section B.3.2), which allowed us to control Heicub via a smartphone for the first time. Following that, a literature review revealed that no efforts were performed in utilizing neural networks for autonomous navigation of humanoid robots. To compensate for this shortcoming, it was investigated whether behavioral cloning and reinforcement learning were possible candidates to train neural networks on autonomous navigation, which are described in the following two sections.

The successfully implemented pattern generation library led to the aim of completely replacing the human user from the control loop of figure 2.1. Since no effort was being made in utilizing neural networks as substitutes, this work focused on replacing the human user by neural networks and, therefore, behavioral cloning was depicted as a possible candidate to train a neural network on autonomous navigation. Preliminary aims had to be achieved, so to utilize depth information as input to the neural network, and since Heicub does not support RGBD cameras, but only stereo RGB cameras (section 3.2), depth maps had to be extracted via existing algorithms from OpenCV [65]. As such, Heicub's cameras got calibrated (figure 4.10) to allow for image rectification, which is a requirement for depth map extraction from stereo cameras. The obtained camera's intrinsic and extrinsic parameters are listed in table 4.3. It was then proven that simple block matching algorithms were not sufficient to allow for a depth map extraction (figure 4.12 (a)), but instead it was shown how confidence weighted least squares disparity maps solved the issue (figure 4.13). For the behavioral cloning, we decided to try and train Heicub on finding a fire extinguisher within a room. The dataset, which was acquired for this task, comprises a total of 2×134401 images, which make up for 4.7 GB (figure 4.14). A newly developed U-Net-Fully-Connected-LSTM network demonstrated the best results on the acquired images (figure 4.15). For comparative reasons, we decided to let Heicub perform benchmarking tests, which were designed to let a human user, as well as the trained neural network, find a fire extinguisher in a previously unseen environment. As benchmarking tasks, we considered straight walking (figures 4.4 and 4.19), curved walking (figures 4.5 and 4.20), obstacle avoidance (figures 4.6 and 4.21), as well as searching for the fire extinguisher (figures 4.7 and 4.22). It could be demonstrated that the uncertainty within the neural network's decisions did not influence the robot's balance (figure 4.25), for which we evaluated the command signal's entropy against the zero moment point deviation from the ideal case of a linear inverted pendulum. Additional tests further showed how the U-Net-Fully-Connected-LSTM was able to interact safely with humans in a dynamic environment and how it learned to understand semantic features (figure 4.27). The second approach to train a neural network then was to utilize reinforcement learning, which is presented in the following section.

As it has recently shown good results for reinforcement learning tasks in general, proximal policy optimization [48] was chosen to train a neural network on autonomous navigation. The well-known algorithm got, therefore, implemented in C++ with PyTorch [66], which is, to the best knowledge of the authors, the first implementation of its kind. The algorithm was first shown to converge for simple navigation tasks in a simulated test environment (figure 4.31), and in contrast to the behavioral cloning approach, it does not rely on images as input, but instead on positional information. Due to temporal constraints, it was further induced that proximal policy optimization was not directly applicable to the real robot. It was therefore shown how proximal policy optimization could be combined with the

nonlinear model predictive control to achieve autonomous navigation for humanoid robots in the same test environment (figure 4.32), which strongly benefited from the speed-up of the presented nonlinear model predictive control implementation (section 4.1.1). The fact that the algorithm had to be trained in simulation, rather than on the real robot, directly leads to some of the implications within this work that are presented in the following section.

5.2 Implications and Limitations

The implications of this work are very versatile, so are the limitations, and they can mainly be drawn with respect to the pattern generation library, as well as the two approaches for utilizing neural networks on autonomous navigation.

The implemented pattern generation library was shown to be fast enough to be used in a reinforcement learning setup. It, therefore, enables the future reader to explore further, and combine reinforcement learning approaches with optimal control. Limitations to the pattern generation library are the theoretically originating limits of the used sequential quadratic programming method, which is built upon the assumption that the current robot's state $\boldsymbol{x}_k$ is close to a solution (section 2.1.2). Divergence of the solution may, therefore, be introduced by the center of mass feedback, which deviates from the linear inverted pendulum. An additional limitation of the implemented nonlinear model predictive control is that it is based on an analytic computation of the zero moment point, which is based on the assumption of a linear inverted pendulum, which the robot clearly does not represent.

It was shown that neural networks could be trained via behavioral cloning to solve navigation tasks. The presented behavioral cloning is general enough, such that it could easily be applied to tasks like corridor navigation, which were previously solved by complex models in [72]. Other multi-behavior state of the art approaches, as presented in the introductory section 1, which are based on fuzzy logic, and that are used on top of existing navigation strategies, may completely be replaced by the presented method, since it combines multiple behaviors and the autonomous navigation into one solution. The great advantage of behavioral cloning over simultaneous localization and mapping approaches is that it works for dynamic environments. Behavioral cloning is further not dependent on a reference coordinate system, and is therefore also not prone to slipping, which is a problem in humanoid robotics applications. The presented method is fast, and it was shown in section 4.2.3 that it takes 4 ms to perform inference with the used neural network. Given that it was run on a GPU, and that tensor processing units are currently under development, which already outperform GPUs and CPUs by $15 - 30\times$ in terms of energy efficiency and speed [73], it becomes clear how similar methods will become increasingly important for energy-critical applications like humanoid robots. Although we could solve the task of finding the fire-extinguisher for three of the four benchmarking cases,

the robot did not manage to avoid the obstacle, and go to the fire extinguisher at the same time. It only managed to avoid the obstacle, and then lost sight of the fire extinguisher. Other limitations are that a broader spacial understanding of the environment is yet missing. The designed behavioral cloning network architecture, furthermore, had a limited temporal understanding, since it only used the five most recent images as input. An additional issue, just as for fuzzy logic based behaviors, is that the neural network performs decisions as a result of incomplete measures of the environment, which may in principle lead to undefined actions. This is, however, a limitation to all existing systems, but in contrast to them, behavioral cloning may also perform undefined decisions on encountering unseen environmental states, which can be compensated for by ensuring a versatile dataset to train from.

Concerning the reinforcement learning part of this thesis, the implemented cutting-edge proximal policy optimization was the first of its kind that got implemented in C++. It was successfully shown how reinforcement learning could be used on top of optimal control to achieve autonomous navigation, which directly considers the system's physics. The implemented algorithm is, moreover, general enough to learn other tasks than autonomous navigation. Limitations are, however, that the presented simulation environment did not support obstacles yet, as it relied on the built-in obstacle avoidance of the nonlinear model predictive control. In contrast to the behavioral cloning approach, the reinforcement learning approach, as it got trained in simulation, does not work on images, but on the current location of the goal. It, therefore, requires an additional environment mapping algorithm to work properly, which directly leads to future work that is presented in the following section.

5.3 Future Work

This work opens many possibilities for future work. The implemented nonlinear model predictive control could be extended by replacing the linear inverted pendulum computation of the zero moment point by a neural network-based computation, for which data had to be acquired by running the implemented pattern generation, and by storing joint angles and their related zero moment points, from which a neural network could learn to map joint angles to the measured zero moment point.

The framework, which got developed in the course of this thesis, can be used to not only train humanoid robots on navigation, but on general behaviors, such as grasping objects. New network architectures can be found very quickly by prototyping with the presented fusion of Python with C++. The future reader should further pay attention to re-investigate on the used neural network's cost function, since the behavior may better be represented by a Kullback-Leibler divergence than by the used mean squared error. However, it is not straight forward to replace one error function by the other, and more research must be done. A network architec-

ture should be developed that allows for a better spacial understanding of the scene. Additionally, a better temporal understanding could be achieved by introducing a longer sequence of past images to the developed U-Net-Fully-Connected-LSTM, or by utilizing attention-based neural networks instead of LSTMs, should memory be an issue.

While the proximal policy optimization turned out to be not suitable for reinforcement learning on the real robot, it showed to be applicable to a simulation environment, from where it can be easily carried to the real robot. However, further steps need to be taken, and a simultaneous localization and mapping algorithm must be utilized, as the neural network, which got trained via reinforcement learning, relies on spatial information. The simultaneous localization and mapping algorithm will, however, directly benefit from the camera calibration, which got carried out within the scope of this thesis. The implemented proximal policy optimization algorithm is, moreover, general enough to be applied to other tasks. In combination with the very fast nonlinear model predictive control it can, for example, be used in order to investigate learning optimal control solutions in a reinforced setting.

Part I

Appendix

A Software Installation

Since all 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 is provided within each following section. For the case of an error that may occur during the build procedure, do not hesitate to open 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`

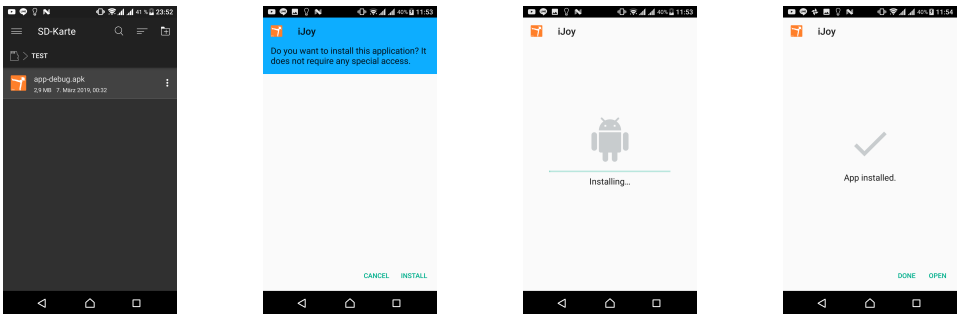
The pattern generator comes with some tests and examples. They can be executed as follows

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)).



(a) File browser. (b) Menu. (c) Installing. (d) Finished.

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 panel, 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 check out this gist to figure out how to perform a clean setup.

B Start-up 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 real-time with the terminal interface or the Android joystick 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 Start-up

1. Turn on the icubsrv (figure B.1), username is `icub`, ask someone at the ORB for the password.

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.

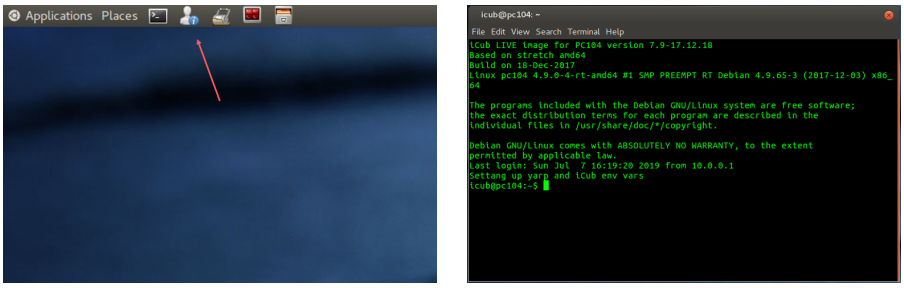
Figure B.2: Power suppliers.

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.



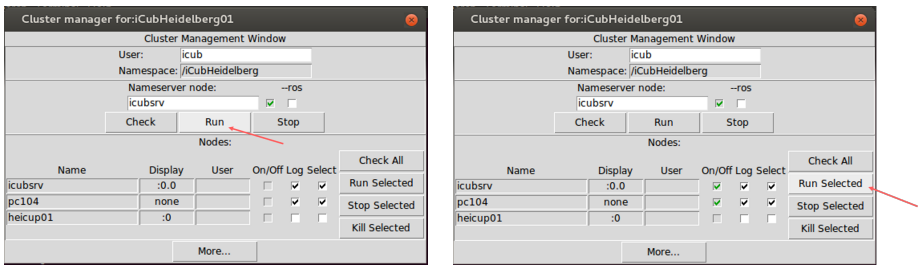
(a) Run ssh to connect to pc104. (b) Terminal to pc104.

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.



(a) Run the YARP nameserver. (b) Run YARP on other devices.

Figure B.5: Run YARP.

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.

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).

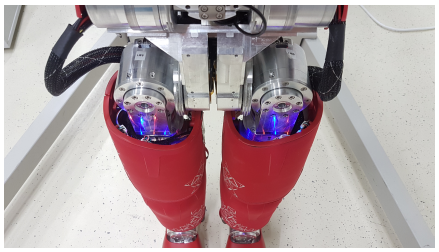
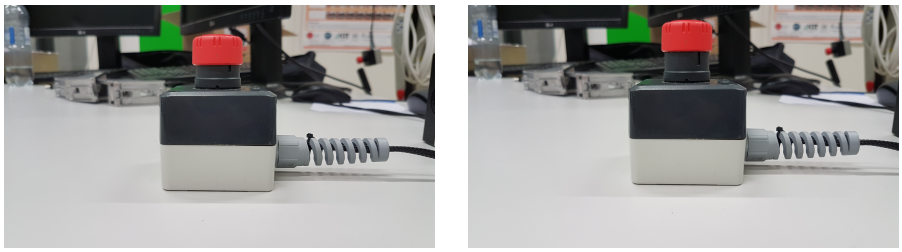


Figure B.6: Blue motor lights.

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

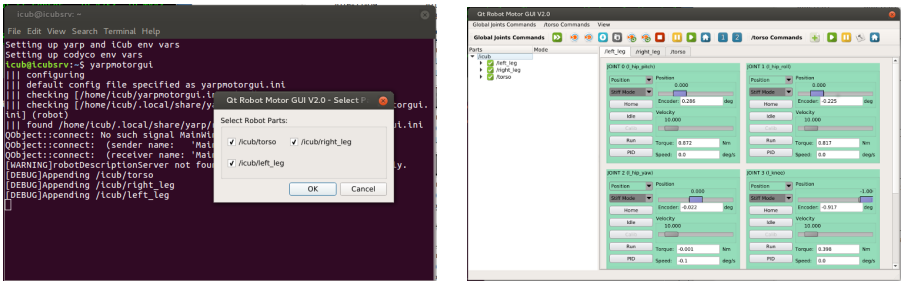


(a) Pressed safety button. (b) Released safety button.

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 occured, 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.

5. 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 yarpmotorgui. 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`.

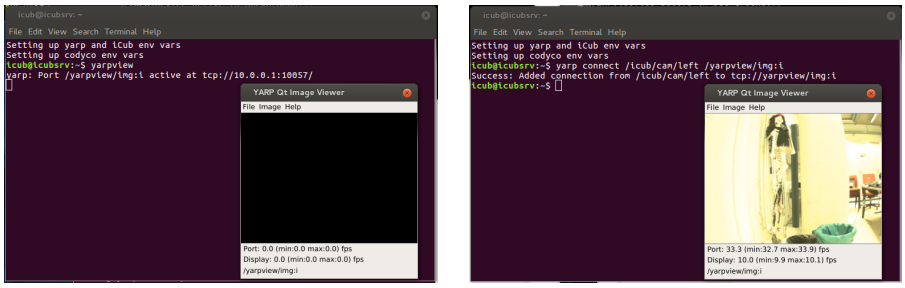
B.1.2 Start the Cameras

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 occured, 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 Start-up

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 start-up 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)).

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

(b) Keyboard user interface.

Figure B.10: Run the keyboard user interface.

4. The user interface should now show up and explain how to control the robot (figure B.10 (b)).

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.

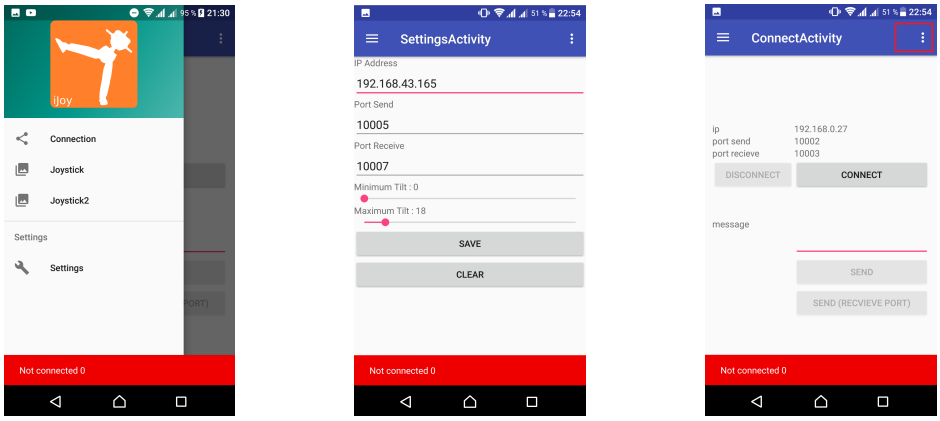
1. Open a hotspot from your phone and connect the icubsrv or your laptop to it.
2. Open a new terminal and go to the shell scripts within the pattern generator folder via
   `cd nmpc_pattern_generator/sh`
   On the icubsrv, this folder is located at `/usr/local/src/robot`. Then, run the app user interface 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)).
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.



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

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

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 4.2. 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`, ask someone at the ORB for the password.

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 shut down 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 yarprobotinterface 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

C.2 List of Tables

D Bibliography

[1] Meet Digit: A Smart Little Robot That Could Change the Way Self-Driving Cars Make Deliveries, 2019.

[2] Olivier Stasse, F Morsillo, Mathieu Geisert, Nicolas Mansard, Maximilien Naveau, and Christian Vassallo. Airbus/future of aircraft factory HRP-2 as universal worker proof of concept. *2014 IEEE-RAS International Conference on Humanoid Robots*, pages 1014–1015, 2014.

[3] D Kinly III. Chernobyl's legacy: Health, environmental and socio-economic impacts and recommendations to the Governments of Belarus, the Russian Federation and Ukraine. The Chernobyl Forum 2003-2005. Second revised version. 2006.

[4] James Kuffner, Koichi Nishiwaki, Satoshi Kagami, Masayuki Inaba, and Hirochika Inoue. Motion planning for humanoid robots. In *Robotics Research. The Eleventh International Symposium*, pages 365–374. Springer, 2005.

[5] Olivier Stasse, Francois Saidi, Kazuhito Yokoi, Bjoern Verrelst, Bram Vanderborght, Andrew Davison, Nicolas Mansard, and Claudia Esteves. Integrating walking and vision to increase humanoid autonomy. *International Journal of Humanoid Robotics*, 5(2):287–310, 2008.

[6] Claire Dune, Andrei Herdt, Olivier Stasse, P-B Wieber, Kazuhito Yokoi, and Eiichi Yoshida. Cancelling the sway motion of dynamic walking in visual servoing. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3175–3180, 2010.

[7] Claire Dune, Andrei Herdt, Eric Marchand, Olivier Stasse, Pierre-Brice Wieber, and Eiichi Yoshida. Vision based control for humanoid robots. In *IROS Workshop on Visual Control of Mobile Robots (ViCoMoR)*, pages 19–26, 2011.

[8] Robert J Griffin, Georg Wiedebach, Stephen McCrory, Sylvain Bertrand, Inho Lee, and Jerry Pratt. Footstep Planning for Autonomous Walking Over Rough Terrain. *arXiv preprint arXiv:1907.08673*, 2019.

[9] Elmer P Dadios, Jazper Jan C Biliran, Ron-Ron G Garcia, D Johnson, and Adranne Rachel B Valencia. Humanoid Robot: Design and Fuzzy Logic Control Technique for Its Intelligent Behaviors. In *Fuzzy Logic-Controls, Concepts, Theories and Applications*. IntechOpen, 2012.

[10] Miomir Vukobratovic and Davor Juricic. Contribution to the synthesis of biped gait. *IFAC Proceedings Volumes*, 2:469–478, 1968.

[11] Miomir Vukobratovic and Davor Juricic. Contribution to the synthesis of biped gait. *IEEE Transactions on Biomedical Engineering*, pages 1–6, 1969.

[12] 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.

[13] Miomir Vukobratovi and Branislav Borovac. Zero-moment point—thirty five years of its life. *International journal of humanoid robotics*, 1(01):157–173, 2004.

[14] 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.

[15] 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.

[16] 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.

[17] Shuuji Kajita, Hirohisa Hirukawa, Kensuke Harada, and Kazuhito Yokoi. *Introduction to humanoid robotics*. Springer, 2014.

[18] Qiang Huang, Kazuhito Yokoi, Shuuji Kajita, Kenji Kaneko, Hirohiko Arai, Noriho Koyachi, and Kazuo Tanie. Stability compensation of a mo-bile manipulator by manipulator motion: Feasibility and planning. *IEEE Transactions on robotics and automation*, 17(3):280–289, 2001.

[19] Raphael Michel. *Dynamic filter for walking motion corrections*. Bachelor's thesis, Heidelberg University, 2017.

[20] Paul T. Boggs and Jon W. Tolle. Sequential quadratic programming. *Acta numerica*, 4:1–51, 1995.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] Jorge J More. The Levenberg-Marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116, 1978.

[26] Tomomichi Sugihara. Solvability-unconcerned inverse kinematics by the Levenberg–Marquardt method. *IEEE Transactions on Robotics*, 27(5):984–911, 2011.

[27] Mikael Haggstrom and Others. Medical gallery of Mikael Haggstrom 2014. *WikiJournal of Medicine*, 1(2), 2014.

[28] Kyoung-Su Oh and Keechul Jung. GPU implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

[29] 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.

[30] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[31] 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.

[32] 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.

[33] 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.

[34] SS Viglione. 4 Applications of pattern recognition technology. In *Mathematics in Science and Engineering*, pages 115–162. 1970.

[35] A G. Ivakhnenko. Polynomial theory of complex systems. *IEEE transactions on Systems, Man, and Cybernetics*, (4):364–378, 1971.

[36] 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.

[37] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[38] 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.

[39] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[41] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[42] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with LSTM. *IET*, 1999.

[43] 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.

[44] David J Montana and Lawrence Davis. Training Feedforward Neural Networks Using Genetic Algorithms. In *IJCAI*, pages 762–767, 1989.

[45] 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.

[46] 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.

[47] 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.

[48] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[49] 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.

[50] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*. 1998.

[51] 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.

[52] 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.

[53] Irwin Sobel. An Isotropic 3x3 Image Gradient Operator. *Presentation at Stanford A.I. Project 1968*, 2014.

[54] 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.

[55] Carlo Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. In *Iccv*, volume 98, page 2, 1998.

[56] C Brown Duane. Close-range camera calibration. *Photogramm. Eng*, 37(8):855–866, 1971.

[57] 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.

[58] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Transactions on pattern analysis and machine intelligence*, 22, 2000.

[59] 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.

[60] Martin Felis. MeshUp, 2012.

[61] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml™) version 1.1. *yaml.org, Tech. Rep*, page 23, 2005.

[62] Gael Guennebaud and Jacob Benoit. Eigen v3, 2010.

[63] Martin L Felis. RBDL: an efficient rigid-body dynamics library using recursive algorithms. *Autonomous Robots*, 41(2):495–511, 2017.

[64] 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.

[65] G. Bradski. The OpenCV Library, 2000.

[66] 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.

[67] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. YARP: yet another robot platform. *International Journal of Advanced Robotic Systems*, 3(1):8, 2006.

[68] 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.

[69] 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.

[70] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[71] Kevin Stein, Yue Hu, Manuel Kudruss, Maximilien Naveau, and Katja Mombaur. Closed loop control of walking motions with adaptive choice of directions for the iCub humanoid robot. In *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, pages 184–190. IEEE, 2017.

[72] Angela Faragasso, Giuseppe Oriolo, Antonio Paolillo, and Marilena Vendittelli. Vision-based corridor navigation for humanoid robots. In *2013 IEEE International Conference on Robotics and Automation*, pages 3190–3195. IEEE, 2013.

[73] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre Luc Cantin, Clifford Chao, Chris Clark,

Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, Part F1286:1–12, 2017.

E Acknowledgements

First of all, I want to thank Katja Mombaur for letting me work at her group Optimization, Robotics, and Biomechanics. Being a part of her group has been very insightful and enjoyable throughout the time that I spent there. It was an honor for me to be given the chance and work on humanoid robots, to which not many have access to.

Secondly, I want to thank Kevin Stein, Yue Hu, and Manuel Kudruss for always lending me a hand, and for supporting the creation of this thesis. I want to thank further Matthew Millard, Giorgos Marinou, Jonas Große Sundrup, Julian Martus, Silvan Lindner, Alex Schubert, and Benjamin Reh for their huge amount of advice that they gave me on several topics, including but not limited to bug-fixing, structuring of this thesis and on general questions.

Additionally, I want to thank Maximilian Staiger for his implementation of the android joystick app, and his very cooperative way of working, when we linked our implementations.

Moreover, I want to thank Patrick Leibersperger, Christoph Rieß, and Lucas Möller for helping me to perform the data acquisition with Heicub. Especially, I want to thank Lucas Möller for providing me with access to powerful GPU clusters, which enabled me to train the neural networks in a reasonable amount of time. Once more, I want to thank Christoph Rieß for proofreading this thesis. Although it is not directly related to this thesis, I want to thank Markus Miltner for letting me stay at his place for the last couple of months, without whom I may not have been able to finish the work.

Finally, I want to thank my parents for their strength and their unconditional support that allowed me to study physics, and to follow my personal aims.

Erklärung:

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

Heidelberg, den (Datum) .....