Learning Motion Skills for a Humanoid Robot

Dissertation
zur Erlangung des akademischen Grades
Dr. rer. nat
an der Fakultät für Mathematik, Informatik und
Naturwissenschaften
der Universität Hamburg

Eingereicht beim Fachbereich Informatik
von Marc Bestmann

03.2023

Gutachter:
Prof. Dr. Jianwei Zhang
Prof. Dr. Janick Edinger

Tag der Disputation: 06.09.2023

Abstract

This thesis investigates the learning of motion skills for humanoid robots. As groundwork, a humanoid robot with integrated fall management was developed as an experimental platform. Then, two different approaches for creating motion skills were investigated. First, one that is based on Cartesian quintic splines with optimized parameters. Second, a reinforcement learning-based approach that utilizes the first approach as a reference motion to guide the learning. Both approaches were tested on the developed robot and on further simulated robots to show their generalization. A special focus was set on the locomotion skill, but a standing-up and kick skill are also discussed.

Zusammenfassung

Diese Dissertation beschäftigt sich mit dem Lernen von Bewegungsfähigkeiten für humanoide Roboter. Als Grundlage wurde zunächst ein humanoider Roboter mit integriertem Fall Management entwickelt, welcher als Experimentalplatform dient. Dann wurden zwei verschiedene Ansätze für die Erstellung von Bewegungsfähigkeiten untersucht. Zu erst einer der kartesische quintische Splines mit optimierten Parametern nutzt. Danach wurde ein Ansatz basierend auf bestärkendem Lernen untersucht, welcher den ersten Ansatz als Referenzbewegung benutzt. Beide Ansätze wurden sowohl auf der entwickelten Roboterplatform, als auch auf weiteren simulierten Robotern getestet um die Generalisierbarkeit zu zeigen. Ein besonderer Fokus wurde auf die Fähigkeit des Gehens gelegt, aber auch Aufsteh- und Schussfähigkeiten werden diskutiert.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Aims of this Thesis . . . 2
1.3 Research Questions . . . 3
1.4 Contributions . . . 3
1.5 Publications . . . 5
1.5.1 Core Academic Publications . . . 5
1.5.2 Further Academic Publications . . . 6
1.5.3 Team Description Papers . . . 7
1.6 Thesis Structure . . . 8

2 Basics **10**
2.1 Humanoid Robots . . . 10
2.2 Servo Motors . . . 11
2.2.1 Dynamixel Servos . . . 11
2.2.2 RS-485 and TTL . . . 13
2.2.3 Dynamixel Protocol . . . 14
2.3 ROS . . . 15
2.4 Splines . . . 16
2.5 RoboCup Humanoid League . . . 20

3 Robot Platform **23**
3.1 Related Work . . . 23
3.1.1 Humanoid Robot Platforms . . . 23
3.1.2 Robustness . . . 28
3.1.3 Spring Based Actuators . . . 28
3.1.4 Low-cost Foot Pressure Sensors . . . 29
3.1.5 Servo and Sensor Interfaces . . . 30
3.2 Overview . . . 32
3.3 Mechanics . . . 35
3.3.1 Torsion Springs . . . 35
3.4 Electronics . . . 38
3.4.1 Sensors . . . 38
3.4.2 Dynamixel Bus Communication . . . 43
3.5 Modeling . . . 47
3.6 Evaluation . . . 50
3.6.1 Servo and Sensor Interface . . . 50
3.6.2 Torque Reduction . . . 55
3.6.3 Computational Performance . . . 57
3.6.4 Qualitative Evaluation . . . 60
3.7 Summary and Future Work . . . 61

4 Fall Management 62
4.1 Related Work . . . 62
4.1.1 Decision Making . . . 62
4.1.2 Robot Software Architecture . . . 65
4.1.3 Fall Handling . . . 66
4.2 Dynamic Stack Decider . . . 67
4.2.1 Elements . . . 67
4.2.2 Domain Specific Language . . . 68
4.2.3 Call Stack . . . 69
4.2.4 Implementation . . . 69
4.3 HCM . . . 71
4.3.1 Requirements for Abstraction . . . 72
4.3.2 Architecture . . . 73
4.3.3 Implementation . . . 74
4.3.4 Fall Detection . . . 78
4.4 Evaluation . . . 79
4.4.1 HCM . . . 80
4.4.2 Latency . . . 80
4.4.3 Fall Detection . . . 85
4.5 Summary and Future Work . . . 90

5 Quintic Spline based Motion Skills 91
5.1 Related Work . . . 92
5.1.1 Pattern Generators . . . 93
5.1.2 Stabilization . . . 95
5.1.3 Learning Approaches . . . 95
5.1.4 Parameter Optimization . . . 95
5.2 Approach . . . 96
5.2.1 Spline Creation . . . 96
5.2.2 Spline Interpolation . . . 98
5.2.3 Parameter Optimization . . . 98
5.2.4 Implementation . . . 99
5.3 Walk . . . 101
5.3.1 Finite State Machine . . . 102
5.3.2 Pattern Generation . . . 104
5.3.3 Stabilization Approaches . . . 106
5.3.4 Odometry . . . 107
5.3.5 Parameter Optimization . . . 107
5.3.6 Other Humanoids . . . 108
5.3.7 Evaluation . . . 110
5.4 Stand Up . . . 119
5.4.1 Implementation . . . 119
5.4.2 Parameter Optimization . . . 119
5.4.3 Evaluation . . . 122
5.5 Kick . . . 125
5.6 Summary and Future Work . . . 127

6 Reinforcement Learning 129
6.1 Related Work . . . 129
6.1.1 RL with Reference Motions . . . 130
6.1.2 Choice of Action and State Space . . . 130
6.2 Approach . . . 131

6.2.1 Reference Motion Generation . . . 132
6.2.2 State . . . 132
6.2.3 Action . . . 133
6.2.4 Neural Network Architecture . . . 134
6.2.5 Reward . . . 136
6.2.6 Environment . . . 137
6.2.7 Implementation . . . 137
6.3 Evaluation . . . 139
6.3.1 Choice of Action and State Space . . . 139
6.3.2 Network Initialization . . . 143
6.3.3 State . . . 145
6.3.4 Action and State Based Reward . . . 146
6.3.5 Phase Representation . . . 146
6.3.6 Adaptive Phase . . . 150
6.3.7 Quality of Reference Motions . . . 153
6.3.8 Usage of Arms . . . 156
6.3.9 Comparison to Spline Approach . . . 156
6.3.10 Generalization . . . 158
6.3.11 Domain Transfer . . . 160
6.4 Summary and Future Work . . . 164

7 Conclusion 165
7.1 Future Work . . . 166

8 Appendix 168

Bibliography 172

Online Sources 189

List of Figures

1.1 Comparison of publication count. 2
1.2 Overview of the thesis structure. 8

2.1 Comparison between an android and a humanoid robot. 11
2.2 Photos of a MX-106 servo. 12
2.3 Overview of the control of a Dynamixel Servo. 12
2.4 Photo of a XH540-W270 servo. 13
2.5 Example of RS-485 transmission. 14
2.6 Structure of a DXL protocol package. 15
2.7 How ROS nodes communicate. 17
2.8 Exemplary quintic spline. 19
2.9 Photo of a RoboCup game. 21
2.10 Screenshot of a HLVS game. 22

3.1 Family tree of Darwin-OP. 24
3.2 Pictures of humanoid robot platforms. 27
3.3 Mass and cost of transportation relationship. 29
3.4 Poppy spring knee joint. 29
3.5 Example images of FPSs. 30
3.6 Block diagram of different interfacing approaches. 31
3.7 Size weight relationship of humanoid robots. 32
3.8 Photo of the Wolfgang-OP robot. 33
3.9 Knee joint PEA. 35
3.10 Visualization of used symbols. 36
3.11 Visualization of different values for x_s. 39
3.12 Plot of the torques for the optimal x_s. 40
3.13 Overview of the electronics. 41
3.14 Annotated images of the IMU module. 42
3.15 Generic schematic for a DXL device. 42
3.16 Annotated image of the FPS. 43
3.17 Photo and exploded CAD drawing of a foot cleat. 44
3.18 Illustration response time. 45
3.19 Comparison of different versions of the R-DXL. 46
3.20 NT-Curve of a MX-64 servo. 48
3.21 Images of different models. 50
3.22 Comparison between actual and theoretical control rates. 53
3.23 Logic analyzer output showing the status delay. 53
3.24 Histogram of status response times. 54
3.25 Torque values of a knee during walking. 57
3.26 Torque values during stand-up from the back. 58
3.27 Torque values during stand-up from the belly. 59

3.28 Comparison between the theoretical and real angle torque relationship. . 59
3.29 Images from Wolfgang-OP in competitions. 61

4.1 Used decision making approaches in the HSL. 63
4.2 Exemplary DSL. 69
4.3 Exemplary DAG. 70
4.4 Exemplary DSD stack. 70
4.5 Control flow of the DSD. 71
4.6 The HCM in a general 3T approach. 73
4.7 The HCM in the RoboCup software stack. 75
4.8 The DAG of the HCM's DSD. 76
4.9 Message latencies different publishing frequencies. 81
4.10 Message latency of different executors. 82
4.11 Message latencies for different ROS versions. 82
4.12 Message latencies for different DDS implementations. 83
4.13 Message latencies in the integrated system. 84
4.14 Image sequence of a frontal fall with HCM active. 85
4.15 Exemplary plot of evaluation data. 87
4.16 Mean lead time of different classifiers. 88
4.17 Min lead time of different classifiers. 88
4.18 Mean lead times for different fall directions. 89
4.19 Computing time of different classifiers. 89

5.1 Generation and usage of motion patterns. 92
5.2 Steps of the OptiQuint approach. 97
5.3 Overview of the walk skill implementation. 102
5.4 Visualization of the walk FSM. 103
5.5 Illustration of the walk pattern creation. 103
5.6 Exemplary splines of a walk motion. 104
5.7 Collection of Webots simulator models. 111
5.8 Plot of optimization history. 112
5.9 Comparison between independent and multivariate MOTPE. 112
5.11 Relationship between robot size and achieved walk velocities. 114
5.12 Durations for different IK solvers. 116
5.13 Walk on stairs. 117
5.14 Wolfgang-OP walking. 118
5.15 NUgus walking. 118
5.16 Push-recovery challenge. 118
5.17 Visualization of the standing up motion. 120
5.18 Stand-up skill optimization history. 123
5.19 Comparison of objective functions and optimization algorithms. 124
5.20 Stand up results on real robot. 125
5.21 Influence of PD controller. 126
5.22 Photos of a kick motion. 127
5.23 Kick in Webots. 127

6.1 Overview of the approach. 132
6.2 Structure of neural network. 134
6.3 Normalized actions with and without initial network bias. 135
6.4 Exemplary plot of the radial basis function kernel for different values of κ. 136
6.5 Implementation of the RL environment. 138
6.6 Comparison of different action spaces. 141
6.7 Comparison of different action spaces using the same hyperparameter. . 142

6.8 IK Error of different policies. 142
6.9 Comparison of using the initial bias. 143
6.10 Different control types and initial biases for Walker2DPybullet. 145
6.11 Investigation of using the joint velocity. 146
6.12 Investigation of using the foot pressure sensor (FPS) data. 147
6.13 Comparison of action and state rewards. 148
6.14 Action plot of policies with action and state rewards. 148
6.15 Comparison of different phase representations. 149
6.16 Action plot of policy without phase. 150
6.17 Influence of using the adaptive phase. 151
6.18 Action plot of the adaptive phase policy. 151
6.19 Influence of the adaptive action when the terrain is used. 152
6.20 Comparison of training with different reference motions. 153
6.21 Objective value and reward relationship. 154
6.22 Objective value and reward relationship. 154
6.23 Comparison of training with different references using same velocities. . 155
6.24 Objective value and reward relationship. 156
6.25 Influence of actuating the arms. 157
6.26 Plot of arm joint actions. 158
6.27 Exemplary plot of the actions produced by the policy. 159
6.28 Training of different robots in Webots. 161
6.29 RL policy on real robot. 161

List of Tables

2.1 Specifications of the used DXL servos. 13
2.2 Number of bytes that are transferred using different instruction types. . 16
2.3 Allowed and Forbidden Sensors in the HSL. 20

3.1 Comparison of Different Low-Cost Humanoid Robot Platforms. 26
3.2 Comparison of devices for controlling RS485 or TTL servo Motors. . . . 31
3.3 Wolfgang-OP Specifications. 34
3.4 Mean update rates. 52
3.5 Achieved Control Loop Rates for Different Configurations. 55
3.6 Comparison of Knee Torques. 56

4.1 Definition of the symbols used in the DSL. 68
4.2 Different classes of sensor information used in the falling experiment. . . 85

5.1 Ranges of the optimized parameters. 110
5.2 Objective values for different robot-sampler combinations. 111
5.3 Maximal walk velocities for different robots. 114
5.4 Darwin-OP forward speed 115
5.5 NAO forward speed 115
5.6 Comparison of different IK solvers. 116
5.7 Generalization of the stand-up skill. 123

6.1 Comparison of Sources for Reference Motions 133
6.2 Comparison between reference and policy 159
6.3 Maximal walk velocities for different robots using DeepQuintic. 162

8.1 Ranges of the optimized parameters. 169
8.2 Hyperparameters for PPO 170
8.3 Training Configuration for Different Robot Types 171

List of Abbreviations

3T three-tier (architecture)

AC alternating current

ADC analog-to-digital converter

AE action element (part of the DSD)

AI artificial intelligence

BT behavior tree

CAD computer-aided design

CMA-ES covariance matrix adaptation evolution strategy (an optimization algorithm)

CNC computer numerical control (a manufacturing method)

CoM center of mass

CoP center of pressure

CORE COntrolling and REgulating (the sub board in the Wolfgang-OP)

CPG central pattern generator

CPU central processing unit

CRC cyclic redundancy check (an error-detecting code)

CVSU constant voltage supply unit

DAG directed acyclic graph

DC direct current

DDS data distribution service

DE decision element (part of the DSD)

DoF degree of freedom

DQ DeepQuintic (see Chapter 6)

DRC DARPA (Defense Advanced Research Projects Agency) Robotics Challenge

DRL deep reinforcement learning

DSD Dynamic Stack Decider (explained in Section 4.2)

DSL domain specific language

DT decision tree

DXL Dynamixel (a series of actuators from Robotis)

FA fused angles

FCNN fully convolutional neural network

FIFA Fédération internationale de Football Association (Intern. Football Federation)

FK forward kinematic

FP false positive

FPS foot pressure sensor

FSM finite state machine

FSR force sensitive resistor

GPIO general-purpose input/output

GPU graphics processing unit

HCM Humanoid Control Module (explained in Section 4.3)

HLVS (RoboCup) Humanoid League Virtual Season

HSL (RoboCup) Humanoid Soccer League

HSM hierarchical state machine

HTN hierarchical task network

I2C Inter-Integrated Circuit (a serial communication interface)

ID identifier

IK inverse kinematic

IMU inertial measurement unit

IO input/output

IPM inverse perspective mapping

KNN k nearest neighbors

LED light-emitting diode

LIDAR light detection and ranging

LIPM linear inverted pendulum model

LiPo lithium-ion polymer (battery)

LSTM long short-term memory

MDP Markov Decision Process

MLP multilayer perceptron

MOCAP motion capture

MOTPE Multi-objective Tree-structured Parzen Estimator (an optimization algorithm)

NCS2 Intel Neural Compute Stick 2 (a tensor processing unit)

NSGA-II Non-dominated Sorting Genetic Algorithm II (an optimization algorithm)

NT-curve speed torque curve

NUC Intel Next Unit of Computing (a mini PC)

OQ OptiQuint (see Chapter 5)

PCB printed circuit board

PD proportional derivative (controller)

PID proportional integral derivative (controller)

PEA parallel elastic actuator

PLA polylactic acid (a kind of plastic)

PoE Power over Ethernet

PPO Proximal Policy Optimization

PSU power supply unit

PWM pulse-width modulation

QoS Quality of Service

R-DXL Rhoban DXL Board

RADAR radio detection and ranging

REP ROS Enhancement Proposal

RGB-D red green blue depth

RGB-LED red green blue LED

RM reference motion

RL reinforcement learning

ROS Robot Operating System (a middleware for robots)

RS-485 Recommended Standard 485 (a standard for serial communications)

RSI reference state initialization

SAC Soft Actor-Critic

SPI Serial Peripheral Interface (a serial communication interface)

SPL (RoboCup) Standard Platform League

SQL Structured Query Language

STRIPS Stanford Research Institute Problem Solver (a planning approach)

SVM support vector machine

TB threshold-based

TP true positive

TPE Tree-structured Parzen Estimator (an optimization algorithm)

TPU thermoplastic polyurethane (a kind of plastic)

TTL transistor–transistor logic

TTS trials till success

UART universal asynchronous receiver-transmitter (a serial communication interface)

URDF Unified Robot Description Format

USB Universal Serial Bus

UTS USB to serial

VAT virtual attitude sensor

ZMP zero moment point

List of Symbols

a	coefficient
a	acceleration [$\frac{\mathrm{m}}{\mathrm{s}^2}$]
a_t	action at time point t
a_t^{ref}	reference action at time point t
b	baud rate [$\frac{\mathrm{bit}}{\mathrm{s}}$]
C_t	set of Cartesian pose at time t
$C(o_t)$	classification of observed state o at time point t
d	data to read [B]
d_i^*	true fall direction
d_i	predicted fall direction
d_k	definition for knot
$d(v)$	achieved walk distance given a velocity v [m]
D	dataset
D_C	command knot definitions
D_k	all knot definitions
D_N	natural knot definitions
D_P	parameter knot definitions
D_S	state knot definitions
f	friction [N m]
f	frequency [Hz]
$f(\Phi)$	objective function
g	gravity constant ($9.81\frac{\mathrm{m}}{\mathrm{s}^2}$)
i	instruction package length [B]
I	current [A]
k	knot
k_d	motor damping constant
k_s	spring constant
k_s^*	optimal spring constant
k_s°	good spring constant
k_ϕ	field constant
k_τ	motor torque constant
K	set of all knots of a spline
$K(\hat{x}, x, \kappa)$	radial basis function kernel
j	jerk [$\frac{\mathrm{m}}{\mathrm{s}^3}$]
l	distance to CoM [m]
m	mass of the robot
n	secondaries to address
o	orientation
o_i	observed state
O_d^e	observed states with fall of direction d at t_e

p	position
P	power [W]
$P^n(x)$	polynomial of order n
q	joint position
r	registers to read/write
r_a	action based imitation reward
r_g	goal reward
r_i	imitation reward
r_s	state based imitation reward
r_t	reward at time point t
R	motor resistance [Ω]
R_b	orientation of the robots base frame
$Q(t)$	quintic pose-spline
$Q_R(t)$	set of quintic pose-spline that describe whole robot pose
s	status package length [B]
s_t	state at time point t
s_τ	torque scaling factor
s_ω	angular velocity scaling factor
$S^n(x)$	spline of order n
t	time point [s]
t_e	end time of a observation stream o_d^e
t_{fp}	time of first false positive prediction
t_l	lead time of fall prediction
t_p	fall prediction time
t_{tp}	time of first true positive prediction
T	duration [s]
v	velocity [$\frac{\text{m}}{\text{s}}$]
v_{cmd}	commanded walk velocity [$\frac{\text{m}}{\text{s}}$]
V	voltage [V]
x_s	spring scaling factor
$\hat{x}$	desired value

α	pendulum angle [rad]
β	spring bending angle [rad]
γ	angle of knee joint [rad]
δ	mounting offset angle of the spring [rad]
θ	parameter
Θ	parameter set
Θ^*	optimal parameter set
κ	scaling factor for radial basis function kernel
μ	mean of Gaussian distribution
π	circle constant (ca. 3.14)
$\pi(a_t\|s_t)$	policy function
Σ	covariance matrix of Gaussian distribution
τ	torque [Nm]
τ_d	knee torque when robot is standing on both legs (double support) [Nm]
τ_n	knee torque if leg has no ground contact [Nm]
τ_e	torque generated by an electrical motor [Nm]
τ_{eff}	effective torque [Nm]
τ_h	stall torque [Nm]
τ_m	knee torque resulting from robot mass [Nm]

τ_s	knee torque when robot is standing on one leg (single support) [Nm]
τ_t	torque generated by torsion spring [Nm]
ϕ	phase
Ψ	set of all possible parameter sets
ω	angular velocity $[\frac{\text{rad}}{\text{s}}]$
ω_b	angular velocity of the robot's base $[\frac{\text{rad}}{\text{s}}]$

Chapter 1

Introduction

1.1 Motivation

In the last decades, robots have proven their usefulness in a wide range of domains, e.g., mass production in factories [SK16, pp.1386-1392] or space exploration [SK16, pp.1424-1437]. Still, they are rare in our everyday lives, as integration into our typical environment is challenging. The human everyday domain is complex and dynamic, making it difficult for robots to perform tasks successfully. Furthermore, it is specifically designed for human abilities, e.g., by using stairs adapted for the human bipedal locomotion system. Therefore, commonly used robot types, i.e., wheeled platforms, can only be used to a certain extent. Humanoid robots, on the other hand, pose the chance of integrating robots into the human domain better [SK16, p. 1790] and can be used to improve our understanding of how humans work by trying to replicate their abilities [SK19]. However, they come with additional challenges due to their complicated hardware and the need for complex control software.

Research on humanoid robots has been growing constantly since the middle of the 1990s (see Figure 1.1). While it is still a small research topic compared to robotics overall, the general public's interest is high. Humanoid robots have been featured in many famous science fiction works. Thus, the viewers have seen how they could benefit from humanoid robots working in their environment. This includes tedious tasks, e.g., cleaning, or fields of work where human laborers are missing, e.g., elderly care. Still, these expectations can not yet be satisfied by the current state of the art in humanoid robotics.

We identified three main issues that need to be solved to allow a wide application of humanoid robots. First, they are not robust enough. Due to their bipedal nature, they are inherently unstable and might fall down. The impact of these falls often damages the robot's hardware, especially the gears, requiring expensive and time-intensive repairs. Therefore, most humanoids are currently used in controlled lab environments where falls can be prevented. Second, they are too expensive to build. Due to their high degree of freedom (DoF) kinematic structure, they require a high number of actuators and linking mechanical elements. These actuators also need to be precise to ensure that the robot can be kept stable, as well as strong enough to achieve the high torque requirements for the leg joints. Third, they lack reliable and simple to create motion skills. In comparison to other robot types, i.e., robotic arms and wheeled platforms, humanoids pose additional challenges as constant balance needs to be ensured and multiple kinematic chains need to be controlled. This prevents or hinders the use of many simple approaches, e.g., teaching a motion trajectory by moving the endeffector manually, as it is often done for robotic arms. Furthermore, generalizing skills for different humanoids is complicated as

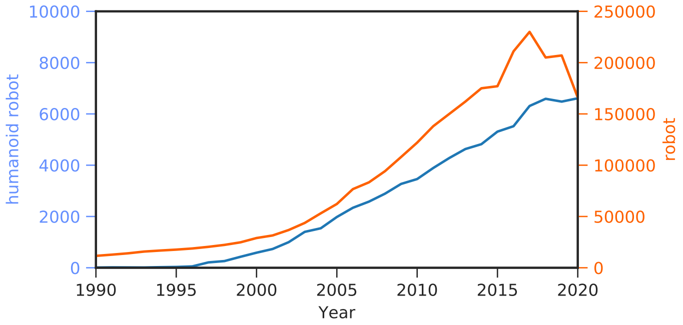


Figure 1.1: Number of publications containing the term "humanoid robot" (**blue**) and the term "robot" (**orange**) per year. Both research areas show an increase in publications over the year, while the general robotics topic has naturally a much higher number of publications. Data is extracted from `scholar.google.com`.

they have different kinematic and dynamic properties.

At the same time, new developments in deep reinforcement learning (DRL) and the constantly increasing computational power of computers provide new chances to tackle these issues. This thesis contributes to all three issues by providing new developments for humanoid hardware, a novel approach for integrated fall management, and, most importantly, new approaches for motion skill creation.

1.2 Aims of this Thesis

This thesis aims to advance research in the field of humanoid robotics in a meaningful way. Therefore, the developed approaches should be tested on multiple robots to ensure that they are generally applicable rather than specific to one kinematic configuration or robot size. Furthermore, the approaches should be tested in realistic conditions. This is difficult, as the capabilities of humanoid robots are far from being able to be applied to real-world scenarios. Robotic competitions, such as RoboCup, provide an excellent intermediate step to this goal since the domain is less controlled than a lab environment but still simpler than the natural human domain.

The requirements for a well-working humanoid robot are manifold, but one fundamental pillar is robust bipedal locomotion, which also includes the management of falls and standing-up motions. Since falls are not entirely avoidable, it is essential that the robot sustains a fall without significant damage and that it can stand up again by itself. Otherwise, the robot could never be fully autonomous, as it would always require the assistance of a human after falls. Therefore, achieving a combination of a bipedal walk with fall management is one of the objectives of this thesis. Since DRL approaches have shown viable results in the past, it should be investigated whether they can be applied to achieve stable locomotion and other humanoid motions. An auspicious direction combines DRL with reference motions (RMs) to guide the learning process. Until now, these were either recorded motion capture data or manually generated as keyframe animations. Rather than using one of these, it will be explored whether conventional walk controllers can provide better RMs.Here, it should be investigated how the performance of the learned policy is related to the quality of the RM.

Another critical aspect of the robot is its price, which limits its use cases. Most current humanoid robots, e.g., Atlas [NSP19a], or Valkyrie [RSH^{+}15], are expensive since they need precise actuators to achieve stable walking. While some low-cost robots exist, e.g., the Darwin-OP [HTA$^+$11], they pose additional challenges due to imprecise actuators, primarily due to backlash, and noisier sensors. Despite this, only low-cost hardware will be used in this thesis. On the one hand, to improve the hardware while maintaining low-cost and, on the other hand, to verify that the software approaches are applicable to hardware that consumers could buy one day.

1.3 Research Questions

This section explains the main research questions investigated by this thesis and why they are relevant.

Improvement of Low-Cost Humanoids Can current low-cost humanoid robots be improved to achieve a higher control loop rate while maintaining the widespread serial bus system? Can the load on the knee joints be reduced so that energy is saved and cheaper servos can be used? These questions are essential to achieve cost-efficient hardware for the other experiments.

Quality of Reference Motion If a reference motion is used during reinforcement learning (RL) to shape the reward, how does the quality of the RM influence the quality of the learned policy? Is there a proportional relationship between the performance of the RM and the policy? Does the training fail if the quality of the RM is below a certain value? Previous approaches have used RM that are not optimized and, therefore, may limit the quality of the resulting policy.

State, Action and Reward Design for DRL Walking Which state and action designs lead to a well-performing policy? Is the usage of Cartesian space superior to joint space? How can orientations be represented best in Cartesian space so that the neural network of the policy can handle it? Does an action-based reward improve the learning speed compared to a state-based one? The quality of the resulting policies depends on these choices.

1.4 Contributions

This section lists the main contributions of this thesis toward the different research fields.

Improving Dynamixel Bus Communication The shortfalls of current interfacing solutions for bus systems with the Dynamixel protocol (see Section 2.2.1) were investigated. Based on this, a novel approach (the QUADDXL) was proposed, allowing a significant increase in the achieved control rate. This system was successfully integrated into the Wolfgang-OP platform, and its applicability was proven. Since Dynamixel servos are one of the most widespread robot servos, the impact on the general robotics community is high. The approach was already adapted by others, i.e., by the Humanoid Soccer League (HSL) team CITBrains in their new platform SUSTAINA-OP [22].

PEA for Knee Actuator Torque Reduction A low-cost parallel elastic actuator (PEA) solution was developed, which reduces the load on the knee actuators of humanoid robots. Thereby reducing energy consumption and allowing the usage of smaller servos. The formula to achieve the optimal torque reduction was deduced, and its validity was tested on real hardware.

Robot Platform The humanoid robot platform Wolfgang-OP, which integrates multiple new approaches, i.e., PEAs in the knees and 3D printed elastic elements was developed. The platform is entirely open source and open hardware. Thus it can be built by others and be used as a reference design when developing new platforms. It also features accurate simulation models for PyBullet [15], and Webots [Mic04]. Furthermore, it is one of the first humanoid robots with a Robot Operating System (ROS) 2 software stack for locomotion and fall management. During the development of the software, contributions were also made to the general ROS 2 community. This platform was developed together with the Hamburg Bit-Bots (see Chapter 3 for details).

Software Abstraction Layer for Humanoid Robots A new architectonic approach, called Humanoid Control Module (HCM), was developed, which builds upon the three-tier (3T) [PBJFG+97] approach. It adds a fourth layer, which abstracts from the fact that the robot is humanoid by taking care of fall management and hardware issues, as well as implementing an actuator control mutex. On the one hand, it allows the motion skills to be decoupled into different modules, and on the other, it allows control of the robot in a similar fashion to a wheeled robot. This simplifies software development for humanoid robots.

Fall Detection Different classification approaches based on different modalities for the detection of falling in humanoid robots were compared. The results allow the detection of falls with higher reliability and a longer lead time, thus allowing counter measurements and thereby reducing damages during falls. Fall detection for humanoid robots is still little investigated, as most robots only work in lab conditions where falls are externally prevented. It is still an important skill to allow the usage of humanoid robots in real-world scenarios.

Decision Making Framework Co-development of a new high-level control architecture (called Dynamic Stack Decider (DSD)) that is the basis for the HCM. It allows programming dynamic behaviors in an understandable way. It has also been used to program the high-level agent behavior for HSL and for building a blackjack dealer robot [FGN+23].

Optimized Spline Motions An approach (called OptiQuint) using parameterized Cartesian quintic spline motions with automatic parameter optimization was developed. It was shown that it can be used for bipedal walking, as well as standing up and kick motions. Its generalizability has been proven by application on various robot platforms. The automatic parameter optimization makes it convenient to use for others. While it may not provide the fastest and most stable walk controller, it is easy to deploy onto a new robot and, therefore, unique. The impact of having such a simple-to-use walk controller is proven as others, e.g., the HSL team NUbots, have already successfully used it.

Reinforcement Learning with Optimized Reference Motion An approach (called DeepQuintic) was developed to create motion policies with DRL. In contrast to existing approaches, the RM was created using OptiQuint, instead of relying on motion capture (MOCAP) or keyframe animations. The relationship of the policy performance with the quality of the RM was investigated, and it could be shown that a better RM leads to an improvement of the policy's performance.

Reinforcement Learning using Position Control The DeepQuintic approach was used to investigate how the space representation influences the policy's performance. It could be shown that the usage of Cartesian space is beneficial for bipedal locomotion. Furthermore, it was shown that the choice of the initial action of the policy has a high impact on the training if position control is used. In addition, different other choices for the design of the training process were evaluated.

1.5 Publications

Parts of this thesis have already been published in research papers. These are listed below with a short explanation of their content. All soft- and hardware that was developed was released open source (see Section 1.6).

1.5.1 Core Academic Publications

This section describes the publications that are directly related to this thesis.

Marc Bestmann, Jasper Güldenstein, and Jianwei Zhang. "High-Frequency Multi Bus Servo and Sensor Communication Using the Dynamixel Protocol." RoboCup Symposium, 2019.

[BGZ19] compares the different available interface solutions for servo-sensor-bus system using RS-485 and the Dynamixel protocol. It investigates the communication bottlenecks that limit their performance and proposes a new approach (called QUADDXL) that allows higher control loop rates. Additionally, a compatible inertial measurement unit (IMU) module and a foot pressure sensor are developed that can be read at a higher rate than previous models. Furthermore, open-source software is provided to directly control the hardware from ROS.

Marc Bestmann, Jianwei Zhang "Humanoid Control Module: An Abstraction Layer for Humanoid Robots" IEEE ICARSC, 2020.

[BZ20] presents a new software architecture approach to decouple parts of the motion software stack as well as handling the fall management of a humanoid robot. Using this approach allows the user to abstract from the complicated bipedal locomotion of a humanoid and thus allows navigation as simple as on a wheeled robot.

Martin Poppinga, **Marc Bestmann** "DSD - Dynamic Stack Decider: A Lightweight Decision Making Framework for Robots and Software Agents" International Journal of Social Robotics, 2021

[PB22] proposes a new decision-making framework called Dynamic Stack Decider. It combines ideas from existing approaches, e.g., behavior tress [CÖ18], to achieve a lightweight and simple-to-use framework that allows the implementation of complex behaviors. Examples of the usage and a qualitative comparison to the existing frameworks are given.

Marc Bestmann, Jasper Güldenstein, Florian Vahl, and Jianwei Zhang "Wolfgang-OP: A Robust Humanoid Robot Platform for Research and Competitions" IEEE Humanoids, 2021.

[BGVZ21] describes the open-source humanoid robot platform "Wolfgang-OP". Special attention is given to the new features of this platform. The main contribution is a new low-cost PEA for the robot's knee, which reduces its load and thereby allows the usage of smaller motors and conserves energy. The paper also includes a formula to derive the necessary spring force for the PEA as well as real-life experiments that verify the approach. Additionally, the integration of the QUADDXL approach into a complete robot and the validation of its performance are shown. Furthermore, 3D-printed elastic elements are presented that reduce the damage to the robot's hardware during falls.

Sebastian Stelter, **Marc Bestmann**, Norman Hendrich, and Jianwei Zhang "Fast and Reliable Stand-Up Motions for Humanoid Robots Using Spline Interpolation and Parameter Optimization" IEEE ICAR, 2021.

[SBHZ21] shows how usage of parameterized Cartesian quintic splines can create standing-up motions for humanoid robots. An approach for automatic parameter optimization is presented and tested on multiple robot models. Additionally, it is shown that this approach successfully reduces the necessary time to stand up and that the parameter optimization is superior to manual parameter tuning. Although this paper was published before [BZ22], it is building up on it.

Marc Bestmann, Jianwei Zhang "Bipedal Walking on Humanoid Robots through Parameter Optimization" RoboCup Symposium, 2022.
(**nominated for best paper award**)

[BZ22] describes a generic approach for bipedal walking based on parameterized Cartesian quintic splines. An automatic parameter optimization method is proposed, and different optimization algorithms are evaluated. Furthermore, it is shown that the approach works on a wide range of robot platforms, and the achieved baselines are provided.

Marc Bestmann, Jianwei Zhang "Reference Motion Quality and Design Choices for Bipedal Walking with DeepRL", **submitted to** International Journal of Humanoid Robotics, 2023

[BZ23] investigates the influence of the reference motion's quality in RL. It uses the walk engine from [BZ22] as references with parameters of different qualities. Furthermore, the paper describes our various experiments on the design of the RL training environment. The paper was submitted briefly before finishing this thesis and is, therefore, at the time of writing still in the peer-review process.

1.5.2 Further Academic Publications

This section describes further publications that are not the core of this thesis but are loosely related, mainly because they use the same robot platform.

Niklas Fiedler, **Marc Bestmann**, Jianwei Zhang "Position Estimation on Image-Based Heat Map Input using Particle Filters in Cartesian Space" IEEE ICPS/MFI, 2019.

[FBZ19] uses a particle filter to improve the position estimation of a ball that is detected with a fully convolutional neural network (FCNN). An early version of the Wolfgang-OP robot was used in the experiments.

Niklas Fiedler, Hendrik Brandt, Jan Gutsche, Florian Vahl, Jonas Hagge, **Marc Bestmann** "An Open Source Vision Pipeline Approach for RoboCup Humanoid Soccer", RoboCup Symposium, 2019.

[FBG+19] describes the open-source vision pipeline used on the Wolfgang-OP robot in the RoboCup competitions. Its runtime and object detection performances are evaluated.

Marc Bestmann, Timon Engelke, Niklas Fiedler, Jasper Güldenstein, Jan Gutsche, Jonas Hagge, and Florian Vahl "TORSO-21 Dataset: Typical Objects in RoboCup Soccer 2021", RoboCup Symposium, 2021.

[BEF+22] provides a standardized dataset for the RoboCup humanoid soccer domain with a baseline for object detection on it. The dataset contains many images recorded by or showing a Wolfgang-OP robot and was used to train the vision networks.

Florian Vahl, Jan Gutsche, **Marc Bestmann**, and Jianwei Zhang "YOEO – You Only Encode Once: A CNN for Embedded Object Detection and Semantic Segmentation" IEEE ROBIO, 2021.

[VGBZ21] presents a novel YOLO-like neural network architecture that allows object detection and image segmentation at the same time. It was later deployed on the Wolfgang-OP for participation in RoboCup.

1.5.3 Team Description Papers

These team description papers also describe parts of this thesis that have been used in RoboCup. They are created for participation in the RoboCup championship and are not peer-reviewed. Thus, they are more like technical reports. Still, they are read and cited by other teams and researchers.

Marc Bestmann et al. "Hamburg Bit-Bots and WF Wolves Team Description for RoboCup 2019 Humanoid KidSize" RoboCup, 2019.

[BBE+19] describes the first developments for the DSD, the walk parameter optimization and the Wolfgang-OP.

Marc Bestmann et al. "Hamburg Bit-Bots Humanoid League 2020" RoboCup, 2020.

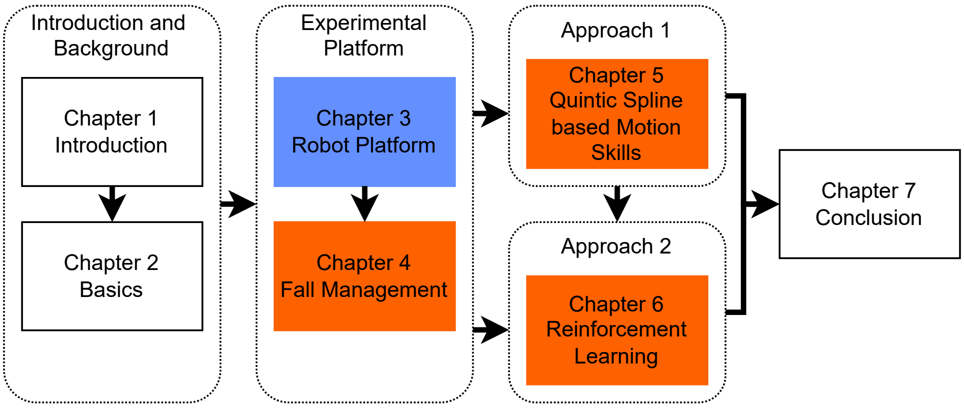

Introduction and Background
Chapter 1 Introduction
Chapter 2 Basics
Experimental Platform
Chapter 3 Robot Platform
Chapter 4 Fall Management
Approach 1
Chapter 5 Quintic Spline based Motion Skills
Approach 2
Chapter 6 Reinforcement Learning
Chapter 7 Conclusion

Figure 1.2: Overview of the thesis structure. Chapters focusing on hardware are marked in **blue**, while chapters focusing on software are marked in **orange**.

[BFGH20] describes issues of integrating the previous version of the walk controller into the software stack and the development of the Wolfgang-OP platform.

Jasper Güldenstein and **Marc Bestmann** "Hamburg Bit-Bots Team Description for RoboCup 2022" RoboCup, 2022.

[GB22] describes the development of the spline-based stand-up motions and the first work on the RL based walking.

1.6 Thesis Structure

Some things described in this thesis have been developed in cooperation with colleagues or students. Leaving these parts out is not feasible as components build up on each other. Instead, a box, as shown below, is used to clarify where others have made a significant contribution and what was not done by me.

This box shows that parts have been done in cooperation with others.

All digital supplementary material for this thesis is aggregated at [1]. This includes links to the open-source repositories and various videos.

This thesis consists of two main parts, the development of the experimental platform and the approaches for creating motion skills (see also Figure 1.2). For the first part, a robust, low-cost humanoid robot platform was developed (see Chapter 3). It is usable for the experiments of this thesis, but it is also used for participation in RoboCup. This way, the thesis results were tested during competitions.Since the later experiments for the motion skills require multimodal sensing capabilities, foot pressure sensors, as well as IMUs, were integrated into the robot. Furthermore, the hardware interface was improved for a fast perception-action-cycle that allows closed-loop control. The issue of high loads on the knee joints was addressed by developing a new PEA based approach for load reduction. Finally, adding elastic elements improved the robustness of falls. These hardware modifications alone are not sufficient to make the robot robust against falls since it also needs to move into a safe pose for the fall. To achieve this, a new software architecture approach for handling fall management is created, and different fall classifiers are investigated (see Chapter 4). Originally, it was planned to use this to train the reinforcement policy directly on the robot. These plans were later changed because the training time would be too long, resulting in too much wear on the hardware.

For the creation of motion skills, two different approaches are presented. The first approach is based on generating motion patterns with parameterized Cartesian quintic splines (see Chapter 5). It is shown how their parameters can be optimized and that the approach is applicable to a wide variety of robots and different types of motion skills. The second approach utilizes a novel approach based on RL that utilizes locomotion based on quintic splines as a reference motion (see Chapter 6). The influence of the reference motion quality, as well as various environment design choices, is investigated, and the generalizability of the approach is demonstrated.

Chapter 2

Basics

This chapter describes the fundamentals that are necessary to understand the thesis. First, humanoid robots are defined in Section 2.1. Then, servo motors are described in Section 2.2 with a focus on the Dynamixel servos that were used in this thesis. Afterward, the ROS middleware is described in Section 2.3 as the software in this thesis uses it. This is followed by an overview of splines in Section 2.4. Finally, the RoboCup competition is described in Section 2.5.

2.1 Humanoid Robots

Humanoid robots are defined rather broadly without an exact and commonly accepted definition. Most definitions are centered around being human-like in shape or function.

- "The field of humanoid robotics focuses on the creation of robots that are directly inspired by human capabilities" [SK16, p.1789]
- "Per definition, humanoid robots are characterized by a human-like appearance and/or functionality." [MFL19]
- "A humanoid robot is a robot that has a human-like shape." [KHHY14, p.1]

A more clearly defined subtype of humanoid robots are androids. These "are designed to have a very human-like appearance with skin, teeth, hair, and clothes" [SK16, p.1937]. Examples of an android robot and a non-android humanoid robot can be seen in Figure 2.1.

There are different reasons for the research on humanoid robots [SK16, pp.1790-1791]. Some of them relate to the field of human-robot interaction, e.g., understanding human psychology through interaction with robots. These typically require human-looking robots (androids) rather than a robot being able to perform human motion skills. Nevertheless, there are also reasons to create robots that may not look human-like but have similar physical properties. First, we know that humans can perform various tasks well. Therefore, imitating our physiology is a promising approach to create a robot with general capabilities. Second, the environment humans created is adapted to them, e.g., we use stairs that are practical for bipedal walking and have high wall shelves that can only be reached by having a vertical body. It would also be possible to adapt the environment to the robot's abilities which is, for example, done in warehouses for logistic robots. Still, this would require extreme efforts and may reduce the usability of the environment by humans. Therefore, it makes sense to adapt the robots to the environment and not vice versa. See also Section 3.1.1 for an overview of humanoid robot platforms.

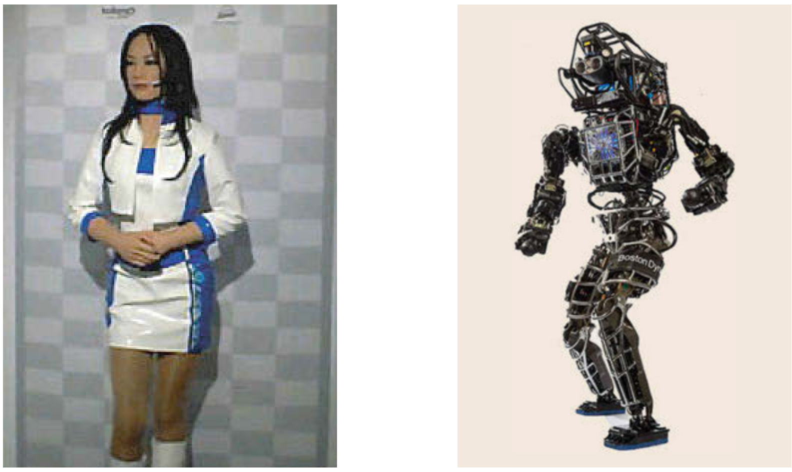


Figure 2.1: Comparison of the *Actroid* android robot (**left**) and the *Atlas* humanoid robot (**right**). [SK16, pp.1791-1792]

2.2 Servo Motors

A servo is a type of actuator that allows it to directly control its position. It usually consists of a direct current (DC) motor, a gearbox, a rotary encoder, and a microcontroller. Additionally, it often includes a bus interface. It utilizes closed-loop control that compares its current state to a desired input state [Poo89, p.98] by using the rotary sensor and the microcontroller. Different types of servos can be classified based on the provided power (AC/DC) and on how the energy is transmitted to the electromagnets (brushed/brushless). Since only brushed DC servos are used in this thesis, the word "servo" will refer to this type of servo.

2.2.1 Dynamixel Servos

One of the most widespread robot servo motor types is the Dynamixel (DXL) series from Robotis. Since this servo type is used in this thesis, it is explained in detail in the following. These servos combine a DC motor, a gearbox, and control electronics into a small casing that allows easy integration into a robot (see Figure 2.2). The servo uses a daisy-chained serial bus for power and half-duplex communication. This reduces the number of cables in the robot, as only one cable is necessary for each limb instead of one for each motor. Thus only five cables are necessary for a humanoid robot (two arms, two legs, one head) instead of the ≥ 20 cables required for parallel communication with each servo. Additionally, it is possible to use the same bus system for different sensors that can communicate with the same protocol. This bus connection and the used protocol are explained in detail in Sections 2.2.2 and 2.2.3, respectively.

The integrated electronics consist of an STM32F013 microcontroller, power management, a bus transceiver, and a rotary encoder to sense the servo's position. The position sensor is an AS5045 spinning current Hall effect sensor with a 12-bit resolution (ca. 0.1° resolution). It measures the rotation of a magnet attached to the output axis.

The servo provides the following control modes: current, velocity, position, multiturn position, current-based position, and pulse-width modulation (PWM) control mode. The microcontroller executes these based on the sensed rotation and, in case of current or current-based position control mode, the applied motor current. The main advantage of using position-based control instead of direct force control is the high internal

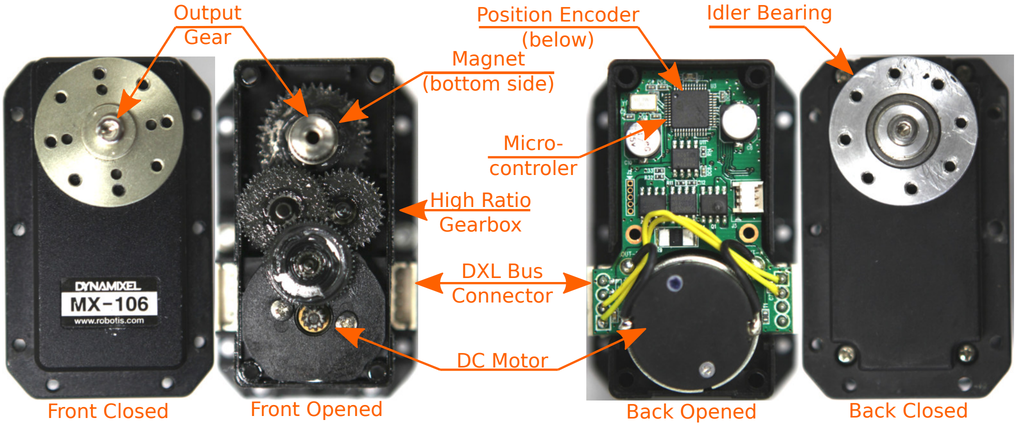


Figure 2.2: Photo of a MX-106 servo from both sides with and without cover. The hall-effect-based position encoder is not visible as it is on the backside of the printed circuit board (PCB). Similarly, the corresponding magnet is not visible as it is on the backside of the output gear. The MX-64 servo has the same structure and electronics but is smaller.

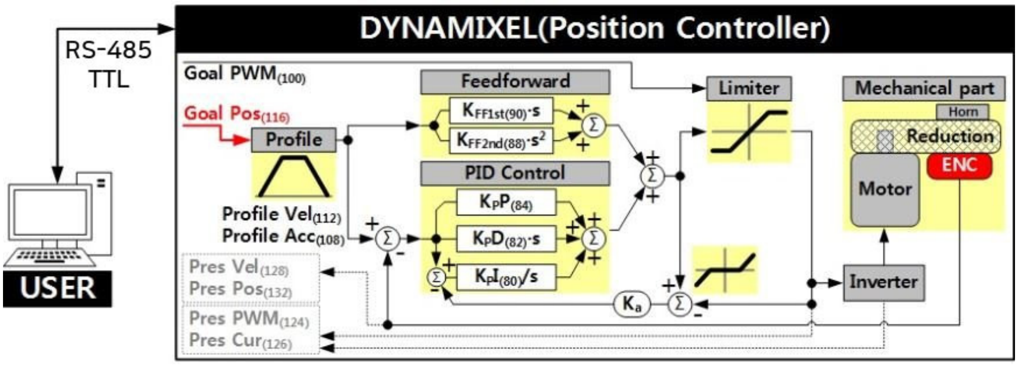


Figure 2.3: Overview of the control of a Dynamixel Servo. The current is sensed by measuring the motor input. The position and velocity of the servo are sensed by a rotary encoder (displayed as “ENC”) that sits on the last gear’s axis. [26]

control loop frequency. If direct force control is used, the low-level control loop of the joint would include the whole bus system. In high-DoF-robots, such as humanoids, this is problematic since the maximal control rate on the bus is comparably low, as many servos need to be communicated with. In contrast to this, the control loop inside of the servo can be much higher and is independent of the number of used servos since no bus communication is involved. See Figure 2.3 for a visual explanation of the position control mode.

There are different series, each of which has a different casing (see Figure 2.2 and Figure 2.4). The casing of the X-series directly includes the mounting threads and allows cable routing through the servo axis. Each series has different types that mainly differ in motor power and gear ratio. In this thesis, the MX-64, MX-106, and XH540-W270 were used (see Table 2.1).

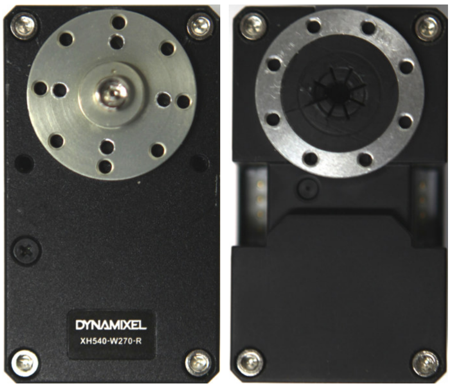


Figure 2.4: Photo of the front (**left**) and back (**right**) of a XH540-W270 servo. Note the hollow bearing, which allows routing cables through the joint axis.

Type	Stall torque @14.8V	No load speed @14.8V	Weight	Backlash
MX-64	7.3 Nm	78 rev/min	135 g	0.33°
MX-106	10.0 Nm	55 rev/min	153 g	0.33°
XH540-W270	11.7 Nm	46 rev/min	165 g	0.25°

Table 2.1: Specifications of the used DXL servos [26].

2.2.2 RS-485 and TTL

Recommended Standard 485 (RS-485) (also known as TIA-485(-A) or EIA-485) is a widespread industry standard for asynchronous serial data transmission [SZC+02]. It uses a differential pair of data wires to increase the robustness against noise from electrical fields and can be used for long-range data transmissions. To do this, the signal is transmitted on one wire, and the inverted signal is transmitted on a second wire. The receiver can reconstruct the original signal by taking the difference between both wires. If the result is $< -1.5V$ it is a 0 bit and if it is $> 1.5V$ it is a 1 bit. Thereby, this procedure may still provide a correct digital value even if electrical interferences shift the voltage levels. It is especially beneficial when combined with robotic servos as the DC-motors in the servos use electromagnets and can thus induce electric disturbances into the wires.

Additionally to the two wires for the RS-485, two wires for DC power supply are used when daisy-chaining the DXL servos. This leads to a four-wire cable for most DXL servo models. Some models of DXL servos only use a three-wire connection. In this case, the data transfer is done via universal asynchronous receiver-transmitter (UART) on a single wire. This is called transistor–transistor logic (TTL) by the manufacturer. Therefore, we will use the same term. The voltage level of the TTL wire is then only compared to the ground, reducing the robustness to noise but reducing the necessary wires. The positive signal of an RS-485 transceiver acts the same as a TTL signal if it is driven with 5 V. Vice versa, a RS-485 signal can be created from a TTL signal by tying the inverted data line to 2.5 V. Since RS-485 requires only a differential voltage of 1.5 V and the TTL level is on 5 V, the transmitted bits will be interpreted correctly (see Figure 2.5). Naturally, the robustness to noise is decreased compared to the standard RS-485 signal, where both lines are actively driven. In the remaining thesis, the communication standard will always be called RS-485 for simplicity, although everything will also work with TTL.

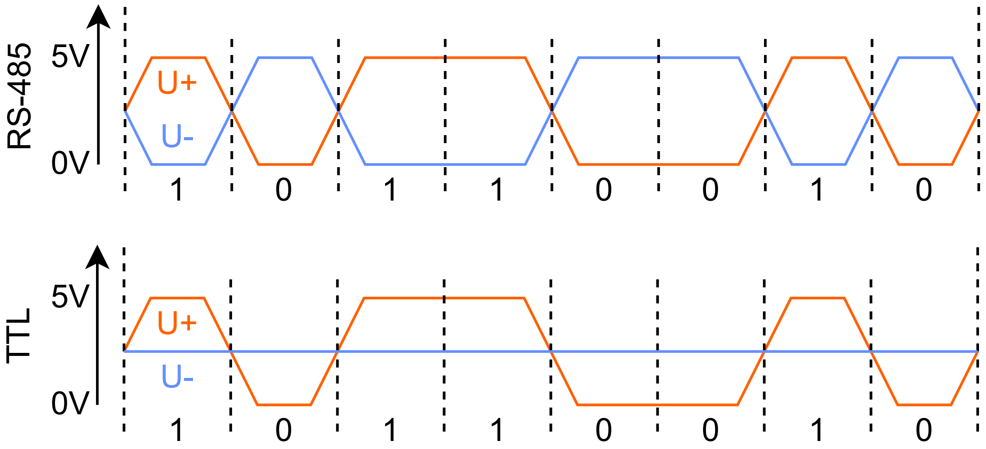


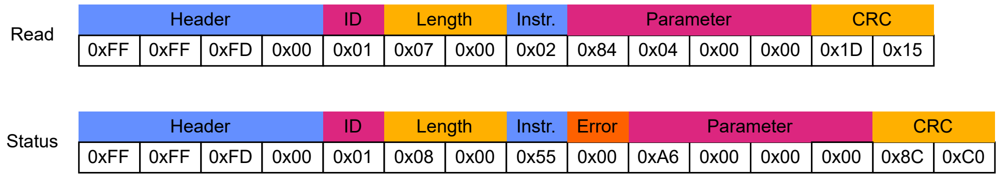
Figure 2.5: Example of a transmitted byte via RS-485 (**top**) via a two-wire interface. It is also possible to directly transfer a transistor–transistor logic (TTL) signal via RS-485 (**bottom**) if the inverted line (U-) is shifted to 2.5V.

2.2.3 Dynamixel Protocol

The Dynamixel protocol is a primary/secondary half-duplex protocol with 8 bit, 1 stop bit, and no parity [26]. All Dynamixel servos, independent of the series, use it. Since it makes sense to only use one protocol for a bus system, different compatible open-source sensors have been created (see Section 3.4.1). There are two versions (1.0 and 2.0) of the Dynamixel protocol. Version 2.0 allows using larger packages and improves the computation of the checksum. Furthermore, it uses byte stuffing to prevent package headers from occurring inside a package. This can happen as the transmitted data may contain, by chance, the same bytes that make up the package header. Thereby, a recipient can be confused as it would wrongly signal the start of a new package. The byte stuffing adds an additional byte at that point so that this data is clearly not a header.

Additionally, it now officially supports all sync and bulk operations (see explanation below), which were only partially possible with version 1.0 and not officially supported. Both versions are incompatible, but older servos can use this version when their firmware is updated. In the following, we will only discuss the newer version.

There are two general types of packages, the instruction package and the status package. The primary (typically a controller board connected to a computer) sends the instruction via the bus to one or more secondaries (typically servos or sensors) which may respond with a status package (dependent on the type of instruction). Each instruction package starts with a header that marks the start of the package (see also Figure 2.6). After that, an identifier (ID) specifies the recipient of the instruction. Due to this ID based protocol, each of the devices on the bus must have a unique ID, which is encoded with one byte. The ID value 255 is not used (probably to reduce the chance of accidentally having header bytes inside the package), and the 254 is reserved for sync instructions (see below). This allows up to 253 devices on the bus. After the ID, the length of the package is specified, followed by a byte that specifies the type of instruction. After that, an arbitrary number of parameters that specify the content of the instruction can be added to the package. Finally, the package ends with a two-byte checksum that ensures that no bits were wrongly transmitted. The CRC-16 algorithm [PB61] is used to compute this checksum.



Read	Header				ID	Length		Instr.	Parameter				CRC	
	0xFF	0xFF	0xFD	0x00	0x01	0x07	0x00	0x02	0x84	0x04	0x00	0x00	0x1D	0x15

Status	Header				ID	Length		Instr.	Error	Parameter				CRC	
	0xFF	0xFF	0xFD	0x00	0x01	0x08	0x00	0x55	0x00	0xA6	0x00	0x00	0x00	0x8C	0xC0

Figure 2.6: Structure of a DXL protocol package presented by an example of a read package (get the current position of a servo) and the corresponding status package that is returning the current position.

A status package is defined very similarly, with just three differences. First, the ID in a status package describes the ID of the sender, in contrast to the recipient ID in an instruction package. Second, the instruction byte is always 0x55 to show that this is a status package. Third, the status package has an extra error byte after the instruction byte to indicate different possible errors, e.g., a checksum error.

There are 15 different instruction packages, including instructions typically not used while running the robot, e.g., factory reset. In the following, we will focus on the different read and write instructions since only these are used to control the robot's motions. The simplest way to control the servos are the standard single read/write instructions. Here, each instruction reads or writes an arbitrary number of bytes from a single secondary. In the case of a read, a single status package with the read data is sent back. The downside of this is that a high number of packages, and therefore protocol overhead, need to be sent over the bus if the number of devices on the bus is high, which is the case for humanoid robots which have many DoFs.

Bulk or sync instructions can be used to reduce the number of packages (see Table 2.2). The sync instructions allow to read or write the same registers of multiple devices with a single instruction. The bulk instructions work similarly but allow specifying different registers for each secondary. This makes the instruction more flexible but also increases its length. In the case of sync/bulk reads, each of the secondaries returns a separate status package. After publishing our results on optimizing control rates for DXL servos [BGZ19] (see also Section 2.2.1), the fast sync read instruction was introduced to the protocol. It is unknown to us whether this was a direct reaction to our findings or just coincidental. It works similarly to the sync read instruction, but all secondaries return a single shared status package. Thus the number of bytes can be further reduced. Unfortunately, only certain newer types of servos, not including the MX series, support this instruction, and therefore it is not used in this thesis.

2.3 ROS

The Robot Operating System (ROS) [QCG$^+$09] is, contrary to its name, not a real operating system but a middleware targeted to the robotics domain. It facilitates developing software stacks for robots by providing message-passing-based communication between different software components (called *nodes*) as well as providing compatible standard solutions for different tasks, e.g., IMU filtering or mobile robot navigation.

There are two different versions of ROS: ROS1 [QCG$^+$09] and ROS 2 [MFG$^+$22]. Both have multiple sub-versions that differ in details, but versions 1 and 2 of ROS have partially different concepts. The thesis was started with ROS 1 but we then switched to ROS 2. Therefore, *ROS* will be used synonymously for both versions in this thesis, Since the final version of the code uses ROS 2, we will only discuss this version in the following.

i: Instruction length [B]
s: Status length [B]
n: Secondaries to address
r: Registers to read/write

read $i = n \cdot 14$ $s = n \cdot (11 + r)$	write $i = n \cdot (12 + r)$
bulk read $i = 10 + n \cdot 5$ $s = n \cdot (11 + r)$	bulk write $i = 10 + n \cdot (5 + r)$
sync read $i = 14 + n$ $s = n \cdot (11 + r)$	sync write $i = 14 + n \cdot (r + 1)$
fast sync read $i = 14 + n$ $s = 8 + n \cdot (4 + r)$	

Table 2.2: Number of bytes that are transferred on the bus using different instruction types, sorted from less to more efficient. Note that not all of these methods are always applicable, e.g., fast sync read is only available for certain newer servo models, and sync commands require to use the same registers on all devices.

Software written with ROS is modularized in *nodes*. Each of these nodes can communicate with other nodes via message passing of three different types (see Figure 2.7). The most used and simplest form are *topics*. These create a one-directional, asynchronous, one-to-many relationship between one node that publishes messages on a topic and any number of other nodes that listen to the same topic and thus receive the messages. A typical usage of a topic would be one node that provides sensor data to all other nodes and sends a message each time it reads the sensor.

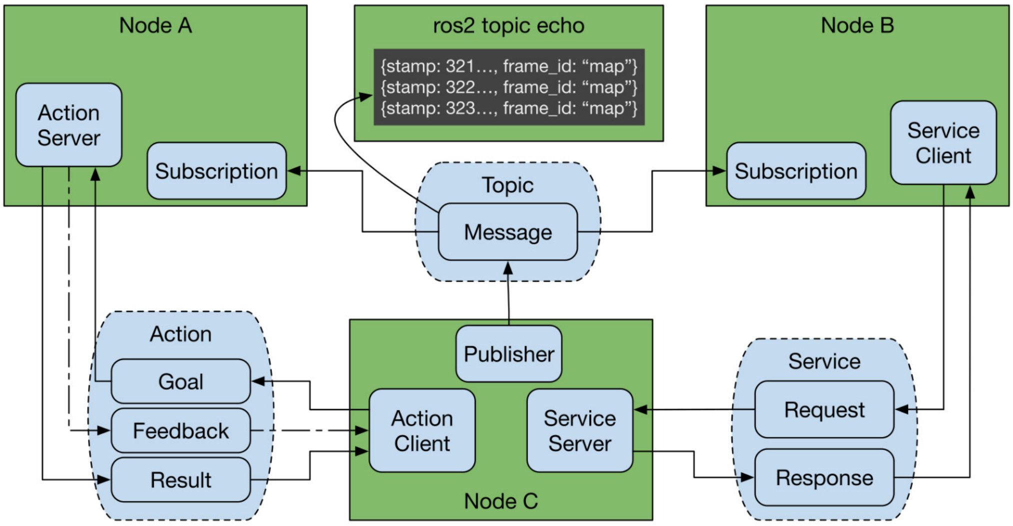
The second type of communication are *services*. These create one-to-one communication between two nodes. One node must provide a service by using a service server. This service can then be called by a client with a *request* message. When this service is processed, a *response* message is returned to the client. Services are typically used to request data from another node or to invoke short-running operations, e.g., switch on a LED on the robot.

The third and most complicated type are *actions*. These also create one-to-one communication between two nodes. One is the client, and the other is the server. The client sends a *goal* message to the server to invoke an action. While the action is executed, the server can send *feedback* messages. During the execution, the client also might cancel an action. When the action is finished (successful or not), the server returns a *result* message. Actions are typically used to invoke long-running operations, e.g., moving the robot to a certain location.

Additionally, ROS comes with multiple tools that simplify the programming and usage of robots. The most important ones are the 3D visualization tool *RViz*, the plugin-based user interface *rqt*, and the plotting tool *PlotJuggler*. These tools can display debug information or send commands to the robot. Additionally, ROS includes management for a node's parameters, including the ability to change these during runtime.

2.4 Splines

A spline $S^n(x)$ is a mathematical function defined between k_0 and k_{m+1} that is piecewise-defined through m polynomials $P^n(x)$ of a given order n [BM08, p. 15f]. The points where two polynomial sections meet $(k_1, ..., k_m)$ and where the outer sections end (k_0, k_{m+1}) are called knots. In contrast to direct polynomial approximation, the decompo-



```
{stamp: 321..., frame_id: "map"}
{stamp: 322..., frame_id: "map"}
{stamp: 323..., frame_id: "map"}
```

Figure 2.7: The three communication methods between nodes: topics, services, and actions. [MFG+22]

sition into multiple polynomials avoids the Runge's phenomenon [Run01].

$$S^n(x) : [k_0, k_{m+1}] \rightarrow \mathbb{R} \tag{2.1}$$

$$S^n(x) = \begin{cases} P_0^n(x - k_0), x \in [k_0, k_1] \\ P_1^n(x - k_1), x \in [k_1, k_2] \\ \ldots \\ P_m^n(x - k_m), x \in [k_m, k_{m+1}] \end{cases} \tag{2.2}$$

$$P_i^n(x) : [k_i, k_{i+1}] \rightarrow \mathbb{R} \tag{2.3}$$

$$P^n(x) = \sum_{j=1}^{n} \mathrm{a}_j x^j = \mathrm{a}_0 + \mathrm{a}_1 x + \mathrm{a}_2 x^2 + \ldots + \mathrm{a}_n x^n \tag{2.4}$$

Splines have various applications, but in the following, we will concentrate on their application in robotics. A typical use case is the description of a robot's end effector pose in Cartesian space over time (t). Here, each knot (k) describes at least the position (p) of the end effector at a given time point but can also describe the derivatives velocity (v), acceleration (a), and jerk (j), if a high order spline is used. Analogically, it is also possible to describe an end effector's orientation (o) over time. For simplicity, we will only discuss the case of the position in the following. Using splines to describe a movement is convenient for the programmer as only specific key points need to be specified. For example, for a grasping skill the start pose of the gripper, the pre-grasp pose, and the final grasp pose need to be defined. These positions are typically known or are computed by another part of the software, e.g., a grasp pose generator. The rest of the trajectory can then be interpolated through the spline.

In the following, we will assume that the spline represents a position over time.

$$x = t \tag{2.5}$$
$$S(t) = p_t \tag{2.6}$$
$$\dot{S}(t) = v_t \tag{2.7}$$
$$\ddot{S}(t) = a_t \tag{2.8}$$
$$\dddot{S}(t) = j_t \tag{2.9}$$

In some applications, sudden changes of discontinuous accelerations are undesirable as these can destabilize the robot. These discontinuities are often encountered when the trajectory is optimized for time, thus leading to high accelerations [BM08, p.26]. This issue can be prevented by defining the position (p), velocity (v), and acceleration (a) of all knots. Thereby, a continuous acceleration profile is achieved between two polynomials. Therefore, each knot is defined as a triple.

$$k_i = (p_i, v_i, a_i) \tag{2.10}$$

Creating a polynomial between two knots (k_i, k_{i+1}) requires fulfilling six conditions for the polynomial.

$$\begin{aligned} &P(t_i) = p_i,\ P(t_{i+1}) = p_{i+1} \\ &\dot{P}(t_i) = v_i,\ \dot{P}(t_{i+1}) = v_{i+1} \\ &\ddot{P}(t_i) = a_i,\ \ddot{P}(t_{i+1}) = a_{i+1} \end{aligned} \tag{2.11}$$

Here, p_i and p_{i+1} describe the positions, v_i and v_{i+i} the velocities and a_i and a_{i+1} the accelerations at t_i and t_{i+1} respectively. Therefore, it is necessary to use a polynomial of the fifth order. The resulting splines are also called quintic splines, hinting at their polynomial order. An exemplary quintic spline is shown in Figure 2.8.

The polynomials of a quintic spline can therefore be defined as

$$\begin{aligned} P^5(t) =& c_0 + c_1(t_{i+1} - t_i) + c_2(t_{i+1} - t_i)^2 + \\ & c_3(t_{i+1} - t_i)^3 + c_4(t_{i+1} - t_i)^4 + c_5(t_{i+1} - t_i)^5 \end{aligned} \tag{2.12}$$

with

$$c_0 = p_i \tag{2.13}$$
$$c_1 = v_i \tag{2.14}$$
$$c_2 = \frac{1}{2}a_i \tag{2.15}$$
$$c_3 = \frac{1}{2T^3}[20h - (8v_{i+1} + 12v_i)T - (3a_i - a_{i+1})T^2] \tag{2.16}$$
$$c_4 = \frac{1}{2T^4}[-30h + (14v_{i+1} + 16v_i)T + (3a_i - 2a_{i+1})T^2] \tag{2.17}$$
$$c_5 = \frac{1}{2T^5}[12h - 6(v_{i+1} + v_i)T + (a_{i+1} - a_i)T^2] \tag{2.18}$$

where $h = p_{i+1} - p_i$ describes the displacement and $T = t_{i+1} - t_i$ the duration [BM08, p. 26ff].

The jerk of a quintic spline is still discontinuous. This can be solved by using splines with polynomials of order seven which also define the jerk at the boundaries.

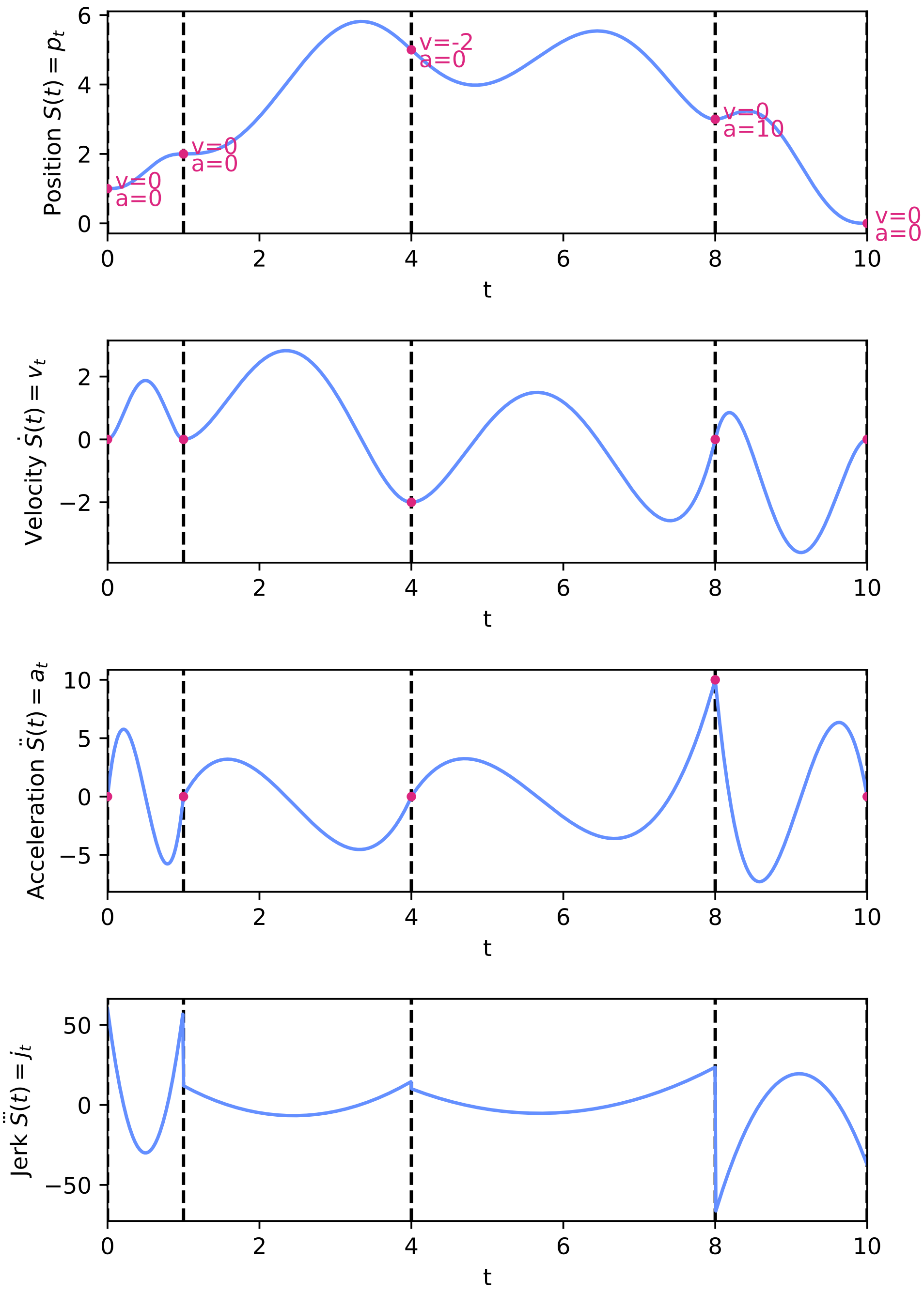


Figure 2.8: Exemplary quintic spline (**blue**) showing a position over time with the derivatives for velocity, acceleration, and jerk. The dashed black lines show the partitioning in different polynomials, and the knots with the corresponding bounding conditions are displayed in **red**.

Table 2.3: Allowed and Forbidden Sensors in the HSL.

	Sensor	Human Sense
Allowed	(Stereo-)Camera	Vision
	Microphone	Hearing
	IMUs	Balance
	Force sensors	Touch
	Rotary joint encoders	Proprioception
	Voltage/current sensors	Proprioception
Forbidden	LIDAR	-
	RADAR	-
	RGB-D Camera	-
	Ultrasonic	-
	Magnetometer	-

2.5 RoboCup Humanoid League

In 1997 the computer *Deep Blue* beat the then-reigning chess world champion Garry Kasparov [CHJH02]. Before, chess was considered difficult to solve, and many chess grandmasters doubted that a computer could win against them [New12, p.1f]. During the development of chess computers from the 1950s to the 1990s, important discoveries were made in the field of artificial intelligence (AI) [New12, p.3f] and search algorithms [KAK$^+$97]. However, after Deep Blue, chess could be regarded as a solved problem. Therefore, the research community looked for a new standard challenge for artificial intelligence. With this in mind, the RoboCup was founded [KAK$^+$97]. Its goal, very analogically to DeepBlue, is the following: "*By mid-21st century, a team of fully autonomous humanoid robot soccer players shall win the soccer game, comply with the official rule of the FIFA, against the winner of the most recent World Cup.*" [BDF$^+$02]

Nowadays, there are multiple leagues in RoboCup, and not all of them are soccer-related [AvS20]. The @Home league, for example, aims to foster the development of service robots by creating competition scenarios in typical household environments. The soccer section is again split into different leagues focusing on various issues. The Small Size League uses small, wheeled robots to reduce hardware costs and circumvent the complicated problem of bipedal walking. The game is played with eight vs. eight robots using a small ball [AvS20]. An external ceiling camera system provides the poses of all objects to the robots. Similarly, the Middle Size League also uses wheeled robots, but larger ones that use a standard soccer ball and need to do onboard computation and vision. The Standard Platform League (SPL) uses the humanoid NAO robot from Aldebaran [GHB$^+$09] with a small soccer ball. Since no changes to the hardware are possible, the teams can focus on the software. Furthermore, the league makes different software approaches comparable since there are no differences in hardware. There are also entirely simulated leagues in a 2D and 3D simulation environment. These focus on strategies and multi-agent communication without accurate physical simulation. Additionally, the agents have physical abilities that are currently not reached by robot hardware. This allows research without the limitations of current hardware but is not feasible to be directly transferred to a real robot.

The most essential league is the Humanoid Soccer League (HSL) which tries to solve the primary goal of RoboCup. To achieve this, the rules are gradually changed to get closer to the ordinary FIFA rules. The hardware of the robots can be changed by the teams, which is necessary since development in hardware is also necessary to play against humans. However, there are certain limitations. The most important one is that only human-like sensors are allowed to be used to make the game fair for the

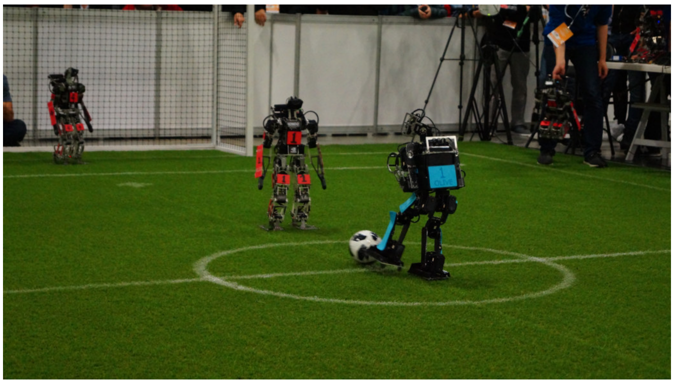


Figure 2.9: Photo of a RoboCup game from 2019 between Rhoban (**blue**) and CITBrains (**red**). Courtesy of Florian Vahl.

human players. Table 2.3 shows a list of allowed sensors and their pendant in humans. Furthermore, the proportions and the weight distribution of the robot are limited to human-like proportions, e.g., the leg length needs to be between 35% and 70% of the robot's height. Based on the height of the robot, the team can participate in the Kid- or Adult-Size League. Since this thesis is related to the Kid-Size League, we are referencing this if we mention the HSL.

A game in the HSL is played over two half times with 10 min each. The field consists of artificial grass with sprayed on white lines and two white goals (see Figure 2.9). The ball is a FIFA size one ball with any markings that are at least 50% white. The robots need to act fully autonomously and can communicate with each other via a wireless network. They also receive the current game state, e.g., the current score, via a wireless network from a dedicated computer controlled by a referee. Each team provides a robot handler that can take out robots on request by the team or the referee, e.g. if the robot performed a rule violation. The robot will get a time penalty, and the team can repair the hardware or change the software. Other manual control of the robot is not allowed. Therefore, it is crucial that the robot can get up by itself after a fall, as it would otherwise, get a time penalty. Additionally, the hardware needs to be robust to prevent timeouts for repairs. The same is true for the robot's battery which should last at least one half-time (currently 10 min) as otherwise timeouts need to be taken to change them. The length of the half-time will be increased incrementally in the future to get closer to the real FIFA rules [31]. Therefore, making the robot as energy-efficient as possible is necessary.

In 2020 and 2021, it was impossible to hold regular RoboCup competitions due to the worldwide COVID-19 pandemic. Therefore, the Humanoid League Virtual Season (HLVS) (see Figure 2.10), a simulated pendant for the HSL, was created using the Webots simulator to allow virtual competitions. The simulated environment tries to resemble the real one as closely as possible, i.e., the ground has similar properties as artificial grass, and the background and lighting conditions change each match. Each team needs to provide a model representing their robot as closely as possible. This includes, for example, sensor noise and joint backlash. Since these robot models are made open source, it is possible to evaluate approaches on different platforms with a comparable small overhead. Since this simulated competition allows a more accessible entry for new teams and a simple way to evaluate the software, competitions are planned to continue in the following years.

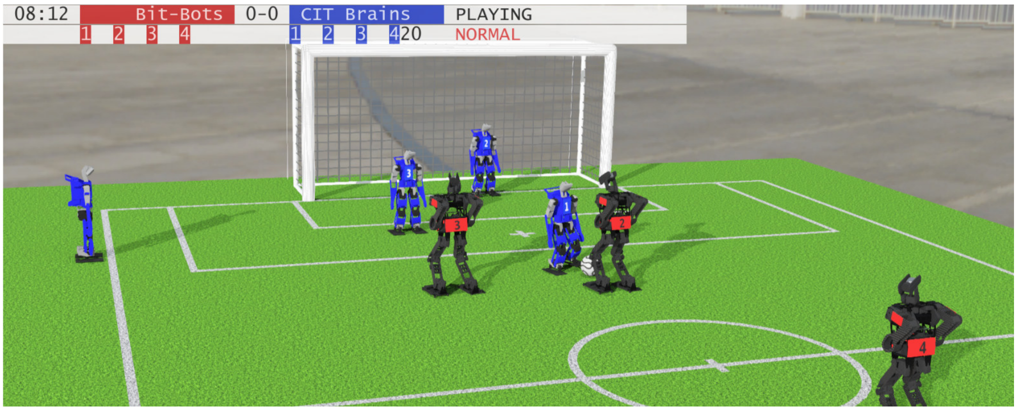


Figure 2.10: Screenshot of the HLVS 21/22 final between the Hamburg Bit-Bots (**red**) and CITBrains (**blue**). The overlay on the top shows the game's current state, including the time, goals, and time penalties for players.

Chapter 3

Robot Platform

For this thesis, a new robot platform was created, as previous ones were not applicable. They were either too small, had too low control loop rates on the hardware bus (Section 3.4.2 discusses the importance of this), lacked FPSs, were not robust against falls, or could not stand up by themselves.

Therefore, a new platform was developed upon the basis of the Nimbro-OP [SSS$^+$12], and the further modifications from Team WF Wolves [BDG$^+$17]. It was chosen since it provides a good height and possibilities to integrate the missing components, such as foot pressure sensors, while still being low cost. The new platform's goal was to be usable beyond this thesis in other research projects and for participation in the HSL. For participation in RoboCup games, but also to allow training and evaluation on the robot, it needs to be robust against falls and be able to stand up autonomously. In regards to the available platform, its processing power and hardware control loop rate should increase to allow faster reaction times, which are essential to stabilize motions. Additionally, some size requirements have to be met to allow participation in the HSL (see Section 2.5).

This platform will be described and evaluated in the following chapter. First, the related works are presented in Section 3.1. Then, a general overview of the Wolfgang-OP will be provided in Section 3.2. This is followed by detailed descriptions of the mechanics (Section 3.3) and electronics (Section 3.4). The kinematic and simulation model creation is discussed in Section 3.5. Afterward, the platform is evaluated in Section 3.6. Finally, a summary is given in Section 3.7.

3.1 Related Work

The following sections give an overview of the current state of the art in the related research field. First, other existing humanoid robot platforms are discussed, with a specific focus on the HSL and low-cost humanoids (Section 3.1.1). Then, other works concerning the robustness of humanoid robots are presented (Section 3.1.2). This is followed by other usages of spring-based actuators in robots and exoskeletons (Section 3.1.3). Then, different FPS are presented (Section 3.1.4). Finally, the existing servo and sensor interface solutions are discussed (Section 3.1.5).

3.1.1 Humanoid Robot Platforms

Humanoid robots have been of interest in research for a long time. The first humanoid robot was the WABOT-1 [Kat73]. Its development was finished in 1973. It already had basic walking abilities but was naturally very limited in its capabilities. Public

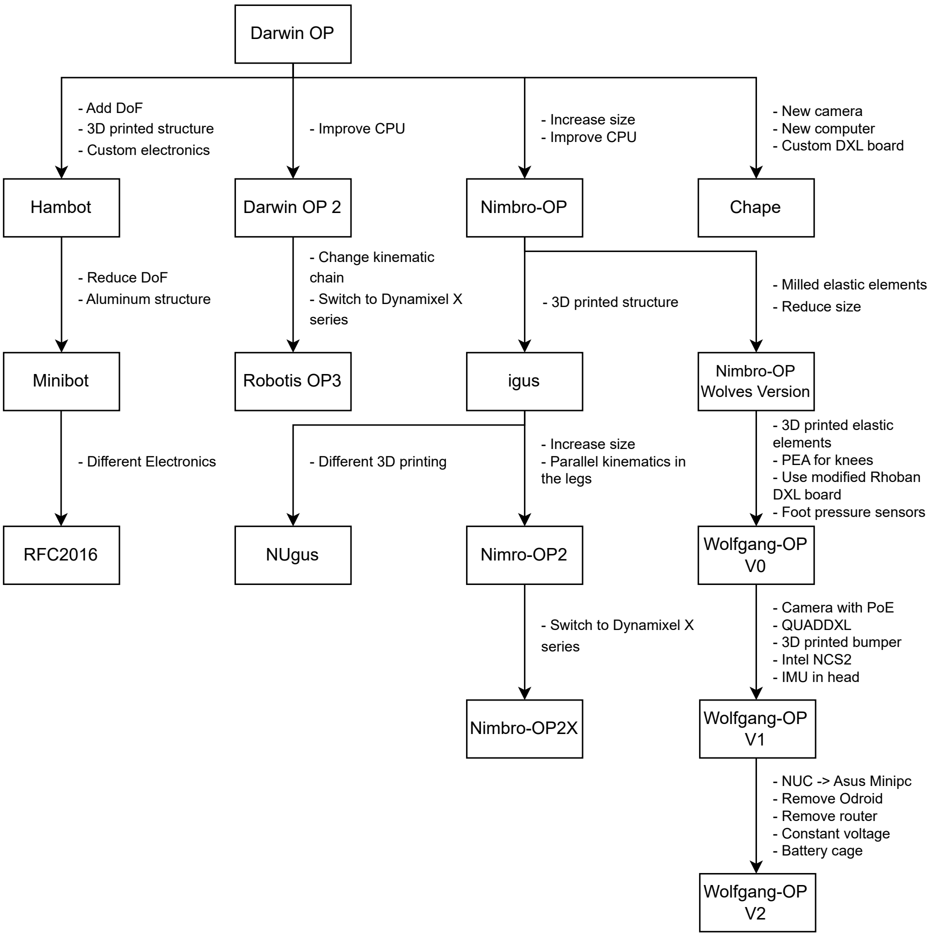


Figure 3.1: Family tree showing how different robot platforms that were developed based on the Darwin-OP. See Table 3.1 for more details on the platforms.

attention to this research field was raised when Honda first presented their ASIMO platform [SGV18] in 2000. It was not only able to walk but also to run. Still, it was only able to perform these skills in controlled environments. In 2002, the HSL was started [17] and proposed a new challenge to humanoids within a partially controlled environment. This led to an increasing development of new humanoid robots that had to be reliable and robust. Additionally, the switch from the previously used four-legged Sony AIBOs [CMQ07] to the humanoid Aldebaran NAOs [GHB+09] in the RoboCup SPL pushed the development in this area. Although, in the latter case, the research was focused on the software since no changes to the hardware are allowed (see 2.5).

In 2011, the Darwin-OP [HTA+11] was released by Robotis. In contrast to the NAO, it fulfilled all rule restrictions of the HSL and was, therefore, a good entry for new teams. Additionally, it was built modularly by using capsuled servo motors for all joints, allowing simple hardware changes. Therefore, multiple new platforms for the HSL were developed on the basis of the Darwin-OP (see Figure 3.1). The manufacturer Robotis created two successor versions. First, the Darwin-OP2 [26], which is very similar and only features an updated main computer. Later, the Robotis-OP3 [26], which is slightly increased in height and uses the never X-Series of the DXL servos (see Section 2.2.1). The main advantage of the Darwin-OP series and the NAO robot, in contrast to established research platforms like the ASIMO, was their price. This low-cost aspect allowed groups to purchase multiple robots to form a team. While a lot of research has been conducted on them, they are limited in their sensor modalities, computational power, and the achieved actuator torques. Additionally, their small size leads to issues with walking on uneven ground, i.e., artificial grass [AvS20], and makes the comparison to human capabilities difficult. Therefore, many researchers have developed these platforms further.

One of its successors was the Nimbro-OP [SSS$^+$12], which kept a similar kinematic structure and electronics but increased the robot's height to 0.95 m. To achieve this, the servo motors were replaced with stronger types from the same DXL MX series (see Section 2.2.1), and parts of the mechanical structure were made out of carbon fiber, which reduces the weight of the robot. Based on this, the mostly 3D printed igus Humanoid Open Platform [AFSB15] was developed. This robot still had similar kinematics and electronics, but the 3D printed manufacturing process allowed "a lightweight and appealing design" [AFSB15]. Based on this, two additional successors were developed. The Nimbro-OP2 [FAFB17] increased the robot's size further to 1.35 m and changed to parallel kinematics in the legs. The latest member of the series, the Nimbro-OP2X [FFB$^+$18], kept similar kinematics but replaced all DXL MX servos with the never X series. Additionally, the computational power was improved by adding a graphics processing unit (GPU) and previously milled parts replaced by 3D printing.

The team NUbots developed the NUgus, a different successor of the igus-OP [ABC$^+$17]. It is similar, but the 3D-printed parts were split into smaller parts to allow easier manufacturing. Additionally, minor changes were made to have better access to the robot's cables [9]. Further robot platforms, i.e., the Bez [25], KAREN [4], and the Chape [25] robot, have been developed on the basis of the Darwin-OP by other HSL teams. The main changes are a slight increase in the robot size and an update of the computing unit.

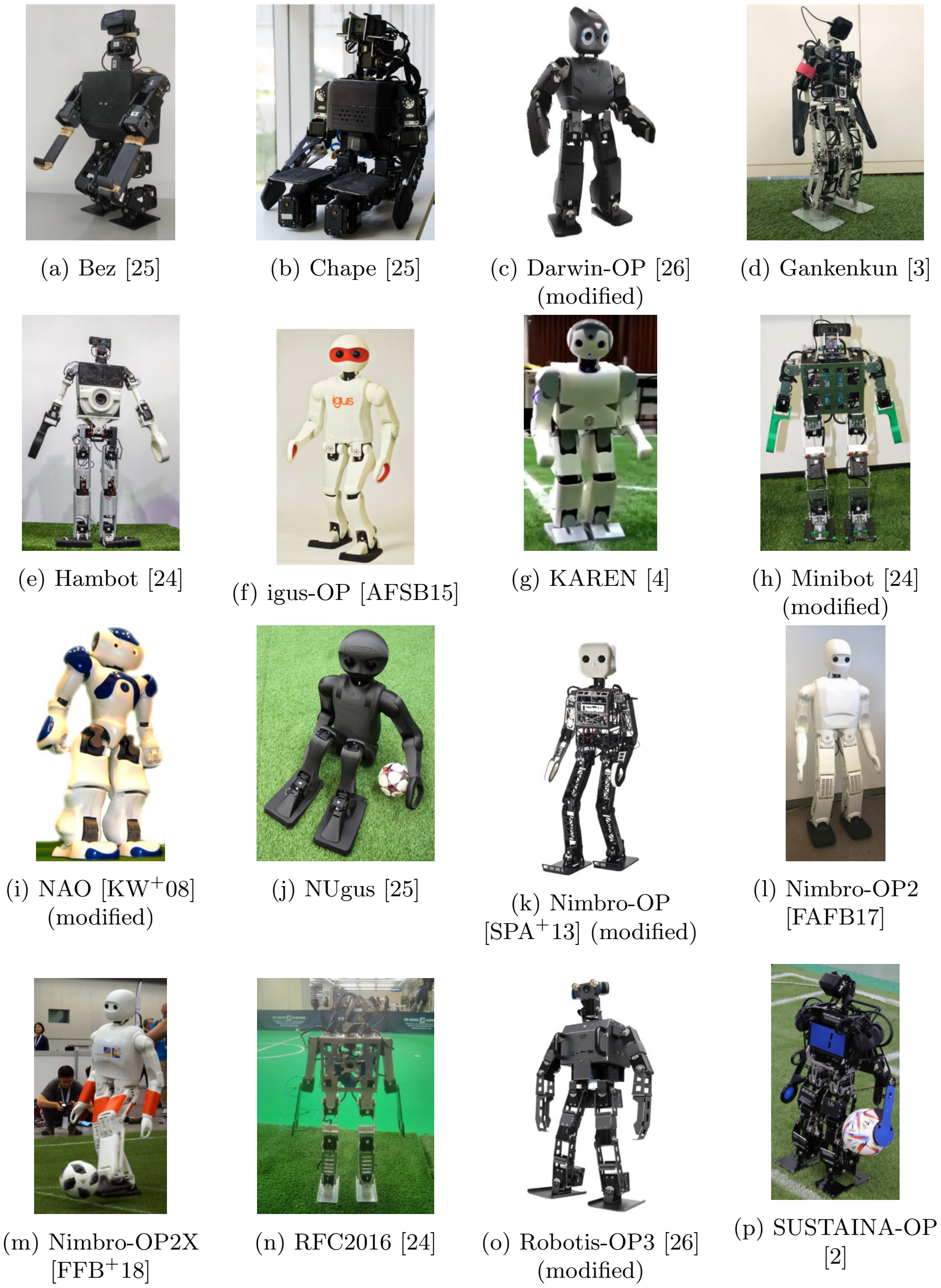
Still, there are other influential and successful robots in the HSL that are no successors of the Darwin-OP. The Sigmaban series [GHLZ$^+$19] of the multiple world champion team Rhoban has been adopted by other teams. It features similar kinematics as the Darwin-OP but uses stronger DXL servos and more rigid mechanical connections. The GankenKun [3] and the successor SUSTAINA-OP [22][Kea22] from team CITBrains is one of the few platforms that feature parallel kinematics in the legs and does not use DXL servos but Kondo servos [5].

The Hamburg Bit-Bots, including myself, developed the 0.87 m high Hambot-OP [BRW15] based on the Darwin-OP. This platform used 3D printing for manufacturing to keep the costs low and to achieve complicated mechanical parts. It changed the typically Darwin-OP-like kinematic structure by adding two DoF in the robot's waist and by adding one DoF in each foot for toe movement. Unfortunately, this introduction of additional joints, in combination with the increased size and the high backlash of the DXL MX servos, led to imprecise movements and instability. Furthermore, the weight of the additional servos was problematic due to the weak knee motors. Still, some effective concepts, e.g., allowing a fast battery change by putting it into a sliding 3D printed battery cage, were later used for the Wolfgang-OP. In the successor Minibot [Aea17], the number of joints and the height were again decreased to counter the stability issues. The mechanical structure relied on aluminum sheet metal which was difficult to produce and sometimes bent when the robot fell. The team FUmanoids used the Minibot as a basis to develop a similar robot platform [Fre16]. Due to internal issues, the team was dissolved, and a new team was founded with the name 01.RFC Berlin. This team participated with a model of this robot, called RFC2016, in the HLVS.

Outside of the influence sphere of RoboCup, further humanoid platforms have been

Table 3.1: Comparison of Different Low-Cost Humanoid Robot Platforms.

Name	Team/Company	Year	Size [m]	Weight [kg]	DoF	Parallel Kinem.	Uses DXL	Bus Interface	FPS	Cleats	Estimated Cost	Reference
Bez	UTRA	2020	0.50	2.4	18	✗	✓	U2D2	✗	✗	?	[25]
Chape	ITAndroids	2017	0.53	3.1	20	✗	✓	Custom	✗	✓	?	[25]
Darwin-OP	Robotis	2010	0.46	2.9	20	✗	✓	CM730	✗	✗	~10,000€	[HTA^{+}11]
Gankenkun	CITBrains	2017	0.64	4.6	19	✓	✗	Custom	✗	✗	?	[3]
Hambot	Hamburg Bit-Bots	2015	0.87	6.0	24	✗	✓	Custom	✗	✗	~7,500€	[BRW15]
igus-OP	NimbRo/igus	2015	0.90	6.6	20	✗	✓	CM730	✗	✗	~20,000€	[AFSB15]
KAREN	MRL-HSL	2020	0.73	?	20	✗	✓	?	✗	✓	?	[4]
Minibot	Hamburg Bit-Bots	2016	0.70	4.0	20	✗	✓	CM730	✗	✓	?	[Aea17]
NAO	Aldebaran	2006	0.57	4.5	21	✗	✗	Custom	✗	✗	~7,000€	[KW^{+}08]
NUgus	NUbots	2018	0.90	7.5	20	✗	✓	CM730	✗	✗	?	[BHMC18]
Nimbro-OP	NimbRo	2013	0.95	6.6	20	✗	✓	CM730	✗	✗	?	[SPA^{+}13]
Nimbro-OP2	NimbRo	2017	1.35	17.5	20	✓	✓	CM730	✗	✗	~40,000€	[FAFB17]
Nimbro-OP2X	NimbRo	2018	1.35	19.0	20	✓	✓	CM740	✗	✗	~33,000€	[FFB^{+}18]
RFC2016	01.RFC Berlin	2016	0.73	4.5	20	✗	✓	Custom	✗	✗	?	[Fre16]
Robotis-OP3	Robotis	2017	0.51	3.5	20	✗	✓	OpenCR	✗	✗	~11,000€	[26]
Sigmaban+	Rhoban	2019	0.7	5.5	20	✗	✓	R-DXL	✓	✓	?	[25]
SUSTAINA-OP	CITBrains	2022	0.65	5.2	19	✓	✗	QUADDXL	✗	✓	?	[22]
Wolfgang-OP V2	Hamburg Bit-Bots	2022	0.80	7.5	20	✗	✓	QUADDXL	✓	✓	~11,000€	[BGVZ21]



(a) Bez [25]

(b) Chape [25]

(c) Darwin-OP [26] (modified)

(d) Gankenkun [3]

(e) Hambot [24]

(f) igus-OP [AFSB15]

(g) KAREN [4]

(h) Minibot [24] (modified)

(i) NAO [KW+08] (modified)

(j) NUgus [25]

(k) Nimbro-OP [SPA+13] (modified)

(l) Nimbro-OP2 [FAFB17]

(m) Nimbro-OP2X [FFB+18]

(n) RFC2016 [24]

(o) Robotis-OP3 [26] (modified)

(p) SUSTAINA-OP [2]

Figure 3.2: Pictures of the humanoid robot platforms listed in Table 3.1.

developed, but typically at higher prices. The child-size iCub [PMN+12] has 53 DoF, but it costs approximately 250,000€. This platform aims to have a robot that can interact with the world similarly to a human child to get insights into the fields of artificial intelligence and human cognition.

Also commercially available are different adult-size robots, such as the Robotis' THORMANG3 [26] and the PAL Robotics' TALOS [SFB+17]. Other humanoids include the Boston Dynamics Atlas [NSP19b], and the Asimov [Shi19]. Further information on humanoids, especially full-size ones, can be found in these survey papers [SJAB19] [SF19] [FB21].

Some platforms are targeted at optimal performance without regard to the costs, e.g., the iCub with its many joints in the fingers. Others are specially designed to be low-cost and try to achieve a sufficient performance with a limited budget, e.g., the above-mentioned Darwin-OP and NAO. From a researcher's perspective, the costs may not seem like a relevant feature of the robot since it should not be commercial. Still, spending more money on the hardware results in less funding for other research. Furthermore, widespread usage of robots in our everyday lives can only be reached if their price is adequate. For this, the problems of controlling inaccurate hardware need to be solved.

3.1.2 Robustness

Another important aspect, besides the costs, is the robustness of a platform against falls. These can create high-impact forces on the robot, especially for larger humanoids. Different approaches have been investigated to ease these impacts and thereby prevent damage to the robot. They can be grouped into active and passive approaches. The latter modifies the robot's mechanical structure, so no additional software is necessary. This can either be done with rigid structures [KKS+17] or elastic elements, e.g., a piano wire [GHLZ+19]. To actively prevent damage from falls, the robot can employ airbags [KCB+16] when a fall is detected or assume a pose that minimizes the damage [FKK+02]. Section 4.1.3 gives a detailed description of active fall management.

3.1.3 Spring Based Actuators

The typical energy consumption of a bipedal robot is currently much higher than the one of humans [SWC+14] (see also Figure 3.3). Different spring-based solutions have been investigated to improve this efficiency in robots. These springs can be part of a tendon-driven system [RMM+11]. They can also be placed parallel to a servo-based joint drive. One example is the Poppy robot [LRO13] where linear springs were used (see Figure 3.4). They are placed to support walking motions (other motions, such as standing or getting up, were not considered). The spring straightens the leg during its support phase and flexes it during its swing phase, thus reducing the power needed for the knee joint. Another example is the WANDERER robot which uses linear compression springs in the hip roll joint [HMS+20]. The current world record holder for the longest traveled distance with one battery charge is the Xingzhe robot [LLTZ16], which also applies springs to improve energy recuperation. Additionally, the usage of torsion springs was investigated in exoskeletons, e.g., to optimize energy while cycling [CGKK17] or in industrial applications [KNY20].

Using springs-based actuators not only reduces the overall energy consumption but may also reduce the peak torques that act on certain motors. This can lead to faster motions as the speed of a DC servo motor is inversely proportional to the torque. It may also allow the usage of smaller motors in those joints, thus reducing the weight and costs of the robot.

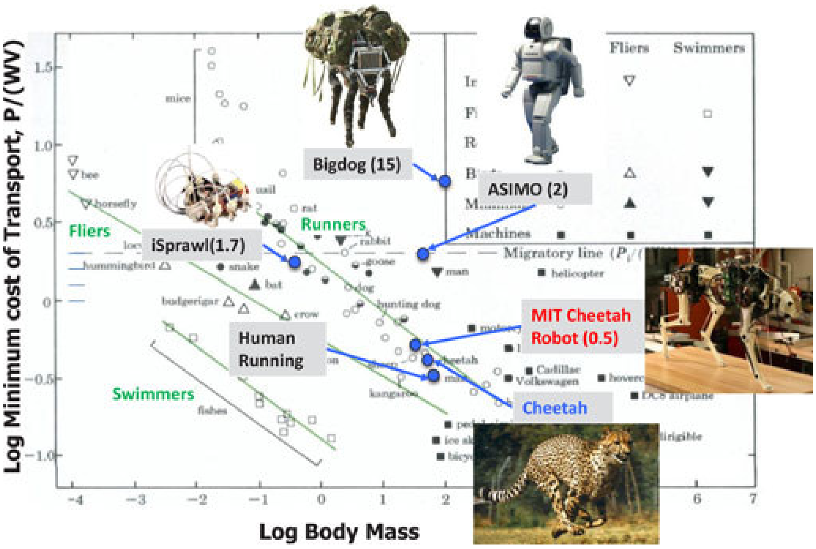


Figure 3.3: Relationship between mass and cost of transportation for different animals and robots. [SWC+14]

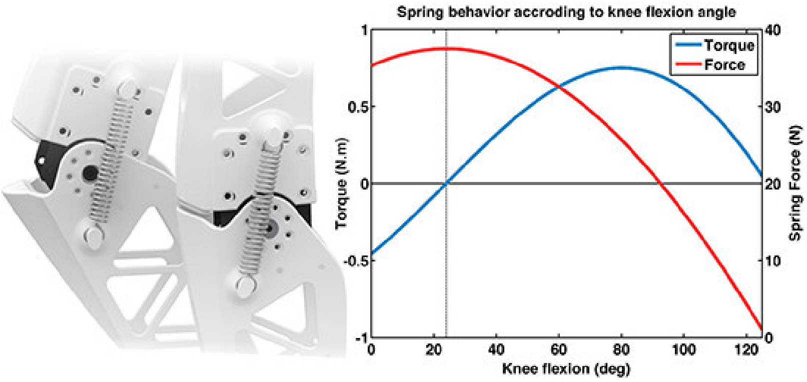


Figure 3.4: Combination of a linear spring with a servo-based knee joint in the Poppy robot (**left**). Torque and force for different knee angles (**right**). [LRO13]

3.1.4 Low-cost Foot Pressure Sensors

Naturally, sensing the forces acting on the foot of a bipedal robot is interesting as it may provide a way to stabilize it or perform its movements more energy efficiently. One typical approach is installing a 6-axis-force-torque sensor in the robot's ankle of the robot [KHHY14, pp. 79-80]. However, these sensors are expensive, each costing in the range of a complete low-cost robot. Therefore, they will not be discussed further in detail.

Multiple low-cost solutions have been proposed to sense the forces between the ground and the robot foot. One of the most widespread is to use force sensitive resistors (FSRs) on the sole of the robot, e.g., in the Poppy robot [LRO13] and as an upgrade to the Darwin-OP [26]. These sensors measure a change in resistance proportional to the applied orthogonal force on the sensor. They are thin and cheap but not precise, and their measurements drift over time [HW06].

The most widespread approach in the HSL is using four load cells with strain gauges

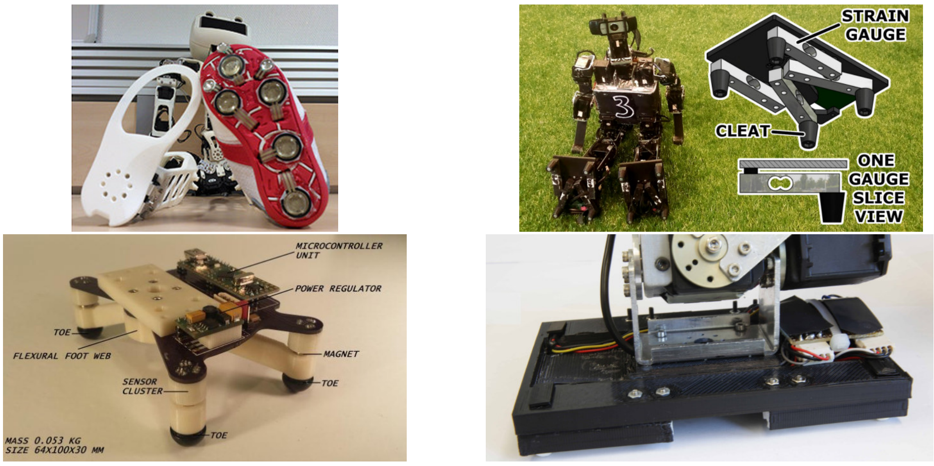


Figure 3.5: Example images for FPSs based on FSR (**top left** [LRO13]), strain gauges (**top right** [RPH$^+$15]), hall sensors (**bottom left** [GVK15]) and optical proximity sensors (**bottom right** [WHFZ18]).

placed at the corners of the foot. An orthogonal force leads to a bending of the load cell and, thus, to elongating the wire in the strain gauge. Thus the resistance increases and the applied force can be measured. This is typically done by utilizing a Wheatstone bridge to sense the small resistance difference [Urb16, pp. 212-215] (these are also often directly integrated into the load cell). Team Rhoban developed such a sensor that connects to the DXL bus system [RPH$^+$15]. Other teams then adapted similar sensors.

While using changes in resistance to measure the applied force is most common, different approaches have been investigated. Gomez et al. [GVK15] used a combination of magnets and hall sensors to measure the deformation and, thus, the force acting on it. They built a three-toe foot which achieved comparable results to the FSR based solution of the Darwin-OP but with reduced costs. Wasserfall et al. [WHFZ18] used a 3D printing approach that used an optical proximity sensor to measure the deformation of a 3D printed beam. They built feet with one of these sensors at each corner and tested it successfully with the Minibot. Examples of all approaches can be seen in Figure 3.5.

3.1.5 Servo and Sensor Interfaces

The DXL servos are widespread in robotics and, thus, also the corresponding protocol. It is prevalent in the HSL where 32 of 34 teams used it in the 2019 championship and 13 out of 14 in the 2022 championship[1]. Additionally, they are common in other RoboCup Leagues, e.g., the Rescue League [18]. Table 3.1 shows a list of humanoid robots that use this protocol (see also Figure 3.2 for photos of the corresponding platforms). Additionally, many other non-humanoid robots use this protocol. Robotis, the manufacturer of the DXL servos, offers multiple platforms. These range from the wheeled mobile robot TurtleBot3 [GA17] to the robotic arm OpenManipulator-X [12]. Furthermore, many research groups developed robots based on this.

[1] Data from team description papers available at https://www.robocuphumanoid.org/

Name	Microchip	Max. Bus Speed [MBaud]	No. Bus	Prot. 2	Cost
CM730[1]	FT232R+STM32F103	4.5	1	no	
OpenCM9.04	STM32F103	2.5 [2]	1	yes	50$
OpenCR	STM32F746	10	1	yes	180$
Arbotix Pro	FT232R+STM32F103	4.5	1	no	150$
Rhoban DXL	STM32F103	4.5 (2.25)[3]	3	yes[4]	∼20$
USB2DXL	FT232R	3	1	yes	50$
U2D2	FT232H	6	1	yes	50$
QUADDXL	FT4232H	12	4	yes	∼40$

Table 3.2: Comparison of the different available interfacing solutions for the DXL bus. The maximal bus speed is the theoretical limit of the board, not regarding limitations from the microprocessor speed. [1] Discontinued; [2] Limited by transceiver on extension board; [3] Bus 1 can operate at 4.5 Mbps, bus 2 and 3 only at 2.25 Mbps; [4] Not supported by original firmware [BGZ19]

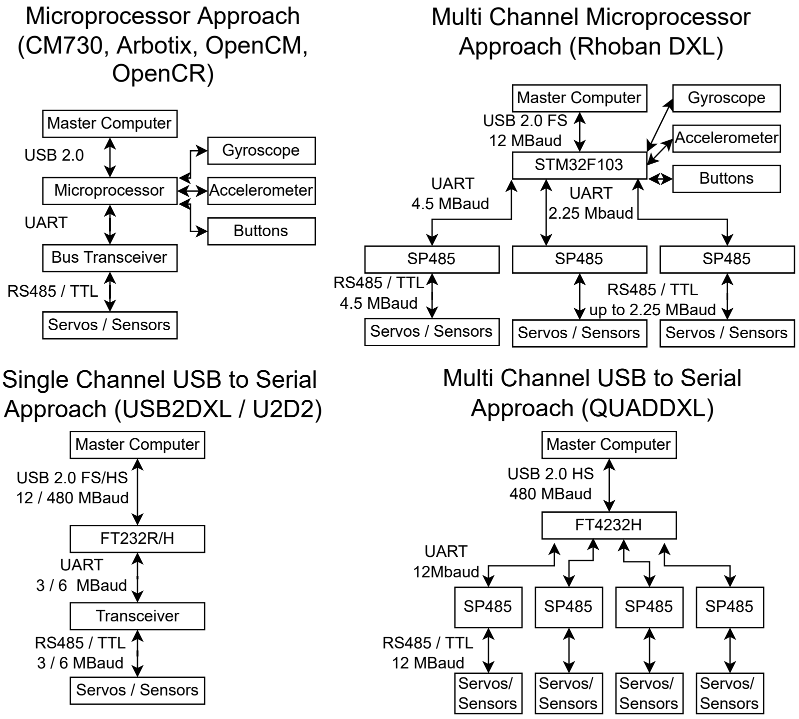


Figure 3.6: Overview of different interfacing solutions between the main computer and the peripheral devices. They can be classified based on the of buses (**horizontal**) and the used chip type (**vertical**). Our approach is the first combining multiple channels with a UTS approach (**bottom right**). [BGZ19]

Different solutions exist to connect these to the main processing unit. Some controller boards are commercially available from Robotis and other manufacturers. Additionally, many custom solutions were created by researchers for their platforms. An overview of the existing solutions can be seen in Table 3.2.

A common numerator is that these all connect to the main processing unit via Universal Serial Bus (USB). Otherwise, they can be grouped into two main approaches. The first approach uses an USB to serial (UTS) converter chip that directly translates the USB signal to UART. A transceiver then converts this to a RS-485 (or TTL) signal. Robotis offers two boards that work on this principle, the *USB2DXL* and *U2D2*. A

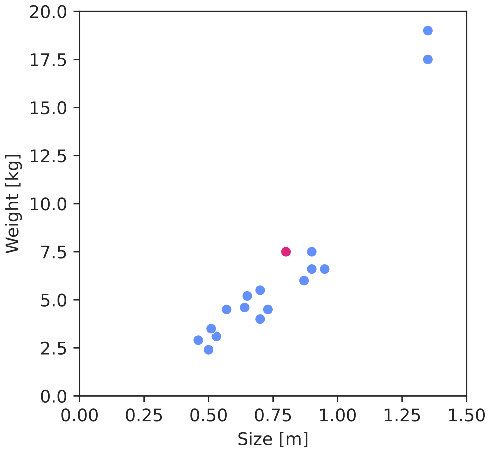


Figure 3.7: Relationship between robot size and weight of all robots in Table 3.1. The Wolfgang-OP is marked in red. A similar relation can be observed for most robots. Outliers are the Nimbro-OP2 and Nimbro-OP2X on the top right.

downside of this approach is that it is impossible to directly connect further devices, i.e., sensors, to the chip.

The second approach uses a standard microcontroller as the main component. It is directly connected to the main processing unit with USB and to different sensors, typically an IMU, via UART, SPI or I2C. The packages received via USB are first parsed and then either directly forwarded to the bus if they are destined for one of the servos or processed by the microcontroller itself if they request data from the directly connected sensors. Figure 3.6 shows a block diagram of the different approaches.

Independent of the used approach, almost all use just a single bus. The only exception is the *Rhoban DXL Board* [ADF$^+$16], which supports three buses. This allows parallelized communication and thereby, theoretically, decreases the time necessary for one control loop cycle. Generally, this would require more cables than a single bus system, e.g., if this approach is applied to a robotic arm. However, a humanoid robot's structure already requires separated cable chains for each extremity. Therefore, having up to five separate buses does not influence the cabling of the robot. The Hambot was already supposed to use a similar three-bus approach with a microcontroller [BRW15], but the development of the electronics was never fully completed. The performances of the different approaches are further discussed in Section 3.4.2.

3.2 Overview

The Wolfgang-OP is an 80 cm height robot that weighs 7.5 kg (see Table 3.3), which is a typical weight for a robot of this size (see Figure 3.7). It features a Darwin-OP-like 20 DoF layout (see Figure 3.8). Three joints (head pitch and both shoulder roll joints) feature a compliant element to protect the gears during falls. The knee joints are PEAs to reduce the peak torques and to save energy. It has two IMUs and two FPSs. Still, the cost of the robot was kept as low as possible and is estimated at ca. 11.000 € for the material, excluding manufacturing and assembly. The presented version of the robot is the Wolfgang-V2. The published paper [BGVZ21] describes the earlier version 1, which has some minor changes, especially to the computer (see also Figure 3.1), but is generally very similar. The following sections describe the platform in more detail.

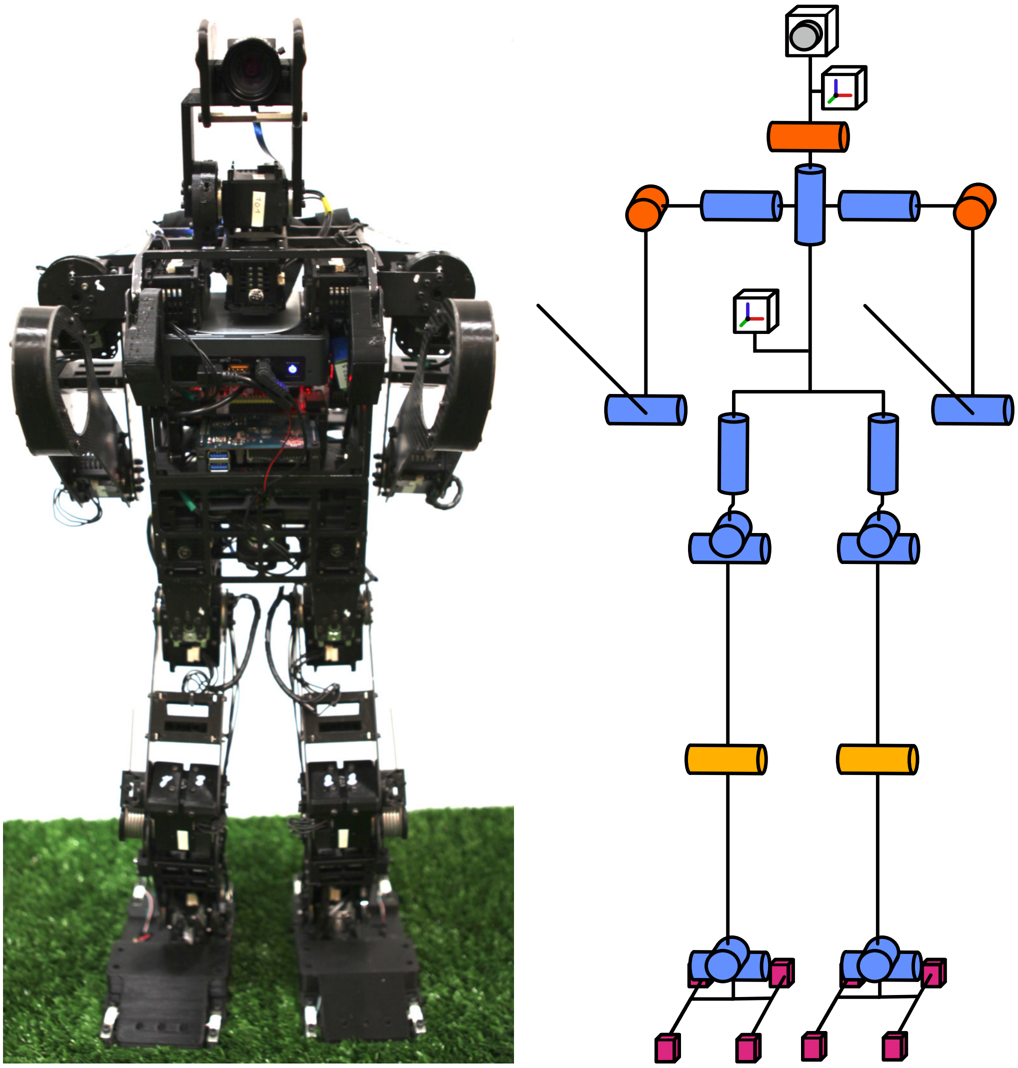


Figure 3.8: Photo of the Wolfgang-OP robot (**left**) and illustration of its kinematic chain (**right**). The joints with elastic elements are colored **orange**, the parallel elastic actuators are colored **yellow**, and the other joints are colored **blue**. The position of the FPSs (**red**), the IMUs, and the camera are indicated. Based on [BGVZ21].

Table 3.3: Wolfgang-OP Specifications. [BGVZ21]

Type	Specification	Value		
General	Height	80cm		
	Weight	7.5kg		
	Battery	3500mAh, 22.2V, 6-cell LiPo		
	Battery Life	25min standing, 10min walking		
	Material	Aluminum, Carbon, PLA, TPU		
	Control loop	700Hz - 1000Hz		
PC	Name	Asus PN51-E1		NCS2
	Processor	Ryzen 5700U		Myriad X
	CPU Cores/Threads	8/16		
	CPU Frequency	4.3GHz		
	Memory	16 GB		
	Network	GigE		
	Wireless	2.4/5GHz		
Controller	Microcontroller	Teensy 4.0 + FT4232H		
	CPU	600MHz Cortex-M7		
	Connection	4 x RS-485 @ 12 Mbaud; USB 2.0 HS @ 480 Mbaud		
Camera	Camera	Basler acA2040-35gc		
	Camera Lens	Computar Lens M0814-MP2 F1.4 f8mm 2/3“		
	Resolution	2048 px x 1536 px		
	Frames per second	36		
	Shutter	Global Shutter		
	Connection	GigE		
IMU	Sensor	MPU6500		
	Gyro Full Scale Range	± 2000 °/sec		
	Gyro Sensitivity	16.4 LSB/°/sec		
	Accel Full Scale Range	± 16 g		
	Accel Sensitivity	2048 LSB/g		
	Microcontroller	ESP32		
	Connection	RS-485 @ 4 Mbaud		
Foot Pressure	Foot Pressure	Custom strain gauge based		
	Range	0-40 kg		
	ADC Resolution	32 bit; ca. 18 bit noise-free		
	Microcontroller	ESP32		
	Connection	RS-485 @ 4 Mbaud		
Actuators	Name	MX-64	MX-106	XH540
	Count	8	10	2
	Stall torque	7.3Nm	10Nm	11.7Nm
	No load speed	78 rev/min	55rev/min	46 rev/min
	Backlash	0.33°	0.33°	0.25°
	Position Sensor	hall based, 12bit resolution		
	Connection	RS-485 @ 4 Mbaud		

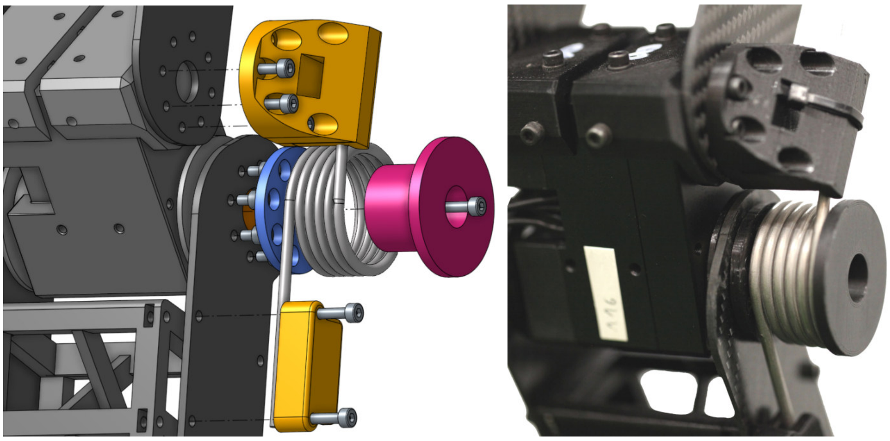


Figure 3.9: Exploded computer-aided design (CAD) model of the PEA (**left**) and photo of the real joint (**right**). The torsion spring (**grey**) is fixed to the upper and lower leg by 3D printed parts (**yellow**). Two additional 3D printed parts ensure that it stays centered on the joint axis (**red**) and that it does not get stuck on the screw heads (**blue**).[BGVZ21]

3.3 Mechanics

The mechanical structure is mainly the same as in the predecessor platform Nimbro-OP [SSS$^+$12]. Slight changes were already made to shorten the legs in the Wolves version and also adopted by us. It consists of four different materials: cut carbon fiber plates, computer numerical control (CNC) milled aluminum, 3D printed polylactic acid (PLA), and 3D printed thermoplastic polyurethane (TPU). Carbon fiber parts are light and strong but only allow simple, mostly two-dimensional, shapes. Therefore they are only used in the legs and arms of the robot. Aluminum can be milled into more complex shapes while still providing good strength for little weight. The 3D printed materials provide the largest freedom of design but do not provide the same strength per weight. Additionally, PLA has the issue that it breaks easily along the axis it was printed due to issues with inter-layer adhesion. It is mainly used for connecting parts. TPU is not as brittle but can be only used for certain parts due to its elastic nature.

3.3.1 Torsion Springs

The Wolfgang-OP (like any humanoid robot without toes, a one-way bendable knee, and rigid leg links) needs to keep its knees bent at most times to avoid the kinematic singularity that exists when the knees are completely straight. If the legs are fully extended, the robot cannot control the position of its center of mass (CoM), making it unstable and prohibiting most motions, especially walking. While keeping the knees bent by ca. 60° mitigates this problem, it comes with the price of a constant additional torque on the knee joint. This torque is undesirable, as it increases energy consumption and limits the remaining torque that the servo can produce for the actual task motion.

There are two possible solutions to this problem. Either the kinematic structure of the robot is changed in a way that does not require constant knee bending, or a counter torque is produced, which reduces the constant load on the knee joint. We took the latter approach since changes to the kinematic structure require more effort and would probably result in a more expensive robot. To produce this counter torque, a torsion spring was added to the side of each knee (see Figure 3.9), turning the knees servos

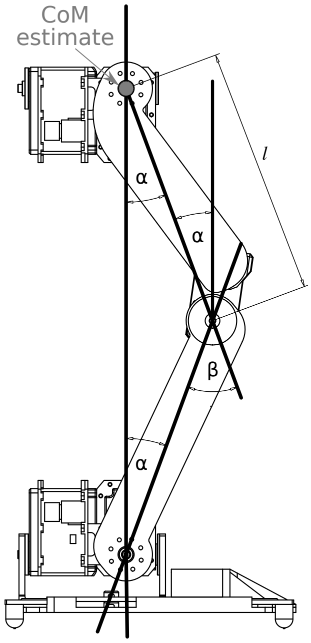


Figure 3.10: Drawing of the Wolfgang-OP's leg from the side to visualize the used mathematical symbols. [BGVZ21]

into PEAs. The torque compensation could further be improved by using a clutching mechanism [PvNWV15], but this would require additional hardware and thus increase costs and complexity.

In the following, we describe our approach for finding the correct torsion spring for any humanoid robot (see Figure 3.10 for a visual explanation). The knee torque (τ_m), which results from the mass of the robot, can be approximated using the equation for an inverted pendulum if only the gravitation force is acting on the robot.

$$\tau_m \approx \frac{1}{2} mgl \cdot \sin\alpha \tag{3.1}$$

Here m is the mass of the robot above the knees (thus excluding the mass of the lower legs and feet), which is divided by two since it is distributed over two legs. Furthermore, g is the gravitation constant, l is the distance to the mass point, and α is the pendulum's angle. Since m, g, and l are constant values, we can see the mass-based torque as a function $\tau_m(\alpha)$ that is only dependent on the angle of the pendulum towards the gravitational vector.

The torque resulting from the torsion spring (τ_t) can be computed using Hooke's law.

$$\tau_t = -k_s\beta \tag{3.2}$$

Here k_s is the spring constant, and β is the bending angle of the spring. The spring can be mounted with an offset angle δ. Then the β can be computed as

$$\beta = \gamma + \delta \tag{3.3}$$

where γ is the current angle of the knee joint. Offsets of $\delta > 0$ are not of interest as they would create no τ_t for small bending angles of the knee. Offsets of $\delta < 0$ would require springs with a large bending range, as the knee requires almost 180° movement, thus limiting the available springs. Additionally, choosing $\delta = 0$ reduces the complexity of the mathematical model and the hardware implementation. Therefore, we chose to omit this possibility and assume that $\beta = \gamma$. Since k_s is constant, the spring-based torque can be written as a function $\tau_t(\beta)$ that only depends on the knee angle.

Based on these, we can define the torque on the knee if the robot is standing on both legs τ_d (double support), as well as when standing on only one leg τ_s (single support) and when the leg has no ground contact τ_n. We neglect the torque resulting from the foot mass for τ_n.

$$\tau_d(\alpha, \beta) \approx \tau_m(\alpha) + \tau_t(\beta) \tag{3.4}$$

$$\tau_s(\alpha, \beta) \approx 2\tau_m(\alpha) + \tau_t(\beta) \tag{3.5}$$

$$\tau_n(\beta) \approx \tau_t(\beta) \tag{3.6}$$

The optimal solution would be to find a k_s^* that minimizes the torques τ_d, τ_s, and τ_n.

$$k^* = \underset{k_s \in K_s, \beta \in [0,\pi], \alpha \in [0,\pi/2]}{\operatorname{argmin}} (\tau_d(\alpha, \beta), \tau_s(\alpha, \beta), \tau_n(\beta)) \tag{3.7}$$

Here K_s is the set of all spring constants of springs which are available on the market and fit from their dimensions to the robot's knees. Furthermore, we assume that the kinematics of the robot limits the knee angle and the pendulum angle is limited by 0 since the knees are always in front of the body and by $\pi/2$ since the torso of the robot would touch the ground at this angle.

Since this optimization depends on both α and β, it can not be solved directly. It would also depend on how much time the robot spends in having certain α, β value combinations, and the ratio of time where the legs have ground contact. For example, having a k_s value that leads to lower τ_d for the typical α and β values of a standing or walking robot reduces the overall power consumption more than having a lower τ_d for angles that are rarely reached. Finding the optimal solution for this would require recording statistical data on the usage of the robot, which is a great effort and only possible when the robot is already built.

Instead, we apply additional approximations to circumvent this and to find a non-optimal but good k_s°. To do this, we exploit the fact that the robot's CoM is typically above its feet when the robot is upright, as it would otherwise fall due to the zero moment point (ZMP) reaching the edge of the support polygon. Additionally, we assume that the robot is primarily upright, as it will stand back up after a fall. Furthermore, the Wolfgang-OP, like most humanoids, has the same link length for the upper and lower legs. The position of the CoM is generally variable and depends on the current pose of all joints since these move around the masses of the head, arm, and leg links. In the case of the Wolfgang-OP, the mass of the torso is high, while the mass of the arms and the head is comparably small. Furthermore, the arms are primarily held in a similar pose close to the torso. The legs have a higher mass, but only the part above the knee is counted. Therefore, the position of the CoM typically does not move significantly in relation to the robot's torso.

We approximate its position at the hip joint, although it is a few centimeters higher than that, as this will allow us to come to an analytic solution. This leads us to a rough estimation of τ_m, which is lower than the actual torque.

Based on this, we can assume:

$$-\beta \approx 2\alpha \tag{3.8}$$

Inserting 3.1, 3.2 and 3.8 into 3.4 will produce the following.

$$\tau_d(\beta, k_s) = \frac{1}{2} mgl \cdot \sin\beta - k_s\beta \tag{3.9}$$

We simplify the equation further by replacing the concrete value of k with a scaling factor x. This makes intuitive sense, as we want to scale the torques of the mass and the spring correctly together rather than finding a concrete k.

$$k_s = x_s \cdot \frac{1}{2} mgl \tag{3.10}$$

Inserting 3.10 into 3.9 leads to the following

$$\tau_d(\beta, x_s) \approx \frac{1}{2} mgl \cdot \sin\left(\frac{1}{2}\beta\right) - x_s \cdot \frac{1}{2} mgl \cdot \beta \tag{3.11}$$

$$\tau_d(\beta, x_s) \approx \frac{1}{2} mgl \cdot \left(\sin\left(\frac{1}{2}\beta\right) - x_s \cdot \beta\right) \tag{3.12}$$

We reached a function that consists of a constant part ($\frac{1}{2}mgl$), and the difference between a sine function and a linear function both depended on β. The factor x_s can be used to pick a trade-off between applied torque in different knee angles for the cases of τ_d, τ_s, and τ_n. These trade-offs are then independent of the concrete mass and leg link length values of the robot platform. A visualization of the influence of x_s on the different torques can be seen in Figure 3.11.

As discussed above, the optimal value for overall torque reduction depends on how much time the robot spends in certain poses. However, it may also be important to reduce the maximum torque that can occur at any angle, as the robot would otherwise not be able to perform certain motions. Both, the overall torque reduction as well as the maximum torque reduction, need to be accounted for the Wolfgang-OP. The energy consumption should be decreased, but the knee servos are close to their limit during stand-up motions (see also Figure 3.26a), which leads to issues when the battery gets empty or when an older servo is used. Still, it is necessary to ensure $\max \tau_n \leq \max \tau_d$ as the robot needs to pull its legs close to the body while lying on its back at the beginning of a stand-up motion. If this movement creates too much torque, the stand-up motion cannot be performed. We can assume that $\beta \in [0, \pi]$, due to the joint limits of the robot. This results in $x_s \approx 0.15$.

Inserting this value in 3.10 will provide the spring constant, which is optimal under our assumptions.

$$k_s^\circ = 0.15 \frac{1}{\text{rad}} \cdot \frac{1}{2} \cdot 6.0\text{kg} \cdot 9.81 \frac{\text{m}}{\text{s}^2} \cdot 0.17\text{m} = 0,75 \frac{\text{Nm}}{\text{rad}} \tag{3.13}$$

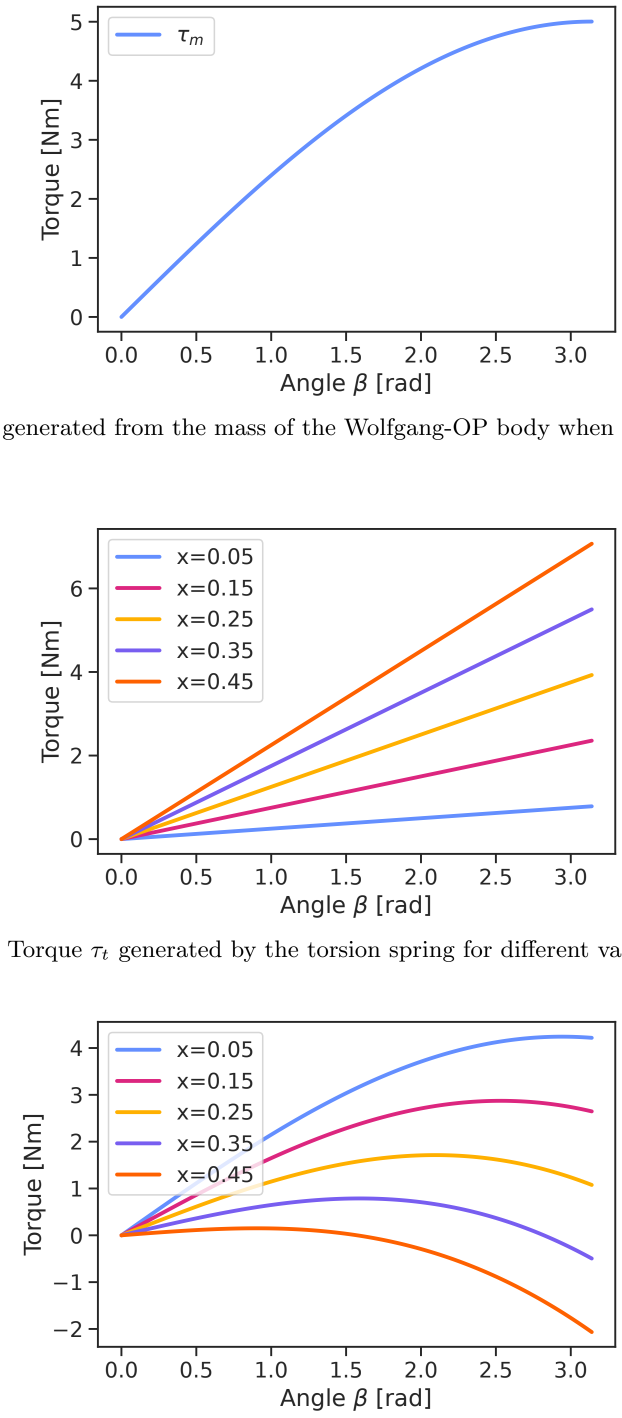
Due to size and availability, we chose a spring with $k_s = 0.755 \frac{\text{Nm}}{\text{rad}}$. This leads to a theoretical reduction of 37% for the maximal τ_d torque. A plot of the theoretical torques is shown in Figure 3.12, and an evaluation of the approach is presented in Section 3.6.2.

3.4 Electronics

The robot's electronics were built from scratch to allow better performance than its predecessor. An overview of the electrical system can be seen in Figure 3.13. The main computer was updated to a newer model, and a tensor processing unit was added for more processing power of neural networks. The complete sensor-servo bus was improved, including adding new sensors. The details are explained in the following subsections.

3.4.1 Sensors

The robot includes different sensors to allow the usage of multi-modal approaches. Each servo already includes a hall-effect-based rotational position sensor with a resolution of



(a) Torque τ_m generated from the mass of the Wolfgang-OP body when it is standing on both legs.

(b) Torque τ_t generated by the torsion spring for different values of x_s.

(c) Torque τ_d (the combination of τ_m and τ_t) for different values of x_s.

Figure 3.11: Visualization of the different torques on the knee joint over the span of the joint's movement. The torque τ_t is displayed without sign for better comparison.

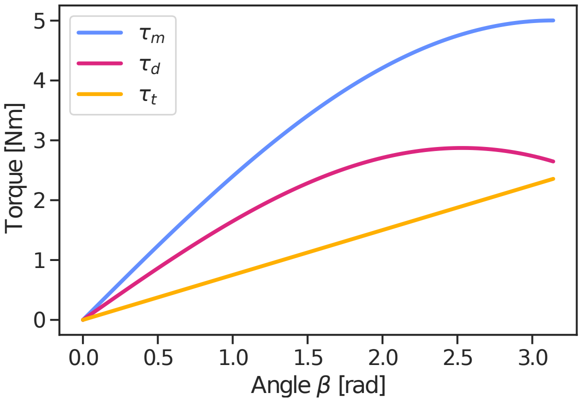


Figure 3.12: Plot showing the torques for the used spring with $x = 0.15$. The torque τ_m (**blue**) that would act on the robot if no PEA would be used is higher than the torque τ_d (**red**) for all joint angles if the robot is standing on both feet. If one leg is without ground contact, the additional torque τ_t (**yellow**) is applied due to the knee spring. Still, the maximal torque that may act on a knee is reduced. The torque τ_t is displayed without sign for better comparison. [BGVZ21]

12bit (see Section 2.2.1 for more details). While each servo also includes a temperature sensor, this is only used to ensure that no overheating happens. Additionally, they provide current measurements of the supplied power. Based on this, the torque that is currently generated by the motor can be computed since the torque provided by a motor (τ_e) equals the provided current (I) times the field constant (k_ϕ) [TP11].

$$\tau_e = k_\phi I \tag{3.14}$$

Naturally, this way of sensing the servo torque is influenced by friction and noise from the current sensing. Still, it provides an estimate of the torque without additional sensors.

The camera is connected to the main computer via Gigabit Ethernet and powered with PoE. All other sensors are powered and communicate through a bus system that uses the DXL Protocol (see Section 2.2.3). This reduces the number of cables drastically, as only a single cable is necessary for each extremity, rather than one for each sensor and actuator. The downside of this is that the bus limits the control loop rate of the robot. We, therefore, investigated how this can be optimized to allow high update rates (see Section 3.4.2).

IMU

The PCB was designed and build by Jasper Güldenstein. Development of the Firmware was done conjointly with him.

There are two IMUs installed in the robot. The main IMU is located in the torso and used to balance the robot during motions, e.g., walking. Another one is located in the head in the same kinematic link as the camera. This allows a better estimation of the camera's orientation which is needed for the inverse perspective mapping (IPM). While it could also be done from the IMU in the torso together with the head joint positions, this is inaccurate due to the elastic element in the neck. Previous platforms have mostly integrated the IMU into a larger board, e.g., the CM730, that also handles other tasks, i.e., connection to motors and power management. We created a small independent

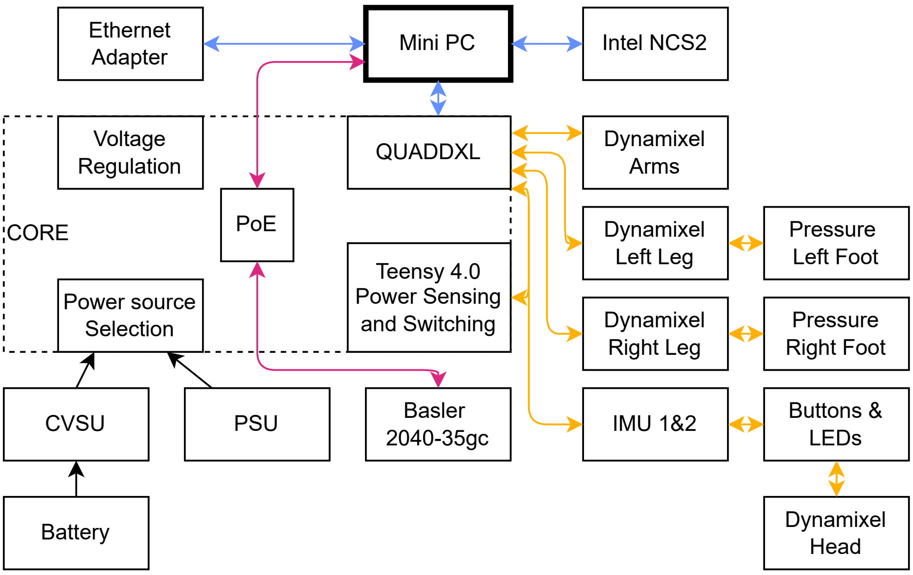


Figure 3.13: Overview of the electrical system in the Wolfgang-OP. The main computing unit (the Asus mini PC) already includes the GPU and is connected to the tensor processing unit in the Intel Neural Compute Stick 2 (NCS2) via USB (**blue**). It is also connected via USB to the QUADDXL, which provides the connection to the four RS-485 Dynamixel buses (**yellow**) with all servos and sensors. It is located on the COntrolling and REgulating (CORE) board which also provides Power over Ethernet (PoE) for the industrial camera that is connected to the main computer via Ethernet (**red**). To still allow connecting to the robot via Ethernet from an external computer, a USB Ethernet adapter is used. The battery is connected to the constant voltage supply unit (CVSU) that ensures a steady voltage level independent of the battery charge. The CVSU then connects to the CORE board. Since the external power supply unit (PSU) always provides a constant voltage, it is directly connected. The CORE then does further voltage regulation since some components require different voltage levels. Based on [BGVZ21].

board (see Figure 3.14) using an InvenSense MPU6500 IMU sensor that connects to the main bus system of the robot, thus allowing free positioning in the robot. It also includes an ESP32 microcontroller with two cores. One handles the bus communication, while the other reads the sensor and runs a complementary filter [VDX15] to estimate the orientation based on accelerometer and gyroscope values. This is a crucial point, as it ensures that the filter runs at a constant rate and no packages or IMU readings are lost. The IMU is connected via Serial Peripheral Interface (SPI) to the ESP32. Reading linear acceleration, angular velocity, and estimated orientation from the IMU requires a DXL package of 37 bytes. Optionally, the IMU sensor module can also support buttons and RGB-LEDs that can be read and set via DXL packages, too.

This approach of using a dual-core microprocessor to make a device directly connectable to a DXL bus can be applied to any peripheral electronics or sensors that have an interface that can be connected to an ESP32, such as GPIO, SPI, UART or I2C. Besides the ESP32, a RS-485 transceiver is required, which connects the ESP32's UART to the DXL bus. Furthermore, a step-down converter is required to provide the correct voltage for the ESP32 since the DXL bus has a higher one. A generic schematic can be seen in Figure 3.15. This approach has also been used for the FPS of the Wolfgang-OP (see next Section) and to create rotary encoders [Ste22].

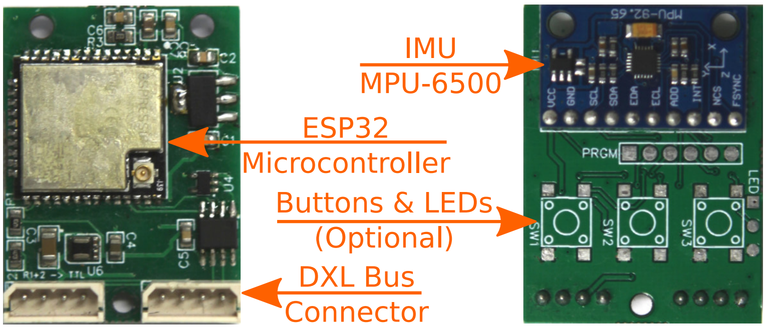


Figure 3.14: Annotated images of the IMU module shown without buttons and light-emitting diodes (LEDs) installed.

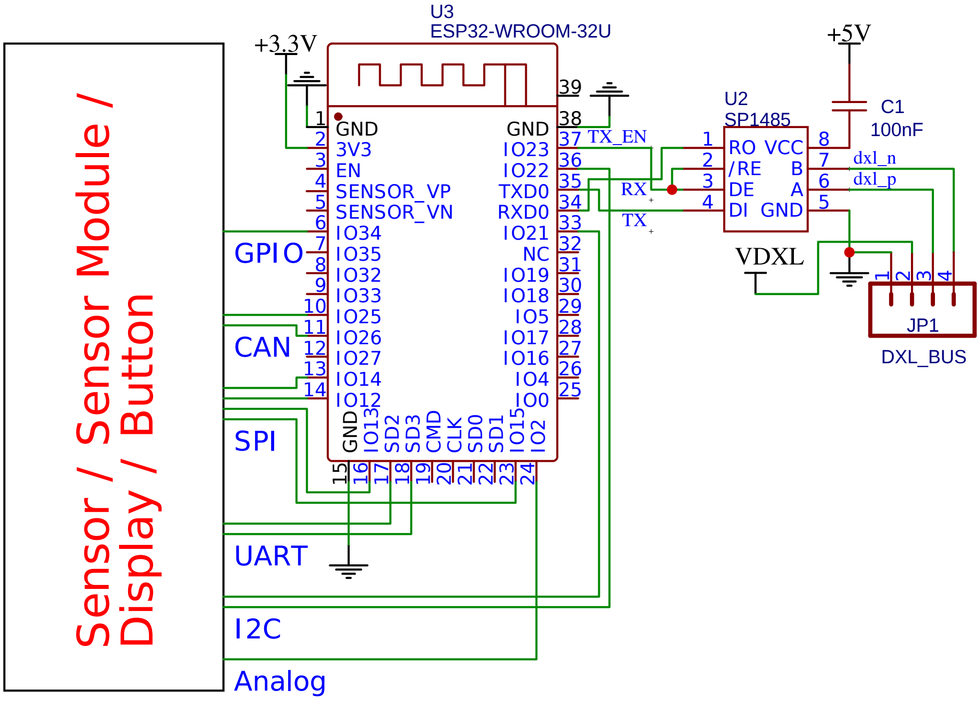


Figure 3.15: Generic schematic for a sensor or periphery module that connects to the DXL bus. An ESP32 connects via UART and a RS-485 transceiver to the DXL bus. Various sensors or input/output (IO) modules can be attached to the ESP32 via standard microcontroller interfaces. A step-down converter provides the necessary voltage for the ESP32. Based on [BGZ19].

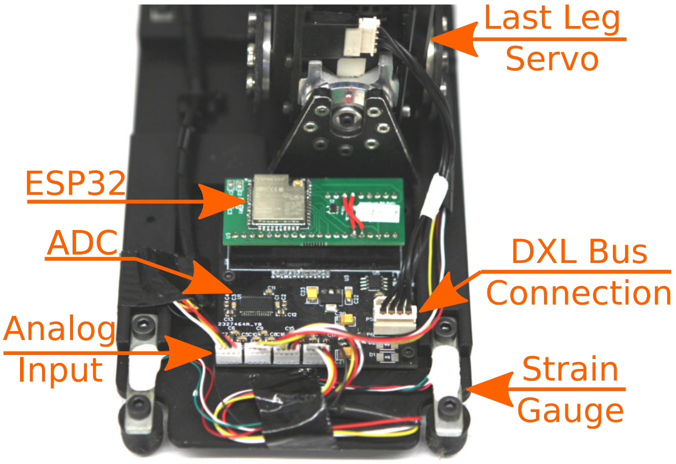


Figure 3.16: Annotated image of the FPS installed on the foot. The 3D printed cover was removed to show the PCB. Each corner of the foot has a cleat with a strain gauge. Their analog signals are fed into the ADC, which is then read by the ESP32 microcontroller. To include the FPS into the bus DXL bus, it is connected to the last servo of the robot.

Foot Pressure Sensor

The PCB was designed and build by Jasper Güldenstein. Development of the Firmware was done conjointly with him.

Another important modality for humanoid robots, besides joint positions and inertia measurements, are forces on the soles of the feet. On the one hand, they can be used to measure the center of pressure (CoP), which is the same as the ZMP, an important stability criterion for bipedal robots [KHHY14, pp.72f]. On the other hand, they provide Boolean information if the foot is in contact with the ground. This allows loop closure for step timings in locomotion skills and the detection of the kidnapped robot problem for localization.

The Wolfgang-OP has four strain gauges at the corners of each foot to measure the one-dimensional vertical ground reaction forces (see Figure 3.16 and 3.17). This approach of sensing the forces is inspired by the sensor of the Sigmaban robot [RPH$^+$15], which also uses four strain gauges. However, this sensor uses a HX711 analog-to-digital converter (ADC) to sense the strain gauges, and an ATMega328PB microprocessor to allow reading them via the Dynamixel protocol. While their approach was generally successful, the used ADC limits the reading rate to 80 Hz. To improve on this, our approach uses the ADS1262 as ADC, which theoretically allows sampling four analog signals with a resolution of up to 32 bit and a rate of up to 38,400 Hz. Furthermore, it already integrates on-chip filtering. The sensor is connected via SPI to an ESP32. Similarly to the IMU, this processor was chosen as it allows bus communication and reading of the sensors simultaneously. The sensor provides the raw data of each strain gauge to allow more flexibility in using this data in the software. Alternatively, it would also be possible to directly provide the CoP, requiring less data to be transmitted on the bus.

3.4.2 Dynamixel Bus Communication

All sensors and actuators in a robot need to be connected to the processing unit. Directly connecting each of them would lead to a high amount of cabling, especially for multi-

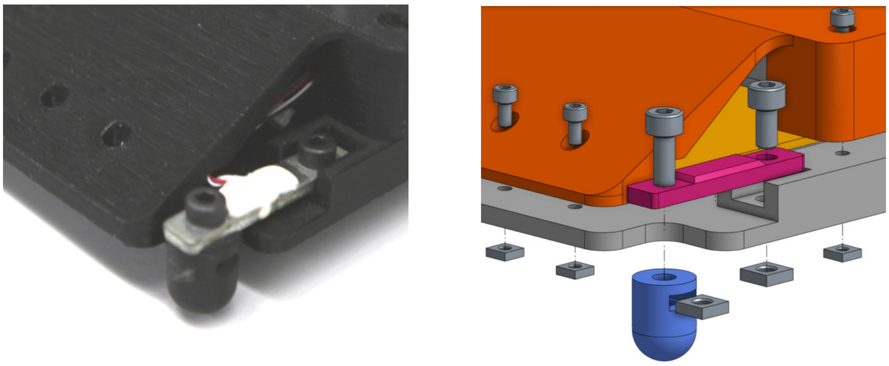


Figure 3.17: Photo (**left**) and exploded CAD drawing (**right**) of a foot cleat. The strain gauge (**red**) is fitted on one side to the 3D printed foot (**gray**) to ensure that it does not move. On the other side, the cleat (**blue**) is mounted. The PCB (**yellow**) is protected by a cover (**orange**).

modal and high DoF humanoids. This problem is typically circumvented by daisy-chaining all parts into a bus system, thus reducing the number of cables to one per extremity. While different bus standards are available, the RS-485 bus (see Section 2.2.2) is used as the Dynamixel motors require it. Similarly, only the Dynamixel protocol (see Section 2.2.3) is discussed, but the findings generalize to other similar command-status protocols. A discussion of previous approaches can be found in Section 3.1.5.

The downside of using a bus system is that all devices have to share the bandwidth of the bus. Therefore, an increase in the number of connected devices results in fewer reads/writes per device per second. This can create a bottleneck in the overall control loop cycle of the robot. Thus leading to slower response times to sensory input, which is undesirable for dynamic and reactive motions. To illustrate the importance, we show how the control loop frequency influences the forces that act on the robot when the swing foot makes contact with the ground. First, we can compute how much time (t) is needed between sensing the ground contact and the servos receiving the stop command. We assume that hardware communication and control are performed asynchronously (as it is typically done when using ROS) and that we communicate with the hardware with a fixed frequency f. We need more than two control cycles, in the worst case almost three, to react to the event of the foot making ground contact (see also Figure 3.18).

$$2\frac{1}{f} < t < 3\frac{1}{f} \tag{3.15}$$

This can also be expressed with a factor e that reflects how the contact's moment relates to the control loop's phase.

$$t = e\frac{1}{f}, e \in [2, 3] \tag{3.16}$$

The traveled distance x of the foot during t can be described as following if we assume a constant velocity v.

$$x = vt \tag{3.17}$$

To estimate the resulting force we can use Hook's law that describes the resulting force (F) from compressing a spring with the spring constant (k_s) over a distance of (x).

$$F = k_s x \tag{3.18}$$

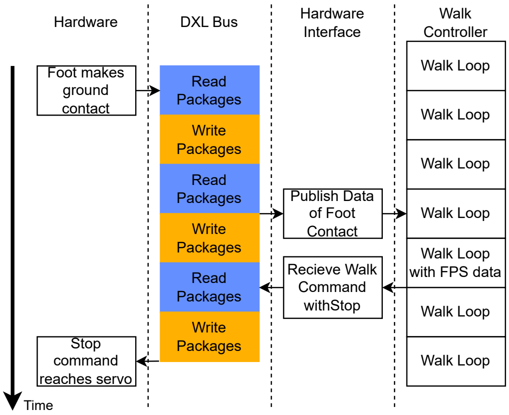


Figure 3.18: Illustration of why it takes multiple control cycles to react on the ground contact.

Most solid materials follow this law as long as the compression is small. Combining the equations leads to the following.

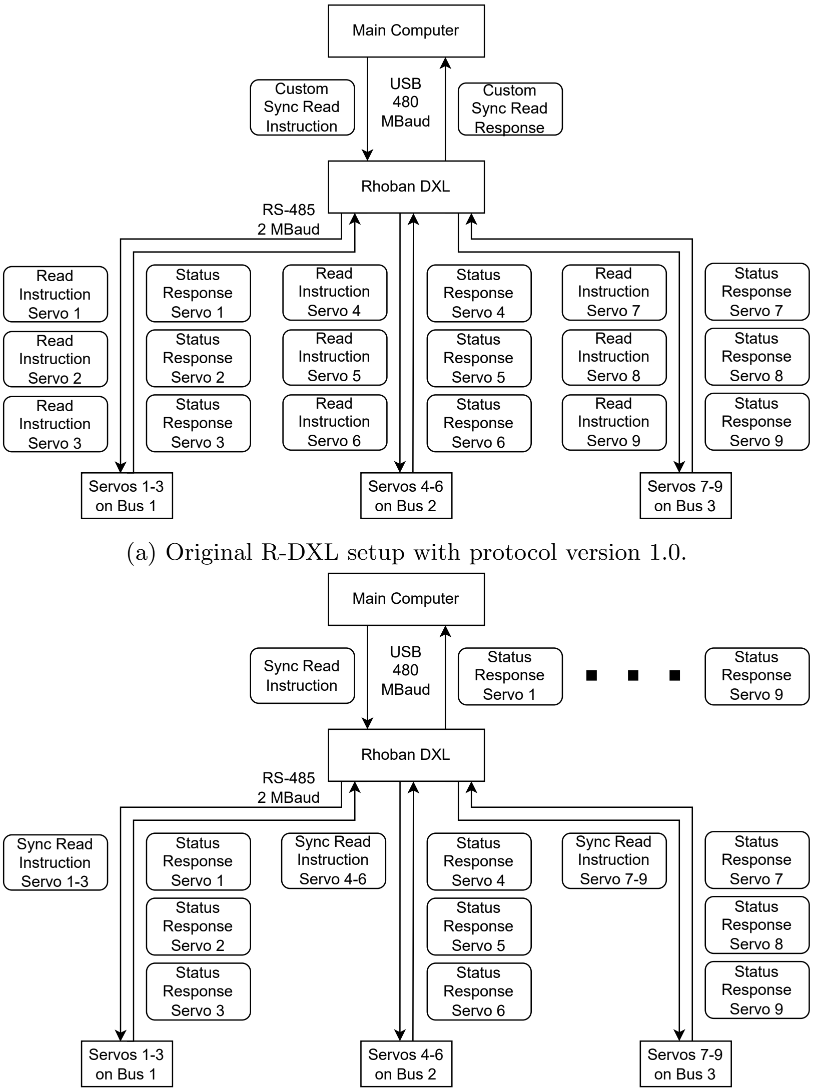
$$F = k_s v f \frac{1}{f} \tag{3.19}$$

Although k, v, f are not known, it is clear that the resulting contact force is anti-proportional to the control rate. This means increasing the control rate decreases the ground contact force when the robot puts its foot on the ground.

The previously used hardware in the Minibot, the CM730, resulted in low control loop rates of around 80Hz. Furthermore, package loss and breaks in communication were frequent. Therefore, two approaches were investigated to improve the control loop time. First, a new firmware was written for the existing Rhoban DXL Board (R-DXL) to enable usage of the protocol version 2.0, thus enabling the usage of sync commands (see Section 2.2.3). Second, a novel approach was investigated based on a four-channel UTS chip. Thereby, a microcontroller-based and a UTS based approach were investigated and could be compared. Both approaches are presented in the following subsections and evaluated in Section 3.6.1.

R-DXL Board

The R-DXL was the only available multi-bus controller. Therefore it was chosen as a basis for further development. It supports three buses, one with 4.5 Mbaud and two with 2.25 MBaud, and can be connected to the main computer via USB 2.0 Full Speed with 12 Mbaud. The original firmware supported only the Dynamixel Protocol 1.0 (compare Section 2.2.3). Additionally, a custom sync read instruction was implemented. When the board receives this command, it is translated into standard read commands for each servo and sends these on the corresponding buses. All returning status packages are then aggregated and returned to the main computer as a single custom response package. While this decreases the number of bytes necessary to send between the computer and the controller board, it does not improve the performance of the servo buses. Any overall improvement from this is doubtful since the bandwidth of the USB bus is higher than the combined bandwidth of the servo buses. Some additional communication over the USB bus is necessary, i.e., to read the IMU, and an additional overhead exists due to the USB protocol. Still, it is improbable that this bus will create a bottleneck due to its high bandwidth. Additionally, these custom packages are not specified in the protocol and therefore violate it. See Figure 3.19 for a visual explanation.



(a) Original R-DXL setup with protocol version 1.0.

(b) Improved version using protocol version 2.0.

Figure 3.19: Exemplary setup of the R-DXL with nine servos equally distributed on the three buses. In the original version (**a**), it was necessary to send a read instruction for each servo on the bus. Additionally, non-standard sync read packages needed to be used. In the improved version (**b**), only one sync read per bus is necessary, and standard sync reads can be used, but the status packages are not aggregated. Still, this is faster as the USB connection has a higher bandwidth than the RS-485 buses.

The protocol version 2.0 specifies a sync read instruction that can be used as a replacement. Switching to this version not only circumvents the violation of the protocol by a custom sync read, but it also allows the use of sync read packages on the servo buses instead of regular reads. Thus a single sync read instruction can be sent from the main computer to the microcontroller. It is then split into three sync read packages for the three buses, each containing the corresponding servo IDs. The returning response packages can then be forwarded from the servos to the main computer in the correct order so that the protocol specifications are not violated. This way, the load on the servo buses, the current bottleneck, can be reduced since the sync reads require fewer bytes than regular reads (compare Section 2.2.3). As the returning packages to the main computer are now single returns for each of the servos instead of an aggregated custom one, the number of bytes on this bus increases. Still, this is not an issue since the USB connection has a higher bandwidth.

Since the board is open-source and open hardware, it is simple to modify the firmware. Two new firmwares that use protocol 2.0 were implemented and are compared in Section 3.6.1. One uses the above-described sync read with three buses, and one uses only a single bus. Since one of the buses allows communication with a higher bandwidth than the other two, using just this at a higher rate might increase performance. Additionally, the computational load on the microcontroller is also decreased as all instructions not targeting the IMU are just forwarded to the bus. All return packages can also be directly forwarded to the main computer. This might reduce the latency from parsing packages.

QUADDXL

As described in Section 3.1.5, the existing solutions for this problem can be distinguished based on the usage of a microcontroller or an UTS chip as the main component. A representative of the first class is described above with the R-DXL. Since there were no previous solutions for a multi-bus controller based on a USB to serial chip, we created a new one that we call *QUADDXL*. Its main component is an FT4232H chip from the company FTDI which converts a single USB 2.0 High-Speed interface with 480 Mbaud to four UART channels with 12 Mbaud each. Thus, the available bandwidth is significantly higher than in any existing approach. Additionally, four SP485 transceivers are used to create an RS-485 interface for each of the UART channels. Virtual com port drivers for this chip have been included in the Linux kernel since version 3.0.0, which allows direct access to each of the buses as if they were single devices. This simplifies the usage from the main computer side. Compared to the microcontroller-based approaches, no firmware is necessary, further simplifying development. It is only necessary to set the chip into the correct mode so that the TX enable works. Generally, this approach is similar to the USB2DXL and U2D2 but offers multiple buses and higher bandwidths. We integrated this into the CORE board.

3.5 Modeling

> The creation of the Webots model was done together with Jasper Güldenstein. The joint modeling was done collaboratively, and he simplified the collision shapes.

An important part of a robot platform is a realistic model. This consists of multiple parts: First, a kinematic model that includes all links (with visual and simplified collision shapes) and the joints that connect them. This can be used to compute forward/inverse kinematics and collision checking. Second, a dynamic model, including all the links'

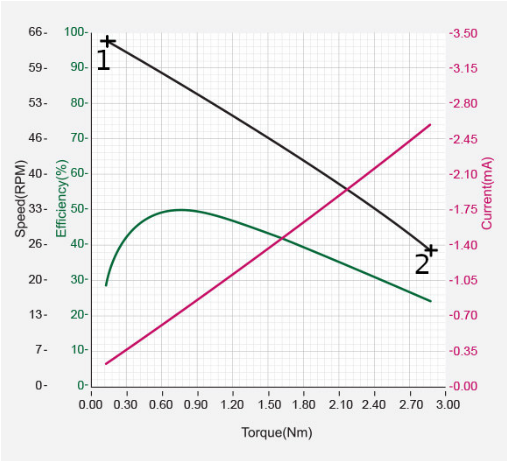


Figure 3.20: speed torque curve (NT-curve) of a MX-64 servo with the points for $\omega_{1/2}$ and $\tau_{1/2}$ marked. Modified from [26].

masses and inertia matrices, for simulating the robot. Third, models of the sensors and their noise to simulate the sensed data.

Creating a model by hand is tedious for such a complex robot. Furthermore, each future change in the hardware needs to be manually reflected in the model, thus complicating further development on the hardware. Therefore, we extracted most of the model directly from our CAD program. We chose to use *OnShape* [11] as CAD program since it is free for research use but still supports all necessary features, e.g., modeling of joints. It is cloud-based and, therefore, platform-independent. It also has a free view functionality which allows easy sharing of the model, which fits well with the open hardware nature of the platform. Furthermore, there was already an existing open source project called *onshape-to-robot* [10], which generates an Unified Robot Description Format (URDF) model directly from OnShape. We improved this project by allowing usage with ROS, by improving the accuracy of the joint representations (as described below), and by fixing bugs. The tool generates a model of all links by using the user-specified material to compute the weight and inertia of the link. Additional frames can also be specified, which is necessary to follow the standards specified in the ROS Enhancement Proposal (REP) 120, e.g., by adding a *l_sole* frame. The joints' name, position, and type can be specified directly through *mates* in OnShape. However, this does not allow to specify further joint properties, i.e., the torque, velocity, damping, and friction. Therefore, the user needs to specify the additional data in a configuration file which will then be used during the generation of the URDF.

A script was written to find these values for the different servos of the Wolfgang-OP (MX-64, MX-106, XH540-W270). This script became the standard for the computation of joint values for the HLVS [6]. The *maxTorque* and *maxVelocity* values can be directly read from the datasheet since they are provided as *stallTorque* (τ_{stall}) and *maxVelocity* (ω_{max}) respectively. The other two joint parameters (*damping* and *friction*) can not be read directly from the datasheet. It is necessary to rely on the NT-curve that is provided in the datasheet (see Figure 3.20). The torque (τ) and angular velocity (ω) are taken from the lowest ($_1$) and highest torque ($_2$) of the curve. All values in between are assumed to have a linear relationship that closely matches the actual NT-curve.

$$y = \mathrm{a}x + \mathrm{b} \tag{3.20}$$

We first solve the slope (a).

$$\mathrm{a} = \frac{\omega_2 - \omega_1}{\tau_2 - \tau_1} \tag{3.21}$$

Then, b can be solved by inserting one of the points into the equation.

$$\mathrm{b} = \omega_1 - \mathrm{a}\tau_1 = \omega_1 - \frac{\omega_2 - \omega_1}{\tau_2 - \tau_1}\tau_1 \tag{3.22}$$

This provides the complete linear equation for the NT-curve.

$$y = \frac{\omega_2 - \omega_1}{\tau_2 - \tau_1}x + \omega_1 - \frac{\omega_2 - \omega_1}{\tau_2 - \tau_1}\tau_1 \tag{3.23}$$

The damping constant (k_d) is the same as the slope of the curve but has, per definition, a positive sign while the slope is negative. Therefore it can be defined as follows.

$$k_d = -\mathrm{a} = -\frac{\omega_2 - \omega_1}{\tau_2 - \tau_1} \tag{3.24}$$

To compute the *friction* parameter, the effective torque must be computed first. The effective torque is not the same as the stall torque since the former is the torque while still moving (thus acting against friction), while the latter is the torque holding a position (thus the combined torque with friction). To compute the effective torque (τ_{eff}), the value of the NT-curve at $\omega = 0$ is computed.

$$0 = \frac{\omega_2 - \omega_1}{\tau_2 - \tau_1}\tau_{eff} + \omega_1 - \frac{\omega_2 - \omega_1}{\tau_2 - \tau_1}\tau_1 \tag{3.25}$$

$$\tau_{eff} = \frac{-(\omega_1 - \frac{\omega_2 - \omega_1}{\tau_2 - \tau_1}\tau_1)}{\frac{\omega_2 - \omega_1}{\tau_2 - \tau_1}} = -\omega_1\frac{\tau_2 - \tau_1}{\omega_2 - \omega_1} + \tau_1 \tag{3.26}$$

The *friction* (f) can then be computed by comparing the stall and effective torque.

$$\mathrm{f} = \tau_{stall} - \tau_{eff} \tag{3.27}$$

Unfortunately, the datasheet only provides the NT-curve for the default supply voltage of 12 V, but the Wolfgang-OP uses 14.8 V. The stall torque and maximal velocity are also provided for 14.8V in the datasheet and can, therefore directly, be used. The values for the NT-curve are then scaled with a factor for the torque (s_τ) and one for the velocity (s_v).

$$s_\tau = \frac{{}_{14.8}\tau_{stall}}{{}_{12.0}\tau_{stall}} \tag{3.28}$$

$$s_\omega = \frac{{}_{14.8}\omega_{max}}{{}_{12.0}\omega_{max}} \tag{3.29}$$

$${}_{14.8}k_d = -\frac{s_\omega\omega_2 - s_\omega\omega_1}{s_\tau\tau_2 - s_\tau\tau_1} \tag{3.30}$$

$${}_{14.8}\mathrm{f} = {}_{14.8}\tau_{stall} + s_\omega\omega_1\frac{s_\tau\tau_2 - s_\tau\tau_1}{s_\omega\omega_2 - s_\omega\omega_1} - s_\tau\tau_1 \tag{3.31}$$

The resulting URDF can directly be used in the PyBullet [15] simulator, but not in Webots since this requires a specific *.proto* format. The proto model can be generated from the URDF using a script [23]. Still, some manual modifications are necessary, i.e., to add the sensors. Images of the different models are shown in Figure 3.21.

Besides the kinematic (URDF) and the simulation (proto) model, a configuration for MoveIt is necessary. This specifies the kinematic chains and the inverse kinematic (IK) solver. It can be semi-automatically generated using the Moveit Setup Assistant [7]. This needs only be adapted if the kinematic chain of the robot changes, which does not happen frequently. Therefore, it is not necessary to completely automate this process.

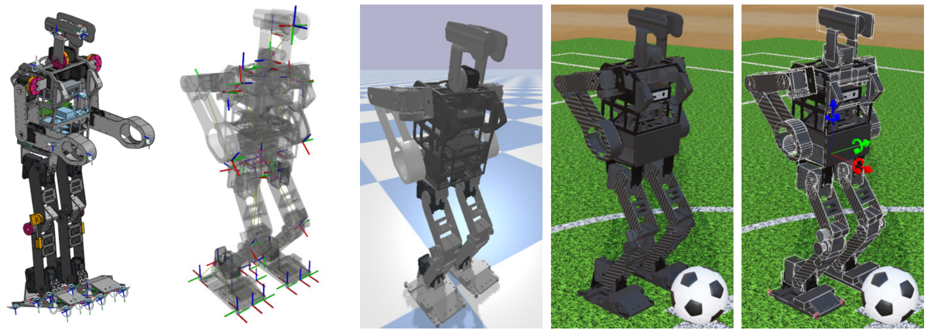


Figure 3.21: From left to right: CAD model in Onshape, URDF displayed in RViZ, PyBullet simulator model, Webots simulator model, Webots model with simplified collision shown.

3.6 Evaluation

This section evaluates different aspects of the Wolfgang-OP platform. First, the QUAD-DXL approach is evaluated and compared to other methods (Section 3.6.1). Then, the reduction of torque and energy usage by the torsion spring based PEA is investigated (Section 3.6.2). Afterward, the computational performance of the motion software stack is evaluated (Section 3.6.3). Finally, a qualitative evaluation is given (Section 3.6.4).

3.6.1 Servo and Sensor Interface

Typically, a continuous control cycle on humanoid robots includes reading the sensor data and writing new goals to the actuators. Unlike other robots, i.e., robotic arms and wheeled platforms, humanoid robots need to do this to keep stable. Thus, staying below a threshold for the time between two cycles is necessary. Also, higher control loop frequencies allow faster reactions to sensory information, for example, during disturbances. Therefore, decreasing the time for a read-write-cycle is preferable (see Section 3.4.2). First, we evaluate which control loop rates can generally be achieved with our approach. Then, we investigate which control loop rates are achieved when the approach is integrated into the Wolfgang-OP.

General Evaluation

The approaches presented in Section 3.4.2 are evaluated. For this, only the sync read-/write instructions are used, as they require the least number of bytes and packages (compare Section 2.2.3). The experimental setup consists of 20 Dynamixel MX-64. This setup was chosen since it is the same number of servos used in the Wolfgang-OP, and many other humanoids have similar numbers of actuators (compare Section 3.1.1). The MX-106 servos and the X-series are assumed to perform identically as they are using the same processor (STM32F103)[26]. For the experiments, position control is used, and all typically interesting information is read from the servo. Therefore, 4 bytes are required for the write command, and 10 bytes are required to read the robot's position, velocity, and drawn current. The latter can be used to estimate the torque that the servo is applying. In the single-bus case, this results in 568 bytes of data per update cycle, including overhead from the packages' checksums and headers (see Table 2.2). In a four-bus case, the necessary data is 163 bytes per bus or 652 bytes overall. The overall data size increases as sync read and sync write packages need to be written to each bus, requiring additional header bytes.

The mean update cycle rate of our approaches (see Section 3.4.2), as well as the USB2DXL, as a baseline, were measured. Additionally, the theoretical maximum that would be achieved if the bandwidth of the bus were ideally exploited is provided as a comparison. The results are shown in Table 3.4. While the servos should be able to communicate on 4.5 Mbaud [26], we were not able to establish a connection with any controller at this rate without a known reason. Thus, the maximal bandwidth that was tested was 4 Mbaud.

For a single bus, the highest rate for 1 and 2 Mbaud is reached by the USB2DXL, but the QUADDXL is only slightly slower. This difference is probably due to handling multiple buses via a single USB interface. Both versions of the R-DXL perform worse. The QUADDXL achieves the highest performance on the single bus if the full 4 MBaud are used. We did not test the U2D2 (as it was not available to us), which would be able to achieve this rate, but we expect that it would be slightly faster in this case, as it is very similar to the USB2DXL. In this configuration, the R-DXL only achieves a very low rate. We believe that the microcontroller is not fast enough to handle packages at this speed, and therefore package loss is happening.

The QUADDXL outperforms the R-DXL in all multi-bus cases significantly. It can be seen that the performance of the R-DXL does not scale with the number of buses as the QUADDXL does. The microcontroller is probably the bottleneck, although we tried to optimize the firmware. The QUADDXL increases its control loop rate with the number of buses and with increased bandwidth. A maximum of 1,373 Hz is reached with four buses, which is three times higher than on a single bus and two times higher than the theoretical limit of the single bus.

Interestingly, there is a significant gap between the theoretical limit and the actual performance for all approaches. A comparison can be seen in Figure 3.22. This difference means that there are significant spans of time on the bus where no data is transmitted. To get qualitative results on the bus usage, the experimental setup was extended with a logic analyzer. An exemplary result can be seen in Figure 3.23. It is clearly visible that the servos have a high reply time, not only initially after the sync read command but also after each response from another servo. This reply time does not change when different baud rates are used. This observation fits the fact that increasing the baud rate does not linearly increase the control loop rate since the waiting times for return packages are not affected by that. The exact time till a reply varies. To further investigate this, the logic analyzer was used to record 61,913 packages, and the reply times were computed (see Figure 3.24). While the delay is mostly comparably short, there are also long delays that are less frequent.

Unfortunately, the firmware of the servos is not open source, so it is impossible to investigate exactly why this is happening. Still, we have the following hypothesis that would explain this phenomenon. The protocol specifies that the return packages need to be ordered in the same way they were requested in the sync read. This ensures that there are never two devices that write their return package simultaneously on the bus. Thus each servo needs to parse the other return packages to know when it is its turn to write the return package. Since the incoming packages can only be verified with their checksum if they are fully parsed, the process of writing the own return package can only start if the preceding package is completely parsed. Depending on the implementation details, the data from the sensors could just then be read, thus requiring extra time.

This problem could be circumvented by changing the process of the return packages. One solution would be to use fixed time frames for each read device based on the position of the ID in the sync read package. Additionally, the return packages for sync reads could be changed to include fewer overhead bytes, i.e., by reducing the number of header bytes. Due to the mentioned closed-source nature of the servos, this is not simple to do. There exists an experimental alternative firmware that is open source [FRP+16]. But

Table 3.4: Mean update rates. [BGZ19]

No. buses	MBaud	Board	Rate [Hz]
1	1	*Theoretical Max*	*176*
		R-DXL Single	132
		R-DXL Multi	125
		USB2DXL	**153**
		QUADDXL	149
	2	*Theoretical Max.*	*352*
		R-DXL Single	215
		R-DXL Multi	179
		USB2DXL	**272**
		QUADDXL	261
	4	*Theoretical Max.*	*704*
		R-DXL Single	40
		QUADDXL	**398**
2	1	*Theoretical Max.*	*336*
		R-DXL Multi	185
		QUADDXL	**285**
	2	*Theoretical Max.*	*671*
		R-DXL Multi	219
		QUADDXL	**497**
	4	*Theoretical Max.*	*1342*
		QUADDXL	**744**
3	1	*Theoretical Max.*	*461*
		R-DXL Multi	224
		QUADDXL	**390**
	2	*Theoretical Max.*	*922*
		R-DXL Multi	250
		QUADDXL	**670**
	4	*Theoretical Max.*	*1843*
		QUADDXL	**1003**
4	1	*Theoretical Max.*	*613*
		QUADDXL	**524**
	2	*Theoretical Max.*	*1227*
		QUADDXL	**923**
	4	*Theoretical Max.*	*2545*
		QUADDXL	**1373**

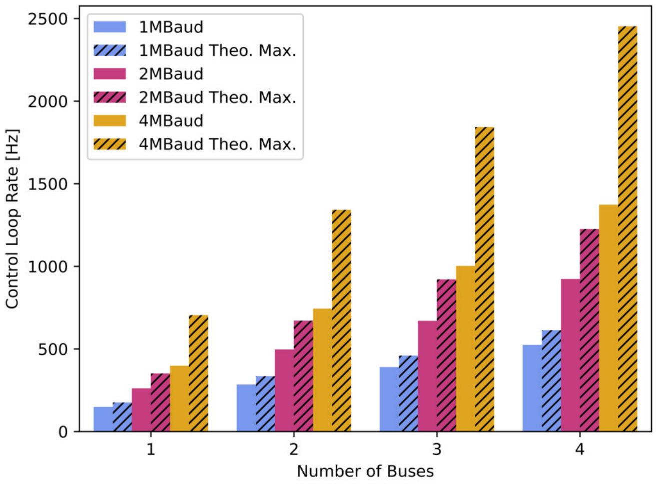


Figure 3.22: Actual and theoretical control loop rates for the QUADDXL with different numbers of busses. It can be observed that the difference to the real result gets larger with higher bus speeds as the response delay is constant while the time per byte decreases. Based on [BGZ19].

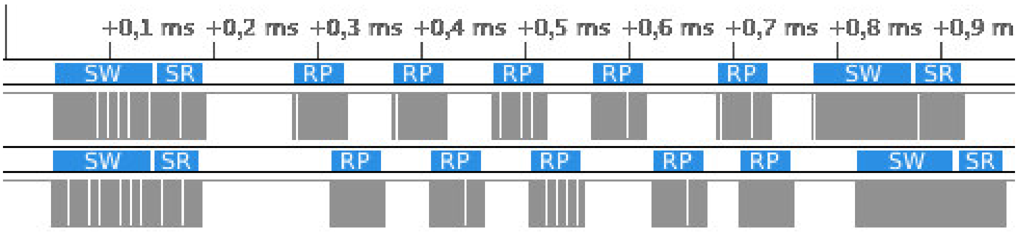


Figure 3.23: Screenshot of a logic analyzer output. Two buses with five servos each are controlled at 4MBaud. The first package is a *sync write* (**SW**), which is followed by a *sync read* (**SR**). Afterward, a series of response packages (**RP**) are visible. Then the control cycle starts again with another sync write. The delay before each status package is clearly visible. [BGZ19]

this is only for the MX-64 and protocol version 1.0. Therefore, using this as a basis to develop an improved behavior of the servos was out of the scope of this thesis.

Based on these findings, it is possible to formulate an equation that estimates the achievable control loop frequency (f) given a number of servos (n), the amount of data to read (d), and the used baud rate (b). As a comparison, the formula for the theoretical reachable control loop frequency (f_t) is shown in Equation 3.32. Since the protocol uses one start and one stop bit, 10 bit are necessary to transfer one byte. Therefore a corresponding factor is added. Equation 3.33 includes an additional part that represents the necessary reply time of the servos. While this is not exact, it gives an estimation (f_e) that can be used during the design of new robots or to validate in a running system that no additional bottlenecks were introduced into a system. A further evaluation of this equation on a real system is presented in Section 3.6.1.

$$f_t\text{Hz} = \left(\frac{d\text{B}\cdot 10\frac{\text{bit}}{\text{B}}}{b\frac{\text{bit}}{\text{s}}}\right)^{-1} \tag{3.32}$$

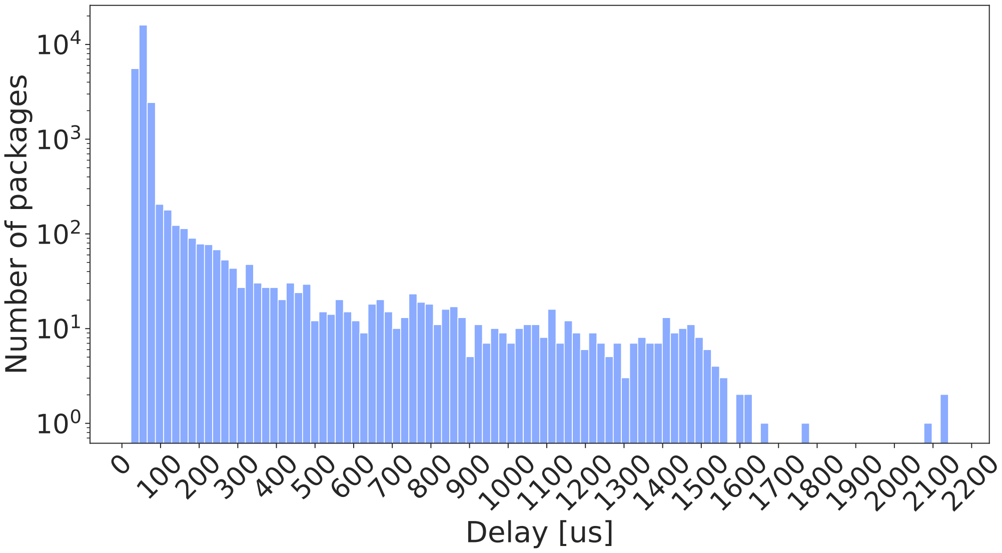


Figure 3.24: Delays between two status response packages are plotted as a histogram. The y-axis is scaled logarithmically to show the outliers with high delays better. The bucket size for each bar is 20us. Based on [BGZ19].

$$f_e\mathrm{Hz} = \left(\frac{d\mathrm{B} \cdot 10 \frac{\mathrm{bit}}{\mathrm{B}}}{b \frac{\mathrm{bit}}{\mathrm{s}}} + n \cdot 0.000050\mathrm{s} \right)^{-1} \tag{3.33}$$

After publishing our results in [BGZ19], the manufacturer introduced a new mode called *fast sync read*. Unfortunately, it is only usable with certain newer servo models. Therefore we could not do the same experiment with this new mode. It reduces the number of necessary bytes for the status packages by omitting the header of every single package and merging them together into one larger return package. While this will reduce the number of bytes and increase the control loop rate, it is unclear if the issue of slow responses from the servos is solved. This should be further investigated.

Platform Evaluation

While the experiments above provide a general evaluation of our approach, integrated tests on the platform together with the sensors are still necessary. To do this, different combinations of the available devices on the bus were chosen, and the resulting control loop frequency was measured (see Table 3.5). This only reflects the reading rate via the bus. The sensors themselves might be read at a lower rate, e.g., around 720Hz for the FPSs, thus providing the same value multiple times for fast reads via the bus.

The theoretical rate, following the Equation 3.33, is also provided. It is necessary to compute this rate for each bus and take the minimum since the devices can not be equally distributed on the bus due to the kinematic structure of the robot. The two leg buses have the highest amount of data since six servos, and one FPS are connected to each. The head chain with two servos and two IMUs is very similar since the IMU needs to transmit the most data. Therefore, not reading the foot pressure sensors leads to an increase in the loop frequency. While the embedding of the sensors as bus devices reduces the achieved control rate, these results are still a significant improvement to all existing solutions.

Comparing the estimated control loop rate following Equation 3.33 with the actually achieved one shows that the equation is not accurate for the integrated case. This

Table 3.5: Achieved Control Loop Rates for Different Configurations.

Servos Write Position	Servo Read Position	Servo Read Velocity	Servo Read Effort	Read Torso IMU	Read Head IMU	Read Foot Pressure	Estimated Rate	Achieved Rate
✓	✓	✓	✓	✓	✓	✓	1,140 Hz	757Hz
✓	✓	✗	✗	✓	✓	✓	1,270 Hz	798Hz
✓	✓	✓	✓	✓	✓	✗	1,388 Hz	758Hz
✓	✓	✗	✗	✗	✗	✗	1,459 Hz	1,194Hz
✓	✓	✓	✓	✗	✗	✗	1,290 Hz	1,078Hz
✗	✗	✗	✗	✓	✗	✗	4,706 Hz	1,811Hz
✗	✗	✗	✗	✓	✓	✗	2,352 Hz	1,228Hz
✗	✗	✗	✗	✗	✗	✓	6,557 Hz	2,172Hz

has probably two reasons. First, it does not take the computational ROS overhead of publishers and subscribers of the hardware interface into account. In the current implementation, the hardware interface is doing these sequentially together with the bus communication. This could be improved by parallelizing these. Second, the estimations for configurations that only include servos are closer to the achieved rate than those containing sensors. Since the equation was developed only based on the response behavior of the servos, possible different behavior of the custom ESP32-based sensors was not taken into account. This could be improved by evaluating their response behavior in more detail and possibly improving the firmware further to improve them. Still, it could also be that using the sensors just creates further computational overhead as more messages need to be published.

Besides the quantitative measurements, it can also be stated that the QUADDXL approach is now used by the team CITBrains in their new SUSTAINA-OP platform [22]. Notably, this robot does not use Dynamixel servos but other RS-485-based ones. Still, the QUADDXL approach is usable and preferable to their previous microcontroller-based solution. It is also part of a new Nimbro-OP2X-based robot platform that is in development by the team HULKS. This and the fact that the protocol was extended with the fast sync read indicates that the presented research had an actual impact on the robotics community.

3.6.2 Torque Reduction

To evaluate the influence of the torsion spring, the torque on the knee joints was recorded during different typical motions of the robot. The torque on the sensor is only approximated by measuring the current drawn by the DC motor. This is possible because the torque (τ) is proportional to the current (I) with a motor constant (k_τ) [TP11, p.55].

$$\tau = k_\tau I \tag{3.34}$$

The actual torque of the servo, including the gearbox, is naturally reduced due to friction. Still, this is not an issue as we only compare the measurements against each other, and both measurements are taken with the identical robot. The torque reduction can be directly seen as a reduction in the power usage (P) since the supply voltage (V) is assumed to be constant.

$$P = V \cdot I \tag{3.35}$$

Table 3.6: Comparison of Knee Torques. [BGVZ21]

Motion		No PEA [Nm]	PEA [Nm]	Relation PEA / no PEA
Standing in a walk ready pose	mean	0.48	0.31	0.66
	max	0.81	0.80	0.99
Walking	mean	2.47	1.91	0.78
	max	9.4	7.29	0.78
Stand up from the front	mean	1.85	1.61	0.87
	max	7.58	6.60	0.87
Stand up from the back	mean	1.90	1.49	0.78
	max	7.03	6.80	0.97
Standing to squatting	mean	1.29	0.48	0.37
	max	2.85	1.17	0.41
Squatting to standing	mean	2.75	1.74	0.63
	max	7.08	3.75	0.53

The torque values also depend on how the motion is performed, e.g., how high the foot is raised during the walking. Since evaluating all possible motion configurations is not feasible, all motions were evaluated with the parameters that were used for the RoboCup competition 2019. The results are presented in Table 3.6. Two metrics were investigated. First, the mean torque was measured to investigate how the power consumption was improved. Second, the max torque is measured to show how the stress of the servo is influenced and to see if the spring could allow the use of a less powerful (meaning smaller and lighter) servo for the joint.

The results show that the mean torque, and thereby the power consumption, is reduced for all motions. Sebastian Stelter performed a similar experiment as part of his thesis where he concluded that the PEA reduces the knee servos power usage by 26% while walking [Ste22, p. 54]. This closely matches our result. The slight difference could arise from using different walking parameters.

This reduction is even present in motions where the spring creates additional torque while the foot has no ground contact. An example is walking. In this case, the leg has no ground contact during the swing phase, and therefore the torque is higher than without a spring (see Figure 3.25). Still, a significant decrease in the mean torque can be observed. A similar situation exists during the stand-up motion from lying on the back (see Figure 3.26a). The robot needs to pull both feet towards the pelvis while only the torso has contact with the ground. Again, the torque with springs is increased in this phase of the motion, but the overall mean is reduced as the torque is lower in the rest of the motion. Additionally, the highest torque peak while getting up from a squatting position is decreased. The stand-up motion from lying on the belly shows a similar behavior (see Figure 3.27). The torque is increased by the PEA when the feet are pulled to the body, but the peak and the mean torque is decreased. The knee joint shows similar performance in tracking its goal position with and without PEA. The values of the proportional derivative (PD) controller were not adapted when the spring was added. Tuning these might further improve the performance of the PEA.

In contrast to this, the maximal torque values show almost no change for some motions. Two factors influence this. First, the robot's weight may not be equally distributed on the two legs, leading to different maximal torques in some repetitions. Second, this value is naturally more influenced by noise, and the measurement via the motor current is noisy, which results in single outliers that increase the maximal value.

We also tested if our theoretical model (see Figure 3.12) fits the measured data. For

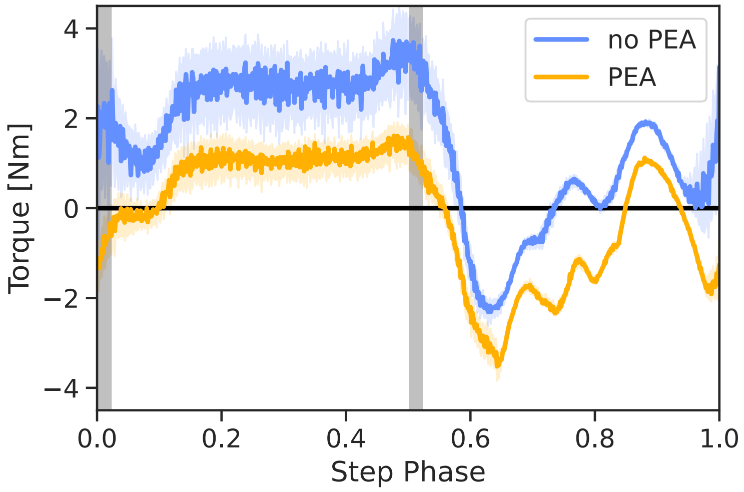


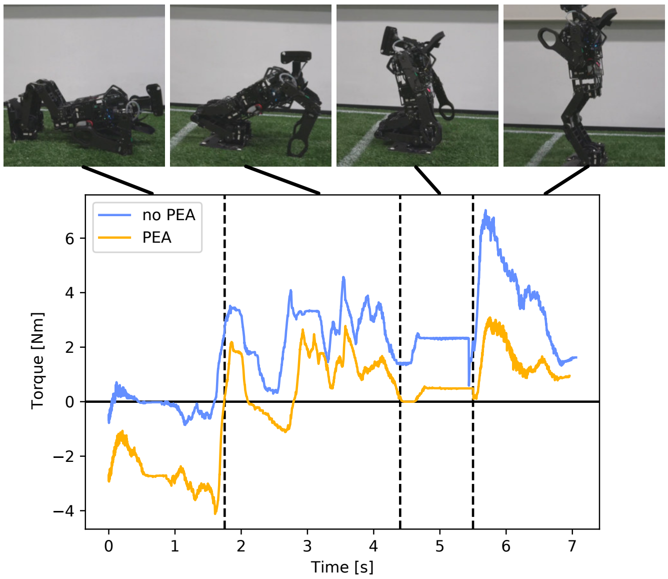
Figure 3.25: A plot of the torque values of one knee with (**yellow**) and without (**blue**) spring from 30s of walking each. The double support areas are marked in **grey**. In the first half, the knee supports the robot's weight. Therefore the spring reduces the torque. In the second half, the knee belongs to the swing leg, and thereby the spring generates additional torque. Modified from [BGVZ21].

this, the robot very slowly moved from squatting to a standing pose and back again. During this, the torque on the knees was recorded over the full joint range of the knees (see Figure 3.28). Interestingly, the torque is considerably different while moving up or down. We hypothesize that the potential energy is used while moving down, thus requiring less current in the DC motor (which is the basis for the torque computation). Another explanation is that static friction needs to be overcome when moving upwards. This is not the case when moving downwards since the torque generated by the mass of the robot will make the joint move as soon as the torque of the motor is reduced. At the highest angle, the torque drops as the hip motors make contact with the ankle motors. Thus the mass of the robot is not creating any torque. Still, the measured torques from moving upwards fit well with the theoretical model, and the torque with PEA is lower in both moving directions.

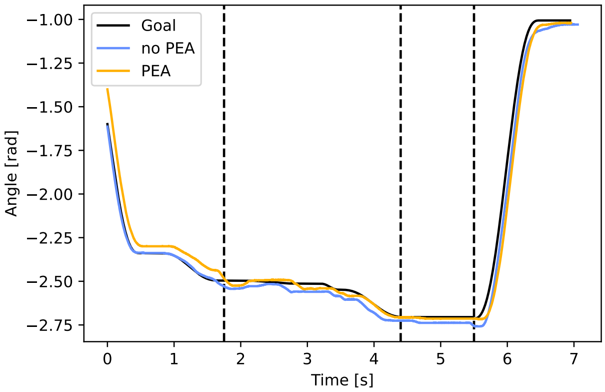
Overall, it can be concluded that the application of the PEA is successful. It reduces the mean torque and, thereby, the robot's power consumption. Additionally, it reduces the peak torques, especially during stand-up motions, thus reducing the stress on the servo and allowing future versions to use smaller servos in this joint. Compared to the approach with linear springs in the Poppy robot [LRO13], our approach does not only provide an advantage for walking but for a large variety of motions.

3.6.3 Computational Performance

As stated above, it is important that it is possible to control the robot in a high-frequency loop. The performance of the bus interface was already proven in Section 3.6.1. Still, it is also important that the ROS based system with its multiple running nodes achieves the same rates. One approach to ensure the correctly timed execution of multiple processes would be the usage of a real-time operating system and specially set scheduling. This would require great effort, and therefore, another approach was chosen. Since the central processing unit (CPU) of the Wolfgang-OP features eight cores with 16 hyper-threads, it is possible to divide the important nodes onto these cores. Thereby, each of them



(a) Plot of the knee torque during a stand-up motion from the back, with (**yellow**) and without (**blue**) torsion spring. [BGVZ21]



(b) Plot of the knee angle, with (**yellow**) and without (**blue**) torsion spring, as well as the goal position (**black**).

Figure 3.26: The different phases are marked with vertical lines. First, the feet are pulled towards the torso, resulting in a high torque from the springs. Second, the robot tilts on its feet and then stands in a squatting pose. Finally, the robot moves upwards to a standing pose. In these last three phases, both feet are on the ground, and, therefore, the overall torque is reduced by the spring. The position control with spring is worse in the first phase due to the additional torque, but better in the other phases.

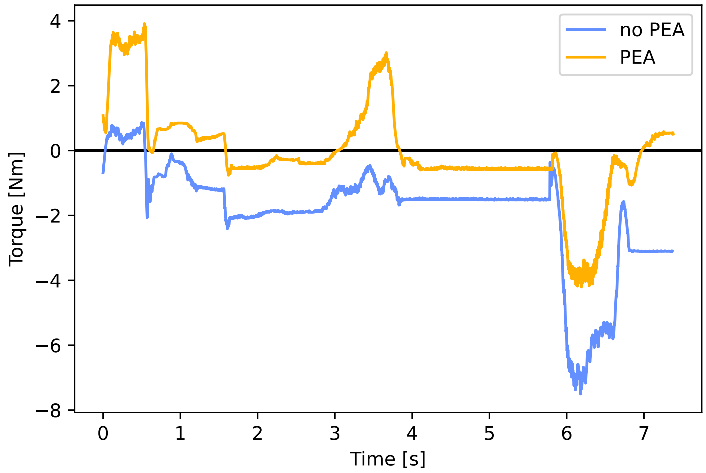
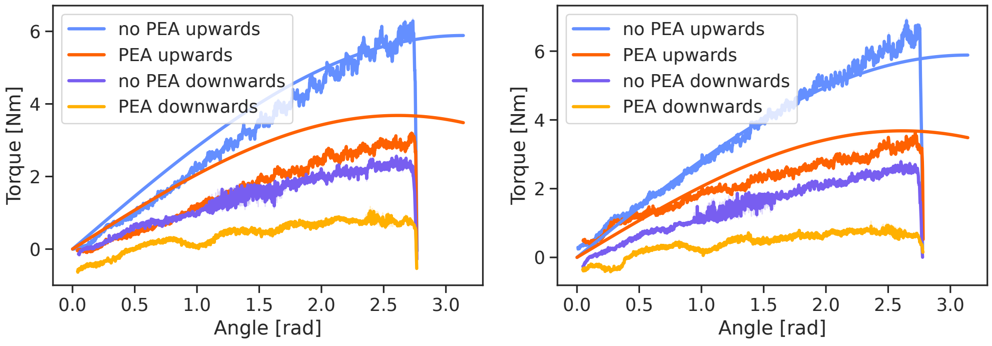


Figure 3.27: Torque of the knee during a stand-up from lying on the belly. First, the torque is higher with PEA while the feet are moved towards the torso. Still, the peak torque while moving from squatting to standing can be reduced.



(a) Left knee

(b) Right knee

Figure 3.28: Comparison between the theoretical angle torque relationship with measured torques while moving upwards and downwards between squatting and standing. A slight difference between the measurements in the knees can be observed. This can either be due to the different behaviors of the servos or to the CoM being not exactly centered laterally.

has its own CPU hyper-thread where it can run without being interrupted by the Linux scheduler. This can easily be realized by using the Linux kernel option *isolcpus* to forbid the Linux scheduler from using certain parts of the CPU. The important ROS nodes can then be set manually to these cores by using the command *taskset*. It may not be the solution that uses the performance of the CPU most efficiently, but it is one of the easiest. If the complete RoboCup software stack is executed without this approach, the control loop becomes unstable as a process like the hardware interface gets interrupted and moved around the processor's cores. This leads to visually observable degradation of the motions by creating jerky movements, which may make the robot fall.

The switch to ROS 2 first created a new problem as the CPU load of the nodes that run at a high frequency was significantly higher than with ROS 1. The reason for this was the inefficient implementation of the default executor that did not scale well with high-frequency topics. The problem was solved by using a new experimental executor called *EventExecutor*, which was at this time not part of the default ROS 2 libraries. Additionally, some nodes needed to be written in C++ instead of Python, as this executor was only available for C++.

After applying these optimizations to the motion stack, we measured the computational load when the robot is actively walking with a control loop frequency of 500 Hz. During this, the hardware interface, the HCM, and the walking each need less than one CPU hyper-thread. Additionally, auxiliary nodes, e.g., the odometry and the robot state publisher, take together less than one hyper-thread. Therefore, only a quarter of the available performance is used for the motion stack. It can be concluded that the CPU of the Wolfgang-OP is powerful enough to run the hardware interface and the motion stack at a high control loop rate while still leaving resources for high-level behavior and computer vision.

3.6.4 Qualitative Evaluation

The Webots model (see Section 3.5) of the platform was used in multiple virtual RoboCup competitions (see Section 2.5). In the simulated world championship 2021, the Hamburg Bit-Bots achieved third place using this robot. Additionally, the robot was awarded third place in the category "best model". In the same year, it also won first place in the simulated RoboCup Brazil Open tournament. In the HLVS 21/22 the Hamburg Bit-Bots achieved second place. Additionally, the model was successfully used to transfer motions from the simulator to the real robot (see Chapter 5). This shows that the model is accurate enough to be useful.

The real robot has been successfully used in its previous versions (see Figure 3.1) in the RoboCup championships in 2018 and 2019. In 2019, it won first place in the teen-size category push-recovery challenge. In this challenge, the goal is to withstand the largest external impact while walking. The latest version was used in the RoboCup 2022 with less success as the switch from ROS 1 to ROS 2 led to various software issues. The elastic elements in the shoulder joints, together with the fall detection (see Chapter 4), drastically reduced the number of gears that needed replacement due to fall damage. After multiple years of usage and many falls, no 3D-printed elastic elements broke.

Still, there is one major issue remaining in the platform which could not be solved during this thesis. This is the backlash issue of the Dynamixel servos. Due to the used gear system, each of them has a nominal backlash of 0.25-0.33° (depending on the model). This backlash increases further as the gears wear down over time of usage. The combination of the backlash from the multiple motors in the leg makes the robot challenging to control. For example, when a robot with old servos is standing straight, the torso can easily be moved by around 1 cm in the sagittal and lateral directions. Unfortunately, this issue can not easily be solved. One option would be to use different

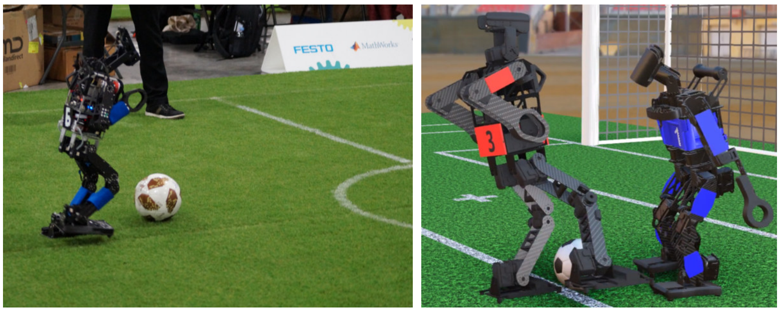


Figure 3.29: Images from Wolfgang-OP in the HSL (**left**) and in the HLVS (**right**). Photo courtesy of Florian Vahl.

motors, especially ones with planetary or strain wave gearboxes. Another would be using multiple servos for one joint and parallel kinematics.

The platform was also used as a basis for multiple bachelor, and master theses [Gül19] [Fie19] [Hag19] [Fle19] [Har19] [Ste20] [Mir20] [Ber20] [Rei20] [Eng21] [Gül22] [Ste22]. It was also used in multiple research papers with other authors (see Section 1.5). This shows that the platform is usable for various research topics and that the goal of creating a robot that is not only used for this thesis was achieved.

3.7 Summary and Future Work

In this chapter, the work on the low-cost Wolfgang-OP was presented. This platform builds upon previous works but offers multiple improvements. A new torsion spring-based approach to reduce the load on the robot's knees was developed and proved its effectiveness in multiple experiments. Additionally, existing approaches for handling the communication with the DXL servos were investigated. This led to the creation of the QUADDXL approach, which outperformed all existing solutions. The complete electronics of the robot were designed based on this approach, and its feasibility could be shown in experiments. Furthermore, the robot platform proved its usefulness in different research projects, theses, and the HSL competition.

In the future, the platform could be improved by using a different type of actuator. Using brushless DC servos with planetary gears would solve multiple issues at the same time. First, this gear system has a lower backlash and would, therefore, allow more precise control of the robot. Second, the different gear ratio makes the servo back drivable, which would allow falling with less damage. Third, they have higher rotational velocities, which would allow faster movements. They have become the standard for quadrupeds [BPK$^+$18], but are still comparatively expensive. A solution would be to use a 3D printable servo [UARM22]. This would also allow easier integration into the robot.

Chapter 4

Fall Management

The fall management for humanoid robots was investigated for two reasons. First, it was planned to train the RL policy on the robot's hardware. This requires the robot to withstand many falls during training and then return to a usable upright pose. Although it turned out later to be not feasible due to the long training time (see Chapter 6), it was also necessary that the robot could execute the trained policy for experiments without breaking. The platform's robustness was increased by adding elastic elements to the hardware. Still, additional software is required to bring the robot into a safe falling pose and let it stand up again. The second reason is that one topic of this thesis is bipedal locomotion. This is not complete without fall management, as falls can always happen, thus rendering the walk controller useless.

Therefore, this chapter presents a novel approach called Humanoid Control Module (HCM), which handles the fall management and other related tasks in a humanoid robot. The HCM is built up upon the DSD decision-making approach, which was partly developed during this thesis. Therefore, it will also be described.

This chapter is structured as follows: First, the related work is presented in Section 4.1. Then, the underlying decision-making approach is described in Section 4.2. Afterward, the HCM is presented in Section 4.3 and evaluated in Section 4.4. Finally, a summary is given in Section 4.5.

4.1 Related Work

This section will present the related work for this chapter. Since the HCM needs to take many decisions, we first investigate the different approaches that can be used for this (Section 4.1.1). Then, we will present other works in the field of software architectures for robots (Section 4.1.2). Finally, we will present other approaches for handling falls of humanoid robots (Section 4.1.3).

4.1.1 Decision Making

Decision-making (also called robot control) "is the process of taking information about the environment through the robot's sensors, processing it as necessary in order to make decisions about how to act, and executing actions in the environment"[SK16, p.308].

It can be grouped into four classes [SK16, pp.308-309]:

Deliberative Using a deliberative approach, the robot builds a plan based on the information it gathered from its sensors. After finishing the plan, it executes it. While it

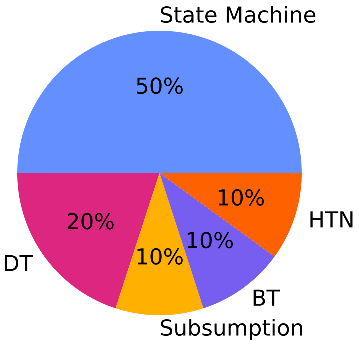


Figure 4.1: Used decision making approaches in the HSL. Note that one team (Rhoban) uses FSM and learns the parameters with RL. The data was taken from personally questioning all other Kid-Size teams during the RoboCup 2022.

allows sophisticated behaviors, it is typically not reactive to changes in the environment. This approach is also often described as sense-plan-act.

Reactive The reactive approach was developed to counter the central issue of the deliberative approach. Here the sensor inputs are more directly connected to the actuator outputs and do not rely on complex planning. This, naturally, results in fast reaction times on stimuli and many problem classes can be solved by it. Still, the approach is not usable for problems that require internal models or memory. This approach is also often described as sense-act.

Hybrid The hybrid approach is a combination of the ideas from the deliberative and reactive approaches. They consist of two components, one deliberative and one reactive. Therefore, it can perform complex behaviors while still reacting quickly to changes.

Behavior-Based The behavior-based approach uses a set of interactive modules. Each receives the data from the sensors and input from other behavior modules and creates commands to the robot's actuators or other behavior modules based on this. Therefore, the system is a decentralized decision network.

Every robot needs a decision-making component in its software to fulfill its task. Therefore, several approaches were developed in the history of robotic research. Many of them already exist for a long time, e.g., finite state machine (FSM) [Gil64]. Since many problems tackled by current robots are still of low or medium complexity, such classic approaches are still widespread (see also Figure 4.1). Therefore, we will also mention them here.

State Machines

FSMs are made of a finite set of discrete states with transitions between them [Gil64]. The agent changes its state based on the available information. It will act based on its current state and/or when the state changes. This approach is simple to understand and implement. Furthermore, it provides a clear state of the robot, which is helpful for a human operator. Still, complex behaviors require a high number of states (*state explosion*), thus leading to bad scalability. Adapting an existing FSM to new goals will typically require changing multiple states and transitions (low maintainability). Testing sub-parts of the decision-making is not directly possible as a FSM can not easily be divided.

The hierarchical state machine (HSM) [Har87] approach tries to address these issues by adding a hierarchical ordering to the state machine. This is done by allowing each state to be a state machine that can have transitions in itself or to other superstates. Thereby, a better organization is introduced, which divides the state machine into subparts. These can be more easily tested independently and are often better understood due to their order. Still, adapting existing HSMs to new tasks might force the developer to modify a large number of transitions [CÖ18].

Planning

Planning approaches define a current state, a goal state, and a set of possible actions. Then they plan a sequence of actions and execute them to reach the goal state. They are the classic example of a deliberative approach. The most widespread member of this class is Stanford Research Institute Problem Solver (STRIPS) [FN71], which is a basis of many newer approaches such as hierarchical task network (HTN) [Ero95]. These approaches allow complex behaviors but often rely on the closed-world assumption and are not reactive.

Subsumption

The Subsumption Architecture [Bro86] focuses on achieving reactive decision-making. It consists of multiple sub-tasks that can run parallel to each other. Each sub-task can subsume other tasks by overwriting their output. They, thereby, form a hierarchical order. This approach is not stateful, which makes it challenging to apply to problems that require knowledge of past decisions or information.

Fuzzy Logic

Fuzzy logic [Zad88] is a reactive approach that uses a non-Boolean logic to decide on the actions that need to be taken. This allows the use of decision rules that are closer to human concepts. To do this, the sensor data is first *fuzzyfied* by matching crisp values to fuzzy linguistic terms. Based on this, the inference engine produces a fuzzy output. This is then *defuzzyfied* into crisp actuator commands.

Decision Tree

The decision tree (DT) [Qui86] is probably the most simple and straightforward approach as it can be created by a simple nested combination of if-else clauses. A DT generally connects multiple decisions to a tree-like structure, resulting in a hierarchical ordering of these decisions. Besides its simplicity, it also allows a clear traceability of how the decision for an action was made. Still, it is a stateless approach, and all decisions need to be evaluated at each iteration of the decision-making. This can also lead to oscillating behaviors when the input data is noisy, as it is typically the case in robotics.

Behavior Tree

The behavior tree (BT) [CÖ18] is, in relation to the others, a recent approach that originates from the computer game industry, where it is used to program the behavior of non-player characters. As the name suggests, it uses a tree-like structure where the leaf nodes are either actions or conditions. All internal nodes control which actions are activated. This also allows to activate multiple parts of the tree simultaneously, thus enabling parallel execution of actions. While they are a powerful tool, they are complicated to program. Furthermore, the fact that decisions are taken implicitly by conditions in the leaf nodes makes the decision process less clear. The current state of a BT can also be unclear as multiple parts of it can be activated simultaneously.

4.1.2 Robot Software Architecture

The HCM creates a hierarchical decomposition of the system. This is generally desirable as it "decreases the overall system complexity and increases overall reliability" [SK16, p. 284]. The three-tier (3T) architecture [PBJFG+97] tries to achieve this and is often used for autonomous robots. As the name suggests, it decomposes the system into three hierarchical layers. The topmost layer is the *deliberative* tier which does long-term planning to achieve the agent's goal. Below this is the *sequencing* tier. Based on the input of the deliberative tier, it invokes skills. These are provided by the lowest layer, the *reactive* tier. A more detailed comparison of our approach to the 3T approach is given in Section 4.3.2

Since our approach is specific to humanoid robots, we will focus in the following on other approaches for this type of robot. Naturally, the teams in the HSL have produced multiple different architectures. The NUbots developed the *motion manager* approach for their *NUPlatform* [KW11]. It has a queue of motion jobs, provided by a higher tier, that are executed in parallel if possible. Based on sensor data, the motion manager can also detect falls. In this case, the current job is interrupted, and fall protection or stand-up motions are invoked. Different *movement handlers* might lock joints during the execution of jobs to prevent clashes between jobs. It is possible to query the currently locked joints from the motion manager, but no semantic state is provided. This locking mechanism was intended as a mutex but did not work [HFL+16]. This was because each movement handler needed to specify the locks by itself. If it did not do so correctly, the mutex was incorrect, and two jobs could actuate the same joint, thus leading to osculations in this joint which in turn then mostly led to falls. Furthermore, the construction of the movement handlers' locking was deemed difficult to maintain.

They replaced this with a new approach called *NUClear* [HFL+16], which they describe as "an extended form of subsumption logic" [HFL+16]. It uses no explicit locking or mutex mechanics, but skills can subsume others, leading to a hierarchy of skills. This also includes fall detection, which will just subsume all other skills.

A typical approach in HSL is to implement all motion skills in a single module that gets commands from a higher tier [FAFB17] [DHW19]. The mutex problem is thereby implicitly solved as only one module controls any joints. Still, such a tight coupling has the disadvantage that exchanging motion skills is complicated as it is not modularized.

Some teams are running such a single motion module on a dedicated microcontroller [RWA+19]. The advantage is that this allows a more direct connection to the sensors and actuators, as no additional hardware interface is needed. Thereby, the latency in the hardware communication is decreased, leading to a more frequent control loop cycle. Still, this makes the development of motion skill software harder since debugging on microcontrollers is more complex. Furthermore, the performance of microcontrollers is limited. Thus, computationally expensive approaches like non-analytic IK solvers can not be applied.

Besides the RoboCup, the DARPA Robotics Challenge (DRC) was another competition that led to the development of architectures for humanoid robots. Here, the teleportation of humanoids was also part of the research. The team THOR used multiple FSMs that each control different kinematic chains of the robot, i.e., legs, arms, the head, and the body [YMV+15]. Additionally, a *Motion FSM* performs movement to stabilize the robot and act in emergencies.

4.1.3 Fall Handling

As already stated above, fall management is important to prevent damage to the robot. Its most crucial part is the detection of the fall, as it is the basis for invoking any counter or protective measurements. Typically, each motion skill will already try to keep the robot stable, e.g., by adapting the joint goals while walking using the ankle strategy [NN08] or for larger disturbances capture steps [PCDG06]. Still, this is only applicable for disturbances up to a certain threshold. If these are exceeded, the robot is not able to control itself. Thus, the only remaining option is to use protective measures, e.g., moving the body to make it fall on certain robust sections. To activate these, the fall needs to be detected first (Section 4.1.3), and then protective measurements need to be invoked (Section 4.1.3). Standing up skills to bring the robot back to a controllable pose are discussed in Chapter 5.

Fall Prediction

Multiple different approaches, using different modalities, have been proposed to predict falls. Ruiz-del-Solar et al. [RdSMPT10] used a virtual attitude sensor (VAT) to detect falls. The VAT uses a Kalman filter [Kal60], which computes the attitude towards the ground and its derivative. This is compared to the ankle joint position, and a fall is predicted if a threshold is exceeded. Kalyanakrishnan et al. [KG11] used a learned decision list of multiple thresholds to classify if the robot is falling or has fallen. As input, they used simulated joint encoders, FPS, and a gyro to compute information about the CoM and how the feet are in contact with the ground. They did not differentiate between the different directions in which the robot is falling and performed all experiments only in simulation.

Muender et al. [MR17] used a model-based approach to predict falls as early as possible. They assumed that a fall would only happen if an external force acted on the robot, as the walking would be generally stable. While this might be a valid assumption for a production system, it is not true for our planned use case, where the robot is still learning to walk. A typical problem of unstable walks is an increasing oscillation that slowly leads to a fall. Furthermore, their approach was only tested on the NAO robot, which has comparably large feet and, therefore, a large support polygon. The results might not translate well to robots with smaller feet.

Wu et al. [WYC+21] used a support vector machine (SVM) classifier to predict falls on a bipedal, but not humanoid, robot. The classifier used the angular velocity from the gyroscope and the angular acceleration, which was computed by differentiating the angular velocity.

A related research topic is detecting falls in humans. The typical use case is the detection of falls for elderly people so that emergency services can be informed. Different approaches with different modalities have been investigated [RP19]. Abeyruwan et al. [ASSV16] developed a novel approach that they tested on humans and humanoid robots both. They used a two-layer approach which classifies in the first layer if a fall is happening and in the second in which direction. The classifiers were built using neural and softmax regression networks. Since they used, among others, a compass sensor, this would not directly be applicable in the HSL.

Protective Measurements

Protective measurements aim to minimize damage to the robot by either minimizing the impact of the fall itself or by making contact with the ground with certain areas of the robot so that damage is minimized. To do this, *Ukemi motions* that originate from martial arts can also be applied to humanoid robots [FKK+02]. They combine multiple strategies. They lower the center of gravity during the fall to reduce the overall impact. Furthermore, they increase the area on which the impact is distributed as well as the time. Finally, they direct the impact on specific parts of the robot that can withstand it best. Another approach uses planning of the contact points [HL15].

Alternatively, the hardware can be modified to increase robustness against falls. More information on this is provided in Section 3.1.2.

4.2 Dynamic Stack Decider

The DSD was originally targeted at developing high-level soccer behaviors for RoboCup, and early predecessor implementations were used without formal specification of the framework since 2015. The final specification and evaluation of the framework were done during this thesis together with Martin Poppinga and published in [PB22]. During this work, the domain specific language (DSL) (see Section 4.2.2) was created by Martin Poppinga. The rest of the work was done by us equally. A clean reimplementation of the DSD library ROS 1 package was done by Timon Engelke and Finn-Thorben Sell. The adaption to ROS 2 was done by Jörn Griepenburg, Timon Engelke, and myself.

The DSD was developed to be usable in dynamically changing domains like the HSL while still being simple to implement and use. In terms of usage, the maintainability and extendability of a behavior over more extended periods of time were especially important. We identified a series of design aspects that we needed to include to achieve the framework's goals. Naturally, this means classical software development goals such as code reuse, modularization, and maintainability. These are put on a more abstract level where they regard decisions and actions in the DSD instead of lines of code. Additionally, the agent's current state must be clear at all times. This means, on the one hand, that the framework is stateful so that decisions are based not only on the current input but also on past decisions. However, it also means that the user can clearly know the agent's state at all times. This simplifies debugging and understanding the agent's behavior. Still, the robot should be able to change its state quickly to adapt to the dynamic environment. This typically requires many checks to determine if previous decisions are still correct under the current conditions. This should be able to be done quickly and without overhead for the developer.

With these goals in mind, the DSD was developed. Its main component (and name giver) is a stack-like structure that orders all currently active parts of the behavior hierarchically. It is assumed that the DSD is called periodically in a control loop. The frequency of this loop is not defined and can be adapted to the domain as well as the computational resources. Each time that the DSD is called, the elements on the stack may be reevaluated, and the topmost one is executed. Based on its output, modules on the stack might be removed or added. The elements on the stack are divided into decision elements (DEs) and action elements (AEs) (see Section 4.2.1). These are programmed independently. The DSD is then created by connecting these to a directed acyclic graph (DAG) by the DSL (see Section 4.2.2). All data for the DSD is aggregated in a blackboard which is shared between all elements.

4.2.1 Elements

Decision Elements

A DE is a representation of one logical decision that can have a discrete set of semantic outcomes. It does not perform any actions by itself and does not directly change anything

\$	$\$Name$ Name of a decision element
@	$@Name$ Name of an action element
$-- >$	“$ReturnValue$” $-- > \$, @Name$ Defining the following element
#	$\#Name$ Defining or using a sub-tree
+	$\{\$, @\}Name + param : p\_value$ Gives initial parameters to the element

Table 4.1: Definition of the symbols used in the DSL. [PB22]

in the agent. The complexity can differ from a simple if-then clause to a multilayer perceptron (MLP). When the DE is called, it makes the decision based on the current information in the blackboard and returns a string representation of the decision. For example, a DE could decide which role a robot in a HSL game should currently take based on the data it has about the game. This decision would then return either *Goalie*, *Defender*, or *Attacker*. Other decision elements would then specify how each of these roles should act. The DAG would then contain the information which DE needs to be pushed next onto the stack.

Besides the normal execution of DEs that are on top of the stack, they can also be *reevaluated* while they are inside of the stack. In this case, the decision will be executed normally, but the result will be compared to the last result that this decision had. If they are not identically, it is assumed that all later taken decisions are not valid anymore. Therefore, all elements on the stack above the reevaluated decision are removed, and new elements are pushed based on the new decision outcome. For example, a robot might have decided to be the goalie at the start of the game. However, when all other teammates are removed due to penalties and the other team has scored more goals, the agent reevaluates this decision from earlier. It now becomes an attacker as this is the only chance to still score goals and win the game. Each DE can decide if it wants to be reevaluated or not. This decision does not need to be static but can also depend on the information in the blackboard. For example, the role decision from above might only want to decide on the robot's role at specific points, e.g., when there is an interruption in the game, and the robots organize themselves.

Action Elements

AEs do not perform any decisions but only execute something that changes the environment or the agent itself, for example, the execution of a kick. Since many actions take a certain time for execution, the AE stays on top of the stack until it removes itself or is removed due to a reevaluation.

4.2.2 Domain Specific Language

As described above, the DEs do not directly define which element is pushed next onto the stack. Therefore, an additional description for the DAG that connects the elements based on their outputs is necessary. In earlier versions, this connection was directly written in code. This approach proved to be cumbersome and badly maintainable. Therefore, a DSL was developed that describes these relations in a clearly readable and arranged format. For this, a text file is written that specifies the modules and their connections with certain symbols (see Table 4.1 and Figure 4.2). This file is then used

```
#Kick
$InKickDistance + kick_threshhold:0.1
  "Yes" --> @KickBall
  "No" --> @GoToBallDirect

#Attack
$ClosestPlayerToBall
  "Yes" --> #Kick
  "No" --> @Wait

--> SampleBehavior
$RoleDecision
  "FieldPlayer" --> $BallPositionAvailable
    "No" --> [...]
    "Yes" --> $DefendAttackDecision
  "Defend" --> $BallInOwnHalf
    "No" --> @Wait
    "Yes" --> #Attack
  "Goalie" --> [...]
    [...] --> #Kick
```

Figure 4.2: Exemplary DSL description of a simplified DSD for the HSL. Two subtrees are defined with the names *Kick* and *Attack*. The start point is the *SampleBehavior*. First, the robot's role is decided, and then further decisions are taken based on this. The [...] means this part was omitted for brevity. [PB22]

to automatically generate the DAG when the DSD is initialized (see Figure 4.3). The DSL also supports giving parameters to elements and sequences of actions.

4.2.3 Call Stack

The stack of the DSD works similarly to typical stacks in a programming language, but in a DSD stack, each element can be accessed. Therefore it is not a stack in the narrowest sense. Still, the name is used as it depicts the general idea. An exemplary stack is depicted in Figure 4.4. Additionally, an overview of how the stack and its elements are used is shown in Figure 4.5. The stack is initialized using its root element during the start and whenever an external interrupt is called. This interrupt can be used to reset the DSD externally, for example, for manual resets. Then, each time the DSD is called, a check for the reevaluation of the elements on the stack is performed, starting from the bottom. If a decision outcome changes during reevaluation, all elements above are cleared. Otherwise, the reevaluation procedure finishes when the top DE is reached. Then the topmost element is executed. Now all pushing and popping operations on the stack are performed until the topmost element is an action which is then executed. This ensures that an actual action is performed during each control loop cycle, and no cycles are spent on changing decisions on the stack. It is reasonable to perform multiple decisions in one cycle as they change nothing and are (principally) instantaneous.

4.2.4 Implementation

The implementation is done in Python as a ROS 2 package. This allows simple integration into the ROS 2 software stack, including visualization in RQT. Python was chosen because it is simple to write and allows access to many libraries, e.g., SciPy or PyTorch. Using Python does not pose any performance issues as the decisions of the DSD are either not computational heavy itself, i.e., being simple if-else clauses, or utilize libraries with C implementation for more complex tasks, e.g., MLPs with PyTorch. Each DE

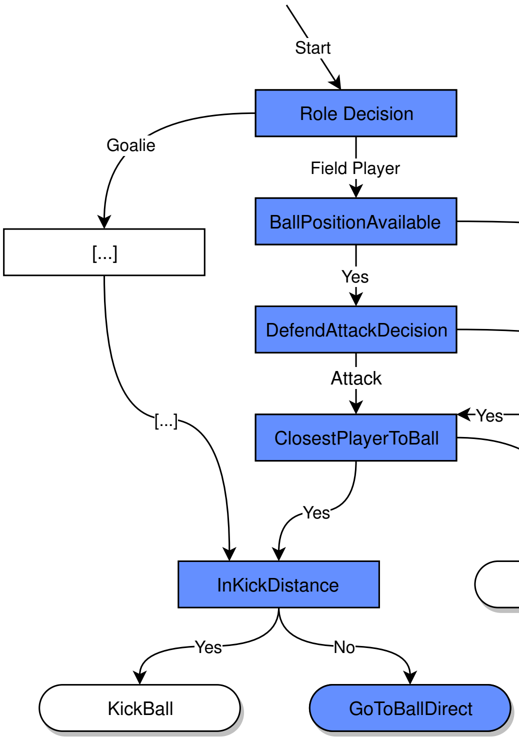


Figure 4.3: Simplified representation of the DAG that is generated by the DSL in Listing 4.2. An exemplary chosen path is marked (**blue**). The agent decides on being an attacker and going to the ball since it is not in kick distance. It can also be observed that there are multiple paths through the DAG that result in the same action. Therefore it is necessary to know the complete stack of previous decisions to know the state of an agent. [PB22]

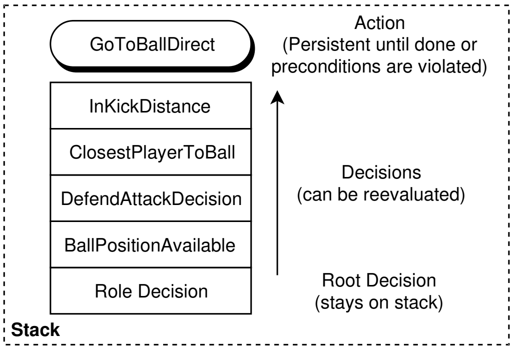


Figure 4.4: Exemplary stack for an agent after following the decisions displayed in Figure 4.3. Each time step, all DEs are checked if they need to be reevaluated, starting from the bottom of the stack. If no decision outcomes changed, the AE on top of the stack is executed. [PB22]

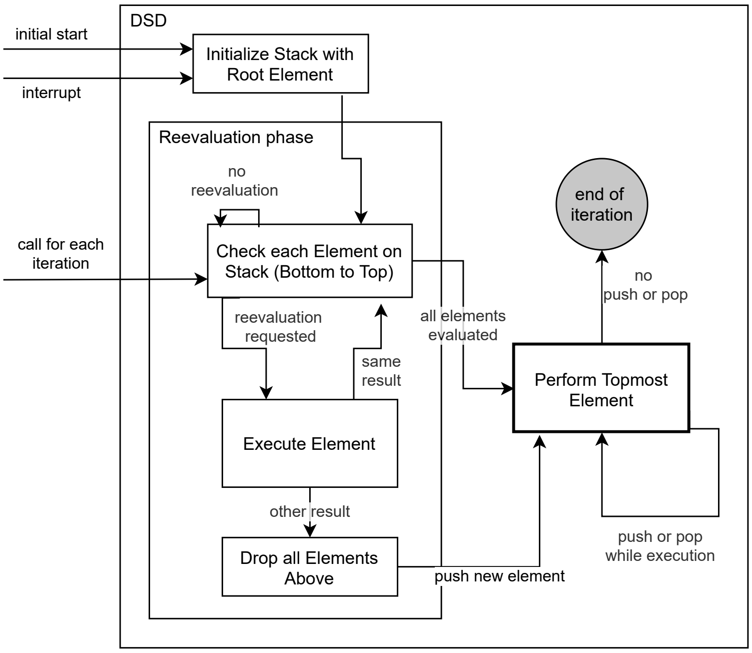


Figure 4.5: Graphical representation of the control flow of the DSD. [PB22]

and AE is implemented as a class that inherits from an abstract class and implements methods for execution and reevaluation. When the DSD is started, the element names in the DSL are matched to the class names, and no further work from the user is required.

4.3 HCM

> The basic idea of the HCM was already developed in my master's thesis [Bes17] but only a very basic version based on a FSM was implemented, it was not yet formalized and lacked multiple of the functions it has now. It was completely redesigned and rewritten using a DSD. Furthermore, the evaluation of different falling detection approaches was completely done during this doctoral thesis.

Humanoid robots promise easy integration into everyday environments due to their similarity to humans. Today, they are still primarily present in research. They have not reached the level of a typical consumer product, in contrast to wheeled robots, which are, for example, used for vacuum cleaning. This difference between humanoid and other mobile robots arises on the one hand from the higher costs due to more complex hardware but also because humanoids are harder to control from a software perspective. Wheeled robots are constantly in a stable and, therefore, controllable state while driving on flat terrain. On the other hand, humanoids can fall easily, even when moving on flat terrain. If this happens, the robot can not perform a sensible action before it achieves a standing pose again. Additionally, the robot may need to detect falls while they happen to minimize the damage by doing counter measurements. These cases must be handled in all software parts that control the robot, making it more complex. Therefore, most of the research in mobile robotics is done with wheeled robots. Additionally, standard implementations for wheeled robots, e.g., for path planning, can not be directly applied. Even manual moving of the robot with teleoperation only works if the human in the same location can pick up a fallen robot.

The HCM solves this issue by introducing an additional abstraction layer that allows other software parts to control the robot as simply as a wheeled one. Thus, teleoperation, as well as the usage of software for wheeled robots, is enabled. Additionally, it allows autonomous reinforcement learning of motions directly on the robot (see Chapter 6) since it can reset the robot back to a standing pose and prevent damage during falls.

4.3.1 Requirements for Abstraction

We identified six issues that need to be addressed to achieve the abstraction mentioned above. Two of those (hardware problems and manual stops) are also present in wheeled robots but have different implications for humanoids.

Hardware Issues During the usage of a robot, different types of hardware issues may occur suddenly at any time. These can be the complete failure of a single component, e.g., a sensor or actuator, as well as losing connection to one or multiple devices due to broken cables. In our experience, connection breaks are especially present on Dynamixel servo-based robots due to impractical cable routing, an issue that has only been addressed in the never X-Series (see Section 2.2.1). In contrast to wheeled robots, this can damage the robot further as it can lead to falls or prevent the execution of fall countermeasures. Therefore, such issues must be autonomously detected, and the robot must stop until a human resolves the issue. While this would only require a cut of the actuator power on wheeled robots, it is not feasible on humanoids as it would result in an uncontrolled fall. In our experience, this also simplifies the debugging of issues that are triggered by this in other parts of the software, e.g., having an unstable walking due to not receiving sensor updates.

Manual Stops Manual stops are necessary to ensure the safety of the robot and its environment. This situation is similar to having a hardware issue, as the robot must be stopped completely until the issue is resolved.

Fall Management Falls on bipedal robots can not be avoided entirely. Even if all motions of the robot by itself are perfectly stable, an external force can occur, e.g., by impacts from humans or other robots. These fall situations need to be managed. Preferably, a counter measurement, e.g., a capture step, would be executed to prevent the robot from falling and bringing it back into a stable pose. If this is not possible, the robot can at least perform safety measurements, e.g., moving the head to a safe position, to minimize the damage from the imminent fall.

Stand Up When a fall has happened, the robot needs to get up so that it can again be controlled by the high-level behavior. Depending on the use case, the time until the robot returns to a standing pose might also be of interest. This can be the case in competitive domains, e.g., RoboCup soccer, or in time-critical situations such as rescue operations.

Joint Goal Mutex The joint actuators of a humanoid robot typically have multiple uses, e.g., the arm joints can be used during grasping but also to stabilize walking. They are, therefore, typically used by different software parts. This leads to problems when two parts try to control the same joint at the same time as PD controller will move to them alternately, creating jittery movements. Therefore, a mutex-like approach needs to be applied to limit the usage of different joints.

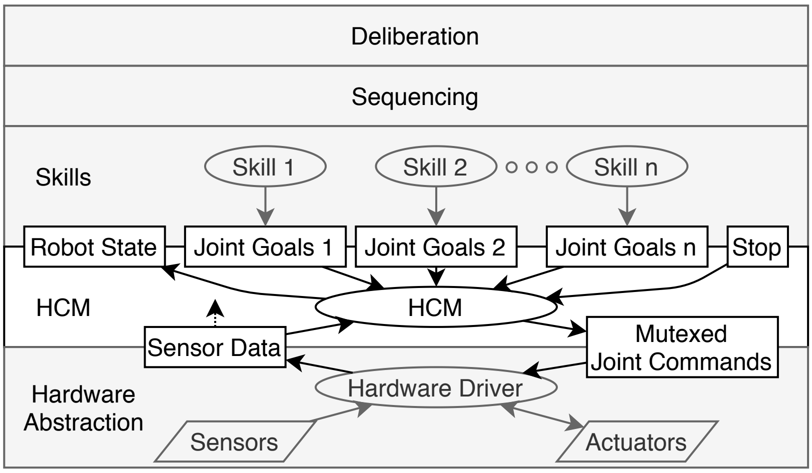


Figure 4.6: Location of the HCM in a classical 3T approach. It needs to be below the skill layer so that it can mutex the joint goals and react quickly when falls happen. The HCM provides a semantic robot state and a manual stop, which can be used to communicate with high-level behavior. The rest of the layers (**gray**) are not changed, allowing simple migration of existing software modules. [BZ20]

Semantic Robot State Although the goal of HCM is that higher-level software parts can handle the robot as a wheeled one and do not need to be aware of the state of the robot, it can be beneficiary in some situations to provide this information to higher-level behavior parts. One important example is the localization which can be improved by informing it about falls. The localization can then add the possible rotation of the robot during the fall into its belief. Providing information about the robot's state to the user allows simplified debugging.

4.3.2 Architecture

Overall Architecture

Fast reactions are necessary for the above-described tasks, especially for the fall handling. Therefore, the HCM can not be placed on a deliberative or sequencing tier (see Section 4.1.2). Furthermore, this would hinder the usage of standard software for wheeled robots that are located on these tiers. Thereby, the HCM needs to be on a lower tier.

Additionally, the HCM should implement a mutex for the actuator control to ensure that only one skill can control an actuator at any time point to avoid issues like in the NUPlatform (see Section 4.1.2). This could be implemented by using subsumption, as it was done in the NUClear framework (see Section 4.1.2). Still, this would not provide a clear semantic state, which is also one of our requirements. Instead, we propose a four-tier architecture where the HCM is an entirely new layer below the skills layer and above the hardware abstraction (see Figure 4.6). Thereby no changes to the other layers are required. All actuator commands are routed through the HCM, which allows enforcing a mutex while having a clear semantic state. The targeted middleware ROS is based on asynchronous message passing (see Section 2.3). This allows routing the joint commands through the HCM while the sensor data can be directly passed to the higher levels as it does not require a mutex. Thus reducing the introduced delay in the control loop cycle.

Choice of Decision-Making Approach

The HCM decides in which state the robot is based on two pieces of information. First, it uses the sensor data from IMU, foot pressure sensors, and joint position encoders to determine its external state, e.g., being picked up. Second, it uses the joint goals from the different skills to determine the robot's internal state, e.g., walking. This state is then used to determine which joint goals are being forwarded and if fall management actions need to be executed. Additionally, it is published as information to the rest of the software.

Knowing the correct state is a typical decision-making problem, and multiple different approaches to this already exist (see Section 4.1.1). The first basic version of the HCM used a FSM as a basis, but as the HCM was developed further, the number of states and transitions increased dramatically. This typical issue of FSMs is also called *state explosion* and leads to low maintainability and understandability. Additionally, the different states of the HCM have a hierarchical ordering, e.g., *Falling* is more critical than Walking. This cannot be clearly represented with a normal FSM.

While an HSM does allow hierarchical ordering in general, it is not well fitted to a scenario like this where the number of hierarchical levels is high. It only allows traversing through hierarchical layers one by one, thus enforcing recurrent conditional clauses. For example, if the prioritization of three states *Stopped>StandinUp>Walking* needs to be encoded, it is necessary to check if the robot is stopped in the *StandingUp* and *Walking* state.

This issue is not present in a decision tree, but as this approach is stateless, other issues arise. The state can only be decided on the current sensor data and not on the previous state. This can, for example, lead to issues when the robot gets a brief push and thus needs to go into the state *Falling*, but in the next cycle, these sensor readings are not present anymore, and the HCM has now forgotten that it is falling.

A behavior tree would allow the creation of a strong hierarchical relationship while remembering previous decisions. We still decided against using one since it is unnecessarily complex for this task. Furthermore, the extraction of the state can be difficult since multiple parts in the behavior tree can be active at the same time.

Based on this reasoning, we decided on using the DSD (see Section 4.2). It can nicely encode many levels of hierarchy while having a clearly defined state that is represented by its stack. Additionally, it is also used for the high-level soccer behavior of the robot and thus lowers the entry barrier for users of the complete software stack.

4.3.3 Implementation

Software Stack Integration

The concrete implementation of the HCM was targeted for the RoboCup soccer domain. Therefore, it includes some small things which are specific to this domain, e.g., handling of the kick. Still, it can generally be used and also easily be adapted to other domain-specific tasks. In the following, its integration into the complete software stack is described exemplarily.

Since the whole software stack relies on this middleware, the HCM is implemented as a single ROS node. It receives all inputs via either ROS message transfers or service calls. All high-level decisions are made by the *Body Behavior*, which represents the deliberative layer of the architecture. It decides where the robot should go. This navigation goal is then processed by *Nav 2*, the ROS 2 standard package for path planning. It splits the task into a global and local path planner. For both, different standard implementations exist which are targeted at wheeled robots. While we were able to navigate using these standard planners, a specialized local planner was later implemented, especially for

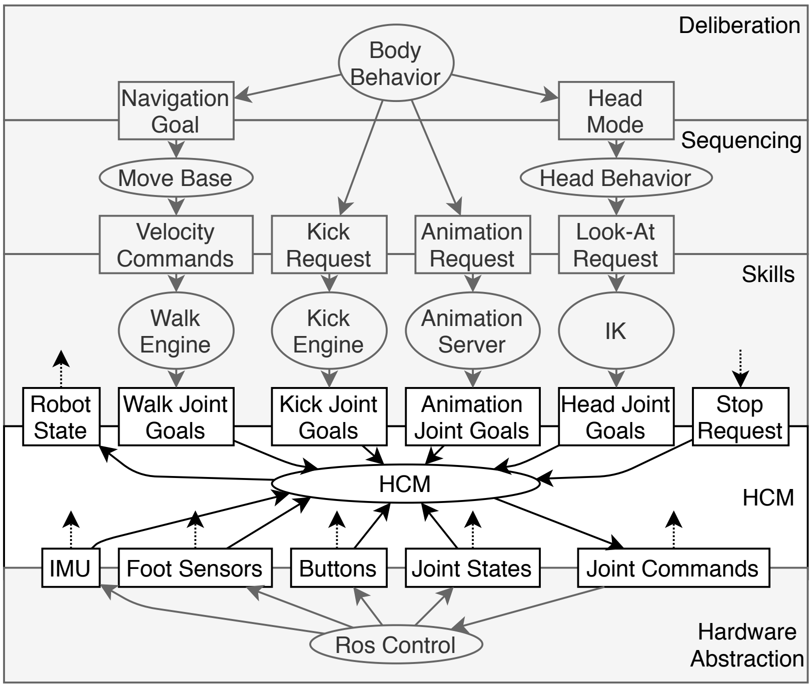


Figure 4.7: Visualization of the HCM with the RoboCup software stack grouped into the different layers. ROS nodes are depicted as **ellipses** and topics as **rectangles**. The dotted arrows represent a direct connection to any other node. The deliberation and sequencing layers can be replaced by a simple interface for manual teleoperation. [BZ20]

faster positioning at the ball before kicking. The navigation sequences the goal location to velocity commands which are normally directly translated to wheel speeds. In our case, the walk engine skill computes the corresponding joint goal positions to match this speed. Implementation of a footstep-based planner would also be possible but was not necessary as the domain is flat without obstacles that can be stepped over.

Additionally, the *Body Behavior* can set a head mode that specifies which information should be preferably obtained by the camera, e.g., the ball position or field features for self-localization. The *Head Behavior* produces Cartesian positions, which should be looked at. These are then again translated by an IK into head joint goal positions.

The *Body Behavior* can also directly invoke a kick with a specified direction, which is then translated to joint goal positions by a kick engine (see Section 5.5). Additionally, a set of keyframe animations, e.g., cheering after scoring a goal, can be directly invoked and are then translated into joint goals by the *Animation Server*.

Altogether, the HCM receives the joint goals from four different skills. Additionally, it may also get stop requests via a ROS service call.

HCM Node Implementation

The implementation of the HCM node consists out of two parts. The DSD runs at a fixed frequency and decides, based on the latest sensor information, if the state needs to be changed. The ROS subscribers have asynchronous callbacks for all joint goal positions and directly forward the goals to the hardware if the current state allows it. This way, the introduced delay by the HCM is minimized. While the DSD part of the HCM is written in Python, the ROS node is written in C++. This was necessary since

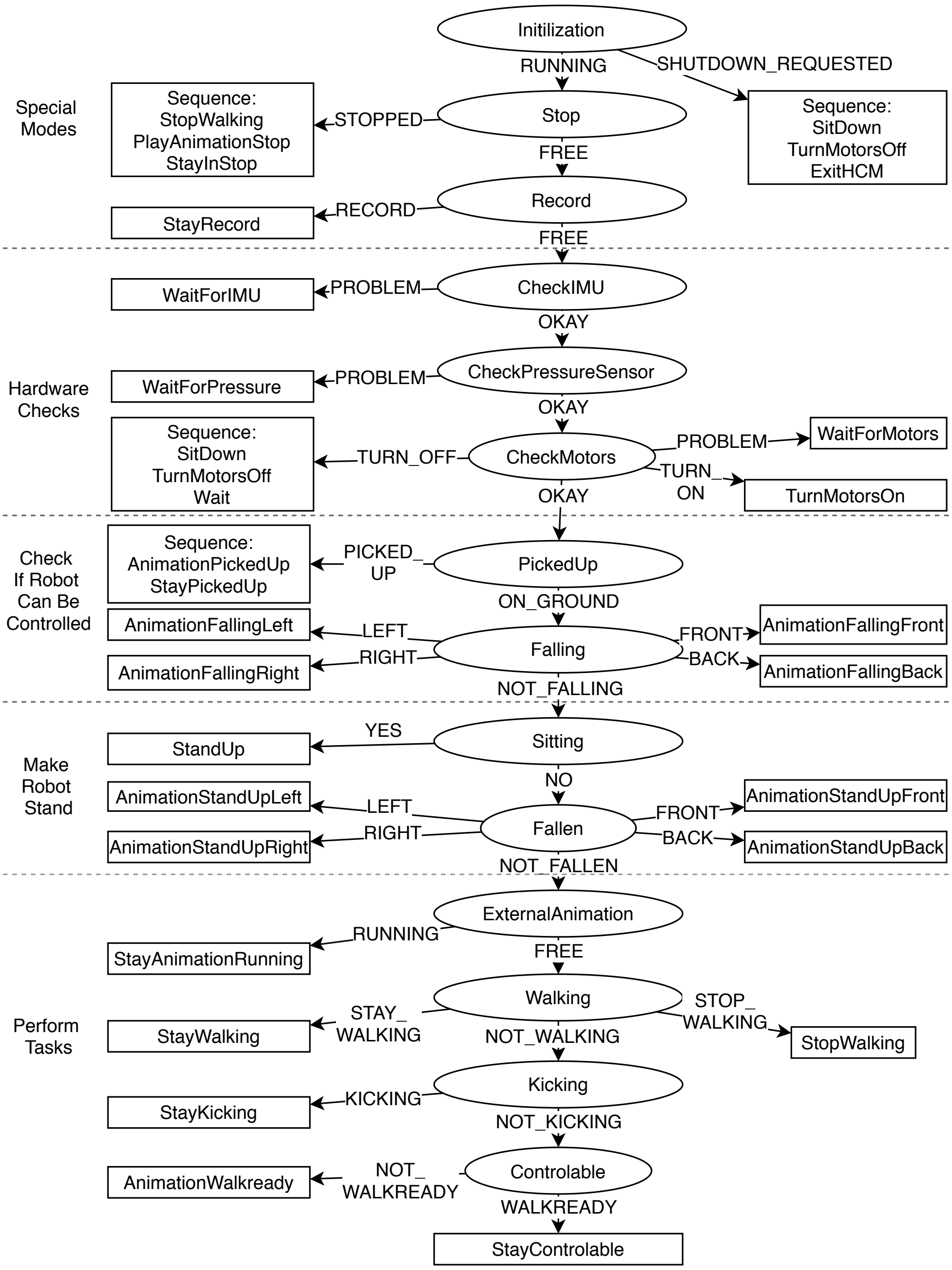


Figure 4.8: The DAG of the HCM's DSD. The elements are grouped for additional clarity. DEs are represented by **ellipses** and AEs by **rectangles**. [BZ20]

the Python executors in ROS 2 were not performant enough (see Section 4.4.2 for more details).

The HCM provides the following states: *Animation Running, Controllable, Falling, Fallen, Getting Up, Kicking, Picked Up, Record, Shutdown, Startup, Stopped, Walking.* Most of these states are self-explanatory. The *Animation Running* state means that a keyframe animation is currently moving the robot's joints. The robot is *Controllable* if it is idle and standing upright in a pose from which it can either start walking or performing other motion skills. If a human is holding the robot above the ground, it is in the *Picked Up* state, which prevents movements and signals to the localization that the robot might have been moved to a different place (kidnapped robot problem). The *Record* state is only used during the recording of keyframe animations on the robot, i.e., to deactivate the fall detection. The *Stopped* state reflects that the robot's movements were currently disabled due to a manual stop request from a human.

The DSD consists of 14 decisions that are ordered hierarchically and can be grouped into five objectives (see Figure 4.7). All of them are reevaluated every time unless some actions block reevaluation, i.e., during a fall.

The first group of decisions checks if special modes are requested. This can be a shutdown request, where the robot first sits down before powering off its servos. It can also be a stop request, which stops the walking and lets the robot stand until the stop request is revoked. Additionally, the HCM can be put into a *record* mode, which is necessary to record keyframe animations on the robot. In this case, the HCM will only let through the goals of the recording software, but it will not perform any fall detection as it is expected that a human is holding the robot while recording poses of it. The fall detection must be deactivated in such a scenario as the robot might be put into unstable poses during this process.

The second group consists of a range of hardware checks. Here, the time since the last value was successfully read from a sensor as well as when the sensor changed its value the last time is used to determine if all sensors are working. This includes the IMU, the foot pressure sensors, and the joint feedback. Additionally, the servo motors are maybe powered off if they got no new commands for a long time. This is done to prevent damage to the robot while idly standing for too long after being forgotten by a human operator. Of course, the robot first sits down before it turns off its motor power. If new commands are received, it will turn the power on again and stand up again (based on a later decision).

The third group checks if the robot is currently in a state where it can move. The robot can recognize if a human operator has lifted it up based on the IMU and foot pressure sensor values. In this case, it will stop moving and go into a standing posture. This allows easier carrying due to no movements and depositing due to a stable standing pose. Additionally, movements introduced by the carrying human might trigger the fall detection, leading to movements that might hurt the operating human. In case the robot is on the ground, and a fall is detected (see also Section 4.3.4), the HCM recognizes in which direction the robot is falling and performs an appropriate movement to prevent damage to the robot, i.e., moving the arms to a pose so that the robot falls on the soft parts.

The fourth group checks if the robot is upright. It could either be sitting, due to previously sitting down, or due to being started in this posture, or it could lie on the floor, after a fall or due to being started this way. If one of these is the case, the corresponding motion is performed to bring the robot back into an upright position where it can be operated (see Section 5.4 for details on how this is done).

The fifth group is responsible for managing the high-level tasks, playing keyframe animations, walking, and kicking, that should be executed. Finally, it is evaluated if the robot is in the so-called *Walkready* pose, which it should always go to while it is idle because it is stable and allows to quickly start walking or perform various other motions.

The state of the HCM can be visualized during runtime using the default DSD visualization rqt plugin.

4.3.4 Fall Detection

Fall detection can be formulated as a typical classification problem. This generally uses a training dataset D of tuples of classifier input ($x_i \in X$) and correct result ($y_i \in Y$), which are typically provided by human labeling [dMP18, p.3]. A classifier (C) is a function that maps the inputs (X) to the labels (Y).

$$D = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\} \tag{4.1}$$

$$C : X \rightarrow Y \tag{4.2}$$

In our case, the input represents the observed robot state (o), and the output is the corresponding true direction of the fall (d^*).

$$x_i = o_i \tag{4.3}$$

$$y_i = d_i^* \tag{4.4}$$

The sensor data is a vector of real numbers whose length (n) depends on the type of modalities we chose.

$$o_i \in \mathbb{R}^n \tag{4.5}$$

The resulting direction is either one of the four natural sides in which a vertical body can fall or the information that the robot is not falling.

$$d_i = \begin{cases} 0, notFallen \\ 1, fallenFront \\ 2, fallenBack \\ 3, fallenLeft \\ 4, fallenRight \end{cases} \tag{4.6}$$

Therefore, our classifier (C) is a function with the following mapping.

$$C : \mathbb{R}^n \rightarrow \mathbb{N}[0:4] \tag{4.7}$$

In the typical usage of classifiers, two consecutive data points (x_i, y_i) and (x_{i+1}, y_{i+1}) are not related. Furthermore, the classifier's quality is measured based on how correctly it predicts the label from the given data. In our use case, we have a slightly different objective. The robot is getting a constant stream of observed states (O_d^e) at a high rate which are in a temporal order and end at time point t_e with an end direction of d. The classifier can be solved at each control loop cycle to decide if the robot is currently falling or not.

$$O_d^e = [o_{t_0}, o_{t_1}, ..., o_{t_e}] \tag{4.8}$$

The time point t_e can either be the impact after a fall or just the point where the robot was deactivated if no fall happened. The correct classification of each of the time points is of less importance as long as a fall is detected early enough and no falls are predicted while the robot is stable. Thus, false positives, which would unnecessarily invoke fall counter actions and disturb the current motion of the robot, are worse than false negatives, which might be corrected in the next control loop cycle. Still, having too many false negatives after each other would result in the robot not detecting the fall and crashing into the ground. Therefore, we introduce a better measurement of the continuous performance of the classifier, the *lead time*. This is defined as the time difference between the successful detection of the fall and the impact on the ground. To express this formally, we first define the times for the first true positive (t_{tp}) and for the first false positive (t_{fp}) for a classifier (C), given a stream of sensor data (O_d^e) leading to a fall in direction d.

$$t_{tp} = \min\{t \in [0, t_e] | C(o_t) \neq 0 \wedge C(o_t) = d_t^*\} \tag{4.9}$$

$$t_{fp} = \min\{t \in [0, t_e] | C(o_t) \neq 0 \wedge C(o_t) \neq d_t^*\} \tag{4.10}$$

Then the prediction time (t_p) can be defined as follows.

$$t_p = \begin{cases} t_{tp}, t_{tp} < t_{fp} \\ t_e, else \end{cases} \tag{4.11}$$

The lead time (t_l) of O_d^e given $d \neq 0$ can then be computed as follows.

$$t_l = t_e - t_p \tag{4.12}$$

This also implies that the lead time is 0 if the fall is not detected correctly. If no fall is happening in O_d^e, t_l is undefined.

We define false positives (FPs) and true positives (TPs) only based on their first decision as we assume that once any protective measures are invoked, the robot will not decide again on the fall direction.

$$FP : t_{fp} < t_{tp} \tag{4.13}$$

$$TP : t_{tp} \leq t_{fp} \tag{4.14}$$

It would also be possible to use observations from the past ($o_{t_{i-m}}, ..., o_{t_i}$), but this is not further investigated as these data can directly be included in the current state, e.g., by using a complementary filter to get the orientation of the robot from the IMU data. Therefore, we will only discuss the observation at a certain time point.

There are various approaches to solving a classification problem. Typical approaches include: k nearest neighborss (KNNs) [FH89], support vector machines (SVMs) [BGV92], and multilayer perceptrons (MLPs) [Mur91]. The Hamburg Bit-Bots previously used a simple threshold-based (TB) classifier based on IMU data (see Algorithm 4). It has already shown its usability in multiple RoboCup competitions. Still, a more sophisticated classifier or additional modality could improve the lead time (see Section 4.4.3 for a comparison).

The TB classifier (Algorithm 4) was written by Florian Vahl.

4.4 Evaluation

This section evaluates different aspects of the presented approach. First, a qualitative evaluation of the HCM is given (Section 4.4.1). Then, the latency aspect will be further investigated (Section 4.4.2). Finally, different classifiers and modalities for fall detection will be compared (Section 4.4.3).

The DSD will not be evaluated quantitatively since this is not possible for an architectural approach. Still, the evaluation of the HCM shows that the DSD works well. Additionally, it has also been successfully used for the high-level RoboCup behavior. The stand-up skill will be evaluated in Section 5.4.3.

4.4.1 HCM

The HCM has been used successfully in multiple simulated and real RoboCup competitions and reduced the number of broken gears (see also Section 3.6.4). During the years of usage, the skill implementations have been changed by successive switching from keyframe animations to spline and RL based solutions (see Chapters 5 and 6). The HCM allowed not only a simple change of the module but also to have old and new skill implementations beside each other. This is practical as it offers a simple fallback solution while the new implementation is not stable yet. This shows that the HCM improves the modularization and maintainability of the architecture.

The goal of the HCM to allow a simple, wheeled-robot-like control was also achieved. It was possible to apply the default path planning pipeline of ROS as well as simple teleoperation. The main behavior of the RoboCup software stack could also be simplified since it did not need to take care of the fall management. Thus complex strategies could be implemented with low implementation complexity.

4.4.2 Latency

One important downside of the HCM approach is the added latency from the joint goal mutex, which adds one additional message transfer. In the following, we investigate how high this latency is to show if this disadvantage is acceptable.

Although the latency might increase with message size, e.g., for image messages, this is not relevant for this evaluation since the HCM joint mutex only affects the joint goal messages. Therefore, no experiments on different message sizes were performed, and the same joint command message was used in all experiments.

There are already multiple benchmarks and benchmarking frameworks available for ROS 2 [MKA16] [KPM$^+$21]. Still, we decided to perform our own experiments as the previous experiments did not use the EventExecutor or targeted different setups, i.e., with larger messages and lower control loop frequencies.

Influence of Idle Function

It was already described that the default Linux scheduling might interrupt nodes during execution, leading to various issues (see Section 3.6.3). Therefore, all experiments were performed on isolated cores. Furthermore, the idle function of the CPU was disabled to reduce latency introduced by it (see Figure 4.9). This was done with the following command.

```
echo 1 | sudo tee /sys/devices/system/cpu/cpu*/cpuidle/state*/disable > /dev/null
```

It is also possible to deactivate the idle function during kernel boot up by setting `cpuidle.off=1`, but this showed roughly a factor two higher latency than using the command above. The reasons for this are not clear. All experiments were performed on the ASUS PN51-E1 main computer of the Wolfgang-OP to ensure that the results are not biased due to a faster CPU, i.e., by using a desktop computer instead. The messages were recorded over a span of 100 s, after discarding the first 100 messages since these often contained outliers with high latencies due to the start-up process of the nodes.

Choice of Executor

ROS 2 provides different *executors* which run nodes and handle the message transfer. The issues with high CPU usage were already discussed in Section 3.6.3. Even though this issue makes only the *EventExecutor* usable for our system, we still compared it to the others to see if its lower CPU usage comes with an increase in latency. Furthermore, it is possible in ROS 2 to run multiple nodes in the same process, thus potentially

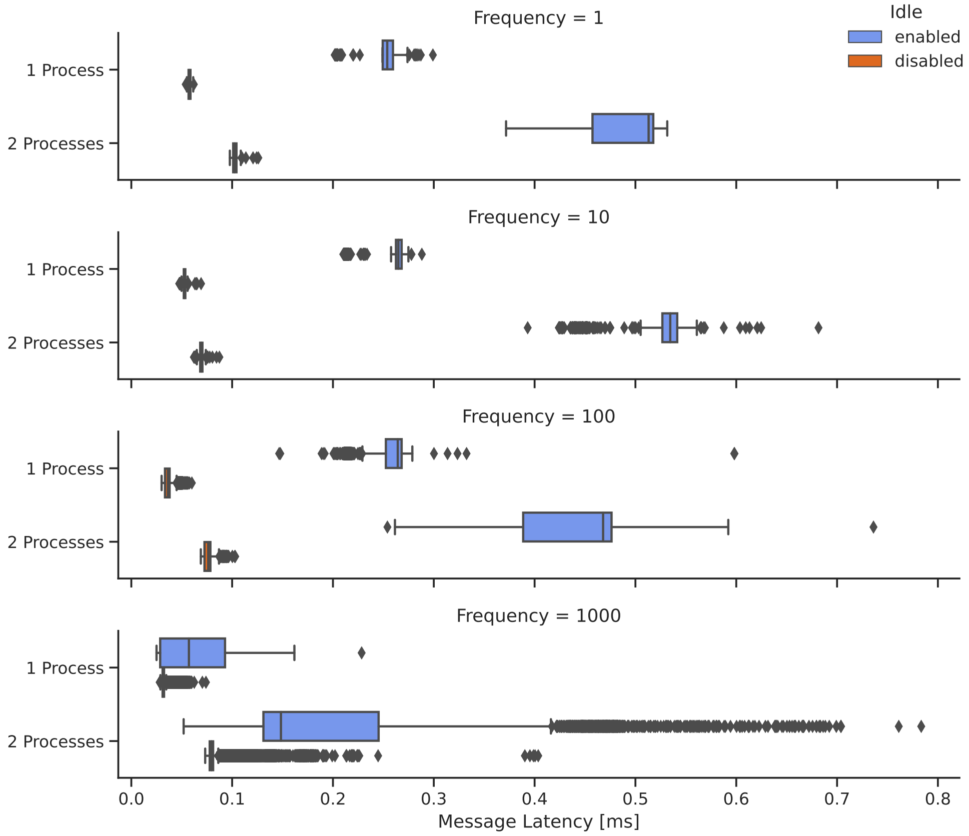


Figure 4.9: Message latencies for different publishing frequencies using a single process or two processes with the idle option enabled (**top**) or disabled (**bottom**). All measurements were conducted with the ROS 2 EventExecutor. The impact of disabling the idle function is larger for the lower frequencies. The 1,000 Hz seem to create enough load so that the CPU does not always go into idle mode, thus reducing the mean latency.

further decreasing the latency of message transfers. The event executor performs worse in separate processes but similarly with a single process (see Figure 4.10). The other executors perform similarly for both process configurations. Generally, running both nodes in the same process leads to lower latencies. This matches previous work [SvD19]. However, they did not test a single message latency but the overall time for processing their software. Therefore, times for copying message contents, especially for large image messages, play a bigger role.

Comparison to ROS 1

Additionally, we compared the results to a ROS 1 message transfer of the same message. No different executors are selectable here as ROS 1 does not have this concept. Still, it is possible to activate a *tcp_nodelay* option that should decrease latency. Therefore, we tested it with this option activated. Besides the executors, ROS 2 also has the concept of Quality of Service (QoS), which influences the latency. To keep the results comparable to ROS 1, we chose the default QoS settings, which are specially designed to mimic the ROS 1 behavior [20]. This setting also makes sense for our use case, as we do not want to lose messages. The results can be seen in Figure 4.11.

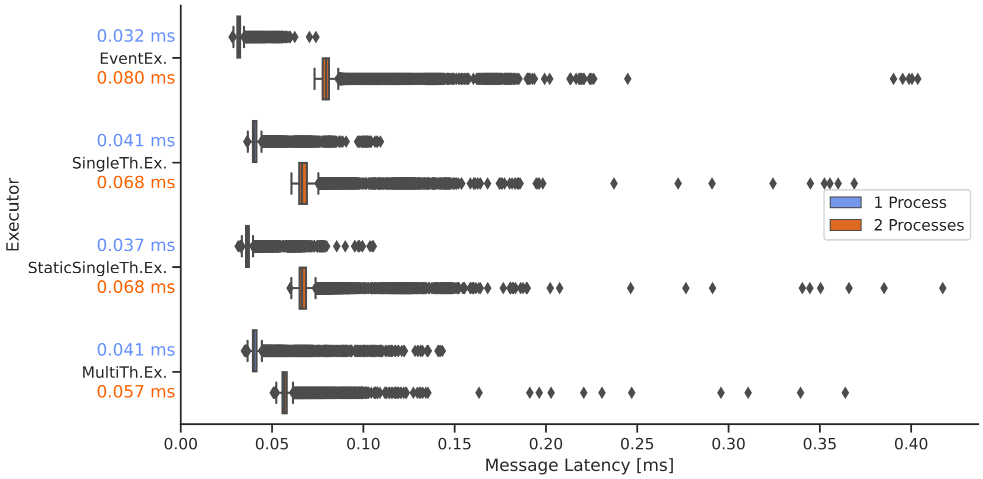


Figure 4.10: Message latency of different executors for ROS 2 C++ nodes running at 1,000Hz. Each is evaluated with both nodes running in the same process and with them running in different processes. Both are done on isolated cores. The mean latency is given on the left. Running both nodes in the same process leads to a smaller latency for all executors. The difference between the executors is not as large as between different numbers of processes.

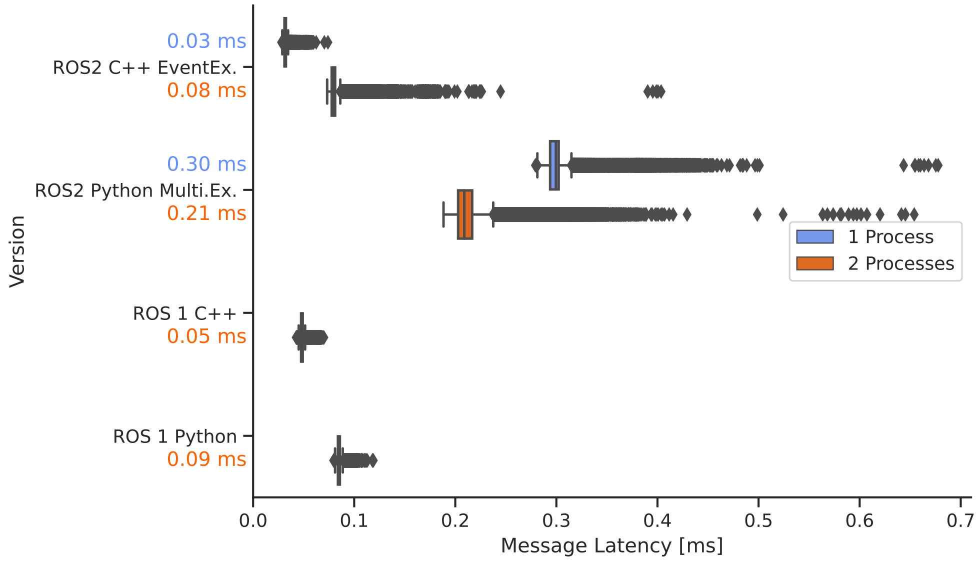


Figure 4.11: Comparison of the message latency of the C++ EventExecutor to the Python MultithreadedExecutor and to ROS 1. The mean latencies are provided on the left. Using C++, ROS 2 is only able to achieve better performances than ROS 1 if both nodes are run in the same process. Using Python, ROS 2 has significantly higher latencies than ROS 1.

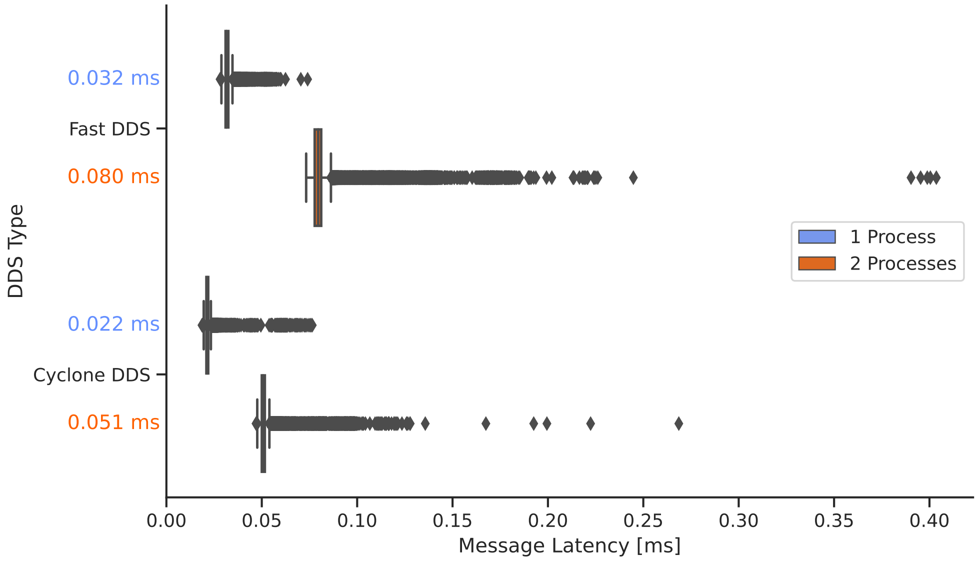


Figure 4.12: Comparison between different data distribution service (DDS) implementations using the EventExecutor at 1,000 Hz with a single (**top**) and two processes (**bottom**). All settings for the DDS were kept to the corresponding default. Cyclone DDS performs slightly better than Fast DDS.

Choice of DDS

It is possible to choose between different DDS implementations in ROS 2. The most commonly used ones are *Fast DDS* and *Cyclone DDS*, which are also open source. Cyclone performs slightly better (see Figure 4.12). This contrasts previous work by Kronauer et al. [KPM$^+$21] where they concluded that Cyclone DDS has the highest latency. They only tested frequencies up to 100 Hz, on different hardware and with a different executor. This could explain the difference in the results.

Integrated Evaluation

Another experiment was performed to see how well these results of our simple test scenario apply to the real world (see Figure 4.13). For this, the complete motion stack was executed on the robot with 500 Hz, and the robot was commanded to walk forward. Cyclone was used as DDS. The latency was measured directly inside the HCM for the received joint command messages that are sent by the walking. The results are compared to the simple publisher subscriber test from the previous experiment. Furthermore, different ways to run the HCM are compared as they significantly influence the latency. All of them use the EventExecutor. Executing the HCM without modifications results in high outliers with up to 4ms. As described in Section 3.6.3, it is necessary to configure the distribution of process to the CPU cores in a ROS 2 system with high-performance nodes. However, only applying taskset without isolating the CPU cores worsens the results. In this case, we gave the HCM three cores. If the cores are isolated, the result improves, and no message transfer takes more than 0.5ms, but the performance is still not comparable to our previous experiment. One reason for this is that ROS 2 runs all callbacks sequentially as default. Since the HCM listens to multiple high-frequency topics, the message from the walking will not be processed instantaneously. Therefore, we set the callbacks to be executed in parallel, but it did not improve the results. After

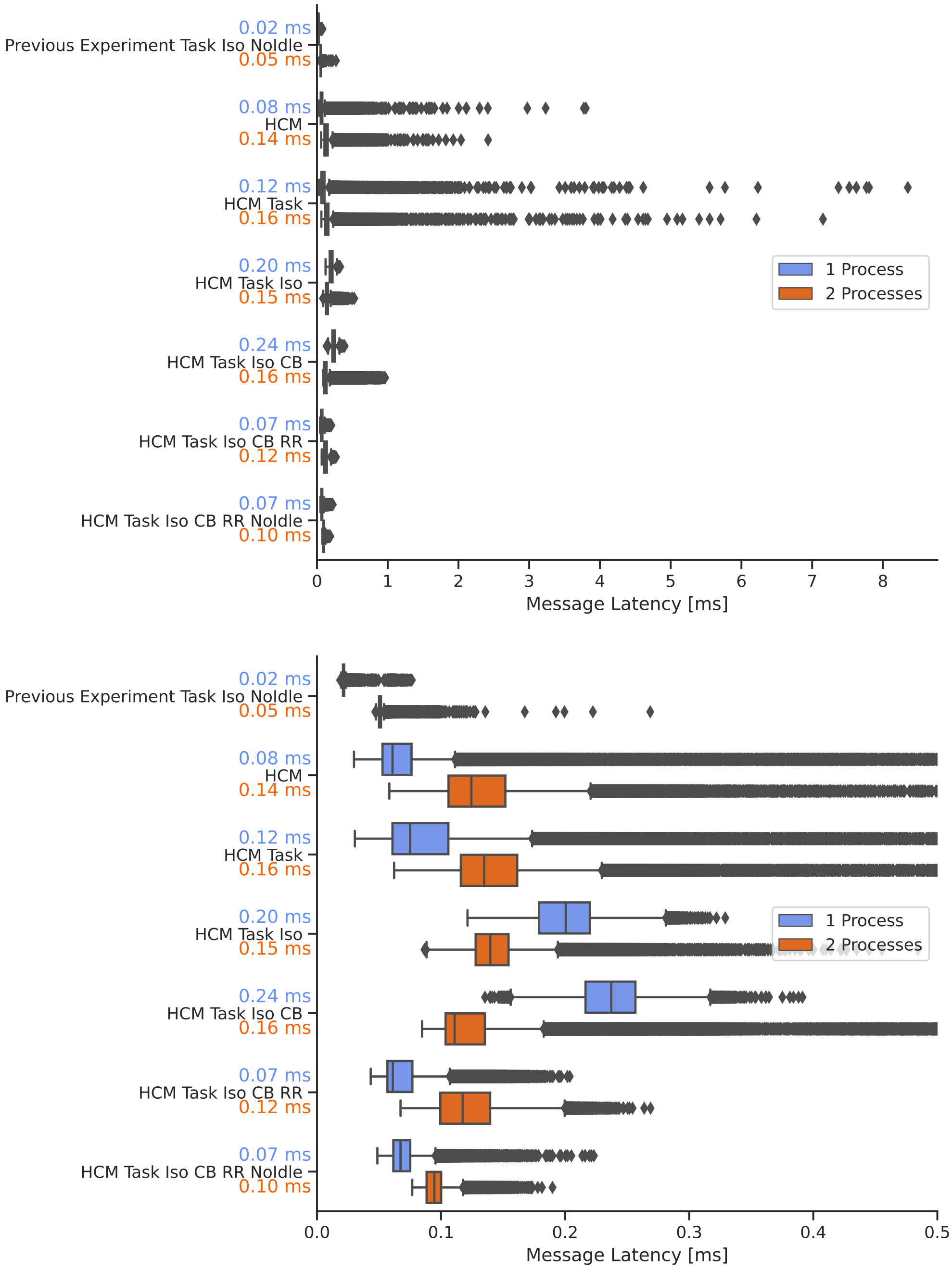


Figure 4.13: Message latencies in the integrated system while executing the motion stack and actively walking. The lower plot shows the same data but zoomed in for better readability. Different options to run the HCM were evaluated. These include using taskset ("task"), isolating the cores ("Iso"), using parallel callbacks ("CB"), using the round robin scheduler ("RR"), and deactivating the idle function of the CPU ("NoIdle"). It can be seen that only an exact configuration of the Linux scheduling prevents outliers with high latency in an integrated test scenario.

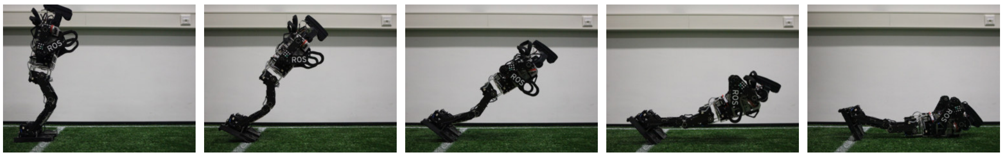


Figure 4.14: Image sequence of a frontal fall with the HCM active. Between pictures two and three, the robot recognizes the fall and moves its arms and head into a safe position. It reaches this position in picture four before hitting the ground. In this example, the threshold-based classifier was used. [BZ20]

Table 4.2: Different classes of sensor information used in the falling experiment.

Name	Dim.	Description
imu	$\mathbb{R}^6$	Accelerometer and gyroscope data.
euler	$\mathbb{R}^3$	Torso orientation in Euler angles based on IMU complementary filter.
fused	$\mathbb{R}^3$	Torso orientation in fused angles based on IMU complementary filter.
joints	$\mathbb{R}^{12}$	Effort feedback of the leg servos.
cop	$\mathbb{R}^4$	Center of pressure of each foot computed from strain gauge sensors.

some investigation, we realized that the usage of taskset leads to only one of the node's threads being executed simultaneously. We solved this issue by using the round-robin scheduler for the HCM node process. This improved the results, especially for the single process case. Additionally, we also tested if the deactivation of the idle function also helps for a node that uses the CPU for more than just sending a message. There is only a small difference for the separate processes case.

Overall, the results show that the introduced latency by the HCM can be kept low enough to allow a fast response to sensor input. When using ROS 2, it is crucial to use the C++ EventExecutor and to execute the HCM with the other motion nodes and the hardware interface in the same process to minimize the latency. Furthermore, the Linux system needs to be correctly configured to prevent interrupts from other processes and to prevent the CPU from going into idle mode.

4.4.3 Fall Detection

Investigation with slow-motion videos showed that it takes around 1 s between giving an impact to the robot and it hitting the ground, depending on the applied force (see Figure 4.14). The robot needs 0.2 s to take a safe pose when it was standing or walking. Therefore the detection must have at least a lead time of 200 ms before impact. Having a higher lead time is still desirable as the time to get into a safe pose can be higher based on the current pose, e. g., while performing a kick motion. Furthermore, having a higher lead time would also enable the usage of different fall poses that minimize the getting up time afterward, e. g., landing on the hands.

Dataset

To investigate this, different types of sensor data and processed orientation estimations were used (see Table 4.2). The IMU data and the derived orientation in Euler angles are commonly used for this issue (see Section 4.1.3). Additionally, the fused angles representation of the torso orientation was used since it was specially developed for balancing [AB15] and could be simpler to classify. Other modalities from the servos and strain gauge sensors were also included to investigate if these could improve the fall detection.

All data types were recorded with 200 Hz (due to the hardware limitations at that time) on the robot while it was standing, walking, and falling in different directions. Manual perturbations were performed to produce edge cases where the robot is almost falling. The robot performed no protective measurements during recording to prevent biases. Instead, the robot was manually caught just before the fall to prevent damage. The recorded frames were manually annotated as stable or with the fall direction. All frames after catching the robot were removed. Therefore, the training set does not contain samples of the actual impact of the robot on the ground. This missing data is not an issue since the prediction of the fall impact at this state is too late for any protective measures. The lead time computation is influenced by it as the recorded impact time is slightly before the actual impact. Therefore, the performance of the classifiers may be underestimated. Still, this does not bias the comparison between the different approaches.

The training set consists of 13,944 frames where the robot is not falling and 7,961 where it is falling in one of the four different directions. For evaluation, a set of 9,436 frames where the robot is not falling, but moving and manually pushed, is used to compute the number of FPs. Furthermore, the impact times for five falls while standing and walking in each direction (together 40) were labeled to compute the lead time (see Figure 4.15 for examples).

Training

Three commonly used classification approaches were used: KNN, SVM, and MLP. They were trained using the scikit-learn library [PVG+11] with five-fold cross-validation. Their hyperparameters were tuned for each type of sensor input using the Tree-structured Parzen Estimator (TPE) [BBBK11] implementation of the Optuna library [ASY+19] with 100 trials each. TPE was also used to optimize the thresholds of the original TB classifier for both input types with 100 trials each.

Results

All classifier-sensor combinations were evaluated for their mean lead time and the number of FPs (see Figure 4.16) as well as for their minimal lead time (see Figure 4.17). The latter is especially interesting to evaluate how much time can be planned to perform protective movements during a fall, while the FPs quantify the classifier's reliability.

The SVM and MLP classifiers seem to perform similarly well for most modalities, while the KNN classifier performs worse, especially in regard to the minimal lead time. Regarding the sensor information, *euler* and *fused* perform inferior in terms of lead time, probably due to the inertness introduced by the complementary filter. In contrast to this, *imu* provides the best lead time but has a high number of FPs. Combining *imu* with either *euler* or *fused* leads to a few FPs while still having a high lead time. The difference between these combinations is small, especially in the mean lead time. This could be due to the fact that the robot mostly falls along one of its axes, thus leading to no complex rotations where the fused angles would be an advantage. Furthermore, the gimbal lock issues of Euler angles only come into play when the robot has already fallen. Therefore, the data of Euler angles and fused angles is very similar leading up to the fall (compare Figure 4.15). The *joints* and *cop* data types can not be used as a sole source of data as their FPs are too high. Combining either of them with the well-performing *imu+euler* combination does not provide any improvements.

The best compromise between mean lead time, min lead time, and FP is provided by *imu+fused* with the MLP classifier. Still, the difference to the baseline TB classifier that uses the same data is small.

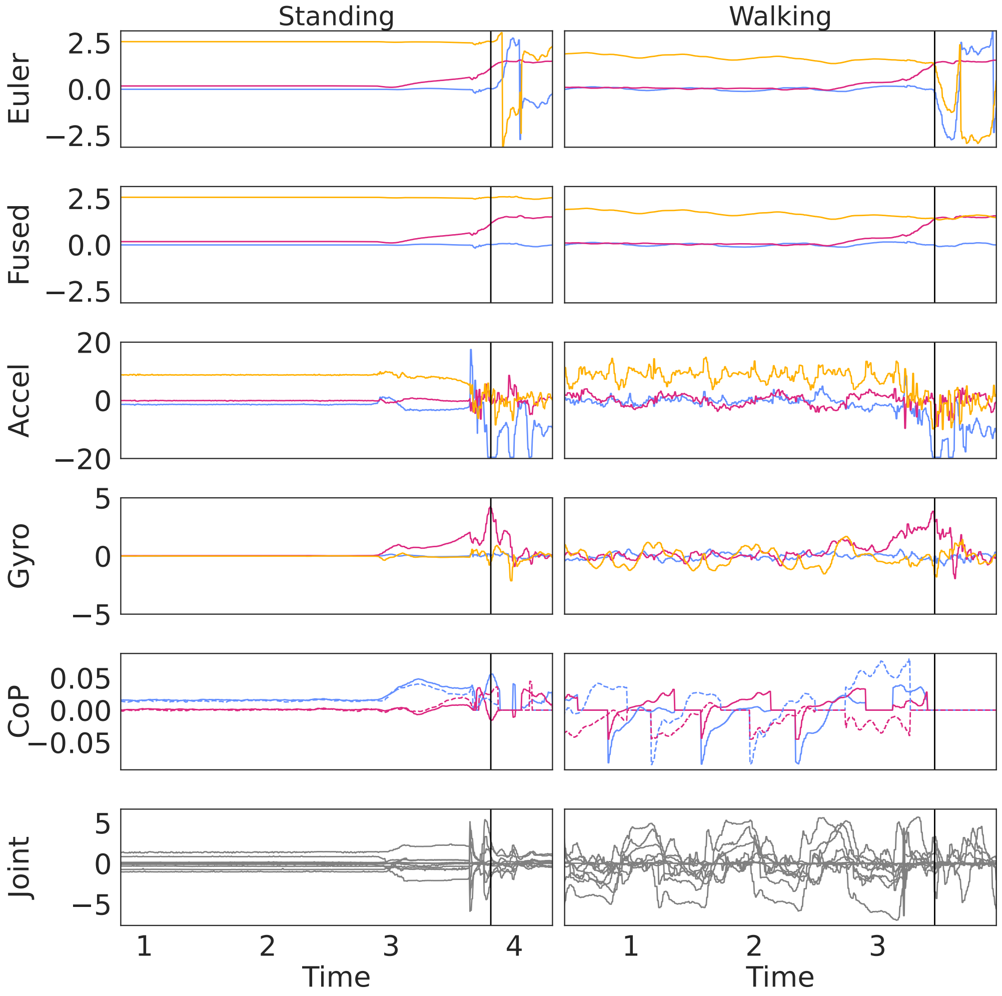


Figure 4.15: An exemplary plot of two falls to the front from the evaluation set. On the **left** the robot was standing, and on the **right** it was walking. The colors correspond to the axis of the robot frame (x: **blue**, y: **red**, z: **yellow**). For the CoP the dashed line indicates the right foot and the other the left foot. The vertical black line indicates the moment when the robot was caught just before impact. Data after this point is not used for the evaluation.

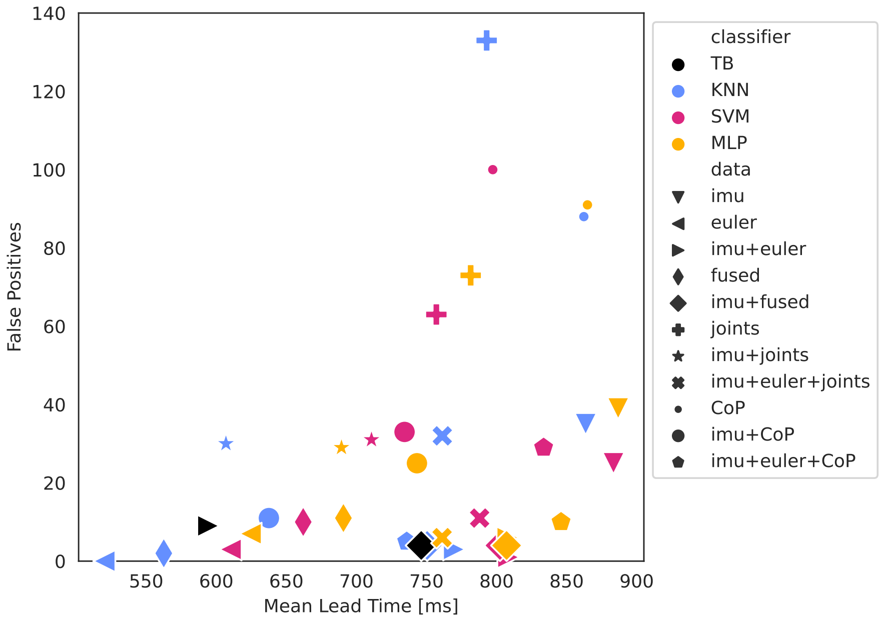


Figure 4.16: Mean lead time (higher is better) and number of FPs (lower is better) for different classifiers and sensor inputs on the validation set. Combinations of *imu+euler* or *imu+fused* data types with MLP and KNN classifiers provide the best results.

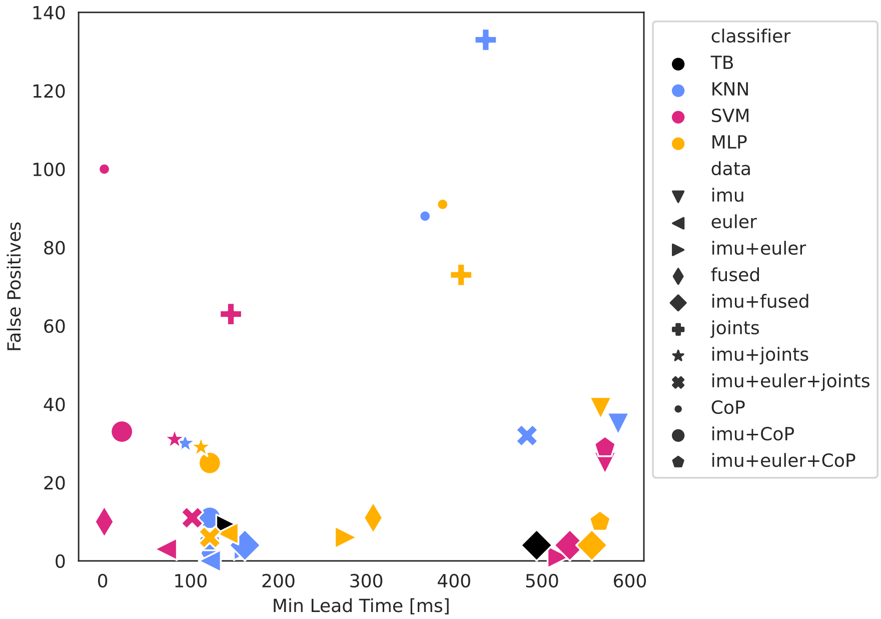


Figure 4.17: Mean lead time (higher is better) and number of FPs (lower is better) for different classifiers and sensor inputs on the validation set. Combinations of *imu+fused* data types provide the best results. The baseline TB classifier is only slightly worse.

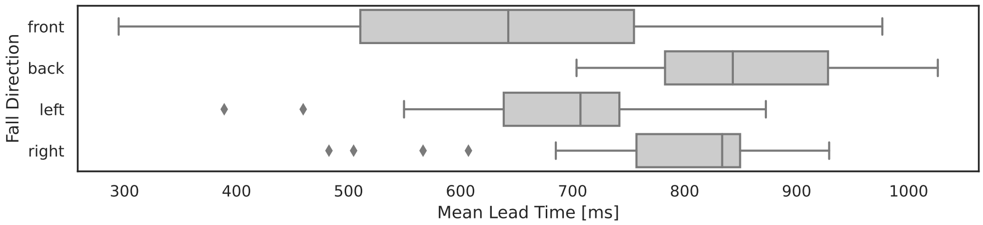


Figure 4.18: Mean lead times for the different classifiers separated into the fall directions. It can be observed that, generally, falls to the front are detected later than to the other sides. This may be related to the fact that the robot was leaned to the front during the experiments due to the walking parameters.

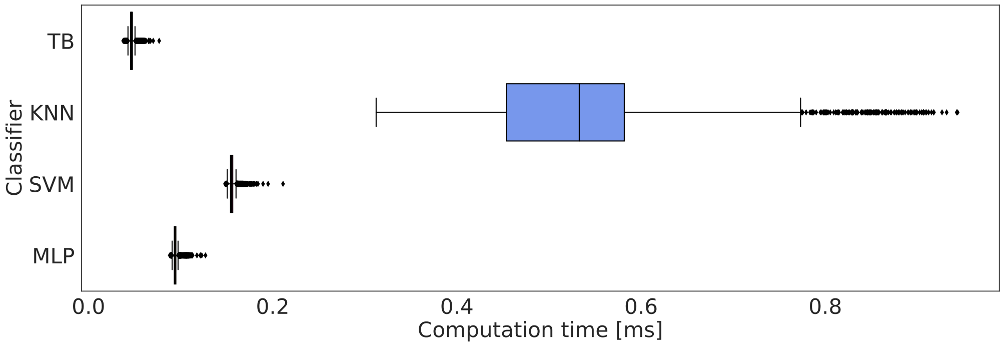


Figure 4.19: Computing time of different classifiers with imu+fused input. Naturally, the threshold-based classifier has the fastest and most constant performance due to its simplicity.

To see if the fall detection works uniformly well for all directions, the mean lead times for each data-classifier combination were separated into the four fall directions (see Figure 4.18). The front fall direction seems to be more difficult to detect, which could be influenced by the fact that the robot was leaned to the front during the experiments due to the used walk parameters.

The fall detector has to be evaluated in every cycle of the HCM (preferably with 1,000 Hz). Therefore, it should take as little time as possible. To investigate this, the time necessary for every prediction in the validation set was measured on the robot's computer (ASUS PN51-E1). The results are shown in Figure 4.19. The threshold-based classifier performs best, MLP and SVM are both slightly inferior, while KNN performs worst. It should also be noted that KNN and SVM runtime is dependent on the number of samples used to fit the classifier, while MLP and TB are not. Therefore, using a larger dataset to create more accurate results would also worsen their runtime.

These results show that it is still possible to improve fall detection by using more sophisticated methods. Still, the difference is not large, and the minimal lead time of the TB classifier is still well enough. Furthermore, the TB solution has an additional advantage, especially when using it in a competition scenario. Its parameters can be manually re-tuned to different conditions, e.g., different floor covers, in a few minutes without the need to record new training data. By changing these thresholds, it is also possible to directly tune the trade-off between lead time and false positives. The classifier choice should also be based on how much time is needed to get into a safe position. If this time is short, a solution with a lower false-positive rate can be picked.

4.5 Summary and Future Work

This chapter described the detection and handling of falls. Different approaches and modalities for detecting falls were evaluated, showing that a simple TB classifier already achieves a sufficient performance. However, MLP and SVM based solutions can achieve slightly higher lead times. In our experiment, adding further modalities besides the IMU did not improve the detection. The fall detection and handling were integrated into the software stack using a new architectural approach called HCM. This approach achieves an abstraction from the fact that the robot is humanoid and, therefore, allows easier control. Furthermore, it removes the necessity of monolithic motion nodes and, thereby, allows a clear separation of motion skills. The HCM is implemented using a novel decision-making framework called DSD. This allows a simple and clear structuring of the many tasks that the HCM needs to perform. Finally, it was shown that the message latency that is introduced by the HCM approach is low enough for it to be usable. It was also shown how the Linux scheduler has to be configured to allow such low latencies.

Still, there are some points which could be improved in the future. The dataset for the falling detection could be improved by including more falls and other robots. It could also be recorded again with a higher data rate since the newest version of the hardware interface allows this now (see Section 3.6.1). The labeling of the data could be automated using an external sensor to reduce inaccuracy from manual labeling. Since the HCM also knows when it has fallen, it could continuously gather information and use lifelong learning to improve its classifiers over time. The current classifiers rely on a single measurement sample. Using multiple previous data samples might improve their performance.

The protective measures are currently very simple. More sophisticated solutions may further reduce the impact force or allow faster standing up afterward. A more human-like approach would be to use the arms for softening the impact. However, new solutions need to be found that ensure that no gears break during such movements.

Chapter 5

Quintic Spline based Motion Skills

An autonomous humanoid robot needs a set of basic skills to achieve goals in the world [JM03]. These can be invoked by higher-level behaviors and produce a motion output (see Section 4.3.2). One naturally given way to classify these skills is based on the type of motion that they produce. This can either be continuous, e.g., walking, or discrete, e.g., standing up. Additionally, we propose the following classification based on their purpose.

Locomotion: A crucial task for a humanoid robot is locomotion [SJAB19]. It allows the robot to change its position in the world, e.g., to move to an object of interest. Locomotion can be implemented by different omnidirectional walking approaches.

Recovery: A humanoid robot is inherently unstable due to its high CoM and its small support polygon, which leads to falls. Since the robot can not perform any meaningful task while lying on the ground, it needs to perform a recovery motion to get up again. Additionally, the robot might already perform a motion during the fall to prevent it or to minimize damage to the hardware.

Manipulation: For most tasks, the robot needs to interact with objects in the world. This includes, most notably, manipulation using the arms and hands, but usage of other body parts, e.g., pedipulation, is also possible.

This chapter presents a general approach (called *OptiQuint*) for creating humanoid skills of all three presented categories by using optimized Cartesian quintic splines. It is inspired by the walk controller of Rouxel et al. [RPH$^+$15]. To show the general applicability of this approach, it is demonstrated for the following skills that cover all classes:

- Walk (continuous, locomotion)
- Stand up (discrete, recovery)
- Kick (discrete, manipulation)

The structure of the chapter is as follows. First, related works to this field are presented in Section 5.1. Then the general approach is explained in Section 5.2. In Section 5.3, the implementation of the walk skill will be discussed in most detail, as

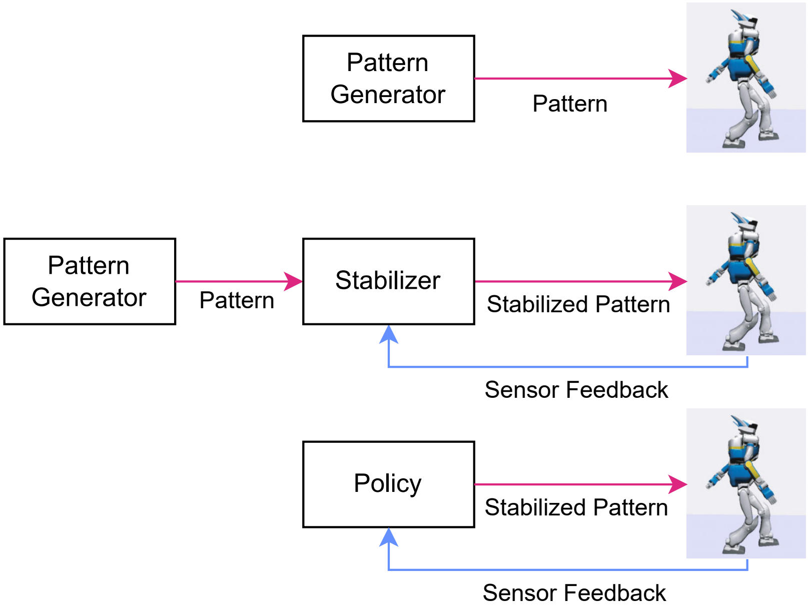


Figure 5.1: A pattern can theoretically be directly applied to a robot (**top**). However, for most real-life scenarios, additional stabilization based on the sensor feedback is required. This can be added by a dedicated stabilizer (**center**), or both parts can be integrated into one module, for example, in a RL policy (**bottom**). Based on [KHHY14, p.106].

it is the most crucial and complex one. Additionally, the analog implementation of a stand-up and a kick skill will be discussed in Sections 5.4 and 5.5. Finally, a summary is given in Section 5.6.

5.1 Related Work

This section presents related works for the control of humanoid motion skills. Most research works don't use a holistic approach that covers all motion skills, but focus on one, with locomotion being the most researched area. Nevertheless, we will organize them in the following by the type of approach rather than the type of skill.

Generally, any motion skill for humanoid robots needs to define a trajectory of one or more endeffectors over time. This trajectory is sometimes, especially for locomotion, called pattern. Theoretically, this pattern is enough to perform the skill, but due to the unstable nature of humanoid robots, typically, additional stabilization is necessary (see Figure 5.1). These modify the pattern based on sensor data, e.g., to compensate for uneven floors or external forces that are applied to the robot. It is also possible to have both pattern generation and stabilization in one unit. This is often the case for RL trained policies. In the following, we present different approaches for pattern generation (Section 5.1.1), stabilization (Section 5.1.2), and learned approaches that combine these (Section 5.1.3). Still, the focus will be on the pattern generation as this will be used as reference motion for the RL approach (see Chapter 6). Since our approach uses parameter optimization, we also present related work in this field (Section 5.1.4).

5.1.1 Pattern Generators

Besides the division in the following subcategories, pattern generators can also be differentiated based on the used space in which the pattern is generated. Typically, either joint or Cartesian space is used, but other custom application-specific space representations are also possible, e.g., the leg space [MB13]. If the joint space is used, no IK is necessary, but the approach is less transferable to other robots as it relies on a certain kinematic configuration.

Central Pattern Generators

The central pattern generator (CPG) is a biologically inspired approach that mimics how rhythmic movements are generated in humans and animals. Biological CPGs consist of multiple connected neurons that can be activated with a non-rhythmic signal and then continue to create rhythmic output even without phasic sensor feedback [BHW09, p.650]. Artificial CPGs try to mimic this by using artificial neural networks. Due to their rhythmic nature, they are especially interesting for continuous motion skills like locomotion [DABI18]. Typically, the produced oscillations each control the goal position of one joint, which is close to how CPGs work in animals, but they can also be used to generate patterns in Cartesian space [PHK10] [PHK13].

CPGs can not only be used to generate patterns but also to stabilize them directly as it is possible to incorporate sensor feedback as input. This input can also be preprocessed before feeding it into the CPG network. An example of this is the work of Auddy et al. [AMW19], which created a locomotion for a humanoid robot using CPGs. In their approach, the CPG network is combined with a MLP that processes the state of the robot's torso into a high-level control signal for the CPG network. Additionally, the current joint angle positions are directly given to the CPG network.

Splines

The basics of splines are already discussed in Section 2.4. They have been used early in robotics research for other types of robots, e.g., to control robotic arms [Pau72]. They have also been used for humanoid robots due to their ability to generate smooth motions, which are more stable [TZS03].

Seiwald et al. used Cartesian quintic splines to describe the movement of the CoM of a humanoid robot during locomotion [SSSR19]. Using a quintic spline instead of a cubic spline improved their stability. They stabilized the generated pattern using hybrid position/force control.

Rouxel et al. developed a spline-based walk engine for their small humanoid robot Sigmaban [RPH$^+$15] [Rou17], which was also the basis for this work. Quintic splines were chosen to create smooth motions of the robot without sudden changes in acceleration. All parameters for the splines were tuned manually, and the skill was only applied to one robot type.

Liu et al. created a kick motion using Cartesian B-Splines [LLL$^+$21]. They concluded that this method leads to better kicks than their previously used method, which is not clearly described but seems to be a keyframe animation.

Splines can also be created by using Bézier curve for each segment. This has been used to create a kick skill for the NAO robot, which was further stabilized with a proportional integral derivative (PID) controller [MLR10].

Keyframe Animations

A similar approach to splines are keyframe animations which have been used for computer graphics for a long time [BW71]. As the name suggests, this approach works by defining certain keyframes. In robotics, these are poses of the robot at specific time points. All poses for the other time points are then generated by interpolating between these keyframes, which can be done linearly or by using other functions. The main difference to splines is that for keyframe animations a complete pose of the robot is defined for a time point, while spline-based solutions can define knots for certain splines at a given time point. These poses are typically defined in joint space through a set of joint positions. This clear definition of full robot poses simplifies the creation of motion skills for the developer, as the different movement dimensions are always synchronized. It also allows the creation of such animations directly on the robot by putting it manually in these keyframe poses, which allows an intuitive balancing of these poses. On the downside, the forced synchronized definition of all movement dimensions limits the expressiveness of this approach, and it is difficult to use sensor data for stabilization. Still, it is often used due to its simplicity.

Keyframe animations have been used in HSL for a long time [SSB06]. Based on a survey we conducted during the RoboCup 2022, all other teams in the HSL still used keyframe animations for their stand-up skill, and only one other team used something different for their kick motion. Before using the OptiQuint approach, the Hamburg Bit-Bots also relied on keyframe animations for stand-up and kick.

Generally, it can be observed that keyframe animations are being used for discrete skills but not for continuous ones.

Motion Capture

Instead of creating motion patterns purely programmatically, they can also be recorded from a human demonstration using motion capture (MOCAP). This approach is often used in computer games [Men00]. Similarly to the keyframes, this enables a more intuitive way to describe a complex motion. Since the poses are not recorded directly on the robot, they need to be transferred. This can be complicated as most humanoid robots have fewer DoF than humans and can not imitate all movements, e.g., due to missing toes. Furthermore, due to different mass distributions between the recording human and the goal platform, the recorded movements may be unstable. Despite its downsides, it has been used, e.g., by Mistry et al., who realized a sit-to-stand skill for a humanoid robot sitting on a chair [MMYH10]. They used MOCAP data from a human demonstration which included contact forces.

Model Based

The previously presented methods used various ways to directly describe the movements of the robot. Another approach is to make a model of the robot and use this to generate the motion. This has the advantage that this motion can be directly described in a physically stable fashion, but it also requires exact modeling of the robot, which is especially problematic for low-cost robots. One of the earliest pattern generators for bipedal walking used the ZMP to define a stable movement trajectory [VS72]. The originally used algorithm was later sped up by solving the central equation in the frequency domain [TTT+89]. Another classic example is the linear inverted pendulum model (LIPM) [KHHY14, 120-135] [KKK+01], which describes the position and velocity of the CoM in relation to the support foot like an inverted pendulum. To allow for a constant height of the torso, the length of the pendulum can be changed linearly, which is a simplification of the leg length change due to knee movement.

5.1.2 Stabilization

Similarly to the pattern generation, the stabilization can also work in different space representations. For locomotion patterns in the joint space, the hip, and the ankle strategy proved their usefulness [NN08, JLO$^+$19]. Other strategies include foot-tilting strategy and stepping strategy [PCDG06]. Simply stopping the walk can also be used [RB06].

Missura et al. [MBB20] modified the walk pattern with capture steps in case of instabilities. They successfully applied this on a real Nimbo-OP2.

Since Euler angles have multiple issues, e.g., gimbal locks, fused angles were proposed as an alternative orientation representation [AB15]. They were also used to stabilize a locomotion skill [AB16].

RL has also been applied to generate stabilization. Yang et al. [YKL17] used a combination of a high-level learned neural network and low-level PD controller. They were able to learn stabilizing movements on a simplified humanoid robot simulation model. Similarly, Yang et al. [YYM$^+$18] learned different strategies for push recovery by combining neural networks and PD controller. Pankert et al. [PKA18] learned capture steps for push recovery by mapping human demonstrations via RL to the robot.

5.1.3 Learning Approaches

Peng et al. [PBYVDP17] used a two-level hierarchical approach to learn walking and dribbling on a simulated biped using neural networks. The lower-level network generates a stable walk for a given step target, while the higher-level network processes a terrain map to generate step targets for the lower network.

Melo et al. [MMdC19] learned kicking and walking on a simulated NAO robot by using supervised learning. Their dataset consisted of keyframe-based movement patterns that were taken from existing solutions. In another work, Melo et al. [MM19] used Proximal Policy Optimization (PPO) to learn running skills for a simulated humanoid without prior knowledge with an action in joint space.

Further approaches that combine RL with reference motions are described in Section 6.1.1.

5.1.4 Parameter Optimization

Approaches for pattern generation and stabilization often have parameters, e.g., the P, I, and D values for a PID controller. The performance of the approach is typically connected to the quality of these parameters. Therefore, finding optimal, or at least sufficiently good, values is important. These parameters can often be tuned manually, either through intuition or by applying a formalized method, e.g., the Ziegler–Nichols method [ZN$^+$42] for PID controllers. Still, automatic methods are to be preferred as they propose less work and may find better parameters. In the following, we present different approaches to optimize the parameters of motion skills.

Shafii et al. [SLR15] optimized the seven parameters of a locomotion skill in simulation using covariance matrix adaptation evolution strategy (CMA-ES) [HO01]. This algorithm was also used by Seekircher et al. [SV16] to optimize the ten model parameters of a LIPM based locomotion on the NAO robot.

Rodriguez et al. [RBB18] applied Bayesian optimization to optimize up to four parameters of their locomotion skill stabilization. They used a combination of simulated and real experiments for the optimization process.

Kasaei et al. [KLP19] optimized eight parameters of a two-mass model-based locomotion skill on a simulated NAO robot. They applied a genetic algorithm to optimize eight parameters.

Silva et al. [SPCB17] applied RL to optimize two parameters of their locomotion approach.

5.2 Approach

The OptiQuint approach uses parameterized quintic splines to generate a model-free Cartesian open-loop trajectory which is then transformed into joint space by using a general IK. Therefore, it is possible to apply this on any bipedal robot, if the correct parameters are found. Since these parameters are few and in direct relation to the robot's movement (e.g. how high the foot is lifted), it is possible to tune them manually. Nevertheless, we provide a simulator-based optimization technique to find a set of them for any given bipedal robot model. To stabilize the open loop trajectory, we apply different stabilization methods, such as phase modification and PID control. See Figure 5.2 for an overview.

5.2.1 Spline Creation

The first step in the approach is to create splines that describe the movement of robot links, e.g., the left foot, in Cartesian space over time. Therefore, for each skill, it is necessary to decide which links of the robot should be moved in relation to which reference frame. For example, for a walk skill, it makes sense to describe the movement of the torso and the moving foot in relation to the support foot. For each of those parts, we need six splines to describe the complete pose using Euler angles $(x, y, z, roll, pitch, yaw)$. We will call such a combination of six splines a *pose-spline*. Since we only use quintic splines in the following, the pose-spline will be denoted with $Q(t)$.

$$Q(t) = (S_x(t), S_y(t), S_z(t), S_{roll}(t), S_{pitch}(t), S_{yaw}(t)) \tag{5.1}$$

Note that this is not the same as a multidimensional spline since the splines do not need to have the knots at the same time points. Typically, a motion skill is controlling multiple endeffectors in a humanoid robot. Therefore, the state of the robot at time t can be described by using multiple pose-splines. We denote such a combination of pose spline with $Q_R(t)$. The usage of different pose splines is possible. One natural example is given in Equation 5.2.

$$Q_R(t) = (Q_{l\_foot}(t), Q_{r\_foot}(t), Q_{l\_hand}(t), Q_{r\_hand}(t)) \tag{5.2}$$

For all splines, the time t is normalized as $t \in [0, 1]$ where 0 denotes the start of the motion and 1 the end. This time representation is called the *phase* of the motion and allows simple scaling of the speed that a motion is performed in.

To define these pose-splines, it is necessary to define their knots $K = \{k_0, k_1, ..., k_n\}$. In the case of a quintic spline, for each knot, the position (p), velocity (v), and acceleration (a) needs to be defined (see Section 2.4). We will describe these three categories of defined knot values as $d_k \in D_k$ in the following. Each d_k is, therefore, one of the following types of tuple.

$$d_k = \begin{cases} (p, t) \\ (v, t) \\ (a, t) \end{cases} \tag{5.3}$$

The set D_k can be constructed from four sources:

$$D_k = D_N \cup D_S \cup D_C \cup D_P \tag{5.4}$$

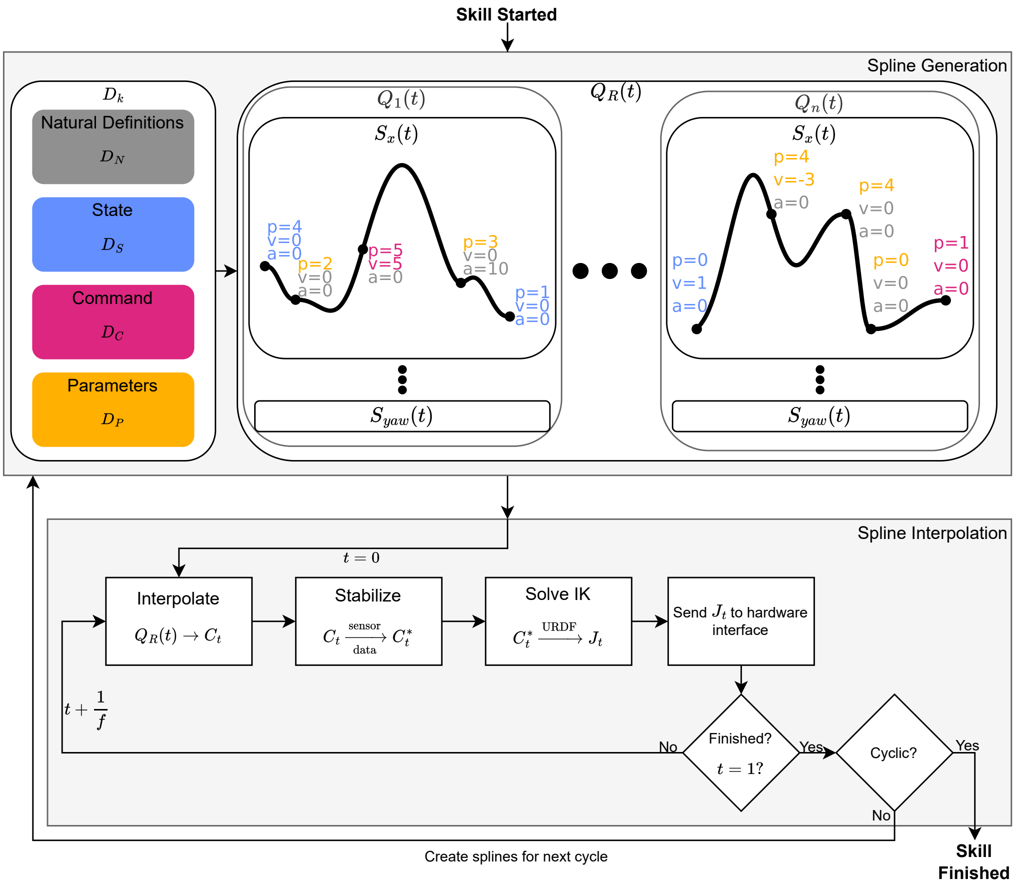


Figure 5.2: Steps of the approach. The skill is invoked by some higher-level behavior. First, the quintic splines are generated based on natural definitions, the current robot state, the command, and the parameters. Then, these splines are continuously interpolated to get the current Cartesian poses. These may then be modified for stabilization. An IK computes the corresponding representation in joint space based on a given URDF description of the robot. These joint positions are then sent to the hardware interface, which executes them on the robot. This process is continued until the phase is completed. If the skill is continuous, the next splines are created. Otherwise, the skill is finished.

Natural Definitions D_N Some members of D_k are given by the nature of the motion. For example, during a walk motion, the pose-spline of the moving foot at the phase where the foot is on its apex is given by $v = a = 0$. An apex is, by definition, the highest point where the movement switches from a rising to a descending movement. Such natural definitions should be used as much as possible to reduce the number of parameters.

State D_S A second source is the current pose of the robot. This is especially useful to define the knots at the start of the motion ($t = 0$). It does not only reduce the number of parameters but also ensures that the movement at the start is smooth.

Command D_C The command that invokes the skill can be another source for values for D_k. In manipulation skills, it often contains information about the object's pose that we want to manipulate. For example, for a kick skill, the ball position is provided in the command, thus defining p of the knots that represent the moment where the ball is touched. But the command can also be used in locomotion skills, as the movement of the foot is also related to the commanded velocity.

Parameters D_P Typically, not all values for D_k can be defined through the above-mentioned sources. For example, it was shown above that v, and a of the foot's apex point during walking are naturally defined. This is not true for the p value of the same knot (the height of the foot above the ground), as this value depends on the kinematics and dynamics of the robot as well as the type of ground, e.g., tall grass requires higher values than a flat floor. These values are represented as parameters of the skill which need to be optimized. Their number ($|D_P|$) should be kept as low as possible since the complexity of the optimization will grow exponentially with this number (curse of dimensionality).

5.2.2 Spline Interpolation

After the splines are created, the skill runs in a control loop with a given frequency f to provide the current joint position commands. While the splines are smooth by definition, we need to discretize them over time which leads to interpolation errors. A high value for f is, therefore, desirable since the time between two loop steps $\Delta t = t_n - t_{n-1}$ becomes smaller, and thus the interpolation is smoother.

In each step of the loop, first, all pose-splines are interpolated at the current phase time t by solving the spline equation (see Equation 2.4). This will provide a set of Cartesian goal poses with one for each controlled link of the robot.

$$C_t = Q_R(t) \tag{5.5}$$

The spline trajectories are only providing an open-loop pattern generation. It is possible to adapt these Cartesian goals to stabilize the motion and to achieve closed-loop control. This is possible through different established control strategies, e.g., by using a PID controller, resulting in a set of stabilized Cartesian end effector goal poses C_t^*. Finally, these are solved by an IK to get the joint goal positions of the robot.

$$J_t = IK(C_t^*) \tag{5.6}$$

These are then given to the hardware or simulator interface.

This process is executed at each time step until $t = 1$. If the skill is periodic (e.g. locomotion), then the phase is reset to $t = 0$, and the splines for the next cycle are generated. Otherwise, the skill is finished. It is also possible to add phase modifications that end a skill early, e.g., to end a locomotion step early when the foot makes contact with the ground. These also need to be checked at each time step.

5.2.3 Parameter Optimization

As stated in Section 5.2.1, the splines are partially defined by parameters. The quality of these parameters highly influences the performance of the skill. Therefore, it is necessary to find good sets of parameters. This can be done manually through trial and error. Since the parameters are directly related to the movement of the robot, it is comparably simple for a human to find good values for them. Still, this requires manual work and will probably not yield an optimal parameter set as humans tend, in our experience, to focus on the first working values and thus run into a local optimum. Additionally, the parameters need to be adjusted if the kinematic or dynamic properties of the robot change. Thus resulting in additional work after changes in hardware which could deter users from hardware developments.

This issue can be mitigated by using a fully automated black-box parameter optimization approach to find a good parameter set for a given robot. The stabilization approaches also rely on parameters. We do not optimize these since they are either simple to find, i.e., the thresholds for the phase manipulations, or have established methods to tune them, i.e., the Ziegler-Nichols method [ZN+42] for the PID controller.

The optimization process can be expressed formally as finding a parameter set Θ^* out of the set of possible parameter sets Ψ that maximizes either a single objective function $f(\Theta)$ (single objective optimization)

$$\Theta^* = \operatorname*{argmax}_{\Theta \in \Psi} \; f(\Theta) \tag{5.7}$$

or multiple objective functions $f_1(\Theta), f_2(\Theta), ..., f_n(\Theta)$ (multi-objective optimization).

$$\Theta^* = \operatorname*{argmax}_{\Theta \in \Psi}(f_1(\Theta), f_2(\Theta), ..., f_n(\Theta)) \tag{5.8}$$

These objective functions quantify how good a given parameter set is for a skill, e.g., for a kick, how far the ball is moved. The objective function is not necessarily naturally given and needs to be designed for each skill. The used objective functions will be discussed in detail in the corresponding skill sections below.

In multi-objective optimization, there is typically not a single parameter set that is the best for all objectives. Instead, the result is a Pareto front of parameter sets. Each of these has one objective function whose value is not surpassed by any other parameter set without reducing the value of another objective function. As only one parameter set can be used, it is necessary to choose one of them. This can be done by choosing manually or by computing a single objective value by taking a weighted sum of the values (scalarization).

In addition to the objective function, it is also necessary to define the parameter space Ψ. We use a continuous range for each parameter. The bounds of some parameters are universal for all used robots, but some are also directly related to the kinematics of the robot, e.g., how far a foot can be raised during a step. Still, these are simple to define, e.g., by testing how far the robot can raise its foot. Furthermore, these ranges do not need to be exact but can be larger. This may prolong the optimization process but will still yield results. Additionally, it is also possible to artificially narrow the parameter ranges to achieve a certain type of motion, e.g., force the robot to raise its feet to a certain height for walking on artificial grass. The detailed parameter ranges are described in the corresponding sections below.

The general optimization process consists of sequentially trying different parameter sets in simulation and computing their objective values by measuring the performance of the executed skill (see Algorithm 1). This might encompass multiple tries of the same parameter set since the skill may not be deterministic due to sensor noise and randomness in the IK solution. The decision of which parameter set is to be evaluated next is made by an optimization algorithm.

5.2.4 Implementation

In this subsection, we will discuss how a skill using this approach can be implemented, relying on established software libraries and frameworks.

Middleware

Our approach tries to create skills that can be put onto a new robot with as least work as possible. This requires easy integration into the robot's software stack, independence from the robot's kinematics, and a simple tuning process. To allow easy integration on

Algorithm 1 Parameter Optimization

```
choose sampler //e.g. MOTPE
define parameter space for each parameter // e.g. $\theta_1 \in [-1, 3]$
parameter_sets = [ ]
objective_values = [ ]
for $i < number\_trials$; $i + +$ do
    sample parameter set
    parameter_sets.append(parameter set)
    execute skill in simulation
    compute objective values
    objective_values.append(objective values)
end for
choose parameter set by scalarization or a posteriori
return parameter set
```

many robots, we rely on the widespread ROS (2) middleware (see Section 2.3), URDF and MoveIt [CSC12] configurations. Each skill can be programmed as a single ROS node, thus resulting in modularity. The node publishes the computed joint goals on a topic. These are then executed on the hardware by ros_control [CMEM+17] or in the simulation via a ROS simulation interface. The commands for the skill node can either come from action calls, for discrete motions, or from a command topic, for continuous motions. The walking, for example, can then be directly controlled from Nav2 stack [8]. One of our applied stabilization methods are PID controllers. These are based on the control_toolbox [19] package is used. The naming and placing of coordinate frames (including the odometry) in the URDF follow REP-120 [16]. This simplifies the usage with other robot models.

However, for some purposes (optimization and learning), the asynchronous communication of ROS is not well fitted. Therefore, we also provide a Python interface based on PyBind11 [28], which allows direct calls of the skill's internal methods and blocks till the result is computed.

IK

To achieve independence from the concrete kinematic configuration of the robot, our approach works in the Cartesian space. However, this requires an IK solver to transfer the goals from Cartesian space to joint space. Since the goal is to allow the usage of different platforms, MoveIt [CSC12] was chosen as an interface for the IK. It provides multiple different IK implementations. *Orocos KDL* [13] is the default IK solver which is based on the inverse Jacobian algorithm. *TrackIK* [BA15] combines an extension to the Newton-based convergence algorithm with Sequential Quadratic Programming. Both run in parallel, and a solution is provided as soon as one approach converges. This makes it more robust. *BioIK* [RHSZ18] uses a memetic approach that integrates gradient methods with evolutionary and particle swarm optimization. It allows to specify multiple different goal types and was shown to successfully avoid getting stuck in local minima.

Some of the used platforms, i.e., the Wolfgang-OP and the NUgus, don't have an analytic IK solution for the legs since the hip joint axes are not intersecting. Therefore, analytic IKs were not used. MoveIt also provides a plugin for *IKFast* [Dia10], which automatically tries to find an analytic algorithm for a given robot model, which is then saved and allows to compute IK solutions quickly during runtime. This plugin did not work for the Wolfgang-OP (maybe due to the not intersecting axes in the hip) and was therefore not further considered as a possible IK solver.

Visualization

An often overlooked factor is providing good additional visualization to the user. In cases where the skill does not perform as required, it can be difficult for the user to understand where the issue lies. If it is only possible to obtain information about the input and output of the skill node, it can be seen as a black box. It is not distinguishable from which step of the approach (spline creation, spline interpolation, stabilization, IK) the error is arising. A simple way to solve this issue is to provide additional debug topics from the skill node that display the internal state. It is possible to only compute and publish this debug data if there are subscribers on these topics, thus not influencing the computational performance otherwise. Two main ROS tools can be used for this. First, RViz allows a 3D visualization of the robot together with additional custom markers. This can, for example, be used to show the spline's knots in 3D space. Second, PlotJuggler allows the simple generation of online plots for data on ROS topics. This can be used to plot the interpolated splines over time.

Parameter Optimization

The Optuna library [ASY$^+$19] provides a clear interface to run an optimization process (called *study*) for a set of different optimization strategies (called *samplers*). These include covariance matrix adaptation evolution strategy (CMA-ES) [HO01], Tree-structured Parzen Estimator (TPE) [BBBK11], Multi-objective Tree-structured Parzen Estimator (MOTPE) [OTWO20], Non-dominated Sorting Genetic Algorithm II (NSGA-II) [DAPM00] and random sampling. Optuna allows the distributed execution of a study, which significantly decreases the time necessary to find good parameters. To do this, multiple instances are created which communicate via a Structured Query Language (SQL) database. When a sampler starts the evaluation (called *trial*) of a new parameter set, it writes the used parameters into the database so that the other instances know which parameters are already explored. When the trial finishes, the resulting objective values are added to the parameter set in the database. This allows each sampler to use the combined experiences of all instances when sampling the next parameter set. Optuna also provides visualization tools that show the current progress of a study based on the data in the database.

5.3 Walk

This section describes the application of the OptiQuint approach to a locomotion skill (see Figure 5.3 for an overview). Walking is a continuous motion that can also be seen as a sequence of discrete step motions. Still, it is complex since the support foot switches at each step, the desired walk velocity may change, and special motions are required for starting and stopping the walk. Additionally, we apply phase modulation techniques to increase the stability of the robot. This is handled by a state machine (Section 5.3.1) that tracks the support foot and decides on what kind of step to perform. The spline engine (Section 5.3.2) creates the splines at the start of a step and these are then interpolated to provide the Cartesian goal poses. These are modified by PID controllers (Section 5.3.3) and then solved by an IK. The resulting joint goal positions are published and further processed by the hardware or simulation interface.

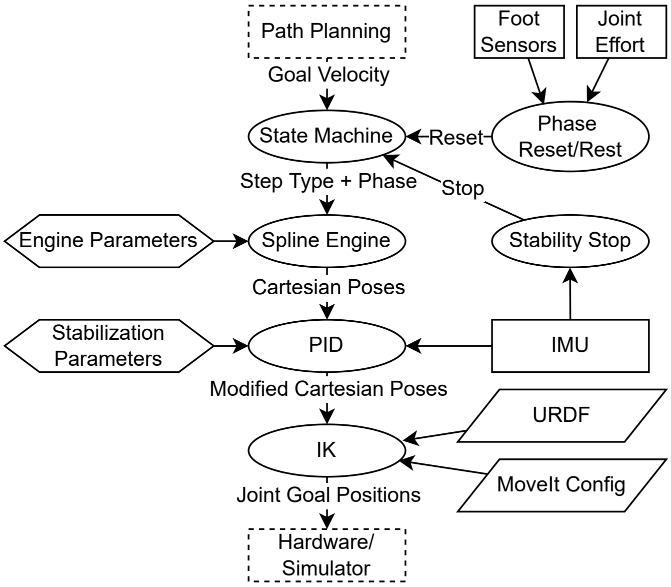


Figure 5.3: Overview of the walk skill implementation. First, the FSM decides on the step type and current phase based on the command and the stabilization methods. Then, the spline engine generates the current Cartesian poses. These are stabilized by a PID controller. Finally, the IK computes the corresponding joint goals and sends them to the hardware interface. Parallelograms represent model information, rectangles represent sensor data, and rhombuses represent parameters. Parts that don't belong to the walking node are dashed. [BZ22]

5.3.1 Finite State Machine

The walk engine has to do different types of steps, e.g., the first step to start walking differs from the following ones. Furthermore, two stability features are added, which can end the step early (*Phase Reset*), prolong a step (*Phase Rest*), or request a break during double support to let oscillations of the robot settle (*Stability Stop*), see also Section 5.3.3. As we use this approach in RoboCup soccer, a small dribbling kick is also integrated. This behavior is implemented as a FSM which uses the information about the previous step, the commanded velocities, and the feedback from the stabilization methods to provide which type of step should currently be applied and what the phase is. An overview of the FSM can be seen in Figure 5.4.

Before the first lifting of a foot can be performed, the CoM needs to be moved over the support foot as it will otherwise leave the support polygon and the robot will fall. Therefore, the walk always starts with the *Start Movement*. Then the first step is performed with a phase offset between the torso and foot (*Start Step*) to initialize the lateral swinging motion of the torso. After this, the walk controller will perform the same kind of step, which is only modified by the commanded walk velocity (*Walking*). While in this state, a brief stop can be made to stabilize the robot if it begins oscillating (*Pause*). Furthermore, small dribbling kicks can be performed, which alter the movement of the flying foot (*Kick*). A similar procedure to the start is done when the robot wants to come to a complete stop. This ensures stopping stably and results in a centered pose which is more stable and allows the execution of other skills, e.g., the kick skill.

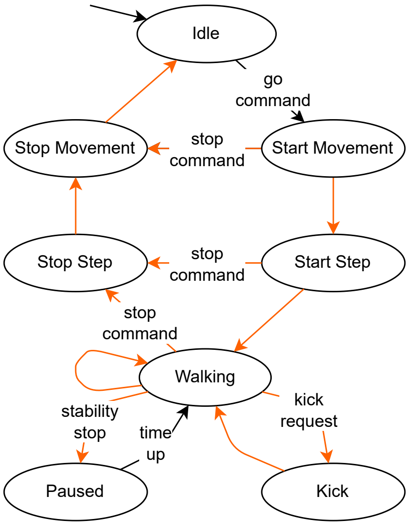


Figure 5.4: Visualization of the FSM, which controls the state of the walk skill. The *orange* state transitions are executed when a step ends, while the *black* transitions are executed whenever the corresponding event happens. [BZ22]

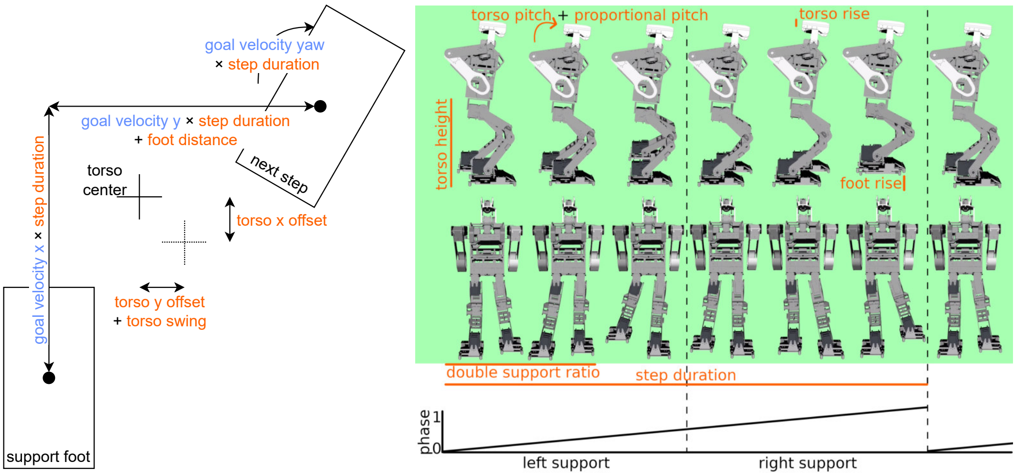


Figure 5.5: Illustration of the walk pattern creation. The next step pose (**left**) is defined by the goal walk velocity (**blue**) and the parameters (**orange**). Based on this and further parameters, the movement during a walk cycle is defined (**right**). A phase variable represents the current time in the cycle (**bottom**). The double support ratio is exaggerated for clarity. The parameter which defines a phase shift between the torso and foot movement is not displayed. [BZ22]

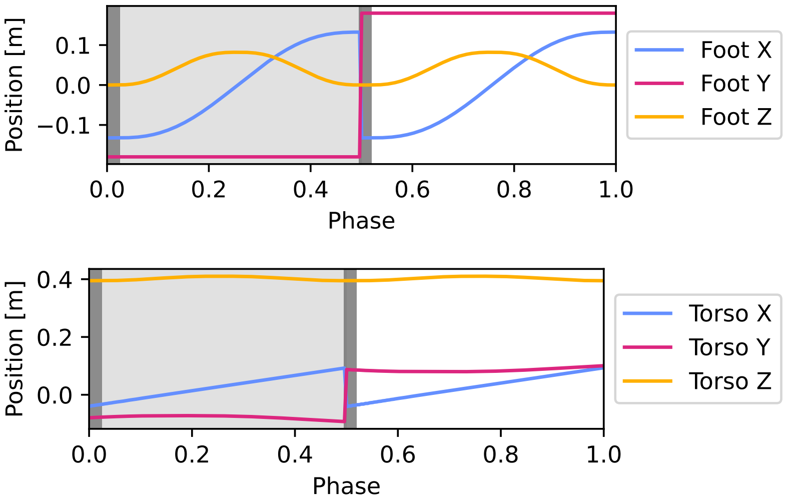


Figure 5.6: Exemplary splines of a simple walk forward with $0.4\frac{\text{m}}{\text{s}}$. One double step over, represented by one phase cycle, is shown. The light gray area indicates that the left foot is the support foot, while the darker area indicates the double support phases. Since the splines are defined in the support foot frame, it looks like there is a break during step exchange. However, from the corresponding reference frame, it leads to a smooth movement. The orientation splines are not shown, as these are constant when the robot is not moving in yaw direction or changing its velocity.

5.3.2 Pattern Generation

The walk spline engine is partly based on *IKWalk* by Rouxel et al. [RPH$^+$15] but has been almost completely rewritten, and the spline definitions have been changed, e.g. by the introduction of a vertical torso movement. It defines the splines for the different step types as required by the FSM. For simplicity, we will only discuss the normal step spline (state *Walking*) in the following, but the other splines are very similarly defined, and their differences have been discussed in the previous section.

The quintic splines are used to represent the Cartesian trajectories of the moving foot Q_{foot} and the torso Q_{torso} in the support foot frame (see Figure 5.6 for an example). Each trajectory is represented with one pose-spline (consisting of six splines that represent one dimension of translation or rotation). These splines are newly created for every single step to adapt to eventually changed commanded velocities.

$$Q_{foot} = (S_{foot\_x}, S_{foot\_y}, S_{foot\_z}, S_{foot\_roll}, S_{foot\_pitch}, S_{foot\_yaw}) \tag{5.9}$$

$$Q_{torso} = (S_{torso\_x}, S_{torso\_y}, S_{torso\_z}, S_{torso\_roll}, S_{torso\_pitch}, S_{torso\_yaw}) \tag{5.10}$$

As described in Section 5.2.1, there are different sources for the definitions of the knot values D. The values for the knots at $t = 0$ of both splines $S_i \in \{Q_{foot}, Q_{torso}\}$ at the current time step i are given by the corresponding last pose of the previous spline. During the step transition, the support foot changes. Therefore, the values need to be transformed from the previous step's coordinate frame to the current one by a transformation function $f_{fk}()$ that can be solved by using forward kinematic (FK).

$$p_0^{S_i} = f_{fk}(S_{i-1}(1)) \tag{5.11}$$

$$v_0^{S_i} = f_{fk}(\dot{S}_{i-1}(1)) \tag{5.12}$$

$$a_0^{S_i} = f_{fk}(\ddot{S}_{i-1}(1)) \tag{5.13}$$

The actual current poses, as provided through joint encoders and FK, are not used. The delay between commanding the joint position and the joint actually reaching this position, as well as the additional delay of reading the sensor and the modification of the spline goal through the PID controller would result in a difference to the last commanded pose, thus violating the smoothness of the trajectory.

Naturally, the foot should stay on the ground until the end of the double support phase. Therefore an additional knot is necessary. The end time of the double support phase (t_d) is defined through a parameter (θ) and represented in the normalized phase time (a value in $[0, 1]$).

$$t_d = \theta_{step\_duration} * \theta_{double\_support\_ratio} \tag{5.14}$$

Since the foot should not move, the further values are naturally defined.

$$p_{t_d}^{S_{foot}} = p_0^{S_{foot}} \tag{5.15}$$

$$v_{t_d}^{S_{foot}} = 0 \tag{5.16}$$

$$a_{t_d}^{S_{foot}} = 0 \tag{5.17}$$

Naturally, the foot needs to rise from the ground during a step. This is easily described by defining the apex point. The time of the apex (t_a) is in the center of the single support phase to allow equal time for rising and descending the foot.

$$t_a = (1 - t_d)/2 \tag{5.18}$$

Only the movement in z dimension needs to be defined. The velocity and acceleration at this point need to be zero since the foot changes its movement direction. How far the foot rises is defined by a parameter.

$$p_{t_a}^{S_foot\_z} = \theta_{foot\_rise} \tag{5.19}$$

$$v_{t_a}^{S_foot\_z} = 0 \tag{5.20}$$

$$a_{t_a}^{S_foot\_z} = 0 \tag{5.21}$$

The three dimensions for the end position of the foot spline are provided by the commanded velocity (v_{cmd}) and parameters (see Figure 5.5).

$$p_1^{S_{foot\_x}} = v_{cmd\_x} * \theta_{step\_duration} \tag{5.22}$$

$$p_1^{S_{foot\_y}} = v_{cmd\_y} * \theta_{step\_duration} + \theta_{foot\_distance} \tag{5.23}$$

$$p_1^{S_{foot\_yaw}} = v_{cmd\_yaw} * \theta_{step\_duration} \tag{5.24}$$

The other three dimensions of position are naturally defined as zero since the pattern is generated for walking on generally flat ground.

$$p_1^{S_{foot\_z}} = 0 \tag{5.25}$$

$$p_1^{S_{foot\_roll}} = 0 \tag{5.26}$$

$$p_1^{S_{foot\_pitch}} = 0 \tag{5.27}$$

The velocity and acceleration of the foot are naturally defined as zero at the end of the step for all dimensions.

$$v_1^{S_{foot}} = 0 \tag{5.28}$$

$$a_l^{S_{foot}} = 0 \tag{5.29}$$

These definitions are sufficient to describe the motion of the foot. The movement of the torso is defined with the same procedure.

5.3.3 Stabilization Approaches

To increase the stability of the walking, different loop-closure approaches based on different modalities are implemented. All of these are model-free approaches that only rely on parameters. We don't use the CoP as a ZMP criterion, as this is not applicable for uneven ground [KHHY14, p.101]. Furthermore, we do not explicitly enforce that the projection of the CoM is not always in the support polygon since this would not allow any dynamic walking [KHHY14, p.105]

We apply two phase modulation approaches for stability. The *phase reset* uses the data of the foot pressure sensors or the joint torque sensors to detect when the moving foot makes contact with the ground. At this moment, the support foot is changed, and the phase is set to the start of the step. This means the step is ended early, which prevents the moving foot from pushing further into the direction of the ground, thus introducing a force on the robot. Similarly, the *phase rest* prevents the starting of a new step until contact is made with the ground. It is possible to rely solely on the joint torque data, but this data is noisy as the servos only estimate the torque based on the applied motor current. Additionally, friction in the joints can differ between robots, e.g., due to wear and tear, therefore leading to different sensor values on different robots. The foot pressure data is more reliant and should be preferred if the robot is equipped with this type of sensor.

The *stability stop* detects if the robot is becoming unstable in the walk, based on the IMU, and pauses the walking at step change for a short amount of time. During this, the oscillations of the robot can settle themselves. When the robot is not oscillating anymore, it starts to walk again. Naturally, this slows the robot down due to the inserted pauses. Still, it is faster than doing a stand-up motion after a fall. Additionally, it is also better for the hardware, as falls are minimized.

Two PID controllers are applied to adapt the Cartesian poses of the robot's torso based on input from the IMU. One controls the robot's fused pitch angle and the other the fused roll angle based on the estimated orientation of the IMU in fused angles (see also Section 5.1.2). The controllers reduce oscillations of the torso that can arise during the walk and can stabilize external pushes, e.g., from bumping into another robot.

Other PID controllers were tried during the implementation of this approach but were not further used due to inferior results during preliminary experiments. Therefore, the focus was set on the fused angle based controller. The following is a list of the tested PID controllers.

- Torso Euler angles based on IMU measured Euler angles
- Torso Euler angles based on angular velocity from the gyroscope
- Final foot pose of the current step based on IMU measured Euler angles (similar to capture steps)
- Hip joint positions based on IMU measured Euler angles (hip strategy)
- Ankle joint positions based on IMU measured Euler angles (ankle strategy)

The arms of the robot are not used for stabilization, although this would be possible. There are two reasons for this. First, there are RoboCup rules (especially in the HLVS), e.g., ball holding, that could be violated accidentally. Second, the hardware of the Wolfgang-OP is optimized to withstand falls, but it requires the arms to be in certain poses. Although the HCM could move the arms to this pose if a fall is detected, the time for this is short. Thus only small movements of the arms would be possible, and, therefore, this was not further investigated.

5.3.4 Odometry

An important additional information that needs to be provided about the locomotion of the robot is the odometry. It represents the measured route that the robot has taken since the start. It can either be used directly for dead reckoning or as information for the update step of a Bayesian localization approach. In the following, we discuss how this information is measured in the presented approach.

There are different ways to measure the traveled route. The simplest one is to just use the given goal walk velocities (v_{cmd}) and integrate them over time. Naturally, the resulting values will have an error, as the robot will not exactly follow the given goal velocities. There are multiple reasons for this. First, the approach only changes the walk velocity when new splines are created, i.e., when a step is finished. Second, the joint servos will not exactly follow their goal position due to the servo's PD controller and the link inertias. Thus, a step is not exactly executed. How much these two factors influence the error of the odometry is also related to the walk parameters, e.g., a higher step frequency leads to the walk velocity being changed more often.

Due to these issues of using the goal velocity directly, a different approach is typically applied, which computes the transform of each step through FK and thus tracks the pose of the current support foot. Since the odometry needs to present the current pose of the base_link frame (see Section 5.2.4), an additional transform between the support foot and the base_link needs to be added. This is also possible through FK. Still, this approach assumes that the support foot is always perfectly aligned with the ground, i.e. the robot is not tilted to any side. This estimation can be improved by including the orientation estimation from the IMU. Since the yaw that is provided by the IMU (which does not contain a magnetometer due to RoboCup rules) drifts over time, it is not used. The yaw, which is computed through the forward kinematics, is more reliable. The IMU is only used for the roll and pitch orientation of the robot. Since this computation is a general problem of bipedal robots, the computation is done by a separate ROS 2 node that can easily be reused for other walk controllers, e.g., the one described in Chapter 6. The walk node itself only publishes the current support foot using a standardized ROS message, thus informing the odometry node that a step was finished.

5.3.5 Parameter Optimization

A simple objective function for bipedal locomotion is how fast the robot can walk without falling down. This is especially interesting for the domain of RoboCup soccer since walking faster than the opponent team will provide an advantage in a game. Since the walk skill is omnidirectional, there is not just one walk direction that we can maximize. The walk velocity is a vector with three entries (x, y, yaw) which are all continuous, leading to an infinitive number of possible directions. By trying out different objective functions, it became clear that optimizing the walk speed for four directions (forward, backward, sideward, turn) is sufficient to reach a parameter set that also works on the combined velocities, e.g., walking to the back and left while turning clockwise. The sideward and turn movements are symmetrical, and therefore only one direction each needs to be evaluated. This results in the objective functions $f_f(\Theta)$ (forward), $f_b(\Theta)$ (backward), $f_s(\Theta)$ (sideward), $f_t(\Theta)$ (turn).

To evaluate these objective functions, we first evaluate if the robot is able to stand with these parameters (see Algorithm 2). If this is not the case we return $f_f(\Theta) = f_b(\Theta) = f_s(\Theta) = f_t(\Theta) = 0$. Otherwise, we measure the maximal distance $d(v_{cmd})$ that a robot can travel in 10 s for a given goal walk velocity v_{cmd}. This includes an acceleration and deceleration phase, which is necessary because it is not possible to go from standing to high speeds directly. Additionally, the procedure waits two seconds after stopping the walk to verify that the robot does not fall down during stopping. The commanded goal velocity is not used as an objective criterion as the parameters may lead to slower walking than commanded, thus v_{cmd} may represent the actually walked velocity inaccurately. Instead, we use the end position provided by the simulator but only take the distance from the start position in the commanded walk dimension. This prevents the robot from getting a high objective value by walking forward while it should only walk sideward.

We increase the v_{cmd} stepwise linearly.

$$v_{cmd}(x) = a * x, x \in \mathbb{N} \tag{5.30}$$

This is done until the robot has either fallen down or the traveled distance has not increased.

$$d(v_{cmd}(i)) \leq d(v_{cmd}(i-1)) \mid v_{cmd}(i) > v_{cmd}(i-1) \tag{5.31}$$

Before each trial, the robot is reset by letting it walk a few steps in the air. This ensures that the robot starts in a fitting pose to the used parameters. Otherwise, bad parameters in the trial before could lead to an unstable start pose and thus wrongly attribute a low objective value to the current parameters.

The presented optimization process provides the four objective values for $f_f(\Theta)$, $f_b(\Theta)$, $f_s(\Theta)$, $f_t(\Theta)$. We can either scalarize the objectives to a single one or use a multi-objective optimization approach and then use an a posteriori function to choose one set from the Pareto front (see Section 5.2.3). For both approaches, we use the following function:

$$f(\Theta) = f_f(\Theta) + f_b(\Theta) + 2 * f_s(\Theta) + 0.2 * f_t(\Theta) \tag{5.32}$$

The weights are chosen based on the observed typical maximum velocities that are reached in the different directions, i.e., most robots only manage to walk half as fast sidewards compared to forward. Since the turn direction is in $\frac{\mathrm{rad}}{\mathrm{s}}$ instead of $\frac{\mathrm{m}}{\mathrm{s}}$, scaling is necessary to get values of a similar dimension. This scalarization ensures that all four directions are contributing equally to the objective value and, thus, that none of these directions is preferred in the optimization process.

By choosing this objective, we value speed and stability over accurately reaching the commanded velocity. Therefore, the resulting parameter set typically leads to a walk which does not exactly reproduce the given command. We solve this by adding a factor to the walk engine for each direction that scales it accordingly. Still, small differences may exist, but in our experience, these were small enough to be handled by the navigation algorithm of the robot.

As described in Section 5.2.3, some parameter range boundaries can be defined identically for all robots, while others depend on the robot, e.g., the length of the legs. The used parameter ranges are shown in Table 5.1. Boundaries that apply to all robots are written in black, while the others are written in gray. For the latter ones, the value used for the Wolfgang-OP is shown exemplarily.

5.3.6 Other Humanoids

A set of humanoid robot models was prepared to show that the walk skill and the parameter optimization work on different platforms. Two things are needed for each platform. First, a simulation model to be able to run the optimization process. Second, a MoveIt configuration and an URDF model for the IK. The currently largest set of simulation models for humanoids robot platforms is coming from the HLVS for the Webots simulation (see also Section 2.5). Additionally, there are some models of widespread commercial platforms already integrated in the Webots simulation. Therefore, a combination of these two sets of models was chosen for the evaluation of the walk skill.

Algorithm 2 Optimization Process for Walk Skill

```
procedure ResetRobot
    Deactivate gravity
    Put robot in the air
    Walk multiple steps
    Stop walk
    Activate gravity
    Set robot on the ground
end procedure
objective_values=[ ]
ResetRobot()
Wait 1 second
if robot has fallen then
    return [0,0,0,0]
else
    for direction in [$forward, backward, sideward, turn$] do
        $v_{cmd}$ = [0,0,0]
        travelled_distance = 0
        while true do
            Increase cmd_vel linearly in walk direction
            ResetRobot()
            while robot not fallen and not 12 seconds passed do
                Walk with $0.25 * v_{cmd}$ for 1 second
                Walk with $0.5 * v_{cmd}$ for 1 second
                Walk with $v_{cmd}$ for 6 seconds
                Walk with $0.5 * v_{cmd}$ for 1 second
                Walk with $0.25 * v_{cmd}$ for 1 second
                Stop walk
            end while
            Measure traveled distance in walk direction
            if robot has fallen or traveled distance did not increase then
                Break while loop
            end if
        end while
        objective_values.append(travelled_distance)
    end for
    return objective_values
end if
```

Table 5.1: Ranges of the optimized parameters.

Parameter	Lower boundary	Upper boundary	Unit
step frequency	1.00	3.00	Hz
double support ratio	0.00	0.50	
foot distance	0.15	0.25	m
foot rise	0.05	0.15	m
torso height	0.39	0.43	m
torso phase offset	-0.50	0.50	
torso lateral swing ratio	0.00	1.00	
torso rise	0.00	0.05	m
torso x offset	0.05	0.05	m
torso y offset	0.05	0.05	m
torso pitch	-0.50	0.50	rad
torso pitch proportional to v_{cmd_x}	-2.00	2.00	$\frac{\text{rad s}}{\text{m}}$
torso pitch proportional to v_{cmd_yaw}	-0.50	0.50	$\frac{\text{rad s}}{\text{m}}$

Webots includes a method to automatically create a corresponding URDF for a given robot model in the simulator's proto format. This URDF still needs some manual modifications to obey the REP 120 (see Section 5.2.4). The creation of the corresponding MoveIt configuration can then simply be done using the MoveIt Setup assistant [7]. Most importantly, two kinematic solvers need to be defined for legs.

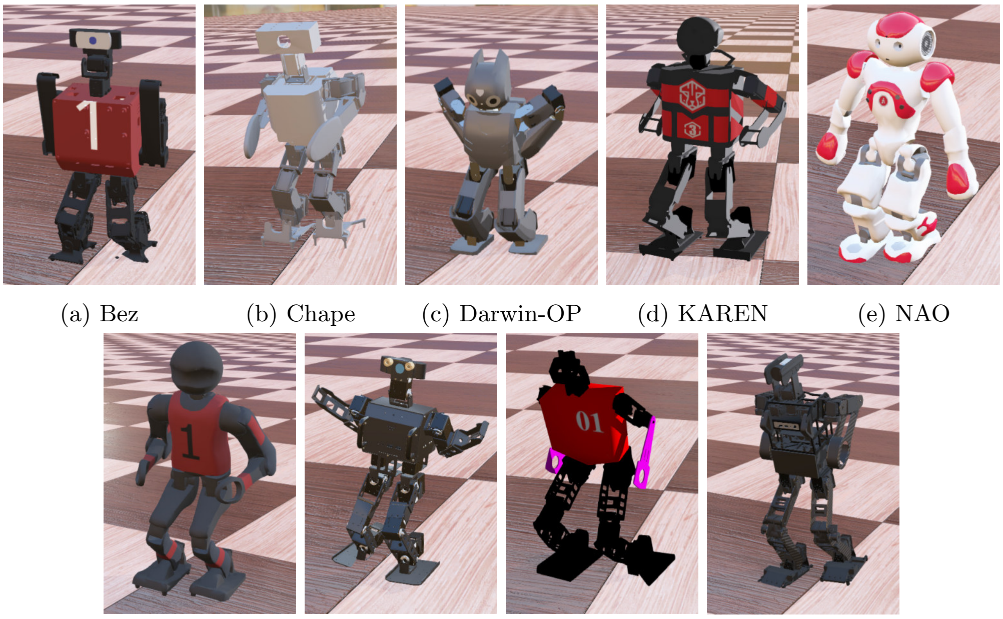
This collection is made public [1] and can therefore be used by other researchers to evaluate their approaches and compare them to the baseline we provide. Images of the used simulator models can be seen in Figure 5.7 and further specifications of these robots can be found in Table 3.1.

5.3.7 Evaluation

Different aspects of the walk skill were investigated. First, the parameter optimization is evaluated to show that the OptiQuint approach is feasible. Then, the generalization to other robot models is shown, and baseline values are provided. This is followed by a comparison of the different IK solvers of MoveIt. Finally, a qualitative evaluation of the walk skill is given.

Parameter Optimization

We evaluated the different optimization approaches provided by the Optuna library (see Section 5.2.4) to see if they all are applicable and to find out which of them performs best. We also tested the multivariate versions of the MOTPE and TPE sampler, as it is claimed that these perform superiorly [FKH18]. For each sampler, 1,000 trials were conducted. Additionally, 1,000 and 10,000 trials were chosen randomly to see if similar good results could also easily be achieved by random sampling. To ensure that the used robot type does not bias this experiment, it was conducted for three different robots, the Wolfgang-OP, the OP3, and the Bez. These were chosen since they have different sizes, weights, foot shapes, and degrees of realism. Each combination of sampler and robot type was only executed once due to the long runtime of the experiment. Since the optimization process is not deterministic (the first trials are always chosen randomly), the results can be influenced by noise. Still, in our experience in preliminary optimization experiments, the variance in achieved objective values for studies with such a high number of trials



(a) Bez (b) Chape (c) Darwin-OP (d) KAREN (e) NAO

(f) NUgus (g) OP3 (h) RFC2016 (i) Wolfgang-OP

Figure 5.7: Collection of the used Webots simulator models to test generalization.

was low. The parameter ranges are chosen as displayed in Table 5.1. The results are displayed in Table 5.2 and Figure 5.8.

The independent MOTPE sampler finds the parameter set with the highest scalarized objective value. The results of its multivariate version are inferior for all robots. Similarly, the independent TPE implementation finds in two out of three a better solution than the multivariate one. Additionally, it can be observed that the independent MOTPE provides better results for all three robots in comparison to the independent TPE. The NSGA-II sampler performed just slightly inferior to MOTPE, but the CMA-ES sampler provided clearly inferior results. Still, all of them outperform the baseline of the 1,000 trials random sampling.

Interestingly, the execution time of these approaches also varies. This is due to the structure of the objective function. Since the trial stops when the robot has fallen in each walking direction, the length it needs to be computed is inversely proportional to

Table 5.2: Scalarized objective values for different robot-sampler combinations (higher is better). Based on [BZ22]

Name	Wolf.	OP3	Bez
Multivariate MOTPE 1,000	1.33	1.81	0.24
Independent MOTPE 1,000	1.39	1.89	0.72
Multivariate TPE 1,000	1.25	1.83	0.31
Independent TPE 1,000	1.35	1.81	0.55
CMA-ES 1,000	0.90	1.53	0.69
NSGA-II 1,000	1.39	1.78	0.63
Random 1,000	0.64	1.36	0.17
Random 10,000	1.09	1.48	0.36
MOTPE Combined 1,500	1.52	1.95	0.74

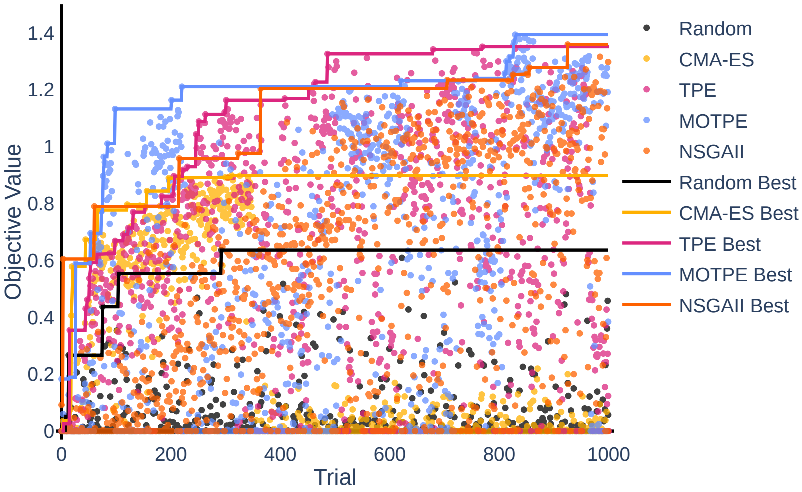


Figure 5.8: Plot of the optimization history of different samplers for the Wolfgang-OP. It can be seen that MOTPE already finds good solutions after a few trials. Both TPE and NSGA-II need more trials to find a similarly good parameter set but improve on it till the end of the trials. CMA-ES becomes stuck at around trial 300 and does not manage to further improve the parameters. Randomly trying parameters does not yield significantly lower results and thereby shows that it is not simple to find good parameters.

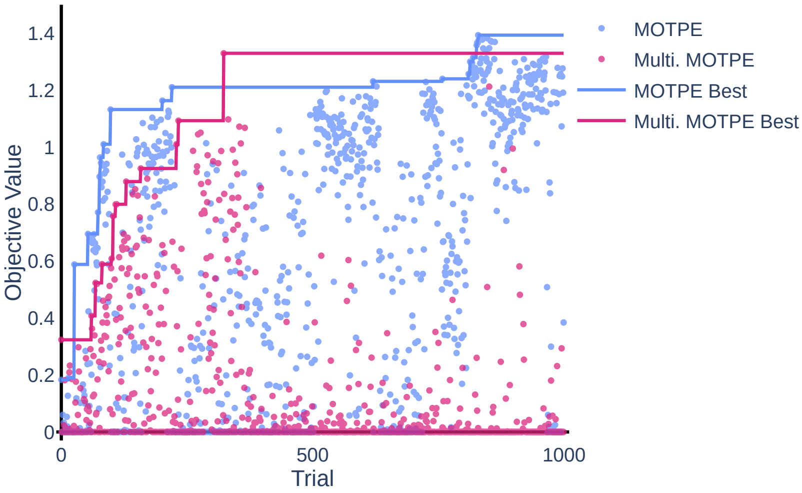


Figure 5.9: Comparison between independent MOTPE and multivariate MOTPE. The independent MOTPE sampler tries more parameter sets that are similar to the current best set, which has, therefore, also a high objective value. The multivariate version diverges more from the current best, thus also creating many sets with low or zero objective values.

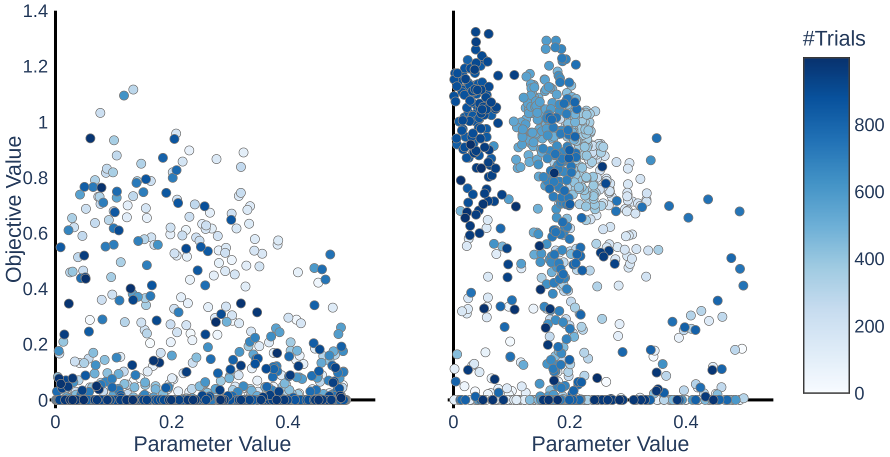


Figure 5.10: Comparison of the sampling between multivariate MOTPE (**left**) and independent MOTPE (**right**). Here, the $\theta_{double\_support\_ratio}$ parameter is shown as an example. It can be observed that the independent version focuses on a certain area in later trials. The multivariate version also samples more equally in later trials resulting in many objective values of zero. Based on [BZ22].

the quality of the tested parameters. The time that the samplers need to compute the next parameter set based on the current knowledge is insignificant in comparison to the time that is necessary to compute the physical simulation. Therefore, samplers that test many low-quality parameters can test more in the same amount of time. This is especially interesting when regarding the multivariate and the independent MOTPE/TPE samplers. The multivariate MOTPE approach needed ca. 8 hours for 1,000 trials on a machine with an AMD Ryzen 59000x 12-core CPU. The independent version thrice as long (ca. 24 hours). The reason for this is the difference in objective values that are achieved in later trials (see Figure 5.9). This is again a result of the difference in sampling strategy. While the independent MOTPE focuses on high-value areas in later trials, the multivariate version explores more of the parameter space (see Figure 5.10). Choosing between the multivariate and independent versions seems to be a trade-off between exploration and exploitation while at the same time influencing the run time of the experiment. We tried to get the best of both worlds by first sampling 1,000 trials using the multivariate approach and then 500 trials using the independent version. This results in a similar runtime as only using the independent approach while providing better results (see *MOTPE Combined* in Table 5.2).

We optimized 13 parameters for each robot. These are more than have been optimized in the related work (see Section 5.1.4). Furthermore, we have optimized not only one walking direction but all of them. Our results seem to indicate that it is possible to optimize all necessary parameters of an omnidirectional walk engine without issues due to the curse of dimensionality. Furthermore, we used a complete black box approach that needed no knowledge of the walk approach. Therefore, we can assume that this optimization approach will also work on other walk skills with a similar amount of parameters.

Generalization

One point of the OptiQuint approach is the usage of Cartesian splines to achieve generally usable skills that do not depend on a certain kinematic configuration. To prove that this goal was actually achieved, the walking skill was applied to the collection of Webots

Table 5.3: Maximal walk velocities for different robots [BZ22]

Robot Platform	Team/Company	Height [m]	Forward [m/s]	Backward [m/s]	Sideward [m/s]	Turn [rad/s]
Bez	UTRA	0.50	0.21	0.05	0.12	2.45
Chape	ITAndroids	0.53	0.30	0.36	0.10	2.09
Gankenkun	CITBrains	0.65	-	-	-	-
KAREN	MRL-HSL	0.73	0.54	0.48	0.18	3.10
NUgus	NUbots	0.90	0.38	0.42	0.26	1.97
RFC2016	01.RFC Berlin	0.73	0.37	0.40	0.36	1.99
Wolfgang-OP	Hamburg Bit-Bots	0.83	0.48	0.51	0.22	1.89
Darwin/OP2	Robotis	0.45	0.29	0.29	0.12	2.47
OP3	Robotis	0.51	0.45	0.54	0.13	2.01
NAO	Aldebaran	0.57	0.43	0.58	0.25	0.85

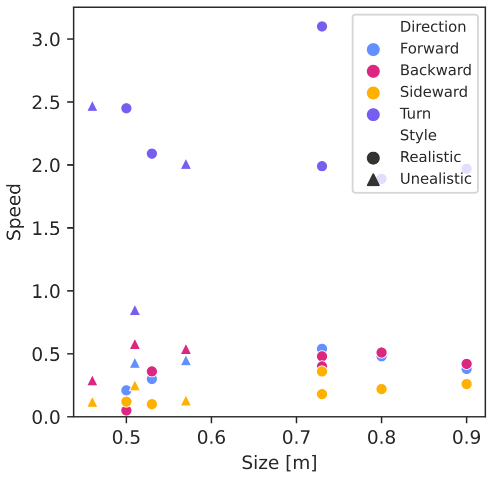


Figure 5.11: Relationship between robot size and achieved walk velocities for the values in Table 5.3. The turn speed is given in $\frac{\text{rad}}{\text{s}}$ and the other directions are given in $\frac{\text{m}}{\text{s}}$. The robots with unrealistic models have triangular markers. No clear correlation is visible. This could also be due to the different degrees of realism in the models.

robot models (see Section 5.3.6). For each robot, the parameters were optimized using the combined MOTPE approach with overall 1,500 trials (see Section 5.3.7). The results are displayed in Table 5.3 and the optimized parameters can be found in Table 8.1.

The approach was successfully applied to nine out of ten tested robots. The Gankenkun robot was the only one which did not work. The reason for this is that it is the only robot in the collection that uses parallel kinematics. Since the used IK solvers rely on URDF as a description for the robot, it was not possible to apply them to the Gankenkun. It is generally impossible to model a parallel robot with URDF. We assume that the walk skill could be applied to the robot if a custom IK solver is implemented, but we did not test it.

We also compared the achieved velocities with the size of the robots (see Figure 5.11). There seems to be no clear correlation. This might be influenced by the fact that the models which are not from the HSL are unrealistic, especially in regard to their joint parameters. Therefore, the comparably small OP3 robot achieves comparable performances as larger platforms.

It is difficult to compare the achieved speeds to existing results since other researchers have not used the same simulation environment with the simulated artificial grass. Furthermore, there are no baseline values for most of the robots from the other RoboCup

Table 5.4: Darwin-OP forward speed

	OptiQuint	Robotis [26]	Silva et al. [SPCB17]	Zhang et al. [ZJF+22]
Forward $[\frac{\text{m}}{\text{s}}]$	0.29	0.24	0.288	0.488
Used Webots	✓	✗	✓	✓

Table 5.5: NAO forward speed

	OptiQuint	Kulk et al. [KW+08]	Gouaillier et al. [GHB+09]	Gil et al. [GCS19]	Kasaei et al [KLP19]
Forward $[\frac{\text{m}}{\text{s}}]$	0.43	0.139	0.1	0.049	0.805
Used Webots	✓	✗	✗	✓	✗

teams. Therefore, we only compare the results of forward walking for the Darwin-OP and the NAO, where previous work exists (see Tables 5.4 and 5.5).

For the Darwin-OP, the OptiQuint approach achieves a higher velocity than the claimed maximal speed given by the manufacturer. It also slightly surpasses the speed achieved by Silva et al. [SPCB17], which also used the Webots simulator but with a harder floor, therefore, in a simpler environment. Zhang et al. [ZJF+22] achieved a higher speed also using the Webots simulation but also did not use a soft ground. Furthermore, they only optimized to walk as fast as possible in the forward direction without exactly following a given command velocity.

For the NAO, the OptiQuint approach outperforms the works of Kulk et al. [KW+08] and Gouaillier et al. [GHB+09], which were tested on the real robot. It also surpasses the speed of Gil et al. [GCS19], which used the Webots simulator. Only the work of Kasaei et al. [KLP19] manages to achieve a higher velocity than our approach. However, this was done in the Simspark simulator, which uses a very unrealistic model of the robot.

IK Comparison

As stated in section 5.2.4, different IK options are available through the MoveIt interface. While earlier work [RHSZ18] already compared these IKs for different robot models, it was always done for random poses. We wanted to investigate this special case of following spline trajectories, where the solution is always near the current position. Our hypothesis is that simple IK approaches, e.g., using a Jacobian, might outperform more complex approaches in this case because they have less initialization overhead.

We evaluated the KDL, TracIK, and BioIK solvers. The latter also provides a gradient and Jacobian mode, with single- and multi-threaded solving, which we also tested. We evaluated each solver for three different robots (Wolfgang-OP, Darwin-OP, NAO) to avoid a bias in our results. The resolution was set to 0.01 mm and the timeout to 10 ms for each leg. Each solver-robot-combination was tested by running it for 100 s while the robot is walking with 0.1 $\frac{m}{s}$ forward. The first iterations were removed from the data since these were always high outliers and they can be avoided by adding an initialization procedure to the walk skill. The time to solve both foot positions was measured. All tests were performed on the same ASUS mini PC (see Table 3.3). Furthermore, the combination of running on isolated CPU cores with the round-robin scheduler, as described in Section 4.4.2, was used. The walk engine was given access to two cores to allow testing of the multi-threaded solvers.

The results of the experiment are displayed in Figure 5.12 and Table 5.6. The simple KDL solver performs best in terms of mean and maximal solving time. We don't know why the BioIK in Jacobian mode has high outliers for the Wolfgang-OP but not for

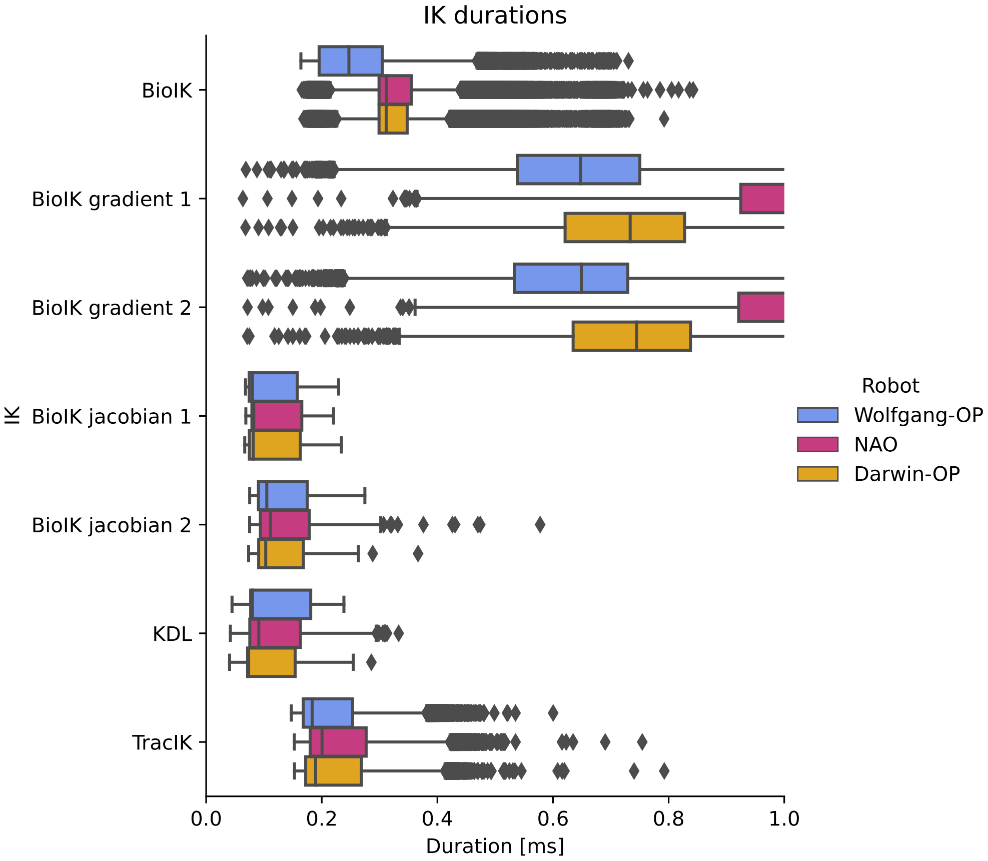


Figure 5.12: Note that this only shows a part of the plot for better visibility. Some outliers are therefore not shown. See Table 5.6.

Table 5.6: Comparison of different IK solvers

IK Name	Wolfgang-OP		NAO		Darwin-OP	
	mean	max	mean	max	mean	max
KDL	0.12	0.24	0.12	0.33	0.11	0.29
Trac IK	0.21	0.60	0.23	0.75	0.23	0.79
BioIK memetic	0.27	0.73	0.33	0.84	0.33	0.79
BioIK gradient 1	0.66	1.83	1.11	2.95	0.74	2.21
BioIK gradient 2	0.62	1.84	1.09	2.17	0.74	1.97
BioIK Jacobian 1	0.21	57.22	0.12	0.22	0.12	0.23
BioIK Jacobian 2	0.13	0.27	0.13	0.58	0.13	0.27

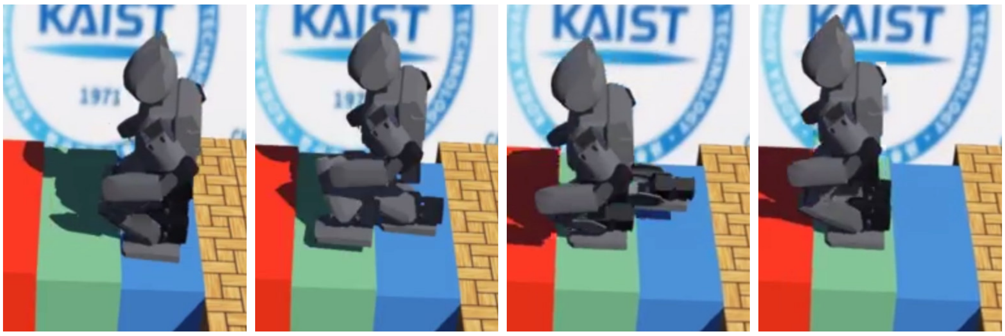


Figure 5.13: Simulated Darwin-OP walking up the stairs for the Running Robot Competition.

the others. One explanation would be the not intersecting axes in the hip joint of the Wolfgang-OP, which make solving the IK more difficult. Otherwise, we could not see large differences between the used robot models.

Not running the IK on isolated cores leads to outliers with a long solving duration. The maximal time for solving both legs using BioIK increased from 0.73 ms to 15.58 ms. This aligns with our findings in Section 4.4.2 and highlights again the strong influence of the Linux scheduler.

In summary, we can conclude that the KDL solver performed best for this special domain of Cartesian poses that are close to each other. BioIK might perform better for arbitrary poses and includes other types of goals that the IK can solve, but both of these factors are not relevant in our case.

Qualitative Evaluation

The Hamburg Bit-Bots applied the walk skill in both in simulation and on the real robot. In the simulation, the team scored third place in the RoboCup 2021, first place in the RoboCup Brazil Open 2021, and second place in the HLVS 21/22. Furthermore, the team also used it in the Running Robot Competition [21] in 2020 and 2021. In this competition, a simulated Darwin-OP robot had to be navigated through a parkour. This included walking on stairs, on a slope, and over a narrow bridge. We were able to modify the walk skill to allow stepping on stairs (see Figure 5.13). The slope was automatically solved by the PID controller stabilization. The narrow bridge was solved by optimizing parameters with a limited $\theta_{foot\_distance}$. In one year, the navigation was done by using a RL approach using footstep planning, and in the other year, using the ROS navigation stack with velocity control.

Earlier versions of the walk skill with manually tuned parameters have been used by us on multiple RoboCup competitions in 2018 and 2019 on the Minibot and Wolfgang-OP (see Figure 5.14). In 2019, the push recovery technical challenge was won by using this walk skill (see Figure 5.16). We also used the walk skill with optimized parameters on the real robot in the RoboCup 2022 on the Wolfgang-OP. Generally, we observed that the optimized parameters from the simulation were not directly applicable to the real robot. Typically, the balance in the sagittal direction was not correct, and θ_{torso_pitch} needed to be manually corrected. This may be due to inaccuracies in the mass distribution of the simulation model or to the real robot having more backlash than the simulated one. However, making the pre-optimized parameters usable on the real robot is less work than manually tuning them from scratch. Furthermore, the automatic optimization found better solutions than our previous manual tuning, especially by using quicker steps.

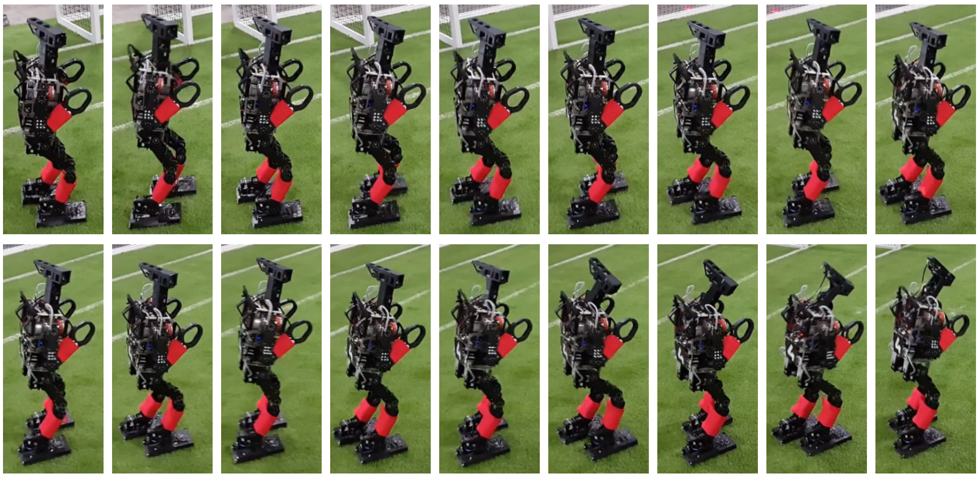


Figure 5.14: Images of the Wolfgang-OP walking during a RoboCup competition. See [1] for the video.

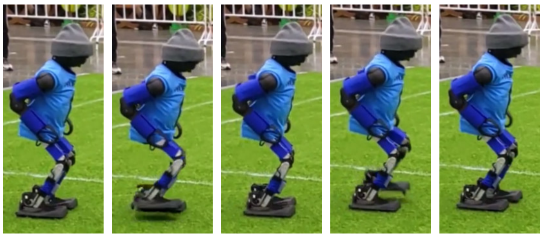


Figure 5.15: Images of the NUgus walking during a RoboCup competition using the OptiQuint approach. See [1] for the video.

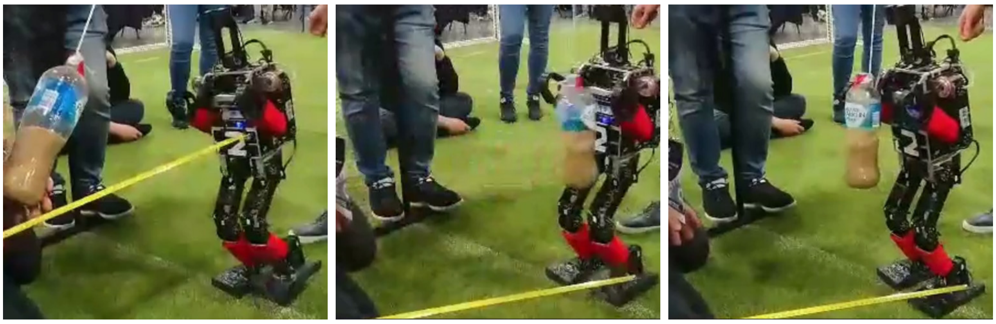


Figure 5.16: Images from the RoboCup push-recovery challenge. In the first image, the deflection of the weight (a bottle filled with sand) is measured, as this is the basis of the points that can be achieved. In the second image, the robot is hit by the weight. In the third image, the robot stopped walking and stabilized itself. See [1] for the video.

Still, the real Wolfgang-OP robot falls significantly more often than the simulated one. We assume that this is also influenced by the backlash of the servos, which further increases over time (see Section 3.6.4).

The team NUbots also used our walk skill on the simulated since 2019 [DHI+22] and successfully used it on their real robot in the RoboCup 2022 (see Figure 5.15). Parts of the walk skill have also been used as a basis for the work of Putra et al. [PMFC20].

5.4 Stand Up

The stand-up skill is based on the OptiQuint approach but implemented by Sebastian Stelter in his Bachelor's thesis supervised by me [Ste20]. The parameter optimization for the stand-up skill was completely done by me. The experiments were done collaboratively and published in [SBHZ21]. Therefore, the focus of this section will be the parameter optimization and the evaluation, while the implementation will be only described very briefly.

This section presents a discrete recovery skill implementation using OptiQuint. First, its implementation is described in Section 5.4.1. Since it is done analogously to the walk skill the section focuses on the differences. Then, the applied parameter optimization and its objective function are described in Section 5.4.2. Finally, the skill is evaluated in Section 5.4.3.

5.4.1 Implementation

There are three main differences to the walk skill. First, the stand-up skill is discrete and therefore ends its spline interpolation when $t = 1$ instead of beginning a new cycle. Due to this, it is started by using a ROS action instead of constantly listening to a ROS topic like the walk. The action also specifies in which direction the robot should stand up.

Second, the stand-up does not only move the legs of the robot but also the arms. Therefore, it uses four pose splines instead of two. All of them are in the frame of the robot's torso. The first knots for all splines are defined by the current pose of the robot. The last knots are defined by a predefined goal pose which allows the robot to start walking afterward directly. The remaining knots are defined by using the same procedure as in the walking, i.e., first relying on natural definitions and then using parameters for the remaining values.

Third, the stabilization is only performed during the last part of the motion. The robot is first lying on the ground, and then it makes ground contact with its arms and legs. During this time, the support polygon is large, and therefore the robot is inherently stable. Furthermore, it is difficult to compute the exact correct orientation for the robot's torso during this period, making any PID control complicated. Therefore, the stabilization is only used during the last phase of the stand-up motion, in which the robot only stands on its two feet.

A visualization of the splines can be seen in Figure 5.17. More details on the implementation can be found in [SBHZ21] and [Ste20].

5.4.2 Parameter Optimization

The parameters in the stand-up skill can be grouped into timing parameters and Cartesian parameters. The first group defines how long different parts of the motion take (see also Figure 5.17) while the second group defines the poses of the endeffectors. There are

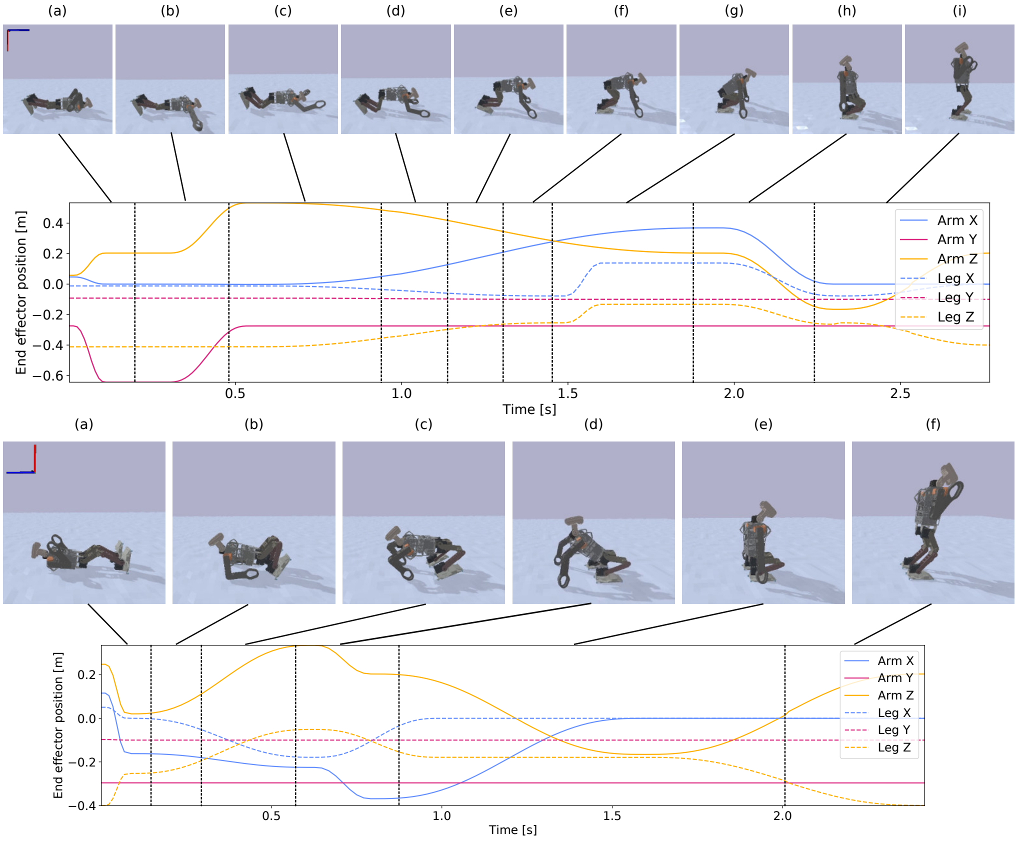


Figure 5.17: Visualization of the splines for the stand-up motion from the front (**top**) and back (**bottom**), together with images from the PyBullet simulation. The plot shows the right arm and foot in relation to the robot's torso with the coordinate system indicated in the first image. The trajectories for the left side of the robot are the symmetric equivalent. The black vertical lines indicate the timing of the different phases of the motion, which is defined by parameters. [SBHZ21]

nine timing parameters and six Cartesian parameters for the front motion. Analogously, there are six timing parameters and nine Cartesian parameters for the back motion. The high amount of timing parameters is understandable as the stand-up consists of multiple phases, while the walk only consists of double and single support phases. The Cartesian parameters of the front motion define the lateral and sagittal position of the hands during the pushing up of the torso, the minimal distance of the feet when they are pulled towards the torso, as well as the angles of the legs and torso during the center phases. The back parameters are similar, but they also define how high the torso is pushed from the ground as well as the positioning of the CoM before the last phase.

The parameters for the front and back motion were optimized independently from each other. This reduces the number of parameters for one optimization process (curse of dimensionality) and has no disadvantages as these motions are also independent of each other. The optimization of these parameters was done in PyBullet since these experiments were conducted before the HLVS was created. The simulation uses a random height map of 1cm to simulate the slightly uneven ground of a RoboCup field. Due to this height map and the randomness in the IK solver, the execution of the stand-up motion is not deterministic. Therefore, each parameter set is evaluated three times, and the objective value is based on the mean of the three repetitions. Similarly to the walking, the execution of the skill is interrupted if a fall has happened or if the IK finds no solution. The latter case can happen if, for example, the goal poses of the hands are at positions that the arms can never reach. An overview of the optimization process is detailed in Algorithm 3.

Algorithm 3 Optimization Process for Stand-up Skill

```
procedure ResetRobot
    Deactivate gravity and self-collision
    Place robot in the air
    Move joints to initial pose
    Place robot on the ground
    Activate gravity and self-collision
end procedure
procedure ExecuteStandup
    Start stand-up
    while True do
        Execute simulation step
        if fallen then
            return False
        end if
        if finished then
            return True
        end if
    end while
end procedure
successes = 0
for $i = 0$; $i < 3$; $i++$ do
    Randomize terrain
    ResetRobot()
    success = ExecuteStandup()
    if success then
        successes += 1
    end if
    Compute objective values
    if multi objective optimization then
        return objective values
    else
        Scalarize objective values
        return single objective value
    end if
end for
```

The naturally given objective function for a stand-up motion is a simple binary decision if the robot is standing afterward or not. While this is generally correct and simple to compute, it does not provide a clear gradient which makes optimization difficult. Another measurement is the time that the skill takes, as a quicker stand-up is generally better. This metric has a clear gradient but does not help to guide the optimization algorithm in finding successful stand-up motions. Therefore, we propose two additional objectives. The first uses the minimal achieved fused pitch angle to provide a gradient on how upright the robot is, and the second uses the maximal achieved height of the robot head to create a gradient on how close the robot comes to standing. This results in the following four objective functions that are based on the success rate ($f_s(\Theta)$), the total time ($f_t(\Theta)$), the minimal reached fused pitch ($f_p(\Theta)$) and the maximal reached head height ($f_h(\Theta)$).

$$f_s(\Theta) = \begin{cases} 0 & \textit{if successful} \\ 100 & \textit{if not successful} \end{cases} \tag{5.33}$$

$$f_t(\Theta) = 10 * min(time\_until\_standing\ [s], 10) \tag{5.34}$$

$$f_p(\Theta) = \min(fused\_pitch\ [deg]) \tag{5.35}$$

$$f_h(\Theta) = 100 - \max(head\_height\ [cm]) \tag{5.36}$$

In $f_t(\Theta)$, the time for the robot to become stable is included. This was important, as otherwise, the optimization learns to quickly perform the motion but lets the torso oscillate afterward. Additionally, we use the fused angles representation in $f_p(\Theta)$ to avoid gimbal lock issues.

They are scaled in themselves to provide values of the same range ($[0, 100]$), although they have different units. Thereby, a single-objective optimization can be performed by taking the sum of them. We will investigate different combinations denoted by the combination of the corresponding letters. In the case of multi-objective optimization, each of the summands is returned as an individual objective value.

$$f_{st}(\Theta) = f_s(\Theta) + f_t(\Theta) \tag{5.37}$$

$$f_{sth}(\Theta) = f_s(\Theta) + f_t(\Theta) + f_h(\Theta) \tag{5.38}$$

$$f_{stp}(\Theta) = f_s(\Theta) + f_t(\Theta) + f_p(\Theta) \tag{5.39}$$

$$f_{sthp}(\Theta) = f_s(\Theta) + f_t(\Theta) + f_h(\Theta) + f_p(\Theta) \tag{5.40}$$

5.4.3 Evaluation

The OptiQuint-based stand-up skill has been successfully used together with the HCM in the HLVS 21/22 and during the RoboCup 2022. Therefore, it has shown its general usability. Still, we evaluate different aspects of this skill in the following. First, the proposed objective functions and available optimization algorithms are compared. Then, the generalization to other robot models is evaluated. Afterward, the transfer of the optimized parameters to the real robot is evaluated. Finally, the influence of the PD controller is shown.

Parameter Optimization

The different possible objective functions (see Section 5.4.2) were investigated regarding two qualities. First, how fast it is possible to find a working solution where at least one of the three executions is successful (trials till success (TTS)) and, second, the minimal stand-up time that was achieved without falls. Additionally, different samplers were evaluated to see if there were differences between them. In this experiment, the NSGA-II sampler was not tested since it was not part of the Optuna library at that time. Each optimization was run for 500 trials (see Figure 5.18 for an exemplary plot). Each combination of objective function and sampler was run five times to account for the randomness in the parameter sampling process. The mean and the worst run of these five executions is compared in Figure 5.19.

Generally, it can be observed that the back motion is harder to optimize than the front motion. This fits well with our experience with manual parameter tuning. The back motion is more dynamic and closer to the kinematic limits, but it is also faster.

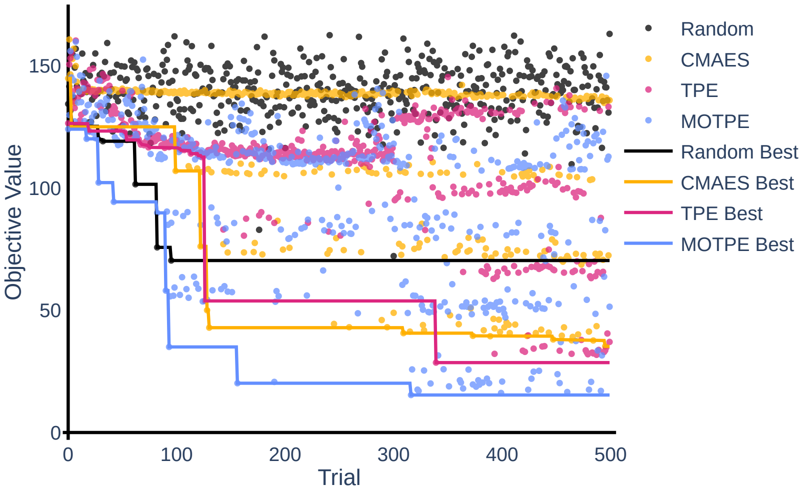


Figure 5.18: An exemplary plot of optimization runs with the different samplers. In this case, the stand-up from the back was optimized using $f_{sth}(\Theta)$ as the objective function. Each dot shows one trial result but only the $f_{st}(\Theta)$ value since these are the ones that are actually important and not just used for guidance. Based on [SBHZ21].

Table 5.7: Optimization results using MOTPE and $f_{sth}(\Theta)$

	front		back	
	Time [s]	TTS	Time [s]	TTS
Wolfgang	2.7	21.0	2.1	29.4
Darwin	2.9	35.8	n/a	n/a
Sigmaban	2.2	34.2	2.2	62.4

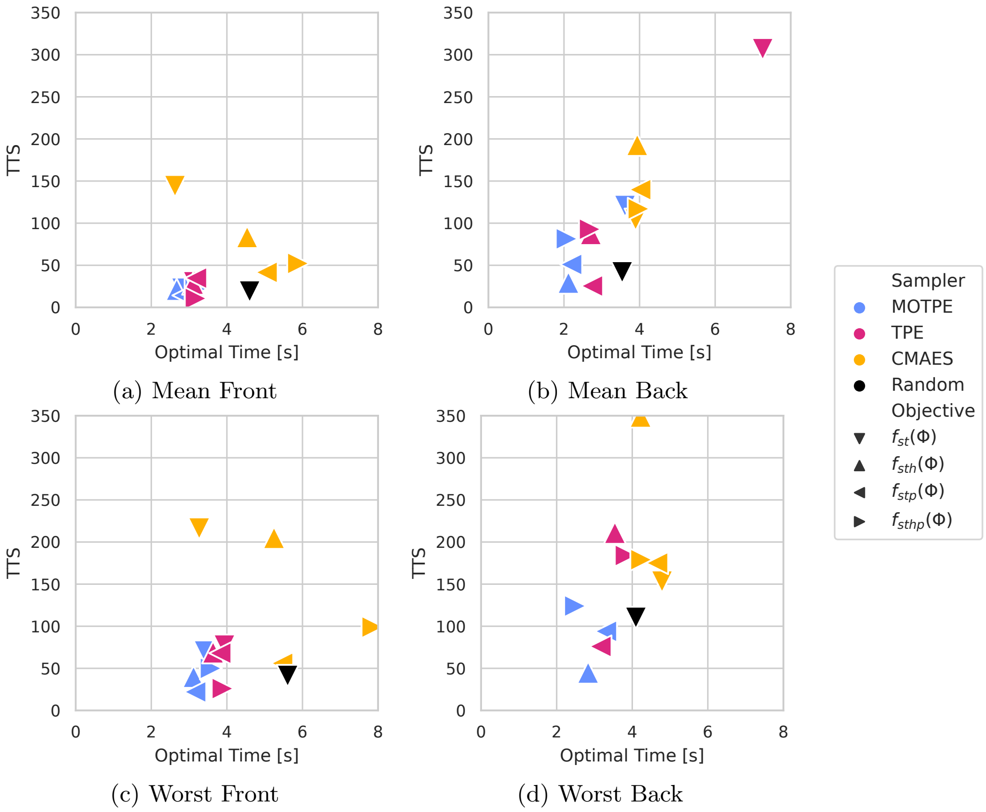
The CMA-ES algorithm performs clearly inferior to the TPE and MOTPE sampler for both metrics of optimal time and TTS. Even in comparison to the random sampler baseline, it does not perform well.

The MOTPE and TPE perform both well and clearly superior to the random baseline. Between them, the MOTPE has a slightly better performance, especially in regards to the achieved optimal time.

Regarding the different objective functions, it can be observed that $f_{st}(\Theta)$ has the highest variance and is the only objective where the TPE and MOTPE did not always find a solution. This was expected due to the missing gradient and supports our initial motivation of adding further objectives. Both of the options ($f_h(\Theta)$ and $f_p(\Theta)$) work in guiding the optimization algorithm, although the results of $f_h(\Theta)$ seem to be slightly superior. Combining both of them seems to lead to no further benefit.

Generalization

To see how well the skill generalizes to other humanoids, it was also tested on the Sigmaban and Darwin-OP robot in PyBullet, as these experiments were performed before the creation of the HLVS. For all robots, the MOTPE sampler and the $f_{sth}(\Theta)$ objective function were chosen since they performed best in the previous experiment. The results are displayed in Table 5.7. The Darwin robot was not able to stand up from the back due to limited movement of the hip joint but was able to stand up from the front. The Sigmaban was able to stand up from both sides.



(a) Mean Front

(b) Mean Back

(c) Worst Front

(d) Worst Back

Figure 5.19: Results of five times optimizing the parameters for standing up from the front (**left**) and back (**right**). In subfigure **c** the marker for the $f_{st}(\Theta)$ combined with TPE and MOTPE is missing since it did not find a solution. Based on [SBHZ21].

Sim2Real Transfer

The optimized stand-up skill was also tested on the real robot to see if a Sim2Real transfer is possible. Two other stand-up motions were used as a baseline. First, a keyframe-based motion that was used by the Hamburg Bit-Bots previous to the introduction of the spline-based skill. Second, the same spline-based solution but with manually tuned parameters.

Not all of the optimized parameter sets were applicable to the real robot. Therefore, multiple optimizations were done to see how many of them could be used. Each objective function was used three times, resulting in twelve parameter sets for each direction. These were tested on the robot, and the best one was chosen manually based on its performance on the robot. Of the twelve parameter sets for the front motion, two worked directly on the robot, five needed small fine-tuning, and five were not usable. As expected, the back direction performed worse as it is less stable. Here, no parameter sets were directly applicable, six could be made to work by minor tuning, and six were not usable. Out of these, the one which needed only a change of one parameter was used for further evaluation.

The largest issue was an unrealistic motor model, which assumed a too-high motor speed. Thus, the motion could not be performed on the robot as quickly as in the simulation and, therefore, failed during the dynamic phases due to being too slow. This could be solved for some parameters by increasing the length of some of the parameter phases.

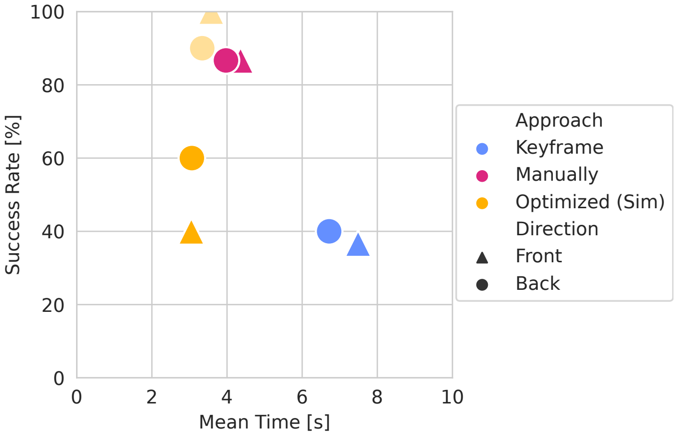


Figure 5.20: Performance comparison between the different stand-up skills for both directions in terms of mean time and success rate. Additionally, the values that the optimized motion achieves in simulation are plotted transparently to visualize the sim-to-real gap. The optimized parameters need the least time but are not as stable as the manually tuned ones. Furthermore, they are less stable than in simulation. Still, the automatically optimized parameters as well as the manually tuned ones, outperform the keyframe animation. Based on [SBHZ21].

All three motions (keyframe, manually, and automatically optimized splines) were evaluated 30 times each for front and back direction with the same Wolfgang-OP robot on the artificial turf with a 3cm blade length. The time was measured till the robot was standing with angular velocities smaller than $0.15 rad/s$. The results are displayed in Figure 5.20.

Stabilization

The influence of the stabilizing PD controller is visualized in Figure 5.21. During the unstable part of the motion (around second two in Figure 5.21c), the PD controller starts to manipulate the torso goal orientation (and thereby the joint goal positions). The correction of the torso pitch orientation can be seen in Figure 5.21d. This allows a very fast motion as the robot can move dynamically from one pose to another and has fewer oscillations of the torso orientation at the end. The open-loop keyframe animation had to perform the same motion more slowly and include small pauses that settle oscillations.

5.5 Kick

Based on the OptiQuint approach, a kick skill was implemented by Frederico Bormann, Timon Engelke, and Finn-Thorben Sell in a project supervised by me [BES19]. The parameter optimization was done by Timon Engelke, based on my implementation of the other skills. Since my contribution to this was small, the kick skill will only be presented very briefly. Still, it is presented here to show that the OptiQuint generalizes to all three types of motion skills (see the introduction of this chapter).

A kick is a discrete motion skill like the stand-up. Still, it has a complex goal, like the walking. This consists of the position of the ball, the direction in which the ball

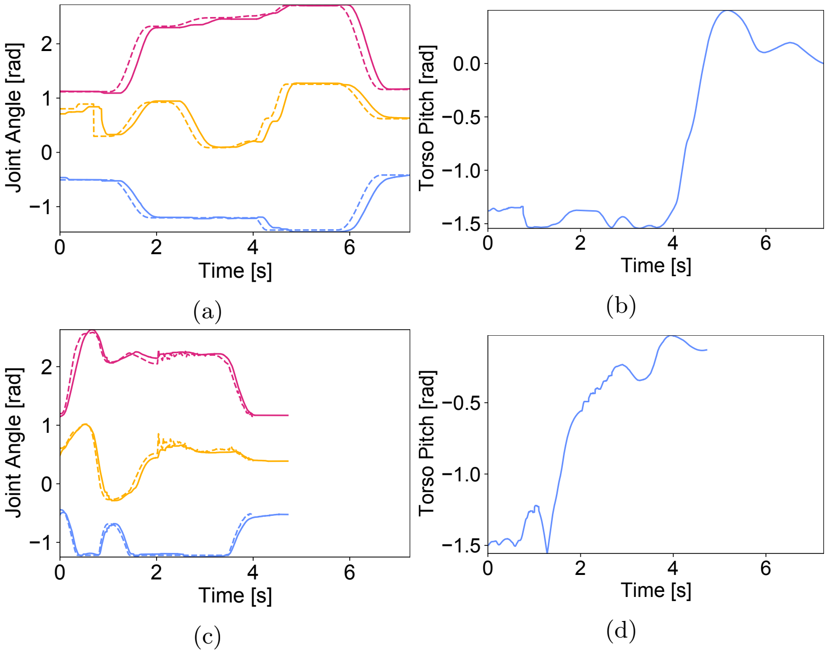


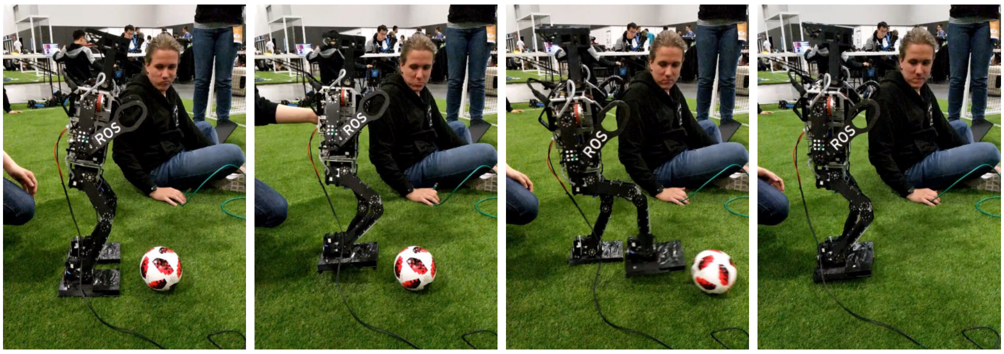
Figure 5.21: Comparison of a stand-up from the back using the keyframe-based motion (**a**) and the spline-based solution with manually tuned parameters (**c**). For both, the leg pitch joint positions (**blue** is the ankle, **red** is the knee, **yellow** is the hip) are plotted together with the joint goal positions as a dashed line. Additionally, the corresponding torso orientation is shown in **b** and **d**. The stabilization by the PD controller is visible at around two seconds in **c** and the influence on the torso pitch is visible in **d**. [SBHZ21]

should be kicked, and with what speed. This information can be used to define the knots of the foot spline at the impact point. Here, the ability of quintic splines to define the velocity at a certain point is especially interesting. This greatly facilitates the control of the kicking speed. Thereby, the most important knot in the motion is already defined. Additionally, the start knots can be defined by the robot's pose when the skill is invoked, and the end knots can be set to the same values so that the robot returns its original pose. The robot also needs to raise its foot before the kick and retract it afterward. The corresponding knots are partially defined by naturally given values and partially by parameters. An image sequence of a forward kick on the real robot can be seen in Figure 5.22, and a sequence of a simulated sideward kick can be seen in Figure 5.23.

The stabilization is also done using a PID controller that modifies the orientation of the torso. Unlike in the other presented skills, the CoP of the support foot is used as input data which works well as the support foot never leaves the ground. The PID control is especially useful to compensate for the additional forces of hitting the ball with the foot.

Previously, a keyframe-based solution was used. This consisted of two independent motions for kicking straight ahead with the left foot and the right foot. It did not allow other kicking directions, and the robot needed to be positioned exactly in relation to the ball so that the foot hit it.

The new skill has been integrated into the HCM-based architecture and has been used in the HLVS 21/22 and on the RoboCups in 2019 and 2022. The ability for side kicks also allowed to implement a new soccer strategy where the robot kicks the ball to one of its teammates if an opponent is standing in front of it. It was not yet tested on



(a) Before the kick (b) Raise a foot (c) Kick the ball (d) Retract the foot

Figure 5.22: Photos of a forward kick motion on the real Wolfgang-OP robot. [Eng21]

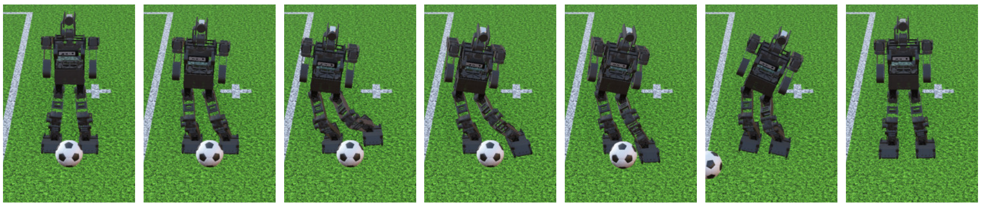


Figure 5.23: Images from a sideward kick in the Webots simulation. [Eng21]

other robots, but we expect it to generalize similarly well as the walk skill, since it also only uses the legs. One limitation of transferring it to other platforms is that it requires an estimation of the CoP and, therefore FPSs. Using IMU based stabilization would allow it to be used by a wider range of robots as these are included in every humanoid (see also Table 3.1).

5.6 Summary and Future Work

This chapter presented the OptiQuint approach, which uses Cartesian quintic splines to generate movement patterns for motion skills. A classification of humanoid motion skills into three classes was proposed. Then it was shown that the OptiQuint approach is able to create skills for all of these by using a walk, a stand-up, and a kick skill as examples.

The main novelty of the approach is the automatic optimization of the spline parameters. Different optimization algorithms were investigated to identify the best choice for this task. This showed that using a posteriori scalarization with multi-objective optimization leads to better results than a priori scalarization with single-objective optimization. Furthermore, the TPE algorithm proved to perform superior to the CMA-ES approach. The optimized skills were able to outperform manually searched parameters in simulation. We also optimized a larger number of parameters than in previous works.

The transfer to the real robot was also investigated. Due to the sim-to-real gap, the optimized parameter sets performed worse on the real robot than in simulation. Still, some of them were directly applicable, and a large proportion could be fixed by minimal tuning, thus still reducing the amount of manual work. The accurate modeling of the motor parameters proved to be especially important for the transferability to the real robot.

It was also shown that the OptiQuint approach is not limited to the Wolfgang-OP platform by applying the optimization to various other humanoids in simulation. The walk skill worked on nine out of ten tested robots and only had problems with parallel kinematics. Since the implementation of the locomotion skill was done in ROS and it works for all humanoid robots with serial kinematics, it can be easily used by others. This is especially interesting for new teams that want to start participating in the HLVS or HSL.

Still, there are many points which could be improved in the future. The walk skill could be tested on larger humanoids which was not possible as no models from the HLVS adult-size were made open source yet. The other presented skills could also be tested on a larger set of models. Here, we expect the stand-up skill to have more problems as it also uses the arms. Identifying these issues could lead to an improvement of this skill. It could also be tested if higher order splines would further improve the motions by limiting jerk.

While the parameter optimization works well, the computation time could be further improved. Instead of linearly increasing the commanded velocity for each parameter set, a binary search approach could be chosen. For this, first, the currently highest velocity would be tested. If it works, we can assume that all slower speeds also work and directly start optimizing from there. If it does not work, the velocity is halved, and the procedure is done again.

Another improvement for the optimization would be a better handling of kinematically unsolvable parameter sets. Before evaluating a new parameter set in simulation, it could be tested if a cycle of the resulting pattern is solvable by the IK. If this is not the case, a negative objective value could be returned to guide the optimizer away from this area of the parameter space. This would also reduce the computation time as IK solving a single cycle is much faster than running the full simulation with a not working parameter set.

Chapter 6

Reinforcement Learning

Bipedal walking is one of the most critical problems in humanoid robotics. It is challenging to solve due to the high center of mass, the changing support polygon, the complex dynamics, and the contact forces with the ground. Still, walking abilities have been achieved with various approaches in the past (see Section 5.1). In recent years, deep reinforcement learning has become a popular approach. First proving its viability in simulation [SML$^+$15, PBYVDP17] and more recently also on the actual hardware [LCP$^+$21, RB21].

One key contributor to achieving feasible policies has been the introduction of a reference motion [PALvdP18], which simplifies the problem and leads to less exploitation of simulation inaccuracies. Typically, these reference motions are created by using MOCAP of humans or animals. Instead, we created a novel approach (called DeepQuintic) which generates reference actions during training by using the OptiQuint approach (see Chapter 5). This reference is only used during training and is not further required afterward. Additionally, we investigated different design choices for training the policy, for example, the used action space and the type of reward.

We will only discuss bipedal walking in this chapter, but the DeepQuintic approach can also be applied for other motions, e.g., by using the OptiQuint-based implementation of a stand-up (see Section 5.4). Timon Engelke used the DeepQuintic approach in his Bachelor's thesis to learn a kick motion based on the OptiQuint-based kick (see Section 5.5) [Eng21], but it will not be discussed further as it just applies the same approach on a different reference motion.

Originally, it was planned to train the policy (partly) on the real robot. Therefore, the robustness of the hardware was improved (see Chapter 3), and an independently working fall management was implemented (see Chapter 4). Finally, this plan was dropped due to the long training times of the policy and the resulting wear of the hardware. Furthermore, the possibilities for accurate simulation improved with the addition of backlash and modeled artificial grass in Webots.

The structure of the chapter is as follows. First, the related work is presented in Section 6.1. Then, the DeepQuintic approach is presented in Section 6.2 and evaluated in Section 6.3. Finally, a summary is given in Section 6.4.

6.1 Related Work

This section presents related works in the field of RL for robot motions. Many of these works were published while this thesis was created, and it was, therefore, not possible to build up on these. We already discussed general learning approaches for motions in Section 5.1.3 and, therefore, we will focus on work that is closely related to the

DeepQuintic approach. The remainder of this section is structured as follows. First, we show different other approaches that utilize reference motions for RL in Section 6.1.1. Then, different choices for action and state space are discussed in Section 6.1.2.

6.1.1 RL with Reference Motions

In recent years, many RL approaches utilized reference motions to guide the learning process for robotic motions. Different sources for these references are possible. One is the usage of MOCAP or manually created keyframe animations [PALvdP18, YYH$^+$20, EPY$^+$22, BTB$^+$22]. Since this requires a comparably high effort, the automatic extraction of motions from videos was investigated in [PKM$^+$18]. These references have not been recorded on the robot itself but on humans or animals with other kinematic and dynamic properties. Therefore, they are generally not optimal for the target platform.

To circumvent this issue, references that are optimized for the target platform can be used. Li et al. [LCP$^+$21] used a library of fifth-order Bézier curves in joint space as a reference and as part of the policy's input. They were optimized using CFROST [HHH$^+$19] for the target platform, but no further investigation on the influence of the quality of this motion was performed. Xie et al. [XBC$^+$18] restricted the policy to learn only corrections to a reference motion in joint space. Similarly, Zhang et al. [ZJF$^+$22] let their policy learn an action offset to the reference motion. Melo et al. [MMdC22] learned corrections to their walk engine to keep stable during pushes. They tried to imitate the ankle and hip strategy for push-recovery by working in joint space, and further restricting the policy to some of the leg joints.

Rodriguez et al. [RB21] successfully learned an omnidirectional walk policy without reference motion. They defined a nominal pose which is used as a regularization term in the reward function. This could also be interpreted as using a single-frame reference. To achieve this, they needed to apply curriculum learning by gradually increasing the velocity. They also used a beta function to limit the actions, which are represented in joint space relative to the current position, to the possible joint positions.

6.1.2 Choice of Action and State Space

Different choices of action space (i.e. joint torque, joint velocity, joint position, and activation for musculotendon units) were investigated by Peng et al. [PvdP17]. While they included Cartesian information in the state (i.e. the positions of all links), they did not investigate actions in Cartesian space nor states in joint space. They suggested that high-level action parameterization, e.g., by using a PD controller, improves the performance of the policy in most motions. In recent years, such PD controller-based joint positions became a common choice when using a reference motion [LCP$^+$21, RB21, PALvdP18, YYH$^+$20] although their usage may lead to inferior performance without reference motion [RTvdP20].

Even though the definition of actions in Cartesian space seems intuitive, the research on this is sparse. Duan et al. [DDG$^+$21] defined their action in a task space represented by a combination of Cartesian positions of the feet relative to the base and joint positions of their hip yaw and foot pitch joints. This action, together with a reference action from a spring-mass model and additional information of the state, is then processed by an inverse dynamics controller to compute the torque that should be applied to the joints. Their state includes, among others, the orientation of the robot's base, which is given as a quaternion, the position of the feet in Cartesian space without their orientation, as well as the position and velocities of all joints. While this work has a partial Cartesian action, it did not investigate complete Cartesian actions, and the trained policy could only walk forward with different velocities.

Outside of bipedal locomotion, Bellegarda et al. [BB19] showed that a policy in Cartesian space outperforms one in joint space for creating a skating controller for a quadruped. The actions in both spaces were modeled as discrete offsets rather than in a continuous space. Martín-Martín et al. [MMLG$^+$19] demonstrated the benefits of using variable impedance in Cartesian spaces instead of joint positions or torques for controlling robotic arms. Actions described as Cartesian offsets from a reference trajectory were used to create a jumping motion [BN20] for a quadruped robot. The same robot learned to walk using absolute Cartesian actions with impedance control [BCLN22]. The results were compared to actions in joint space and simple PD control qualitatively, and it was observed that using a joint-space-based action leads to unnatural motions when transferring to another simulator. In all four works, the stated dimensionality of the actions indicates that Euler angles have been used, although it was not explicitly discussed.

The importance of choosing the optimal state representation was investigated, showing, among others, that including Cartesian coordinates can improve learning, especially for complex environments [RTvdP20, KH20].

In summary, while different Cartesian-space-based actions have already been investigated, a direct comparison between absolute, continuous Cartesian, and joint space actions has not been made yet. Furthermore, different orientation representations have not been investigated. Another commonly used way for representation are quaternions. These don't have the gimbal lock issue of Euler angles, but their discontinuity can lead to issues if they are used for learning approaches [SDN09]. Another option are fused angles [AB15], which work similarly to Euler angles but are also not continuous. As a solution for the continuity issue, a 5D and 6D representation have been proposed [ZBL$^+$19]. It was shown that the 5D and 6D representation improves learning tasks for object orientation estimation and inverse kinematics. In both tasks, the 6D representation performed better than the 5D representation.

6.2 Approach

The problem of omnidirectional bipedal walking can be modeled as a Markov Decision Process (MDP) [SB18], where the robot interacts with an environment. During each time step t, the robot is given the current state of the environment s_t and uses its policy $\pi(a_t|s_t)$ to choose an action a_t. This action is applied to the environment, and the subsequent state s_{t+1} is given together with a scalar reward r_t that describes the desirability of this transition. There are different methods to learn a policy π that maximizes the reward. We apply Proximal Policy Optimization (PPO) [SWD$^+$17], an on-policy, model-free algorithm that has proven its applicability in this domain [LCP$^+$21] and has become the current de facto standard.

We compare two different approaches utilizing a policy in Cartesian space π_c and one in joint space π_j, as illustrated in Figure 6.1. We hypothesize that mapping states to actions is more straightforward for π_c than for π_j. Both policies use a quintic-spline-based walk engine to generate a reference action at each time step, based on the commanded velocity v_{cmd} and a phase ϕ that synchronizes it with the policy. The reference action is represented by positions p^{ref} and orientations o^{ref} of the feet in Cartesian space. These are not part of the control loop, as in other recent works [LCP$^+$21, DDG$^+$21], but only part of the reward function during training. To improve comparability, the Cartesian actions of π_c are transformed to joint space and applied to the same PD controllers as the actions of π_j, rather than directly applying Cartesian control. The details of this approach are described in the following sections.

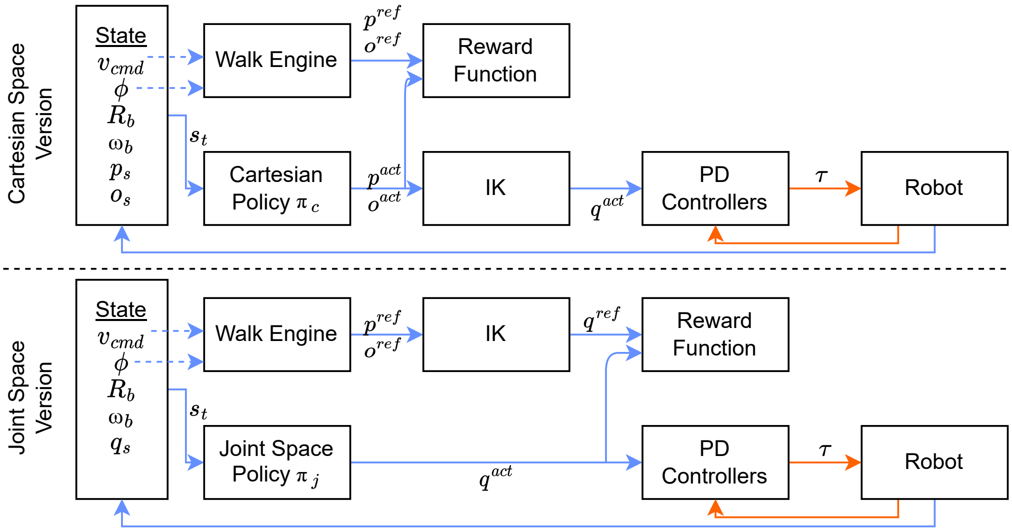


Figure 6.1: Overview of the approach and comparison between Cartesian policy π_c (**top**) and joint space policy π_j (**bottom**). They run at 30 Hz (**blue**), while the PD controllers run at the same frequency as the simulation (**orange**), i.e., 240Hz in PyBullet. The walk engine receives as input the commanded velocity and the phase (**dashed**), which are also part of the policy's state. Both share the same walk engine to generate reference actions. [BZ23]

6.2.1 Reference Motion Generation

The reference actions are created online at each training step by calling the walk engine with the currently commanded velocity v_{cmd} and phase ϕ, which represents the progress of one walk cycle. This engine is the one described in Section 5.3. Only the spline generation and interpolation are used to create the movements of the robot in Cartesian space. None of the stabilization approaches is used since this may bias the learning of the stabilization strategy. We only want to guide the RL process towards a good walk pattern but force it to explore the stabilization since we don't want to exactly reproduce the spline-based approach. The used parameters are the result of the optimization process as described in Section 5.3.5.

In comparison to other approaches using reference motions (see Section 6.1.1), this has multiple advantages. The most important one is that the reference motion is already optimized for the robot. In other cases, e.g., when using human motion capture data, it is not optimal and could lead the reinforcement learning to a suboptimal solution. Since the reference motion is parameterized, certain characteristics can be freely chosen by the user through the fixation of some parameters during optimization. This allows, for example, to create a walking with feet close to each other for going over a narrow bridge. Common problems when learning walking in simulation, e.g., lifting the foot not high enough from the ground, can be avoided by limiting the search space of the parameters. A comparison between the different approaches for reference motions is given in Table 6.1.

6.2.2 State

The state only includes information that is available on a real robot. Two entries, the commanded walk velocity $v_{cmd} \in \mathbb{R}^3$ and the current phase ϕ, are shared with the

Table 6.1: Comparison of Sources for Reference Motions

Motion Capture	Keyframe Animation	Spline-based Engine
Only possible for human or animal-like robots	Possible for any kind of robot	Possible for any kind of robot
Transfer to different kinematics needed	Designed for one specific platform	Adapts to different platforms
Reference not executable	Reference may be executable	Reference executable, provides baseline
Not optimal	Difficult to optimize	Can be optimized for the platform
One walk velocity	One walk velocity	Any walk velocity
Represents state	Represents state	Represents Action
Data recording needed	Manual creation needed	Programming needed

walk engine to synchronize the reference action. In previous work, the phase was either represented by a single value ($\mathbb{R}$) [PALvdP18], which resets to 0 at the end of a walk cycle, or by the sine and cosine value of this ($\mathbb{R}^2$) [YYH+20]. The latter approach tries to achieve a continuous phase input to the network and thus prevents the introduction of non-smoothness to the network's learning behavior. To the best of our knowledge, no direct comparison of the two approaches has been made yet. Therefore we evaluated both methods (see Section 6.3.5).

Additionally, the state encompasses the orientation of the robot's base R_b in the world frame without the yaw component, as this can not be measured reliably by an IMU that complies with the HSL rules. This value is not directly taken from the simulation, but computed by a complementary filter using simulated accelerometer and gyroscope values. Since we want to evaluate different kinds of orientation representations, the dimensionality of R_b depends on which of them is chosen. Generally, quaternions ($\mathbb{R}^4$) are chosen in robotics, but it has been observed that artificial neural networks have difficulties processing them [ZBL+19]. Therefore, we also evaluated Euler angles ($\mathbb{R}^2$ since only roll and pitch are used), fused angles [AB15] ($\mathbb{R}^2$ since only roll and pitch are used), and 6D [ZBL+19] ($\mathbb{R}^6$). Fused angles were chosen as they were specifically designed for keeping balance with bipedal robots, and 6D was chosen since it was designed for usage with artificial neural networks (see also Section 6.1.2). Furthermore, the state includes the angular velocity of the robot's base $\omega_b \in \mathbb{R}^3$.

If the Cartesian policy is used, the state includes the Cartesian positions of both feet $p_s \in \mathbb{R}^6$ and their orientations $o_s \in \mathbb{R}^{\{4,8,12\}}$, which are again represented by one of the earlier mentioned methods, relative to the robot's base. Otherwise, the current leg joint positions $q_s \in \mathbb{R}^{12}$ are added for joint space policies.

6.2.3 Action

For the Cartesian policy, the action is defined as the absolute goal position of both feet $p_a \in \mathbb{R}^6$ and the absolute orientation $o_a \in \mathbb{R}^{\{6,8,12\}}$ (depending on the choice of representation, see Section 6.2.2) in relation to the robot's torso. For each foot, the actions are limited to a convex Cartesian space that is defined in reference to the robot's base (see Table 8.3 for exact values). These limits are defined by sampling poses in this space and ensure that at least two-thirds are solvable by the IK. Still, not all poses in this space can have a valid IK solution due to the low number of degrees of freedom in the robot's leg. Therefore, invalid actions can occur and need to be handled. This can be done either by using a negative reward for such actions, terminating the episode early, or

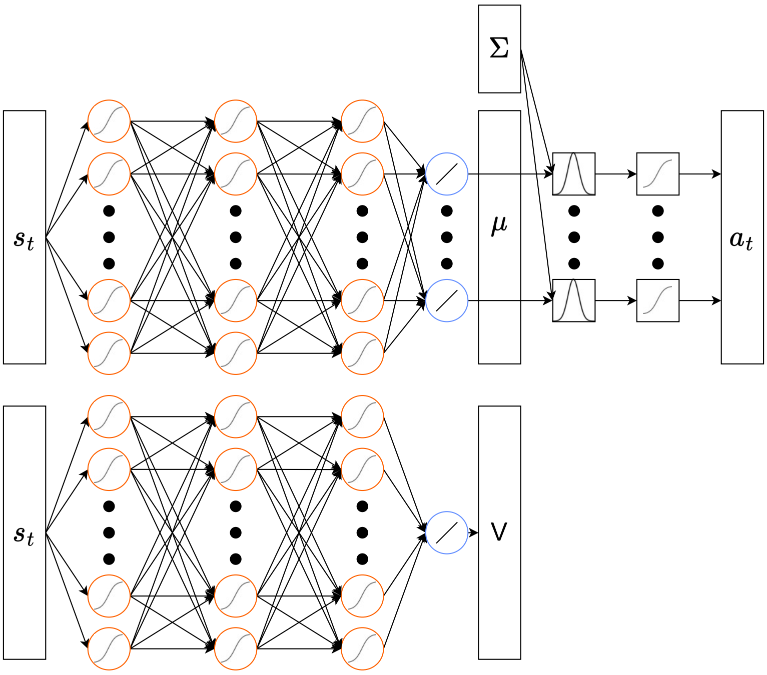


Figure 6.2: Structure of the used neural network. The policy and value functions are two separate networks with similar structures but no shared weights. The first layers use a tanh activation function (**orange**), and the final layer a linear activation (**blue**). The output of the last layer is used as a mean for a multi-dimensional Gaussian distribution with a fixed Σ that is treated as a hyperparameter. The distribution output is squashed using a tanh function to ensure action bounds. [BZ23]

by using an approximate IK solution. Due to early experiences during the development of the approach, we chose the approximation approach. Using an early termination is problematic as the choice of the action in training is partially random due to the used stochastic distribution. Therefore, it will too often produce non-solvable poses at the beginning of the training. In the end, it does not really matter if an approximation has been used, as only the result counts.

When using the joint space policy, the action is defined as joint goal positions for the leg joints $q^{act} \in \mathbb{R}^{12}$. For each joint, the action range is bound to the joint limits. Therefore, all actions are possible, and no further measures need to be taken to ensure this.

In both spaces, the policy output is normalized between $[-1, 1]$ and then scaled accordingly. This is important as we use Gaussian distributions with the same fixed variance for explorations. If the action magnitudes would differ, the exploration would not work well.

6.2.4 Neural Network Architecture

The policy π consists of an artificial neural network that takes a state s_t as input and provides an action a_t as output. The exact network structure is considered a hyperparameter. All possible architectures consist of two or three fully connected layers with 64, 128, 258, 512, or 1024 neurons each, with *tanh* activation function and then a linear layer (see Figure 6.2). The action distribution is modeled as Gaussian with a mean μ that is given by the network's output and a fixed diagonal covariance matrix Σ,

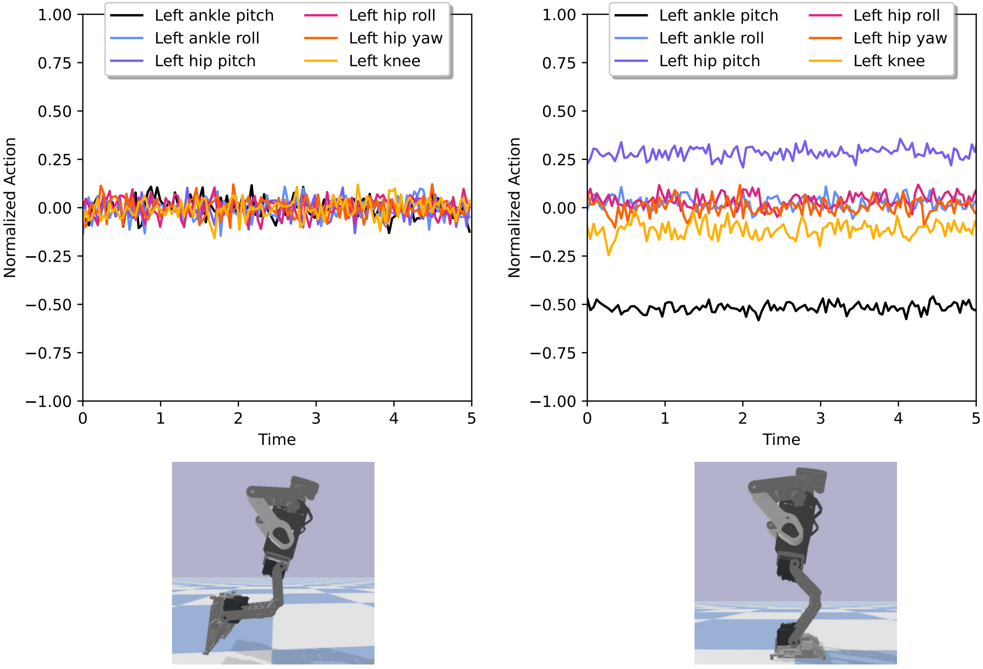


Figure 6.3: Normalized actions of an untrained policy without (**left**) and with (**right**) initial network bias. The difference in the initial exploration is clearly visible. Below the corresponding pose of the robot is displayed. [BZ23]

similar to [PALvdP18]. The variance is fixed to prevent an early decrease in exploration. The output of the Gaussian sampling is squashed with a tanh function to ensure action bounds. Otherwise, boundary effects can degrade the learning performance [CMS17]. Using a tanh function was also later advised by Andrychowicz et al. [ARS+20]. An alternative would be the usage of a different distribution type, i.e., a beta distribution [CMS17], but in our preliminary experiments, this did not perform as well. The reason is not clear, but it might be due to having two parameters that need to be specified, and none of them directly represents the deterministic action. The value network is separated but has the same size of fully connected layers as the policy.

We observed that the network initialization influences the initial starting point for exploration significantly. A Cartesian action space has its natural center with the feet centered below the upper body and the soles parallel to it. It does therefore provide a comparably stable starting pose for exploration. A joint-based action space is more dependent on the exact kinematic structure of the robot. Still, one common feature in most bipeds is a knee that only bends in one direction. Therefore, the zero point of the knee angle is typically at around 90° when this action space is normalized between the joint limits. This leads to an initial exploration around an unstable pose. Furthermore, this pose typically differs significantly from the robot's pose at the start of the episode. Thereby, the joint goal positions of the first action differ from the current positions, leading to the PD controllers commanding high torques and thus to the robot falling quickly.

We tackled this problem by initializing the bias of the neurons that represent the μ value in the network (the last linear layer) so that the initial exploration is done around the pose of the reference motion at $\phi = 0$ and $v_{cmd} = (0, 0, 0)$ (see Figure 6.3 and video at [1]). It would also be possible to achieve a similar bias by adding it to the output

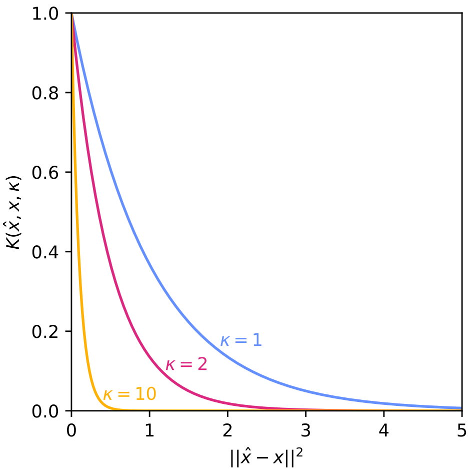


Figure 6.4: Exemplary plot of the radial basis function kernel for different values of κ.

of the policy. However, this does not allow the network to change this value during training. An evaluation of this approach is provided in Section 6.3.2.

6.2.5 Reward

We use a straightforward definition for the reward without additionally shaping it. It consists of two equally weighted parts, the actual goal r_g and an imitation reward for the reference motion r_i.

$$r = \frac{1}{2} \cdot r_g + \frac{1}{2} \cdot r_i \tag{6.1}$$

A radial basis function kernel is used to bound each partial reward to $r \in [0, 1]$, where $\hat{x}$ is the vector of desired values, x the vector of actual values, and κ a scaling hyperparameter.

$$K(\hat{x}, x, \kappa) = e^{-\kappa(\|\hat{x}-x\|)^2} \tag{6.2}$$

The influence of the κ parameter can be seen in Figure 6.4. It has been chosen manually for the following functions (which have input vectors of different dimensions) to achieve a similar scaling. Handling the κ as a hyperparameter and optimizing it is not possible as it would naturally choose small values, as they always increase the reward.

The goal reward r_g is computed by how accurately the robot achieves the given command velocity v_{cmd} in x, y, and yaw direction.

$$r_g = K(v_t^{cmd}, v_t^{state}, 5) \tag{6.3}$$

The imitation reward can be defined in two different ways. In previous works, the state of a robot after executing an action was compared to a reference motion [LCP+21, PALvdP18]. This makes sense if the reference is a trajectory of recorded states. In our case, the reference is provided by a walk engine and can therefore be interpreted as a reference action. Even if the policy chooses at time point t the same action a_t^{ref}, this does not necessarily reflect in the state s_{t+1} since this depends on further factors such as the previous foot positions p_t and orientations o_t. Furthermore, the correct state s_{t+1} could have also been reached due to an approximated IK solution of an action a_t that can not exactly be solved. We define both an action-based imitation reward r_{ia} and a state-based imitation reward r_{is} for the Cartesian and joint space to evaluate if this design choice influences the learning performance (see Section 6.3.4). In the following, the double plus operator ($+\!\!+$) denotes a concatenation of vectors. An example is given below in Equation 6.4.

$$(1, 2) +\!\!+ (3, 4) = (1, 2, 3, 4) \tag{6.4}$$

$$r_{ia}^{cartesian} = K(p_t^{ref} +\!\!+ o_t^{ref}, p_t^{act} +\!\!+ o_t^{act}, 1) \tag{6.5}$$

$$r_{ia}^{joint} = K(q_t^{ref}, q_t^{act}, 2) \tag{6.6}$$

$$r_{is}^{cartesian} = K(p_t^{ref} +\!\!+ o_t^{ref}, p_{t+1}^{state} +\!\!+ o_{t+1}^{state}, 50) \tag{6.7}$$

$$r_{is}^{joint} = K(q_t^{ref}, q_{t+1}^{state}, 2) \tag{6.8}$$

Note that we used the geodesic distance when we compared two orientations since directly taking the difference between values of two quaternions does not reflect the actual distance between them.

6.2.6 Environment

Training is done in simulation using the earlier described model of the Wolfgang-OP (see Section 3.5). Each training episode is started using reference state initialization (RSI) [PALvdP18] by choosing randomly $\phi \in [0, 1]$, $v_{cmd} \in ([-0.6, 0.6]\frac{\mathrm{m}}{\mathrm{s}}, [-0.25, 0.25]\frac{\mathrm{m}}{\mathrm{s}}, [-2, 2]\frac{\mathrm{rad}}{\mathrm{s}})$ (different for other robots see Table 8.3) and resetting the robot's pose to the corresponding reference action. The randomly chosen v_{cmd} is tested (without running the simulation) for one complete gait cycle to check if the robot is able to kinematically solve the reference motion completely. If this is not the case, another v_{cmd} is chosen randomly until one is found that is solvable.

Additionally, different methods of domain randomization can be applied. The start pose can be randomized by applying a random offset of up to 0.1 rad to each joint to improve generalization [RTvdP20]. The model can be modified by varying all masses, inertias, maximal motor torques, maximal motor velocities, restitution, as well as lateral, rolling, and spinning friction coefficients. A terrain can be created using randomized squares of $0.01\mathrm{m}^2$ in a height range of $[0, 0.05]\mathrm{m}$. To simulate external forces and to increase difficulty, a random force of up to 10 N and a torque up to 20 Nm in each direction can be applied to the robot's base at each time step of the environment.

During training, the policy runs at 30 Hz while the simulation and PD controllers run at 240 Hz. The episode is either terminated early when the robot is fallen down (measured by its orientation) or after 10 s. We use bootstrapping to handle the time limit issue [PTLK18].

6.2.7 Implementation

We implemented our approach following the standard OpenAI Gym interface [BCP+16], which defines that training is done in an environment with step and reset functionalities. Our environment is kept modular to allow changes and comparisons, e.g., by using two different reward functions. An overview of it can be seen in Figure 6.5. Two objects of the robot class are used. One for the agent robot and one for the reference motion model without any physics or collisions. This has the advantage that the reference motion can be displayed alongside the agent. For additional debug purposes, a ROS-based debug interface was created. This allows further insight into the system through standard ROS tools like PlotJuggler.

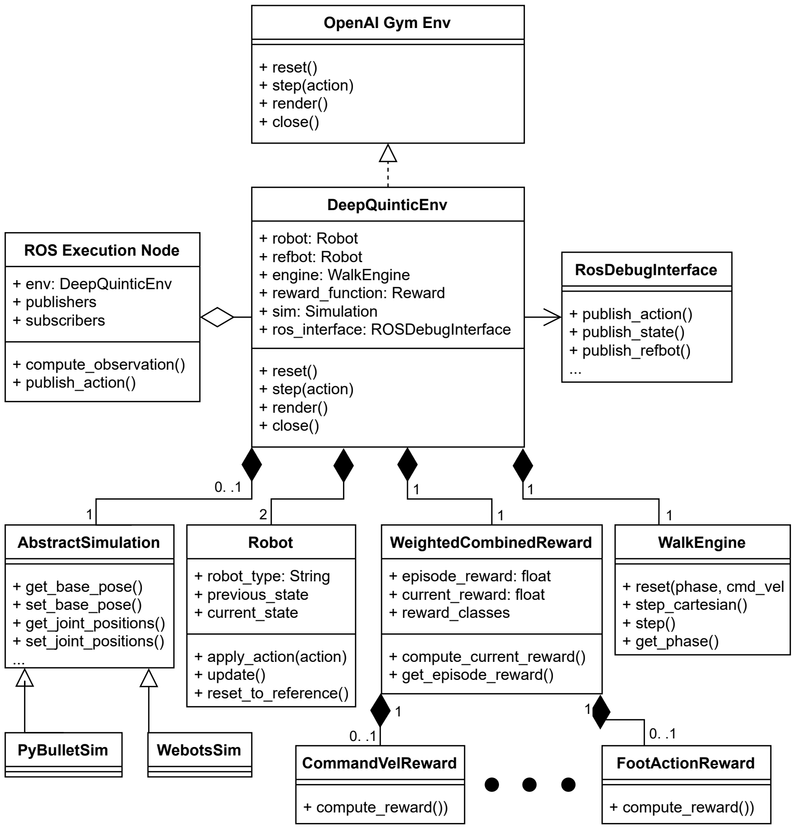


Figure 6.5: Simplified overview of the environment implementation as a class diagram. The main component is the RL environment (DeepQuinticEnv). This implements the Gym interface and is used during training, but it can also be used to run a trained model as a ROS node. In this case, the internal simulation and reference motion are deactivated.

Different simulators can be used. For this, a common interface is defined that allows a simple exchange of the simulation. We implemented a PyBullet and Webots version based on the model described in Section 3.5. The Webots version is more accurate since it includes a simulation of the joint backlash and the torsion springs in the knee joints. However, it needs more computational resources of the training computer and therefore allows less parallel training. Furthermore, it exhibited simulation glitches which happened especially during hyperparameter optimization. Therefore, most of the experiments were conducted in PyBullet.

The walk engine to generate the reference motion pattern uses the OptiQuint-based approach described in Section 5.3. It uses the direct Python interface instead of using any ROS (see Section 5.2.4).

The PPO implementation of Stable Baselines3 [30] was used, and training was done with RL Baselines3 Zoo [29]. Pueue [14] was used to start the experiments and Weights & Biases [27] was used to track the experiments during training. Optuna's implementation of TPE was used for optimizing the hyperparameter of PPO.

The trained agent can be executed in ROS with a custom node that internally holds the environment without simulation. It subscribes to multiple topics and, thereby, aggregates the data for the policies state. The chosen action is then published again to the ROS system.

6.3 Evaluation

We evaluated different design choices for the RL environment to show which of them influences the achieved performance of the trained policy. Most of the experiments were performed using the PyBullet simulation as it took fewer resources (see Section 6.2.7). We compared the results for the different configurations with each other but not directly to the results of researchers for two reasons. First, since we used a custom environment with our own robot and different action definitions, there was no comparable research for most experiments. Second, the influence of the used PPO implementation is high [EIS$^+$20] and, therefore, it can not be ensured that differences in the reward are not due to these differences.

As described above, we handled the network structure and the variance of the Gaussian distribution as hyperparameters. Additionally, PPO itself has multiple hyperparameters. We optimized these using TPE with 100 trials (the first ten are chosen randomly). Each trial was trained for 5M steps and evaluated each 0.5M steps for 100 episodes. The high number of episodes is necessary to reduce the noise that comes from choosing the goal velocity randomly. During this evaluation, the actions were chosen deterministically to avoid biases from different variance parameters. Additionally, median pruning was applied to remove not promising trials earlier. In the following experiments, the hyperparameters were optimized separately for different experiment configurations to ensure that the observed difference between configurations does not stem from the hyperparameters being more fitting for one of the cases. First, the hyperparameters were optimized for the default Cartesian Euler angles policy (see Section 6.3.1). The hyperparameters for the other policies were optimized in the same procedure, but the best parameter set of the Euler policy was given as an initial guess. This should prevent the case where the optimization of another policy does not find a similar good parameter set and therefore performs worse than the Euler in the following experiment. As a downside, this could mean that the other policies further improve on this parameter set and are thereby at an advantage against the Euler policy. Therefore, the same procedure of optimizing the hyperparameters with the best set as initial guess was also performed for the Euler policy but did not further improve on this parameter set. See Table 8.2 for a complete list of the optimized hyperparameters. The most interesting finding in this is the network structure. For most cases, it is notably smaller than previous works [PALvdP18, RB21].

If not otherwise stated, each policy was trained for 25M steps using PPO. During the training, they were evaluated each 0.5M steps 100 times using deterministic actions (same as during the optimization). Note that a reward (for both imitation and goal) of 300 was the maximum that could be achieved since the episode time was limited to 10 s and the policy was executed with 30 Hz. The computation for this (using PyBullet simulation) takes around 24 hours on a computer with AMD Ryzen 9 3900X CPU and Nvidia Titan X GPU. Depending on the environment configuration, the policy typically already converges at around 5M steps and therefore can already be used after around 5 hours of training.

The following subsections discuss our further experiments in detail.

6.3.1 Choice of Action and State Space

One of our hypotheses is that a policy in Cartesian space can learn the task easier as it does not need to learn the kinematics of the robot. To investigate if this is true, we compared the joint space and Cartesian policies, as well as the different representations of orientation (Euler angles, fused angles, quaternion, 6D). Each space representation was used for both state and action. Although it would also be possible to represent the state in one space and the action in another, this was not evaluated as we expect it to perform inferiorly due to the additional space transfer in the policy.

We trained each configuration five times with different random seeds. The results are displayed in Figure 6.6. While the goal reward is of most interest, the other rewards and the episode length are also shown. The fused angle policy performs best for overall reward and goal reward. The Euler angle and joint policies achieve equal but slightly lower rewards than the fused angle policy. The 6D policy receives an again slightly lower reward. Finally, the quaternion-based policy achieves the lowest reward with a large difference. All policies achieve similar episode lengths, although the quaternion and 6D policies need more training steps to achieve the same level. Still, all policies were able to produce a working omnidirectional walk.

To further eliminate possible influence on the results due to better hyperparameters, the same training was executed with the hyperparameters of the fused angle policy as it performed best in the previous experiment. We will denote these in the following as FA hyperparameter. In this experiment, each policy was trained with only three different seeds since the difference between training runs was small in the previous experiment. The results are displayed in Figure 6.7, and an exemplary video is available at [1]. It can be seen that with the same parameters, the Euler angle policy performs identically to the fused angle-based one. This indicates that the difference in the first experiment was only due to the difference in hyperparameters. The fused angles have, to the best of our knowledge, not yet been used with RL. Although they were specially designed for the problem of bipedal walking, they seem to provide no advantage over simple Euler angles in RL. Both Euler and fused angles policies perform better than the joint policy. The 6D and quaternion-based policies take longer to converge but achieve a higher reward than with the previous parameters. Still, both can not outperform the Euler angles policy.

As mentioned before, quaternions are not well suited for neural networks, and therefore, a comparably bad performance was expected. This also aligns with previous results [ZBL+19]. The 6D representation, on the other hand, was specially designed to be easy to learn for neural networks. Still, it performed worse than the simple Euler angles. We identified two reasons for this. First, the 6D representation needs six action values for each rotation, while Euler only needs three, thus resulting in a larger action dimension. Second, the action bounds for the Euler and fused angles were set so that the robot's foot can only be rotated by 30°, as higher rotations are not necessary during walking. This was not done with the 6D representation, as it can not be represented easily by limiting each of the numbers. The quaternions have the same disadvantage. This is also visible in the IK error (see Figure 6.8). Euler and fused angle policies already have very small errors in the first evaluation. Quaternion and 6D policies need to first learn which rotations are possible and never achieve such a low error. Interestingly, both policies still achieve a working walk by exploiting the approximation of the IK.

This result may not apply to other motion skills. In bipedal locomotion, all rotations (R_b, o_s, and o_a) are very limited walking. The feet only rotate slightly in roll and pitch for stabilization and in yaw for walking in curves. If the torso is rotated too much, it will fall, and in the training, the episode is terminated early. Therefore, the typical issues of Euler angles, i.e., gimbal locks and non-linearities due to sequential rotations, are not encountered. This would be different for a stand-up skill. Here, the body is typically rotated by 90° at the start, and the feet may also rotate more in the pitch axis. Therefore, the Euler angles might perform worse in this case. Still, other skills that have no strong rotations, e.g., a kick (see Section 5.5), should perform similarly.

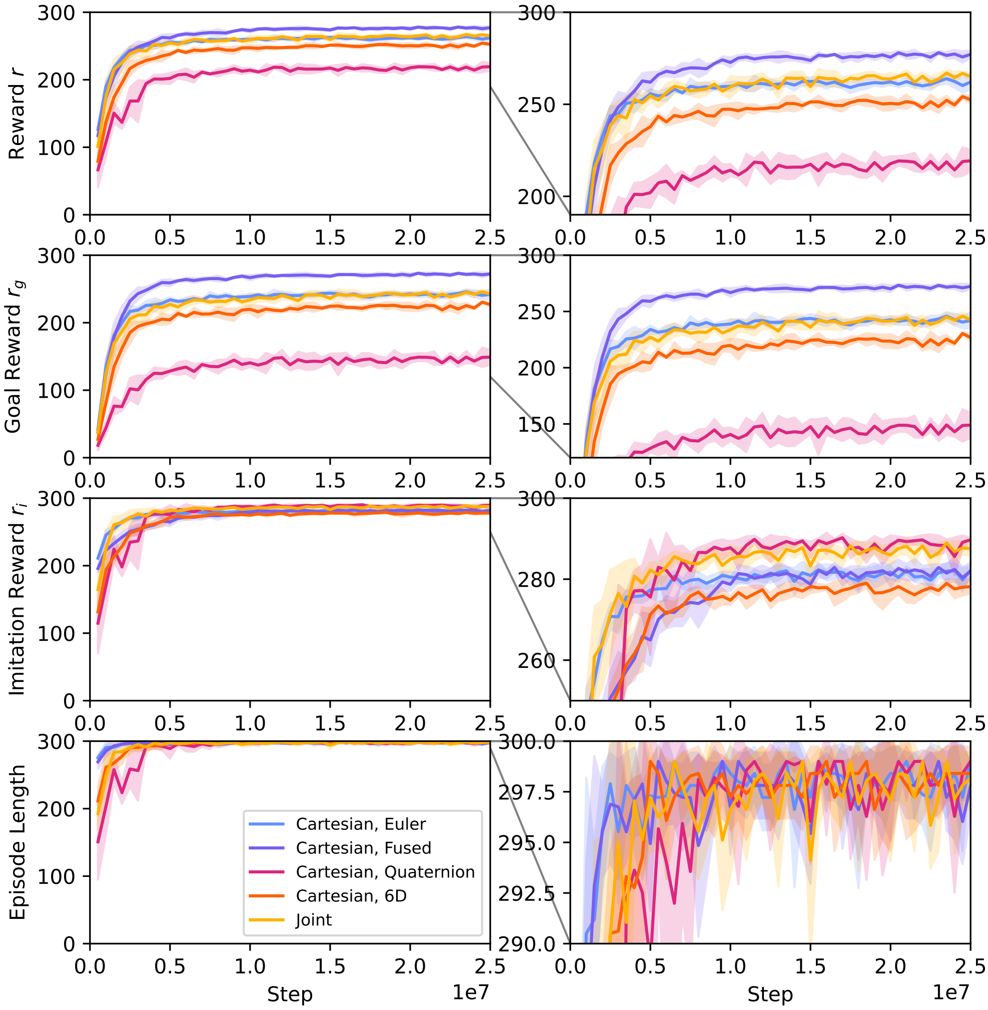


Figure 6.6: Comparison of the different action spaces. For each configuration, the results of the five trainings with standard deviation are shown. The right column provides a magnified plot to improve visibility. The fused angles policy clearly outperforms the other in terms of goal and overall reward. No difference in the finally achieved episode length can be observed. Based on [BZ23].

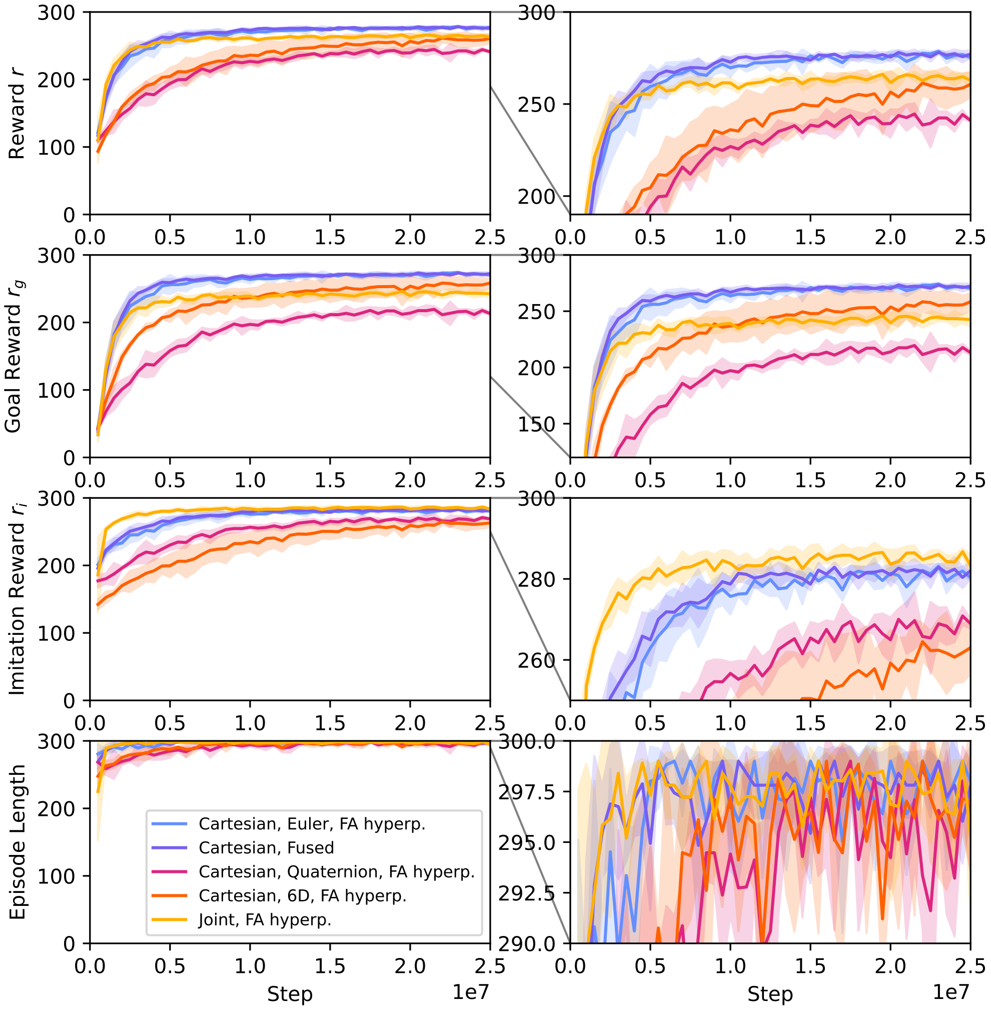


Figure 6.7: Comparison of different action spaces using the same hyperparameter. The Euler and fused angle policies perform equally well and outperform all others in goal and overall reward. Based on [BZ23].

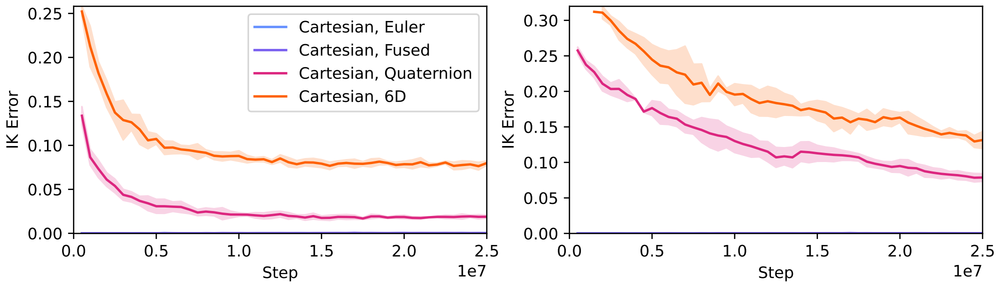


Figure 6.8: IK Error of different policies with their individual hyperparameters (**left**) and the fused angles (FA) hyperparameters (**right**). Note that the error of the Euler and fused angles policies is so small that it is not displayed well. [BZ23]

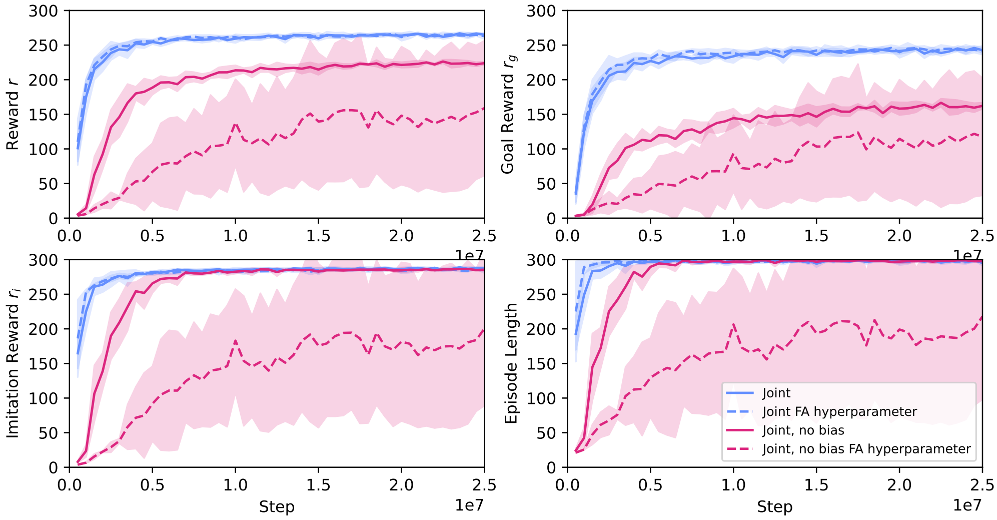


Figure 6.9: Comparison between using the initial network and not using it for the individually optimized and the FA hyperparameters. Based on [BZ23].

6.3.2 Network Initialization

In the previous experiments, all policy networks were initialized as described in Section 6.2.4 since this provides a more comparable start initialization. We perform an additional experiment to ensure that it does not reduce the achieved performance. Furthermore, we test our hypotheses that the network initialization procedure accelerates the learning of the policy. We optimized the hyperparameters for the policy without initial bias and trained it. To ensure that the difference between both is not only due to different hyperparameter, both policies were also trained using the FA hyperparameters. The Cartesian policies were not evaluated without an initial bias as their zero initialization is already a stable pose close to the one that is created by the network initialization (see Section 6.2.4). Therefore, the difference is small. The results are displayed in Figure 6.9, and an exemplary video is available at [1].

The policy without initial bias starts on a lower reward and episode length. This was expected as this policy first needs to learn to stand. It still manages to achieve the same imitation reward, although later. Still, it does not converge to the same goal reward as the other policy. The discrepancy in goal reward after training for 25M steps is higher than most of the other Cartesian policies in the last experiment.

The performance difference is even larger if the FA hyperparameters are used. In this case, the policy with network initialization performs exactly the same as with the parameters that were optimized for this policy. In contrast to this, the policy without initial initialization performs worse and has a large discrepancy between trainings with different random seeds. This can be an indicator for using not fitting hyperparameters.

These results show two interesting points. First, the discrepancy in achieved rewards between the initialized and not initialized joint policy is higher than the one between the initialized joint policy and most Cartesian policies. Second, the FA hyperparameters work well for the initialized joint policy but not for the not initialized joint policy. These two points might indicate that the difference in used space (between Cartesian and joint space) is smaller than the difference due to the different initializations of the network. Previous works, e.g., [BB19], have not taken the initialization into account, and, therefore, their result may be influenced by the different initializations.

We performed another experiment to ensure that this result is not limited to the used robot model or the fact that a reference trajectory is used. We chose the *Walker2D-BulletEnv-v0* gym environment since it also has the goal of learning bipedal walking but with a much simpler robot that only learns to walk forward and can not fall in the lateral direction. It normally uses torque control. Therefore, we created a modified version that uses position control. To ensure that both environments are comparable, the maximal torques of the PD controllers were given the same limits as the torque control uses.

The environment uses no reference motion, which can be used to define the initial bias pose. Therefore, we tested two poses which both let the robot stand stably. The first keeps the legs exactly straight, which is also the starting pose of the agent. The second bends the knees by 0.6rad and the hip/ankle joints by 0.4rad, which resembles roughly the start pose that we used in the experiments with the Wolfgang-OP. The pose was not optimized in any way, instead, the first stable pose with bent knees was taken.

Since the original environment uses torque control, we also investigated if the usage of an initial bias improves the training. However, we can not use the same technique of using a stable standing pose since we can not define a pose by torque values. Instead, we chose to initialize the actions to the mean torque values that a trained policy produces.

We compared the achieved reward of these different configurations. Each configuration was trained three times for only 5 M steps since the simplicity of this environment enabled quicker training. Similarly to the previous experiments, we optimized the hyperparameters individually but also trained all environment configurations with the best-performing hyperparameter set. We did not find better hyperparameters for the straight leg configuration than one that we used for the bent configuration. Therefore, the straight leg configuration is only shown once in the results using these parameters. For the torque-based environment, we used the hyperparameters provided by the rl-baselines3-zoo [29].

The results (see Figure 6.10 and exemplary video [1]) show a large difference between the different configurations. This difference is especially visible between the differently initialized position-based control configurations. The one using a pose with bent knees outperforms all others. Using a straight leg pose leads to lower rewards but it is still better than not using any initial bias. This is interesting since it indicates that the initial action should be close to a high reward achieving motion rather than being close to the initial pose at the episode start. Using the original torque-based control almost reaches the same reward as the uninitialized position control configuration. Using an initialization for the torque-based policy does not seem to lead to any improvements.

The experiments indicate that our approach of setting an initial bias to the network is advantageous, as it converges to a higher reward. Additionally, the number of falls during the training is reduced drastically. This is especially interesting if the training should be performed on a real robot. The joint space policy with initial bias performed similarly well as the best Cartesian policies. Therefore, the performance difference between Cartesian and joint space seems to arise mostly from the different initial poses and not from the different complexity of the space representation.

The presented approach provides an alternative to the utilization of pre-trained weights, which was shown to be inefficient for reinforcement learning [RTvdP20]. Andrychowicz et al. [ARS^{+}20] also stated that the initial policy has a high impact. They observed that a small standard deviation and a mean of 0 are crucial for achieving high performances in multiple gym environments, including a humanoid. It is important to consider that these environments use direct torque control and not position control like the DeepQuintic one. Therefore, the mean of the action is not directly related to its position. Our results seem to indicate that this rule does not generalize to environments with position control. Additionally, a change of the policy network to ensure a small standard deviation is not necessary in our case, as we treat the standard deviation of

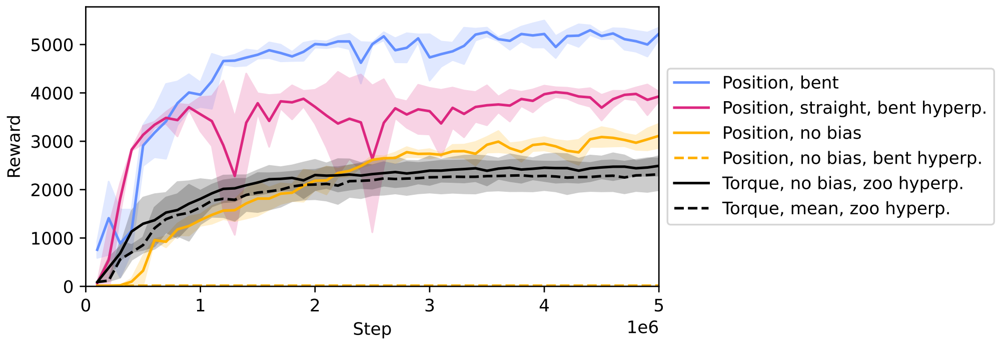


Figure 6.10: Training of the Walker2DPybullet gym environment for different control types and initial biases. The position-based policy which was initialized with bent knees, performs best. Using a straight leg initialization still performs inferior to it but still gives a benefit in comparison to no initialization. The not initialized position configuration did not work, resulting in near-zero rewards. [BZ23]

the policy as a hyperparameter and do not set it via the policy output. Other works, e.g., [RB21], may have circumvented this issue by using relative position goals instead of absolute ones. Thereby, an initial position action of zero is more similar to a zero torque action.

One result of Reda et al. [RTvdP20] was that using PD controller goal positions instead of directly using torque has a limited advantage. Our experiments contradict their results since we could observe a large difference between both types of policies. We hypothesize that they did not take the initial action of their position control network into consideration. In fact, our result of the unbiased policy also results in a comparably small advantage. However, when the initial action is set to a bent pose, the position-based policy outperforms the torque-based one clearly.

6.3.3 State

We also investigated if adding further data to the state might improve the training. One typical addition is the joint velocity, as this provides the policy information on how fast the robot is moving, which it can not directly know by having only the current positions. Therefore, we trained a joint space based policy with this additional information and compared it to the earlier described joint space policy (see Figure 6.11). Both were trained using the same hyperparameter of the joint space policy for 25M steps each. The additional does not improve the policy (see Figure 6.11). The imitation reward remains similar, but the goal reward decreases slightly.

We also evaluated an additional modality by adding FPSs based data to the state. The simulated sensors provide eight unidirectional forces for the corners of both feet, similar to the sensor of the real robot (see Section 3.4.1). We trained it using the previously mentioned method and compared it to the same policy without FPS data (see Figure 6.12). The additional FPS data seems to provide no advantage for the policy as it leads to lower rewards. It could be possible that the hyperparameters need to be optimized separately for this scenario. However, it is more likely, that the policy does not require any FPS data on flat terrain. This also aligns with previous findings from [KH20, RTvdP20]. Another test with terrain is done in Section 6.3.6.

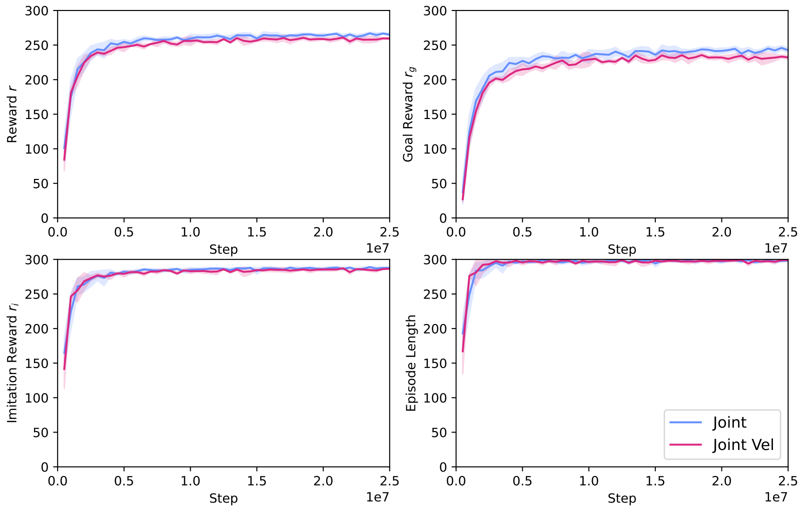


Figure 6.11: Investigation of using the joint velocity in the state. The additional data does not lead to a higher reward and, therefore, provides no advantage.

6.3.4 Action and State Based Reward

We presented two different approaches for defining the reward either based on the action or the state of the robot (see Section 6.2.5). Our hypothesis was that an action-based reward might perform superior since the state-based reward might be influenced by the previous state of the robot as well as approximate solutions of the IK. To evaluate this, we trained the Cartesian policy with Euler angles using both rewards. To eliminate any influence from the hyperparameters, we trained with optimized parameters and with the FA hyperparameters. Each combination was trained three times for 25 M steps.

The action-based reward r_a and the state-based reward r_s both manage to train a walking policy in a similar amount of time steps (see Figure 6.13). They converge to the same goal reward if the same set of hyperparameters is used. Therefore, we assume that the slight difference between both approaches when using the individually optimized parameters is only due to differences in these hyperparameters.

The results of this experiment seem to indicate that there is no difference in using a state- or action-based reward as the achieved goal reward is similar. However, the actions that the trained policies choose are different (see figure 6.14). The state reward based policy's actions oscillate more. Our hypothesis is that the inertness of the system allows such oscillations in the actions as the robot's state only changes slowly. Such oscillations are not possible with the action-based reward since the action is directly compared with the reference. These oscillations lead to no significant reduction of the achieved goal reward. Still, they might be of interest for some domains, e.g., when this oscillating behavior works in simulation but does not transfer well to the real robot.

An action-based reward might additionally have an advantage as it is slightly faster to compute. The state-based reward requires querying the state from the simulation, while the action-based one can be solely computed by the output of the network. This difference is very small, though.

6.3.5 Phase Representation

We compared a non-cyclic phase representation using a single scalar value to a cyclic representation with the sine and cosine value of this scalar (as described in Section 6.2.2).

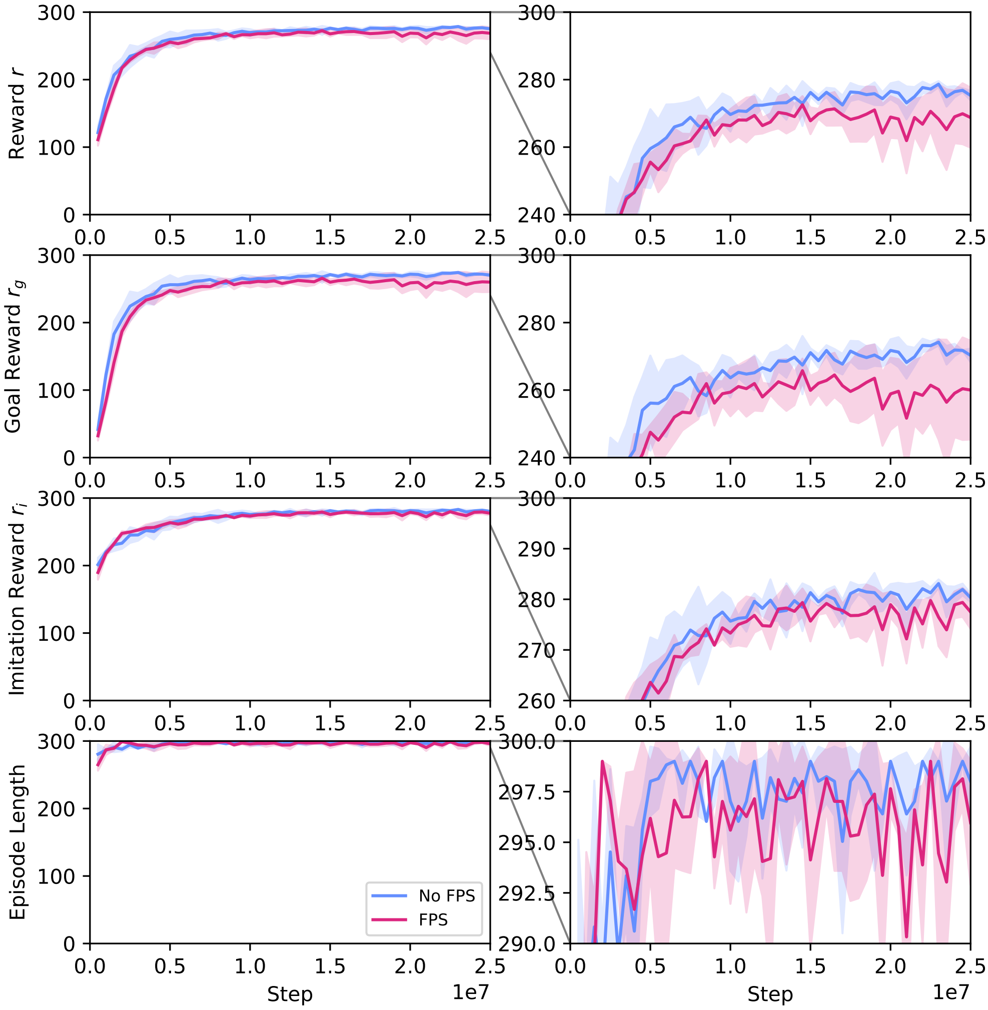


Figure 6.12: Investigation of using the FPS data in the state. Adding this leads to slightly lower rewards and less stable convergence. This indicates that the FPS data provides no advantage.

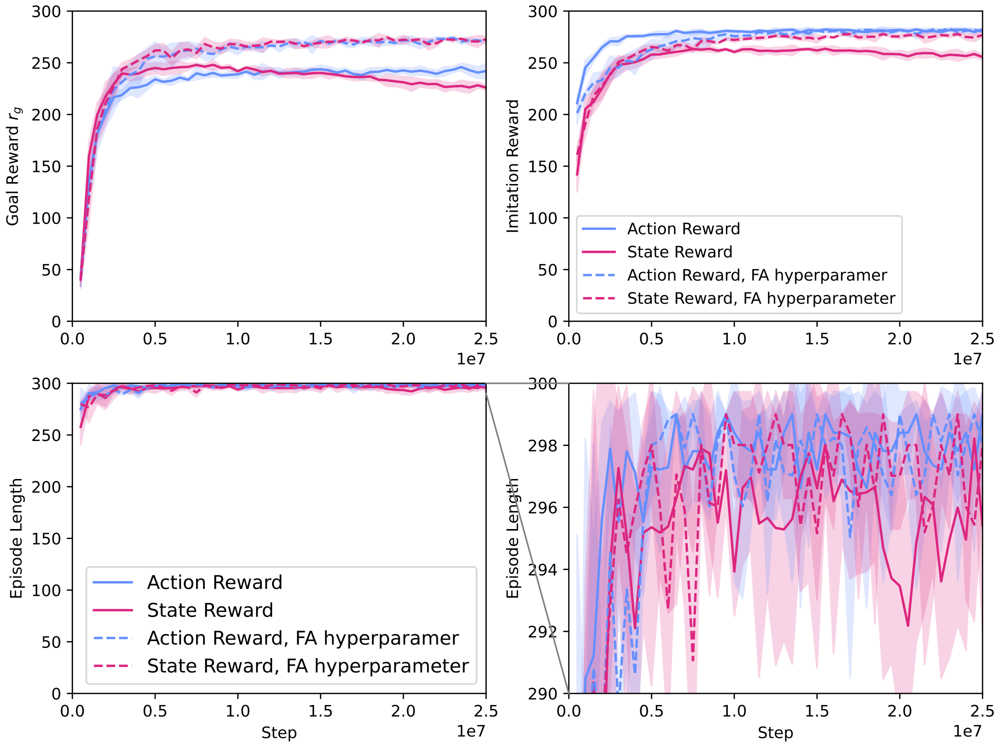


Figure 6.13: Comparison of using a state based reward (**red**) instead of an action based one (**blue**). The dashed lines indicate that the FA hyperparameters have been used.

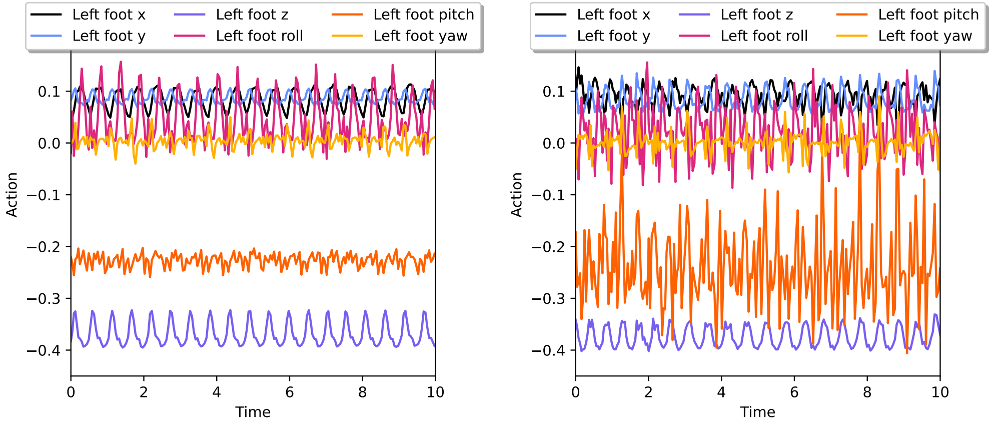


Figure 6.14: Action plot of policies with action (**left**) and state rewards (**right**). For both, only the actions for the left foot are plotted, but the right foot acts similarly. The state reward based policy produces more oscillations.

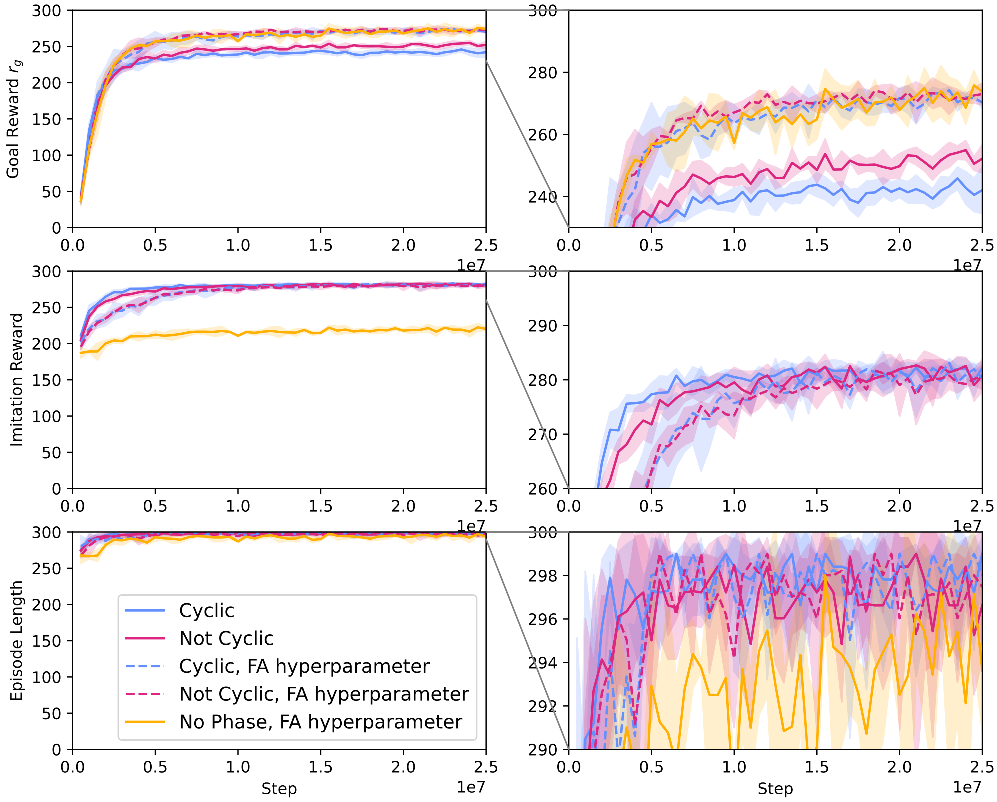


Figure 6.15: Comparison of cyclic (**blue**) and non-cyclic (**red**) phase representations, as well as training the policy without any phase representation (**yellow**). The dashed lines indicate that the FA policy's hyperparameters have been used. There is only a difference in goal reward between the cyclic and non-cyclic policy if not the same hyperparameters are used. The phaseless policy achieves equal goal rewards but is not able to imitate the reference motion correctly.

Additionally, we also trained a policy without any phase representation in the state to demonstrate the necessity of this information. Similarly to the previous experiments, we compared the achieved reward using individually optimized hyperparameters and using the best-performing hyperparameter set from the experiment in Section 6.3.1. All trained policies use the Cartesian Euler space representation.

In our experiment, the cyclic and non-cyclic phase policies achieve a similar reward when the same hyperparameters are used (see Figure 6.15). Using the individually optimized parameters results in worse goal rewards with only a slight difference. Therefore, we assume that this is only due to the difference in hyperparameters. We can not validate the hypotheses that there is a benefit in using a cyclic phase representation as it was stated in [YYH$^+$20].

The policy without phase representation in the state still achieves comparably high rewards. However, it only manages this by exploiting the simulator. It generates actions that let the robot vibrate over the floor instead of performing actual steps (see Figure 6.16). This type of policy would not transfer well into a real-world scenario, and it also does not imitate the reference motion correctly. The result was expected.

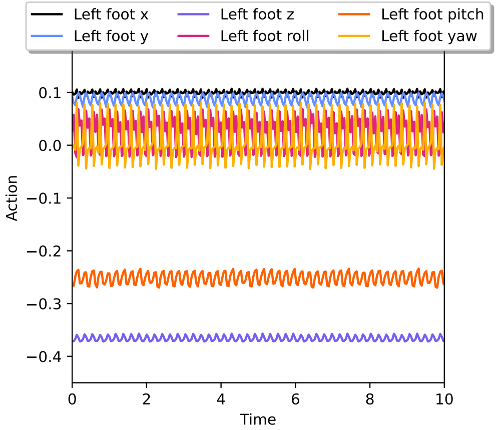


Figure 6.16: A plot of the actions of a trained policy without phase representation in the state. Note the difference to Figure 6.14. Only small vibrating movements are generated. Only the left foot is shown, but the right one is equivalent. See also [1] for a video.

6.3.6 Adaptive Phase

The presented DeepQuintic approach has a fixed gait cycle length. This is a disadvantage compared to the OptiQuint walk skill, which can modify the phase. Theoretically, it is possible for the policy to diverge from the reference motion to, for example, end a step earlier by moving the foot differently. However, the policy will not be able to start a new step and would need to stay in this pose. Otherwise, it would desynchronize with the reference motion, which would lead to a bad imitation reward. Additionally, since the state only includes the current phase and the policy has no information about past actions, it would not know in the next time step that it has ended a step early.

This issue of having a forced cycle length due to the synchronization with the reference motion also exists in other works (see Section 6.1.1) but has not been discussed yet. We propose a novel approach that allows the policy to modify the phase and, therefore, might improve its performance since the phase is not fixed anymore. For this, an additional action is added to the policy, which decides how much the phase of the reference motion is advanced. This allows the policy to learn if it wants to finish a step earlier or prolong it. The bias of this action in the policy's network is initialized with the otherwise used value for achieving 30 Hz.

We tried two versions of this approach. In the first, no reward is given based on the value of the phase action, thus only guiding the learning of this action through the existing goal and imitation reward. In the second, the phase action is included in the imitation reward with the default value of 30 Hz as a reference. All were trained using the FA hyperparameters. The results are shown in Figure 6.17 and an exemplary video is available at [1]. No significant difference between the different configurations can be observed in terms of the achieved reward. Additionally, the policy's outputs were compared to see if the adaptive phase is actually used (see Figure 6.18).

Both adaptive policies use their phase action in a cyclic pattern. Not providing an imitation reward results in a higher variance of this action and the usage of long time steps. When the imitation reward is applied, the action stays closer to the original time step and mostly shortens it.

One reason for the adaptive phase not increasing the performance of the policy is that it provides no advantage if the robot walks on even ground. Therefore, we performed

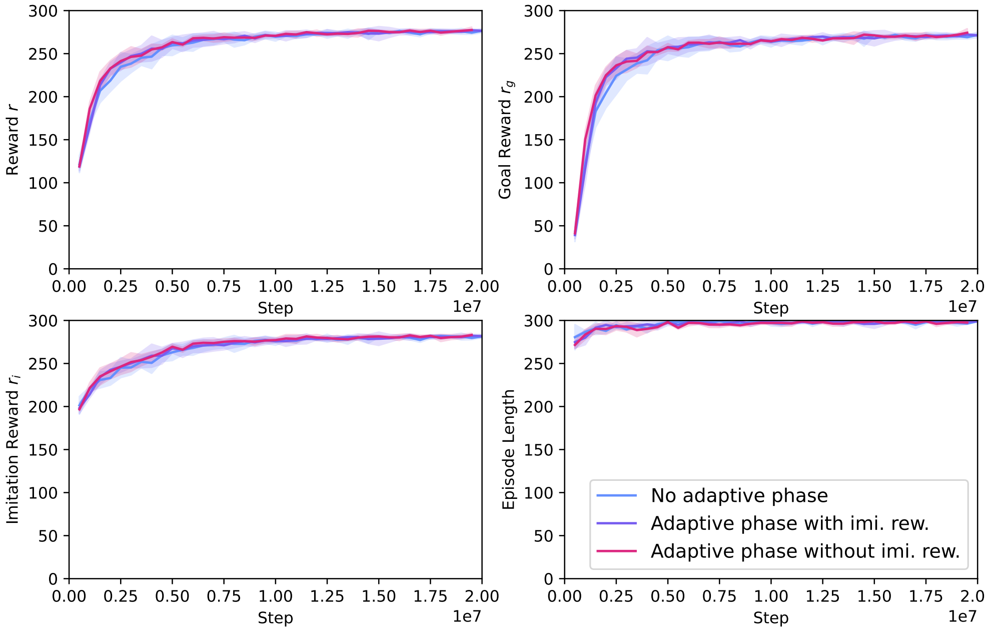


Figure 6.17: Influence of using the adaptive phase with an 30 Hz reward and without. No difference in achieved reward can be observed between the different versions of the adaptive reward, as well as in comparison to a fixed-cycle policy.

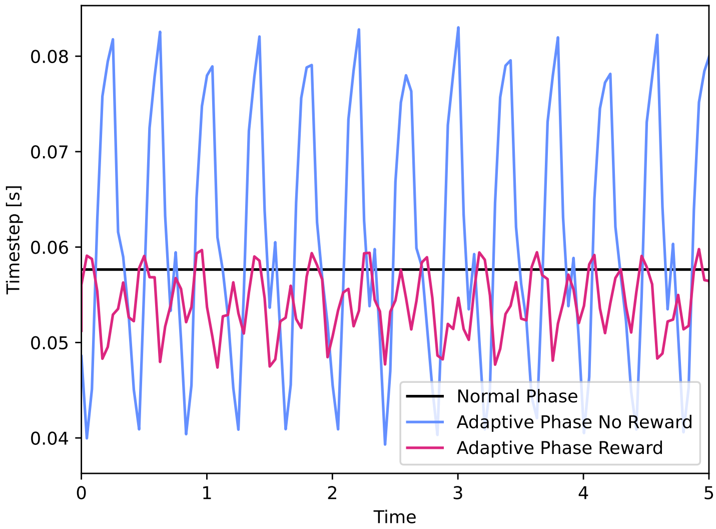


Figure 6.18: Action plot of the two adaptive phase policy versions with the fixed-cycle policy as a baseline. The robot was walking slowly forward during the recording of the action outputs. The action is shown as the corresponding time step length. It can be observed that both adaptive policies use the ability to change the phase step in a cyclic pattern. The imitation reward leads to a smaller variance of the phase action.

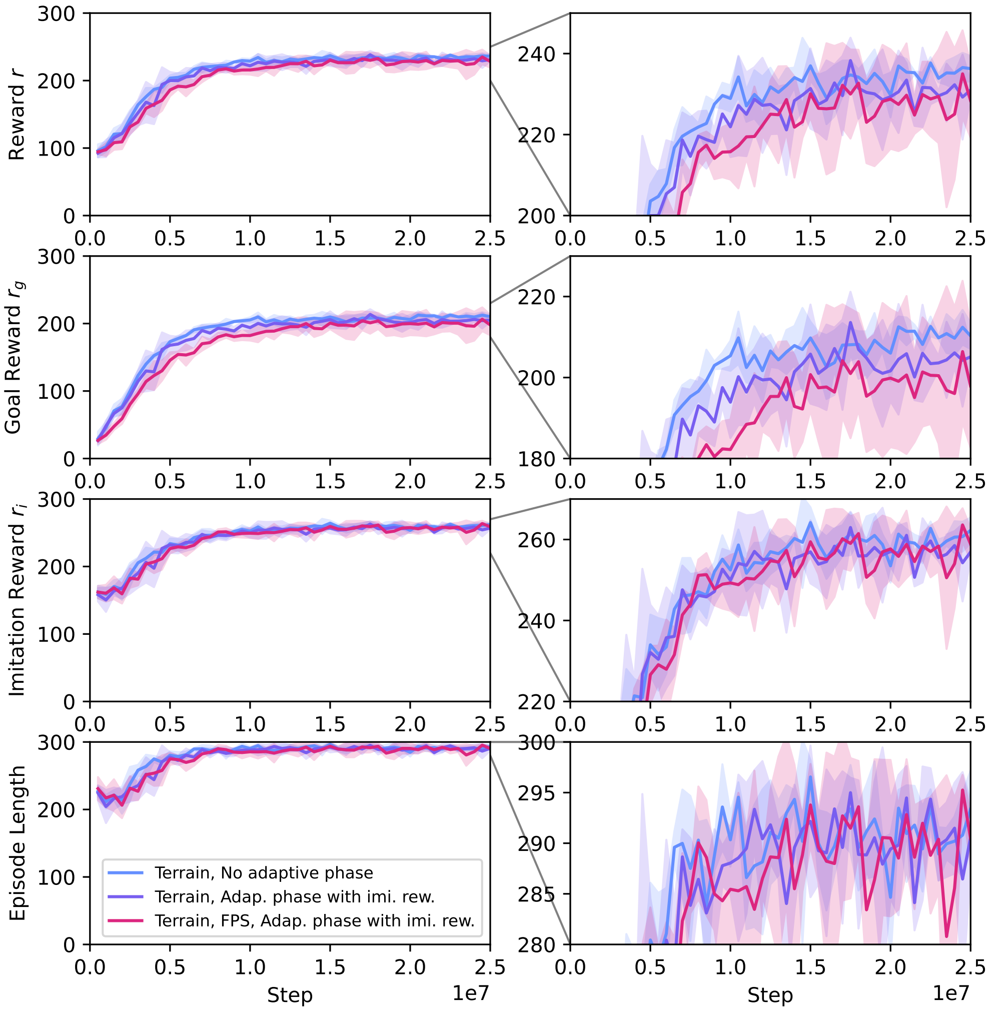


Figure 6.19: Influence of the adaptive action when the terrain is used, with and without FPS. No increase in achieved reward can be observed for the adaptive phase policy.

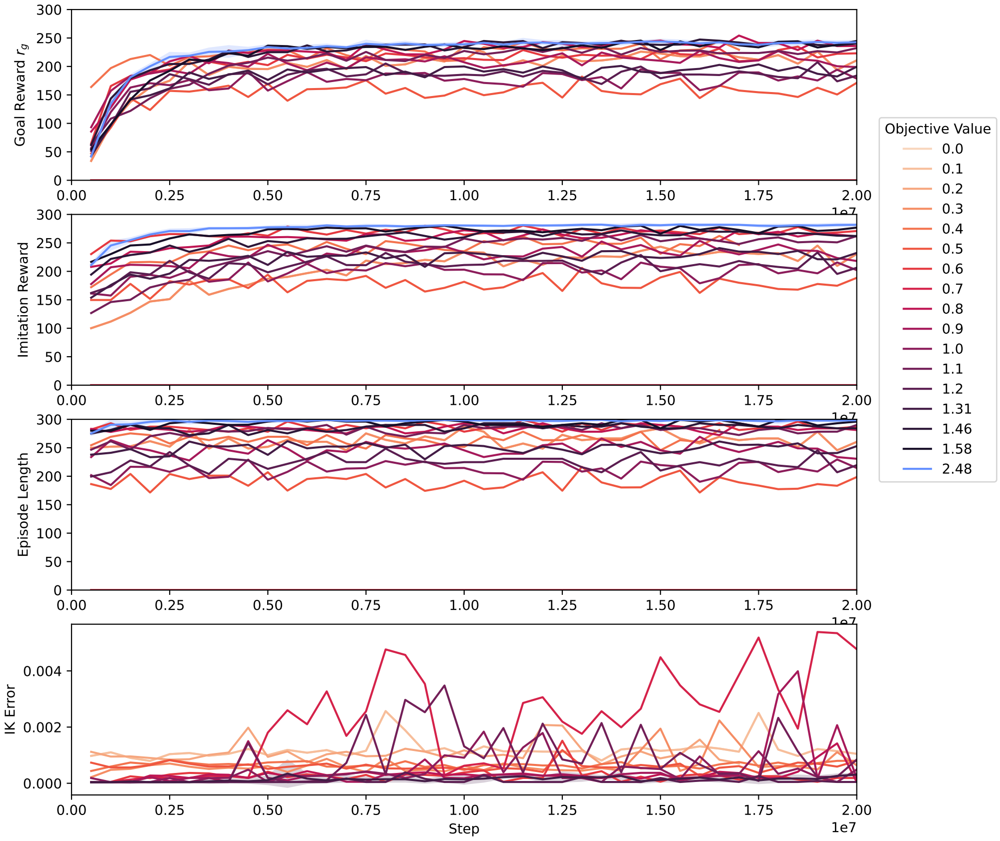


Figure 6.20: Comparison of the training using reference motions with different qualities of parameters. The previously used policy with the best parameter set is displayed in **blue**.

a second experiment with terrain and FPSs input, which should allow the policy to recognize when the foot makes contact with the ground (see Figure 6.19). Still, the performance of the different configurations remains the same.

It can be concluded that the adaptive phase approach does not improve the policy's performance.

6.3.7 Quality of Reference Motions

One of our main hypotheses for building the DeepQuintic approach was that the quality of the reference motion influences the achieved reward of the trained policy (see Section 6.1.1). It is difficult to quantify the quality of a reference motion, especially if it was generated using MOCAP data. In our case, we have an advantage as our reference motion is generated by a parameterized walk skill. We have already shown in Chapter 5 that the quality of these parameters defines how well the walk skill performs. Therefore, we can use intentionally bad or suboptimal parameters for our reference motion to lower its quality. Furthermore, it is possible to quantify their quality using the objective function that we used for their optimization (see Section 5.3.5).

We generated 10,000 sets of parameters using random sampling to ensure that the similarity between these sets is not influenced by the optimization algorithm. Due to the inferior performance of the random search approach, the best-performing parameter

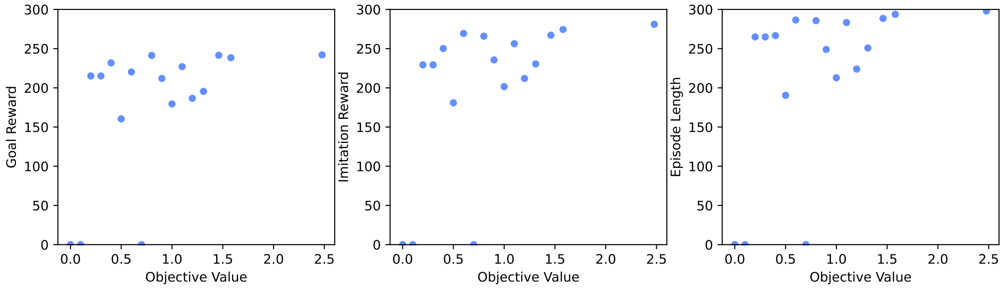


Figure 6.21: Relationship between the objective value of the used reference motion and the reached reward of the policy after 20M steps. In this experiment, all policies only executed command velocities that were kinematically solvable by their reference motions.

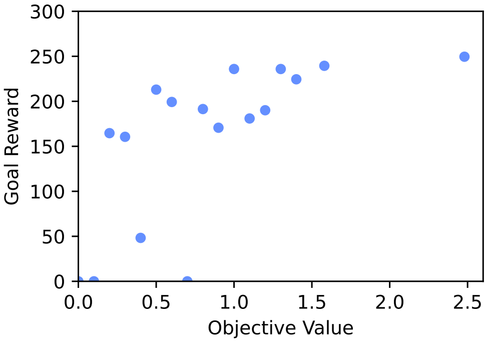


Figure 6.22: Relationship between the objective value of the used reference motion and the reached goal reward. In this experiment, all policies were evaluated using the same command velocities as the policy with the best parameter set. [BZ23]

set had an objective value of 1.58, which is lower than the 2.48 that was achieved by the parameter set that we used in all previous experiments. Additionally, we took one parameter set for each 0.1 step of objective value, to get equally distributed samples. This was not exactly possible for the higher objective values, as there were too few sets that reached these values. We then trained a Cartesian Euler angles policy with the FA hyperparameters for each parameter set. Each policy was trained for 20M steps. The results are shown in Figure 6.20 and 6.21. See [1] for an exemplary video showing the qualitative differences between the policies.

The results show that the policies indeed perform differently when using different parameter sets for the goal reward, imitation reward, the episode length, and also for the IK error of the policy. Still, they don't show a clear linear relationship. Some bad parameter sets even manage to achieve almost equal performance as the best parameter set, which has a significantly higher objective value. We assume that these results are influenced by the choice of v_{cmd}. Since we only train the policy for velocities that are kinematically solvable, some parameters might result in having lower values for v_{cmd} and, thereby, an easier evaluation. Three of the low-quality parameter sets led to gaits that were not solvable by the IK without large errors and, therefore, never found any command velocities that fit. They are displayed with achieving a zero reward and episode length.

We evaluated the goal reward of the trained policies again using the same range for

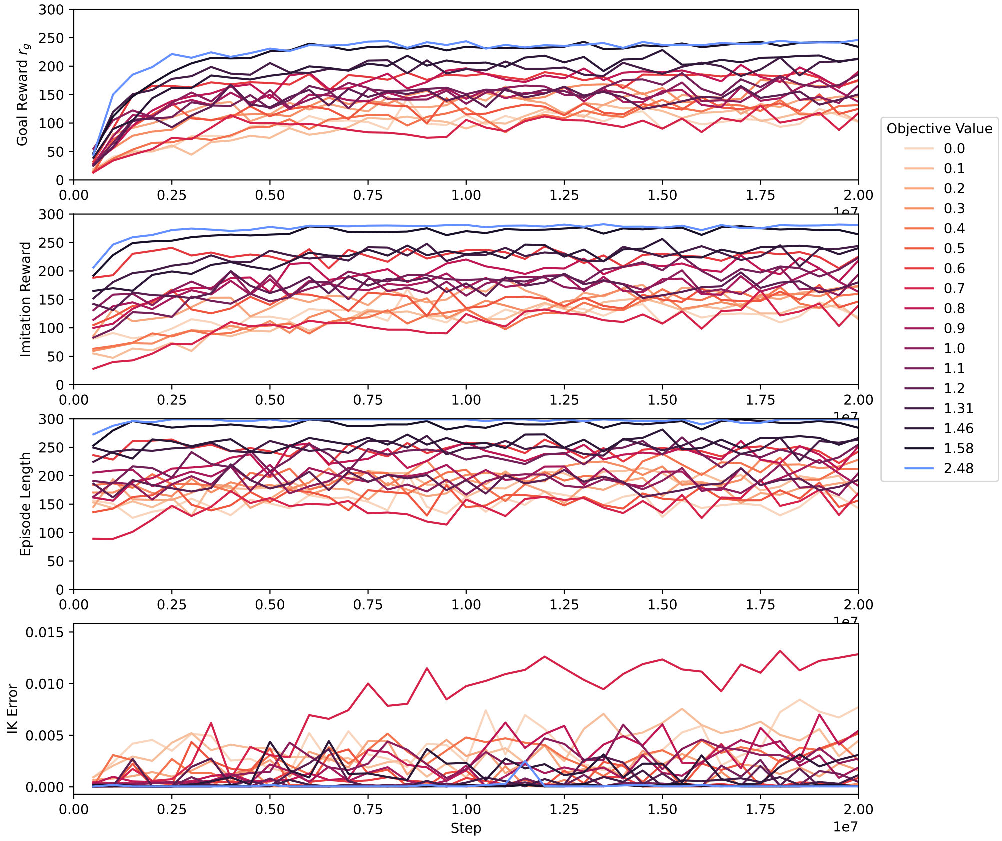


Figure 6.23: Comparison of the training using reference motions with different qualities of parameters but using the same possible command velocities. The previously used policy with the best parameter set is displayed in **blue**.

v_{cmd} that the policy with the best parameter set has. Since the reference motion is not used for the execution of a trained policy, it does not matter if it can be solved by the IK without errors. The result shows that the previous experiment was, indeed, influenced by having different v_{cmd} (see Figure 6.22).

This result could still be unfair to the other parameter sets since they did not encounter some of the command velocities during training. Therefore, we also trained policies again with a deactivated check for kinematic solvability. Thereby, all policies were trained with the same velocities even if the reference motion could not completely solve them. The results are displayed in Figure 6.23 and Figure 6.24. The performance difference between low- and high-quality parameters is more distinctive. This is expected since the reference motions are partially giving wrong poses since there is no kinematic solution.

We can conclude from these experiments that reference motions of lower quality reduce the achieved performance of the trained policy. However, the relationship between the achieved goal reward and the objective value of the parameter set is not perfectly linear. Some parameters with similar objective values lead to different goal rewards. This might also be influenced by the algorithm that we use to determine the objective value, which only depends on the achieved maximal velocity. A parameter set might lead to an early fall due to a single bad parameter, e.g., the torso being leaned too much forward. The rest of the motion might be good, so the RL policy can easily adapt

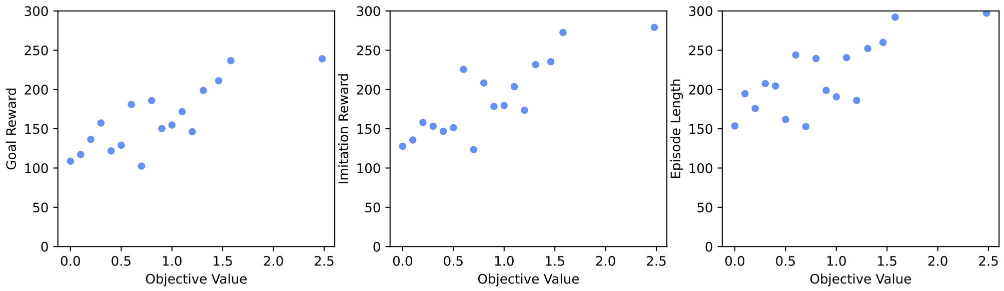


Figure 6.24: Relationship between the objective value of the used reference motion and the reached reward. In this experiment, all policies were trained and evaluated using the same command velocities as the policy with the best parameter set. Based on [BZ23]

the orientation of the torso and thus achieve a high reward. Using a different objective function might lead to clearer results. Still, overall the policy with the best reference motion was not outperformed by any of the others.

6.3.8 Usage of Arms

The arms of a humanoid robot can be used for stabilization during walking. We have already argued why we don't want to allow arbitrary arm movements in our case (see Section 5.3.3). Still, smaller arm movements would be possible without further issues. In contrast to other approaches that rely on MOCAP, we don't have goals for the arms in our reference. Therefore, it is interesting to see if the policy still achieves a sensible motion for the arms. Additionally, we want to test how the network initialization influences the movement of the arms.

We tested three scenarios. In the first scenario, we added the default pose of the arms to the reference motion and used it in the imitation reward. The arm actions in the network were also initialized to this pose. Therefore, the arms should learn to keep close to this pose. In the second scenario, we also initialized the network as before but did not give any imitation reward based on the arm movements. In the third scenario, we did not initialize the network and gave no imitation reward for the arms. Thus, we expected them to move more freely. Additionally, we use the joint space policy without arm movement as a baseline. All experiments were trained for 25 M steps using the same hyperparameters of the joint policy and including network initialization for the legs. The results are shown in Figure 6.25. Actuating the arms through the policy leads to no increase in the achieved goal reward. Other researchers have also removed actions for the arms in their RL locomotion policies due to their small influence [ZJF$^+$22].

We also investigated if the arms are moved differently based on the usage of imitation reward and network initialization (see Figure 6.26). It is interesting to see how the network's initialization influences the arms' pose. In both cases, the policy learns a solution close to the action of the untrained network. Therefore, the policy without network initialization uses an unnatural pose while the other stays close to the typical poses of the robot's arms. Naturally, it would also be possible to provide a more sophisticated imitation reward if the reference motion includes splines for the movement of the arms. However, we do not expect that this would improve the performance significantly.

6.3.9 Comparison to Spline Approach

We have already shown that the OptiQuint-based walking that we use as reference motion can also directly be used as a walk skill (see Section 5.3). The additional effort

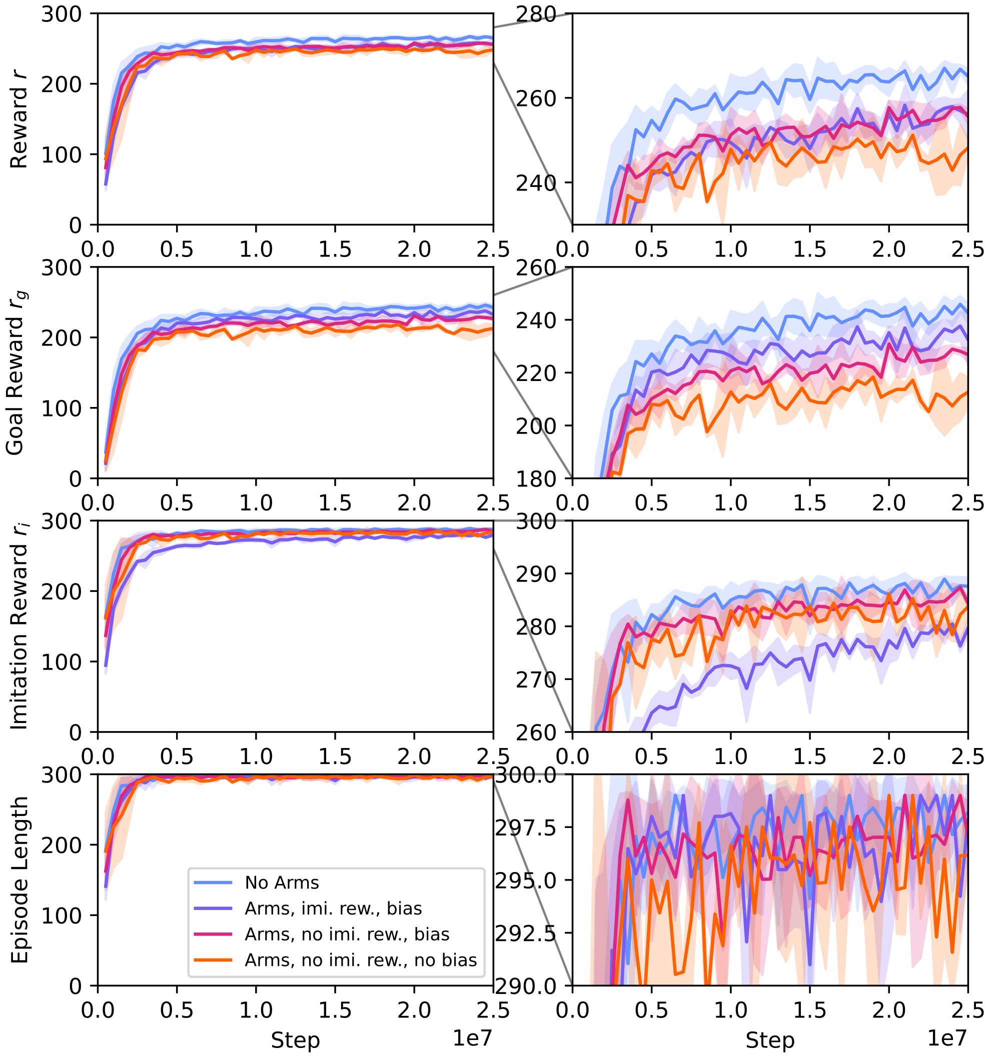
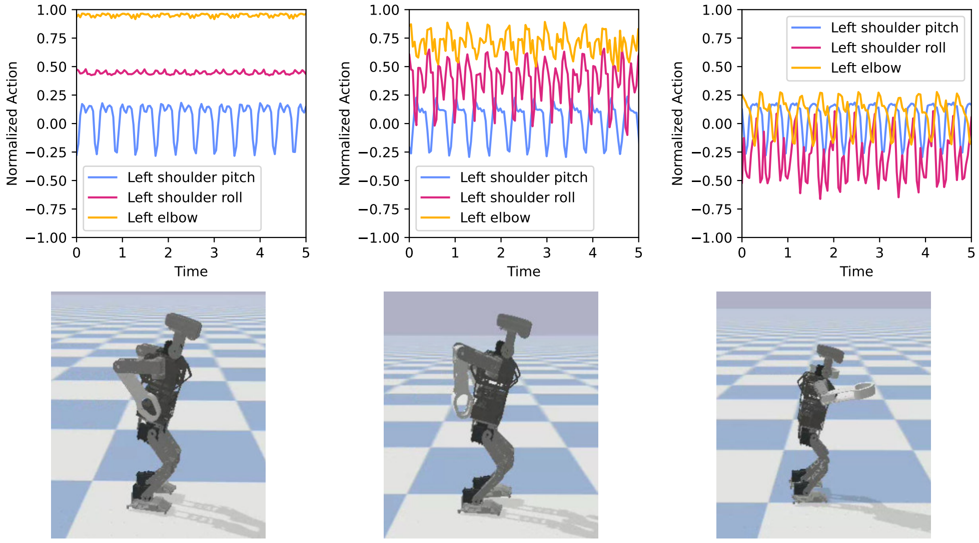


Figure 6.25: Influence of actuating the arms, with and without imitation reward for these actions, as well as with and without initial network action bias. Actively controlling the arm joints decreases the achieved goal reward in all cases.



(a) Imi. rew, bias (b) No imi. rew., bias (c) No imi. rew., no bias

Figure 6.26: Plot of the normalized arm joint actions. It can be observed that the usage of the initial network bias strongly influences the resulting motion. A video showing these different policies can be seen at video [1].

that is needed to train the RL policy is only sensible if the resulting skill is superior to the reference skill. Therefore, we compared the performance of the OptiQuint-based walk skill with the best policy (the Euler angles one with the fused angles hyperparameter set). We evaluated the OptiQuint skill once without stabilization to show how much the RL policy can improve in comparison to its reference motion. Additionally, we evaluate the OptiQuint skill with activated PID stabilization. Four different scenarios were tested. The default environment without changes, additional random head movements (similar to the behavior of the head during RoboCup), random forces acting on the robot's base, and a random terrain with a high difference of 5cm. Each combination was tested for 100 episodes. The results are displayed in Table 6.2.

The stabilized reference motion only slightly improves on the open-loop one and mostly in terms of fewer falls (longer episode length). The RL policy clearly outperforms the reference motion both in following the commanded velocity and in preventing falls. This also shows that the RL does not only learn to replicate the reference motion but actually improves on them. An exemplary comparison between the reference action and the corresponding action of the RL policy is shown in Figure 6.27.

6.3.10 Generalization

It was already shown that the OptiQuint-based reference motion is applicable to other robot types as long as they don't use parallel kinematics (see Section 5.3.7). Similar to this, it was evaluated if the DeepQuintic approach also generalizes to all of these robots. Since these models are only available in Webots, the training was performed in this simulator. For each robot, the Euler angle-based policy with FA hyperparameters was trained one time without further hyperparameter optimization. Since the robots have different sizes, their Cartesian action spaces need to be defined individually. Additionally, the speed these platforms can achieve is different. Therefore, the commanded velocities were also set differently (see Table 8.3). Otherwise, the same training proce-

Table 6.2: Comparison between reference and policy

		Ref.	Stab. Ref.	Pol.
No Change	Goal rew.	141	141	**275**
	Ep. length	292	293	**300**
Head	Goal rew.	140	140	**272**
	Ep. length	290	292	**300**
Force	Goal rew.	129	130	**262**
	Ep. length	281	289	**300**
Terrain	Goal rew.	19	22	**211**
	Ep. length	59	68	**288**

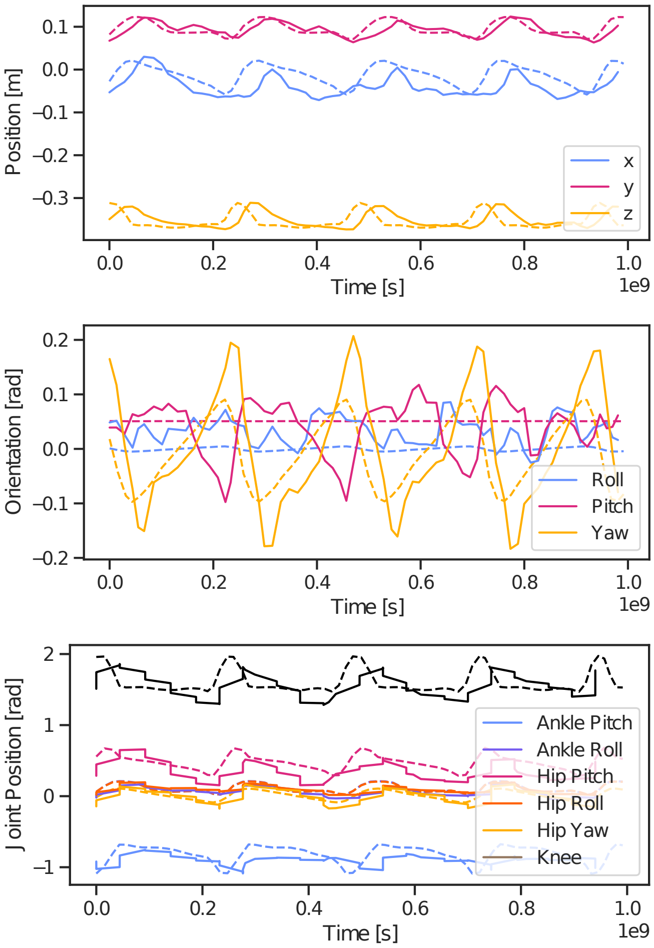


Figure 6.27: Exemplary plot of the actions produced by the policy π_c for the left foot. The Cartesian position (**top**) and orientation (**center**) are displayed together with the corresponding reference action (dashed). Furthermore, the resulting joint position commands (**bottom**) are shown. The robot was walking with $v_{cmd} = (0.2\frac{m}{s}, 0.1\frac{m}{s}, -0.5\frac{rad}{s})$. It is visible that the policy diverges from the reference trajectory to stabilize the robot.

dure as described in Section 6.3.1 was used. The training is shown in Figure 6.28. The achieved walk velocities of the trained policies are displayed in Table 6.3 together with the results from the OptiQuint approach (from Table 5.3).

The training of all robots is successful, showing the generalization of the approach. The RL-based approach outperforms the spline-based one for most robots and directions. The other cases might be explained by not well-chosen action spaces for some robots or the hyperparameters not being optimized for the individual robots.

6.3.11 Domain Transfer

Training a RL policy in simulation has many benefits, e.g., no wear of the hardware. However, it does not guarantee that the result is also applicable to the real robot. This problem is also called the sim-to-real transfer [ZQW20]. There are multiple factors that lead to the performance difference between the simulation and the real robot. These can be inaccuracies of the robot model, e.g., wrong values for the link masses or the joint torques, as well as inaccuracies of the environment model, e.g., wrong friction values of the ground. Additionally, the policy is often trained using discrete steps in which the policy's state is computed from the current simulator state. On a real robot, especially one that uses a message passing based middleware like ROS, the policy's state is always lagging behind the actual state of the robot. Furthermore, some of the sensor information may have different latency than others, based on how the system is designed.

In recent years, domain randomization has been used frequently [ZQW20]. This method changes model and simulation parameters randomly for each training episode. Thereby, a policy is learned that performs well in all of these configurations, and it is expected that the real system is one of these. Additionally, some authors have introduced a latency in the training environment by providing a state to the policy that is always a certain number of time steps behind [RB21].

We also applied these methods by randomizing the link masses and inertias, as well as the joint maximal torques and velocities. Additionally, we randomized the contact properties of the feet and floor by changing the restitution as well as lateral, spinning, and rolling friction values. To model sensor noise, we added Gaussian noise to the state orientation and angular velocity entries in the state. Finally, we used the state of the previous time step to model a latency of 33 ms, which should be sufficient based on our previous experiments (see Section 4.4.2). Although we applied all of this, the final result on the robot was not satisfactory. Therefore, we separated the sim-to-real transfer into three steps in order to better understand where the problem stems from.

As a first step, we trained our policy in the PyBullet simulation and then executed it in the Webots simulation while still directly accessing the simulation without the message passing of ROS. This was successful and showed us that the policy can handle the difference between the simulations. Especially interesting is that the Webots model includes a simulated backlash and the knee torsion spring (see Section 3.3.1), which are not included in the PyBullet model. The usage of PD controllers instead of direct torque control might have helped to simplify the transition. Especially the different behavior of the knee joint is probably directly compensated by the PD controller.

As the second step, we executed the policy again in the Webots simulation but accessed the simulation through the ROS software stack. For this, we used the ROS interface of the policy (see Section 6.2.7). This was again successful (see video [1]). It shows that the latency that is introduced by the message passing does not pose an issue to the policy. This also shows that there are no errors coming from the implementation of the policy's ROS interface or different IMU filter behavior.

As a final step, we executed the policy using the ROS interface on the real robot

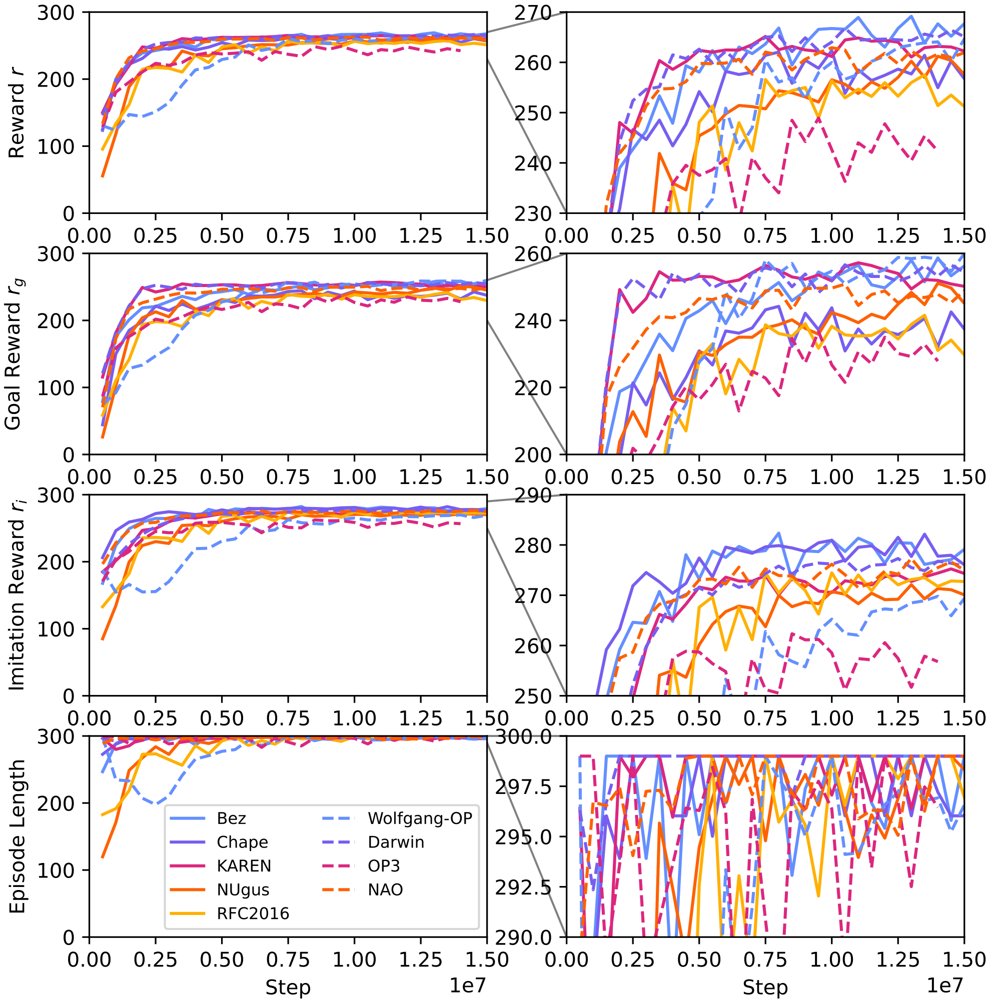


Figure 6.28: Training of different robots in Webots. It can be seen that the training works similarly well on all platforms. Note that the goal reward is not directly comparable as the robots have different possible goal velocities. An exemplary video of the trained policies can be seen at [1].

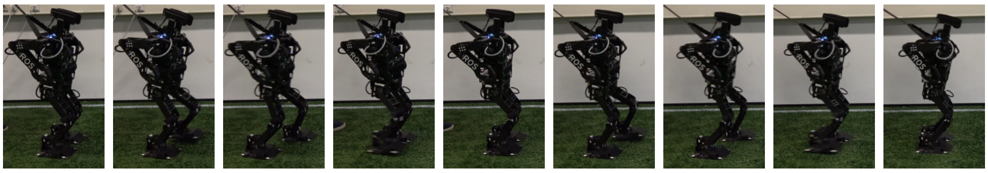


Figure 6.29: The trained policy walking a few steps forward. Note that the robot needs to be slightly stabilized by a rope at its back. The corresponding video can be seen at [1].

Table 6.3: Maximal walk velocities for different robots using DeepQuintic compared to OptiQuint

Robot Platform	Team/Company	Height	Forward		Backward		Sideward		Turn	
		[m]	[m/s]		[m/s]		[m/s]		[rad/s]	
			DQ	OQ	DQ	OQ	DQ	OQ	DQ	OQ
Bez	UTRA	0.50	0.17	0.21	0.31	0.05	0.1	0.12	2.35	2.45
Chape	ITAndroids	0.53	0.25	0.30	0.39	0.36	0.17	0.10	2.57	2.09
KAREN	MRL-HSL	0.73	0.54	0.54	0.61	0.48	0.22	0.18	3.22	3.10
NUgus	NUbots	0.90	0.56	0.38	0.55	0.42	0.35	0.26	2.2	1.97
RFC2016	01.RFC Berlin	0.73	0.46	0.37	0.34	0.40	0.31	0.36	2.45	1.99
Wolfgang-OP	Hamburg Bit-Bots	0.83	0.69	0.48	0.67	0.51	0.21	0.22	2.39	1.89
Darwin/OP2	Robotis	0.45	0.20	0.29	0.22	0.29	0.26	0.12	2.17	2.47
OP3	Robotis	0.51	0.51	0.45	0.67	0.54	0.28	0.13	2.13	2.01
NAO	Aldebaran	0.57	0.3	0.43	0.29	0.58	0.16	0.25	1.75	0.85

(see Figure 6.29). While the robot was able to perform walking motions with the commanded directions, it was not stable and kept tilting forward. Investigating this issue further showed that the IMU of the robot is always slightly offset due to its mounting. Even this small orientation difference made a large impact on the policy's action. This was unexpected since we used random noise on the IMU orientation during training. However, this was Gaussian distributed and sampled for each time step. Therefore, it was not a constant offset. We solved this problem by adding parameters that could be tuned manually to correctly calibrate the IMU orientation offsets. This resolved the behavior of constant tilting. However, the policy was still not able to walk stably (see video [1]).

We also observed that the policy constantly operated the servos at full load, so they were running hot after just a brief moment of walking. This has not been the case with the OptiQuint approach. We assume that this is due to oscillations of the goal joint positions that are generated by the policy since it always just reacts to the current state. This might be one issue for not being able to walk stably since the servo's performance degrades if it becomes hot. Since this motor behavior is not simulated, it did not create any issues when running the policy in the simulation. There are multiple possible solutions for this. The actions could be filtered, e.g., by a butterworth filter [B$^+$30] to reduce their oscillation as it was done by Rodriguez et al. [RB21]. It would also be possible to adapt the reward function to include a term that punishes oscillating actions or power consumption. Finally, using a different network for the policy, e.g., a long short-term memory (LSTM), might improve this since the action would not only rely on the current state and thereby might oscillate less. Another issue that leads to the walk being unstable is the large backlash in the robot's legs (see Section 3.6.4). Unfortunately, this could not be resolved as it would have required a complete redesign of the platform.

Although we were not able to achieve a completely stable walk on the real robot, we observed that different measures had a qualitatively observable positive influence. We, therefore, list them here as advice for further research. Since the policy only runs at 30 Hz, limiting the step frequency of the reference motion is crucial. Otherwise, it only has very few actions for every single step, which leads to jerky motions. Executing the policy with a higher frequency than it was trained, e.g., 60 Hz instead of 30 Hz, seemed to lead to smoother movements. Using a reference motion with a wide lateral distance between the foot is important as the policy otherwise might positions them close to each other, making them collide due to the backlash. For the domain randomization, we advise not to randomize the joint values, i.e., maximal velocity and torque, upwards but only downwards, since the real servos basically never perform better than their values in the data sheet but are often worse especially if they are older. The above-mentioned calibration of the IMU orientation offset is also important. Furthermore, it is necessary to use a complementary filter to estimate the robot's orientation during training instead of directly getting these values from the simulator (see Section 6.2.2). The filter introduces a latency to the estimated orientation, which will also be present on the real robot.

The sim-to-real transfer of the OptiQuint approach (see Section 5.3.7 and 5.4.3) was simpler as it was possible to change parameters manually while executing the approach on the real robot. This was only possible because the parameters of the approach were few, and their influence on the motion is easily understandable for a human. This is not the case for the RL policy since it has a higher number of parameters and their relation to the motion is not directly understandable. A different network design for the policy, which enforces more structure, might simplify manual changes post after training.

We did not change the command velocity during an episode in the training of the policy. Therefore, it never explicitly learned to change its walk velocity. Still, it was able to walk stably in simulation with changing commands as long as no extreme changes to the command were made. This also fits our observation of the OptiQuint approach, which did not do this either during optimization but still performed well.

6.4 Summary and Future Work

Our approach of combining optimized quintic-spline-based reference actions with deep reinforcement learning achieved omnidirectional walking and generalized to multiple different other robot platforms. This was done with a minimal reward function and without curriculum learning. The optimization of the reference motion could also be seen as an optimization of the reward function.

We investigated different design choices for training bipedal walking with reinforcement learning. Specific effort was made to prevent influences due to the hyperparameters of PPO. In our opinion, this has been a neglected point in some previous publications. First, we evaluated policies with different orientation representations in Cartesian space and compared them to joint-based policies. The Euler and fused angles performed best in the tested scenario since all rotations are limited, but we do not expect that this holds true for all types of motions. Still, it may be preferable to use them for bipedal walking or other motions with limited rotations on the end effectors. Second, we showed that the achieved reward correlates to the quality of the reference motion. This is an important finding since most previous approaches relied on motion capture data from humans, which is not optimal for the robot and thus may limit the achieved performance of the policy. Third, we showed that the initial action of an untrained policy has a large influence if position control is used. This might be the most interesting result since it may have long-reaching implications for the training of policies position control. It was also interesting to see that many changes in the environment, e.g., adding more sensor information to the state, did not change the achieved reward of the policy.

All our experiments were done using PPO, and the results could differ for other RL algorithms. Therefore, further experiments with other RL algorithms would be good to verify the results. Moreover, a comparison between using the optimized reference motion and one from human motion capture data should be made. The orientation representations need to be investigated for other humanoid motions, e.g., standing up, to see if the Euler angles still perform best in these cases where gimbal locks can be an issue. It should be investigated if optimizing the hyperparameters only based on the achieved goal reward leads to further improvements. Furthermore, different other approaches that lead to non-fixed-cycle policies should be investigated, e.g., a binary action that decides if a step is done could be used. The approach of giving an initial bias to the policy's action could be verified on further RL problems that have a similar problem like locomotion. An increase in the policy's control rate should also be investigated, as this might lead to quicker reactions to external forces. However, during the work on this, we recognized that it is more difficult to train the policy with a higher frequency since the effects of the policy's action on the state are smaller. The software framework's structure and the Sim2Real transfer might be improved by using the EagerX framework [vdHLF+22]. Finally, (partial) training on the real robot is something that should be further considered as most ground works have been created by this thesis. However, the problem of wear on the robot's hardware must be solved for this.

Chapter 7

Conclusion

The aim of this thesis was to contribute to the field of humanoid robotics by setting up a robust low-cost robot platform and by investigating the learning of motion skills, especially bipedal locomotion. This was achieved, and further, we were able to advance the state of the art in multiple research areas. The most important finding advances reinforcement learning with reference motions by investigating their influence on the training outcome. We have shown that non-optimal reference motions can limit the performance of the trained policy and should, therefore, be avoided. We hypothesize that these non-optimal references lead to non-optimal specifications of the reward in the imitation term. This finding is especially relevant since previous approaches have extensively relied on MOCAP data and keyframe animations that were not optimized for the target platforms. We also devised an approach to generate optimized reference motions for different motion skills and showed that it generalizes well to various robots. The usage of these optimized motions as references significantly improved the performance of our policy.

During our experiments, we identified another important factor influencing the training of policies that use position-based actions with PD controllers. This factor is the initial action bias used by the policy at the start of the training. We could show in two different bipedal walking environments that an initial action close to a good motion not only accelerates the training but also leads to a better performance in the trained policy. We were also able to outperform the previous baseline in a standard gym environment. This finding has a great impact as the combination of RL policies with PD joint controllers are often used to aid in the sim-to-real transfer. Furthermore, our results indicate that previous works that used position control have not considered this design choice carefully enough and have, therefore, attributed it a too-low performance.

We investigated different design choices for the training of a bipedal locomotion skill using PPO. We identified the choice of the space in which the policy operates as the most important one. Using Cartesian space instead of joint space leads to better-performing policies. We hypothesize that this difference arises from the more direct relationship between the state and the action. In joint space, the policy might need to implicitly learn the IK. Although we only tested it for the problem of bipedal walking, we assume that the usage of Cartesian space might be beneficial for other motion skills, too.

To allow the execution of the aforementioned RL experiments, we needed to solve multiple other issues, which led to further findings and contributions. We compared the performance of different black-box optimization algorithms for the parameter optimization of our motion skills. Thereby, we discovered that the usage of multi-objective optimization with a posteriori scalarization performs superior to single-objective optimization with a priori scalarization. The optimized motion skills could be transferred to the real robot and were used in different competitions. Further, the approach is already in use by other researchers, showing the benefit of such a simple-to-use approach.

Originally, we planned to train motion skills directly on the real hardware. Although these plans were later changed due to the high wear, research was made to create an autonomous handling of falls that would allow such automatic learning on real hardware. The devised HCM approach turned out to be useful in general. It simplifies the deliberative layer of the robot to a wheeled-robot-like complexity. To ensure that the fall protection can be invoked early enough, we investigated how falls can be detected by comparing various approaches and modalities. It turned out that a simple threshold-based classifier using IMU data is sufficient. However, the usage of an MLP classifier can further improve the lead time for detecting falls. We also showed how the latencies introduced by the ROS message transfer can be reduced and compared different versions of ROS in this regard. Furthermore, a novel decision-making framework, the DSD, was created as a basis for the HCM approach. This framework also turned out to be generally useful and was already applied in other contexts, e.g., for a blackjack dealer robot [FGN+23].

As an experimental platform, the Wolfgang-OP was developed. To improve its bus communication, the problems of existing approaches were investigated, and a novel solution was found that outperforms all previous works. This work already impacted other researchers, who then used our approach in their hardware. Additionally, an equation was devised to compute the optimal torsion spring strength for a PEA minimizing the load on the robot's knee joints. Its validity was confirmed by experiments with the real robot.

7.1 Future Work

While most of the set goals were reached, there are still parts that require further improvements. We tried to transfer our trained policy from the training environment to the real robot using domain adaption. The policy was successfully transferred to another simulator and integrated into the existing ROS stack, but its transfer to the real robot was only partially successful. While the policy was able to walk omnidirectional, it was not stable enough. We identified the backlash of the robot's joints as one of the main issues that remain to be solved. This can only be done by changing the used servo type to another one, e.g., a brushless DC servo with a planetary-style gearbox. This comes with the additional benefit that the different gear ratio might allow faster movements. Due to the typical dimensions of the gearbox type, further changes to the robot would be necessary to integrate them. The new BRUCE robot [LSZ+22] features low backlash, faster joints, and is able to produce dynamic movements. A 3D printed approach, i.e., [UARM22], might allow achieving lower costs.

Our RL based approach was only tested for bipedal walking and kicking. Testing further skills, especially standing-up, is necessary to ensure that the advantage of a Cartesian policy generalizes to various types of motions. A direct comparison between using optimized reference motions and MOCAP data might show the resulting performance difference more clearly. All of our experiments were conducted using the PPO algorithm. Therefore, similar experiments should also be conducted using other approaches, e.g., Soft Actor-Critic (SAC) [HZAL18].

While the usage of approximate solutions to approach unreachable Cartesian poses was successful, it should be investigated if this issue could be resolved in a better way. One alternative is the description of the action space as a polytope, exactly describing the space of valid solutions. The action output of the policy could still remain as normalized values between -1 and 1 if a mapping to the polytope can be described.

We showed that the initial action of an untrained network has a large influence on the resulting performance and provided a method to set a bias to it. Nonetheless, this was only tested on two environments with bipedal walking skills. Therefore, it should be further tested with other motion types to evaluate whether our findings generalize to all motion skills.

We tested both of our approaches on a wide variety of simulated robots. However, none of those had the size of an adult human, as no such model was available. In the future, these approaches should be tested on larger humanoids to ensure that they also generalize to these.

The fall detection could be improved by a more generalized dataset that is recorded on more than one robot type. Furthermore, lifelong learning could be applied to improve the performance of the HCM in this regard over time. More sophisticated solutions for the fall protection actions could further reduce the impact forces. This would be especially interesting for larger robots where passive elastic elements might not be sufficient for damage prevention.

Chapter 8

Appendix

Algorithm 4 Check Falling

```
procedure FallQuantification(angle, ang_vel)
    moving_more_upright = sign(angle) $\neq$ sign(ang_vel)
    over_point_of_no_return = abs(angle) > angle_threshold
    if not moving_more_upright or over_point_of_no_return then
        skalar = max((angle_threshold - abs(angle)) / angle_threshold, 0)
        if ang_vel_threshold * skalar < abs(ang_vel) then
            return abs(ang_vel) * (1 - skalar)
        end if
    end if
    return 0
end procedure

roll_quantification = FallQuantification(roll_angle, roll_ang_vel)
pitch_quantification = FallQuantification(pitch_angle, pitch_ang_vel)
if roll_quantification == 0 and pitch_quantification == 0 then
    result = “stable”
else
    if pitch_quantification > roll_quantification then
        if pitch_angle < 0 then
            result = “back”
        else
            result = “front”
        end if
    else
        if roll_angle < 0 then
            result = “left”
        else
            result = “right”
        end if
    end if
end if
return result
```

Table 8.1: Ranges of the optimized parameters.

Parameter	Bez	Chape	Darwin-OP	KAREN	NAO	NUgus	OP3	RFC2016	Wolfgang-OP
step frequency	2.376	2.671	2.060	2.950	2.846	2.882	2.840	2.994	1.981
double support ratio	0.045	0.060	0.034	0.044	0.302	0.021	0.050	0.211	0.032
foot distance	0.118	0.085	0.100	0.152	0.080	0.166	0.098	0.188	0.158
foot rise	0.031	0.028	0.055	0.072	0.060	0.067	0.074	0.118	0.060
torso height	0.150	0.223	0.195	0.250	0.298	0.426	0.214	0.326	0.396
torso phase offset	-0.150	0.072	-0.174	-0.123	0.128	-0.108	-0.023	-0.253	-0.005
torso lateral swing ratio	0.512	0.024	0.606	0.255	0.016	0.030	0.476	0.426	0.265
torso rise	0.031	0.028	0.055	0.007	0.024	0.004	0.017	0.021	0.017
torso x offset	-0.021	-0.002	0.008	0.038	0.007	0.023	0.015	-0.007	-0.018
torso y offset	-0.002	0.006	-0.002	0.001	0.001	0.007	-0.001	-0.001	0.000
torso pitch	0.102	0.308	0.236	0.115	-0.055	0.396	-0.085	0.419	0.135
torso pitch proportional to v_{cmd_x}	-0.544	-0.696	-1.097	-0.278	-.0126	-1.099	1.313	-1.365	0.222
torso pitch proportional to v_{cmd_yaw}	0.017	0.079	0.215	-0.053	-0.267	0.230	-0.194	-0.007	0.166

Table 8.2: Hyperparameters for PPO

Parameter name	Euler	Fused	Quaternion	6D	Joint	Joint without bias	Euler state reward	Euler non cyclic
batch_size	64	256	512	512	256	128	128	64
clip_range	0.3	0.3	0.1	0.2	0.4	0.3	0.3	0.3
gae_lambda	0.9	0.8	0.8	0.8	0.8	0.9	0.9	0.9
gamma	0.95	0.95	0.9	0.95	0.9	0.9	0.95	0.95
learning_rate	0.83e-4	1.24e-4	0.75e-4	0.48e-4	0.12e-4	0.16e-4	0.39e-4	0.40e-4
log_std_init	-2.93	-3.62	-2.63	-2.68	-3.46	-3.01	-3.54	-2.97
max_grad_norm	0.3	2.0	0.9	5.0	5.0	0.9	0.9	0.3
n_epochs	20	20	1	10	80	10	40	20
n_steps	1024	1024	128	256	512	128	1024	1024
network layers	[64,64]	[64,64,64]	[1024,1024,1024]	[1024,1024]	[64,64]	[64,64,64]	[256,256]	[64,64]
activation_fn	tanh	tanh	tanh	tanh	tanh	tanh	tanh	tanh
ent_coef	0	0	0	0	0	0	0	0
n_envs	8	8	8	8	8	8	8	8
vf_coef	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

Table 8.3: Training Configuration for Different Robot Types

Robot	v_{cmd_x}	v_{cmd_y}	v_{cmd_yaw}	x	y	z	roll	pitch	yaw
Bez	$[-0.20, 0.20]$	$[-0.20, 0.20]$	$[-2.00, 2.00]$	$[-0.07, 0.18]$	$[0.00, 0.18]$	$[-0.20, -0.10]$	$[\pi/3, \pi/3]$	$[\pi/6, \pi/6]$	$[\pi/4, \pi/4]$
Chape	$[-0.25, 0.25]$	$[-0.25, 0.25]$	$[-2.00, 2.00]$	$[-0.05, 0.2]$	$[0.00, 0.15]$	$[-0.26, -0.16]$	$[\pi/3, \pi/3]$	$[\pi/6, \pi/6]$	$[\pi/4, \pi/4]$
Darwin-OP	$[-0.23, 0.23]$	$[-0.25, 0.25]$	$[-2.00, 2.00]$	$[-0.05, 0.20]$	$[0.0, 0.20]$	$[-0.26, -0.12]$	$[\pi/3, \pi/3]$	$[\pi/6, \pi/6]$	$[\pi/4, \pi/4]$
KAREN	$[-0.48, 0.54]$	$[-0.18, 0.18]$	$[-3.10, 3.10]$	$[-0.2, 0.25]$	$[0.00, 0.25]$	$[-0.30, -0.15]$	$[\pi/3, \pi/3]$	$[\pi/6, \pi/6]$	$[\pi/4, \pi/4]$
NAO	$[-0.25, 0.25]$	$[-0.25, 0.25]$	$[-1.00, 1.00]$	$[-0.10, 0.12]$	$[0.00, 0.10]$	$[-0.34, -0.25]$	$[\pi/3, \pi/3]$	$[\pi/6, \pi/6]$	$[\pi/6, \pi/6]$
NUgus	$[-0.60, 0.50]$	$[-0.30, 0.30]$	$[-2.00, 2.00]$	$[-0.25, 0.35]$	$[-0.05, 0.30]$	$[-0.45, -0.24]$	$[\pi/4, \pi/4]$	$[\pi/6, \pi/6]$	$[\pi/4, \pi/4]$
OP3	$[-0.60, 0.50]$	$[-0.30, 0.30]$	$[-2.00, 2.00]$	$[-0.20, 0.25]$	$[0.00, 0.22]$	$[-0.25, -0.10]$	$[\pi/3, \pi/3]$	$[\pi/6, \pi/6]$	$[\pi/4, \pi/4]$
RFC2016	$[-0.40, 0.40]$	$[-0.35, 0.35]$	$[-2.00, 2.00]$	$[-0.15, 0.30]$	$[-0.02, 0.28]$	$[-0.44, -0.15]$	$[\pi/3, \pi/3]$	$[\pi/6, \pi/6]$	$[\pi/4, \pi/4]$
Wolfgang-OP	$[-0.60, 0.60]$	$[-0.25, 0.25]$	$[-2.00, 2.00]$	$[-0.17, 0.27]$	$[0, 0.27]$	$[-0.44, -0.24]$	$[\pi/3, \pi/3]$	$[\pi/6, \pi/6]$	$[\pi/4, \pi/4]$

Bibliography

[AB15] Philipp Allgeuer and Sven Behnke. Fused angles: A representation of body orientation for balance. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 366–373. IEEE, 2015.

[AB16] Philipp Allgeuer and Sven Behnke. Omnidirectional bipedal walking with direct fused angle feedback mechanisms. In *International Conference on Humanoid Robots (Humanoids)*, pages 834–841. IEEE, 2016.

[ABC$^+$17] Matthew Amos, Alex Biddulph, Stephan Chalup, Daniel Ginn, Alexandre Mendes, Josephus Paye, Ysobel Sims, Peter Turner, and Taylor Young. The NUbots team description paper 2019. Technical report, University of Newcastle, 2017.

[ADF$^+$16] Julien Allali, Louis Deguillaume, Rémi Fabre, Loic Gondry, Ludovic Hofer, Olivier Ly, Steve N'Guyen, Grégoire Passault, Antoine Pirrone, and Quentin Rouxel. Rhoban football club: Robocup humanoid kid-size 2016 champion team paper. In *Robot World Cup*, pages 491–502. Springer, 2016.

[Aea17] Rami Aly et al. Application from hamburg bit-bots for robocup 2017. Technical report, Universität Hamburg, 2017.

[AFSB15] Philipp Allgeuer, Hafez Farazi, Michael Schreiber, and Sven Behnke. Child-sized 3d printed igus humanoid open platform. In *International Conference on Humanoid Robots (Humanoids)*, pages 33–40. IEEE, 2015.

[AMW19] Sayantan Auddy, Sven Magg, and Stefan Wermter. Hierarchical control for bipedal locomotion using central pattern generators and neural networks. In *International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob)*, pages 13–18. IEEE, 2019.

[ARS$^+$20] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Leonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, et al. What matters for on-policy deep actor-critic methods? a large-scale study. In *International conference on learning representations*, 2020.

[ASSV16] Saminda W. Abeyruwan, Dilip Sarkar, Faisal Sikder, and Ubbo Visser. Semi-automatic extraction of training examples from sensor readings for fall detection and posture monitoring. *IEEE Sensors Journal*, 16(13):5406–5415, 2016.

[ASY$^+$19] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

[AvS20] Minoru Asada and Oskar von Stryk. Scientific and technological challenges in robocup. *Annual Review of Control, Robotics, and Autonomous Systems*, 3:441–471, 2020.

[B$^+$30] Stephen Butterworth et al. On the theory of filter amplifiers. *Wireless Engineer*, 7(6):536–541, 1930.

[BA15] Patrick Beeson and Barrett Ames. TRAC-IK: An open-source library for improved solving of generic inverse kinematics. In *International Conference on Humanoid Robots (Humanoids)*, pages 928–935. IEEE, 2015.

[BB19] Guillaume Bellegarda and Katie Byl. Training in task space to speed up and guide reinforcement learning. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 2693–2699. IEEE, 2019.

[BBBK11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[BBE$^+$19] Marc Bestmann, Hendrik Brandt, Timon Engelke, Niklas Fiedler, Alexander Gabel, Jasper Güldenstein, Jonas Hagge, Judith Hartfill, Tom Lorenz, Tanja Heuer, Martin Poppinga, Ivan David Ria no Salamanca, and Daniel Speck. Hamburg bit-bots and wf wolves team description for RoboCup 2019 humanoid kidsize. Technical report, Universität Hamburg, 2019.

[BCLN22] Guillaume Bellegarda, Yiyu Chen, Zhuochen Liu, and Quan Nguyen. Robust high-speed running for quadruped robots via deep reinforcement learning. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 10364–10370. IEEE, 2022.

[BCP$^+$16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[BDF$^+$02] Hans-Dieter Burkhard, Dominique Duhaut, Masahiro Fujita, Pedro Lima, Robin Murphy, and Raul Rojas. The road to RoboCup 2050. *IEEE Robotics & Automation Magazine*, 9(2):31–38, 2002.

[BDG$^+$17] Tobias Bolze, Vitor Dallasta, Reinhard Gerndt, Alexander Gabel, Tanja Heuer, Oliver Krebs, Tom Lorenz, Ivan David Ria no Salamanca, Frank Stiddien, and Rodrigo da Silva Guerra. WF Wolves and Taura Bots – humanoid teensize team description for RoboCup 2017. Technical report, Ostfalia University of Applied Sciences, Wolfenbüttel, Germany, 2017.

[BEF$^+$22] Marc Bestmann, Timon Engelke, Niklas Fiedler, Jasper Güldenstein, Jan Gutsche, Jonas Hagge, and Florian Vahl. Torso-21 dataset: Typical objects in robocup soccer 2021. In *Robot World Cup*, pages 65–77. Springer, 2022.

[Ber20] Eric Claus Bergter. Image based robot localization and orientation classification using CGI and photographic data. Bachelor's thesis, Universität Hamburg, 2020.

[Bes17] Marc Bestmann. Towards using ROS in the robocup humanoid soccer league. Master's thesis, Universität Hamburg, 2017.

[BES19] Frederico Bormann, Timon Engelke, and Finn-Thorben Sell. Developing a reactive and dynamic kicking engine for humanoid robots. Technical report, Universität Hamburg, 2019.

[BFGH20] Marc Bestmann, Niklas Fiedler, Jasper Güldenstein, and Jonas Hagge. Hamburg bit-bots humanoid league 2020. Technical report, Universität Hamburg, 2020.

[BGV92] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Annual workshop on Computational learning theory*, pages 144–152, 1992.

[BGVZ21] Marc Bestmann, Jasper Güldenstein, Florian Vahl, and Jianwei Zhang. Wolfgang-OP: A robust humanoid robot platform for research and competitions. In *International Conference on Humanoid Robots (Humanoids)*, pages 90–97. IEEE, 2021.

[BGZ19] Marc Bestmann, Jasper Güldenstein, and Jianwei Zhang. High-frequency multi bus servo and sensor communication using the dynamixel protocol. In *Robot World Cup*, pages 16–29. Springer, 2019.

[BHMC18] Alexander Biddulph, Trent Houliston, Alexandre Mendes, and Stephan K Chalup. Comparing computing platforms for deep learning on a humanoid robot. In *International Conference on Neural Information Processing*, pages 120–131. Springer, 2018.

[BHW09] Marc D. Binder, Nobutaka Hirokawa, and Uwe Windhorst. *Encyclopedia of neuroscience*, volume 3166. Springer Berlin, 2009.

[BM08] Luigi Biagiotti and Claudio Melchiorri. *Trajectory planning for automatic machines and robots*. Springer Science & Business Media, 2008.

[BN20] Guillaume Bellegarda and Quan Nguyen. Robust quadruped jumping via deep reinforcement learning. *arXiv preprint arXiv:2011.07089*, 2020.

[BPK$^+$18] Gerardo Bledt, Matthew J. Powell, Benjamin Katz, Jared Di Carlo, Patrick M. Wensing, and Sangbae Kim. MIT cheetah 3: Design and control of a robust, dynamic quadruped robot. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2245–2252. IEEE, 2018.

[Bro86] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE journal on robotics and automation*, 2(1):14–23, 1986.

[BRW15] Marc Bestmann, Bente Reichardt, and Florens Wasserfall. Hambot: an open source robot for robocup soccer. In *Robot Soccer World Cup*, pages 339–346. Springer, 2015.

[BTB$^+$22] Steven Bohez, Saran Tunyasuvunakool, Philemon Brakel, Fereshteh Sadeghi, Leonard Hasenclever, Yuval Tassa, Emilio Parisotto, Jan Humplik, Tuomas Haarnoja, Roland Hafner, et al. Imitate and repurpose: Learning reusable robot movement skills from human and animal behaviors. *arXiv preprint arXiv:2203.17138*, 2022.

[BW71] Nester Burtnyk and Marceli Wein. Computer-generated key-frame animation. *Journal of the SMPTE*, 80(3):149–153, 1971.

[BZ20] Marc Bestmann and Jianwei Zhang. Humanoid control module: An abstraction layer for humanoid robots. In *International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 263–268. IEEE, 2020.

[BZ22] Marc Bestmann and Jianwei Zhang. Bipedal walking on humanoid robots through parameter optimization. In *Robot World Cup*. Springer, 2022.

[BZ23] Marc Bestmann and Jianwei Zhang. Reference motion quality and design choices for bipedal walking with DeepRL. Submitted, 2023.

[CGKK17] Ronnapee Chaichaowarat, Diego Felipe Paez Granados, Jun Kinugawa, and Kazuhiro Kosuge. Passive knee exoskeleton using torsion spring for cycling assistance. In *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017.

[CHJH02] Murray Campbell, Joseph Hoane Jr., and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[CMEM$^+$17] Sachin Chitta, Eitan Marder-Eppstein, Wim Meeussen, Vijay Pradeep, Adolfo Rodríguez Tsouroukdissian, Jonathan Bohren, David Coleman, Bence Magyar, Gennaro Raiola, Mathias Lüdtke, and Enrique Fernández Perdomo. ros_control: A generic and simple control framework for ros. *The Journal of Open Source Software*, 2(20):456–456, 2017.

[CMQ07] Stephan K Chalup, Craig L Murch, and Michael J Quinlan. Machine learning with AIBO robots in the four-legged league of RoboCup. *IEEE Transactions on Systems, Man, and Cybernetics*, 37(3):297–310, 2007.

[CMS17] Po-Wei Chou, Daniel Maturana, and Sebastian Scherer. Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution. In *International conference on machine learning*, pages 834–843. PMLR, 2017.

[CÖ18] Michele Colledanchise and Petter Ögren. *Behavior trees in robotics and AI: An introduction.* CRC Press, 2018.

[CSC12] Sachin Chitta, Ioan Sucan, and Steve Cousins. Moveit! *IEEE Robotics & Automation Magazine*, 19(1):18–19, 2012.

[DABI18] Florin Dzeladini, Nadine Ait-Bouziad, and Auke Ijspeert. CPG-based control of humanoid robot locomotion. *Humanoid Robotics: A Reference*, pages 1–35, 2018.

[DAPM00] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In *International Conference on Parallel Problem Solving from Nature*, pages 849–858. Springer, 2000.

[DDG$^+$21] Helei Duan, Jeremy Dao, Kevin Green, Taylor Apgar, Alan Fern, and Jonathan Hurst. Learning task space actions for bipedal locomotion. In *International Conference on Robotics and Automation (ICRA)*, pages 1276–1282. IEEE, 2021.

[DHI+22] Nicholas Dziura, Abigail Hall, Yuji Ishikawa, Sam McFarlane, Alexandre Mendes, Cameron Murtagh, Alana Noonan, Josephus Paye II, Mikyla Peters, Thomas O'Brien, et al. The nubots team extended abstract 2022. Technical report, University of Newcastle, 2022.

[DHW19] Klaus Dorer, Ulrich Hochberg, and Michael Wülker. The sweaty 2019 robocup humanoid adultsize team description. Technical report, University of Applied Sciences Offenburg, 2019.

[Dia10] Rosen Diankov. *Automated construction of robotic manipulation programs.* Dissertation, Carnegie Mellon University, 2010.

[dMP18] Rodrigo F. de Mello and Moacir A. Ponti. Machine learning: a practical approach on the statistical learning theory. *Springer*, 2018.

[EIS+20] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep policy gradients: A case study on ppo and trpo. *arXiv preprint arXiv:2005.12729*, 2020.

[Eng21] Timon Engelke. Learning to kick from demonstration with deep reinforcement learning. Bachelor's thesis, Universität Hamburg, 2021.

[EPY+22] Alejandro Escontrela, Xue Bin Peng, Wenhao Yu, Tingnan Zhang, Atil Iscen, Ken Goldberg, and Pieter Abbeel. Adversarial motion priors make good substitutes for complex reward functions. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 25–32. IEEE, 2022.

[Ero95] Kutluhan Erol. *Hierarchical task network planning: formalization, analysis, and implementation.* University of Maryland, College Park, 1995.

[FAFB17] Grzegorz Ficht, Philipp Allgeuer, Hafez Farazi, and Sven Behnke. NimbRo-OP2: Grown-up 3d printed open humanoid platform for research. In *International Conference on Humanoid Robotics (Humanoids)*, pages 669–675. IEEE, 2017.

[FB21] Grzegorz Ficht and Sven Behnke. Bipedal humanoid hardware design: A technology review. *Current Robotics Reports*, 2(2):201–210, 2021.

[FBG+19] Niklas Fiedler, Hendrik Brandt, Jan Gutsche, Florian Vahl, Jonas Hagge, and Marc Bestmann. An open source vision pipeline approach for robocup humanoid soccer. In *Robot World Cup*, pages 376–386. Springer, 2019.

[FBZ19] Niklas Fiedler, Marc Bestmann, and Jianwei Zhang. Position estimation on image-based heat map input using particle filters in cartesian space. In *International Conference on Industrial Cyber Physical Systems (ICPS)*, pages 269–274. IEEE, 2019.

[FFB+18] Grzegorz Ficht, Hafez Farazi, André Brandenburger, Diego Rodriguez, Dmytro Pavlichenko, Philipp Allgeuer, Mojtaba Hosseini, and Sven Behnke. NimbRo-OP2X: Adult-sized open-source 3d printed humanoid robot. In *International Conference on Humanoid Robots (Humanoids)*, pages 1–9. IEEE, 2018.

[FGN+23] Niklas Fiedler, Jasper Güldenstein, Theresa Naß, Michael Görner, Norman Hendrich, and Jianwei Zhang. A multimodal robotic blackjack dealer: Design, implementation, and reliability analysis. Submitted, 2023.

[FH89] Evelyn Fix and Joseph Lawson Hodges. Discriminatory analysis. nonparametric discrimination: Consistency properties. *International Statistical Review/Revue Internationale de Statistique*, 57(3):238–247, 1989.

[Fie19] Niklas Fiedler. Distributed multi object tracking with direct FCNN inclusion in robocup humanoid soccer. Bachelor's thesis, Universität Hamburg, 2019.

[FKH18] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, pages 1437–1446. PMLR, 2018.

[FKK+02] Kiyoshi Fujiwara, Fumio Kanehiro, Shuuji Kajita, Kenji Kaneko, Kazuhito Yokoi, and Hirohisa Hirukawa. UKEMI: Falling motion control to minimize damage to biped humanoid robot. In *International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 2521–2526. IEEE, 2002.

[Fle19] Tanja Flemming. Evaluating and minimizing the reality gap in the domain of robocup humanoid soccer. Bachelor's thesis, Universität Hamburg, 2019.

[FN71] Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[Fre16] Lutz et al. Freitag. Berlin united - FUmanoids team description paper for robocup 2016. Technical report, Freie Universität Berlin, 2016.

[FRP+16] Rémi Fabre, Quentin Rouxel, Grégoire Passault, Steve N'Guyen, and Olivier Ly. Dynaban, an open-source alternative firmware for dynamixel servo-motors. In *Robot World Cup*, pages 169–177. Springer, 2016.

[GA17] Erico Guizzo and Evan Ackerman. The TurtleBot3 teacher. *IEEE Spectrum*, 54(8):19–20, 2017.

[GB22] Jasper Güldenstein and Marc Bestmann. Hamburg bit-bots team description for RoboCup 2022 humanoid league kidsize. Technical report, Universität Hamburg, 2022.

[GCS19] Cristyan R. Gil, Hiram Calvo, and Humberto Sossa. Learning an efficient gait cycle of a biped robot based on reinforcement learning and artificial neural networks. *Applied Sciences*, 9(3):502, 2019.

[GHB+09] D. Gouaillier, V. Hugel, P. Blazevic, C. Kilner, J. Monceaux, P. Lafourcade, B. Marnier, J. Serre, and B. Maisonnier. Mechatronic design of NAO humanoid. In *International Conference on Robotics and Automation*. IEEE, 2009.

[GHLZ+19] Loïc Gondry, Ludovic Hofer, Patxi Laborde-Zubieta, Olivier Ly, Lucie Mathé, Grégoire Passault, Antoine Pirrone, and Antun Skuric. Rhoban football club: Robocup humanoid kidsize 2019 champion team paper. In *Robot World Cup*. Springer, 2019.

[Gil64] Arthur Gill. *Introduction to the Theory of Finite-state Machines.* McGraw-Hill book company, Inc., New York, 1964.

[Gül19] Jasper Güldenstein. Comparison of measurement systems for kinematic calibration of a humanoid robot. Bachelor's thesis, Universität Hamburg, 2019.

[Gül22] Jasper Güldenstein. Footstep planning using reinforcement learning for humanoid robots. Master's thesis, Universität Hamburg, 2022.

[GVK15] Sergio Castro Gomez, Marsette Vona, and Dimitrios Kanoulas. A three-toe biped foot with hall-effect sensing. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 360–365. IEEE, 2015.

[Hag19] Jonas Hagge. Using FCNNs for goalpost detection in the robocup humanoid soccer domain. Bachelor's thesis, Universität Hamburg, 2019.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[Har19] Judith Hartfil. Feature-based monte carlo localization in the robocup humanoid soccer league. Master's thesis, Universität Hamburg, 2019.

[HFL$^+$16] Trent Houliston, Jake Fountain, Yuqing Lin, Alexandre Mendes, Mitchell Metcalfe, Josiah Walker, and Stephan K Chalup. Nuclear: A loosely coupled software architecture for humanoid robot systems. *Frontiers in Robotics and AI*, 3:20, 2016.

[HHH$^+$19] Ayonga Hereid, Omar Harib, Ross Hartley, Yukai Gong, and Jessy W Grizzle. Rapid trajectory optimization using c-frost with illustration on a cassie-series dynamic walking biped. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 4722–4729. IEEE, 2019.

[HL15] Sehoon Ha and C Karen Liu. Multiple contact planning for minimizing damage of humanoid falls. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 2761–2767. IEEE, 2015.

[HMS$^+$20] Clinton G Hobart, Anirban Mazumdar, Steven J Spencer, Morgan Quigley, Jesper P Smith, Sylvain Bertrand, Jerry Pratt, Michael Kuehl, and Stephen P Buerger. Achieving versatile energy efficiency with the wanderer biped robot. *IEEE Transactions on Robotics*, 36(3), 2020.

[HO01] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[HTA$^+$11] Inyong Ha, Yusuke Tamura, Hajime Asama, Jeakweon Han, and Dennis W Hong. Development of open humanoid platform DARwIn-OP. In *SICE annual conference*, pages 2178–2181. IEEE, 2011.

[HW06] Avrum Hollinger and Marcelo M. Wanderley. Evaluation of commercial force-sensing resistors. In *Proceedings of the International Conference on New Interfaces for Musical Expression, Paris, France*, pages 4–8. Citeseer, 2006.

[HZAL18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[JLO+19] Hyobin Jeong, Inho Lee, Jaesung Oh, Kang Kyu Lee, and Jun-Ho Oh. A robust walking controller based on online optimization of ankle, hip, and stepping strategies. *IEEE Transactions on Robotics*, 35(6):1367–1386, 2019.

[JM03] Odest Chadwicke Jenkins and Maja J Mataric. Automated derivation of behavior vocabularies for autonomous humanoid motion. In *International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 225–232, 2003.

[KAK+97] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. Robocup: The robot world cup initiative. In *International conference on autonomous agents*, pages 340–347, 1997.

[Kal60] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal Basic Engineering*, 82:35−45, 1960.

[Kat73] Ichiro Kato. Development of WABOT 1. *Biomechanism*, 2:173–214, 1973.

[KCB+16] S. Kajita, R. Cisneros, M. Benallegue, T. Sakaguchi, S. Nakaoka, M. Morisawa, K. Kaneko, and F. Kanehiro. Impact acceleration of falling humanoid robot with an airbag. In *International Conference on Humanoid Robots (Humanoids)*. IEEE, 2016.

[Kea22] Masato Kubotera et al. Extended abstract CIT brains for RoboCup 2022. Technical report, Chiba Institute of Technology, 2022.

[KG11] Shivaram Kalyanakrishnan and Ambarish Goswami. Learning to predict humanoid fall. *International Journal of Humanoid Robotics*, 8(02):245–273, 2011.

[KH20] Joanne Taery Kim and Sehoon Ha. Observation space matters: Benchmark and optimization algorithm. *arXiv preprint arXiv:2011.00756*, 2020.

[KHHY14] Shuuji Kajita, Hirohisa Hirukawa, Kensuke Harada, and Kazuhito Yokoi. *Introduction to humanoid robotics*. Springer, 2014.

[KKK+01] Shuuji Kajita, Fumio Kanehiro, Kenji Kaneko, Kazuhito Yokoi, and Hirohisa Hirukawa. The 3d linear inverted pendulum mode: A simple modeling for a biped walking pattern generation. In *International Conference on Intelligent Robots and Systems*, volume 1, pages 239–246. IEEE, 2001.

[KKS+17] Y. Kakiuchi, M. Kamon, N. Shimomura, S. Yukizaki, N. Takasugi, S. Nozawa, K. Okada, and M. Inaba. Development of life-sized humanoid robot platform with robustness for falling down, long time working and error occurrence. In *International Conference on Intelligent Robots and Systems (IROS)*, 2017.

[KLP19] Mohammadreza Kasaei, Nuno Lau, and Artur Pereira. A fast and stable omnidirectional walking engine for the nao humanoid robot. In *Robot World Cup*, pages 99–111. Springer, 2019.

[KNY20] Hyung Joo Kim, Jaeho Noh, and Woosung Yang. Knee-assistive robotic exoskeleton (KARE-1) using a conditionally singular mechanism for industrial field applications. *Applied Sciences*, 10(15), 2020.

[KPM$^+$21] Tobias Kronauer, Joshwa Pohlmann, Maximilian Matthé, Till Smejkal, and Gerhard Fettweis. Latency analysis of ROS2 multi-node systems. In *International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, pages 1–7. IEEE, 2021.

[KW$^+$08] Jason Kulk, James Welsh, et al. A low power walk for the NAO robot. In *Australasian Conference on Robotics & Automation (ACRA)*, pages 1–7, 2008.

[KW11] Jason Kulk and James S Welsh. A nuplatform for software on articulated mobile robots. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 31–45. Springer, 2011.

[LCP$^+$21] Zhongyu Li, Xuxin Cheng, Xue Bin Peng, Pieter Abbeel, Sergey Levine, Glen Berseth, and Koushil Sreenath. Reinforcement learning for robust parameterized locomotion control of bipedal robots. In *International Conference on Robotics and Automation (ICRA)*, pages 2811–2817. IEEE, 2021.

[LLL$^+$21] Chih-Cheng Liu, Yi-Chung Lin, Wei-Fan Lai, Ching-Chang Wong, et al. Kicking motion planning of humanoid robot based on b-spline curves. *Journal of Applied Science and Engineering*, 25(4):623–631, 2021.

[LLTZ16] Qingdu Li, Guodong Liu, Jun Tang, and Jianwei Zhang. A simple 2d straight-leg passive dynamic walking model without foot-scuffing problem. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 5155–5161. IEEE, 2016.

[LRO13] Matthieu Lapeyre, Pierre Rouanet, and Pierre-Yves Oudeyer. The poppy humanoid robot: Leg design for biped locomotion. In *International Conference on Intelligent Robots and Systems*. IEEE, 2013.

[LSZ$^+$22] Yeting Liu, Junjie Shen, Jingwen Zhang, Xiaoguang Zhang, Taoyuanmin Zhu, and Dennis Hong. Design and control of a miniature bipedal robot with proprioceptive actuation for dynamic behaviors. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 8547–8553. IEEE, 2022.

[MB13] Marcell Missura and Sven Behnke. Self-stable omnidirectional walking with compliant joints. In *Workshop on Humanoid Soccer Robots, International Conference on Humanoid Robots*. IEEE, 2013.

[MBB20] Marcell Missura, Maren Bennewitz, and Sven Behnke. Capture steps: Robust walking for humanoid robots. *International Journal of Humanoid Robotics*, page 1950032, 2020.

[Men00] Alberto Menache. *Understanding motion capture for computer animation and video games*. Morgan kaufmann, 2000.

[MFG$^+$22] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[MFL19] Thomas Mergner, Michael Funk, and Vittorio Lippi. Embodiment and humanoid robotics. *Philosophisches Handbuch Künstliche Intelligenz*, pages 1–27, 2019.

[Mic04] Olivier Michel. Cyberbotics Ltd. Webots™: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):5, 2004.

[Mir20] Robin Mirow. Embedded debug interface for robots. Bachelor's thesis, Universität Hamburg, 2020.

[MKA16] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ROS2. In *International Conference on Embedded Software*, pages 1–10, 2016.

[MLR10] Judith Müller, Tim Laue, and Thomas Röfer. Kicking a ball–modeling complex dynamic motions for humanoid robots. In *Robot Soccer World Cup*, pages 109–120. Springer, 2010.

[MM19] Luckeciano Carvalho Melo and Marcos Ricardo Omena Albuquerque Máximo. Learning humanoid robot running skills through proximal policy optimization. In *2019 Latin american robotics symposium (LARS), 2019 Brazilian symposium on robotics (SBR) and 2019 workshop on robotics in education (WRE)*, pages 37–42. IEEE, 2019.

[MMdC19] Luckeciano Carvalho Melo, Marcos Ricardo Omena Albuquerque Maximo, and Adilson Marques da Cunha. Learning humanoid robot motions through deep neural networks. *arXiv preprint arXiv:1901.00270*, 2019.

[MMdC22] Dicksiano C Melo, Marcos ROA Maximo, and Adilson Marques da Cunha. Learning push recovery behaviors for humanoid walking using deep reinforcement learning. *Journal of Intelligent & Robotic Systems*, 106(1):8, 2022.

[MMLG$^+$19] Roberto Martín-Martín, Michelle A Lee, Rachel Gardner, Silvio Savarese, Jeannette Bohg, and Animesh Garg. Variable impedance control in end-effector space: An action space for reinforcement learning in contact-rich tasks. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 1010–1017. IEEE, 2019.

[MMYH10] Michael Mistry, Akihiko Murai, Katsu Yamane, and Jessica Hodgins. Sit-to-stand task on a humanoid robot from human demonstration. In *International Conference on Humanoid Robots*, pages 218–223. IEEE, 2010.

[MR17] Thomas Muender and Thomas Röfer. Model-based fall detection and fall prevention for humanoid robots. In *Robot World Cup*, pages 312–324. Springer, 2017.

[Mur91] Fionn Murtagh. Multilayer perceptrons for classification and regression. *Neurocomputing*, 2(5-6):183–197, 1991.

[New12] Monty Newborn. *Kasparov versus Deep Blue: Computer chess comes of age*. Springer Science & Business Media, 2012.

[NN08] Dragomir N. Nenchev and Akinori Nishio. Ankle and hip strategies for balance recovery of a biped subjected to an impact. *Robotica*, 26(5):643–653, 2008.

[NSP19a] Gabe Nelson, Aaron Saunders, and Robert Playter. The petman and atlas robots at boston dynamics. *Humanoid Robotics: A Reference*, 169:186, 2019.

[NSP19b] Gabe Nelson, Aaron Saunders, and Robert Playter. The PETMAN and Atlas Robots at Boston Dynamics. In *Humanoid Robotics: A Reference*. Springer, 2019.

[OTWO20] Yoshihiko Ozaki, Yuki Tanigaki, Shuhei Watanabe, and Masaki Onishi. Multiobjective tree-structured parzen estimator for computationally expensive optimization problems. In *Genetic and Evolutionary Computation Conference*, pages 533–541, 2020.

[PALvdP18] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Transactions on Graphics (TOG)*, 37(4):143, 2018.

[Pau72] Richard Paul Collins Paul. *Modelling, trajectory calculation and servoing of a computer controlled arm*. Stanford University, 1972.

[PB61] William Wesley Peterson and Daniel T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, 1961.

[PB22] Martin Poppinga and Marc Bestmann. DSD - dynamic stack decider: A lightweight decision making framework for robots and software agents. *International Journal of Social Robotics*, 14(1):73–83, 2022.

[PBJFG$^+$97] R Peter Bonasso, R James Firby, Erann Gat, David Kortenkamp, David P Miller, and Mark G Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2-3):237–256, 1997.

[PBYVDP17] Xue Bin Peng, Glen Berseth, KangKang Yin, and Michiel Van De Panne. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, 36(4):41, 2017.

[PCDG06] Jerry Pratt, John Carff, Sergey Drakunov, and Ambarish Goswami. Capture point: A step toward humanoid push recovery. In *International Conference on Humanoid Robots (Humanoids)*, pages 200–207. IEEE, 2006.

[PHK10] Chang-Soo Park, Young-Dae Hong, and Jong-Hwan Kim. Full-body joint trajectory generation using an evolutionary central pattern generator for stable bipedal walking. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 160–165. IEEE, 2010.

[PHK13] Chang-Soo Park, Young-Dae Hong, and Jong-Hwan Kim. Evolutionary-optimized central pattern generator for stable modifiable bipedal walking. *IEEE/ASME Transactions on Mechatronics*, 19(4):1374–1383, 2013.

[PKA18] Johannes Pankert, Lukas Kaul, and Tamim Asfour. Learning efficient omni-directional capture stepping for humanoid robots from human motion and simulation data. In *International Conference on Humanoid Robots (Humanoids)*, pages 431–434. IEEE, 2018.

[PKM$^+$18] Xue Bin Peng, Angjoo Kanazawa, Jitendra Malik, Pieter Abbeel, and Sergey Levine. Sfv: Reinforcement learning of physical skills from videos. In *SIGGRAPH Asia 2018 Technical Papers*, page 178. ACM, 2018.

[PMFC20] Bagaskara Primastya Putra, Gabrielle Satya Mahardika, Muhammad Faris, and Adha Imam Cahyadi. Humanoid robot pitch axis stabilization using linear quadratic regulator with fuzzy logic and capture point. *arXiv preprint arXiv:2012.10867*, 2020.

[PMN$^+$12] Alberto Parmiggiani, Marco Maggiali, Lorenzo Natale, Francesco Nori, Alexander Schmitz, Nikos Tsagarakis, José Santos Victor, Francesco Becchi, Giulio Sandini, and Giorgio Metta. The Design of the iCub Humanoid Robot. *International Journal of Humanoid Robotics*, 09(04), 2012.

[Poo89] Harry H. Poole. *Fundamentals of robotics engineering*. Springer Science & Business Media, 1989.

[PTLK18] Fabio Pardo, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. Time limits in reinforcement learning. In *International Conference on Machine Learning*, pages 4045–4054. PMLR, 2018.

[PvdP17] Xue Bin Peng and Michiel van de Panne. Learning locomotion skills using deeprl: Does the choice of action space matter? In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 1–13, 2017.

[PVG$^+$11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[PvNWV15] M. Plooij, M. van Nunspeet, M. Wisse, and H. Vallery. Design and evaluation of the Bi-directional Clutched Parallel Elastic Actuator (BIC-PEA). In *International Conference on Robotics and Automation (ICRA)*. IEEE, 2015.

[QCG$^+$09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3. Kobe, Japan, 2009.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[RB06] Reimund Renner and Sven Behnke. Instability detection and fall avoidance for a humanoid using attitude sensors and reflexes. In *International Conference on Intelligent Robots and Systems*, pages 2967–2973. IEEE, 2006.

[RB21] Diego Rodriguez and Sven Behnke. Deepwalk: Omnidirectional bipedal gait by deep reinforcement learning. In *International Conference on Robotics and Automation (ICRA)*, pages 3033–3039. IEEE, 2021.

[RBB18] Diego Rodriguez, André Brandenburger, and Sven Behnke. Combining simulations and real-robot experiments for bayesian optimization of bipedal gait stabilization. In *Robot World Cup*, pages 70–82. Springer, 2018.

[RdSMPT10] Javier Ruiz-del Solar, Javier Moya, and Isao Parra-Tsunekawa. Fall detection and management in biped humanoid robots. In *International Conference on Robotics and Automation*, pages 3323–3328. IEEE, 2010.

[Rei20] Wulf Adrian Reichardt. Creation of a spline-based throwing engine for robocup soccer. Bachelor's thesis, Universität Hamburg, 2020.

[RHSZ18] Philipp Ruppel, Norman Hendrich, Sebastian Starke, and Jianwei Zhang. Cost functions to specify full-body motion and multi-goal manipulation tasks. In *International Conference on Robotics and Automation (ICRA)*, pages 3152–3159. IEEE, 2018.

[RMM$^+$11] Katayon Radkhah, Christophe Maufroy, Moritz Maus, Dorian Scholz, Andre Seyfarth, and Oskar Von Stryk. Concept and design of the Biobiped1 robot for human-like walking and running. *International Journal of Humanoid Robotics*, 8(03), 2011.

[Rou17] Quentin Rouxel. *Apprentissage et correction des imperfections des robots humanoïdes de petite taille: application à l'odométrie et à la synthèse de mouvements [Learning and correction of imperfections in small humanoid robots: application to odometry and movement synthesis]*. PhD thesis, Université de Bordeaux, 2017. Original document in French.

[RP19] Lingmei Ren and Yanjun Peng. Research of fall detection and fall prevention technologies: A systematic review. *IEEE Access*, 7:77702–77722, 2019.

[RPH$^+$15] Quentin Rouxel, Gregoire Passault, Ludovic Hofer, Steve N'Guyen, and Olivier Ly. Rhoban hardware and software open source contributions for robocup humanoids. In *Workshop on Humanoid Soccer Robots, International Conference on Humanoid Robots (Humanoids)*. IEEE, 2015.

[RSH$^+$15] Nicolaus A Radford, Philip Strawser, Kimberly Hambuchen, Joshua S Mehling, William K Verdeyen, A Stuart Donnan, James Holley, Jairo Sanchez, Vienny Nguyen, Lyndon Bridgwater, et al. Valkyrie: NASA's first bipedal humanoid robot. *Journal of Field Robotics*, 32(3):397–419, 2015.

[RTvdP20] Daniele Reda, Tianxin Tao, and Michiel van de Panne. Learning to locomote: Understanding how environment design matters for deep reinforcement learning. In *SIGGRAPH Conference on Motion, Interaction and Games*, pages 1–10, 2020.

[Run01] Carl Runge. Über empirische funktionen und die interpolation zwischen äquidistanten ordinaten [about empiric functions and the interpolation between equidistant ordinates]. *Zeitschrift für Mathematik und Physik*, 46(224-243):20, 1901. Original document in German.

[RWA$^+$19] Aulia Khilmi Rizgi, Ryan Satria Wijaya, Ilham Fakhrul Arifin, Mochamad Ayuf Basthomi, Cipta Priambodo, Rokhmat Febrianto, Ibrahim Musthofainal Akhyar, Miftahul Anwar, Anhar Risnumawan, and Achmad Subhan Khalilullah. EROS - team description paper for humanoid kidsize league, robocup 2019. Technical report, Politeknik Elektronika Negeri Surabaya, 2019.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[SBHZ21] Sebastian Stelter, Marc Bestmann, Norman Hendrich, and Jianwei Zhang. Fast and reliable stand-up motions for humanoid robots using spline interpolation and parameter optimization. In *International Conference on Advanced Robotics (ICAR)*, pages 253–260. IEEE, 2021.

[SDN09] Ashutosh Saxena, Justin Driemeyer, and Andrew Y Ng. Learning 3-d object orientation from images. In *International conference on robotics and automation*, pages 794–800. IEEE, 2009.

[SF19] Olivier Stasse and Thomas Flayols. An overview of humanoid robots technologies. In *Biomechanics of Anthropomorphic Systems*, pages 281–310. Springer, 2019.

[SFB+17] Olivier Stasse, Thomas Flayols, Rohan Budhiraja, Kevin Giraud-Esclasse, Justin Carpentier, Joseph Mirabel, Andrea Del Prete, Philippe Souères, Nicolas Mansard, Florent Lamiraux, et al. Talos: A new humanoid research platform targeted for industrial applications. In *International Conference on Humanoid Robotics (Humanoids)*, pages 689–695. IEEE, 2017.

[SGV18] Satoshi Shigemi, Ambarish Goswami, and Prahlad Vadakkepat. Asimo and humanoid robot research at honda. *Humanoid robotics: A reference*, pages 55–90, 2018.

[Shi19] Satoshi Shigemi. ASIMO and Humanoid Robot Research at Honda. In *Humanoid Robotics: A Reference*. Springer, 2019.

[SJAB19] Saeed Saeedvand, Masoumeh Jafari, Hadi S. Aghdasi, and Jacky Baltes. A comprehensive survey on humanoid robot development. *The Knowledge Engineering Review*, 34:e20, 2019.

[SK16] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. Springer, 2016.

[SK19] Bruno Siciliano and Oussama Khatib. Humanoid robots: historical perspective, overview and scope. *Humanoid robotics: a reference*, pages 3–8, 2019.

[SLR15] Nima Shafii, Nuno Lau, and Luis Paulo Reis. Learning to walk fast: Optimized hip height movement for simulated and real humanoid robots. *Journal of Intelligent & Robotic Systems*, 80(3):555–571, 2015.

[SML+15] 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.

[SPA+13] Max Schwarz, Julio Pastrana, Philipp Allgeuer, Michael Schreiber, Sebastian Schueller, Marcell Missura, and Sven Behnke. Humanoid teensize open platform nimbro-op. In *Robot Soccer World Cup*, pages 568–575. Springer, 2013.

[SPCB17] Isaac J Silva, Danilo H Perico, A Helena Reali Costa, and Reinaldo AC Bianchi. Using reinforcement learning to optimize gait generation parameters of a humanoid robot. *XIII Simpósio Brasileiro de Automaçao Inteligente*, 2017.

[SSB06] Jörg Stückler, Johannes Schwenk, and Sven Behnke. Getting back on two feet: Reliable standing-up routines for a humanoid robot. In *IAS*, pages 676–685, 2006.

[SSS$^+$12] Max Schwarz, Michael Schreiber, Sebastian Schueller, Marcell Missura, and Sven Behnke. NimbRo-OP humanoid teensize open platform. In *Workshop on Humanoid Soccer Robots, International Conference on Humanoid Robots (Humanoids)*. IEEE, 2012.

[SSSR19] Philipp Seiwald, Felix Sygulla, Nora-Sophie Staufenberg, and Daniel Rixen. Quintic spline collocation for real-time biped walking-pattern generation with variable torso height. In *International Conference on Humanoid Robots (Humanoids)*, pages 56–63. IEEE, 2019.

[Ste20] Sebastian Stelter. Creating dynamic stand-up motions for bipedal robots using spline interpolation. Bachelor's thesis, Universität Hamburg, 2020.

[Ste22] Sebastian Stelter. Learning error-corrections for series elastic actuators on humanoid robots. Master's thesis, Universität Hamburg, 2022.

[SV16] Andreas Seekircher and Ubbo Visser. A closed-loop gait for humanoid robots combining LIPM with parameter optimization. In *Robot World Cup*, pages 71–83. Springer, 2016.

[SvD19] Marcus M. Scheunemann and Sander G. van Dijk. ROS 2 for robocup. In *Robot World Cup*, pages 429–438. Springer, 2019.

[SWC$^+$14] Sangok Seok, Albert Wang, Meng Yee Chuah, Dong Jin Hyun, Jongwoo Lee, David M Otten, Jeffrey H Lang, and Sangbae Kim. Design principles for energy-efficient legged locomotion and implementation on the MIT cheetah robot. *IEEE/ASME Transactions on Mechatronics*, 20(3):1117–1129, 2014.

[SWD$^+$17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[SZC$^+$02] Manny Soltero, Jing Zhang, Chris Cockrill, et al. 422 and 485 standards overview and system configurations. *Texas Instruments Application Report*, pages 1–33, 2002.

[TP11] Kok Kiong Tan and Andi Sudjana Putra. *Drives and control for industrial automation*. Springer Science & Business Media, 2011.

[TTT$^+$89] A Takanishi, M Tochizawa, T Takeya, H Karaki, and I Kato. Realization of dynamic biped walking stabilized with trunk motion under known external force. In *Advanced Robotics: 1989*, pages 299–310. Springer, 1989.

[TZS03] Zhe Tang, Changjiu Zhou, and Zenqi Sun. Trajectory planning for smooth transition of a biped robot. In *International Conference on Robotics and Automation*, volume 2, pages 2455–2460. IEEE, 2003.

[UARM22] Karthik Urs, Challen Enninful Adu, Elliott J Rouse, and Talia Y Moore. Design and characterization of 3d printed, open-source actuators for legged locomotion. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 1957–1964. IEEE, 2022.

[Urb16] Gerald Urban. Jacob fraden: Handbook of modern sensors: physics, designs, and applications, 2016.

[vdHLF$^+$22] Bas van der Heijden, Jelle Luijkx, Laura Ferranti, Jens Kober, and Robert Babuska. Eagerx: Engine agnostic graph environments for robotics. *GitHub repository*, 2022.

[VDX15] Roberto G Valenti, Ivan Dryanovski, and Jizhong Xiao. Keeping a good attitude: A quaternion-based orientation filter for imus and margs. *Sensors*, 15(8):19302–19330, 2015.

[VGBZ21] Florian Vahl, Jan Gutsche, Marc Bestmann, and Jianwei Zhang. YOEO – you only encode once: A CNN for embedded object detection and semantic segmentation. In *2021 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 619–624. IEEE, 2021.

[VS72] Miomir Vukobratović and Juri Stepanenko. On the stability of anthropomorphic systems. *Mathematical biosciences*, 15(1-2):1–37, 1972.

[WHFZ18] Florens Wasserfall, Norman Hendrich, Fiedler Fiedler, and Jianwei Zhang. 3d-printed low-cost modular force sensors. In *International Conference on Climbing and Walking Robots (CLAWAR)*, pages 485–492. World Scientific, 2018.

[WYC$^+$21] Tong Wu, Zhangguo Yu, Xuechao Chen, Chencheng Dong, Zhifa Gao, and Qiang Huang. Falling prediction based on machine learning for biped robots. *Journal of Intelligent & Robotic Systems*, 103(4):1–14, 2021.

[XBC$^+$18] Zhaoming Xie, Glen Berseth, Patrick Clary, Jonathan Hurst, and Michiel van de Panne. Feedback control for cassie with deep reinforcement learning. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 1241–1246. IEEE, 2018.

[YKL17] Chuanyu Yang, Taku Komura, and Zhibin Li. Emergence of human-comparable balancing behaviours by deep reinforcement learning. In *International Conference on Humanoid Robotics (Humanoids)*, pages 372–377. IEEE, 2017.

[YMV$^+$15] Seung-Joon Yi, Stephen G McGill, Larry Vadakedathu, Qin He, Inyong Ha, Jeakweon Han, Hyunjong Song, Michael Rouleau, Byoung-Tak Zhang, Dennis Hong, et al. Team thor's entry in the darpa robotics challenge trials 2013. *Journal of Field Robotics*, 32(3):315–335, 2015.

[YYH$^+$20] Chuanyu Yang, Kai Yuan, Shuai Heng, Taku Komura, and Zhibin Li. Learning natural locomotion behaviors for humanoid robots using human bias. *IEEE Robotics and Automation Letters*, 5(2):2610–2617, 2020.

[YYM$^+$18] Chuanyu Yang, Kai Yuan, Wolfgang Merkt, Taku Komura, Sethu Vijayakumar, and Zhibin Li. Learning whole-body motor skills for humanoids. In *International Conference on Humanoid Robots (Humanoids)*, pages 270–276. IEEE, 2018.

[Zad88] Lotfi A. Zadeh. Fuzzy logic. *Computer*, 21(4):83–93, 1988.

[ZBL$^+$19] Yi Zhou, Connelly Barnes, Jingwan Lu, Jimei Yang, and Hao Li. On the continuity of rotation representations in neural networks. In *Conference on Computer Vision and Pattern Recognition*, pages 5745–5753. IEEE, 2019.

[ZJF$^+$22] Weiyi Zhang, Yancao Jiang, Fasih Ud Din Farrukh, Chun Zhang, Debing Zhang, and Guangqi Wang. Lorm: a novel reinforcement learning framework for biped gait control. *PeerJ Computer Science*, 8:e927, 2022.

[ZN$^+$42] John G. Ziegler, Nathaniel B. Nichols, et al. Optimum settings for automatic controllers. *trans. ASME*, 64(11), 1942.

[ZQW20] Wenshuai Zhao, Jorge Peña Queralta, and Tomi Westerlund. Sim-to-real transfer in deep reinforcement learning for robotics: a survey. In *2020 IEEE symposium series on computational intelligence (SSCI)*, pages 737–744. IEEE, 2020.

Online Sources

[1] Additional online material for this thesis. https://sammyramone.github.io/phd_thesis. accessed 03.01.2023.

[2] Citbrains website. https://citbrains.studio.site/1-1. accessed 31.08.2022.

[3] Gankenkun webpage. https://github.com/citbrains/OpenPlatform. accessed 23.08.2022.

[4] Karen robot. https://www.youtube.com/watch?v=lILpHR8pccg&feature=emb_imp_woyt. accessed 09.12.2022.

[5] Kondo servo webpage. kondo-robot.com. accessed 23.08.2022.

[6] Model specification hlvs. https://cdn.robocup.org/hl/wp/2021/06/v-hsc_model_specification_v1.05.pdf. accessed 27.11.2022.

[7] Moveit setup assistant. https://github.com/ros-planning/moveit2/tree/main/moveit_setup_assistant. accessed 27.11.2022.

[8] Nav2 navigation stack. https://navigation.ros.org. accessed 20.12.2022.

[9] Nugus hardware overview. https://nubook.nubots.net/system/hardware/overview. accessed 18.05.2022.

[10] Onshape to robot. https://github.com/Rhoban/onshape-to-robot. accessed 27.11.2022.

[11] Onshape website. www.onshape.com. accessed 03.01.2023.

[12] Openmanipulator-x manual. https://emanual.robotis.com/docs/en/platform/openmanipulator_x/overview/. accessed 17.01.2022.

[13] Orocos kinematics and dynamics. ,http://www.orocos.org. Accessed: 2020-10-14.

[14] Pueue. https://github.com/Nukesor/pueue. accessed 03.01.2023.

[15] Pybullet webpage. http://pybullet.org. accessed 27.11.2022.

[16] Rep-120. https://www.ros.org/reps/rep-0120.html. accessed 20.12.2022.

[17] Robocup humanoid webpage. https://humanoid.robocup.org/. accessed 18.05.2022.

[18] Robocup team description papers. https://tdp.robocup.org/tdp/. accessed 20.12.2022.

[19] Ros control toolbox. https://github.com/ros-controls/control_toolbox. accessed 20.12.2022.

[20] Ros documentation. https://docs.ros.org/en/rolling/Concepts/About-Quality-of-Service-Settings.html. accessed 20.03.2023.

[21] Running robot competition. http://www.running-robot.net/. accessed 03.01.2023.

[22] Sustaina-op webpage. https://github.com/citbrains/SUSTAINA-OP. accessed 23.08.2022.

[23] Urdf2webots. https://github.com/cyberbotics/urdf2webots. accessed 03.01.2023.

[24] Robot Specifications RoboCup 2022. https://humanoid.robocup.org/hl-2016/teams/, 2016. accessed 09.12.2022.

[25] Robot Specifications RoboCup 2022. https://humanoid.robocup.org/hl-2022/teams/, 2022. accessed 09.12.2022.

[26] Robotis e-Manual. http://emanual.robotis.com/, 2022. accessed 21.04.2022.

[27] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[28] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 — seamless operability between c++11 and python, 2017. https://github.com/pybind/pybind11.

[29] Antonin Raffin. Rl baselines3 zoo. https://github.com/DLR-RM/rl-baselines3-zoo, 2020.

[30] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable baselines3. https://github.com/DLR-RM/stable-baselines3, 2019.

[31] Humanoid League TC. Roadmap from 2022 to 2050. http://humanoid.robocup.org/wp-content/uploads/roadmap_2022.pdf, 2022. accessed 13.10.2022.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrift selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe.
Außerdem versichere ich, dass dieses Exemplar der Dissertation, das elektronisch eingereichte Dissertationsexemplar (über den Docata-Upload), so wie das zur Archivierung eingereichte gebundene Exemplar und das Exemplar auf dem der gebundenen Fassung beiliegendem elektronischen Speichermedium identisch sind.

I hereby declare upon oath that I have written the present dissertation independently and have not used further resources and aids than those stated in the dissertation.
I also declare that I did not submit this dissertation in any other examination procedure before.
Furthermore, I declare that this copy of the dissertation is identical to the one submitted in electronic form (via the Docata upload), as well as to the archived bound copy of the dissertation and the one on the electronic storage medium attached to the bound copy.

Hamburg, 20.03.2023