Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Analysis of State-of-the-Art Quadruped Locomotion Controllers for Use in Simulation

Jakub Jon

Supervisor: Mgr. Martin Pecka, PhD.
May 2024

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Jon Jakub** Personal ID number: **507665**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Cybernetics and Robotics**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Analysis of State-of-the-Art Quadruped Locomotion Controllers for Use in Simulation

Bachelor’s thesis title in Czech:

Analýza pokročilých řídicích algoritmů pro chůzi čtyřnohých robotů se zaměřením na užití v simulaci

Guidelines:

1. The goal of the thesis is to compare several state-of-the-art quadruped locomotion controllers implemented in simulation, develop guidelines suggesting which controller is better in which situation, and finally, implementing an example decision strategy that could switch between the controllers based on the current context.
2. Initially, the student should review available literature and implementations of quadruped motion controllers.
3. Next, the student should create necessary applications that would allow running all controllers in the same simulator for fair comparison. The student should as well design appropriate simulated scenarios which will be used for comparing the controllers.
4. The student should propose and evaluate suitable metrics that will allow choosing the best controller for a given task. A simple user-oriented guideline should be extracted from the results of the comparison.
5. Last, the student should implement a context-aware strategy that will be able to switch between multiple motion controllers to achieve superior performance in the designed metrics in some of the designed scenarios.

Bibliography / sources:

[1] Joonho Lee et al., Learning quadrupedal locomotion over challenging terrain. Sci. Robot. 5, (2020). DOI:10.1126/scirobotics.abc5986
[2] Takahiro Miki et al., Learning robust perceptive locomotion for quadrupedal robots in the wild. Sci. Robot. 7 (2022). DOI:10.1126/scirobotics.abk2822
[3] Ruben Grandia, Perceptive and Dynamic Locomotion through Nonlinear Model Predictive Control. Doctoral Thesis, ETH Zurich (2022).
[4] Fabian Jenelten et al., DTC: Deep Tracking Control. Sci. Robot. 9 (2024). DOI:10.1126/scirobotics.adh5401

Name and workplace of bachelor’s thesis supervisor:

Mgr. Martin Pecka, Ph.D. Vision for Robotics and Autonomous Systems FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **09.02.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

Mgr. Martin Pecka, Ph.D.	prof. Dr. Ing. Jan Kybic	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____ Date of assignment receipt

_____ Student's signature

Acknowledgements

I would like to express my deepest gratitude to my family for their patience and unwavering support throughout this project.

I am also incredibly grateful to my supervisor, Mgr. Martin Pecka, Ph.D., for the insightful weekly discussion sessions and continuous guidance. Additionally, I extend my thanks to the Department of Cybernetics for providing access to a DGX computer with its abundant computational resources, which were essential for conducting my deep-learning experiments.

Thank you all for your support.

Declaration

I hereby declare that I wrote this thesis on my own and that I cited all the used information sources.

In Prague, 24. May 2024

Abstract

The field of robotics has experienced a surge of advanced robots entering the market in recent years. Many robots today can perform tasks that were unimaginable only a few years ago. Among the various types of robots—such as drones, bipedal robots, or hexapods —four-legged robots have gained significant attention due to their ability to navigate complex terrains and perform dynamic movements.

In this work, we explore state-of-the-art methods for controlling quadrupedal robots. Specifically, we focus on implementing four walking controllers for the Anymal D robot, one of the most advanced four-legged machines available. These controllers include a model predictive controller (MPC) in combination with a whole-body tracking controller, a reinforcement-learning-based controller, and two hybrid controllers that combine an MPC controller with a neural-network based tracking controller. We verify the functionality and performance of these controllers in the Gazebo simulator, evaluating their ability to effectively traverse various types of terrains. Finally, we develop a context-aware strategy that dynamically switches between two of the implemented controllers based on the available information.

Keywords: Reinforcement Learning, Model Predictive Control

Supervisor: Mgr. Martin Pecka, PhD.

Abstrakt

Robotika zažívá v posledních letech prudký nástup pokročilých robotů na trhu. Mnoho robotů dnes dokáže vykonávat úkoly, které byly ještě před několika lety nepředstavitelné. Mezi různými typy robotů - jako jsou drony, humanoidi nebo hexapodi - si čtyřnozí roboti získali značnou pozornost díky své schopnosti pohybovat se ve složitém terénu a dynamicky se pohybovat.

V této práci zkoumáme nejmodernější metody řízení čtyřnohých robotů. Konkrétně se zaměřujeme na implementaci tří kontrolerů chůze pro robota Anymal D, jednoho z nejpokročilejších čtyřnohých robotů vůbec. Tyto regulátory zahrnují MPC kontrolér v kombinaci s WBC regulátorem, dálé potom dva regulátory založené na posilovaném učení a hybridních regulátorech, který kombinují prediktivní kontrolér s tracking regulátorem založeným na neuronových sítích. Funkčnost těchto regulátorů ověřujeme v simulátoru Gazebo a hodnotíme jejich schopnost efektivně překonávat různé typy terénů. Nakonec vyvíjíme strategii zohledňující kontext, která dynamicky přepíná mezi dvěma implementovanými regulátory na základě dostupných informací.

Klíčová slova: Posilované Učení, Prediktivní řízení

Překlad názvu: Analýza pokročilých řídicích algoritmů pro chůzi čtyřnohých robotů se zaměřením na užití v simulaci

Contents

Figures

Tables

Chapter 1

Introduction

The field of robotics has experienced a surge of new robots entering the market over the last couple of years. Advancements have been made both on the hardware and software side of things. Many robots these days can perform stunts that were unimaginable only a few years back. There are different types of robots, including drones, bipedal robots, quadrupedal robots or hexapods, to name a few.

In this work, we will dive deep into some of the methods used to control one type of these machines, namely the four-legged machines, implement three different walking controllers, and verify their functionality in simulation.

1.1 Literature review

People have been building quadrupedal robots for a few decades now. Four-legged machines of various shapes and sizes are being used for diverse industry applications these days, ranging from simple inspection routines inside manufacturing plants [1] to last-mile deliveries to homes in many cities [18]. Consequently, there are many robots available to perform testing on. In this work, however, we will be using the Anymal robot [8], one of the most advanced four-legged machines in the world at the time of writing this thesis.

(a) : Anymal B [3] **(b) :** Anymal D[2]

Figure 1.1: Anymal and two of its iterations

Anymal has twelve actuated joints, and because it is a floating-base robot, it has an additional six unactuated degrees of freedom. Three of these define

the robot's position with respect to a user-defined fixed frame, and the last three define the robot's orientation to this frame. These six unactuated degrees of freedom make Anymal an underactuated robot [19], as they cannot be driven independently of the 12 actuated joints.

Because of the underactuation and the high number of DOFs, performing highly dynamic motions is challenging, and only applying PD regulators to control the entire system will not suffice.

A relatively novel way of controlling Anymal was introduced in [4], where Hutter et al. introduced, among other ideas, a whole-body controller built on top of hierarchical quadratic programming.

Whole-body controllers generate control signals solely based on the current state and thus do not consider the future when optimizing the next command to be sent to the joint actuators. While these controllers are incredibly useful for controlling highly dynamic robots, they are usually not used by themselves, as performing dynamic movements requires the anticipation of future contacts of the machine with the environment. This is where model-predictive controllers come into play. MPC controllers are, in our settings, reference trajectory generators for the WBC controllers. MPCs usually plan hundreds of milliseconds into the future and produce reference trajectories for the WBCs to track. This planning into the future comes at a cost, which is smaller update frequencies. Generally, neither of the two controllers mentioned above are used alone. It is the combination of the WBC's high update frequency and the MPC's future planning that makes this duo so powerful. The paper [17] succinctly describes how an MPC controller for four-legged robots can be formulated as an optimization problem and summarises centroidal dynamics, an efficient way of describing the motion of the robot's center of mass.

To safely and efficiently navigate the environment while performing a task, legged robots generally need to use as much information as possible about their state and local terrain. In [5], Fankhauser et al. proposed a method to generate terrain maps in a Kalman-filter fashion, fusing state estimates, point clouds from onboard depth cameras, and terrain map models from the previous timestep. After its creation, a local terrain map can be further analyzed, and many useful quantities can be computed, such as traversability scores, signed distance fields, or surface normals.

Local terrain maps, or gridmaps, can be used for various purposes, and the community has applied them in many successful projects. One such project is TAMOLS [9], which uses quantities computed from gridmaps to formulate terrain-aware trajectory optimization problems for legged machines. The paper itself proposed many new ways of thinking about controlling quadrupedal robots. There are quantities that MPCs cannot optimize over, as including them in the MPC's optimization problem formulation would lead to orders of magnitude longer solve times. This paper proposed several ways of pre-computing quantities necessary for generating naturally looking gaits, including base position and base orientation with respect to the ground and introduced an MPC formulation that takes these pre-computed quantities

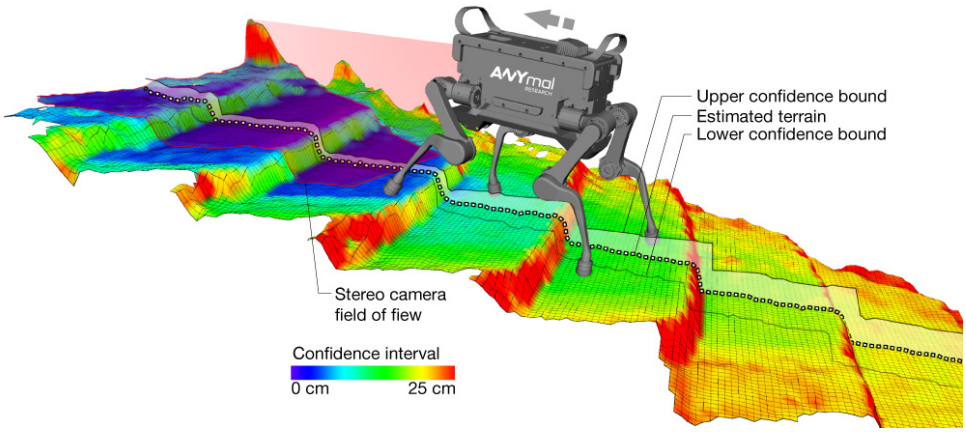


Figure 1.2: Gridmap [5]

into account.

In a follow-up project [7], Hutter et al. introduced the notion of a perceptive nonlinear MPC. The biggest drawback of the work presented in [9] is that the utilized MPC assumed the single rigid body dynamics (SRBD). For highly dynamic motions, however, this assumption is not justified, as the leg dynamics play a significant role in the evolution of the robot's state.

Therefore, the authors used centroidal dynamics instead. Moreover, the authors claimed that unintended collisions with the environment reduced the overall performance of the MPC. Their solution to this problem was incorporating a signed distance field into the MPC formulation. More specifically, the SDF was used in the formulation of a soft constraint incetivizing some of the robot's links to stay away from ground.

Finally, a better method for enforcing footholds had been sought after. The authors used a method combining plane segmentation, convex region extraction, and foothold inequality constraints.

So far, all the papers discussed have only used traditional means of controlling quadrupedal robots, specifically, an MPC controller whose generated trajectories are tracked using a low-level tracking controller, generally some variant of the WBC controller. In the last couple of years, however, researches have started experimenting with deploying neural networks to control their four-legged machines. A key paper on the topic of controlling four-legged robots with neural networks was [12], which proposed a blind walking policy fully trained in simulation. The paper itself contributed to the research community by describing important details pertaining both training and deployment.

Building upon the work introduced in [12], another publication,[14], enhanced the walking policy by giving the neural network the ability to see the local environment by retrieving the terrain heights around the robot's feet and using that information as an input to the underlying neural network. The majority of training details stayed the same, though a new way of policy distillation was introduced in this paper, removing the previously used CNN and using an RNN instead.

When training reinforcement learning agents, using vast quantities of data is paramount. It used to be very difficult to simulate hundreds of robots in parallel and perform all the training efficiently in the past. That changed in 2021 when Nvidia introduced its Isaac Gym framework, a physics simulator fully implemented on GPU, aimed specifically to tackle many reinforcement learning applications [13].

The authors of [12] and [14] took advantage of this simulator's capabilities and built a gym environment [1] for training quadrupedal robots on top of it. Their work was summarized in [15] where they described and demonstrated their claim that four-legged machines could be taught to walk in mere minutes instead of hours or days, which had been the norm prior to this work.

Traditional control policies with MPCs exhibit great planning capabilities. However, regular assumption violations during deployment in the real world reduce their robustness. Reinforcement-learning-based controllers, on the other hand, usually generate robust walking motions. Nevertheless, in situations where valid footholds are sparse, RL policies usually struggle, and their performance is mostly inferior to that of MPC-based controllers.

In [10], Hutter et al. proposed a hybrid control architecture that combined the merits of both worlds. The proposed control architecture is essentially the same as that of an MPC-based controller with a WBC, except the WBC is superseded with a neural network.

It is important to acknowledge that the papers discussed above represent only a small fraction of the ongoing advancements in the field of quadrupedal robotics. There are numerous other significant contributions that have not been mentioned here, though their role in the development of four-legged robotic systems is just as important. We only picked publications whose influence on this work was the greatest.

[1] https://github.com/leggedrobotics/legged_gym

Chapter 2

Implementation details

This chapter will first give a high-level overview of the project, introducing important pieces and notions, including that of a `Controller`, `CentralController`, `JointController`, `StatePublisher`, and `StateSubcriber`. These are namely the main building blocks on top of which the walking controllers introduced and discussed in the previous chapter are implemented in this work. Subsequently, having a general overview of the basic interactions between the main parts, we will take a closer look at each of the implemented walking strategies. Five different types of controllers have been implemented as part of this work. We have called them the `StaticController`, the `MpcController`, the `BobController`, the `DtcController` and lastly the `JoeController`. The `MpcController` and the `BobController` have their corresponding blind and perceptive variants, depending on whether the exteroceptive information from Anymal's cameras is considered during deployment or not. `DtcController` and `JoeController` were implemented as a perceptive controllers only, unlike the `StaticController` which is completely blind.

All the implementation is open-source and the following chapters are best read while having the code open on the side. As the project is publicly available on Github, improvement suggestions or even contributions are welcome.

- **tbai** - *https://github.com/lnotspotl/tbai*:
  Repository containing main code for controller deployment in Gazebo
- **tbai_isaac** - *https://github.com/lnotspotl/tbai_isaac*
  Repository containing code for training reinforcement learning based controllers - `BobController`, `DtcController` and `JoeController`
- **tbai_bindings** - *https://github.com/lnotspotl/tbai_bindings*
  Repository providing python bindings for the **ocs2**[1]'s MPC implementation for use during training of `DtcController` and `JoeController`

The code, as it was at submission, can be found in the `24.5.2024` branch.

[1] https://leggedrobotics.github.io/ocs2/

2.1 High level overview

Before discussing the introduced building blocks, let us define two data structures used to communicate details about the system's (Anymal's) state and the control signals (joint torques). The first of these data structures is named `RbdState`. Its declaration is in listing 2.1.

```
struct RbdState {
    double rbd_state[36];
    bool contacts[4];
};
```

Listing 2.1: RbdState declaration

`RbdState` consists of two lists, namely `rbd_state` and `contacts`. `rbd_state` defines Anymal's full state and is made of the following parts:

- base link orientation w.r.t the world frame, XYZ euler angles, in rad (3)
- base link position w.r.t the world frame in m (3)
- base link angular velocity expressed in base frame in rad/s (3)
- base link linear velocity expressed in base frame in m/s (3)
- joint angles in rad (12)
- joint velocities in rad/s (12)

`contacts` defines four flags indicating whether Anymal's individual feet are touching the ground or not. Next, we declare `JointCommandArray` in listing 2.2.

```
struct JointCommandArray {
    struct JointCommand {
        string joint_name;
        double desired_position;
        double desired_velocity;
        double kp;
        double kd;
        double torque_ff;
    };
    JointCommand joint_commands[12];
};
```

Listing 2.2: JointCommandArray declaration

As shown, `JointCommandArray` comprises of a list with twelve `JointCommands`. A `JointCommand` is a data structure composed of the following fields:

- `joint_name`: name of the joint to be controlled
- `desired_position`: desired joint position in rad
- `desired_velocity`: desired joint velocity in rad/s

- `kp`: P value for the underlying PD regulator
- `kd`: D value for the underlying PD regulator
- `torque_ff`: feed-forward torque in N·m

Having the `RbdState` and `JointCommandArray` data structures defined, we can now take a look at Figure 2.5, giving an overview of how the project's building blocks are structured and interconnected.

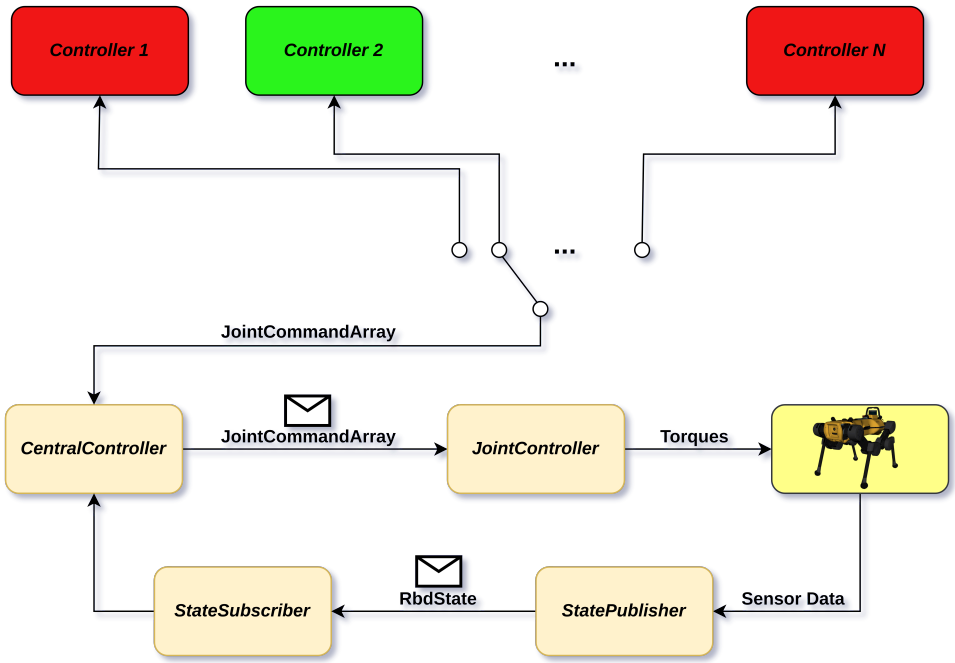


Figure 2.1: System overview

Let us now give a brief overview of the building blocks present in Figure 2.5.

2.1.1 `CentralController`

The entire control system consists of a feed-back loop made out of five blocks, one of which is the Anymal robot itself. The central piece of this feed-back loop is `CentralController`, a building block that communicates with `Controller` instances and retrieves `JointCommandArrays` from them. It also regulates the feed-back loop frequency, depending on the currently active `Controller` instance. At every single point in time, only one `Controller` instance is active. For instance, in Figure 2.5, `Controller` 2 is active, denoted by the green color, and the rest of the `Controller` instances are inactive, indicated by the red color.

2.1.2 `Controller`

The `Controller` class is an abstract class defining a unified interface for all the controller realizations (e.g `MpcController`, `BobController`, etc.) to

conform to. There are seven functions in total that each controller realization needs to implement, as declared in listing 2.3.

```
class Controller {
public:
    virtual JointCommandArray getCommandMessage(double
        currentTime, double dt);
    virtual void visualize();
    virtual bool isSupported(string controllerType);
    virtual void changeController(string controllerType,
        double currentTime);
    virtual void stopController();
    virtual bool checkStability();
    virtual double getRate();
};
```

Listing 2.3: Controller abstract class

Following is a brief explanation of the responsibility for each one of the functions.

- `virtual JointCommandArray getCommandMessage(...)`:
  Produce `JointCommandArray`. The arguments are `currentTime`, which is the number of seconds from a specific point in time defined at program start, and `dt`, time elapsed since the last feed-back loop iteration.

- `virtual void visualize()`:
  Visualize controller-specific data, usually `RbdState` together with other quantities.

- `virtual bool isSupported(...)`:
  When switching between controllers, `CentralController` asks each of the `Controller` instances whether it is of the desired type, represented as a string. `isSupported` can return `true` for two or more distinct inputs. This can be used to introduce aliases or make one `Controller` have two different *modes* of operation, where the *mode* is encoded into the argument. For example, `StaticController`'s `isSupported` method returns `true` for two inputs - `"SIT"` and `"STAND"`.

- `virtual void changeController(...)`:
  Change to this controller. This function is mostly used for initialization tasks when the `CentralController` switches to this controller. Note that if this function is called it is assumed that `isSupported` evaluates to `true`.

- `virtual void stopController()`:
  Perform cleanup tasks when `CentralController` switches to a different controller.

- `virtual bool checkStability() const`:
  Check whether controller is stable. If not, `CentralController` may switch to a fallback controller, usually the `StaticController`.

- `virtual scalar_t getRate() const`:
  Return desired feed-back loop frequency.

2.1.3 JointController

Another important block is the `JointController`. This block unpacks the latest received `JointCommandArray` data structure from the `CentralController` and produces torques to be set for each of the joints. The torque value for every joint is internally computed as follows:

$$\tau = \text{clip}(\tau_{\text{ff}} + k_p \cdot (\theta_d - \theta) + k_d \cdot (\dot{\theta_d} - \dot{\theta}) \mid \tau_{\min}, \tau_{\max}) \tag{2.1}$$

with clip defined as

$$\text{clip}(a \mid b, c) = \max\{\min\{a, c\}, b\} \tag{2.2}$$

In equation 2.1 for calculating the joint torque, τ is the actual torque applied, τ_{ff} is set to `torque_ff`, k_p and k_d evaluate to `kp` and `kd` respectively. We substitute `position_desired` for the symbol θ_d and `velocity_desired` for $\dot{\theta}_d$. The symbols θ and $\dot{\theta}_d$ are replaced with the numerical values of the current joint position and joint velocity respectively. $\tau_{\min}$ and $\tau_{\max}$ represent the minimum and maximum possible torques and are defined as $\tau_{\min} = -$`limit` and $\tau_{\max} = +$`limit` with `limit` ≥ 0 being an effor limit specified in Anymal's *URDF* description.

2.1.4 StatePublisher

The `StatePublisher` block is a piece that directly communicates with the simulator and produces `RbdState` instances. As of writing this work, there are two distint ways how the `RbdState` instance gets populated with numerical values. Either, we assume full observability and the data is taken directly from the simulator, or partial observability as assumed and an extended Kalman filter is utilized to produce the observation. The second of the two options is still experimental as of writing this thesis.

2.1.5 StateSubscriber

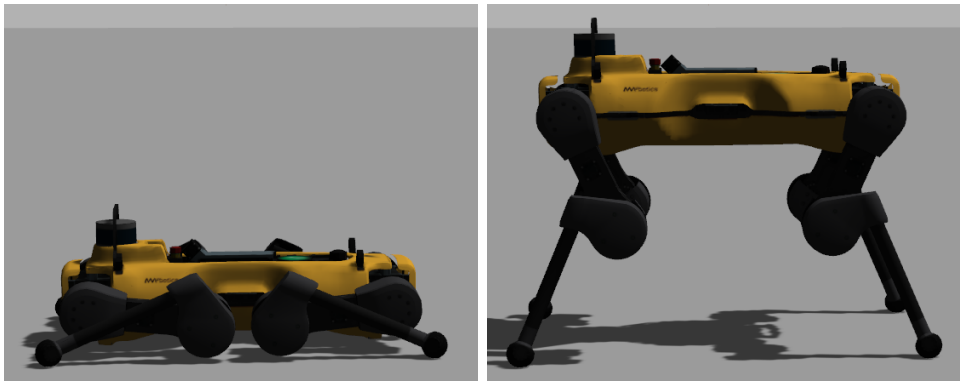
The `StateSubscriber`'s job is to store the latest `RbdState` instance produced by the `StatePublisher` and perform a few preprocessing steps before handing the information about the robot's state and contact flags to the active controller. `StateSubscriber` does no more than that.

2.2 Implemented controllers

2.2.1 StaticController

The simplest of the implemented controllers is the `StaticController`. Internally, it stores the *sit* joint angles as well as the *stand* joint angles and when

switched to, the controller linearly interpolates from the current joint position to the requested joint position over a configurable period of time. Both the `torque_ff` and the `velocity_desired` values are set to zero for each of the `JointCommand` instances. The parameters `kp` and `kd` are set to high values.



(a) : Anymal D sitting **(b) :** Anymal D standing

Figure 2.2: Stand and sit joint angles

2.2.2 `MpcController`

`StaticController` was the simplest of the four implemented controllers. However, it cannot produce the walking motions needed for our quadruped to move around in space. That is where `MpcController` comes into the picture. Even though there's only the MPC part in its name, it is a controller combining an MPC controller with a WBC tracking controller, as shown in Figure 2.3.

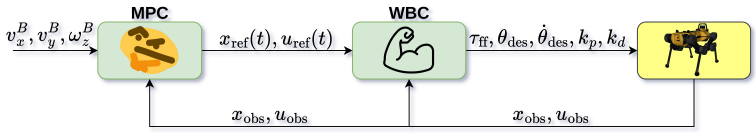


Figure 2.3: Overview of `MpcController`'s architecture

The controller's input is a reference linear velocity v_x^B and v_y^B for the base as well as a reference yaw rate ω_z^B, each expressed in the robot's base frame. Internally, based on the reference velocities, the quadruped's state is extrapolated using the Euler integration method. The generated trajectory serves as a reference for an internal MPC controller, whose main purpose is to fill in more details to the trajectory, taking dynamics, constraints and a cost function into account. An example of a reference trajectory produced by the MPC controller is depicted in Figure 2.4.

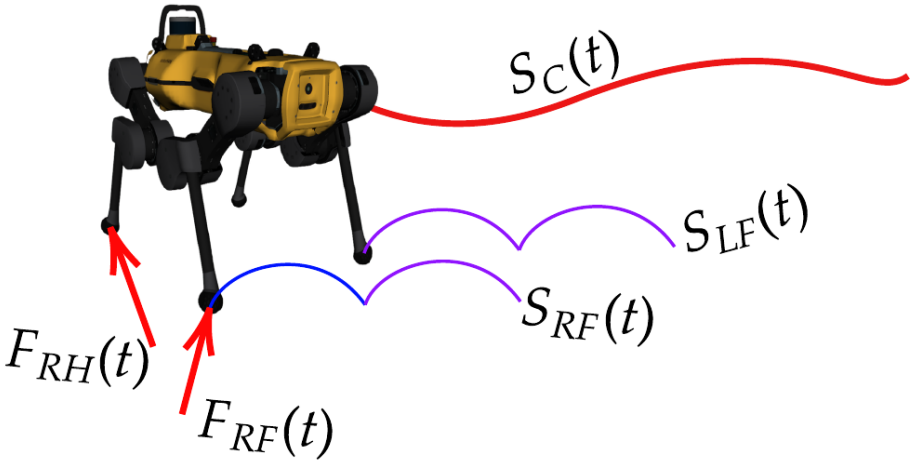


Figure 2.4: Reference trajectory produced by the MPC controller: $S_C(t)$ is an optimized trajectory describing the robot's base link motion, $F_{(.)}(t)$ is an optimized ground reaction force, RF is an abbreviation for *right-front*, $S_{RF}(t)$ is a trajectory describing the motion of the *right-front* foot. Note that not all trajectories produced by the MPC are shown in this image.

The produced trajectory, which is a continuous function of time, is passed to the WBC tracking controller that, based on the current time and state, produces joint torques that are to be set for each of the joint controllers.

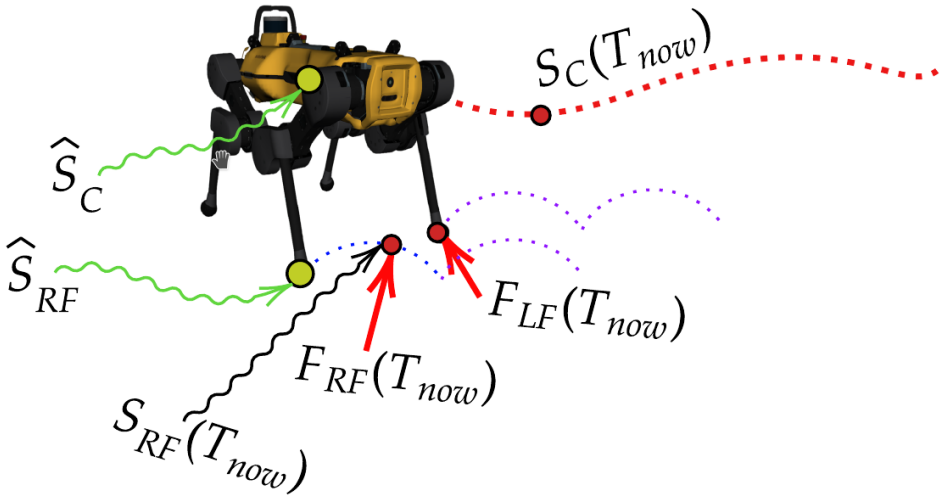


Figure 2.5: The WBC tracking controller is tasked to track the optimized state trajectory $S_{(.)}(t)$, taking the current state $\hat{S}_{(.)}$ and optimized ground reaction forces into account.

Trajectories described by continuous functions can be paremeterized in a myriad of different ways. The way trajectories are represented in this project is as follows. We sample the trajectory (a function of time) at N+1 different points, denoted as $t_0, t_1, \ldots, t_N$ with $t_i < t_{i+1}$. The sampled values together

with the sampling times are stored in an array. When we later want to evaluate the trajectory at time T, we first find t_i and t_{i+1} with $t_i \leq T \leq t_{i+1}$ and perform a linear interpolation between $S(t_i)$ and $S(t_{i+1})$, where $S(t_i)$ and $S(t_{i+1})$ are the sampled values stored in our array. Extrapolation is not defined. For Ts satisfying $T < t_0$ we simply return $S(t_0)$. The upper bound can be handled in a similar manner.

To build the `MpcController`, we used a modified [2] implementation of an MPC controller designed specifically to control Anymal built on top of the **ocs2** project[3]. The MPC controller is the same as described in [7].

We implemented two different types of a WBC tracking controller, more specifically the *Hierarchical Quadratic Program* (HQP) WBC controller and the *Single Quadratic Program* (SQP) WBC controller, each described in [4] and [7], respectively. As the underlying QP solver, we are using `qpOASES`, a high-performance optimizer implemented specifically for MPC applications [6].

The optimized joint torques are packed into `JointCommand`s with the desired position and velocity set to the optimized values from the MPC controller. The PD regulator constants are set to small numbers, meaning the robot is mostly controlled using the computed torques.

[2] https://github.com/lnotspotl/ocs2/tree/tbai
[3] https://leggedrobotics.github.io/ocs2/

2.2.3 BobController

`BobController` is the name of the controller we gave to the trained policy implemented[4] based on [14]. It is a combination of encoders, decoders, multi-layer perceptrons and recurrent neural networks with proprioceptive and exteroceptive information as an input. The model's output is an action, a vector of twelve joint angles, directly used to control Anymal. A high-level overview of this control architecture is show in Figure 2.6.

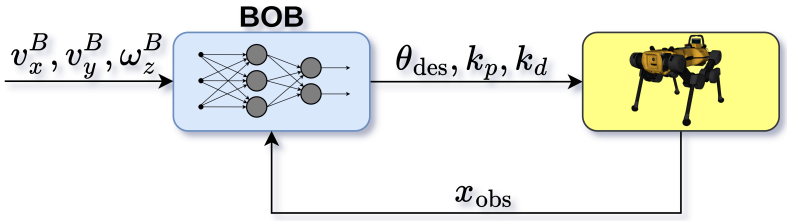


Figure 2.6: Overview of `BobController`'s architecture

We used a PD regulator to track the joint angles produced by the neural network and set the k_p and k_d values to be 80 and 2, respectivelly, as proposed in the original paper [14]. Details about the training process as well a brief description of the inputs to the underlying neural network can be found in [11]. The whole training, a process combining teacher traininig and student policy distillation, takes around ninety minutes on a single DGX A100 station, which is significantly less than the time needed to train the `DtcController` or the `JoeController`.

2.2.4 DtcController

The penultimate controller implemented in this work is the `DtcController`. As stated in section 1.1, the idea for this controller was first proposed in [10]. We took the concepts from the original paper and implemented a controller similar to the one proposed in [10], with a couple of modifications.

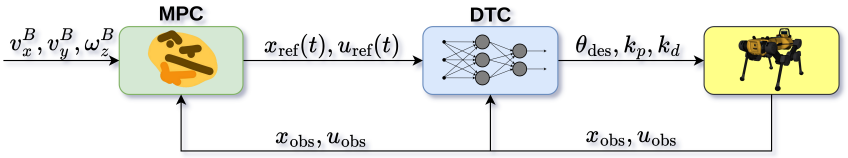


Figure 2.7: Overview of `DtcController`'s architecture

We trained in a simpler environment, the one used in [11], rather than the one described in [10] with the intent that the training would not take fourteen days as reported but just a fraction of that time. With the current

[4] Code for training the policy can be found under this link: https://github.com/lnotspotl/tbai_isaac

implementation, a policy can be trained in just under 4 hours on a single DGX A100 station, though more optimization can be done to reduce the training time even futher.

The trained policy is a simple multi-layer perceptron, whose inputs are briefly summarized in section 2.2.4 and the ouputs are twelve joint angles, just as was the case for `BobController`. These are then tracked using a PD regulator with the k_p and k_d constants set to 80 and 2, respectively.

Input	Dimension
base linear velocity	3
base angular velocity	3
projected gravity	3
command	3
joint angles	12
joint velocities	12
last action	12
planar footholds	8
time left in phase	4
height samples	4x10
CPG information	8

Table 2.1: Observations for `DtcController`'s action neural network

The first 10 observations are succintly described in [10], the *CPG information* is a piece we added with the intent it would help the action neural network reason about which part of the gait cycle it is in at every point in time. This observation was taken from [14].

We are using the PPO algorithm[5] [16] to train our policy. PPO generally uses two distinct neural networks, one which produces actions (joint angles in our case) - *actor* - and one which estimates the value for each possible observation - *critic*. Apart from the observations given as an input to the *actor*, we are also providing the *critic* with information about the optimized trajectory produced by the MPC controller. This additional information is desired base pose, velocity and acceleration, desired joint angles and velocities and finally optimized foot positions and velocities (calculated using forward kinematics). Futhermore, ground reaction forces are given to the *critic* network.

Details about the rewards are given in [10]. We added a couple more rewards terms, namely negative rewards for not following the desired joint angles and velocities as well as the optimized foot positions and velocities. Additionally, to help Anymal stand up at the beginning of training, we added an over time attenuated reward punishing base height deviations from a predifined value.

[5]We are using a modified version of an open-source PPO implementation available at https://github.com/leggedrobotics/rsl_rl

2.2.5 `JoeController`

The final controller we came up with is the `JoeController` inspired by the way the WBC tracking controller works. The inputs to the WBC controller used in `MpcController` are optimized trajectories describing the base pose, velocity and acceleration as well as joint angles and velocities. We decided to take the controller architecture for the NN-based controller introduced in section 2.2.4 and instead of giving the touched upon quantities to the *critic* network, we decided to feed them directly to the *actor* network.

The rest of the details pertaining training stayed the same. The controller architecture is given in Figure 2.8.

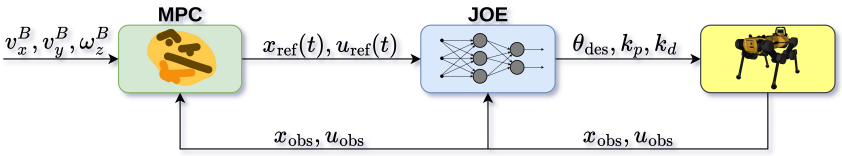


Figure 2.8: Overview of `JoeController`'s architecture

Chapter 3

Controller benchmarking

Benchmarking and comparing two distinct walking controllers is impossible, as the no-free lunch theorem applies to walking policies just as it does to any other type of algorithms. At the end of the day, a walking policy is just an algorithm whose sole purpose is to generate walking motions. There is therefore no universal walking policy that would be better in every possible regard than any other walking algorithm. In this chapter, we will be comparing the implemented walking policies, all described in the previous chapter, for one specific scenario, and thus when we say one policy is better than the other, what we really mean is that this policy is better than the other on the used benchmark we will briefly introduce now.

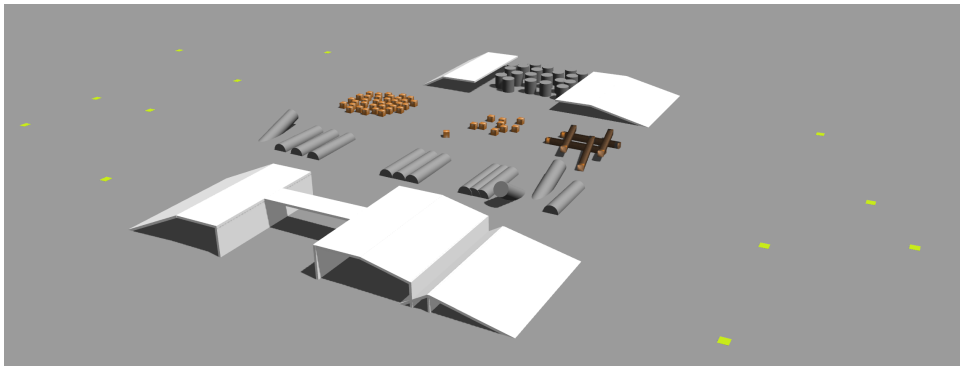
3.1 Benchmark map in Gazebo

We designed a simple obstacle course in Gazebo to test and compare the implemented walking controllers. The entire Gazebo map is depicted in Figure 3.1. The obstacle course consists of multiple stages, each testing the controllers' ability to handle different scenarios.

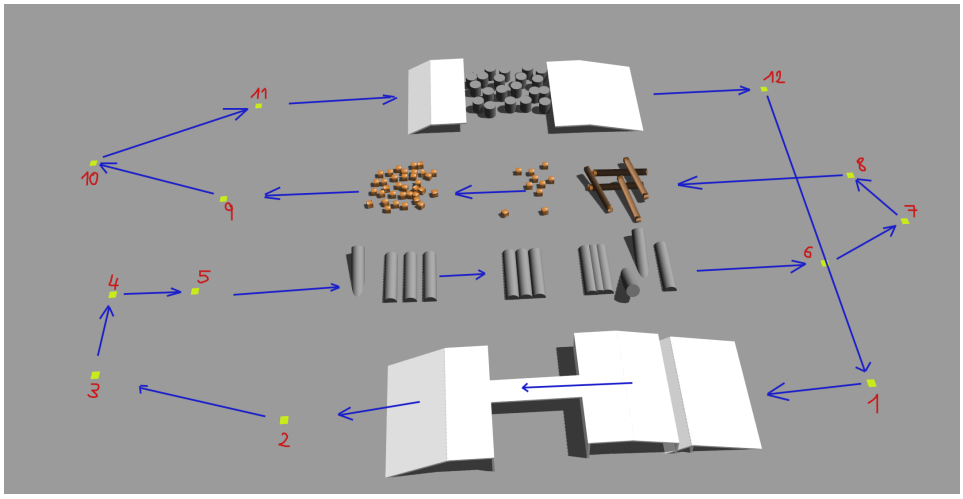
Let us now introduce a simple notation that we will use to discuss the obstacle course. When we write $X \Rightarrow Y$, we refer to the course section in which the robot is supposed to go from waypoint X to waypoint Y.

Some of the sections contain no obstacles whatsoever and only serve the purpose of preparing the robot's pose for traversing the next section, while other sections do contain some obstacles and test our robot's various abilities to cope with different terrains.

Section $1 \Rightarrow 2$ contains a ramp followed by a narrow section. This obstacle tests the controller's ability to reason about its footsteps, as the narrow part cannot be traversed without placing the robot's feet closer together. Section $5 \Rightarrow 6$ contains a couple of stationary cylinders, all scattered around with varying orientations and distances to the nearest other cylinder. When traversing section $8 \Rightarrow 9$, the robot has to go over a pile of wooden logs followed by two regions filled with wooden cubes. Both the cubes and the logs are dynamic obstacles because they are floating body objects, the opposite of static objects. Section $11 \Rightarrow 12$ contains the obligatory stepping stones. No real legged-robot benchmark can lack these. Lastly, section $12 \Rightarrow 1$ is a



(a) : Isometric view



(b) : Top view with annotated waypoints

Figure 3.1: Benchmark map

long stride where the robot's ability to follow the commanded velocity can be tested.

3.2 Reference twist generation

A simple policy was designed to generate twist commands (v_x, v_y and ω_z: desired velocity in X and Y directions and desired yaw rate, all expressed in the robot's base frame) for the robot to follow. A pseudocode for the algorithm is given next.

Algorithm 1 Twist generator for controller benchmarking

Input: simulation state
Output: v_x, v_y, ω_z: all doubles

$x_c \leftarrow$ `currentBasePositionX()`
$y_c \leftarrow$ `currentBasePositionY()`
$\theta_c \leftarrow$ `currentBaseYaw()`
$x_d \leftarrow$ `desiredBasePositionX()`
$y_d \leftarrow$ `desiredBasePositionY()`
$\theta_d \leftarrow$ `atan2`$(y_d - y_c, x_d - x_c)$
$\theta_{\text{diff}} \leftarrow$ `anglediff`(θ_d, θ_c)
$v_x \leftarrow 0$
$v_y \leftarrow 0$
if `abs`$(\theta_{\text{diff}}) \leq \frac{\pi}{3}$ **then**
    $v_x \leftarrow 0.7$
end if
$\omega_z \leftarrow$ `clip`$(0.5 \cdot \theta_{\text{diff}}, -0.7, 0.7)$

The desired base position is set to be a point one meter ahead of the robot on the track in the direction of the next waypoint.

3.3 Experiments

Using the reference velocity generator introduced above, we let each of the implemented controllers autonomously traverse the abstacle course starting at waypoint *1* a moving along the course as indicated by the arrows in Figure 3.1b. For each of the course section involving obstacles, we measure the time it takes for the robot to get from the start to the finish, i.e. from waypoint X to waypoint $X + 1 \mod 12$. Additionally, for the stride $12 \Rightarrow 1$, we also measure the mean joint power, that is the average value of $\sum_{i=1}^{12} \tau_i \cdot \dot{\theta}_i$, where τ_i is the i-th joint torque and $\dot{\theta}_i$ is the i-th joint velocity. We denote this mean value by $\mathrm{P}_{12\Rightarrow 1}$.

3.4 Results

Table 3.1 summarizes the results for experiments presented in the last section.[1]

Controller	1⇒2	5⇒6	8⇒9	11⇒12	12⇒1	$P_{12⇒1}$
`MpcController perceptive`	14.1	15.9	-	15.6	10.9	181.1
`MpcController blind`	-	-	-	-	11.0	186.3
`BobController perceptive`	**14.0**	15.6	17.8	-	9.5	**145.5**
`BobController blind`	-	**15.3**	**16.4**	**15.1**	**9.4**	148.3
`DtcController perceptive`	15.7	18.7	20.1	19.1	11.4	237.6
`JoeController perceptive`	16.9	22.7	-	-	11.0	252.2

Table 3.1: Time taken to traverse course section, given in seconds. The last column is the average joint power in watts on the stride 12⇒1. "-" means the controller failed.

The results given in table 3.1 indicate that the blind `BobController` is usually the fastest while the perceptive `DtcController` and `JoeController` are the slowest. The distance between waypoints 12 and 1 is around 7 meters and therefore, since the commanded velocity is 0.7 m/s, the controllers should ideally take around 10 seconds to get from the starting point to the finish. Both the perceptive and blind `BobController`s are the closest to this quantity and thus we can conclude that on flat terrain, these reinforcement-learning-based controllers can track the commanded velocity the best.

Judging from the recorded videos[1] made while conducting the above experiments, the `BobController`s usually push through and exhibit less visually pleasing movements compared to the perceptive `MpcController` on some parts of the course, especially the two containing the stepping stones and the narrow passage. On those two, the `MpcController` performed better thanks to its ability to plan into the future.

Overall, the `BobController`s work the best in terrains where there are many valid footholds or dynamic and slippery obstacles in the way. `MpcController`s do a great job in cases where planning of future footsteps is paramount, i.e. in scenarios where valid footholds are scarce. On the other hand, when a slip occurs during `MpcController`'s deployment, it often results in the controller's failure. The same goes for the `JoeController` that relies on the underlying MPC controller to a large extent and fails to function when the tracked reference trajectories are inconsistent. The `DtcController` is more robust than `JoeController` in this regard as only desired footholds are extracted from the otpimized trajectories generated by the MPC. These footholds are essential for `DtcController`'s operation and give it a slightly better ability to traverse terrains where valid footholds are mostly lacking while not making

[1] Videos are available here: https://github.com/lnotspotl/tbai/tree/thesis

it too reliant on the MPC controller that can easily diverge in case some unmodeled phenomenom happens.

Lastly, while not being trained for that specifically, our `BobController`s use the least power and are thus more energy efficient than the `MpcController`s, the `DtcController` and the `JoeController`, which ended up losing in this regard.

For benchmarking other types of algorithms on quadrupedal platforms, sometimes the ability to traverse complex terrains is not as important as the walking controller's low computational requirements. Therefore, for each of the implemented controllers, we tested its average CPU utilization. This testing was performed on a machine with the AMD Ryzen 7 5800X 8-Core 3.8 Ghz processor using the `atop` utility. We summarized our findings in table 3.2.

Controller	Average CPU utilization [%]
`MpcController perceptive`	64
`MpcController blind`	64
`BobController perceptive`	265
`BobController blind`	11
`DtcController perceptive`	52
`JoeController perceptive`	53

Table 3.2: Controller CPU utilization. 100 % means full utilization of a single core.

For each controller utilizing an MPC controller, around 45 % CPU utilization is dedicated just for the MPC part (running at 30 Hz). The rest is consumed by other calculations. For MPC-based controller, therefore, the most heavy-lifting takes place on **ocs2**'s end.

Chapter 4

Context-aware controller

Taking the results presented in the last chapter into account, we implemented a simple context-aware strategy that, based on the circumstances, switches between the perceptive `BobController` and the perceptive `MpcController`. The strategy is as follows:

Start with the perceptive `MpcController`. If any foot slip is detected along the way while going across the obstacles, switch to the perceptive `BobController` and use it until the next waypoint has been reached. Upon reaching the next waypoing, switch back to the `MpcController` if it is not the currently active controller.

We detect foot slips using a simple heuristic and the algorithm is given next:

Algorithm 2 Foot slip detection algorithm

Input: feet, simulation state
Output: slip: bool

```
slip ← false
for each foot ∈ feet do
    if foot not in contact then          ▷ RbdState contains contact flags
        continue
    end if
    v_x ← getGlobalVelocityX(foot)
    v_y ← getGlobalVelocityY(foot)
    if v_x^2 + v_y^2 ≥ T then            ▷ T is a tunable threshold value
        slip ← true
        break
    end if
end for
```

The devised strategy was benchmarked[1] in the same way as the individual controllers themselves in the last chapter and the results are once again given in the form of a table, table 4.1.

[1] Videos are available here: https://github.com/lnotspotl/tbai/tree/thesis

Controller	1⇒2	5⇒6	8⇒9	11⇒12	12⇒1	$P_{12\Rightarrow1}$
`MpcController perceptive`	14.1	15.9	-	15.6	10.9	181.1
`MpcController blind`	-	-	-	-	11.0	186.3
`BobController perceptive`	**14.0**	15.6	17.8	-	9.5	**145.5**
`BobController blind`	-	**15.3**	**16.4**	**15.1**	**9.4**	148.3
`DtcController perceptive`	15.7	18.7	20.1	19.1	11.4	237.6
`JoeController perceptive`	16.9	22.7	-	-	11.0	252.2
`Context-aware`	**14.0**	16.1	18.5	16.2	11.0	181.5

Table 4.1: Time taken to traverse course sections, given in seconds. The last column is the average joint power in watts on the stride 12⇒1. The first five rows (gray rows) are copied results from table 3.1 and are included here for easier comparison.

Our `Context-aware` controller was able to get through the entire obstacle course without a single failure. While not being the fastest, reliability might be of more importance in many use-cases. The devised `Context-aware` controller combined the advantages of both the perceptive `MpcController` and the perceptive `BobController`, giving it an ability to better cope with a wider range of terrains.

Chapter 5

Conclusion

In this thesis, we explored, analyzed and tested state-of-the-art quadruped locomotion controllers implemented for the Anymal D robot within the Gazebo simulator. Four primary controllers were implemented: the `MpcController`, combining model predictive control with a whole body tracking controller; the `BobController`, a reinforcement learning-based control strategy; and finally `DtcController` with `JoeController`, two hybrid architectures merging an MPC controller with a neural network-based tracking controller.

Each of the controllers was benchmarked on a custom-built obstacle course designed to test various aspects of quadrupedal locomotion. The results showed that:

- the `MpcController` excells in terrains where precise foot placements are paramount, such as narrow passages or stepping stones
- the `BobController` work best in terrains where precise footholds are not important but robustness is, for instance slippery terrains or passages with dynamic obstacles
- the `DtcController`, a promising control architecture combining MPC with a neural-network based tracking controller, having part of the robustness exhibited by the `BobController` and part of the foresight inherent in the `MpcController`
- the `JoeController`, an architecture similar in nature to `DtcController` relying more on the optimized trajectories from an MPC controller making it a bit less robust

Taking the benchmark results into account, we created a simple context-aware strategy switching between the perceptive `MpcController` and the perceptive `BobController` dynamically. This strategy was benchmarked and resulted in a superior performance on the obstacle course, enhancing `MpcController`'s robustness by switching to the perceptive `BobController` when foot slipping is detected.

Bibliography

[1] *ANYbotics - autonomous robotic inspection solutions*. 2024. URL: https://www.anybotics.com/ (visited on 05/24/2024).

[2] *ANYmal D walking robot, ready for inspection duties*. URL: https://www.electronicsweekly.com/blogs/gadget-master/robot/anymal-d-walking-robot-ready-inspection-duties-2021-05/ (visited on 05/24/2024).

[3] *ANYmal: a unique quadruped robot conquering harsh environments*. URL: https://researchfeatures.com/anymal-unique-quadruped-robot-conquering-harsh-environments/ (visited on 05/24/2024).

[4] C. Dario Bellicoso et al. "Perception-less terrain adaptation through whole body control and hierarchical optimization". In: *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)*. 2016, pp. 558–564. DOI: 10.1109/HUMANOIDS.2016.7803330.

[5] P. Fankhauser, M. Bloesch, and M. Hutter. "Probabilistic Terrain Mapping for Mobile Robots With Uncertain Localization". In: *IEEE Robotics and Automation Letters* 3.4 (2018), pp. 3019–3026. DOI: 10.1109/LRA.2018.2849506.

[6] H. Ferreau et al. "qpOASES: A parametric active-set algorithm for quadratic programming". In: *Mathematical Programming Computation* 6.4 (2014), pp. 327–363.

[7] R. Grandia et al. "Perceptive Locomotion Through Nonlinear Model-Predictive Control". In: *IEEE Transactions on Robotics* 39.5 (2023), pp. 3402–3421. DOI: 10.1109/TRO.2023.3275384.

[8] M. Hutter et al. "ANYmal - a highly mobile and dynamic quadrupedal robot". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2016, pp. 38–44. DOI: 10.1109/IROS.2016.7758092.

[9] F. Jenelten et al. "TAMOLS: Terrain-Aware Motion Optimization for Legged Systems". In: *IEEE Transactions on Robotics* 38.6 (2022), pp. 3395–3413. DOI: 10.1109/TRO.2022.3186804.

[10] F. Jenelten et al. “DTC: Deep Tracking Control”. In: *Science Robotics* 9.86 (Jan. 2024). ISSN: 2470-9476. URL: http://dx.doi.org/10.1126/scirobotics.adh5401.

[11] J. Jon. *Improving Spot’s Walking Abilities in Simulation*. 2024. URL: https://github.com/lnotspotl/tbai/blob/thesis/RL_project.pdf.

[12] J. Lee et al. “Learning quadrupedal locomotion over challenging terrain”. In: *Science Robotics* 5.47 (Oct. 2020). ISSN: 2470-9476. URL: http://dx.doi.org/10.1126/scirobotics.abc5986.

[13] V. Makoviychuk et al. “Isaac Gym: High Performance GPU Based Physics Simulation For Robot Learning”. In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*. Ed. by J. Vanschoren and S. Yeung. Vol. 1. 2021. URL: https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/28dd2c7955ce926456240b2ff0100bde-Paper-round2.pdf.

[14] T. Miki et al. “Learning robust perceptive locomotion for quadrupedal robots in the wild”. In: *Science Robotics* 7.62 (Jan. 2022). ISSN: 2470-9476. URL: http://dx.doi.org/10.1126/scirobotics.abk2822.

[15] N. Rudin et al. “Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning”. In: *Proceedings of the 5th Conference on Robot Learning*. Ed. by A. Faust, D. Hsu, and G. Neumann. Vol. 164. Proceedings of Machine Learning Research. PMLR, Aug. 2022, pp. 91–100. URL: https://proceedings.mlr.press/v164/rudin22a.html.

[16] J. Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[17] J.-P. Sleiman et al. “A Unified MPC Framework for Whole-Body Dynamic Locomotion and Manipulation”. In: *IEEE Robotics and Automation Letters* 6.3 (2021), pp. 4688–4695. DOI: 10.1109/LRA.2021.3068908.

[18] *Swiss-Mile*. 2024. URL: https://www.swiss-mile.com/ (visited on 05/24/2024).

[19] R. Tedrake. *Underactuated Robotics. Algorithms for Walking, Running, Swimming, Flying, and Manipulation*. 2023. URL: https://underactuated.csail.mit.edu.