Dynamic Resonance Frequency Identification for Energy-Efficient Movement of Legged Microbots

By

Luca Russo
B.S., Politecnico di Torino , 2022

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2024

Chicago, Illinois

Defense Committee:
Prof. Amit Ranjan Trivedi, Chair and Advisor
Prof. Pranav Bhounsule
Prof. Marcello Chiaberge, Politecnico di Torino

Copyright by

Luca Russo

2024

To my family and to my grandpa, my life example.

ACKNOWLEDGMENT

I am incredibly grateful to my MS advisor, Professor Amit Ranjan Trivedi, for entrusting me with the captivating microbot project. His unwavering support, expert guidance, and profound wisdom were instrumental in successfully completing my Master's thesis. I would also like to extend my heartfelt thanks to the distinguished members of my MS committee: Professor Marcello Chiaberge and Professor Pranav Bhounsule. The expert advice, constructive discussions, and thoughtful feedback greatly enriched this journey..

Then, I want to express my deep gratitude to my parents for their unwavering support throughout my life journey. I deeply appreciate the understanding and encouragement they have shown, especially during my decision to study abroad. Thank you for always believing in me and being my pillar of strength. I am truly blessed to have such amazing parents.

To Aurora, thank you for your immense love and support, even though we are miles apart. I am truly blessed to have you as a sister. Keep shining brightly, and I look forward to witnessing your continued success in life!

To my grandparents, for setting such a powerful example of strength and resilience every day of your lives. To my aunt Emilia ('La zi') for the constant emotional and mental support you've given me throughout my life, especially this year. To all my family and the people who make my life better: my uncles Luca, Fabio, Linda, Amalia, Mario, Ornella, Franco, Eralda, Donato and Rosella, Claudio, Daniele, Salvatore, Lorenz, Stefano, Emanuela and Pietro.

ACKNOWLEDGMENT (Continued)

To my friends with whom I shared this incredible life journey that is University: Galle, Alice, Marta, Sam, Lorenzo, Gaia, Ricky, Giò, Vittorio, and Laura. These years wouldn't have been the same without you.

To my wonderful roommates and all the amazing people I crossed paths with while abroad, thank you for being so extraordinary!

Finally, I want to express my gratitude to my best friend, my lover, my source of comfort, my everything, Lisa. I am thankful to you for helping me become the person I am today. You see potential in me that I often overlook, and you always inspire me to aim higher. Your unwavering support has been invaluable throughout this journey, and that's why I want to end by thanking you. Thank you for being so extraordinary.

LR

CONTRIBUTIONS OF AUTHORS

Most of the results and discussions presented in this thesis are taken from an under-review paper we recently submitted to IEEE ICCR 2024, whose authors are: L. Russo, E. Chandler, K. Jayaram, A. R. Trivedi. After all of them had concealed the idea, L. Russo led the investigations and produced the analytical analysis, technical results and write-up. E. Chandler, K. Jayaram, and A. R. Trivedi contributed to the review of the manuscript write-up.

TABLE OF CONTENTS

CHAPTER			PAGE
1	**INTRODUCTION**		1
	1.1	MICROBOTS	1
	1.2	THE mCLARI ROBOT	4
2	**CONTROL LOGIC ALGORITHM**		8
	2.1	PRELIMINARY ANALYSIS	8
	2.1.1	LEGS	8
	2.1.2	OVERALL	9
	2.2	CONTROL LOGIC	10
3	**MICROBOT PHYSICAL SIMULATION**		16
	3.1	STRUCTURE DESIGN	16
	3.1.1	COMPLETE MODEL	17
	3.1.2	FIRST SEMPLIFICATION	20
	3.1.3	FINAL MODEL	21
	3.2	ACTUATOR DESIGN	23
	3.3	ENVIRONMENT DESIGN	27
4	**REINFORCEMENT LEARNING TRAINING**		32
	4.1	REINFORCEMENT LEARNING	32
	4.1.1	OBSERVATION SPACE	34
	4.1.2	ACTION SPACE	37
	4.1.3	AGENT	38
	4.1.4	REWARD FUNCTION	38
	4.2	GYM ENVIRONMENT	40
	4.2.1	**STEP**	40
	4.2.2	**RESET**	41
	4.2.3	RENDER & CLOSE	42
	4.2.4	**COMPUTE_REWARD**	43
	4.2.5	**CHECK_DONE**	43
	4.2.6	OTHERS	44
	4.3	TRAINING	45
	4.3.1	STABLE BASELINE 3	45
	4.3.2	HYPERPARAMETER TUNING	49
5	**RESULTS**		52
	5.1	PHASE ANALYSIS	52

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

5.2 POWER ANALYSIS 54
5.2.1 EFFICIENCY 58
5.3 RL RESULTS 62

6 CLOSING REMARKS AND FUTURE WORK 67
6.1 REMARKS 67
6.2 FUTURE WORKS 68

CITED LITERATURE 69

VITA 73

LIST OF TABLES

TABLE		PAGE
I	COEFFICIENT FOR THE THREE SLIPPERINESS CONDITIONS	31
II	TABLE OF THE RL ALGORITHMS AVAILABLE IN SB3, V IMPLIES THAT THE ALGORITHM SUPPORTS THE SPACE WHILE X DOESN'T. TABLE INSPIRED (1).	46

LIST OF FIGURES

FIGURE		PAGE
1	SCHEMATIC OF THE MCLARI LEG. KABUTZ ET AL.(2), PROCEEDINGS OF THE IEEE, 2023, WITH PERMISSION OF THE AUTHOR. . . .	5
2	VISUAL REPRESENTATION OF THE MORPHING CAPABILITIES OF THE MCLARI ROBOT.KABUTZ ET AL.(2), PROCEEDINGS OF THE IEEE, 2023, WITH PERMISSION OF THE AUTHOR.	7
3	EFFECT OF THE INTRA-LEG PHASES DIFFERENCE ON THE LEG OUTPUT MOVEMENT IN THE LIFT-SWING PLANE. KABUTZ ET AL. (3), PROCEEDINGS OF THE IEEE, 2023, WITH PERMISSION OF THE AUTHOR.	10
4	POSSIBLE GAITS OF THE HAMR MICROBOTS WITH DIFFERENT INTER-LEGS PHASE SHIFTS (4).	11
5	SCHEMATIC OF THE CONTROLLER LOGIC AND SYSTEM MODEL. .	15
6	KINEMATIC CHAIN OF THE MCLARI ROBOT'S LEG, IN YELLOW THE POINTS IN WHICH THE STRUCTURE WAS OPENED AND THE TYPOLOGY OF CONSTRAINTS USED.	17
7	CONNECTIONS BETWEEN THE LEGS OF THE ROBOT, THE FOUR LEGS ARE CONNECTED THROUGH TWO EQUALITY CONSTRAINTS.	19
8	COMPLETE MODEL OF THE ROBOT OBTAINED DIRECTLY FROM THE CAD FILES.	19
9	FIRST SIMPLIFICATION MODEL OF THE MCLARI ROBOT.	20
10	FINAL MODEL OF THE ROBOT.	22
11	BODE PLOT OF THE TRANSMISSION CHAIN OF THE LEG'S ACTUATOR IN THE MCLARI ROBOT. IN THE FIGURE, THE RATIO BETWEEN THE AMPLITUDES OF THE LEG TIP AND THE ACTUATOR.	23
12	FREQUENCY RESPONSE OF THE LIFT ACTUATOR: IN RED THE CONTROL SIGNAL IMPOSED, IN BLUE THE EFFECTIVE MOVEMENT OF THE LEG.	26
13	ROUGH SURFACE WITH HEIGHT FIELD VALUES TAKEN BY A GAUSSIAN DISTRIBUTION $N(0, 0.1)$; ON THE UPPER RIGHT CORNER THE PNG IMAGE USED.	29
14	ROUGH SURFACE WITH HEIGHT FIELD VALUES TAKEN BY A GAUSSIAN DISTRIBUTION $N(0, 0.1)$ SMOOTHED BY A 3X3 MEAN FILTER; ON THE UPPER RIGHT CORNER THE PNG IMAGE USED.	30
15	TRAJECTORIES FOLLOWED BY THE ROBOT DURING 1 MINUTE OF SIMULATION FOR DIFFERENT ANGLE GROUPS AND WITH DIFFERENT FREQUENCIES.	57

LIST OF FIGURES (Continued)

FIGURE		PAGE
16	POWER CONSUMPTION OF THE LIFT AND SWING ACTUATORS IN 1 SECOND OF SIMULATION FOR ANGLE GROUP (0,0,0) AND FREQUENCY 1 HZ	59
17	POWER CONSUMPTION, MEAN VELOCITY AND EFFICIENCY OF ROBOT MOVEMENT FOR DIFFERENT FREQUENCIES WITH FIXED INTER-LEG PHASE SHIFTS TRIPLES OF (0,0,0), (π,π,0) AND (π,0,π). .	61
18	EFFICIENCIES RESULT OF THE TRAINED MODEL ON DIFFERENT ATTEMPTS ON DIFFERENT SURFACES.	66

LIST OF FIGURES (Continued)

FIGURE **PAGE**

SUMMARY

Advances in robotics and manufacturing processes have enabled the development of extremely small robotic devices, even as tiny as a penny. In this domain, legged microrobots (a.k.a. *microbots*) offer numerous potential applications and characteristics to explore. However, controlling such small multi-legged robots presents significant challenges in achieving the desired behavior. Primarily, due to the robot's small size, it can only operate with a tiny battery, therefore, an extremely computationally efficient controller is needed. Tiny robots are also susceptible to damage and control methodologies are needed that also ensure longevity. This thesis work presents a novel approach for creating a control algorithm for a multi-legged system that dynamically identifies and operates a microbot at its resonance frequency of movement. At the resonance frequency, a microbot's leg oscillations achieve maximum amplitude with minimal energy, resulting in optimal locomotion efficiency. After imposing a sinusoidal control for actuating each of the robot's legs, a two-part controller is developed. The controller is then trained using soft actor-critic-based reinforcement learning on a custom model of the mClari microbot. A successful simulation-based demonstration of the learning-based controller is shown by varying the underlying surface (such as wet, soft, etc.) of the microbot where the controller dynamically identifies and switches to the corresponding resonance frequency, achieving efficient and adaptive locomotion.

CHAPTER 1

INTRODUCTION

1.1 MICROBOTS

The field of robotics is constantly inspired by the marvels of Mother Nature. Among the most captivating sources of inspiration is the remarkable world of insects. These extraordinary creatures, with their ubiquitous presence and petite size, possess a remarkable ability to access and execute tasks that larger animals cannot. Take, for instance, the remarkable cooperation and coordination within bee colonies, enabling them to efficiently tackle intricate tasks. It's truly awe-inspiring to witness the limitless potential that nature offers to inspire innovation in robotics.

In today's exciting era, manufacturing processes have achieved remarkable precision, resulting in the creation of incredibly accurate machines. Simultaneously, the shrinking size of microcontrollers is opening doors to the development of small-scale autonomous systems. Furthermore, the continuous advancement of wireless technologies is simplifying the cooperation between devices, making connectivity more seamless and efficient than ever before. With all this in mind, it's evident that research into these small autonomous devices is experiencing remarkable success. Known as "microbots," these tiny robotic systems have recently made their debut, presenting exciting potential to address various challenges. The rapid progress and promising capabilities of these microbots are truly inspiring and offer hope for solving a

wide range of problems. Just picture this: small, bee-like microbots with cameras pinpointing infected plants to stop disease spread in agriculture. Likewise, beetle-like microbots could maneuver through tight spots in industrial plants, like gas pipelines, to detect leaks. In case of earthquakes, ant-shaped microbots could aid in search and rescue efforts by navigating building debris. At home, mosquito-like microbots could keep an eye out for intruders, while also monitoring CO2 levels, humidity and temperature.

Let's dive into the fascinating realm of microbots! These incredible devices are classified according to their mobility and the surfaces they operate on: land, water, air, or a combination of these. For the purpose of this thesis, we'll focus on land-based robots. Let's delve deeper into this category. In terms of movement, microbots mainly fall into two categories: wheeled and legged. A standout example of wheeled microbots is the Jasmine platform (5), which comprises several small robots on wheels and enables synchronized movements with swarms of microbots.

Let's direct our attention to the world of legged robots. There are two primary categories: single-legged and multi-legged systems. Single-legged robots, such as the impressive Salto-1P, harness the reactive force of a single leg to achieve remarkable jumping and movement capabilities (6). On the other hand, multi-legged systems are marvels of complexity, often boasting four, six, or even eight legs. Remarkable examples in this category include the HAMR-VI developed by Harvard University, the ingenious spider-inspired mClari (2) and the innovative cockroach-inspired CRAM (7), both from the University of Colorado Boulder. These microbots, no larger than a penny, blend their small size for accessing confined spaces with agile

legged locomotion, enabling them to navigate diverse terrains with impressive flexibility, thus overcoming the limitations of traditional wheeled locomotion. It's truly thrilling to witness the potential of these robotic innovations! Microbots have enormous potential, but controlling their movement poses significant challenges. Given their reliance on small batteries, conserving power for movement is paramount. Additionally, due to their limited energy capacity, microbots can currently only function with a select few light sensors and actuators. Prioritizing efficiency is key, as adding more power or components would increase the microbot's size and weight. The situation becomes even more intricate when we consider that the energy efficiency of the robots' movement depends on multiple time-varying factors, which can vary between different devices. For instance, these robots typically use piezoelectric actuators, which have inherent frequency-dependent characteristics. Furthermore, the movement of these small robots is achieved through a complex actuation kinematic chain made up of 3D-printed links and joints, where imperfections during manufacturing can greatly impact the frequency response across different devices. Even the slightest variations in density or surface roughness can influence the overall performance of the robot, leading to different behaviors. Moreover, advanced devices like mClari and HAMR-VI have intricately coupled systems, where each leg controls two different degrees of freedom (DoF) - lift and swing - creating a highly interdependent system.

Nevertheless, as of today, there has been significant progress in maximizing the area/energy efficiency of onboard robotic workload processing. Shukla et al. (2021, 2023) (8; 9) demonstrated a novel compute-in-memory approach to remarkably minimize the necessary energy for robotics algorithms such as visual odometry. Additionally, Darabi et al. (2022, 2024) (10; 11; 12)

demonstrated novel computing architectures and technologies to enable robotic reasoning for localization and path-planning under uncertainties. Furthermore, Tayebati et al. (2024), Darabi et al. (2023), and Shylendra et al. (2020) (13; 14; 15) demonstrated ultra-low-power solutions to check the reliability of sensing as well as leveraging generative AI methods to minimize the need for complex sensors such as LiDAR, while Suleiman et al. (2019, 2018) (16; 17) demonstrated chip designs operating within 2 mW power to support a variety of robotic navigation algorithms.

We were determined to create a controller that could effectively tackle the mentioned challenges and confidently regulate the actions of the microbot. Our journey began with a thorough study of the mClari microbot (2), diving into its key features, and then crafting a fitting controller.

1.2 THE mCLARI ROBOT

The mCLARI robot (2), is the evolution of the CLARI (Compliant Legged Articulated Robotic Insect) (3) developed by the University of Colorado Boulder. With a weight of 0.96g and a neutral body length of around 20 mm, the mCLARI robot is a perfect example for studying the behavior of legged microbots.

The robot is made up of four legs, each with two independent movements: lifting and swinging. Two piezoelectric actuators control these movements. The legs are part of a closed kinematic chain that includes links and passive joints. Please consult the diagram presented in figure Figure 1 for a detailed view of the leg's transmission line configuration.

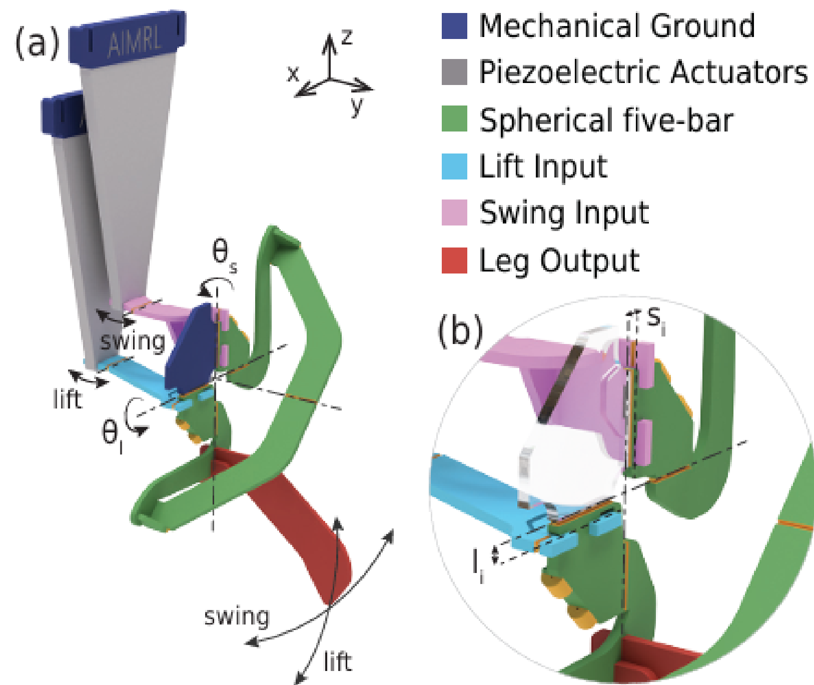


Figure 1: Schematic of the mCLARI leg. Kabutz et al.(2), proceedings of the IEEE, 2023, with permission of the author.

Each piezoelectric actuator is then actuated through a sinusoidal voltage signal of the form:

$$V_i(t) = V_{b,i} \sin(2\pi f_i t + \phi_i). \tag{1.1}$$

The leg movement can be described as follows: different voltages applied to the piezoelectric cause them to move back and forth. This movement is then transmitted through two different transmission lines (shown in light blue and pink in Figure 1). These two lines transmit the movement to both the spherical five-bar linkage (shown in green in Figure 1) and to the body

of the robot itself (shown in blue in Figure 1), which represents the mechanical ground of the system. The five-bar mechanism connects the swing and lift degrees of freedom (DoF) and contributes to the overall movement of the leg output (shown in red in Figure 1). This component exhibits the lift and swing movements and represents the foot of the robot, making actual contact with the ground.

The four legs are connected by flexible joints in a circular arrangement, allowing the robot to stretch, maneuver through tight spaces and achieve different walking patterns and motions. This shape-shifting capability is passive, meaning it requires external forces to stretch the system, but it has a wide range of potential applications. In Figure 2, you can see an example of this behavior.

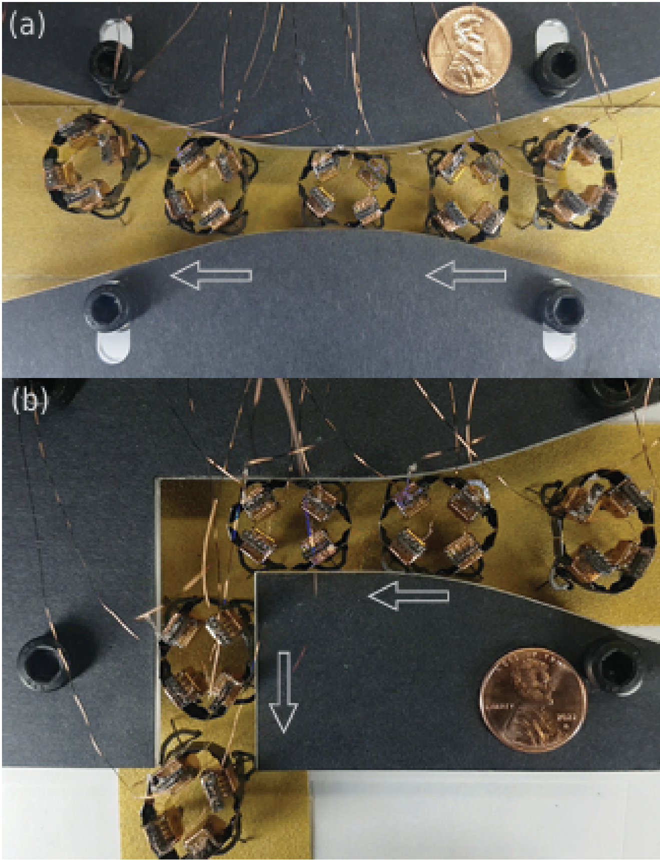


Figure 2: Visual representation of the morphing capabilities of the mClari robot.Kabutz et al.(2), Proceedings of the IEEE, 2023, with permission of the author.

CHAPTER 2

CONTROL LOGIC ALGORITHM

2.1 PRELIMINARY ANALYSIS

After carefully considering the obstacles in creating a controller for these small devices, it's clear that integrating a *physics-informed* control mechanism is the optimal path. This approach will lead to a lightweight, efficient and robust controller, ensuring exciting possibilities for these tiny devices. As mentioned earlier, the mCLARI robot is designed with four independent legs, each identical to the others and all connected through passive joints.

The controller needs to take into consideration two separate levels: the low level, which consists of the motion of the leg itself, with both lift and swing actuation; and a high level, composed of the overall system. Each level has unique characteristics and behaviors, so let's analyze them separately.

2.1.1 LEGS

The leg model of the mCLARI robot is theoretically similar to the one used by both CLARI (Kabutz et al., 2023 (3)) and HAMR (18). Our first step was to review the literature for any characterization of the system's behavior. We found in Kabutz et al. (2023) (3) a characterization of the single leg of the CLARI robot using different frequencies. The paper shows that the robot's leg behavior is highly affected by frequency. Specifically, considering one actuator at a time, the relationship between the actuation input and the resulting movement in the

respective direction follows a behavior similar to a second-order linear damped system. This implies that for each actuator, there exists a certain frequency (**resonance frequency**) that, for the same input used, maximizes the amplitude of the movement. Utilizing movement at this frequency can maximize the efficiency of the robot's movement in terms of the ratio between power consumed and the actual gait performed.

While the above is valid for a single actuator, the system is coupled as the lift and swing actuators are connected and the overall movement is a combination of the two. Unfortunately, we were unable to find references in the literature that describe the behavior of the coupled lift-swing system with respect to frequency. However, since both the lift and swing systems have similar resonance frequencies, we expect a linear behavior outside the resonance region and high non-linearity within the resonance region.

However, we have found that the behavior of the leg tip in the lift-swing plane (xz plane) also depends on the phase-angle difference between the actuation signals. Figure 3 presents an idea of the leg tip's behavior for different phase angles; it can be observed that the overall shape resembles an ellipse and, for phases close to $\pi/2$, a circle.

2.1.2 OVERALL

To understand the behavior of the overall system, we initially delved into the theoretical framework governing legged systems and their ability to switch contact points with the ground, as detailed in (19). Although primarily theoretical, this study facilitated the identification of potential movement pathways through the analysis of contact points.

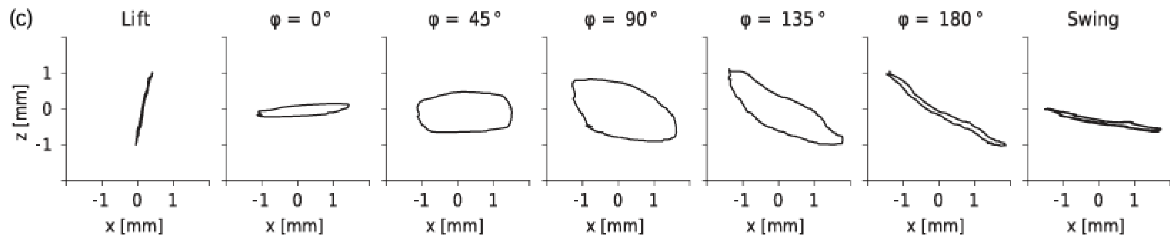


Figure 3: Effect of the intra-leg phases difference on the leg output movement in the lift-swing plane. Kabutz et al. (3), proceedings of the IEEE, 2023, with permission of the author.

However, due to the novelty of mCLARI, a comprehensive analysis of the robot's gait performance is currently unavailable. Nevertheless, findings on a phase control system for the HAMR device were presented in (20). As each piezoelectric actuator is driven by a sinusoidal voltage, the paper suggests exploring the results obtained by adjusting the phase shift between inter-leg angles. Consequently, the system can explore different gait patterns while maintaining a constant frequency and voltage base and manipulating the actuation angles. Figure 4 displays several possible combinations of inter-leg phase angles to achieve diverse movements.

2.2 CONTROL LOGIC

The previous considerations showed two main characteristic of the microbot studied, first of all each leg have an intrinsic frequency-dependent behavior which have to be taken into account and second, controlling the movement of the complete robot by using inter-leg phase differences can be useful to perform several different gaits and movements. Furthermore, the frequency behavior of the overall robot have not really been fully developed while this aspect presents high potential in controlling the robot.

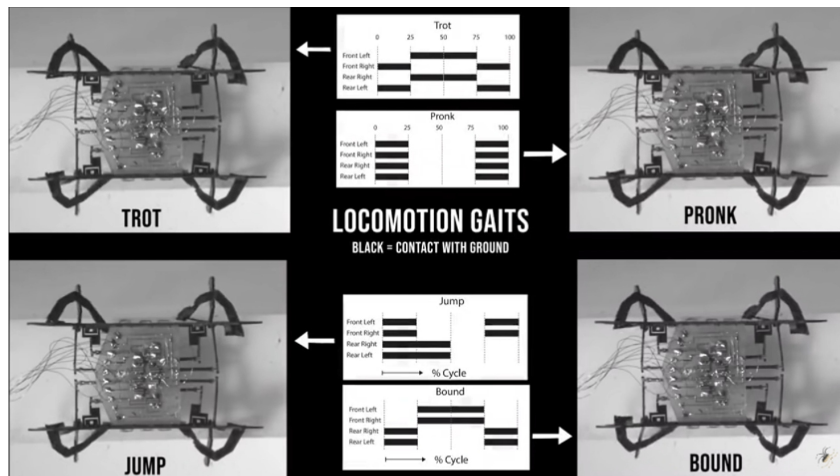


Figure 4: Possible gaits of the HAMR microbots with different inter-legs phase shifts (4).

We, therefore, decided to take into account these two variables: frequency and phase differences. Anyway, here comes the problem: which range of values can we assign to the frequencies and the phase differences and how can we choose them to maximize the efficiency of the system and perform the desired behavior?

To try to solve this problem we started by following a *physic-informed* approach, we took as starting point the actuation signals given to the robot and then, to simplify the controller as more as possible, we tried to reduce the number of controllable variables to the least amount. As stated above, mClari (2) is a four-legged microrobot actuated by eight piezoelectric actuators, each driven by a sinusoidal signal:

$$V_i(t) = V_{b,i} \sin(2\pi f_i t + \phi_i). \tag{2.1}$$

Each actuator is, therefore, controlled by three variables (V_i, f_i, ϕ_i), thus totaling 24 controllable variables. Considering the need for a resource-efficient on-board controller, to simplify this expensive control space, we biased all voltages ($V_{b,i}$) to a constant fixed value across all actuators. Due to robot's symmetry, we also considered a single frequency variable for the entire robot. The reason for that lies in the fact that, inside each leg, the piezoelectric actuators have to work at the same frequency to avoid strange behavior of the system due to the coupling of the lift and swing actuation transmission lines. Now, let's suppose to have four different frequencies, one for each of the legs; this would create a complex and highly unpredictable system to control; that is why, by fixing the frequency to be unique and by working on the phase shift, a more efficient control can be implemented.

Let's now focus on the phase difference variables. We have a total of 8 values that can be controlled: four for the lift actuators and four for the swing. Each leg behaves quite independently from each other and the movement performed by the tip of the output leg with different phase shift between the lift and swing control input is shown in Figure 3. Our idea was to choose a shape from those presented in Figure 3 and use it for all the control signals. We chose to use an intra-leg phase difference of $\pi/2$, as this angle allowed us to have an almost circular shape which is good for moving, being less aggressive when touching the ground. By doing this, we have been able to reduce the number of actual controllable variables to one angle for each leg. This means only four variables are needed to be found, i.e. the main phase angles of each leg. By fixing one of the legs as a reference, three variables are then needed. Even if this number may seem small, when dealing with all the possible combinations of angles, the

number of variables can become incredibly large. We delved into the physics of the system once again. Since these angles represent the phase shift of a sinusoidal input signal, there are some symmetrical properties that can be taken into account. For a microbot like the HAMR, the gaits are performed by coupling the phase shift for the legs; i.e., two pairs of legs move at the same time. The leg pair is what really identifies the difference between the gaits. mCLARI has a similar but different structure and we weren't sure whether this consideration could be useful or not. Therefore, we worked on a more general case study. As mentioned earlier, the required variables for the angles have been reduced to three values. However, each of these values can actually represent an infinite number of values. We needed to find some characteristic angles to take into account. To do so, we used the properties of the sinusoidal function, which is the function used for the actuation signal.

This function is periodic, with a period of 2π, which means that a shift of π between two legs will produce an opposite movement. For example, when one leg is up, the other will be down touching the ground; when one is completely forward, the other will be completely backward and so on. A shift of $\pi/2$, on the other hand, will produce a delay in the movement. For instance, when one leg is touching the ground, the other is completely forward, or when one is forward, the other will be completely lifted. Concerning the angle $3\pi/2$, the behavior will be similar to that of $\pi/2$ but opposite. Any angle between 0 and $\pi/2$ will show a similar but less pronounced behavior in the difference between the legs. Similar considerations are valid for the angles between $\pi/2$ and π. Therefore, it was clear to us to choose just three possible values for references: 0, $\pi/2$ and π. From these three angles, we evaluated all the possible

permutations and found 27 sets of angles. A more detailed analysis of the behavior for each set will be provided in 5. The grouping of phase differences into 27 groups has allowed us to simplify the controlling variable to a single number ranging from 1 to 27. This provides a significant advantage in terms of the computational cost of the controller.

The controller for mClari was designed to achieve three main objectives within a limited control space (consisting of a frequency value and a group angle, both integers between 1 and 150 and 1 and 27, respectively):

i) Following the desired path.

ii) Ensuring longevity by avoiding excessive leg stretches that could cause long-term damage.

iii) Maintaining energy efficiency due to its small battery.

The remaining task was to find an algorithm or controller logic capable of discovering these two values and updating them. The main difficulties stemmed from the highly nonlinear nature of the system and the unpredictable behavior of the microrobot. A traditional controller would not have been sufficiently reliable for this type of system, or the associated computational cost would have been too high.

In our research, we drew inspiration from previous studies on legged robots (Castillo et al., 2022 (21)) and utilized reinforcement learning (RL)-based control. While the structure of our system was based on the aforementioned article, we opted for a different approach. The previous paper introduced a neural network (NN) control system that integrated both endogenous signals from the robot (such as power consumed by actuators, robot velocity and

center of mass position) and exogenous commands given to the robot (like the desired center of mass position), producing output in a reduced control space without clearly defining its physical significance. In our approach, we also considered both endogenous and exogenous signals, but our output was physically meaningful as it represented the two specified values (frequency and inter-leg phase group number). The controller we selected was a deep neural network (DNN)-based reinforcement learning agent.

In previous research on microbots (Duisterhof, 2019 (22)), it was demonstrated that two-layer neural networks (NNs) are effective for learning and adaptation in similar environments. We used a similar NN structure for our controller. More detailed information about the training of the controller can be found in Section 4. However, the reinforcement learning reward function was designed to prioritize both maximizing the efficiency of the robot's movement and reaching the desired target.

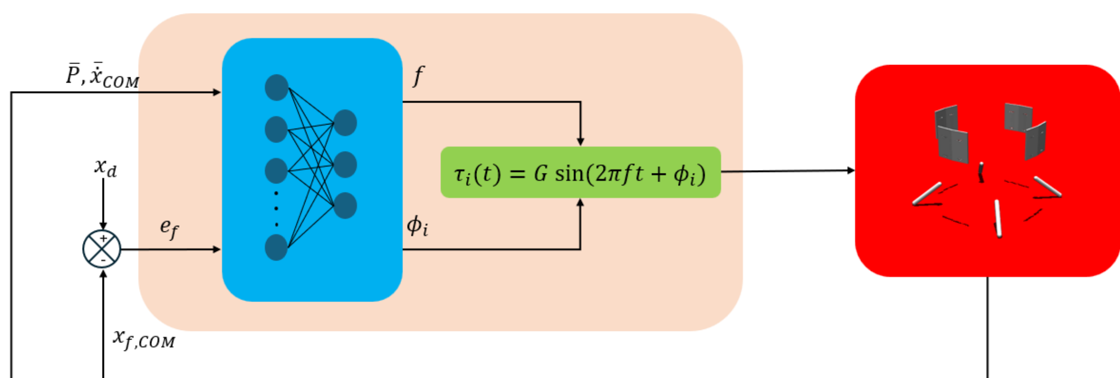


Figure 5: Schematic of the controller logic and system model.

CHAPTER 3

MICROBOT PHYSICAL SIMULATION

3.1 STRUCTURE DESIGN

To work on the robot, we needed an environment that allowed us to recreate the robot's movement and to understand how the environment affected it. We needed a way to actually mimic the response of the environment and a platform to test our control algorithm.

We used the open-source physics engine MuJoCo(23). It is a solver that allows us to create and visualize the overall mechanical system together with the environment in which the robot has to move. It also offers the possibility to implement different types of controllers on the system and observe the behavior of the robot by also providing all the possible variables and sensors that could be useful to actively control the system. Since we wanted to deal with the mClari robot, the first thing we did was take the CAD files of the device. We found them in (24).

We then defined the appropriate system of units. Due to the small dimensions of the robot, we used MMGS (millimeter-gram-second) units instead of the standard MKS (meter-kilogram-second). We did so to avoid computational issues and to be consistent with the dimensions of the CAD files. We, therefore, scaled the gravity acceleration and the dimensions of the environment accordingly.

We developed a three-step process for modeling the system, starting from the overall system and simplifying it step by step.

3.1.1 COMPLETE MODEL

Once taken the CAD files of all the components of the robot's leg; we obtained all the mesh files needed to model both the links and joints. Despite having a complete system, the model encountered issues due to the closed-chain kinematics of each leg. MuJoCo is designed to work with open kinematic chains and closed loops require opening the chains at some points and imposing equivalent constraints. Each mClari's leg has a kinematic chain closed four times, which is why we opened the chain in all the points highlighted in Figure 6.

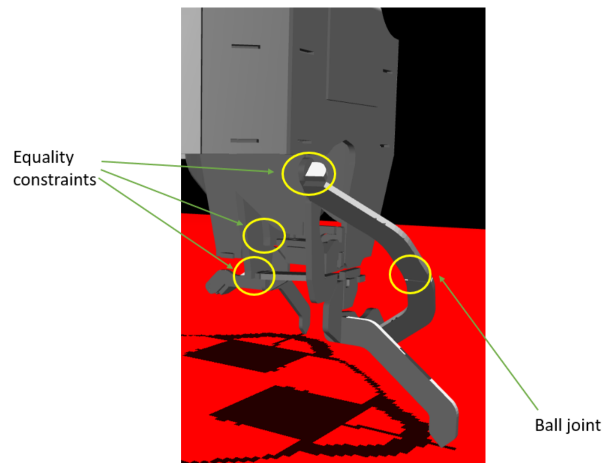


Figure 6: Kinematic chain of the mClari robot's leg, in yellow the points in which the structure was opened and the typology of constraints used.

MuJoCo offers several types of equality constraints, but we focused mostly on two of them:

- **connect**: it is an equality constraint that connects two bodies at one point, it can be useful as a ball joint.
- **weld**: it strongly connects two bodies and removes all the movement degrees of freedom between them.

These constraints are soft, in the sense that they can be violated by the simulation if needed.

Regarding each single leg, the constraints we used were mostly of the type **weld**, in fact, instead of opening the chain at the joints level, we thought that it would have been better to open it in points that were supposed to be strictly connected together. The only **connect** constraint we used was at the junction between the lift and the swing transmission line. In that point, in fact, real experiments on the robot showed that this particular joint behaves actually as a ball joint and the **connect** constraint was therefore the best way to represent it.
For the complete model, we used the same leg model 4 times by rotating each of 90 degrees along the Z axis (the axis normal to the ground) and we joined them together by using two **connect** equality constraints one on the top and the other at the bottom of the connection element as presented in Figure 7. That was done to allow the relative movement between the four legs, as it is in the real robot model. In Figure 8 an image of the overall robot model.

The system created wasn't representative enough of the real robot since the large amount of equality constraints led to a misbehavior of the actuators and the robot didn't move properly.

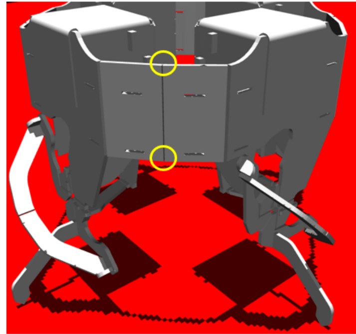


Figure 7: Connections between the legs of the robot, the four legs are connected through two equality constraints.

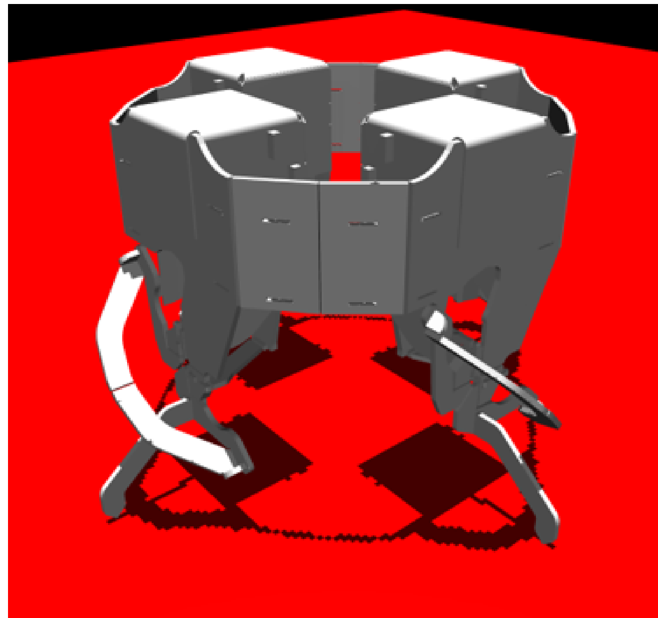


Figure 8: Complete model of the robot obtained directly from the CAD files.

3.1.2 FIRST SEMPLIFICATION

Considering the difficulties mentioned, we had to simplify the model in the second iteration by focusing just on the robot's main features: the robot is quadruped and has the ability to stretch. This implies that the only parts that actually matter are the leg end-effectors and the leg's connection elements. The first is needed for the actual contact of the robot with the ground and the second is for the robot's overall movement. Since each leg has a lift and a swing DoF we used two motors at the beginning of the leg end-effector and used them for moving the system. For the connection between the different parts we used the same approach used before. In Figure 9 the simplified model of the robot.

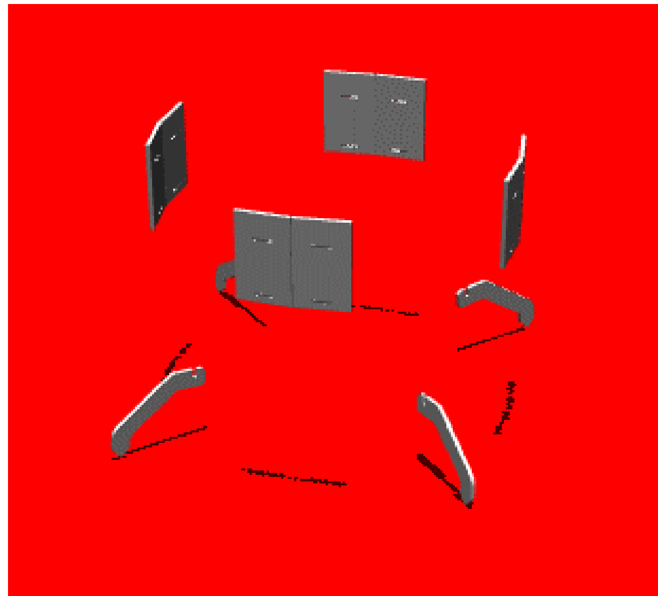


Figure 9: First simplification model of the mClari robot.

3.1.3 FINAL MODEL

The previous model was conceptually correct, but we still used mesh files to represent each leg's tip, causing high computational costs in identifying ground contacts and reaction forces. MuJoCo, in fact, evaluates the contact between two geometries by creating a convex hull or some control surfaces around the two geometries, like bounding boxes. This leads to an increase in the computational cost of the simulation (it is obvious that by working with the real system, none of these problems will actually matter, but, in simulation, it does; the higher the computational cost, the higher the time spent in training the algorithm.)

Inspired by the ant model in MuJoCo(25), we replaced the legs' mesh files with four capsules (default shapes in MuJoCo). This made the simulation lighter, faster in identifying contact points and more precise. Furthermore, as stated above, the robot was modeled as four legs connected to each other through two equality constraints; these were sufficient to impose the closeness of the structure but no force was generated towards the center of the robot, so that, any component of the friction force tangential to the plane was not successfully countered. These forces, indeed, crumple the robot on itself and make it not able to walk properly.

Our robot was indeed able to stretch but with some limitations, that is why we needed something that behaved like a spring when the legs were too close, in order to prevent the overall robot from collapsing by impressing a force towards outside the robot, but that, at the same time, allowed the legs to move and to stretch. We found out that **tendons** were perfect for this job; they are passive actuators that have many tunable parameters like stiffness, duping coefficient, etc... These elements are represented by ropes and have several parameters that can

be tuned, like the stiffness coefficient below and above which they are activated. Of all the parameters, the one that we used was the stiffness. We didn't use the damping coefficient since we wanted to allow full movement to the robot until a certain distance and a complete block after that. We crossed two tendons so that each tendon connected the legs facing each other. Since the distance between two legs is about 10 mm we imposed no forces until the distance was lower than 8 mm and, after that, the force impressed by a spring with a stiffness of 1000 g/s^2. Figure Figure 10 shows the final model implemented.

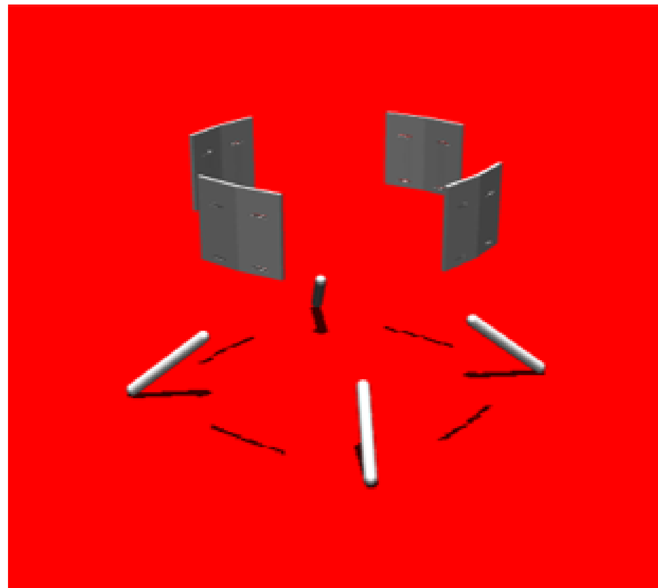


Figure 10: Final model of the robot.

3.2 ACTUATOR DESIGN

Once the mechanical and kinematic system was correctly modeled, it was essential to fully characterize the actuation system for each leg. We began with the physical results obtained from Doshi et al.(3), deriving the Bode plot of the frequency response for each leg's kinematic chain Figure 11.

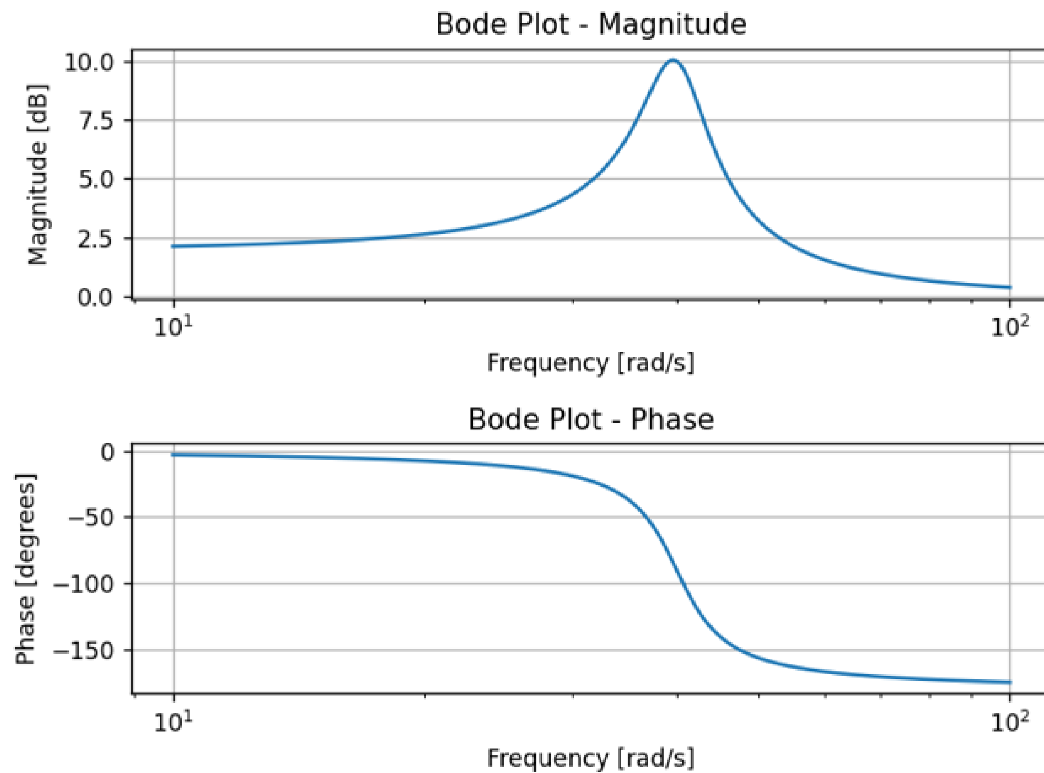


Figure 11: Bode plot of the transmission chain of the leg's actuator in the mClari robot. In the figure, the ratio between the amplitudes of the leg tip and the actuator.

The paper indicates that each actuator chain behaves like a second-order system, allowing us to identify the inertia, damping and stiffness coefficients for the actuation motor. The paper presented for lift and swing the ratio between the amplitude output of the leg effector actuator (in mm) with the amplitude of the piezoelectric actuation (in mm) with respect to the frequency. We were therefore able to find out the motion transfer function:

$$H(s) = \frac{3200}{s^2 + 8s + 1600} \tag{3.1}$$

To model the overall actuator's transmission line, we needed to model two different elements: the joints, which are the elements that are indeed actuated and the actuators, i.e., what moves the joints. Due to the simplifications done above, we imposed revolute joints and, therefore, motors as actuators.

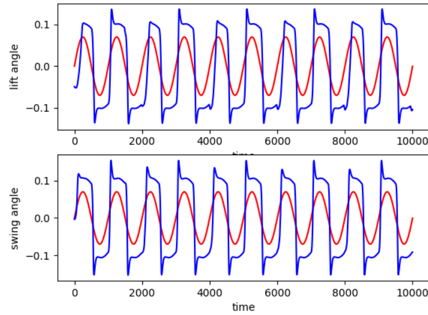
Let's start with the joints, which are the most complex. For modeling them, we started by focusing on their main axes of motion. For each leg, we imposed two different axes of motion: the first is along the x-axis to represent the swing movement and the second is along the y-axis to represent the lift. Then, we worked on their range of actuation: we fixed each joint to work between -5° and 5° since this is the maximum extension allowed by the leg in the real robot, both for the lift and the swing angles.

Furthermore, MuJoCo gives the possibility to set the characteristic of the joint by modeling it as a mass-spring-damper system. Here is where we used the Bode plot presented before. To set the characteristics of the system we have two different possibilities: the first is to manually

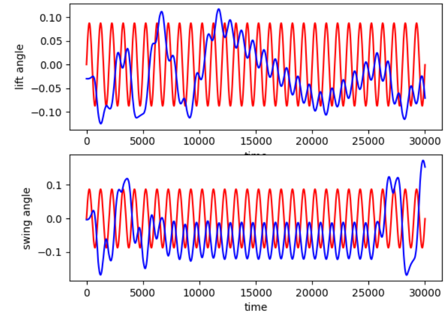
set the values of the armature (which here represents the inertia of the joint along the rotation axis), the stiffness and the damping coefficient; the second is to get both the time constant τ of the overall system and the damping ratio ζ, this can be done by using the **springdamper** attribute. We chose to follow this second approach since we have the transfer function and we derived from that a time constant $\tau = 0.25s$ and a damping ratio $\zeta = 0.1$

Regarding the actuator, we chose to implement the motors and the only important parameter to tune was the **gear**, i.e., the gain constant of the motion. This was, indeed, the hardest parameter to find since the transfer function has as input the elongation of the piezoelectric while, in our case, we were using a motor, therefore, we needed to tune the gear in another way, in the end, we found that having a gear of 300 would have been enough to guarantee a movement throughout all the chosen frequencies.

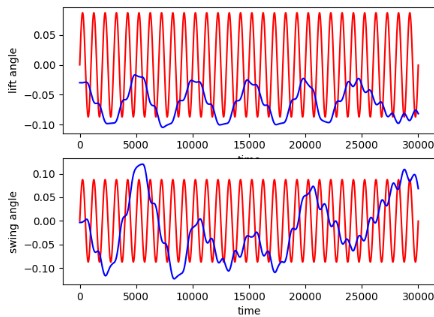
Below in Figure 12 is presented a comparison between the actuation signal (in red) at different frequencies and the real output from the joint angle. The images have been obtained by simulating the real robot and the contact with the ground, that is the reason why the signal, a part from the first frequency, doesn't follow perfectly the behavior of the input. These results can be explained for several reasons: first of all the bode plot presented in Figure 11 are referred to the actuation of one single DoF, either swing or lift, while the results presented in Figure 12 present the coupled behavior. Furthermore, also the presence of the ground constraint affects the performances at high frequencies.



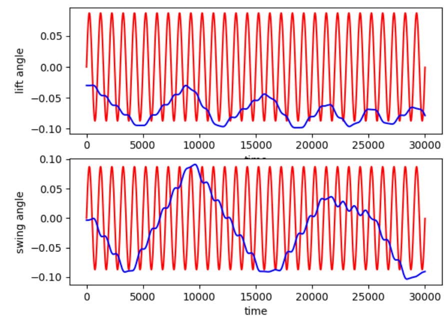
(I) Frequency response at 1Hz



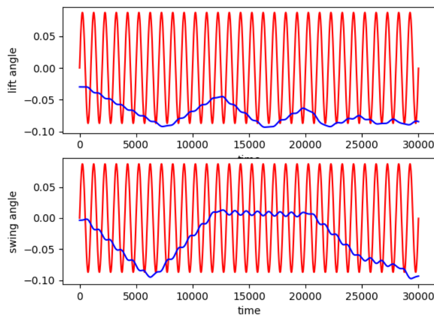
(II) Frequency response at 20Hz



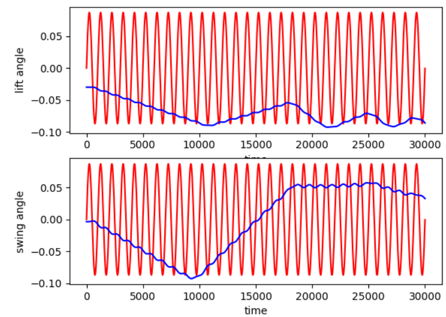
(III) Frequency response at 40Hz



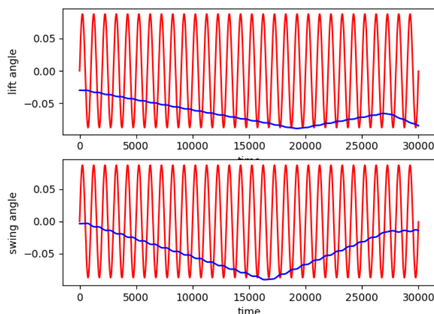
(IV) Frequency response at 60Hz



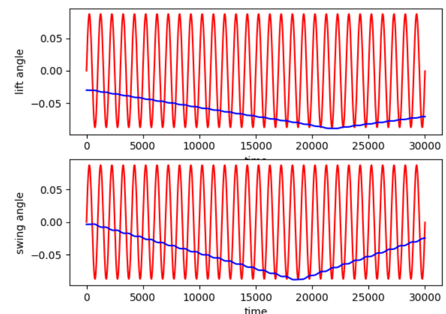
(V) Frequency response at 80Hz



(VI) Frequency response at 100Hz



(VII) Frequency response at 140Hz



(VIII) Frequency response at 150Hz

Figure 12: Frequency response of the lift actuator: in red the control signal imposed, in blue the effective movement of the leg.

3.3 ENVIRONMENT DESIGN

Lastly, we designed different types of surfaces on which the algorithm could be trained to find the optimal frequency. Six types of surfaces were implemented based on two classifications: surface texture and slipperiness level. The surface texture can be either rough or plain (with or without a height field), while the slipperiness level can be categorized as normal, wet, or dry.

To define a surface in MuJoCo there are several parameters that can be tuned. First of all, the ground surface is defined as a rectangle normal to the z-axis; to successfully characterize it, the **size** attribute can be used. It has three parameters, one along each direction (x,y,z axes). The x and y values represent half of the length of the plane in the chosen direction, while the z value represents the offset of the plane along the z-axis. In our case, the robot has an overall dimension of around 20 mm, we decided that we wanted a surface plane that was almost infinite for the robot, which is why we set the dimension to be 500 mm along x and y and no offset along z; this allowed us to have a surface defined as a square of 1mx1m which is big enough for the robot to move on.

Regarding the surface texture, this can be implemented on MuJoCo by defying a height field. To do so, a 2D matrix needs to be created. The size of the matrix depends on the resolution of the field you want to implement, a bigger matrix implies a more refined surface but a higher computational cost in terms of the evaluation of the contact points. We decided to implement a 200x200 matrix since we found it to be a good compromise between resolution and efficiency. Once the height matrix is created, the overall surfaces are divided into a mesh grid the same size as the matrix and, to each mesh element, the height is assigned according to the

matrix. Each value of the matrix is a float in the range [0, 1] and it represents the normalized values of the height; this value is then scaled by MuJoCo, in fact, just for the height field, the **size** parameter has 4 elements: the half dimension along x and y, the amplitude of the hfield along z (which is the parameter that is multiplied to the values in the height matrix) and the offset along -z (which is a way to create holes in the surface). Once the matrix is created, in MuJoCo there are three possible ways to realize a height field:

1. storing the values into a PNG greyscale image, the float in the range [0,1] as to be scaled into a uint8 in the range [0,255] then MuJoCo will rescale the values accordingly;
2. storing the values into a binary file composed by an int32 in the first two lines that specify the number of rows and columns and then a third line with all the float values organized according to a row-major order;
3. leaving the values undefined and just defining the number of rows and columns. This will allocate the needed space in the memory by imposing 0 to all the heights and while compiling these values can be changed.

We decided to use the first approach since it is more useful to have an image of the height field for debugging purposes. What we wanted was a rough surface, that was representative of the superficial roughness. We, therefore, started by taking all the matrix values from a Gaussian distribution $N(0, 0.1)$, we then scaled the obtained values between 0 and 1 by taking the minimum and maximum values obtained (z_{min}, z_{max}) and by using the relationship

$$z_{norm} = \text{uint8}\left(255\left(\frac{z}{z_{max} - z_{min}} - z_{min}\right)\right) \tag{3.2}$$

By casting everything as a uint8 we were able to represent the grayscale image. In Figure 13 the greyscale image obtained and the resulting surface.

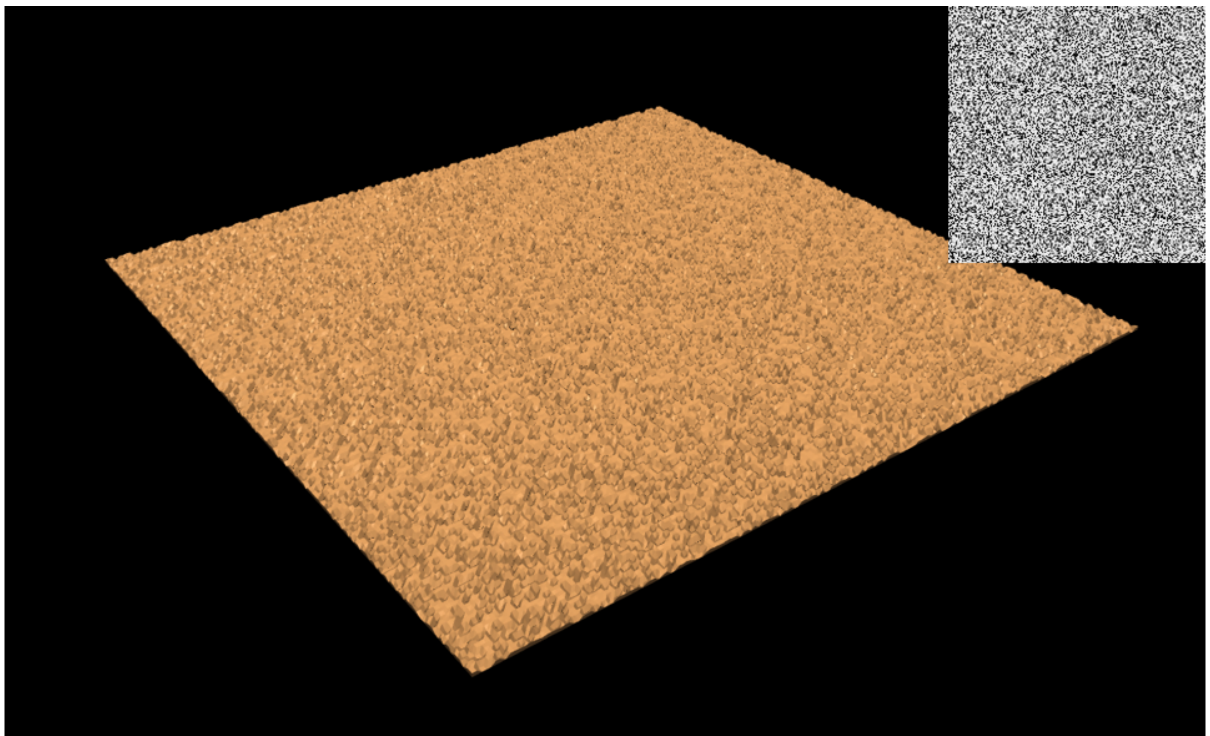


Figure 13: Rough surface with height field values taken by a Gaussian distribution $N(0, 0.1)$; on the upper right corner the png image used.

From the image above, it can be seen that having a 200x200 matrix size is indeed enough for having a good representation of the surface; the problem here is that the environment resulted in being a bit too pointy and a movement on top of it wouldn't be so effective and easy to

perform, furthermore, this surface wouldn't represent correctly a rough surface or any surface in which the robot would actually be able to move on. That is the reason why we decided to smooth the surface a bit by applying a low pass filter to the height image, to eliminate the high frequencies. The filter we chose is a mean 3x3 filter whose elements are $\frac{1}{9}$; the resulting image is a blurred version of the image before, taking just the low frequencies and eliminating the high ones. Figure 14 presents the used PNG image along with the obtained surface.

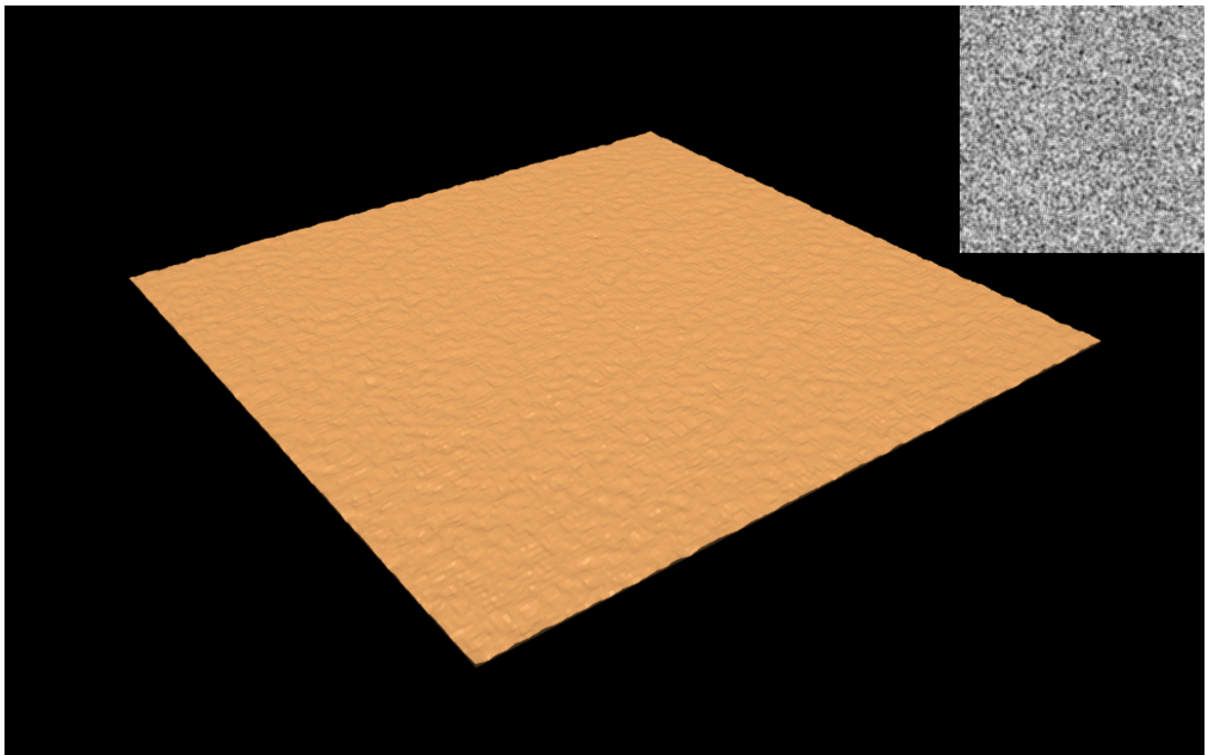


Figure 14: Rough surface with height field values taken by a Gaussian distribution $\mathsf{N}(0, 0.1)$ smoothed by a 3x3 mean filter; on the upper right corner the png image used.

Regarding the slipperiness level, MuJoCo gives the possibility to set the friction coefficients of each of the geometries for contact purposes. The **friction** attribute presents three different friction coefficients:

1. sliding (SC), it is the coefficient responsible for the generation of the force along the tangential axis of the contact plane.
2. torsional (TC), it is the coefficient responsible for the generation of the torque around the axis normal to the contact plane (z-axis)
3. rolling (RC), it is the coefficient responsible for the generation of the torque around the axes tangential to the contact plane.

To fully characterize the behavior of the different surfaces, we worked on the three coefficients above. MuJoCo already has some default coefficients; furthermore, to obtain a higher slipperiness (i.e., a wet surface), we decreased the friction coefficients, while for obtaining an extra dry surface, we increased them. The table below lists the values used for the three surfaces:

TABLE I: COEFFICIENT FOR THE THREE SLIPPERINESS CONDITIONS

surface	SC	TC	RC
normal	1.0	0.005	0.0001
wet	0.1	0.1	0.1
extra dry	1.0	1.0	1.0

CHAPTER 4

REINFORCEMENT LEARNING TRAINING

4.1 REINFORCEMENT LEARNING

As expressed in Section 2.2, the approach we wanted to use for training the control algorithm is by using Reinforcement Learning. Before going further in the discussion, let's introduce what reinforcement learning is and what its main components are. As with all the greatest ideas, Reinforcement Learning results from observing how Nature works. All living beings need to learn the environment they are in and adapt to it to survive.

As stated in Sutton *et al.,*: *Reinforcement learning is learning what to do—how to map situations to actions so as to maximize a numerical reward signal.*(26). This type of learning is usually presented as a third option between supervised(27) and unsupervised learning(28).

All three types of learning imply the presence of an agent, which is the system that learns and of input signals coming from the environment and on whom the agent can learn. The big difference is in how the agent learns what to do. In supervised learning, the agent has not only the input data but also the expected output, so it has to model the relationship between inputs and outputs and apply it to new input data. For unsupervised learning, instead, the agent has just the input data and the learning consists of working on collections of inputs by looking for hidden patterns or relationships that can be useful. For the last type of learning, instead, the agent has the input data from which an action is derived; the action then produces

a reward or a penalty from the environment and the agent uses this signal to train to maximize the expected reward.

Let's now focus on the main components needed to implement Reinforcement Learning and how they apply to our system. As said above, the main component of a Reinforcement Learning system is the **agent**, it moves around in an **environment** and from it, the agent receives as input data the current configuration of the environment, the **state** and from that the agent compute the next **action** to perform on the environment and obtains a **reward** from it. A well-known approach for a decision-making system is to implement a Markov Decision Process (MDP)(29); it is used for decision-making in partly random situations. A MDP can be seen as a 4-tuple presenting the set of all possible states of a system (called State/Observation Space or S), the set of all possible actions performed by the agent (called Action Space or A), the probability function that expresses the probability of given a certain action a to transition from a state s to a new state *s'* (a.k.a. P_a) and the reward function associated to transitioning from a state s to a state *s'* given the action a(a.k.a. R_a). The goal of the agent is to maximize the overall reward and to do so, the agent has to follow a strategy, also called **policy**, that, given the current state of the system, gives, in return, the action that maximizes the expected overall reward.

Regarding the agent, since the main goal is to learn the policy, which is a multi-variables function that relates the actions with the states, recently, some approach that uses Reinforcement Learning integrated with Neural Networks(30) have been presented (31).

The power of Neural Networks (from now on NN) lies in their ability to be efficient for approximating functions whose equation is unknown. Their working principle consists of approximating any function as the weighted sum and composition of simple functions (hyperbolic tangents or ReLu, which is defined as a line for positive inputs and zero otherwise). By tuning the weights of the function sums, the initial function is approximated. If using NN or any other kind of system, it is necessary to implement a way for learning the policy that the agent has to follow and to do so, usually, the training is performed by using a discrete-time system. To do so, we start from an initial state s_0 and from that, an action a_0 is performed; the action is responsible for the transition of the state from s_0 to s_1 obtaining a reward r_0. We called this transition a **step**. Furthermore, since we want the system to train with different initial states or also with the same initial state, but we want to test whether the action taken is actually the best of all, an **episode** is created. The episode is nothing but a collection of a finite number of steps. Each episode is on its own and will produce an overall reward function and this can be used for training the system for long term rewards.

For what concerns our system, we decided to implement a Reinforcement Learning approach with a NN agent; below is a more detailed discussion of all the components we used.

4.1.1 OBSERVATION SPACE

It is the set representing the state of the system at each step. Now, the robot on its own is a very complicated device and there are many possible variables that can be controlled. But, due to the robot's small size, only a few sensors can be placed on it. Furthermore, a high number of state variables implies increasing difficulties of the function needed and a bigger memory needed

to store them. Also, we needed to consider signals and input values that could be obtained both in simulation (as in our approach) and in a real-time system.

The main difficulty is to find the least amount of variables needed to have the most precise knowledge of the system at each step. To find them, we started by considering the goal that our controller has to achieve. We decided to use just four state space variables. The first two are endogenous, i.e., variables obtained by the system itself and the overall power consumed by the actuators, as well as the mean velocity of the robot during one step.

For the mean power used by the motor actuators, this can be obtained from MuJoCo or, in a real scenario, from the piezoelectric actuators. In MuJoCo, in fact, we have, for each actuator, the value of the actuator force/torque impressed by it; also, for each joint, the position and the velocity (linear or angular, depending on the type of the actuator) of it at each simulation timestep can be obtained. The power of each actuator is then calculated by multiplying the torque of each motor by the angular velocity of the respective joint. Something to take care of is that, in MuJoCo, while the velocity and in our case, the angular velocity of the joint, has indeed a sign indicating whether the movement is clockwise or counterclockwise, the actuation torque is expressed as the module of the torque impressed, therefore always positive.

It is important here to make a clear difference between step and timestep. When talking about **steps**, we will be referring to the reinforcement learning meaning, i.e. the 'time' elapsing passing from state s_i to the state s_{i+1}, when dealing with **timesteps** instead we will be talking in the MuJoCo sense, i.e. the integration time used by the physical solver for evaluate the

position, forces etc... working on the robot. The difference is quite big since the step is usually more than 500 times bigger than a timestep.

The power obtained by the motor is evaluated once every timestep or once every multiple of it (better explications will be given in Section 4.3). Therefore, we obtain a function $P(t)$ with positive and negative values. To have a mean power dissipated, we take the mean of the norm of all the power values. We then sum all the mean power obtained by the actuators. The second value is the mean velocity of the center of mass of the robot; in this case, we determined it once every step as the difference in the position of the robot's center before and after a step. This is done since the step we decided is very small in time (around 1 s); therefore, we can approximate the movement of the robot as almost linear (no back-and-forth movements). Computation-wise, MuJoCo allows obtaining the position of the CoM of the robot in the system reference frame; we consider just the first two values since the position along the z-axis, the normal to the plane, can be subjected to oscillations, but it isn't what we really care about. We took the initial and final position and since the step is of 1 second, we simply subtract them and we obtain the two norm of the resulting vector. The last two values are a mix of exogenous and endogenous values: there are errors along the x and y axes between the target position of the robot and its actual position. We decided to take these values since the robot's position is needed above all else. Therefore, there aren't any additional costs related to them. Furthermore, these values can be easily obtained by installing an IMU on the robot or using external localization systems.

4.1.2 ACTION SPACE

For the action space, a similar consideration to the state space has been used. We need an action space that is the smallest possible but allows us, at the same time, to fully control the device. As expressed in Section 2.2, just two variables are needed to control the robot: the frequency and the phase angles group.
The range of allowed frequencies spaced from 1 to 150, anyway, sudden frequency changes would create large jumps that could damage the robot. In a simulation environment, this wouldn't create any problems since MuJoCo is just a solver and it treats the actuation signals as mere numbers; in a real environment, frequent huge frequency shifts could seriously damage the piezoelectric actuators; that is why we decided to constraint the variations in the action space.

The first variable to be controlled is related to the frequency, but it is not representative of it; we decided to make it a variable between -1.0 and 1.0 (a normalized variable results in better performances of the NN training) that represents a decrease of an increase of the frequency. In detail, by calling the previous frequency f_i, the successive f_{i+1} and the first value of the action $a_{i,0}$ we have:

$$f_{i+1} = f_i + a_{i,0} \cdot \Delta f \tag{4.1}$$

where Δf is a fixed value. The frequency is then cast into an integer since we will work with integer frequencies. The second action variable, instead, selects the optimal angle combination. Section 5 will present more results on the phase angles combination, but we saw that a good gait is achieved when the legs have angles similar to two at two, we therefore reduced the possible

combinations from 27 to 8, having a reduced memory space for angles and less possible choices. Again, we chose a float number between -1.0 and 1.0 (always for performances) as the second value and we scaled the select values in an integer between 0 and 7.

4.1.3 AGENT

For the agent, we implemented a two-layer neural network with 10 nodes in the first layer and 3 nodes in the second layer. Since it takes the 4 observation space values as input and computes two normalized float numbers as output, we used the hyperbolic tangent as an activation function.The agent is trained using the SAC algorithm implemented by the sb3 library (32), anyway, a further discussion will be presented in Section 4.3.

4.1.4 REWARD FUNCTION

The reward function is, maybe, the most difficult yet important function to be determined. It is what the agent learns while moving; therefore, if designed incorrectly, the behavior of the robot won't be the expected one.

In developing the reward function, two main objectives were kept in mind: maximizing the robot's movement speed while minimizing overall energy consumption and reaching the target position. For the first objective, we focused on maximizing the overall system efficiency η (see Equation 5.1). It is defined as a sort of inverse of the cost of transport (CoT), which is a well-used measure to characterize the energy consumed by legged robots. The first thing we did was perform some simulations and evaluate the value of η in different conditions (Section 5 will present a more detailed discussion on this topic), from those, we saw that the value of the efficiency was always around the first decimal (range 0.01-0.1). The value of the efficiency

was, therefore, already normalized and this is the reason why we took exactly this value as the reward associated with each step; in this way, a reward for high efficiency was given. For the second objective, i.e., for reaching the desired position, we, at first, thought about minimizing the position errors, giving a reward to the movement in the correct direction by evaluating the distance between the desired and the actual position of the robot's center of mass. The idea was to find a variable related to this distance and to have an overall reward function as the weighted sum between this and the efficiency variable.

After testing the system with this reward function, we noticed that in some simulations, more weight was given to efficiency rather than getting to the correct target. Therefore, we decided to implement a different approach. We decided to give a reward based on just efficiency and to give a reward to the robot once the target was reached, the reward given by reaching the target was actually big (+10). By doing so, we actually solved other problems; for example, by just giving a final reward whether the target is reached, we avoid imposing a defined trajectory on the robot, but, on the other hand, we let it decide freely the path to follow. A penalization function was also implemented: when the robot fell, for any reason, a -100 penalization was given and a new episode started.

4.2 GYM ENVIRONMENT

Once the physical system was fully characterized, we needed an environment able to help us deal with reinforcement learning. We used GYM (33) for this purpose. GYM is a toolkit developed by OpenAI that can be used for developing and comparing different reinforcement learning (from now on RL) algorithms.

Gym presents a lot of already implemented environments and algorithms for reinforcement learning. Furthermore, it has a big community around it that makes the software easy to use thanks to the wide range of tutorials available. The greatness of GYM is that it provides a standardized environment on which many other libraries and toolkit can be attached to; the only catch is that the standard has to be followed accurately.

We started by creating our own custom gym environment. It would help us in both having a platform for actually controlling and testing the control logic and it will make it easier to implement already existing RL algorithms. A Gym environment is defined as a class with many objects, i.e. functions belonging to the class. Below is a list of the objects needed for creating a custom gym environment and how we defined them (in the description below we will be using the name of the actual function implemented in the gym environment class) .

4.2.1 STEP

The *step* function is maybe the most important function of them all. It is responsible for everything related to any step of any episode of the simulation. Its role is to take the actions obtained by the agent and guide the system throughout the performance of the desired action. After that, or together with that, it is responsible for evaluating the observation, computing

the reward and checking whether the termination conditions are met. The standard convention is that the step function takes as input the action to be performed and returns five variables: **observation**, which is a NumPy(34) array of the size of the observation space, **reward** which is a float value that stores the reward obtained by the step, **terminated** and **truncated** which are two boolean variables signifying whether the episode has ended and if it has ended before the maximum number of steps and **info** which is a dictionary stating which termination condition the episode has met (more in 4.2.5).

What is stated above is the general idea of the step function, in our custom function, here are the steps we perform: once the agent determines the action to be performed (i.e., the frequency and phase groups), the initial position of the robot is recorded, the simulation timestep is changed accordingly to the frequency (in order to always have a good representation of the robot's behavior), the microbot's movement is then simulated for one second using MuJoCo. Once every 2ms of simulation, the power consumed by the actuators is registered. After one second of simulation, the final position of the robot is obtained along with the mean power consumed by all the actuators. The observation states and the reward are then computed and the termination conditions are checked. Algorithm 1 the pseudocode of the function

4.2.2 RESET

The *reset* function is also very useful; it is used each time an episode is terminated and it resets the environment so that a new episode can begin. It doesn't need any particular values as input. Nevertheless, it may accept a **seed** that can be used to reset the environment and set the initial parameter accordingly and a **options** variable that again can be used for customizing

Algorithm 1 step function of the gym environment

1: compute frequency, angles from the action
2: timesteps = 500*frequency
3: simulation_timestep = 1/timesteps
4: step = 0
5: compute initial_position
6: **while** step ¡ timesteps **do**
7: compute actuation signal
8: simulate one step
9: **if** 2 ms has passed **then**
10: actuator_power.append(get actuators power)
11: **end if**
12: **end while**
13: computes final_position
14: observation = f(initial_position, final_position,actuator_power)
15: reward = compute_reward(observation,action)
16: terminated, truncated, info = check_done(observation)
17: **return** observation, reward, terminated, truncated, info

the new environment created. As output, instead, two variables are requested: **observation** (which is the same as above and gives some information on the new environment) and the **info** dictionary that can be used to store whether the reset has been successful and any other relevant information.

Regarding our environment, MuJoCo offers an already implemented function that resets the environment (it is called mj_resetData) and comes back to the initial setup and that is what we used.

4.2.3 RENDER & CLOSE

The *render* and *close* functions are indeed different functions but we paired them up in this discussion since their aim is related. In detail, they both work on the rendering of the

environment, i.e. they are responsible for showing or not the simulation running during the episode. The *render* function takes care of showing the behavior of the movement on the terminal and it is responsible on when to update the rendering. On the other hand, in the *close* function, all the commands needed to stop the showing of the simulation running are stored.

MuJoCo presents some already implemented functions for rendering and synchronizing the physical environment. ***viewer***=**mujoco.viewer.launch_passive()** creates an object (***viewer***) which is stored in the MuJoCo environment and has as an internal function **sync()**, which is used to synchronize the previous object to the latest update and to launch a window on the terminal with the updated image. In the *close* function, the window is closed and the ***viewer*** object is simply deleted.

4.2.4 COMPUTE_REWARD

The *compute_reward* function is responsible for taking the observations as input and returning the computed reward accordingly.
In our custom environment we implemented the reward function as specified in 4.1.4

4.2.5 CHECK_DONE

The *check_done* function is another fundamental function since its role is to identify whether the termination conditions are met or not and store this information into a variable usually called **terminated**. Furthermore, each episode has a fixed maximum number of steps and this function also checks if the episode has ended before the maximum time or not by storing this into two different variables: the **truncated** which is True whether the episode has terminated before the maximum number of timesteps and False otherwise and **info** which is a dictionary

that stores the reason why the episode has terminated. In our custom environment, we set the maximum length of an episode to 180 steps (3 minutes), this was done to have, during the training of the logic algorithm, fast episodes that would lead to showing the robot a wider range of scenarios. Besides reaching the maximum number of steps, an episode can terminated due to two other major conditions: the robot falling on the ground or out of the geometry space (which also results in a penalty) or the robot reaching the chosen target (which instead results in a reward).

4.2.6 OTHERS

The functions presented above are the ones that every gym environment needs to have, or better, that a gym environment is expected to have (at least at the time in which this thesis is written). Anyway, we implemented many other functions in the gym environment to help us create an easier readable code and for performing repetitive tasks. Some of them are *find_pose*, which gets the position of the microbot at the desired instant, or *get_power*, which evaluates the power consumed by the actuators, and more...

4.3 TRAINING

Once all the components of the RL process and the GYM environment are defined, we can now proceed with the training of the model. Having a GYM environment opened us the possibility to use a lot of existing libraries that could really speed up the training process. Among them, we found the Stable Baselines3 (sb3) library (32) to be very useful. This is a collection of Reinforcement learning algorithms and presents a fast and efficient way for both training and testing a NN agent. In the next subsections, a description of the sb3 library and the hyperparameters tuning process will be presented.

4.3.1 STABLE BASELINE 3

Stable Baseline 3 is, at the time of this document, the last version of the Stable Baseline library (35). It offers a collection of already implemented Reinforcement Learning training algorithms and an easy-to-use environment for implementing them and training an agent. In Table II a list of some of the algorithms available, together with the type of action and state space they can work with. Sb3, indeed, divides the possible sets into 4 groups: **Box**, **Discrete**, **MultiDiscrete**, **MultiBinary**.

1. The **Box** space is a continuous N-dimensional space, it is used for implementing actions that have a continuous value (such as a variable from -1.0 to 1.0).
2. The **Discrete** space is a list of a finite number of actions, at each step of the simulation only one action can be chosen.
3. The **MultiDiscrete** is a list of Discrete spaces, at each step only one action for each space can be chosen.

4. The **MultiBinary** is a list of possible actions and at each step any action and combination of actions can be chosen.

TABLE II: TABLE OF THE RL ALGORITHMS AVAILABLE IN SB3, V IMPLIES THAT THE ALGORITHM SUPPORTS THE SPACE WHILE X DOESN'T. TABLE INSPIRED (1).

Algorithms	Box	Discrete	MultiDiscrete	Multibinary
ARS	V	V	X	X
A2C	V	V	V	V
DDPG	V	X	X	X
DQN	X	V	X	X
HER	V	V	X	X
PPO	V	V	V	V
QR-DQN	X	V	X	X
RecurrentPPO	V	V	V	V
SAC	V	X	X	X
TD3	V	X	X	X
TQC	V	X	X	X
TRPO	V	V	V	V
Maskable PPO	X	V	V	V

Regarding our own environment, the action space is composed by two different values, the frequency increase/decrease and the angle phase group (see Section 4.1 for more details), this two values can be obtained by using a Box space or a MultiDiscrete one, that implies that

the DQN algorithm[1] or anything related to it that couldn't be used. After analyzing all the different algorithms presented by also focusing on the computational resources associated with them, we narrowed down the choice between two different algorithms: PPO and SAC.

The PPO, or Proximal Policy Optimization algorithm (37), is a method of the family of the policy gradient methods, i.e. algorithms whose aim is to improve directly the policy to follow. Since the policy is itself a function, this method tries to identify its fundamental parameters and to improve them by evaluating some of the rewards associated with each policy and by computing the gradients of the rewards with respect to the policy's parameters. The greatness of this algorithm is that the updates of the policy are done by using a lower and an upper limit which are defined by a clipping function, this allows the policy to converge to the optimal one by using a step that is neither too small to increase the computational time neither too big to deviate too much from the original policy and avoid overshooting. The positive aspects of this algorithm are the simplicity, the stability and the efficiency.

The computational cost of PPO is lower than some other algorithms since it uses a first-order approximated function (the clipping function) to constrain the policy update. Furthermore, having a function that constrains the update to be too big implies that there is no need for hyperparameter tuning since the update is already limited. One of the drawbacks is due to the fact that the algorithm is on-policy, which means that the algorithm, at each step, uses

---

[1] The DQN algorithm(36), is an algorithm that approximate the Q-policy function by using a DNN, a Deep Neural Network. The $Q(s, a)$ function is a function that predicts the total cumulative rewards given by performing action a in the state s

the policy, evaluates the policy changes and then modifies the policy again. This will lower the efficiency since, at each step, the policy needs to be updated before moving again.

The on-policy nature of the PPO algorithm makes it inefficient for our setup; in fact, when thinking about swarm robots, an off-policy mechanism would be more efficient since it will allow us to parallelize the work and improve faster. That is mainly the reason why we moved to the SAC algorithm.

The SAC or Soft-Actor-Critic Algorithm (38), is an off-policy algorithm widely used in the field of deep reinforcement learning. The goal of the algorithm is to evaluate at the same time three different functions: the policy and two Q-functions. The aim of the algorithm is to maximize at the same time the expected reward and the entropy of the Q-function. The entropy is a measure of the randomness of the movements, and by maximizing both, the goal is to succeed in the task also moving as randomly as possible. This implies a high exploration of the system and avoids the policy from being stuck into a local maximum. Furthermore, having two different Q-functions is used not to overestimate the effect of an action on the Q-value (the value function is a measure of the expected long-term reward related to the short term by taking an action $\mathfrak{a}$). These are just some of the reasons why this algorithm has shown to outperform other on and off-policies. The computational cost is a bit bigger than the cost of other algorithms like the PPO, but having an off-policy infrastructure can lead to better results when considering a swarm configuration since the workload can be divided and more observations can be taken.

4.3.2 HYPERPARAMETER TUNING

We, therefore, chose the SAC algorithm for training the system. That said, the sb3 library offers a lot of ready-to-use functions that make the training very easy to perform. The first thing to do is to create an ad hoc model of the system and the parameter needed is a first string to set the type of NN used to approximate the policy, some examples are:

- **'MlpPolicy'**: Multi-Layer Perceptron, is the simplest one and presents a fully connected structure; it accepts vectorial inputs.
- **'CnnPolicy'**: Convolutional Neural Network structure, it is useful for handling images.
- **'MultiInputPolicy'**: it is useful for combining different kinds of inputs like in environments in which the agent receives both visual and vector inputs.
- **'XLstmPolicy'**: where X can be either Mlp or Cnn it is a combination of Mlp or Cnn with a Long-Short-Term-Memory, in more sophisticated cases in which memory is needed.

We used the 'MlpPolicy' since our input has a vectorial form. Once the shape of the NN has been defined, some of its characteristics can be modified, we can act on different parameters:

1. **network architecture**: the number of layers and the number of neurons in each layer can be modified.
2. **activation function**: the type of activation function can be chosen between 'ReLu', 'tanh', 'LeakyReLu' and others...
3. **learning rate**: the value of the learning rate can be modified and is one of the hardest parameters to tune.

4. **optimizer**: also the optimizer can be identified and chosen from a variety (like 'Adam','SDG',...)

5. **others**...

In our case, we set the network architecture to be a two-layer with 10 neurons on the first layer and 3 on the second to make the system faster.
As an activation function, we chose the 'tanh' since we wanted to have a normalized final value between -1.0 and 1.0. For the learning rate, we did some tests, and we found that 0.01 and 0.1 gave us successful results that will be presented in Section 5.

Once the NN is defined, the other parameters needed to create the sb3 model are the environment and a **verbose** parameter that asks whether feedback on the training will be outputted or not. Once the model is created, it is a class containing a lot of functions like **learn**, **save** and **predict**.

Learn needs as only a parameter the number of steps the training will run; it represents the total number of steps regardless of the number of elapsed episodes.

Save is used to save the trained model in a file such to have it there already trained without the need to train it every time.

Predict is used to take the model and, given an observation, it will output the action chosen by the model.

Another important function that can be added to the model is a **callback**. This can be either custom or standard, and it is used for tracking the behavior of the training in a custom way by taking the desired parameters. With the sb3 environment, training the system is very easy, below in Algorithm 2 is a pseudo-code of what the environment performs.

Algorithm 2 Reinforcement learning step

1: compute initial observation
2: **while** terminated == TRUE **do**
3:     compute action
4:     execute step
5:     compute reward, new observation
6:     episode reward += γ*reward
7: **end while**
8: **return** episode reward, terminated

CHAPTER 5

RESULTS

Once the physical model (in MuJoCo) and the Reinforcement Learning Environment(Gym) have been characterized, we can proceed on testing the system and evaluate the behaviors of our model. In the next sections various results will be presented, each of them will cover a different characteristic of the model.

5.1 PHASE ANALYSIS

Once the system model was complete and the actuators fully characterized, we aimed to determine if the results from (39) could be repurposed and validated for our mClari robot simulation. In detail, we aimed to understand the gait behavior of the microbot influenced by phase angles and frequencies.

As stated already in Section 2.2, we narrowed the number of controlled variables to just two, the frequency and the phase angle groups. While the frequencies vary from 1 to 150 Hz, for the phase angle groups, we decided to use 3-tuples composed of permutations of the angles $(0, \pi/2, \pi)$, having a total of 27 groups. Having an already-implemented GYM environment helped us perform simulations for all the different groups. In detail, for each 3-tuple, we ran eight simulations of the MuJoCo model described in Sec. 3.1.3 for 1 minute, varying the frequency from 1 Hz to 150 Hz ([1, 20, 40, 60, 80, 100, 120, 150]).

We, therefore, ran 60 steps of the simulations and, at each step, we recorded the position of the robot's center-of-mass and took the x and y coordinates (it wasn't something new since the environment already evaluated the final position at each step). At the end of each simulation, we plotted the trajectory followed by the microbot in 1 minute. We decided to represent 27 graphs, one for each 3-tuple, and in each of them to report all the gaits followed by the robot for each frequency. The results are presented in Figure 15; from those, it can be seen that, for each triple a wide variety of movements can be performed by just changing the frequency, at the same time, by looking at the same frequency in different angles group, it can be noticed that the behavior is also widely different and therefore difficult to characterize theoretically a priori. This indicated that frequency and phase angles are highly interconnected, and by choosing the correct tuple (frequency, phase angles) any type of movement and direction can be achieved.

A remark that is to be made is that the model we are dealing with is just a simulation of the robot and not the real one. This implies having a behavior that can differ, also significantly, from the one of the real system. Furthermore, having a so simplified system with respect to the real robot and also transforming the two piezoelectric actuators with all the transmission lines into two rotational motors is a huge simplification. Those aspects may produce movements that could not appear in the real system; for example, we noted that for some frequencies, a continuous rotation of the robot keeps rotating on itself while moving, which is a behavior that the real robot usually doesn't have. Nevertheless, the results presented here are a characterization of the behavior of our model, and the important aspect underlying is that the model

moves similarly to what the real microbot does in terms of dependencies of both frequency and phase angles.

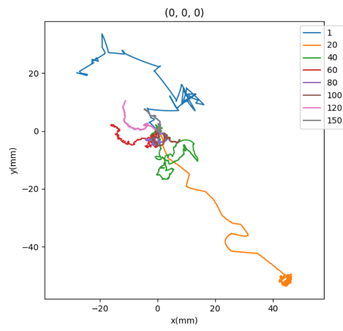
5.2 POWER ANALYSIS

After identifying the trajectories performed by the microbot and having highlighted the dependencies from both frequency and phase angles, it is time to focus on the characterization of the power consumed by the actuators during movement and give an idea regarding the efficiency of the movement.

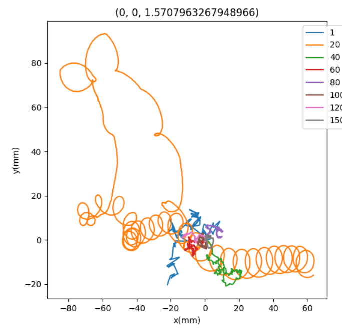
We started by looking at the power consumed by the actuators, as already stated, MuJoCo offers a wide range of variables that can be controlled and that can be worked on. For characterizing the actuators, in fact, MuJoCo gives as output, at each timestep, the torque given to the motors; also, for each joint, the position and angular velocity (in our case, we are dealing with rotational joints) are given. Therefore, the overall power consumed by the actuators can be obtained by simply multiplying, at each timestep, these two variables for each actuator.

At first, we wanted to see the behavior of the power with respect to time; therefore, we ran a simulation of the system for 10 seconds with angle group (0,0,0) and frequency 1 Hz, and we chose, as environment, the plain one with the normal slipperiness level. In Figure 16 are presented the eight powers consumed by the actuators with respect to time.

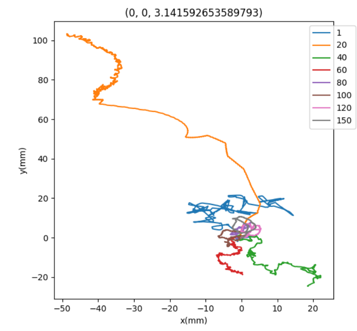
From the picture, it can be seen that, as expected, the power consumed is symmetric on the lift and swing actuators, a peak is presented in all the graphs and the behavior is periodical, this is something we actually expected from the system since the movement of the leg is composed by a phase in which the leg is lifted and a phase in which it actually touches the ground giving



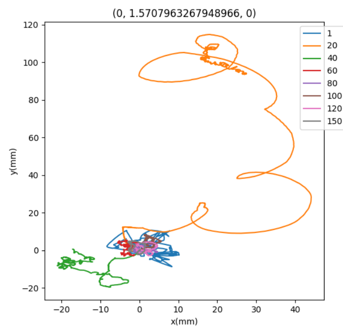
(I) Group $(0, 0, 0)$



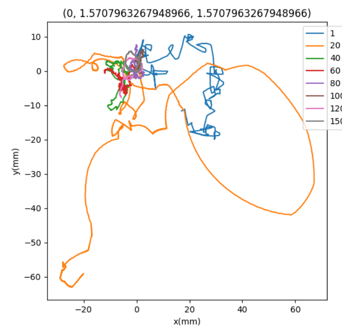
(II) Group $(0, 0, \pi/2)$



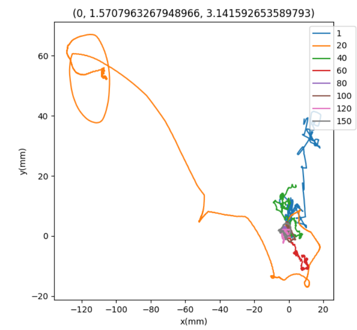
(III) Group $(0, 0, \pi)$



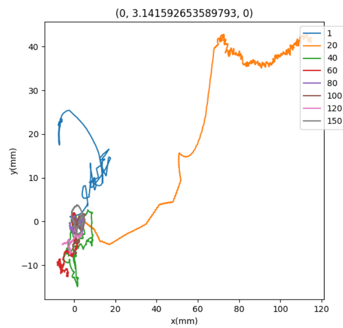
(IV) Group $(0, \pi/2, 0)$



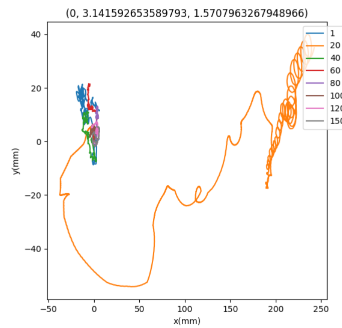
(V) Group $(0, \pi/2, \pi/2)$



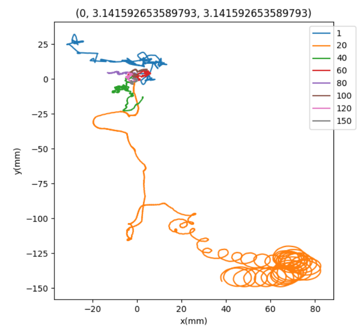
(VI) Group $(0, \pi/2, \pi)$



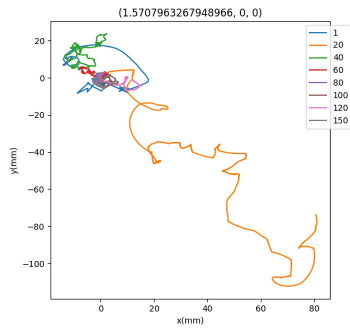
(VII) Group $(0, \pi, 0)$



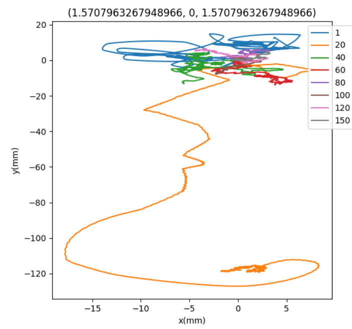
(VIII) Group $(0, \pi, \pi/2)$



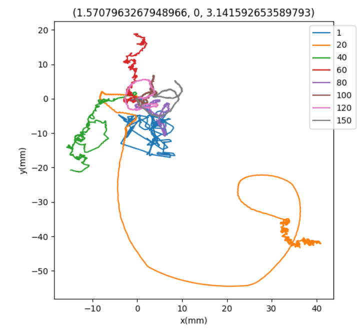
(IX) Group $(0, \pi, \pi)$



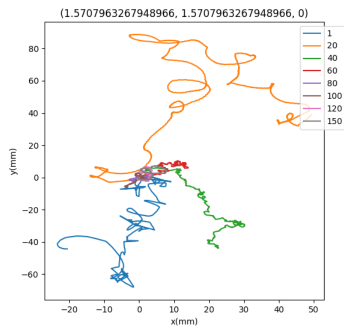
(X) Group $(\pi/2, 0, 0)$



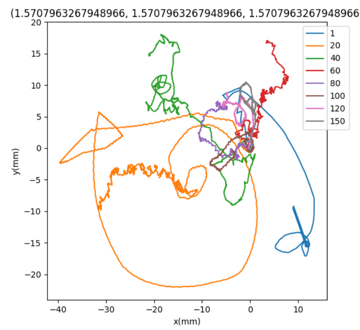
(XI) Group $(\pi/2, 0, \pi/2)$



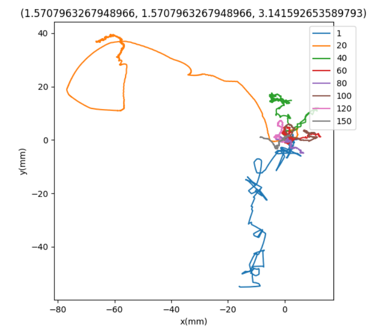
(XII) Group $(\pi/2, 0, \pi)$



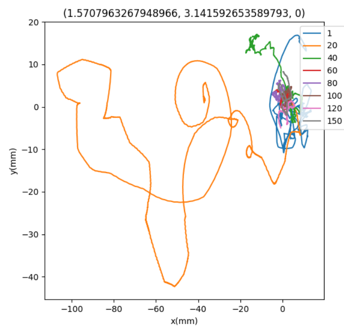
(XIII) Group $(\pi/2, \pi/2, 0)$



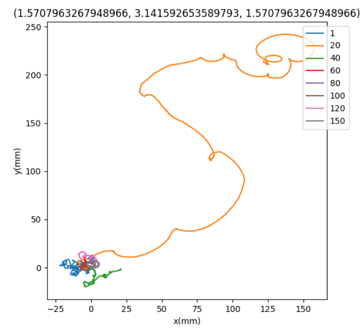
(XIV) Group $(\pi/2, \pi/2, \pi/2)$



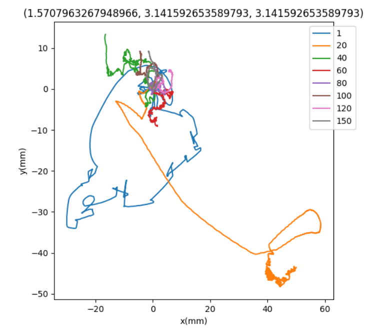
(XV) Group $(\pi/2, \pi/2, \pi)$



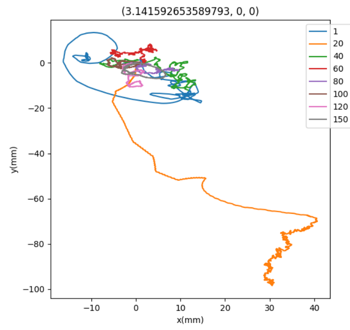
(XVI) Group $(\pi/2, \pi, 0)$



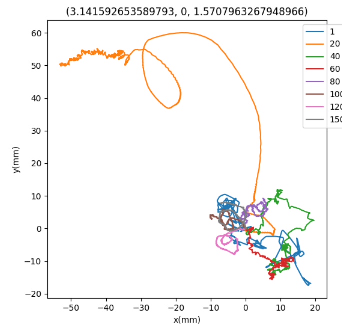
(XVII) Group $(\pi/2, \pi, \pi/2)$



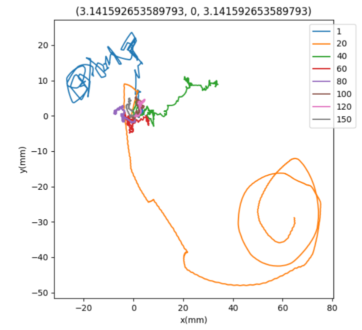
(XVIII) Group $(\pi/2, \pi, \pi)$



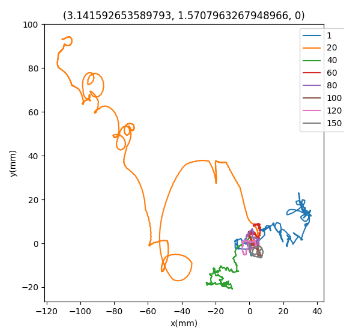
(XIX) Group $(\pi, 0, 0)$



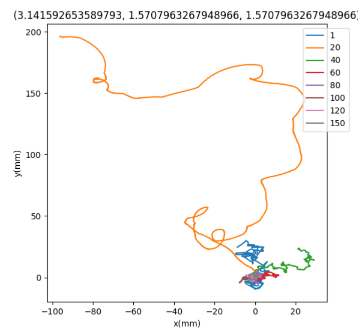
(XX) Group $(\pi, 0, \pi/2)$



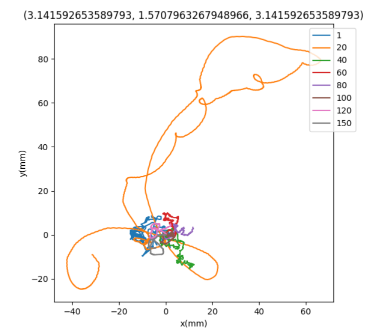
(XXI) Group $(\pi, 0, \pi)$



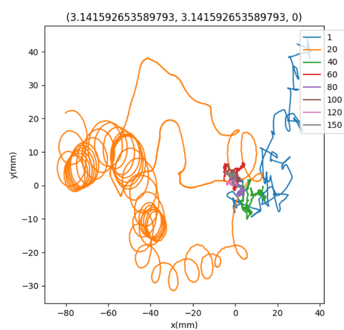
(XXII) Group $(\pi, \pi/2, 0)$



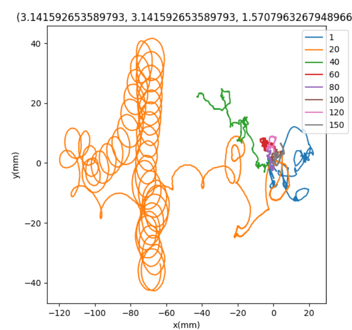
(XXIII) Group $(\pi, \pi/2, \pi/2)$



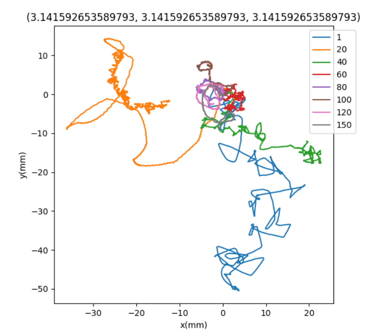
(XXIV) Group $(\pi, \pi/2, \pi)$



(XXV) Group $(\pi, \pi, 0)$



(XXVI) Group $(\pi, \pi, \pi/2)$



(XXVII) Group (π, π, π)

Figure 15: Trajectories followed by the robot during 1 minute of simulation for different angle groups and with different frequencies.

a peak of the torque generated by the actuators. Another aspect to care about is related to the value obtained by the power itself, since we assume that all the energy is given by the robot and not to the robot itself, we expect a power behavior that is always positive, but this is not the case, since the value obtained is both positive and negative. MuJoCo, in fact, gives the torque with a positive sign always, since it is the module of the given torque, for the angular velocity of the actuator, instead, it assigns it a sign to identify the direction of motion; each actuator is, indeed, defined with an axis of motion, therefore an angular velocity in the accordingly direction will have a positive sign, while the one in the opposite way will have a negative one.

By characterizing the behavior of the power consumed by the robot, we were able to use this information for characterizing the real parameter we wanted to deal with: the **efficiency** of movement.

5.2.1 EFFICIENCY

As said before, to maximize the operative life of the robot, we need a variable to control and we chose the efficiency η of the gait, intended as the inverse of the cost of control (CoT), we defined therefore η as:

$$\eta = \frac{mgv}{\sum_{i=1}^{8} \bar{P}_i} \tag{5.1}$$

Where **m** is the mass of the microbot, we set it to be around 0.97g, **g** is the acceleration of gravity (9.81 m/s^2), **v** is the mean velocity of the robot after one step and $\bar{P}_i$ is the mean power consumed by the i-th actuator expressed as the mean of the norm 2 of the power vector whose elements are the powers evaluated at each timestep.

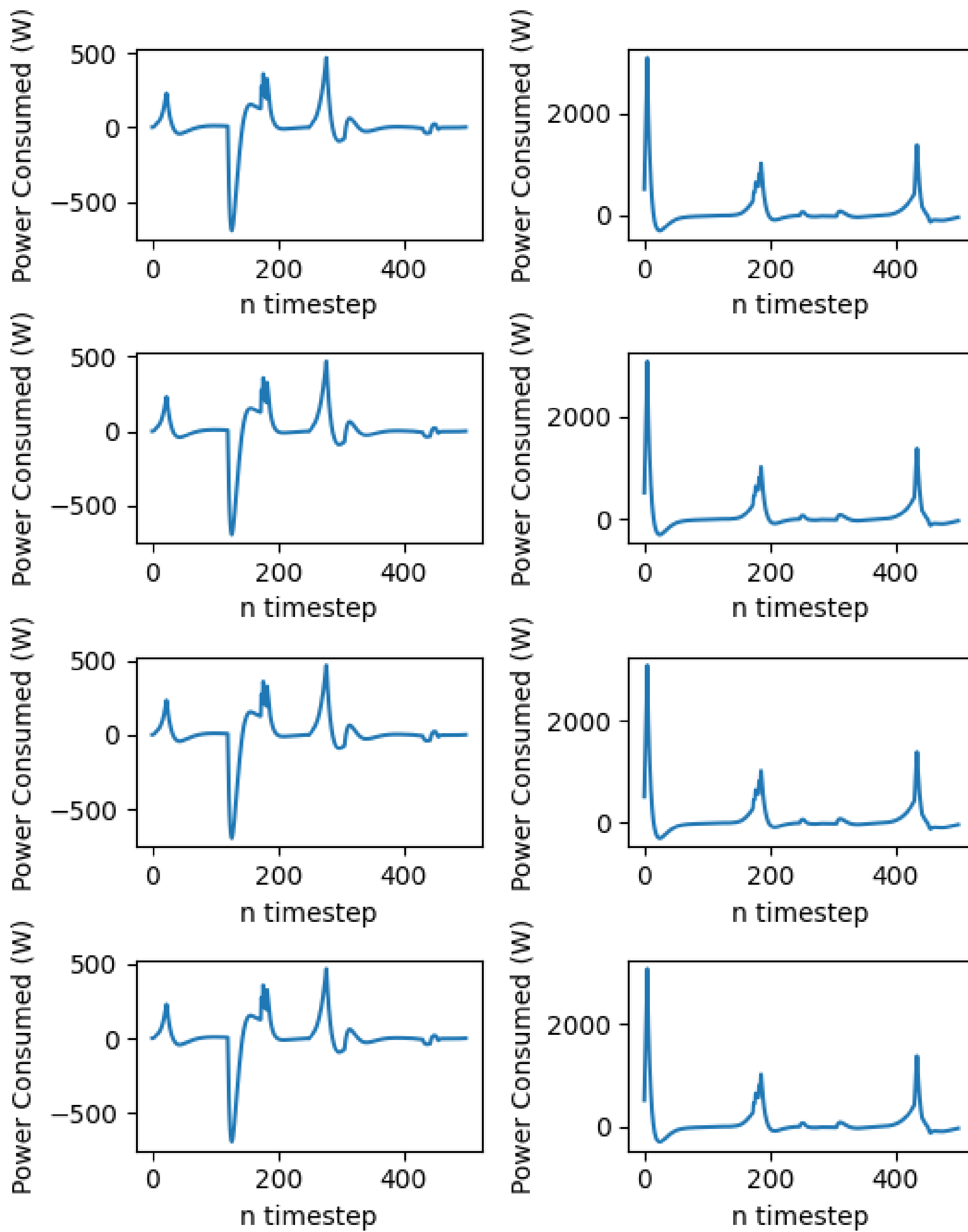


Figure 16: Power consumption of the lift and swing actuators in 1 second of simulation for angle group (0,0,0) and frequency 1 Hz

Since the efficiency has to be adimensional, we need the numerator and the denominator to have the same measurement units. As said before, the chosen units are MMGS, therefore the angular velocity obtained by the software is in mm/s while the power is in $g \cdot \frac{\mathrm{mm}^2}{\mathrm{s}^2}$ or nW. We decided to transform everything into SI units after the computation so that both the numerator and denominator were expressed in W.

To have a first idea of how the efficiency changes with respect to both the frequency and the phase angle groups, by using the same 27 angle groups, we ran eight 10-second simulations (it means 10 steps) by using again the eight frequencies used above. For each frequency and phase angle, we evaluated the efficiency for each step and then we averaged them together. This has been done since, at the beginning of the simulation, the robot may take some time to start moving, and this can lead to a very small velocity, which can affect the value of the efficiency, giving a bad frequency response affected by a not yet developed movement.

To have a better understanding of the efficiency behavior, Figure 17 presents the plots of power consumed, mean velocity and efficiency with respect to frequency for three groups of angles. The results indicated that while power consumption is nearly the same across different angle combinations, velocity and efficiency had specific frequency values that maximized performance (resonance frequency). Furthermore, these optimal frequencies strictly depend on the chosen angle group, making stronger our hypothesis of inter-correlation between frequency and phases.

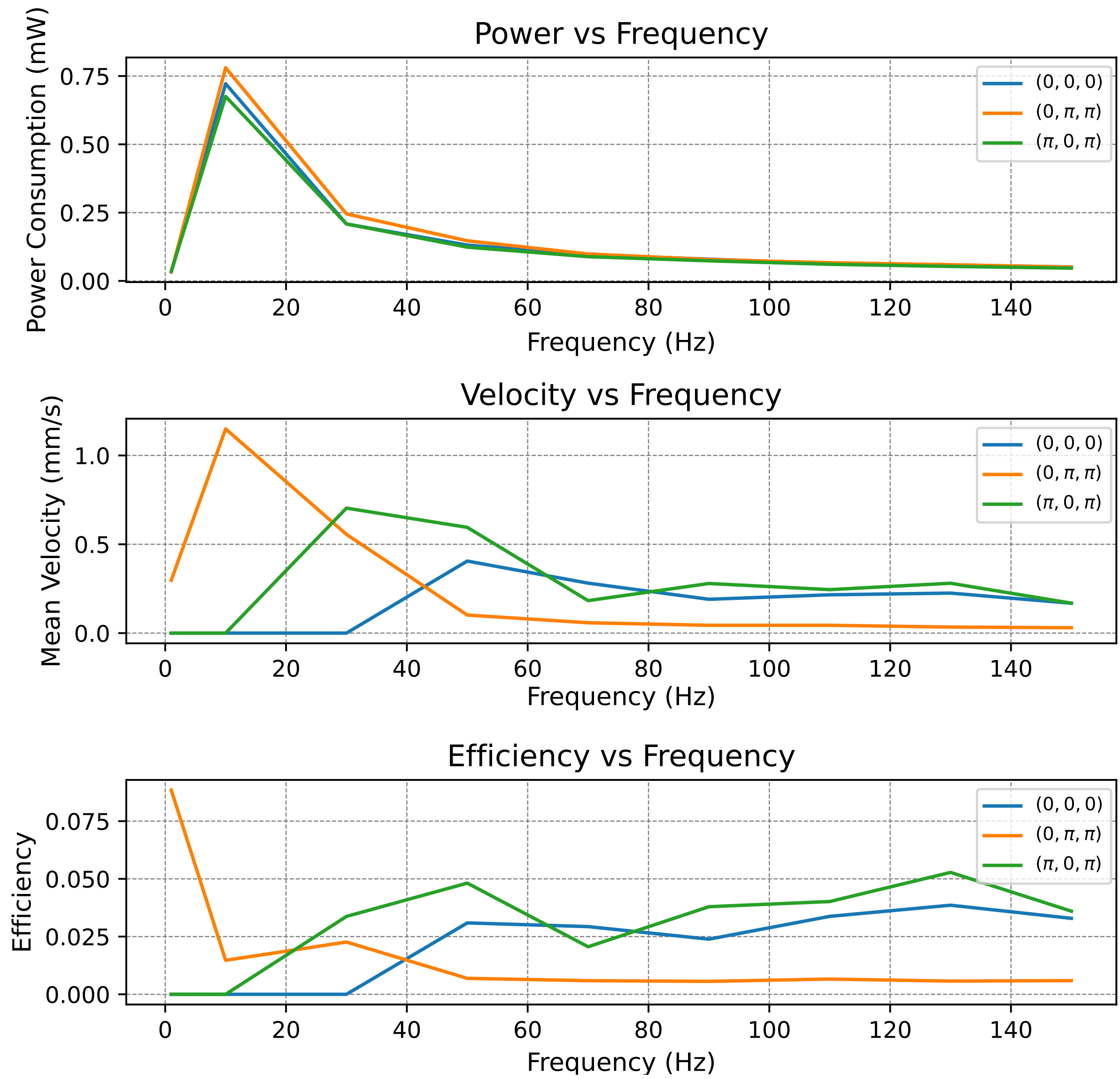


Figure 17: Power consumption, mean velocity and efficiency of robot movement for different frequencies with fixed inter-leg phase shifts triples of (0,0,0), (π,π,0) and (π,0,π).

It is important to notice that the maximum efficiencies is obtained with different frequencies and, furthermore, the maximum doesn't always have the same amplitude but there are higher efficiencies.

5.3 RL RESULTS

In the previous Sections, we dived into the overall environment we created to train and test our model using Reinforcement Learning. It is now time to present the result of the training. Let's have a brief recap of the setup we used. We started by creating the physical environment on which to test all our control logic. We then created a gym environment to use Reinforcement Learning for testing the logic, and lastly, we used the sb3 library to train the algorithm using the Soft Actor-Critic (SAC). After tuning the hyperparameters (learning rates, size of the NN layers, etc.), we found that training the algorithm for 500,000 steps was indeed sufficient for having some good results in terms of increasing the efficiency of the movement.

To train the model to be as reactive as possible to new environments we proceeded in training by constantly changing the type of terrains the robot moves on. Since the sb3 training deals just with the total number of steps the model is trained for, the number of episodes is not decided before. Each time an episode ended, we changed the type of terrain randomly so that the new one featured a different terrain for the robot to navigate together with a different target to reach. Even if the SAC algorithm is already thought to maximize the entropy of the system, i.e. to choose a moving strategy that is as random as possible, by using this approach we aim to train the neural network to adapt quickly to changes, improve system robustness and promote generalized behavior by avoiding focus on a single terrain type.

After the training phase, we tested the resulting model on all six different terrains each for more than an entire episode. We also recorded, for each episode, the efficiency of the robot's movement and plotted it over time. Figure 18 shows some of the obtained results for three of the tested terrains, the plain ones. It can be seen that, in most of them, the agent can correctly identify the movement that leads to high efficiency and the research is quite fast, considering that it usually takes around 1 to 2 minutes of simulation.

It can be also seen, in other images, that the episode length is lower than 180 seconds, which is the standard one, sometimes it is due to the fact that the system is actually able to reach the target and these are the episodes that produce the best results (they are usually the one ending at 120 s more or less), on the other hand, there are some cases in which the robot falls already at the beginning of the simulation and this is a problem we found more rarely by increasing the number of training timesteps. In fact, we started the training with 5000 and 50000 timesteps, but the number of falls was actually a higher percentage (around 50% and 30% of the total tested episodes); with 500000, we succeeded in falling once every 10 episodes, but we weren't able to go below this percentage.

Also, another aspect we found is that the learned algorithm tends to work always on low frequencies (<5 Hz), even if we start with high ones since these ones are responsible for very low power consumed by the actuators and still produce a minimum velocity of the system. Anyway, even if higher frequency performs very fast movements, they usually have a bigger power consumed, the efficiency for this still remains lower than the one related to low frequencies.

The last aspect we want to talk about is the computational cost of the simulation. By reducing the number of control variables to this few amount (just two variables) and by using a training algorithm that is complex but quite fast if related to a NN with just 13 nodes in two layers, we obtain a very fast control algorithm and the training could result in a small amount of time consumed. Everything said above is valid if some criteria are actually met, i.e. the algorithm trains fast with certain initial conditions.

Of all the components of the algorithm, the real-time consumption is the simulation itself and the step function. In the step, as expressed before, the equation of motion is applied to the robot and propagated to obtain the behavior of the system in reaction to external forces and moments. To have a system that is as close as possible to the real one, we need to have a timestep small enough to actually take care of the frequency of the actuation input. For example, let's say we have an input force that moves at 100 Hz, i.e. that each $(1/100)s = 0.01s$ the action repeats itself periodically, if the timestep is of about 0.01 or even 0.005, the actuation input will be constant or a square wave function from -1 to 1, we needed to have a timestep which was considerably for having a smooth actuation signal that resembles the original one, that is why we opted for a timestep frequency related, of $1/(500f)$, where f is the frequency of actuation. This will allow us to have a correct input signal and a timestep small enough to correctly integrate the motion equation. The drawback is, anyway, that when using high frequencies for training the system, each step will have a higher number of computations, making the simulation very slow.

For this reason, whenever the simulation started with an initial frequency of 1Hz at the beginning of each episode, the train would be fast, for example, a 5000 steps training would take less than 5 minutes; but if we impose higher frequencies, the computational time will grow and, by starting with a frequency of 150 Hz, we will need up to 5 hours to run a 5000 steps training.

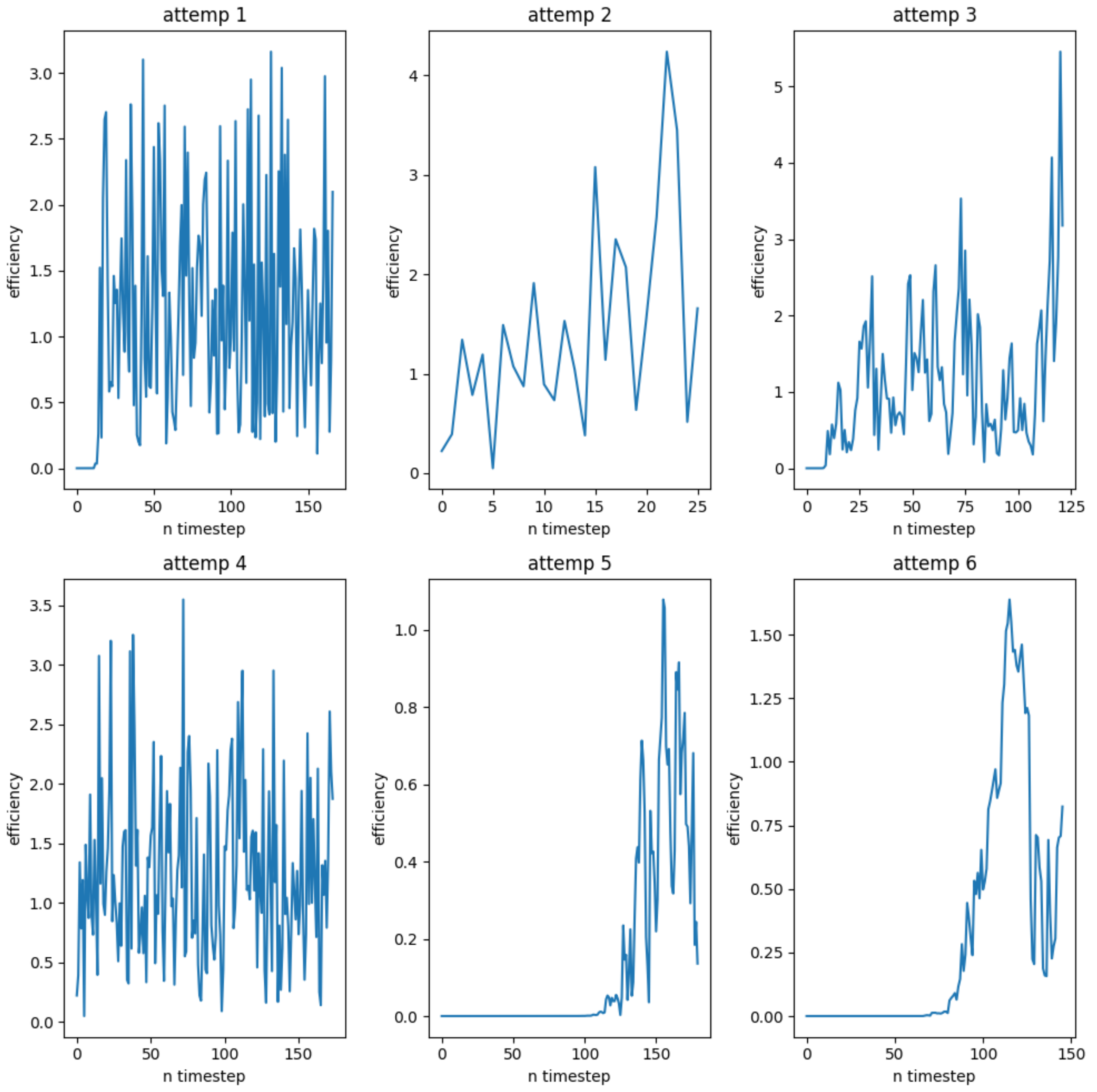


Figure 18: Efficiencies result of the trained model on different attempts on different surfaces.

CHAPTER 6

CLOSING REMARKS AND FUTURE WORK

6.1 REMARKS

This paper presents a novel approach to control microbots using just two variables: frequency and phase shifts. These two parameters enable full control over the robot's position and direction by also implementing, at the same time, a very simple controller due to the small number of controlled variables.

The controller successfully finds the tuple frequency angle that best optimizes the movement efficiency together with identifying the resonance frequency depending on the surface on which the motion happens. A physical model of the robot was presented together with a framework that could be used in the future for simulating the system and implementing different control strategies.

Overall the thesis presents a foundation work on the control of these tiny microbots, by obtaining some great results in terms of improvement of the efficiency of the overall movement of the robot, the architecture presented here paves the way for the implementation of a structure that could be used in a wide variety of other systems by making little changes and adapting to its intrinsic characteristics.

6.2 FUTURE WORKS

Since this thesis presents a foundation work, several future research directions may be explored and many details of this work have to be improved. Firstly, the current work is confined to simulations. In this regard, the object of future research can be implementing a better characterization of the physical environment by using real devices as a reference, having simulation results that are closer to the real system, and, therefore, having a more reliable platform for the implementation of other algorithms. Also, an experiment can be thought of where the logic is trained on the real device to have a clearer characterization of the computational power needed and a definition of the overall hardware components needed to implement an on-board controller to make the robot fully autonomous.

Future research could explore advanced algorithms or hybrid approaches to improve trajectory tracking performance in real systems. In this thesis, we wanted to demonstrate that working with learning the best movement frequency was something possible and this is what we achieved, but next work could dive into more complex tasks to be handled.

Furthermore, the results presented here are related to a single microbot, but extending the results to a swarm of microbots is a promising avenue. By enabling robots to operate independently and subsequently merge their findings, the computational cost of determining optimal frequencies could be significantly reduced. This approach could enhance the efficiency and scalability of microbot systems, allowing for more complex and adaptive behaviors in swarm applications.

CITED LITERATURE

1. Stable-Baselines3: Reinforcement learning algorithms.

2. Kabutz, H., Hedrick, A., McDonnell, W. P., and Jayaram, K.: mclari: a shape-morphing insect-scale robot capable of omnidirectional terrain-adaptive locomotion in laterally confined spaces. In 2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 8371–8376. IEEE, 2023.

3. Kabutz, H. and Jayaram, K.: Design of clari: A miniature modular origami passive shape-morphing robot. Advanced Intelligent Systems, 5(12):2300181, 2023.

4. Motion, A. I. and Lab, R.: Hamr-jr: small and dextrous. presented at icra 2020, 2020. YouTube video.

5. Swarmrobot.org: Swarmrobot.org: Micro-robotic platforms and swarm robotics, 2023. Accessed: 2024-08-03.

6. Haldane, D. W., Yim, J. K., and Fearing, R. S.: Repetitive extreme-acceleration (14-g) spatial jumping with salto-1p. In 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 3345–3351. IEEE, 2017.

7. Beetz, M., Mösenlechner, L., and Tenorth, M.: Cram—a cognitive robot abstract machine for everyday manipulation in human environments. In 2010 IEEE/RSJ international conference on intelligent robots and systems, pages 1012–1017. IEEE, 2010.

8. Shukla, P., Muralidhar, A., Iliev, N., Tulabandhula, T., Fuller, S. B., and Trivedi, A. R.: Ultralow-power localization of insect-scale drones: Interplay of probabilistic filtering and compute-in-memory. IEEE transactions on very large scale integration (VLSI) systems, 30(1):68–80, 2021.

9. Shukla, P., Sureshkumar, S., Stutts, A. C., Ravi, S., Tulabandhula, T., and Trivedi, A. R.: Robust monocular localization of drones by adapting domain maps to depth prediction inaccuracies. In ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 1–5. IEEE, 2023.

10. Shukla, P., Nasrin, S., Darabi, N., Gomes, W., and Trivedi, A. R.: Mc-cim: Compute-in-memory with monte-carlo dropouts for bayesian edge intelligence. IEEE Transactions on Circuits and Systems I: Regular Papers, 70(2):884–896, 2022.

11. Darabi, N., Shukla, P., Jayasuriya, D., Kumar, D., Stutts, A. C., and Trivedi, A. R.: Navigating the unknown: Uncertainty-aware compute-in-memory autonomy of edge robotics. In 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1–6. IEEE, 2024.

12. Lee, C., Rahimifard, L., Choi, J., Park, J.-i., Lee, C., Kumar, D., Shukla, P., Lee, S. M., Trivedi, A. R., Yoo, H., et al.: Highly parallel and ultra-low-power probabilistic reasoning with programmable gaussian-like memory transistors. Nature Communications, 15(1):2439, 2024.

13. Tayebati, S., Tulabandhula, T., and Trivedi, A. R.: Sense less, generate more: Pre-training lidar perception with masked autoencoders for ultra-efficient 3d sensing. arXiv preprint arXiv:2406.07833, 2024.

14. Darabi, N., Tayebati, S., Ravi, S., Tulabandhula, T., Trivedi, A. R., et al.: Starnet: Sensor trustworthiness and anomaly recognition via approximated likelihood regret for robust edge autonomy. arXiv preprint arXiv:2309.11006, 2023.

15. Shylendra, A., Shukla, P., Mukhopadhyay, S., Bhunia, S., and Trivedi, A. R.: Low power unsupervised anomaly detection by nonparametric modeling of sensor statistics. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 28(8):1833–1843, 2020.

16. Suleiman, A., Zhang, Z., Carlone, L., Karaman, S., and Sze, V.: Navion: A 2-mw fully integrated real-time visual-inertial odometry accelerator for autonomous navigation of nano drones. IEEE Journal of Solid-State Circuits, 54(4):1106–1119, 2019.

17. Suleiman, A., Zhang, Z., Carlone, L., Karaman, S., and Sze, V.: Navion: A fully integrated energy-efficient visual-inertial odometry accelerator for autonomous navigation of nano drones. In 2018 IEEE symposium on VLSI circuits, pages 133–134. IEEE, 2018.

18. Jayaram, K., Shum, J., Castellanos, S., Helbling, E. F., and Wood, R. J.: Scaling down an insect-size microrobot, hamr-vi into hamr-jr. In 2020 IEEE International Conference on Robotics and Automation (ICRA), pages 10305–10311, 2020.

19. Prasad, H. K. H., Hatton, R. L., and Jayaram, K.: Geometric mechanics of contact-switching systems. IEEE Robotics and Automation Letters, 8(12):8343–8349, 2023.

20. Doshi, N., Jayaram, K., Goldberg, B., and Wood, R. J.: Phase control for a legged micro-robot operating at resonance. In 2017 IEEE International Conference on Robotics and Automation (ICRA), pages 5969–5975. IEEE, 2017.

21. Castillo, G. A., Weng, B., Zhang, W., and Hereid, A.: Reinforcement learning-based cascade motion policy design for robust 3d bipedal locomotion. IEEE Access, 10:20135–20148, 2022.

22. Duisterhof, B. P., Krishnan, S., Cruz, J. J., Banbury, C. R., Fu, W., Faust, A., de Croon, G. C., and Reddi, V. J.: Learning to seek: Autonomous source seeking with deep reinforcement learning onboard a nano drone microcontroller. arXiv preprint arXiv:1909.11236, 2019.

23. Todorov, E., Erez, T., and Tassa, Y.: Mujoco: A physics engine for model-based control. In 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 5026–5033. IEEE, 2012.

24. Animal-Inspired Motion and Robotics Lab: Cad_mclari, 2024. GitHub repository.

25. Todorov, E., Erez, T., and Tassa, Y.: Mujoco ant model. `http://www.mujoco.org/`, 2012. XML model file: ant.xml.

26. Sutton, R. S. and Barto, A. G.: Reinforcement learning: An introduction. MIT press, 2018.

27. Mitchell, T. M. and Mitchell, T. M.: Machine learning, volume 1. McGraw-hill New York, 1997.

28. Hinton, G. and Sejnowski, T. J.: Unsupervised learning: foundations of neural computation. MIT press, 1999.

29. Bellman, R.: A markovian decision process. Journal of mathematics and mechanics, pages 679–684, 1957.

30. Rosenblatt, F.: The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review, 65(6):386, 1958.

31. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. nature, 518(7540):529–533, 2015.

32. Raffin, A., Hill, A., Ernestus, M., Gleave, A., Kanervisto, A., and Dormann, N.: Stable baselines3. `https://github.com/DLR-RM/stable-baselines3`, 2021.

33. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W.: Openai gym, 2016.

34. Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., et al.: Array programming with NumPy. Nature, 585(7825):357–362, 2020.

35. Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y.: Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

36. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M.: Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, 2013.

37. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

38. Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S.: 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.

39. Goldberg, B., Doshi, N., and Wood, R. J.: High speed trajectory control using an experimental maneuverability model for an insect-scale legged robot. In 2017 IEEE International Conference on Robotics and Automation (ICRA), pages 3538–3545. IEEE, 2017.

VITA

NAME	Luca Russo
EDUCATION	B.Sc. Aerospace Engineering, Politecnico di Torino, Italy. July 2022. M.Sc. in Mechatronics Engineering, Politecnico di Torino, Italy. Expected October 2024. M.Sc. Electrical and Computer Engineering, University of Illinois Chicago, Chicago, IL, USA. Expected December 2024.
SUBMITTED CONFERENCE PAPER	Luca Russo, Elyssa Chandler, Kaushik Jayaram, Amit Ranjan Trivedi "Dynamic Resonance Frequency Identification for Energy Efficient Legged Movement of Microbots" Manuscript under review for the 2024 Annual Conference of the IEEE International Conference on Control,Mechatronics and Automation (ICCMA, 2024).