Evolutionary robotics

Group T

Tifaine Mezencev
EPFL Robotics Master Student
Lausanne
tifaine.mezencev@epfl.ch
325347

Zoé Frottier
EPFL Robotics Master Student
Lausanne
zoe.frottier@epfl.ch
341115

***Abstract*—This project investigates methods to improve quadruped locomotion, with a focus on enabling a robot to navigate stairs and other challenging environments. By combining Central Pattern Generators (CPGs), Proportional-Derivative (PD) control, and Deep Reinforcement Learning (DRL), we developed and tested multiple locomotion strategies.**

The primary success was achieved through a hybrid approach using Joint PD control and DRL, with a two-stage training process: first, training on flat terrain to develop stable, straight-line walking, followed by training specifically designed for stair traversal. Additionally, we trained a separate policy using a combination of CPG and DRL to handle slopes and environments with random obstacles, as with CPG control the robot's movements looked more natural but less adapted to stair climbing. Our results show the advantages and challenges of different control methods for making quadruped robots move and offer ideas for creating robots that can adapt to various tasks.

I. Introduction

Robotics is making great progress in creating robots that move like animals. Quadruped robots are becoming more advanced thanks to techniques inspired by nature and modern technology. This project focuses on two methods to improve/model how these robots move: Central Pattern Generators (CPGs) and Deep Reinforcement Learning (DRL).

In the first part, we used CPGs, which are models inspired by how animals naturally control their movements with an oscillatory motion. These allow the robot to perform various walking styles, or gaits, such as walking and trotting. By fine-tuning parameters like step size and stance frequency, we tried to make the locomotion smooth and efficient on flat terrain.

The second part leverages DRL to teach the robot how to handle more complex challenges like slopes and stairs. Using algorithms like Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC), we created a framework for the robot to learn through trial and error.

This project was built on the code from the Robotics course MICRO-507, titled "Legged Robots." We expanded and refined this structure to meet the demands of our project. For the CPG component, we tuned parameters to get stable locomotion at varying speeds, verifying that the performance remains consistent and reliable while using different gaits. For the DRL component, most of our effort went into designing and modifying the action space, observation space, and, most importantly, the reward function. The reward function was the main object shaping the robot's learning process and results, and we drew inspiration from a paper by Bellegarda and Ijspeert that integrated CPG and RL frameworks [1].

To validate our methods, we trained the DRL model multiple times, experimenting with different configurations to identify the most effective approach. While our main objective was to make the robot walk on stairs successfully, we also tested its performance on other (simpler) environments, such as slopes. We explored hybrid approaches, combining DRL with CPG or alternative control strategies. The outcomes of these experiments are discussed in this report.

II. Central Pattern Generators

A. CPG states

We implemented four different gaits using CPG: Trot, Walk, Pace, and Bound. The parameters of the CPG visualized in the figures include the amplitude, the phase, and their derivatives with respect to time.

We generated plots of the CPG parameters for each gait by sampling data at the end of the simulation to ensure the gait had stabilized. The green curve represents the phase θ. Notably, the slope of θ changes at $\theta = \pi$, indicating the transition between the swing phase (when the leg is off the ground) and the stance phase (when the leg is on the ground). $\dot{\theta}$ oscillates between two values: ω_{swing} and ω_{stance}. The synchronization of the legs is specific for each gait.

The next figures shown represent the plots of the gait parameters optimized for stability and for medium/maximum speed. The CPG parameters visualized include amplitude, phase, and their derivatives with respect to time.

TROT The legs are synchronized in a diagonal way (front left with rear right, and inversely).

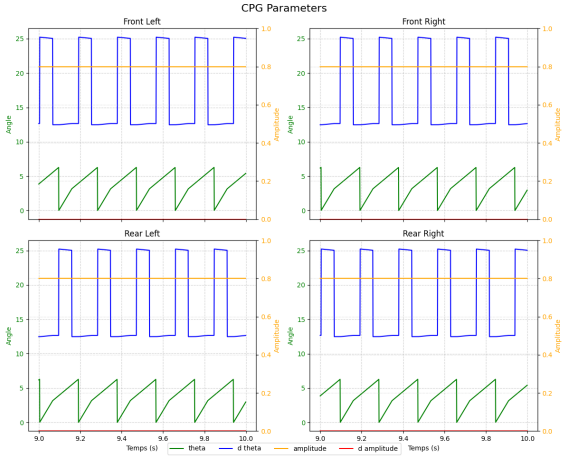
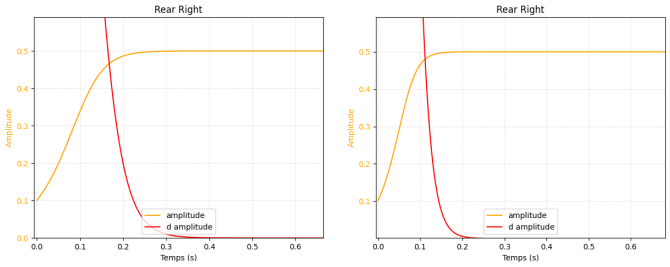


Fig. 1: CPG parameters for TROT gait at Medium speed

The convergence of the amplitude occurs in the very beginning of the simulation, making it less visible in the main plot where we only show the last few seconds. Figures 2a and 2b show the amplitude convergence (at the beginning of the simulation) as a function of the convergence factor α. A higher α results in faster convergence.



(a) $\alpha = 30$ (b) $\alpha = 50$

Fig. 2: Comparison of amplitude convergence for different values of α for the TROT gait

WALK The walking motion is more complex, the synchronization of the legs can be seen in the figure. There are always at least 2 legs touching the ground.

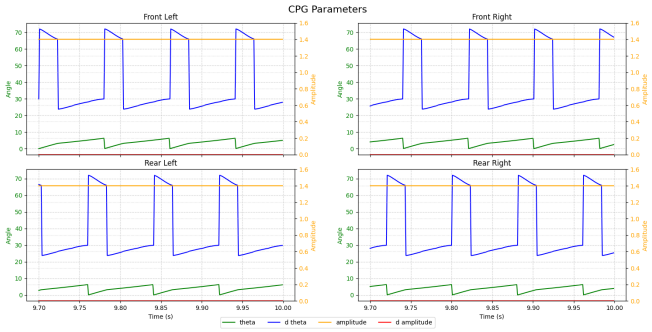


Fig. 3: CPG parameters for the WALK gait at Max speed

PACE In this gait, the legs are synchronized by side (right side moves together, same for left).

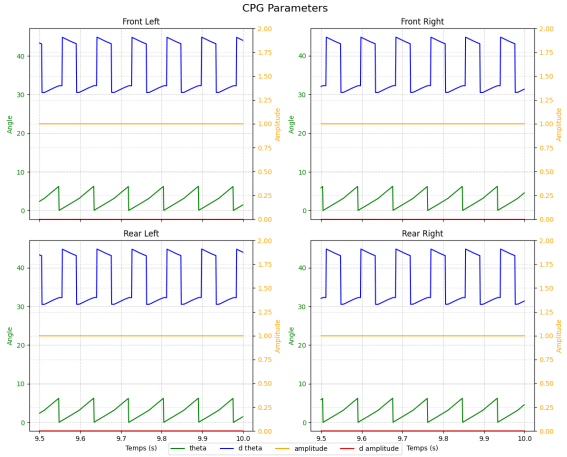


Fig. 4: CPG parameters for the PACE gait at Max speed

BOUND

Figure 5 illustrates the CPG parameters for the bound gait at maximum speed. Here, the front legs are synchronized together, same for the rear legs.

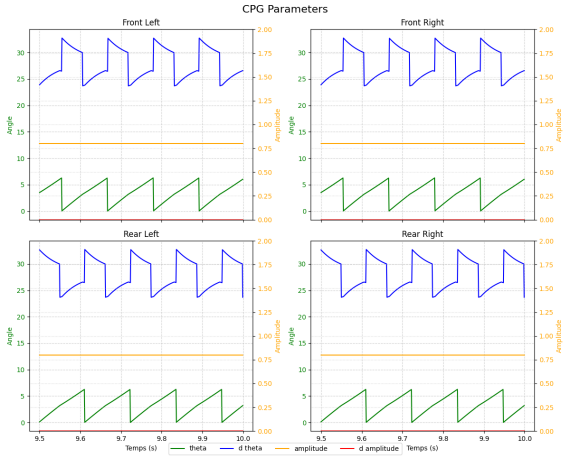
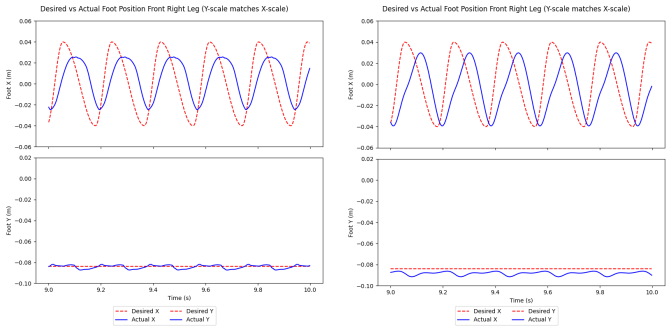


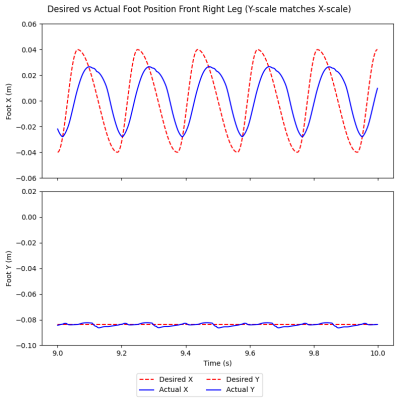
Fig. 5: CPG parameters for BOUND gait at Max speed

B. Desired VS Actual foot position and joint angle for TROT gait

Figure 6 shows the difference between the Desired Foot Position (DFP) and the Actual Foot Position (AFP) of the Front Right Leg for various controllers. We initially tuned a controller combining both joint PD and Cartesian PD to achieve optimal results. To understand each controller's impact, we also tested them individually. The results indicate that joint PD control has a greater influence than Cartesian PD with the selected gains. Therefore, we use joint PD as the primary controller and Cartesian PD for minor adjustments. It's important to note that we did not separately tune each PD controller; the results reflect weights from both joint and individual tuning. Additionally, while Cartesian PD shows better tracking of the desired x-position, this occurs because the robot has fallen and is no longer interacting with the ground, simplifying the tracking process.

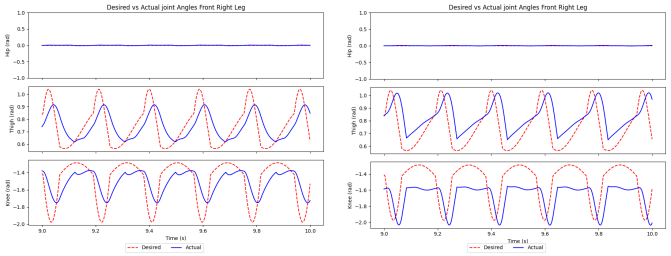


(a) Joint PD (b) Cartesian PD

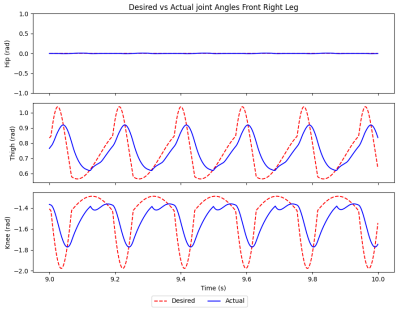


(c) Joint PD + Cartesian PD

Fig. 6: Comparison of Desired vs Actual Foot Positions: Joint PD, Cartesian PD, and Both for TROT gait at Medium speed



(a) Joint PD (b) Cartesian PD



(c) Joint PD + Cartesian PD

Fig. 7: Comparison of Desired vs Actual Joint Angles: Joint PD, Cartesian PD, and Both for TROT gai=t at Medium speed

Here in figure 7 we see again how the main effect comes again from the joint PD control (we can see that just by looking at how similar figure 7a and figure 7care). Here again, the Cartesian PD plot is made while the robot has fallen, making the tracking of the desired angle look better than what it would actually be.

1) PD Gains in TROT Medium speed: We combined joint PD and Cartesian PD. The gains used are:

For the Joint PD:

- $K_p = [200,\ 150,\ 150]$
- $K_d = [5,\ 5,\ 5]$

For the Cartesian PD:

- CartesianK$_p$ = diag$([500] \times 3)$
- CartesianK$_d$ = diag$([20] \times 3)$

Here, `diag` denotes a diagonal matrix with the specified values repeated three times.

These parameters were determined iteratively. Increasing K_p improves tracking performance but reduces system stability. Conversely, increasing K_d enhances stability but degrades tracking accuracy. We tried to balance tracking performance and stability by visually checking the simulation to find the best parameters. The three values of K_p and K_d correspond to the motors controlling the hip, thigh, and knee, respectively.

C. Results

1) Hyperparameters: The hyperparameters we tuned include ω_{swing}, ω_{stance}, μ (amplitude), desired step length, gains (Kp, Kd, Cartesian Kp, Cartesian Kd), ground clearance, ground penetration, α (amplitude convergence factor), and coupling strength.

However, the most important parameters to tune were ω_{swing} and ω_{stance}, which determine the proportion of time the leg is in contact with the ground. ω_{swing} represents the frequency during the swing phase (no contact with the ground), while ω_{stance} represents the frequency during the stance phase (contact with the ground). We also adjusted the desired step length to modify the average body speed. The other parameters were mainly used to stabilize the robot.

Parameters	**Trot**	**Walk**	**Pace**	**Bound**
μ	0.8^2	1.4^2	1	0.8^2
ω_{swing}	$5 \times 2\pi$	$11 \times 2\pi$	$5 \times 2\pi$	$5 \times 2\pi$
ω_{stance}	$7 \times 2\pi$	$4.3 \times 2\pi$	$7 \times 2\pi$	$4 \times 2\pi$
α	50	500	50	50
Coupling strength	0.5	1	1	3
Ground clearance	0.08	0.07	0.06	0.09
Ground penetration	0.02	0.01	0.01	0.02
Desired step length	0.185	0.075	0.12	0.16
Joint kp	$[150, 150, 150]$	$[100, 100, 100]$	$[180, 180, 300]$	$[200, 30, 300]$
Joint kd	$[5, 5, 5]$	$[2.7, 2.7, 2.7]$	$[7, 9, 5]$	$[1, 6, 8]$
Cartesian kp	diag$([600] * 3)$	diag$([500] * 3)$	diag$([500] * 3)$	diag$([600] * 3)$
Cartesian kd	diag$([10] * 3)$	diag$([20] * 3)$	diag$([20] * 3)$	diag$([20] * 3)$
Max speed in x [m/s]	4.22	1.84	1.277	1.18

TABLE I: Hyperparameters for different gaits at MAX speed

We achieved both the maximum and minimum speeds using the TROT gait, which proved to be the most stable. In contrast, maintaining stable locomotion with the PACE gait at minimum speed was particularly difficult, as the lateral symmetry of the gait made it difficult for the robot to maintain equilibrium during slow movements.

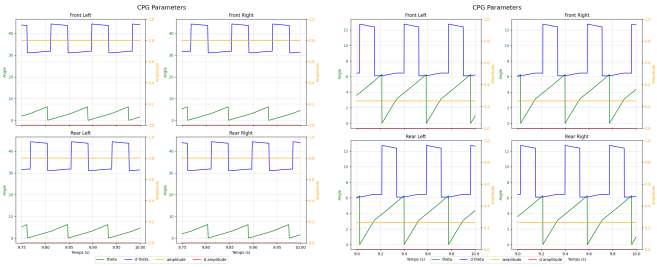
Parameters	Trot	Walk	Pace	Bound
μ	0.5^2	0.4^2	0.5^2	0.8^2
ω_{swing}	$1 \times 2\pi$	$1.4 \times 2\pi$	$3 \times 2\pi$	$3 \times 2\pi$
ω_{stance}	$1 \times 2\pi$	$0.5 \times 2\pi$	$3 \times 2\pi$	$2 \times 2\pi$
α	30	50	30	30
Coupling strength	0.5	1	1	3
Ground clearance	0.08	0.04	0.03	0.07
Ground penetration	0.02	0.01	0.01	0.02
Desired step length	0.0038	0.04	0.05	0.05
Joint kp	$[200, 150, 150]$	$[100, 100, 100]$	$[180, 180, 300]$	$[150, 30, 200]$
Joint kd	$[5, 5, 5]$	$[2, 2, 2]$	$[7, 9, 5]$	$[1, 6, 8]$
Cartesian kp	diag($[500] * 3$)	diag($[500] * 3$)	diag($[500] * 3$)	diag($[600] * 3$)
Cartesian kd	diag($[10] * 3$)	diag($[20] * 3$)	diag($[20] * 3$)	diag($[20] * 3$)
Min speed in x [m/s]	0.0005	0.009	0.3779	0.179

TABLE II: Hyperparameters for different gaits at MIN speed

2) Duty Cycle Ratio for TROT Gait: Definition:

$$\text{Duty Cycle} = \frac{\text{Stance Duration}}{\text{Stride Duration}} \tag{1}$$

Stride Duration is the total time for a complete cycle. Stance Duration is the time the foot is in contact with the ground.



(a) MAX speed (b) MIN speed

Fig. 8: CPG parameters for the TROT gait at MAX and MIN Speeds

Figure 8 shows the stance and swing durations, identified by the constant values of $\dot{\theta}$ (in blue). For instance, at maximum speed, the higher derivative value ($\omega_{\text{swing}} = 7 \times 2\pi$) corresponds to the stance phase, while the lower value ($\omega_{\text{swing}} = 5 \times 2\pi$) corresponds to the swing phase. From these graphs, we derive the following durations:

Parameters	Values at MAX speed	Values at MIN speed
Stance Duration [s]	0.035	0.247
Swing Duration [s]	0.050	0.127
Step Duration [s]	0.085	0.374
Duty Cycle	0.41	0.66

TABLE III: Duty cycle and stride durations at MAX and MIN speed

Metric	TROT	WALK	PACE	BOUND
CoT at MAX Speed	0.79246	0.68614	4.38632	6.91208
CoT at MIN Speed	36.99156	7.39531	35.88216	13.53178
Energy (MAX) [J]	3242.898	1267.381	5940.434	7289.986
Energy (MIN) [J]	170.816	87.320	7146.044	5084.334
Distance (MAX) [m]	33.495	15.117	11.085	8.633
Distance (MIN) [m]	0.038	0.097	1.630	3.075

TABLE IV: Cost of Transport (CoT), energy used, and distance traveled for different gaits at MAX and MIN speeds. Values calculated using the last 8 seconds of the simulation.

3) Cost of Transport: Definition:

$$\text{CoT} = \frac{E}{mgd}$$

where E is the total energy consumed, mg is the robot's weight, and d is the distance traveled. We calculated the CoT using data from the last 8 seconds of the simulation to allow the robot sufficient time to stabilize after starting.

It is important to note that the gaits PACE and BOUND were not prioritized during the optimization. Instead, we focused our efforts on TROT and WALK, as these gaits are more likely to be used in future implementations with the RL-based CPG. Therefore, the CoT results for PACE and BOUND are less efficient compared to those of TROT and WALK.

The results show that the CoT values at MAX speed are always lower than those at MIN speed. This happens because the CoT formula divides by d, and the distances traveled at MIN speed were very small. As a result, the CoT becomes generally much larger for MIN speed.

We worked on improving the CoT for the WALK gait. By keeping all other parameters the same as for MAX speed but increasing the joint K_d values from 2.7 to 5, we reduced the CoT from 0.68614 to 0.28598. This change made the gait a bit slower but much more energy-efficient (using one third of the energy) and comparable to the CoT of an animal.

Why Increasing K_D Made It More Efficient

Increasing the joint K_D values improves how the robot moves in two main ways:

1) **Less Wasted Movement:** Higher K_D values reduce joint oscillations and overshooting, making the movements smoother and more stable. This saves energy.
2) **Better Energy Use:** Smoother movements mean more energy is used for forward motion instead of correcting instability or vibrations.

However, if K_D is too high, the robot can become too slow and less effective. It's important to find the right balance.

D. Controllers extension

Our controllers can be extended by incorporating feedback loops and descending control signals, which would make locomotion more adaptive and robust in varying environments.

One approach to achieve this is through Virtual Model Control (VMC). VMC improves posture stability and terrain adaptability by using virtual springs to stabilize the robot's roll and pitch on uneven terrain. Additionally, stumbling reflexes, triggered by contact sensors, make the robot able to adjust its leg positions when detecting obstacles, preventing collisions and maintaining smooth motion. Furthermore, another feature of VMC called directional control uses virtual forces to correct the robot's direction [2].

In addition to VMC, feedback mechanisms can be used. For example, force feedback helps the robot adapt its leg movements to changes in the terrain (like friction or incline), while proprioceptive feedback, using joint angle and velocity data, improves the accuracy of gait execution. Descending

control signals can dynamically adjust CPG parameters, such as speed, step length, and gait type, allowing the robot to transition between different gaits and respond quickly to environmental changes [3].

Simulation studies of these extensions have shown that, for example, combining CPG with VMC gives a 96% success rate in handling randomized terrains, improving stability, adaptability, and energy efficiency [2].

Even though these extensions were not implemented in our project, they show us valuable directions for future work.

III. Deep Reinforcement Learning

A. Action Space

During our training, we tested different environments to understand their characteristics. Using the *CPG motor control mode*, we successfully navigated the SLOPE environment provided. Initially, we trained the robot on flat ground with no obstacles, using the default `reward_fwd_locomotion` function. We then continued the training by reusing the previous simulation and applying the reward functions we designed for the STAIRS environment to train the robot in the SLOPE environment. This approach allowed us to achieve stable and efficient performance, enabling the robot to traverse slopes with ease.

We then decided to focus on the STAIRS environment, as it presents more of a challenge.

To complement this environment, we selected the *Proportional-Derivative (PD) control mode* for motor actuation. The PD control mode provides direct joint position control for our 12 motors, enabling smooth and precise adjustments to the robot's leg movements, which is critical for climbing stairs. Although CPG offers fluid and stable movement, we preferred PD control for its adaptability to uneven terrain, like in our case for the stairs.

B. Observation Space

In the `DEFAULT` mode, the observation space includes joint angles, joint velocities, and ground orientation, bounded by the physical limits of the robot, which are enough for our robot's locomotion on flat terrain.

Standardizing the observation space ensures that the different measurements have comparable scales. If these dimensions (joint angles and speeds, for example) are not standardized, some of them may overpower others simply because their scale is larger, leading to an imbalance in learning. Although practical for basic tasks, this method lacks the precision needed for difficult terrain such as stairs.

For our observation space we significantly improved the observation space `LR_COURSE_OBS` to better deal with the complexities of crossing stairs. In addition to the DEFAULT parameters, we have added:

- **CPG states:** The oscillatory parameters $r, \dot{r}, \theta, \dot{\theta}$ give an understanding of the gait patterns generated by the CPG. These were added only for the DRL that uses CPG for locomotion
- **Contact measures:** Valid/invalid contact counts, normal forces for each foot, and contact Boolean arrays.
- **Orientation Parameters:** Orientation data ensures precise body posture control during stair climbing.

We noticed these additions improve the robot's understanding of its environment and seem to make the training faster. The margins were defined to be sufficiently wide (in cases we did not clearly now the limit) to allow the robot to perform the necessary movements while staying within its physically achievable limits.

C. Reward Function

The reward function is *the* essential element of reinforcement learning, designed to facilitate and guide the robot's training. In this work, several reward functions have been combined to guide our robot toward the desired goals while penalizing unwanted actions. Below is a description of each reward function we used:

1) **Velocity Tracking Reward :**
This function rewards the agent for maintaining a desired forward velocity along the x-axis, encouraging efficient locomotion.
This is in general the most rewarding function, and the one that will mostly define the overall curve of our total reward function.
$R_{\text{velocity}} = 0.35 \cdot \exp\left(-\frac{(\text{desired\_}\nu_x - \text{current\_}\nu_x)^2}{0.1}\right) - 0.03$

2) **Progress Reward :**
This function is slightly complementary to the *Velocity Tracking Reward*, as it no longer focuses on the robot's target speed over time, but directly encourages forward movement by rewarding the distance covered along the desired trajectory, no matter the speed profile.
$R_{\text{progress}} = 10 \cdot (x_t - x_{t-1})$

3) **Yaw Tracking Reward :**
Encourages the robot to maintain a desired yaw orientation, penalizing for turning in the other directions.
This function has the same level of importance as the *Progress reward* function and has been manually tuned to give it a similar weight, which encourages the robot to stand upright in order to maintain its stability and orientation towards its target, contributing to more natural movements.
$R_{\text{yaw}} = 0.1 \cdot \exp\left(-\frac{(\text{desired\_yaw} - \text{current\_yaw})^2}{0.25}\right)$

4) **Smoothness Reward :**
Promotes smooth transitions in motor angles, velocities, and torques while maintaining consistent contact forces by penalizing abrupt changes and excessive torques. This function plays an important role in the robot's long-term stability. However, we assigned it minimal weight to avoid impeding the robot's mobility, allowing it to serve

as a supplementary function for smoothing the robot's movements.

$$R_{\text{angles}} = -0.5 \left(\exp\left(\frac{\|\theta_t - \theta_{t-1}\|}{6} \right) - 1 \right) \in [-0.5, 0]$$

$$R_{\text{velocity}} = 0.01 \cdot \left(-\exp\left(\frac{\|v_t - v_{t-1}\|}{10} - 4.5 \right) + 0.3 \right) \in [-0.5, 0.5]$$

$$R_{\text{force}} = -0.5 \cdot \text{normalized\_forces},$$

$$\text{normalized\_forces} = \frac{\text{actual\_force} - \text{min\_force}}{\text{max\_force} - \text{min\_force}}$$

$$\text{actual\_force} \in [\text{min\_force}, \text{max\_force}],$$

$$R_{\text{smoothness\_tot}} = 0.1 \cdot (R_{\text{angles}} + R_{\text{torque}} + R_{\text{force}})$$

5) **Drift Penalty :**
This function penalizes deviations from the center-line in the y-direction to get straight movement. The negative weight of this function varies a lot during the learning of our robot. At the beginning of the learning process, its penalty is very high due to the random and erratic movements of the robot, which often lead to lateral drift in the y-direction. But as learning progresses, this value becomes less and less important and its weight becomes negligible.
$R_{\text{drift}} = -0.05 \cdot |y|$

6) **Oscillation Reduction Reward :**
This reward minimizes oscillatory behavior in motor velocities by penalizing too frequent sign changes.
The same 4 motors (3 different motors in each leg) have their own 'desired oscillation' and their own 'margin', which were defined empirically by comparing the values obtained when our robots move according to different gaits defined in the CPG part (without RL) as described in the previous sections.

$$R_{\text{oscillation}} = \sum_{\text{groups}} 0.01 \cdot \left[\exp\left(-\frac{(\text{desired\_}\xi - \text{actual\_}\xi)^2}{\text{margin}} \right) - 0.4 \right]$$

ξ = number of oscillations during a certain period
margin = 1000, per default

As you can see, many reward functions are defined in the following form: $\text{Reward}(x) = a \cdot \exp\left(-\frac{(b-x)^2}{c}\right) + d$. This was inspired by the reward functions taken from the article of Bellegarda and Ijspeert [1]. It gives a reward in the form of a "revised Gaussian function". This allows a higher reward if the robot precisely follows the desired objective, but still rewards it if it deviates slightly from it, which may be necessary for the robot to rebalance itself, for example. Finally, the term d, which is often negative, has been added to reduce the reward function if the robot does not follow the objective for a sufficiently long time (if it gets stuck, for example).

We've used Geogebra's graphical visualization to fine-tune the rewards function to give us the expected rewards. Figure 9 gives an example of the function for the final speed visualized on Geogebra.

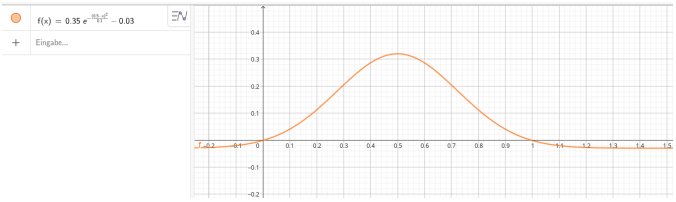


Fig. 9: Geogebra's function for the robot's speed

The parameters were defined either empirically by 'trial and error', or by comparing the values obtained when our robots move according to different gaits defined in the CPG part (without RL). The weights were defined according to the importance we wanted to give to each aspect/parameter that the robot had to learn.

Note that the reward functions were sometimes tuned, modified or not taken into account as we rerun the simulations. For example, after realizing that the smoothness reward function increased the difficulty of crossing stairs, as the robot tended to "freeze" one of its legs and use only 3 of them, we stopped using it for our latest simulations.

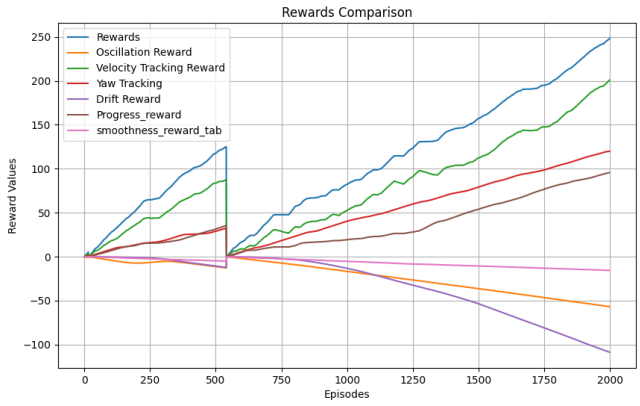


Fig. 10: Reward function comparison

Figure 10 illustrates the 'priorities' that we have given to each of the reward functions described above. It shows the rewards granted during two simulations (the second simulation starts at $550ms$ when all the reward functions are reset to 0 on the graph) after the training has run for around 3 million steps. The robot manages to cross the stairs that have been placed in its environment.

D. Deep RL Algorithms

PPO versus SAC:

Proximal Policy Optimization (PPO) is an on-policy algorithm that updates its policy based on recent data. It uses a clipping mechanism to prevent the policy update to deviate too much from the previous policy. It is simple to implement

and effective across various tasks.

Soft Actor-Critic (SAC), on the other hand, is an off-policy algorithm that can utilize data from a replay buffer. It includes an entropy term (that is set to 0 in the PPO algorithm) in the reward function to encourage exploration and policy randomness. SAC is very strong in tasks requiring a lot of exploration, but it can be more computationally demanding.

Our Choice of Algorithm:

For this project, we chose to use the PPO approach due to its simplicity and stability. We choose to target mainly the `STAIRS` environment. Our setup involves real-time collection of data (on-policy), making PPO's structure (which doesn't require replay buffers) a good fit. The on-policy approach aligns well with our environment, where data collection is straightforward, and the clipping mechanism updates smoothly despite the complex terrain.

Neural network Architecture:

The architecture of the neural network plays an important role in the success of our reinforcement learning. In our case, the neural network used is a Multi-Layer Perceptron (MLP). It consists of two fully connected layers (256 units each) that allow the model to learn the nonlinear mappings between the observations and actions required to walk on stairs.

The *policy network* is responsible for deciding which action to take at each step. The two 256-unit hidden layers help the network process the input observation (the robot's state) and output an action, which is then executed by the robot.

Hyperparameters choices:

We decided to keep most of the default hyperparameters provided by Stable Baselines3, as they enabled us to achieve a consistent improvement in the learning policy depending on the rewards function we provided to our simulation.

The most important hyperparameters for us are the ones that directly influence the learning stability, convergence, and the quality of the learning policy. Here is a description of some of the hyperparameters we consider to be the most important and how they benefit our learning policy.

- The *learning rate* determines how quickly the model adjusts its weights during training. A high learning rate leads to overshooting optimal solutions, while a low learning rate causes slow convergence. We started the first simulations with a learning rate of $1 \cdot 10^{-4}$ before gradually decreasing it as we reran our simulations based on previous optimizations while augmenting the difficulty of our environment (augmenting the stairs hight). This allowed us to tune more and more precisely the robots movements: first he simply managed to move forward, then he managed to climb the stairs without falling, more and more smoothly,...
- The *n_steps* hyperparameter, set to 4096, is also important as it determines the size of the rollout buffer, which affects how much data the algorithm gathers before performing an update.
- Like mentioned before, the PPO algorithm uses a clipping strategy to ensure the policy learning stays consistent and to prevent instability. The *clip_range* hyperparameter is set to 0.2, guaranteeing smooth learning updates without over-correction.
- *Gamma* is an hyperparameter that can be set from 0 to 1. A value close to 1, like the one chosen (0.99), gives priority to smooth movements and long-term planning, which are the kind of movements we want to prioritize for our quadruped robot.

E. Environment

After focusing on the `STAIRS` environment, we adjusted the `EPISODE_LENGTH` to range between 15 and 20 seconds, depending on the simulation, as the maximum time before the environment resets. This episode length makes it possible for the robot to navigate the stairs despite blockages at the start of the climb. It gives the robot an opportunity to realign or recover and "try again" if the simulation has not failed, enhancing the robot's learning and adaptability to different situations.

Once we defined most of our reward functions, we reran the simulation multiple times, each with 2-3 million steps, while progressively increasing the difficulty of the environment (stair heights of 0.03 meters, then 0.04 and 0.05 meters). This iterative process improved the robot's training and policy robustness, following the principle of the "Adaptive Terrain Curriculum" described in *Lee et al.*'s article [4]. Additionally, we conducted simulations with staircases of different heights and step counts to evaluate the robot's ability to navigate each staircase successfully in a single visualization.

F. Discussion and Results

As mentioned before, we used a PD motor control method combined with the STAIRS environment, which is quite complex. PD produces less stable results than CPG, leading to unpredictable outcomes in the robot's simulation.

This combination was deliberately chosen to test the limits of what reinforcement learning can achieve in challenging locomotion tasks. Although the PD control method is less stable than CPG, it gives greater flexibility and adaptability to meet the varying demands of stair climbing.

The reward functions guide the robot to move forward without deviating significantly from the center-line or pointing in the wrong direction, making it able to successfully navigate the stairs. It performs this reasonably well without falling. A big achievement is the robot's ability to maintain stability and continue its ascent even after encountering disturbances. The reward functions, designed to prioritize forward movement and directional accuracy, effectively guided the robot to traverse the stairs. In particular, the robot demonstrated an ability to recover from deviations, such as tripping or bouncing against steps, and realign itself to continue climbing.

However, a drawback of using PD is that the robot exhibits unpredictable and less fluid movements to progress. Depending on its reward functions, it may move erratically and unnaturally, using more energy, yet still manage to climb the stairs. We tried to enforce more natural movements by adding reward functions for movement fluidity and penalizing

excessive motor accelerations or velocity changes, but these increased the difficulty of stair climbing.

We acknowledge that this results in a high CoT (Cost of Transport). After computation, we observed values fluctuating around 3, depending on the run. Furthermore, transferring the simulation to reality presents challenges. Real-world hazards such as uneven terrain or varying friction coefficients, absent in our simulation, would complicate the transfer. Additionally, the rapid speed and acceleration changes exhibited by the robot in simulation to climb stairs could damage its motors or be difficult to replicate in real-world conditions.

Several improvements could be made to our simulation. Fine-tuning the reward functions to balance task performance, energy efficiency, and natural movement could improve fluidity. Introducing more realistic terrains, such as varying step heights across simulations and adding random disturbances and noise, would improve adaptability. To solve simulation-to-reality transfer, environment randomization and robustness tests could be incorporated to handle unexpected disturbances, guaranteeing smoother and more reliable performance in both simulation and real-world environments.

Figures 11 and 12 show two examples of the learning policy based on simulations, each with a total of 2 million steps. Figure 11 illustrates the early stages of the robot's learning process, where rapid learning and increasing rewards are observed. Figure 12 shows the fine-tuning of the reward policy after modifications to the environment difficulty and reward functions.

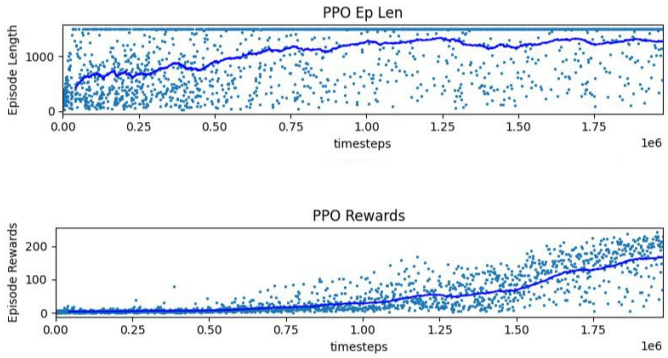


Fig. 11: Learning policy reward, early stage of the learning

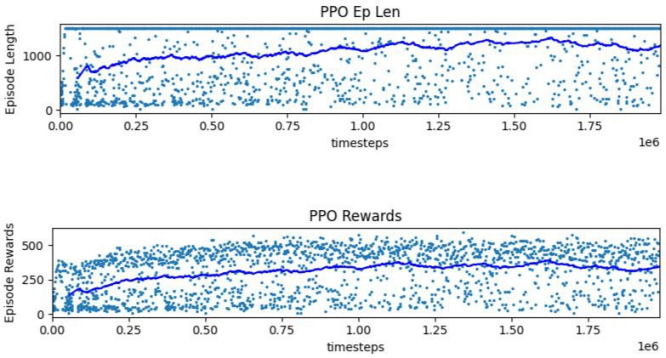


Fig. 12: Learning policy reward, fine-tuning of the learning

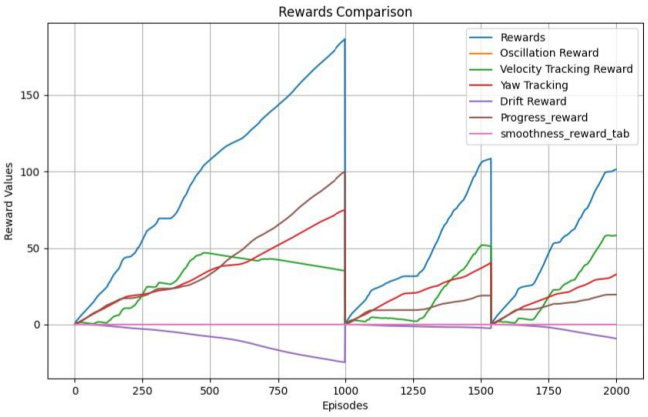


Fig. 13: Reward functions comparison when climbing high stairs

Figure 13 illustrates the reward of each reward function used in the last simulation to traverse a 13-step staircase with a staircase height of 0.05. We can see in the first green curve (velocity tracking reward) that the function gently descends after 500 ms. This is due to the fact that the robot has passed the steps and is continuing on a flat surface. It then goes faster than expected by the reward function, resulting in a negative reward as described in section *III.C.1*.
Similarly, the second curve shows that velocity tracking and progress reward stagnate at first, before increasing. This indicates that the robot first stumbled on the first few steps, before correctly starting its ascent a few seconds later. However, we can also observe in this second curve, by looking at the drift reward which is very close to 0, that the robot hasn't deviated at all from its trajectory and remains perfectly aligned with the x axis.

IV. Conclusion

This project was divided in two parts: Quadruped locomotion for different gaits behavior with a Central Pattern Generators (CPGs) and Deep Reinforcement Learning (DRL) in specific environments.

By implementing different gaits (TROT, WALK, PACE, BOUND), we demonstrated the flexibility and adaptability of CPGs, by optimizing parameters to obtain stable locomotion at high and low speeds. We then analyzed the various behaviors. Videos of each gait in action were included with this report.

The second part focused on using DRL to address a more challenging task: climbing stairs. By designing specific action and observation spaces, as well as a set of reward functions, we succeeded in training the robot to climb stairs despite the complexity of the environment. Gradually increasing the difficulty of the task during the training allowed us to fine-tune the policy and achieve a balance between stability, adaptability, and task performance.

This project helped us better understand the advantages and disadvantages of CPG and DRL for quadruped locomotion and why it is becoming more and more important in mobile robotics today. Our reward functions could be further improved

by increasing energy efficiency and incorporating simulation-to-reality transfer strategies to create robust, adaptive robots capable of moving across different terrains.

REFERENCES

[1] G. Bellegarda and A. Ijspeert, "CPG-RL: Learning central pattern generators for quadruped locomotion," IEEE Robotics and Automation Letters, vol. 7, no. 1, pp. 67–74, Jan. 2022.

[2] M. Ajallooeian, S. Pouya, A. Sproewitz, and A. J. Ijspeert, "Central pattern generators augmented with virtual model control for quadruped rough terrain locomotion," *2013 IEEE International Conference on Robotics and Automation*, Karlsruhe, Germany, pp. 3321–3328, May 2013.

[3] S. Coros, A. Karpathy, B. Jones, L. Reveret, and M. van de Panne, "Locomotion skills for simulated quadrupeds," *ACM Transactions on Graphics (TOG)*, vol. 30, no. 4, pp. 1–12, Jul. 2011.

[4] Lee et al, Learning quadrupedal locomotion over challenging terrain. Robotic Systems Lab, ETH-Zürich, Switzerland, 2020