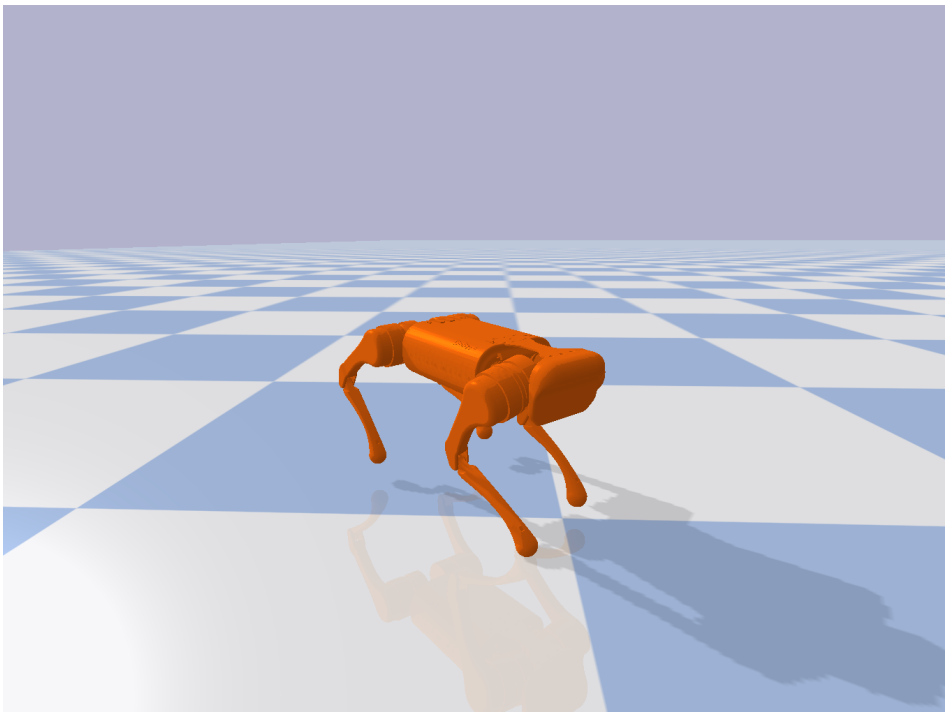
EPFL

MICRO-507 - Legged Robots

Autumn Semester 2021

Report on Project 2 :

Quadruped Locomotion with Central Pattern Generators and Deep Reinforcement Learning



Benjamin Mottis
Antoine Perrin
Matteo Righi

Contents

1 Introduction **1**

2 Central Pattern Generators **1**

2.1 Chosen CPG Parameters 1

2.2 Tuning of the PD Controller Parameters 2

2.3 Results 2

2.3.1 Plots for the WALK Gait 3

2.3.2 Plots for the TROT Gait 5

2.3.3 Plots for the PACE Gait 7

2.3.4 Plots for the BOUND Gait 9

2.3.5 Other Locomotion Metrics 11

2.4 Discussion 12

3 Deep Reinforcement Learning **13**

3.1 Observation Space 13

3.2 Action Space 13

3.3 Reward Function 15

3.4 Neural Network Hyperparameters 17

3.5 Environment 19

3.6 Results 21

3.7 Discussion 22

4 Conclusion **22**

5 References **24**

1 Introduction

The goal of this project is to produce smooth and stable locomotion on the Unitree A1 quadruped robot using two different approaches: central pattern generators (CPGs) and reinforcement learning (RL). The different control methods are developed using a code in Python which makes use of the Gym toolkit, and then tested in a PyBullet simulation.
The former method is inspired by [1] and uses a control architecture using sets of sinusoidal pattern generators at the limbs. The latter method uses existing RL algorithms to train a control policy which maximizes a chosen reward function.
In this report, the methodology for the choice and the tuning of the parameters is explained. The results obtained are presented, and routes for possible development in the future are discussed.

2 Central Pattern Generators

In this section, we will present the methodology and results obtained using a CPG control method, as described in [1].

2.1 Chosen CPG Parameters

We developed 4 different walking and running gaits, denoted hereon on out as WALK, TROT, PACE, and BOUND. These are shown in figure 1; note that the numbers on each limb represent the proportion of time that limb spends in contact with the ground at each cycle:

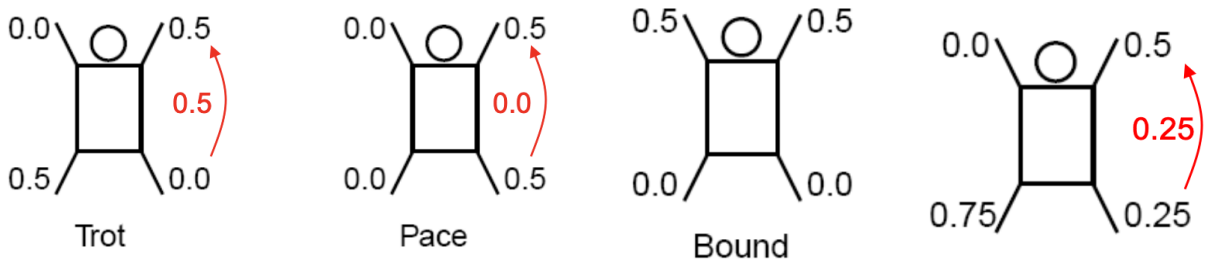


Figure 1: Different implemented gaits: TROT, PACE, BOUND, WALK)

We found that the most important parameters to achieve stable locomotion were the swing and the stance frequencies ω_{stance} and $\omega_s wing$. We started from the values suggested in the assignment document: a high value for ω_{stance} and 2-4 times that for ω_{swing}. We then tuned these parameters by hand to optimize the stability and the smoothness of the gait. We found that increasing both of these improved stability.

The final values which were used are:

WALK :	TROT :	PACE :	BOUND :
$\omega_{stance} = 18\pi$	$\omega_{stance} = 16\pi$	$\omega_{stance} = 24\pi$	$\omega_{stance} = 24\pi$
$\omega_{swing} = 40\pi$	$\omega_{swing} = 36\pi$	$\omega_{swing} = 60\pi$	$\omega_{swing} = 48\pi$

Surprisingly, we found that we were able to achieve the desired gaits without needing to tune other parameters than ω_{stance} and $\omega_s wing$.

2.2 Tuning of the PD Controller Parameters

Additionally, the proportional and derivative gains, Kp and Kd, of the two PD controllers were tuned to increase the speed of the robot.

For the CARTESIAN_PD controller we started from the proposed values; we found the best results with increasing Kp from 2500 to 3000 on each diagonal element and decreasing Kd from 40 to 30 on each diagonal element:

$$Kp^{C\_PD} = \begin{bmatrix} 3000 & 0 & 0 \\ 0 & 3000 & 0 \\ 0 & 0 & 3000 \end{bmatrix} \quad Kd^{C\_PD} = \begin{bmatrix} 30 & 0 & 0 \\ 0 & 30 & 0 \\ 0 & 0 & 30 \end{bmatrix}$$

This gives us smoother and faster locomotion: as shown in figure 2, the mean steady-state velocity goes up from approximately 0.9 to approximately 1.4m/s, which represents a gain of over 50%.

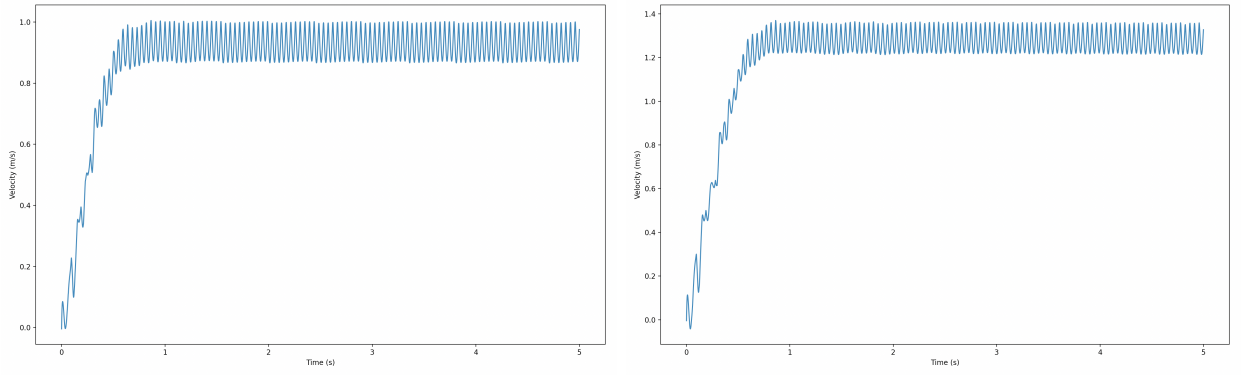


Figure 2: Left: base x- velocity using the old parameters; Right: base x- velocity using the new parameters

For the PD controller, we found that the robot was not always stable using the default gains. By decreasing all the values of both the Kp and the Kd matrices, we were able to improve the result. Our result was not perfect, and notably it tends to lead the robot to turn on the yaw direction, but the locomotion was visibly smoother and the stability was improved.
The parameters which we found to work best were:

$$Kp^{PD} = \begin{bmatrix} 50 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 10 \end{bmatrix} \quad Kd^{PD} = \begin{bmatrix} 1.5 & 0 & 0 \\ 0 & 0.3 & 0 \\ 0 & 0 & 0.3 \end{bmatrix}$$

2.3 Results

In this section we will detail the quantitative results of the gaits using this control structure.

2.3.1 Plots for the WALK Gait

Below and in the following three subsections are given, for each gait and for each leg, plots of the amplitude R the phase θ of the corresponding oscillator, plots of their derivatives $\dot{R}$ and $\dot{\theta}$, plots of the actual vs desired foot positions using the PD and CARTESIAN_PD controllers, and plots of the actual vs desired joint angles using the PD and CARTESIAN_PD controllers.
All plots are produced using the new tunings of the PD and CARTESIAN_PD controllers gains. The plots of R, θ, $\dot{R}$, $\dot{\theta}$ are produced using the CARTESIAN_PD controller.

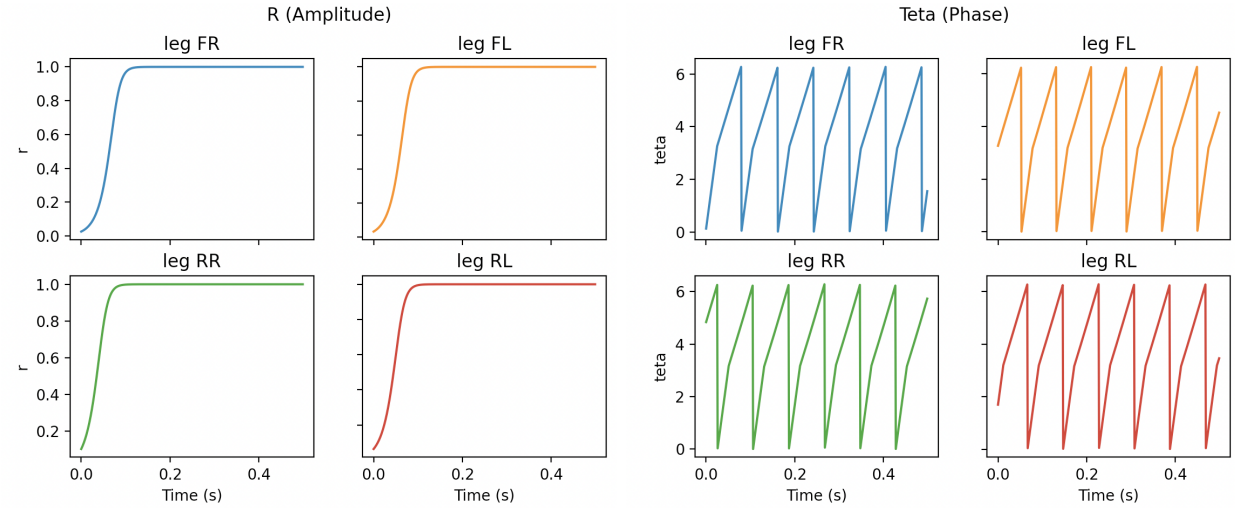


Figure 3: Resulting amplitude r and phase θ for the WALK gait

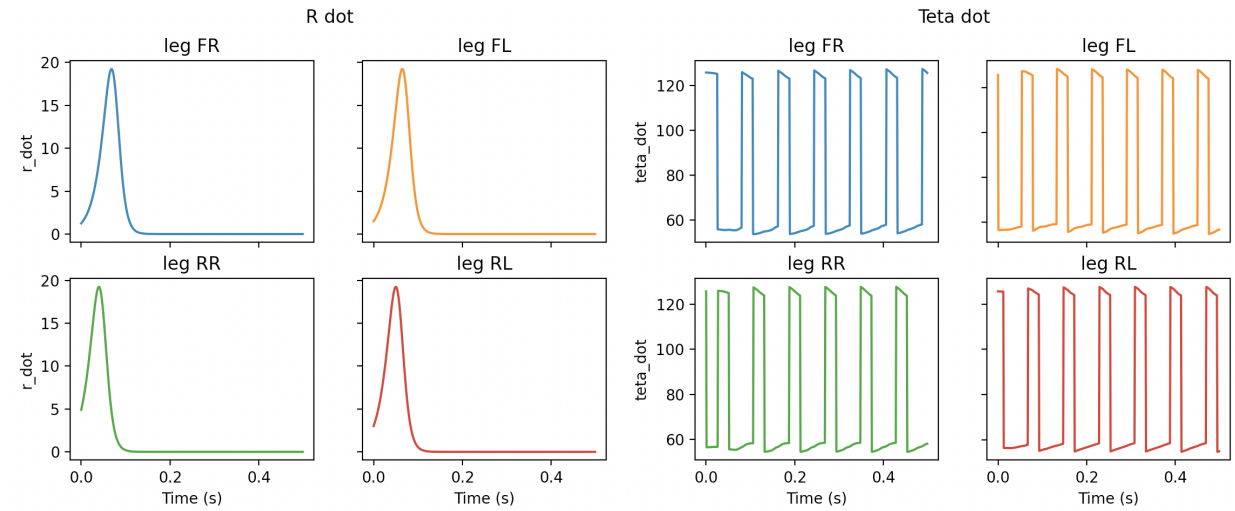


Figure 4: Resulting $\dot{r}$ and $\dot{\theta}$ for the WALK gait

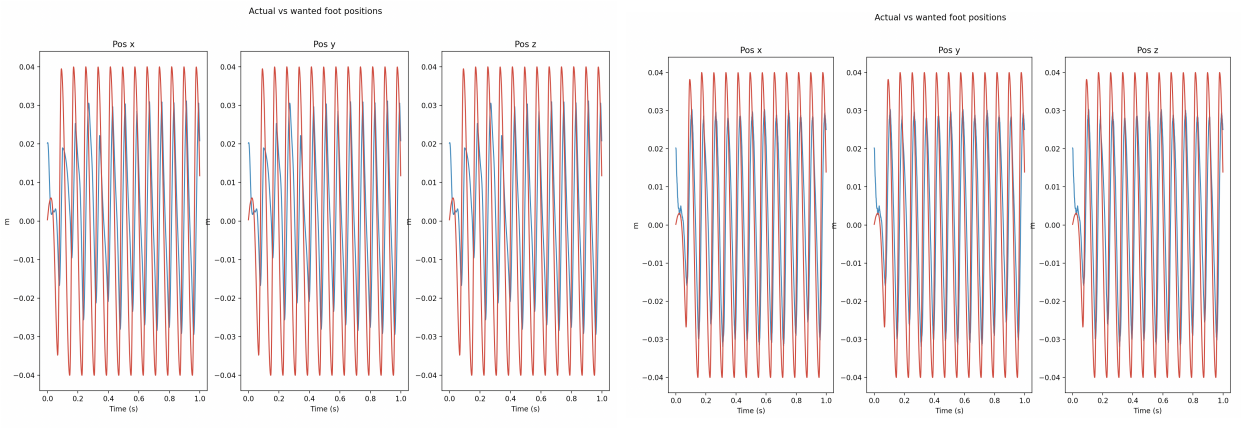


Figure 5: Actual (blue) vs desired (red) foot positions for the CARTESIAN_PD and PD controller (left and right, respectively)

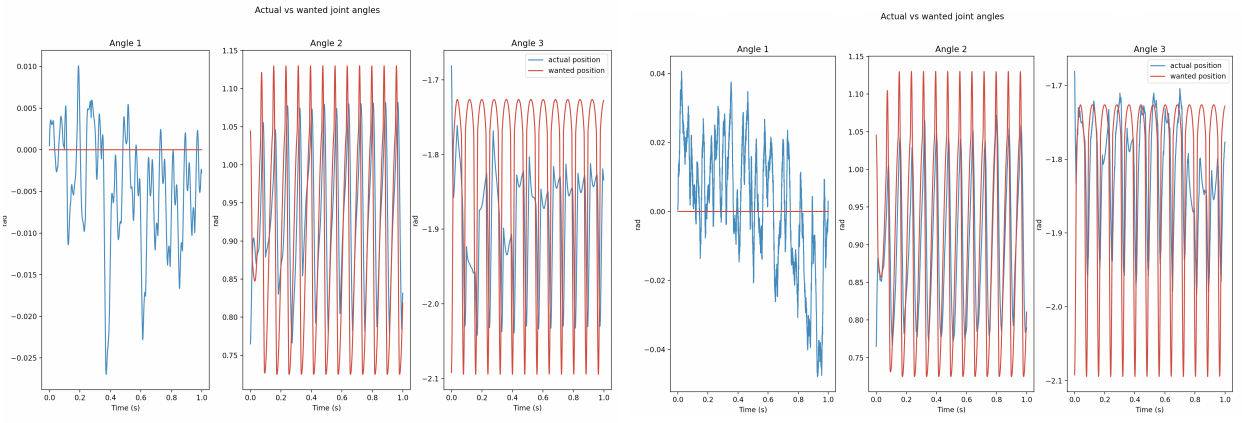


Figure 6: Actual (blue) vs desired (red) joint angles for the CARTESIAN_PD and PD controller (left and right, respectively)

2.3.2 Plots for the TROT Gait

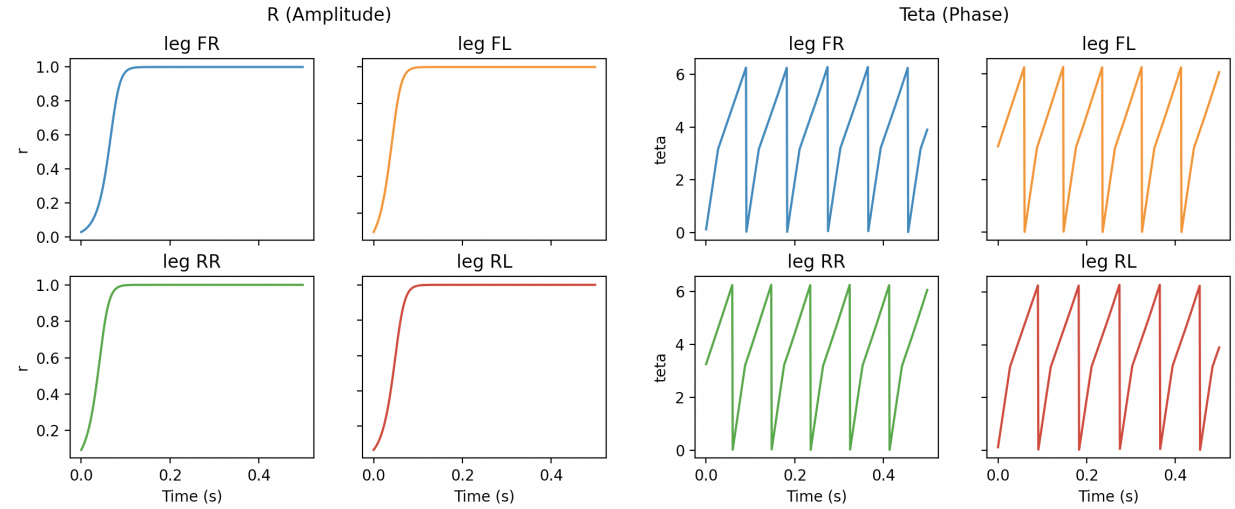


Figure 7: Resulting amplitude r and phase θ for the TROT gait

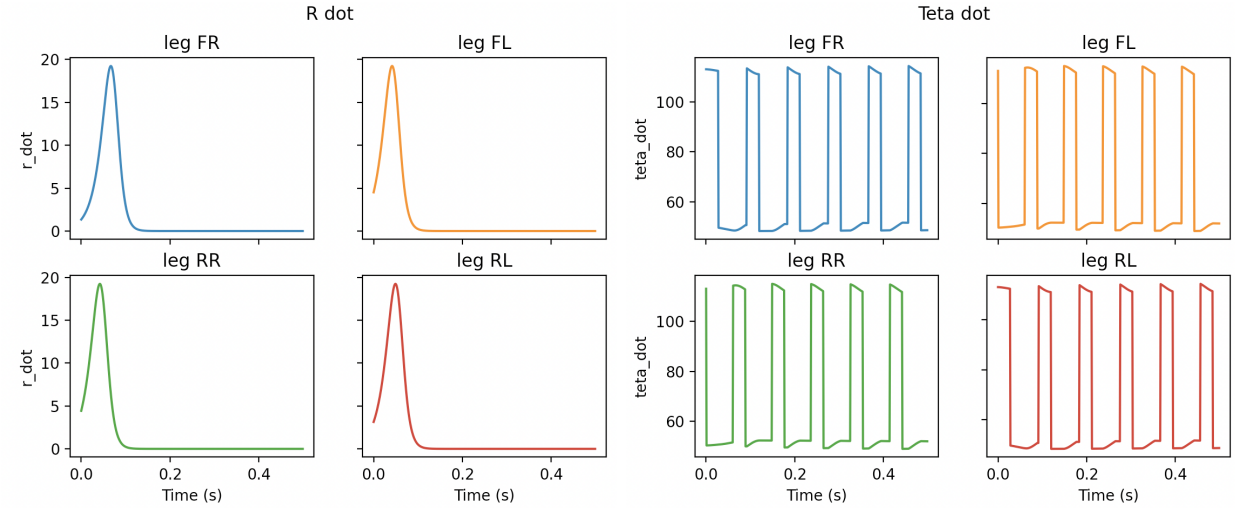


Figure 8: resulting $\dot{r}$ and $\dot{\theta}$ for the TROT gait

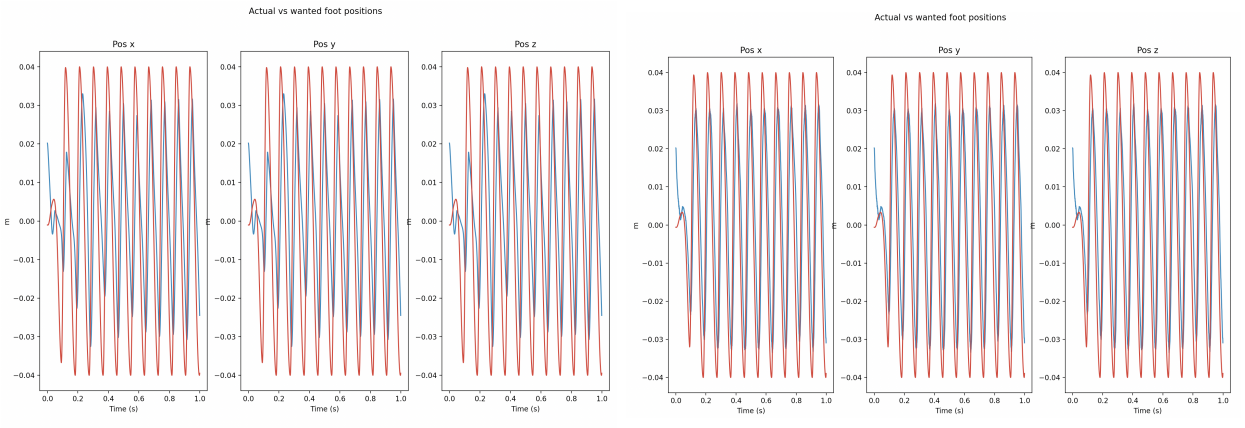


Figure 9: Actual (blue) vs desired (red) foot positions for the CARTESIAN_PD and PD controller (left and right, respectively)

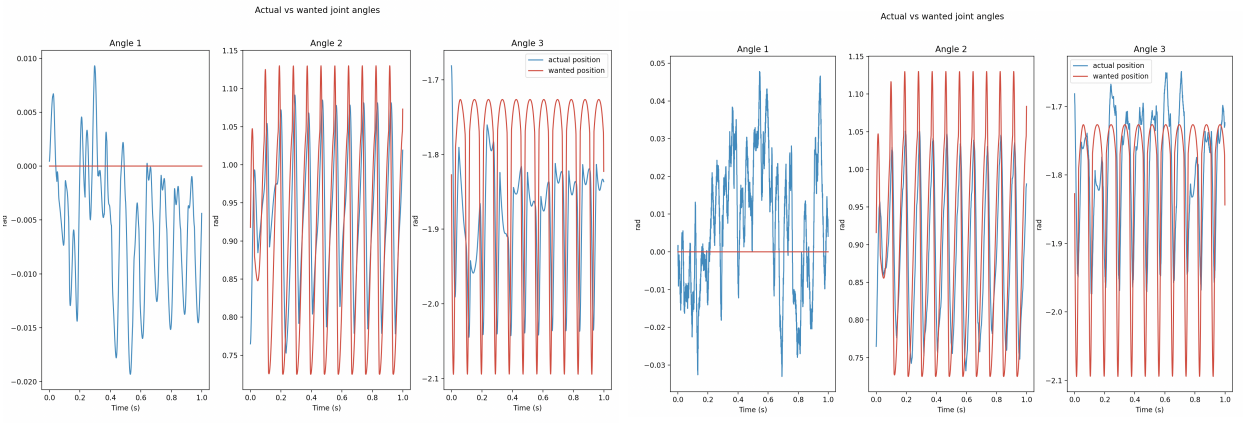


Figure 10: Actual (blue) vs desired (red) joint angles for the CARTESIAN_PD and PD controller (left and right, respectively)

2.3.3 Plots for the PACE Gait

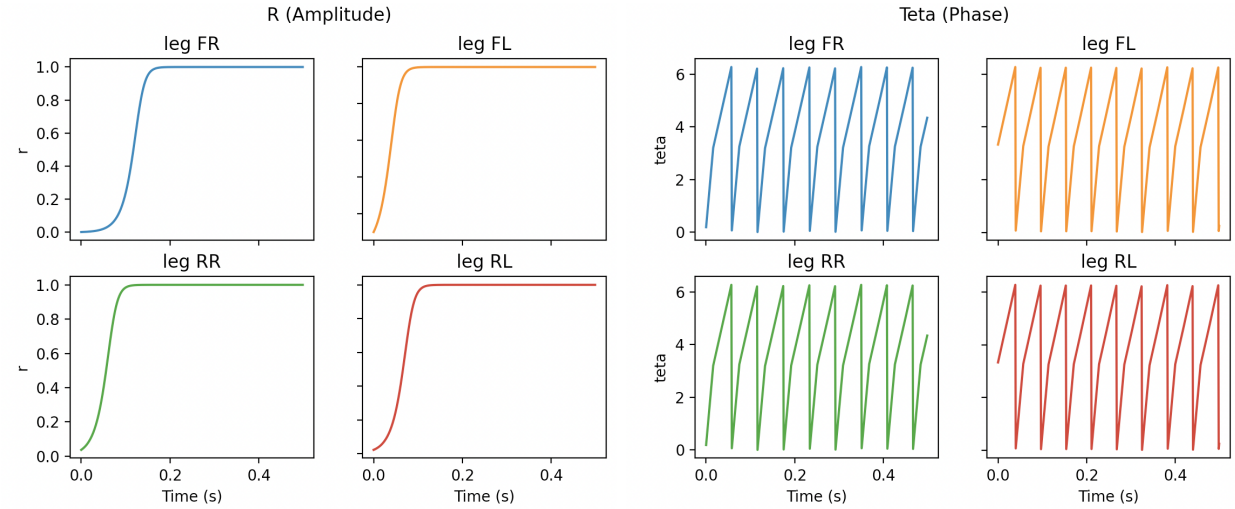


Figure 11: Resulting amplitude r and phase θ for the PACE gait

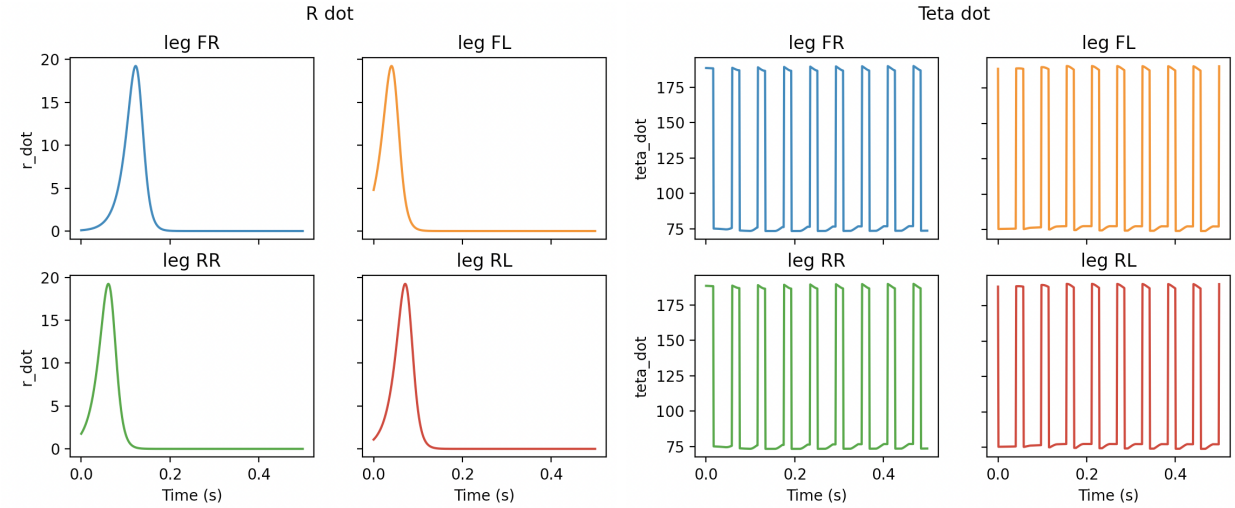


Figure 12: resulting $\dot{r}$ and $\dot{\theta}$ for the PACE gait

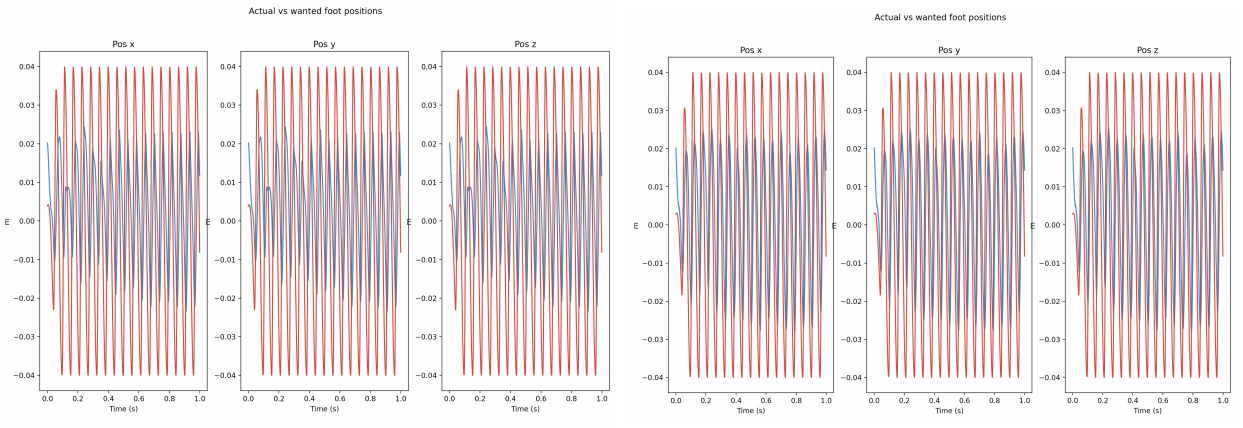


Figure 13: Actual (blue) vs desired (red) foot positions for the CARTESIAN_PD and PD controller (left and right, respectively)

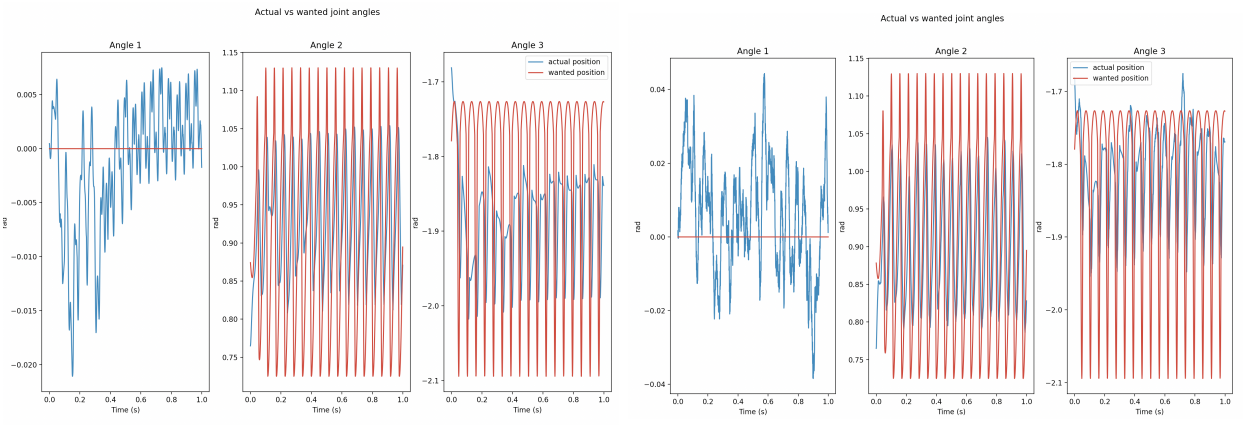


Figure 14: Actual (blue) vs desired (red) joint angles for the CARTESIAN_PD and PD controller (left and right, respectively)

2.3.4 Plots for the BOUND Gait

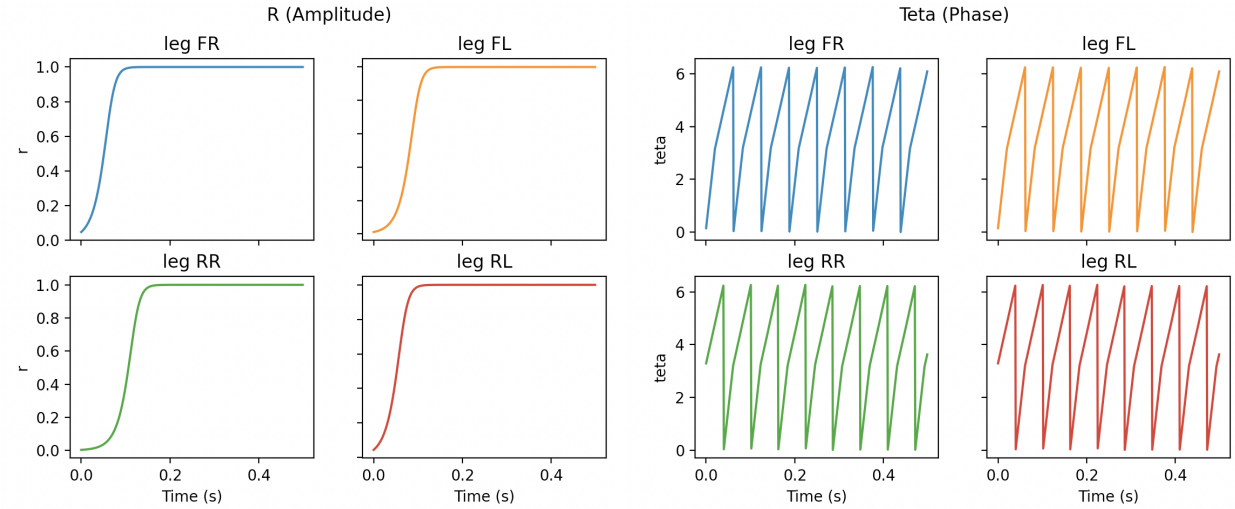


Figure 15: Resulting amplitude r and phase θ for the BOUND gait

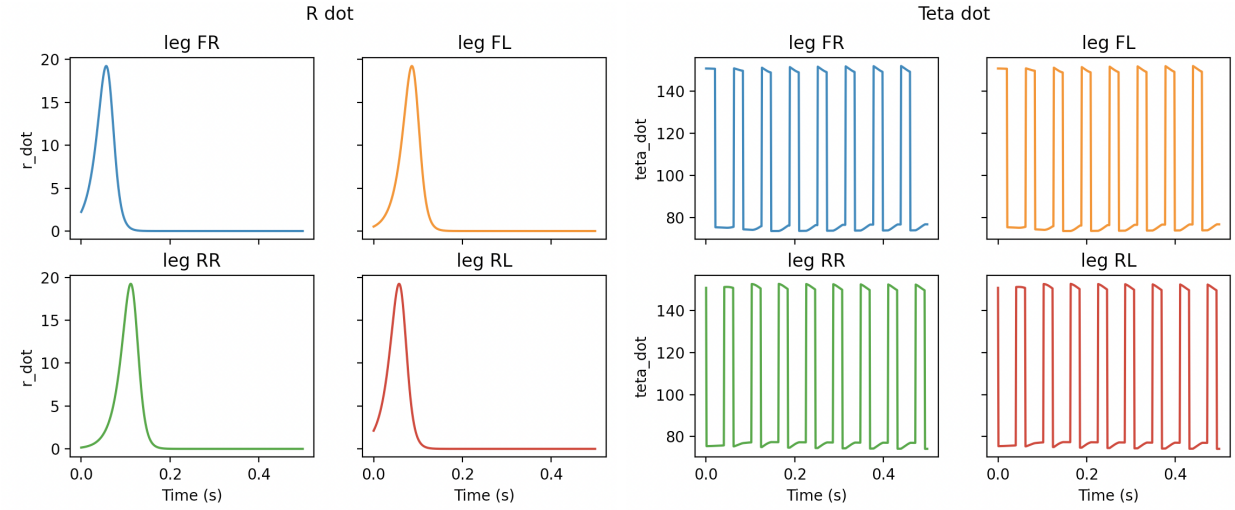


Figure 16: resulting Amplitude and phase for the "BOUND" gait

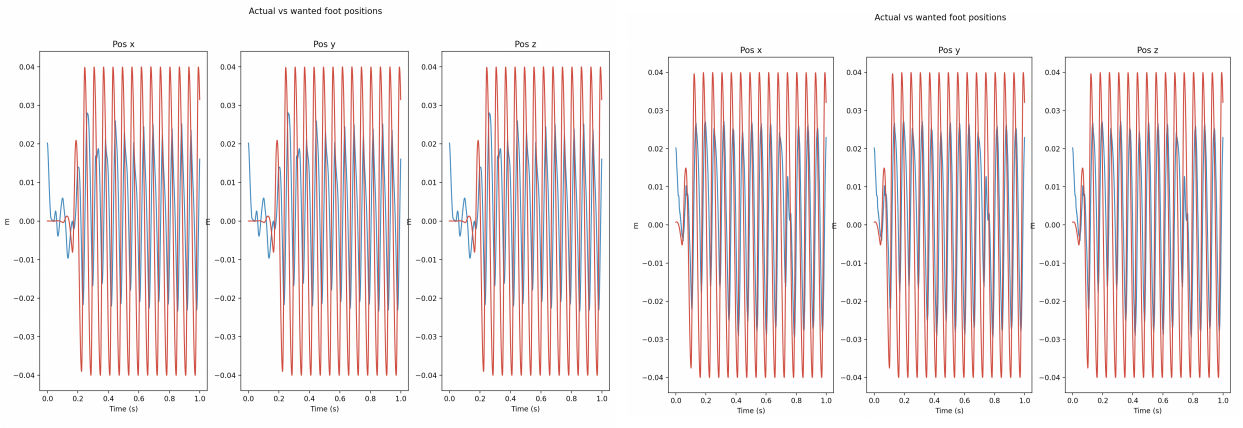


Figure 17: Actual (blue) vs desired (red) foot positions for the CARTESIAN_PD and PD controller (left and right, respectively)

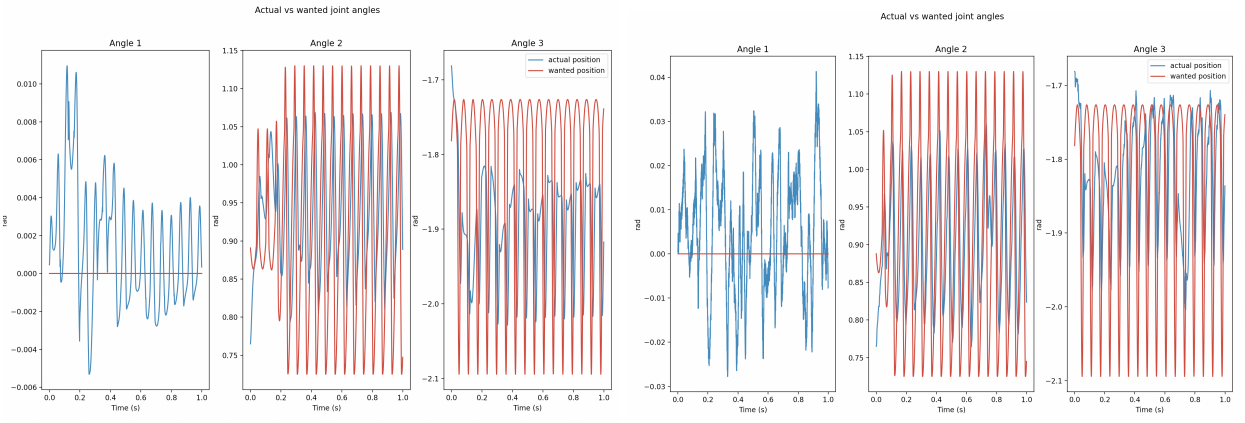


Figure 18: Actual (blue) vs desired (red) joint angles for the CARTESIAN_PD and PD controller (left and right, respectively)

2.3.5 Other Locomotion Metrics

In this subsection we detail some other locomotion metrics of the achieved gaits.

The velocity is given both for the default and for the new controller gains; the energy and corresponding CoT are given considering three possible interpretations of a negative work measure, as explained below.

	WALK	TROT	PACE	BOUND
$\frac{\omega_{stance}}{\omega_{swing}}$	0.45	0.44	0.40	0.50
Velocity(old k) [m/s]	0.93	0.92	0.97	0.97
Velocity(new k) [m/s]	1.4	1.3	1.6	1.55
"Real" CoT (E/mgd)	0.78	0.83	0.63	0.66
"Zero-energy" CoT	2.19	1.99	2.4	2.58
"Absolute-energy" CoT	3.54	3.2	4.19	4.49
Power consumption [W]	94.67	81.76	75.98	71.2
Zero Power consumption [W]	245.6	208.47	289.3	275
Absolute Power consumption [W]	396.5	355.18	502.66	479

In particular, the energy consumption used to calculate the CoT and the power consumption is calculated as follows:

$$E = \sum_t \delta E_t = \sum_t \tau_t \cdot d\theta_t = \sum_t \tau_t \cdot (\theta_t - \theta_{t-1})$$

Where the motor torques τ_t and angles θ_t at each timestep t are measured by the robot.
Note that from this formula, it is possible to get negative work measures when the torque is in the opposite direction to the angular velocity. In reality, on conventional motor structures this corresponds to the motor acting as a generator. However, this would probably be at such a low efficiency that it would be more correct to consider a zero energy consumption in this case.
For the sake of being complete, we consider three different cases: assuming that the motor works as a generator and neglecting losses, we can simply add up the energy values at each step without additional processing ("real-energy" case in the table above). Considering, as mentioned above, that the energy generated is negligible, we can choose to set to zero any negative energy values before summing the energy at each step ("zero-energy" case in the table above). Finally, assuming that we are using some type of structure which does not act as a generator, we must assume that even negative work results in power consumption. In this case, we sum the absolute value of the energy consumption at each step ("absolute-energy" case in the table above).

2.4 Discussion

Throughout our testing, we were able to see that the type of controller which is applied has a significant effect on the robot's behavior. As is to be expect, the PD controller is better at having the angles obey correspond to the desired angles, while the Cartesian PD is better at having the foot position correspond to the desired foot position.
In our tests, we had more success with the Cartesian PD. Visually, we could see in the PyBullet simulation that with the PD controller, the robot would tend to deviate more from the forward x-direction.
We also observed that the Cartesian PD resulted in a more uniform velocity, as demonstrated in figure 19.

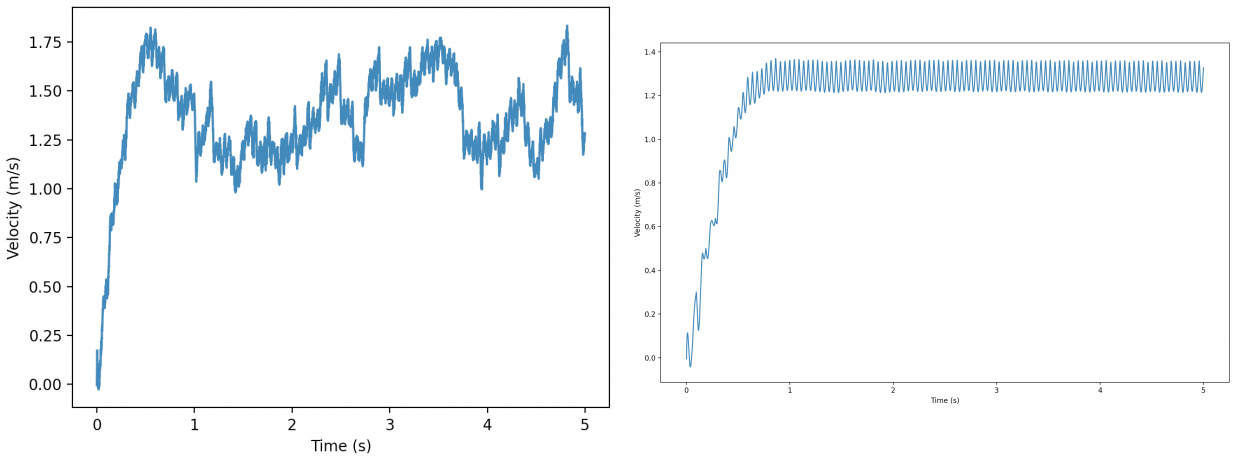


Figure 19: PD velocity curve against CARTESIAN_PD velocity curve

Ultimately, our implementation is good, but could be improved. We can observe in the desired vs actual graphs from the sections above that there is still some divergence between the desired and the actual positions and angles, meaning that the controller gains could still be tuned better.
Regarding the terrain, we could add a controller that can change the gait, the speed or the robots parameters (ground_clearance, ground_penetration, robot_height,etc...) for the robot to adapt to this new terrain.

More generally, we could think to broaden the scope of this architecture by implementing some high-level controllers to send signals to modulate the output of the CPGs to achieve high-level, non-periodic tasks such as tuning or changing speed; this is the principle of descending modulation discussed in [1] and other sources. For example, such a controller could impose a phase shift between the CPGs on either side to produce yaw rotation, or it could vary the amplitude of the CPG outputs to transition from a walking gait to a running gait.
To develop even further, the CPG architecture presented here could be complemented by adding a feedback structure, similarly to what was discussed in class [2]. This results in a system which is very robust due to the high level of redundancy.

3 Deep Reinforcement Learning

3.1 Observation Space

The observation space which we chose to use consists of of the following measurements:

- Motor angles
- Motor velocities
- Motor torques
- Base orientation
- True base angular velocities
- Base linear velocities

All measurements on the motors are useful to evaluate and optimize the energy consumption of the robot. Other measurements such as the base orientation, the base angular velocities and the base linear velocities are useful to evaluate the robot performance and its behavior in the environment. This allows us to know if the robot is moving straight forward, in what direction, at what speed and whether it is stable. This is especially in roll and pitch.

On real hardware, the noise level of these measurements will depend very strongly on the quality of the sensors used. Generally speaking, the measurements made in the motors are less noisy than the IMU measurements.

3.2 Action Space

Two different action spaces were used: the joint PD control space and the Cartesian PD control space. The former was already implemented in the program, the latter had to be completed by us in the `ScaleActionToCartesianPos` function of `quadruped_gym_env.py` using the following equations:

$$\tau_{Cartesian} = J^T(q) * (K_{p,Cartesian}(p_d - p) + K_{d,Cartesian}(v_d - v))$$

We performed tests training policies with the two different action spaces but keeping all the other parameters constant: default environment, PPO algorithm with the default hyperparameters, and with the following reward function

$$R_t = 2 * (x_t - x_{t-1}) - (0.1 * \dot{\psi}_t)^2 - 0.01 * (0.1 * \dot{theta}_t)^2 - 0.01 * (0.1 * \dot{phi}_t)^2$$

Where x_t is the x-direction position at the timestep and $\dot{\theta}$, $\dot{phi}$, $\dot{\psi}$ are the pitch, roll and yaw rates, respectively.
Note that the weights used are not exactly those from our final reward function; however, the rewards and penalties used are the same, so we believe that this comparison is nonetheless relevant to the final policy which we trained. More details regarding the development of the reward function are given in section 3.3.

A first observation which we made was that the Cartesian PD action space resulted in slower training. This is due to the Jacobian calculation, which requires more computing operations.

Figures 20, 21, 25 and 22 below show the comparison between the two environments:

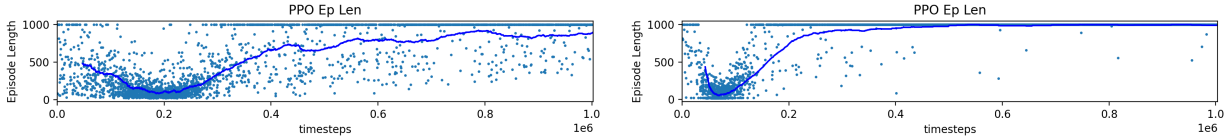


Figure 20: Mean Episode Length: PD (left) and Cartesian PD (right)

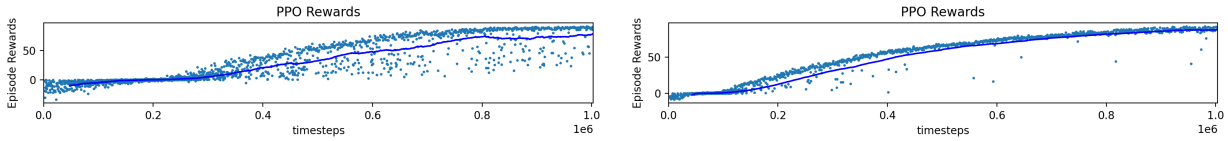


Figure 21: Mean Episode Reward: PD (left) and Cartesian PD (right)

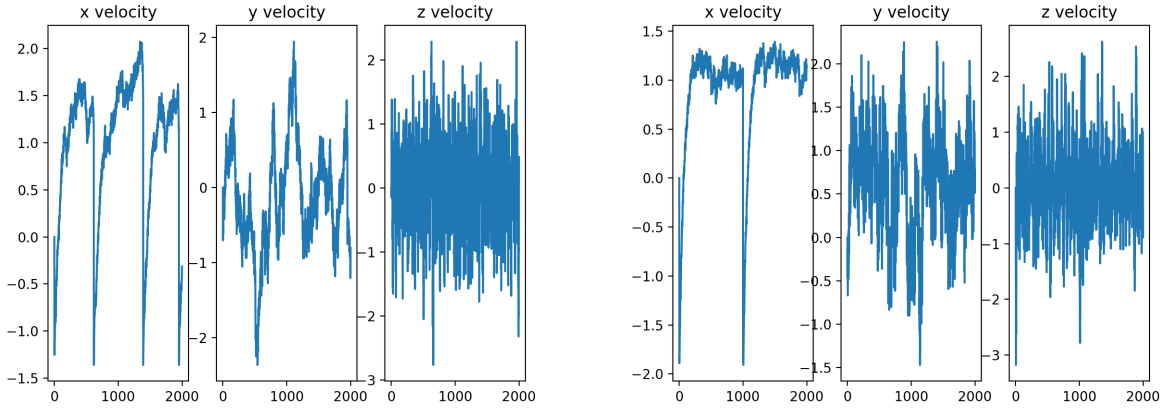


Figure 22: Linear velocities: PD (left) and Cartesian PD (right)

We can see from the graphs above that for our reward function, there is a significant difference in performance between the 2 actions space. Figures 20 and 21 show that the Cartesian PD results in a policy which is more stable, and which converges faster; figure 22 shows that the Cartesian PD policy resulted in a gait which tended to be faster; figure 25 shows that the Cartesian PD policy tends to have less roll and pitch. In the PyBullet simulation, we were able to verify visually that the Cartesian PD policy tended to be smoother. The videos of these 2 runs can be found in the videos folder provided with the report, under the names PD_Action_Space and Cartesian_PD_Action_Space.

It seems logical to us that the Cartesian PD action space had more success than the PD action space: intuitively, it seems easier to distinguish a good gait from a bad gait, especially in terms of minimizing yaw and maximizing displacement, by looking at foot placements than by looking at joint angles. In terms of the neural network, there might be more separation between good and bad gaits in the former environment than in the latter.

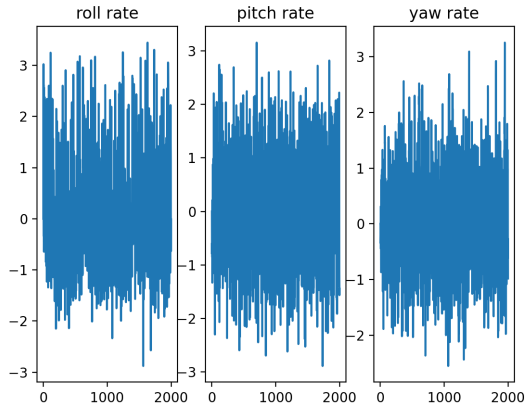


Figure 23: PD

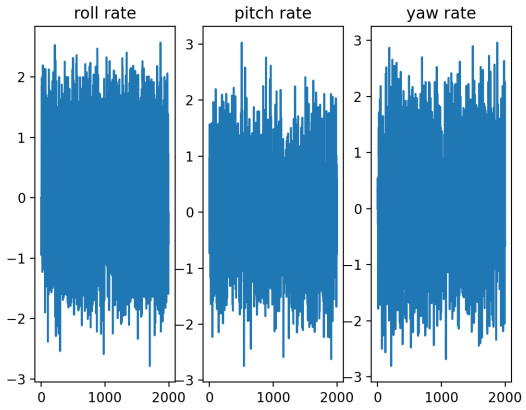


Figure 24: Cartesian PD

Figure 25: Angular Velocities

3.3 Reward Function

The ultimate goal would be to create a smooth, omnidirectional locomotion policy that is energy efficient and capable of moving within a continuous range of forward, backward, and lateral velocities and yaw rates. For this initial testing, we focused on reaching a robust policy where the robot moves forward at a trot or running pace. To achieve this, we considered several different parameters in our reward function:

- **Distance**
  We tried setting a reward on the distance covered in each timestep, as done in the example provided. This yields a reward of the form:
  $$R_t^d = c^d \cdot (x_t - x_{t-1})$$
  Note that it is important to reward the distance traveled at each step, rather than the total distance traveled, otherwise the reward for forward movement increases as the robot moves further from the origin.
  We realized later that because the timestep duration is fixed, this effectively corresponds to maximizing the instantaneous velocity at each step.
  We found this parameter very effective at producing smooth and stable gaits.
- **Velocity**
  As mentioned in the previous point, maximizing the distance covered on each timestep is equivalent to maximizing the velocity of the robot. This yielded similar results.
- **Target velocity**
  We thought to try assigning a penalty to deviation from a target velocity, to make the robot move at a desired velocity. This was implemented using a penalty of the form:
  $$P_t^{vd} = -c^{vd} \cdot |v_x - v_{x,desired}|$$
  We found this to provide less good results than just maximizing the velocity. This might be because for certain parameters, the robot has certain velocities which are most efficient to move at. With

more tuning and potentially some additional rewards and penalties this could be improved, but we found that it was not the most efficient parameter at producing smooth and stable gaits.

- **Angle of the base**
  To make the robot move in a straight line along a desired axis, we thought to set a penalty on non-zero yaw values.
  Note that this does not necessarily guarantee that the robot moves in a straight line, it only ensures that the robot is facing in a certain direction while it moves. Depending on the other rewards used, this might be more or less useful; for example, maximizing forward displacement and minimizing energy consumption will theoretically result in the robot learning to walk in a forwards direction without constraints on the angle. However, the angle constraint might help the model to converge faster.
  Additionally, we thought to penalize excessive pitch and roll. Here, we observed empirically that some amount of pitch and roll is necessary to provide smooth motion, so we chose to use small weights on these penalties.
  To further avoid excessive penalization of natural rotations necessary to the movement, we thought to implement non-linear penalties which would penalize large deviations from the desired angle much more that smaller deviations. This results in a polynomial penalty of the form:

$$P_t^{angle} = c_1^{angle} \cdot |\psi|^{n_1} + c_2^{angle} \cdot (|\theta|^{n_2} + |\phi|^{n_2}); n_{1,2} > 1$$

  Where ψ is the yaw angle and θ, ϕ are, respectively, the pitch and roll angles.
  In our testing, we found that this was a useful penalty to add and worked well to ensure a rectilinear trajectory.
- **Angular velocity of the base**
  For the same reason we thought to penalize the pitch, roll and yaw, we could also penalize the rates of these rotations.
  In our testing, we found this to provide similar results to penalizing the angles. The main difference is that penalizing yaw instead of yaw rate results in the robot taking time to correct its orientation whenever it deviates, rather than continuing in a different direction.
  To maximize the rectilinear motion over the limited timeframe, we thought that the rates would be more effective than the actual angles; this way, the robot will not waste time turning around when it deviates. Other than this, the choice would depend on whether the desired task is to move roughly in a straight line, or to follow a true desired direction.
  Similarly to the rotations, we thought to use non-linear penalization of the rotation rates. We found that it tended to be harder to correctly weigh the penalty on the angular rates, as it varied more and in a less intuitive way than the angles.
- **Energy consumed**
  Finally, the last goal of our desired policy is to minimize energy consumption. By extension, this can in some cases produce more natural-looking gaits, as often locomotion in living beings has a component of energy-efficiency.
  The energy consumption was calculated following the formula detailed in section 2.3. For this case, we chose to set negative energy to zero, so assuming that the motors can work as generators but at very low efficiency.
  We found that this parameter was not optimal in training a policy for smooth, fast locomotion. Depending on the relative weight of the energy and velocity terms, this tended to produce some gaits where the robot was very conservative in moving its limbs. This could potentially be added

to an existing, functioning policy to optimize the energy consumption, but we found it to be impractical.

A major issue we had when testing reward functions was that we found in some initial tests that the network tended to converge to a solution where the robot falls down immediately, as shown in figure 26.

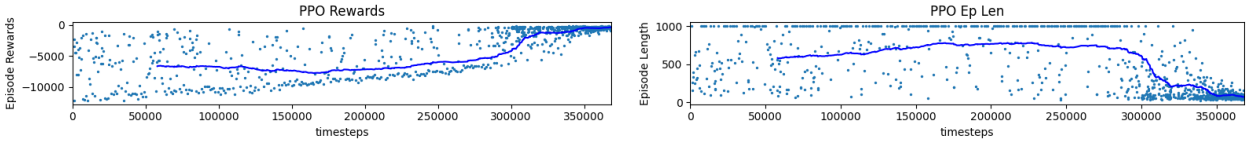


Figure 26: Rewards curve (left) and episode length (right) for a negative reward

This was due to implementing the penalties simply as a negative reward $R_t = -P_t$, which resulted in a negative final reward. This then penalized longer runs, which accumulated a more negative total reward.
To solve this problem, we implemented penalties as a reward of the form:

$$R_t = c \cdot (1 - \frac{P_t}{P_{max}});$$

To avoid confusion with the previous notations for penalties, note that following this notation P_t is unweighted and then c is the weight corresponding to this reward.
This way the reward is always positive, and the reward is maximized when the penalty is minimal.

Following a large number of tests, we settled on the following reward function:

$$R = 2 \cdot (x_t - x_{t-1}) - 0.01 \cdot \dot{\psi}^2 - 0.0001 \cdot \dot{\theta}^2 - 0.001 \cdot \dot{\phi}^2$$

Note that because the reward term is significantly greater than the penalty terms, it is not necessary to use the format explained above to write the penalties as rewards to ensure that the overall reward is positive.

3.4 Neural Network Hyperparameters

The first RL algorithm which was proposed is Proximal Policy Organization or PPO. Developed in 2017, this is a popular RL algorithm [3][5].
PPO is in the class of policy gradient methods, meaning that a gradient descent method is used to find policies which optimize the expected return.
In the clipped PPo implementation, the main objective function is given by:

$$\mathbb{E}\left[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

A_t is the difference between the discounted rewards and the baseline estimate; in other words, it describes whether this policy improved on the estimated rewards. $r_t(\theta)$ is defined as $r_t(\theta) = \frac{\pi(a_t|s_t)}{\pi_{old}(a_t|s_t)}$ and describes whether a given action $a_t|s_t$ became more or less likely under the new policy.

This algorithm favors policies which maximize $r_t(\theta)\hat{A}_t$. Adittionally, there is a clipping of the objective function to the range $[(1-\epsilon) * \hat{A}_t, (1+\epsilon) * \hat{A}_t]$; this has the effect of limiting the output in the case of policies which are too different (case $r_t(\theta)$ is very large), to prevent from moving too far from policies which have been found in previous iterations to work well.

The hyperparameters of the PPO algorithm which are referenced in the code are the following [4]:

- *gamma*: the discount factor: a reward R_k at k+1 steps into the future is valued $\gamma^k \cdot R_k$; describes how much the network cares about rewards in the distant future compared to immediate rewards
- *n_steps*: the number of steps to run in each environment per update
- *ent_coeff*: the entropy coefficient for the loss calculation; describes how "predictable" the actions of the agent are
- *learning_rate*: the learning rate of the network; describes the change in the model following the estimated error at each iteration
- *vf_coef*: the value function coefficient for the loss calculation (particular to this specific implementation)
- *max_grad_norm*: the maximum value for the gradient clipping
- *gae_lambda*
- *batch_size*: the minibatch size
- *n_epochs*: the number of epochs when optimizing surrogate loss
- *clip_range*: the clipping parameter; corresponds to the ϵ term described above
- *clip_range_vf*: the clipping parameter for the value function
- *verbose*: the terminal output as the program runs: 0 for no output, 1 for info, 2 for debugging
- *tensorboard_log*: the log location for tensorboard
- *_init_setup_model*: whether or not to build the network at the creation of the instance
- *policy_kwargs*: additional arguments to be passed to the policy upon creation. For us this is the architecture of the network (ie the layout of the neurons).
- *device*: run the network on CPU or GPU; the program runs faster on GPU

Typically, for us the *gamma* coefficient should be high. Indeed, we want to favor more periodic movements, which results in the rewards at each timestep varying. Using a small *gamma*, these movements would be penalized in favor of more constant movements where each timestep maximizes the instantaneous reward. Using a high *gamma*, the network will instead favor policies which maximize the reward over a whole period of motion.
Potentially, it could be interesting to vary *clip_range*, as this is one of the particularities of the algorithm. It should be tuned for our specific implementation to maximize the convergence speed.

The second RL algorithm proposed to us was Soft Actor Critic or SAC; this algorithm is especially popular for robotics applications [6] [8].
In this algorithm, the objective function is given by:

$$\mathbb{E}\left[\sum_t r(s_t, a_t) - \alpha log(\pi(a_t|s_t))\right]$$

Where the first summand describes the expected rewards, and the second summand describes the expected entropy. Thus, this algorithm is different to the previous one in that it tries to maximize the randomness of the model. There are several reasons for this; intuitively, more random models

"explore" more, so they can converge sooner. Also, higher randomness reduces the dependence on the hyperparameters.
Note especially the α term; this is called the temperature term, and it is used to balance the reward term with the entropy term. Setting α to 0, this algorithm becomes a more conventional algorithm to maximize only the reward.
In empirical tests, it has been found that SAC converges faster than other RL algorithms for robotcs applications.

The hyperparameters of the SAC algorithm which are referenced in the code are the following [7]; parameters which are different from PPO are not explained again:

- *learning_rate*
- *buffer_size*: the size of the replay buffer
- *batch_size*
- *ent_coef*: the α term described above
- *gamma*
- *tau* is the soft update coefficient
- *train_freq* is the frequency of the model update
- *gradient_steps* is how many gradient steps do do after each rollout
- *learning_starts*: how many steps of the model to collect transitions for before learning starts
- *verbose*
- *tensorboard_log*
- *policy_kwargs*
- *seed*: seed for the pseudo random generators; set a fixed seed so that the algorithm will always converge to the same result
- *device*

In this algorithm, the most interesting parameter to vary would be *ent_coef*; this is application-dependent, and better convergence could be reached when tuning this parameter correctly.

We found that the PPO algorithm with the pre-set parameters was sufficient for our purposes, so we did not change any parameters.
We did some initial testing with the SAC algorithm, and found that it converged less well than the PPO. However, given that SAC is designed specifically for robotics, it could be interesting to see which results can be found with this algorithm.

3.5 Environment

In order to challenge the robot and to determine the robustness of the policy, several changes of the environment during testing were made on different parameters such as the friction coefficient of the ground, or by adding boxes of random sizes and shapes.

First, the friction coefficient of the ground was modified using the `add_noise` parameter from the program `load_sb3.py`. This has the effect of randomly modifying the friction coefficient and thus allowing the robot's performance to be evaluated on different terrains. In spite of this modification to the environment, the robot still managed to accomplish its goal and remained stable, which shows good robustness of the policy. However, this had an impact on the speed: figure 27 shows 2 different runs of the robot at friction coefficients of 0.626 and 0.913, respectively. Note that the first coefficient

is active at timesteps 0 to 1000, the second at timesteps 1000 to 2000. We can see that the robot actually goes slightly faster under the lower friction coefficient.

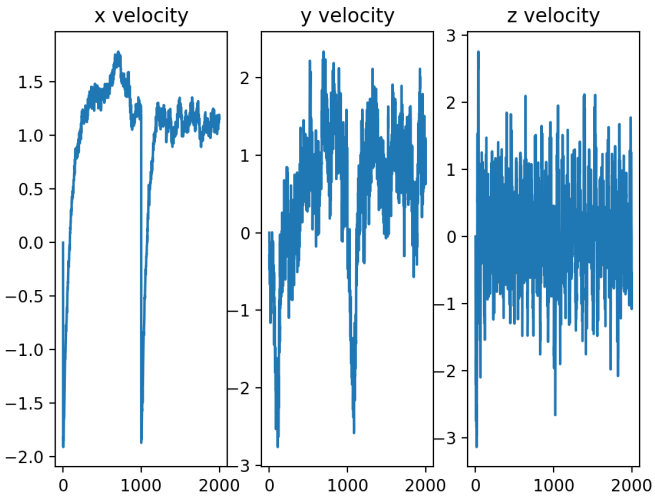


Figure 27: Impact of the friction coefficient of the ground on the velocities

We then tested the robot in a more complex environment consisting of a terrain containing boxes of random shapes and sizes, and making the robot carry a box on its base to vary its mass and inertia, as shown in figure 28.

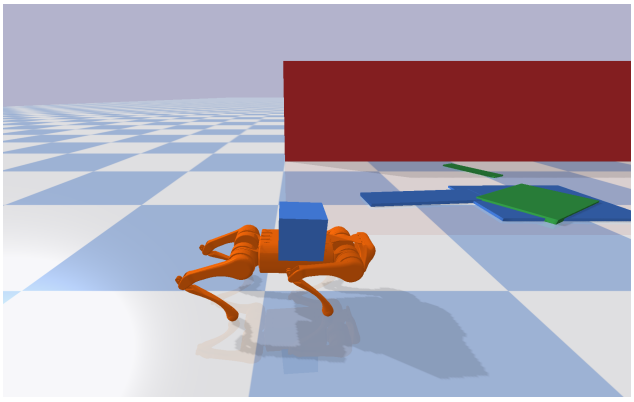


Figure 28: Test environment

Due to the random and uneven nature of the environment, the results are fairly random and vary greatly between different runs. In most runs, we found that the robot was not able to cross the ground with the obstacles: it tended to fall either due to the box it was carrying or because of the terrain.
On some occasions, the robot was able to complete the run without falling. A video of one of these runs is given in the video folder under the name Final_run_with_env.
The two main problems of the robot in this terrain were that the robot was very often unbalanced

when the load put on it was too heavy, and that it tended to bump its legs into the obstacles. This shows that our policy is not as robust as desired.

A video named Test_Env shows different tests in this challenging environment.

3.6 Results

As explained in the previous section, the policy obtained from Deep Reinforcement Learning is relatively robust. The robot behaves well and obtains good results in terms of speed and stability when the environment is simple or only slightly modified (coefficient of friction of the ground); on the other hand the results are less satisfactory on more complex terrain.

The chosen parameters allows the reinforcement learning algorithm to converge well: indeed, the control policy managed to converge in under 1 million timesteps. The episode length curve converged in only 400 000 timesteps (as shown in figure 29) while the reward curve took about 900 000 timesteps (as shown in figure 30).

As shown in Figure 31, using our trained policy the robot has an average speed of between 1 and 1.5 m/s depending on the runs. Note that this seems realistic, as anything above 5m/s would be impossible and around 2-3m/s is really fast for this robot. Its angular rotation rates are between -2 and 2 rad/s (see Figure 32), which indicates a good general stability of the robot.

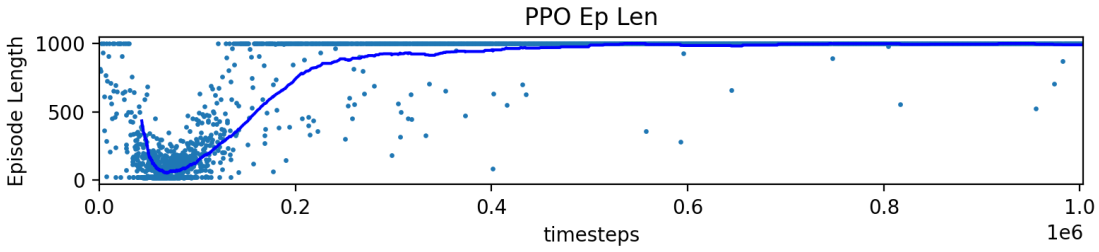


Figure 29: Episode Length

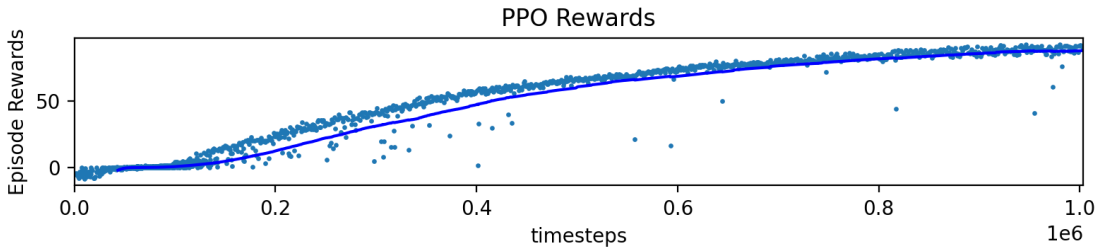


Figure 30: Episode Reward

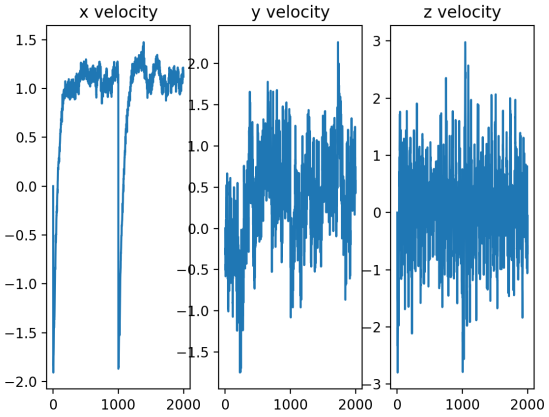


Figure 31: Linear velocities

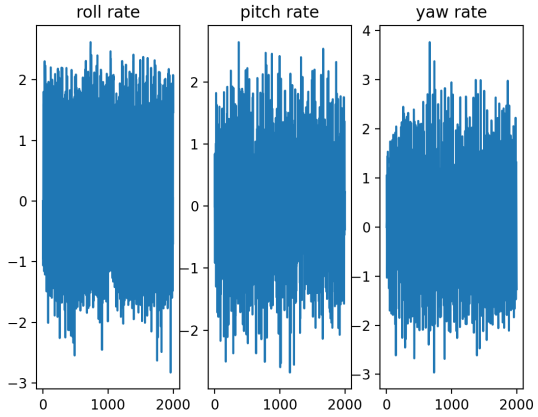


Figure 32: Angular velocities

The video corresponding to the runs of the figures above is named *Final_run* and is available in the folder containing all the videos.

Using the "zero-energy" CoT calculation developed in section 2.3, we calculated a cost of transport of 0.88 for this trained gait.

3.7 Discussion

We remarked that this particular implementation of RL had some shortcomings when applied to this task. Often, the outputs of the network to changes in the RL parameters were unintuitive, making the tuning difficult. In particular, using a single reward function for the entire training is probably not optimal for fast convergence or for tuning; we might instead use a "curriculum" of increasingly difficult skills which lead up to the final task.

Note that the method of using RL to train a control policy has some problems when transferring to real life. As explained to us notably during the presentation by prof. Marco Hutter, which summarizes some of the main points of [10], there will always be problem of sim-to-real transfer: because the simulation can never correspond exactly to real life, policies trained only in simulation often work poorly in real life. The solution to this is to combine some runs with the real-life robot in the RL training. Obviously, for us this is not possible.

4 Conclusion

Overall, this is very complete an interesting look at two different methods used in current research to generate forward locomotion in quadruped robots. It is interesting to remark that the two methods are conceptually extremely different, yet we found them to reach results which were quite similar: indeed, we were able to see visually that the gaits produced in the two policies are somewhat similar. This project has been a valuable hands-on experience with what would otherwise be very theoretical

concepts.

5 References

[1] L.Righetti and A.J. Ijspeert, “Pattern generators with sensory feedback for the control of quadruped locomotion,” in *2008 IEEE International Conference on Robotics and Automation*, 2008, pp. 819–824.

[2] F. Dzeladini, J. van den Kieboom and A.J Ijspeert, "The contribution of a central pattern generator in a reflex-based neuromuscular model", in *Frontiers in Human Neuroscience*, 2014

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” in *arXiv preprint arXiv:1707.06347*, 2017.

[4] Explanation of the PPO algorithm implementation from the Stable Baselines website
`https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html`
Accessed 7/1/2022

[5] "An Introduction to Policy Gradient Methods", YouTube video by user Arxiv Insights
`https://www.youtube.com/watch?v=5P7I-xPq8u8`
Accessed 7/1/2022

[6] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” in *International conference on machine learning*. PMLR, 2018, pp. 1861–1870.

[7] Explanation of the SAC algorithm implementation from the Stable Baselines website
`https://stable-baselines3.readthedocs.io/en/master/modules/sac.html`
Accessed 7/1/2022

[8] T. Haarnoja, V. Pong, K. Hartikainen, A. Zhou, M. Dalal, and S. Levine, "Soft Actor Critic - Deep Reinforcement Learning with Real-World Robots", from the Berkely BAIR website, 2018
`https://bair.berkeley.edu/blog/2018/12/14/sac/`
Accessed 7/1/2022

[9] Description of the A1 robot from the Unitree website
`https://www.unitree.com/products/a1/`
Accessed 7/1/2022

[10] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, "Learning agile and dynamic motor skills for legged robots", in *Science Robotics*, 2019