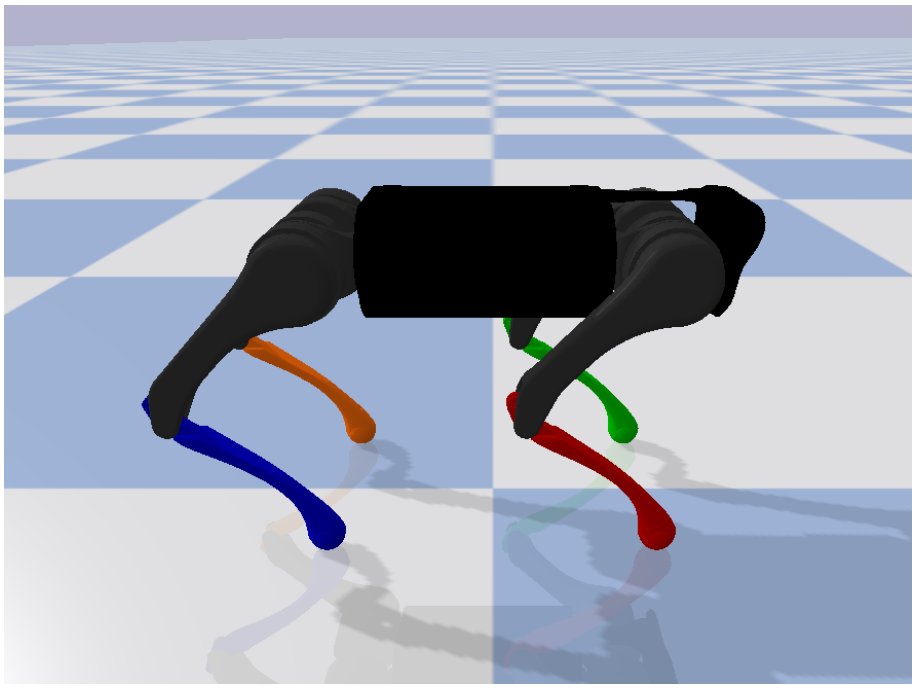
École Polytechnique Fédérale de Lausanne

Project 2 - Quadruped Locomotion with Central Pattern Generators and Deep Reinforcement Learning

MICRO-507 Legged Robotics

Cameron Bush
Marie Ethvignot
Marine Moutarlier



Contents

1 Problem Statement and Setup

In this study, we aim to investigate two different approaches to quadruped locomotion: Central Pattern Generators (CPG's) with predefined gaits, and reinforcement learning using a Markov Decision Process (MDP). The general goal in using these different methods is to generate a quadruped which is capable of stable locomotion. We will additionally investigate the effects of the two aforementioned methods on the velocity of the quadruped and the cost of transport. We used a Pybullet simulation of a standard quadruped with three actuated joints per limb, making a total of 12 joints. The reference frame is defined with respect to the robot, with the x-direction being the intended forward direction, y being the direction perpendicular to robot motion (positive toward robots left), and z being the vertical.

1.1 General Modeling and Control

Each leg has 3 joints which are actuated with their own motors. The actuation is modeled as a simple torque delivered to the different leg links. The hip joint controls the yaw of the leg, or its rotation around an axis parallel to the forward motion of the robot. The thigh joint controls the angle of the upper leg link, and the calf joint controls the angle of the lower leg link. We use the Jacobian to map the foot position/velocity to a specific joint configuration/velocity using forward kinematics (equations 2 and 3), and the inverse of the Jacobian to find the joint states for a specific desired foot position. We also use the Jacobin to map specific force requirements at the feet to torques in the joints using equation 3. In addition to using the CPG and reinforcement learning to provide high level control, we use PD control with respect to both the joint positions and the cartesian position of the foot (shown in equations 5 and 6).

$$p = f(q) \tag{1}$$

$$p = J(q) \cdot q \tag{2}$$

$$v = J(q) \cdot \dot{q} \tag{3}$$

$$\tau = J^T(q)F \tag{4}$$

$$\tau_{joint} = K_{p,join}(q_d - q) + K_{d,joint}(\dot{q_d} - \dot{q}) \tag{5}$$

$$\tau_{cartesian} = J^T(q)[\, K_{p,cartesian}(p_d - p) + K_{p,cartesian}(v_d - v)\,] \tag{6}$$

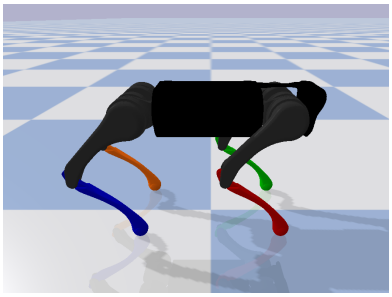


Figure 1: Robot with his legs

The legs are numbered 0-3 in order Front Right (FR, 0), the one in red, Front Left (FL, 1), the one in green, Rear Right (RR, 2), the one in blue, Rear Left (RL, 3), the one in orange.

1.2 Performance Evaluation

Performance was measured by the average forward velocity, and the cost of transport of the robot. The cost of transport (COT) was calculated by integrating the motor torques over the corresponding change in joint angle for a time step in the simulation. This yielded the power required to move the robot per time step, which we then used in the following equation to calculate the COT.

$$COT = \frac{P}{mgv} \tag{7}$$

P is the total power consumed by the motors, m is the mass of the robot, g is earth's gravitational force, and v is the instantaneous velocity of the robot. The COT was calculated for each time step, and then averaged over the duration for the simulation to yield one final value. Our implementation of this concept in Python is shown below

```
def calculate_COT(torque_matrix, joint_matrix, forward_velocity, mass, gravity, time_step):
    num_motors, num_time_steps = torque_matrix.shape
    power = np.zeros([num_motors, num_time_steps])
    cot = np.zeros([num_motors, num_time_steps])
    average_cot = 0
    for i in range(num_motors):
        torque_i = torque_matrix[i, :]
        joint_i = joint_matrix[i, :]
        joint_i = np.concatenate([joint_i, [joint_i[-1]]])
        # Adjust the shape of joint_i to match torque_i
        power[i,:] = np.abs((torque_i*np.diff(joint_i))/time_step)
        cot[i,:] = power[i,:]/(mass*gravity*forward_velocity)

    average_cot = np.average(cot)
    return average_cot
```

2 Central Pattern Generator

2.1 CPG Theory and Oscillator

A CPG is a neural circuit that can produce rythmic outputs which drive some sort of rhythmic biological process like breathing or walking. In vertebrates, the spine is a CPG which takes inputs from brain via descending modulation, and converts them to a rhythmic central pattern. This allows for a distributed control scheme that takes some of the computational demand off the brain, as it no longer needs to inform the exact movements of the entire body at all times. Rather, it has low bandwidth communication with the spinal cord, which then encodes a central rhythmic pattern, and enforces local control to match that pattern. For this reason, the CPG control scheme has been used in legged robotics to reduce computational load, and is helpful when attempting to enforce a steady state walking gait. The neural circuit of the CPG can be modeled as a system of coupled oscillators which exhibit limit cycle behavior, meaning that a steady state pattern can be generated. The state of each oscillator can be defined by its phase θ , and amplitude r which are defined by the following equations.

$$\dot{r_i} = \alpha(\mu - r_i^2)r_i \tag{8}$$

$$\dot{\theta}_i = \omega_i + \sum_j r_j w_{ij} \sin(\theta_j - \theta_i - \phi_{ij}) \tag{9}$$

In our project, each leg had a corresponding oscillator which it would take commands from. By looping through each leg, we were able to apply the above equations. Next, we used numerical integration to calculate r and θ. The following code displays this implementation.

```
for i in range(4):
      # Initialize r and theta
      r, theta = X[:,i]
      r_dot = self._alpha*(self._mu-r**2)*r
      theta = theta % (2*np.pi)

      # Assign natural frequency based on the phase the leg is in (swing or stance)
      if 0 <= theta <= np.pi:
        theta_dot = self._omega_swing
      else:
        theta_dot = self._omega_stance

      # Make seperate variable containing the value we take sine of in theta_dot equation
      sin_matrix = np.sin(X[1,:]-theta-self.PHI[i,:])

      # Carrying out the summation in theta_dot equation
      if self._couple:
        theta_dot += np.sum(X[0,:]*self._coupling_strength*sin_matrix)

      # set X_dot[:,i]
      X_dot[:,i] = [r_dot, theta_dot]

# Perform Euler integration
self.X = X + (X_dot_prev + X_dot) * self._dt / 2
self.X_dot = X_dot
# mod phase variables to keep between 0 and 2pi
self.X[1,:] = self.X[1,:] % (2*np.pi)
```

Here, w_{ij} represents the coupling weight between oscillators. In our simulations, the coupling weight is constant between all of the oscillators, so this is simply a scalar variable. If they differed, the coupling weights would be packaged into a matrix of dimensions i x j. ϕ represents the phase coupling between oscillators. This variable can be used to define how and when the limbs move relative to one another. So, by defining ϕ, we can define the gait of the quadruped:

$$\phi_{\mathbf{trot}} = \begin{bmatrix} 0 & \pi & \pi & 0 \\ -\pi & 0 & 0 & -\pi \\ -\pi & 0 & 0 & -\pi \\ 0 & \pi & \pi & 0 \end{bmatrix}$$

$$\phi_{\mathbf{bound}} = \begin{bmatrix} 0 & 0 & \pi & \pi \\ 0 & 0 & \pi & \pi \\ -\pi & -\pi & 0 & 0 \\ -\pi & -\pi & 0 & 0 \end{bmatrix}$$

$$\phi_{\mathbf{walk}} = \begin{bmatrix} 0 & \pi & -\pi/2 & \pi/2 \\ \pi & 0 & \pi/2 & -\pi/2 \\ \pi/2 & -\pi/2 & 0 & \pi \\ -\pi/2 & \pi/2 & \pi & 0 \end{bmatrix}$$

$$\phi_{\mathbf{pace}} = \begin{bmatrix} 0 & \pi & 0 & \pi \\ -\pi & 0 & -\pi & 0 \\ 0 & \pi & 0 & \pi \\ -\pi & 0 & -\pi & 0 \end{bmatrix}$$

2.2 CPG Results

The implementation of the CPG was successful in introducing stable gaits to the simulated robot. Figure 2 shows the outputs of the state variables r, $\dot{r}$, θ, and $\dot{\theta}$ for the thigh joint on each leg.

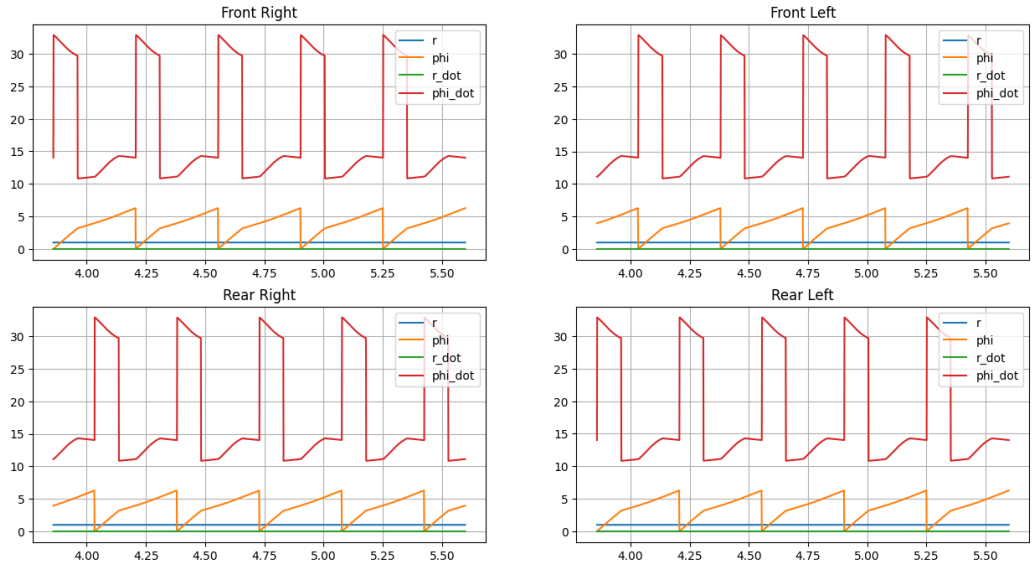


Figure 2: Oscillator state variables for the Trot gait

Figure 2 shows the gait after convergence, and the foot strike pattern is clearly visible in looking at the plot of θ. Contralateral limbs are coordinated (Front right and Rear Left for example), just as they should be for a trot. In contrast, if we rerun the simulation with a Bound gait, we get the following plot of state variables for the same oscillator.

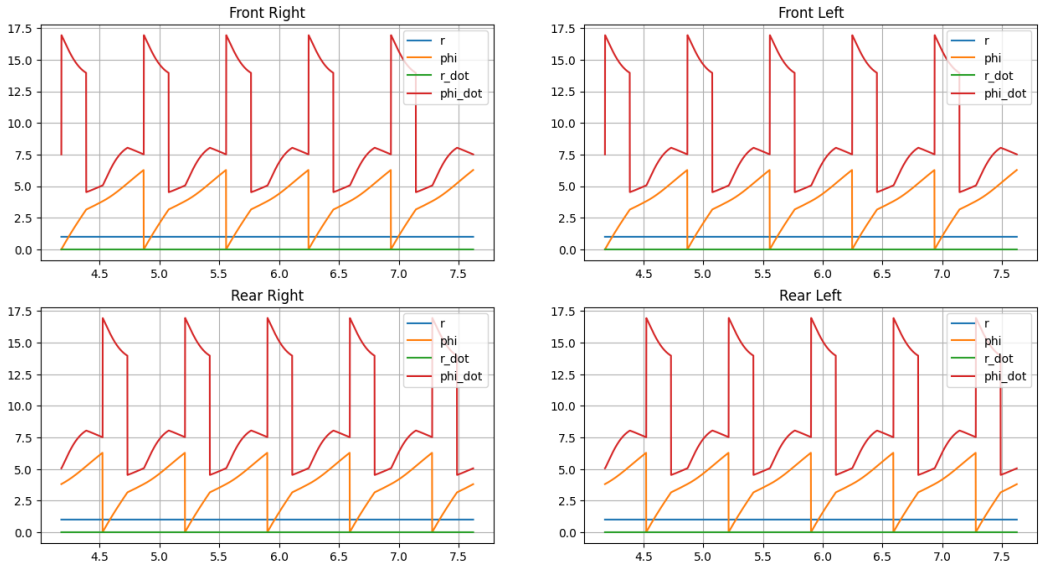


Figure 3: Oscillator state variables for the Bound gait

Here we see the difference in gait cycle, where rather than contralateral coupling, we have simple lateral coupling. The front legs are in phase, as are the rear legs. It should be noted that in both plots of oscillator states, the amplitude is constant, and therefore the time derivative of the amplitude is zero. This is because we assign a constant desired amplitude to each oscillator, as is characteristic of a limit cycle. Any perturbation of the amplitude would result in a quick convergence back to stability. In the transient period before a limit cycle is achieved, the amplitude does ramp up to match its desired value.

Although the gaits were successfully implemented to the limb oscillators, the oscillator states had to be converted to joint commands. To achieve this, we mapped the oscillator states to foot positions in the Cartesian plane using the following equations and code.

$$x_{\text{foot}} = -d_{\text{step}}\, r_i \cos(\theta_i) \tag{10}$$

$$z_{\text{foot}} = \begin{cases} -h + g_c \sin(\theta_i) & \text{if } \sin(\theta_i) > 0 \\ -h + g_p \sin(\theta_i) & \text{otherwise} \end{cases} \tag{11}$$

```
for i in range(4):
     if np.sin(self.X[1,i])>0:
       z[i] = -self._robot_height +self._ground_clearance*np.sin(self.X[1,i])
     else:
       z[i] = -self._robot_height +self._ground_penetration*np.sin(self.X[1,i])

   # scale x by step length
   if not self.use_RL:
     x = -self._des_step_len*self.X[0,:]*np.cos(self.X[1,:])
     return x, z
```

These Cartesian positions were used to apply the Cartesian control described in equation 6. The positions were mapped to corresponding joint positions using inverse kinematics and the Jacobian to apply the joint control described by equation 5. A comparison between the Trot gait with Cartesian control and without Cartesian control lead us to the conclusion that Cartesian control leads to a notable decrease in COT. This can be attributed to a tighter containment of the feet along their respective trajectories. Therefore, less correction needs to be done along each step, and less motor power needs to be spent over time. Using the default values for the joint gains and the frequencies, we achieved a modestly successful trot with an average forward velocity of 0.443 m/s and a Cost of transport of 0.935. When the cartesian control was added, we saw a slight increase in velocity to 0.451 m/s, and a drop in the COT to 0.633. The time evolution of the foot and joint states for each test are shown in figures 4 through 5. Immediately, the similarity between the two is apparent. The most notable difference is the tendency of the foot to deviate more in the y direction without Cartesian control. Its likely that including the Cartesian control makes the gait more stable against local perturbations, like these horizontal movements, but doesn't greatly affect an already stable gait.

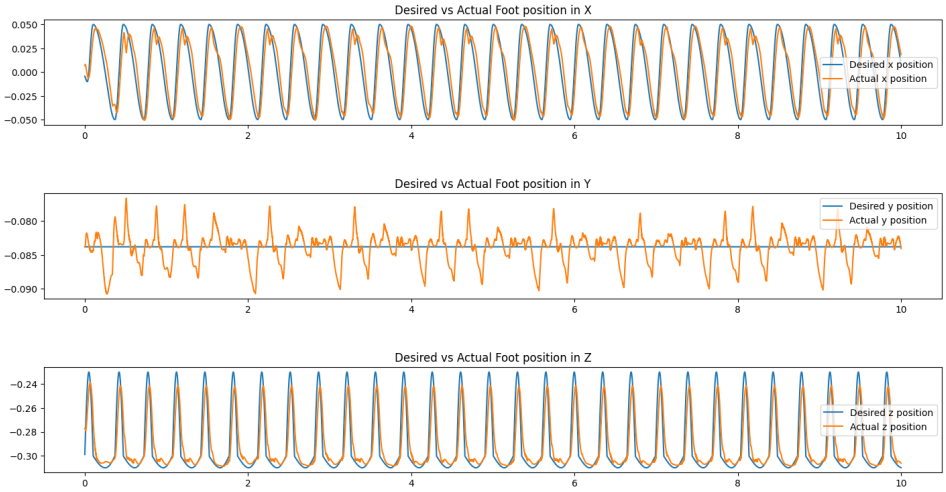


Figure 4: Desired foot position vs Actual Foot position for Front Left foot with no Cartesian PD control applied

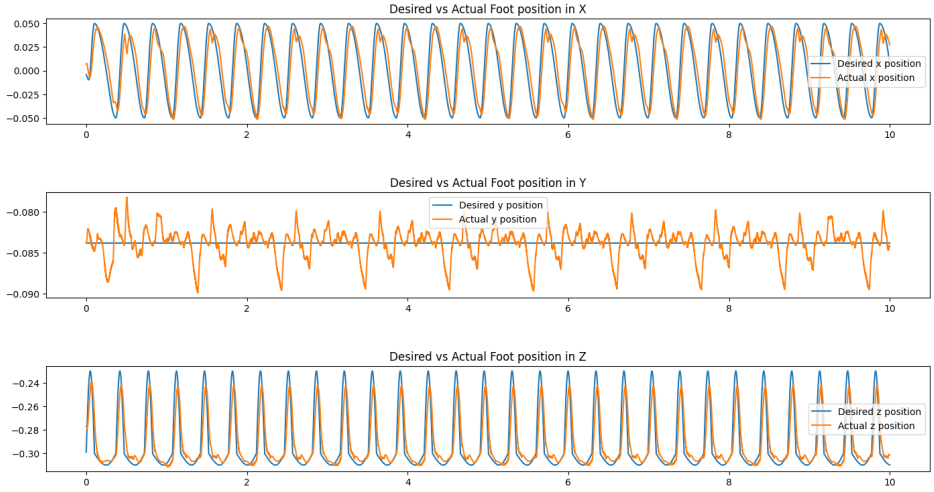


Figure 5: Desired foot position vs Actual Foot position for Front Left foot with Cartesian PD applied

Test Number	v_{avg}	COT	ω_{swing}	ω_{stance}	DC_{swing}	$K_{P,Cart}$	$K_{D,Cart}$	$K_{P,Joint}$	$K_{D,Joint}$
1	0.443	0.935	2π	4π	29.1	0	0	300	6
2	0.451	0.633	2π	4π	29.1	500	20	300	6
3	0.452	0.723	2π	4π	29.1	300	10	300	6
4	0.447	0.603	2π	4π	29.1	100	5	300	6
5	0.435	0.530	2π	4π	29.1	50	2	300	6
6	0.448	0.621	2π	4π	29.1	25	2	300	6
7	0.449	0.612	2π	4π	29.1	10	1	300	6
8	0.295	0.113	2π	4π	29.1	10	1	100	2
9	0.252	1.03	2π	4π	29.1	10	1	50	1
10	0.413	0.069	2π	4π	29.1	10	1	200	4
11	0.4479	0.590	2π	4π	29.1	10	1	300	4
12	0.147	0.369	2π	2π	50	10	1	300	4
13	0.547	0.126	2π	6π	37	10	1	300	4
14	0.648	0.220	20π	8π	29.1	10	1	300	4

Table 1: Testing and performance parameters for each test conducted while tuning the Trotting Gait

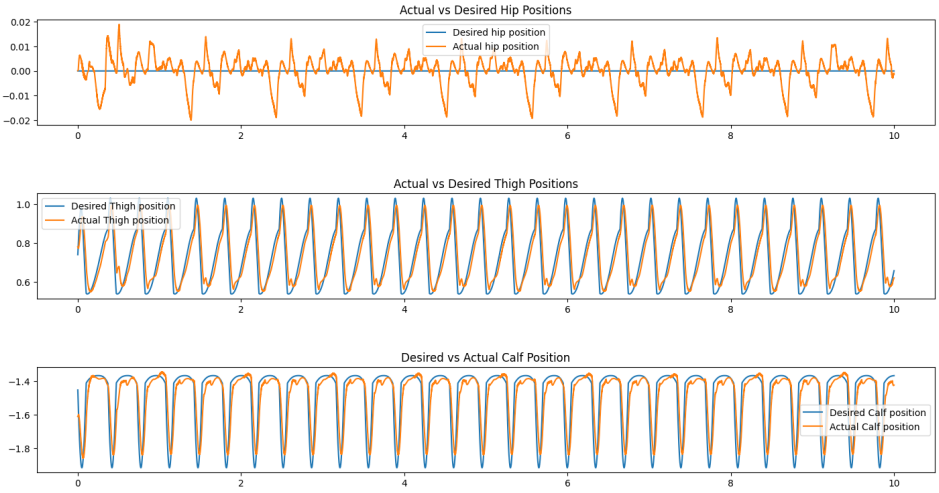


Figure 6: Desired joint position vs Actual joint position for Front Left Thigh with Cartesian PD applied

The stability of the gait depends on the control gains and the values of other parameters like the swing and stance frequencies. These frequencies are used to define the time evolution of θ, as shown in equation 9. For our purposes, the the stance phase was defined as the first half of the oscillation $(0 - \pi)$ and the swing phase was defined as the second half of the oscillation $(\pi - 2\pi)$. The values for each of these frequencies were varied in order to achieve quick body velocities and stable gaits.

Following this first system runs with default parameters, we began tuning the stance and swing frequencies along with the PD gains in attempts to improve the performance of the system. All of the tests we conducted (with Trot gait) and their associated parameters are displayed in Table 1. The highest body velocity achieved was 0.6479 m/s with a COT of 0.22 in test 14. This was achieved by increasing the swing and stance frequency, while keeping the swing duty cycle around 29%, and altering the PD gains. Specifically, the swing frequency was 20π and the stance frequency was 8π. We found the robot to be more sensitive to changes in the joint PD, so we changed the $K_{p,joint}$ to 300, and the $K_{d,joint}$ to 4. The $K_{p,cart}$ was set to 10 and the $k_{d,cart}$ was set to a low value of 1.

We also changed the foot swing height from the default of 0.07 to 0.03 in an attempt to decrease the

cost of transport. It did slightly improve the high performance gait from a COT of 0.220 to 0.204. After successfully achieving the highest velocity we could with the Trot gait, we moved on to testing other gaits. The first alternate gait we attempted was the bounding gait, whose Hopf states can be seen in figure 3. The reason we attempted this gait was because it seemed to be the next most inherently stable. At all points (assuming the frequency was low enough) two feet would be in contact with the ground. After some trial and error, we achieved a stable gait with the parameters listed in Table 1. The resulting foot and joint positions are shown in figures 7 and 8.

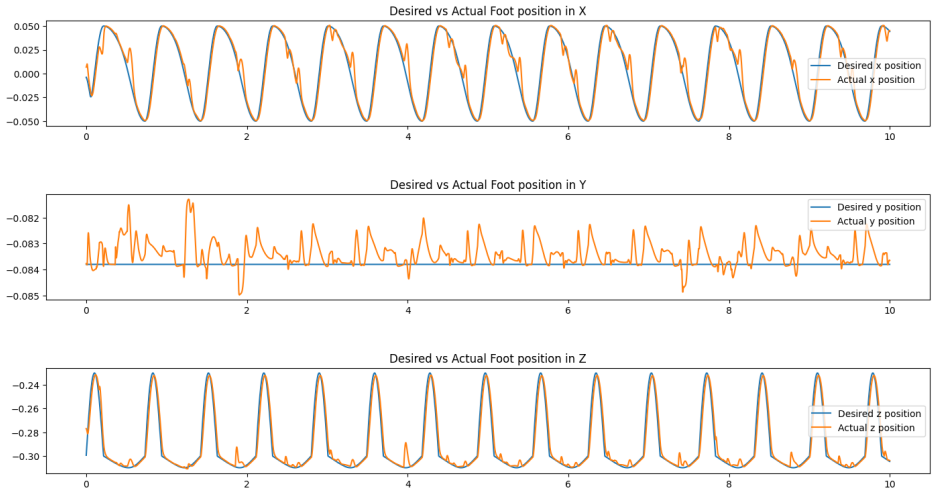


Figure 7: Desired foot position vs Actual Foot position for Front Left foot

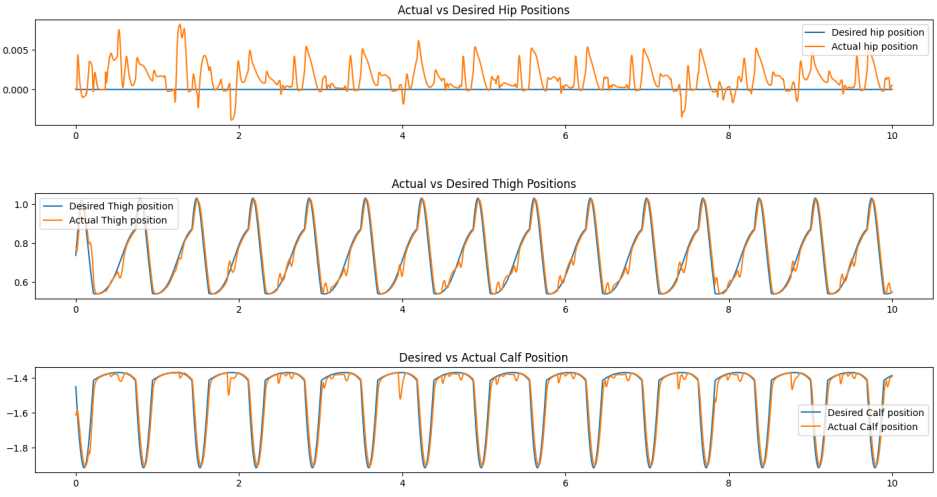


Figure 8: Desired joint position vs Actual joint position for Front Left Thigh

The end result yielded a velocity lower than the Trot gait, presumably because we were forced to lower the leg frequencies to ensure stability. We also tweaked the step height to make sure the back of the robot stayed relatively horizontal. We did attempt to implement gaits with the pace and the walk, but were unsuccessful. These gaits require more dynamic movement, and are therefore more sensitive to slight variations in gait caused by environmental interference. We tried numerous combinations of hyperparameters, but without some sort of optimization, reaching a stable gait proved difficult. This represents the drawback of manual tuning, and is something that we attempt to mitigate by using reinforcement learning in the following section.

3 Deep Reinforcement Learning

In this section, we will present the design of a Markov Decision Process (MDP) for the locomotion of our quadruped robot. We will present the different parts of the Reinforcement Learning method (action space, observation space) and then explain our choice of Reinforcement Learning algorithm and the design of our policy.

Reinforcement learning is a method of Machine Learning which consists of training an agent to learn to operate in a set environment by setting rewards for wanted behaviors and setting penalties for unwanted behaviors. To do so, the agent uses a policy that dictates the actions to take as a function of the agent's state and the environment. The goal is to maximize the reward so that the agent ultimately learns to execute the optimal wanted behaviors in an environment.

In legged robots, we can use Reinforcement Learning rather than traditional optimal control (where one develops specific mathematical models) in order for the robot to learn how to do certain behaviors such as simply walking, jumping over obstacles or walking in a specific complex terrain. Indeed, since the robot learns how to adapt its actions in a consequent number of random training episodes, it learns to react in a lot of different scenarios and can therefore be a very useful learning method in the case of an unstructured, rough and unknown terrain.

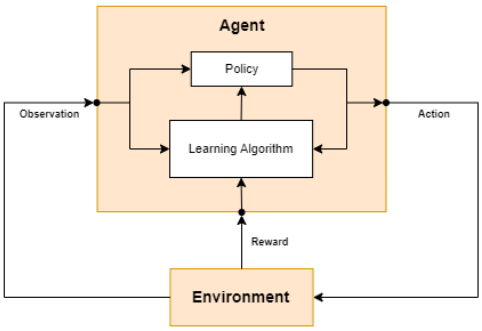


Figure 9: Reinforcement Learning method diagram

As shown on this diagram, the agent takes certain actions dictated by the policy based on the rewards (or penalties) and observations received by the environment. The action space and observation space of our project are detailed in the following sections.

3.1 Action Space

The Action Space is the set of all possible actions that an agent can take in a given environment. Agents make decisions by selecting actions from this action space to maximize some notion of cumulative reward. In our case the Action Space can be implemented with different methods. (1) joint position action space with joint PD control, (2) Cartesian space action space with Cartesian PD control, (3) learning to modulate CPG parameters and map these to joint commands (CPG-RL).

3.1.1 Different methods for the action space

We started by testing the DEFAULT parameters for the action which gave us this result:

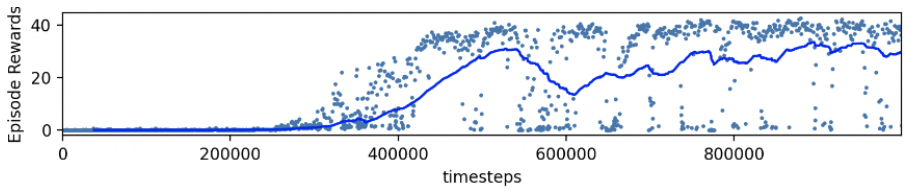


Figure 10: Reward function for PD controller with SAC with PD control

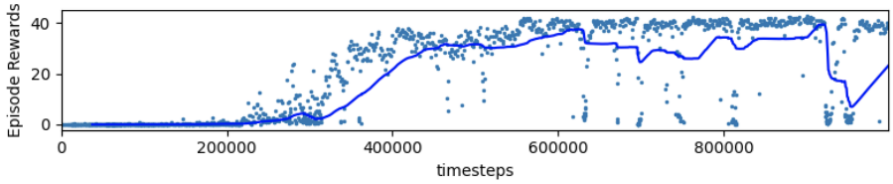


Figure 11: Reward function for PD controller with SAC with Cartesian PD control

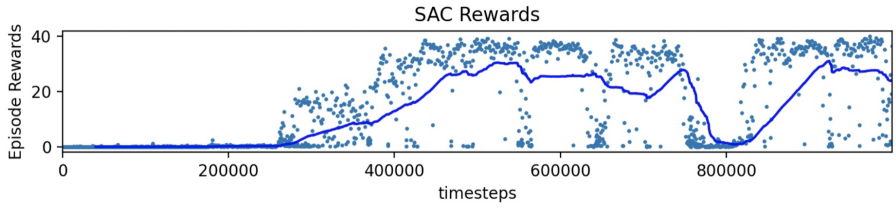


Figure 12: Reward function for PD controller with SAC with CPG Control

We run this default configuration with PD, Cartesian PD and CPG to figure out which motor control approach gave the best result. The primary distinction between the third control methods discussed earlier lies in the dimensionality of the state space. In the context of Cartesian PD, Torque, and PD controllers, individual control is exerted over each of the 12 motors across the four legs, resulting in a total of 12 control variables. This approach is mirrored in the CPG action space, where we allocate 4 states for the amplitudes of the oscillators and an additional 4 states for the phases of the oscillators.

In here we can see that the reward is better with the Cartesian PD control, than for the PD and the CPG. In these plots, all controllers seem to improve performance over time, as indicated by the increasing trend in rewards. However, the Cartesian PD controller Fig. 9 shows a more consistent and higher reward towards the later timesteps compared to the PD controller Fig. 10 and the CPG in Fig.11, suggesting that the Cartesian PD controller is leading to better performance. We will now focus on the Cartesian PD.

3.2 Observation Space

An observation space in the context of reinforcement learning refers to the set of all possible states that an agent can observe from the environment. An "observation" typically represents the information or sensory input available to the agent at a given point in time. The observation space is critical because it directly influences the agent's ability to make decisions and learn from the environment. Changing the observation space may help the reward function to be more accurate since we are looking at more variables to better fit the model. We had the following for the *Default* observation space mode:

```
if self._observation_space_mode == "DEFAULT":
     self._observation = np.concatenate((self.robot.GetMotorAngles(),
                                         self.robot.GetMotorVelocities(),
                                         self.robot.GetBaseOrientation() ))
```

For the LR COURSE observation space, we decided to add the foot position of the robot, the foot velocity, the torques of the motors and the boolean value of whether the foot is in contact with the ground or not:

```
elif self._observation_space_mode == "LR_COURSE_OBS":
    foot_position = np.zeros(12)
    foot_velocity = np.zeros(12)
    for i in range(4):
        J, ee_pos_legFrame = self.robot.ComputeJacobianAndPosition(i)
        foot_position[3*i:3*i+3] = ee_pos_legFrame
        foot_velocity[3*i:3*i+3] = J @ self.robot.GetMotorVelocities()[i*3:i*3+3]

    self._observation = np.concatenate((self.robot.GetMotorAngles(),
                                        self.robot.GetMotorVelocities(),
                                        self.robot.GetBaseOrientation(),
                                        foot_position,
                                        foot_velocity,
                                        self.robot.GetContactInfo()[3],
                                        self.robot.GetMotorTorques() ))
```

We also implemented a CPG observation space, in which we added the CPG phase and amplitude of the steps of our robot, as well as the derivatives of this phase and amplitude :

```
elif self._observation_space_mode == "CPG":
      foot_position = np.zeros(12)
      foot_velocity = np.zeros(12)
      for i in range(4):
        J, ee_pos_legFrame = self.robot.ComputeJacobianAndPosition(i)
        foot_position[3*i:3*i+3] = ee_pos_legFrame
        foot_velocity[3*i:3*i+3] = J @ self.robot.GetMotorVelocities()[i*3:i*3+3]

      self._observation =np.concatenate((self.robot.GetMotorAngles(),
                                    self.robot.GetMotorVelocities(),
                                    self.robot.GetBaseOrientation(),
                                    foot_position,
                                    foot_velocity,
                                    self.robot.GetContactInfo()[3],
                                    self.robot.GetMotorTorques(),
                                    self._cpg.get_r(),
                                    self._cpg.get_theta(),
                                    self._cpg.get_dr(),
                                    self._cpg.get_dtheta()))
```

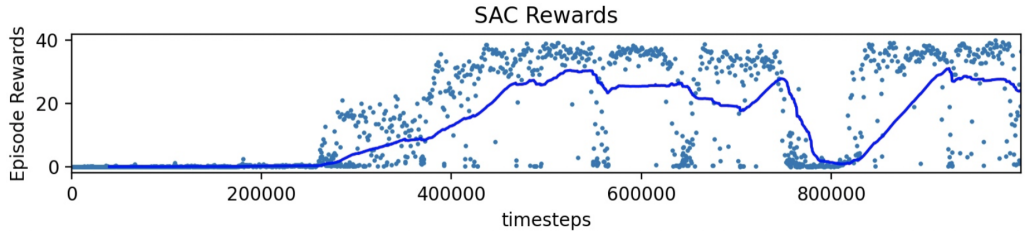


Figure 13: Reward fwd with a default observation space, for SAC and CPG motor mode

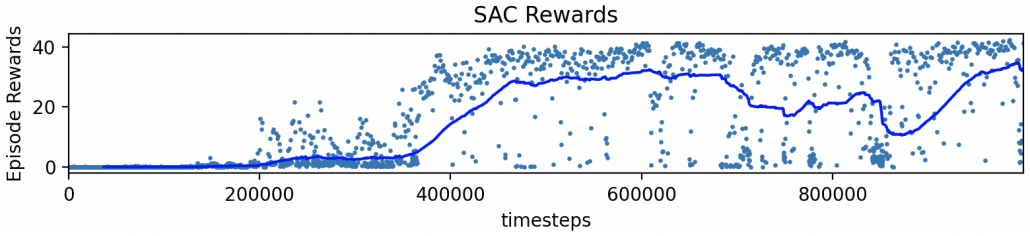


Figure 14: Reward fwd with our lr course observation space, for SAC and CPG motor model

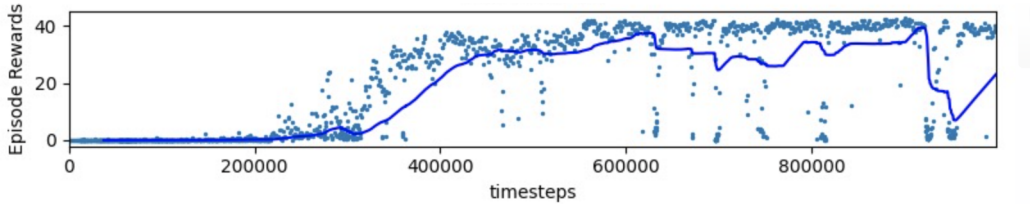


Figure 15: Reward fwd with default observation space, for SAC and cartesian_pd motor model

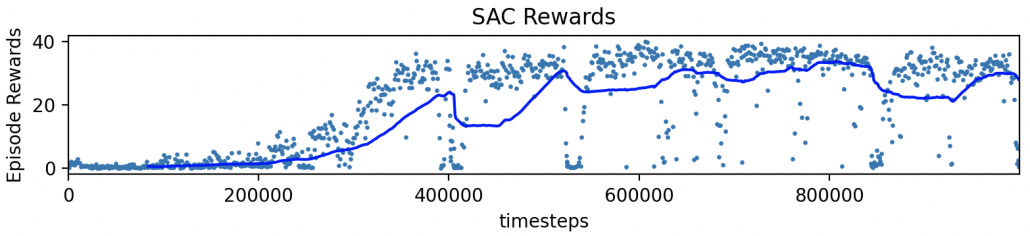


Figure 16: Reward fwd with our cpg observation space, for SAC and cartesian_pd motor model

Based on the two first plots, our observation space LR_COURSE could be considered better for this example because it likely provides information that enables the agent to learn more effectively and to have a more consistent policy. This is due to the fact that our observation space contains more features that are beneficial for the task, allowing the algorithm to better interact with the environment to achieve its goals.

With the two last plots, we can compare our CPG observation space to the default one with the Cartesian PD motor control. We observe that the default observation space gives a smoother and more stable value for 0 to 600 000 time-steps but after that, our CPG observation space gives a smoother and more stable value. We believe that if we could train it over more time-steps our CPG observation space might be better than the default one.

3.3 Reward function

As previously explained, the goal of the agent (quadruped robot in our case) is to learn an optimal, or nearly-optimal, policy that maximizes the reward function. A reward function associates an environment's state with a numerical value, giving the reward or penalty linked to this specific state. Maximizing this function allows the agent to learn to engage in behaviors that optimize these rewards. The design of the reward function is crucial in reinforcement learning since it contains the goals of the agent and dictates to the agents the actions to take to achieve them. Indeed, a well constructed reward function enables the agent to learn optimal policies that align with the task objectives. On the other hand, a poor reward function may result in unexpected agent behaviors and the agent may not achieve to learn the goal tasks and actions that the training was meant for.

In the case of this project, the main goals of our reward function are for the quadruped robot to walk forward without falling over.

The following reward function was already implemented for the "FWD_LOCOMOTION" observation space mode. As we can see, it encourages the robot to walk straight forward at a steady desired pace. It achieves it by setting penalties for the yaw angle (to walk straight), the lateral drift of the robot and its used energy. It sets rewards for the tracking of the velocity which means that the smaller the difference between the desired velocity and its actual velocity, the better.

```
def _reward_fwd_locomotion(self, des_vel_x=0.5):
        """Learn forward locomotion at a desired velocity. """
        # track the desired velocity
        vel_tracking_reward = 0.05 * np.exp(-1 / 0.25 * (self.robot.GetBaseLinearVelocity()[0] -
            ↪ des_vel_x) ** 2)
        # minimize yaw (go straight)
        yaw_reward = -0.2 * np.abs(self.robot.GetBaseOrientationRollPitchYaw()[2])
        # don't drift laterally
        drift_reward = -0.01 * abs(self.robot.GetBasePosition()[1])
        # minimize energy
        energy_reward = 0
        for tau, vel in zip(self._dt_motor_torques, self._dt_motor_velocities):
            energy_reward += np.abs(np.dot(tau, vel)) * self._time_step

        reward = vel_tracking_reward \
                + yaw_reward \
                + drift_reward \
                - 0.01 * energy_reward \
                - 0.1 * np.linalg.norm(self.robot.GetBaseOrientation() - np.array([0, 0, 0, 1]))
```

We implemented our own reward function:

```
def _reward_lr_course(self, des_vel_x=1):
    """Learn left-right course navigation."""
    # track the desired velocity
    vel_tracking_reward = 0.05 * np.exp(-1 / 0.25 * (self.robot.GetBaseLinearVelocity()[0] - des_vel_x
        ↪ ) ** 2)
    # minimize yaw (go straight)
    yaw_reward = -0.2 * np.abs(self.robot.GetBaseOrientationRollPitchYaw()[2])

  # reward facing forwards
    heading_reward = np.sum(
           np.cos(self.robot.GetBaseOrientationRollPitchYaw())) * self._time_step * self.
               ↪ _action_repeat
     # penalize crouching
    crouching_penalty = -abs(
           self.robot.GetBasePosition()[2] - self.robot._GetDefaultInitPosition()[2]) * self.
               ↪ _time_step
    # don't drift laterally
    drift_reward = -0.01 * abs(self.robot.GetBasePosition()[1])

     # penalize sideways motion
    current_base_position = self.robot.GetBasePosition()
    sideways_penalty = -np.abs(current_base_position[1] - self._last_base_position[1])

    # minimize energy
    energy_reward = 0
    for tau, vel in zip(self._dt_motor_torques, self._dt_motor_velocities):
        energy_reward += np.abs(np.dot(tau, vel)) * self._time_step

    reward = vel_tracking_reward \
                + yaw_reward \
```

```
            + heading_reward \
            + 1.5 * crouching_penalty \
            + drift_reward \
            - 0.00001 * sideways_penalty \
            - 0.01 * energy_reward \
            - 0.1 * np.linalg.norm(self.robot.GetBaseOrientation() - np.array([0, 0, 0, 1]))

    return max(reward, 0) # keep rewards positive
```

While maintaining the original goal of velocity tracking, the new function sets a higher desired speed, which raises the challenge for the robot to maintain a quicker pace. This is complemented by the addition of a heading reward, which explicitly incentivizes the robot to keep its orientation aligned with the direction of travel. The new reward structure introduces a crouching penalty that discourages significant vertical deviations from a set baseline height, ensuring the robot maintains a consistent posture, which is crucial for stability and obstacle avoidance. The sideways penalty in the new reward function is designed to prevent unwanted lateral or sideways movement. This penalty is intended to discourage the robot from drifting off course.

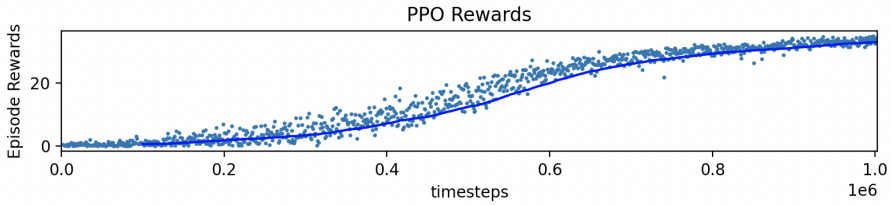


Figure 17: FWD_LOCOMOTION reward function with our lr_course observation space, for PPO algorithm and CARTESIAN_PD motor model

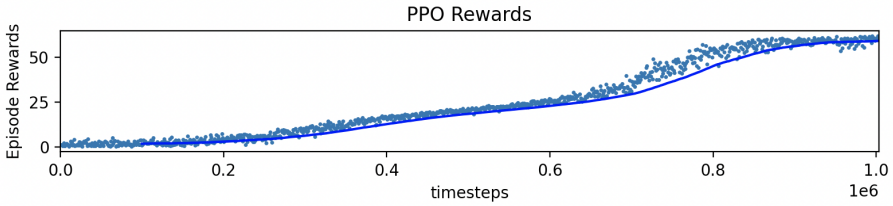


Figure 18: Our LR_COURSE_TASK reward function with our lr_course observation space, for PPO algorithm and CARTESIAN_PD motor model

The two previous plots show the evolution of the rewards over the time-steps for the forward locomotion reward function (Figure 17) and for our lr_course_observation reward function both with the PPO algorithm, Cartesian PD control and our lr_course observation space. As we can see, our reward function converges slightly more at a million time-steps and it converges to higher reward value. Indeed, it converges to a value around 70 as opposed to around 40 for the forward locomotion reward function.

3.4 Deep reinforcement learning algorithms: PPO and SAC

3.4.1 Definitions of the algorithms

Fo our project, we have two possible algorithm implementation to make our robot walk and be stable. These ttwo are PPO and SAC.

Proximal Policy Optimization (PPO) is the first type of reinforcement learning algorithm. It iteratively updates policies that define the agent's behavior, ensuring small stable changes by clipping

the policy update rate, preventing drastic policy performance drops. PPO balances performance and computational efficiency. Its key advantage lies in its simplicity.

Soft Actor-Critic (SAC) is a state-of-the-art reinforcement learning algorithm that optimizes a policy in an off-policy way, emphasizing the trade-off between exploration and exploitation. It works well in situations where actions are continuous and complex, like controlling a robot's movement.It's known for its sample efficiency, stability, and robustness, particularly in continuous action spaces. SAC is efficient and stable because it constantly balances trying out new things with using what it already knows.This results in a more balanced and effective learning process.

3.4.2 Hyperparameters

These are the hyperparameters of the PPO algorithm, they could be find in the run_sb3.py file that is available in our project.

Parameter	**Value/Type**	**Explanation**
gamma	0.99	Discount factor for calculating future rewards.
n_steps	4096	Number of steps to collect before updating the policy.
ent_coef	0.0	Coefficient for the entropy term in the loss function, encouraging exploration.
learning_rate	1e-4	Step size for policy updates.
vf_coef	0.5	Coefficient for the value function loss in the total loss.
max_grad_norm	0.5	Maximum norm for gradient clipping, to prevent large updates.
gae_lambda	0.95	Trade-off parameter for bias-variance in Generalized Advantage Estimation.
batch_size	128	Number of samples in each mini-batch for gradient descent.
n_epochs	10	Number of passes over the entire dataset for updating the policy.
clip_range	0.2	Clipping parameter for policy update, to prevent large updates.
clip_range_vf	1	Clipping parameter for value function updates.
verbose	1	Verbosity of the training process (0 = silent, 1 = progress bar, 2 = full info).
tensorboard_log	None	Directory for TensorBoard logs.
_init_setup_model	True	Whether to set up the model when the PPO instance is created.
policy_kwargs	[-256,256]	Additional keyword arguments for the policy.
device	gpu_arg: auto or cpu	Device to use for computation (CPU or GPU).

Table 2: Hyperparameters for PPO Configuration

Parameter	**Value/Type**	**Explanation**
learning_rate	1e-4	Step size for updating the weights of the neural network.
buffer_size	300000	Size of the replay buffer used to store past transition information.
batch_size	256	Number of samples to use when performing a gradient update.
ent_coef	'auto'	Coefficient of entropy regularization in the loss function. 'Auto' for automatic tuning.
gamma	0.99	Discount factor for future rewards.
tau	0.005	Soft update coefficient for the target network.
train_freq	1	Frequency of the training process.
gradient_steps	1	Number of gradient steps to take for each training iteration.
learning_starts	10000	Number of steps before training starts to populate the replay buffer.
verbose	1	Verbosity level of the training output (0 = silent, 1 = progress bar, 2 = full info).
tensorboard_log	None	Directory for TensorBoard logging.
policy_kwargs	[-256,256]	Additional keyword arguments for the policy network architecture.
seed	None	Random seed for reproducibility.
device	gpu_arg: auto or cpu	Device to run computations on

Table 3: Hyperparameters for SAC Configuration

The primary differences between PPO and SAC based on their hyperparameters are:

- **PPO:**
  - Employs a fixed number of steps per update (`n_steps`).
  - Uses clipping (`clip_range`) to ensure stable policy updates.
  - Focuses on on-policy learning, optimizing the policy that is being used to interact with the environment.
- **SAC:**
  - Utilizes a replay buffer (`buffer_size`) indicative of its off-policy learning approach.
  - Includes entropy coefficients (`ent_coef` to promote exploration.
  - Adopts a different policy update frequency (`train_freq` and `gradient_steps`).

These hyperparameters highlight PPO's focus on stable policy updates and SAC's emphasis on exploration and learning from a large range of experiences.

3.4.3 Comparison between the two algorithms

We have used these two algorithms to train the same settings and same models to compare them. We tried these two algorithms for the Lr_course observation space, which is the one that we made,

the Cartesian pd control as well as the default FWD_LOCOMOTION reward function and got the following results:

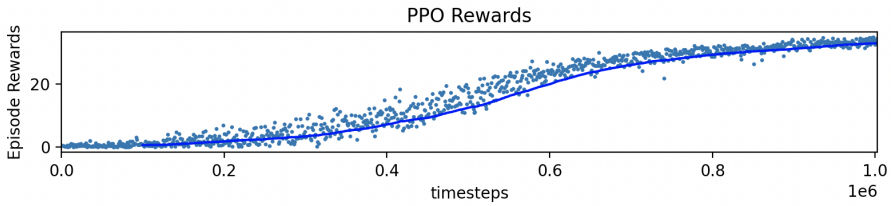


Figure 19: FWD_LOCOMOTION reward function with our lr_course observation space, for PPO algorithm and CARTESIAN_PD motor model

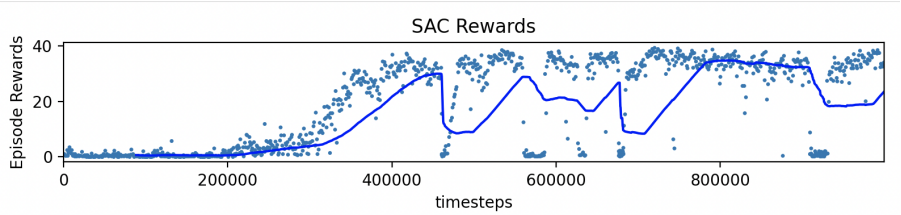


Figure 20: FWD_LOCOMOTION reward function with our lr_course observation space, for SAC algorithm and CARTESIAN_PD motor model

These two graphs show the learning progress of PPO and SAC algorithms through rewards over time. We can observe that the PPO's reward increases slowly but steadily over 1 million time-steps and then slowly converges. On the other hand, the SAC's reward increases more quickly before 500,000 time-steps but then varies a lot (by decreasing and then increasing multiple times) without increasing steadily. Generally, if quicker learning is needed, SAC may be preferable, while PPO might be better for consistent performance over time. We can also observe that the reward function of the PPO algorithm approaches a linear function between approximately 400 000 time-steps until 800 000 time-steps, where as the SAC reward function does not show any linearity. Moreover, we observe that there more outliers with the SAC algorithm. Finally, in terms of computation time, we noticed that the SAC algorithm is slower that the PPO algorithm. In our case, we believe that using the PPO algorithm is better and will do so for the rest of this project.

3.5 Environment details and changes

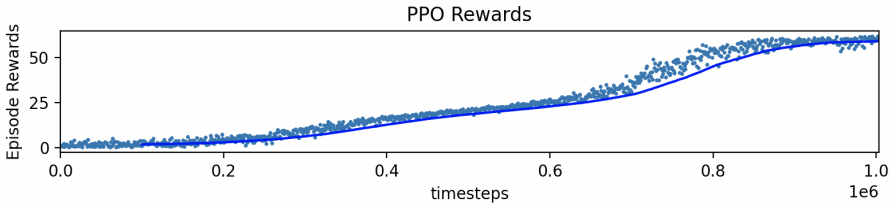


Figure 21: Our LR_COURSE_TASK reward function with our lr_course observation space, for PPO algorithm and CARTESIAN_PD motor model

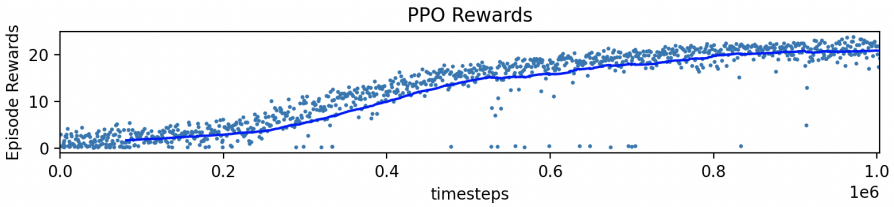


Figure 22: Our LR_COURSE_TASK reward function with our lr_course observation space, for PPO algorithm and CARTESIAN_PD motor model, trained with the environment

We tested out our robot on different environments such as the test and competition environment. These environments present obstacles of different heights.

Even though both use the same reward function and observation space, the presence of obstacles in the environment for the second scenario introduces additional complexity to the problem that the PPO algorithm is trying to solve.

Without Obstacles (Figure 20): The PPO algorithm's learning process is likely more straightforward since it does not have to account for obstacles. The reward function and the observation space are sufficient to learn a policy that maximizes rewards, potentially leading to a smoother learning curve as represented by the trend line. The absence of obstacles means the agent's policy can focus solely on optimizing the primary task without any additional constraints.

With Obstacles (Figure 21): The training process becomes more complex. The algorithm must now learn to navigate or avoid obstacles while still trying to maximize the reward function. This can lead to a more erratic set of episode rewards, especially early in training, as the algorithm tries to figure out how to deal with these new elements. Over time, if the algorithm learns effectively, it should still show an upward trend in rewards, indicating that it is learning to overcome obstacles while performing the task.

If obstacles significantly impact the agent's ability to achieve high rewards, this would usually result in a lower overall trend in the rewards, which would explain how the general value of the reward goes down from around 60 to 22.

To improve our model on such environments it could be interesting to add to the reward function a factor on how high the leg goes during the step, as well as a higher weight for the crouching We haven't changed the environment beside that. On the following pictures we can observe the different possible environments.

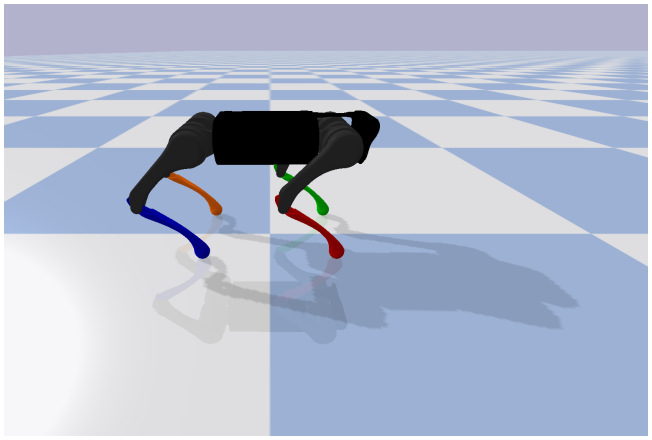


Figure 23: Normal environment

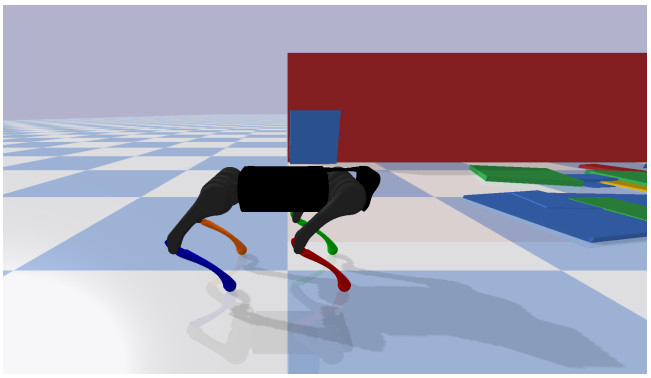


Figure 24: Test environment

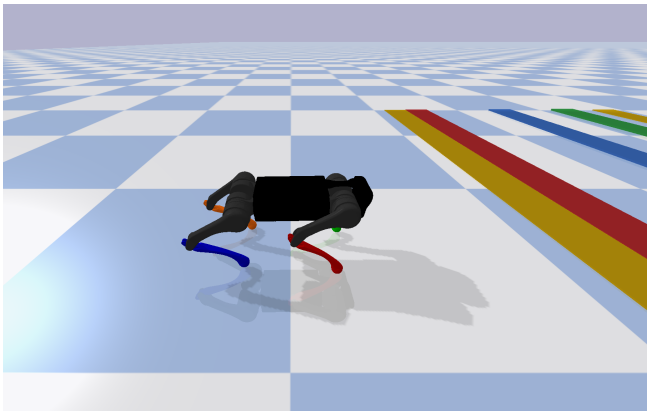


Figure 25: Competition environment

3.6 Robustness

In this section, we will discuss the robustness of our model and computed policy and the challenges we may encounter when transferring it from simulation to the real world. As useful as the PPO algorithm is for our simulations, we may encounter some difficulties when trying to implement it on a real-world application. For example, one of the possible challenges is the sample efficiency. Indeed, the PPO algorithm requires a significant amount of data to learn effectively .Real-life applications often have limited data and the PPO algorithm might not collect enough to learn an optimal policy in this case. Real-world environments can also be more complex and often have high-dimensional state spaces so it can be difficult to simulate the environment accurately. Moreover there might be physical limitations to the training, like the possibility to execute a lot of training episodes. However,if there are less episodes, less data is collected.

Another potential challenge is due to the fact that, like many other algorithms, the PPO algorithm makes the assumption that the observation states received by the environment are accurate, and changes its actions based on them. However, in the real-world, the observations and measurements by sensors or other systems, are usually noisy and not accurate, so the agent might need to estimate the state of this system from these observations. This issue might make it difficult for the agent to learn an an efficient and optimal policy since it lacks access to information crucial for making optimal decisions.

To conclude, it's important to highlight that while the PPO algorithm (as well as other reinforcement learning algorithms) might successfully learn an efficient policy within a simulation, there's no assurance that this policy will successfully be implemented on real-world applications. There are many disparities between simulation and real-world that make this transfer really challenging.

In the case of our implemented Reinforcement Learning model, we believe that our policy is not very robust yet and might not transfer successfully from simulation to the real world. However, we could make it more robust by adding, for example, reflexes or a CPG controller.

3.7 Quantification of the control policy

During our numerous trainings, we observed that the tracking of the desired velocity of the robot is best achieved when we use the PPO algorithm, the CARTESIAN_PD controller and our reward function reward_lr_course(). When we set the desired speed to 1 m/s, with this observation space, reward function and controller, the resulting speed of the robot is around 0.79 m/s (with 1 million time-steps), as shown on the resulting plot below.

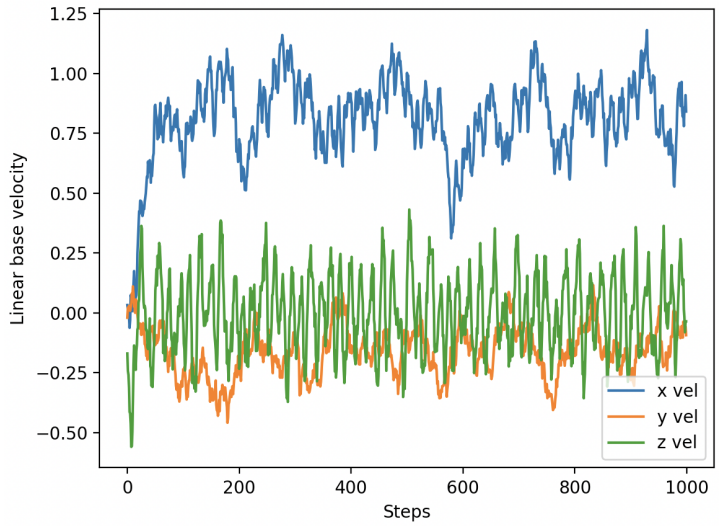


Figure 26: Plot of the average velocity

The corresponding duty cycles and Cost of Transport are the following:

```
episode_reward [50]
Final base position (7.971403537860976, -1.6569985293860896, 0.24713385617847033)
CoT:  0.04800509997854696
Average velocity:  0.7971403537860976
Duty cycles:  [[0.433 0.4 0.336 0.453]]
```

Figure 27: Average body velocity, CoT and duty cycle

The resulting duty cycles are slightly different from one leg to another but do not differ too much. This is not surprising since we didn't constraint any specific synchronized gait, as we did in CPG. Moreover, we have a COT of approximately 0.05 which is extremely low. This shows that the motion of our robot is extremely energy efficient.

Since our reward function aims at maximizing the tracking of a desired velocity and minimizing the energy used by the robot to move forward, these results suggest that our control policy is rather effective.

Another goal of our control policy was for our robot to have a stable gait and locomotion. To achieve this, we included multiple parameters in our reward function such as a penalty for its crouching.

We can observe on the following figure the resulting joint positions of the robot, that are stable over time. This shows once again that our control policy is rather effective and enables our robot to have a stable, dynamic and energy efficient gait to a desired speed in an environment.

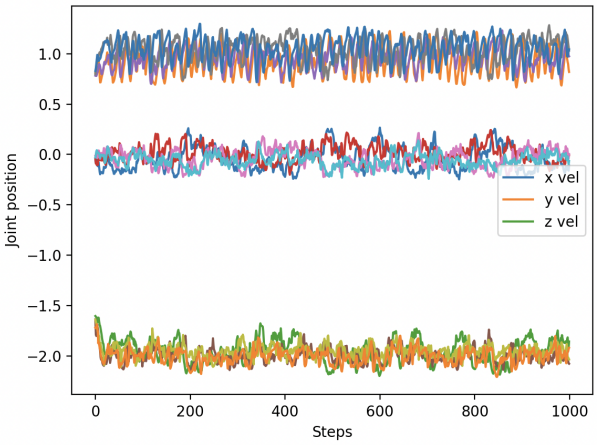


Figure 28: Joint position with respect to the time-steps

Our reward function already maximizes the difference between a desired chosen speed and the actual speed of the robot, to train the robot to walk forward at a desired set speed. In order to train a policy that we can command at test time to achieve any arbitrary speed, we would have to train multiple times our policy with a wide range of speeds. This would be achievable but would take a lot of training time.

To deal with uneven terrain, we could do multiple things such as adding reflexes or taking the contact forces on the feet of the robot as observations in our observation space.

3.8 Performance of the goal-following controller

We implemented the reward and the observation space for 'FLAGRUN' in this part, and we trained our robot. The robot is supposed to go to a set of randomized goals and to follow a certain sequence.

We also computed the speed of the turn, using the yaw angle.

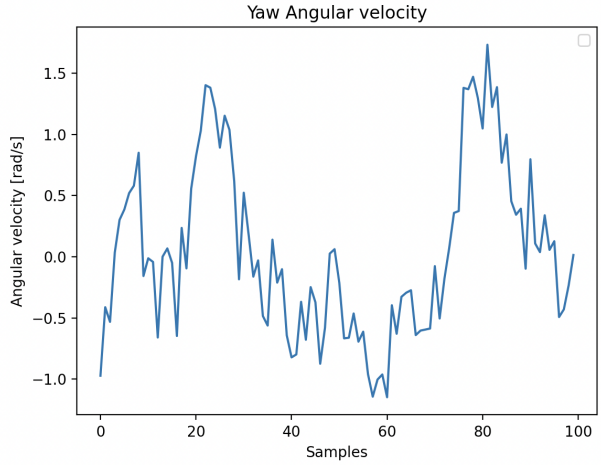


Figure 29: Angular velocity for the flagrun

The random sampling of the goal is tough on the robot. Its angle velocity and ability to turn is not as fast as the change of the goal in the simulation. We cannot visualize if it reaches the goal. This is probably due to the fact that in the simulation, the visualization of the robot's movement is slower than its computation and during the mean time the goal changes places.

To extend the MDP if the goal is know in advance we could add a few parameters to the reward function. It could account for sequential goal achievement. The reward can be designed to encourage not just reaching the current goal but also positioning the robot advantageously for the next goal. We could also update the observation space and allow it to include information about all known goals, possibly with prioritization or sequencing information

By adjusting these aspects of the MDP and training process, we could effectively train a robot to handle scenarios with multiple known consecutive goals.

4 Conclusion

This interesting project allowed us to compare two methods for quadruped robots locomotion: implementing different gaits with Central Pattern Generator and design a Markov Decision Process (MDP) to be used by deep reinforcement learning algorithms to train our robot to execute desired locomotion.

In the first part, we implemented a CPG based control, where we effectively produced a walking gait. In order to maximize the velocity of the robot, and minimize the COT, we manually tuned parameters such as the gains, swing and stance frequencies, and step height. We continued to experiment with other gaits, and were able to achieve a successful, albeit much slower, bounding gait. The walking and trotting gaits proved much more difficult to tune, as they are more dynamic, and therefore are more prone to instabilities.

In the second part, we really enjoyed comparing the different possible Reinforcement Learning algorithms that could be used, as well as the observation spaces, motor control model and the importance of the design of an effective reward function. Indeed, it was really interesting to visualize the resulting gaits and see the impact of all the parameters and parts of the Reinforcement Learning. However, we also realized what a important computational time it could take to train an agent with Reinforcement Learning (up to 10 hours with some of our computers). We therefore believe, that if we had more time available, we could have developed a better control policy with the the Deep Reinforcement Learning method. Indeed, our resulting control policy is rather effective to allow the quadruped robot to walk forward in a steady gait at a desired speed. However, it is not the most robust implementation and would probably not be successfully transferred to the real world.

To conclude, this project enabled us to compare hands-on two very popular methods for the locomotion of legged robots. We witnessed how the model-based method is more simple and faster to develop whereas the Reinforcement Learning method is more time consuming and computationally more expensive. Model-based methods can be well suited for planning and for anticipatory behavior but might struggle to adapt to unexpected or unmodeled dynamics and environment. Additionally, without any parameter optimization, its difficult to guess values that will allow the robot to walk more quickly and/or efficiently. So, without a complete picture of the environment or how to exactly tune a robot, Deep Reinforcement learning would offer an advantage. Through trial and error, Deep Reinforcement Learning allows the robot to learn more complex locomotion behaviors without explicitly modeling the environment, making them potentially more adaptable to various conditions and terrains.

References

[1] Reinforcement Learning diagram: https://swishdata.com/reinforcement-learning/