Ecole Polytechnique Fédérale de Lausanne
Legged Robots
MICRO-507

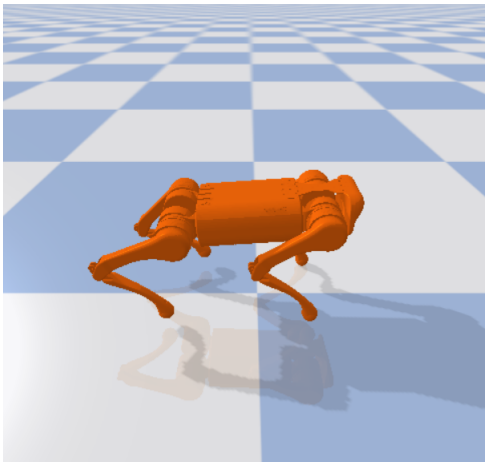
Quadruped Locomotion with Central Pattern Generators and Deep Reinforcement Learning

Legged Robots - Project 2

Catarina Pires 346796
Dimitri Hollosi 247227
Richard Gao 342177

Supervising Teacher:

Auke Ijspeert



Lausanne, January 2022

Table of Contents

1 Introduction **2**

1.1 Context . . . 2

1.2 Quadruped Modeling and Control . . . 2

1.2.1 Quadruped Modeling . . . 2

1.2.2 Control . . . 3

1.3 Report Outline . . . 3

2 Locomotion Control based on Central Pattern Generator (CPG) **4**

2.1 Theoretical Insights . . . 4

2.2 Methods . . . 5

2.2.1 Gait Implementation . . . 6

2.2.2 Parameter Tuning Methodology . . . 8

2.3 Results . . . 8

2.3.1 Trot Gait . . . 8

2.3.2 Pace Gait . . . 10

2.3.3 Lateral Sequence Walk Gait . . . 12

2.3.4 Bound Gait . . . 14

2.3.5 Summary . . . 15

2.4 Discussion . . . 16

3 Deep Reinforcement Learning for Quadruped Locomotion **17**

3.1 Theoretical Insights . . . 17

3.2 Methods . . . 17

3.2.1 Observation Space . . . 17

3.2.2 Action Space . . . 18

3.2.3 Reward Function . . . 21

3.2.4 Deep Reinforcement Learning . . . 21

3.3 Discussion . . . 22

4 Conclusive Remarks **24**

Bibliography **25**

1. Introduction

1.1 Context

For the scope of this project, it is of interest to study two different methods for quadruped locomotion. Different gaits shall firstly be implemented by means of Central Pattern Generator (CPG), after which a Markov Decision Process (MDP) shall be designed by using state-of-the-art Deep Reinforcement Learning (DRL) algorithms. Both of these methods will be implemented in Python, namely by using a customised "quadruped" environment which will be simulated in PyBullet.

1.2 Quadruped Modeling and Control

1.2.1 Quadruped Modeling

The quadruped model to be used in PyBullet has 4 legs with 3 joints each for hip, thigh and calf (q_0, q_1 and q_2 respectively, see fig.1.1) for a total of 12 motors. This therefore implies 12-length arrays shall be used when handling data back and from the motors, which will always be done in the aforementioned joint order.

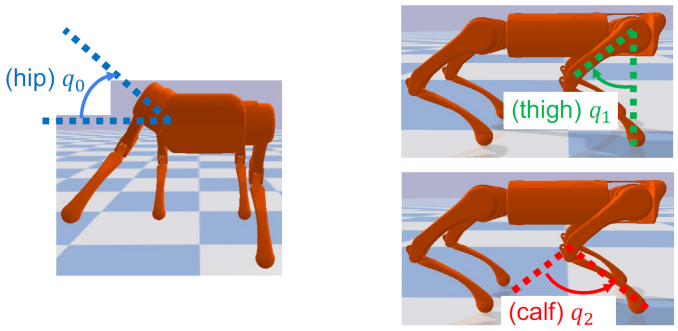


Figure 1.1: Quadruped Model Joint References

Naturally, it is important to gain a detailed understanding of the coordinate frame of the quadruped as well as the leg frame, numbered from 0-3 as Front Right (FR,0), Front Left (FL, 1), Rear Right (RR,2) and Rear Left (RL,3)), which are shown in figures 1.2 and 1.3 respectively. As an illustrative example, the coordinates of the front right end effector (i.e right foot) will intuitively be $(x, y, z) = (0, \text{- hip length}, -0.25)$ meters (the CoM is located at 0.25m from the ground).

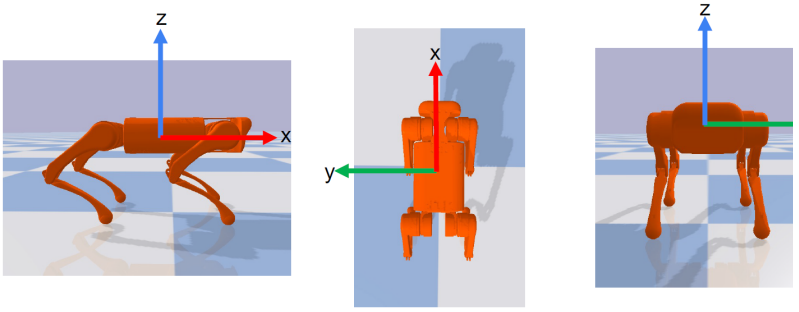


Figure 1.2: Quadruped Model Reference Frame

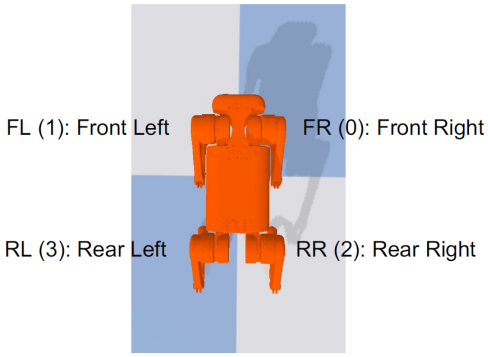


Figure 1.3: Quadruped Model Leg References

The forward kinematics $f(\cdot)$ of a single leg with respect to the leg frame portrays the relationship between the joint angles q and foot posisition p, which is recalled in equation 1.1.

$$p = f(q) \tag{1.1}$$

Differentiating the above relationship allows to extract the foot velocity:

$$v = J(q) \cdot \dot{q} \tag{1.2}$$

Which introduces the single leg Jacobian matrix $J(q) \in \mathbb{R}^{3x3}$ of a foot position with regards to the leg frame. The latter namely comes into use for mapping a desired force at the end effector to joint torques with the operational space formulation (equ. 1.3).

$$\tau = J(q)^T F \tag{1.3}$$

1.2.2 Control

Contrarily to the previous projects, it is now desired to implement both the joint PD control and Cartesian PD control within the quadruped locomotion framework. The position control in the joint space for a particular leg angle needing to be actuated to a desired angle is given by relation 1.4 shown below.

$$\tau_{joint} = K_{p,joint}(q_d - q) + K_{d,joint}(\dot{q_d} - \dot{q}) \tag{1.4}$$

Tracking performance can further be improved by using a Cartesian PD controller, which related the desired foot positions p_d and velocities v_d in the leg frame from desired joint angle and velocity trajectories (q_d and $\dot{q_d}$). From a current foot position and velocity, the associated motor torques can be computed as shown in equation 1.5 to track the desired quantities.

$$\tau_{Cartesian} = J^T(q)[K_{p,Cartesian}(p_d - p) + K_{d,Cartesian}(v_d - v)] \tag{1.5}$$

As a final note, it may also be of interest to monitor the potential contributions from both of these controllers by combining the overall torque, i.e $\tau_{final} = \tau_{joint} + \tau_{Cartesian}$

1.3 Report Outline

From the above mentioned quadruped modelling, two different locomotion tuning methods will be assessed. The first one shall be based on Central Pattern Generation, and therefore attempt to modulate parameters such as speed and heading all the while implementing common gaits found in quadruped vertebrates. Their performance will be evaluated and compared, and finally validated based on results obtained from the implemented controllers.

The second part of this report shall focus on designing a Markov Decision Process (MDP) in a Deep Reinforcement Learning framework. The agent will be trained according to the joint PD and Cartesian PD action spaces to achieve specific rewards. Additionally, the quadruped is tested in a competition environment to ensure successful implementation of the reward space after the training is complete.

Finally, the two methods shall be compared and evaluated both on their respective added benefits. A discussion on how to improve both design methodologies shall conclude this report.

2. Locomotion Control based on Central Pattern Generator (CPG)

2.1 Theoretical Insights

Central Pattern Generators are used to generate rhythmic (or patterned) outputs despite the input being of different nature, which can then be modeled with coupled oscillators to produce locomotion gaits. In the scope of this project, the CPG consists of coupled Cartesian space Hopf oscillators. The latter's corresponding equations for each leg i are given in equations 2.1 and 2.2 (amplitude and phase respectively, in polar form, based on [1]):

$$\dot{r_i} = \alpha(\mu - r_i^2)r_i \tag{2.1}$$

$$\dot{\theta_i} = \omega_i + \sum_{j=0}^{3} r_j w_{ij} sin(\theta_j - \theta_i - \phi_{ij}) \tag{2.2}$$

Where

- r_i is the current amplitude, $\sqrt{\mu}$ the desired amplitude of the oscillator and α a positive constant that controls the speed of convergence to the limit cycle.
- θ_i is the phase of the oscillator, ω_i the natural frequency of oscillations (gait dependent), w_{ij} the coupling strength between oscillators i and j, and ϕ_{ij} is the desired phase offset between oscillators i and j.

This simple network allows to extract valuable information such as the amplitude and the phase for each of the oscillators, which are fully coupled, as shown in equation 2.2. The obtained CPG states can then be mapped back to Cartesian space foot coordinates through Inverse Kinematics to obtain the desired outputs x_{foot} and z_{foot} (equs. 2.3 and 2.4), and control them accordingly. In the absence of coupling (i.e considering only one leg), these outputs can be visually represented as shown in figure 2.1.

$$x_{foot} = -d_{step} \cdot r_i cos(\theta_i) \tag{2.3}$$

$$z_{foot} = \begin{cases} -h + g_c sin(\theta_i) & \text{if } sin(\theta_i) > 0 \text{ (swing phase)} \\ -h + g_p sin(\theta_i) & \text{otherwise (stance phase)} \end{cases} \tag{2.4}$$

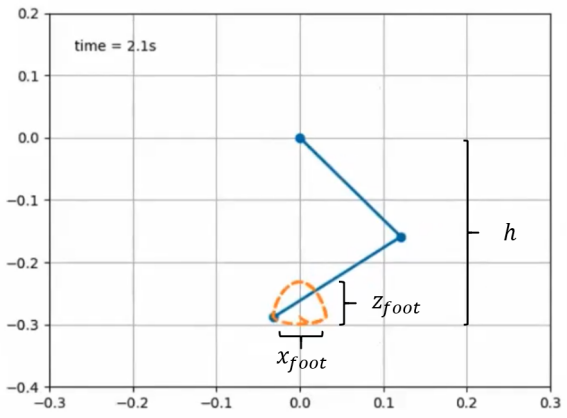


Figure 2.1: CPG States to Foot Position mapping

It is worth mentioning that the natural frequency of the oscillators ω is split into ω_{swing} amd ω_{stance}. The swing phase is defined as when the foot is in the air, hence when the $0 \leq \theta_i \leq \pi$, and the stance phase conversely defined as when $\pi \leq \theta_i \leq 2\pi$ (see equation 2.4). The max ground clearance is given by g_c and the max ground penetration as g_p. The latter is naturally surface dependant and is introduced in order to assist in potential stability related issues. In order to assess and qualitatively compare the different gait, key terms such as the stride duration (the period of one complete cycle) and Duty cycle $D = T_{stance}/T_{swing}$ are introduced.

2.2 Methods

The control method was implemented on the `hopf_network.py` module. This file contains the class *HopfNetwork* which starts out by defining all the parameters needed for the methods in the class and initialising the oscillator states. In this class there are two main methods: *_set_gait* which defines the coupling matrix for the chosen gait, and *_integrate_hopf_equations* which calculates the current amplitude of the oscillator and the phase from $\dot{r}$ and $\dot{\theta}$ using the equations 2.1 and 2.2. Another noteworthy method is *update* that takes new CPG variables and converts them to xz foot positions using the equations 2.3 and 2.4. As for the execution of the code, when the *hopf_network.py* module is run, the necessary classes and parameters are initialised and, for each step, the desired foot positions from the CPG are calculated. The current motor torques and velocities are obtained, and the control action for each leg is calculated using the PD joint controller equation, and similarly for the Cartesian PD controller. The resulting implementation in the code is shown in listing 2.1 below. The condition for the switching of the natural frequencies ω_i is shown in lines 40-43, and the final obtained $\dot{\theta}$ in line 51.

Listing 2.1: Implementation of Methods in HopfNetwork class

```
1       def update(self):
2       """ Update oscillator states. """
3
4       # update parameters, integrate
5       self._integrate_hopf_equations()
6
7       # map CPG variables to Cartesian foot xz positions (Equations 8, 9)
8       r = self.X[0,:]
9       theta = self.X[1,:] % (2*np.pi)
10      x = -self._des_step_len * r * np.cos(theta) # [TODO]
11      z = np.zeros(x.shape)
12      i = 0
13      for theta_i in theta:
14        if np.sin(theta_i) > 0:
15          z[i] = -self._robot_height+self._ground_clearance*np.sin(theta_i)
16        else:
17          z[i] = -self._robot_height+self._ground_penetration*np.sin(theta_i)
18        i+=1
19      # [TODO]
20
21      return x, z
22
23
24    def _integrate_hopf_equations(self):
25      """ Hopf polar equations and integration. Use equations 6 and 7. """
26      # bookkeeping - save copies of current CPG states
27      X = self.X.copy()
28      X_dot = np.zeros((2,4))
29      alpha = 50
30
31      # loop through each leg's oscillator
32      for i in range(4):
33        # get r_i, theta_i from X
```

```
34          r = X[0,i] # [TODO]
35          theta  = X[1,i] % (2*np.pi)
36          # compute r_dot (Equation 6)
37          r_dot = alpha*(self._mu-r**2)*r # [TODO]
38          # determine whether oscillator i is in swing or stance phase to set natural ...
                frequency omega_swing or omega_stance (see Section 3)
39
40          if (0 <= theta <= np.pi):
41            theta_dot = self._omega_swing # [TODO]
42          elif (np.pi < theta <2*np.pi):
43            theta_dot = self._omega_stance
44
45          # loop through other oscillators to add coupling (Equation 7)
46          if self._couple:
47            for j in range(4):
48              if j!=i:
49                r_j = X[0,j]
50                theta_j = X[1,j] % (2*np.pi)
51                theta_dot += r_j * self._coupling_strength  * np.sin(theta_j - theta - ...
                      self.PHI[i,j] ) # [TODO]
52
53          # set X_dot[:,i]
54          X_dot[:,i] = [r_dot, theta_dot]
55
56        # integrate
57        self.X = X + X_dot*self._dt # [TODO]
58        # mod phase variables to keep between 0 and 2pi
59        self.X[1,:] = self.X[1,:] % (2*np.pi)
60        self.X_dot = X_dot
```

2.2.1 Gait Implementation

The gaits implemented in this project were: trot, pace, walk and bound, which yield the following coupling matrices ϕ_{gait}, all being symmetric gaits except for the asymmetric *bound* gait [**?**].

$$\phi_{walk} = \begin{bmatrix} 0 & -\pi & -\frac{\pi}{2} & \frac{\pi}{2} \\ \pi & 0 & \frac{\pi}{2} & -\frac{\pi}{2} \\ \frac{\pi}{2} & -\frac{\pi}{2} & 0 & \pi \\ -\frac{\pi}{2} & \frac{\pi}{2} & -\pi & 0 \end{bmatrix} \qquad \phi_{trot} = \begin{bmatrix} 0 & -\pi & -\pi & 0 \\ \pi & 0 & 0 & \pi \\ \pi & 0 & 0 & \pi \\ 0 & -\pi & -\pi & 0 \end{bmatrix} \qquad \phi_{pace} = \begin{bmatrix} 0 & \pi & 0 & \pi \\ -\pi & 0 & -\pi & 0 \\ 0 & \pi & 0 & \pi \\ -\pi & 0 & -\pi & 0 \end{bmatrix}$$

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

The coupling matrices were obtained taking into consideration the footfall sequences of each gait in one cycle, figure 2.2, and that the entry ij of the matrix corresponds to the phase difference between leg_i and leg_j. The matrices are skew-symmetric as the phase difference between leg_i and leg_j will be the symmetric of the phase difference between leg_j and leg_i.

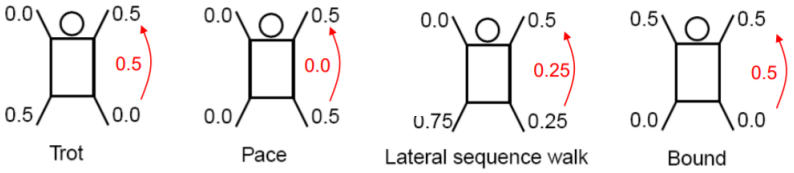


Figure 2.2: Footfall sequences for each of the gaits

The first three gaits represented in figure 2.2 are symmetric, meaning that their respective footfalls of a pair of limbs are equally spaced in time; the bound gait is conversely an asymmetric gait. The tuning of the coupling matrices shown above was done by deducing the required phase offset for each pair of limbs to achieve the corresponding gaits. This could be verified by activating the on_rack in the QuadrupedGymEnv constructor (see fig. 2.3). Indeed, this allowed to simulate the produced gaits to ensure coherent footfall patterns prior to testing it in the standard environment.

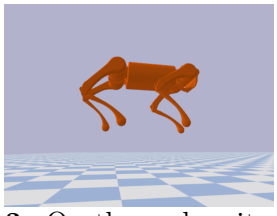


Figure 2.3: On the rack gait monitoring

It is worth noting that despite satisfactory gait patterns on the rack, the resulting implementation did not always yield satisfactory results due to the stringent influence of the parameters on the different physical parameters of the quadruped system. This namely implied that fine tuning of appropriate PD gains as well as stance and swing natural frequencies (amongst others) was required. Another way to evaluate whether the gaits were coherent would be to further examine the phase states and if a corresponding limit cycle is obtained. For completeness, an example of such a graph for the *trot* gait is shown in fig. 2.4 below.

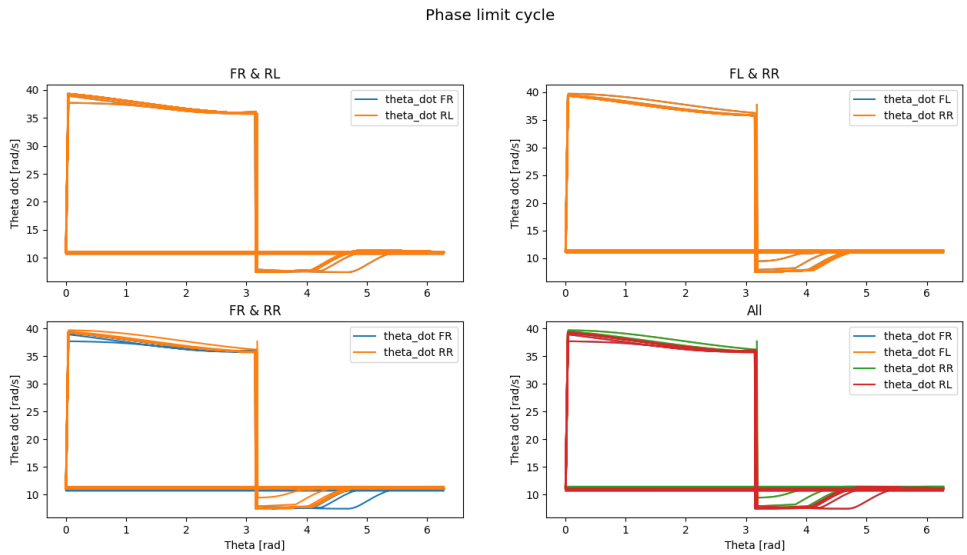


Figure 2.4: Phase Limit Cycle for trot gait (adapted from [2])

As was to be expected, the diagonal legs (FR with RL and FL with RR - see top left and right graphs) are perfectly overlapping, implying that the corresponding footfall sequences shown in figure 2.2 are indeed

respected (i.e both pair of limbs touch the ground at the same time). The cyclic nature of this plot is due to the repetitive switching of the stance and swing phases (see equation 2.2) .

Due to the chaotic response from the lack of the Cartesian PD controller on the quadruped , it was decided to evaluate the reponse of the foot positions and joint angles on the rack directly (in both the absence/presence of the Cartesian PD). Indeed, after numerous attempts, successful gait implementations with solely the joint PD controller being used amounted to no avail.

2.2.2 Parameter Tuning Methodology

When running the simulations, it was found most useful to tune ω_{stance} and ω_{swing} and the relationship between them for each gait. Indeed, they are key parameters behind determining the associated duty cycles/ratios, which is here defined as a measure of the amount of time the quadruped is in stance phase as opposed to swing phase $D = T_{stance}/T_{swing}$. Hence, a duty cycle larger than 1 implies that more time is spent in contact with the ground, and consequently result in more balanced performances. This was a desired outcome when first implementing the gaits, as they all proved to be very unstable upon first executions. Interestingly, this is not a natural phenomenon when considering for typical fast paced gaits in quadruped vertebrates. As one would expect, typical running gaits (such as fast trot, pace and bound) would imply lower swinging frequencies, and thus a duty cycle inferior to 1, for efficient forward locomotion.

The parameters shown in table 2.1 were kept constant for all gaits.

Table 2.1: Hopf oscillator and simulation parameters

α	50
μ	1
coupling strength	1
time step	0.001
robot height	0.25

2.3 Results

2.3.1 Trot Gait

The natural frequencies used for the slowest trot gait were: $\omega_{stance} = 3\pi$ and $\omega_{swing} = 12\pi$, resulting in a Duty cycle of $D = \frac{\omega_{swing}}{\omega_{stance}} = 4$. The chosen natural frequencies result in a stance period of 0.67 seconds and swing period of 0.17 seconds, making the total time for a leg to take a step equal to 0.87 seconds. For this gait the *ground clearance* parameter was changed from 0.05 (default value) to 0.03, as well as the *desired step length* which as changed from 0.04 to 0.03. These changes, though small, aimed to increase the quadruped stability by making it take shorter but faster steps. The *ground penetration* was kept at default value as 0.01.

The Joint and Cartesian PD controllers gains used are shown on the table 2.2

Table 2.2: Joint and Cartesian PD controllers gains

	q1	**q2**	**q3**
Kp	150	70	70
Kd	5	1	1
	X foot	**Y foot**	**Z foot**
Kp,c	2500	2500	2500
Kd,c	40	40	40

The natural frequencies used for the highest speed trot gait were: $\omega_{stance} = 5\pi$ and $\omega_{swing} = 15\pi$, resulting in a Duty cycle of $D = \frac{\omega_{swing}}{\omega_{stance}} = 3$. The natural frequencies chosen result in a stance period of 0.4 seconds

and swing period of 0.13 seconds, making the total time for a leg to take a step equal to 0.53 seconds. For this gait the *ground clearance* parameter was kept as 0.05 (default value), the *desired step length* was changed from 0.04 to 0.03, and the *ground penetration* was kept at default value as 0.01. These changes, though small, aimed to increase the quadruped stability by making the quadruped take shorter but faster steps.

The Joint and Cartesian PD controllers gains used are shown on the table 2.3

Table 2.3: Joint and Cartesian PD controllers gains

	q1	**q2**	**q3**
Kp	150	70	70
Kd	5	1	1
	X foot	**Y foot**	**Z foot**
Kp,c	2500	2500	2500
Kd,c	100	100	100

The CPG states for each leg during motion, comparison of the front right foot's positions with the desired positions with and without Cartesian PD control, and comparison of the front right leg's joint angles with the desired joint angles with and without Cartesian PD control can be seen below in figures 2.5, 2.6, 2.7 (these were taken from the slower trot gait).

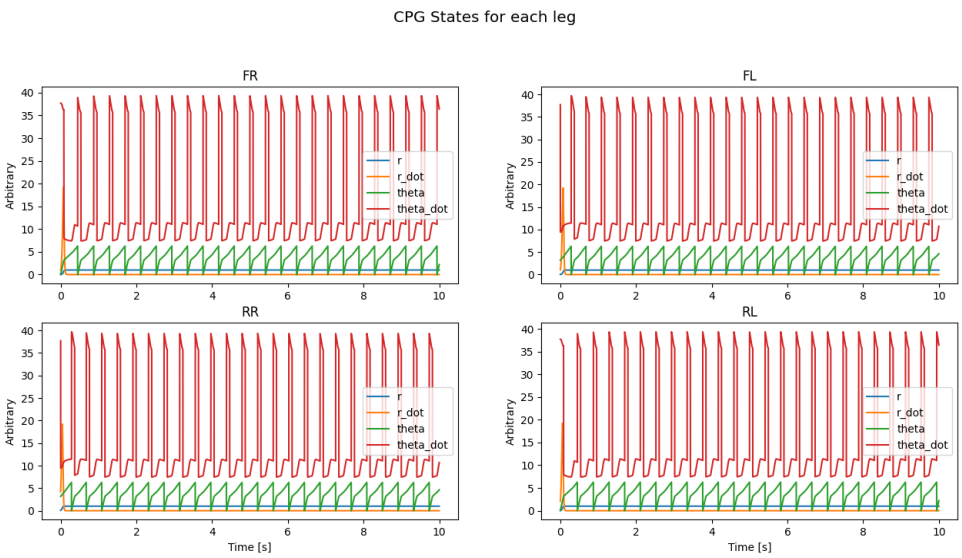
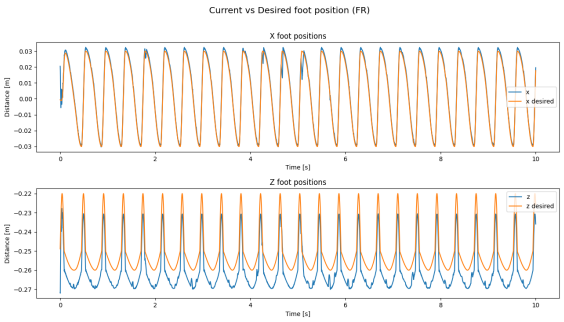
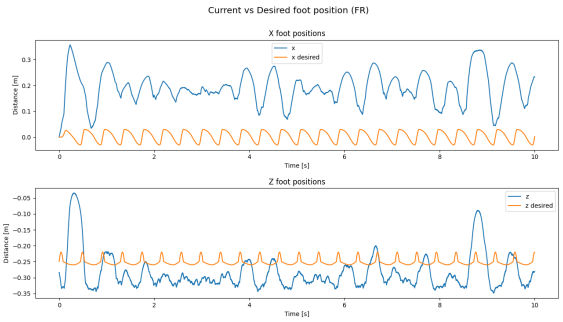


Figure 2.5: CPG states for Trot gait

Note that, analysing the figure 2.5, the plots for the front left leg and the rear right leg are identical, and the same applies to the plots for the front right leg and rear left leg. Moreover, these two couples of plots are out of phase with each other by half a cycle (π), this is what was expected for the trot gait.

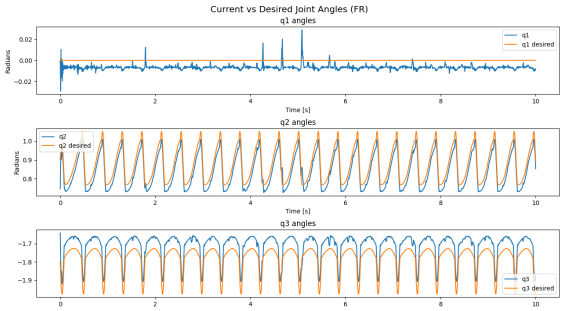


(a) Foot positions vs Desired foot positions with contribution of the Cartesian PD controller

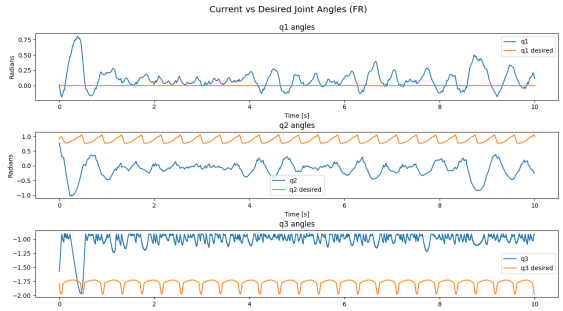


(b) Foot positions vs Desired foot positions without contribution of the Cartesian PD controller

Figure 2.6: Foot positions vs Desired foot positions



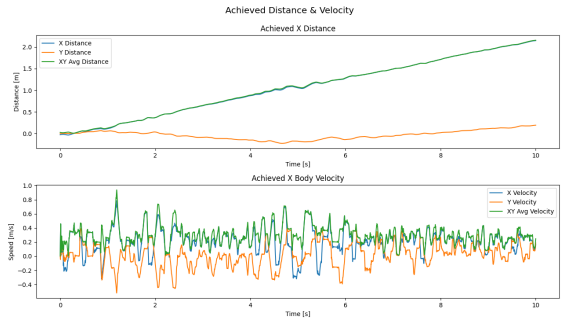
(a) Joint angles vs Desired joint angles with contribution of the Cartesian PD controller



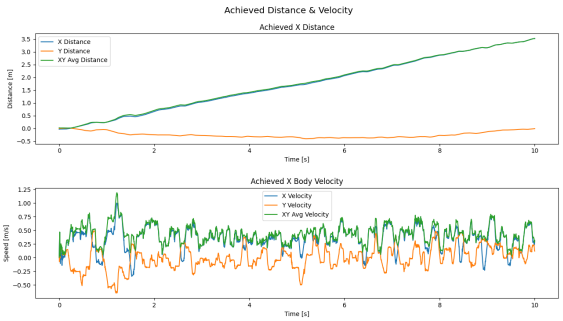
(b) Joint angles vs Desired joint angles without contribution of the Cartesian PD controller

Figure 2.7: Joint angles vs Desired joint angles

The fastest trot gait achieved and linear velocity of 0.355m/s, and the corresponding Cost of Transport (CoT) of 1.932. The slowest trot gait achieved and linear velocity of 0.218 m/s, and the corresponding Cost of Transport (CoT) of 0.959. The distance travelled by the quadruped and its velocity for both speeds are shown on figure 2.8.



(a) Distance travelled and velocity of the quadruped for the lowest speed



(b) Distance travelled and velocity of the quadruped for the highest speed

Figure 2.8: Distance travelled and velocity of the quadruped

2.3.2 Pace Gait

The natural frequencies used for the pace gait were: $\omega_{stance} = 4\pi$ and $\omega_{swing} = 12\pi$, resulting in a Duty cycle of $D = \frac{\omega_{swing}}{\omega_{stance}} = 3$. The natural frequencies chosen result in a stance period of 0.5 seconds and swing period of 0.17 seconds, making the total time for a leg to take a step equal to 0.67 seconds. For this gait the

default parameters for *ground clearance*, *ground penetration*, and desired step length were kept at default value as 0.05, 0.01 and 0.04 respectively. The controller gains used were the same as for the slow trot gait.

The CPG states for each leg during motion, comparison of the front right foot's positions with the desired positions with and without Cartesian PD control, and comparison of the front right leg's joint angles with the desired joint angles with and without Cartesian PD control can be seen below in figures 2.9, 2.10, 2.11.

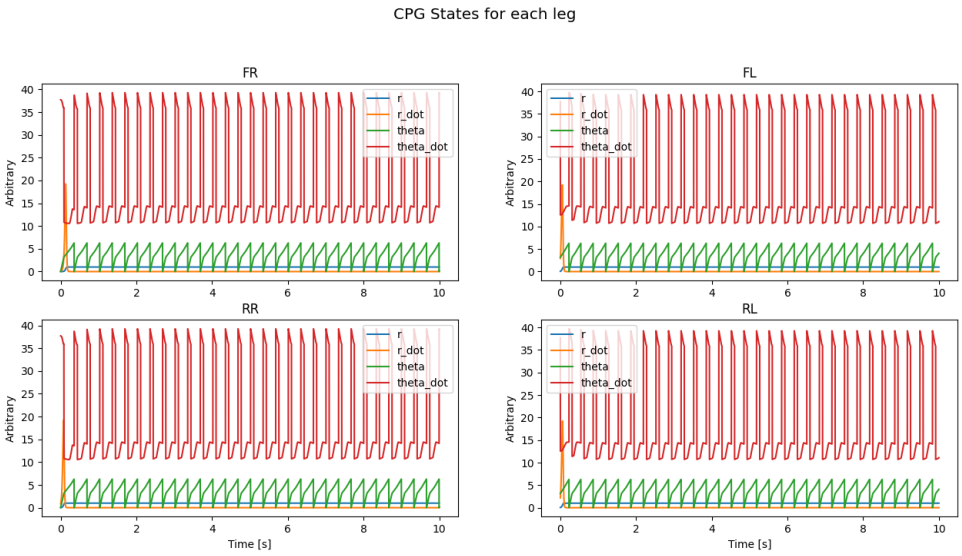
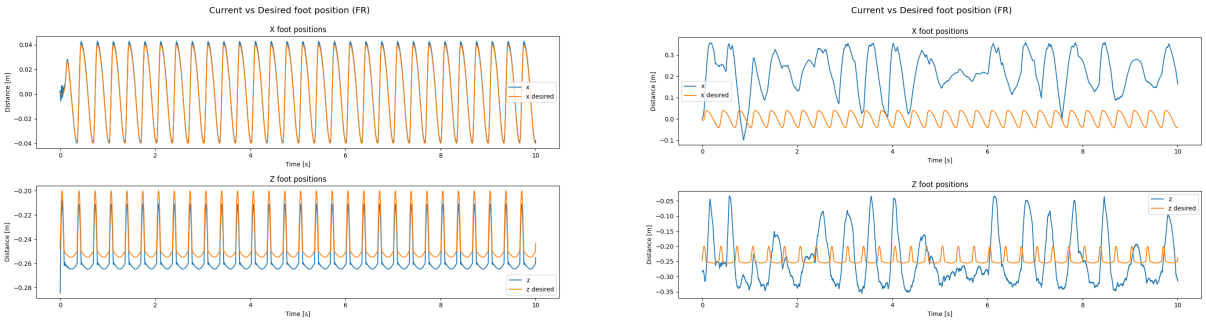


Figure 2.9: CPG states for Pace gait

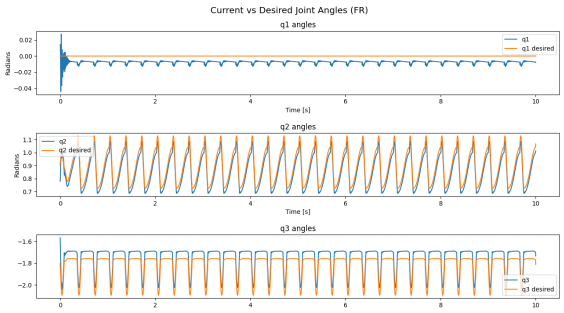
Note that, analysing the figure 2.9, the plots for the front left leg and the rear left leg are identical, and the same applies to the plots for the front right leg and rear right leg. Additionally, these two couples of plots are out of phase with each other by half a cycle (π), this is what was expected for the pace gait.



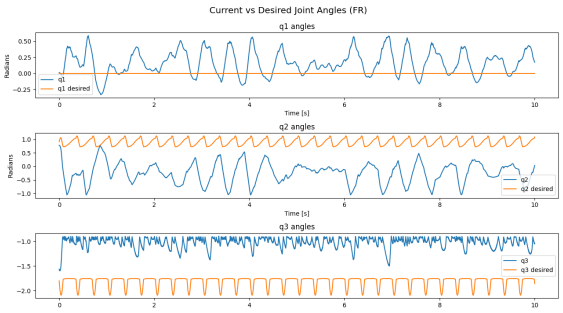
(a) Foot positions vs Desired foot positions with contribution of the Cartesian PD controller

(b) Foot positions vs Desired foot positions without contribution of the Cartesian PD controller

Figure 2.10: Foot positions vs Desired foot positions



(a) Joint angles vs Desired joint angles with contribution of the Cartesian PD controller



(b) Joint angles vs Desired joint angles without contribution of the Cartesian PD controller

Figure 2.11: Joint angles vs Desired joint angles

After intensive tuning it was still not possible to achieve a stable pace gate on the ground, therefore no CoT value is presented for this gait.

2.3.3 Lateral Sequence Walk Gait

The natural frequencies used for the fastest walk gait were: $\omega_{stance} = 3\pi$ and $\omega_{swing} = 12\pi$, resulting in a Duty cycle of $D = \frac{\omega_{swing}}{\omega_{stance}} = 4$. The natural frequencies chosen result in a stance period of 0.67 seconds and swing period of 0.17 seconds, making the total time for a leg to take a step equal to 0.87 seconds. For this gait the *ground clearance* parameter was changed from 0.05 (default value) to 0.03, as well as the *desired step length* which as changed from 0.04 to 0.03. These changes, though small, aimed to increase the quadruped stability by making it take shorter but faster steps. The *ground penetration* was kept at default value as 0.01.

The Joint PD controller gains used are shown on the table 2.4 (Cartesian PD gains remained the same as the pace gait).

Table 2.4: Joint PD controller gains

	q1	**q2**	**q3**
Kp	150	72	72
Kd	2	1	1

The natural frequencies used for the slowest walk gait were: $\omega_{stance} = \pi$ and $\omega_{swing} = 4\pi$, resulting in a Duty cycle of $D = \frac{\omega_{swing}}{\omega_{stance}} = 4$. The natural frequencies chosen result in a stance period of 2 seconds and swing period of 0.5 seconds, making the total time for a leg to take a step equal to 2.5 seconds. All other parameters (ground clearance, ground penetration and desired step length) remained the same.

The Joint PD controller and Cartesian PD controller remained basically the same, only the proportional gain for q_1 and q_2 was changed from 72 to 70.

The CPG states for each leg during motion, comparison of the front right foot's positions with the desired positions with and without Cartesian PD control, and comparison of the front right leg's joint angles with the desired joint angles with and without Cartesian PD control can be seen below in figures 2.12, 2.13, 2.14 (these were taken from the fastest walk gait)

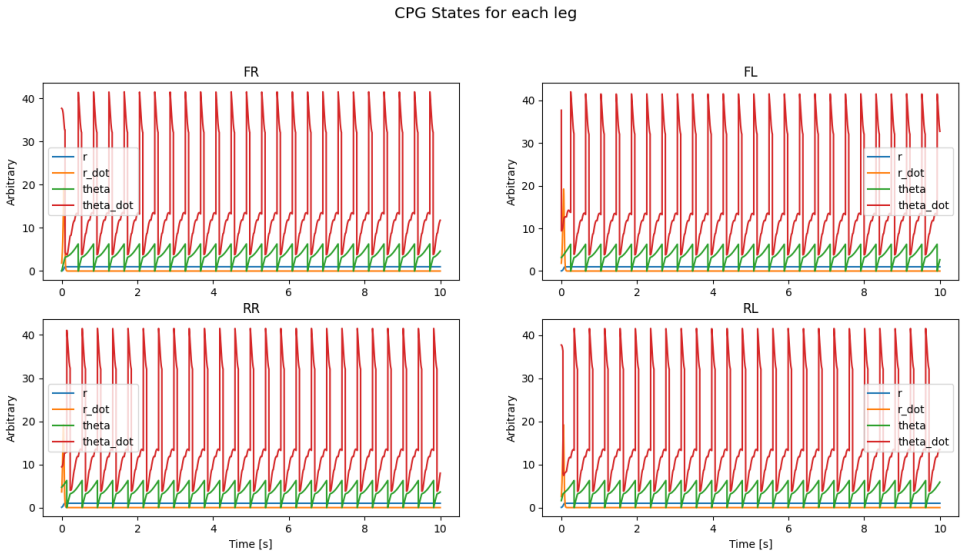
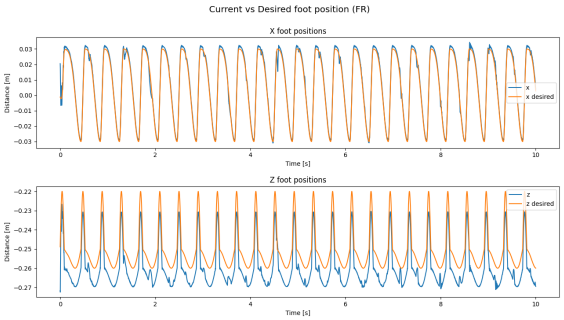
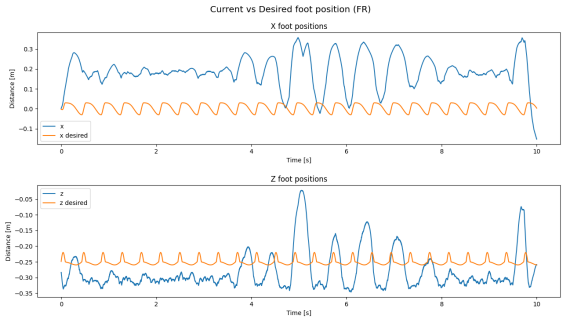


Figure 2.12: CPG states for Walk gait

Analysing the figure 2.12 more closely, note that the θ and $\dot{\theta}$ plots for the front right leg are a quarter of a cycle out of phase with the rear left leg plots, which in turn are a quarter of a cycle out of phase with front left leg's plot, which, in turn, shows the same phase difference when compared with rear right leg. These plots were indeed what was expected for the walking gait.

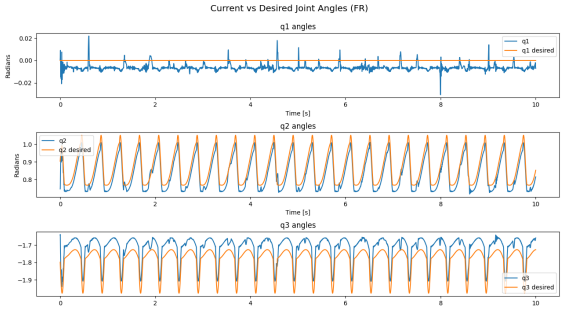


(a) Foot positions vs Desired foot positions with contribution of the Cartesian PD controller

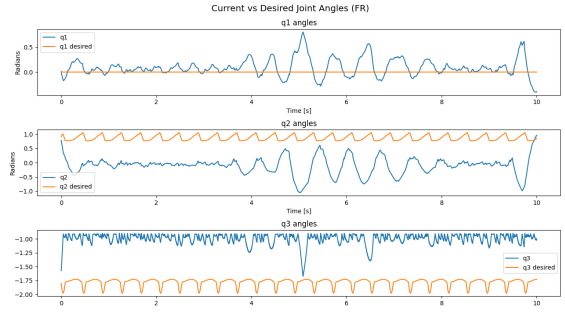


(b) Foot positions vs Desired foot positions without contribution of the Cartesian PD controller

Figure 2.13: Foot positions vs Desired foot positions



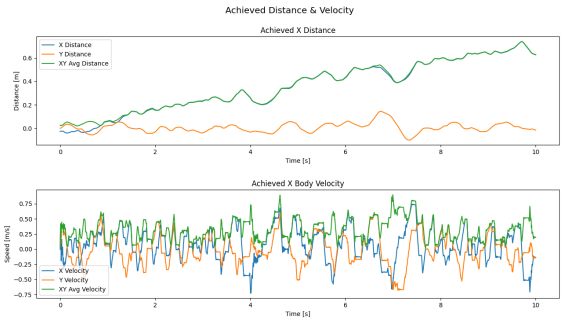
(a) Joint angles vs Desired joint angles with contribution of the Cartesian PD controller



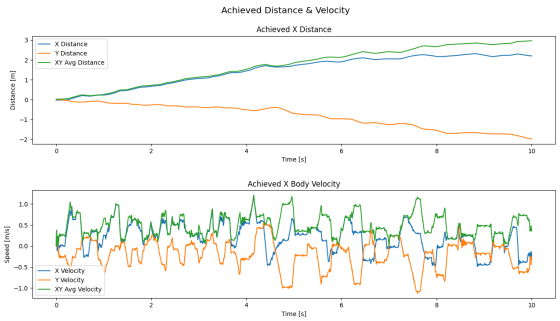
(b) Joint angles vs Desired joint angles without contribution of the Cartesian PD controller

Figure 2.14: Joint angles vs Desired joint angles

The fastest walk gait achieved and linear velocity of 0.298 m/s, and the corresponding Cost of Transport (CoT) of 1.937. The slowest walk gait achieved and linear velocity of 0.065 m/s, and the corresponding Cost of Transport (CoT) of 3.183. The distance travelled by the quadruped and its velocity for both speeds are shown on figure 2.15.



(a) Distance travelled and velocity of the quadruped for the lowest speed



(b) Distance travelled and velocity of the quadruped for the highest speed

Figure 2.15: Distance travelled and velocity of the quadruped

2.3.4 Bound Gait

The natural frequencies used for the walk gait were: $\omega_{stance} = 5\pi$ and $\omega_{swing} = 15\pi$, resulting in a Duty cycle of $D = \frac{\omega_{swing}}{\omega_{stance}} = 3$. The natural frequencies chosen result in a stance period of 0.4 seconds and swing period of 0.13 seconds, making the total time for a leg to take a step equal to 0.53 seconds. For this gait the *ground clearance* parameter was changed from 0.05 (default value) to 0.03, as well as the *desired step length* which as changed from 0.04 to 0.03. These changes, though small, aimed to increase the quadruped stability by making it take shorter but faster steps. The *ground penetration* was changed from 0.01 to 0.005 so that the feet wouldn't sink into the ground as much to help the robot easily change between the swinging the front or hind legs. the controller gains used were the same as for the trot.

The CPG states for each leg during motion, comparison of the front right foot's positions with the desired positions with and without Cartesian PD control, and comparison of the front right leg's joint angles with the desired joint angles with and without Cartesian PD control can be seen below in figures 2.16, 2.17, 2.18.

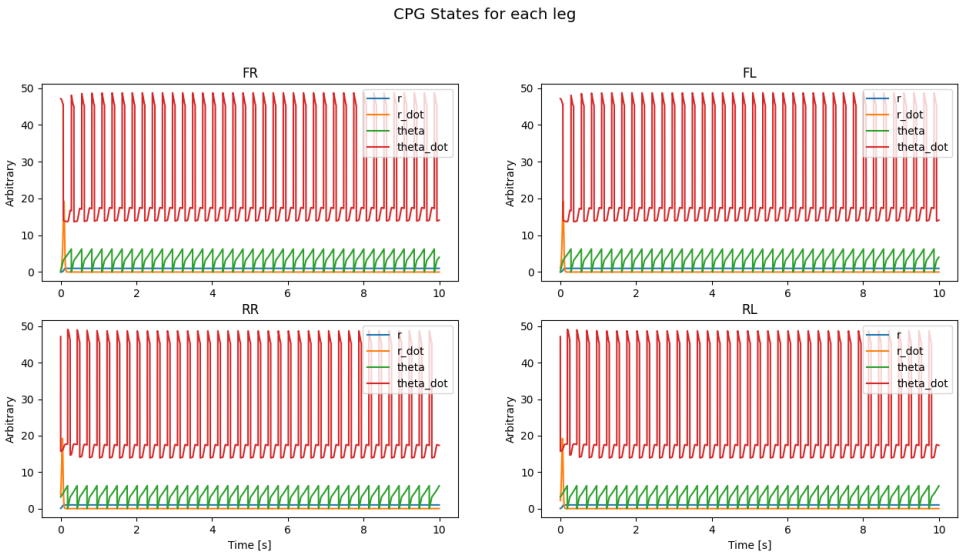
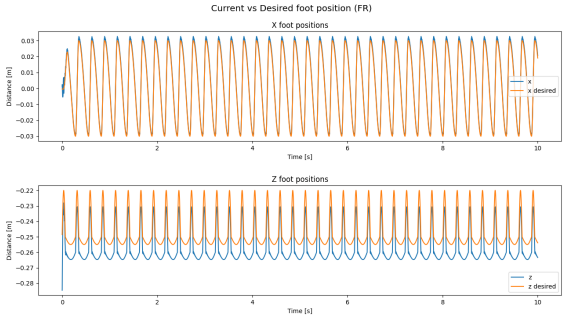
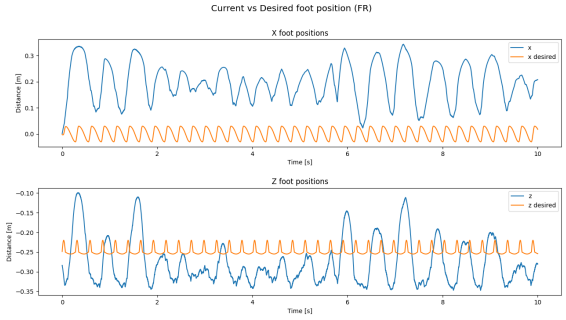


Figure 2.16: CPG states for Bound gait

Note that, analysing the figure 2.16, the plots for the front legs are identical, and the same applies to the plots for the rear legs. Moreover, these two couples of plots are out of phase with each other by half a cycle (π), this is what was expected for the bound gait.

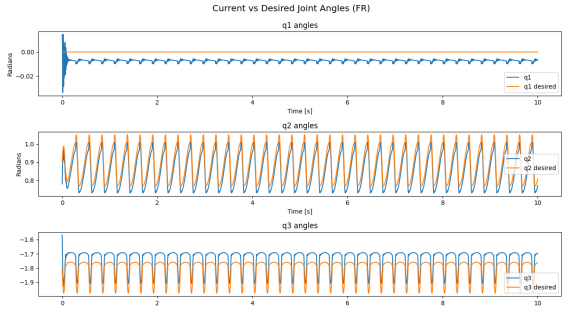


(a) Foot positions vs Desired foot positions with contribution of the Cartesian PD controller

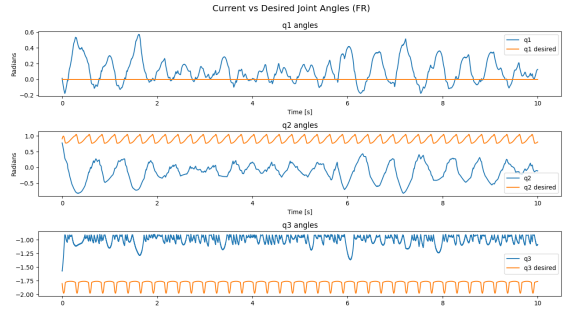


(b) Foot positions vs Desired foot positions without contribution of the Cartesian PD controller

Figure 2.17: Foot positions vs Desired foot positions



(a) Joint angles vs Desired joint angles with contribution of the Cartesian PD controller



(b) Joint angles vs Desired joint angles without contribution of the Cartesian PD controller

Figure 2.18: Joint angles vs Desired joint angles

After intensive tuning it was still not possible to achieve a stable pace gate on the ground, therefore there is no CoT value presented for this gait.

2.3.5 Summary

The table 2.5 below shows a summary of the obtained Cost of Transport as well as their corresponding velocities. As previously mentioned, the other *bound* and *pace* gaits (namely typical running gaits) did not manage successful episode runs in the test environment, and therefore their performance is not presented here, despite having achieved coherent footfall patterns on the rack.

Table 2.5: Recap of obtained performances per successful gait

	Trot		**Walk**	
	COT [-]	Velocity [m/s]	COT [-]	Velocity [m/s]
Fast	1.932	0.355	1.937	0.298
Slow	0.959	0.218	3.183	0.065

It is reminded that all of the above achieved gaits yielded a Duty cycle $D = 4$ except for the fast trot which instead had $D = 3$ at rather high frequencies ($w_{stance} = 5\pi and w_{swing} = 15\pi$).

2.4 Discussion

Taking into consideration the results presented above it is possible to conclude that the most important parameters to tune for each gait were the stance and swing natural frequencies, though small changes in ground clearance, ground penetration and desired step length also proved useful for added stability. The PD joint gains were also increased from the default values of $[2, 0.5, 0.5]$ with the intent of minimizing joint oscillations during the beginning of motion. Regarding the CPG states for each gait, the plots showcase multiple spikes in the $\dot{\theta}$ which is due to the change between ω_{stance} and ω_{swing}. The state r stabilized at 1 for all gaits.

Regarding the comparison of the tracking performance when the Cartesian PD controller is and is not added, it is clear to see that the added contribution of this controller greatly improves the quadruped's response to the reference angles and foot positions.

It can be ascertained that the obtained results all yielded duty cycles above 1, which in this context would imply prolonged duration in the stance phase with regards to the swing phase. This indeed makes sense for the slow gaits, such as lateral sequence walking, as it always bears the property of having 3 limbs constantly touching the ground. However, starting from medium speed gaits, such as the trot, this may come as a surprise. Generally, quadruped vertebrates would have a stance phase which is lower than the swing phase in the overall stride duration [3] and thus would result in a duty cycle less than 1. As was shown in the results, only the gaits which had a duty cycle *above* 1 were successfully implemented, which could explain the relatively low speeds that were obtained. Surprisingly, the most efficient gait seemed to be the *slow trot* as it has a lower CoT than for the lateral sequence walk, although not by a vast amount. Regarding the fast-paced gaits such as the *pace* and *bound* gaits, one would have expected successful implementation for swinging frequencies lower than stance frequencies, although this was not achieved despite numerous efforts. Considerations for future work to improve these performances could be to have a better attitude control, as they often seemed to deviate from a "straight" trajectory. Stumbling correction as well as improved heading control could further increase the prospect of meeting the desired performances.

3. Deep Reinforcement Learning for Quadruped Locomotion

3.1 Theoretical Insights

It is desired here to design a Markov Decision Process (MDP) for the quadruped locomotion task as well as train the control policies with two different reinforcement learning algorithms, namely Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC). The well-known $stable-baselines3$ library shall be used to implement these algorithms.

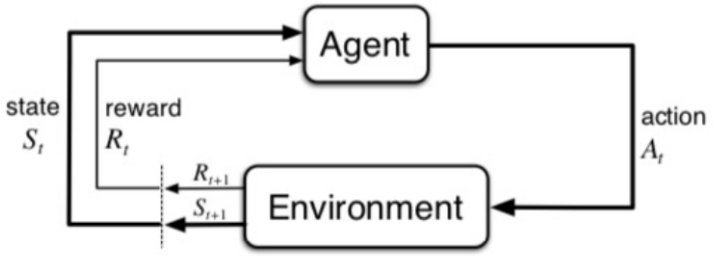


Figure 3.1: Reinforcement Learning- Markov Decision Process

Generally speaking, Reinforcement Learning aims to find the best sequence of actions that an Agent can take on an Environment to determine the most optimal outcome, i.e collecting the most reward. For terminology, the Environment takes a commanded task or simulation (action) that is the output of the Agent, which refers to the entity making the decisions based on the received rewards and states (i.e the policy which is constantly updated with the RL algorithm). Based on the action, the Environment then produces a reward for that particular action. States are here defined as body observations, for example the body height and velocity, angular state (roll, pitch and yaw) or potential joint states (obtained observational measurement from sensors); the reward function aims to "shape" future actions, and may encourage forward locomotion and/or penalise energy expenditure. The action space may be comprised of the motor positions/torques and affect the joints directly (operating in the joint space with Joint PD control), or instead make the policy choose a desired end effector position in Cartesian space with Cartesian PD control.

3.2 Methods

Regarding the code structure for implementation of the RL algorithms, the OpenAI Gym environment is used to provide an interface to the MDP which is defined in the file quadruped_gym_env.py. This file also generates the instance of the quadruped robot which is used for simulation and training by the RL algorithms through the run_sb3.py file or to visualise the results through the load_sb3.py file.

A step-by-step approach was taken in designing the MDP for this project to minimise the time required for testing, which was significant as each simulation required several hours to yield results. Each stage of the MDP was tuned and finalised before moving onto the next stage and this was done in the order of: observation space, action space, reward function. This section will discuss further the methods and reasoning behind selecting the final MDP for Deep Reinforcement Learning.

3.2.1 Observation Space

The observation space is the set of values which reflect the effects that the agent has on the environment, or in this case the effect that the commanded motor actions have on the robot state. These observations can include a number of measurements such as body position, joint angles and feet positions but these values should be observable on a real robot by its on-board sensors. For this project, the robot state

measurements which were considered relevant and observable are the body height, orientation quaternion, linear and angular velocities, joint position and velocities, feet position and velocities, and foot in-contact booleans. These values were chosen as they are directly effected by the robot's locomotion and can be easily gathered through common on-board sensors such as an inertial measurement unit and feet pressure sensors, or they can be discerned from inverse kinematics. Given that the quadruped robot has four legs and three joints per leg, the final observation space contains sixty three dimensions. It should also be noted that these values are normalised for training purposes as each measurement will be of a different scale and thus affect the algorithm disproportionately unless they are normalised first.

The observation space was selected purely from a theoretical standpoint and practically the observation space could be made more compact to reduce the effect of noisy measurements. From [7] the benefit of doing this would make the robot more robust and it has been shown in [6] that a more compact observation space can yield similar results to what is shown here. However, since this project is conducted entirely through simulation, it is difficult to discern which hardware measurements are likely to be more noisy than others and should thus be left out. Additionally, to allow more time for testing the action space and reward function, it was decided not to tune the observation space further and leave it in a state similar to [5] which bears great similarity to this project.

3.2.2 Action Space

The action space maps the policy network output to physical actions for the robot. One of the goals of this project was to develop the action space in Cartesian space, learning desired foot positions, and compare the output with a joint position action space, learning desired joint angles. It was expected that the Cartesian space control be more accurate and produce smoother locomotion as it provides an exact mapping of the feet and legs to the environment. The results of both action spaces were compared, with all other parameters being equal, and the better performing type of control was carried forward to the next stage.

To implement the Cartesian PD controller using the Cartesian action space, recall equation 1.5 which calculates the required motor torques given desired foot positions and current foot positions, velocities and leg Jacobians:

$$\tau_{Cartesian} = J^T(q)[K_{p,Cartesian}(p_d - p) + K_{d,Cartesian}(v_d - v)] \tag{3.1}$$

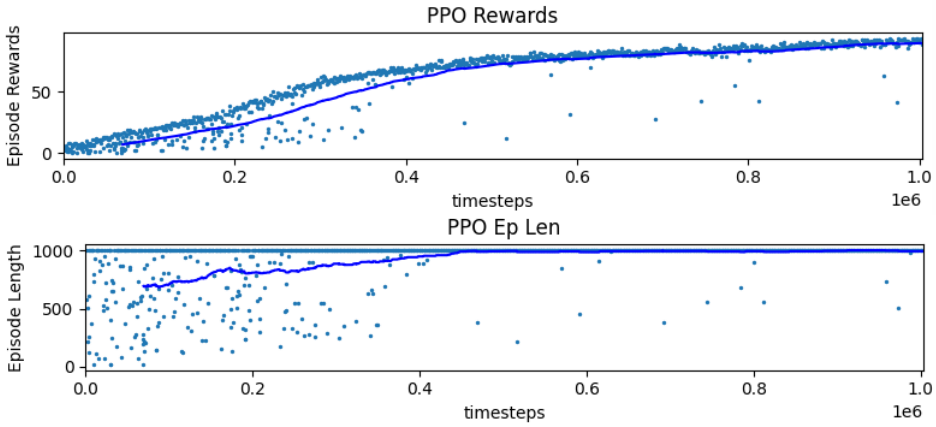
This is implemented in the quadruped_gym_env.py file as shown in Listing 3.1. Note that the scale factor for the foot positions in line 8 has been tuned and this will be further explained later.

Listing 3.1: Implementation of the Torque equation in code

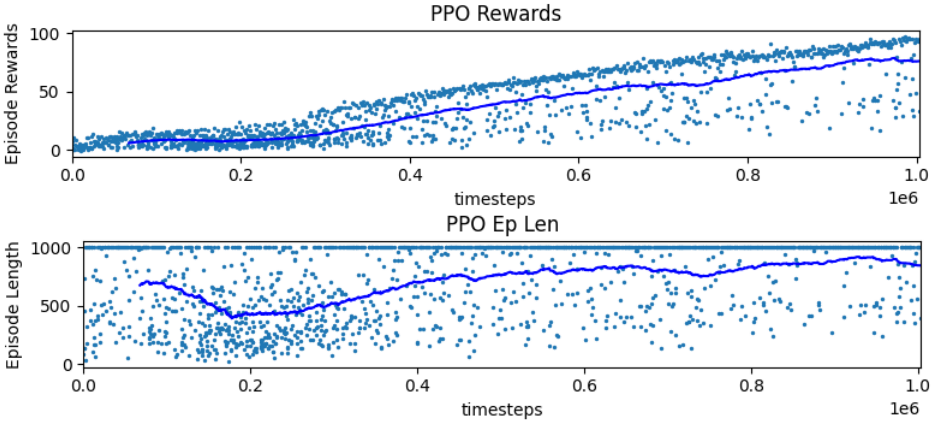
```
1   def ScaleActionToCartesianPos(self,actions):
2     """Scale RL action to Cartesian PD ranges.
3     Edit ranges, limits etc., but make sure to use Cartesian PD to compute the torques.
4     """
5     # clip RL actions to be between -1 and 1 (standard RL technique)
6     u = np.clip(actions,-1,1)
7     # scale to corresponding desired foot positions (i.e. ranges in x,y,z we allow ...
          the agent to choose foot positions)
8     scale_array = np.array([0.15, 0.05, 0.12]*4)
9     # add to nominal foot position in leg frame (what are the final ranges?)
10    des_foot_pos = self._robot_config.NOMINAL_FOOT_POS_LEG_FRAME + scale_array*u
11
12    # get Cartesian kp and kd gains (can be modified)
13    kpCartesian = self._robot_config.kpCartesian
14    kdCartesian = self._robot_config.kdCartesian
15    # get current motor velocities
16    qd = self.robot.GetMotorVelocities()
17
18    action = np.zeros(12)
```

```
19      for i in range(4):
20        # get Jacobian and foot position in leg frame for leg i (see ...
              ComputeJacobianAndPosition() in quadruped.py)
21        J, p = self.robot.ComputeJacobianAndPosition(i)
22        # desired foot position i (from RL above)
23        Pd = des_foot_pos[3*i:3*i+3]
24        # desired foot velocity i
25        vd = np.zeros(3)
26        # foot velocity in leg frame i (Equation 2)
27        v = np.dot(J, qd[3*i:3*i+3])
28        # calculate torques with Cartesian PD (Equation 5) [Make sure you are using ...
              matrix multiplications]
29        tau = np.matmul(np.transpose(J), (np.matmul(kpCartesian,(Pd-p)) - ...
              np.matmul(kdCartesian,v)))
30
31        action[3*i:3*i+3] = tau
32
33      return action
```

Now with two types of control in two different action spaces, it is possible to compare the performance of each based on their training curves and simulation performance. In obtaining the figures below, the observation space is the same as described in section 3.2.1, the reward function is designed for simple forward locomotion and the PPO algorithm is used for training.



(a) Cartesian PD



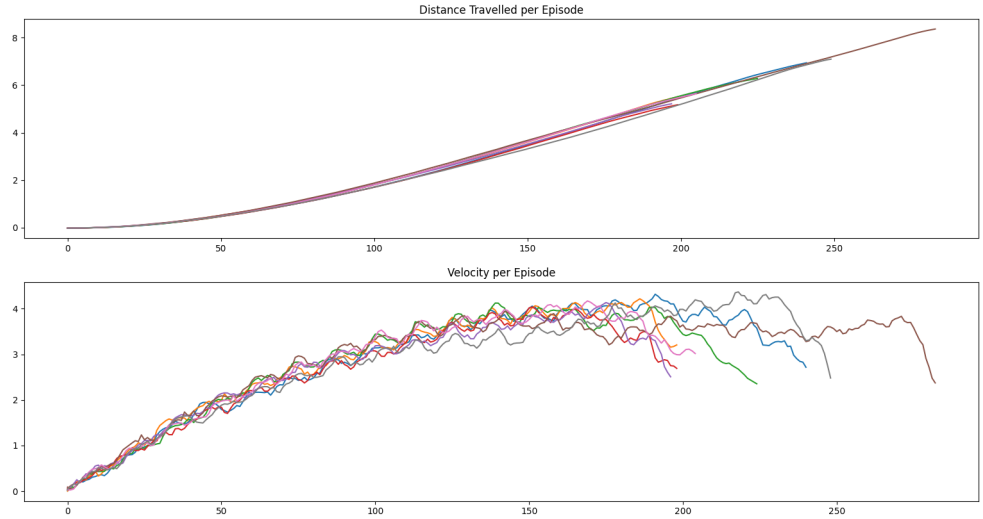
(b) Joint PD

Figure 3.2: Training Curves for Cartesian and Joint Space Control

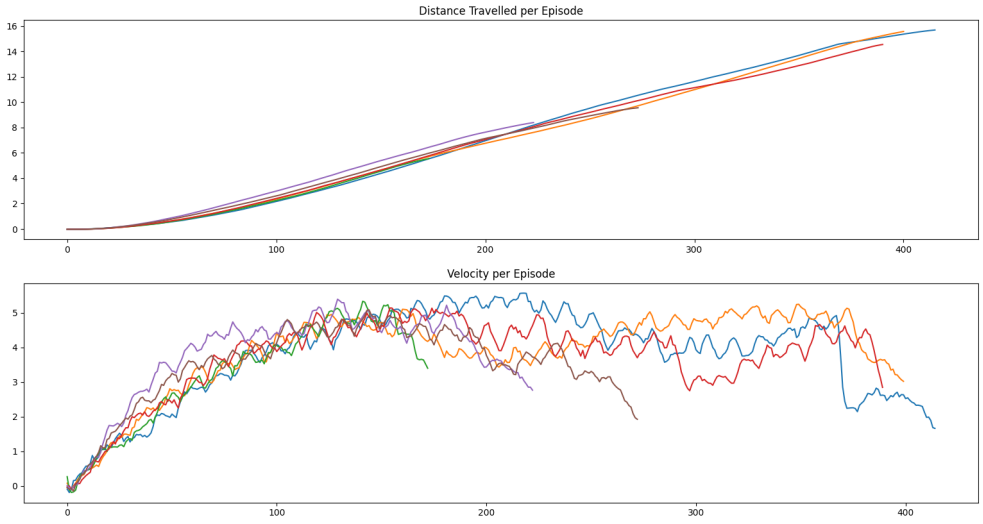
It can clearly be seen from Figure 3.2 that Cartesian PD control converges to a higher reward and episode length after the one million time steps. In practice this means that the robot under Cartesian PD control

can travel further in the intended direction and with a much lower chance of falling. This result can also be obtained qualitatively upon examination of the videos produced by each action space. Using Cartesian PD control, it can be seen that the robot has learned a galloping gait similar to many quadruped mammals and appears smooth and stable. Using Joint PD control, the movement of the feet appears erratic and shaky, an undesirable outcome. Therefore, the initial expectation of the Cartesian action space being more accurate and able to produce a smoother locomotion was proved correct and this action space was chosen to continue testing the MDP.

However, before the action space can be finalised, it is required to tune the action scaling array seen in line 8 of Listing 3.1. Similarly to the observation space, the actions of the policy network must be scaled to appropriate ranges for the feet x, y and z positions. From an initial scale array of $[x = 0.1, y = 0.05, z = 0.08]$, the x and z values were increased to 0.15 and 0.12 respectively so as to allow for greater stride lengths in the x direction and greater foot clearance over obstacles in the z direction. The plots in Figure 3.3 show the velocity and distance travelled by the robot over the same uneven terrain for the model trained initially and a new model trained with the updated scale array. It is clear to see that the increased x and z values have significantly improved the robot's ability to navigate uneven terrain as, on average, it travels further and faster over the same terrain.



(a) Initial Scale Array



(b) Updated Scale Array

Figure 3.3: Distance and Velocity Plots for Initial and Updated Scale Arrays on Uneven Terrain

3.2.3 Reward Function

The final stage of designing the MDP is creating the reward function which dictates the task that the system will train to complete. The models trained up to this point have rewarded distance travelled in the positive x direction only. To improve upon this, it was decided that the robot should also be energy efficient, travel in a straight line without deviating from the x axis and travel at a specified velocity. In order to achieve this, it is important to first understand how a reward function is constructed and the value that it returns. Typically a reward function will contain multiple terms which seek to reward or punish a particular action of the system. Each term must be weighted according to its desired importance in the overall reward and also to normalise the effects of different terms. The summation of these terms is a scalar output which the RL algorithm will seek to maximise over the training time. Thus, by rewarding positive actions and punishing negative actions, the system will eventually learn the desired behaviour.

In order to construct the reward function for this project, the function must contain terms measuring the robot's progress in the x direction, energy consumption, orientation and velocity. This is shown in equation 3.2:

$$R = w_d(x_t - x_{t-1}) - w_e\Delta_t(\tau * \dot{q}) - w_\theta|\theta| - w_v|\dot{x}_t - \dot{x}_{target}| \tag{3.2}$$

Where $w_d = 2$, $w_e = 0.008$, $w_\theta = 0.01$, $w_v = 0.001$, x is the base x coordinate, subscript t denotes the current simulation time, Δ_t is the length of a time step, τ are the motor torques, $\dot{q}$ are the motor velocities and θ is the base orientation. The weights for the reward function were initially chosen intuitively to normalise the effects of each term before being iteratively tuned to produced the desired effect on the model. The weight for the first term, distance travelled in the positive x direction, is the highest as this number is small compared to the others as well as being considered the most important. Implementation of this function in quadruped_gym_env.py can be see in Listing 3.2.

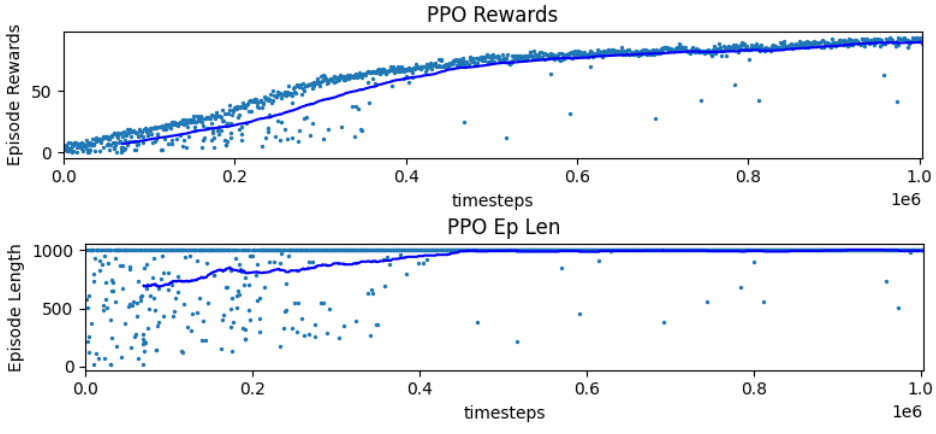
Listing 3.2: Implementation of the Final Reward Function

```
1   def _reward_lr_course(self):
2     """ Implement your reward function here. How will you improve upon the above? """
3     current_base_position = self.robot.GetBasePosition()
4     forward_reward = current_base_position[0] - self._last_base_position[0]
5     self._last_base_position = current_base_position
6     energy_expenditure = self._time_step * abs(np.dot(self.robot.GetMotorTorques(), ...
          self.robot.GetMotorVelocities()))
7     current_base_orientation = self.robot.GetBaseOrientation()
8     yaw_cost = abs(current_base_orientation[2])
9     current_base_velocity = self.robot.GetBaseLinearVelocity()
10    velocity_error = abs(current_base_velocity[0] - 5)
11    # clip reward to MAX_FWD_VELOCITY (avoid exploiting simulator dynamics)
12    if MAX_FWD_VELOCITY < np.inf:
13      # calculate what max distance can be over last time interval based on max ...
            allowed fwd velocity
14      max_dist = MAX_FWD_VELOCITY * (self._time_step * self._action_repeat)
15      forward_reward = min(forward_reward, max_dist)
16
17    return self._distance_weight * forward_reward - self._energy_weight * ...
        energy_expenditure -\
18          self._yaw_weight * yaw_cost - self._velocity_weight * velocity_error + 0.01
```

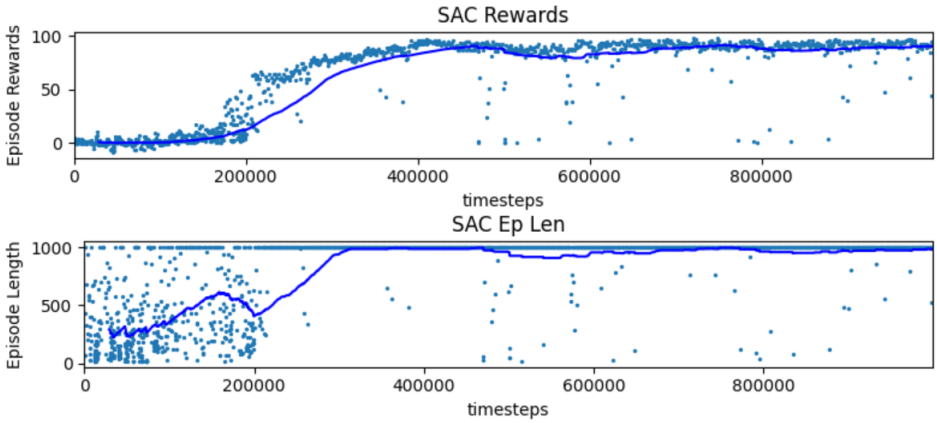
3.2.4 Deep Reinforcement Learning

The MDP has been completed as described above and this can now be passed to an RL algorithm to train the robot. The two options considered in this project were PPO and SAC. As the algorithms themselves were not the primary subject of this project, a simple quantitative comparison of the training curves between the two algorithms, using the same MDP, is used to determine the most suitable option. From Figure 3.4,

it can be seen that the reward and episode length after one million time steps converge to similar values. However, the PPO training curves display a much clearer upward trend compared to those for SAC, which contain several dips as the algorithm has trained a "bad habit". Indeed the videos of the two models also show that, despite the similar final rewards and episode lengths, the PPO trained model appears to move more smoothly and make better use of its dynamics to be more efficient.



(a) PPO Training Curves



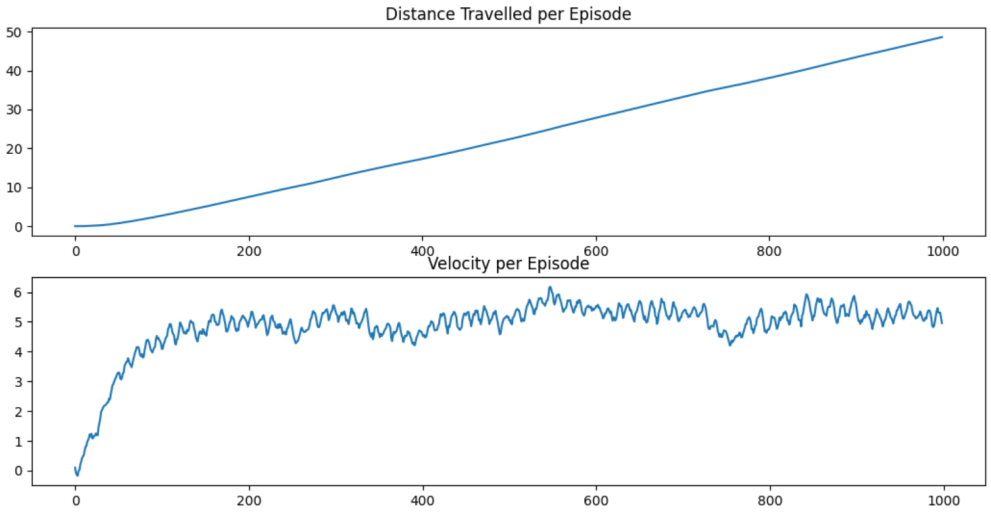
(b) SAC Training Curves

Figure 3.4: Training Curves for PPO and SAC trained models

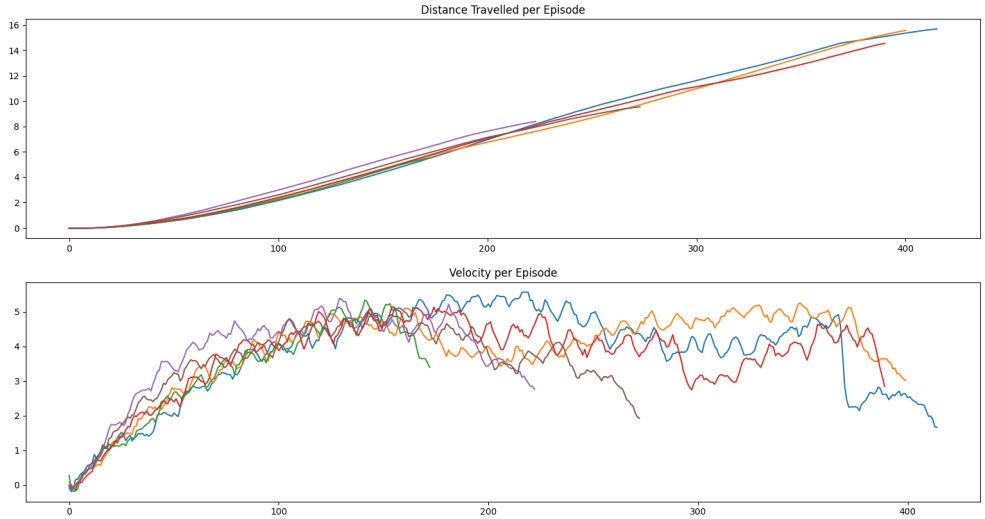
Each RL algorithm is used with a set of hyperparameters whose values can influence the performance at training time. Unfortunately, due to the length of time that it takes to run each algorithm for one million time steps, it was decided that these hyperparameters could not feasibly be tuned for this project.

3.3 Discussion

This section has presented the theory and steps taken to produce the resulting control policy for the robot. As instructed, the robot was trained in a uniform flat environment but measures were taken to improve the robot's robustness including varying the coefficient of friction on the surface and increasing the limit for the robot's feet in the z direction to better navigate low obstacles. Figure 3.5 shows the performance of the robot in the training environment and in uneven terrain. It can be seen that in the training environment the robot has achieved the desired velocity of 5 m/s and is able to complete the entire duration of an episode, while showing some adaptability to the uneven terrain owing to the higher foot step height. In order to make the robot more robust to obstacles and other disturbances, it could be trained with random obstacles and forces applied to the body. In addition, it can be seen from the PPO Rewards in Figure 3.4 that the reward for this control policy has just barely converged after one million time steps and may benefit from a longer training period.



(a) Performance in the Training Environment



(b) Performance in Uneven Terrain

Figure 3.5: Distance and Velocity Plots in the Training Environment and over Uneven Terrain

As mentioned previously, the observation space for the control policy described here is perhaps overly comprehensive and may face difficulties in transference to real world platforms due to noisy and imprecise sensor measurements. Hardware testing could be performed in an effort to account for poor measurements and could also be used to construct a more compact observation space. Another potential sim-to-real issue that was encountered in development was the scaling of the learned actions. It was found that very slight modifications to the scale array could drastically alter the performance of a control policy not trained for such scaling. Real world motors are likely to have some variance in their torque output and would need to be finely tuned to comply with the RL control policy.

Finally the control policy trained here is quantified using the Cost of Transport metric. Knowing the mass m, distance travelled d and energy consumption of the motors E, the CoT has been computed as follows:

$$CoT = \frac{E}{mgd} = 0.2 \tag{3.3}$$

This can be used to compare the RL control policy with that of the CPG control policy described in the first part of this report.

4. Conclusive Remarks

This report has presented the methods and results of quadruped locomotion control using both Central Pattern Generators and Deep Reinforcement Learning.

The CPGs used here consist of coupled Cartesian space Hopf oscillators. Typical quadruped vertebrate locomotion gaits are deduced and constructed from the oscillator coupling matrix, linking the phase offsets between each limb in order to specifically achieve desired footfall sequences. Validation of the obtained gaits was found by simulating the robot on the rack and examining various parameters for thorough inspection. As seen in section 2.2, successful walking and trotting gaits were achieved by imposing higher swinging frequencies, coherent with typical characteristics of these two common slow to medium speed gaits found in nature. On the other hand, the two commonly high speed gaits were found to require more extensive parameter tuning than that of the two previous, which would consist in additional future work. These could perhaps be achieved by extending the current controllers with various feedback loops, which would help attaining better tracking of the foot positions as well as the hip, thigh and calf joint angles. In any event, all the gaits seemed to require the addition of a Cartesian PD controller, as neither of them showed satisfactory results when using solely joint PD (for both the foot position and joint angles).

Chapter 3 focused on the Markov Decision Process design prior to be passed to an RL algorithm to train the robot based on the corresponding Observation, Action and Reward spaces. Similarly to the CPG locomotion, the addition of a Cartesian PD action space proved to be much more suitable for the considered applications. Careful tuning of the reward functions allowed meticulous shaping of the desired trained quadruped locomotion attributes so as to better make use of the robot's dynamics to be more efficient. Furthermore, two common DRL algorithms, namely SAC and PPO, were compared prior to final testing in an obstacle environment, where satisfactory results were obtained.

Once again, the Cost of Transport comparison metric is established to evaluate the performance obtained from the Reinforcement Learning to that of the more manual tuning implied from Central Pattern Generation. Despite the trained quadruped obtaining more efficient locomotion (longer distances at higher speeds, even in the presence of obstacles), it is hard to predict whether natural looking gaits would be achieved. Conversely, the control engineer has much more flexibility regarding the implementation of natural-looking gaits when using Central Pattern Generation, which ultimately could result in a more effective sim-to-real transfer.

Bibliography

[1] Ludovic Righetti and Auke Jan Ijspeert , *Pattern generators with sensory feedback for the control of quadruped locomotion*, IEEE International Conference on Robotics and Automation Pasadena, CA, USA, May 19-23, 2008

[2] H.-W. Park, M. Y. Chuah, and S. Kim, , *Quadruped bounding control with variable duty cycle via vertical impulse scaling*, IEEE/RSJ International Conference on Intelligent Robots and Systems 2014, pp. 3245–3252

[3] Ludovic Righetti and Auke Jan Ijspeert , *Design methodologies for central pattern generators: an application to crawling humanoids*, Conference: Robotics: Science and Systems II, University of Pennsylvania, Philadelphia, Pennsylvania, USA, 2006

[4] Cost of Transport - Wikipedia,
`https://en.wikipedia.org/wiki/Cost_of_transport` (accessed: 15.11.2021)

[5] G. Bellegarda and Q. Nguyen, "Robust high-speed running for quadruped robots via deep reinforcement learning," *arXiv preprint arXiv:2103.06484*, 2021.

[6] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke, "Sim-to-real: Learning agile locomotion for quadruped robots," in *Proceedings of Robotics: Science and Systems*, Pittsburgh, Pennsylvania, June 2018.

[7] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, "Learning agile and dynamic motor skills for legged robots," *Science Robotics*, vol. 4, no. 26, 2019.