LEGGED ROBOTS

PROJECT 2 : QUADRUPED LOCOMOTION WITH CENTRAL PATTERN GENERATORS AND DEEP REINFORCEMENT LEARNING

TOM RATHJENS, OWEN SIMON, SYLVAIN BEURET

Contents

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION QUADRUPED GAIT CONTROL

In this project, we treated a typical quadruped robot like the *Spot* robot from Boston dynamics. In the first part, we implemented four different gaits with Central Pattern Generators and controlled them with different methods (Cartesian PD, joint PD, torque). And the second part, we designed Markov reward functions for stable_baseline's deep reinforcement learning (DRL) algorithms to train our quadruped for running after specific targets.

1.2 CODE STRUTURE

All the code is done in python. It has multiple files, the main ones being :

- "hopf_network.py" which provides a CPG class skeleton for various gaits.
- "quadruped_gym_env.py" in which we implemented the reward functions for the training, the observation space and the different motor control modes.
- "quadruped.py" which contains the robot specific functionalities.
- "run_cpg.py" which maps these joint commands to be executed given by hopf_network on an instance of the quadruped_gym_env class.
- "load_sb3.py" : to check the results of our trained model
- "run_sb3.py" : to train our model with RL.

1.3 SOFTWARE AND INTRO TO METHODS

We used python *version : 3.9.18* with the following librairies :

1. **gym** *version 0.19.0* : Is used for testing and developing reinforcement learning agents.
2. **matplotlib** *version 3.3.4* : It's a 2D plotting library for creating static, animated, and interactive visualizations in Python
3. **numpy** *version 1.19.5* : It's used for all kind of mathematical operation. Indeed we used it a lot for matrix multiplications for example.

4. **pybullet** *version 3.2.0* : It's was used for simulating physical interactions of robotic and mechanical systems in a virtual environment.
5. **stable_baseline3** *version 1.3.0* : It provide us a set of high-quality reinforcement learning algorithms in Python.
6. **pyqt5** *version 5.15.6* : It's useful for creating a graphical user interface.

1.4 ACCRONYMES

Here we will but all the accronymes that are gonna be used in the report.

1. CPG : Central pattern generator
2. ANN : Artificial neural network
3. DL : Deep learning
4. RL : Reinforcement learning
5. DRL : Deep reinforcement learning
6. MDP : Markov Decision Process

CHAPTER 2

METHODS

2.1 CONTROL

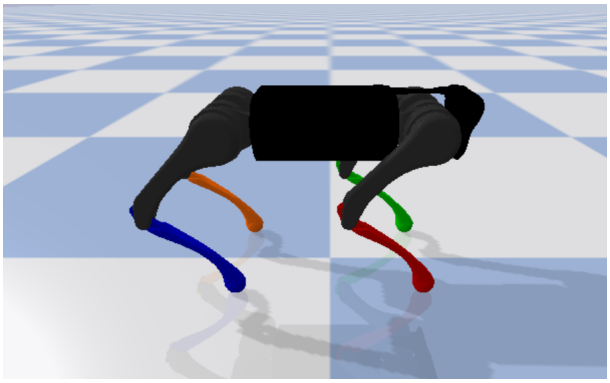


FIGURE 2.1
Quadruped model in the simulation

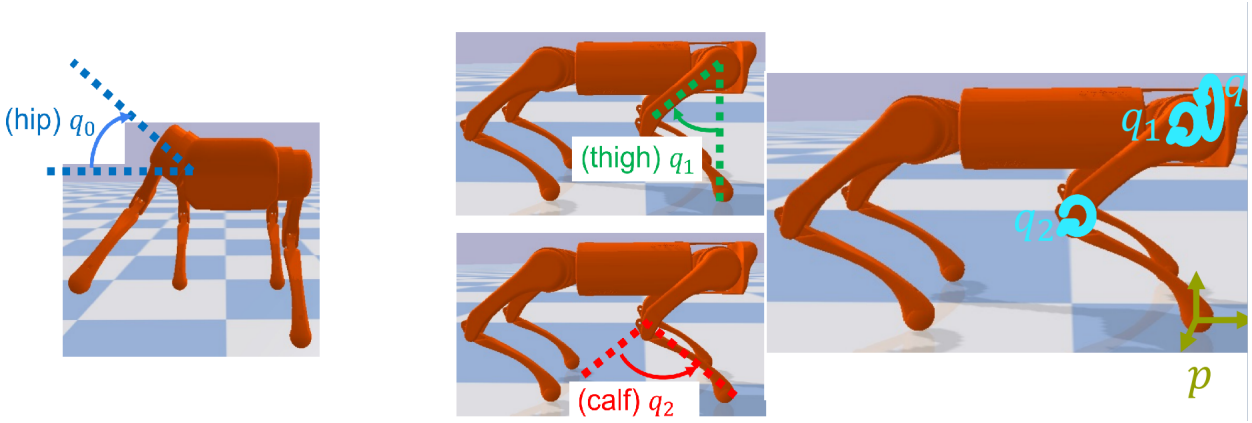


FIGURE 2.2
Angles quadruped

2.1.1 PROPORTIONAL DERIVATIVES CONTROLLERS

JOINT

In this part, we control the quadruped with a classic PD controller that regulates the **joint angle** position and velocity to the desired ones given by the CPG.

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

Where :
q=current joint angles= $\begin{bmatrix} q_0 \\ q_1 \\ q_2 \end{bmatrix}$
$\dot{q}$= current joint angle velocity
q_d=desired joint angle
K_p, K_D= diagonal matrices of proportional and derivative gains

2.1.2 CARTESIAN

Here we controlled the quadruped with a classic PD controller that regulates the **absolute joint** position and velocity to the desired ones given by the CPG. The control is done by computing the torques

$$\tau_{cartesian} = J^T(q)(K_{p,cartesian}(p_d - p) + K_{d\_cartesian}(\dot{p_d} - \dot{p}))$$

Where :
p=current foot position
p_d=desired foot position
$\dot{p}$= current foot velocity
J(q) = foot Jacobian
K_p, K_D= diagonal matrices of proportional and derivative gains

2.2 JOINT + CARTESIAN PD

In the project we use a mix of both the Cartesian and the joint controller therefore giving us

$$\tau_{\text{des}} = \tau_{\text{Cartesian}} + \tau_{\text{joint}}$$

Indeed, we are able to do so as when we get an xyz position we can always compute the inverse kinematics and therefore the desired angles.

2.3 CENTRAL PATTERN GENERATORS

2.3.1 CENTRAL PATTERN GENERATOR SUMMARY

CPG or Central Pattern Generators is a higher level system control scheme to create harmonic and frequency driven patterns. It's a good method to mimic rhythmic and synchronized movements found in walking and running. The CPG is a generator in which we input μS and ω and it gives us as output a set of positions. We can then apply a PD controller and potentially a Cartesian PD controller to control the quadruped joint positions and velocities to the optimised ones. And from the PD controller we get the input torques applied on each leg actuator. The CPG works by giving the desired leg position relative to the frame. This position is given by $\vec{x}_{\text{foot}} = \begin{bmatrix} -d_{\text{step}} r_i \cos(\theta_i) \\ 0 \\ \begin{cases} -h + g_c \sin(\theta_i) & \text{if } \sin(\theta_i) > 0 \\ -h + g_p \sin(\theta_i) & \text{otherwise} \end{cases} \end{bmatrix}$ Where g_c is the

maximal ground clearance and g_p is the maximum ground penetration. The $\dot{r}_i$ and the $\dot{\theta}_i$ are actually where the hand-tuned or the fed in parameters come into play.

$$\dot{r}_i = \alpha(\mu - r_i^2)r_i$$

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

where w_{ij} are the coupling strengths and ω_i, μ and w_{ij} are hyper parameters.

2.3.2 CENTRAL PATTERN GENERATOR IN CODE

The quadruped has 4 legs, each with 3 joints (hip, thigh, calf), for a total of 12 motors. The legs are numbered 0-3 in order Front Right (FR, 0), Front Left (FL, 1), Rear Right (RR, 2), Rear Left (RL, 3).

We implemented the quadrupeds CPG in the ***hopf_network*** python class. In it we implemented and derived the offset necessary for the trot, the pace, the walk and the bound and the equations necessary to its operation. We tuned the different parameters of the CPG to produce stable locomotion for the 4 gaits. In this file, the following variables were used :
$X = ([2, 4]) =$ CPG states. It has the amplitude of each leg in row 0, and phase in row 1
$r_i =$ current amplitude of oscillator
$\theta_i =$ phase of oscillator
$\sqrt{\mu} =$ desired amplitude of the oscillator
$\alpha =$ a constant that controls the speed of convergence to the limit cycle
$\omega_i =$ natural frequency of oscillation
$w_{ij} =$ coupling strength between oscillator i and j
$\phi_{ij} =$ desired phase offset between oscillator i and j

The method for each gate is :

- First : Defined the coupling matrix : The coupling matrix phi is the phase shift between each leg. It's done in the following function.

```
def _set_gait(self, gait):
```

- We model the oscillators as Hopf oscillators, which have the following equations. :

$$\dot{r}_i = \alpha(\mu - r_i^2)r_i$$

$$\dot{\theta}_i = \omega_i + \sum_{j=0}^{3} r_i \omega_{ij} \sin(\theta_j - \theta_i - \phi_{ij})$$

  The ω_i above being the natural frequency of oscillation, we also differentiated between the swing phase from 0 to π where ω="self._omega_swing" and from π to 2π where ω="self._omega_stance".

- Then the previous CPG states are updated with the new ones from the next time step, which we get by integrating with the Euler backwards method. $self.X = X + (X\_dot + X\_dot\_prev)/2 * self.\_dt$

```
    def update(self):
```

2.3.3 CONTROL

The CPG computes the desired foot positions. The Control part in the code has to find and adjust for each time step the correct torque inputs on the actuators of the joints, to bring the legs close to the desired position. We implemented the underlying lower level control with PD and Cartesian PD.

2.3.4 DISCUSSION CPG PART

The plot of the CPG states can be found here 3.1.1. In the first 2 graphs, the Cartesian position of each of the 4 joints is compared with the desired one. They were controlled with Cartesian PD with a proportional gain K_p = 500 (for all 3 coordinates) and a gain K_d = 20 (for all 3 coordinates). We tuned them by hand, through testing and we choose them because they gave us a quick and precise control. The derivative gain is responsible for anticipating the future trend of the error by considering its rate of change. and the proportional gain determines the strength of the response of the controller to the current error.

The hyper-parameters of the CPG that we played with (in Hopf_network) are μ = intrinsic amplitude : The higher the μ, the higher the quadruped jumps. We kept μ at 1 because the quadruped was not well synchronised for a higher μ. Then we increased omega_swing = 20 * 2π and omega_stance=8 * 2π. With this duty cycle-ratio (=$\frac{time_{leg\_on\_ground}}{time_{leg\_in\_air}}$) of around 40% we got our fastest quadruped with trot, with resulting body velocity of around 0.8[m/s]. We have t_{stance}=0.06[s] and t_{swing}= 0.04[s]. The lowest stable speed we reached was 0.25[m/s]. We expected to get a higher speed in this part but we didn't manage to get a stable quadruped for higher angular speeds, or higher μ unfortunately.

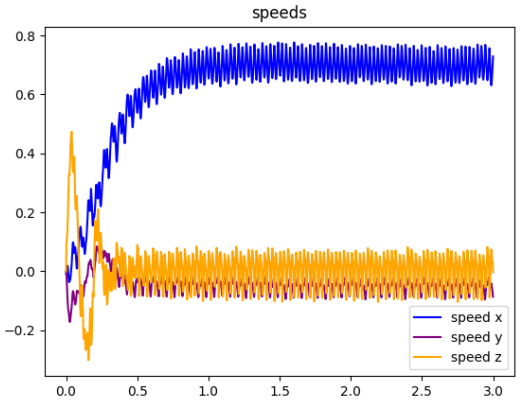


FIGURE 2.3
Highest quadruped speed reached

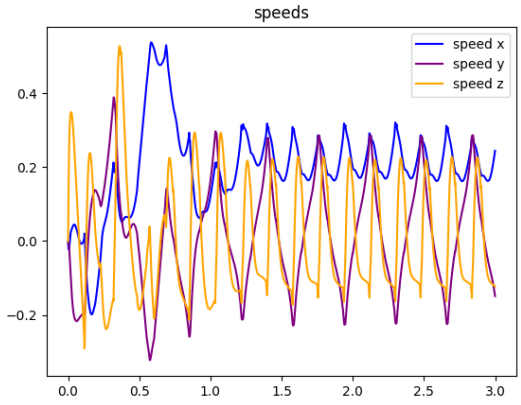


FIGURE 2.4
Lowest quadruped speed reached

2.4 DEEP REINFORCEMENT LEARNING

2.4.1 INTRODUCTION

To train our model, tune its parameters and find the best controller possible for our task, we trained it with each of the Motor control mode with Deep Reinforcement learning.
What is DRL : First : deep learning is a type of machine learning which uses Artificial Neural Network (ANN) to choose a "good" set of output for an input set. Then Reinforcement learning is : a type of machine learning in which an agent learns to make decisions through feedback in our case a reward function. This reward function should incorporate states and variables to have a high value when the wanted task is done successfully and a low value when it failed. Mathematically RL uses a Markov decision process (MDP). Finally Deep Reinforcement Learning is a mixed of Deep Learning and Reinforcement Learning. It's used when the states of the MDP have a higher dimension.
We used 2 different DRL learning algorithms on python, Soft Actor Critic (SAC) and Proximal Policy Optimisation (PPO). They have differences but we mostly used SAC because after some training, it appeared to work better and it is much more efficient with GPU. We did not modify the different hyperparameters of those algorithms, mostly because we focused on the effect of the reward function and

observation space.
These are the steps we used to improve our simulated gait to reach a faster, more robust and sturdier gait.

- Firstly in order to access the efficiency of different control modes we trained each model to interact with one of them. We trained them for the default "FWD_LOCOMOTION" reward function.
- Then we corrected/modified some of the reward functions and compared them.
- Then with our best model, we will train it further for the task "FLAGRUN". This means the folder with the FWD_LOCOMOTION containing our network will be the base input of the training for the FLAGRUN training. This means the observation space has to be the same for both task_env. This task is to follow and reach Targets the fastest possible. We iterate until we get a fast enough quadruped and a reliable controller.
- The last part is to improve the reward function and to make it more robust (to unsteady terrains / obstacles for example) and even faster.

During training we monitor 2 variables : The first one is the average duration of each simulation *ep_len_mean* and the second is the reward function value for each simulation *ep_rew_mean*. The reward function should globally increase with the number of simulations, and so should the episode length as we stop the restart the episode when the robot falls. So when the robot doesn't fall in the simulation anymore, each *ep_rew_mean* should average 1000 (=10[s]) which is the time after which the simulation automatically times out and resets.
A typical training should converge withing 1000000 time steps. For this reason and for time saving, most of our training last 1000000 episodes.

2.4.2 OBSERVATION SPACE

We trained with multiple observation spaces and observed the effect on the results, many inputs from the robot can be read and useful. Those values differ in scale thus it is important to normalize the observation space to avoid disproportionate importance on the output. The three main spaces used are:

- DEFAULT: In $\mathbb{R}^{28}$ only with the motor angles and velocities as well as the base orientation.
- LR_COURSE_OBS: Tried to give a maximum of observations to improve results. Ended up in $\mathbb{R}^{78}$. In addition to those from DEFAULT, we added The base linear and angular velocities, the motor torque, the position and velocity of the feet. We also have some contact information. Indeed, we assessed that the richer the information our robot would receive the more robust it would become through its ability to rely.
- OBS: Lastly, this observation space is a compromise between the two other as more is not always better and some observations might be redundant. We kept the motor and base inputs to end up in $\mathbb{R}^{46}$.

2.4.3 TASK_ENV AND REWARD FUNCTIONS

Multiple reward functions have been used to train a better model and try to build upon the last ones. We will discuss some of them and the reasons for their shortcomings and assess their results. The key challenge of a reward functions is to find the right weight for each term which can be tricky as we will see. The main theoretical steps used to achieve an effective model was to first train it to be stable in forward locomotion and then teach it to go increasingly faster. Lastly, to make it omnidirectional, training it in FLAGRUN. The expected results were not met immediately as the robot is often finding loopholes in the reward functions.

THE DEFAULT _REWARD_FWD_LOCOMOTION FUNCTION

```
    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]))
```

RESULTS This reward function enabled the robot to attain stability in about 400'000 steps, depending on the motor control. The main drawback of this method is that the robot puts forward the need for stability before anything else. This hence leads the robot to take advantage of the limitless precision of the simulation's motors in order to not fall over by taking minuscule steps.

REWARD WITH TORQUES PENALTY WITH FREQUENCIES AND PENALIZING TORQUES

OBJECTIVES Get the robots to lift its feet for longer periods of time. Try to influence the torques in order to get the robot to take longer strides. One of the problem with the approach of trying to influence the torques was that we did not find a good way to encourage them.

IMPLEMENTATION In order to implement this function we had to create a method to calculate the frequencies at which the robot touches the ground

```
        _, _, _, contacts = self.robot.GetContactInfo()
        frequencies = [0, 0, 0, 0]
        for index, contact in enumerate(contacts):
            # updating counters logic
            if self.__previous_touch[index] and contact == 0:
                self.__previous_touch[index] = False
            elif not self.__previous_touch[index] and contact == 1:
                self.__contact_counter[index] += 1
                self.__previous_touch[index] = True

            # for the first 1000 timesteps of sim the values are not representative
            if self._env_step_counter < 500:
                return frequencies

            frequencies[index] = self.__contact_counter[index] \
                / (self._env_step_counter * self._time_step)
        return np.array(frequencies)
```

and then implemented a _reward_fwd_locomotion

```
    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

        # take into account the frequency
        frequency_penalty = 0.05 * np.mean(self.__calculate_frequency())

        # take into account max indices
        max_indices = np.argsort(self.robot.GetMotorTorques())[-2:]
        large_torques_reward = np.clip(0.05 * np.linalg.norm(np.array(self.robot.
                                                GetMotorTorques()[max_indices])),0
                                                , 4)

        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])) \
            - frequency_penalty \
            + large_torques_reward

        return max(reward, 0)
```

We used a debugger in order to set the different weights to try to ensure that certain rewards and penalties did not engulf the reward function. We also clipped the values of the torques predicting that the robot would try to abuse these mechanics by having spasms. We also only tried to limit the robot's abuse of mechanics by taking only the two highest torques thinking that otherwise the robot would spasm with the other legs while they were not touching the ground.

RESULTS The robot outsmarted us and minimized the frequency penalty by jumping as high as possible in the air in order to minimize the frequencies penalties as he would never be touching the ground. He also took the opportunity of being in the air to spasm with two legs in order to maximize the torques reward.

_REWARD_FWD_LOCOMOTION V 2

OBJECTIVES Through this reward we were thinking that we would minimize the frequencies to kill two birds with one stone. We hoped that minimizing torques would lead our robot to be forced to take longer strides as he would have less strength to make small implosions

IMPLEMENTATION We could now take into account all the motor joints.

```
    large_torques_penalty = np.clip(0.03*np.linalg.norm(np.array(self.robot.
                                            GetMotorTorques())),0, 4)
    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])) \
            - frequency_penalty \
            - large_torques_penalty
```

RESULTS The SAC algorithm converged to a barely functionnal gait that was very low to the ground

_REWARD_FWD_LOCOMOTION V 3

OBJECTIVES The objective of this reward function was to get the body back at the right height.

IMPLEMENTATION We added a height penalty

```
    height_penalty = 200 * np.abs(self.robot.GetBasePosition()[2] - 0.3)
```

RESULTS The gait did not converge anymore. The major lesson learned through this trial was to try and keep the reward function as simple as possible. Our reward function was becoming far too complex and volatile. We therefore decided to backtrack to the original reward function and tried different avenues such as increasing observation space. This was a great learning experience trying to understand the complexity of getting the algorithms to converge.

_REWARD_FWD_LOCOMOTION V 4

OBJECTIVES This reward function was simplified to only keep the essential i.e. the distance traveled.

IMPLEMENTATION Reward function

```
    current_position = self.robot.GetBasePosition()
    fwd_reward = current_position[0] - self._last_position[0]
    self._last_position = current_position
    forward_reward = min( forward_reward, max_dist)
    return self._distance_weight * forward_reward
```

RESULTS The results were good as it moves faster but as expected, the stability was not perfect.

2.4.4 ACTION SPACES

Three main action spaces were used in our training. The different outputs are scaled between [-1,1], this enables to constrain the output values to avoid physically infeasible states, it also helps to stabilize the training as extreme outputs might influence the model of an undesirable amount.

PD

The PD mode uses the classical control of proportional and derivative strategy. It calculates the difference between the desired and actual joint angles and modulates them with some gains to give the right torques to reach optimal position of the limbs. In this case the DNN outputs in angle space. This method main advantage is the simplicity of use and calculation but can lack some precision.

CARTESIAN PD

This mode is an extension of the PD control mode to the Cartesian space as it modulates the actual position of the limbs instead of the motor angles. As seen previously, from the desired position fed by the DNN we compute the inverse kinematics to try to get the joints to also converge to the right value. It gives better results as it more robust.

CPG

This action space used the CPG we defined in the prior section and modulates the parameters μ and ω_i to make the model adapt to what is happening around it through the observation space. The advantage of using CPG is that with the DNN we will be interfacing with a higher level controller that already has a very animal like gait and making it more robust. Hence, it should lead to the most animal like gaits.

CHAPTER 3

RESULTS

3.1 CPG

Ïn the following pages, we can see the different graph showing our results for the different CPG gaits. The desired and actual position are close enough to have good results.

3.1.1 PLOTS OF DIFFERENT CPG STATES OF THE QUADRUPED TROT GAIT

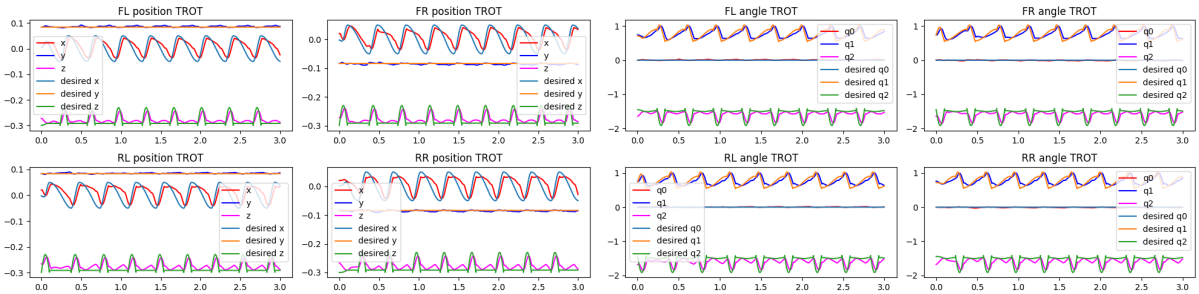
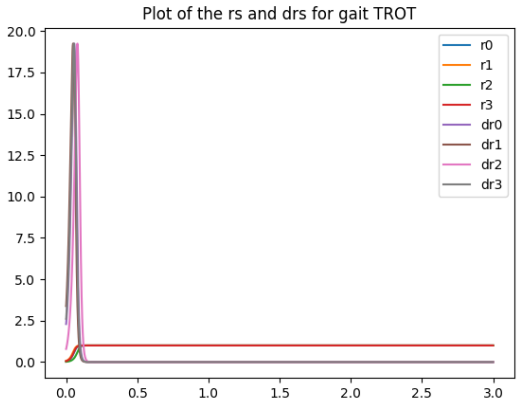


FIGURE 3.1

FIGURE 3.2



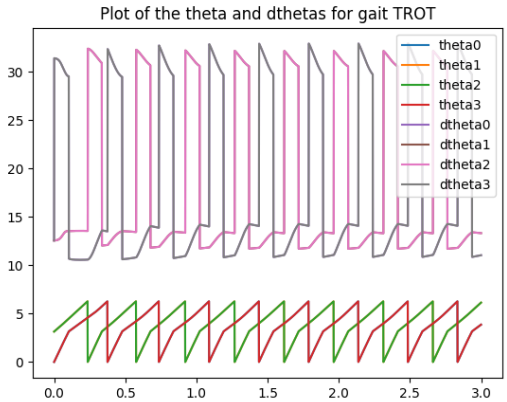


FIGURE 3.3

FIGURE 3.4

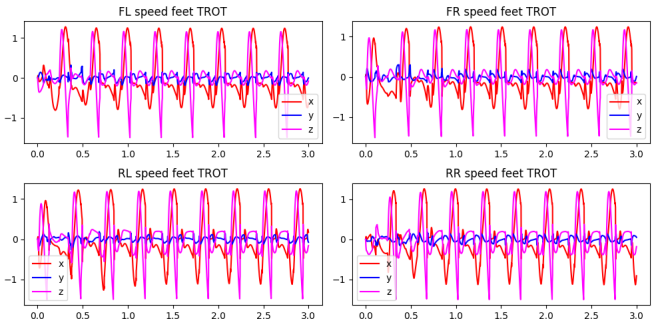


FIGURE 3.5

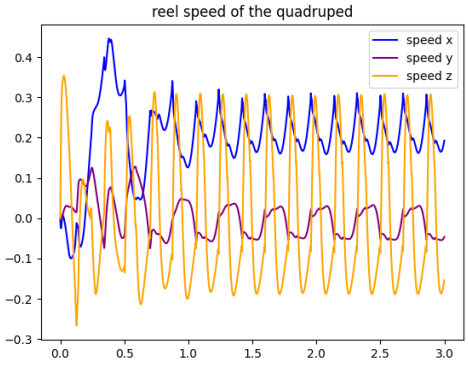


FIGURE 3.6

3.1.2 PLOTS OF DIFFERENT CPG STATES OF THE QUADRUPED BOUND GAIT

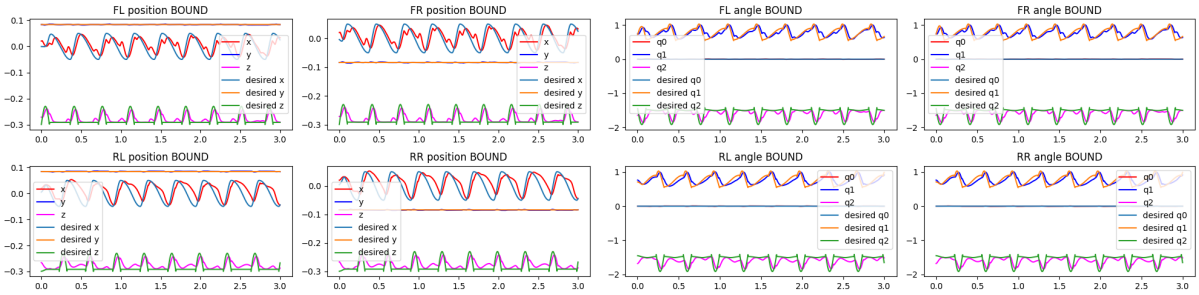


FIGURE 3.7

FIGURE 3.8

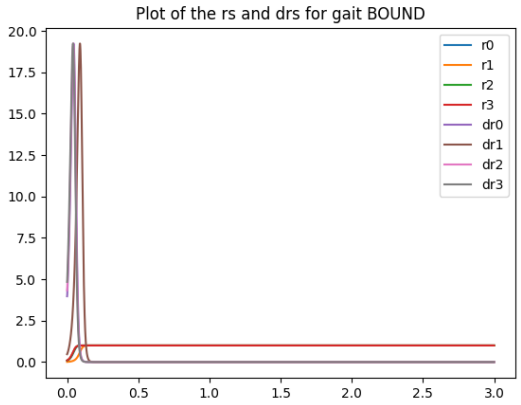


FIGURE 3.9

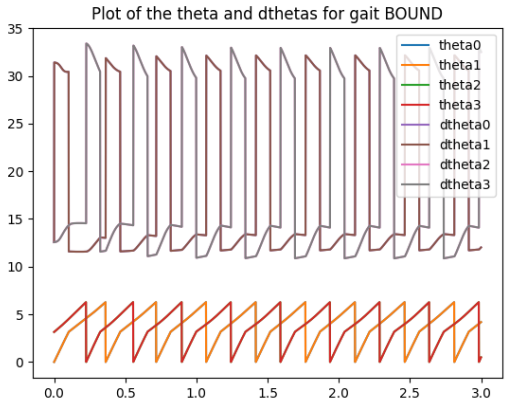


FIGURE 3.10

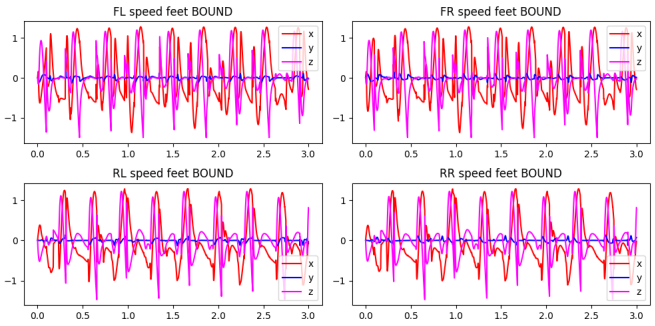


FIGURE 3.11

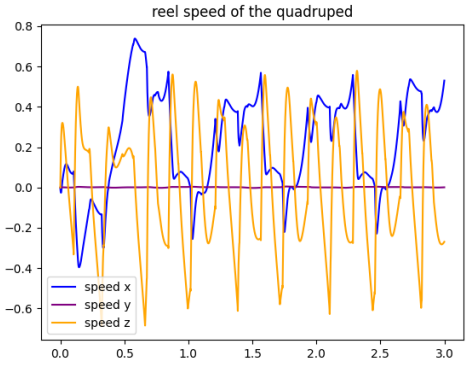


FIGURE 3.12

3.1.3 PLOTS OF DIFFERENT CPG STATES OF THE QUADRUPED PACE GAIT

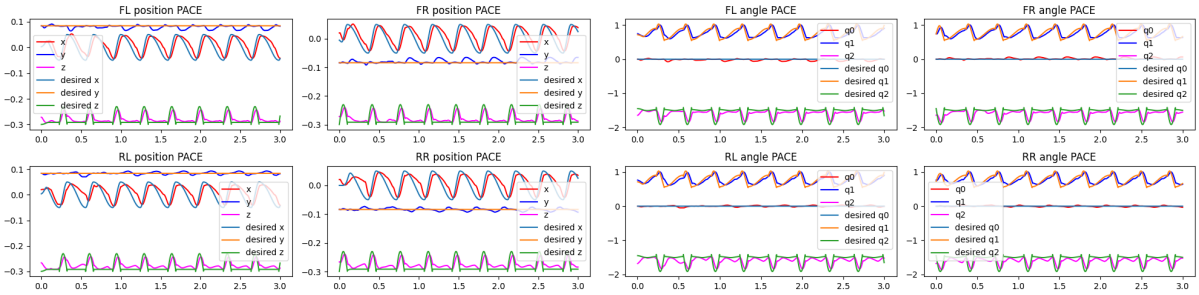


FIGURE 3.13

FIGURE 3.14

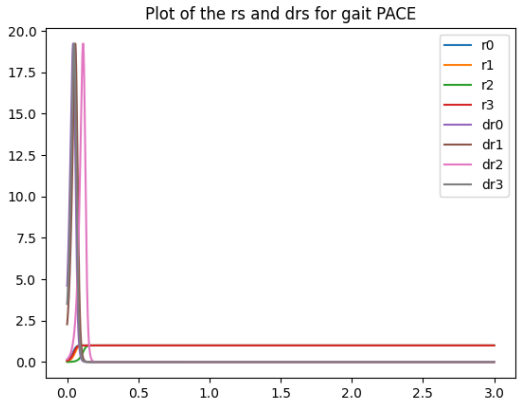


FIGURE 3.15

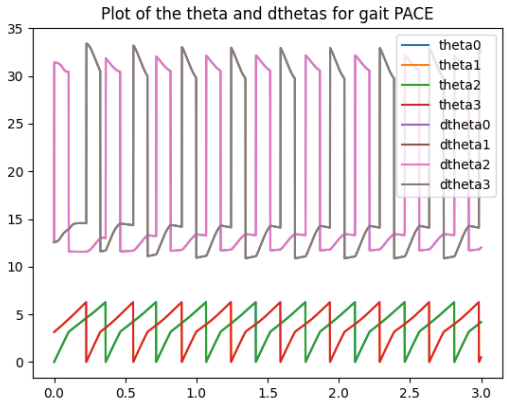


FIGURE 3.16

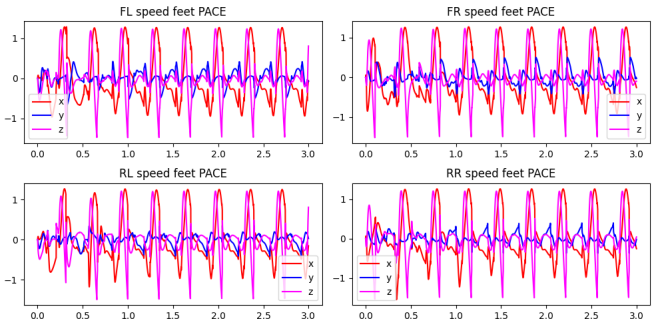


FIGURE 3.17

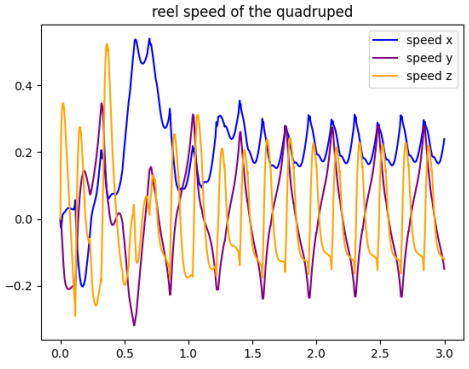


FIGURE 3.18

3.1.4 PLOTS OF DIFFERENT CPG STATES OF THE QUADRUPED WALK GAIT

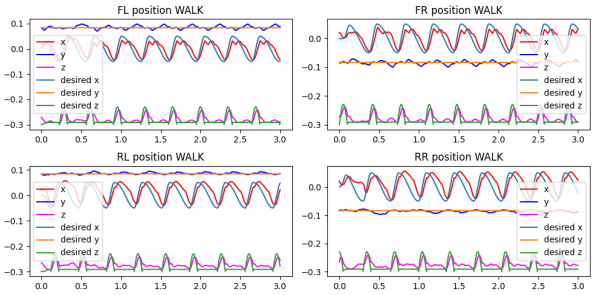


FIGURE 3.19

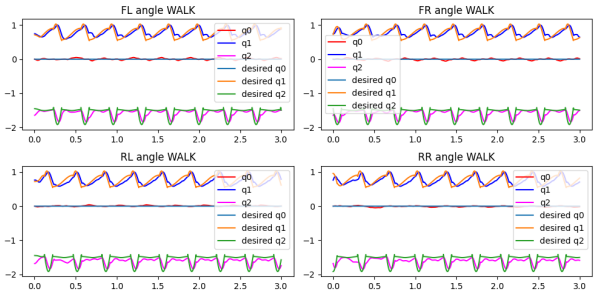


FIGURE 3.20

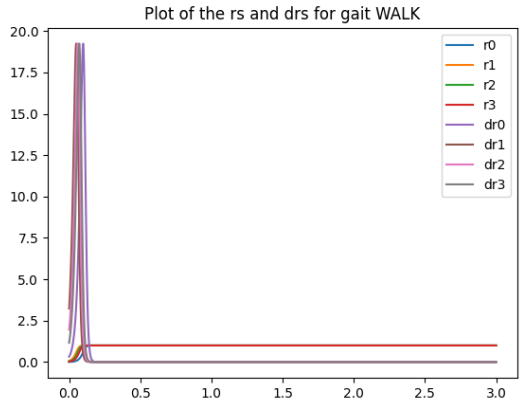


FIGURE 3.21

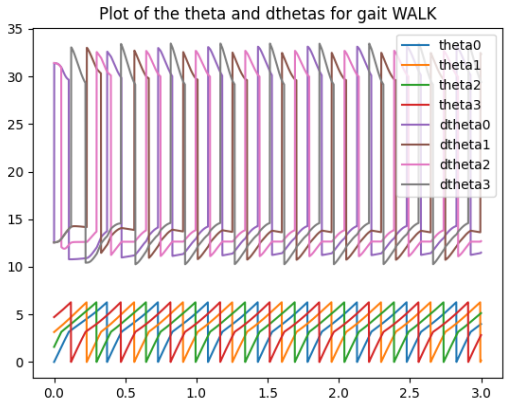


FIGURE 3.22

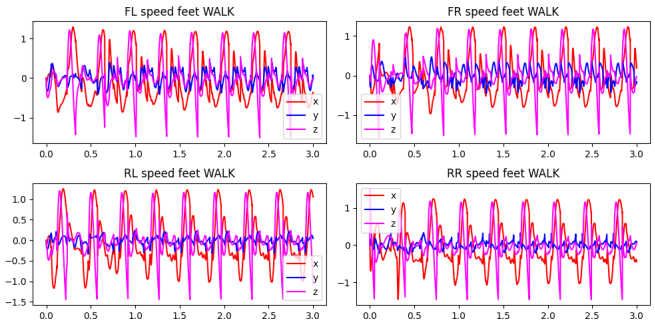


FIGURE 3.23

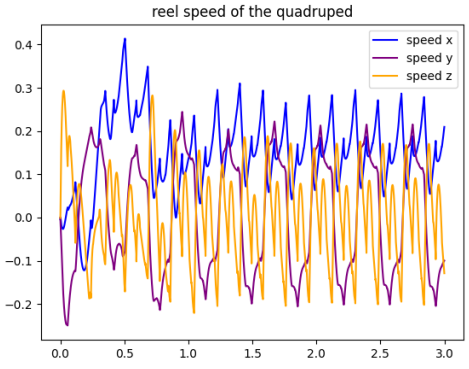


FIGURE 3.24

3.2 REINFORCEMENT LEARNING

We will discuss the different steps done in our training and the results associated for each type of control mode.

3.2.1 TRAINED MODELS

In this section, different models are discussed to better understand the stages occurring in training and the challenges associated. Three types of control mode have been trained i.e. PD, Cartesian PD and CPG.

PD_v1

This first model was used to better understand the influence of each input. It is very simple and uses the DEFAULT functions. The first curve of rewards obtained is almost perfect but the actual gait are only tiny steps which are slow and prone to failure on uneven ground. The next step was to fix this, it was more difficult than expected as the robot tried to outsmart our reward function at every turn. The weights in the reward functions have a huge impact on the final result but difficult to really estimate the best choice before training thus making it a little bit of a haphazard. We still continued the training by gradually increasing the desired velocity and finally training it for the FLAGRUN. The results were not perfect as the robot was only using its front legs and struggled to find the goals.
In the following graphs showing the results of the second stage of training where the goal was to go forward and the speed was augmented to 1m/s which it almost meets. At 600 timesteps, he trips but then recovers. In the speeds graph, we can see that the actual speed oscillates around the desired speed value. We can also see the problem of step height in the z position as it is almost constant meaning that the robot doesn't lift it's feet much.

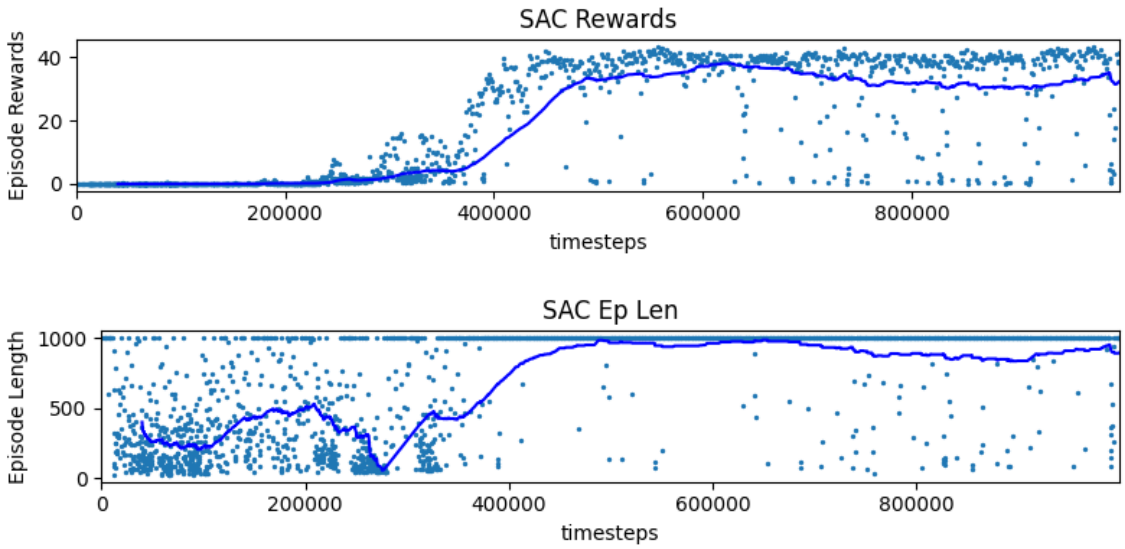


FIGURE 3.25
Curve of PD_v1 for FWD_LOCOMOTION

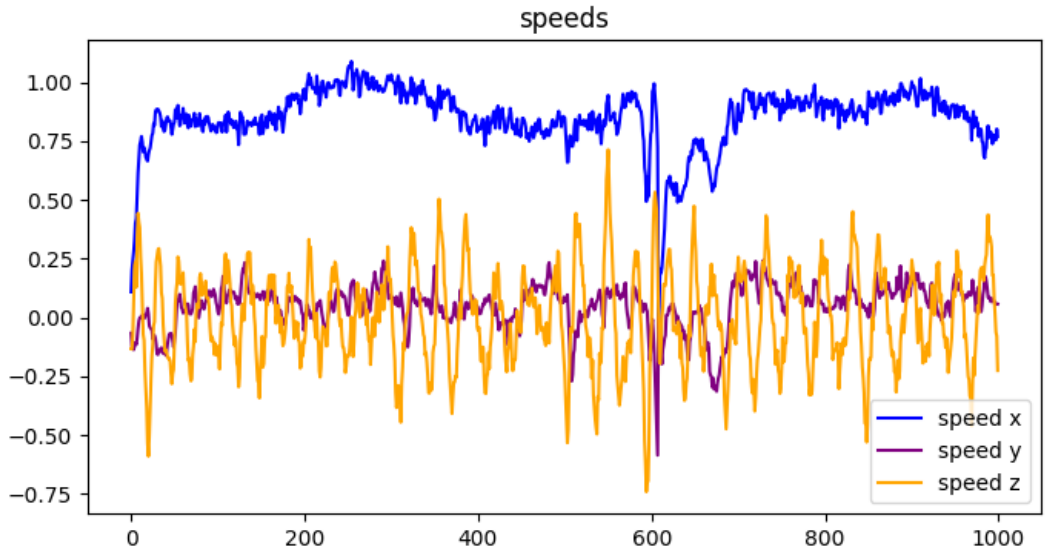


FIGURE 3.26
Graph of the speeds of PD_v1 for FWD_LOCOMOTION

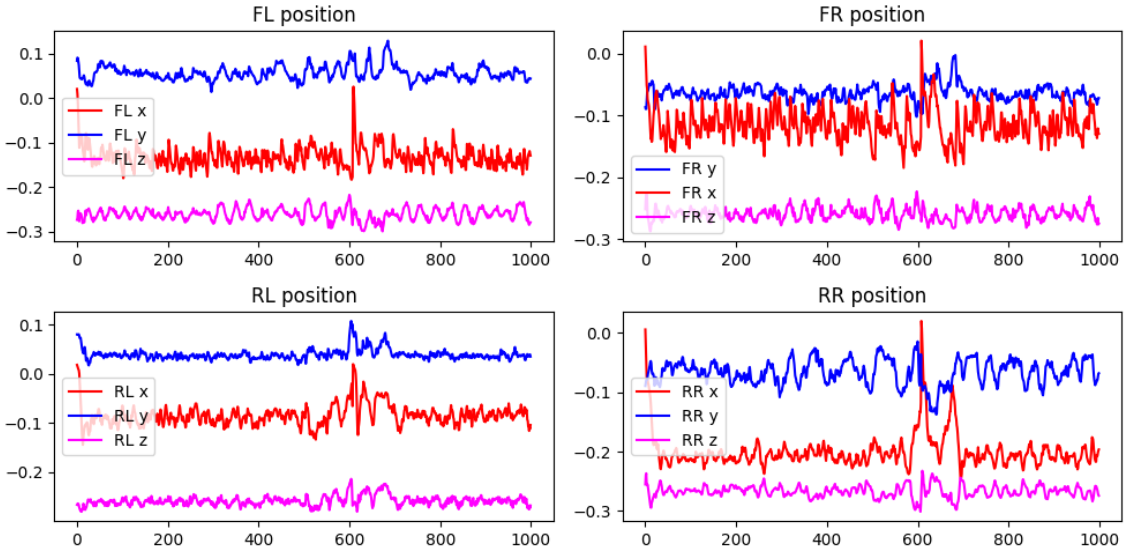


FIGURE 3.27
Graph of the feet positions of PD_v1 for FWD_LOCOMOTION

PD_v2

After some encouraging results from the first part and results coming from training with the other type of control mode, we tried to change the observation space to *OBS*, giving the model more information to improve itself with more feedback. The different steps in the construction of this model were similar to the other i.e. first trained to go straight and then in different directions. Multiple problems occurred in the process. First it did not lift its feet then when trying to fix this problem, it only used the front ones. Then we only kept the final distance as a reward which worked but lacked stability. In the end, the model is still not perfect even if it sometimes achieves some beautiful gaits but struggle to initiate it. For the FLAGRUN part, it manages to go in the right direction but gets lost before reaching the goal as we can see in Figure 3.25.
In the following graph, we can see the gait it created in the training part where only the final x position mattered. The peaks are synchronized in the front and back legs creating a bound gait.

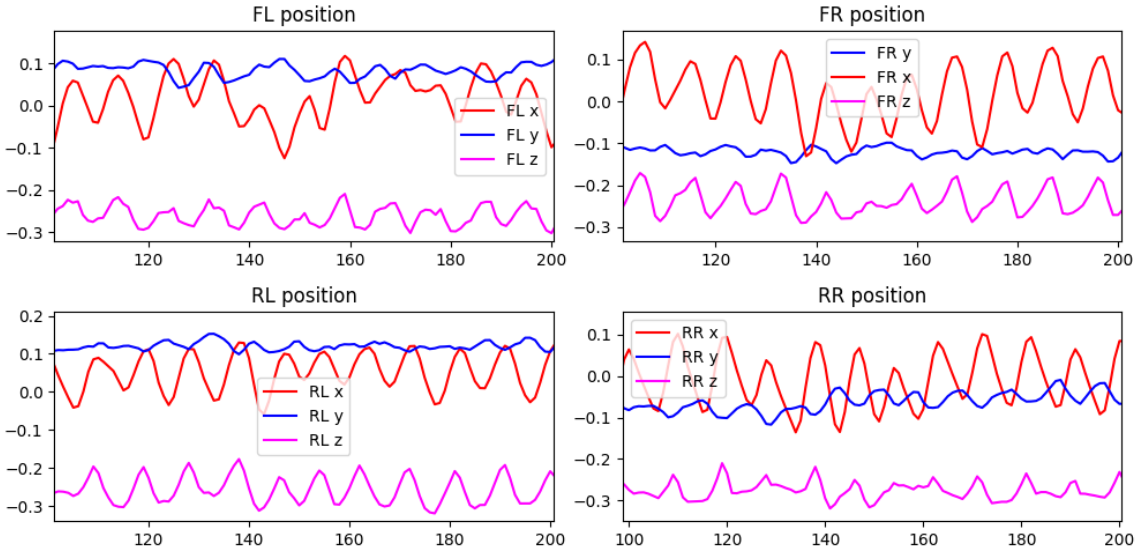


FIGURE 3.28
Graph of the feet positions of PD_v2 for FWD_LOCOMOTION

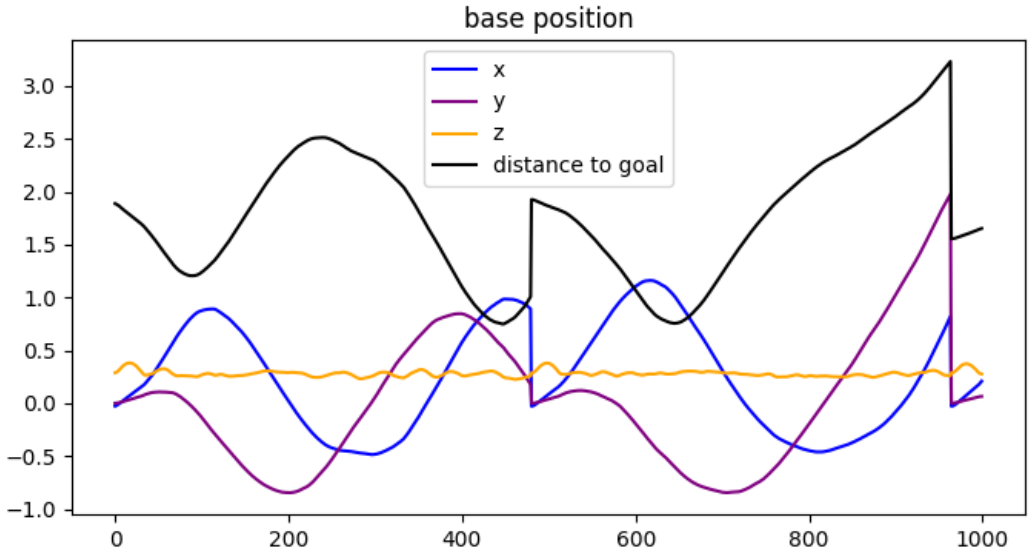


FIGURE 3.29
Position and distance to goal for PD_v2 in FLAGRUN

CARTESIANPD_V1

The first version of Cartesian PD used the DEFAULT observation space. The reward function curve looks promising but the actual model is not very stable. A lot of tries were made trying to improves the results by changing the different reward function or action space but nothing conclusive.

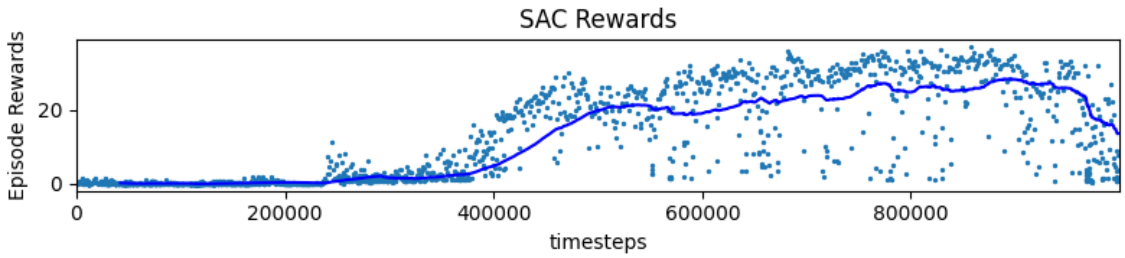


FIGURE 3.30
Reward curve of CPD_v1

CARTESIANPD_V2

To achieve better results than the previous version, the observation space was changed to have more inputs. LR_COURSE_OBS was used. A first model was trained but stopped at 300'000 steps, the curve begins to rise and the actual model works but could need longer training as it is not perfectly stable. Therefore, a model was trained with the same inputs but with 1.5m steps but the result was worse than with 300'000 as it was just standing at the origin doing breakdance moves. This has shown us that RL has some part of randomness and is frustrating as the training takes time.
After this failure, we changed the reward function to _Reward_fwd_Locomotion_v_4 which is simpler, taking only the distance in x. Results are the best yet, lack a bit of stability and might trip in case of uneven ground because the stride is not very high. The next step was to train it for the FLAGRUN. Nothing worked well, the instabilities were accentuated and the robot couldn't reach the goals but still moves in their direction.

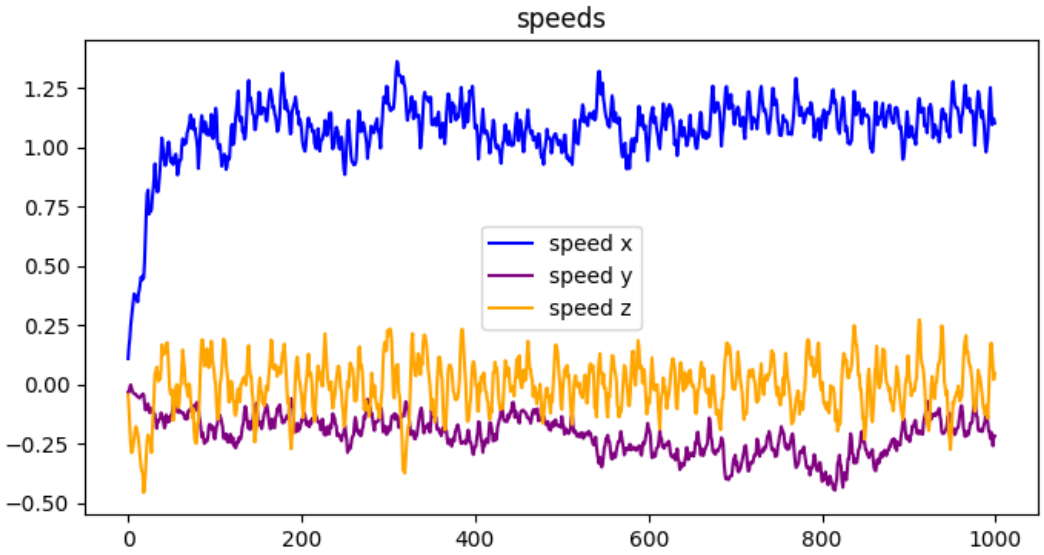


FIGURE 3.31
Speeds with Reward_fwd_Locomotion_v_4

CPG

We also trained with the CPG control mode, at first with the FWD_LOCOMOTION task and a reward function assuring the stability and trying to reach a desired speed of 1m/s. The result is almost perfect as

we can see in the following Figures. Firstly with the reward curves and then with the contact of the feet with ground. The gait achieve is a trot as the front right and rear left are synchronized. We then increased the desired speed but it didn't work anymore. Lastly, training it for FLAGRUN gave disappointing results.

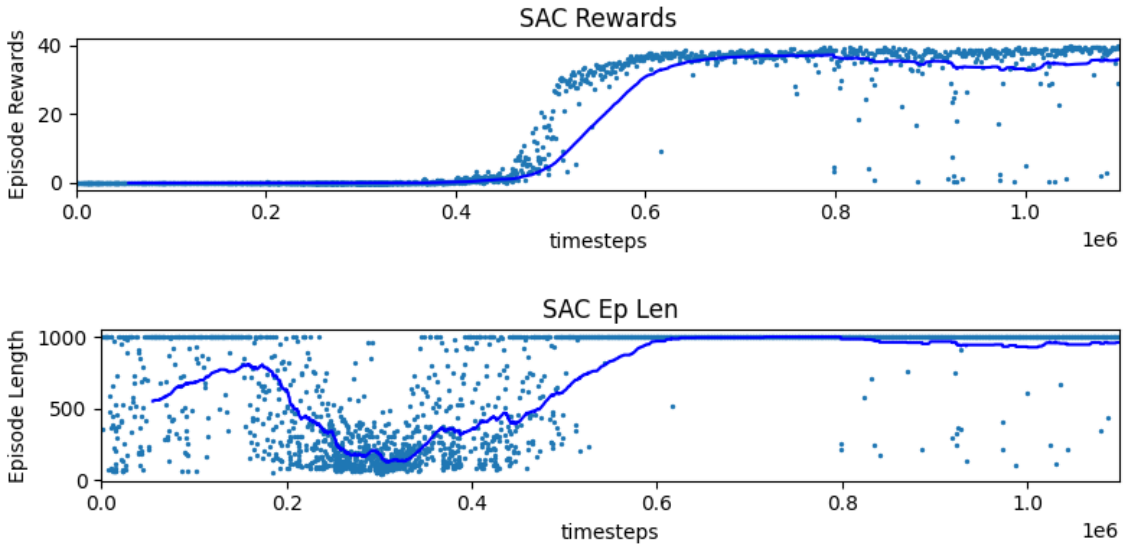


FIGURE 3.32
Curve of CPG for FWD_LOCOMOTION

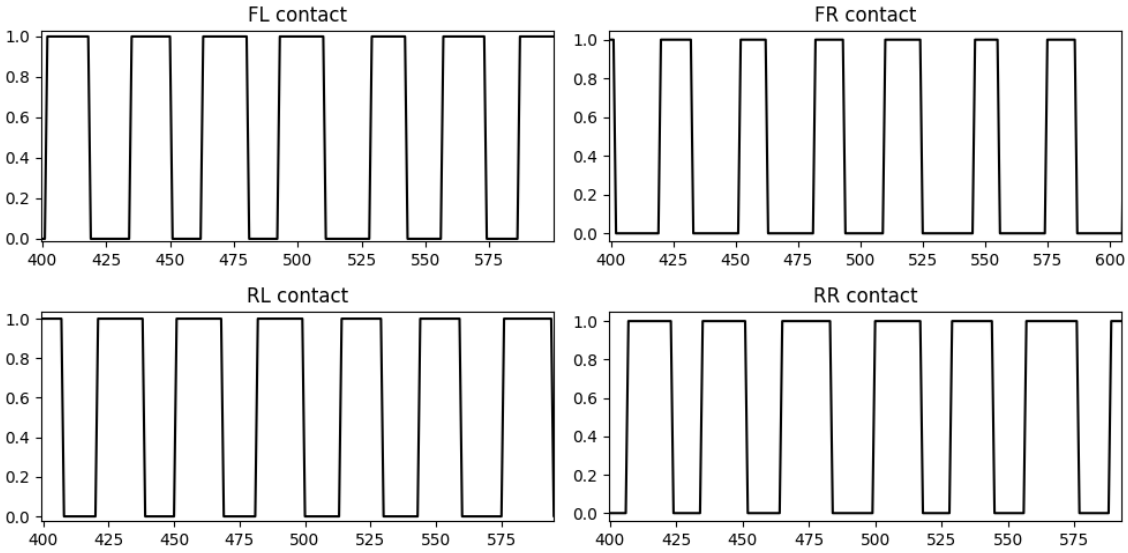


FIGURE 3.33
Contact timing of CPG for FWD_LOCOMOTION

CHAPTER 4

DISCUSSION AND CONCLUSION

4.1 EVALUATION OF OUR SYSTEM

In the end of this long and interesting project we have understood how to train a robot how to walk towards objects. Finally we have reached a model in cartesian PD that is quite animal like.

4.1.1 THE LIMITS OF OUR SYSTEM

UNPREDICTABLE SITUATIONS We noticed our system has an extremely hard time to react to unforeseen situations such as on the competition course in which when it encounters the slightest bump in the road or a sliperry surface, just stops working or continues as if nothing had changed.

NO FORMAL PROOF OF OUR SYSTEM WORKING This lack of predictability leads us to have a system with no formal proof of convergence, and that therefore could never be applied to real world applications next to humans

4.2 HOW TO IMPROVE THE SYSTEM

TRAINING OUR SYSTEM MORE Let us start by the most obvious and attainable solution which is to train our system in as many situations as possible to make it as robust as possible. Indeed, the major weakness of RL is its inability to handle unexpected situations therefore if we train it enough, similarly to the quadruped from Anybotics, it should become stable and resilient enough to handle any problem seen in simulation.

CREATION OF A MORE SEMANTIC REWARD FUNCTION It was very inspiring to hear Kim talk about the creation of a more semantic programming language and it would be an interesting alley to explore. Indeed, just in this project it would have been interesting idea trying to create a class which creates and homogenises reward function parameters automatically to avoid these parameters taking too much space. It would also be interesting to have that class adapt the reward function as the robot as a function of the performances. A changing reward function over time of sorts.

MAKING A CONVEX OPTIMIZATION FUNCTION Trying to transform the problem into a convex optimization problem could be potentially beneficial as if done successfully could offer a more robust proof of the stability and consistency of the system.

RELY MORE ON LITERATURE Using more the literature and the different results could have helped, especially in the beginning to have a better model to start which would have given us more time to train efficient models. For example [1] gives more details on a reward function that was proved to work but was not used in our models.

4.3 DISCUSSION ON REINFORCEMENT LEARNING

Overall, the RL part was not as good as expected. We succeeded in multiple situation to make it move forward but struggled to have effecient models to build on for the FLAGRUN part. Each resolution of a problem brought many more with it and the time it takes to see a training through was debilitating. We discussed the most interesting models in the development but most of them failed, even with 30-40 models trained the best results were not excellent. The transition to a real world application seems possible but would probably bring a lot of problems as the noises in inputs and outputs would affect the motion even if some was added during training. Despite this we still manage to have some interesting models. Also, the different action spaces used showed interesting results, The cartesian PD action space seemed to be the more reliable and adjustable. To end on a light note it was sometimes funny to train a robot that would act as a rebellious child trying to get around every rule and use every opportunity to not do what you want it to do.

CHAPTER 5

BIBLIOGRAPHY AND ANNEXES

5.1 BIBLIOGRAPHY

1. G. Bellegarda, Y. Chen, Z. Liu, and Q. Nguyen, “Robust high-speed running for quadruped robots via deep reinforcement learning,” in 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2022, pp. 10 364–10 370.

5.2 ANNEXES

5.2.1 VIDEOS OF SIMULATIONS

1. TROT_HIGH_0,8ms.mp4
2. TROT_LOW_0.25.mp4
3. PD_v1_FWD_vdes=0.5.mp4
4. PD_v1_FWD_vdes=1.mp4
5. PD_v1_FLAGRUN.mp4
6. PD_v2_FWD_OBS.mp4
7. PD_v2_FWD_OBS_distmax.mp4
8. PD_v2_FLAGRUN_OBS.mp4
9. CPD_V2_FWD_LR_COURSE_OBS.mp4
10. CPD_V2_FWD_LR_COURSE_OBS_1.5msteps.mp4
11. CPD_V2_FWD_LR_COURSE_OBS_distmax.mp4
12. CPD_V2_FLAGRUN_LR_COURSE_OBS.mp4
13. CPG_FWD_OBS_vdes=1.mp4
14. CPG_FWD_OBS_vdes=2.mp4
15. CPG_FLAGRUN_OBS.mp4