VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER SCIENCE AND ENGINERING

REINFORCEMENT LEARNING BASED LEGGED LOCOMOTION OF QUADRUPEDAL IN A COMPLEX ENVIRONMENT

CSE6099: MASTER THESIS

FINAL REVIEW REPORT

SUBMITTED BY:

ANURUP SALOKHE [19MCS0060]

SUBMITTED TO:

SCIENTIST G. MADHAV M KUBER

VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering

DECLARATION

I, hereby declare that the project entitled **"Reinforcement Learning Based Legged Locomotion of Quadrupedal in a Complex Environment"** submitted by me to the School of Computer Science and Engineering, Vellore Institute of Technology, Vellore-14 towards the partial fulfilment of the requirements for the award of the degree of **Master of Technology in Computer Science and Engineering** is a record of bonafide work carried out by me under the supervision of **Scientist G. Madhav M Kuber.** I further declare that the work reported in this project has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma of this institute or of any other institute or university.

Name: Anurup Salokhe
Reg. No: 19MCS0060

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering

CERTIFICATE

The project report entitled **"Reinforcement Learning Based Legged Locomotion of Quadrupedal in a Complex Environment"** is prepared and submitted by **Anurup Salokhe (Register No: 19MCS0060),** has been found satisfactory in terms of scope, quality and presentation as partial fulfillment of the requirements for the award of the degree of **Master of Technology in Computer Science and Engineering** in Vellore Institute of Technology, Vellore-14, India.

Guide

(Name & Signature)

ACKNOWLEDGEMENT

The project **"Reinforcement Learning Based Legged Locomotion of Quadrupedal in a Complex Environment"** was made possible because of inestimable inputs from everyone involved, directly or indirectly. I would first like to thank my guide, **Scientist G. Madhav M Kuber,** who was highly instrumental in providing not only a required and innovative base for the project but also crucial and constructive inputs that helped make my final product. My guide has helped me perform research in the specified area and improve my understanding in the area of web development and internet of things and I am very thankful for her support all throughout the project.

I would also like to acknowledge the role of the HOD, **Dr. Sasikala R.,** who was instrumental in keeping me updated with all necessary formalities and posting all the required formats and document templates through the mail, which I was glad to have had.

Finally, I would like to thank **Vellore Institute of Technology**, for providing me with a flexible choice and execution of the project and for supporting my research and execution related to the project.

CONTENT TABLE

Chapter Titles **Page No.**

LIST OF ABBREVIATIONS

RL	Reinforcement Learning
MDP	Markov Decision Process
PPO	Proximal Policy Optimzation
RWF	Reward Function
CNN	Convolution Neural Network
PD	Proportional Derivative
LLP	Legged Locomotion Policy
STF	State Transfer Function
UML	Unified Modelling Language

LIST OF FIGURES

ABSTRACT

In this work, we propose a Reinforcement learning based legged locomotion quadruped comprising active model that learns to localize itself in a space known by the system in complex cluttered environment. Legged robots, like quadrupeds, would be introduced in terrains with difficult-tomodel and predict geometries, so they must be fitted with the ability to generalize well to unexpected circumstances, according to the problem statement. We proposed a novel method for training neural-network policies towards terrainaware functionality in this paper, which incorporates state-ofthe-art frameworks based-model for robot navigation and reinforcement learning. Our strategy is as follows instead of using physical simulation, we formulate Markov decision processes (MDP) based on the assessment of complex feasibility parameters. Using both exteroceptive and proprioceptive measurements, we employ policy-gradient approaches that develop policies that plan and execute foothold and baseline movements in a variety of environments independently. We put our method to the test on a difficult range of virtual terrain scenarios that include features like narrow bridges, holes, and stepping stones, as well as train policies that successfully locomotive in all situations. The simulation can be achieved on high performing CPU system using Open AI GYM and also using TensorFlow on Rex GYM environment tool.

Chapter 1

INTRODUCTION

INTRODUCTION

1.1Background

The problem of engineering agile locomotion for quadruped robots has been studied for several years. This is on the grounds that controlling an under-activated robot performing exceptionally powerful movements that require complicated equilibrium is troublesome. Classical methods often necessitate a great deal of practice and time-consuming manual tuning. Is it possible to automate this procedure? Deep reinforcement learning has made considerable strides recently. These algorithms are capable of solving locomotion problems without the need for much human interaction. In any case, most of these trials are done in recreation, and a regulator prepared in reproduction regularly fizzles in reality. Model inconsistencies between the virtual and actual physical systems create this reality difference. This inconsistency is brought about by an assortment of variables, including unmodeled elements, mistaken reproduction boundaries, and mathematical blunders. Worse, this gap is intensified in locomotion tasks. The transitions of touch circumstances split the control space into broken fragments while a robot moves quickly with regular contact changes. Any small model difference can be amplified, resulting in bifurcated outcomes. It's difficult to bridge the perception gap. Another choice is to learn the task on the physical structure directly.

1.2 Objective

Although this has been exhibited effectively in mechanical grasping, it is hard to apply this way to deal with velocity assignments because of the difficulties of consequently reconfiguring the investigations and social occasion information consistently. Furthermore, any fall while learning has the potential to harm the robot. Learning in

recreation is thusly more appealing for headway exercises since it is speedier, less expensive, and more secure. We present a thorough learning structure for deft movement in this venture, in which control strategies are created in recreation and executed on genuine robots. There are two main challenges:

1) Learning locomotion policies (LLP) that can be regulated.

2) The policies are being transferred to the physical system.

1.3. Motivation

On a quadruped robot, we test our method with two locomotion tasks: tossing and galloping. We exhibit that exceptionally dexterous motion steps can arise consequently utilizing profound RL. We also show how our framework allows users to easily select the locomotion style.

When we compare our learned gaits to expert-crafted gaits at the same running pace, we discover that our learned gaits are much more energy efficient. We show how we can effectively execute policies trained in simulation to the physical system using an effective dynamics simulation and robust control policies. The main contributions of this project are:

1) A full learning framework for agile locomotion is proposed. It gives users complete control over the learned policies, ranging from completely restricted toward a user specified gait to completely trained from scratch.

2) We demonstrate how a number of methods can be used to close the fact gap and perform systematic tests of their efficacy.

3) We show that dexterous velocity walks like jogging and dashing can be prepared consequently, and thusly these strides can be utilized on mechanical frameworks with no extra actual model preparing.

1.4. Reinforcement Learning

Support learning (SL) is a region of AI stressed over how smart experts should take actions in an environment to grow consolidated honor. Backing learning is one of three fundamental AI ideal models, nearby managed learning and independent learning.

Backing taking in contrasts from coordinated learning in not needing named input/yield sets be presented, and in not needing dangerous exercises to be explicitly amended. Maybe the consideration is on finding a concordance between examination (of odd space) and misuse (of current data).

The situation is ordinarily communicated as a Markov decision cycle (MDP), considering the way that various help learning estimations for this setting use dynamic programming systems. The essential differentiation between the old-style dynamic programming systems and backing learning estimations is that the keep going don't expect data on a cautious mathematical model of the MDP and they target tremendous MDPs where clear procedures become infeasible.

Due to its distortion, support learning is packed in various orders, similar to game speculation, control theory, exercises research, information speculation, amusement-based upgrade, multi-expert systems, swarm information, and experiences. In the exercise's assessment and control composing, support learning is called unpleasant incredible programming, or neuro-dynamic programming. The issues of income in help learning have in like manner been amassed in the theory of ideal control, which is worried generally with the presence and depiction of ideal plans, and computations for their precise estimation, and less with learning or assessment, particularly without a mathematical model of the environment. In monetary matters and game theory, support learning may be used to explain how concordance may arise under restricted acumen.

A fundamental support learning specialist AI cooperates with its current circumstance in discrete time steps. At each time t, the agent receives the current state s_t and reward r_t. It then chooses an action a_t from the set of available actions, which is subsequently

sent to the environment. The environment moves to a new state s_{t+1} and the reward r_{t+1} associated with the transition is determined. The goal of a reinforcement learning agent is to learn a *policy*:

$$\boldsymbol{\pi : A * S \rightarrow [0,1], \pi(a,s) = \Pr(a_t = a \mid s_t = s)}$$

Chapter 2

LITERATURE SURVEY

LITERATURE SURVEY

2.1. Background

In this consider a few investigate papers containing the usage of reinforcement Learning and Information are looked into to induce a generally thought like how to bargain with the policy gradient data, which calculations ought to be connected, how the accuracy can be made strides to form a proficient framework. Underneath are the reviews of a few considered containing strategy and technique utilized together with the conclusion.

2.2. Review in quadrupedal robots

First, in robotics leg locomotion, both organized and unstructured, in non-flat terrain is a major challenge. In such settings, autonomous operation necessitates highlighting multi-contact motion preparation concerns and management. ANYmal, for example, is a four-legged robot [1], must be able to pick suitable footholds about the terrain though maintaining a constant sense of balance in order to navigate complex environments autonomously. This research focuses on the issue of using proprioceptive and exteroceptive detecting to prepare and execute foothold sequences in restricted non-flat terrain allowing quadrupedal mobility.

Walking dynamically on non-flat geography, computation of both continuous state-input pathways, such as in the motion, is needed of the foundation, and differentiated decision variables, including which surface to make contact with and when. Model-based methods, even those that combine other heuristics with probabilistic optimization methods [2], [3] [4], have been used to devise motions for both foundation and the feet. While a few of the previously described methods [2], [3] can solve these issues for both discrete and continuous variables, they are still too much time consuming to be used on

the platform. Thus, Kino static way of approach [4], [5] had thus far been effective in performing online foothold preparation. In attempt to make the quest for important footholds easier, most methods use some form of terrain parameterization or classification [4], [6], [7].

Attempting to deal with the stochastic problem that emerges from the large number of possible terrain contact configurations, is one of the most difficult aspects of multihold planning for multi-limb structures. Using samplingbased analytic techniques [7], [8] or assuming the movement patterns [4], [5] are two common solutions. There are also works [5], [6], [9] that incorporate optimization and random samples approaches. However, since they appear to ignore the dynamics of the system, these usually decouple the acquisition of footholds from the localization of ground movements, remaining Kino static.

Machine-learning methods have also been used in some works to aid terrain descriptions [8], [10], [11]. Others have used Reinforcement Learning (RL) [12], [13] to achieve locomotion that is conscious of the terrain from start to finish. However, using the latter also comes with a number of drawbacks, including: (1) How to remove unwanted emergent behavior while retaining emergent activity that is advantageous and (2) Decrease the total complexion of the study and efficiently trained policies.

We suggest a new approach for to cross complicated non-flat terrain, quadrupedal frameworks are used that incorporates model-free and state-of-the-art based model strategy. Our formulation consisting of: (1) Aware terrain coordinator that produces foothold and base motion sequences that guide the robot in the direction of a goal approaching and (2) A foot base hold and framework acceleration control system that carries out the above sequences also preserving equilibrium and coping through disturbances. Also the planning and the controller were implemented as of deterministic

policies that are determined using NN function approximation and programmed using cutting-edge Deep Reinforcement Learning (DRL) algorithms.

A new approach for training Kino dynamic movement planners that uses a Trajectory Optimization (TO) technique to determine the feasibility of transitioning between separate support phases that used a coarse simulation environment. This eliminates the need to have a planner system to communicate with both mobility and a controller during preparation, allowing the two policies to learned separately and reducing overall elemental complexity significantly. The easy formula for creating a dynamic gait control system that depend solely on proprioceptive sensing and use target footholds as references. This allows us to build a controller system that fully utilizes the robot's dynamic behavior to monitor arbitrary target footholds, regardless of the planner that produced them.

Using a physics simulator, we test the efficiency of our system in a variety of difficult locomotion scenarios and report the results. Our experiments indicated that the planner is capable of generalizing over a variety of terrain types and that the control system can track relative footholds while keeping the robot balanced. Furthermore, we compare our system to a state-of-the-art based model approaching [4] to demonstrate its benefits.

2.3. Summary of the Chapter

In this chapter different thinks about were looked into on the premise of the strategy utilized, device utilized, calculation utilized and subsequently finishing with the conclusion. These thinks about made a difference the creators to urge arranged for the up-and-coming circumstance.

Chapter 3

ANALYSIS AND DESIGN

3. ANALYSIS AND DESIGN

Before developing this system, the analysis of the requirements and designing of the model is conducted so as to make a virtual structure of the system and to determine what is required by the system.

3.1. Introduction

End-to-end reinforcement learning is a promising approach to enable robots to acquire complicated skills. However, this requires numerous samples to be implemented successfully. The issue is that it is often difficult to collect the sufficient number of samples. To accelerate learning in the field of robotics, knowledge gathered from robotics engineering and previously learned tasks must be fully exploited. Specifically, we propose using a sample-efficient curriculum to establish quadrupedal robot control in which the walking and turning tasks are divided into two hierarchical layers, and a robot learns them incrementally from lower to upper layers.

To develop such a curriculum, two core components are designed. First the fractal design of neural networks in reservoir computing is aimed at allocating the tasks to be learned to respective modules in fractal networks. This allows mitigating the problem of catastrophic forgetting in neural networks and achieves the capability of continuous learning. The second task includes hierarchical task decomposition according to robotics knowledge for controlling legged robots. Owing to the combination of these two components, the proposed curriculum enables a robot to tune the lower layer even when the upper layer is optimized.

As a result of implementing the proposed design, we confirm that a quadrupedal robot in a dynamical simulator succeeds in learning skills hierarchically according to the given curriculum, starting from moving legs and finally, walking/turning, unlike the considered conventional curriculums that are unable to achieve such results.

3.2. Requirement Analysis

This section determines what type of software and hardware is required for the development of the system as mentioned below:

3.2.1. Software Requirement

- Ubuntu 16 or above/ Windows 7 or above
- Python
- Packages: Numpy, Sklearn, Matplotlib, Pandas, roboschool, Rviz
- Open AI Rex GYM
- PyBullet
- TensorFlow
- MeshLab

3.2.2. Hardware Requirement

- 4 GB RAM or above
- 1 TB ROM or above
- Processor Speed 1.4 GHz or above

3.3. Detailed Design

In this unit various diagrams are stated for better understating of the study.

3.3.1. Waterfall Model

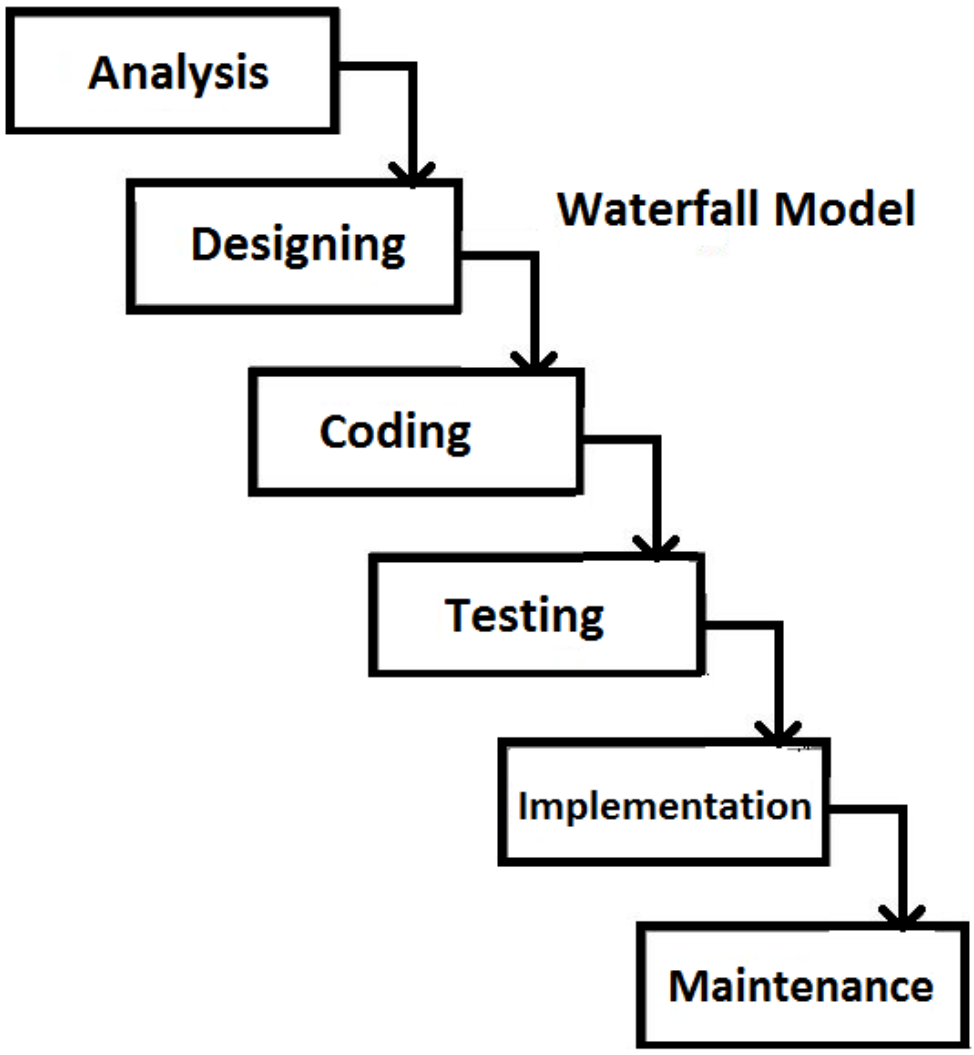


Figure- 3.3.1. shows how the workflow of project is processed by the system i.e., the step-by-step blocks of the data.

3.3.2. Data-Flow-Diagram

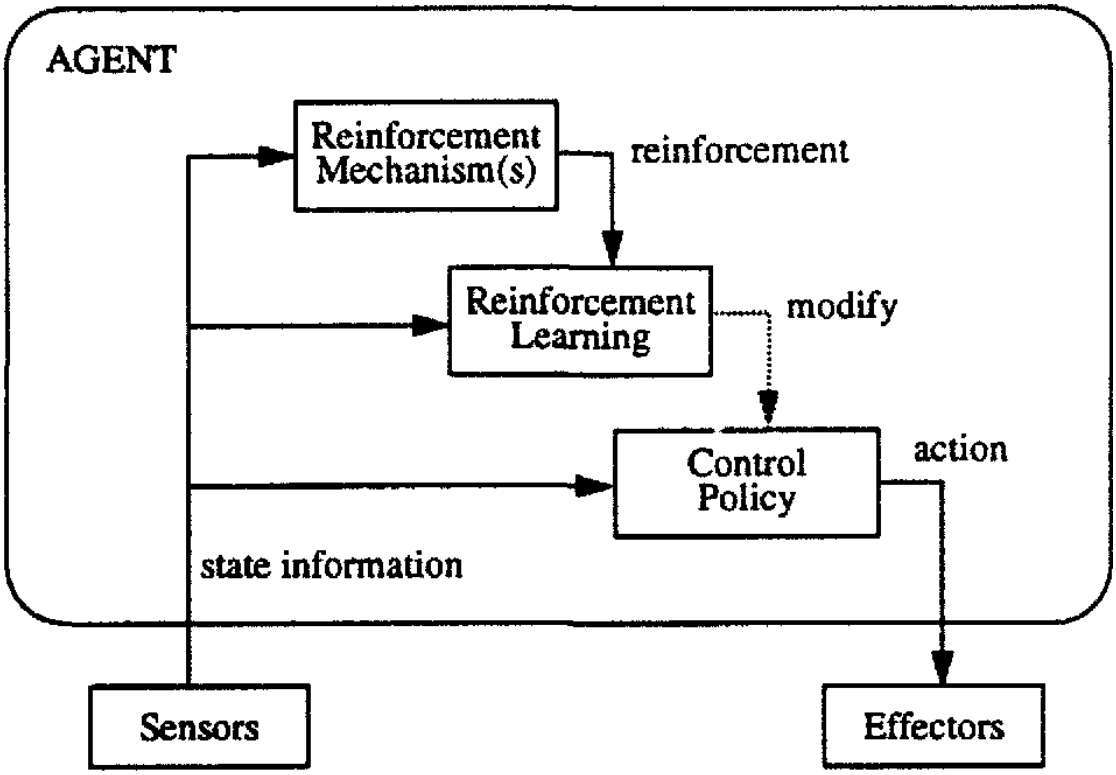


Figure- 3.3.1 Data-Flow-Diagram

Figure- 3.3.1 shows how the data is processed by the system i.e., the step-by-step flow of the data.

3.3.3. UML Class Diagram

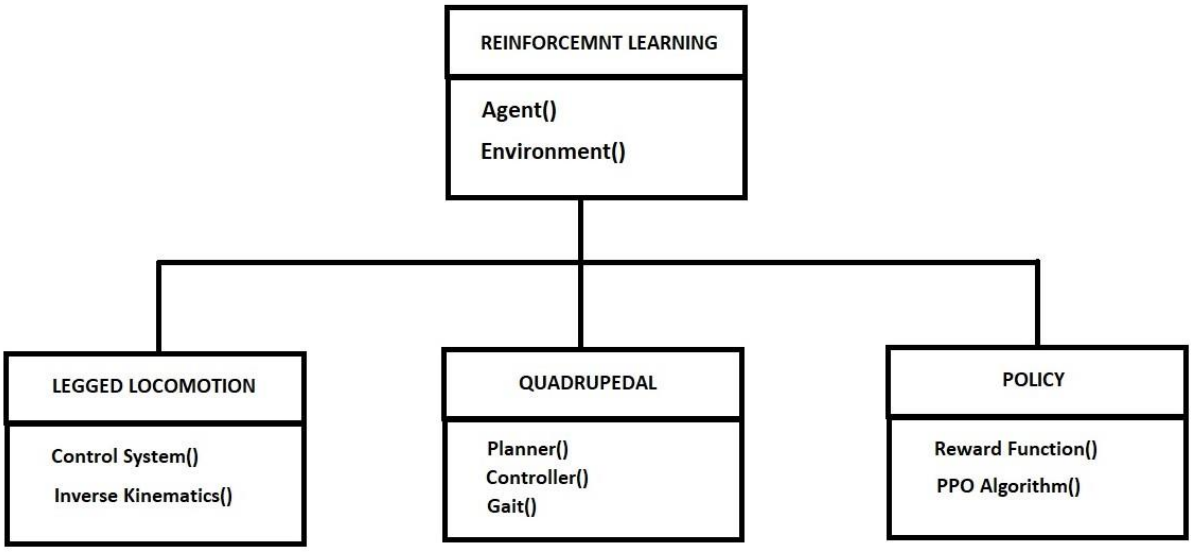


Figure- 3.3.3 UML Class Diagram

3.3.3. Control Sequence Diagram

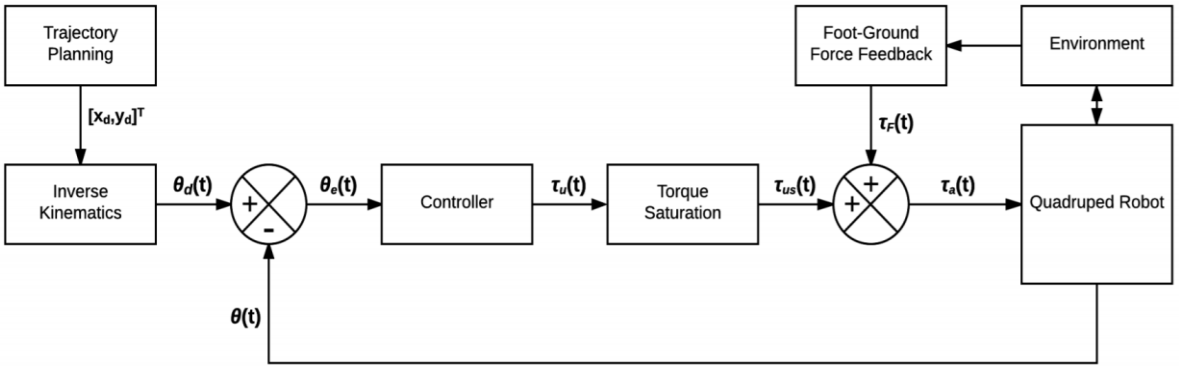


Figure- 3.3.4 Sequence -Diagram

3.4 Summary_of_the_Chapter

This chapter defines the several use cases and classes essential for the completion of the study by explaining various diagrams related to it.

Chapter 4

IMPLEMENTATION DETAILS

4.1 Introduction

Below are the detailed description of the various requirements tools, techniques, libraries and testing required for the study.

4.1.1. Functional Requirements

- **Terrain:** Locomotion of legged robots is a challenging multidimensional control problem. It requires the specification and coordination of motions in all of the robots' legs while accounting for factors such as stability and surface friction.
- **Inverse Kinematics:** For the purposes of this paper, we adopt forward speed as the sole objective function. That is, as long as the robot does not actually fall over, we do not optimize for any form of stability.
- **Policy Gradient:** We formulate the problem as a policy gradient reinforcement learning problem by considering each possible set of parameter assignments as defining open-loop policy that can be executed by the robot. Assuming that the policy is differentiable with respect to each of the parameters, we estimate the policy's gradient in parameter space, and then follow it towards a local optimum. Since we do not know anything about the true functional form of the policy, we cannot calculate the gradient exactly. Furthermore, empirically estimating the gradient by sampling can be computationally expensive if done naively, given the large size of the search space and the temporal cost of each evaluation.

4.1.2. Non-Functional Requirements

- **Languages and Reasoning:** Much work focuses on capturing NFRs in visual modeling languages, sometimes with an underlying metamodel and semantics,

facilitating automated qualitative and quantitative methods to support decision making. Usually, approaches allow users to use NFRs to select among possible alternative functional requirements.

- **Runtime, Adaptation, and Evolution:** NFR approaches were extended to consider a requirements-based view of runtime system operation, where functional and quality requirements could be monitored at runtime, based on data from the running system. Work in this area went further to consider requirements-based runtime adaptation, e.g., a certain quality aspect is not sufficiently satisfied at runtime, thus the system will evolve and adjust to try to gain better performance or quality, all while considering quality trade-offs.
- **Linking Data to Quality:** A related line of work uses an adaption of common requirements notations to link business data to organizational goals, including qualities, allowing for continuous goal-based business intelligence. More recent work focuses on the design of data analytic systems for business, which may include RL algorithms. This work focuses on finding designs which fit domain-specific analytic questions, considering aspects of quality performance for various RL options.

4.2. Tools Used

Various tools are used for this study i.e. Python 3.7 & Open AI Gym.

4.2.1. Python 3.7

Python is a scripting language which has a lot of inbuild libraries to do the complex computation of carious Machine Learning algorithm is seconds. Libraries like numpy, sklearn, pandas, flask, jolly are used.

4.2.2. Open AI Gym

Open AI Gym is used to train the models using Python as it provides good kernel interface for python libraries. The users can also set their own environments so that a certain project library will not interfere with one another.

4.2.3. Libraries

4.2.3.1. Numpy

Numpy is a package in python which is used to execute powerful calculation on multidimensional array which usually are datasets.

4.2.3.2. Sklearn

Sklearn in python deliver support to several machine learning algorithms like splitting the data into training and testing, accuracy, precision, recall, confusion matrix and so on.

4.2.3.3. Pandas

Pandas in python contains of data examination tool and data structures for the dataset.

4.2.3.5. Pickle

Pickle in python makes a pickle record of the prepared show so as to utilize it on web apps in such a way that it can test the information entered by the client.

4.2.3.6. Tensorflow

Tensorflow in python permits the input data into the pickle file for the testing.

4.2.3.7. Matplotlib

It is a package in python which is used to plot the graphs for the accuracy.

4.3. METHODOLOGY

4.3.1 Algorithm Description

Prior to RL, an agent could learn an optimal strategy for obtaining the maximum return (i.e., the number of rewards) from a setting. The following tuple is used to represent the Markov decision process (MDP): (S, A, R, p_0, p_T, γ). The MDP method can be summarized as follows: The agent begins by randomly obtaining the initial state $s_0 \in S : s_0 \sim p_0 (s_0)$. The agent chooses action $a_t \in A$ by the policy π above the present state $s_t \in S$: $a_t \sim \pi(a_t \mid s_t)$ at time stage $t \in$ N. According to the transfer probability model $p_T : s_t{+}1 \sim p_T (s_{t+1} \mid s_t , a_t)$, firstly a_t is introduced in the environment and the agent continues to receive the successor state $s_t{+}1$. At the very same time, the agent receives a $r_t \in R$ reward obtained from the reward function: $r_t = r(s_t , a_t , s_{t+1})$. The agent seeks to optimize the return $R_t = \sum^{\infty}_{k=0} \gamma^k r_{t+k}$ because of the factor of discount $\gamma \in [0, 1)$ by enhancing the policies effectiveness to $\pi*$. Our use of the actor–critic methodology with the student-t policy and regular temporal difference training at this stage. In this case, the function of value is also extended to include the

expected value of the return from the present state: $V(s) = E[R_t \mid s_t]$. The value function is used in such algorithm for the ds (dimensional state) space and the da (dimensional action) space $V(s) \in R$ and policy's three conditions are: location $\mu(s) \in R^x_y$, scale $\sigma(s) \in R^x_{y+}$ and degrees of freedom $v \in R_+$, using function approximations, they must be approximated.

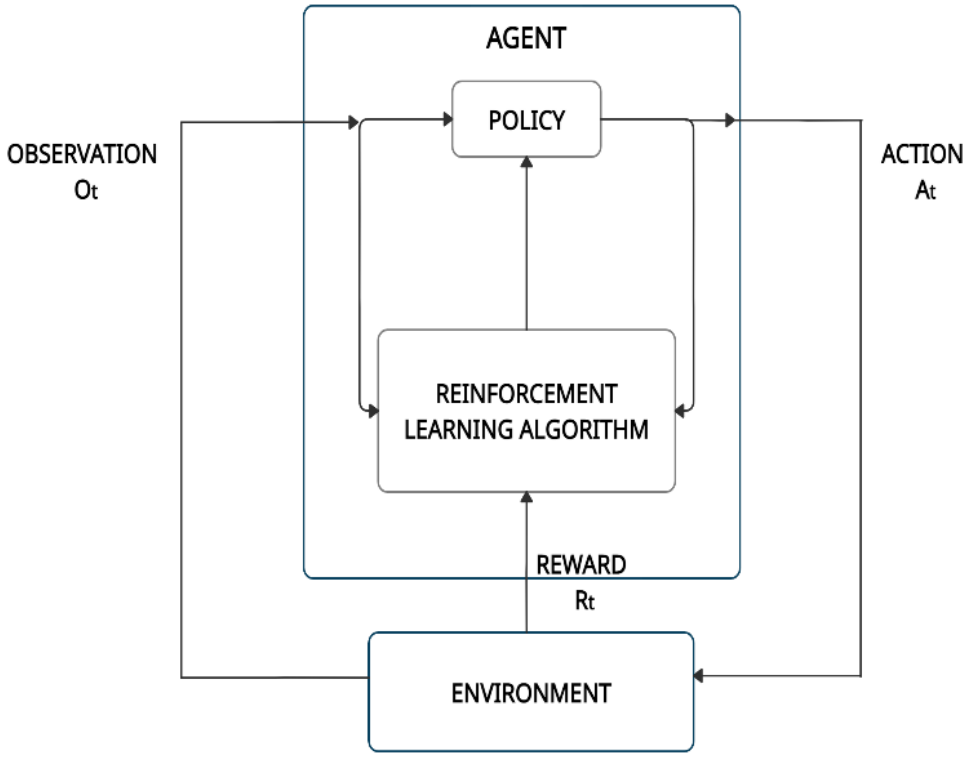


Fig.4.3.1. Reinforcement learning framework: an agent communicates with a climate by playing out an activity inspected from a strategy over a present status; the climate at that point returns the following state dependent on elements and the compensation to be augmented.

The properties and tests from the dataset are pre-processed to correct the invalid values after that different Machine Learning calculations are connected on the corrected data to create the output as appeared.

4.3.2 Proposed Architecture

4.3.2.1 Planning of Gait:

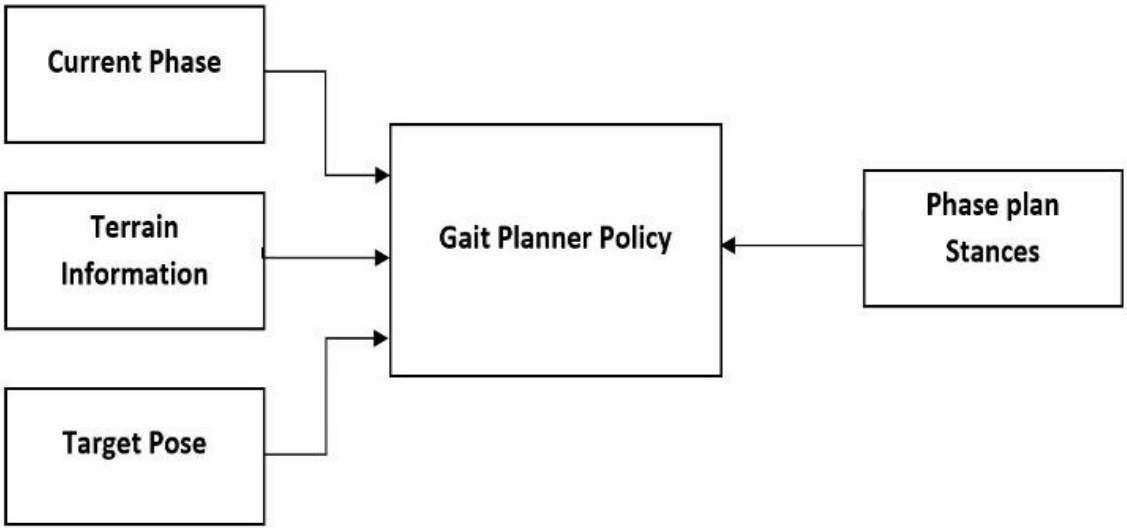


Figure. 4.5.1 Planning of Gait: represents as a localized terrain-aware planner that generates a finite succession of support periods, i.e., a phase plan, using both exteroceptive and proprioceptive measures.

The Planning of Gait generates the aforementioned step plan by sequentially querying the planning policy $\pi_{\theta m}$. When proposing phase transitions, we formulate the Markov Decision Process in order to train $\pi_{\theta m}$ using RL, with the aim of ensuring that resulting policies learn to respect the robot's Kino dynamic properties and limits and also touch constraints. We can train $\pi_{\theta m}$ to gather a distribution on top of phase transitions that is not only feasible but also maximizes locomotion efficiency using the MDP that results.

4.3.2.2 Controlling of Gait:

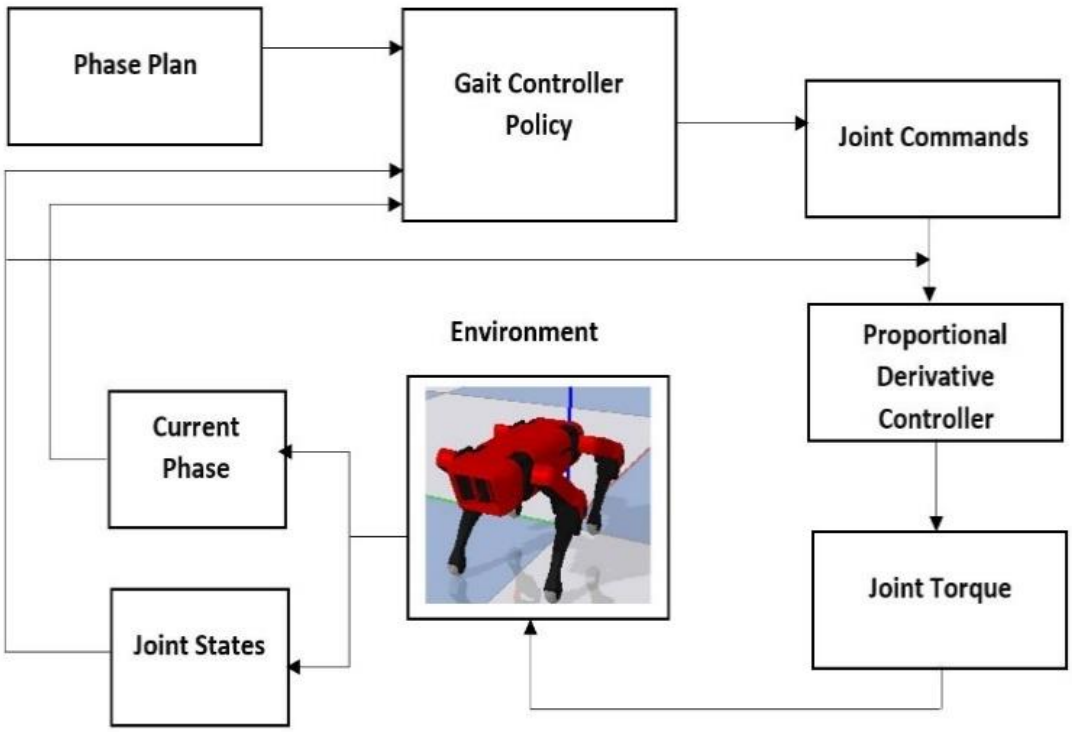


Fig. 3. Controlling of Gait: represents as the hybrid movement of planning system and controller that output's joint location references using only proprioceptive sensing and the aforementioned step plan. Finally, a combined PD controller (with null goal joint velocity) computes joint torques using these joint position comparisons.

4.3.3 Policy Description

While gaining without any preparation may eliminate the requirement for human mastery and, at times, improve effectiveness, having command over the learned approaches is basic for mechanical applications. For instance, we should determine walk data.

Therefore, we split the movement regulator into two segments: an open circle part that empowers the client give the reference directions, and a reaction segment that changes the leg presents dependent on the examination.

$$a(t, o) = a^*(t) + \pi(o)$$

where π(o) is the feedback component and a*(t) is the open loop component, which is normally a periodic signal. Users can quickly communicate their optimal gait using an open loop stimulus, and learning will take care of the rest, such as core stability, which is time-consuming to design individually. This hybrid policy is a broad formulation that provides users with a wide range of control options. It can be modified at any time, from completely user-specified to completely learned from scratch.

We can draw both the lower and upper lines of π(o) to nothing on the off chance that we decide to utilize a client indicated strategy. Set a*(t) = 0 and give the criticism segment π(o) an expansive yield range on the off chance that we need an approach that is gained without any preparation. We can screen how much client control is applied to the gadget by fluctuating the open circle signal and the yield bound of the input variable.

We showed two examples in this paper: figuring out how to dash without any preparation and figuring out how to run with a client gave manage. We use a neural network to represent the feedback element π and Proximal Policy Optimization to solve the POMDP above.

4.3.4 Reward Function

Reinforcement learning optimizes a policy $\pi : O \rightarrow A$ that significantly increases the anticipated returns (accumulated rewards) R. A partial observation $o \in O$, rather than a complete state in $s \in S$, is observed at each control level.

We create a reward mechanism that motivates the agent to learn habits such as hitting the target position as quickly as possible, approaching the destination as fast as possible, reducing kinetic effort throughout phase transitions by avoiding long stances phases. The overall R_M i.e. reward function is composed of multiplicative and additive expressions as follows:

$$R_M\,(S_M,t,\ S_M,t+1) = r_m \cdot r^2_h \cdot r_k - r_c$$

where r_m awards the agent with the aim of moving the baseline foot hold position near to the target and inflict a penalty on the agent for driving it away, r_h penalizes the robot for not approaching the desired position, r_k inflict a penalty on for driving the gap between the marginal footholds under the shoulder and r_c inflict a penalty on for not raising a foot on the top of several steps, as a result, exploration is encouraged and such policies are kept from being trapped into the maximum of maintaining a constant stance.

In addition, we'd like to call attention to a few characteristics of the multiplicative word in the reward function above. This term produces a sanction that is low when r_p is less, i.e. at the start of session and high when r_p is more, i.e. at the end of the session, results in an automatic ascend of the total multiplication terms. We found that utilizing these multiplicative prizes produces advantageous slopes across all preparation emphases because their attributes are never too big to stifle experimentation and really not that small to have no impact.

4.3.5 Algorithm – PPO

PPO is a policy-based algorithm. PPO can be used in either discrete or continuous action space settings. Because of its easy usability and high performance, PPO has now become OpenAI's default reinforcement learning algorithm. To update the policy by maximizing the PPO objective is given by:

$$\boldsymbol{\theta_{k+1} = \arg max_{\theta}} \frac{1}{|D_k|T} \sum_{\tau \epsilon D_k} \sum_{t=0}^{T} \boldsymbol{min}\left(\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta}}) \right)$$

Fit Value Function is given by:

$$\emptyset_{k+1} = \arg max_{\emptyset} \frac{1}{|D_k|T} \sum_{\tau \epsilon D_k} \sum_{t=0}^{T} (V_{\emptyset}(s_t) - R_t)^2$$

4.4 Summary_of_the_Chapter

This chapter describes the various tools, methodology, algorithms used in this study along with the various system testing methodologies applied to it.

Chapter 5

EXPERIMENTAL RESULTS AND TESTING

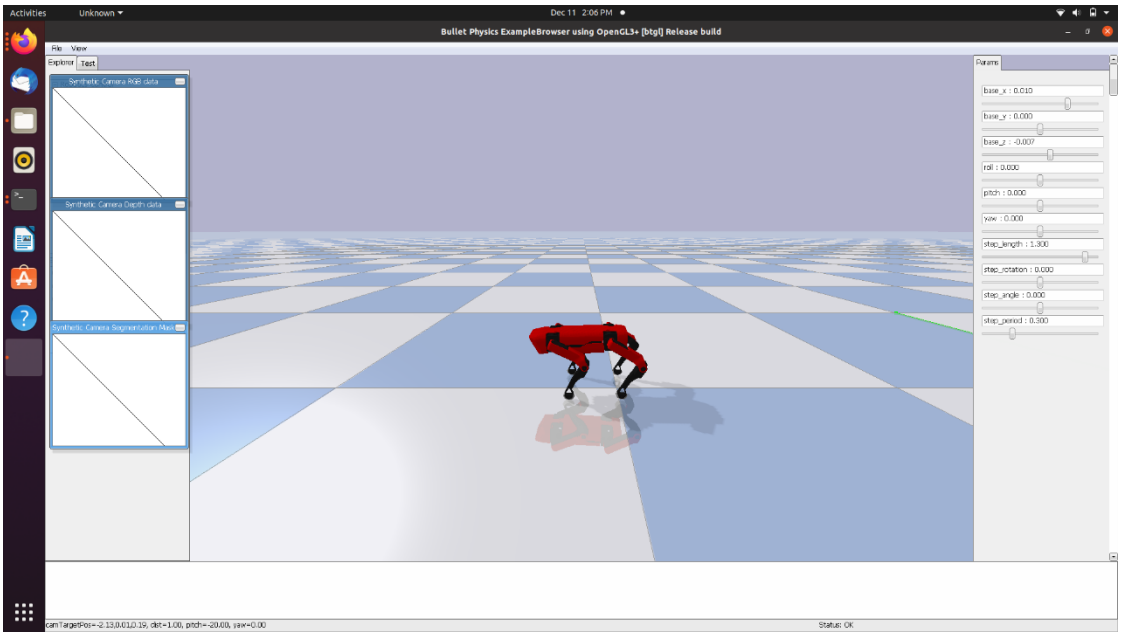
5.1. INTRODUCTION

Following that, we created a collection of terrain-based scenarios for priming and evaluating the controlling Gait and planning Gait agent's in order to evaluate our approach. The first and simplest scenario involves an infinite flat surface known as Flat World, which we infer to create an output and behavior baseline.

The terrain scenario is Temple-Ascent, which is a hybrid terrain that includes both flat and non-flat terrain areas. The RaiSim multi-body physics engine was used to create the MDP environment of the controlling Gait. All RL calculations were executed utilizing the TensorFlow3. For applying the algorithms on the quadrupedal model, Open AI Rex GYM tool is used. From above-mentioned algorithms, the outcomes concluded are: -

5.1.1 Simulations of quadruped in complex terrains

1) Flat Plane Terrain: Basic approach towards simulation



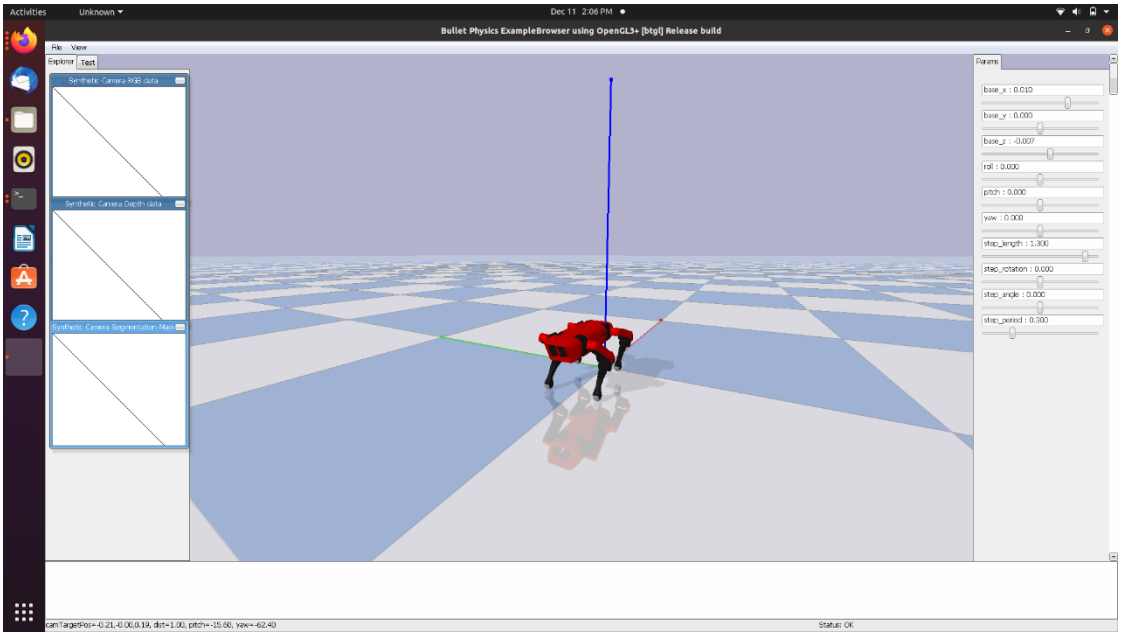
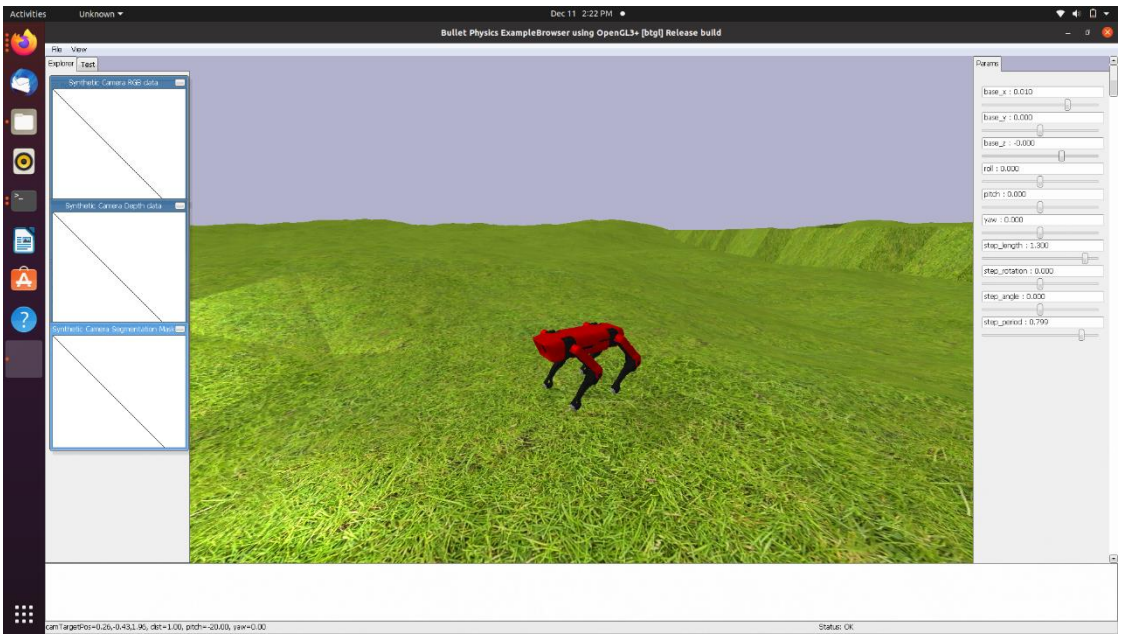


Fig. 5.1.1.1. Flat Plane Terrain: Basic approach towards simulation

2) Hills non-flat terrain: Outdoor non-flat grass surface



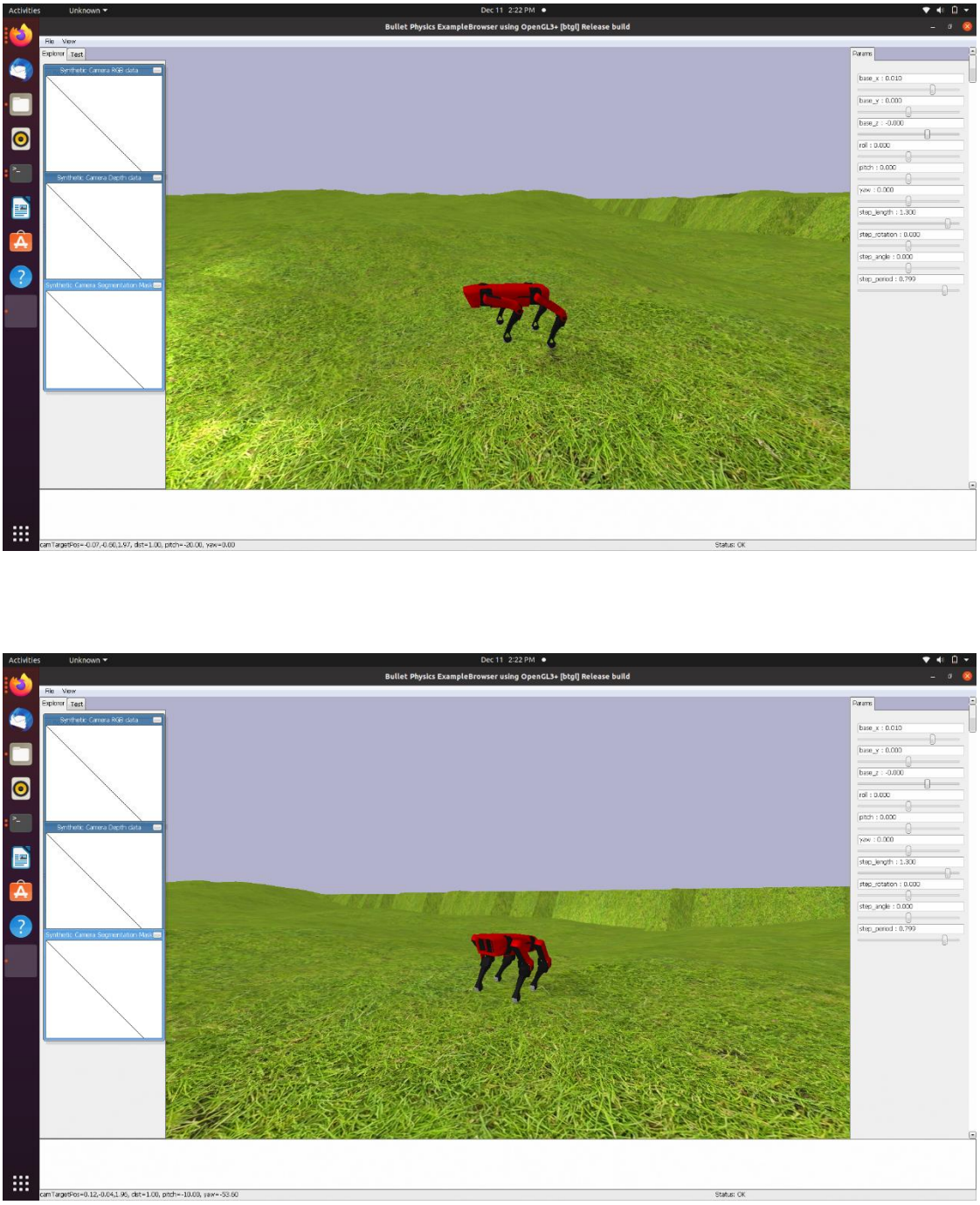
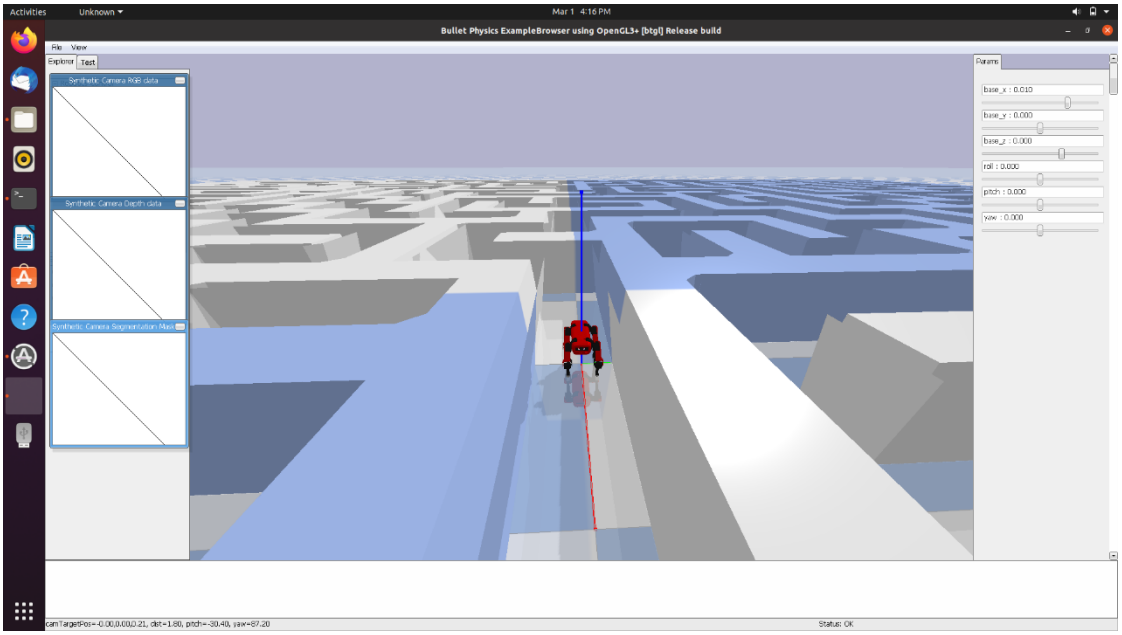
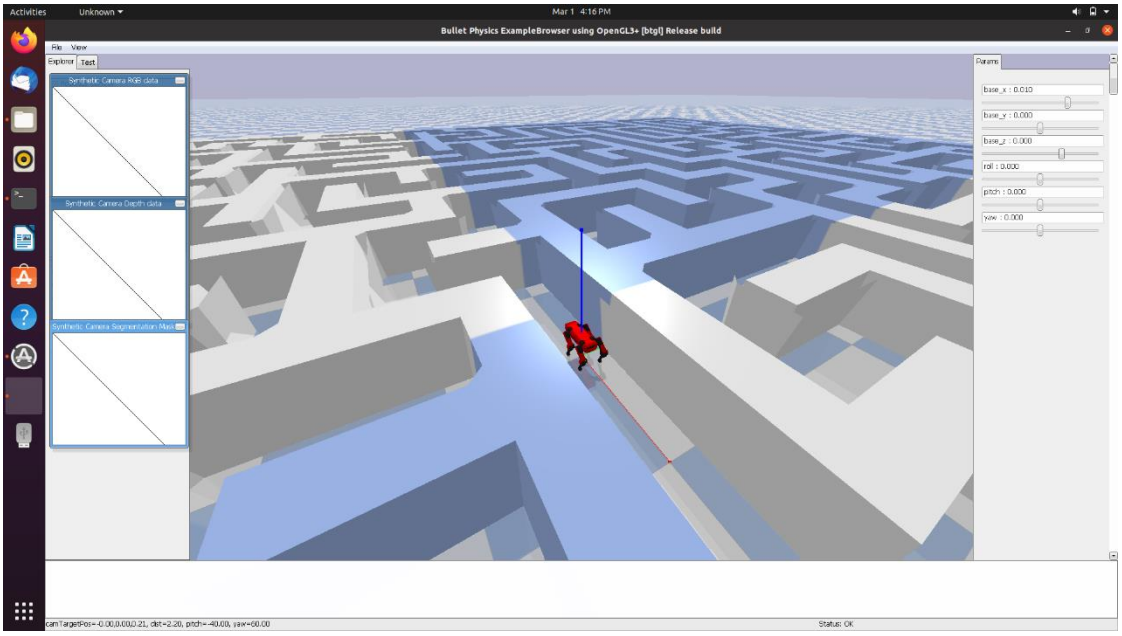


Fig. 5.1.1.b. Hills non-flat terrain: Outdoor non-flat grass surface

3) Maze flat terrain: Obstacles based approach for movement





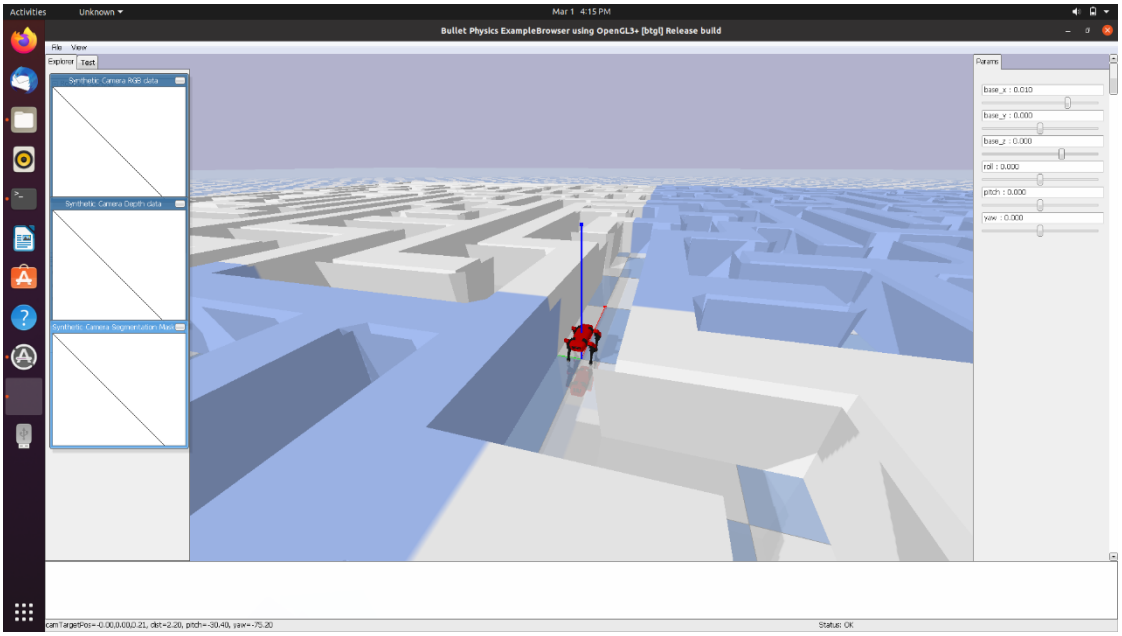
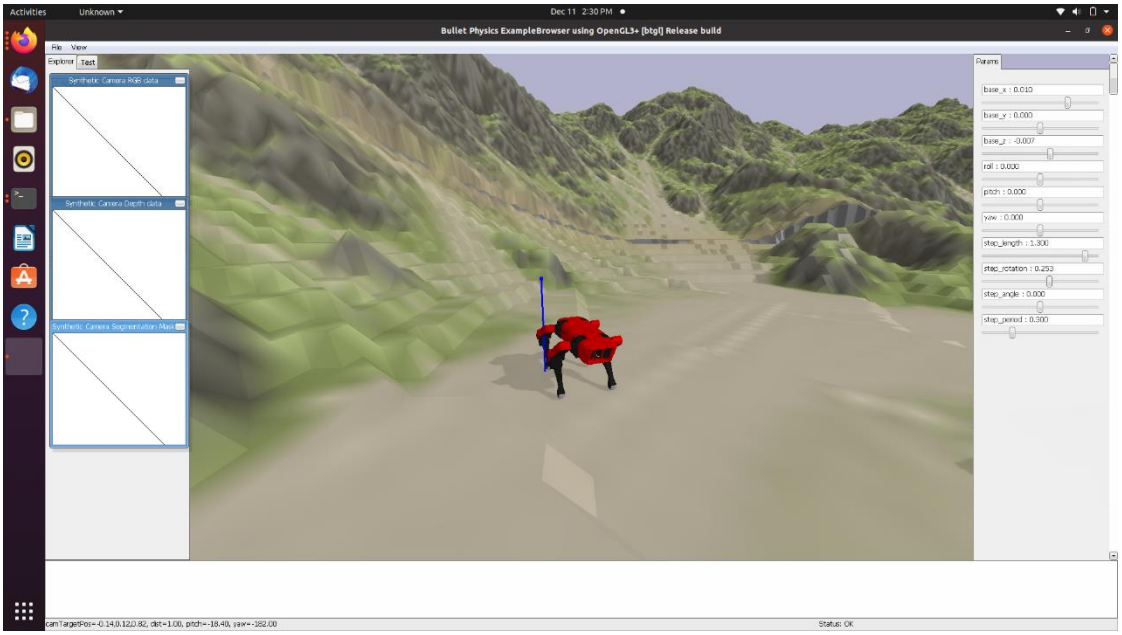


Fig. 5.1.1.c. Maze flat terrain: Obstacles based approach for movement

4) Mountain non-flat terrain: Outdoor extreme complex non-flat surface



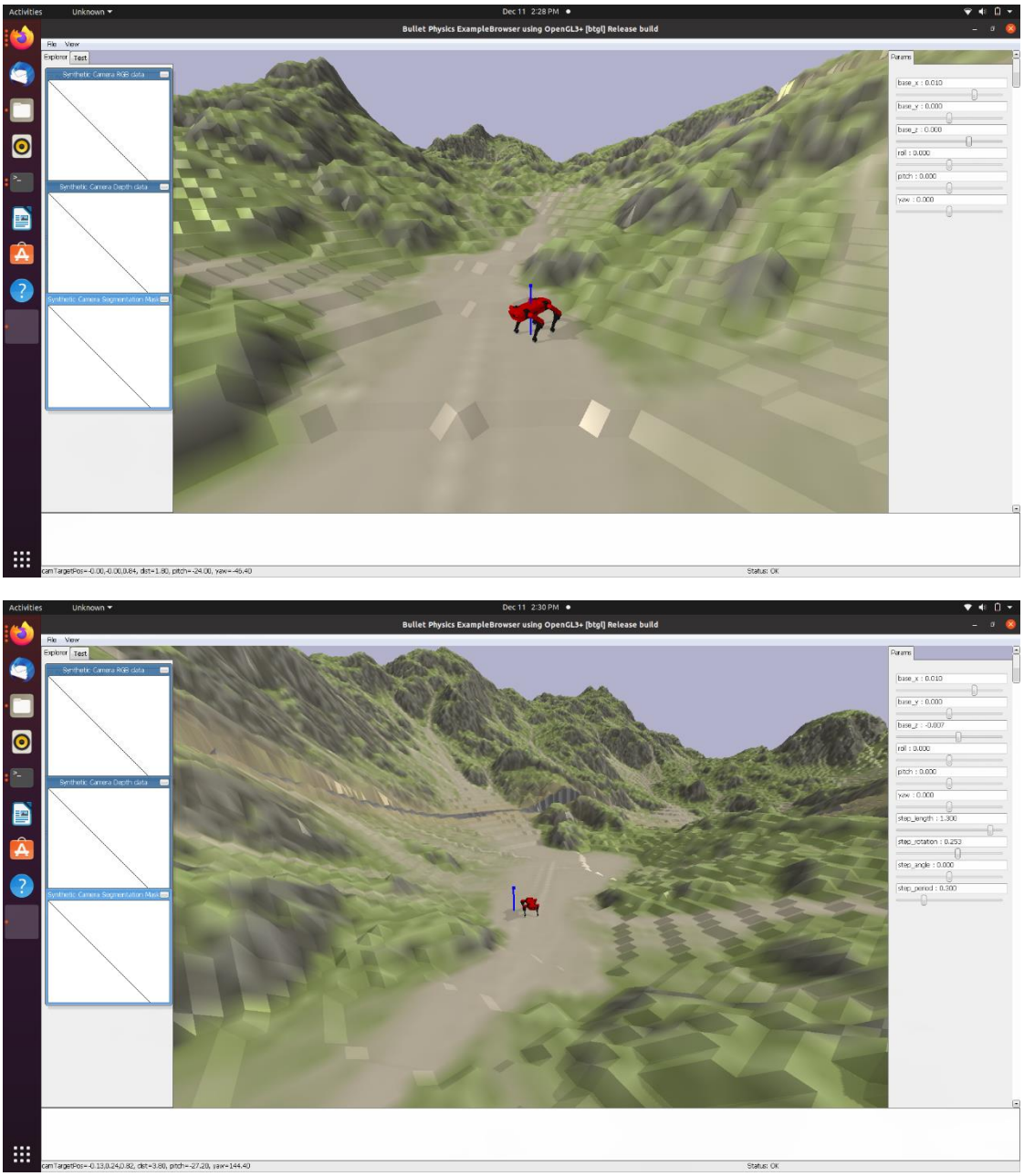


Fig. 5.1.1.d. Mountain non-flat terrain: Outdoor extreme complex non-flat surface

5) Sand non-flat terrain: Outdoor complex sand non-flat surface

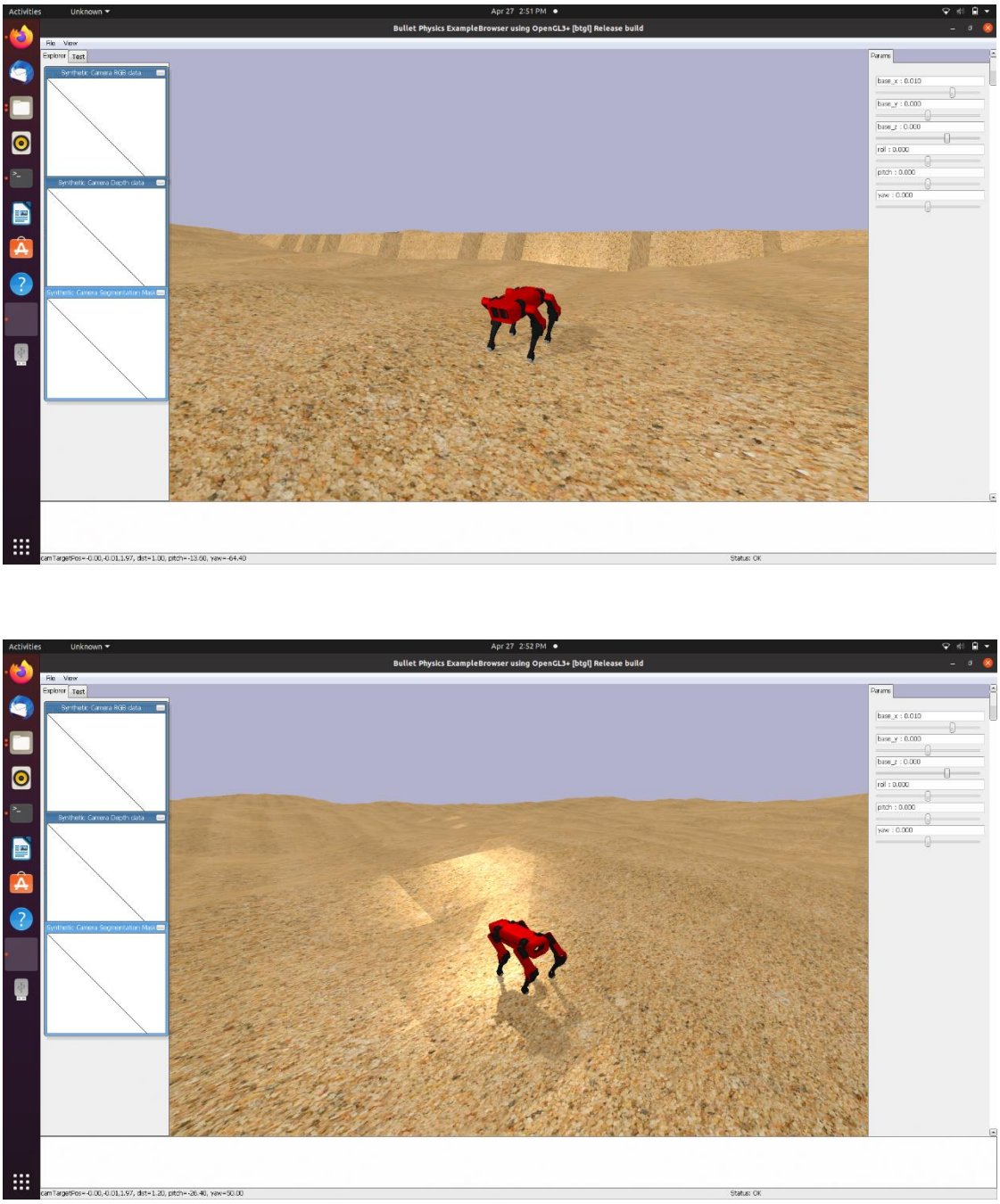
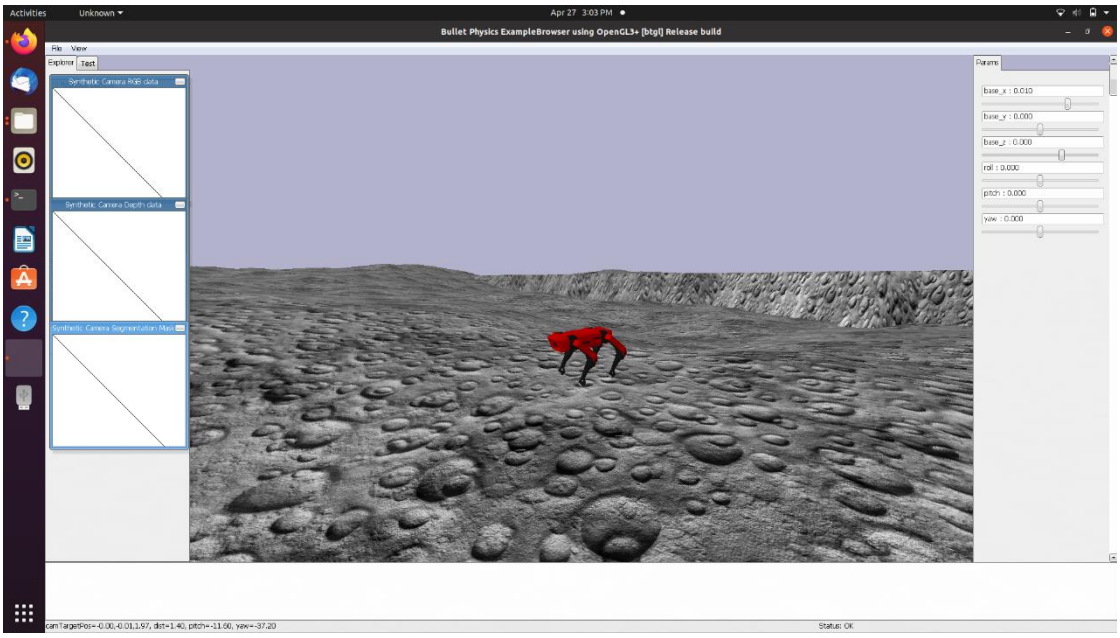


Fig. 5.1.1.e. Sand non-flat terrain: Sand particles-based approach for locomotion

6) Space flat terrain: complex flat surface space scenario



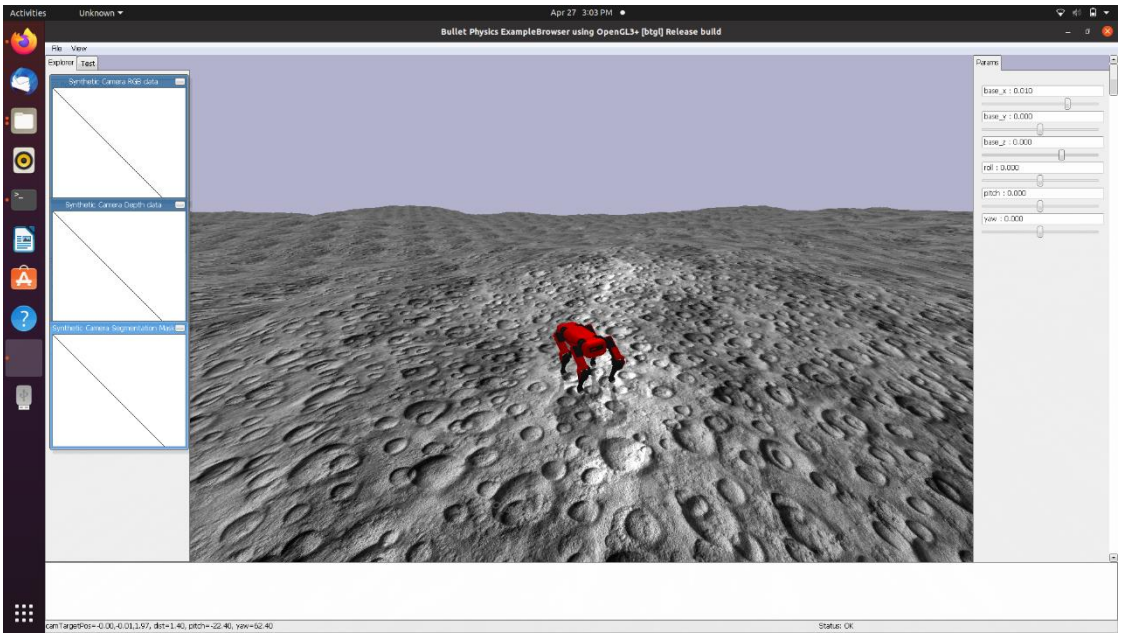


Fig. 5.1.1.f. Space flat terrain: Space hard flat complex surface based terrain

5.2. Graphs & Maps

5.2.1. Graph of cumulative reward function

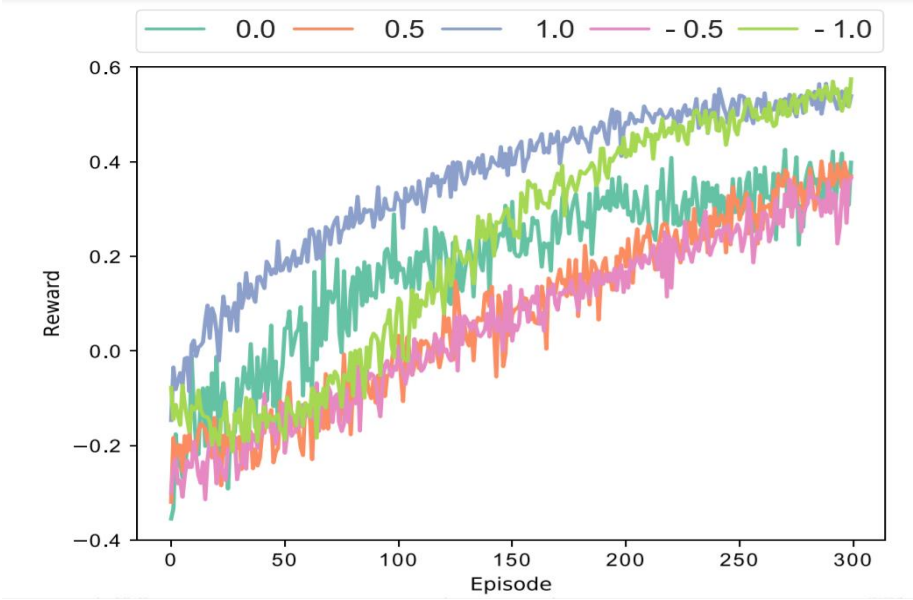


Fig. 5.1.2. Reward: In order to maximize reward, the agent must walk with respect to the set of episodes.

The proposed system has the three capabilities of reusing the knowledge already learned: modularity for continuous learning; transferring knowledge about leg control; the hierarchy suitable for locomotion. To fully exploit these capabilities, we need to establish a curriculum for learning locomotion. First, a quadrupedal robot needs to learn how to control a leg on the ground as a part of the actuation layer. The normalized phase deviation ω is defined $\{0, 0.5, 1.0, -0.5, -1.0\}$ in order. At this point, other legs are fixed at the initial states to reduce the randomness of the environment. Owing to continuous learning, we can avoid random commands for ω, which would require considerable time and would be inefficient. A robot can focus on learning the skills for constant ordered commands, and interpolation is naturally expected by preserving all skills for the given commands appropriately. After learning this step, the knowledge about control of a leg is transferred to other legs to complete the actuation layer.

5.2.2. Graph of lateral body posture & torqure

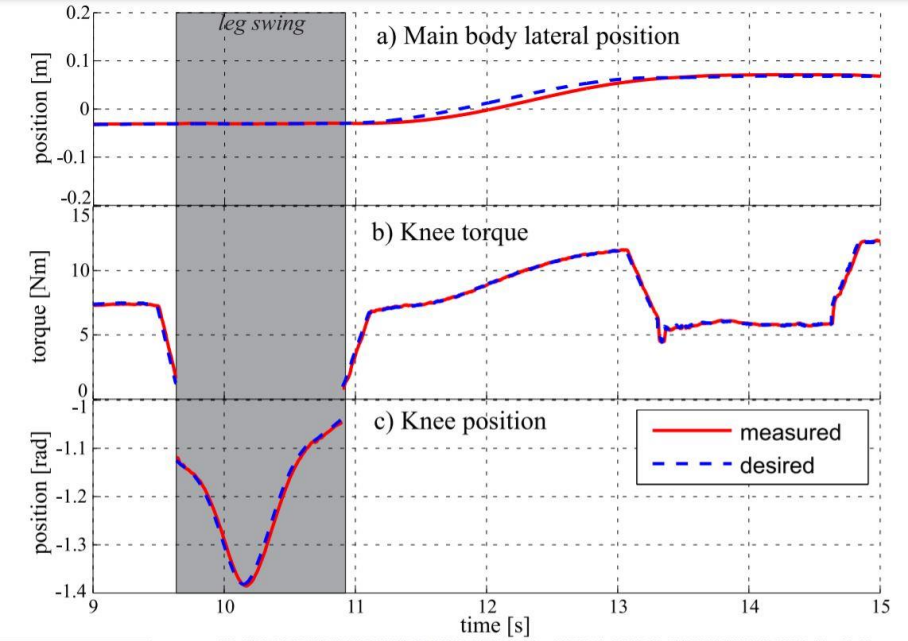


Fig. 5.1.3 Position and torque measured with respect to time

As illustrated in above figure, joint torques and positions are followed very accurately during the entire gait cycle and hence also the base position can be accurately moved according to the preplanned trajectory.

It is important to know that the latter follows only from virtual model control (task space control) at the base and without any joint position or impedance regulation. By applying a classical planner forward locomotion results in a very smooth and almost straight line of the base as illustrated.

Chapter 6

CONCLUSION AND FUTURE -WORK

6.1 CONCLUSION

We've shown that RL can be utilized to show robots coordinated velocity all alone. We have shown a total structure that utilizes profound support learning in virtual conditions and can gain without any preparation or permit the client to coordinate the learning cycle. We effectively conveyed the regulators prepared in reproduction on genuine robots utilizing an appropriate physical model and reliable controllers. We had the option to fabricate two dexterous motion walks for a quadruped robot utilizing this framework: trotting and galloping. Learning transferable locomotion policies is the subject of our paper. We used a simple incentive feature and a simple setting for this: optimizing running speed on a flat surface. To navigate the dynamic physical world in real-world situations, robots should see the climate, change their speed, and turn rapidly.

6 .2 FUTURE -WORK

This suggests two promising directions for future research. To begin with, we need to learn velocity arrangements that can change running pace and course powerfully. Second, by incorporating vision as part of sensory feedback, this work could be extended to manage complex terrain structures.

REFERENCES

REFERENCES

[1] M. Hutter et al., "ANYmal-a highly mobile and dynamic quadrupedal robot," in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst., 2016, pp. 38–44.

[2] S. Kuindersma et al., "Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot," Auton. Robots, vol. 40, pp. 429–455, 2016.

[3] A. W. Winkler, C. D. Bellicoso, M. Hutter, and J. Buchli, "Gait and trajectory optimization for legged systems through phase-based end-effector parameterization," IEEE Robot. Autom. Lett., vol. 3, no. 3, pp. 1560–1567, Jul. 2018.

[4] P. Fankhauser, M. Bjelonic, C. D. Bellicoso, T. Miki, and M. Hutter, "Robust rough-terrain locomotion with a quadrupedal robot," in Proc. IEEE Int. Conf. Robot. Autom., 2018, pp. 1–8.

[5] D. Belter, P. Abcki, and P. Skrzypczyski, "Adaptive motion planning for autonomous rough terrain traversal with a walking robot," J. Field Robot., vol. 33, pp. 337–370, 2016.

[6] M. Kalakrishnan, J. Buchli, P. Pastor, M. Mistry, and S. Schaal, "Learning, planning, and control for quadruped locomotion over challenging terrain," Int. J. Robot. Res., vol. 30, pp. 236–258, 2011.

[7] S. Tonneau, N. Mansard, C. Park, D. Manocha, F. Multon, and J. Pettré, "A reachability-based planner for sequences of acyclic contacts in cluttering environment," in Robotics Research. Berlin, Germany: Springer, 2018, pp. 287–303.

[8] X. B. Peng, E. Coumans, T. Zhang, T.-W. Lee, J. Tan, S. Levine, Learning agile robotic locomotion skills by imitating animals. arXiv:2004.00784 [cs.RO] (2 April 2020).

[9] S. Ha, P. Xu, Z. Tan, S. Levine, J. Tan, Learning to walk in the real world with minimal human effort. arXiv:2002.08550 [cs.RO] (20 February 2020).

[10] Y. Yang, K. Caluwaerts, A. Iscen, T. Zhang, J. Tan, V. Sindhwani, Data efficient reinforcement learning for legged robots, in Conference on Robot Learning (PMLR, 2019).

[11] T. Haarnoja, S. Ha, A. Zhou, J. Tan, G. Tucker, S. Levine, Learning to walk via deep reinforcement learning (Robotics: Science and Systems, 2019).

[12] Z. Xie, P. Clary, J. Dao, P. Morais, J. Hurst, M. van de Panne, Learning locomotion skills for Cassie: Iterative design and sim-to-real, in Conference on Robot Learning (PMLR, 2019).

[13] J. Lee, J. Hwangbo, M. Hutter, Robust recovery controller for a quadrupedal robot using deep reinforcement learning. arXiv:1901.07517 [cs.RO] (22 January 2019).

[14] R. Wang, J. Lehman, J. Clune, K. O. Stanley, Paired open-ended trailblazer (POET): Endlessly generating increasingly complex and diverse learning environments and their solutions. arXiv:1901.01753 [cs.NE] (7 January 2019).

[15] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, V. Vanhoucke, Sim-to-real: Learning agile locomotion for quadruped robots (Robotics: Science and Systems, 2018).

[16] C. D. Bellicoso, F. Jenelten, C. Gehring, M. Hutter, Dynamic locomotion through online nonlinear motion optimization for quadrupedal robots. IEEE Robot. Autom. Lett. 3, 2261–2268 (2018).

[17] X. B. Peng, M. Andrychowicz, W. Zaremba, P. Abbeel, Sim-to-real transfer of robotic control with dynamics randomization, in IEEE International Conference on Robotics and Automation (ICRA) (IEEE, 2018).

[18] R. Hartley, J. Mangelson, L. Gan, M. G. Jadidi, J. M. Walls, R. M. Eustice, J. W. Grizzle, Legged robot state-estimation through combined forward kinematic and preintegrated contact factors, in 2018 IEEE International Conference on Robotics and Automation (ICRA) (IEEE, 2018), pp. 1–8.

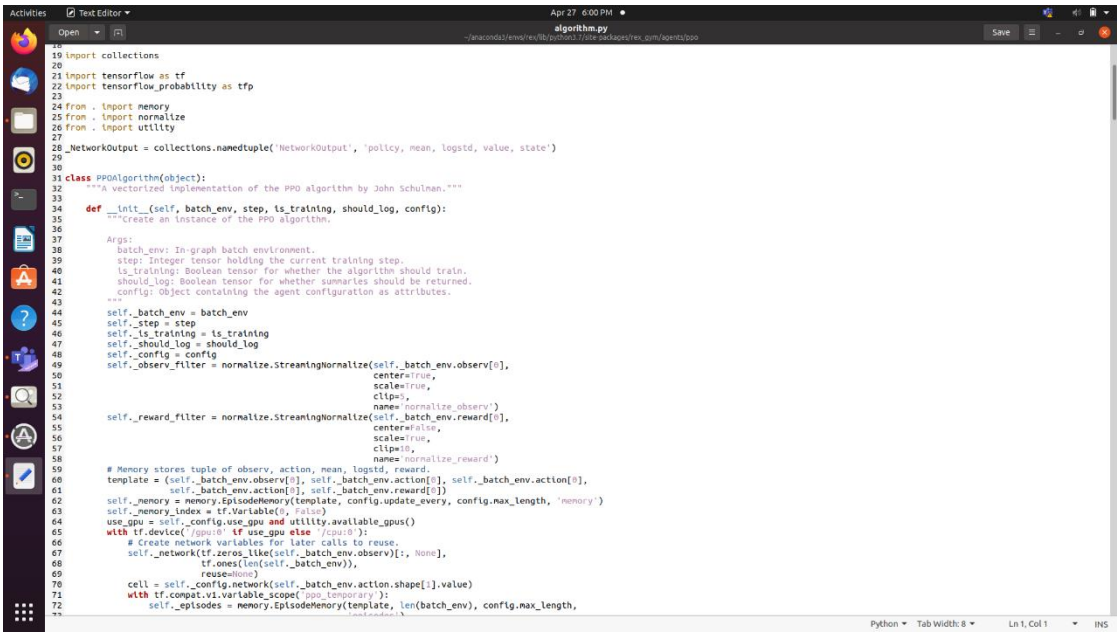
[19] 7. M. Camurri, M. Fallon, S. Bazeille, A. Radulescu, V. Barasuol, D. G. Caldwell, C. Semini, Probabilistic contact estimation and impact detection forstate estimation of quadruped robots. IEEE Robot. Autom. Lett. 2, 1023–1030 (2017).

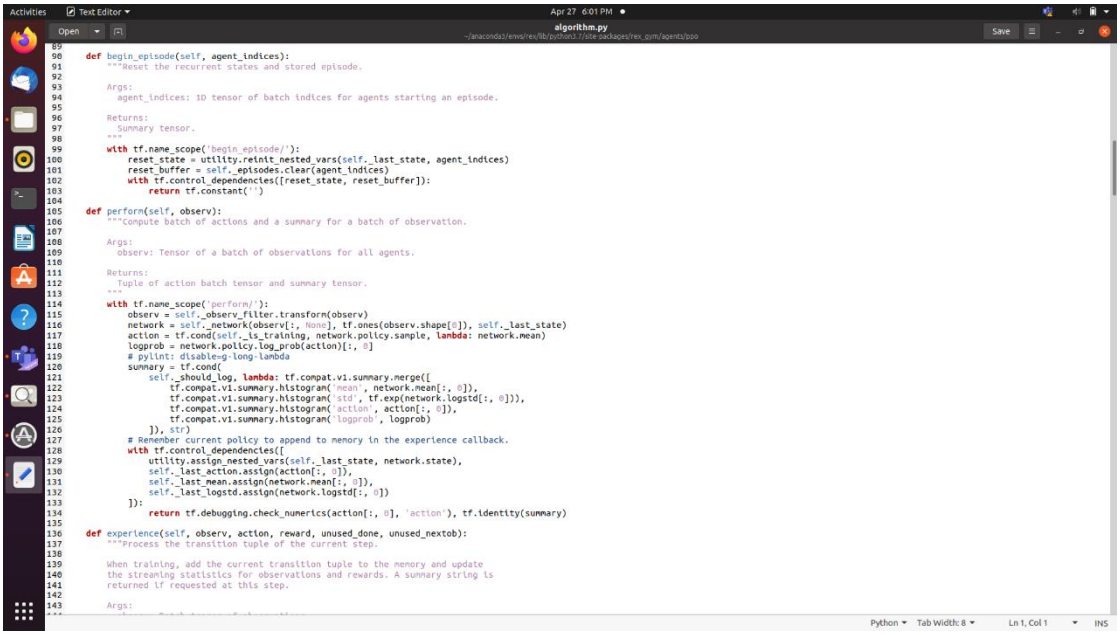
[20] C. Gehring, C. D. Bellicoso, S. Coros, M. Bloesch, P. Fankhauser, M. Hutter, R. Siegwart, Dynamic trotting on slopes for quadrupedal robots, in 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE, 2015), pp. 5129–5135.

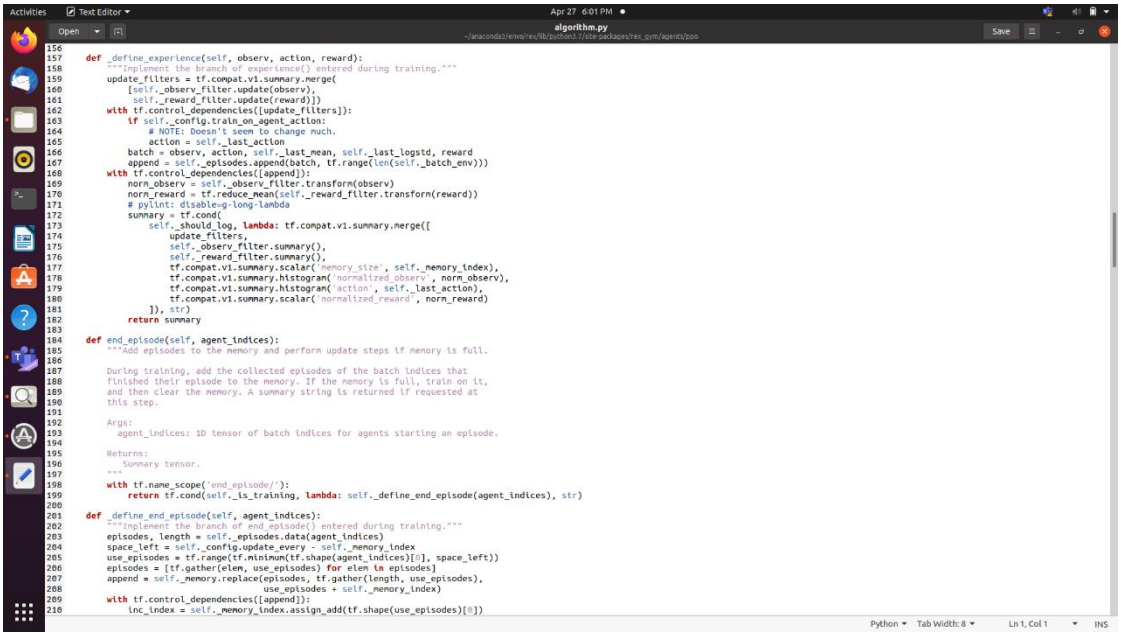
CODE

CODE

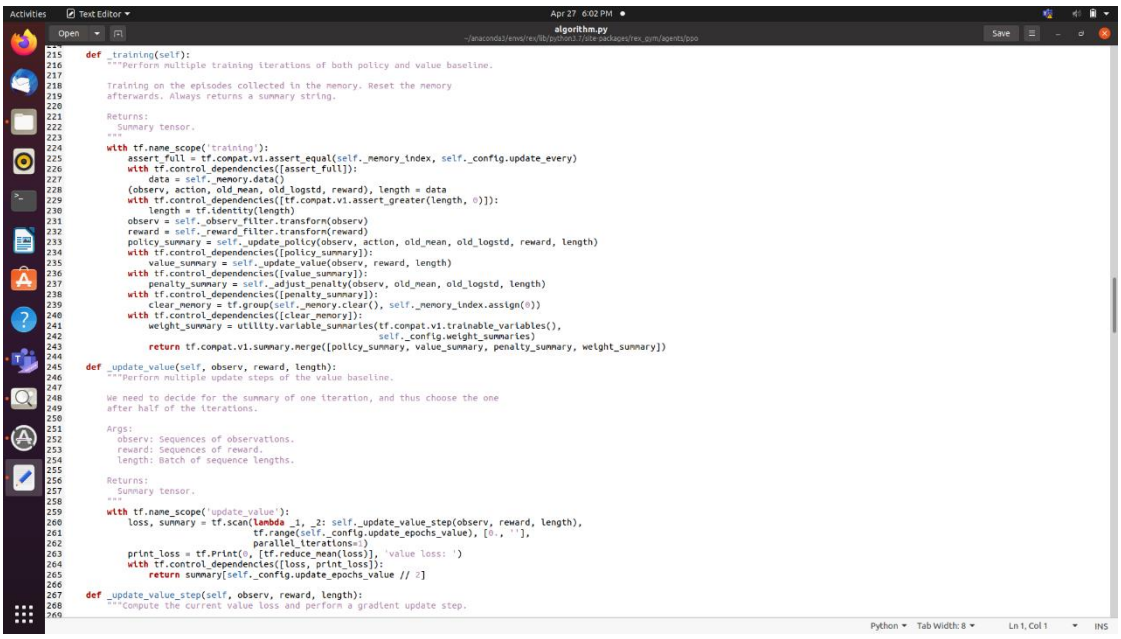
PPO Algorithm:





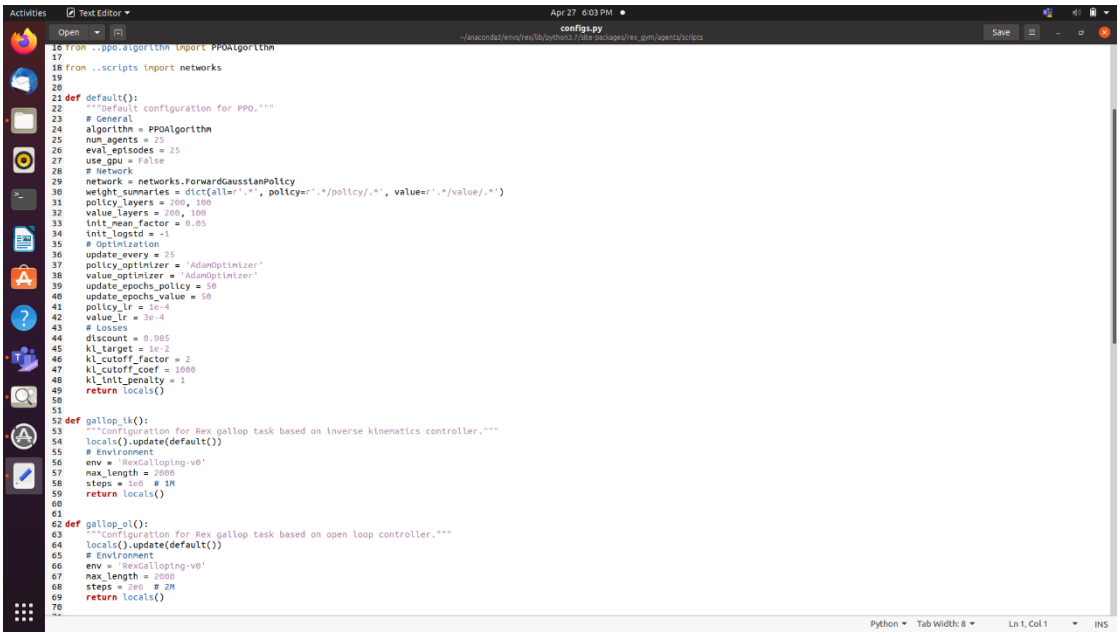


```
def _define_experience(self, observ, action, reward):
    """Implement the branch of experience() entered during training."""
    update_filters = tf.compat.v1.summary.merge(
        [self._observ_filter.update(observ),
         self._reward_filter.update(reward)])
    with tf.control_dependencies([update_filters]):
        if self._config.train_on_agent_action:
            # NOTE: Doesn't seem to change much.
            action = self._last_action
        batch = observ, action, self._last_mean, self._last_logstd, reward
        append = self._episodes.append(batch, tf.range(len(self._batch_env)))
    with tf.control_dependencies([append]):
        norm_observ = self._observ_filter.transform(observ)
        norm_reward = tf.reduce_mean(self._reward_filter.transform(reward))
        # pylint: disable=g-long-lambda
        summary = tf.cond(
            self._should_log, lambda: tf.compat.v1.summary.merge([
                update_filters,
                self._observ_filter.summary(),
                self._reward_filter.summary(),
                tf.compat.v1.summary.scalar('memory_size', self._memory_index),
                tf.compat.v1.summary.histogram('normalized_observ', norm_observ),
                tf.compat.v1.summary.histogram('action', self._last_action),
                tf.compat.v1.summary.scalar('normalized_reward', norm_reward)
            ]), str)
        return summary

def end_episode(self, agent_indices):
    """Add episodes to the memory and perform update steps if memory is full.

    During training, add the collected episodes of the batch indices that
    finished their episode to the memory. If the memory is full, train on it,
    and then clear the memory. A summary string is returned if requested at
    this step.

    Args:
      agent_indices: 1D tensor of batch indices for agents starting an episode.

    Returns:
      Summary tensor.
    """
    with tf.name_scope('end_episode/'):
        return tf.cond(self._is_training, lambda: self._define_end_episode(agent_indices), str)

def _define_end_episode(self, agent_indices):
    """Implement the branch of end_episode() entered during training."""
    episodes, length = self._episodes.data(agent_indices)
    space_left = self._config.update_every - self._memory_index
    use_episodes = tf.range(tf.minimum(tf.shape(agent_indices)[0], space_left))
    episodes = [tf.gather(elem, use_episodes) for elem in episodes]
    append = self._memory.replace(episodes, tf.gather(length, use_episodes),
                                  use_episodes + self._memory_index)
    with tf.control_dependencies([append]):
        inc_index = self._memory_index.assign_add(tf.shape(use_episodes)[0])
```

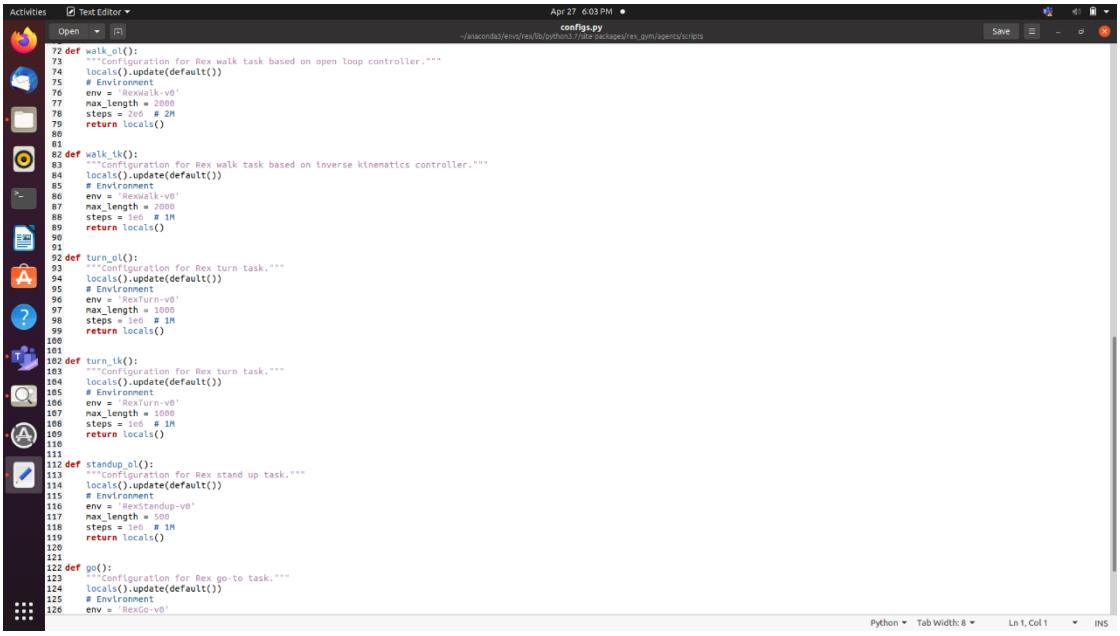


```
def _training(self):
    """Perform multiple training iterations of both policy and value baseline.

    Training on the episodes collected in the memory. Reset the memory
    afterwards. Always returns a summary string.

    Returns:
      Summary tensor.
    """
    with tf.name_scope('training'):
        assert_full = tf.compat.v1.assert_equal(self._memory_index, self._config.update_every)
        with tf.control_dependencies([assert_full]):
            data = self._memory.data()
        (observ, action, old_mean, old_logstd, reward), length = data
        with tf.control_dependencies([tf.compat.v1.assert_greater(length, 0)]):
            length = tf.identity(length)
        observ = self._observ_filter.transform(observ)
        reward = self._reward_filter.transform(reward)
        policy_summary = self._update_policy(observ, action, old_mean, old_logstd, reward, length)
        with tf.control_dependencies([policy_summary]):
            value_summary = self._update_value(observ, reward, length)
        with tf.control_dependencies([value_summary]):
            penalty_summary = self._adjust_penalty(observ, old_mean, old_logstd, length)
        with tf.control_dependencies([penalty_summary]):
            clear_memory = tf.group(self._memory.clear(), self._memory_index.assign(0))
        with tf.control_dependencies([clear_memory]):
            weight_summary = utility.variable_summaries(tf.compat.v1.trainable_variables(),
                                                        self._config.weight_summaries)
            return tf.compat.v1.summary.merge([policy_summary, value_summary, penalty_summary, weight_summary])

def _update_value(self, observ, reward, length):
    """Perform multiple update steps of the value baseline.

    We need to decide for the summary of one iteration, and thus choose the one
    after half of the iterations.

    Args:
      observ: Sequences of observations.
      reward: Sequences of reward.
      length: Batch of sequence lengths.

    Returns:
      Summary tensor.
    """
    with tf.name_scope('update_value'):
        loss, summary = tf.scan(lambda _1, _2: self._update_value_step(observ, reward, length),
                                tf.range(self._config.update_epochs_value), [0., ''],
                                parallel_iterations=1)
        print_loss = tf.Print(0, [tf.reduce_mean(loss)], 'value loss: ')
        with tf.control_dependencies([loss, print_loss]):
            return summary[self._config.update_epochs_value // 2]

def _update_value_step(self, observ, reward, length):
    """Compute the current value loss and perform a gradient update step.
```

Configuration:

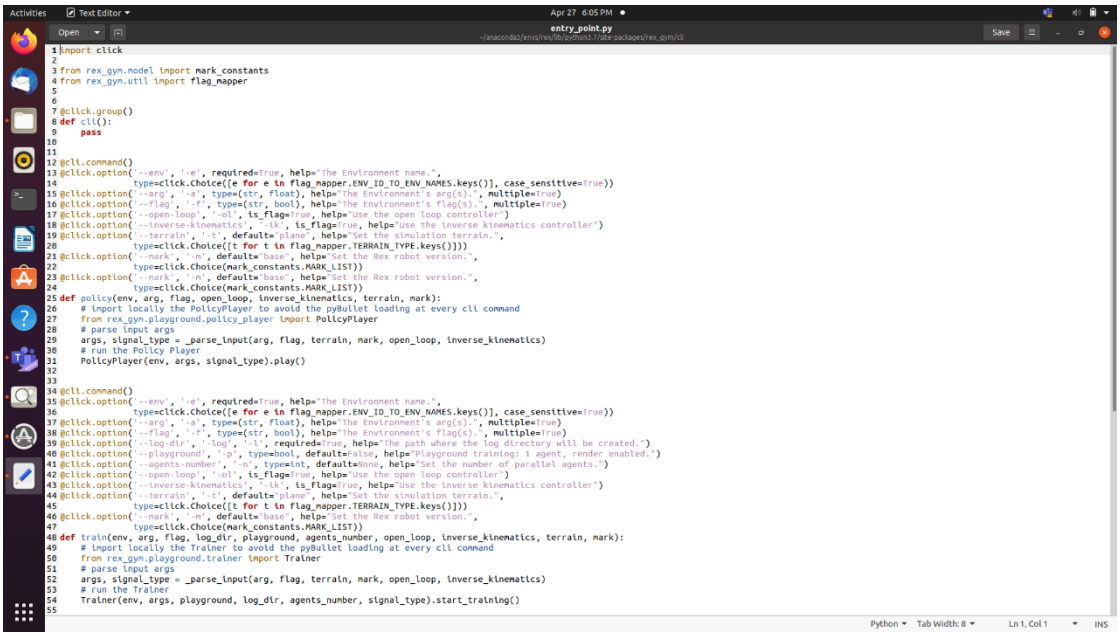



```python
from ..ppo.algorithm import PPOAlgorithm

from ..scripts import networks


def default():
    """Default configuration for PPO."""
    # General
    algorithm = PPOAlgorithm
    num_agents = 25
    eval_episodes = 25
    use_gpu = False
    # Network
    network = networks.ForwardGaussianPolicy
    weight_summaries = dict(all=r'.*', policy=r'.*/policy/.*', value=r'.*/value/.*')
    policy_layers = 200, 100
    value_layers = 200, 100
    init_mean_factor = 0.05
    init_logstd = -1
    # Optimization
    update_every = 25
    policy_optimizer = 'AdamOptimizer'
    value_optimizer = 'AdamOptimizer'
    update_epochs_policy = 50
    update_epochs_value = 50
    policy_lr = 1e-4
    value_lr = 3e-4
    # Losses
    discount = 0.985
    kl_target = 1e-2
    kl_cutoff_factor = 2
    kl_cutoff_coef = 1000
    kl_init_penalty = 1
    return locals()


def gallop_ik():
    """Configuration for Rex gallop task based on inverse kinematics controller."""
    locals().update(default())
    # Environment
    env = 'RexGalloping-v0'
    max_length = 2000
    steps = 1e6  # 1M
    return locals()


def gallop_ol():
    """Configuration for Rex gallop task based on open loop controller."""
    locals().update(default())
    # Environment
    env = 'RexGalloping-v0'
    max_length = 2000
    steps = 2e6  # 2M
    return locals()
```

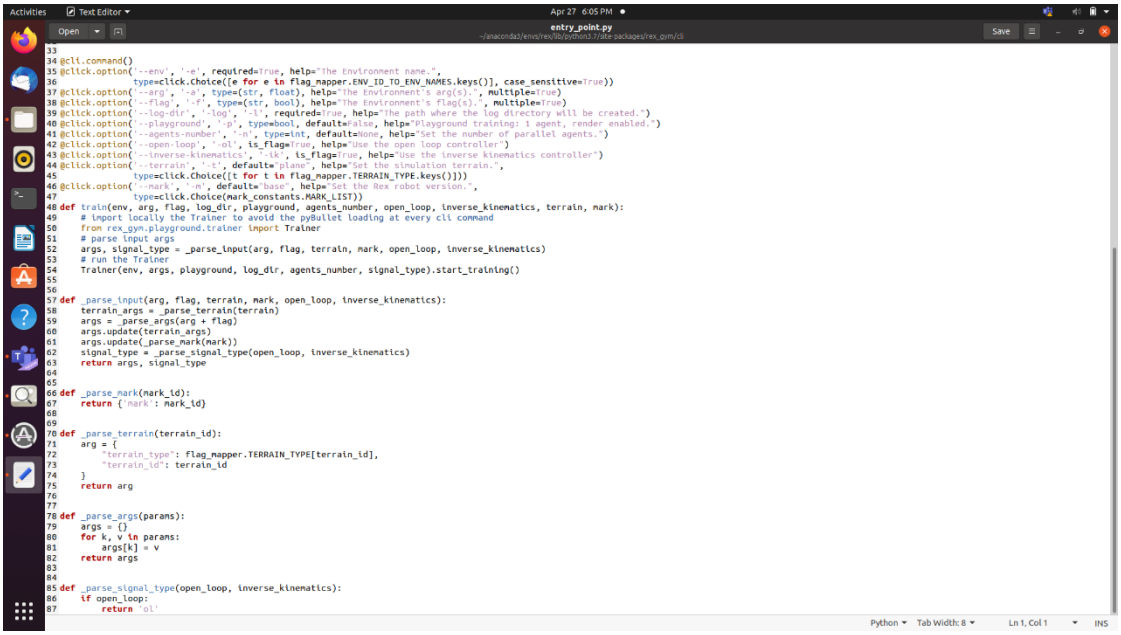



```python
def walk_ol():
    """Configuration for Rex walk task based on open loop controller."""
    locals().update(default())
    # Environment
    env = 'RexWalk-v0'
    max_length = 2000
    steps = 2e6  # 2M
    return locals()


def walk_ik():
    """Configuration for Rex walk task based on inverse kinematics controller."""
    locals().update(default())
    # Environment
    env = 'RexWalk-v0'
    max_length = 2000
    steps = 1e6  # 1M
    return locals()


def turn_ol():
    """Configuration for Rex turn task."""
    locals().update(default())
    # Environment
    env = 'RexTurn-v0'
    max_length = 1000
    steps = 1e6  # 1M
    return locals()


def turn_ik():
    """Configuration for Rex turn task."""
    locals().update(default())
    # Environment
    env = 'RexTurn-v0'
    max_length = 1000
    steps = 1e6  # 1M
    return locals()


def standup_ol():
    """Configuration for Rex stand up task."""
    locals().update(default())
    # Environment
    env = 'RexStandup-v0'
    max_length = 500
    steps = 1e6  # 1M
    return locals()


def go():
    """Configuration for Rex go-to task."""
    locals().update(default())
    # Environment
    env = 'RexGo-v0'
```

Entry Point:

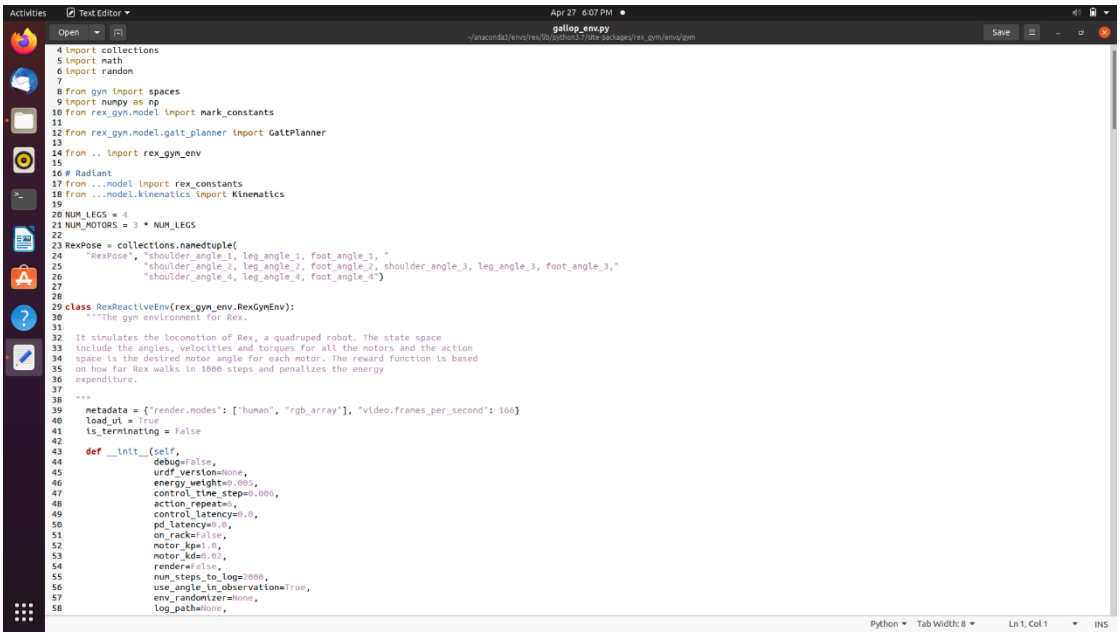


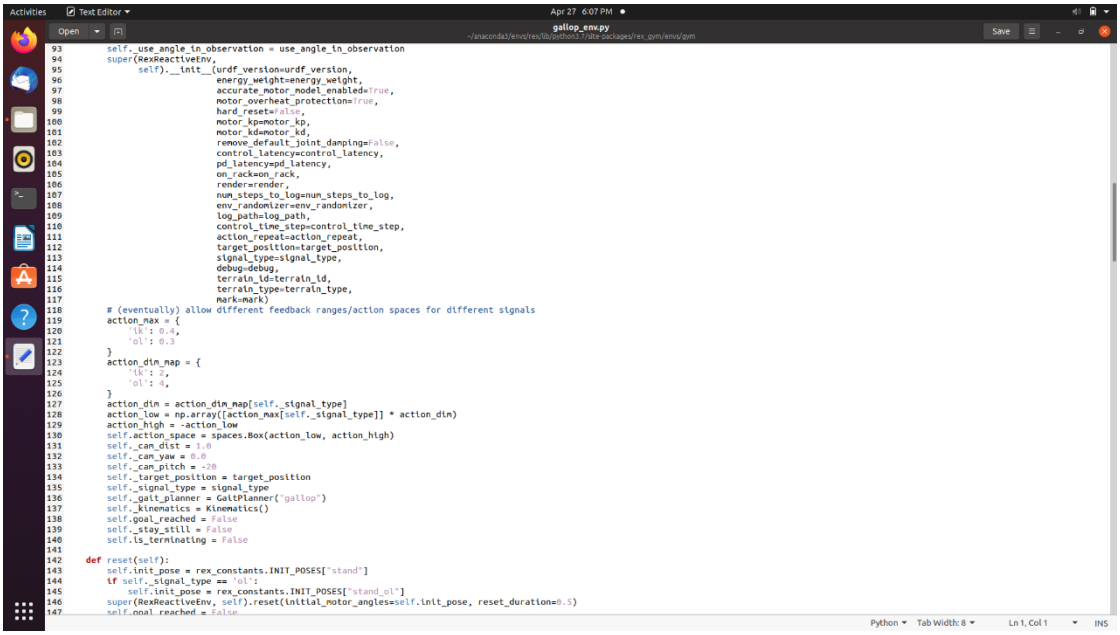

```python
import click

from rex_gym.model import mark_constants
from rex_gym.util import flag_mapper


@click.group()
def cli():
    pass


@cli.command()
@click.option('--env', '-e', required=True, help="The Environment name.",
              type=click.Choice([e for e in flag_mapper.ENV_ID_TO_ENV_NAMES.keys()], case_sensitive=True))
@click.option('--arg', '-a', type=(str, float), help="The Environment's arg(s).", multiple=True)
@click.option('--flag', '-f', type=(str, bool), help="The Environment's flag(s).", multiple=True)
@click.option('--open-loop', '-ol', is_flag=True, help="Use the open loop controller")
@click.option('--inverse-kinematics', '-ik', is_flag=True, help="Use the inverse kinematics controller")
@click.option('--terrain', '-t', default="plane", help="Set the simulation terrain.",
              type=click.Choice([t for t in flag_mapper.TERRAIN_TYPE.keys()]))
@click.option('--mark', '-m', default="base", help="Set the Rex robot version.",
              type=click.Choice(mark_constants.MARK_LIST))
@click.option('--mark', '-m', default="base", help="Set the Rex robot version.",
              type=click.Choice(mark_constants.MARK_LIST))
def policy(env, arg, flag, open_loop, inverse_kinematics, terrain, mark):
    # Import locally the PolicyPlayer to avoid the pyBullet loading at every cli command
    from rex_gym.playground.policy_player import PolicyPlayer
    # parse input args
    args, signal_type = _parse_input(arg, flag, terrain, mark, open_loop, inverse_kinematics)
    # run the Policy Player
    PolicyPlayer(env, args, signal_type).play()


@cli.command()
@click.option('--env', '-e', required=True, help="The Environment name.",
              type=click.Choice([e for e in flag_mapper.ENV_ID_TO_ENV_NAMES.keys()], case_sensitive=True))
@click.option('--arg', '-a', type=(str, float), help="The Environment's arg(s).", multiple=True)
@click.option('--flag', '-f', type=(str, bool), help="The Environment's flag(s).", multiple=True)
@click.option('--log-dir', '-log', '-l', required=True, help="The path where the log directory will be created.")
@click.option('--playground', '-p', type=bool, default=False, help="Playground training: 1 agent, render enabled.")
@click.option('--agents-number', '-n', type=int, default=None, help="Set the number of parallel agents.")
@click.option('--open-loop', '-ol', is_flag=True, help="Use the open loop controller")
@click.option('--inverse-kinematics', '-ik', is_flag=True, help="Use the inverse kinematics controller")
@click.option('--terrain', '-t', default="plane", help="Set the simulation terrain.",
              type=click.Choice([t for t in flag_mapper.TERRAIN_TYPE.keys()]))
@click.option('--mark', '-m', default="base", help="Set the Rex robot version.",
              type=click.Choice(mark_constants.MARK_LIST))
def train(env, arg, flag, log_dir, playground, agents_number, open_loop, inverse_kinematics, terrain, mark):
    # Import locally the Trainer to avoid the pyBullet loading at every cli command
    from rex_gym.playground.trainer import Trainer
    # parse input args
    args, signal_type = _parse_input(arg, flag, terrain, mark, open_loop, inverse_kinematics)
    # run the Trainer
    Trainer(env, args, playground, log_dir, agents_number, signal_type).start_training()
```

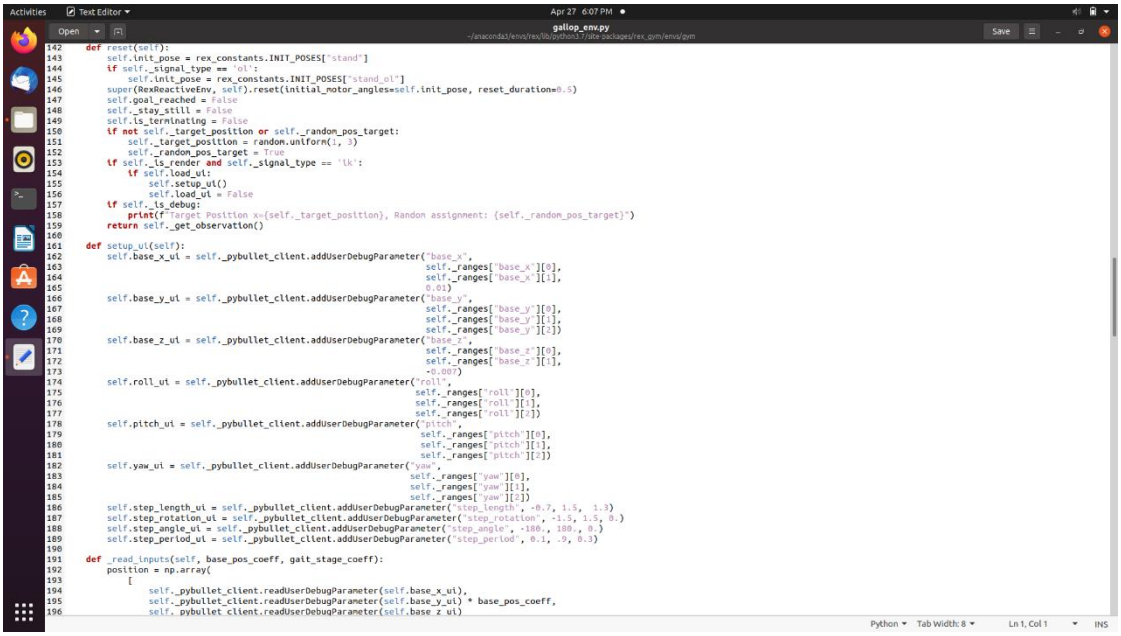



```python

@cli.command()
@click.option('--env', '-e', required=True, help="The Environment name.",
              type=click.Choice([e for e in flag_mapper.ENV_ID_TO_ENV_NAMES.keys()], case_sensitive=True))
@click.option('--arg', '-a', type=(str, float), help="The Environment's arg(s).", multiple=True)
@click.option('--flag', '-f', type=(str, bool), help="The Environment's flag(s).", multiple=True)
@click.option('--log-dir', '-log', '-l', required=True, help="The path where the log directory will be created.")
@click.option('--playground', '-p', type=bool, default=False, help="Playground training: 1 agent, render enabled.")
@click.option('--agents-number', '-n', type=int, default=None, help="Set the number of parallel agents.")
@click.option('--open-loop', '-ol', is_flag=True, help="Use the open loop controller")
@click.option('--inverse-kinematics', '-ik', is_flag=True, help="Use the inverse kinematics controller")
@click.option('--terrain', '-t', default="plane", help="Set the simulation terrain.",
              type=click.Choice([t for t in flag_mapper.TERRAIN_TYPE.keys()]))
@click.option('--mark', '-m', default="base", help="Set the Rex robot version.",
              type=click.Choice(mark_constants.MARK_LIST))
def train(env, arg, flag, log_dir, playground, agents_number, open_loop, inverse_kinematics, terrain, mark):
    # Import locally the Trainer to avoid the pyBullet loading at every cli command
    from rex_gym.playground.trainer import Trainer
    # parse input args
    args, signal_type = _parse_input(arg, flag, terrain, mark, open_loop, inverse_kinematics)
    # run the Trainer
    Trainer(env, args, playground, log_dir, agents_number, signal_type).start_training()


def _parse_input(arg, flag, terrain, mark, open_loop, inverse_kinematics):
    terrain_args = _parse_terrain(terrain)
    args = _parse_args(arg + flag)
    args.update(terrain_args)
    args.update(_parse_mark(mark))
    signal_type = _parse_signal_type(open_loop, inverse_kinematics)
    return args, signal_type


def _parse_mark(mark_id):
    return {'mark': mark_id}


def _parse_terrain(terrain_id):
    arg = {
        "terrain_type": flag_mapper.TERRAIN_TYPE[terrain_id],
        "terrain_id": terrain_id
    }
    return arg


def _parse_args(params):
    args = {}
    for k, v in params:
        args[k] = v
    return args


def _parse_signal_type(open_loop, inverse_kinematics):
    if open_loop:
        return 'ol'
```

Galloping Environment:



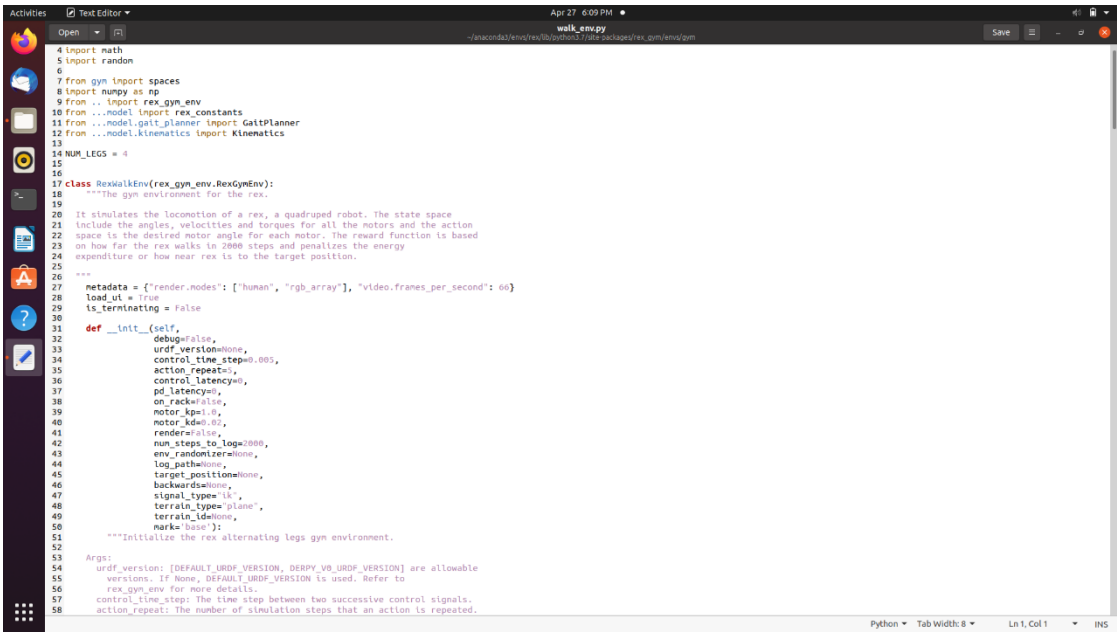




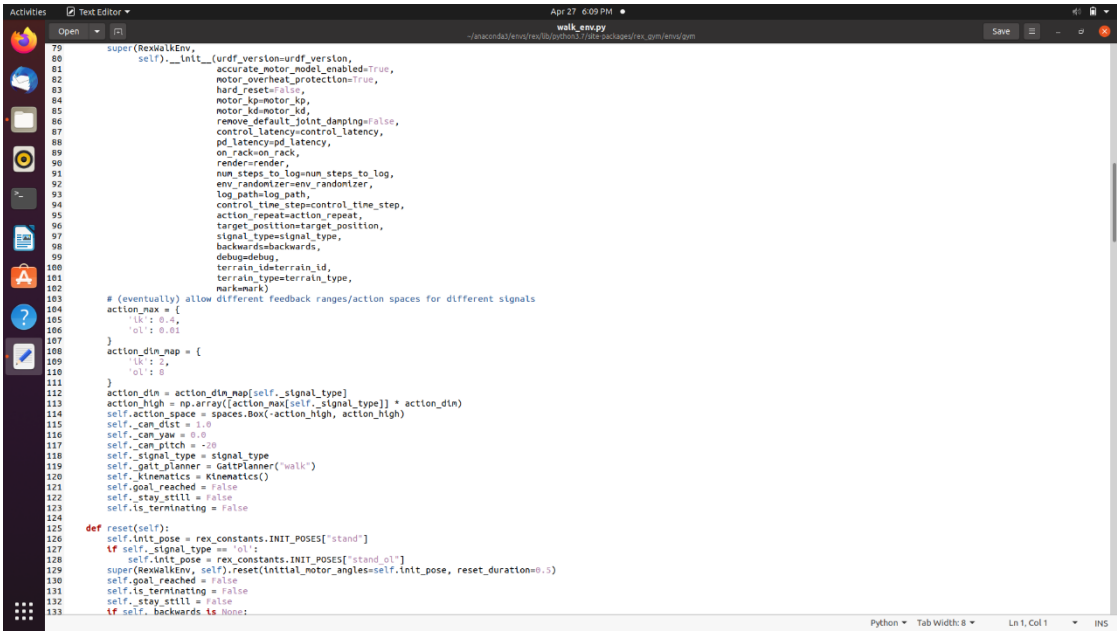
```
    def reset(self):
        self.init_pose = rex_constants.INIT_POSES["stand"]
        if self._signal_type == 'ol':
            self.init_pose = rex_constants.INIT_POSES["stand_ol"]
        super(RexReactiveEnv, self).reset(initial_motor_angles=self.init_pose, reset_duration=0.5)
        self.goal_reached = False
        self._stay_still = False
        self.is_terminating = False
        if not self._target_position or self._random_pos_target:
            self._target_position = random.uniform(1, 3)
            self._random_pos_target = True
        if self._is_render and self._signal_type == 'ik':
            if self.load_ui:
                self.setup_ui()
                self.load_ui = False
        if self._is_debug:
            print(f"Target Position x={self._target_position}, Random assignment: {self._random_pos_target}")
        return self._get_observation()

    def setup_ui(self):
        self.base_x_ui = self._pybullet_client.addUserDebugParameter("base_x",
                                                                      self._ranges['base_x'][0],
                                                                      self._ranges['base_x'][1],
                                                                      0.01)
        self.base_y_ui = self._pybullet_client.addUserDebugParameter("base_y",
                                                                      self._ranges['base_y'][0],
                                                                      self._ranges['base_y'][1],
                                                                      self._ranges['base_y'][2])
        self.base_z_ui = self._pybullet_client.addUserDebugParameter("base_z",
                                                                      self._ranges['base_z'][0],
                                                                      self._ranges['base_z'][1],
                                                                      -0.007)
        self.roll_ui = self._pybullet_client.addUserDebugParameter("roll",
                                                                   self._ranges['roll'][0],
                                                                   self._ranges['roll'][1],
                                                                   self._ranges['roll'][2])
        self.pitch_ui = self._pybullet_client.addUserDebugParameter("pitch",
                                                                    self._ranges['pitch'][0],
                                                                    self._ranges['pitch'][1],
                                                                    self._ranges['pitch'][2])
        self.yaw_ui = self._pybullet_client.addUserDebugParameter("yaw",
                                                                  self._ranges['yaw'][0],
                                                                  self._ranges['yaw'][1],
                                                                  self._ranges['yaw'][2])
        self.step_length_ui = self._pybullet_client.addUserDebugParameter("step_length", -0.7, 1.5,  1.3)
        self.step_rotation_ui = self._pybullet_client.addUserDebugParameter("step_rotation", -1.5, 1.5, 0.)
        self.step_angle_ui = self._pybullet_client.addUserDebugParameter("step_angle", -180., 180., 0.)
        self.step_period_ui = self._pybullet_client.addUserDebugParameter("step_period", 0.1, .9, 0.3)

    def _read_inputs(self, base_pos_coeff, gait_stage_coeff):
        position = np.array(
            [
                self._pybullet_client.readUserDebugParameter(self.base_x_ui),
                self._pybullet_client.readUserDebugParameter(self.base_y_ui) * base_pos_coeff,
                self._pybullet_client.readUserDebugParameter(self.base_z_ui)
```

```
    def _check_target_position(self, t):
        if self._target_position:
            current_x = abs(self.rex.GetBasePosition()[0])
            # give 0.15 stop space
            if current_x >= abs(self._target_position):
                self.goal_reached = True
                if not self.is_terminating:
                    self.end_time = t
                    self.is_terminating = True

    @staticmethod
    def _evaluate_stage_coefficient(current_t, end_t=0.0, width=0.001):
        # sigmoid function
        beta = p = width
        if p - beta + end_t <= current_t <= p - (beta / 2) + end_t:
            return (2 / beta ** 2) * (current_t - p + beta) ** 2
        elif p - (beta/2) + end_t <= current_t <= p + end_t:
            return 1 - (2 / beta ** 2) * (current_t - p) ** 2
        else:
            return 1

    @staticmethod
    def _evaluate_brakes_stage_coeff(current_t, action, end_t=0.0, end_value=0.0):
        # ramp function
        p = 1. + action[0]
        if end_t <= current_t <= p + end_t:
            return 1 - (current_t - end_t)
        else:
            return end_value

    @staticmethod
    def _evaluate_gait_stage_coeff(current_t, action, end_t=0.0):
        # ramp function
        p = 1. + action[1]
        if end_t <= current_t <= p + end_t:
            return current_t
        else:
            return 1.0

    def _signal(self, t, action):
        if self._signal_type == 'ik':
            return self._IK_signal(t, action)
        if self._signal_type == 'ol':
            return self._open_loop_signal(t, action)

    def _IK_signal(self, t, action):
        base_pos_coeff = self._evaluate_stage_coefficient(t, width=1.5)
        gait_stage_coeff = self._evaluate_gait_stage_coeff(t, action)
        if self._is_render:
            position, orientation, step_length, step_rotation, step_angle, step_period = \
                self._read_inputs(base_pos_coeff, gait_stage_coeff)
        else:
            position = np.array([0.01,
                                 self._base_y * base_pos_coeff,
                                 -0.007])
```

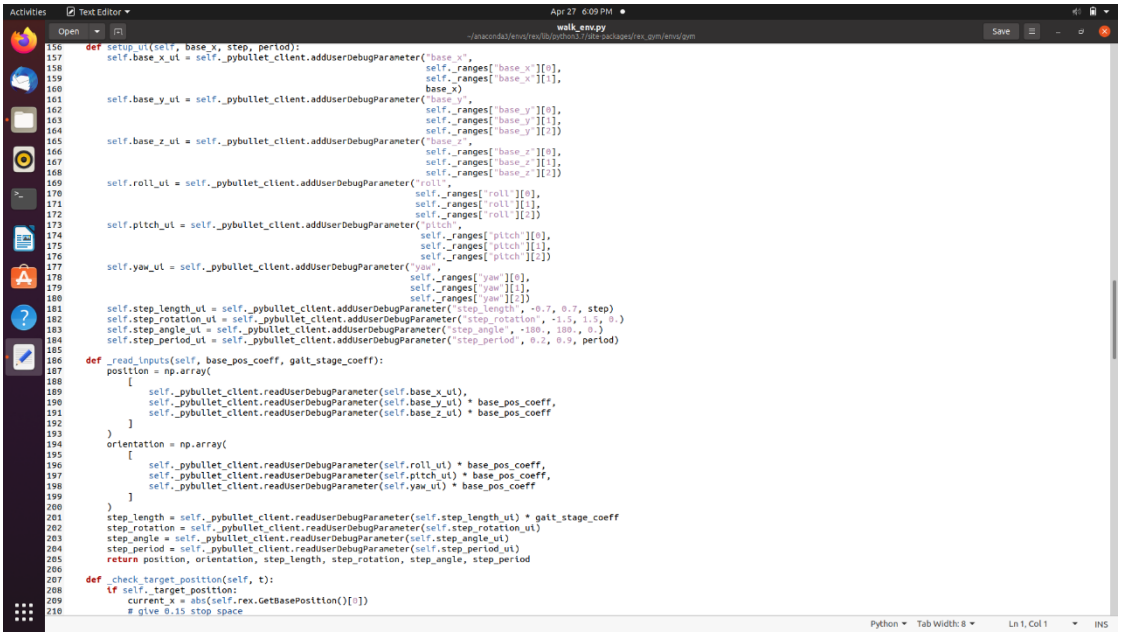
Walking Environment:




```python
import math
import random

from gym import spaces
import numpy as np
from .. import rex_gym_env
from ...model import rex_constants
from ...model.gait_planner import GaitPlanner
from ...model.kinematics import Kinematics

NUM_LEGS = 4


class RexWalkEnv(rex_gym_env.RexGymEnv):
  """The gym environment for the rex.

  It simulates the locomotion of a rex, a quadruped robot. The state space
  include the angles, velocities and torques for all the motors and the action
  space is the desired motor angle for each motor. The reward function is based
  on how far the rex walks in 2000 steps and penalizes the energy
  expenditure or how near rex is to the target position.

  """
  metadata = {"render.modes": ["human", "rgb_array"], "video.frames_per_second": 66}
  load_ui = True
  is_terminating = False

  def __init__(self,
               debug=False,
               urdf_version=None,
               control_time_step=0.005,
               action_repeat=1,
               control_latency=0,
               pd_latency=0,
               on_rack=False,
               motor_kp=1.0,
               motor_kd=0.02,
               render=False,
               num_steps_to_log=2000,
               env_randomizer=None,
               log_path=None,
               target_position=None,
               backwards=None,
               signal_type='ik',
               terrain_type="plane",
               terrain_id=None,
               mark='base'):
    """Initialize the rex alternating legs gym environment.

  Args:
    urdf_version: [DEFAULT_URDF_VERSION, DERPY_V0_URDF_VERSION] are allowable
      versions. If None, DEFAULT_URDF_VERSION is used. Refer to
      rex_gym_env for more details.
    control_time_step: The time step between two successive control signals.
    action_repeat: The number of simulation steps that an action is repeated.
```

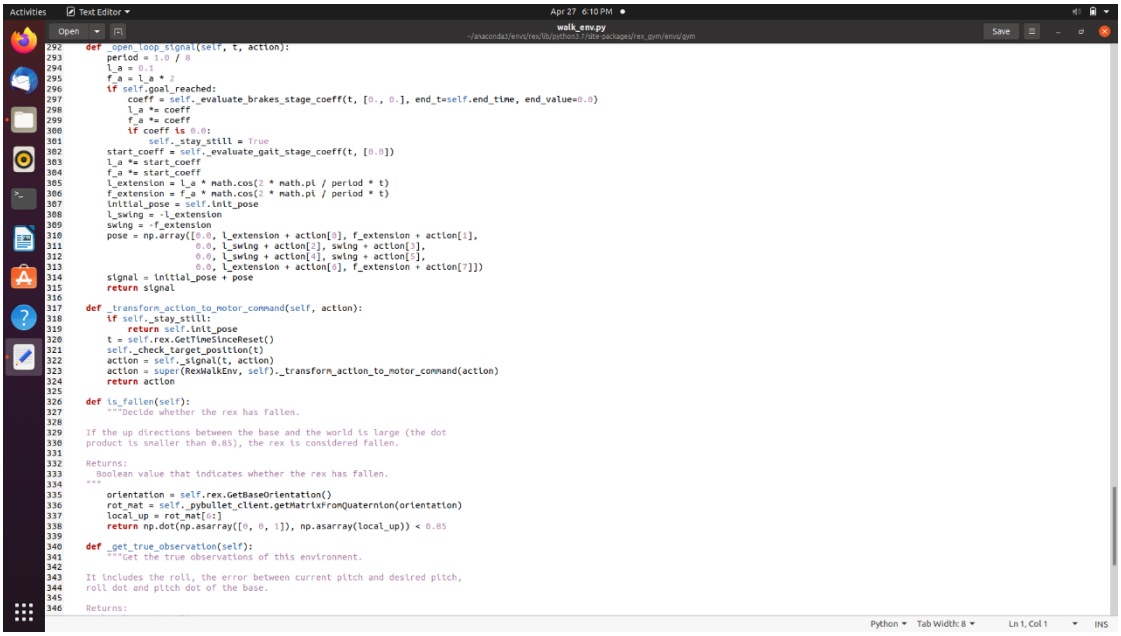


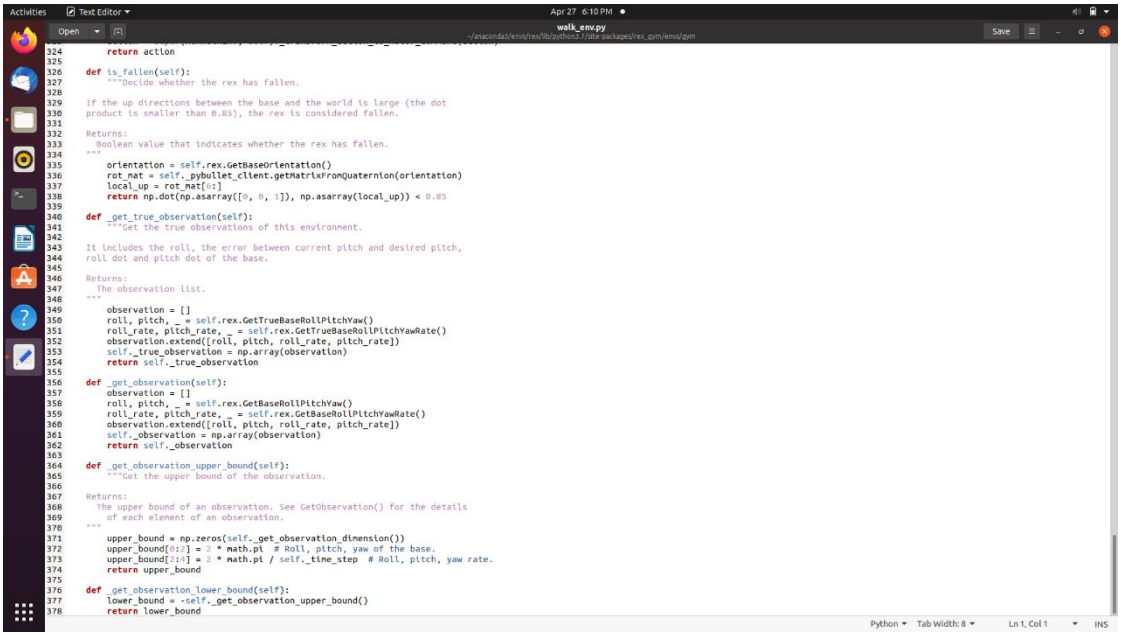

```python
      super(RexWalkEnv,
            self).__init__(urdf_version=urdf_version,
                           accurate_motor_model_enabled=True,
                           motor_overheat_protection=True,
                           hard_reset=False,
                           motor_kp=motor_kp,
                           motor_kd=motor_kd,
                           remove_default_joint_damping=False,
                           control_latency=control_latency,
                           pd_latency=pd_latency,
                           on_rack=on_rack,
                           render=render,
                           num_steps_to_log=num_steps_to_log,
                           env_randomizer=env_randomizer,
                           log_path=log_path,
                           control_time_step=control_time_step,
                           action_repeat=action_repeat,
                           target_position=target_position,
                           signal_type=signal_type,
                           backwards=backwards,
                           debug=debug,
                           terrain_id=terrain_id,
                           terrain_type=terrain_type,
                           mark=mark)
      # (eventually) allow different feedback ranges/action spaces for different signals
      action_max = {
          'ik': 0.4,
          'ol': 0.01
      }
      action_dim_map = {
          'ik': 2,
          'ol': 8
      }
      action_dim = action_dim_map[self._signal_type]
      action_high = np.array([action_max[self._signal_type]] * action_dim)
      self.action_space = spaces.Box(-action_high, action_high)
      self._cam_dist = 1.0
      self._cam_yaw = 0.0
      self._cam_pitch = -20
      self._signal_type = signal_type
      self._gait_planner = GaitPlanner("walk")
      self._kinematics = Kinematics()
      self.goal_reached = False
      self._stay_still = False
      self.is_terminating = False

  def reset(self):
      self.init_pose = rex_constants.INIT_POSES["stand"]
      if self._signal_type == 'ol':
          self.init_pose = rex_constants.INIT_POSES["stand_ol"]
      super(RexWalkEnv, self).reset(initial_motor_angles=self.init_pose, reset_duration=0.5)
      self.goal_reached = False
      self.is_terminating = False
      self._stay_still = False
      if self._backwards is None:
```



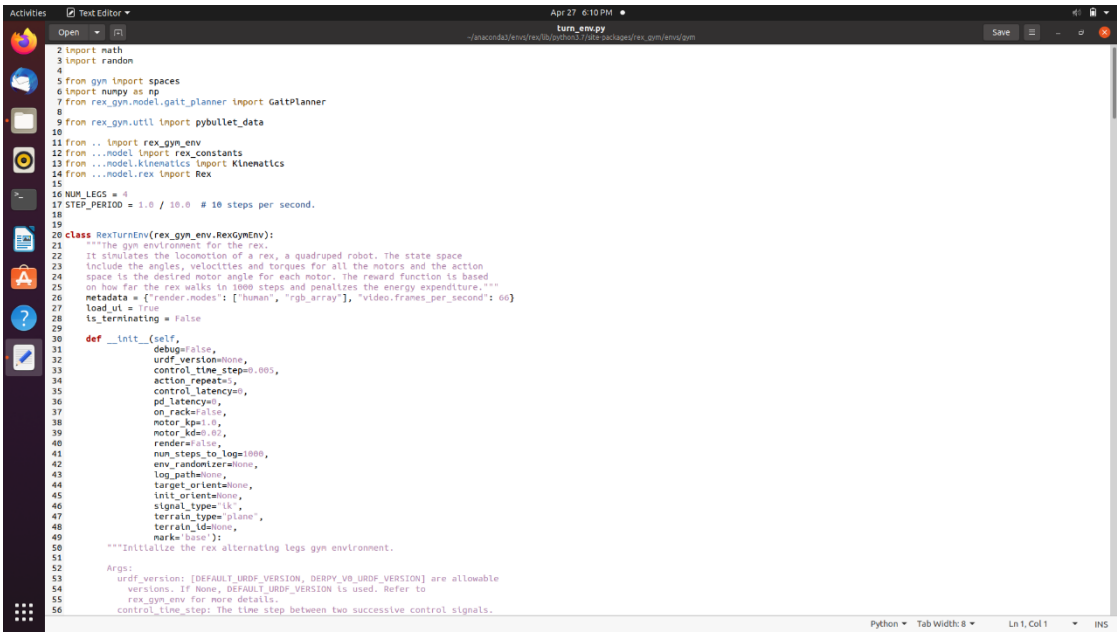
```
156     def setup_ui(self, base_x, step, period):
157         self.base_x_ui = self._pybullet_client.addUserDebugParameter("base_x",
158                                                                       self._ranges["base_x"][0],
159                                                                       self._ranges["base_x"][1],
160                                                                       base_x)
161         self.base_y_ui = self._pybullet_client.addUserDebugParameter("base_y",
162                                                                       self._ranges["base_y"][0],
163                                                                       self._ranges["base_y"][1],
164                                                                       self._ranges["base_y"][2])
165         self.base_z_ui = self._pybullet_client.addUserDebugParameter("base_z",
166                                                                       self._ranges["base_z"][0],
167                                                                       self._ranges["base_z"][1],
168                                                                       self._ranges["base_z"][2])
169         self.roll_ui = self._pybullet_client.addUserDebugParameter("roll",
170                                                                    self._ranges["roll"][0],
171                                                                    self._ranges["roll"][1],
172                                                                    self._ranges["roll"][2])
173         self.pitch_ui = self._pybullet_client.addUserDebugParameter("pitch",
174                                                                     self._ranges["pitch"][0],
175                                                                     self._ranges["pitch"][1],
176                                                                     self._ranges["pitch"][2])
177         self.yaw_ui = self._pybullet_client.addUserDebugParameter("yaw",
178                                                                   self._ranges["yaw"][0],
179                                                                   self._ranges["yaw"][1],
180                                                                   self._ranges["yaw"][2])
181         self.step_length_ui = self._pybullet_client.addUserDebugParameter('step_length', -0.7, 0.7, step)
182         self.step_rotation_ui = self._pybullet_client.addUserDebugParameter('step_rotation', -1.5, 1.5, 0.)
183         self.step_angle_ui = self._pybullet_client.addUserDebugParameter('step_angle', -180., 180., 0.)
184         self.step_period_ui = self._pybullet_client.addUserDebugParameter("step_period", 0.2, 0.9, period)
185
186     def _read_inputs(self, base_pos_coeff, gait_stage_coeff):
187         position = np.array(
188             [
189                 self._pybullet_client.readUserDebugParameter(self.base_x_ui),
190                 self._pybullet_client.readUserDebugParameter(self.base_y_ui) * base_pos_coeff,
191                 self._pybullet_client.readUserDebugParameter(self.base_z_ui) * base_pos_coeff
192             ]
193         )
194         orientation = np.array(
195             [
196                 self._pybullet_client.readUserDebugParameter(self.roll_ui) * base_pos_coeff,
197                 self._pybullet_client.readUserDebugParameter(self.pitch_ui) * base_pos_coeff,
198                 self._pybullet_client.readUserDebugParameter(self.yaw_ui) * base_pos_coeff
199             ]
200         )
201         step_length = self._pybullet_client.readUserDebugParameter(self.step_length_ui) * gait_stage_coeff
202         step_rotation = self._pybullet_client.readUserDebugParameter(self.step_rotation_ui)
203         step_angle = self._pybullet_client.readUserDebugParameter(self.step_angle_ui)
204         step_period = self._pybullet_client.readUserDebugParameter(self.step_period_ui)
205         return position, orientation, step_length, step_rotation, step_angle, step_period
206
207     def _check_target_position(self, t):
208         if self._target_position:
209             current_x = abs(self.rex.GetBasePosition()[0])
210             # give 0.15 stop space
```

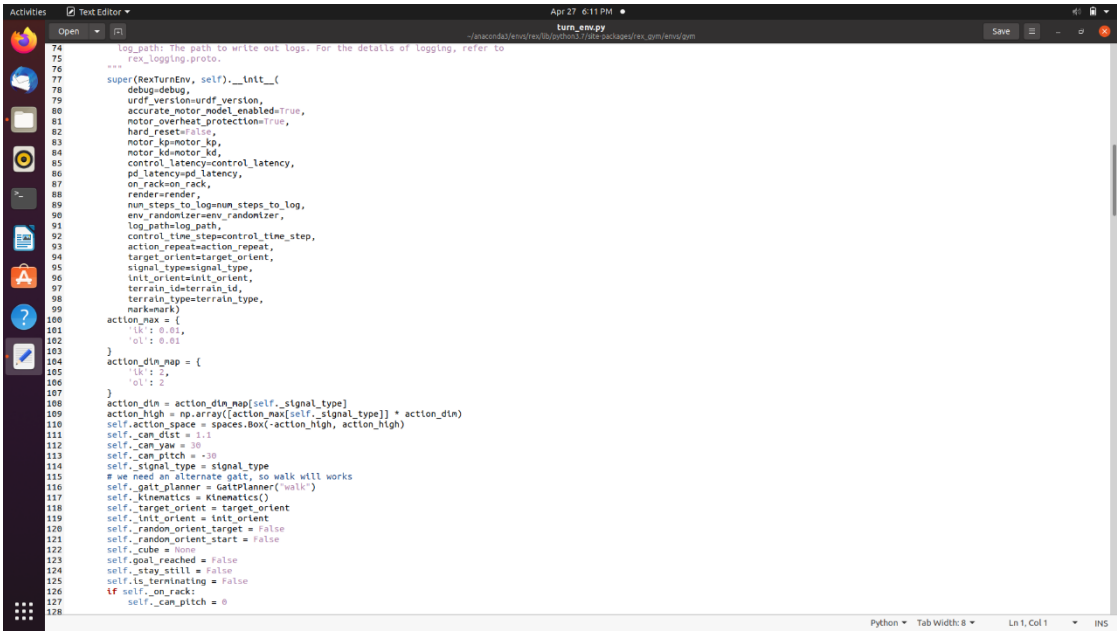
```
207     def _check_target_position(self, t):
208         if self._target_position:
209             current_x = abs(self.rex.GetBasePosition()[0])
210             # give 0.15 stop space
211             if current_x >= abs(self._target_position) - 0.15:
212                 self.goal_reached = True
213                 if not self.is_terminating:
214                     self.end_time = t
215                     self.is_terminating = True
216
217     @staticmethod
218     def _evaluate_base_stage_coeff(current_t, end_t=0.0, width=0.001):
219         # sigmoid function
220         beta = p = width
221         if p - beta + end_t <= current_t <= p - (beta / 2) + end_t:
222             return (2 / beta ** 2) * (current_t - p + beta) ** 2
223         elif p - (beta/2) + end_t <= current_t <= p + end_t:
224             return 1 - (2 / beta ** 2) * (current_t - p) ** 2
225         else:
226             return 1
227
228     @staticmethod
229     def _evaluate_gait_stage_coeff(current_t, action, end_t=0.0):
230         # ramp function
231         p = 0.8 + action[0]
232         if end_t <= current_t <= p + end_t:
233             return current_t
234         else:
235             return 1.0
236
237     @staticmethod
238     def _evaluate_brakes_stage_coeff(current_t, action, end_t=0.0, end_value=0.0):
239         # ramp function
240         p = 0.8 + action[1]
241         if end_t <= current_t <= p + end_t:
242             return 1 - (current_t - end_t)
243         else:
244             return end_value
245
246     def _signal(self, t, action):
247         if self._signal_type == 'ik':
248             return self._IK_signal(t, action)
249         if self._signal_type == 'ol':
250             return self._open_loop_signal(t, action)
251
252     def _IK_signal(self, t, action):
253         base_pos_coeff = self._evaluate_base_stage_coeff(t, width=1.5)
254         gait_stage_coeff = self._evaluate_gait_stage_coeff(t, action)
255         step = 0.6
256         period = 0.65
257         base_x = self._base_x
258         if self.backwards:
259             step = -.3
260             period = .5
261             base_x = .0
```

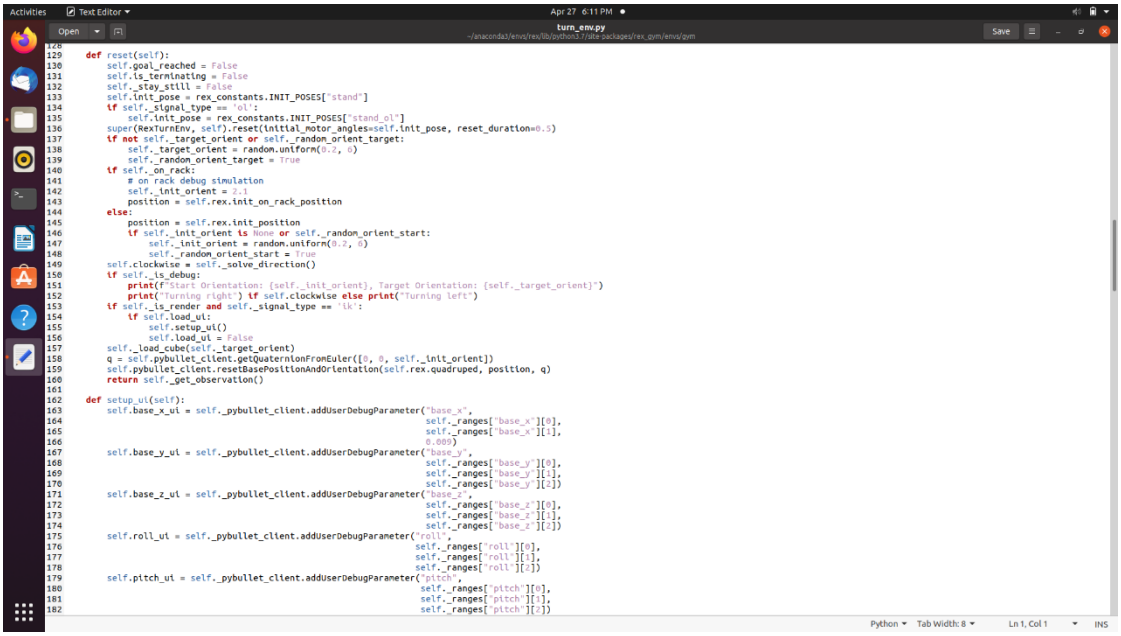

Activities
Text Editor
Apr 27 6:10 PM
Open
walk_env.py
~/anaconda3/envs/rex/lib/python3.7/site-packages/rex_gym/envs/gym
Save
def _open_loop_signal(self, t, action):
period = 1.0 / 8
l_a = 0.3
f_a = l_a * 2
if self.goal_reached:
coeff = self._evaluate_brakes_stage_coeff(t, [0., 0.], end_t=self.end_time, end_value=0.0)
l_a *= coeff
f_a *= coeff
if coeff is 0.0:
self._stay_still = True
start_coeff = self._evaluate_gait_stage_coeff(t, [0.0])
l_a *= start_coeff
f_a *= start_coeff
l_extension = l_a * math.cos(2 * math.pi / period * t)
f_extension = f_a * math.cos(2 * math.pi / period * t)
initial_pose = self.init_pose
l_swing = -l_extension
swing = -f_extension
pose = np.array([0.0, l_extension + action[0], f_extension + action[1],
0.0, l_swing + action[2], swing + action[3],
0.0, l_swing + action[4], swing + action[5],
0.0, l_extension + action[6], f_extension + action[7]])
signal = initial_pose + pose
return signal
def _transform_action_to_motor_command(self, action):
if self._stay_still:
return self.init_pose
t = self.rex.GetTimeSinceReset()
self._check_target_position(t)
action = self._signal(t, action)
action = super(RexWalkEnv, self)._transform_action_to_motor_command(action)
return action
def is_fallen(self):
"""Decide whether the rex has fallen.
If the up directions between the base and the world is large (the dot
product is smaller than 0.85), the rex is considered fallen.
Returns:
Boolean value that indicates whether the rex has fallen.
"""
orientation = self.rex.GetBaseOrientation()
rot_mat = self._pybullet_client.getMatrixFromQuaternion(orientation)
local_up = rot_mat[6:]
return np.dot(np.asarray([0, 0, 1]), np.asarray(local_up)) < 0.85
def _get_true_observation(self):
"""Get the true observations of this environment.
It includes the roll, the error between current pitch and desired pitch,
roll dot and pitch dot of the base.
Returns:
Python
Tab Width: 8
Ln 1, Col 1
INS


Activities
Text Editor
Apr 27 6:10 PM
Open
walk_env.py
~/anaconda3/envs/rex/lib/python3.7/site-packages/rex_gym/envs/gym
Save
return action
def is_fallen(self):
"""Decide whether the rex has fallen.
If the up directions between the base and the world is large (the dot
product is smaller than 0.85), the rex is considered fallen.
Returns:
Boolean value that indicates whether the rex has fallen.
"""
orientation = self.rex.GetBaseOrientation()
rot_mat = self._pybullet_client.getMatrixFromQuaternion(orientation)
local_up = rot_mat[6:]
return np.dot(np.asarray([0, 0, 1]), np.asarray(local_up)) < 0.85
def _get_true_observation(self):
"""Get the true observations of this environment.
It includes the roll, the error between current pitch and desired pitch,
roll dot and pitch dot of the base.
Returns:
The observation list.
"""
observation = []
roll, pitch, _ = self.rex.GetTrueBaseRollPitchYaw()
roll_rate, pitch_rate, _ = self.rex.GetTrueBaseRollPitchYawRate()
observation.extend([roll, pitch, roll_rate, pitch_rate])
self._true_observation = np.array(observation)
return self._true_observation
def _get_observation(self):
observation = []
roll, pitch, _ = self.rex.GetBaseRollPitchYaw()
roll_rate, pitch_rate, _ = self.rex.GetBaseRollPitchYawRate()
observation.extend([roll, pitch, roll_rate, pitch_rate])
self._observation = np.array(observation)
return self._observation
def _get_observation_upper_bound(self):
"""Get the upper bound of the observation.
Returns:
The upper bound of an observation. See GetObservation() for the details
of each element of an observation.
"""
upper_bound = np.zeros(self._get_observation_dimension())
upper_bound[0:2] = 2 * math.pi # Roll, pitch, yaw of the base.
upper_bound[2:4] = 2 * math.pi / self._time_step # Roll, pitch, yaw rate.
return upper_bound
def _get_observation_lower_bound(self):
lower_bound = -self._get_observation_upper_bound()
return lower_bound
Python
Tab Width: 8
Ln 1, Col 1
INS

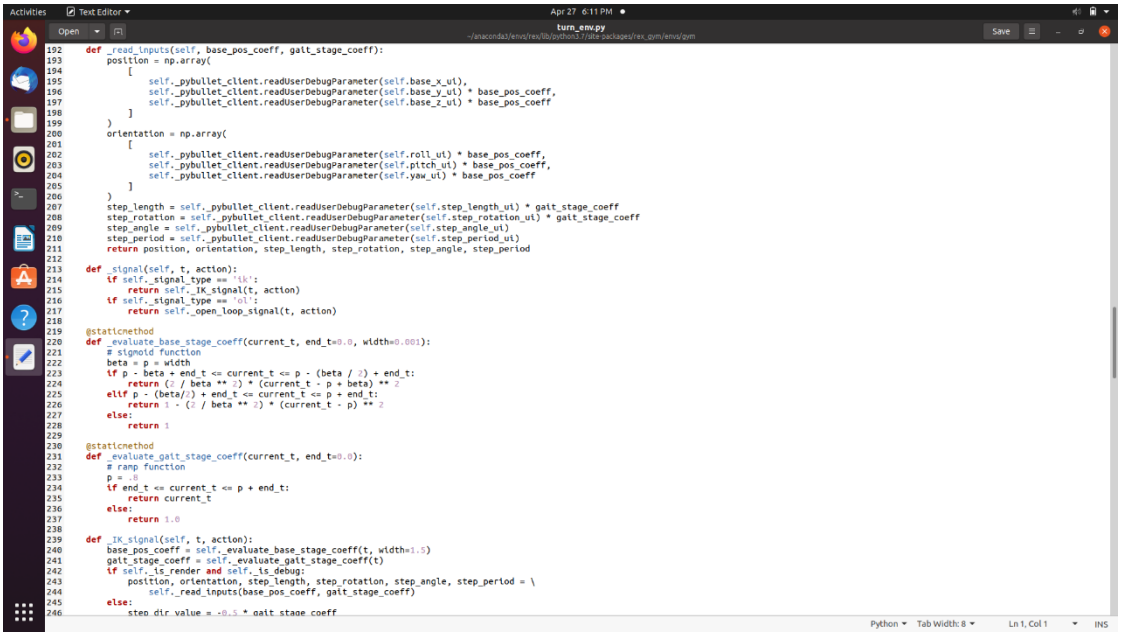
Turn Environment:





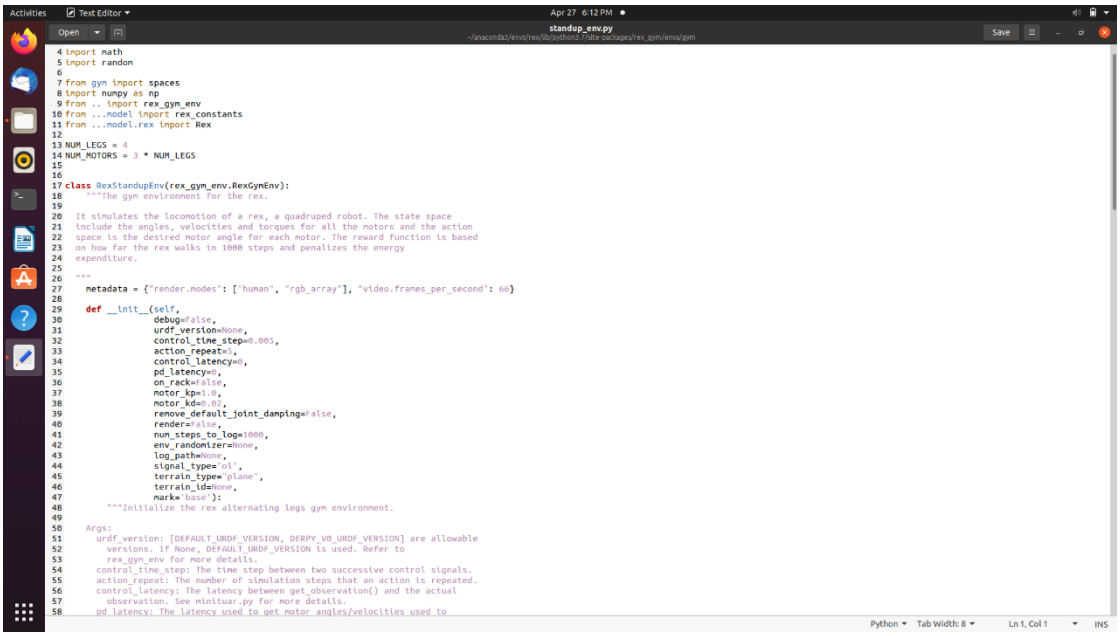


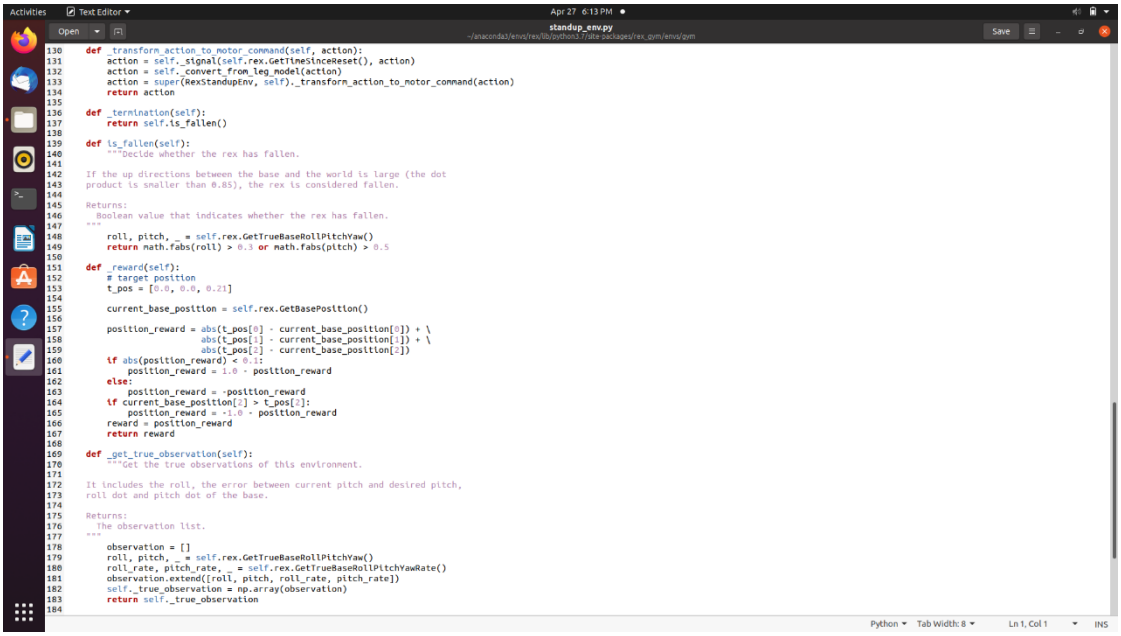
```
    def reset(self):
        self.goal_reached = False
        self.is_terminating = False
        self._stay_still = False
        self.init_pose = rex_constants.INIT_POSES["stand"]
        if self._signal_type == 'ol':
            self.init_pose = rex_constants.INIT_POSES["stand_ol"]
        super(RexTurnEnv, self).reset(initial_motor_angles=self.init_pose, reset_duration=0.5)
        if not self._target_orient or self._random_orient_target:
            self._target_orient = random.uniform(0.2, 0)
            self._random_orient_target = True
        if self._on_rack:
            # on rack debug simulation
            self._init_orient = 2.1
            position = self.rex.init_on_rack_position
        else:
            position = self.rex.init_position
            if self._init_orient is None or self._random_orient_start:
                self._init_orient = random.uniform(0.2, 0)
                self._random_orient_start = True
        self.clockwise = self._solve_direction()
        if self._is_debug:
            print(f"Start Orientation: {self._init_orient}, Target Orientation: {self._target_orient}")
            print("Turning right") if self.clockwise else print("Turning left")
        if self._is_render and self._signal_type == 'ik':
            if self.load_ui:
                self.setup_ui()
                self.load_ui = False
        self._load_cube(self._target_orient)
        q = self.pybullet_client.getQuaternionFromEuler([0, 0, self._init_orient])
        self.pybullet_client.resetBasePositionAndOrientation(self.rex.quadruped, position, q)
        return self._get_observation()

    def setup_ui(self):
        self.base_x_ui = self._pybullet_client.addUserDebugParameter("base_x",
                                                                      self._ranges['base_x'][0],
                                                                      self._ranges['base_x'][1],
                                                                      0.000)
        self.base_y_ui = self._pybullet_client.addUserDebugParameter("base_y",
                                                                      self._ranges['base_y'][0],
                                                                      self._ranges['base_y'][1],
                                                                      self._ranges['base_y'][2])
        self.base_z_ui = self._pybullet_client.addUserDebugParameter("base_z",
                                                                      self._ranges['base_z'][0],
                                                                      self._ranges['base_z'][1],
                                                                      self._ranges['base_z'][2])
        self.roll_ui = self._pybullet_client.addUserDebugParameter("roll",
                                                                   self._ranges['roll'][0],
                                                                   self._ranges['roll'][1],
                                                                   self._ranges['roll'][2])
        self.pitch_ui = self._pybullet_client.addUserDebugParameter("pitch",
                                                                    self._ranges['pitch'][0],
                                                                    self._ranges['pitch'][1],
                                                                    self._ranges['pitch'][2])
```



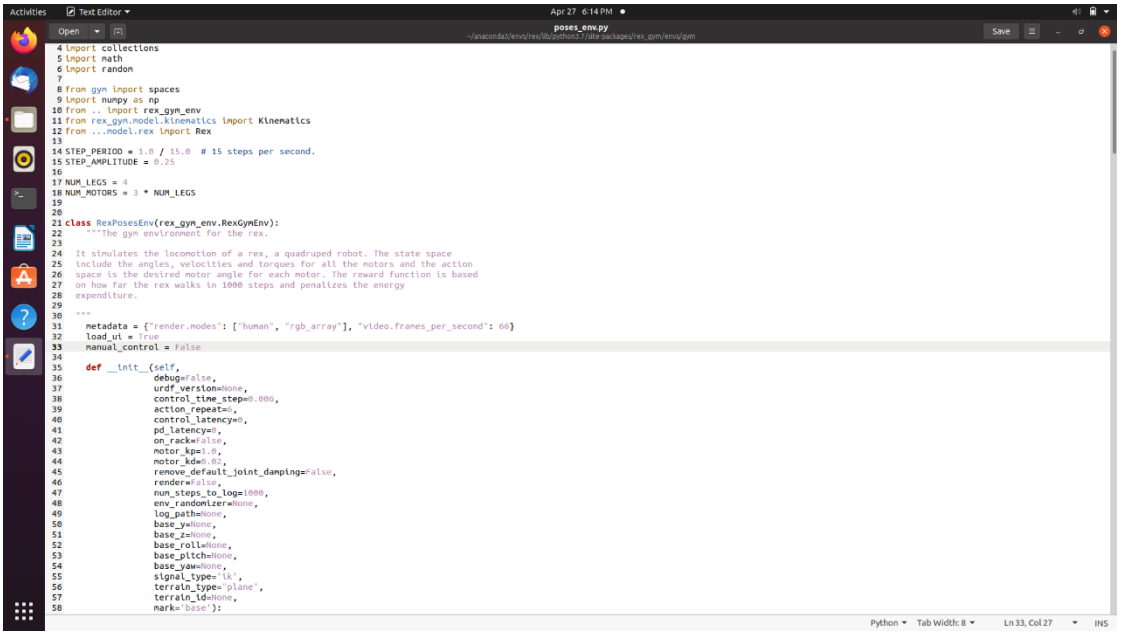
```
    def _read_inputs(self, base_pos_coeff, gait_stage_coeff):
        position = np.array(
            [
                self._pybullet_client.readUserDebugParameter(self.base_x_ui),
                self._pybullet_client.readUserDebugParameter(self.base_y_ui) * base_pos_coeff,
                self._pybullet_client.readUserDebugParameter(self.base_z_ui) * base_pos_coeff
            ]
        )
        orientation = np.array(
            [
                self._pybullet_client.readUserDebugParameter(self.roll_ui) * base_pos_coeff,
                self._pybullet_client.readUserDebugParameter(self.pitch_ui) * base_pos_coeff,
                self._pybullet_client.readUserDebugParameter(self.yaw_ui) * base_pos_coeff
            ]
        )
        step_length = self._pybullet_client.readUserDebugParameter(self.step_length_ui) * gait_stage_coeff
        step_rotation = self._pybullet_client.readUserDebugParameter(self.step_rotation_ui) * gait_stage_coeff
        step_angle = self._pybullet_client.readUserDebugParameter(self.step_angle_ui)
        step_period = self._pybullet_client.readUserDebugParameter(self.step_period_ui)
        return position, orientation, step_length, step_rotation, step_angle, step_period

    def _signal(self, t, action):
        if self._signal_type == 'ik':
            return self._IK_signal(t, action)
        if self._signal_type == 'ol':
            return self._open_loop_signal(t, action)

    @staticmethod
    def _evaluate_base_stage_coeff(current_t, end_t=0.0, width=0.001):
        # sigmoid function
        beta = p * width
        if p - beta + end_t <= current_t <= p - (beta / 2) + end_t:
            return (2 / beta ** 2) * (current_t - p + beta) ** 2
        elif p - (beta/2) + end_t <= current_t <= p + end_t:
            return 1 - (2 / beta ** 2) * (current_t - p) ** 2
        else:
            return 1

    @staticmethod
    def _evaluate_gait_stage_coeff(current_t, end_t=0.0):
        # ramp function
        p = 1.0
        if end_t <= current_t <= p + end_t:
            return current_t
        else:
            return 1.0

    def _IK_signal(self, t, action):
        base_pos_coeff = self._evaluate_base_stage_coeff(t, width=1.5)
        gait_stage_coeff = self._evaluate_gait_stage_coeff(t)
        if self._is_render and self._is_debug:
            position, orientation, step_length, step_rotation, step_angle, step_period = \
                self._read_inputs(base_pos_coeff, gait_stage_coeff)
        else:
            step_dir_value = -0.5 * gait_stage_coeff
```

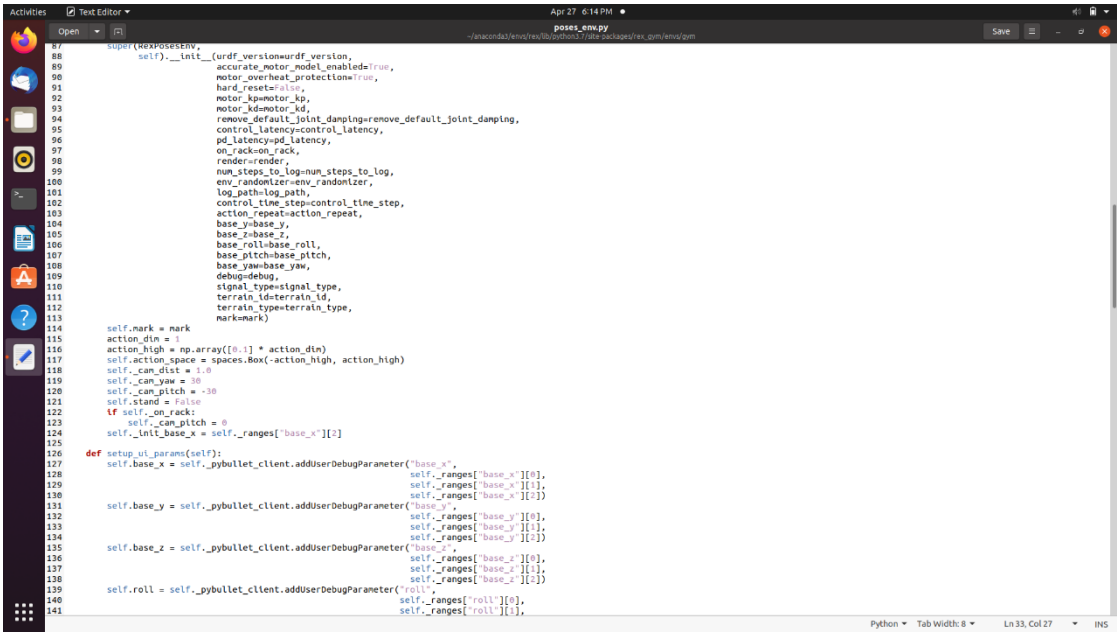
Standup Environment:



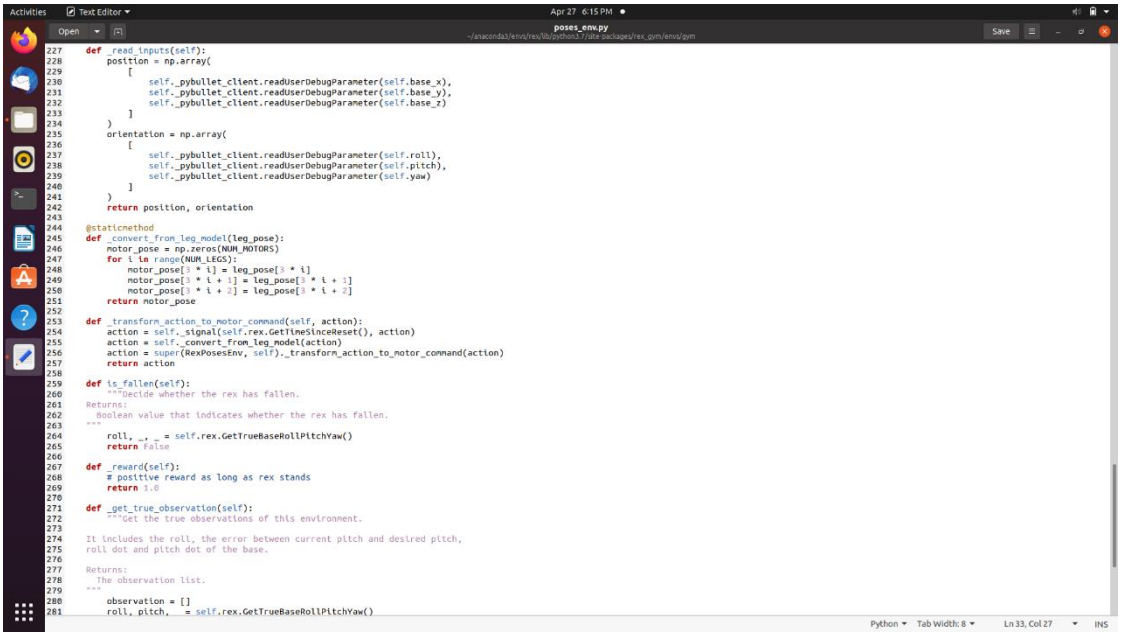


Posing Environment:



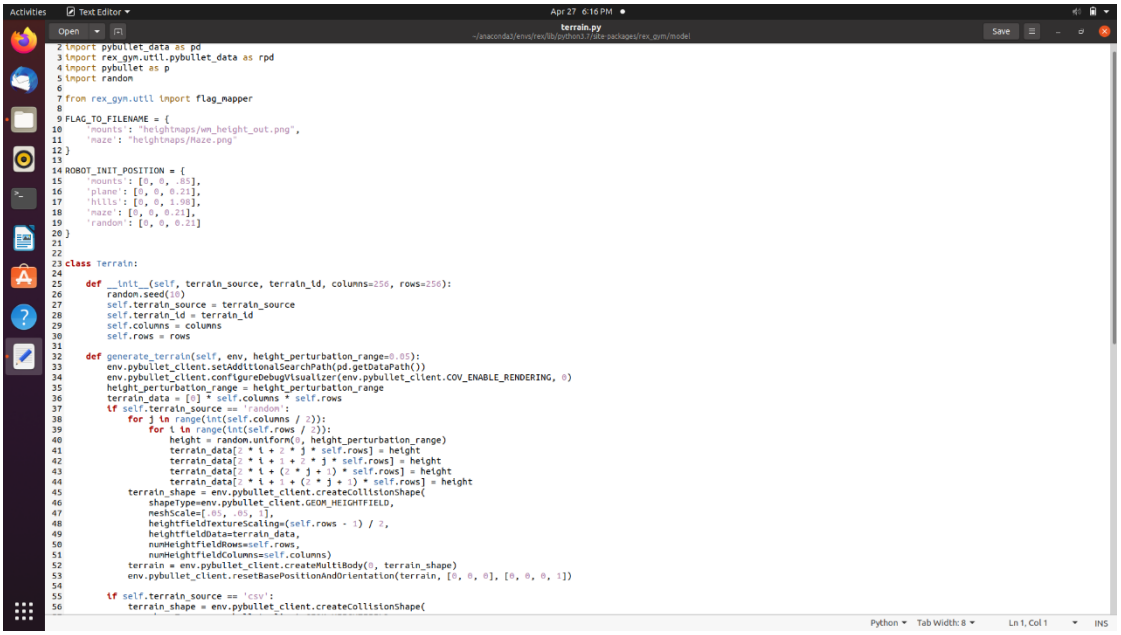

Activities
Text Editor
Apr 27 6:14 PM
Open
poses_env.py
~/anaconda3/envs/rex/lib/python3.7/site-packages/rex_gym/envs/gym
Save
super(RexPosesEnv,
self).__init__(urdf_version=urdf_version,
accurate_motor_model_enabled=True,
motor_overheat_protection=True,
hard_reset=False,
motor_kp=motor_kp,
motor_kd=motor_kd,
remove_default_joint_damping=remove_default_joint_damping,
control_latency=control_latency,
pd_latency=pd_latency,
on_rack=on_rack,
render=render,
num_steps_to_log=num_steps_to_log,
env_randomizer=env_randomizer,
log_path=log_path,
control_time_step=control_time_step,
action_repeat=action_repeat,
base_y=base_y,
base_z=base_z,
base_roll=base_roll,
base_pitch=base_pitch,
base_yaw=base_yaw,
debug=debug,
signal_type=signal_type,
terrain_id=terrain_id,
terrain_type=terrain_type,
mark=mark)
self.mark = mark
action_dim = 1
action_high = np.array([0.1] * action_dim)
self.action_space = spaces.Box(-action_high, action_high)
self._cam_dist = 1.0
self._cam_yaw = 30
self._cam_pitch = -30
self.stand = False
if self._on_rack:
self._cam_pitch = 0
self._init_base_x = self._ranges["base_x"][2]
def setup_ui_params(self):
self.base_x = self._pybullet_client.addUserDebugParameter("base_x",
self._ranges["base_x"][0],
self._ranges["base_x"][1],
self._ranges["base_x"][2])
self.base_y = self._pybullet_client.addUserDebugParameter("base_y",
self._ranges["base_y"][0],
self._ranges["base_y"][1],
self._ranges["base_y"][2])
self.base_z = self._pybullet_client.addUserDebugParameter("base_z",
self._ranges["base_z"][0],
self._ranges["base_z"][1],
self._ranges["base_z"][2])
self.roll = self._pybullet_client.addUserDebugParameter("roll",
self._ranges["roll"][0],
self._ranges["roll"][1],
Python
Tab Width: 8
Ln 33, Col 27
INS


Activities
Text Editor
Apr 27 6:14 PM
Open
poses_env.py
~/anaconda3/envs/rex/lib/python3.7/site-packages/rex_gym/envs/gym
Save
def reset(self):
super(RexPosesEnv, self).reset()
if self._is_render:
if self.load_ui:
self.setup_ui_params()
self.load_ui = False
self.manual_control = True
else:
if self._base_y is not None or self._base_z is not None or self._base_roll is not None \
or self._base_pitch is not None or self._base_yaw is not None:
self.fill_next_pose_and_target()
else:
self.next_pose = self._queue.popleft()
# requeue element
self._queue.append(self.next_pose)
self.target_value = random.uniform(self._ranges[self.next_pose][0], self._ranges[self.next_pose][1])
self.values = self._ranges.copy()
return self._get_observation()
def fill_next_pose_and_target(self):
if self._base_y != 0.0:
self.next_pose = "base_y"
self.target_value = self._base_y
elif self._base_z != 0.0:
self.next_pose = "base_z"
self.target_value = self._base_z
elif self._base_roll != 0.0:
self.next_pose = "roll"
self.target_value = self._base_roll
elif self._base_pitch != 0.0:
self.next_pose = "pitch"
self.target_value = self._base_pitch
else:
self.next_pose = "yaw"
self.target_value = self._base_yaw
@staticmethod
def _evaluate_stage_coefficient(current_t, action, end_t=0.0):
# ramp function
p = 0.8 + action[0]
if end_t <= current_t <= p + end_t:
return current_t
else:
return 1.0
def _signal(self, t, action):
if not self.manual_control:
stage_coeff = self._evaluate_stage_coefficient(t, action)
staged_value = self.target_value * stage_coeff
self.values[self.next_pose] = (self.values[self.next_pose][0],
self.values[self.next_pose][1],
staged_value)
self.position = np.array([
self.values["base_x"][2],
Python
Tab Width: 8
Ln 33, Col 27
INS

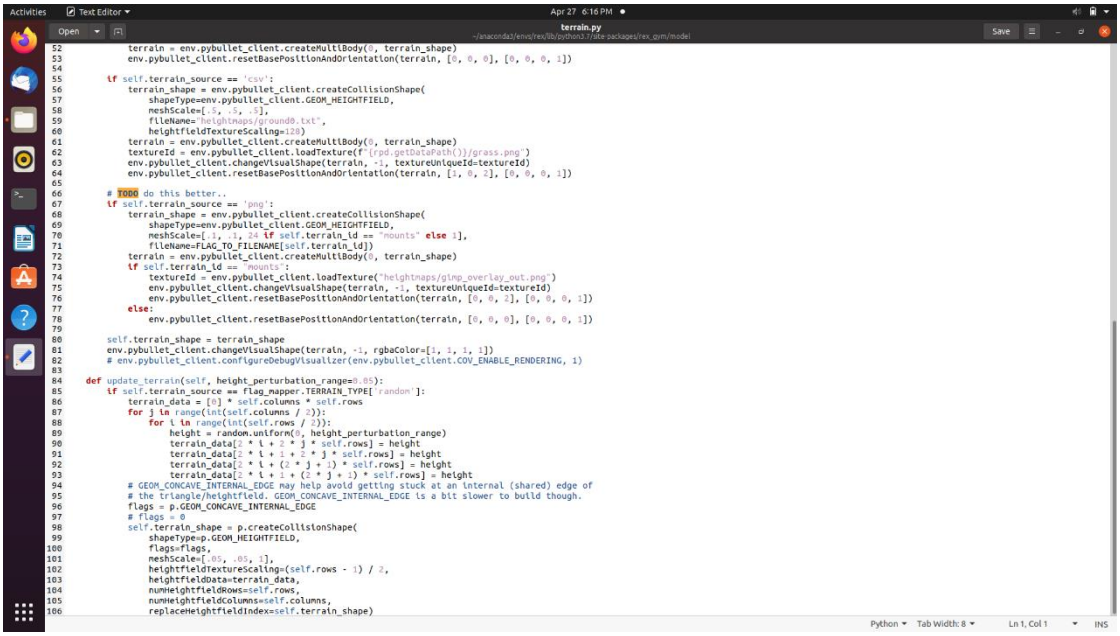


```
    def _read_inputs(self):
        position = np.array(
            [
                self._pybullet_client.readUserDebugParameter(self.base_x),
                self._pybullet_client.readUserDebugParameter(self.base_y),
                self._pybullet_client.readUserDebugParameter(self.base_z)
            ]
        )
        orientation = np.array(
            [
                self._pybullet_client.readUserDebugParameter(self.roll),
                self._pybullet_client.readUserDebugParameter(self.pitch),
                self._pybullet_client.readUserDebugParameter(self.yaw)
            ]
        )
        return position, orientation

    @staticmethod
    def _convert_from_leg_model(leg_pose):
        motor_pose = np.zeros(NUM_MOTORS)
        for i in range(NUM_LEGS):
            motor_pose[3 * i] = leg_pose[3 * i]
            motor_pose[3 * i + 1] = leg_pose[3 * i + 1]
            motor_pose[3 * i + 2] = leg_pose[3 * i + 2]
        return motor_pose

    def _transform_action_to_motor_command(self, action):
        action = self._signal(self.rex.GetTimeSinceReset(), action)
        action = self._convert_from_leg_model(action)
        action = super(RexPosesEnv, self)._transform_action_to_motor_command(action)
        return action

    def is_fallen(self):
        """Decide whether the rex has fallen.
    Returns:
      Boolean value that indicates whether the rex has fallen.
    """
        roll, _, _ = self.rex.GetTrueBaseRollPitchYaw()
        return False

    def _reward(self):
        # positive reward as long as rex stands
        return 1.0

    def _get_true_observation(self):
        """Get the true observations of this environment.

    It includes the roll, the error between current pitch and desired pitch,
    roll dot and pitch dot of the base.

    Returns:
      The observation list.
    """
        observation = []
        roll, pitch,  = self.rex.GetTrueBaseRollPitchYaw()
```

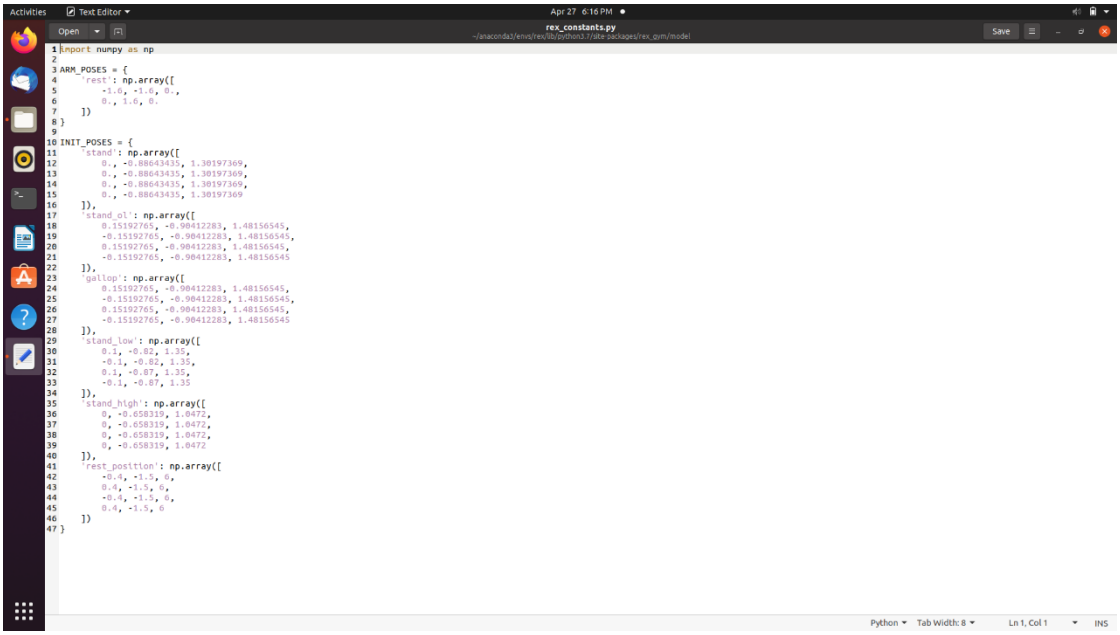
Terrain Implementation:



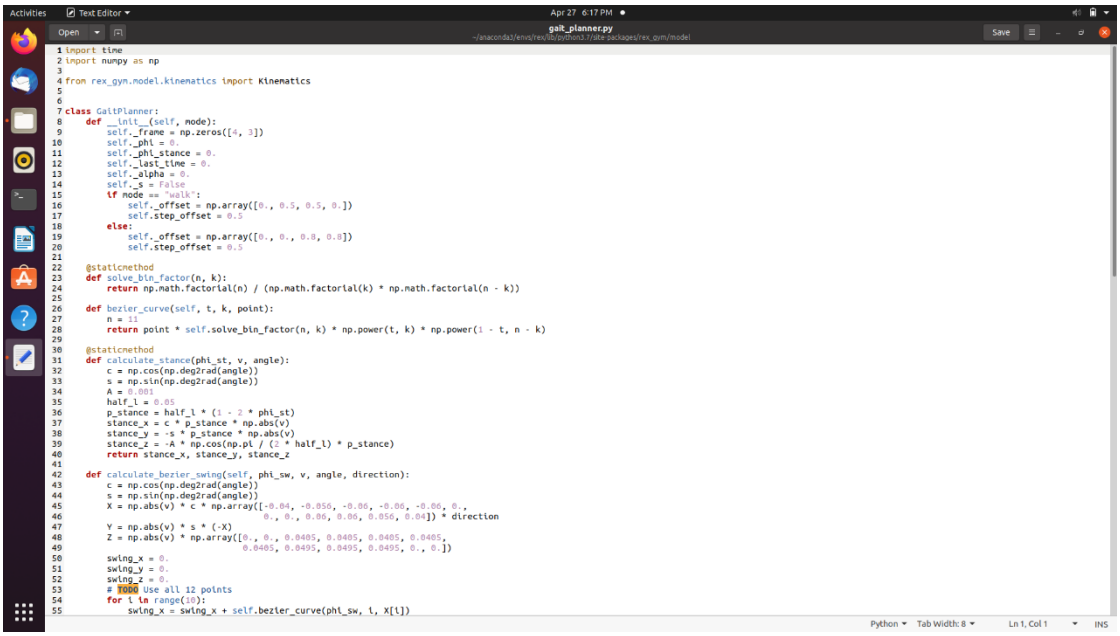
```
import pybullet_data as pd
import rex_gym.util.pybullet_data as rpd
import pybullet as p
import random

from rex_gym.util import flag_mapper

FLAG_TO_FILENAME = {
    'mounts': "heightmaps/wm_height_out.png",
    'maze': "heightmaps/Maze.png"
}

ROBOT_INIT_POSITION = {
    'mounts': [0, 0, .85],
    'plane': [0, 0, 0.21],
    'hills': [0, 0, 1.98],
    'maze': [0, 0, 0.21],
    'random': [0, 0, 0.21]
}


class Terrain:

    def __init__(self, terrain_source, terrain_id, columns=256, rows=256):
        random.seed(10)
        self.terrain_source = terrain_source
        self.terrain_id = terrain_id
        self.columns = columns
        self.rows = rows

    def generate_terrain(self, env, height_perturbation_range=0.05):
        env.pybullet_client.setAdditionalSearchPath(pd.getDataPath())
        env.pybullet_client.configureDebugVisualizer(env.pybullet_client.COV_ENABLE_RENDERING, 0)
        height_perturbation_range = height_perturbation_range
        terrain_data = [0] * self.columns * self.rows
        if self.terrain_source == 'random':
            for j in range(int(self.columns / 2)):
                for i in range(int(self.rows / 2)):
                    height = random.uniform(0, height_perturbation_range)
                    terrain_data[2 * i + 2 * j * self.rows] = height
                    terrain_data[2 * i + 1 + 2 * j * self.rows] = height
                    terrain_data[2 * i + (2 * j + 1) * self.rows] = height
                    terrain_data[2 * i + 1 + (2 * j + 1) * self.rows] = height
            terrain_shape = env.pybullet_client.createCollisionShape(
                shapeType=env.pybullet_client.GEOM_HEIGHTFIELD,
                meshScale=[.05, .05, 1],
                heightfieldTextureScaling=(self.rows - 1) / 2,
                heightfieldData=terrain_data,
                numHeightfieldRows=self.rows,
                numHeightfieldColumns=self.columns)
            terrain = env.pybullet_client.createMultiBody(0, terrain_shape)
            env.pybullet_client.resetBasePositionAndOrientation(terrain, [0, 0, 0], [0, 0, 0, 1])

        if self.terrain_source == 'csv':
            terrain_shape = env.pybullet_client.createCollisionShape(
```



Rex Constant:



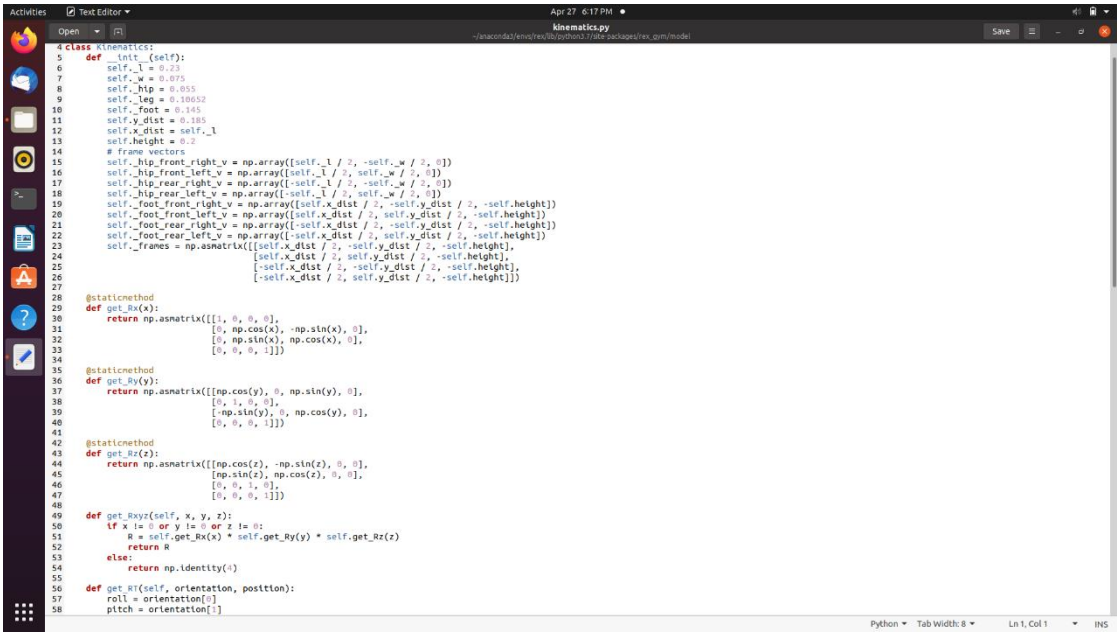
Gait Planner:



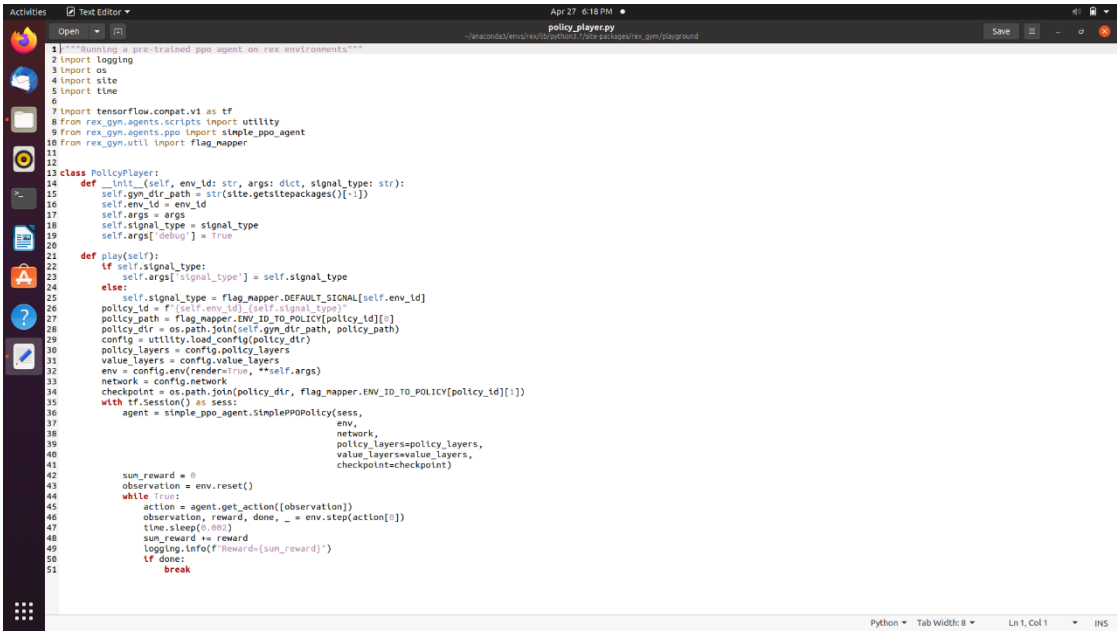
```python
import time
import numpy as np

from rex_gym.model.kinematics import Kinematics


class GaitPlanner:
    def __init__(self, mode):
        self._frame = np.zeros([4, 3])
        self._phi = 0.
        self._phi_stance = 0.
        self._last_time = 0.
        self._alpha = 0.
        self._s = False
        if mode == "walk":
            self._offset = np.array([0., 0.5, 0.5, 0.])
            self.step_offset = 0.5
        else:
            self._offset = np.array([0., 0., 0.8, 0.8])
            self.step_offset = 0.5

    @staticmethod
    def solve_bin_factor(n, k):
        return np.math.factorial(n) / (np.math.factorial(k) * np.math.factorial(n - k))

    def bezier_curve(self, t, k, point):
        n = 11
        return point * self.solve_bin_factor(n, k) * np.power(t, k) * np.power(1 - t, n - k)

    @staticmethod
    def calculate_stance(phi_st, v, angle):
        c = np.cos(np.deg2rad(angle))
        s = np.sin(np.deg2rad(angle))
        A = 0.001
        half_l = 0.05
        p_stance = half_l * (1 - 2 * phi_st)
        stance_x = c * p_stance * np.abs(v)
        stance_y = -s * p_stance * np.abs(v)
        stance_z = -A * np.cos(np.pi / (2 * half_l) * p_stance)
        return stance_x, stance_y, stance_z

    def calculate_bezier_swing(self, phi_sw, v, angle, direction):
        c = np.cos(np.deg2rad(angle))
        s = np.sin(np.deg2rad(angle))
        X = np.abs(v) * c * np.array([-0.04, -0.050, -0.06, -0.06, -0.06, 0.,
                                      0., 0., 0.06, 0.06, 0.056, 0.04]) * direction

        Y = np.abs(v) * s * (-X)
        Z = np.abs(v) * np.array([0., 0., 0.0405, 0.0405, 0.0405, 0.0405,
                                  0.0405, 0.0495, 0.0495, 0.0495, 0., 0.])

        swing_x = 0.
        swing_y = 0.
        swing_z = 0.
        # TODO Use all 12 points
        for i in range(10):
            swing_x = swing_x + self.bezier_curve(phi_sw, i, X[i])
```

```python
    def step_trajectory(self, phi, v, angle, w_rot, center_to_foot, direction):
        if phi >= 1:
            phi = phi - 1.
        r = np.sqrt(center_to_foot[0] ** 2 + center_to_foot[1] ** 2)
        foot_angle = np.arctan2(center_to_foot[1], center_to_foot[0])
        if w_rot >= 0.:
            circle_trajectory = 90. - np.rad2deg(foot_angle - self._alpha)
        else:
            circle_trajectory = 270. - np.rad2deg(foot_angle - self._alpha)

        if phi <= self.step_offset:
            # stance phase
            phi_stance = phi / self.step_offset
            stepX_long, stepY_long, stepZ_long = self.calculate_stance(phi_stance, v, angle)
            stepX_rot, stepY_rot, stepZ_rot = self.calculate_stance(phi_stance, w_rot, circle_trajectory)
        else:
            # swing phase
            phiSwing = (phi - self.step_offset) / (1 - self.step_offset)
            stepX_long, stepY_long, stepZ_long = self.calculate_bezier_swing(phiSwing, v, angle, direction)
            stepX_rot, stepY_rot, stepZ_rot = self.calculate_bezier_swing(phiSwing, w_rot, circle_trajectory, direction)
        if center_to_foot[1] > 0:
            if stepX_rot < 0:
                self._alpha = -np.arctan2(np.sqrt(stepX_rot ** 2 + stepY_rot ** 2), r)
            else:
                self._alpha = np.arctan2(np.sqrt(stepX_rot ** 2 + stepY_rot ** 2), r)
        else:
            if stepX_rot < 0:
                self._alpha = np.arctan2(np.sqrt(stepX_rot ** 2 + stepY_rot ** 2), r)
            else:
                self._alpha = -np.arctan2(np.sqrt(stepX_rot ** 2 + stepY_rot ** 2), r)
        coord = np.empty(3)
        coord[0] = stepX_long + stepX_rot
        coord[1] = stepY_long + stepY_rot
        coord[2] = stepZ_long + stepZ_rot
        return coord

    def loop(self, v, angle, w_rot, t, direction, frames=None):
        if frames is None:
            k_obj = Kinematics()
            x_dist = k_obj.x_dist
            y_dist = k_obj.y_dist
            height = k_obj.height
            frames = np.asmatrix([[x_dist / 2, -y_dist / 2, -height],
                                  [x_dist / 2, y_dist / 2, -height],
                                  [-x_dist / 2, -y_dist / 2, -height],
                                  [-x_dist / 2, y_dist / 2, -height]])
        if t <= 0.01:
            t = 0.01
        if self._phi >= 0.99:
            self._last_time = time.time()
        self._phi = (time.time() - self._last_time) / t
        step_coord = self.step_trajectory(self._phi + self._offset[0], v, angle, w_rot,
                                          np.squeeze(np.asarray(frames[0, :])), direction)  # FR
        self._frame[0, 0] = frames[0, 0] + step_coord[0]
        self._frame[0, 1] = frames[0, 1] + step_coord[1]
```

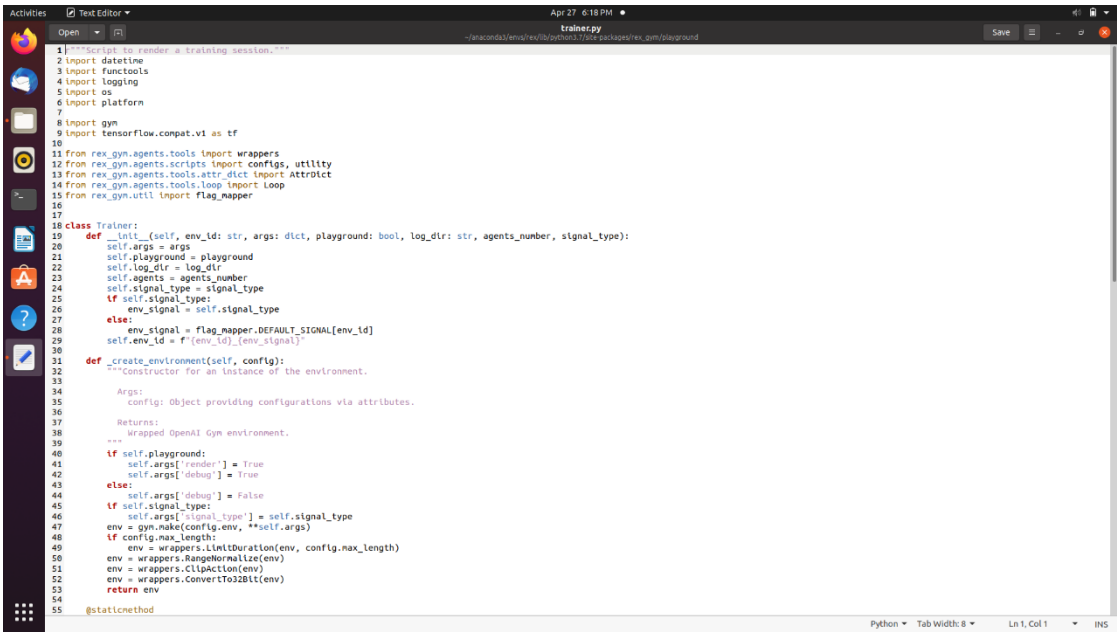
Kinematics:

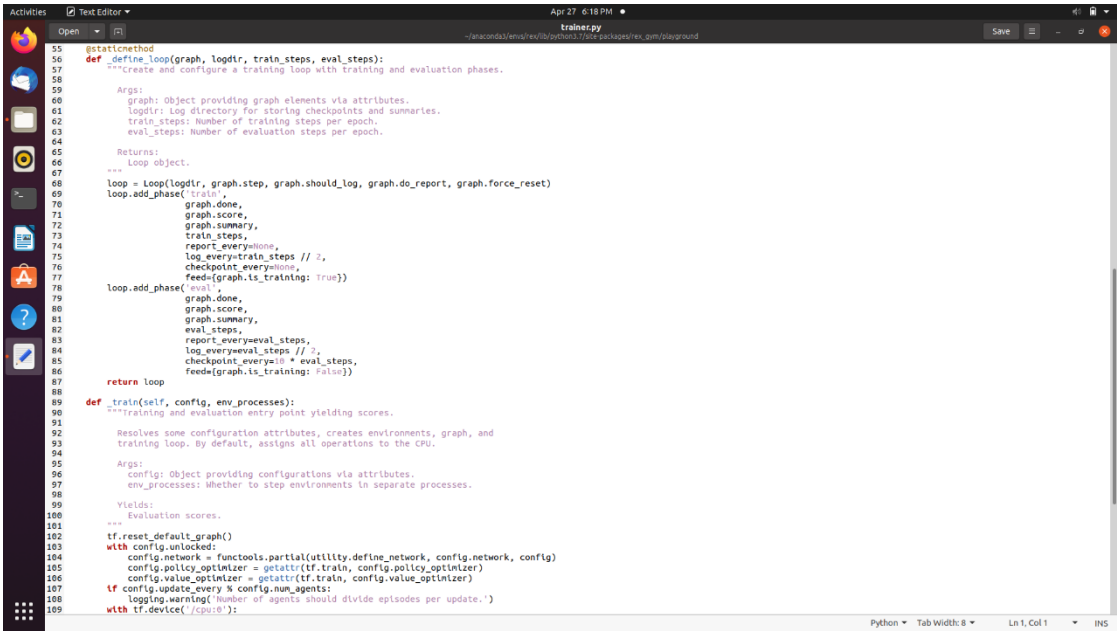


Policy Player:



Trainer:

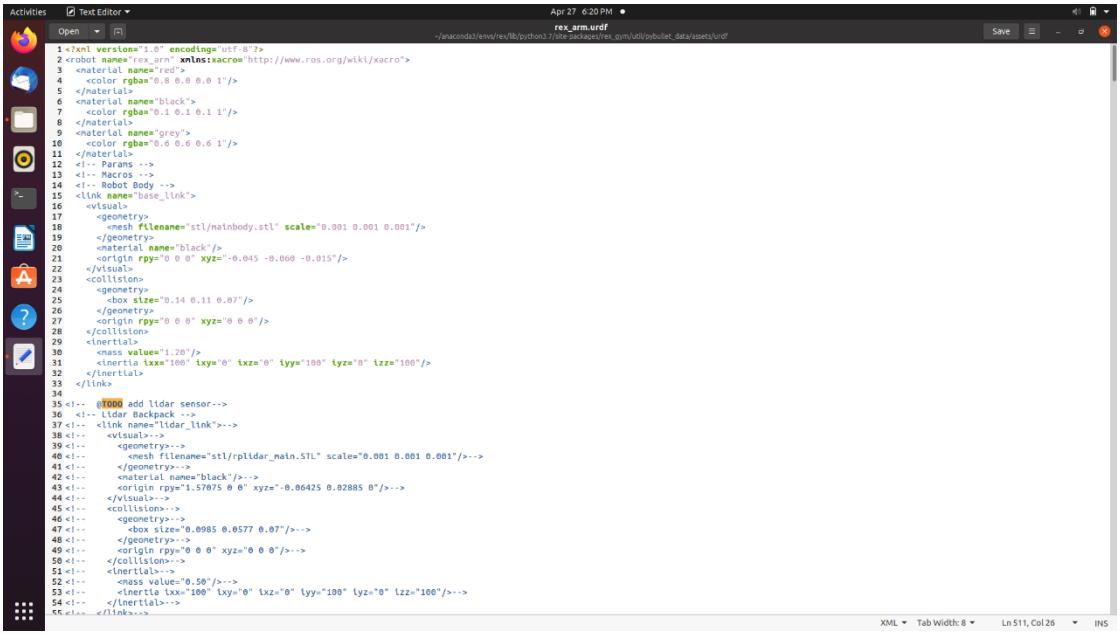

Activities
Text Editor
Apr 27 6:18 PM
Open
trainer.py
~/anaconda3/envs/rex/lib/python3.7/site-packages/rex_gym/playground
Save
"""Script to render a training session."""
import datetime
import functools
import logging
import os
import platform
import gym
import tensorflow.compat.v1 as tf
from rex_gym.agents.tools import wrappers
from rex_gym.agents.scripts import configs, utility
from rex_gym.agents.tools.attr_dict import AttrDict
from rex_gym.agents.tools.loop import Loop
from rex_gym.util import flag_mapper
class Trainer:
def __init__(self, env_id: str, args: dict, playground: bool, log_dir: str, agents_number, signal_type):
self.args = args
self.playground = playground
self.log_dir = log_dir
self.agents = agents_number
self.signal_type = signal_type
if self.signal_type:
env_signal = self.signal_type
else:
env_signal = flag_mapper.DEFAULT_SIGNAL[env_id]
self.env_id = f'{env_id}_{env_signal}'
def _create_environment(self, config):
"""Constructor for an instance of the environment.
Args:
config: Object providing configurations via attributes.
Returns:
Wrapped OpenAI Gym environment.
"""
if self.playground:
self.args['render'] = True
self.args['debug'] = True
else:
self.args['debug'] = False
if self.signal_type:
self.args['signal_type'] = self.signal_type
env = gym.make(config.env, **self.args)
if config.max_length:
env = wrappers.LimitDuration(env, config.max_length)
env = wrappers.RangeNormalize(env)
env = wrappers.ClipAction(env)
env = wrappers.ConvertTo32Bit(env)
return env
@staticmethod
Python
Tab Width: 8
Ln 1, Col 1
INS


Activities
Text Editor
Apr 27 6:18 PM
Open
trainer.py
~/anaconda3/envs/rex/lib/python3.7/site-packages/rex_gym/playground
Save
@staticmethod
def _define_loop(graph, logdir, train_steps, eval_steps):
"""Create and configure a training loop with training and evaluation phases.
Args:
graph: Object providing graph elements via attributes.
logdir: Log directory for storing checkpoints and summaries.
train_steps: Number of training steps per epoch.
eval_steps: Number of evaluation steps per epoch.
Returns:
Loop object.
"""
loop = Loop(logdir, graph.step, graph.should_log, graph.do_report, graph.force_reset)
loop.add_phase('train',
graph.done,
graph.score,
graph.summary,
train_steps,
report_every=None,
log_every=train_steps // 2,
checkpoint_every=None,
feed={graph.is_training: True})
loop.add_phase('eval',
graph.done,
graph.score,
graph.summary,
eval_steps,
report_every=eval_steps,
log_every=eval_steps // 2,
checkpoint_every=10 * eval_steps,
feed={graph.is_training: False})
return loop
def _train(self, config, env_processes):
"""Training and evaluation entry point yielding scores.
Resolves some configuration attributes, creates environments, graph, and
training loop. By default, assigns all operations to the CPU.
Args:
config: Object providing configurations via attributes.
env_processes: Whether to step environments in separate processes.
Yields:
Evaluation scores.
"""
tf.reset_default_graph()
with config.unlocked:
config.network = functools.partial(utility.define_network, config.network, config)
config.policy_optimizer = getattr(tf.train, config.policy_optimizer)
config.value_optimizer = getattr(tf.train, config.value_optimizer)
if config.update_every % config.num_agents:
logging.warning('Number of agents should divide episodes per update.')
with tf.device('/cpu:0'):
Python
Tab Width: 8
Ln 1, Col 1
INS

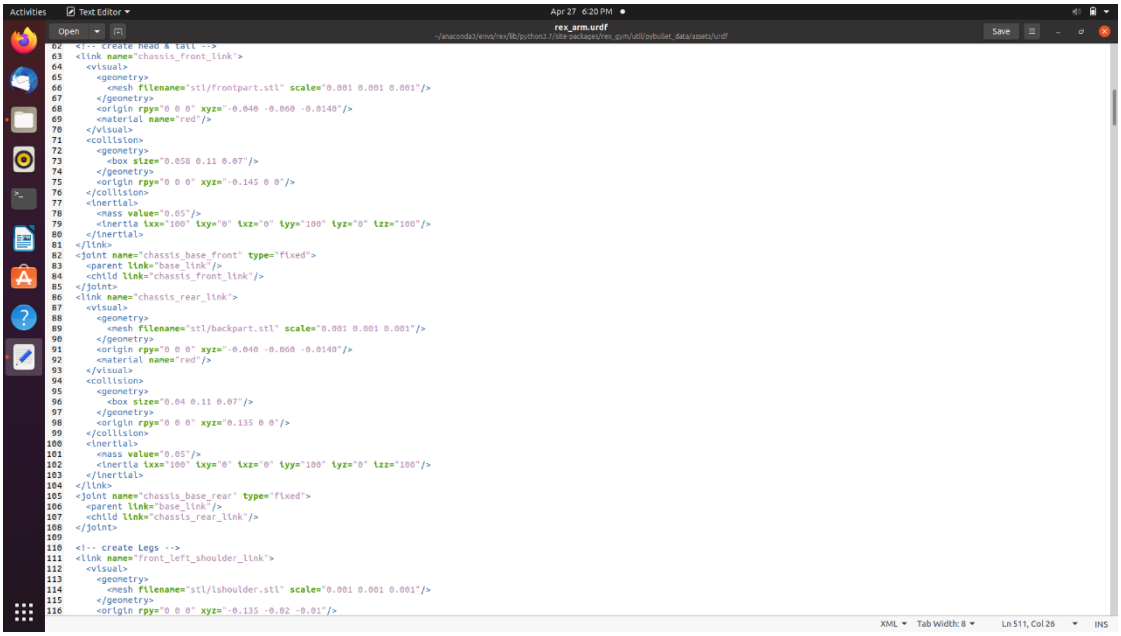
Flag Mapper:


```
ENV_ID_TO_POLICY = {
    'gallop_ol': ('rex_gym/policies/gallop/ol', 'model.ckpt-4000000'),
    'gallop_ik': ('rex_gym/policies/gallop/ik', 'model.ckpt-2000000'),
    'walk_ik': ('rex_gym/policies/walk/ik', 'model.ckpt-2000000'),
    'walk_ol': ('rex_gym/policies/walk/ol', 'model.ckpt-4000000'),
    'standup_ol': ('rex_gym/policies/standup/ol', 'model.ckpt-2000000'),
    'turn_ik': ('rex_gym/policies/turn/ik', 'model.ckpt-2000000'),
    'turn_ol': ('rex_gym/policies/turn/ol', 'model.ckpt-2000000'),
    'poses_ik' : ('rex_gym/policies/poses', 'model.ckpt-2000000')
}

ENV_ID_TO_ENV_NAMES = {
    'gallop': 'RexReactiveEnv',
    'walk': 'RexWalkEnv',
    'turn': 'RexTurnEnv',
    'standup': 'RexStandupEnv',
    'go': 'RexGoEnv',
    'poses': 'RexPosesEnv'
}

DEFAULT_SIGNAL = {
    'gallop': 'ik',
    'walk': 'ik',
    'turn': 'ol',
    'standup': 'ol',
    'go': 'ik',
    'poses': 'ik'
}

TERRAIN_TYPE = {
    'mounts': 'png',
    'maze': 'png',
    'hills': 'csv',
    'random': 'random',
    'plane': 'plane'
}
```

Rex Arm:




```
<?xml version="1.0" encoding="utf-8"?>
<robot name="rex_arm" xmlns:xacro="http://www.ros.org/wiki/xacro">
  <material name="red">
    <color rgba="0.8 0.0 0.0 1"/>
  </material>
  <material name="black">
    <color rgba="0.1 0.1 0.1 1"/>
  </material>
  <material name="grey">
    <color rgba="0.6 0.6 0.6 1"/>
  </material>
  <!-- Params -->
  <!-- Macros -->
  <!-- Robot Body -->
  <link name="base_link">
    <visual>
      <geometry>
        <mesh filename="stl/mainbody.stl" scale="0.001 0.001 0.001"/>
      </geometry>
      <material name="black"/>
      <origin rpy="0 0 0" xyz="-0.045 -0.060 -0.015"/>
    </visual>
    <collision>
      <geometry>
        <box size="0.14 0.11 0.07"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </collision>
    <inertial>
      <mass value="1.20"/>
      <inertia ixx="100" ixy="0" ixz="0" iyy="100" iyz="0" izz="100"/>
    </inertial>
  </link>

<!--  TODO add lidar sensor-->
  <!-- Lidar Backpack -->
<!--  <link name="lidar_link">-->
<!--    <visual>-->
<!--      <geometry>-->
<!--        <mesh filename="stl/rplidar_main.STL" scale="0.001 0.001 0.001"/>-->
<!--      </geometry>-->
<!--      <material name="black"/>-->
<!--      <origin rpy="1.57075 0 0" xyz="-0.06425 0.02885 0"/>-->
<!--    </visual>-->
<!--    <collision>-->
<!--      <geometry>-->
<!--        <box size="0.0985 0.0577 0.07"/>-->
<!--      </geometry>-->
<!--      <origin rpy="0 0 0" xyz="0 0 0"/>-->
<!--    </collision>-->
<!--    <inertial>-->
<!--      <mass value="0.50"/>-->
<!--      <inertia ixx="100" ixy="0" ixz="0" iyy="100" iyz="0" izz="100"/>-->
<!--    </inertial>-->
<!--  </link>-->
```


Activities
Text Editor
Apr 27 6:20 PM
Open
rex_arm.urdf
~/anaconda3/envs/rex/lib/python3.7/site-packages/rex_gym/util/pybullet_data/assets/urdf
Save
<!-- create head & tail -->
<link name="chassis_front_link">
<visual>
<geometry>
<mesh filename="stl/frontpart.stl" scale="0.001 0.001 0.001"/>
</geometry>
<origin rpy="0 0 0" xyz="-0.040 -0.060 -0.0140"/>
<material name="red"/>
</visual>
<collision>
<geometry>
<box size="0.050 0.11 0.07"/>
</geometry>
<origin rpy="0 0 0" xyz="-0.145 0 0"/>
</collision>
<inertial>
<mass value="0.05"/>
<inertia ixx="100" ixy="0" ixz="0" iyy="100" iyz="0" izz="100"/>
</inertial>
</link>
<joint name="chassis_base_front" type="fixed">
<parent link="base_link"/>
<child link="chassis_front_link"/>
</joint>
<link name="chassis_rear_link">
<visual>
<geometry>
<mesh filename="stl/backpart.stl" scale="0.001 0.001 0.001"/>
</geometry>
<origin rpy="0 0 0" xyz="-0.040 -0.060 -0.0140"/>
<material name="red"/>
</visual>
<collision>
<geometry>
<box size="0.04 0.11 0.07"/>
</geometry>
<origin rpy="0 0 0" xyz="0.135 0 0"/>
</collision>
<inertial>
<mass value="0.05"/>
<inertia ixx="100" ixy="0" ixz="0" iyy="100" iyz="0" izz="100"/>
</inertial>
</link>
<joint name="chassis_base_rear" type="fixed">
<parent link="base_link"/>
<child link="chassis_rear_link"/>
</joint>
<!-- create Legs -->
<link name="front_left_shoulder_link">
<visual>
<geometry>
<mesh filename="stl/lshoulder.stl" scale="0.001 0.001 0.001"/>
</geometry>
<origin rpy="0 0 0" xyz="-0.135 -0.02 -0.01"/>
XML
Tab Width: 8
Ln 511, Col 26
INS