



SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

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. |
|---------------------------------------|-----------------|
| Declaration..... | II |
| Certificate..... | III |
| Acknowledgement..... | IV |
| Content Table..... | V-VIII |
| Abbreviations List..... | IX |
| Figures List..... | X |
| Abstract..... | XI |
| | |
| 1. Introduction..... | 12 |
| 1.1 Background..... | 13 |
| 1.2 Objective..... | 13 |
| 1.3 Motivation..... | 14 |
| 1.4 Reinforcement Learning..... | 15 |
| | |
| 2. Literature survey..... | 17 |
| 2.1 Background..... | 18 |
| 2.2 Review in quadrupedal robots..... | 18 |

| | |
|--|-----------|
| 2.3 Summary of the Section..... | 20 |
| 3. Analysis & Design..... | 21 |
| 3.1 Introduction..... | 22 |
| 3.2 Analysis of Requirements..... | 23 |
| 3.2.1 Software Requirement..... | 23 |
| 3.2.2 Hardware Requirement..... | 23 |
| 3.3 Detailed Design..... | 23 |
| 3.3.1 Waterfall Diagram..... | 24 |
| 3.3.2 Dataflow Diagram..... | 25 |
| 3.3.3 UML Class Diagram..... | 26 |
| 3.3.4 Sequence Diagram..... | 26 |
| 3.4 Summary of the Chapter..... | 27 |
| 4. Implementation Details | 28 |
| 4.1 Introduction..... | 29 |
| 4.1.1 Functional Requirements..... | 29 |
| 4.1.2 Non-functional Requirements..... | 29 |
| 4.2 Tool Used..... | 30 |
| 4.2.1 Python..... | 30 |
| 4.2.2 Open AI Gym..... | 31 |

| | |
|---|-----------|
| 4.2.3 Libraries..... | 31 |
| 4.2.3.1 Numpy..... | 31 |
| 4.2.3.2 Sklearn..... | 31 |
| 4.2.3.3 Pandas..... | 31 |
| 4.2.3.4 Pickle..... | 31 |
| 4.2.3.5 Tensorflow..... | 32 |
| 4.2.3.6. Matplotlib..... | 32 |
| 4.3. Proposed Methodology..... | 32 |
| 4.3.1. Algorithm Description..... | 33 |
| 4.3.2. Proposed Architecture..... | 34 |
| 4.3.2.1. Panning of Gait..... | 34 |
| 4.3.2.2. Controlling Gait..... | 35 |
| 4.3.3. Policy Description..... | 35 |
| 4.3.4. Reward Function..... | 36 |
| 4.3.5. PPO Algorithm..... | 38 |
| 4.4. Summary of the Chapter..... | 38 |
| | |
| 5. Experimental Result and Analysis..... | 39 |
| 5.1. Introduction..... | 40 |
| 5.1.1. Simulations of Complex Terrains..... | 40 |
| 5.2. Graphs & Maps..... | 48 |

| | |
|--|--------------|
| 5.2.1. Graph of reward function..... | 48 |
| 5.2.2. Graph of lateral body posture & torque..... | 49 |
| 6. Conclusion & Future Work..... | 50 |
| 6.1. Conclusion..... | 51 |
| 6.2. Future-Work..... | 51 |
| References..... | 52-54 |
| Appendix | |
| Code..... | 55 |

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

| Figure No | Name | Page No. |
|------------------|-------------------------------------|-----------------|
| Figure. 3.3.1 | Waterfall Model | 24 |
| Figure. 3.3.2 | Data Flow Diagram | 25 |
| Figure. 3.3.3 | UML Class Diagram | 26 |
| Figure. 3.3.4 | Sequence Diagram | 26 |
| Figure. 4.3.1 | RL Architecture | 33 |
| Figure. 4.3.2.1 | Planning of Gait | 34 |
| Figure. 4.3.2.2 | Controlling of Gait | 35 |
| Figure. 5.1.1.1 | Flat Plane Terrain | 40 |
| Figure. 5.1.1.2 | Hills Non-flat Terrain | 41 |
| Figure. 5.1.1.3 | Maze Flat terrain | 43 |
| Figure. 5.1.1.4 | Mountain Non-flat Terrain | 44 |
| Figure. 5.1.1.5 | Sand Non-flat Terrain | 46 |
| Figure. 5.4.1.6 | Space Flat Terrain | 47 |
| Figure. 5.2.1 | Graph of Cumulative Reward Function | 48 |
| Figure: 5.2.2 | Graph of Body Posture & Torque | 49 |

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-to-model 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 terrain-aware functionality in this paper, which incorporates state-of-the-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 locomote 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

1.1 Background

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*:

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

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 understanding of the study.

3.3.1. Waterfall Model

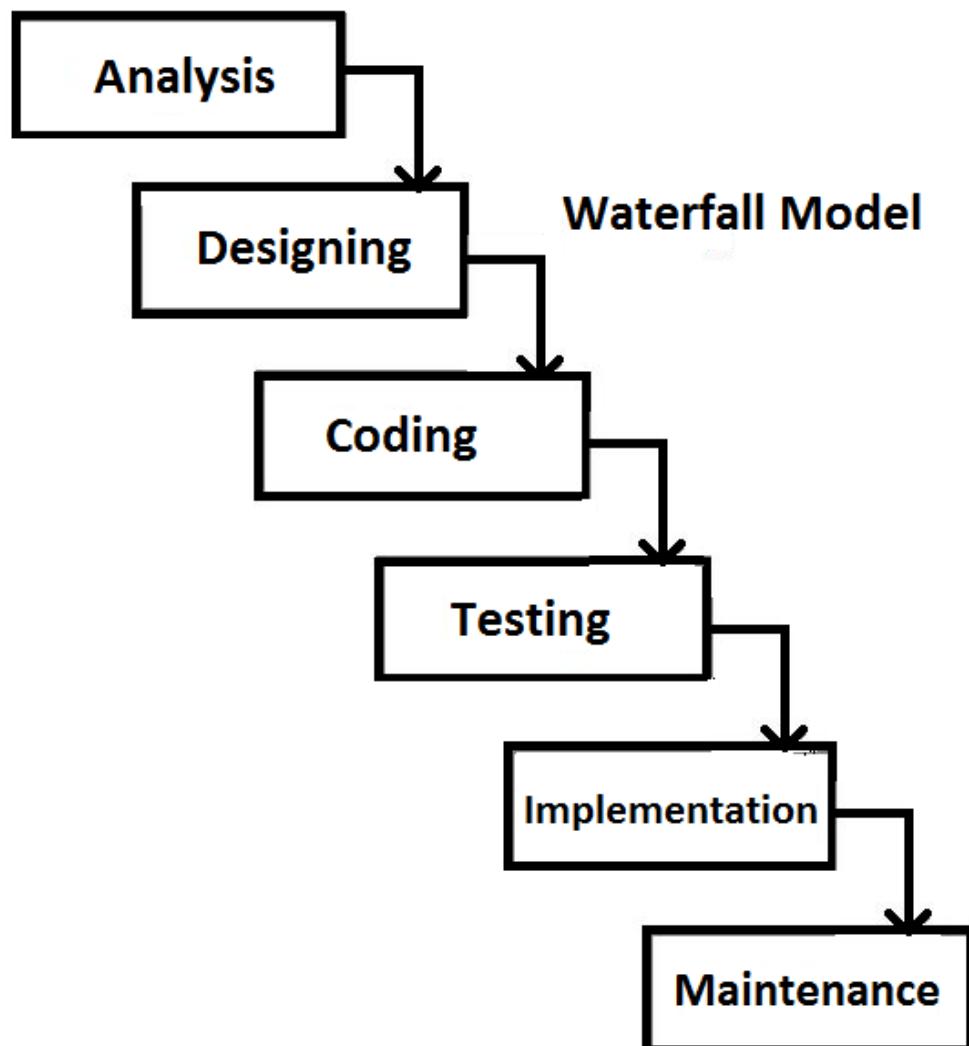


Figure- 3.3.1. shows how the workflow of project is processed by the system i.e., the step-by-step blocks of the data.

3.3.2. Data-Flow-Diagram

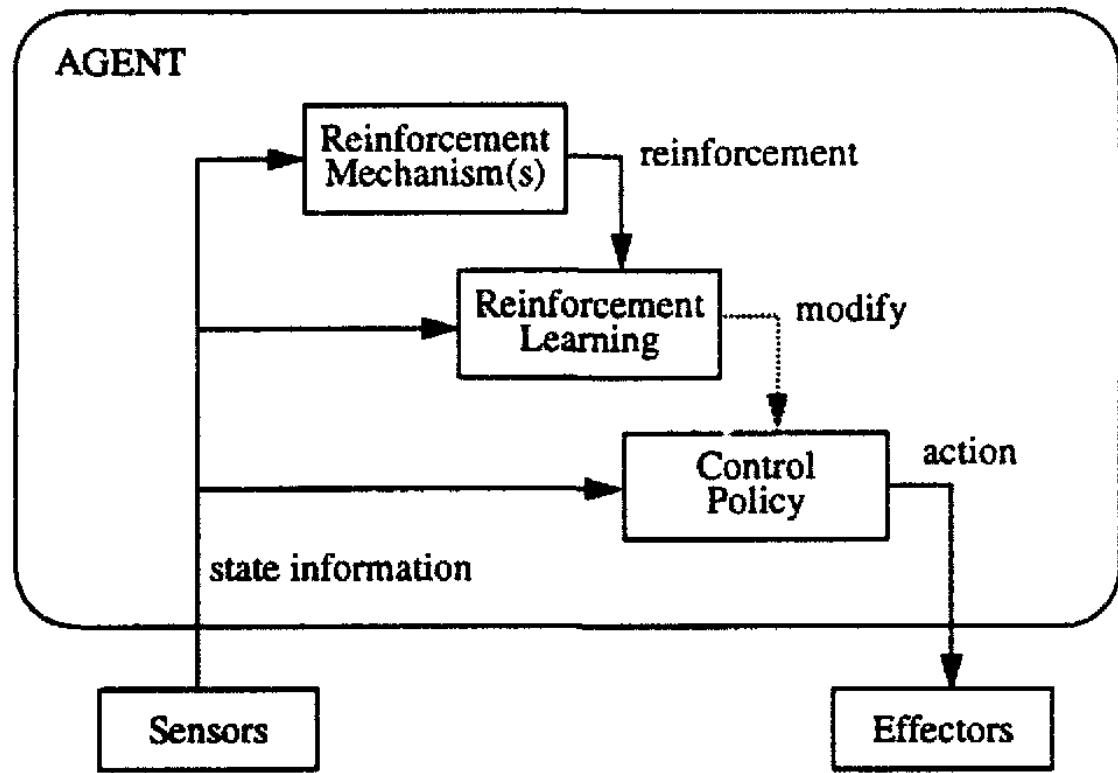


Figure- 3.3.1 Data-Flow-Diagram

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

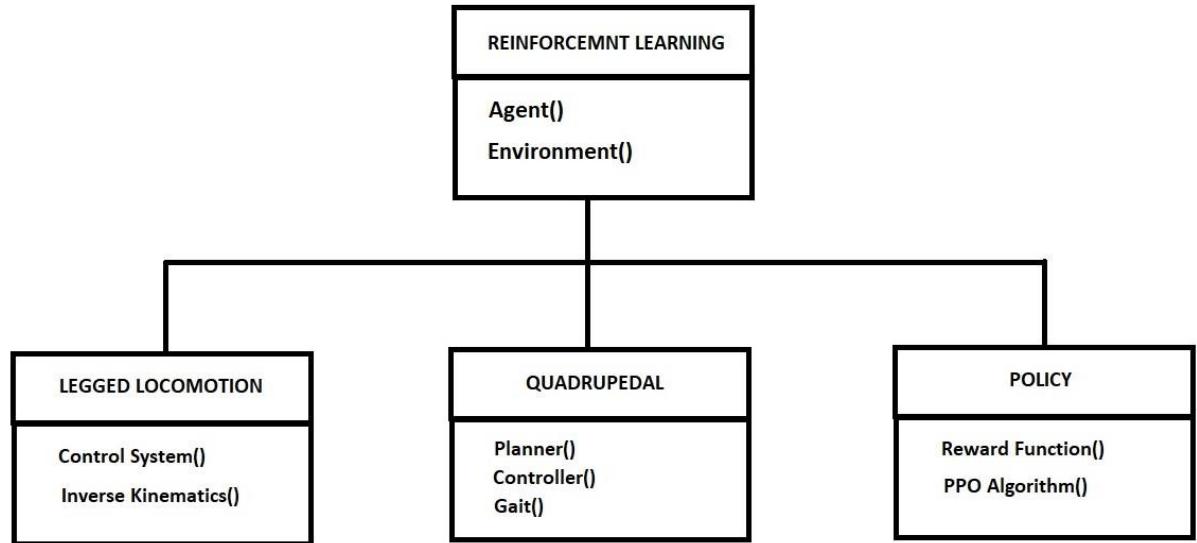


Figure- 3.3.3 UML Class Diagram

3.3.3. Control Sequence Diagram

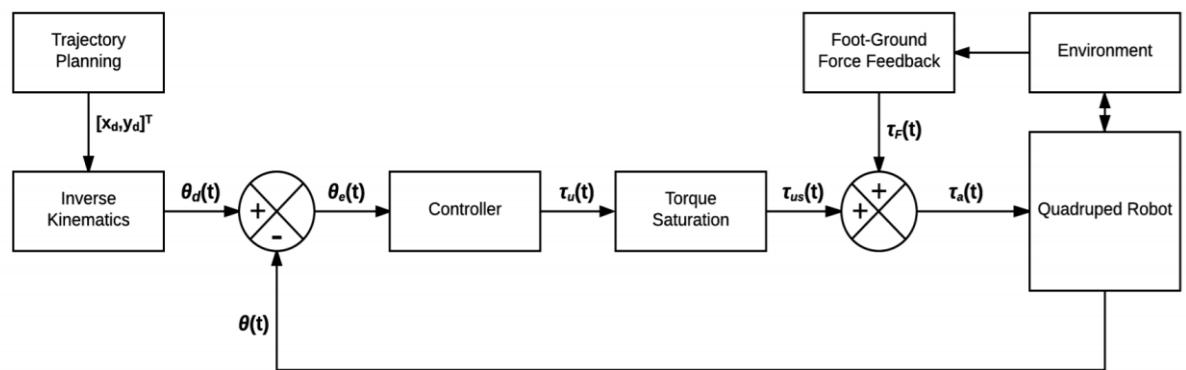


Figure- 3.3.4 Sequence -Diagram

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): $(S, A, R, p_0, p_T, \gamma)$. 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 $s_t \in S: a_t \sim \pi(a_t | s_t)$ at time stage $t \in \mathbb{N}$. According to the transfer probability model $p_T : s_{t+1} \sim p_T(s_{t+1} | s_t, a_t)$, 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 $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ because of the factor of discount $\gamma \in [0, 1]$ by enhancing the policies effectiveness to π^* . 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: $V(s) = E[R_t | s_t]$. The value function is used in such algorithm for the ds (dimensional state) space and the da (dimensional action) space $V(s) \in \mathbb{R}$ and policy's three conditions are: location $\mu(s) \in \mathbb{R}^x_y$, scale $\sigma(s) \in \mathbb{R}^x_y +$ and degrees of freedom $\nu \in \mathbb{R}_+$, using function approximations, they must be approximated.

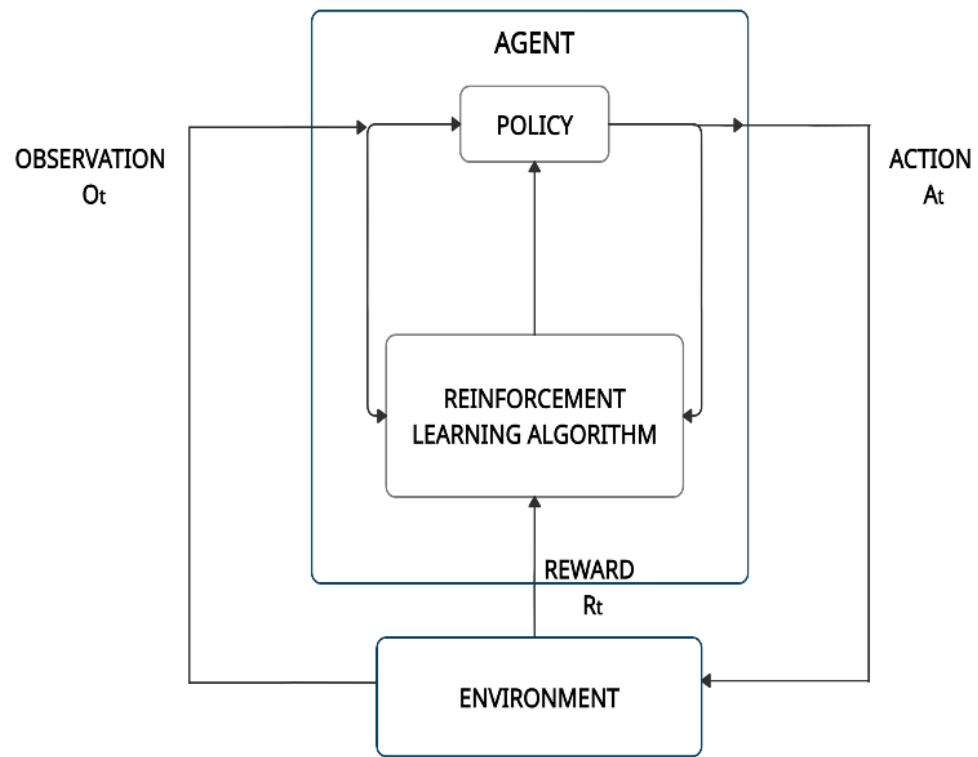


Fig.4.3.1. Reinforcement learning framework: an agent communicates with a climate by playing out an activity inspected from a strategy over a present status; the climate at that point returns the following state dependent on elements and the compensation to be augmented.

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:

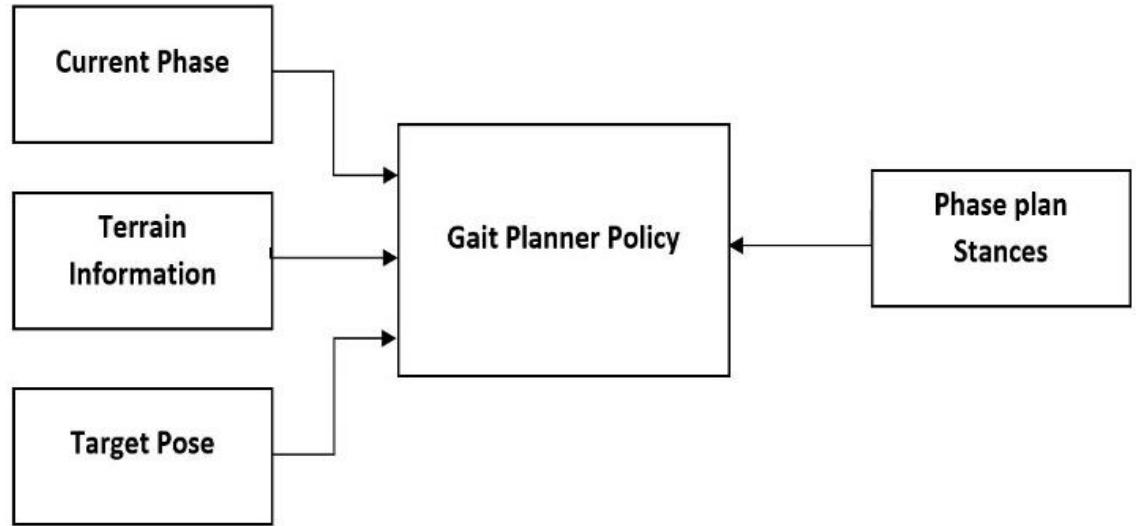


Figure. 4.5.1 Planning of Gait: represents as a localized terrain-aware planner that generates a finite succession of support periods, i.e., a phase plan, using both exteroceptive and proprioceptive measures.

The Planning of Gait generates the aforementioned step plan by sequentially querying the planning policy π_{θ_m} . When proposing phase transitions, we formulate the Markov Decision Process in order to train π_{θ_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 π_{θ_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:

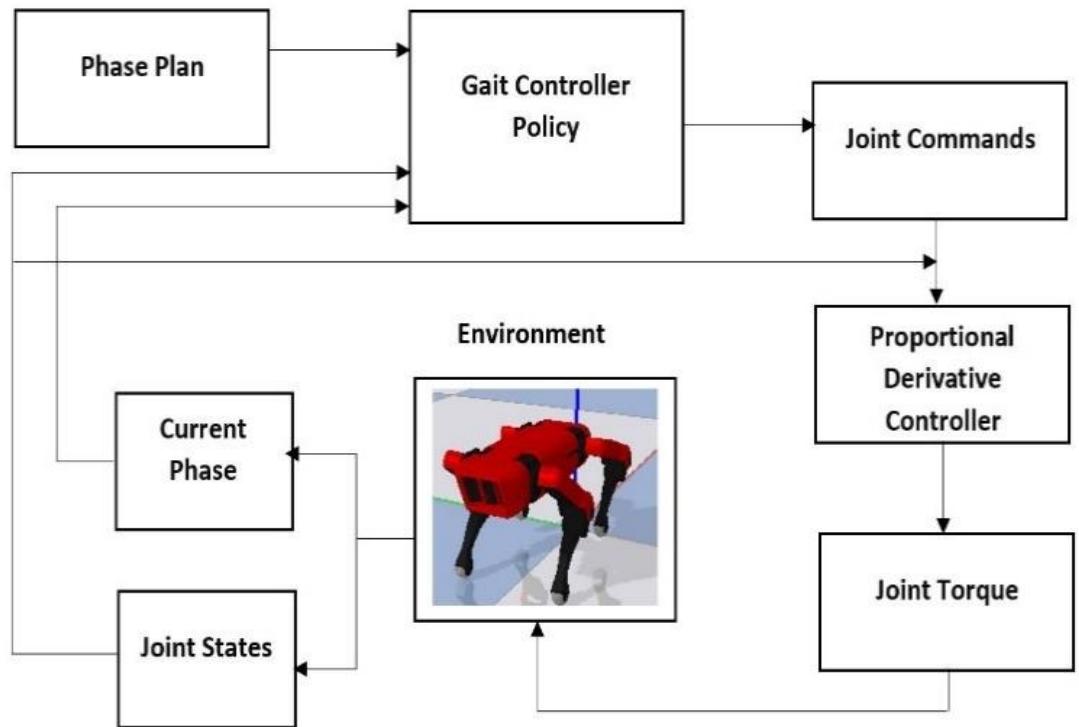


Fig. 3. Controlling of Gait: represents as the hybrid movement of planning system and controller that output's joint location references using only proprioceptive sensing and the aforementioned step plan. Finally, a combined PD controller (with null goal joint velocity) computes joint torques using these 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 $\pi(o)$ 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 $\pi(o)$ to nothing on the off chance that we decide to utilize a client indicated strategy. Set $a^*(t) = 0$ and give the criticism segment $\pi(o)$ 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:

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

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:

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

Fit Value Function is given by:

$$\phi_{k+1} = \arg \max_{\phi} \frac{1}{|D_k|T} \sum_{t \in D_k} \sum_{t=0}^T (V_{\phi}(s_t) - R_t)^2$$

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

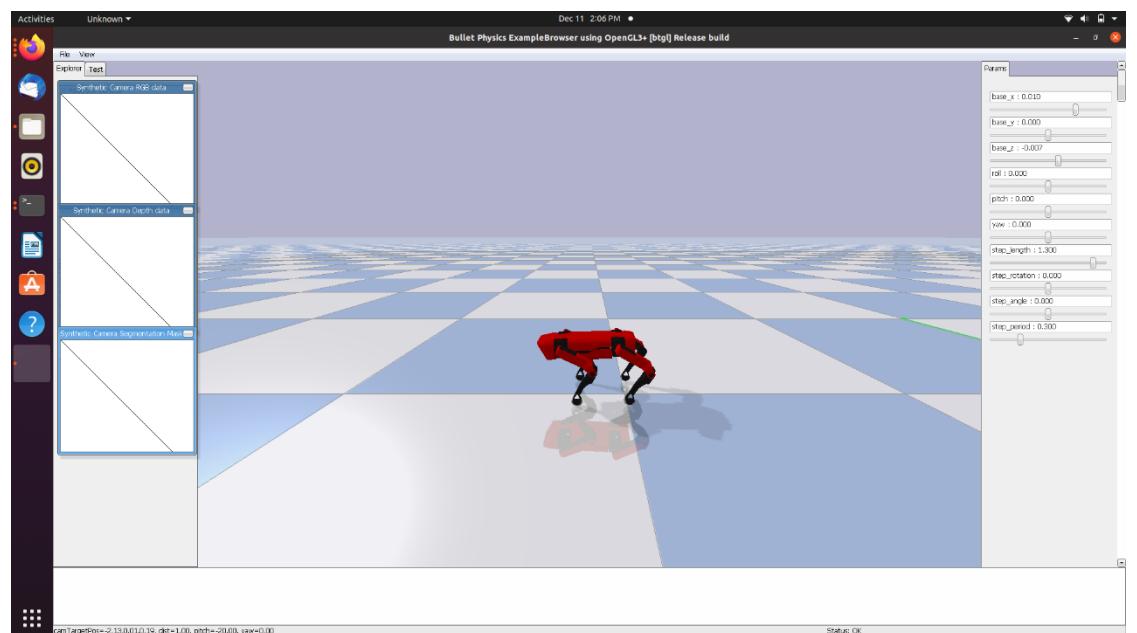
5.1. INTRODUCTION

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



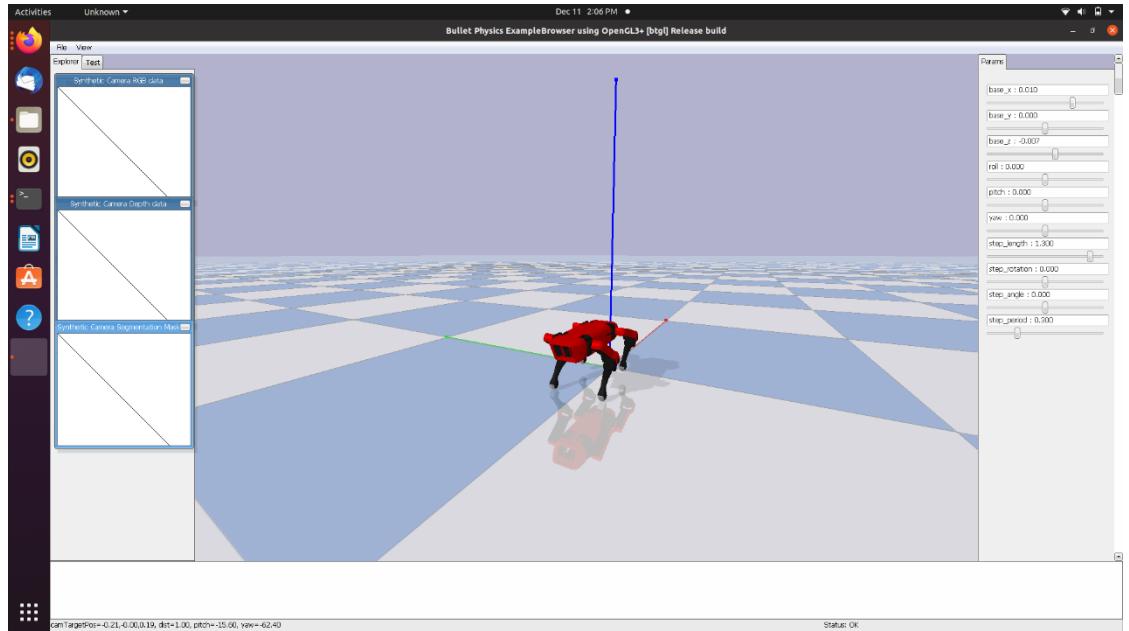
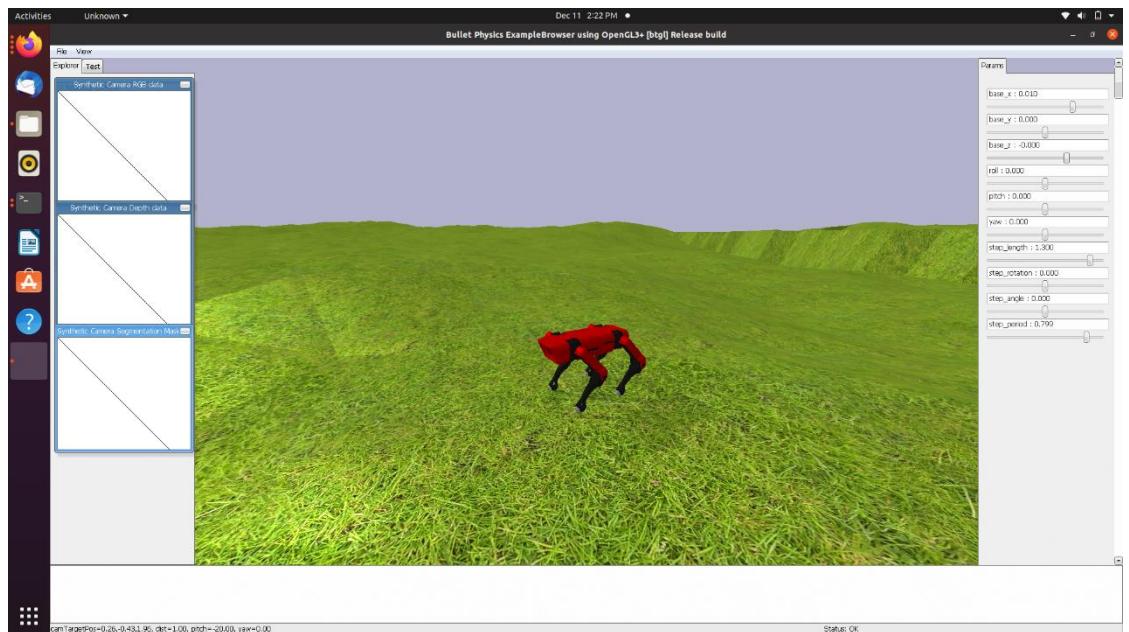


Fig. 5.1.1.1. Flat Plane Terrain: Basic approach towards simulation

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



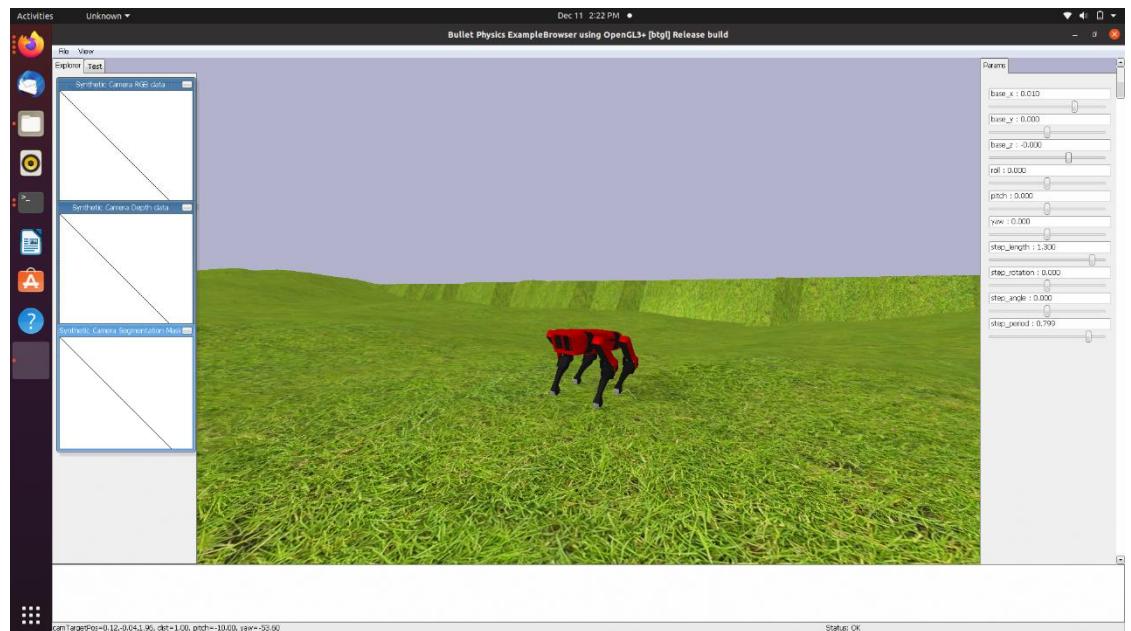
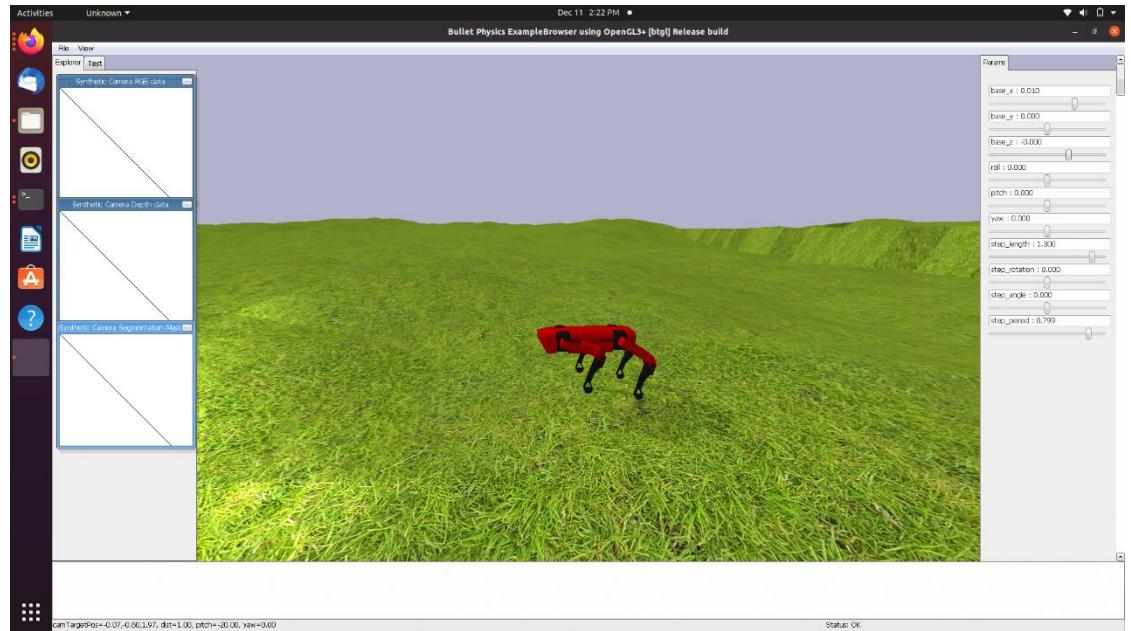
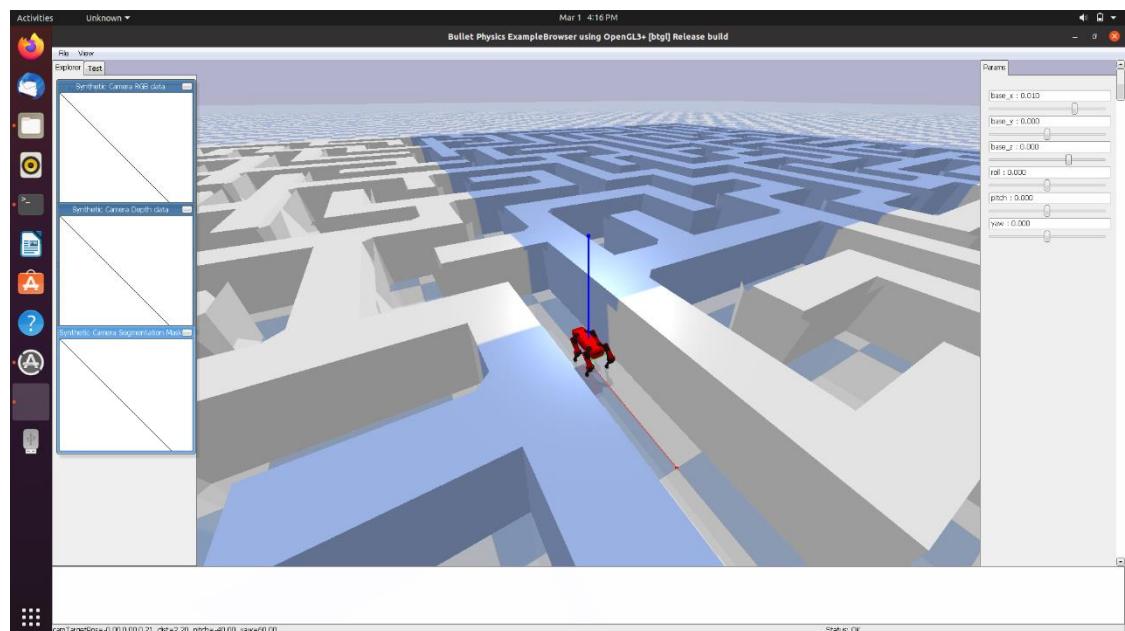
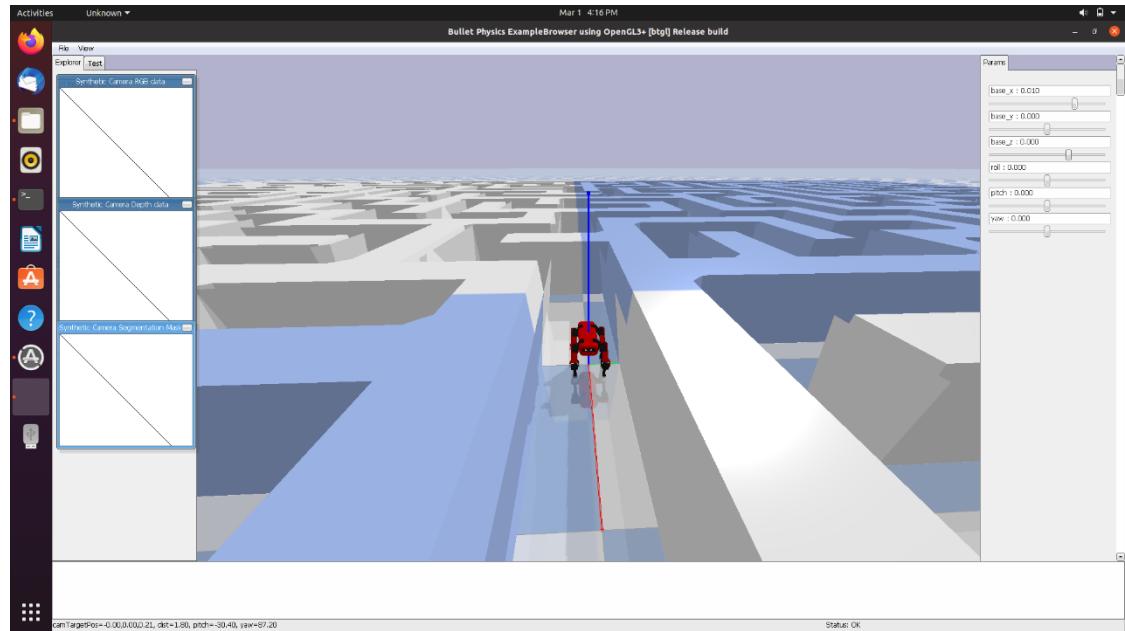


Fig. 5.1.1.b. Hills non-flat terrain: Outdoor non-flat grass surface

3) Maze flat terrain: Obstacles based approach for movement



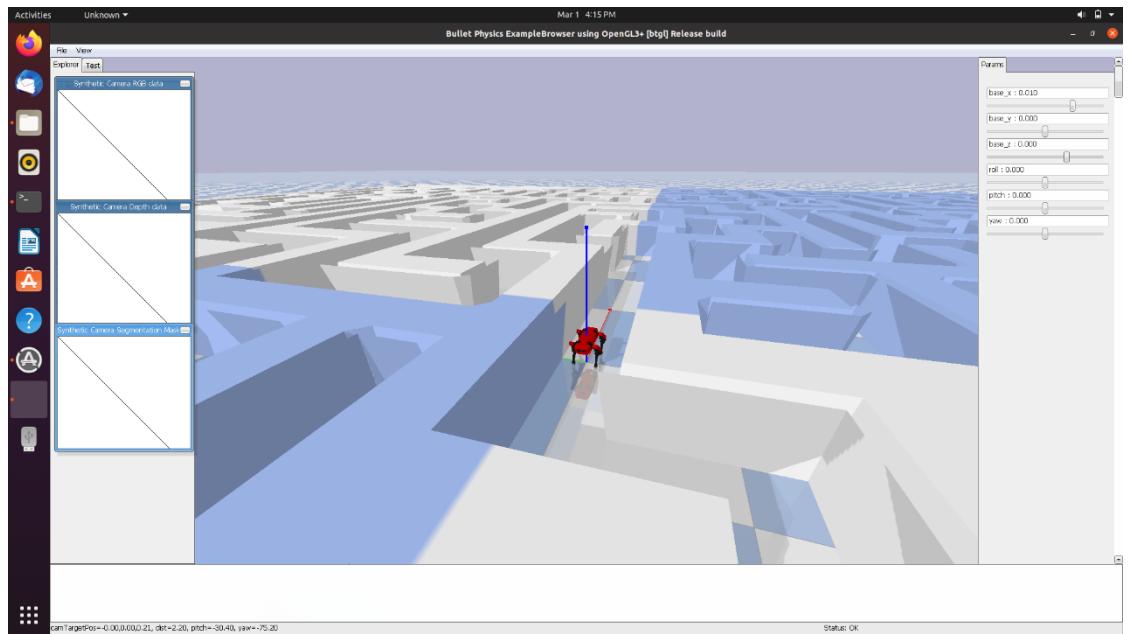
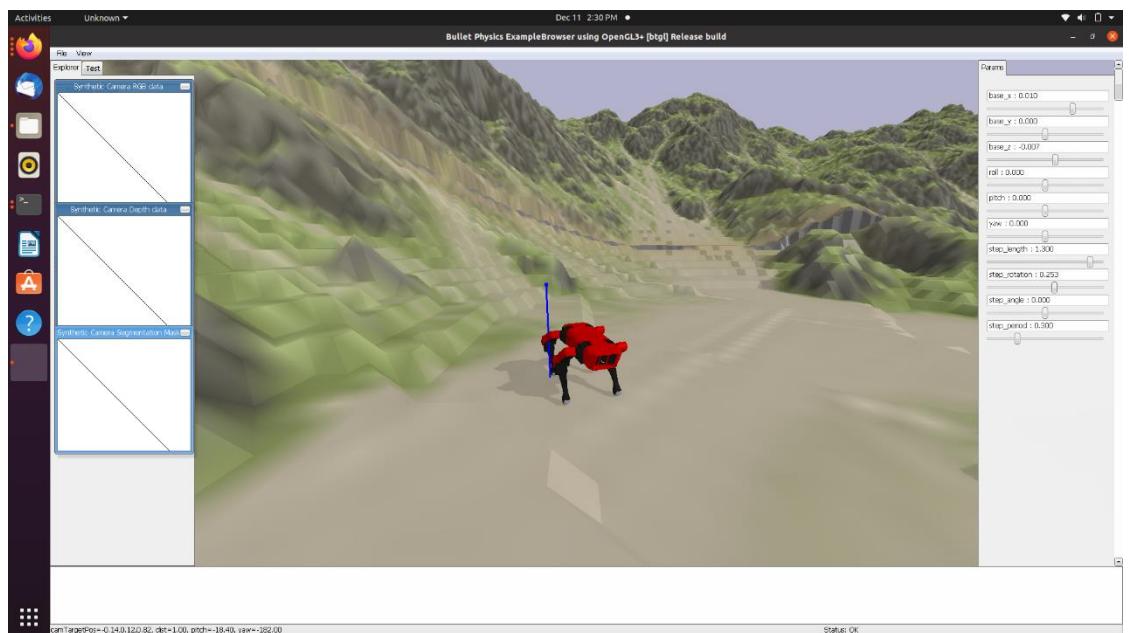


Fig. 5.1.1.c. Maze flat terrain: Obstacles based approach for movement

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



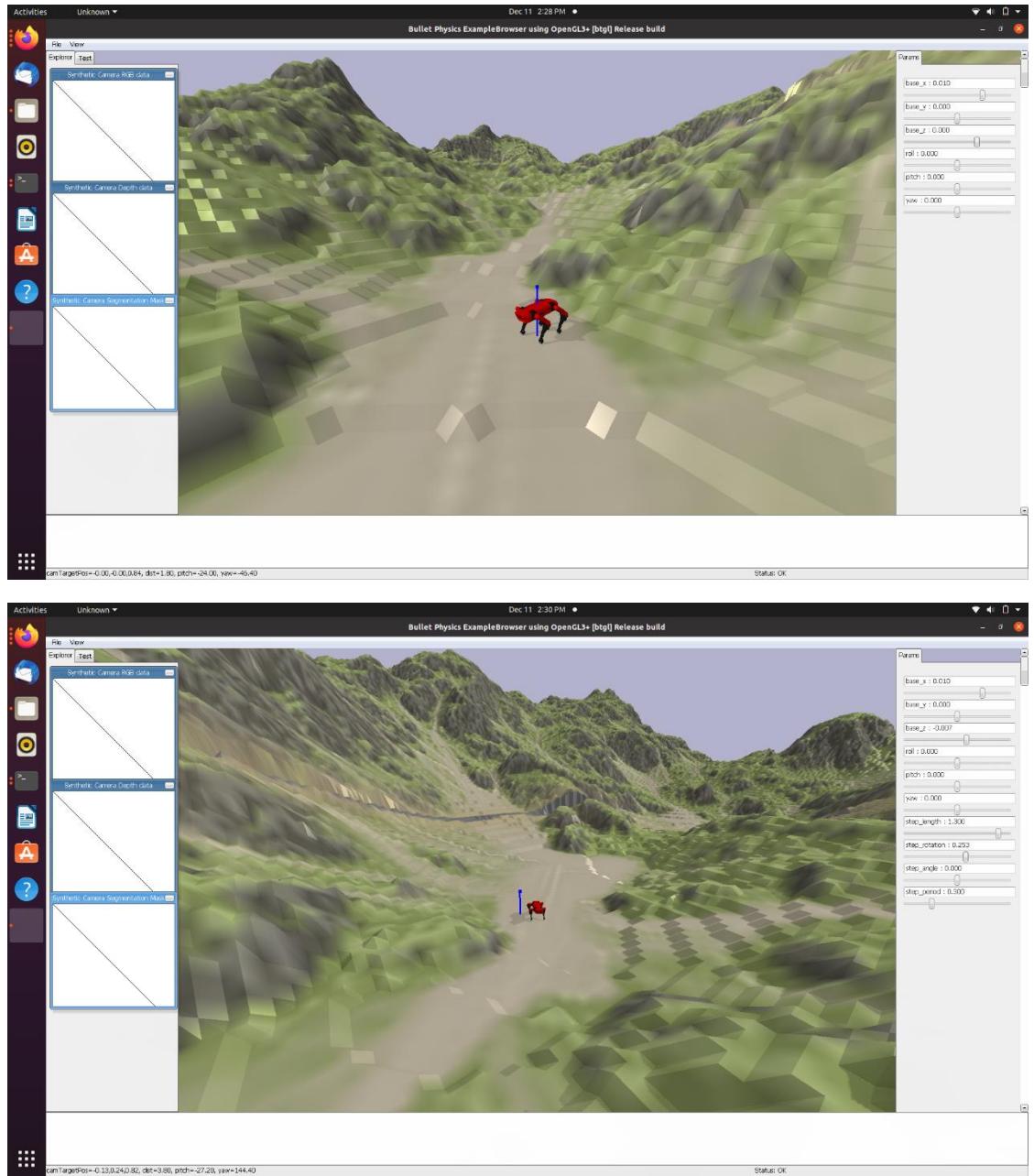


Fig. 5.1.1.d. Mountain non-flat terrain: Outdoor extreme complex non-flat surface

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

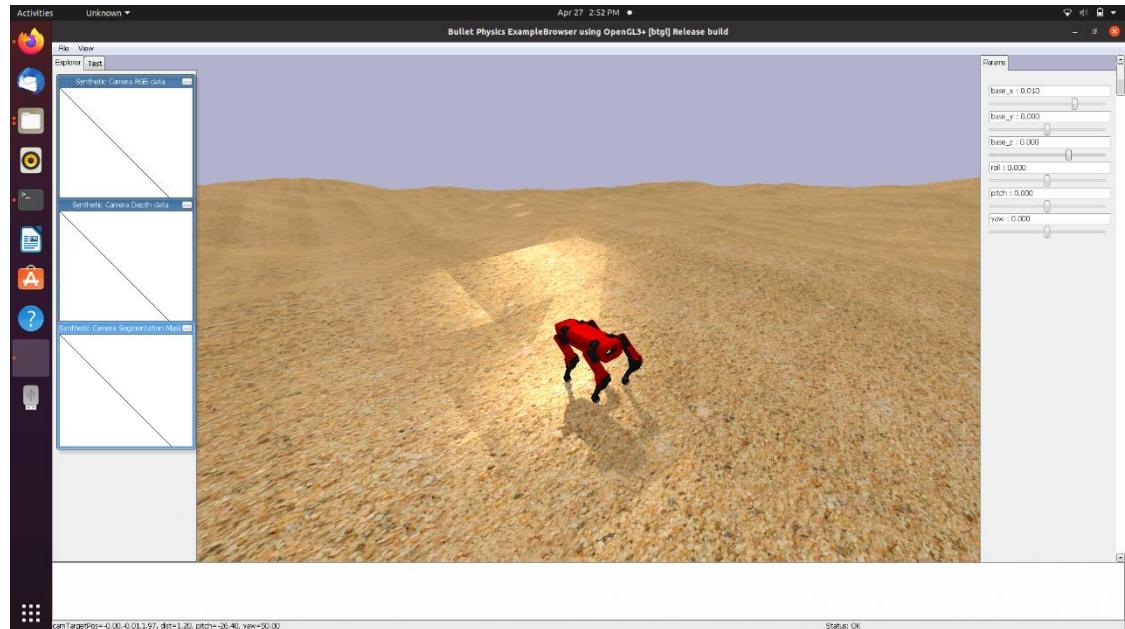
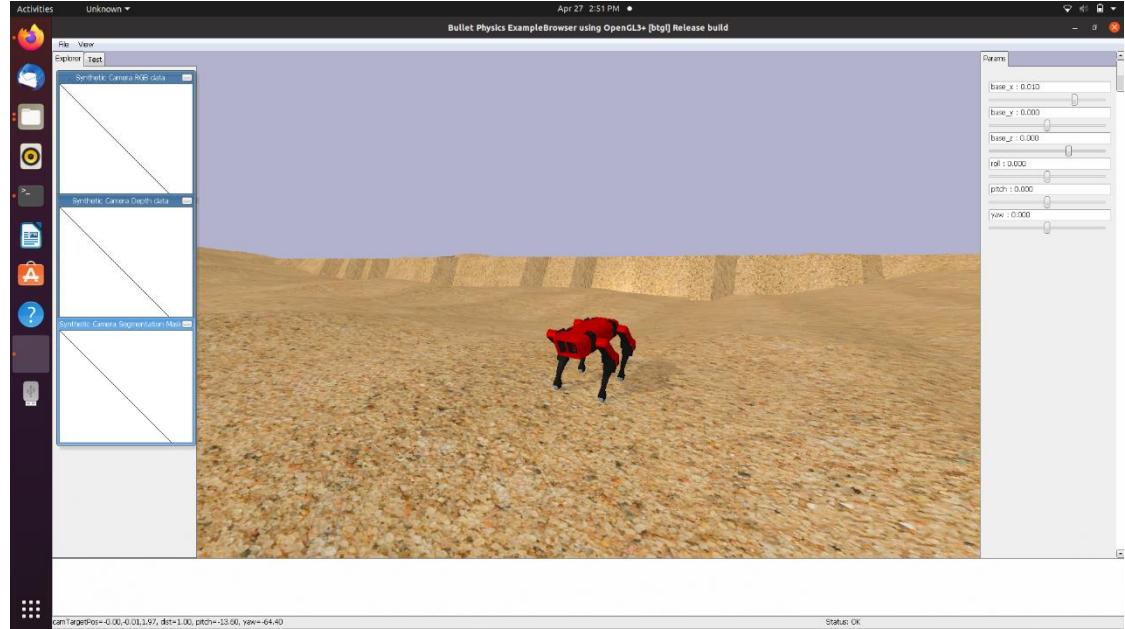


Fig. 5.1.1.e. Sand non-flat terrain: Sand particles-based approach for locomotion

6) Space flat terrain: complex flat surface space scenario

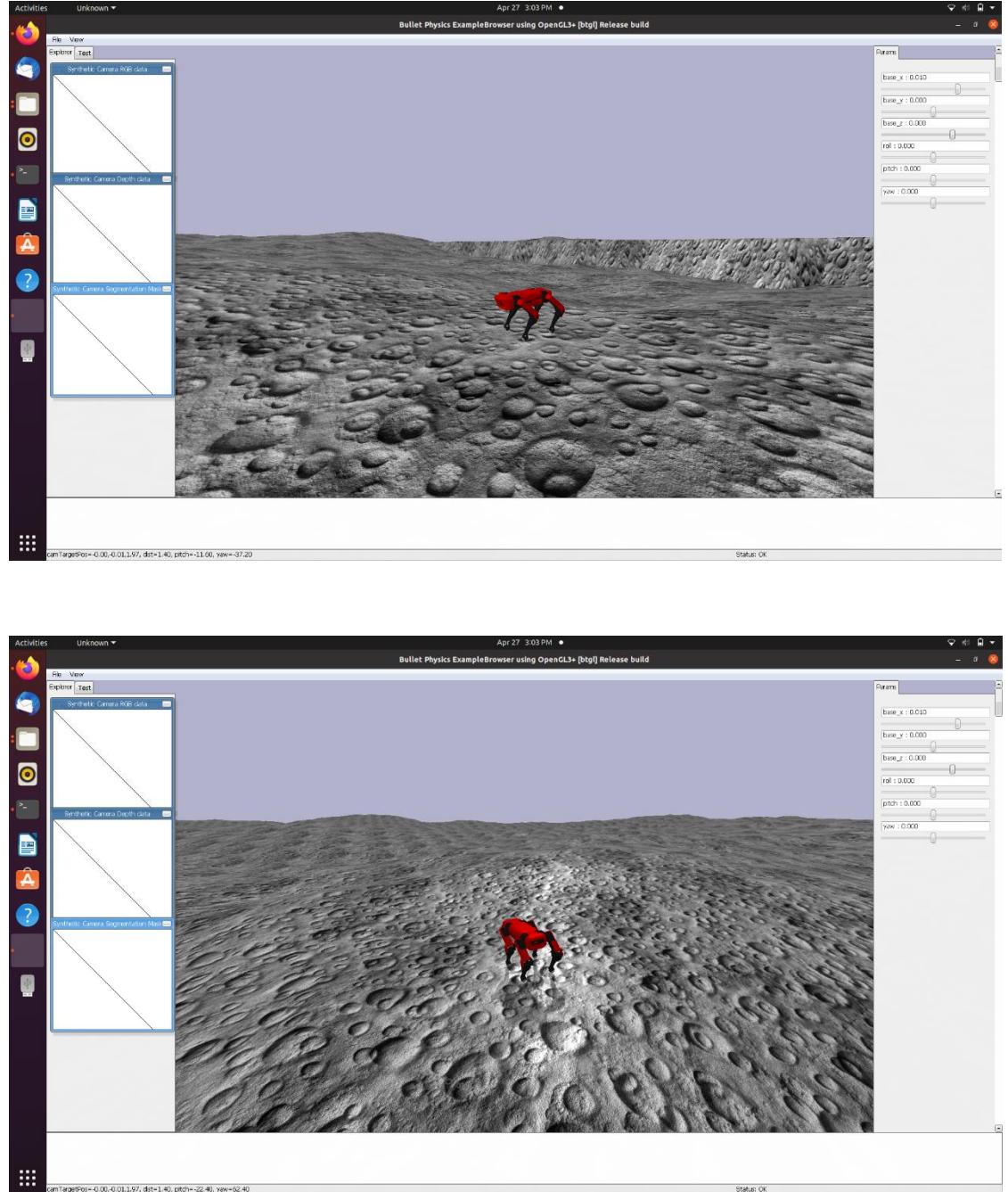


Fig. 5.1.1.f. Space flat terrain: Space hard flat complex surface based terrain

5.2. Graphs & Maps

5.2.1. Graph of cumulative reward function

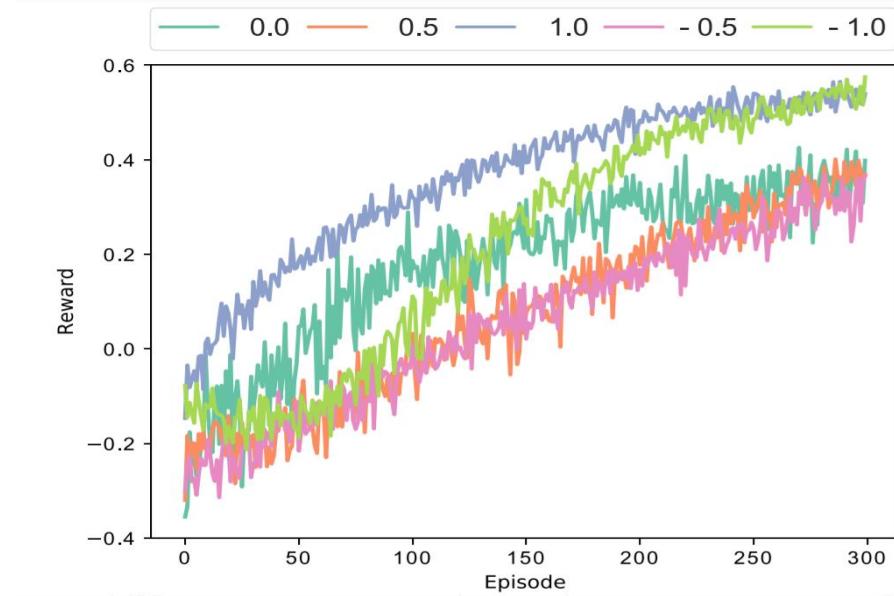


Fig. 5.1.2. Reward: In order to maximize reward, the agent must walk with respect to the set of episodes.

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 & torque

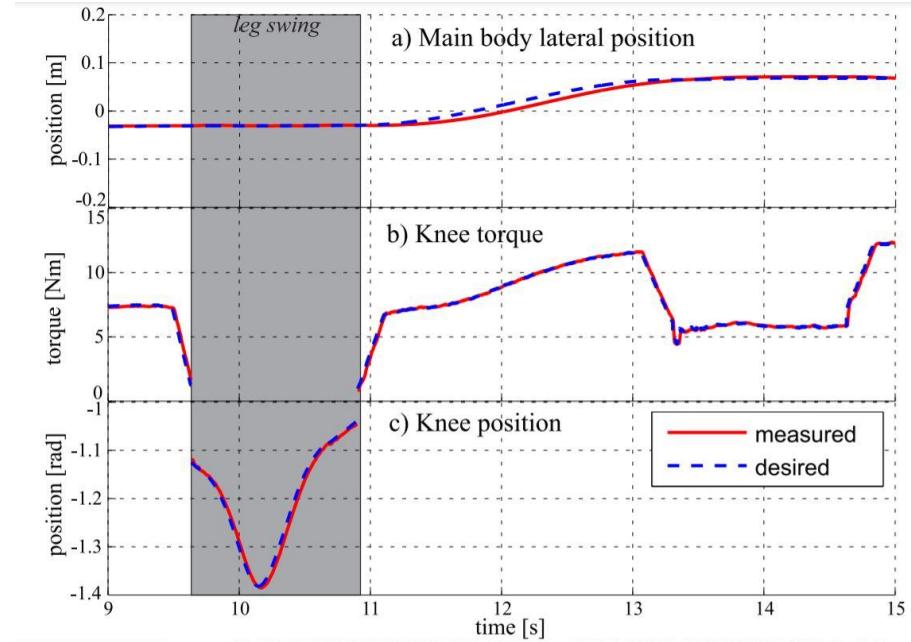


Fig. 5.1.3 Position and torque measured with respect to time

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] T. M. Camurri, M. Fallon, S. Bazeille, A. Radulescu, V. Barasuol, D. G. Caldwell, C. Semini, Probabilistic contact estimation and impact detection for state 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:

The screenshot shows a Jupyter Notebook interface with a single code cell containing Python code for a PPO algorithm. The code imports TensorFlow and its probability module, defines a `_NetworkOutput` tuple, and implements a `PPOAlgorithm` class. The class takes a `batch_env`, `step`, `is_training`, `should_log`, and `config` as arguments. It initializes `_batch_env`, `_step`, `_is_training`, `_should_log`, and `_config`. It also initializes `_observe_filter` and `_reward_filter` as `StreamingNormalizer` objects. The code then defines a template for memory storage, which includes `_batch_env.observe[0]`, `_batch_env.action[0]`, and `_batch_env.reward[0]`. It uses `tf.Variable` to store `_memory_index` and `_available_gpus`. Finally, it creates network variables for later calls to reuse, and defines a `_network` function that takes `tf.zeros_like` of `_batch_env.observe[0]`, `None`, and `len(self._batch_env)` as inputs. The code then uses `tf.compat.v1.variable_scope` to define `_episodes` and `_memory` using `EpisodeMemory` with `template`, `len(batch_env)`, and `config.max_length`.

```
16 #import collections
17 #import tensorflow as tf
18 #import tensorflow_probability as tfp
19 #
20 #from . import memory
21 #from . import normalize
22 #from . import utility
23 #
24 #_NetworkOutput = collections.namedtuple('NetworkOutput', 'policy, mean, logstd, value, state')
25 #
26 class PPOAlgorithm(object):
27     """A vectorized implementation of the PPO algorithm by John Schulman."""
28
29     def __init__(self, batch_env, step, is_training, should_log, config):
30         """Create an instance of the PPO algorithm.
31
32         Args:
33             batch_env: An episode batch environment.
34             step: Integer tensor holding the current training step.
35             is_training: Boolean tensor for whether the algorithm should train.
36             should_log: Boolean tensor for whether summaries should be returned.
37             config: Object containing the agent configuration as attributes.
38
39         """
40         self._batch_env = batch_env
41         self._step = step
42         self._is_training = is_training
43         self._should_log = should_log
44         self._config = config
45         self._observe_filter = normalize.StreamingNormalize(self._batch_env.observe[0],
46                                                            center=True,
47                                                            scale=True,
48                                                            clip=0,
49                                                            name='normalize_observe')
50         self._reward_filter = normalize.StreamingNormalize(self._batch_env.reward[0],
51                                                            center=False,
52                                                            scale=True,
53                                                            clip=0,
54                                                            name='normalize_reward')
55
56     # Memory stores tuple of observe, action, mean, logstd, value, state
57     template = (self._batch_env.observe[0], self._batch_env.action[0], self._batch_env.action[0],
58                 self._batch_env.reward[0], self._batch_env.value[0], self._batch_env.state[0])
59
60     self._memory = memory.EpisodeMemory(template, config.update_every, config.max_length, 'memory')
61     self._memory_index = tf.Variable(0, False)
62     self._available_gpus = utility.available_gpus()
63     use_gpu = tf.gpu_device_count() > 0
64     with tf.device('/gpu:0' if use_gpu else '/cpu:0'):
65
66         # Create network variables for later calls to reuse.
67         self._network = tf.zeros_like(self._batch_env.observe[0], None,
68                                      len(self._batch_env))
69
70         reuse_variables = tf.AUTO_REUSE
71         cell = self._config.network(self._batch_env.action.shape[1].value)
72         with tf.compat.v1.variable_scope('ppo_temporary'):
73             self._episodes = memory.EpisodeMemory(template, len(batch_env), config.max_length,
```

```
Activities Text Editor ▾ Open ▾ Save ▾ Apr 27 6:01 PM • algorithm.py /anaconda3/envs/rex/lib/python3.6/site-packages/rex_gym/agents/bpo
```

```
89
90     def begin_episode(self, agent_indices):
91         """Reset the recurrent states and stored episode.
92
93         Args:
94             agent_indices: ID tensor of batch indices for agents starting an episode.
95
96         Returns:
97             Summary tensor.
98
99         with tf.name_scope('begin_episode'):
100             reset_state = utility.reinit_nested_vars(self._last_state, agent_indices)
101             reset_buffer = self._episodes.clear(agent_indices)
102             with tf.control_dependencies([reset_state, reset_buffer]):
103                 return tf.constant('')
104
105     def perform(self, observe):
106         """Compute batch of actions and a summary for a batch of observation.
107
108         Args:
109             observe: Tensor of a batch of observations for all agents.
110
111         Returns:
112             Tuple of action batch tensor and summary tensor.
113
114         with tf.name_scope('perform'):
115             observe = self._observe_transform(observe)
116             mean, std = network.sample(observ[:, -1], self._last_state)
117             action = tf.cond(self._is_training, network.policy.sample, lambda: network.mean)
118             logprob = network.policy.log_prob(action[:, :], 0)
119             # pytlint: disable=long-lambda
120             summary = tf.summary.merge([
121                 self._should_log(lambda: tf.compat.v1.summary.merge([
122                     tf.compat.v1.summary.histogram('mean', network.mean[:, 0]),
123                     tf.compat.v1.summary.histogram('std', tf.exp(network.logstd[:, 0])),
124                     tf.compat.v1.summary.histogram('action', action[:, 0]),
125                     tf.compat.v1.summary.histogram('logprob', logprob)
126                 ]), str),
127                 # Remember current policy to append to memory in the experience callback.
128                 tf.cond(self._is_training, lambda: utility.assign_nested_vars(self._last_state, network.state),
129                         self._last_action.assign(action[:, 0])),
130                 self._last_mean.assign(network.mean[:, 0]),
131                 self._last_logstd.assign(network.logstd[:, 0])
132             ])
133
134             return tf.debugging.check_numerics(action[:, 0], 'action'), tf.identity(summary)
135
136     def experience(self, observe, action, reward, unused_done, unused_nextobs):
137         """Process the transition tuple of the current step.
138
139         When training, add the current transition tuple to the memory and update
140         the streaming statistics for observations and rewards. A summary string is
141         returned if requested at this step.
142
143         Args:
```

```

Activities Text Editor *
Open algorith.py
Apr 27 6:01 PM
~/anaconda3/envs/ex/lib/python3.7/site-packages/rex_gym/agents/ppo
Save
X
156     def _define_experience(self, observe, action, reward):
157         """Define experience class. This is used during training."""
158         update_filters = tf.compat.v1.summary.merge([
159             self._observe_filter.update(observe),
160             self._reward_filter.update(reward)])
161         with tf.control_dependencies([update_filters]):
162             if self._config.train_on_agent_action:
163                 # NOTE: Didn't seem to change much.
164                 # action, self._last_mean, self._last_logstd, reward
165                 # append = self._episodes.append(batch, tf.range(len(self._batch_env)))
166                 # with tf.control_dependencies([append]):
167                 #     self._observe_filter.transform(observe)
168                 #     norm_reward = tf.reduce_mean(tf._reward_filter.transform(reward))
169                 #     # pylint: disable=long-lambda
170                 summary = tf.cond(
171                     self._config.log, lambda: tf.compat.v1.summary.merge([
172                         update_filters,
173                         self._observe_filter.summary(),
174                         self._reward_filter.summary()],
175                         tf.compat.v1.summary.scalar('agent_mean_size', self._memory_index),
176                         tf.compat.v1.summary.histogram('normalized_observ', norm_observ),
177                         tf.compat.v1.summary.histogram('action', self._last_action),
178                         tf.compat.v1.summary.scalar('normalized_reward', norm_reward)
179                     ), str)
180             return summary
181     def end_episode(self, agent_indices):
182         """Add episodes to the memory and perform update steps if memory is full.
183         """
184         During training, add the collected episodes of the batch indices that
185         finished their episode to the memory. If the memory is full, train on it,
186         and then clear the memory. A summary string is returned if requested at
187         this step.
188         Args:
189             agent_indices: ID tensor of batch indices for agents starting an episode.
190         Returns:
191             Summary tensor.
192         """
193         with tf.name_scope('end_episode'):
194             return tf.cond(self._is_training, lambda: self._define_end_episode(agent_indices), str)
195     def define_end_episode(self, agent_indices):
196         """Implement the branch of end_episode() entered during training."""
197         assert len(agent_indices) == data[agent_indices]
198         space_left = self._config.update_every - self._memory_index
199         use_episodes = tf.range(tf.minimum(tf.shape(agent_indices)[0], space_left))
200         episodes = [tf.gather(elem, use_episodes) for elem in episodes]
201         append = self._memory.replace(episodes, tf.gather(length, use_episodes),
202                                     use_episodes + self._memory_index)
203         with tf.control_dependencies([append]):
204             inc_index = self._memory_index.assign_add(tf.shape(use_episodes)[0])
205
Python Tab Width: 8 ▾ Ln 1, Col 1 ▾ INS

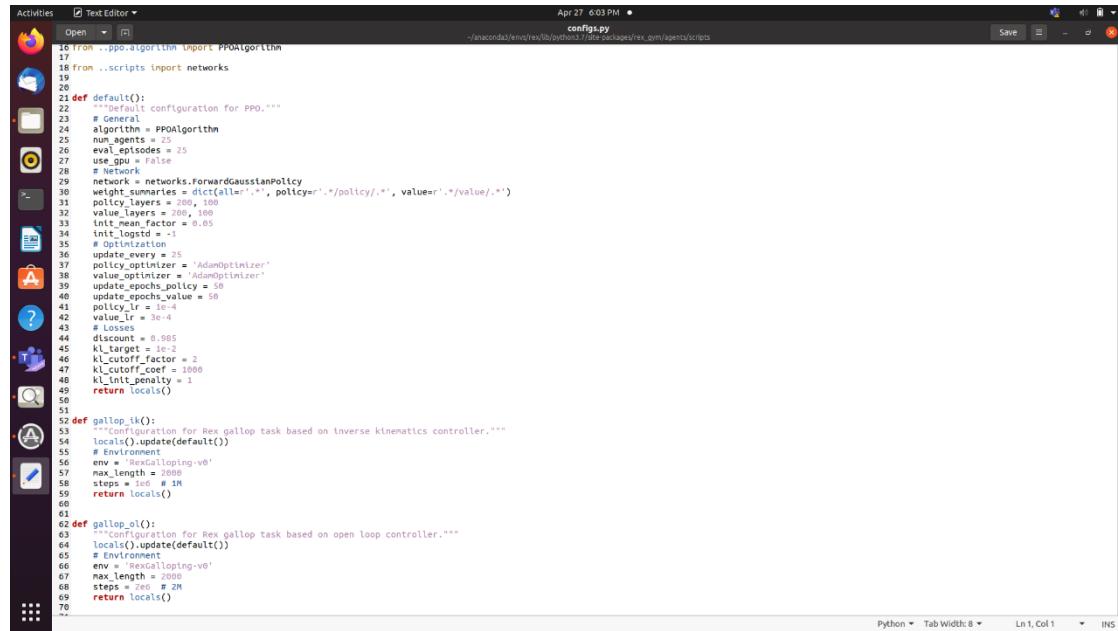
```

```

Activities Text Editor *
Open algorith.py
Apr 27 6:02 PM
~/anaconda3/envs/ex/lib/python3.7/site-packages/rex_gym/agents/ppo
Save
X
215     def training(self):
216         """Perform multiple training iterations of both policy and value baseline.
217         """
218         Training on the episodes collected in the memory. Reset the memory
219         afterwards. Always returns a summary string.
220
221         Returns:
222             Summary tensor.
223
224         with tf.name_scope('training'):
225             assert_full = tf.compat.v1.assert_equal(self._memory_index, self._config.update_every)
226             with tf.control_dependencies([assert_full]):
227                 data = self._memory.data
228                 (observe, action, old_mean, old_logstd, reward, length =
229                  (self._observe_filter.update(data[agent_indices]),
230                   self._action_mean_update(data[agent_indices]),
231                   self._logstd_update(data[agent_indices]),
232                   self._reward_filter.update(data[agent_indices]),
233                   tf.compat.v1.assert_greater(length, 0)))
234                 with tf.control_dependencies([tf.compat.v1.assert_greater(length, 0)]):
235                     length = tf.identity(length)
236                     observe = self._observe_filter.transform(observe)
237                     reward = self._reward.filter.transform(reward)
238                     policy_summary = self._policy.update_policy(observe, action, old_mean, old_logstd, reward, length)
239                     with tf.control_dependencies([policy_summary]):
240                         value_summary = self._value.update_value(observe, old_mean, old_logstd, reward, length)
241                         with tf.control_dependencies([value_summary]):
242                             penalty_summary = self._adjust_penalty(observe, old_mean, old_logstd, length)
243                             with tf.control_dependencies([penalty_summary]):
244                                 clear_memory = tf.group(self._memory.clear(), self._memory_index.assign(0))
245                                 with tf.control_dependencies([clear_memory]):
246                                     weight_summary = utility.variable_summaries(tf.compat.v1.trainable_variables(),
247                                         self._config.weight_summaries)
248                                     return tf.compat.v1.summary.merge([policy_summary, value_summary, penalty_summary, weight_summary])
249
250     def _update_value(self, observe, reward, length):
251         """Perform multiple update steps of the value baseline.
252
253         We need to decide for the summary of one iteration, and thus choose the one
254         after half of the iterations.
255
256         Args:
257             observe: Sequences of observations.
258             reward: Sequences of reward.
259             length: Batch of sequence lengths.
260
261         Returns:
262             Summary tensor.
263
264         with tf.name_scope('update_value'):
265             loss, summary = tf.scan(lambda _1, _2: self._update_value_step(observe, reward, length),
266                                     tf.range(self._config.update_epochs_value), [0, '']),
267             print.loss = tf.Print(0, [tf.reduce_mean(loss)], 'value loss: ')
268             with tf.control_dependencies([loss, print.loss]):
269                 return summary[self._config.update_epochs_value // 2]
270
271     def _update_value_step(self, observe, reward, length):
272         """Compute the current value loss and perform a gradient update step.
273
Python Tab Width: 8 ▾ Ln 1, Col 1 ▾ INS

```

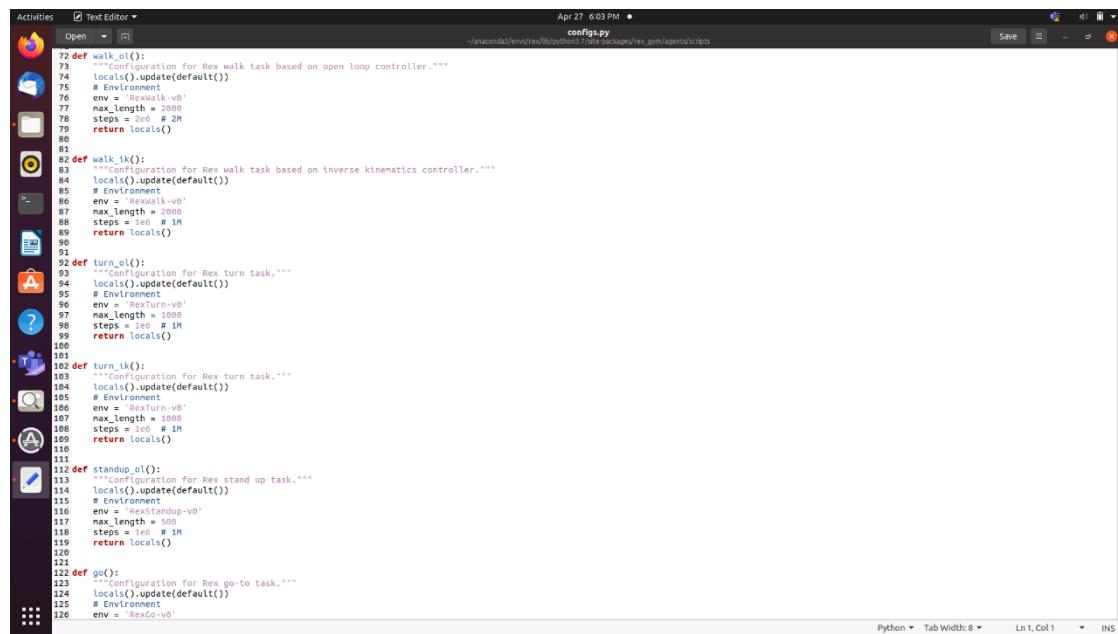
Configuration:



```

Activities Text Editor • config.py
Apr 27 6:03 PM •
16 from .ppo.algorithms import PPOAlgorithm
17 from ..scripts import networks
18
19
20
21 def default():
22     """Default configuration for PPO."""
23     # General
24     algorithm = PPOAlgorithm
25     num_episodes = 25
26     eval_episodes = 25
27     use_gpu = False
28
29     # Network
30     network = networks.ForwardGaussianPolicy
31     weight_summaries = dict(all=r'~.*policy/.*', value=r'~.*value/.*')
32     policy_layers = [200, 100]
33     value_layers = [200, 100]
34     value_lr = 0.0001
35     init_logstd = -1
36     update_epochs = 25
37
38     policy_optimizer = 'AdamOptimizer'
39     update_epochs_policy = 50
40     update_epochs_value = 50
41     policy_lr = 1e-04
42     value_lr = 3e-04
43
44     # Losses
45     discount = 0.985
46     kl_tolerance = 2
47     kl_cutoff_factor = 2
48     kl_cutoff_coeff = 1000
49     kl_init_penalty = 1
50
51     return locals()
52
53 def gallop_ik():
54     """Configuration for Rex gallop task based on inverse kinematics controller."""
55     locals().update(default())
56
57     # Environment
58     env = 'RexGalloping-v0'
59     max_length = 2000
60     steps = 1e6 # 1M
61     return locals()
62
63 def gallop_ol():
64     """Configuration for Rex gallop task based on open loop controller."""
65     locals().update(default())
66
67     # Environment
68     env = 'RexGalloping-v0'
69     max_length = 2000
70     steps = 1e6 # 1M
71     return locals()
72
73
74 def walk_ol():
75     """Configuration for Rex walk task based on open loop controller."""
76     locals().update(default())
77
78     # Environment
79     env = 'RexWalk-v0'
80     max_length = 2000
81     steps = 2e6 # 2M
82     return locals()
83
84
85 def walk_ik():
86     """Configuration for Rex walk task based on inverse kinematics controller."""
87     locals().update(default())
88
89     # Environment
90     env = 'RexWalk-v0'
91     max_length = 2000
92     steps = 1e6 # 1M
93     return locals()
94
95
96 def turn_ol():
97     """Configuration for Rex turn task."""
98     locals().update(default())
99
100    # Environment
101    env = 'RexTurn-v0'
102    max_length = 2000
103    steps = 1e6 # 1M
104    return locals()
105
106
107 def turn_ik():
108     """Configuration for Rex turn task."""
109     locals().update(default())
110
111    # Environment
112    env = 'RexTurn-v0'
113    max_length = 2000
114    steps = 1e6 # 1M
115    return locals()
116
117
118 def standup_ol():
119     """Configuration for Rex stand up task."""
120     locals().update(default())
121
122    # Environment
123    env = 'RexStandup-v0'
124    max_length = 500
125    steps = 1e6 # 1M
126    return locals()
127
128
129 def get():
130     """Configuration for Rex go-to task."""
131     locals().update(default())
132
133    # Environment
134    env = 'RexGo-v0'
135
136
137

```



```

Activities Text Editor • config.py
Apr 27 6:03 PM •
17 from ..scripts import networks
18
19
20
21 def walk_ol():
22     """Configuration for Rex walk task based on open loop controller."""
23     locals().update(default())
24
25     # Environment
26     env = 'RexWalk-v0'
27     max_length = 2000
28     steps = 2e6 # 2M
29     return locals()
30
31
32 def walk_ik():
33     """Configuration for Rex walk task based on inverse kinematics controller."""
34     locals().update(default())
35
36     # Environment
37     env = 'RexWalk-v0'
38     max_length = 2000
39     steps = 1e6 # 1M
40     return locals()
41
42
43 def turn_ol():
44     """Configuration for Rex turn task."""
45     locals().update(default())
46
47     # Environment
48     env = 'RexTurn-v0'
49     max_length = 2000
50     steps = 1e6 # 1M
51     return locals()
52
53
54 def turn_ik():
55     """Configuration for Rex turn task."""
56     locals().update(default())
57
58    # Environment
59    env = 'RexTurn-v0'
60    max_length = 2000
61    steps = 1e6 # 1M
62    return locals()
63
64
65 def standup_ol():
66     """Configuration for Rex stand up task."""
67     locals().update(default())
68
69    # Environment
70    env = 'RexStandup-v0'
71    max_length = 500
72    steps = 1e6 # 1M
73    return locals()
74
75
76
77 def get():
78     """Configuration for Rex go-to task."""
79     locals().update(default())
80
81    # Environment
82    env = 'RexGo-v0'
83
84
85

```

Entry Point:



The screenshot shows a Jupyter Notebook interface with the following details:

- Activities**: Shows "Text Editor" as the active tab.
- File**: Options include Open, Save, and Exit.
- Code Content**: The notebook cell contains a Python script for a reinforcement learning environment named "entry_point.py". The code includes imports for click, rexml, and flag_mappers, and defines a command-line interface (CLI) using the click library. It handles arguments for environment name, flags, training mode, and other parameters like terrain type and number of agents.
- Environment**: The code is run in an Anaconda environment named "env", located at "/anacondas/envs/experiments/rate-packages/rex_gym/ci".
- Timestamp**: The notebook was last modified on April 27, 6:05 PM.
- Toolbar**: Standard Jupyter Notebook toolbar icons for back, forward, search, and help.

Galloping Environment:

The screenshot shows a Jupyter Notebook interface with the following details:

- Activities**: Shows icons for a browser, file manager, terminal, and other notebooks.
- Text Editor**: The active tab, showing Python code for a robot environment.
- Open**: A dropdown menu with options like "New Notebook", "Open", "Save", and "Close".
- File**: A standard file menu with options like "File", "Edit", "Cell", "Kernel", "Help", and "About".
- Code Content**:
 - Imports: collections, math, random, gym, spaces, np, reor_gym, model, mark_constants, GaitPlanner, rex_gym_env.
 - Variables: NUM_LEGS = 4, NUM_MOTORS = 3 * NUM_LEGS.
 - Class Definition:
 - RexPose**: A named tuple representing joint angles for four legs.
 - RexEnv**: A class derived from `rex_gym_env.RexGymEnv`. It includes a docstring describing the environment as simulating the locomotion of a quadruped robot (Rex) with 3 motors per leg, 12 angles per leg, and a reward function based on walking 1000 steps while minimizing energy expenditure.
 - Metadata**: A dictionary specifying render modes ("human", "rgb_array"), video frames per second (100), and a flag for loading UI (True).
 - Initialization**: The `__init__` method takes various parameters including debug, urdf, energy, control_time_step, action_repeat, control_latency, pd_latency, on_ramp, motor_kp, motor_kd, render, max_timestep, max_stuck_timestep, log, use_angle_in_observation, env_randomizer, and log_path.

```
Activities TextEditor Apr 27 6:07 PM gallop.env.py Save
Open
self._use_angle_in_observation = use_angle_in_observation
super().__init__(self._use_angle_in_observation)
self._curr_version = self._curr_version + 1
env_id = "gallop"
accurate_motor_model_enabled = True,
motor_overheat_protection = True,
hard_reset = False,
motor_kp = 1000.0,
motor_kd = 10.0,
remove_default_joint_damping = False,
control_latency = control_latency,
on_rack = on_rack,
render = render,
num_steps_to_log = num_steps_to_log,
env_randomizer = env_randomizer,
control_time_step = control_time_step,
action_repeat = action_repeat,
get_position = target_position,
signal_type = signal_type,
debug = debug,
terrain_id = terrain_id,
terrain_type = terrain_type,
mark = mark)

# (eventually) allow different feedback ranges/action spaces for different signals
action_max = {
    'lK': 0.4,
    'oL': 0.3
}
action_dim_map = {
    'lK': 2,
    'oL': 4
}
action_dim = action_dim_map[self._signal_type]
action_low = np.array([action_max[self._signal_type]] * action_dim)
action_high = -action_low
action_space = spaces.Box(action_low, action_high)
self._can_dist = 1.0
self._can yaw = 0.8
self._can pitch = -0.2
target_position = target_position
self._signal_type = signal_type
self._gait_planner = GaitPlanner("gallop")
self._kinematics = Kinematics()
self._is_standing = 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(ReactiveEnv, self).reset(initial_motor_angles=self.init_pose, reset_durations=0.5)
    self.goal_reached = False
```

```

Activities Text Editor * Apr 27 6:07PM • gallop.envy
Open -/anaconda/envs/rex/lib/python3.7/site-packages/ex_gym/envs/gym
Save X
142     def reset(self):
143         self._init_pose = rex_constants.INIT_POSES['stand']
144         if self._target_x == None:
145             self._init_pose = rex_constants.INIT_POSES['stand_ol']
146         super(RexReactiveEnv, self).reset(initial_motor_angles=self._init_pose, reset_duration=0.5)
147         self._goal_reached = False
148         self._stage_still = 0
149         self._is_terminating = False
150         if not self._target_position or self._random_pos_target:
151             self._target_position = random.uniform(-.3, .3)
152             self._random_pos_target = True
153         if self._is_render and self._signal_type == 'UK':
154             if self._load_ui:
155                 self._load_ui()
156             self._load_ui = False
157             if self._debug:
158                 print(f'Target Position {self._target_position}, Random assignment: {self._random_pos_target}')
159             return self._get_observation()
160
161     def setup_ui(self):
162         self.base_x_ui = self._pybullet_client.addUserDebugParameter("base_x",
163             self._ranges['base_x'][0], self._ranges['base_x'][1],
164             0.01)
165         self.base_y_ui = self._pybullet_client.addUserDebugParameter("base_y",
166             self._ranges['base_y'][0], self._ranges['base_y'][1],
167             self._ranges['base_y'][2])
168         self.base_z_ui = self._pybullet_client.addUserDebugParameter("base_z",
169             self._ranges['base_z'][0], self._ranges['base_z'][1],
170             -0.007)
171         self.roll_ui = self._pybullet_client.addUserDebugParameter("roll",
172             self._ranges['roll'][0], self._ranges['roll'][1],
173             self._ranges['roll'][2])
174         self.pitch_ui = self._pybullet_client.addUserDebugParameter("pitch",
175             self._ranges['pitch'][0], self._ranges['pitch'][1],
176             self._ranges['pitch'][2])
177         self.yaw_ui = self._pybullet_client.addUserDebugParameter("yaw",
178             self._ranges['yaw'][0], self._ranges['yaw'][1],
179             self._ranges['yaw'][2])
180
181         self.step_length_ui = self._pybullet_client.addUserDebugParameter("step_length",
182             self._ranges['step_length'][0], self._ranges['step_length'][1],
183             0.7, 1.3)
184         self.step_rotation_ui = self._pybullet_client.addUserDebugParameter("step_rotation",
185             self._ranges['step_rotation'][0], self._ranges['step_rotation'][1],
186             -1.5, 1.5)
187         self.step_angle_ui = self._pybullet_client.addUserDebugParameter("step_angle",
188             self._ranges['step_angle'][0], self._ranges['step_angle'][1],
189             -180, 180)
190         self.step_periud_ui = self._pybullet_client.addUserDebugParameter("step_periud",
191             0.1, 0.3)
192
193     def read_inputs(self, base_pos_coeff, gait_stage_coeff):
194         position = np.array([
195             self._pybullet_client.readUserDebugParameter(self.base_x_ui),
196             self._pybullet_client.readUserDebugParameter(self.base_y_ui),
197             self._pybullet_client.readUserDebugParameter(self.base_z_ui)])
198
199     @staticmethod
200     def check_target_position(self, t):
201         if self._target_x < abs(self.rex.getBasePosition()[0]):
202             # give 0.15 stop space
203             if self._target_x >= self._target_position:
204                 self._goal_reached = True
205                 if not self._is_terminating:
206                     self.end_time = t
207                     self._is_terminating = True
208
209     @staticmethod
210     def evaluate_stage_coefficient(current_t, end_t=0.0, width=0.001):
211         # sigmoid function
212         beta = p = width
213         if p - beta + end_t <= current_t <= p - (beta / 2) + end_t:
214             return 1 - ((current_t - p + beta / 2) ** 2) / (width ** 2)
215         elif p - (beta / 2) + end_t <= current_t <= p + end_t:
216             return 1 - ((current_t - p) ** 2) / (width ** 2)
217         else:
218             return 1
219
220     @staticmethod
221     def evaluate_brakes_stage_coeff(current_t, action, end_t=0.0, end_value=0.0):
222         # step function
223         p = action[0]
224         if end_t <= current_t <= p + end_t:
225             return 1 - (current_t - end_t)
226         else:
227             return end_value
228
229     @staticmethod
230     def evaluate_gait_stage_coeff(current_t, action, end_t=0.0):
231         # ramp function
232         p = 1. + action[1]
233         if end_t <= current_t <= p + end_t:
234             return current_t
235         else:
236             return 1.0
237
238     def signal(self, t, action):
239         if self._signal_type == 'UK':
240             return self._UK_signal(t, action)
241         if self._signal_type == 'IK':
242             return self._IK_signal(t, action)
243         if self._signal_type == 'RL':
244             return self._rl_maze_loop_signal(t, action)
245
246     def _UK_signal(self, t, action):
247         base_pos_coeff = self._evaluate_stage_coefficient(t, width=.5)
248         gait_stage_coeff = self._evaluate_gait_stage_coeff(t, action)
249         if self._is_render:
250             position, orientation, step_length, step_rotation, step_angle, step_periud =
251             self._read_inputs(base_pos_coeff, gait_stage_coeff)
252         else:
253             position = np.array([0.0,
254                                 self._base_y * base_pos_coeff,
255                                 -0.007])
256
257

```

```

Activities Text Editor * Apr 27 6:08PM • gallop.envy
Open -/anaconda/envs/rex/lib/python3.7/site-packages/ex_gym/envs/gym
Save X
Python Tab Width: 8 Ln 1, Col 1 INS
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266

```

Walking Environment:

```

Activities Text Editor • walk_env.py Apr 27 6:09 PM ● /anaconda3/envs/rex/lib/python3.7/site-packages/rex_gym/envs
Save □ ×

4 import math
5 import random
6
7 from gym import spaces
8 import numpy as np
9 from .. import rex_gym_env
10 from ...model import rex_constants
11 from ...model.gait_planner import GaitPlanner
12 from ...model.kinematics import Kinematics
13
14 NUM_LEGS = 4
15
16
17 class RexWalkEnv(rex_gym_env.RexGymEnv):
18     """The gym environment for the rex.
19
20     It simulates the locomotion of a rex, a quadruped robot. The state space
21     include the angles, velocities and torques for all the motors and the action
22     space contains motor angle for each leg. The reward function is based
23     on how far the rex walks in 2000 steps and penalizes the energy
24     expenditure or how near rex is to the target position.
25
26     """
27     metadata = {'render.modes': ['human', 'rgb_array'], 'video.frames_per_second': 60}
28     load_urdf = True
29     ts_terminating = False
30
31     def __init__(self,
32                  debug=False,
33                  urdf_version=None,
34                  control_time_step=0.005,
35                  action_repeat=10,
36                  control_latency=0,
37                  pd_latency=0,
38                  on_rack=False,
39                  motor_kp=1.0,
40                  motor_kd=0.02,
41                  render=True,
42                  num_steps_to_log=2000,
43                  env_randomizer=None,
44                  log_path=None,
45                  target_position=None,
46                  backwards=None,
47                  signal_type="ik",
48                  terrain_type="plane",
49                  terrain_id=0,
50                  mark_base=False):
51         """Initialize the rex alternating legs gym environment.
52
53         Args:
54             urdf_version: [DEFAULT_URDF_VERSION, DERPY_V0_URDF_VERSION] are allowable
55             versions. If None, DEFAULT_URDF_VERSION is used. Refer to
56             rex_urdf.py for more details.
57             control_time_step: The time step between two successive control signals.
58             action_repeat: The number of simulation steps that an action is repeated.
59
60
61
62
63
64
65
66
67
68
69
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133

```

```

Activities Text Editor • walk_env.py Apr 27 6:09 PM ● /anaconda3/envs/rex/lib/python3.7/site-packages/rex_gym/envs
Save □ ×

79     super(RexWalkEnv, self).__init__(urdf_version=urdf.version,
80                                     accurate_motor_model_enabled=True,
81                                     motor_overheat_protection=True,
82                                     hard_reset=True,
83                                     motor_kp=motor_kp,
84                                     motor_kd=motor_kd,
85                                     remove_default_cont_damping=False,
86                                     control_time_step=control_time_step,
87                                     action_repeat=action_repeat,
88                                     pd_latency=pd_latency,
89                                     on_rack=on_rack,
90                                     render=render,
91                                     num_steps_to_log=num_steps_to_log,
92                                     env_randomizer=env_randomizer,
93                                     log_path=log_path,
94                                     control_time_step=control_time_step,
95                                     action_repeat=action_repeat,
96                                     target_position=target_position,
97                                     signal_type=signal_type,
98                                     backwards=backwards,
99                                     debug=False,
100                                    terrain_id=terrain_id,
101                                    terrain_type=terrain_type,
102                                    markmark=False)
103
104     # (eventually) allow different feedback ranges/action spaces for different signals
105     action_max = {
106         'ik': 0.4,
107         'ol': 0.01
108     }
109     action_dim_map = {
110         'ik': 2,
111         'ol': 0
112     }
113     action_dim = action_dim_map[self._signal_type]
114     action_high = np.array([action_max[self._signal_type]] * action_dim)
115     set_action_space = spaces.Box(-action_high, action_high)
116     self._can_dil = 1.0
117     self._can_yaw = 0.0
118     self._can_pitch = -20
119     self._gait_planner = GaitPlanner("walk")
120     self._kinematics = Kinematics()
121     self._goal_reached = False
122     self._stay_still = False
123     self._terminating = False
124
125     def reset(self):
126         if self._int_pose == rex_constants.INIT_POSES["stand"]:
127             if self._signal_type == 'ol':
128                 self._int_pose = rex_constants.INIT_POSES["stand_ol"]
129             super(RexWalkEnv, self).reset(initial_motor_angles=self._init_pose, reset_duration=0.5)
130             self._goal_reached = False
131             self._terminating = False
132             self._stay_still = False
133             self._backwards.ts=None:

```

```

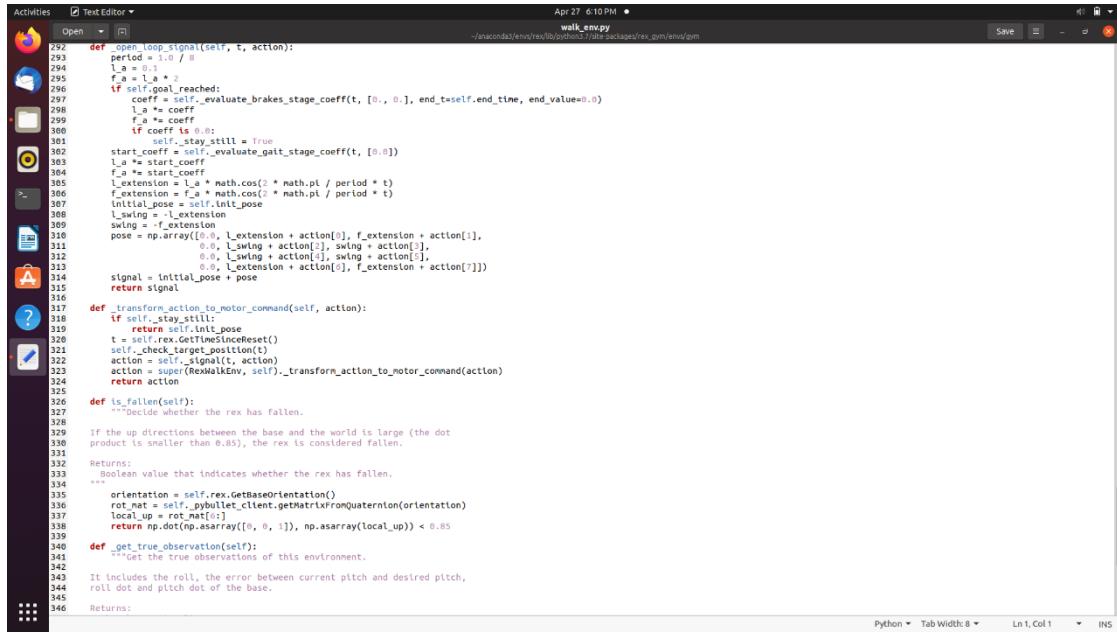
Activities  Text Editor *
Open ... walk.envy
File / anacoda/envs/rei/bullet3/site-packages/ex_gym/envs/gym
Save ... X
156     def setup_ul(self_, base_x, step, period):
157         self_.base_x_ut = self_._pybullet_client.addUserDebugParameter("base_x",
158             self_.ranges["base_x"][0],
159             self_.ranges["base_x"][1],
160             base_x)
161         self_.base_y_ut = self_._pybullet_client.addUserDebugParameter("base_y",
162             self_.ranges["base_y"][0],
163             self_.ranges["base_y"][1],
164             base_y)
165         self_.base_z_ut = self_._pybullet_client.addUserDebugParameter("base_z",
166             self_.ranges["base_z"][0],
167             self_.ranges["base_z"][1],
168             base_z)
169         self_.roll_ut = self_._pybullet_client.addUserDebugParameter("roll",
170             self_.ranges["roll"][0],
171             self_.ranges["roll"][1],
172             self_.ranges["roll"][2])
173         self_.pitch_ut = self_._pybullet_client.addUserDebugParameter("pitch",
174             self_.ranges["pitch"][0],
175             self_.ranges["pitch"][1],
176             self_.ranges["pitch"][2])
177         self_.yaw_ut = self_._pybullet_client.addUserDebugParameter("yaw",
178             self_.ranges["yaw"][0],
179             self_.ranges["yaw"][1],
180             self_.ranges["yaw"][2])
181         self_.step_length_ut = self_._pybullet_client.addUserDebugParameter("step_length",
182             -0.7, 0.7, step)
183         self_.step_rotation_ut = self_._pybullet_client.addUserDebugParameter("step_rotation",
184             -1.5, 1.5, 0.)
185         self_.step_angle_ut = self_._pybullet_client.addUserDebugParameter("step_angle",
186             -100, 100, 0.)
187         self_.step_period_ut = self_._pybullet_client.addUserDebugParameter("step_period",
188             0.2, 0.9, period)
189
190     def _read_inputs(self_, base_pos_coeff, gait_stage_coeff):
191         position = np.array([
192             self_._pybullet_client.readUserDebugParameter(self_.base_x_ut),
193             self_._pybullet_client.readUserDebugParameter(self_.base_y_ut) * base_pos_coeff,
194             self_._pybullet_client.readUserDebugParameter(self_.base_z_ut) * base_pos_coeff
195         ])
196         orientation = np.array([
197             self_._pybullet_client.readUserDebugParameter(self_.roll_ut) * base_pos_coeff,
198             self_._pybullet_client.readUserDebugParameter(self_.pitch_ut) * base_pos_coeff,
199             self_._pybullet_client.readUserDebugParameter(self_.yaw_ut) * base_pos_coeff
200         ])
201         step_length = self_._pybullet_client.readUserDebugParameter(self_.step_length_ut) * gait_stage_coeff
202         step_rotation = self_._pybullet_client.readUserDebugParameter(self_.step_rotation_ut)
203         step_angle = self_._pybullet_client.readUserDebugParameter(self_.step_angle_ut)
204         step_period = self_._pybullet_client.readUserDebugParameter(self_.step_period_ut)
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
211             if current_x >= 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 1 - ((current_t - p + beta)**2 / (width**2))
223         elif p - (beta/2) + end_t <= current_t <= p + end_t:
224             return 1 - ((current_t - p)**2 / (width**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 = 0.0 + 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 = 0.0 + action[1]
241         if end_t <= current_t <= p + end_t:
242             return 1 - (current_t - p - 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 == 'open_loop':
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=1.0)
254         gait_stage_coeff = self_.evaluate_gait_stage_coeff(t, action)
255         step = 0.0
256         perlin = 0.05
257         base_x = self_.base_x
258         if self_.backwards:
259             step = -.3
260             period = .5
261             base_x = .0

```

```

Activities  Text Editor *
Open ... walk.envy
File / anacoda/envs/rei/bullet3/site-packages/ex_gym/envs/gym
Save ... X
156     def setup_ul(self_, base_x, step, period):
157         self_.base_x_ut = 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_ut = self_._pybullet_client.addUserDebugParameter("base_y",
162             self_.ranges["base_y"][0],
163             self_.ranges["base_y"][1],
164             base_y)
165         self_.base_z_ut = self_._pybullet_client.addUserDebugParameter("base_z",
166             self_.ranges["base_z"][0],
167             self_.ranges["base_z"][1],
168             base_z)
169         self_.roll_ut = self_._pybullet_client.addUserDebugParameter("roll",
170             self_.ranges["roll"][0],
171             self_.ranges["roll"][1],
172             self_.ranges["roll"][2])
173         self_.pitch_ut = self_._pybullet_client.addUserDebugParameter("pitch",
174             self_.ranges["pitch"][0],
175             self_.ranges["pitch"][1],
176             self_.ranges["pitch"][2])
177         self_.yaw_ut = self_._pybullet_client.addUserDebugParameter("yaw",
178             self_.ranges["yaw"][0],
179             self_.ranges["yaw"][1],
180             self_.ranges["yaw"][2])
181         self_.step_length_ut = self_._pybullet_client.addUserDebugParameter("step_length",
182             -0.7, 0.7, step)
183         self_.step_rotation_ut = self_._pybullet_client.addUserDebugParameter("step_rotation",
184             -1.5, 1.5, 0.)
185         self_.step_angle_ut = self_._pybullet_client.addUserDebugParameter("step_angle",
186             -100, 100, 0.)
187         self_.step_period_ut = self_._pybullet_client.addUserDebugParameter("step_period",
188             0.2, 0.9, period)
189
190     def _read_inputs(self_, base_pos_coeff, gait_stage_coeff):
191         position = np.array([
192             self_._pybullet_client.readUserDebugParameter(self_.base_x_ut),
193             self_._pybullet_client.readUserDebugParameter(self_.base_y_ut) * base_pos_coeff,
194             self_._pybullet_client.readUserDebugParameter(self_.base_z_ut) * base_pos_coeff
195         ])
196         orientation = np.array([
197             self_._pybullet_client.readUserDebugParameter(self_.roll_ut) * base_pos_coeff,
198             self_._pybullet_client.readUserDebugParameter(self_.pitch_ut) * base_pos_coeff,
199             self_._pybullet_client.readUserDebugParameter(self_.yaw_ut) * base_pos_coeff
200         ])
201         step_length = self_._pybullet_client.readUserDebugParameter(self_.step_length_ut) * gait_stage_coeff
202         step_rotation = self_._pybullet_client.readUserDebugParameter(self_.step_rotation_ut)
203         step_angle = self_._pybullet_client.readUserDebugParameter(self_.step_angle_ut)
204         step_period = self_._pybullet_client.readUserDebugParameter(self_.step_period_ut)
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
211             if current_x >= 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 1 - ((current_t - p + beta)**2 / (width**2))
223         elif p - (beta/2) + end_t <= current_t <= p + end_t:
224             return 1 - ((current_t - p)**2 / (width**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 = 0.0 + 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 = 0.0 + action[1]
241         if end_t <= current_t <= p + end_t:
242             return 1 - (current_t - p - 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 == 'open_loop':
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=1.0)
254         gait_stage_coeff = self_.evaluate_gait_stage_coeff(t, action)
255         step = 0.0
256         perlin = 0.05
257         base_x = self_.base_x
258         if self_.backwards:
259             step = -.3
260             period = .5
261             base_x = .0

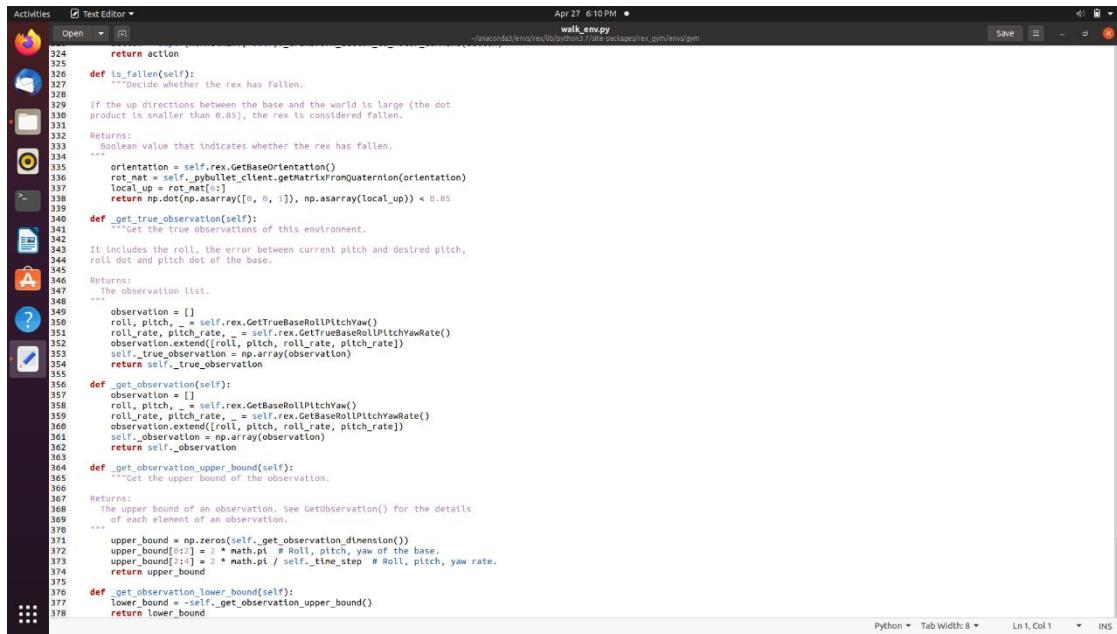
```



```

Activities Text Editor * walk.py
Open ... Apr 27 6:10PM ● /anaconda/envs/rex/lib/python3.7/site-packages/ex_gym/envs/gym
Save X
292     def _open_loop_signal(self, t, action):
293         period = 1.0 / 8
294         l_a = 1.1
295         f_a = l_a * 2
296         if self._goal_reached:
297             l_a *= evaluate_brakes_stage_coeff(t, [0., 0.], end_t=self.end_time, end_value=0.0)
298             f_a *= coeff
299             l_a *= coeff
300             if coeff == 0.0:
301                 self._stay_still = True
302             start_coeff = self._evaluate_gait_stage_coeff(t, [0.0])
303             l_a *= start_coeff
304             f_a *= start_coeff
305             l_extension = f_a * math.cos(2 * math.pi / period * t)
306             f_extension = f_a * math.cos(2 * math.pi / period * t)
307             initial_pose = self.int_pose
308             l_swing = -l_extension
309             swing = -f_extension
310             pose = np.array([0.0, Lextension + action[0], f_extension + action[1],
311                             0.0, Lswing + action[2], swing + action[3],
312                             0.0, Lswing + action[4], swing + action[5],
313                             0.0, Lswing + action[6], f_extension + action[7]])
314             signal = initial_pose + pose
315             return signal
316
317     def transform_action_to_motor_command(self, action):
318         if self._stay_still:
319             return self.int_pose
320         t = self.rex.GetTimeStep()
321         self.rex.SetTargetPosition(t)
322         action = self._signal(t, action)
323         action = super(RexWalkEnv, self).transform_action_to_motor_command(action)
324         return action
325
326     def is_fallen(self):
327         """decide whether the rex has fallen.
328
329         If the up directions between the base and the world is large (the dot
330         product is smaller than 0.85), the rex is considered fallen.
331
332         Returns:
333             Boolean value that indicates whether the rex has fallen.
334         """
335         orientation = self.rex.GetBaseOrientation()
336         rot_mat = self.pybullet_client.getMatrixFromQuaternion(orientation)
337         local_up = rot_mat[:, 2]
338         return np.dot(np.asarray([0., 0., 1]), np.asarray(local_up)) < 0.85
339
340     def get_true_observation(self):
341         """Get the true observations of this environment.
342
343         It includes the roll, the error between current pitch and desired pitch,
344         roll dot and pitch dot of the base.
345
346         Returns:
347             The observation list.
348         """
349         observation = []
350         roll, pitch, yaw = self.rex.GetBaseRollPitchYaw()
351         roll_rate, pitch_rate, yaw_rate = self.rex.GetTrueBaseRollPitchYawRate()
352         observation.extend([roll, pitch, roll_rate, pitch_rate])
353         self._true_observation = np.array(observation)
354         return self._true_observation
355
356     def get_observation(self):
357         observation = []
358         roll, pitch, yaw = self.rex.GetBaseRollPitchYaw()
359         roll_rate, pitch_rate, yaw_rate = self.rex.GetBaseRollPitchYawRate()
360         observation.extend([roll, pitch, roll_rate, pitch_rate])
361         self._observation = np.array(observation)
362         return self._observation
363
364     def get_observation_upper_bound(self):
365         """Get the upper bound of the observation.
366
367         Returns:
368             The upper bound of an observation. See GetObservation() for the details
369             of each element of an observation.
370
371             upper_bound = np.zeros(self._get_observation_dimension())
372             upper_bound[0:3] = 2 * math.pi # Roll, pitch, yaw of the base.
373             upper_bound[3:6] = 2 * math.pi / self._time_step # Roll, pitch, yaw rate.
374
375     def _get_observation_lower_bound(self):
376         lower_bound = -self._get_observation_upper_bound()
377
378         return lower_bound

```



```

Activities Text Editor * walk.py
Open ... Apr 27 6:10PM ● /anaconda/envs/rex/lib/python3.7/site-packages/ex_gym/envs/gym
Save X
324         return action
325
326     def is_fallen(self):
327         """decide whether the rex has fallen.
328
329         If the up directions between the base and the world is large (the dot
330         product is smaller than 0.85), the rex is considered fallen.
331
332         Returns:
333             Boolean value that indicates whether the rex has fallen.
334         """
335         orientation = self.rex.GetBaseOrientation()
336         rot_mat = self.pybullet_client.getMatrixFromQuaternion(orientation)
337         local_up = rot_mat[:, 2]
338         return np.dot(np.asarray([0., 0., 1]), np.asarray(local_up)) < 0.85
339
340     def get_true_observation(self):
341         """Get the true observations of this environment.
342
343         It includes the roll, the error between current pitch and desired pitch,
344         roll dot and pitch dot of the base.
345
346         Returns:
347             The observation list.
348         """
349         observation = []
350         roll, pitch, yaw = self.rex.GetBaseRollPitchYaw()
351         roll_rate, pitch_rate, yaw_rate = self.rex.GetTrueBaseRollPitchYawRate()
352         observation.extend([roll, pitch, roll_rate, pitch_rate])
353         self._true_observation = np.array(observation)
354         return self._true_observation
355
356     def get_observation(self):
357         observation = []
358         roll, pitch, yaw = self.rex.GetBaseRollPitchYaw()
359         roll_rate, pitch_rate, yaw_rate = self.rex.GetBaseRollPitchYawRate()
360         observation.extend([roll, pitch, roll_rate, pitch_rate])
361         self._observation = np.array(observation)
362         return self._observation
363
364     def get_observation_upper_bound(self):
365         """Get the upper bound of the observation.
366
367         Returns:
368             The upper bound of an observation. See GetObservation() for the details
369             of each element of an observation.
370
371             upper_bound = np.zeros(self._get_observation_dimension())
372             upper_bound[0:3] = 2 * math.pi # Roll, pitch, yaw of the base.
373             upper_bound[3:6] = 2 * math.pi / self._time_step # Roll, pitch, yaw rate.
374
375     def _get_observation_lower_bound(self):
376         lower_bound = -self._get_observation_upper_bound()
377
378         return lower_bound

```

Turn Environment:

The screenshot shows a Jupyter Notebook interface with the following details:

- Activities**: Shows the current tab is "Text Editor".
- File Menu**: Includes "Open", "Save", and "Exit".
- Code Area**: The code is written in Python and defines a class `RexEnv` for a quadruped robot simulation. The code includes imports for `math`, `random`, `spaces`, `numpy`, `GaitPlanner`, `pybullet_data`, and various modules from `reinforcement_learning_gym`. It sets parameters like `NUM_LEGS=4` and `STEP_PERIOD=1.0 / 10.0`. The class `RexEnv` has an __init__ method with many parameters, including `control_time_step=0.005` and `control_latency=0`. It also includes a docstring describing the environment's purpose and a metadata section.
- Output Area**: Shows the command `!run env.py` and the output "Apr 27 6:10 PM •".
- Bottom Bar**: Shows tabs for "Python", "Tab Width: 8", and "Ln 1, Col 1".

The screenshot shows a Jupyter Notebook interface with the following details:

- Activities**: Shows the current tab is "Text Editor".
- File**: Includes options like Open, Save, and Exit.
- Code Content**: The code is in a file named `tum_enm.py` located at `-/anaconda3/envs/rex/lib/python3.7/site-packages/rex_gym/rex_gym/gym`. The code defines a class `reset(self)` for a robot simulation. It handles initial poses, target orientations, and camera setup. It also includes methods for setting up UI parameters and adding user debug parameters for base and roll/pitch angles.

```
Activities Text Editor ● April 27 6:11 PM ● turn_enmpy -[anaconda]/envs/reverb/lib/python3.7/site-packages/reverb/gym
Save □ ×

192     def _read_inputs(self, base_pos_coeff, gait_stage_coeff):
193         position = np.array([
194             [
195                 self._pybullet_client.readUserDebugParameter(self.base_y_u1),
196                 self._pybullet_client.readUserDebugParameter(self.base_y_u1) * base_pos_coeff,
197                 self._pybullet_client.readUserDebugParameter(self.base_z_u1) * base_pos_coeff
198             ]
199         ])
200         orientation = np.array([
201             [
202                 self._pybullet_client.readUserDebugParameter(self.roll_u1) * base_pos_coeff,
203                 self._pybullet_client.readUserDebugParameter(self.pitch_u1) * base_pos_coeff,
204                 self._pybullet_client.readUserDebugParameter(self.yaw_u1) * base_pos_coeff
205             ]
206         ])
207         step_length = self._pybullet_client.readUserDebugParameter(self.step_length_u1) * gait_stage_coeff
208         step_rotation = self._pybullet_client.readUserDebugParameter(self.step_rotation_u1) * gait_stage_coeff
209         step_angle = self._pybullet_client.readUserDebugParameter(self.step_angle_u1)
210         step_period = self._pybullet_client.readUserDebugParameter(self.step_period_u1)
211
212         return position, orientation, step_length, step_rotation, step_angle, step_period
213
214     def _signal(self, t, action):
215         if self._signal_type == 'ik':
216             return self._IK_signal(t, action)
217         if self._signal_type == 'open':
218             return self._open_loop_signal(t, action)
219
220     @staticmethod
221     def _evaluate_base_stage_coeff(current_t, end_t=0.0, width=0.001):
222         # Sigmoid function
223         beta = p = width
224         if current_t < end_t < current_t + p - (beta / 2) + end_t:
225             return 1 - ((beta / 2) * (current_t - p + beta)) ** 2
226         elif p - (beta / 2) + end_t <= current_t <= p + end_t:
227             return 1 - ((beta / 2) * (current_t - p)) ** 2
228         else:
229             return 1
230
231     @staticmethod
232     def _evaluate_gait_stage_coeff(current_t, end_t=0.0):
233         # Ramp function
234         p = width
235         if end_t <= current_t <= p + end_t:
236             return current_t
237         else:
238             return 1.0
239
240     def _IK_signal(self, t, action):
241         gait_stage_coeff = self._evaluate_gait_stage_coeff(t)
242         if self._is_render and self._is_debug:
243             position, orientation, step_length, step_rotation, step_angle, step_period = \
244                 self._read_inputs(base_pos_coeff, gait_stage_coeff)
245         else:
246             step_dir.value = -0.5 * gait_stage_coeff
```

Standup Environment:

```

4 import math
5 import random
6
7 from gym import spaces
8 import numpy as np
9 from .. import rex_gym_env
10 from ...model import rex_constants
11 from ...model.rex import Rex
12
13 NUM_LEGS = 4
14 NUM_MOTORS = 3 * NUM_LEGS
15
16
17 class RexStandupEnv(rex_gym_env.RexGymEnv):
18     """The gym environment for the rex.
19
20     It simulates the locomotion of a rex, a quadruped robot. The state space
21     include the angles, velocities and torques for all the motors and the action
22     include the desired motor angle for each motor. The reward function is based
23     on how far the rex walks in 1000 steps and penalizes the energy
24     expenditure.
25
26     """
27     metadata = {'render.modes': ['human', "rgb_array"], "video.frames_per_second": 60}
28
29     def __init__(self,
30                  debug=False,
31                  urdf_version=None,
32                  control_time_step=0.005,
33                  action_repeat=1,
34                  control_latency=0,
35                  pd_latency=0,
36                  on_rack=False,
37                  motor_kd=0.0,
38                  motor_kdmg=0.0,
39                  remove_default_joint_damping=False,
40                  render=False,
41                  num_steps_to_log=1000,
42                  env_randomizer=None,
43                  log_path=None,
44                  signal_type='rl',
45                  terrain_id='plane',
46                  terrain_id_name='',
47                  mark_base=''):
48         """Initialize the rex alternating legs gym environment.
49
50         Args:
51             urdf_version: [DEFAULT_URDF_VERSION, DERPY_V0_URDF_VERSION] are allowable
52             versions. If None, DEFAULT_URDF_VERSION is used. Refer to
53             rex_urdf.py for more details.
54             control_time_step: The time step between two successive control signals.
55             action_repeat: The number of simulation steps that an action is repeated.
56             control_latency: The latency between get_observation() and the actual
57             observation. See initmotor.py for more details.
58             pd_latency: The latency used to set motor_angles/velocities used to
59
59

```

```

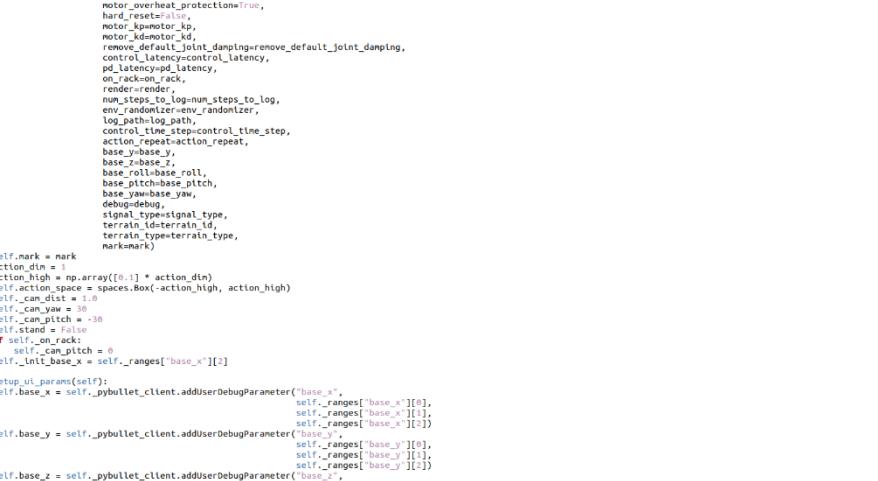
75     """super(RexStandupEnv,
76             self).__init__(urdf_version=urdf_version,
77                            accurate_motor_model_enabled=True,
78                            motor_overheat_protection=True,
79                            hard_safety_limits=True,
80                            motor_kp=motor_kp,
81                            motor_kd=motor_kd,
82                            remove_default_joint_damping=remove_default_joint_damping,
83                            control_latency=control_latency,
84                            pd_latency=pd_latency,
85                            on_rack=on_rack,
86                            render=render,
87                            num_steps_to_log=num_steps_to_log,
88                            env_randomizer=env_randomizer,
89                            log_path=log_path,
90                            control_time_step=control_time_step,
91                            action_repeat=action_repeat,
92                            signal_type=signal_type,
93                            signal_type=signal_type,
94                            debug=debug,
95                            terrain_id=terrain_id,
96                            terrain_type=terrain_type,
97                            mark_base=mark_base)
98
99     action_dim = 3
100    action_high = np.array([0.1] * action_dim)
101    self.action_space = spaces.Box(-action_high, action_high)
102    self._can_dist = 1.0
103    self._can_pitch = 30
104    self._can_pitch = -30
105    self._on_rack = False
106    self._can_pitch = 0
107
108    def reset(self):
109        super(RexStandupEnv, self).reset(initial_motor_angles=rex_constants.INIT_POSES['rest_position'],
110                                         reset_duration=0.5)
111        return self._get_observation()
112
113    def _signal(self, t, action):
114        if t < 0:
115            return rex_constants.INIT_POSES['stand']
116        t += 1
117        # apply a 'brake' function
118        signal = rex_constants.INIT_POSES['stand'] * ((1.0 + action[0]) / t + 1.0)
119        return signal
120
121    @staticmethod
122    def _convert_from_leg_model(leg_pose):
123        motor_pose = np.zeros(NUM_MOTORS)
124        for i in range(NUM_LEGS):
125            motor_pose[i * 3 + 0] = leg_pose[i * 3 + 0]
126            motor_pose[i * 3 + 1] = leg_pose[i * 3 + 1]
127            motor_pose[i * 3 + 2] = leg_pose[i * 3 + 2]
128        return motor_pose
129

```

The screenshot shows a Jupyter Notebook interface with the following details:

- File Path:** /anaconda3/envs/rex/lib/python3.7/site-packages/ex_gym/envs/gym
- Code Content:** The code is a Python class for a Rex environment. It includes methods for transforming actions to motor commands, checking termination, determining if the Rex has fallen, calculating reward based on position, and getting true observations. It also handles roll, pitch, and roll rate calculations.
- Code Lines:** The code spans from line 130 to 944, with line numbers visible on the left.
- Environment:** The interface includes icons for file operations (Open, Save, etc.) and a toolbar at the top.

Posing Environment:



The screenshot shows a Jupyter Notebook interface with a single code cell containing a Python script. The script is titled 'poses_env.py' and is located at the path '/anaconda3/envs/ex1/lib/python3.7/site-packages/re_oxym/envs/gym'. The code defines a class 'posesEnv' that inherits from 'gym.Env'. It includes methods for initializing the environment, performing actions, and observing states. The code uses PyBullet for physics simulation and ROS for communication. It also handles camera parameters and terrain types.

```
    7     super().__init__(urdf_version=urdf_version,
  8         self).__init__(accurate_motor_model_enabled=True,
  9         motor_overheat_protection=True,
 10        motor_kp=1000,
 11        motor_kpmotor_kp,
 12        motor_kd=100,
 13        motor_kdsmotor_kd,
 14        remove_default_joint_damping=remove_default_joint_damping,
 15        control_latency=control_latency,
 16        pd_latency=pd_latency,
 17        on_rack=on_rack,
 18        render=render,
 19        run_stuck=run_stuck,
 20        log_num_steps_to_log=log_num_steps_to_log,
 21        env_randomizer=env_randomizer,
 22        log_path=log_path,
 23        control_time_step=control_time_step,
 24        action_repeat=action_repeat,
 25        action_repeat_action_repeat,
 26        base_y_base_y,
 27        base_z_base_z,
 28        base_x_base_x,
 29        base_x_base_x,
 30        base_pitch=base_pitch,
 31        base_yaw=base_yaw,
 32        debug=debug,
 33        signal_type=signal_type,
 34        terrain_id=terrain_id,
 35        terrain_type=terrain_type,
 36        markmark)
 37
 38     self.mark = mark
 39     action_dim = 1
 40     action_high = np.array([0.1 * action_dim])
 41     self._cam_dist = 1.0
 42     self._cam_yaw = 30
 43     self._cam_pitch = -30
 44
 45     if self._on_rack:
 46         self._cam_pitch = 0
 47         self._cam_base_x = self._ranges["base_x"][[2]]
 48
 49     def setup_ur_params(self):
 50         self.base_x = self._pybullet_client.addUserDebugParameter("base_x",
 51             self._ranges["base_x"][[0]],
 52             self._ranges["base_x"][[1]],
 53             self._ranges["base_x"][[2]])
 54
 55         self.base_y = self._pybullet_client.addUserDebugParameter("base_y",
 56             self._ranges["base_y"][[0]],
 57             self._ranges["base_y"][[1]],
 58             self._ranges["base_y"][[2]])
 59
 60         self.base_z = self._pybullet_client.addUserDebugParameter("base_z",
 61             self._ranges["base_z"][[0]],
 62             self._ranges["base_z"][[1]],
 63             self._ranges["base_z"][[2]])
 64
 65         self.roll = self._pybullet_client.addUserDebugParameter("roll",
 66             self._ranges["roll"][[0]],
 67             self._ranges["roll"][[1]]),
 68
 69
 70
 71
 72
 73
 74
 75
 76
 77
 78
 79
 80
 81
 82
 83
 84
 85
 86
 87
 88
 89
 90
 91
 92
 93
 94
 95
 96
 97
 98
 99
 100
 101
 102
 103
 104
 105
 106
 107
 108
 109
 110
 111
 112
 113
 114
 115
 116
 117
 118
 119
 120
 121
 122
 123
 124
 125
 126
 127
 128
 129
 130
 131
 132
 133
 134
 135
 136
 137
 138
 139
 140
```

The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** Activities, Text Editor, Open, poset_env.py, Save.
- Header:** April 27 6:14 PM
- Code Area:** The code is written in Python and defines a class `PosetEnv` with methods like `reset`, `full_next_pose_and_target`, `evaluate_stage_coefficient`, and `_signal`. It uses various imports from `__future__` and `os` modules, along with `numpy` and `gym`.
- Left Sidebar:** Includes icons for file operations (New, Open, Save, etc.), a search bar, and a help icon.
- Bottom Status Bar:** Tab Width: 8, Ln 33, Col 27, INS.

```

Activities Text Editor * Open ... pose.py -/anaconda3/envs/rex/lib/python3.7/site-packages/ex_gym/envs/gym
227     def _read_inputs(self):
228         position = np.array([
229             self._pybullet_client.readUserDebugParameter(self.base_x),
230             self._pybullet_client.readUserDebugParameter(self.base_y),
231             self._pybullet_client.readUserDebugParameter(self.base_z)
232         ])
233
234         orientation = np.array([
235             self._pybullet_client.readUserDebugParameter(self.roll),
236             self._pybullet_client.readUserDebugParameter(self.pitch),
237             self._pybullet_client.readUserDebugParameter(self.yaw)
238         ])
239
240     return position, orientation
241
242     @staticmethod
243     def _convert_from_leg_model(leg_pose):
244         motor_pose = np.zeros(NUM_MOTORS)
245         for l in range(NUM_LEGS):
246             motor_pose[l * i] = leg_pose[i]
247             motor_pose[l * i + 1] = leg_pose[i + 1]
248             motor_pose[l * i + 2] = leg_pose[i + 2]
249             motor_pose[l * i + 3] = leg_pose[i + 3]
250
251         return motor_pose
252
253     def _transform_action_to_motor_command(self, action):
254         action = self._signal(self.rex.GetTimeSinceReset(), action)
255         action = self._convert_from_leg_model(action)
256         action = super(RexPosesEnv, self)._transform_action_to_motor_command(action)
257         return action
258
259     def _is_fallen(self):
260         # Decide whether the rex has fallen.
261         Returns: Boolean value that indicates whether the rex has fallen.
262         ...
263         roll, _, _ = self.rex.GetTrueBaseRollPitchYaw()
264
265         return False
266
267     def _reward(self):
268         # positive reward as long as rex stands
269         return 1.0
270
271     def _get_true_observation(self):
272         """Get the true observations of this environment.
273
274         It includes the roll, the error between current pitch and desired pitch,
275         roll dot and pitch dot of the base.
276
277         Returns:
278             The observation list.
279             ...
280
281         observation = []
282         roll, pitch, _ = self.rex.GetTrueBaseRollPitchYaw()

```

The screenshot shows a terminal window titled 'Activities Text Editor *' with the file 'pose.py' open. The code is a Python script for a 'RexPosesEnv' environment. It includes methods for reading inputs from a PyBullet client, converting leg model poses to motor commands, determining if the rex has fallen, calculating rewards, and getting true observations. The code uses NumPy arrays for positions and orientations.

Terrain Implementation:

```

Activities Text Editor * Open ... terrain.py -/anaconda3/envs/rex/lib/python3.7/site-packages/ex_gym/envs
1 #!/usr/bin/env python
2
3 import pybullet_data as pd
4 import gym.util.pybullet_data as rpd
5 import pybullet as p
6 import random
7
8 from rex_gym.util import flag_mapper
9
10 FLAG_TO_FILENAME = {
11     'maze': 'heightmaps/maze_height_out.png',
12     'random': 'heightmaps/random.png'
13 }
14
15 ROBOT_INIT_POSITION = [
16     'mounts': [0, 0, .85],
17     'plane': [0, 0, 0.21],
18     'walls': [0, 0, 1.0],
19     'maze': [0, 0, 0.1],
20     'random': [0, 0, 0.21]
21 ]
22
23 class Terrain:
24
25     def __init__(self, terrain_source, terrain_id, columns=256, rows=256):
26         Random.seed(0)
27         self.terrain_source = terrain_source
28         self.terrain_id = terrain_id
29         self.columns = columns
30         self.rows = rows
31
32     def generate_terrain(self, env, height_perturbation_range=0.05):
33         env.pybullet_client.setAdditionalSearchPath(pd.getDataPath())
34         env.pybullet_client.configureDebugVisualizer(env.pybullet_client.COV_ENABLE_RENDERING, 0)
35         height_perturbation_range = height_perturbation_range
36         terrain = np.zeros((self.rows, self.columns))
37         if self.terrain_source == 'random':
38             for j in range(int(self.columns / 2)):
39                 for i in range(int(self.rows / 2)):
40                     height = random.uniform(-height_perturbation_range, height_perturbation_range)
41                     terrain_data[i + 1 + j * self.rows] = height
42                     terrain_data[i + 1 + 2 + j * self.rows] = height
43                     terrain_data[i + 1 + (j + 1) * self.rows] = height
44                     terrain_data[i + 1 + (j + 2) * self.rows] = height
45             terrain.shape = env.pybullet_client.createCollisionShape(
46                 shapeType=gym_util.pybullet_client.GEOM_HEIGHTFIELD,
47                 meshScale=[0, 0, 1],
48                 heightfieldData=height * np.ones(self.rows * 2),
49                 heightfieldDataDownsampled=height * np.ones(self.rows),
50                 numHeightfieldBanks=1,
51                 numHeightfieldDivisions=1)
52             terrain = env.pybullet_client.createMultiBody(0, terrain.shape)
53             env.pybullet_client.resetBasePositionAndOrientation(terrain, [0, 0, 0], [0, 0, 0, 1])
54
55         if self.terrain_source == 'maze':
56             terrain_shape = env.pybullet_client.createCollisionShape(

```

The screenshot shows a terminal window titled 'Activities Text Editor *' with the file 'terrain.py' open. The code is a Python script for generating terrain. It defines a 'Terrain' class with methods for initializing terrain based on source ('random' or 'maze') and generating heightmaps. The terrain is represented as a 2D array of heights, which are then converted into collision shapes for use in a PyBullet environment. The code uses NumPy and PyBullet libraries.

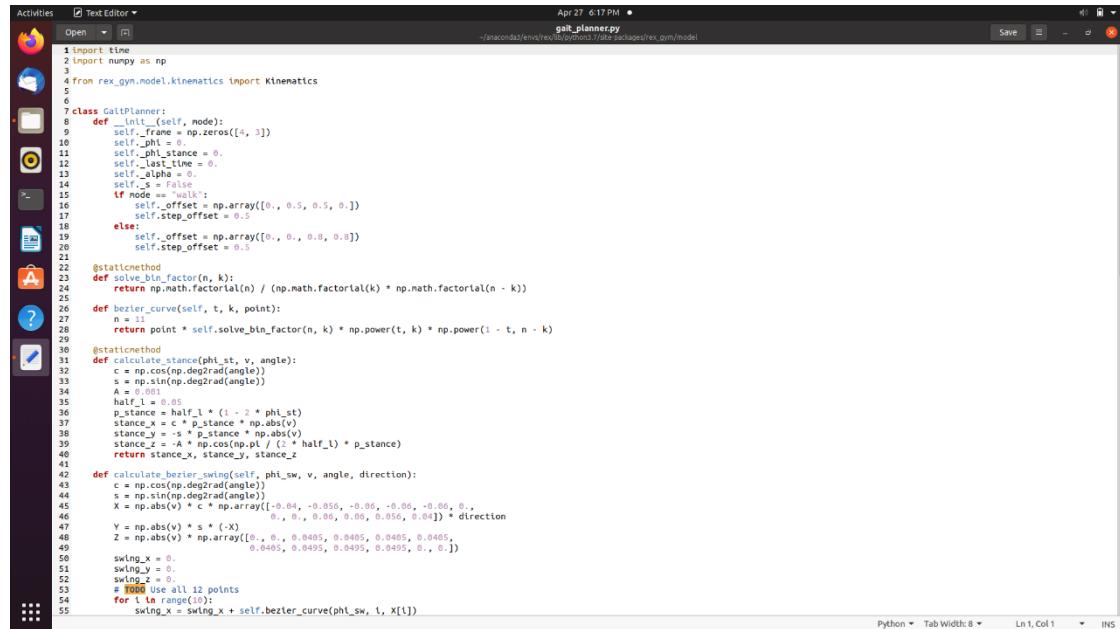
```
Activities Text Editor Apr 27 16:16PM • train.py -> (anaconda)envs/lib/python3.7/site-packages/ex_01/mode...
Open Save x
52 terrain = env.pybullet_client.createMultiBody(0, terrain_shape)
53 env.pybullet_client.resetBasePositionAndOrientation(terrain, [0, 0, 0], [0, 0, 0, 1])
54
55 if self.terrain_source == 'CMF':
56     terrain_shape = env.pybullet_client.createCollisionShape(
57         shapeType=env.pybullet_client.GEOM_HEIGHTFIELD,
58         meshScale=[.05, .05, 1],
59         fileName="heightfield/ground0.txt",
60         heightfieldTextureScaling=120)
61     terrain = env.pybullet_client.createMultiBody(0, terrain_shape)
62     textureId = env.pybullet_client.loadTexture("fcpd/getFolderPath()//grass.png")
63     env.pybullet_client.changeVisualShape(terrain, -1, textureUniqueId=textureId)
64     env.pybullet_client.resetBasePositionAndOrientation(terrain, [1, 0, 0], [0, 0, 0, 1])
65
66 # Todo: do this better
67 if self.terrain_source == 'long':
68     terrain_shape = env.pybullet_client.createCollisionShape(
69         shapeType=env.pybullet_client.GEOM_HEIGHTFIELD,
70         meshScale=[.05, .05, 1] if self.terrain_id == "mounts" else 1,
71         fileName="heightfield/toy_mountain0.txt")
72     terrain = env.pybullet_client.createMultiBody(0, terrain_shape)
73     if self.terrain_id == "mounts":
74         textureId = env.pybullet_client.loadTexture("heightmaps/gimp_overlay_out.png")
75         env.pybullet_client.changeVisualShape(terrain, -1, textureUniqueId=textureId)
76         env.pybullet_client.resetBasePositionAndOrientation(terrain, [0, 0, 0], [0, 0, 0, 1])
77     else:
78         env.pybullet_client.resetBasePositionAndOrientation(terrain, [0, 0, 0], [0, 0, 0, 1])
79
80     self.terrain_shape = terrain_shape
81     env.pybullet_client.changeVisualShape(terrain, -1, rgbaColor=[1, 1, 1, 1])
82     env.pybullet_client.configureDebugVisualizer(env.pybullet_client.COV_ENABLE_RENDERING, 1)
83
84 def update_terrain(self, height_perturbation_range=.85):
85     if self.terrain_source == flag_mapper.TERRAIN_TYPE['random']:
86         terrain_data = [0] * self.columns * self.rows
87         for i in range(int(self.rows / 2)):
88             for l in range(int(self.rows / 2)):
89                 height = random.uniform(0, height_perturbation_range)
90                 terrain_data[i * 2 + 2 * j + self.rows * l] = height
91                 terrain_data[i * 2 + 2 * j + 1 + self.rows * l] = height
92                 terrain_data[i * 2 + 2 * (j + 1) + self.rows * l] = height
93                 terrain_data[i * 2 + 1 + (2 * j + 1) * self.rows * l] = height
94
95     # GEOM_CONCAVE_INTERNAL_EDGE may help avoid getting stuck at an internal (shared) edge of
96     # the triangle/heightfield. GEOM_CONCAVE_INTERNAL_EDGE is a bit slower to build though.
97     flags = p.GEOM_CONCAVE_INTERNAL_EDGE
98     # flags = 0
99     self.terrain_shape = p.createCollisionShape(
100         shapeType=p.GEOM_HEIGHTFIELD,
101         meshScale=[.05, .05, 1],
102         heightfieldTexRescaling=(self.rows - 1) / 2,
103         heightfieldRows=1,
104         numHeightfieldRows=1,
105         numHeightfieldColumns=self.columns,
106         replaceHeightfieldIndex=self.terrain_shape)
```

Rex Constant:



```
Activities Text Editor Open Save □ x
1 import numpy as np
2
3 CRM_POSES = [
4     "rest": np.array([
5         -1.0, -1.6, 0.,
6         0., 1.6, 0.,
7     ]),
8 ]
9
10 INIT_POSES5 = [
11     "stand": np.array([
12         0., -0.88643435, 1.30197369,
13         0., -0.88643435, 1.30197369,
14         0., -0.88643435, 1.30197369,
15         0., -0.88643435, 1.30197369
16     ]),
17     "stand_0": np.array([
18         0.15192765, -0.20412283, 1.48156545,
19         -0.15192765, -0.98412283, 1.48156545,
20         0.15192765, -0.98412283, 1.48156545,
21         -0.15192765, -0.98412283, 1.48156545
22 ]),
23     "gallow": np.array([
24         0.15192765, -0.4142283, 1.48156545,
25         -0.15192765, -0.98412283, 1.48156545,
26         0.15192765, -0.98412283, 1.48156545,
27         -0.15192765, -0.98412283, 1.48156545
28 ]),
29     "stand_low": np.array([
30         0.1, -0.62, 1.35,
31         -0.1, -0.62, 1.35,
32         0.1, -0.87, 1.35,
33         -0.1, -0.87, 1.35
34 ]),
35     "stand_high": np.array([
36         0.1, -0.658319, 1.0472,
37         -0.1, -0.658319, 1.0472,
38         0., -0.658319, 1.0472,
39         0., -0.658319, 1.0472
40 ]),
41     "rest_position": np.array([
42         -0.4, -1.5, 6,
43         0.4, -1.5, 6,
44         -0.4, -1.5, 6,
45         0.4, -1.5, 6
46 ])
47 }
```

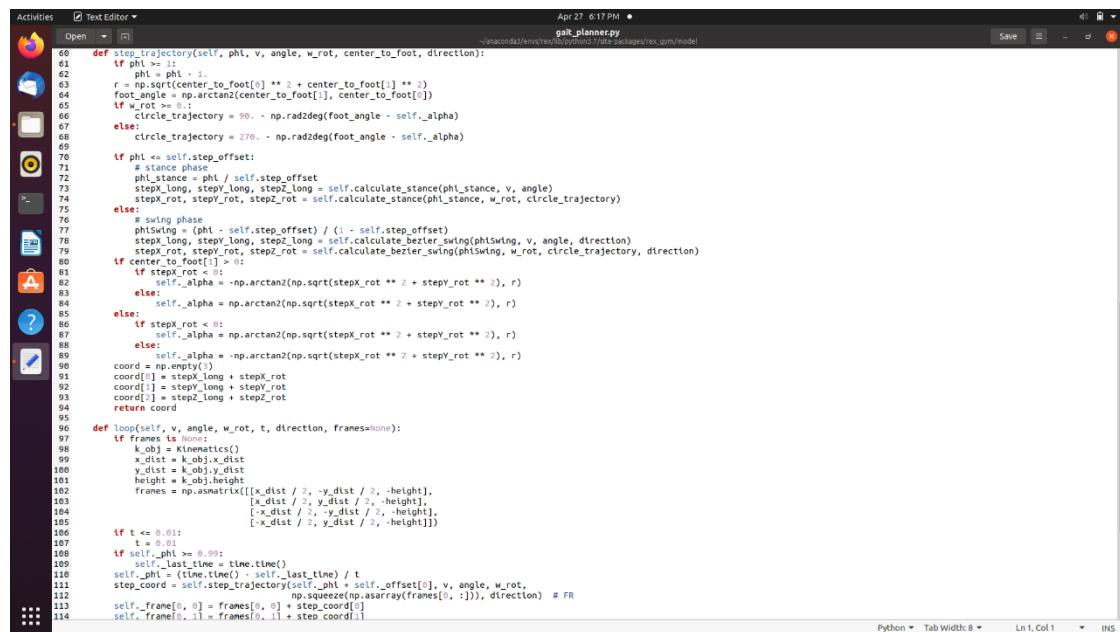
Gait Planner:



```

Activities Text Editor * April 27 6:17 PM • gait_planner.py -/jacobson3/ens/rex/obj/rex3/rls/packages/rex_gym/mode
Open [ ] Save [ ] Python Tab Width: 8 ▾ Ln 1, Col 1 ▾ INS
1 import time
2 import numpy as np
3 from rex_gym.model.kinematics import Kinematics
4
5
6 class GaitPlanner:
7     def __init__(self, mode):
8         self._frame = np.zeros([4, 3])
9         self._phi = 0.
10        self._p_stance = 0.
11        self._last_time = 0.
12        self._alpha = 0.
13        self._s = False
14
15    @property
16        self._offset = np.array([0., 0.5, 0.5, 0.])
17        self.step_offset = 0.5
18
19    else:
20        self._offset = np.array([0., 0., 0.8, 0.0])
21        self.step_offset = 0.1
22
23    @staticmethod
24        solve_bin_factor(n, k):
25            return np.math.factorial(n) / (np.math.factorial(k) * np.math.factorial(n - k))
26
27    def bezier_curve(self, t, k, point):
28        n = 11
29        return point * self.solve_bin_factor(n, k) * np.power(t, k) * np.power(1 - t, n - k)
30
31    @staticmethod
32        calculate_stance(phi_st, v, angle):
33            c = np.cos(np.deg2rad(angle))
34            s = np.sin(np.deg2rad(angle))
35            A = np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]])
36            half_l = 0.85
37            p_stance = half_l * (1 - v * phi_st)
38            stance_x = c * p_stance + v * abs(v)
39            stance_y = s * p_stance + v * abs(v)
40            stance_z = A * np.cos(np.pi / (2 * half_l)) * p_stance
41            return stance_x, stance_y, stance_z
42
43    def calculate_bezier_swing(self, phi_sw, v, angle, direction):
44        c = np.cos(np.deg2rad(angle))
45        s = np.sin(np.deg2rad(angle))
46        x = np.abs(v) * c * np.array([-0.04, -0.05, -0.06, -0.06, 0.,
47            0., 0., 0.06, 0.06, 0.06]) * direction
48        y = np.abs(v) * s * np.array([0., 0., 0.0405, 0.0405, 0.0405,
49            0.0405, 0.0405, 0.0405, 0.0405, 0., 0.])
50        swing_x = 0.
51        swing_y = 0.
52        swing_z = 0.
53        # Use all 12 points
54        for l in range(10):
55            swing_x = swing_x + self.bezier_curve(phi_sw, l, x[l])
56            swing_y = swing_y + self.bezier_curve(phi_sw, l, y[l])
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114

```



```

Activities Text Editor * April 27 6:17 PM • gait_planner2.py -/jacobson3/ens/rex/obj/rex3/rls/packages/rex_gym/mode
Open [ ] Save [ ] Python Tab Width: 8 ▾ Ln 1, Col 1 ▾ INS
1 def step_trajectory(self, phi, v, angle, w_rot, center_to_foot, direction):
2     if phi >= 1:
3         r = np.sqrt(center_to_foot[0]**2 + center_to_foot[1]**2)
4         foot_angle = np.arctan2(center_to_foot[1], center_to_foot[0])
5         if w_rot < 0:
6             circle_trajectory = 90. - np.rad2deg(foot_angle - self._alpha)
7         else:
8             circle_trajectory = 270. - np.rad2deg(foot_angle + self._alpha)
9
10    if phi <= self.step_offset:
11        # stance phase
12        phi = phi / self.step_offset
13        stepx_long, stepy_long, stepz_long = self.calculate_stance(phi_stance, v, angle)
14        stepx_rot, stepy_rot, stepz_rot = self.calculate_stance(phi_stance, w_rot, circle_trajectory)
15
16        # swing phase
17        phiSwing = (phi - self.step_offset) / (1 - self.step_offset)
18        stepx_long, stepy_long, stepz_long = self.calculate_bezier_swing(phiSwing, v, angle, direction)
19        stepx_rot, stepy_rot, stepz_rot = self.calculate_bezier_swing(phiSwing, w_rot, circle_trajectory, direction)
20
21        if stepx_rot == 0:
22            self._alpha = np.arctan2(np.sqrt(stepx_rot**2 + stepy_rot**2), r)
23        else:
24            self._alpha = np.arctan2(np.sqrt(stepx_rot**2 + stepy_rot**2), r)
25
26        if stepx_rot != 0:
27            self._alpha = np.arctan2(np.sqrt(stepx_rot**2 + stepy_rot**2), r)
28
29        else:
30            self._alpha = np.arctan2(np.sqrt(stepx_rot**2 + stepy_rot**2), r)
31
32        coord = np.empty()
33        coord[0] = stepx_long + stepx_rot
34        coord[1] = stepy_long + stepy_rot
35        coord[2] = stepz_long + stepz_rot
36
37        return coord
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
59
60
61
62
63
64
65
66
67
68
69
69
70
71
72
73
74
75
76
77
78
79
79
80
81
82
83
84
85
86
87
88
89
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
109
110
111
112
113
114

```

Kinematics:

```
4 class Kinematics:
5     def __init__(self):
6         self._l = 0.15
7         self._w = 0.075
8         self._hip = 0.055
9         self._foot = 0.0652
10        self._foot_w = 0.145
11        self._y_dist = 0.185
12        self._x_dist = self._l
13        self._height = 0.2
14        # frame vectors
15        self._hip_front_right_v = np.array([self._l / 2, -self._w / 2, 0])
16        self._hip_front_left_v = np.array([self._l / 2, self._w / 2, 0])
17        self._hip_rear_right_v = np.array([-self._l / 2, -self._w / 2, 0])
18        self._hip_rear_left_v = np.array([-self._l / 2, self._w / 2, 0])
19        self._foot_front_right_v = np.array([self._x_dist / 2, -self._y_dist / 2, -self._height])
20        self._foot_front_left_v = np.array([self._x_dist / 2, self._y_dist / 2, -self._height])
21        self._foot_rear_right_v = np.array([-self._x_dist / 2, -self._y_dist / 2, -self._height])
22        self._foot_rear_left_v = np.array([-self._x_dist / 2, self._y_dist / 2, -self._height])
23        self._frames = np.asmatrix([[self._x_dist / 2, -self._y_dist / 2, -self._height],
24                                   [0, 0, 1],
25                                   [-self._x_dist / 2, -self._y_dist / 2, -self._height],
26                                   [-self._x_dist / 2, self._y_dist / 2, -self._height]])
27
28    @staticmethod
29    def get_Bx(x):
30        return np.asmatrix([[1, 0, 0, 0],
31                            [0, np.cos(x), -np.sin(x), 0],
32                            [0, np.sin(x), np.cos(x), 0],
33                            [0, 0, 0, 1]])
34
35    @staticmethod
36    def get_By(y):
37        return np.asmatrix([[np.cos(y), 0, np.sin(y), 0],
38                            [0, 1, 0, 0],
39                            [0, np.sin(y), 0, np.cos(y), 0],
40                            [0, 0, 0, 1]])
41
42    @staticmethod
43    def get_Bz(z):
44        return np.asmatrix([[np.cos(z), -np.sin(z), 0, 0],
45                            [np.sin(z), np.cos(z), 0, 0],
46                            [0, 0, 1, 0],
47                            [0, 0, 0, -1]])
48
49    def get_Rxyz(self, x, y, z):
50        if x <= 0 or y <= 0 or z <= 0:
51            R = self.get_Bx(x) * self.get_By(y) * self.get_Bz(z)
52        else:
53            R = self.get_By(y) * self.get_Bz(z) * self.get_Bx(x)
54        return R
55
56    def get_E(self, orientation, position):
57        roll = orientation[0]
58        pitch = orientation[1]
```

Policy Player:



The screenshot shows a Jupyter Notebook interface with a Python script titled "policy_player.py". The code imports necessary libraries and defines a "PolicyLayer" class. It then initializes the environment ID, signal type, and policy path. The script uses a flag mapper to get the policy ID and creates policy and value layers based on the configuration. It then runs a loop to interact with the environment, taking actions, observing rewards, and updating the checkpoint.

```
1 """Running a pre-trained ppo agent on rex environments"""
2 import logging
3 import os
4 import site
5 import time
6
7 import tensorflow.compat.v1 as tf
8 from rex_gym.agents import utility
9 from rex_gym.agents.ppo import simple_ppo_agent
10 from rex_gym.util import flag_mapper
11
12
13 class PolicyLayer:
14     def __init__(self, env_id: str, args: dict, signal_type: str):
15         self.gym_dlr_path = str(site.getsitepackages()[-1])
16         self.env_id = env_id
17         self.args = args
18         self.signal_type = signal_type
19         self.args[debug] = True
20
21     def play(self):
22         if self.signal_type:
23             self.args['signal_type'] = self.signal_type
24         else:
25             self.signal_type = flag_mapper.DEFAULT_SIGNAL[self.env_id]
26         policy_id = f'{self.env_id}_{self.signal_type}'
27         policy_path = flag_mapper.ENV_ID_TO_POLICY[policy_id][0]
28         policy_dlr = os.path.join(flag_mapper.gym_dlr_path, policy_path)
29         config_utility.load_config(policy_dlr)
30         policy_layers = config.policy.layers
31         value_layers = config.value.layers
32         env = config.env(render=True, **self.args)
33         env.seed(0)
34         checkpoint = os.path.join(policy_dlr, flag_mapper.ENV_ID_TO_POLICY[policy_id][1])
35         with tf.Session() as sess:
36             agent = simple_ppo_agent.SimplePPOPolicy(sess,
37                                                     env,
38                                                     network,
39                                                     policy_layers=policy_layers,
40                                                     value_layers=value_layers,
41                                                     checkpoint=checkpoint)
42
43         sum_reward = 0
44         observation = env.reset()
45         while True:
46             action = agent.get_action([observation])
47             observation, reward, done, _ = env.step(action[0])
48             time.sleep(0.002)
49             sum_reward += reward
50             logging.info(f'Rewards:{sum_reward}')
51             if done:
52                 break
```

Trainer:

```

Activities Text Editor Apr 27 6:18 PM •
trainer.py -/anaconda/envs/rei/lib/python3.7/site-packages/ex_gym/playground
Save □ ×

1 """Script to render a training session."""
2 import datetime
3 import functools
4 import logging
5 import os
6 import platform
7
8 import gym
9 import tensorflow.compat.v1 as tf
10
11 from rex_gym.agents.tools import wrappers
12 from rex_gym.agents.scripts import configs, utility
13 from rex_gym.agents.tools.attr_dict import AttrDict
14 from rex_gym.agents.tools.loop import Loop
15 from rex_gym.util import FlagMapper
16
17
18 class Trainer:
19     def __init__(self, env_id: str, args: dict, playground: bool, log_dir: str, agents_number, signal_type):
20         self.args = args
21         self.playground = playground
22         self.log_dir = log_dir
23         self.agents_number = agents_number
24         self.signal_type = signal_type
25         if self.signal_type:
26             env_signal = self.signal_type
27         else:
28             env_signal = FlagMapper.DEFAULT_SIGNAL[env_id]
29         self.env_id = f'{env_id}_{env_signal}'
30
31     def create_environment(self, config):
32         """Constructor for an instance of the environment.
33
34         Args:
35             config: Object providing configurations via attributes.
36
37         Returns:
38             Returns:
39             ... Wrapped OpenAI Gym environment.
40
41         If self.playground:
42             self.args['render'] = True
43             self.args['debug'] = True
44         else:
45             self.args['debug'] = False
46             if self.signal_type:
47                 self.args['signal_type'] = self.signal_type
48                 env = gym.make(config.env, **self.args)
49             if config.max_length:
50                 env = wrappers.ReweightNormalizer(env, config.max_length)
51             env = wrappers.ClipAction(env)
52             env = wrappers.ConvertTo32Bit(env)
53
54         return env
55
56     @staticmethod

```

Python Tab Width: 8 Ln 1, Col 1 INS

```

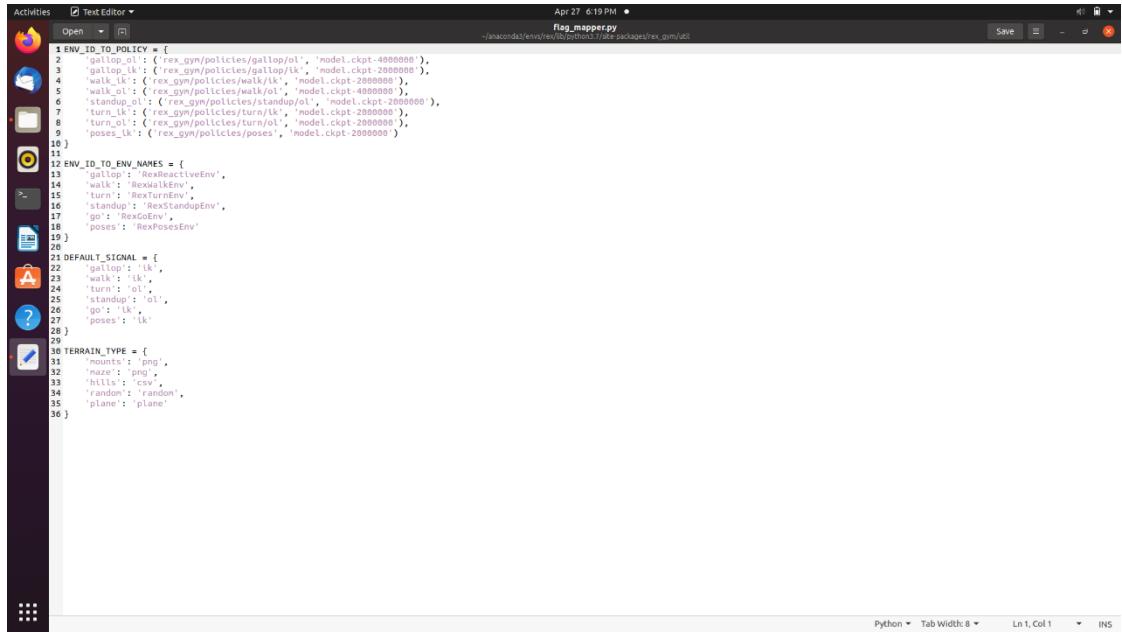
Activities Text Editor Apr 27 6:18 PM •
trainer.py -/anaconda/envs/rei/lib/python3.7/site-packages/ex_gym/playground
Save □ ×

55 @staticmethod
56     def _define_loop(graph, logdir, train_steps, eval_steps):
57         """Create and configure a training loop with training and evaluation phases.
58
59         Args:
60             graph: Object providing graph elements via attributes.
61             logdir: Log directory for storing checkpoints and summaries.
62             train_steps: Number of training steps per epoch.
63             eval_steps: Number of evaluation steps per epoch.
64
65         Returns:
66             ... Loop object.
67
68         loop = Loop(logdir, graph.step, graph.should_log, graph.do_report, graph.force_reset)
69         loop.add_phase('train',
70             graph.done,
71             graph.score,
72             graph.summary,
73             train_steps,
74             report_every=1000,
75             log_every_train_steps // 2,
76             checkpoint_every=1000,
77             feed=(graph.ts.training: True))
78         loop.add_phase('eval',
79             graph.done,
80             graph.score,
81             graph.summary,
82             eval_steps,
83             report_every=eval_steps,
84             log_every_eval_steps // 2,
85             checkpoint_every=10 * eval_steps,
86             feed=(graph.ts.training: False))
87
88     return loop
89
90     def train(self, config, env_processes):
91         """Training and evaluation entry point yielding scores.
92
93         Resolves some configuration attributes, creates environments, graph, and
94         training loop. By default, assigns all operations to the CPU.
95
96         Args:
97             config: Object providing configurations via attributes.
98             env_processes: Whether to step environments in separate processes.
99
100        Yields:
101            ... Evaluation scores.
102
103        tf.reset_default_graph()
104        with config.unlocked():
105            config.network = mcmcools.partial(utility.define_network, config.network, config)
106            config.policy_optimizer = getattr(tf.train, config.policy_optimizer)
107            config.value_optimizer = getattr(tf.train, config.value_optimizer)
108            tf.config.update_every = config.num_agents
109            logging.warning(f'Number of agents should divide episodes per update.')
110            with tf.device('/cpu:0'):

```

Python Tab Width: 8 Ln 1, Col 1 INS

Flag Mapper:

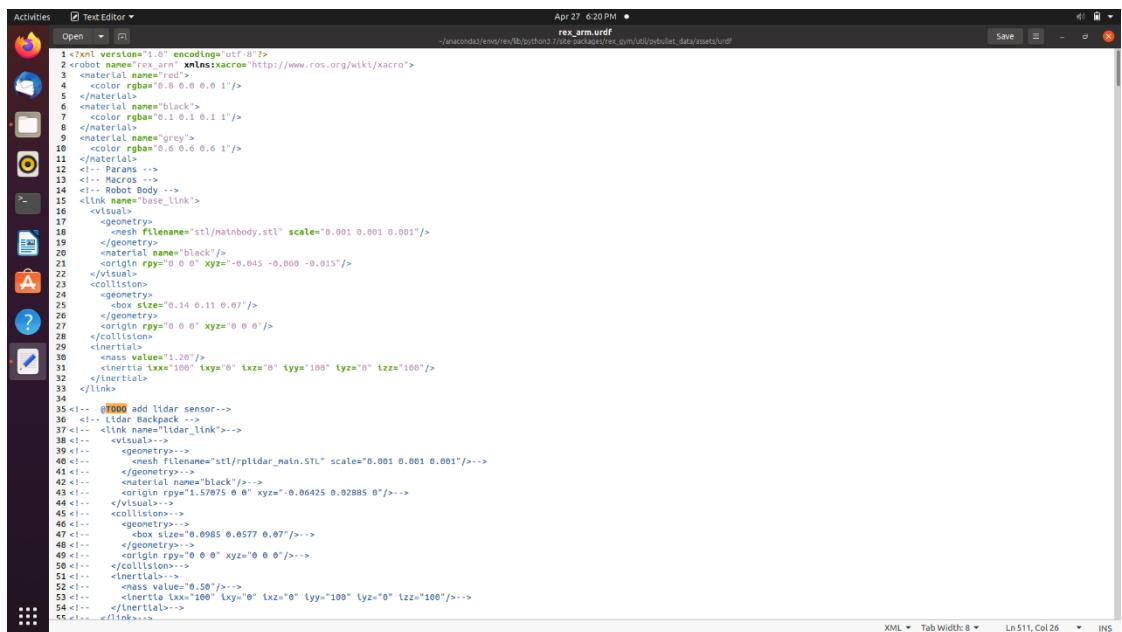


```

1 ENV_ID_TO_POLICY = {
2     'gallop_ol': ('rex_gym/policies/gallop/ol', 'modelckpt-4000000'),
3     'gallop_llk': ('rex_gym/policies/gallop/llk', 'modelckpt-2000000'),
4     'walk_ol': ('rex_gym/policies/walk/ol', 'modelckpt-4000000'),
5     'walk_llk': ('rex_gym/policies/walk/llk', 'modelckpt-2000000'),
6     'standup_ol': ('rex_gym/policies/standup/ol', 'modelckpt-2000000'),
7     'turn_llk': ('rex_gym/policies/turn/llk', 'modelckpt-2000000'),
8     'turn_ol': ('rex_gym/policies/turn/ol', 'modelckpt-2000000'),
9     'poses_llk': ('rex_gym/policies/poses', 'modelckpt-2000000')
10 }
11
12 ENV_ID_TO_ENV_NAMES = {
13     'gallop': 'RexReactiveEnv',
14     'walk': 'RexWalkEnv',
15     'turn': 'RexTurnEnv',
16     'standup': 'RexStandupEnv',
17     'go': 'RexGoEnv',
18     'poses': 'RexPosesEnv'
19 }
20
21 DEFAULT_SIGNAL = {
22     'gallop': 'llk',
23     'walk': 'llk',
24     'turn': 'llk',
25     'standup': 'ol',
26     'go': 'llk',
27     'poses': 'llk'
28 }
29
30 TERRAIN_TYPE = {
31     'mountain': 'png',
32     'maze': 'png',
33     'hills': 'csv',
34     'random': 'random',
35     'plane': 'plane'
36 }

```

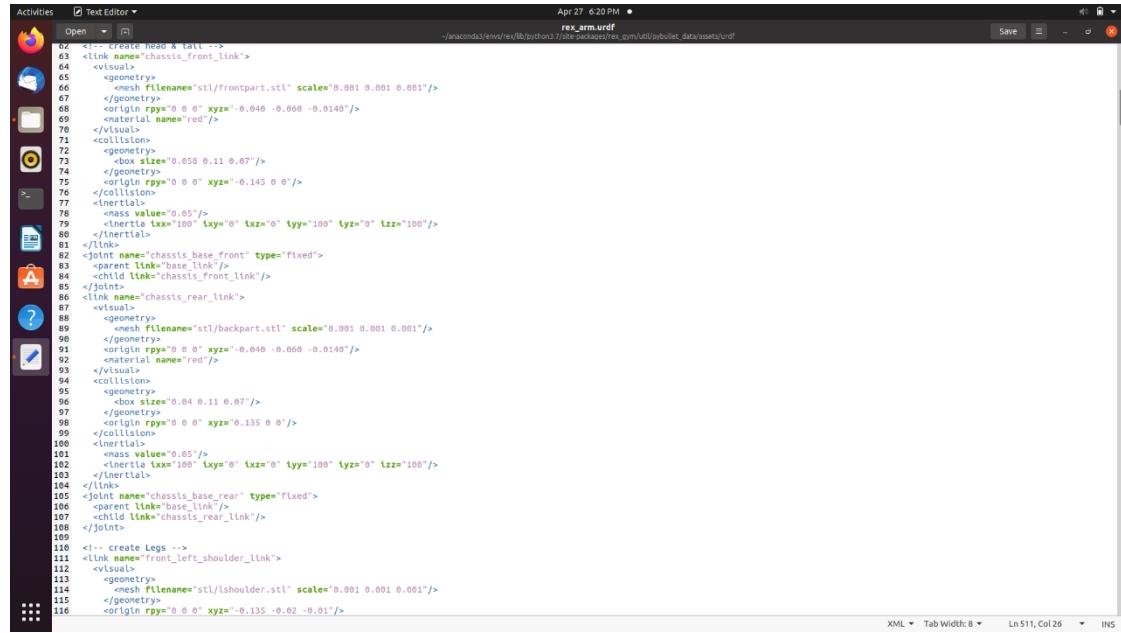
Rex Arm:



```

1 <?xml version="1.0" encoding="utf-8"?>
2 <robot name="rex_arm" xmlns:xacro="http://www.ros.org/wiki/xacro">
3   <material name="red">
4     <color rgba="0.8 0.0 0.6 1"/>
5   </material>
6   <material name="black">
7     <color rgba="0.1 0.1 0.1 1"/>
8   </material>
9   <material name="grey">
10    <color rgba="0.6 0.6 0.6 1"/>
11  </material>
12  <!-- Params -->
13  <!-- Macros -->
14  <!-- Base -->
15  <link name="base_link">
16    <visual>
17      <geometry>
18        <mesh filenam="stl/mainbody.stl" scale="0.001 0.001 0.001"/>
19      </geometry>
20      <material name="black"/>
21      <origin rpy="0 0 0" xyz="-0.045 -0.000 -0.015"/>
22    </visual>
23    <collision>
24      <geometry>
25        <mesh size="0.14 0.11 0.07"/>
26      </geometry>
27      <origin rpy="0 0 0" xyz="0 0 0"/>
28    </collision>
29    <inertial>
30      <mass value="1.20"/>
31      <inertia ixz="100" ixy="0" iyz="0" izx="100" iyy="100" izz="100"/>
32    </inertial>
33  </link>
34
35 <!-- @TODO add lidar sensor-->
36  <!-- Lidar Backpack -->
37  <!-- <link name="lidar_link">-->
38  <!-- <visual>-->
39  <!-- <geometry>-->
40  <!-- <mesh filenam="stl/rplidar_main.STL" scale="0.001 0.001 0.001"/>-->
41  <!-- <geometry>-->
42  <!-- <material name="black"/>-->
43  <!-- <origin rpy="1.57075 0 0" xyz="-0.06425 0.02885 0"/>-->
44  <!-- <collision>-->
45  <!-- <geometry>-->
46  <!-- <mesh size="0.0985 0.0577 0.07"/>-->
47  <!-- <origin rpy="0 0 0" xyz="0 0 0"/>-->
48  <!-- <collision>-->
49  <!-- <origin rpy="0 0 0" xyz="0 0 0"/>-->
50  <!-- <collision>-->
51  <!-- <inertial>-->
52  <!-- <inertia ixz="100" ixy="0" iyz="0" izx="100" iyy="100" izz="100"/>-->
53  <!-- <inertia ixz="100" ixy="0" iyz="0" izx="0" iyy="100" izz="100"/>-->
54 </inertial>-->
55 </link>-->

```



The screenshot shows a terminal window with a text editor displaying URDF (Unified Robot Description Format) code. The file is named `rx_arms.urdf`. The code defines a robot arm with various links, joints, and collision geometry. Key parts of the code include:

- Links: `chassis_front_link`, `base_front`, `chassis_rear_link`, `chassis_base_rear`.
- Joints: `base_link`, `chassis_front_link`, `chassis_rear_link`.
- Collision Geometry: `base_backpart.stl`, `lshoulder.stl`.
- Visuals: `frontpart.stl`, `lshoulder.stl`.
- Materials: `red`.

```
Apr 27 6:20PM • rx_arms.urdf
-/anaconda3/envs/ex/lib/python3.7/site-packages/ex_gym/drl/pybullet_data/assets/urdf

62 <!-- create head & tail -->
63 <link name="chassis_front_link">
64   <visual>
65     <geometry>
66       <mesh filenamex="stl/frontpart.stl" scale="0.001 0.001 0.001"/>
67       <origin rpy="0 0 0" xyz="-0.040 -0.060 -0.0140"/>
68       <material name="red"/>
69   </visual>
70   <collision>
71     <geometry>
72       <box size="0.058 0.11 0.07"/>
73     </geometry>
74     <origin rpy="0 0 0" xyz="-0.145 0 0"/>
75   </collision>
76   <inertial>
77     <mass value="0.05"/>
78     <center_of_mass ixz="100" ixy="0" ixz="0" iyy="100" izx="0" iyz="100"/>
79     <inertia ixx="100" iyy="100" izz="100"/>
80   </inertial>
81 </link>
82 <link name="base_front" type="fixed">
83   <parent link="base_link"/>
84   <child link="chassis_front_link"/>
85 </joint>
86 <link name="chassis_rear_link">
87   <visual>
88     <geometry>
89       <mesh filenamex="stl/backpart.stl" scale="0.001 0.001 0.001"/>
90       <origin rpy="0 0 0" xyz="-0.040 -0.060 -0.0140"/>
91       <material name="red"/>
92   </visual>
93   <collision>
94     <geometry>
95       <box size="0.04 0.11 0.07"/>
96     </geometry>
97     <origin rpy="0 0 0" xyz="0.135 0 0"/>
98   </collision>
99   <inertial>
100    <mass value="0.05"/>
101   <center_of_mass ixz="100" ixy="0" ixz="0" iyy="100" izx="0" iyz="100"/>
102   <inertia ixx="100" iyy="100" izz="100"/>
103 </inertial>
104 </link>
105 <link name="chassis_base_rear" type="fixed">
106   <parent link="base_link"/>
107   <child link="chassis_rear_link"/>
108 </joint>
109 <!-- create Legs -->
110 <link name="front_left_shoulder_link">
111   <visual>
112     <geometry>
113       <mesh filenamex="stl/lshoulder.stl" scale="0.001 0.001 0.001"/>
114     </geometry>
115     <origin rpy="0 0 0" xyz="-0.135 -0.02 -0.01"/>
```