TocabiLab: A Mass Parallel Capable Torque Controlled Robot Training Framework

by

ZHIXIN FANG

B. E., ShanghaiTech University, 2021

A Project Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

TocabiLab: A Mass Parallel Capable Torque Controlled Robot Training Framework

by

ZHIXIN FANG

B. E., ShanghaiTech University, 2021

Supervisory Committee

Dr. Brandon Haworth, Supervisor

(Department of Computer Science)

Dr. Teseo Schneider, Departmental Member

(Department of Computer Science)

ABSTRACT

Robotic deep RL tasks are intrinsically hard to parallelize, as they usually require cooperation between the RL agent, the simulator and the physics engine. Context barriers can be hard to cross between those systems, making device-host transfer costs during training a challenge. Furthermore, the rapidly evolving world of machine learning also makes integration increasingly difficult for researchers, as incorporating a new learning algorithm usually requires huge engineering effort. TocabiLab is developed as an extension to NVIDIA Isaac Lab with the capability of conducting GPU-native robotics deep RL tasks. Its unified API inherited from Isaac Lab reduces the difficulty of integration, as well as enabling data to be swiftly passed between components at a large scale, accelerating training under computation intensive scenarios. This project report describes the novel architecture applied in the proposed system and a set of experiments designed upon a previously published research to validate the system's correctness, efficiency and versatility. Evaluation result shows that the system is capable of handling more than 300% load than the industry standard systems while providing comparable performance.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

All members of the GAIDG Lab, for the emotional and academic support I received.

Dr. Brandon Haworth, for connecting me with external resources and the wonderful world of RL.

Donghyeon Kim and Dr. Mathew Schwartz, for providing reference materials and technical support.

While I do not believe in conservatism ideologies, I constantly feel pressured from the privilege I received from my family, my country, my friends and supporters. Nobility as a class is no more, yet privileged people are still entitled. Nobility extends beyond mere entitlement, and we must do more than what we received for the people we love. In a time when nobility is honorary but privileges persist, one can still choose to be noble.

Zhixin 'Shin' Fang

DEDICATION

I dedicate this project to my mentor, Mr. Brian Cox, who has been supporting me for over 7 years.

To my parents, who are always supporting me and encouraging me to find and fulfill my duty.

To Matt Salsamendi and his team, who took me on an amazing journey to create All Out Games and helped to build my career.

To my amazing peers like Ali Ahangarpour, Steven Bobyn and David Kim, thank you for coming through these two years of spirited struggle with me.

Chapter 1

Introduction

In recent years, the field of robotics has seen significant advancements, particularly in the area of reinforcement learning (RL)[26, 23]. However, the training of RL models for robotics applications remains a computationally intensive task, often requiring substantial time and resources. To address these challenges, this thesis introduces "TocabiLab", a novel framework designed to enable efficient and parallelized RL training for robotics tasks on GPUs. By leveraging the parallel processing capabilities of GPUs, The proposed system significantly increased the scalability of robotics training environments, making it feasible to develop more sophisticated and capable robotic systems.

One of the key features of the proposed system is its versatility in switching between different RL libraries, such as stable baselines, rsl_rl, and rl_games, with ease. This flexibility allows researchers and developers to experiment with various RL algorithms and select the most suitable one for their specific application. Additionally, The proposed system supports a wide range of control models, including torque control, proportional-derivative (PD) control, and model-based controllers. This adaptability ensures that the framework can be applied to a diverse set of robotic tasks,

from simple manipulations to complex, dynamic interactions.

Furthermore, The proposed system is designed with user-friendliness in mind, providing an intuitive interface that simplifies the process of setting up and managing RL experiments. By streamlining the workflow, The proposed system reduces the barrier to entry for researchers and developers, enabling them to focus on innovation and experimentation rather than the intricacies of RL training. Ultimately, this project report aims to demonstrate how the system can be used for more efficient, versatile, and powerful robotics applications.

The system is designed as an extension of NVIDIA's Isaac Lab[17], which allowed it to utilize the high-fidelity rendering pipeline provided by the Omniverse toolkit. More importantly, GPU native support enabled it to communicate directly between the RL library and the physics engine, eliminating the context barrier so that the RL algorithms can directly fetch the robots' states with no device-host transfers. A novel GPU-driven architecture is presented in Chapter 3, which accelerates training and makes the environments more scalable.

We also reproduced a published research [12] in the proposed system as the starting task for future users and showed the correctness and versatility by 3 experiments in Chapter 4. We also achieved a significant performance boost from using GPU-native architecture, as will be demonstrated in Chapter 5.

1.1 Contribution

The proposed system aims to solve the versatility and training speed issues that are frequently found in robotic deep RL research projects. **Versatility** and **efficiency** will be the two keywords across this paper.

We define **versatility** in Robotic RL systems as the ability to rapidly iterate and

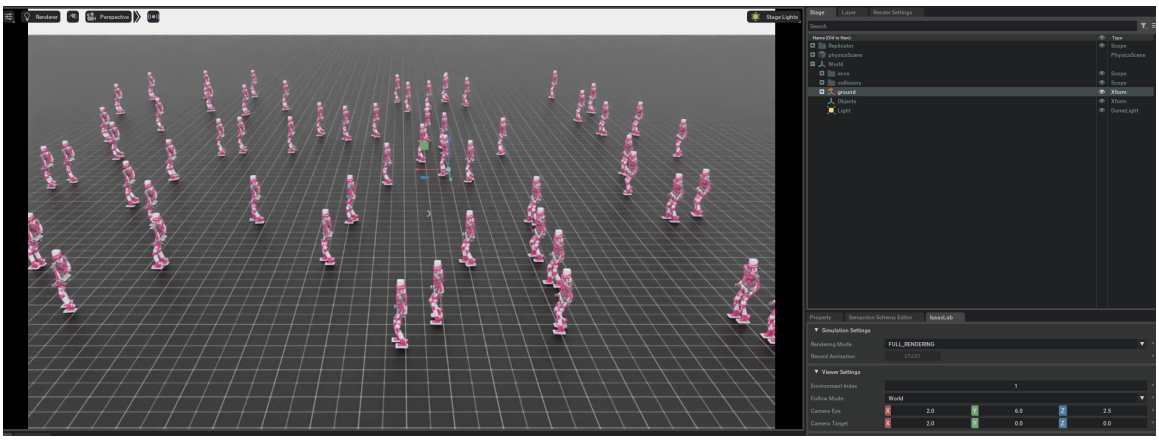

Figure 1.1: Training 64 TOCABI in real time on a regular PC (RTX Enabled at 30FPS)

integrate new RL libraries or methods. Novel policy optimization methods like Proximal Policy Optimization (PPO)[29], architectural improvements such as DD-PPO[38] and more optimized APIs like rsl_rl[13] are constantly evolving. It can be difficult to keep track of new advancements as robotic RL tasks are frequently conducted on an entire suite of software including simulator, controller and RL libraries. Researchers are usually challenged by extensively time-consuming engineering tasks when they want to try out the new methods, as there is no guarantee that the new methods or tools will fit the existing API. The proposed system extends the unified API provided by Isaac Lab [17] to fit torque controlled robots, enabling tasks using the torque-based action space to be transferred between different supported RL toolboxes with little engineering effort.

Simulated robots ultimately serve the purpose of mirroring their real world counterparts, thus a physics engine is usually used to accept control input and predict what would happen in the real world. When GPU parallelization and reinforcement learning are introduced to this scenario, we naturally encounter a context barrier: communication between the physics engine and the RL policy. Fetching the status of a robot from physics engine and feeding it to the RL policy is a necessity. Conventionally, it has to be done by transporting the data to the CPU first and distributing it to the another end. This introduces excessive device-host transfer and impacts the **efficiency** of the training greatly. The proposed system utilized the communication pathway between the PhysX engine and Issac Lab, enabling zero-transfer communication between reinforcement learning and the physics engine. It also removed the need of copying multiple scenes by effectively use GPU instancing. This improves the scalability and speed of the system by a notable margin.

In the following chapters, we will introduce the architecture of the system and the experiment we performed to prove the correctness and versatility of the system. A

scalability test will also be conducted between the proposed system and the reference system, comparing their efficiency.

1.2 Structure of the Report

The rest of the project report is organized as follows:

Chapter 2 introduces existing ideas in the crossing field of Reinforcement Learning, Robotics and GPU Programming.

Chapter 3 explains the novel architecture design we introduced for the proposed system, and compares it against the traditional architecture found in the reference system[12]. Critical physics model comparison is also included in this chapter for future users' reference.

Chapter 4 conducts 3 experiments, aiming to reproduce results from the reference system, showing the correctness and robustness of the proposed system.

Chapter 5 compares the scalability and efficiency of different systems.

Chapter 6 concludes the project and discusses its limitation and future research potential.

Chapter 2

Related Work

In recent years, Graphics Processing Units (GPUs) are becoming more and more important in our hardware architectures, particularly due to the growing need of computational power in the topics related to machine learning. Reinforcement learning on legged robots is one such topic. Reinforcement learning provides an intuitive way to model the objective feedback of the control strategy[32], and is therefore widely used in robotics. Compared to wheel-supported robots, legged robots are fairly complex and typically need to be trained for longer than to accomplish a task. Apart from the need of training, physics simulation also requires huge GPU resources. The computation-intensive nature of RL tasks proposed researchers in the robotics field to utilize their GPU more efficiently.

This chapter looks at closely related work that have utilized RL and physics simulator for robotic tasks, as well as papers of the platforms we used to construct our mass-parallel capable robotic RL framework.

2.1 RL on Legged Robots

Legged robots face many challenges in real world applications, such as locomotion[8], dynamic terrain traversal[25] and body coordination[3]. RL-based methods have shown superiority compared to model-based ones, as such methods usually require less domain knowledge and exploit full body dynamics better[14].

One particular challenge lies in the locomotion process of legged robots. By discretely making contact with the ground, legged robots are naturally more capable of trajectory correction, but this makes controlling them more challenging as well, since their trajectory is unpredictable during their locomotion. Common ways to address this challenge is to employ a phase model [39] or design a footstep planner [3]. In legged robots RL problems, phase modulation is more commonly used, as phase can be easily modeled as $\phi \in [0, 1]$ and adapted into the RL agent's observation space[12, 25, 26].

GenLoco [8] designed generalized locomotion controllers that can be applied on quadrupeds by feeding data from the training of a diverse set of procedurally generated robots. It shows that by training locomotion tasks on a very diverse set of legged robots, the RL agent eventually learns to generalize and obtained a control strategy that can be directly transferred to novel simulated and real-world robots with diverse morphologies. This work also hints the need of an efficient parallelization approach, as intensive training and huge data sets are needed to obtain generalizable results.

Controlling the gait of the legged robots is another challenge in their locomotion tasks. Different from a model-based controller, it is difficult to control the gait of a walking robot when it is controlled by an RL-based agent. Peng et al.'s DeepMimic [23] provides a novel way to solve this problem. By adding rewards for mimicking the joint position, velocity and end effector status from the reference motion, the legged robots are actively encouraged to follow the predetermined gait of given motions.

Regulation rewards are also modeled to encourage the RL agent to use less extreme movements, adding reliability to Sim-to-Real scenarios. With the growing popularity of this method and RL in general, our framework must have support for importing captured motions and utilizing DeepMimic-like methods.

Stable-Baselines3 (SB3)[28] was developed to address the common reproducibility issues among reinforcement learning works. One of the RL algorithms frequently used with it, Policy Proximal Optimization(PPO)[29] is commonly employed in robotic research, as PPO enables multiple epochs to perform minibatch updates, which will allow the policy to prioritize long-term outcomes of the policy. PPO generally works very well with tasks like robotic locomotion and game playing and has been then widely used in similar RL tasks[9, 12, 40], and have shown to be superior to other methods like Deep-Q Networks(DQN)[33] and Asynchronous Advantage Actor-Critic(A3C)[4] in terms of performance and stability. PPO is also shown to be compatible with distributed reinforcement learning, as Erik Wijmans et al. [38] demonstrated DD-PPO's capability of mass parallelized training of 2.5 billion agents.

In conclusion, RL on legged robots is a very active subject that requires lots of data and training time, various methods are developed and tested on a range of different tasks and different robots. The proposed system is designed to be modular enough that it handles large scale training on regular hardware efficiently, and is capable of rapid prototyping and testing of an array of actuator/task/RL library combinations. In Chapter 3, we will discuss how the proposed system achieved this modularity.

2.2 Torque Controlled Robot

As this project aims to create a versatile, GPU-native framework for legged robot reinforcement learning tasks, it is essential to select a task with an appropriate problem

space and complexity as a starting point. Peng et al.[26] discussed multiple action spaces that can be used in legged robot RL locomotion tasks. This work trained multiple articulated figures on various reference actions for different gaits and compared four different action spaces, namely target joint angles, target joint velocities, torque and muscle activation.

While torque, muscle activation, target position and target velocity are all valid action spaces, outputting torque as actions of an RL agent is considered to be decently stable and requires least amount of domain knowledge. Although carefully tuned PD controllers can easily outperform torque controllers in the learning process, torque controller is easier to setup and still have comparable performance. We thus determined it is the appropriate starter actuator model for the framework.

The Torque-controlled Compliant Biped (TOCABI) [30] was developed for research and application of torque-based control humanoid robots. It has 50 degrees of freedom, which we consider to be fairly complex compared to MuJoCo's built-in humanoid (29 DoF)[35]. Despite being specifically designed for torque-based control, TOCABI is capable of handling model-based or RL-based PD (Proportional Derivative) Controller as well. PD control is the industry standard for articulated robots [10, 36], and it's frequently compared to torque control. Although we eventually decided to start with torque-control, the capability of implementing PD control in the framework is preserved.

Kim et al.'s *Torque-based Deep Reinforcement Learning for Task-and-Robot Agnostic Learning on Bipedal Robots Using Sim-to-Real Transfer*[12] used TOCABI in various tasks including standing, squatting, walking and running and proved torque controllers trained with PPO and DeepMimic reference action are stable enough for Sim-to-Real transfer. The paper also provided training performance records, reference solution and all their reference data captured from DeepMimic. We therefore choose

the tasks from this work as our reference and starting point. Our framework's objective is set to reliably reproduce results from the reference project, while achieving better visualization, higher performance and higher framework versatility. In Chapter 4's experiments, we will review this work and go over the key difference between the reproduced task and its original counterpart to demonstrate the proposed system's advantages.

2.3 Physics Engine and Simulator in Robotics

Physics engines and simulators have long been used in robotic deep RL research, as it serves as a necessary verification stage before deploying the control strategy to costly real robots. In simulated domains, agents have been developed to perform a diverse array of challenging tasks in simulated environments supported by physics engine[18, 15, 5], and approaches to directly transfer control to their real counterparts (known as Sim-to-Real) are also developed.[24, 41]

Multi-Joint dynamics with Contact (MuJoCo) [35] is a physics engine designed specifically for articulated robots. It offers powerful contact report API and is still actively maintained by Google, leading to its frequent presence in robotics RL research. However, being a system developed 14 years ago, its graphics API is outdated and does not utilize the GPU very well. It performed poorly when running mass paralleled tasks when the parallelization exceeds the capacity of the CPU, as our experiments will later demonstrate.

Erez et al. [6] published a report comparing a few frequently used physics engine for model-based robotics, namely Bullet, MuJoCo, Havok, ODE and PhysX. We are particularly interested in the comparison between MuJoCo and PhysX, as most recent robotics RL research are implemented in MuJoCo, while PhysX is the default physics

engine in Nvidia's Omniverse toolchain. Erez's work deemed PhysX unsuitable for robotics due to the fact that it ignores the Coriolis force. However, this work was published before PhysX 5's release. This version of PhysX added support for Coriolis force on all rigid bodies. It is also worth noting that MuJoCo performed best when less disconnected rigidbodies are present, which is a good trait since many tasks involves one robot and one terrain. However, in some cases, we might want more than one robots to appear in the scene, MuJoCo's performance rapidly degrades when more and more bodies participate in collision resolution, while PhsyX is able to maintain a relatively consistent performance.

Isaac Sim [19] is Nvidia's dedicated simulation tool for robotics applications. It provides a specialized set of APIs built upon PhysX[21] for robotics. This project aims to create a framework using it and its capability of going entirely GPU native. It is capable of transferring data directly from the RL environment to the physics engine, thus eliminating device-host transfer. Isaac Lab (previously named Orbit)[17] is an Isaac Sim extension that provides unified API for reinforcement learning purposes. This framework simplified the process of task creation and RL policy customization and can also provide great performance and rendering quality, and is thus selected as the basis of this work. Rendering quality wise, Isaac Sim uses the Omniverse RTX renderer, which provides efficient real-time ray-traced visualization capabilities.

Projected Gauss-Seidel Solver (PGS)[2] is a widely used numerical solver in rigid-body simulation scenarios[37, 31, 7]. While PhysX provides a more efficient method called Temporal Gauss-Seidel, as clear from the previous section, the proposed project needs a reproduction of a previous work [12] done in MuJoCo, we've decided to use the PGS solver as it's supported in both MuJoCo and PhysX.[35]

2.4 GPU Programming in RL

The efficiency of machine learning algorithms in parallelization is a pivotal factor in their rapid advancement and widespread application. This efficiency stems from the inherent nature of many machine learning tasks, which can be decomposed into smaller, independent operations. For instance, matrix multiplications, which are fundamental to deep learning, can be parallelized effectively, as each element of the resulting matrix can be computed independently. In many scenarios including robotic RL, data parallelism allows for the simultaneous processing of data batches during training, significantly accelerating the learning process. Model parallelism further enhances efficiency by distributing large models across multiple GPUs, enabling the handling of complex distributed architectures on multiple GPUs[38]. Moreover, tasks such as hyperparameter tuning and ensemble methods, which involve running multiple independent experiments, are inherently parallelizable. As a result, many high level GPU programming frameworks such as Caffe[11], Tensorflow[1] and PyTorch[22] emerged, providing accessible and productive ways for researchers to use their GPUs.

PyTorch[22] was created to address the challenge of parallelizing tasks in Python. Due to the Global Interpreter Lock (GIL) that existed until Python 3.12 preventing a Python process to use multiple threads, it is notoriously difficult to parallelize a Python environment. PyTorch bypassed the GIL by writing a C++ based core library that allows operations to be dispatched from Python within minimal overhead. Stable-Baselines3 [28] utilized PyTorch extensively to handle the matrix operations, making it an ideal choice for researchers using MuJoCo with its Python bindings. However, as mentioned in documentation[27], Stable-Baselines3's implementation of PPO is intended to be run in CPU mode, although it uses GPU for matrix operations. More specifically, Stable-Baselines3's PPO runs its core loops in Python, meaning PyTorch is only used to handle large matrix operations such as policy updates, while the

buffer collected from the learning environment still resides in host memory. Running this version of PPO will require data transportation to the CUDA device, which incurs excessive device-host transfer, as buffers need to be moved to and from the host memory and the GPU's VRAM.

Rsl_rl[13] is a dedicated framework designed by ETH's Robotic Systems Lab for robotics tasks. It re-implements PPO and vectorized environments in a different way that allows legged robotics learning tasks to be fully on GPU. Rsl_rl solves Stable-Baselines3's buffer transfer overhead by migrating PPO entirely to PyTorch-GPU, making it more scalable for mass paralleled training tasks. TocabiLab will modularly support both RL libraries and remain compatible to other RL libraries as will be discussed in Chapter 3. An experiment to compare the scalability of Rsl_rl against Stable-Baselines3 will also be conducted in Chapter 5.

Chapter 3

Design and Architecture

Rrobotics is one of the scenarios where reinforcement learning is frequently used[23, 41, 38]. The proposed system aims to provide mass parallel capability for robotics reinforcement learning tasks with versatility and efficiency. It provides modular components that can be swapped and customized, high parallelization capability, unified RL API and modern visual quality, thus reducing future research efforts and time commitment while producing better results.

3.1 Design Purpose

TocabiLab is named after the two essential previous research projects it is based on - Torque-controlled Compliant Biped (TOCABI)[30] and Isaac Lab[17]. While Isaac Lab provides our rendering capability and interfaces with the PhysX physics engine, TOCABI is chosen to be used for the first task that we implement in the system to prove the correctness of the system and provide a performance baseline. See appendix for more information on the base systems. We will hereafter refer the TOCABI paper as "reference work", and the codes it provides as "the MuJoCo codebase".

The proposed system aims to fulfill the need of performing mass paralleled robotic

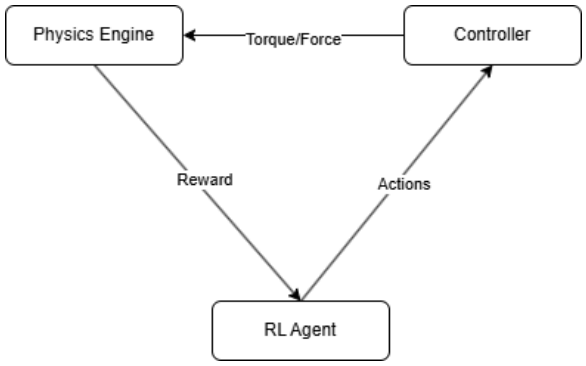

Figure 3.1: Generic Architecture of a robotics RL task

reinforcement learning tasks with **versatility** and **efficiency**.

Versatility by Modular Design

In robotics RL, the cooperation of three components, namely physics, learning and control, is critical. Figure 3.1 shows a generic architecture of a robotics reinforcement learning system, where an RL agent generates actions that get translated to force or torques controlling the robot, altering the environment and receive a reward. The system generally loops over this process, observing the correlation between actions, observations and rewards to learn an optimal strategy.

Problems arise when new RL libraries or controller models appear and researchers want to test them in their current environments. It is frequently the case that researchers need to start all from scratch so that novel methods or libraries can be integrated, since the new API can differ drastically from the old ones. This need is acknowledged by Issac Lab [17], and the proposed system thus extends the unified API provided by Isaac Lab to fulfill this purpose on torque-controlled legged robots.

Efficiency by GPU-driven Environment

As previously discussed in Chapter 2.4, parallelization in Python for RL tasks is largely handled by PyTorch, due to the lack of a thread model in the language

itself. However, using Stable-Baselines3 as-is wastes computational power due to additional host-device memory transfer. To address this issue, TocabiLab takes advantage of Nvidia's ecosystem and implemented a completely GPU-driven pipeline for the robots.

3.2 System Design

The TocabiLab Framework is a direct expansion of the concept presented in Figure 3.1, where we use PhysX as our physics engine and implement a parallel capable framework that contains the learning algorithm and robotic controller under a unified API.

Additionally, TocabiLab implements the ideal torque actuator that controls the interaction between RL agent and the physics engine, one locomotion task and integrated two RL agents. These contents serve as a tool to verify TocabiLab's correctness in reproducing previous results from the reference work, as well as a base example project of future research.

3.2.1 System Diagram

Figure 3.2 provides an overview of the components within the system. A booting script is executed first, starting the policy runner. The policy runner initialize the RL tool that the user specified, which starts the scene creation process. A unified API is designed to make this policy runner universal, and it does so by routing APIs from different RL libraries (such as SB3, rsl_rl) to a set of API that the policy runner can universally execute. As a result, the policy runner and the the DirectRLEnv base class don't need any modification when users wish to run their RL tasks using different RL libraries, as long as they implement their own unified wrapper.

The task itself extends the DirectRLEnv class and its APIs. The task will sample actions from the policy runner and progress the physics simulation with the action sampled. The controller applies torques or forces to the articulations it controls using the actuator model it owns. Lastly, the physics engine updates the scene and provides sensor data, such as measured forces and torques and contact reports. The task updates its states, and calculate a reward to feed to the policy runner.

We can notice that the system is designed to provide a layer of abstraction so that researchers can easily define new actuator models, tasks and customized RL policies without the need to touch the base workflow constructed by the policy runner and the base classes it uses. This design choice ensures the proposed system's ability to switch reinforcement learning backends with minimal engineering efforts, enabling researchers to do rapid prototyping to choose the optimal algorithm and RL library for their tasks.

3.2.2 System Interface and Specification

The proposed system uses a data-driven approach to ensure effortless transition of the robot assets and its properties between different RL tools.

Physics Properties

In robotics reinforcement learning tasks, we generally want the subjects to handle a predefined task autonomously. While Isaac Sim still provides interfaces to modify the training environment, they are rarely used at runtime. Instead, users should define the physics properties in the configuration files. Figure 3.3 and 3.4 shows the configuration interface for the Articulation Root and Physics Scene which govern the robots' joint force solver and the scene physics solver respectively. These parameters are contained in the task specification, as they should remain unchanged when the

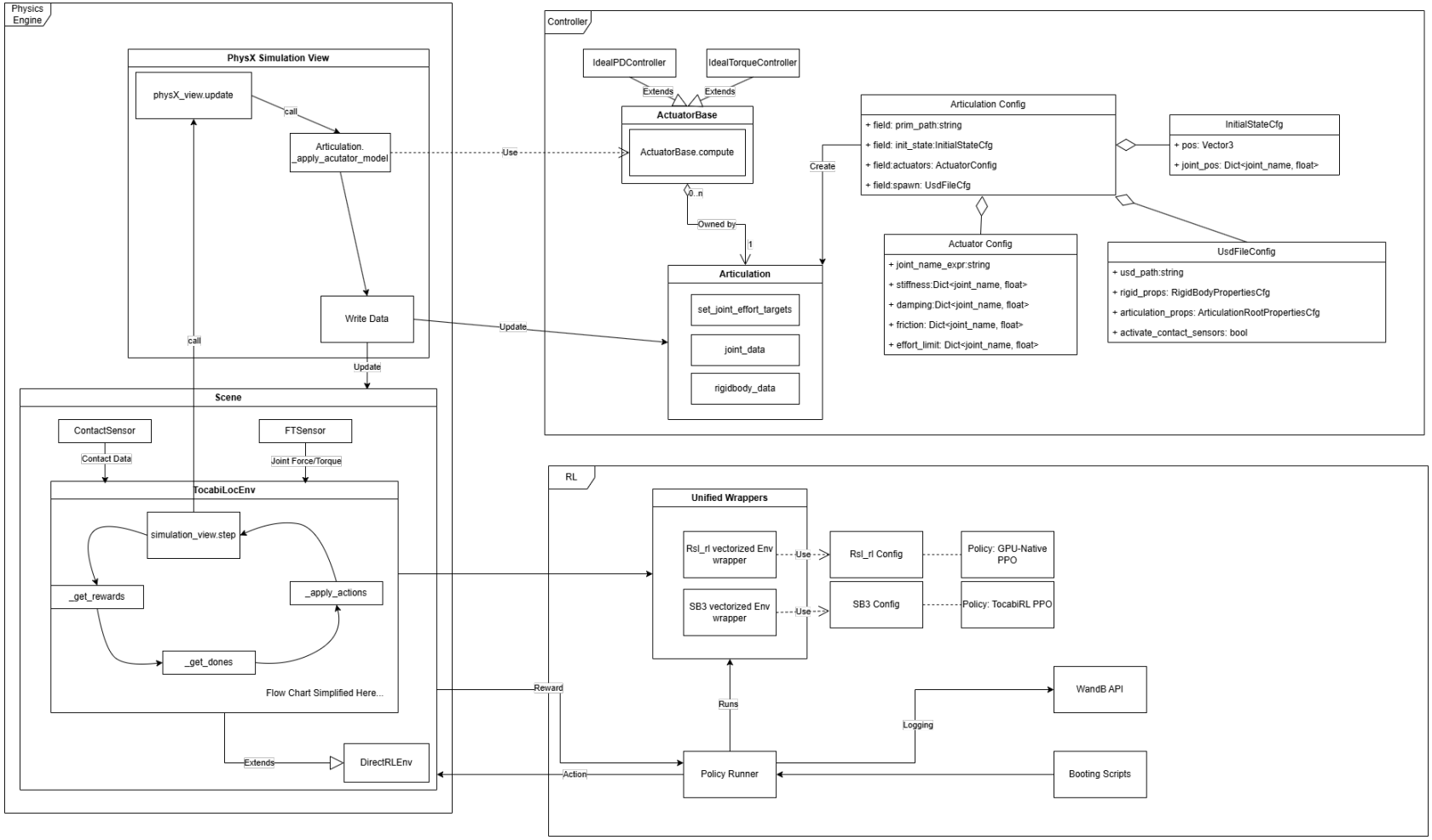

Figure 3.2: Diagram Overview of the System

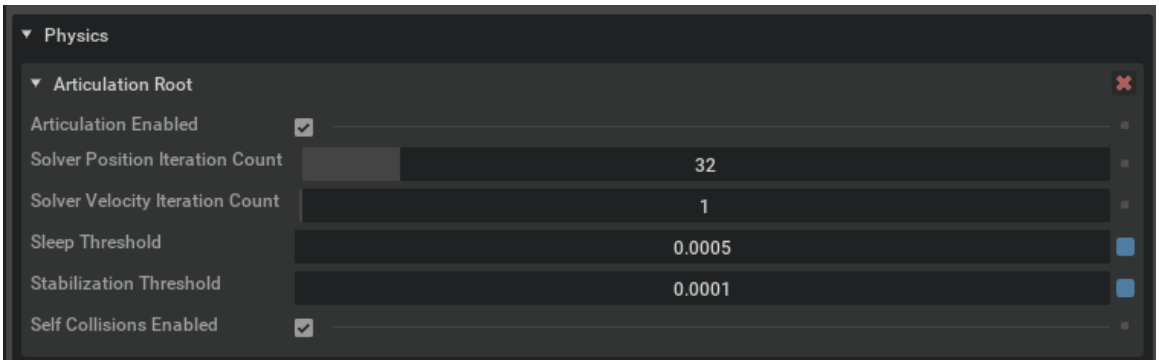

Figure 3.3: PhysX Articulation Setting Graphical Interface

task is trained under different RL tools.

PhysX uses Nvidia's closed-source Lula kinematic solver to resolve articulation kinematics, we therefore do not have mathematical insights towards the mechanism of the Articulation Root API. The two iteration counts here determines the solver's accuracy in approximating a joint's position and velocity, sleep threshold determines the minimum kinetic energy a joint must preserve to not be ignored by PhysX, and stabilization threshold determines the minimum kinetic energy for it to participate in stabilization. In practice, we try to tune these parameters manually to strike a balance between performance and stability.

The physics scene controls the scene's collision and contact resolution. In most cases in robotics, only rigid bodies are involved. As our experiment chapter involves a reproduced task from MuJoCo, we selected the Projected Gauss-Seidel Solver, which is supported by both MuJoCo and PhysX. The Gauss-Seidel method works by putting n active rigid bodies' kinetic equation and constraints to form an Linear Complimentary Problem (LCP) $Ax = b$, and iteratively solving it by running a loop for rigidbody

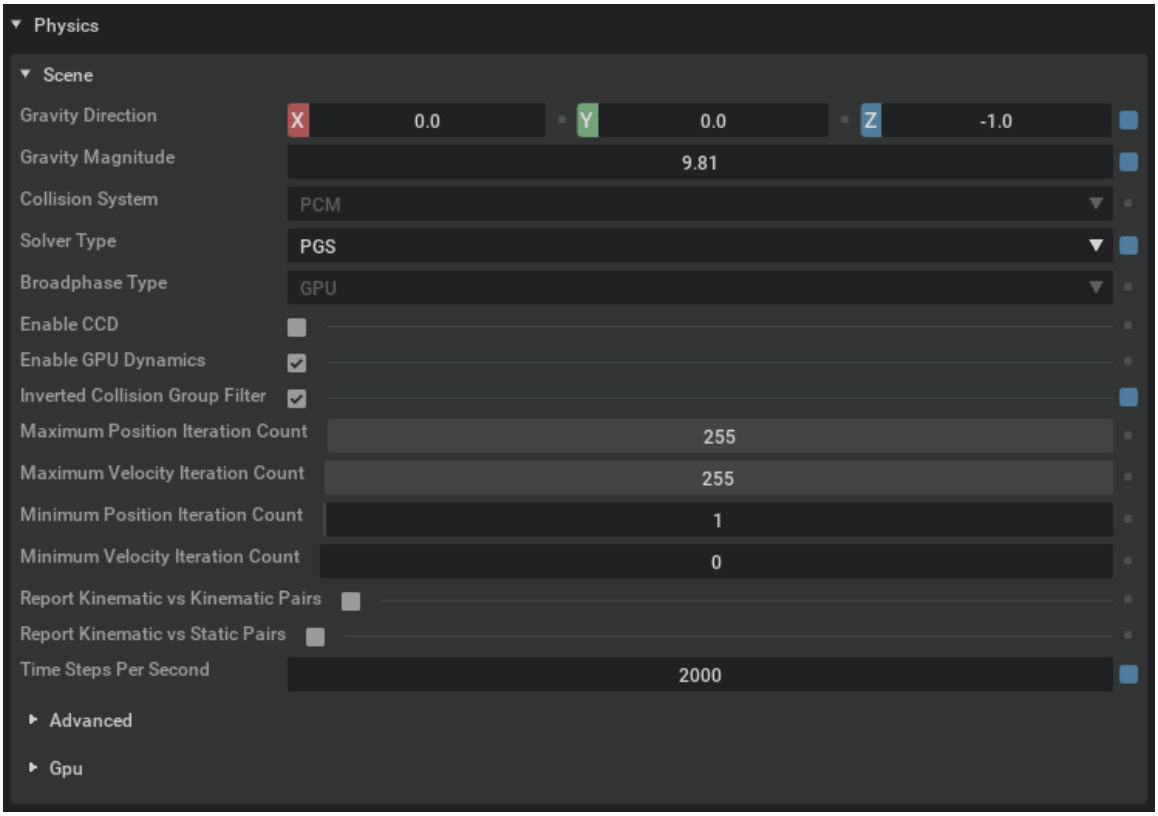

Figure 3.4: PhysX Scene Setting Graphical Interface

$i \in [0, n]$:

$$\Delta x_i = [b_i - \sum_{j=1}^{n} A_{ij} x_j]/A_{ii} \tag{3.1}$$

$$x_i = x_i + \Delta x_i \tag{3.2}$$

until the algorithm converges, i.e. $|\Delta x_i| < \epsilon, \forall i$. Check Catto's work [2] for details of the problem's formulation. In the proposed system, we setup our solver to get the maximum accuracy possible (255 iterations); The PhysX engine is set to update at 2000Hz, meaning the solver is executed 2000 times per second. Both settings align with the reference work [12].

Import Robots

A simulated robot is a combination of many rigid bodies and articulations connecting them. The proposed system uses a MuJoCo Modeling XML File (MJCF) importer tool provided by Issac Sim to directly import robot assets from the MuJoCo codebase. Similar to the PhysX backend settings, Isaac Sim also provides graphics interfaces for the robot's joints and bodies which are also rarely changed at runtime. The import tool copies the mass and center of mass properties from MJCF and the joint structures when importing the robot, then the user is required to create an ArticulationCfg configuration class to set its articulation properties and actuators.

Figure 3.5 illustrates the structure of this config file. The task will load this config class with a .usd file path to load the robot asset, and configure PhysX Articulation and Scene settings. It also handles the robots' initial world position and initial joint position.

The robot's actuator settings also resides in this configuration file. We can set the type of controller (e.g. PD, torque, model-free) and its properties such as stiffness,

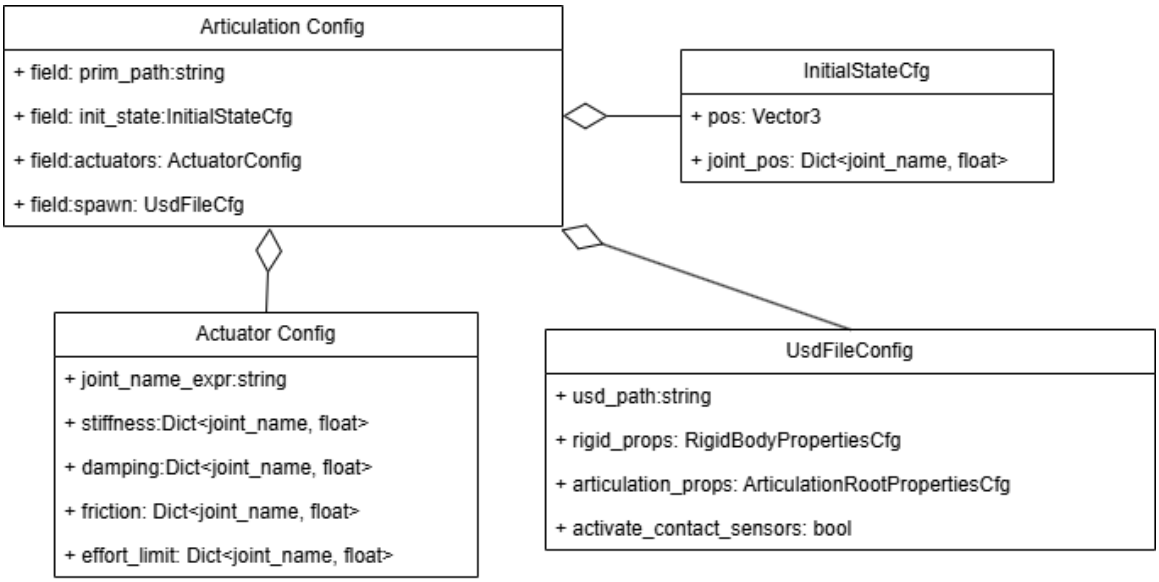

Figure 3.5: Articulation Config Structure

damping, torque limit and friction here. This part of the config will be discussed in more details in section 3.3.

Task Definition

We define a task as a universal workflow that can be extended to learn different control strategies to solve different problems. Figure 3.6 shows a flow chart of a running task workflow. Users add new task by simply extend the DirectRLEnv base class and implement these three functions and register it in the system. The task should then become executable for any RL libraries that is currently supported in the system.

Using the API wrapper provided by Isaac Lab, we are able to wrap popular reinforcement learning tools such as stable-baselines3 and rsl_rl under a unified API. This enables us to define tasks without the RL tool's context. The users will only need a few steps to run the task under a new RL policy or a different RL framework. As long as the task comply to the unified API provided by Isaac Lab, no changes will be needed for the task itself. This greatly reduced the effort needed from researchers

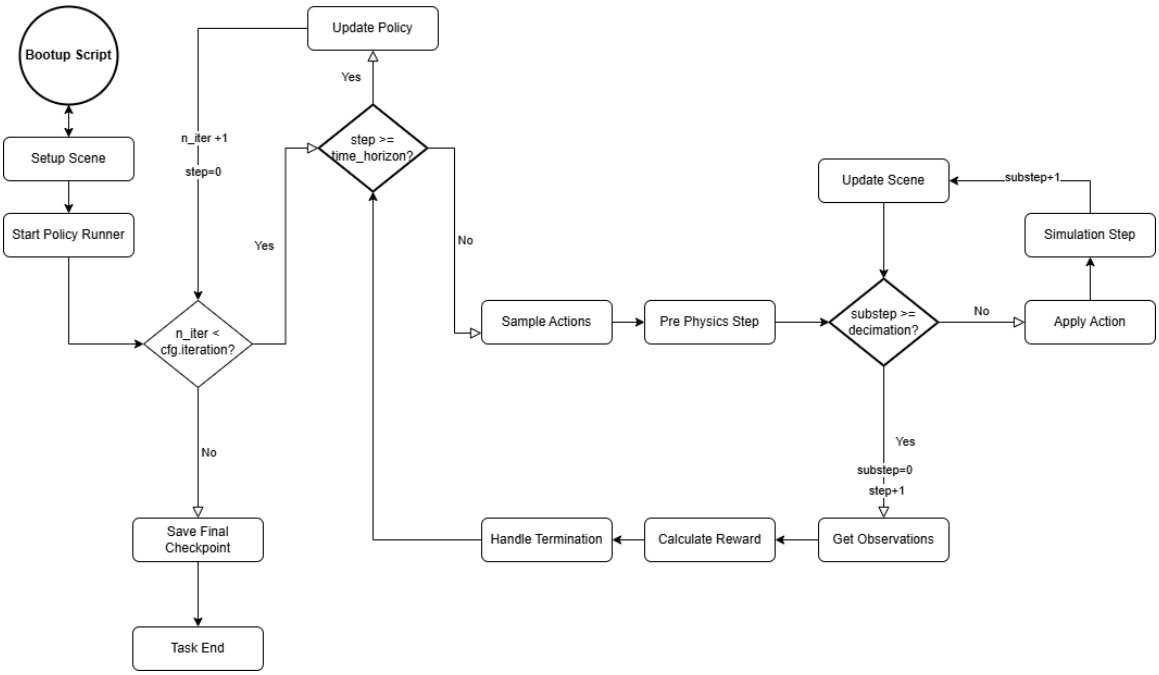

Figure 3.6: Policy Runner Task Flow

when they need to test out the performance of novel RL algorithms.

GPU-driven Pipeline

TocabiLab is a GPU-native system. In ideal situations, all data of TocabiLab will live within the GPU's VRAM. Since Isaac Lab and PhysX are both developed by NVIDIA, when using CUDA as computation backends, TocabiLab will be able to directly communicate between PhysX and the RL tool's data buffer (provided that the buffer is GPU-native). This drastically reduced the device-host transfer overhead and improved the speed and scalability of the training system by a notable margin, as we will show in Chapter 4 and 5.

This pipeline works differently from the MuJoCo codebase. To enable GPU-driven control over the learning process, we need to expose data-level parallelism for the RL problem. Effectively, this means all environments' data needs to be collectively handled. To achieve this parallelism, the CPU portion (non-PyTorch Python code)

must be synchronized at all times, meaning no branches can occur. However, it is very common for RL agents running in parallel to have different states, thus we cannot remove branches from the problem. Instead, we can ensure that a copy of data that only contains those need to be updated to be sent to the compute kernel. Namely, we need to form boolean buffers by state and slice the PyTorch tensor by that buffer, then we run the branchless kernel and write back to the flagged positions in the tensor.

The task we will later run in the experiment chapter is a good example to illustrate this difference. The agent only has two states, 'walking' and 'dead'. The `IsDead()` function will return True/False in both workflows based on the input shape, in it lies a few conditions that determines if the robot(s) queried should be terminated.

Figure 3.7 shows a typical multi threads/processes CPU-driven RL pipeline, where `n` environments learn independently but collectively update the policy. The state of each agent is determined and handled individually. This model is widely used because it's relatively easy to implement, as users can write the RL task in a single-threaded fashion and the RL library can handle the rest by vectorizing the data when the collective update happens.

Figure 3.8 shows the GPU-driven pipeline that TocabiLab uses. We removed the two branches presented in the CPU pipeline by introducing two buffers, `Timeout Buffer` and `IsDead Buffer`. The main difference is that all actions, observations and states are now gathered in a data buffer that includes all environments - for example, if the action space has a length of 13, then the action buffer will be a tensor of shape `(n,12)`. Similarly, when we update the controllers (denoted in the flow graph as "Apply Actions"), this buffer will be sent to the controller, then translated to `(n,12)` torque values we apply simultaneously to an array of joints by calling a vectorized PhysX interface function.

When branching is needed, such as in the `Reset Env` function, the `ResetFlag`

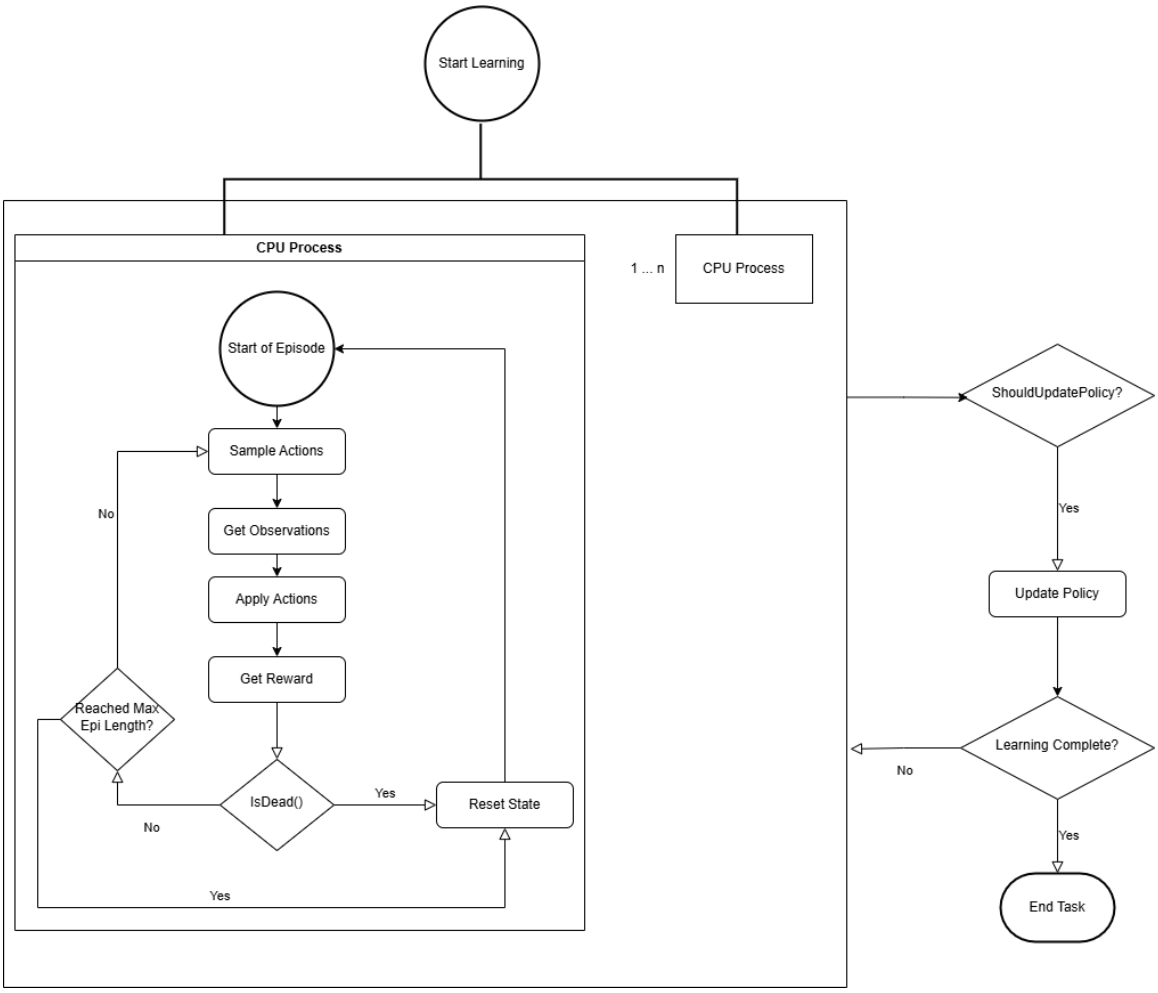

Figure 3.7: Typical CPU-driven RL Pipeline

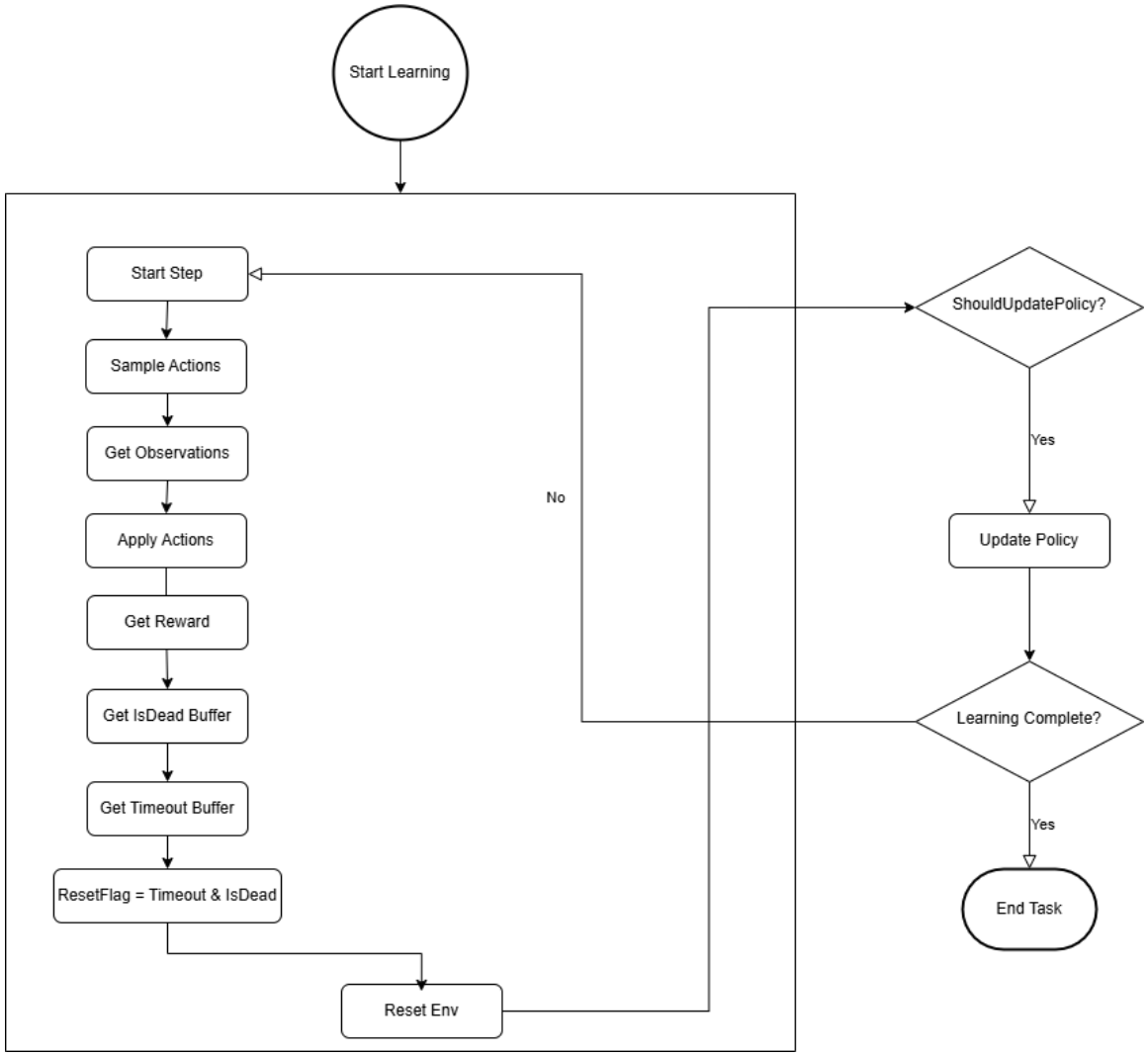

Figure 3.8: GPU-driven RL Pipeline

`Buffer` we formed from bitwise operation will be fed to a similarly vectorized reset function, which simply writes the default state back to the flagged environments, effectively resetting them without branching.

We can see that in TocabiLab's GPU driven pipeline, one CPU process is enough to govern any number of parallelized environments provided their scale does not exceed the capacity of the GPU, while the reference work's CPU-driven pipeline is bound by the CPU's thread count. We removed this bound by moving all relevant data into the VRAM, and CPU is thus only used to dispatch GPU calls.

3.3 Torque Control on TOCABI

In the proposed system, there are multiple actuators available to control the robot. We implemented an ideal torque controller and an ideal PD controller under a unified API, enabling them to be swapped at the user's disposal. More actuators can be added to extend this system.

3.3.1 Controller Model

The controller refers to an interface that takes a set of actions of shape `(n,a)`, where `n` is the number of parallelized environments and `a` is the size of action space, and convert them to actuation on the robots. The actuation can be forces on a rigid body or torque applied to a revolute joint. In TocabiLab, we only support revolute joints as all joints found on TOCABI are revolute joints, and other types of joints (such as 3 DoF ball joints) can be modeled as multiple 1-DoF controllers. All actuators output torque, but they can be controlled by different models.

Multiple controller models are present in the proposed system, there are two main categories of controllers that are applied on TOCABI in the referenced work,

namely position-based and direct torque. They share the same interface, where they essentially implement a function for a group of joints:

$$\tau_{cmd} = F_{con}(\pi_\theta) \tag{3.3}$$

where τ is the command torque, π_θ is the policy output (actions).

Position-based Controller

In the position-based deepRL method, the policy outputs the target joint angle, and the target joint angle is tracked by a low-level PD controller. PD stands for "Proportional-Derivative", which is a sub model of PID (Proportional-Integral-Derivative) feedback-based control loop mechanism. Position-based PD controller gets feedback in the form of position (Proportional) and speed (Derivative), outputting torque with the following equation:

$$\tau_{cmd} = K_p \cdot (q_{target,\pi_\theta} - q) + K_d \cdot (-\dot{q}) \tag{3.4}$$

The policy output here is a target angle, and the output torques are generated based on two parameters K_p and K_d. In practice, they are often called stiffness and damping, as higher K_p usually makes the controller snap faster to the target angle, while higher K_d makes the controller slow down quicker. PD controller is widely used in articulated robots, as it can react to predefined tasks quickly and precisely. However, the tuning of PD controllers requires domain knowledge, and usually don't work well when the operational environment changes[26]. In Experiment 4 and 6 in the reference work [12], torque control shows superiority adapting to tracking errors.

In practice, the policy will output $a \in [-1, 1]$ with shape (n, a) corresponding to $n * a$ controlled joints in the scene. Each element in a will then be mapped to the

angle limit of the robots. We then apply this action as a target angle directly on PD-controlled joints.

Torque-based Controller

Torque controller is simpler in robotics deep RL context, since the controller will directly output a torque value from the policy. The policy output π_θ will now be mapped to the torque limit of the joint directly, as opposed to a target angle.

$$\tau_{cmd} = \pi_\theta \tag{3.5}$$

This means no parameter tuning is required on the controller itself. The removal of tuning also means that researchers with less domain knowledge can work with robotic problems more productively. This method is also shown to be stable and applicable to Sim-to-Real scenarios[12].

3.3.2 Actuator Model

Currently, all actuators in the proposed system are ideal models, meaning that the command torque are only affected by the friction model:

$$\tau = \tau_{cmd} - F_{resist}(\tau_{cmd}, F_{trans}) \tag{3.6}$$

where τ is the actual torque that is applied to the joint in the physics engine. F_{resist} models the friction resistance of the joint, which reduces the actual torque.

3.3.3 Comparison to the Reference System

Since this framework's first deep RL task involves a reproduction of an existing research project, the comparison between the reference system and the new system is essential.

The control model is interchangeable between the two systems, as the weight unit (kg) and torque unit (N·m) are the same. As a result, when importing an MJCF file into the proposed system, the body mass will be automatically copied and applied. Joint stiffness and damping values need to be set manually in the robot's configuration file, this is by the data-driven design of the system, since we want to adjust joints configurations from the codes rather than modifying the usd file itself.

The joints' naming is the same in both system. However, due to PhysX's articulation system not yet supporting joint reordering, we will have to remap some data (particularly the DeepMimic reference action) to MuJoCo's order so that we can get correct results. This functionality is implemented as `get_mj_joints_access_array` and `get_mj_bodies_access_array` in the proposed system, they essentially convert an array slice in MuJoCo to an access array in TocabiLab that accesses the same joints or bodies.

The control frequency is kept the same across both systems. Both systems update the controller at a frequency of 250Hz, and the controller applies action to the robots at 2000Hz.

Lastly, the most important difference is friction modeling in the two systems. This is an intrinsic trait built into the physics engine that we can't really change, and indeed PhysX models friction differently from MuJoCo.

PhysX documentation [21] shows that frictions between surfaces use the Coulomb

friction model, which is governed by this equation:

$$F_f \leq \mu F_n \tag{3.7}$$

where μ is the friction coefficient between the two materials, F_n is the normal force.

On joints, a different model that still simulates static and Coulumb friction is applied. This model applies a force to resist joint motion based on force and torque transmitted per-axis from the parent body to the child body. Omniverse documentation listed the model [20] as:

$$F_{resist} \leq \mu(|F_{trans} + T_{trans}|) \tag{3.8}$$

where F_{trans} T_{trans} are the transmitted force and torque. This is not a physical model, but chosen because the calculating transmitted force and torque is very cheap for joints. Note that this model is used to model both static and dynamic friction, as the transmitted torque and force are non-zero even when the joint is not moving.

For our reference system, MuJoCo's original paper [35] described its friction modeling. Friction between rigidbodies in MuJoCo is also using Coulomb friction model. The joint friction model, however, is different. Chapter II, Section C of MuJoCo's theory paper [34] explained MuJoCo's model of dry friction. We can notice that there are not a model specifically for dynamic friction - the reason is that dynamic friction is included in the contact constraints in MuJoCo. In other words, the dynamic joint friction is modeled by the Coulumb model as well. For static friction, MuJoCo define a constant $\eta(i)$ for each joint. This value is represented by a constant field `friction loss` in the MJCF file, and is essentially the maximum dry friction force in Newtons(N) associated with a joint.

From above analysis, we conclude that there's not a direct equivalent way to

model MuJoCo's dry friction in PhysX. We set the friction parameter in TocabiLab's configuration to be proportional to the MuJoCo static value in order to approximate its dry friction. In practice, we simply set the friction coefficient in TocabiLab to be:

$$\mu = 0.02 * F_{static} \tag{3.9}$$

This is tested to work pretty well, as we will show in the training and joint trajectory experiments in the following chapter. For dynamic friction, as both system simulates Coulomb friction, we keep default $\mu = 0.05$ for both systems. Although the two systems use different ways to represent the normal force F_n, it is reasonable to assume that they still perform relatively similarly.

Chapter 4

Correctness Experiments

Chapter 4 and 5 will examing the correctness and efficiency of the system thoroughly by performing multiple experiments. This chapter will be geared towards correctness, while the next chapter will be more on the efficiency and scalability side.

In this chapter, we present 3 experiments designed to verify the correctness of the system and compare the performance (in terms of training and locomotion quality) across multiple systems including our reference[12]. We will implement TOCABI's walking task in the proposed system with GPU-driven architecture.

The first experiment port the learning algorithm(Stable-baselines3's PPO) directly from the MuJoCo codebase and run the experiment to show that we can still achieve similar training results in a new physics engine; The next two experiments examine how well the robots in the new system learns the reference motion and then reproduced an obstacle experiment from the reference work. While this chapter focuses more on the correctness expressed by new training frameworks, more performance metrics will be presented in Chapter 5.

4.1 Experiment 1: Reproducing Reinforcement Learning Task from TocabiRL

A robot control problem in the proposed system is modeled as a discrete-time Markov Decision Process (MDP) [12]. The robot's state at timestep t is modeled as s_t, and the policy runner samples an action $a_t \sim \pi_\theta(\cdot|s_t)$. The given action is passed to the actuator, the actuator generates torque or forces to transition the robot to state $s_{t+1} \sim p(\cdot|s_t, a_t)$. Reward for this step $r_t = r(s_t, a_t, s_{t+1})$ is calculated subsequently. The agent wrapped in the policy runner then tries to learn a policy through interaction with the environment that maximizeds the expected return $J(\pi_\theta)$, which is the expected cumulative discounted reward over a finite-horizon T

$$J(\pi_\theta) = E_{\tau \sim p(\tau|\pi_\theta)}[\sum_{t=0}^{T-1} \gamma^t r_t] \tag{4.1}$$

where τ is the command torque when executing the policy π_θ.

An RL task is formed by implementing the reward function $r_t = r(s_t, a_t, s_{t+1})$, the stepping function $p(\cdot|s_t, a_t)$ and an observation function $s_t = obs(t)$. In the stepping function, the user-defined actuator converts the action a_t to a command that changes the trajectory in the physics engine, τ.

4.1.1 Task Design

We chose to reproduce the reference work's walking task where TOCABI tries to walk with a given target speed. We made a few changes to the design of the environment aiming to simplify the problem space, since this task is the starter task in this framework and making it simpler aligns with its purpose of getting new users on board. The design of observations, rewards and actions remain largely unchanged,

but we removed most noises and randomness, which was originally implemented in the reference codebase for a more stable Sim-to-Real behavior.

Action Space

The action space remains unchanged from the reference work. The upper body of the robot is controlled by a PD controller to remain in the reference pose at all times, and the lower body (12 DoFs) are controlled by the policy generated action $A_{jnt} \in R^{12}$. In addition, one action that modulates the phase $\phi_t \in [0, \Delta t]$ is generated. The phase variable controls the reference phase we load from DeepMimic cyclic motion, which can be expressed as:

$$\phi_{t+1} = fmod(\phi_t + \frac{\Delta t}{T_{ref}} + a_{\delta\phi,t}, 1.0) \tag{4.2}$$

The action space $A = [A_{jnt}, a_{\delta\phi,t}] \in R^{13}$ will update at 250Hz with 8 decimations, meaning the controller is executing at 2000Hz.

State Space

The state space is kept the same as the reference work $S \in R^{31}$. It consists of the robot's base orientation in Euler angles $\alpha_t \in R$, $\beta_t \in R$, $\gamma_t \in R$. The lower body joint position $q_t \in R_{12}$, and joint velocity $\dot{q}_t \in R_{12}$, the phase information represented by sine and cosine $sin(2\pi\phi_t) \in R$ and $cos(2\pi\phi_t) \in R$, a command velocity $v_t^{cmd} \in R^2$.

Differences compared to the reference work include:

1. Removed low pass filtering of the joint velocity for simplicity.
2. Added y-axis movement for the command velocity. This allows the robot to move slightly sideways.

These changes are also applied to the MuJoCo codebase for experiments. The noise injected to the joint position sensors is removed as well.

The observation buffer we feed to the policy includes more information. In addition to the above state space S, we add supporting force of both feet on the Z axis $F_{z,l}, F_{z,r} \in R$, roll / yaw torque on the feet $T_{x,l}, T_{x,r} \in R$, $T_{y,l}, T_{y,r} \in R$, forming $O \in R^{37}$. From here, we then use two circular buffers to form an observation buffer of the previous 4 steps of state transition, including 5 observation spaces $O_{t-4}, O_{t-3}, O_{t-2}, O_{t-1}, O_t$ and corresponding actions $A_{t-4}, A_{t-3}, A_{t-2}, A_{t-1}$. In total, $O_{bf} \in R^{237}$.

Training Process

Both system run the same episode length of 8000 steps, meaning a robot can run for 8000 steps if it isn't terminated prematurely. At 250Hz, the simulation length is capped at 32 seconds. During this 32 seconds, we change the command velocity v_t^{cmd} each 2000 steps (8s) to simulate a user input. The velocity $v_{cmd} \in R^2$ is sampled from two uniform distributions $v_{cmd,x} \sim U(0.0, 0.7)$, $v_{cmd,y} \sim U(-0.2, 0.2)$, except for the initial speed on X axis where we sample from $U(0.2, 0.7)$ for a higher starting speed - this reduces the possibility of converging to a standing local optimum.

The episode terminates prematurely under the following conditions:

1. When a self collision happens on the robot.
2. When any part of the robot except the feet contacts the ground.
3. When the torso height of the robot is less than 0.6m.
4. When the neck height of the robot is less than 0.8m.

We also removed Gravity Pre-training and Dynamic Randomization from the reference work for simplicity.

Rewards

In table 4.1, we show the mathematical definition of the reward components. They are kept similar to the reference work, albeit implemented differently with GPU buffers. The buffer names will be included with following interpretation of the component. Like the reference work, the reward components are heavily based on Peng et al.'s DeepMimic[23] with specific components for cyclic motions on bipedal robots.

$r_t^{base,imitate}$ `(mimic_body_orientation_reward)` is the base reward for keeping body orientation in the same direction of the reference motion. This reward is calculated by calculating the quaternion error of the base frame compared to the reference motion.

$r_t^{q,imitate}$ `(qpos_regulation)` is the DeepMimic imitation reward. This component calculates a regulated reward based on the total deviation of all controlled joints from the reference action. The reference animation data is captured in the MuJoCo codebase and directly copied to TocabiLab, which contains 3600 frames of joint angle record and feet with 2 cyclic walking animation in total.

$r_t^{c,imitate}$ `(foot_contact_reward)` is calculated based on the current support phase indicated by the reference motion. The agents only receive this reward when their feet contact situation matches with the current reference motion. There are in total 3 support phase defined for our reference motion: Φ_{DSP} (Double Support Phase), where both feet make contact with the ground; $\Phi_{SSP,l/r}$ where only one foot makes contact with the ground. We use contact sensors on TOCABI's feet to determine if the agents receive the reward.

$r_t^{v,task}$ (`body_vel_reward`) is given to the agent by calculating the world frame speed's difference with the command speed v_t^{cmd}.

$r_t^{\dot{q},reg}$ (`qvel_regulation`) is a regulation reward that encourages the robot to regulate joint speeds. $\dot{q}$ is the current angular speed measured on all joints on the robot

in rad/s.

$r_t^{\ddot{q},reg}$ (`qacc_regulation`) is a regulation reward that encourages the robot to take smoother actions. The robot is rewarded if it tries to apply less acceleration to the joints.

$r_t^{F,reg}$ (`contact_force_penalty`) is applied to the agent based on the readings from the feet FT (Force & Torque) sensors. While $r_t^{c,imitate}$ encourages the robot to make periodic feet contact according to the reference action, this less-weighted component makes the agent try to keep the contact force on the robot's feet minimal.

$r_t^{\Delta F,reg}$ (`contact_force_diff_regulation`) regulates the stepping process of the robot, encouraging it to lift/lower the feet in a smoother manner.

$r_t^{\tau,reg}$ (`torque_regulation`) and $r_t^{\Delta\tau,reg}$ (`torque_difference_regulation`) regulate the command torque similarly, encouraging gradual change of the commanding torque instead of extreme actions.

$r_t^{F,imitate}$ (`force_ref_reward`) takes the recorded contact force readings in the reference motion as reference. The robot is thus encouraged to use similar level of actuation force to imitate the reference cyclic motion.

Additionally, two penalty components are applied to prevent sudden change of supporting forces, which is usually detrimental in real world scenarios. Table 4.2 shows their definitions. The two thresholds are defined as:

$$F_{spf}^{threshold} = 1.4 \cdot G \tag{4.3}$$

$$F_{\Delta spf}^{threshold} = 0.2 \cdot G \tag{4.4}$$

where G is the total gravity force applied on the robot.

$$G = 9.81 \cdot \sum_{i=0}^{n} m_i \tag{4.5}$$

Reward	Definition
$r_t^{base,imitate}$	$0.3 \cdot \exp(-13.2\|q_{base,t}^{ref} - q_{base,t}\|)$
$r_t^{q,imitate}$	$0.35 \cdot \exp(-4.0\|q_t^{ref} - q_t\|_2^2)$
$r_t^{c,imitate}$	$\begin{cases} 0.2 \quad \text{if} \begin{cases} c_{r,t} = 1, c_{l,t} = 1 \quad \text{if } \phi_t \in \Phi_{DSP} \\ c_{r,t} = 1, c_{l,t} = 0 \quad \text{if } \phi_t \in \Phi_{SSP,r} \\ c_{r,t} = 0, c_{l,t} = 1 \quad \text{if } \phi_t \in \Phi_{SSP,l} \end{cases} \\ 0 \quad \text{else} \end{cases}$
$r_t^{v,task}$	$0.3 \cdot exp(-3.0\|\|v_t^{cmd} - v_t\|\|_2^2)$
$r_t^{\dot{q},reg}$	$0.05 \cdot exp(-0.01\|\|\dot{q}_t\|\|_2^2)$
$r_t^{\ddot{q},reg}$	$0.05 \cdot exp(-20.0\|\|\ddot{q}_t\|\|_2^2)$
$r_t^{F,reg}$	$0.1 \cdot exp(-0.0005(\|\|F_{r,t}\|\|_2 + \|\|F_{l,t}\|\|_2))$
$r_t^{\Delta F,reg}$	$0.2 \cdot exp(-0.0005(\|\|\Delta F_{r,t}\|\|_2 + \|\|\Delta F_{l,t}\|\|_2))$
$r_t^{\tau,reg}$	$0.05 \cdot exp(-0.01(\|\|\tau_{cmd,t}\|\|_2))$
$r_t^{\Delta\tau,reg}$	$0.6 \cdot exp(-0.01(\|\|\Delta\tau_{cmd,t}\|\|_2))$
$r_t^{F,imitate}$	$0.1 \cdot [exp(-0.001(\|\|F_{r,t} - F_{r,t}^{ref}\|\|)) + exp(-0.001(\|\|F_{l,t} - F_{l,t}^{ref}\|\|))]$

Table 4.1: Reward Components (Torque Experiment)

Penalty	Definition
p_t^{spf}	$\begin{cases} \text{-0.08} & \text{if } \|\|F_{r,t}\|\| > F_{spf}^{threshold} \text{ or } \|\|F_{l,t}\|\| > F_{spf}^{threshold} \\ 0 & \text{else} \end{cases}$
$p_t^{\Delta spf}$	$\begin{cases} \text{-0.05} & \text{if } \|\|F_{r,t} - F_{r,t-1}\|\| > F_{\Delta spf}^{threshold} \text{ or } \|\|F_{l,t} - F_{l,t-1}\|\| > F_{\Delta spf}^{threshold} \\ 0 & \text{else} \end{cases}$

Table 4.2: Penalty Components (Torque Experiment)

p_t^{spf} (`force_thres_penalty`) is applied when either foot's supporting force exceeded the total gravity on the robot by 40%. This usually happens when the robot start or finishes a jump. We apply this penalty to stop the robot from learning to jump.

$p_t^{\Delta spf}$ (`force_thres_diff_penalty`) is applied when the difference in supporting force on either foot in one controller update differs more than 20% of the robot's total gravity. This penalty works with `contact_force_diff_regulation` to regulate the supporting force on the robot's feet in order to prevent acute changes which are generally undesirable.

4.1.2 Learning Algorithm

Both the reference work and the proposed system use variants of PPO (Proximal Policy Optimization) algorithm [29]. The versatility of the proposed system enables us to directly port the original algorithm implemented in the reference work. Our first experiment is a **correctness** experiment that compares the training curve and the results directly from the ported algorithm.

The reference work trains the task defined above in section 4.1.1 using a modified version PPO. Namely, the standard deviation of the policy output Σ where $\pi_\theta \sim N(\mu_\theta, \Sigma^2)$ is fixed and scheduled value which is calculated using the following method:

$$\log \Sigma = \begin{cases} \log(exp(\Sigma_{start} * (2P - 1) + \Sigma_{end} * (2 - 2P))) & \text{if } P > 0.5 \\ \log(exp(\Sigma_{end})) & \text{else} \end{cases} \tag{4.6}$$

$$P = n/N_{step} \tag{4.7}$$

where P indicates the current progress of the training, calculated by the current timesteps count divided by the maximum timesteps allowed in this training run. Here we write Σ in log form since it is expressed this way in the original PPO code, as `log_std`. This method ensures the agent explores the action space enough in the first half of the training, and exploit the knowledge it learns in the later half by fixing the log standard deviation of actions to a lower value.

The proposed system is capable of running the original customized PPO directly, but the performance degrades significantly due to device-host transfer overheads. As previously mentioned in Chapter 2.4, Stable Baselines 3 introduces excessive device-host transfer and is not suitable for GPU-native applications like the proposed system. We will show the difference in Chapter 5 through experiments.

Instead of SB3, we are using Rsl_rl with its implementation of PPO. To simulate the scheduling behavior, a specific set of hyper parameters can be used with Rsl_rl's adaptive std mode, and in our results it demonstrated similar behaviors compared to the manually scheduled PPO. Following (Figure 4.1) is a comparison between the standard deviation scheduling.

4.1.3 Results

This experiment shows that the migration of the reference system to Isaac Sim remains stable, while still generating a similar training process under higher parallelization

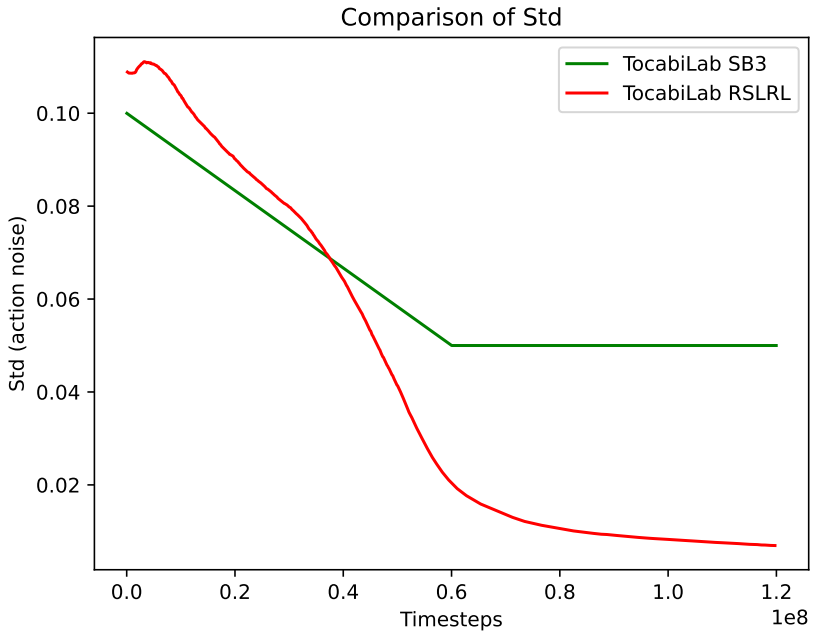

Figure 4.1: Std Scheduling difference

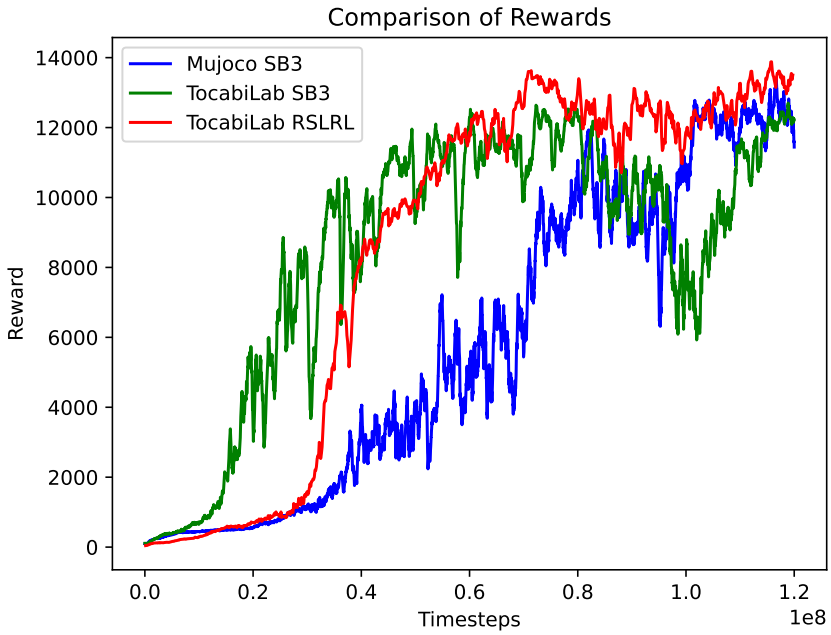

Figure 4.2: Reward Curve

workloads. A "full training" refers to a process where the robot controller is trained for $1.2 * 10^8$ timesteps. For this experiment, we run 8 environments in MuJoCo (iteration = 7324), 16 environments in TocabiLab with SB3 (iteration = 3662), and 64 environments in TocabiLab with Rsl_rl (iteration = 916) and do a full training in each of these setups.

By quantitatively compare the reward curve, we can get a rough picture that all three different setups are training and converging to similar optimum in Figure 4.2.

The reward curve alone is not a sufficient evidence of the system's correctness, although it does prove that we did not introduce any instability or local optimum in the process of reproducing this experiment. To prove that we get a similar controlling policy with the proposed system, more quantitative and qualitative results will be shown in Experiment 2.

Reward $r_t^{q,imitate}$ `(qpos_regulation)` directly reflects how well the robot is learn-

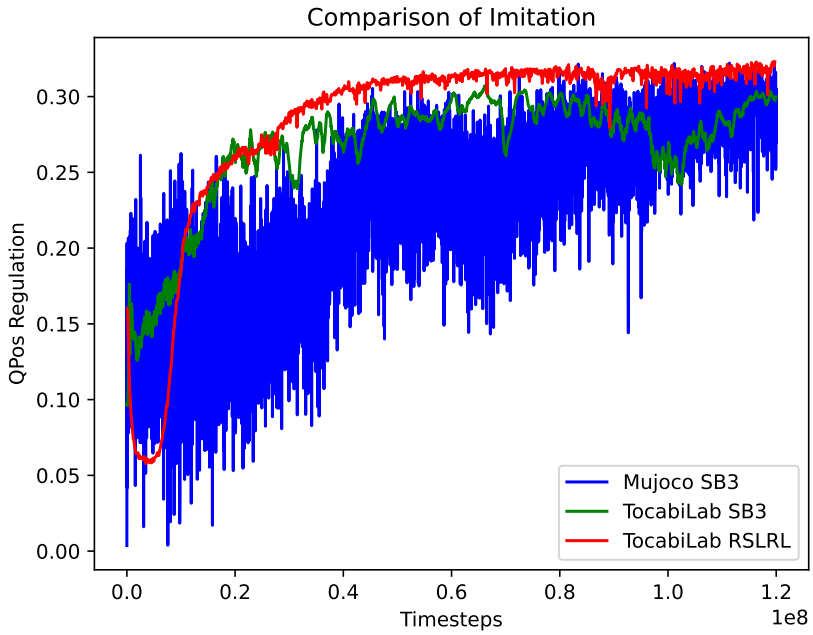

Figure 4.3: Imitation Reward Curve

ing to imitate the reference pose. In both systems, the reference motion is the same, therefore the similarity in this reward reflects the similarity in motion. Figure 4.3 illustrates this similarity. Note that the episodic rewards are reported differently in MuJoCo because of the difference in the thread model. MuJoCo reports episodic rewards each time an environment finishes an episode, while TocabiLab updates this value once per iteration (with a fixed timestep interval), therefore we have approximately 8x more data points from MuJoCo.

We can conclude from this experiment that the proposed system trains similarly to the reference system, and is still imitating the reference motion relatively well.

4.2 Experiment 2: Gait Comparison

Our reinforcement learning task is designed to make TOCABI learn to walk in a simulated environment. In this chapter, we will compare the gait of the robots running

learned policy in the proposed system and the original system. Quantitatively, we will present the joint torque comparison of several critical joints in the first few seconds on the trained policy; Qualitatively, we will present robot gait time-lapse shots as well as video evidences.

To make this experiment as consistent as possible, we will set the initial command speed to a fixed value of $v_t = (0.5, 0)$ in inference mode instead of randomizing it. We will also let each system run for 8 seconds instead of 32, which does not trigger a random speed change. The expected behavior of the robot is they walk forward for approximately 4m in a manner that resembles the reference pose.

4.2.1 Visualization of Gait

There will be 3 groups for comparison:

1. TocabiLab SB3 (Torque)
2. TocabiLab Rsl_rl (Torque)
3. TocabiLab Rsl_rl (PD)

The camera (fixed at position $(2, 6, 2.5)$) will start capture a frame when $t = 0s$, and each follow up shot will take an interval of $\Delta t = 2s$, resulting in 5 screen captures from the side. These photo shots are then edited together for comparison. The robot walks from the right side of the camera to the left, and each grid on the ground represents a 1m distance.

The video stips show that different systems work relatively well when walking on plain terrain they learned on. The gaits of the robots are shown to be similar to the reference motion visually. In the next subsection, we will compare the trajectory and torque output between different systems quantitatively. The Rsl_rl policy walks slightly faster than the other two, but the pose did not deviate from acceptable

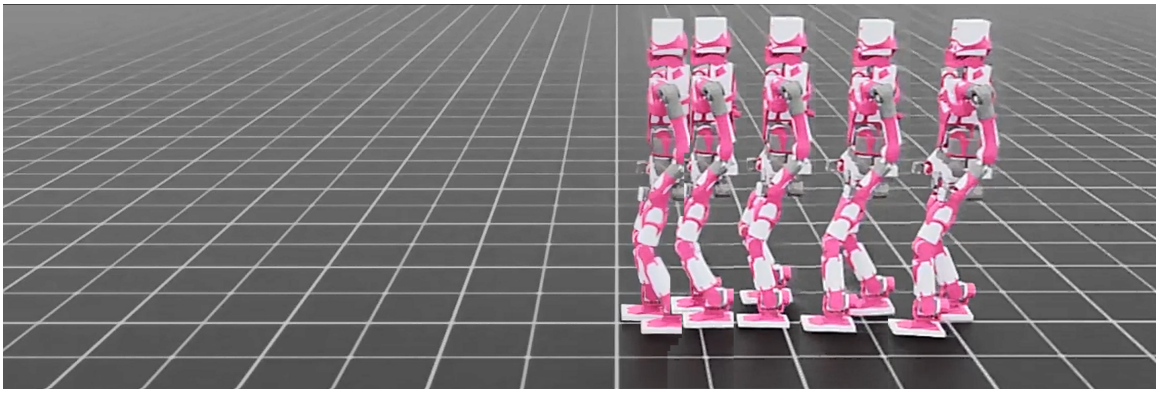

Figure 4.4: Inference using SB3 trained policy

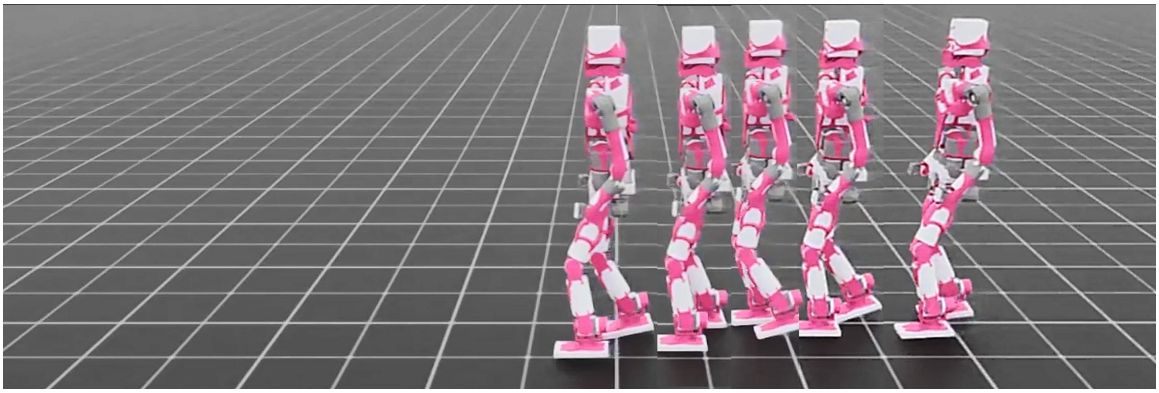

Figure 4.5: Inference using RSLRL trained policy

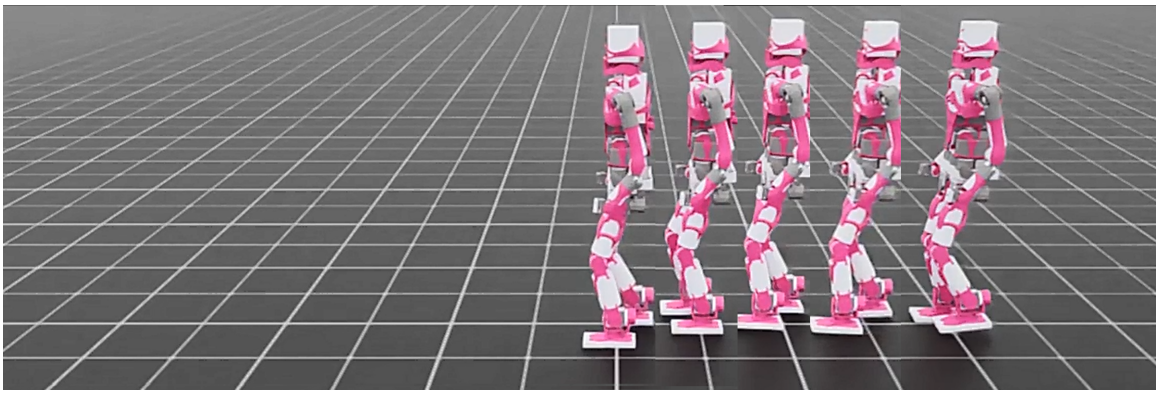

Figure 4.6: Inference using Position-based policy

range. All 3 groups of this experiment walked approximately 4m which align with the command speed.

4.2.2 Joint Trajectory

We can also show that the 3 different groups of systems are learning the reference motion by comparing the joint trajectory to the reference motion.

In the charts, the joint position is shown in radian $\theta \in [-\pi, \pi]$, the reference motion is plotted in black. Figure 4.7 shows the left knee's joint position compared to reference; Figure 4.8 shows the right knee. Since the phase variable partially controls the system, we can see a slight deviation at times, but all 3 groups maintained in motion cyclically. The result shows that TocabiLab reached acceptable level of control on the robot with different combination of RL libraries and control methods, proving its correctness and versatility.

4.2.3 Result

We can conclude that the proposed system can be used to train robotic locomotion tasks effectively using different controllers or different RL libraries, as all three combination of RL library and controllers managed to successfully imitate the reference motion. We can also notice that the higher initial noise we set for the rsl_rl system (see Figure 4.1 from last section) resulted in slightly more impulsive actions and slightly higher speeds, but all three groups managed to stay in a cyclic motion manner in locomotion.

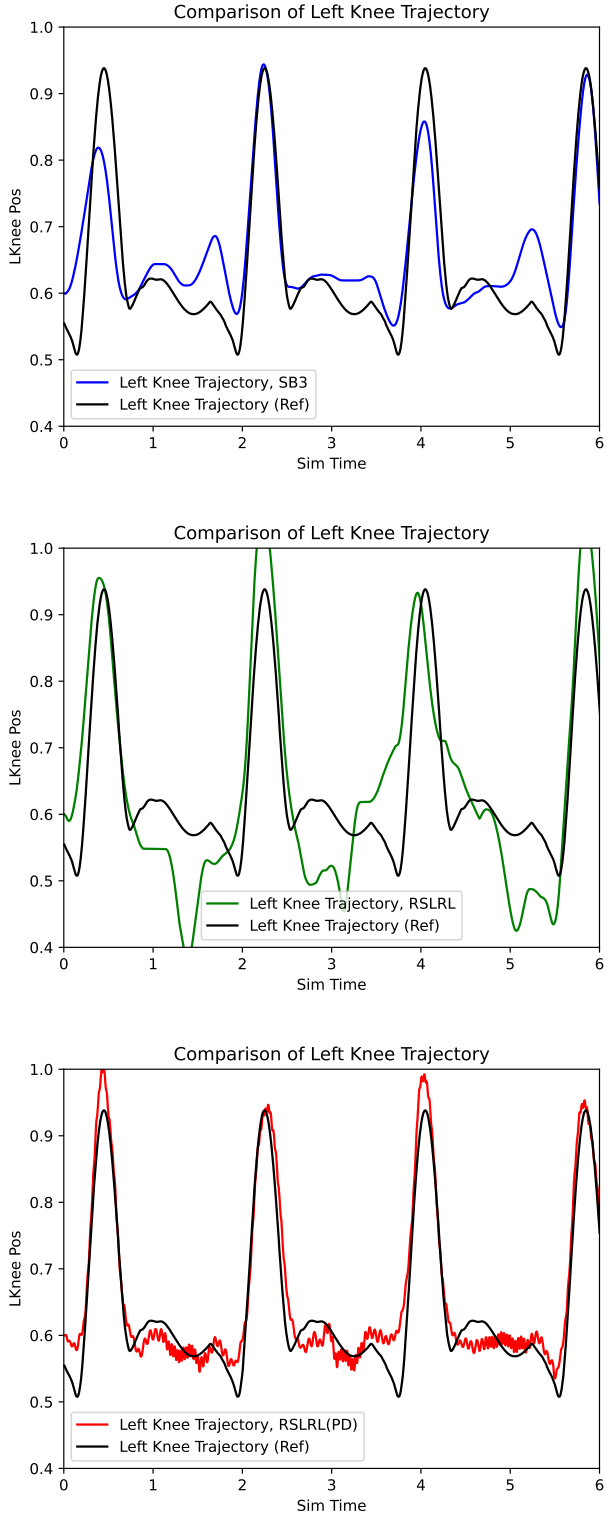

Figure 4.7: Left Knee Trajectory in 6s

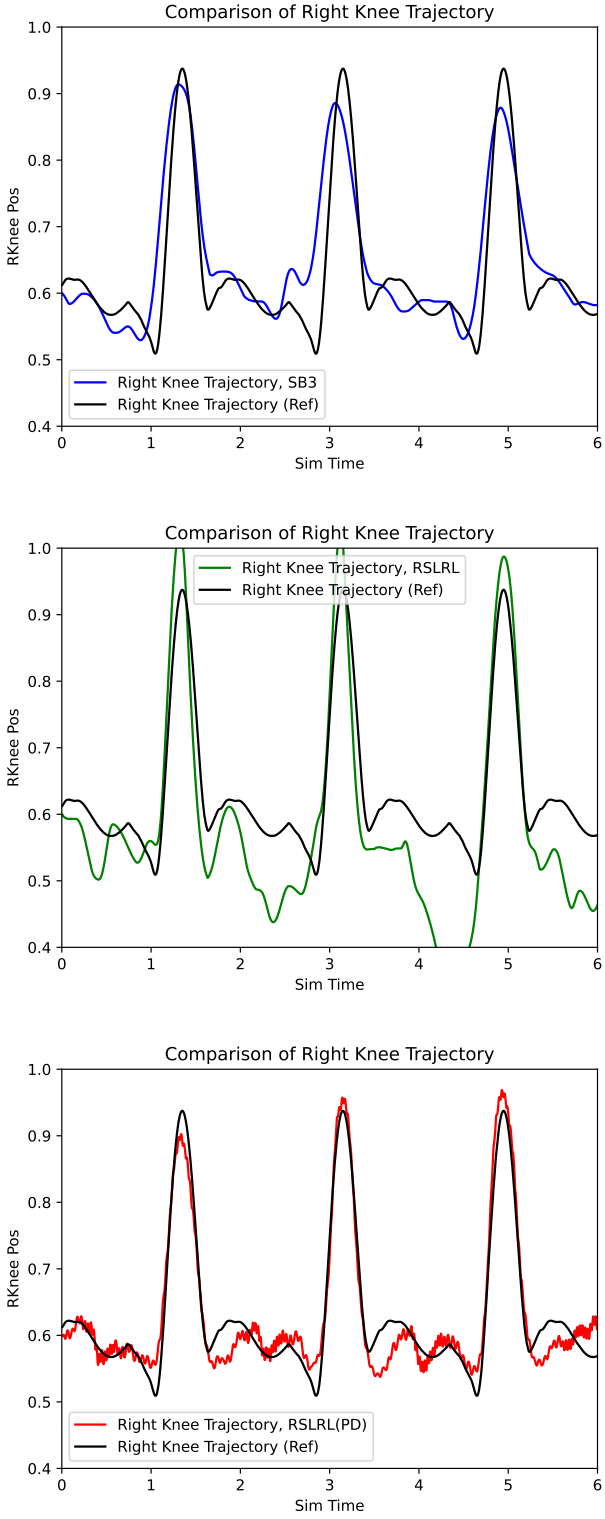

Figure 4.8: Right Knee Trajectory in 6s

4.3 Experiment 3: Handling Obstacles

The reference work showed interesting results in Chapter. V, Experiment 4, where a small obstacle is added to the scene and the robot's handling of unexpected contact is tested. While the MuJoCo codebase used a separated code branch to implement PD control, in the proposed system, this experiment is reproduced in a cleaner way. This experiment reproduced the expected result and demonstrated the versatility of our unified API.

Similar to the original experiment, we put an additional obstacle on the path of the robot. The obstacle is a thin cuboid that has variable height. We will put it at approximately 60 cm distance to the robot's left foot and set the target velocity to $(0.5, 0)$ so that the robot steps on the box when taking the first step. The policy has no perception of the obstacle, and the obstacle will not move as we set its mass to ∞.

4.3.1 Handling unexpected contact

We experiment on 3 different heights (5mm, 12mm, 20mm) with the torque-based policy and position-based policy, both obtained from rsl_rl. The experiment result is classified as "Failure" if the robot triggers the episode termination after stepping onto the obstacle.

The PD controlled robot handles unexpected contacts poorly, and creates oscillations on contact no matter if there is an obstacle or not. This could be due to poor tuning, which further emphasize the importance of domain knowledge required for a PD controller. The PD controlled robot successfully passed an 5mm obstacle, but fails immediately after stepping on the 12mm and 20mm obstacles.

On the other hand, the torque controller proves to be more resistant to small

obstacles. It succeeded in passing the obstacles in all 3 experiments, but started deviation from its original path in the 20mm experiment.

The obstacle takes 3 steps to go through, we therefore observe the first 3 steps starting at the robot's first step on the obstacle for each experiment.

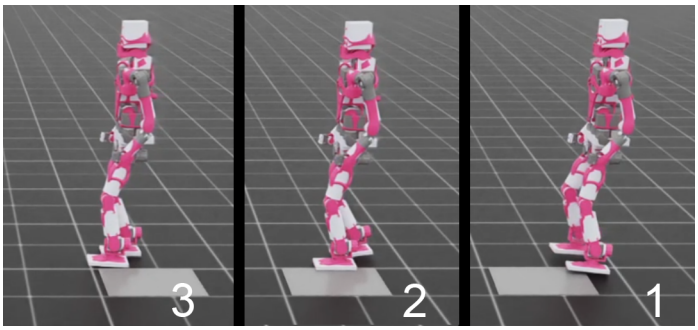

Figure 4.9: PD - 5mm obstacle

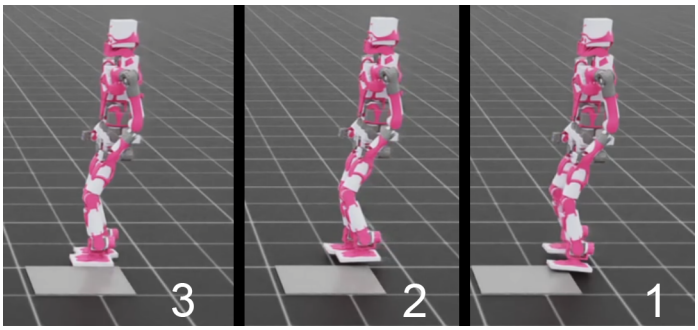

Figure 4.10: PD - 12mm obstacle

4.3.2 Observation

The PD controller performed poorly in all three experiments. It managed to not trip over from the 5mm obstacle shown in Figure 4.9, but a tendency of sideway propulsion can be observed, and it started to move to the right-hand side after the unexpected contact.

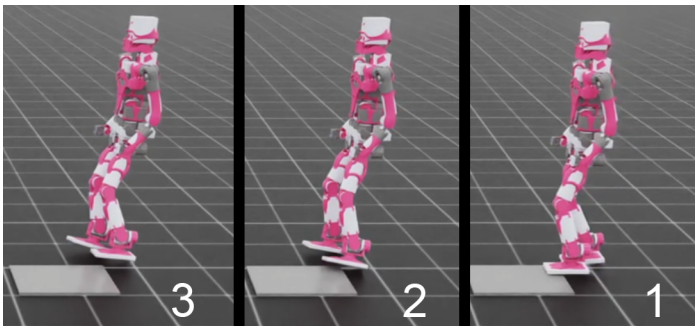

Figure 4.11: PD - 20mm obstacle

It failed both 12mm (Figure 4.10) and 20mm (Figure 4.11) experiments. In the former, a self collision between feet happened after the second step and terminated the experiment. The latter sees the robot propels backwards and fall over after the first step.

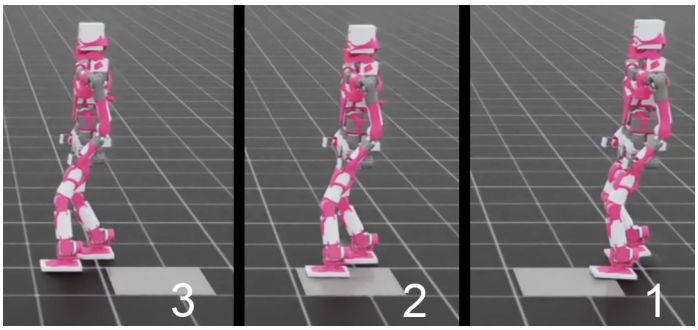

Figure 4.12: Torque - 5mm obstacle

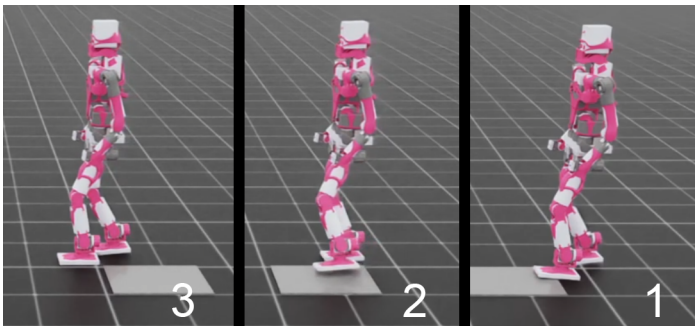

Figure 4.13: Torque - 12mm obstacle

The torque controller passed through the 5mm(Figure 4.12) and 12mm (Figure 4.10) with no obvious deficiency observed. The 20mm obstacle (Figure 4.14) caused a slight deviation in moving directions.

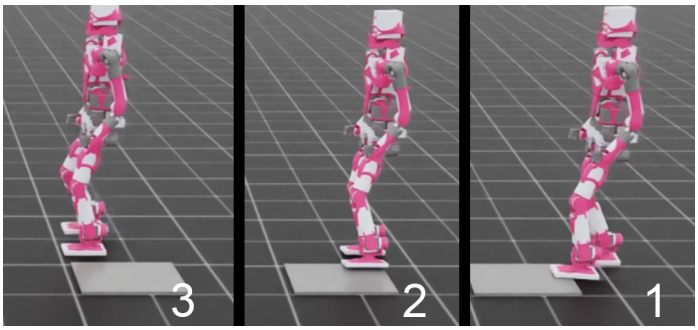

Figure 4.14: Torque - 20mm obstacle

4.3.3 Torque variation caused by the obstacle

We also monitored the torque output from the left ankle joint. While we did not tune the PD controller to the smoothness of the reference work[12], we observed the similar patterns as the Experiment 4 in that paper.

Figure 4.15 and 4.16 shows the left ankle torque output from both policies with obstacles compared to without. The unexpected contact happens at around $t = 2.6s$. We observed similar pattern to the reference work [12] with a similar obstacle (5mm), and obtained comparable results. We also experimented on larger obstacles than the original experiment where instabilities started to appear on torque controllers, but torque controllers recovers to normal stance within $0.2s$ in both experiments.

4.3.4 Results

We can see from the chart and the screenshots that PD control struggles to stabilize with the lowest obstacle. Although it eventually managed to not fall over, the path deviates from the reference. For higher obstacles, PD-controlled robot fails to step through, resulting in early termination. Torque-control managed to pass all three experiments and continue on walking. Some instabilities appeared when stepping on higher obstacles, but it returns to normal torque output quickly and maintained an overall stable pose.

This experiment also serves to further proof the versatility offered by the system, as virtually no engineering is required in modifying the scene (such as the obstacle or the terrain) or the action space of the robot (PD and torque).

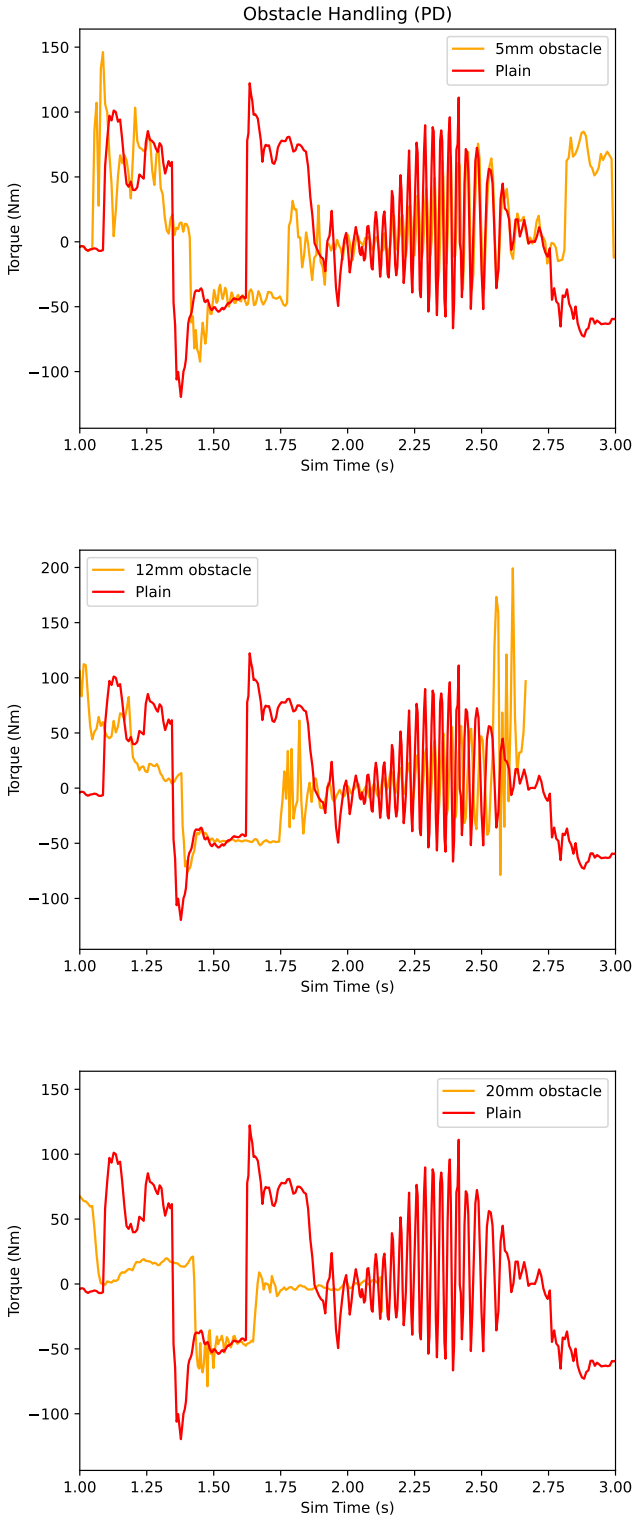

Figure 4.15: PD Controlled Torque

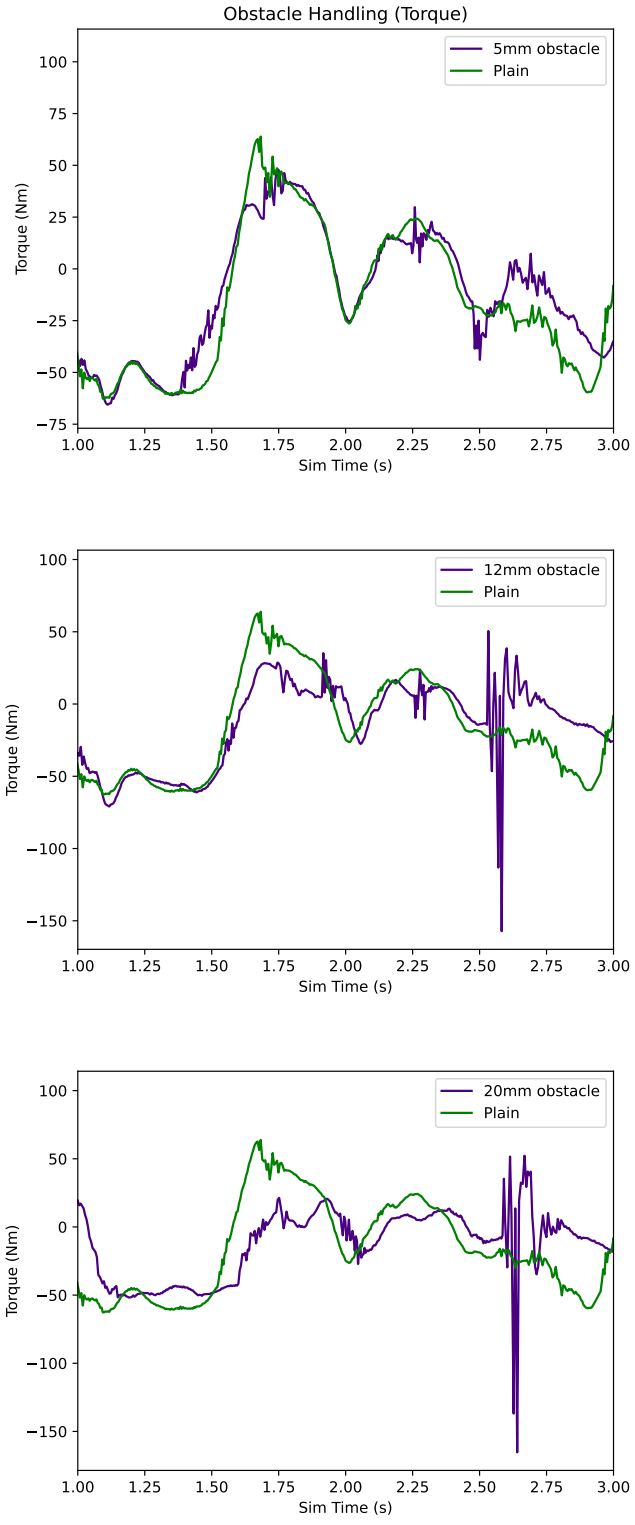

Figure 4.16: Torque Controlled Torque

Chapter 5

Efficiency and Scalability Evaluation

Chapter 4 examined the correctness of the system by running both training and inference on a reproduced publication and comparing them to reference. While the experiments are successful, it is more important that we see the proposed system's key advantage over MuJoCo - scalability.

In this chapter, we run three training sections for 3 system combination below to measure the performance of the performance and efficiency of the proposed system. All three groups will be running the torque-based controller. The system will be denoted by a single letter in Table 5.1. We measure training time by starting the timer when the environment setup is complete, and stopping the timer before the process ends.

1. MuJoCo with Stable-Baselines 3 (M)
2. TocabiLab with Stable-Baselines 3 (S)
3. TocabiLab with rsl_rl (R)

No.	System	Time Horizon	Env. Count	Iteration	Total Steps	Training Time(s)
1	M	2048	8	60	983040	1535.22
2	S	2048	8	60	983040	8847.47
3	R	2048	8	60	983040	9013.46
4	M	2048	32	15	983040	979.49
5	S	2048	32	15	983040	2672.08
6	R	2048	32	15	983040	2522.88
7	M	2048	48	10	983040	7832.77
8	S	2048	96	5	983040	1076.25
9	R	2048	96	5	983040	850.37
10	M	2048	96	5	983040	Failed

Table 5.1: Training Speed Result

5.1 Training Efficiency Benchmarks

We examine the training time under different loads to quantitatively compare the performance of different setups under different loads. We set the time horizon to 2048, and control the total timesteps by adjusting the amount parallelized environments and the maximum iteration count. We run a total of 983040 steps, which gives the cache on both processors enough time to warm up.

The PC used for this test runs on a 32-core i9-13900 CPU and an RTX 4070Ti-Super GPU, with 32 GB of RAM. The proposed system runs on Windows while the reference system is hosted on WSL on the same machine, and all systems will run headlessly to prevent rendering impacting performance.

We expect to see CPU bound when above 32 environments are parallelized (the CPU has 32 cores), which means all three system should get a similar performance boost from low to mid parallelization loads, while the GPU-based system gets more from mid to high parallelization loads.

In practice, we noticed that the CPU-based solution gained less performance than expected from more threads being dispatched, even when the number is below the core counts. This could be due to architecture reasons, as our test machine has a

8P16E setup, meaning 16 threads on the Eco cores are lower powered than the other 16 threads on performance cores. At 32 threads, the P cores are already bottle-necked by the E cores, thus less performance benefits are achieved by increasing thread counts from 8 to 32.

The MuJoCo system failed to run 96 threads. we then lowered the environment counts to 48 and encountered significantly worsened performance, indicating the CPU is struggling to keep up with the task.

We can convert the training time T to an efficiency metric by

$$E = S/(T * 480) \tag{5.1}$$

$$[N_{env} * N_{iter} = 480] \tag{5.2}$$

$$S = 983040 \tag{5.3}$$

which indicates the steps that are learned per environment per second under each setup.

As shown in Figure 5.1, the GPU-based environments are significantly inferior to CPU under low loads, but they can scale roughly to the parallelization load without harming per-environment efficiency. The CPU-based system, while performs better under lower load, begins to experience sharp drops in efficiency once it passed the capability of the CPU, and cannot handle mass parallelized environments.

Meanwhile, we also notice slight deviations between performances of the two GPU-based solutions. As previously mentioned, although SB3 is wrapped to work on the GPU in TocabiLab, it does not support GPU-native buffers, which means more and more device-host transfer will happen when the parallelization load grows. We anticipate this difference to grow larger when more environments are parallelized.

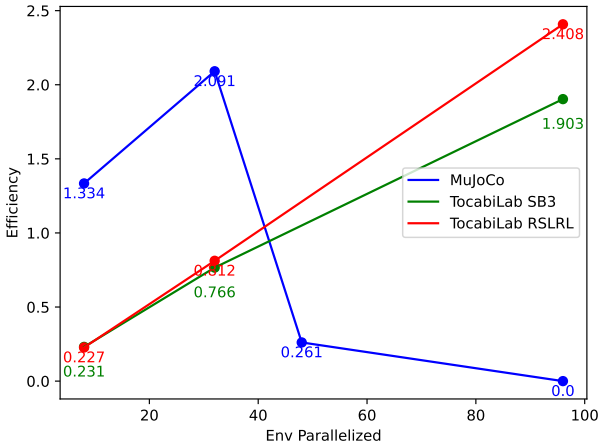

Figure 5.1: Efficiency Plot

Host RAM Usage(MB) / Env Count	8	16	32
MuJoCo SB3	6430.3	9638.8	15562.2
TocabiLab SB3	5439.8	5507.7	5554.5
TocabiLab RSLRL	5406.4	5496.4	5574.1

Table 5.2: Host RAM Usage

Device VRAM Usage(MB) / Env Count	8	16	32
MuJoCo SB3	280	328	383
TocabiLab SB3	4796	4992	5089
TocabiLab RSLRL	4739	4766	4828

Table 5.3: Device VRAM Usage

5.2 RAM and VRAM Usage Benchmarks

Since the MuJoCo solution is already failing at higher loads, for the memory benchmarks we'll be lowering the experiments to run on 8, 16, 32 environments. Each experiment will run one iteration and report the average RAM consumption. We extract host RAM data using the PyCharm profiler, and VRAM usage by NSight profiler during said time.

RAM and VRAM usage is critical for a system's scalability. The maximum amount of objects that a system can host on a fixed memory budget can greatly affect the parallelization capability as well as calculation efficiency. TocabiLab supports GPU-instancing which essentially creates only one instance of a robot and copying the data to create duplicates of it, instead of directly create more robots. The RAM/VRAM usage is expected to increase by a noticeably smaller margin compared to non-instancing methods, as we are only copying variable data (such as rigidbody, transform and physics properties) but not mesh and vertices which took lots of space.

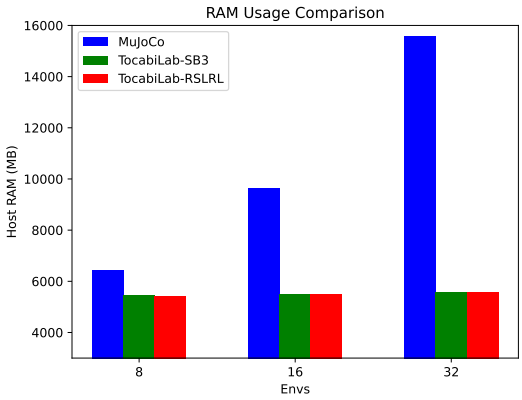

Figure 5.2: Host RAM Scalability

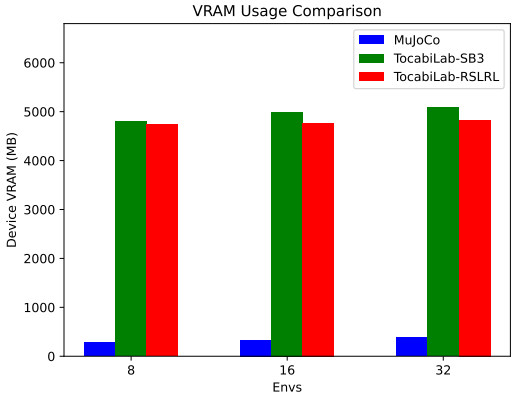

Figure 5.3: Device VRAM Scalability

The reason for MuJoCo failing for higher loads is highly likely RAM overflowing, as we can observe a pattern of linear RAM usage growth. Running 96 MuJoCo processes can take up to 47GB RAM which is more than the test machine's capability. Running 48 environments also exceeded the test machine's physical memory capacity and introduced lag from virtual memory page exchange, resulting in much poorer performance.

Figure 5.2 and 5.3 show that both GPU-based experiment groups in TocabiLab proved to be very scalable, as we did not observe linear growth in either RAM or VRAM. The system is GPU-native and utilized GPU-instancing technology, therefore

CPU Utilization(%) / Env Count	8	16	32
MuJoCo SB3	22.5	39.1	88.6
TocabiLab SB3	23.2	23.6	26.1
TocabiLab RSLRL	18.6	19.2	18.9

Table 5.4: CPU Utilization

we do not introduce additional robot instances when we parallelize more environments in training, this prevents redundant data existing in the memory space while retains the ability of the system to individually assess and control different environment instances. SB3 uses slightly more RAM and VRAM than rsl_rl when load grows, which is likely caused by the device-host transfer overhead.

Unfortunately, TocabiLab does not scale beyond 96 robots at this moment due to a bug related to FT sensors (see the next chapter and appendix) although it is theoretically possible to run 4096 TOCABIs in the proposed system from the statistics. We expect this issue to be fixed soon as Isaac Lab will add support for FT sensors in newer versions.

5.3 CPU Usage Benchmarks

In this section, we run the same experiment as last section, collecting CPU utilization rates. The rates will be a rough estimate collected from PyCharm profiler. Table 5.3 shows the result.

We can observe the same pattern as the previous section from Figure 5.4 where TocabiLab generally remains very scalable when the number of environments grows, making it very suitable for mass paralleled applications compared to MuJoCo. The CPU's workload stayed around the same across experiments for TocabiLab, which indicates it's less reliant on the CPU for computation. Meanwhile, MuJoCo quickly approaches the limit of physical cores with the growth of load as expected, since it is

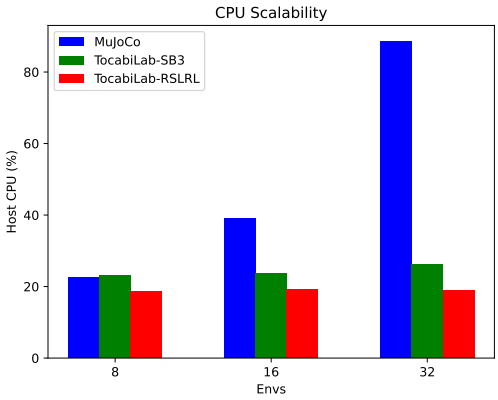

Figure 5.4: CPU Scalability

creating copies of the entire environment when vectorizing the training process.

5.4 Conclusion

Regarding scalability, the proposed system shows great benefit gained from the novel GPU-native architecture, as we can grow the parallelization load without introducing significant overhead. The system is also less reliant on host RAM, VRAM and CPU when scaled, albeit requiring a higher starting hardware specification compared to MuJoCo.

Chapter 6

Conclusions, Limitation and Future Work

6.1 Conclusion

Through reproducing a well established research under the proposed system, we completed the framework development of the proposed system and proved its versatility and efficiency. The unified API for reinforcement learning already support a wide range of available algorithms and frameworks and can be easily extended or customized. The novel GPU-native architecture successfully overcomes the context barrier between RL systems and physics engine and achieved comparable performance under normal load and avoided CPU bound under higher loads. Modular design of the controller API supports different joint control models.

We also proved the transition from MuJoCo to the proposed system can be done with relatively low efforts, and provided the experiment design as part of the report as well as a starter environment of the system. Overall, TocabiLab proves to b a valid new framework that can be used in future robotics research. In broader terms, this

system can be further extended to support generalized physical character control, and the mass-parallelization capabilities enables the potential for application in physical crowd simulation as well.

6.2 Limitations

While we managed to reproduce a highly-sophisticated research in this project, we removed some features that were deemed unnecessary to validate the reproduction, such as gravity pre-training and dynamic randomization. We also did not tune the PD controller to match the reference work's performance, resulting in less stability on PD-controlled robots.

Due to the system being GPU-native, programming in the proposed system can be more difficult compared to MuJoCo. Most data manipulations and control must be done in a matrix operation fashion, and branching must be done with index buffers - this makes the codes less readable and harder to maintain. However, we still believe this system to be a good investment, as it provides much better scalability than CPU based system.

The proposed system is less feasible when used under low loads. Some traffics between the host and the GPU are still happening, such as logging and kernel dispatches. This extra cost outweighs the performance boost before we hit the CPU bound, resulting worse performance below 32 environments. CPU mode is supported in Isaac Lab, but a existing bug prevents us from using it, as the contact sensors do not work in CPU mode. Another bug is causing PhysX to stall when more than 96 environments are created. Both of these issues are reported to NVIDIA and will be fixed soon, as listed in the appendix.

Isaac Lab is a relatively new framework and is updated frequently. Users are

advised to upgrade TocabiLab along with Isaac Lab until the parent system reaches a stable Long-Term-Support version.

6.3 Future Work

The system itself is very robust and modular, thus the future work will mostly be expanding its current functionalities. More RL framework wrappers and controllers can be implemented and integrated, and we can validate it with more tasks and other robots by migrating previous works from other lab members.More efforts are required to keep it updated, as we will need bugfixes and new features in the future.

Looking beyond the current scope of the project, while this report only explores parallelization of single-robot environments in the proposed system as of now, it is entirely possible to be used to handle multi-agent systems. The perceptions and contacts of the robots are all available natively on the GPU in the form of sensors, and the mass parallelization capability makes the system a great tool for future research on physical crowd simulation.

On the ease-of-use side, there are also efforts from NVIDIA that aim to making GPU-programming easier. Warp [16] is a Python-based programming framework that JIT compiles single-thread fashion python codes to GPU-kernels. Currently, NVIDIA is working on supporting the Warp toolkit in Isaac Lab, and we may be able to start experiment Warp usage soon in the proposed system.

Appendix A

Additional Information

A.1 Hyperparameter Configuration for Different Systems

All of our experiments are conducted using the PPO algorithms , despite under two different frameworks (Stable-Baselines 3 and rsl_rl).

The checkpoint for the original system is obtained with the following hyperparameters:

Parameter Name	seed	n_steps	batch_size	n_epochs	gae_lambda	clip_range	vf_coef	max_grad_norm	activation	net_arch (actor/critic)
Value	42	2048	128	3	0.65	0.2	0.5	0.5	relu	[256,256] / [256,256]

Table A.1: Hyperparameter settings: SB3 PPO

The torque-controller trained using migrated SB3 are using the same set of hyperparameters.

Rsl_rl is used to train another torque controller and the PD controller. The hyper parameters are as follows:

Both systems use virtually the same hyperparameters except the network architecture. The learning rates are not shown here because they are both scheduled. See Chapter 4.

Parameter Name	seed	num_steps	mini_batches	learning_epochs	lam
Value	42	2048	128	3	0.95

clip_param	value_loss_coef	max_grad_norm	activation	net_arch (actor/critic)
0.2	0.5	0.5	relu	[400,200,100] / [400,200, 100]

Table A.2: Hyperparameter settings used in rsl_rl PPO

A.2 Parent Toolchain Version

As extensions of Isaac Lab, the proposed system relies on correct version of the simulator (Isaac Sim) and base framework (Isaac Lab) to work. NVIDIA usually update these two systems together.

The system started developments in Isaac Sim 4.0.0 and Isaac Lab 1.2.0. It was later updated to Isaac Sim 4.2.0 and Isaac Lab 1.4.1. At time of writing (Feb. 2025), both has been updated again to Isaac Sim 4.5.0 and Isaac Lab 2.0.0. NVIDIA updates this toolchain approximately two times a year, and it's highly suggested to keep this extension updated.

TocabiLab, as of now, runs on Isaac Sim 4.2.0 and Isaac Lab 1.4.1.

A.3 Related Github Issues

As previously mentioned in the Limitation section, there are many existing bugs and missing features in the system that prevents the system from using CPU mode and run more than 96 robots in parallel. This is normal considering Isaac Lab is a relatively new system, and all of these issues are reported to NVIDIA during development and the community and the project team are actively working on address them. Users are advised to keep an eye on these issues.

IsaacLab Issue #1725 Additional PhysX View used for FT Sensor, causing the scene to stall under high load.

IsaacLab Issue #1390 Contact sensor does not work in CPU mode.

IsaacLab Issue #1335 `max_episode_length` causing crash when truncated.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Erin S. Catto. Iterative dynamics with temporal coherence. 2005.

[3] Robin Deits and Russ Tedrake. Footstep planning on uneven terrain with mixed-integer convex optimization. In *2014 IEEE-RAS International Conference on Humanoid Robots*, pages 279–286, 2014.

[4] Alberto Del Rio, David Jimenez, and Javier Serrano. Comparative analysis of a3c and ppo algorithms in reinforcement learning: A survey on general environments. *IEEE Access*, 2024.

[5] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International conference on machine learning*, pages 1329–1338. PMLR, 2016.

[6] Tom Erez, Yuval Tassa, and Emanuel Todorov. Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In *2015 IEEE*

International Conference on Robotics and Automation (ICRA), pages 4397–4404, 2015.

[7] Kenny Erleben. Numerical methods for linear complementarity problems in physics-based animation. In *Acm Siggraph 2013 Courses*, pages 1–42. 2013.

[8] Gilbert Feng, Hongbo Zhang, Zhongyu Li, Xue Bin Peng, Bhuvan Basireddy, Linzhu Yue, Zhitao Song, Lizhi Yang, Yunhui Liu, Koushil Sreenath, and Sergey Levine. Genloco: Generalized locomotion controllers for quadrupedal robots, 2022.

[9] Mario S Holubar and Marco A Wiering. Continuous-action reinforcement learning for playing racing games: Comparing spg to ppo. *arXiv preprint arXiv:2001.05270*, 2020.

[10] Jamshed Iqbal. Modern control laws for an articulated robotic arm. *Engineering, Technology & Applied Science Research*, 9(2):4057–4061, 2019.

[11] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[12] Donghyeon Kim, Glen Berseth, Mathew Schwartz, and Jaeheung Park. Torque-based deep reinforcement learning for task-and-robot agnostic learning on bipedal robots using sim-to-real transfer. *IEEE Robotics and Automation Letters*, 8(10):6251–6258, October 2023.

[13] LeggedRobotics. Legged robotics rsl_rl. `https://github.com/leggedrobotics/rsl_rl`. Accessed: 2025-01-24.

[14] Zhongyu Li, Xuxin Cheng, Xue Bin Peng, Pieter Abbeel, Sergey Levine, Glen Berseth, and Koushil Sreenath. Reinforcement learning for robust parameterized locomotion control of bipedal robots. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2811–2817, 2021.

[15] TP Lillicrap. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[16] Miles Macklin. Warp: A high-performance python framework for gpu simulation and graphics. `https://github.com/nvidia/warp`, March 2022. NVIDIA GPU Technology Conference (GTC).

[17] Mayank Mittal, Calvin Yu, Qinxi Yu, Jingzhou Liu, Nikita Rudin, David Hoeller, Jia Lin Yuan, Ritvik Singh, Yunrong Guo, Hammad Mazhar, Ajay Mandlekar, Buck Babich, Gavriel State, Marco Hutter, and Animesh Garg. Orbit: A unified simulation framework for interactive robot learning environments. *IEEE Robotics and Automation Letters*, 8(6):3740–3747, June 2023.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[19] Nvidia. Nvidia isaac sim. `https://developer.nvidia.com/isaac/sim`. Accessed: 2025-01-23.

[20] NVIDIA. PxArticulationJointReducedCoordinate 2014; physx 5.4.0 documentation — nvidia-omniverse.github.io. `https://nvidia-omniverse.github.io/PhysX/physx/5.4.0/_api_build/class_px_articulation_joint_reduced_coordinate.html`. [Accessed 04-02-2025].

[21] NVIDIA Omniverse. Nvidia physx. `https://github.com/NVIDIA-Omniverse/PhysX`. Accessed: 2025-01-29.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[23] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Trans. Graph.*, 37(4):143:1–143:14, July 2018.

[24] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 3803–3810. IEEE, 2018.

[25] Xue Bin Peng, Glen Berseth, and Michiel van de Panne. Dynamic terrain traversal skills using reinforcement learning. *ACM Trans. Graph.*, 34(4), July 2015.

[26] Xue Bin Peng and Michiel van de Panne. Learning locomotion skills using deeprl: does the choice of action space matter? In *Proceedings of the ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, SCA '17, New York, NY, USA, 2017. Association for Computing Machinery.

[27] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable baselines3. `https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html`, 2021. Accessed: 2025-01-31.

[28] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[29] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[30] Mathew Schwartz, Jaehoon Sim, Junewhee Ahn, Soonwook Hwang, Yisoo Lee, and Jaeheung Park. Design of the humanoid robot tocabi. In *2022 IEEE-RAS 21st International Conference on Humanoid Robots (Humanoids)*, pages 322–329, 2022.

[31] Morten Silcowitz, Sarah Niebe, and Kenny Erleben. Projected gauss–seidel subspace minimization method for interactive rigid body dynamics - improving animation quality using a projected gauss–seidel subspace minimization method. In *Proceedings of the International Conference on Computer Graphics Theory and Applications - Volume 1: GRAPP, (VISIGRAPP 2010)*, pages 38–45. INSTICC, SciTePress, 2010.

[32] Richard S Sutton, Andrew G Barto, et al. Reinforcement learning. *Journal of Cognitive Neuroscience*, 11(1):126–134, 1999.

[33] Christopher Ryan Thompson, Rajasekhar Reddy Talla, JCS Gummadi, and A Kamisetty. Reinforcement learning techniques for autonomous robotics. *Asian Journal of Applied Science and Engineering*, 8(1):85–96, 2019.

[34] Emanuel Todorov. Convex and analytically-invertible dynamics with contacts and constraints: Theory and implementation in mujoco. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6054–6061, 2014.

[35] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[36] Claudio Urrea, John Kern, and Víctor Torres. Design, simulation, and comparison of advanced control strategies for a 3-degree-of-freedom robot. *Applied Sciences*, 14(23), 2024.

[37] Xiao-Ping Wang, Carlos J. Garcia-Cervera, and Weinan E. A gauss–seidel projection method for micromagnetics simulations. *Journal of Computational Physics*, 171(1):357–372, 2001.

[38] Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. Dd-ppo: Learning near-perfect point-goal navigators from 2.5 billion frames, 2020.

[39] Alexander W. Winkler, C. Dario Bellicoso, Marco Hutter, and Jonas Buchli. Gait and trajectory optimization for legged systems through phase-based end-effector parameterization. *IEEE Robotics and Automation Letters*, 3(3):1560–1567, 2018.

[40] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative multi-agent games. *Advances in Neural Information Processing Systems*, 35:24611–24624, 2022.

[41] Wenshuai Zhao, Jorge Peña Queralta, and Tomi Westerlund. Sim-to-real transfer in deep reinforcement learning for robotics: a survey. In *2020 IEEE symposium series on computational intelligence (SSCI)*, pages 737–744. IEEE, 2020.