



Advanced Control for Robotics

Final Project

Siyuan WANG

Han BAO

instructor: Prof. Wei ZHANG

Department of Electric and Electronic Engineering

Southern university of science and technology

January 2025

Abstract

In this project, we employed deep reinforcement learning methodology to train a locomotion controller for Limx TRON1-A bipedal robot. The trained policy enables the robot to achieve stable walking on rough terrain conditions, which shows robustness against external disturbances. Furthermore, the trained policy was deployed in the MuJoCo simulator for sim-to-sim validation. Additionally, we developed a control policy for single-leg hopping, which demonstrated promising performance in the Isaac Gym simulator. A sim-to-sim test video is available here: [bilibili_video_link](#). We also plan to make our code open-sourced after the code documentation was well implemented. Scheduled open-source link: [github_repo_link](#)

Keywords: Bipedal Robot, Robotic Control, Deep Reinforcement Learning

Contents

Abstract	ii
1 Codebase Structure	iv
1.1 Simulation Setup	iv
1.1.1 RL Environment setup	iv
1.1.2 Environment forward step	v
1.1.3 Termination	vi
1.2 PPO algorithm	vii
1.2.1 Overview of PPO	vii
1.2.2 Core Components of PPO in rsrl	vii
1.2.3 Collecting Experience: Rollouts	viii
1.2.4 Advantage Estimation	viii
1.2.5 Clipped Surrogate Objective	ix
1.2.6 Training the PPO Agent	ix
1.2.7 Hyperparameters and Tuning	ix
1.2.8 Conclusion	x
1.3 Agent	x
1.3.1 Compute Observations	x
1.3.2 Compute Rewards	xii
1.4 MuJoCo Sim-to-sim Code	xii
2 Details Techniques for Enhancing Robustness	xviii
2.1 Reward Terms and Weights	xviii
2.2 Domain Randomization	xix
3 Special Tasks Training on Flat Ground	xx
3.1 Single Leg Hopping	xx
3.2 Kneel Down	xx

1 Codebase Structure

Our codebase is built on top of *legged_gym* [7] [2], and is composed of four parts: the reinforcement learning environment in Isaac Gym, the PPO Actor-Critic reinforcement learning algorithm implemented by *rsl_rl* [3], and a self-implemented sim-to-sim policy validation by us.

1.1 Simulation Setup

In this project, the RL environment is implemented within Isaac Gym. Isaac Gym is NVIDIA’s prototype physics simulation environment for robotic reinforcement learning research. Isaac Gym allows developers to experiment with massive parallel GPU based multi-rigidbody simulation, which makes simulation and RL pipelines can both be run on GPU, accelerating the training pipeline.

This all GPU pipeline feature permits thousands of environments to be simultaneously simulated on a single GPU, allowing experiments that previously may have required an entire data center to run on a single workstation.

Robotic System Lab at ETH Zurich released their accompanying code of ‘Learning to walk in minutes using massively parallel deep reinforcement learning’, which is called Legged Gym, leveraging Isaac Gym to train thousands of legged robots in parallel and achieved outstanding sim-to-real performance [7]. The following figure shows a demonstration of an Anymal-C robot legged gym.

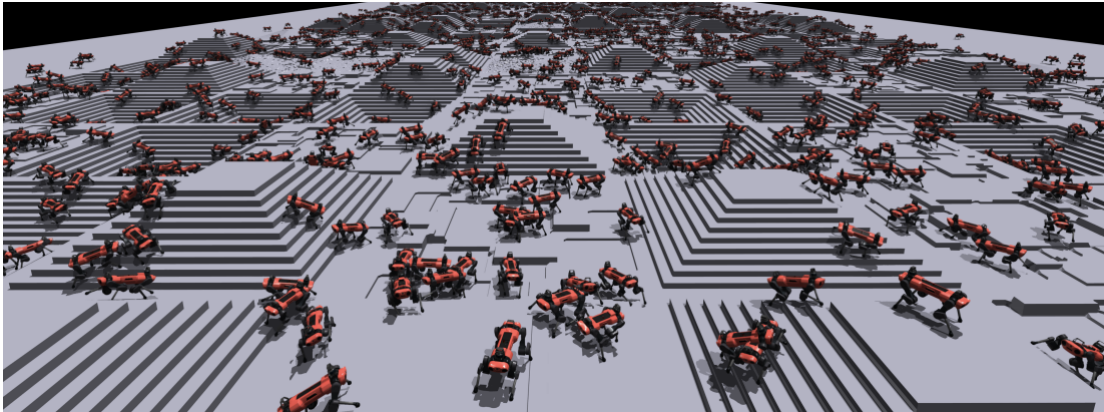


Figure 1: Thousands of legged robots learning to walk in simulation.

1.1.1 RL Environment setup

The RL environment utilized several interfaces provided by Isaac Gym, to set up the interaction between the agent and environment, allowing the retrieval of the robot’s state and the writing of new states to the robot. The RL environment setup is wrapped within the `PointFoot` class.

For the massive parallel robot environment, it needs to be initialized first, which is implemented in the `__init__()` function of `PointFoot`. The `__init__()` function loads configurations, initializes the device, allocates training buffers, prepares reward functions, etc.

The training process is inspired by a game curriculum. In the beginning, robots are evenly placed in terrains of different difficulty levels. As long as the robot passed the episode without termination, the robot will be placed in a terrain with higher difficulty

level, vice versa for robots failed to pass the episode. In general, if we train the controller policy from scratch, all robots should fall quickly to the easiest terrains, due to the policy is just randomized, and learned no useful locomotion skills. A picture of this phenomenon is shown below.

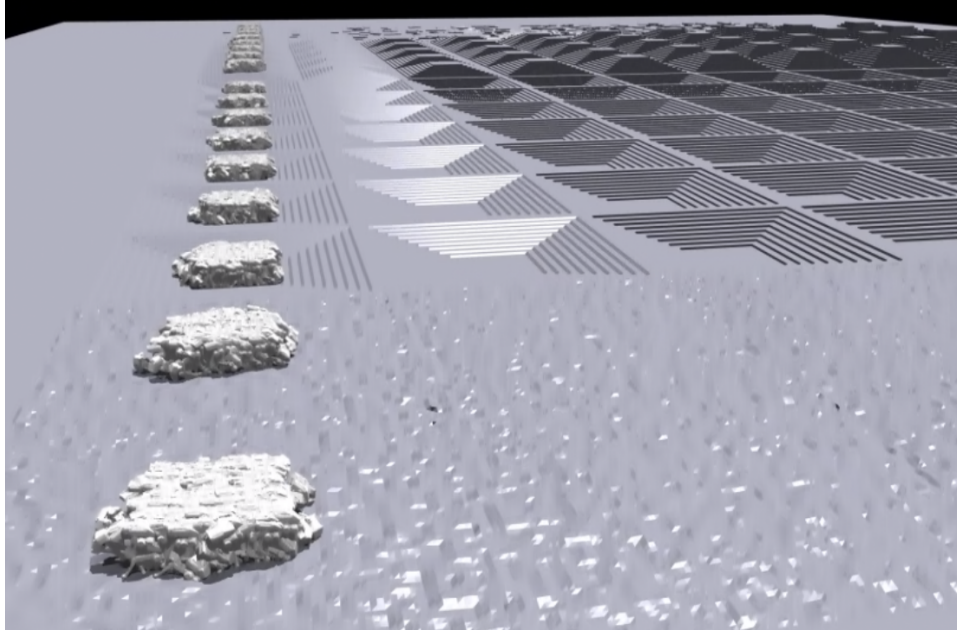


Figure 2: Robots failed to traverse difficult terrains at first if trained from scratch

1.1.2 Environment forward step

After the environment is initialized, it will call `step()` function to update. The `step()` function is responsible for applying actions, running the physics simulation, and returning updated environment states. Besides, it will check which environments need to be reset.

```

1 def step(self, actions):
2     # Apply actions, simulate, call post_physics_step()
3     clip_actions = self.cfg.normalization.clip_actions
4     self.actions = torch.clip(actions, -clip_actions, clip_actions).to(self.device)
5     # Step physics and render each frame
6     self.render()
7     for _ in range(self.cfg.control.decimation):
8         self.torques = self._compute_torques(self.actions).view(self.torques.shape)
9         self.gym.set_dof_actuation_force_tensor(self.sim, gymtorch.unwrap_tensor(self.torques))
10        self.gym.simulate(self.sim)
11        self.gym.fetch_results(self.sim, True)
12        self.gym.refresh_dof_state_tensor(self.sim)
13    self.post_physics_step()
14
15    # Return clipped obs, clipped states (None), rewards, dones, and infos
16    clip_obs = self.cfg.normalization.clip_observations
17    self.proprioceptive_obs_buf = torch.clip(self.proprioceptive_obs_buf, -clip_obs, clip_obs)
18    if self.privileged_obs_buf is not None:
19        self.privileged_obs_buf = torch.clip(self.privileged_obs_buf, -clip_obs, clip_obs)
20    return self.proprioceptive_obs_buf, self.privileged_obs_buf, self.rew_buf, self.reset_buf,
        self.extras

```

Below is a detailed breakdown of its functionality:

- Action Clipping and Device Transfer:

-
- The `actions` tensor, with shape `(num_envs, num_actions_per)`, represents the actions for each environment.
 - The actions are clipped to the range `[-clip_actions, clip_actions]` using `torch.clip`.
 - The clipped actions are then transferred to the correct device (either CPU or GPU) based on the configuration using `to(self.device)`.

- **Physics Step and Rendering:**

- The function first calls `self.render()` to handle rendering if enabled.
- Then, it performs simulation steps for each action using `cfg.control.decimation`, iterating the following steps:
 - * Compute the torques corresponding to the actions: `self._compute_torques(self.actions)`.
 - * Apply the torques to the simulation using: `self.gym.set_dof_actuation_force_tensor(self.sim, gymtorch.unwrap_tensor(self.torques))`.
 - * Advance the simulation using: `self.gym.simulate(self.sim)`.
 - * Fetch results after simulation: `self.gym.fetch_results(self.sim, True)`.
 - * Refresh the degrees of freedom (DOF) states with: `self.gym.refresh_dof_state_tensor(self.sim)`.

- **Post-Physics Step:**

- After the physics simulation, the function calls `self.post_physics_step()`, which likely handles further updates or actions based on the current simulation state.

- **Clipping Observations:**

- The proprioceptive observations (`self.proprioceptive_obs_buf`) and privileged observations (`self.privileged_obs_buf`) are clipped to the range `[-clip_obs, clip_obs]`.

- **Return Values:**

- The function returns the following:
 - * Clipped proprioceptive observations: `self.proprioceptive_obs_buf`.
 - * Clipped privileged observations (if available): `self.privileged_obs_buf`.
 - * Rewards for each environment: `self.rew_buf`.
 - * Reset flags for each environment: `self.reset_buf`.
 - * Additional information: `self.extras`.

1.1.3 Termination

The termination check should be properly set, in case the robot will learn unexpected and strange behaviors. In our experiments, at first we only add the robot terminates if the base link endure forces, and only added survival reward, not added termination penalization reward. The policy unexpectedly learned the "kneeled-down" behavior by chance.

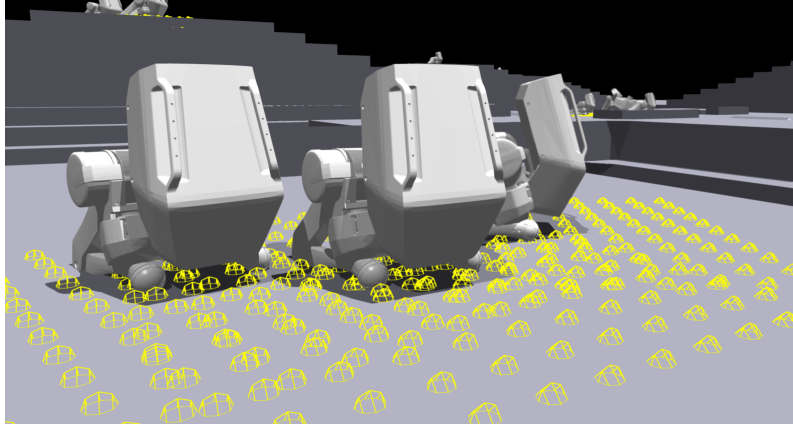


Figure 3: Unexpected "kneeled down" behavior when termination condition is not properly set

After that, we employed both survival and termination reward, as well as set that the robot episode should terminate as long as its abad link and base link endure forces. In such way, the kneeled down behavior never occurs.

1.2 PPO algorithm

In this section, we describe the implementation of the Proximal Policy Optimization (PPO) algorithm provided by the `rsrl` library. The following steps break down the core components of PPO along with the code snippets that implement them.

1.2.1 Overview of PPO

PPO is a policy gradient algorithm that aims to optimize a policy by directly adjusting the policy network to maximize the expected cumulative reward. Unlike traditional Q-learning, PPO focuses on directly optimizing the policy using a clipped surrogate objective. The core idea is to restrict large policy updates by clipping the objective function, making it both stable and efficient.

1.2.2 Core Components of PPO in rsrl

The PPO algorithm in `rsrl` consists of the following key components:

- **Policy Network:** The policy network outputs the probability distribution over actions for discrete action spaces or the mean and variance for continuous action spaces.
- **Value Network:** The value network estimates the value of each state, and this value is used to calculate the advantage function.

In the following code, we initialize the PPO agent with a policy network, value network, optimizer, and key hyperparameters such as the learning rate, discount factor, and epsilon for clipping:

```

1 class PPOAgent:
2     def __init__(self, env, policy_net, value_net, config):
3         self.env = env
4         self.policy_net = policy_net
5         self.value_net = value_net
6         self.optimizer = torch.optim.Adam(self.policy_net.parameters(), lr=config['lr'])
7         self.gamma = config['gamma'] # Discount factor for future rewards
8         self.epsilon = config['epsilon'] # Clipping range for PPO

```

```
self.entropy_coef = config['entropy_coef'] # Coefficient for entropy regularization
```

In this setup, `policy_net` and `value_net` are neural networks that represent the policy and value function, respectively. The `optimizer` is used to optimize the policy and value networks.

1.2.3 Collecting Experience: Rollouts

PPO requires rollouts, which are trajectories collected by interacting with the environment. Each trajectory consists of states, actions, rewards, and advantages. These rollouts are used to calculate the advantage function and update the policy.

In the following code snippet, the agent collects rollouts by interacting with the environment:

```
1 def collect_rollouts(self, num_steps):
2     states = []
3     actions = []
4     rewards = []
5     values = []
6     for _ in range(num_steps):
7         state = self.env.reset()
8         done = False
9         while not done:
10             action = self.policy_net(state)
11             next_state, reward, done, _ = self.env.step(action)
12
13             states.append(state)
14             actions.append(action)
15             rewards.append(reward)
16             values.append(self.value_net(state))
17             state = next_state
18     return states, actions, rewards, values
```

Here, the `collect_rollouts` function generates a batch of rollouts by interacting with the environment. For each step, it collects the current state, action, reward, and value estimation, and stores them for later use.

1.2.4 Advantage Estimation

The next step in PPO is to calculate the advantage, which quantifies how much better (or worse) an action performed compared to the average behavior. In PPO, the advantage is typically calculated using Generalized Advantage Estimation (GAE).

The following code shows how the advantage is computed:

```
1 def compute_advantages(self, rewards, values, gamma, lambda_):
2     advantages = []
3     last_advantage = 0
4     for t in reversed(range(len(rewards))):
5         delta = rewards[t] + gamma * values[t+1] - values[t]
6         last_advantage = delta + gamma * lambda_ * last_advantage
7         advantages.insert(0, last_advantage)
8     return advantages
```

Here, the `compute_advantages` function computes the advantage using GAE. It iterates over the trajectory in reverse order, calculating the temporal difference (TD) error, and then computes the advantage recursively.

1.2.5 Clipped Surrogate Objective

The key feature of PPO is its clipped surrogate objective, which helps to prevent overly large updates to the policy. The objective function is clipped such that the ratio between the new and old policy probabilities is restricted within a specified range.

The following code snippet shows how the clipped objective is computed in PPO:

```
1 def compute_ppo_loss(self, states, actions, old_probs, advantages, epsilon):
2     new_probs = self.policy_net(states)
3     ratio = new_probs / old_probs
4     clip_ratio = torch.clamp(ratio, 1 - epsilon, 1 + epsilon)
5
6     # Surrogate loss
7     loss = -torch.min(ratio * advantages, clip_ratio * advantages).mean()
8
9     # Value function loss
10    value_loss = ((self.value_net(states) - rewards) ** 2).mean()
11
12    # Entropy loss (to encourage exploration)
13    entropy_loss = -torch.mean(new_probs * torch.log(new_probs))
14
15    return loss + 0.5 * value_loss - 0.01 * entropy_loss
```

In this function, `compute_ppo_loss` calculates the PPO loss, which includes: - The clipped surrogate loss. - The value function loss, which minimizes the error between predicted values and actual rewards. - The entropy loss, which encourages exploration by penalizing deterministic policies.

1.2.6 Training the PPO Agent

The PPO agent is trained by iterating through multiple episodes, collecting rollouts, calculating advantages, and updating the policy using the computed PPO loss. The agent is updated in batches over several epochs.

Here's the training loop for the PPO agent:

```
1 def train(self, num_epochs, batch_size, num_steps):
2     for epoch in range(num_epochs):
3         states, actions, rewards, values = self.collect_rollouts(num_steps)
4         advantages = self.compute_advantages(rewards, values, self.gamma, 0.95)
5
6         # Perform updates in mini-batches
7         for _ in range(batch_size):
8             old_probs = self.policy_net(states)
9             loss = self.compute_ppo_loss(states, actions, old_probs, advantages, self.epsilon)
10
11             self.optimizer.zero_grad()
12             loss.backward()
13             self.optimizer.step()
```

In this loop: - Rollouts are collected by calling `collect_rollouts`. - The advantages are calculated using `compute_advantages`. - The PPO loss is computed and used to update the policy and value networks using backpropagation.

1.2.7 Hyperparameters and Tuning

In the `rsrl` library, several hyperparameters control the behavior of the PPO algorithm:

- **Learning Rate:** Determines the step size for updating the network weights.

- **Discount Factor (γ):** Controls the weight of future rewards in the value function and advantage estimation.
- **Clipping Range (ϵ):** Specifies the allowable range for the probability ratio in the clipped objective.
- **Batch Size:** Defines how many rollouts are used in each update.
- **GAE Lambda (λ):** Balances bias and variance when calculating the advantage using Generalized Advantage Estimation.

Adjusting these hyperparameters allows you to fine-tune the agent's performance and stability during training.

1.2.8 Conclusion

The PPO implementation in the `rsrl` library provides a robust and scalable solution for training reinforcement learning agents. By using a clipped objective function and regularization techniques such as entropy regularization, PPO offers a balance between exploration and exploitation. Its ability to efficiently update policies while maintaining stability makes it a powerful tool for tackling complex reinforcement learning problems.

1.3 Agent

The agent follows the MDP paradigm, where each step includes obtaining observations, calculating rewards, checking if a reset is needed, and other steps.

1.3.1 Compute Observations

The following function, `compute_observations()` compute the observations of robots, and add noise to improve generalization ability.

```

1  def compute_observations(self):
2      """Computes observations
3      """
4      self.compute Proprioceptive_observations()
5      self.compute_privileged_observations()
6
7      self._add_noise_to_obs()
```

The `compute Proprioceptive_observations` function computes the proprioceptive observations for the agent. Proprioceptive observations are sensory inputs related to the agent's own state, such as joint positions, velocities, and potentially height information. These observations are used by the agent to understand its own position and behavior in the environment.

```

1  def compute Proprioceptive_observations(self):
2      self._compose Proprioceptive_obs_buf_no_height_measure()
3      if self.cfg.terrain.measure_heights_actor:
4          self Proprioceptive_obs_buf = self._add_height_measure_to_buf(self.
5              Proprioceptive_obs_buf)
6      if self Proprioceptive_obs_buf.shape[1] != self.num_obs:
7          raise RuntimeError(
8              f"obs_buf_size_{self Proprioceptive_obs_buf.shape[1]}_does_not_match_num_obs_{self.
9                  num_obs}")
```

- **Compose initial proprioceptive observations:**

The function first calls `_compose_proprioceptive_obs_buf_no_height_measure()` to generate the initial proprioceptive observations buffer (`proprioceptive_obs_buf`) without including height information.

- **Add height measurement if required:**

The function then checks whether height measurements for the actor are required, based on the configuration (`self.cfg.terrain.measure_heights_actor`). If true, it calls `_add_height_measure_to_buf()` to append the height data to the `proprioceptive_obs_buf`.

- **Validate the observations' shape:**

Finally, the function verifies if the shape of the `proprioceptive_obs_buf` matches the expected number of observations (`num_obs`). If the shape does not match, a `RuntimeError` is raised to signal the mismatch and prevent any inconsistency in the buffer.

This function computes the "privileged observations", which are additional information that might be useful to the agent during training but are not necessarily available in the real environment. Privileged observations can enhance training efficiency and help the agent perform better during training.

```

1 def compute_privileged_observations(self):
2     if self.num_privileged_obs is not None:
3         self._compose_privileged_obs_buf_no_height_measure()
4         # add perceptive inputs if not blind
5         if self.cfg.terrain.measure_heights_critic:
6             self.privileged_obs_buf = self._add_height_measure_to_buf(self.privileged_obs_buf)
7         if self.privileged_obs_buf.shape[1] != self.num_privileged_obs:
8             raise RuntimeError(
9                 f"privileged_obs_buf_size_{self.privileged_obs_buf.shape[1]} does not match_{num_privileged_obs}_{self.num_privileged_obs}")

```

- **Check if privileged observations are configured:**

The function first checks if the number of privileged observations (`num_privileged_obs`) is configured. If not, the function will not proceed with the following steps.

- **Compose initial privileged observations:**

If the configuration exists, the function calls `_compose_privileged_obs_buf_no_height_measure()` to generate the initial `privileged_obs_buf`. This method likely generates the observations without including height information.

- **Add height information if required:**

Then, the function checks the configuration to see if height information should be included in the privileged observations (`self.cfg.terrain.measure_heights_critic`). If true, it calls `_add_height_measure_to_buf()` to add the height data to the `privileged_obs_buf`.

- **Validate the shape of privileged observations:**

Finally, the function checks if the shape of `privileged_obs_buf` matches the expected number of privileged observations (`num_privileged_obs`). If there is a mismatch, a `RuntimeError` is raised to ensure the dimensions match.

1.3.2 Compute Rewards

The `compute_reward` function computes the rewards for the agent during the simulation. This function sums the rewards from all the individual reward functions that have a non-zero scale, and adds them to the total reward. It also handles any additional reward logic, such as ensuring only positive rewards or adding termination rewards.

```
1 def compute_reward(self):
2     """Compute rewards
3     Calls each reward function which had a non-zero scale (processed in self.
4     _prepare_reward_function())
5     Adds each term to the episode sums and to the total reward
6     """
7     self.rew_buf[:] = 0.
8     for i in range(len(self.reward_functions)):
9         name = self.reward_names[i]
10        rew = self.reward_functions[i]() * self.reward_scales[name]
11        self.rew_buf += rew
12        self.episode_sums[name] += rew
13    if self.cfg.rewards.only_positive_rewards:
14        self.rew_buf[:] = torch.clip(self.rew_buf[:], min=0.)
15    # add termination reward after clipping
16    if "termination" in self.reward_scales:
17        rew = self._reward_termination() * self.reward_scales["termination"]
18        self.rew_buf += rew
19        self.episode_sums["termination"] += rew
```

- **Reset the reward buffer:** The function starts by setting all the values in the reward buffer (`rew_buf`) to zero. This ensures that the rewards are calculated from scratch at each step.
- **Iterate over the reward functions:** The function then loops over the reward functions stored in `reward_functions`. For each function, it computes the reward by multiplying the result of the function by its corresponding scale factor (found in `reward_scales`). The calculated reward is added to both the total reward buffer (`rew_buf`) and the individual episode sum for that reward term (`episode_sums[name]`).
- **Clipping the reward:** If the configuration specifies that only positive rewards should be allowed (`self.cfg.rewards.only_positive_rewards`), the reward buffer is clipped so that any negative rewards are set to zero.
- **Add termination reward:** After the rewards are computed and possibly clipped, if a termination reward scale exists (`"termination" in self.reward_scales`), the function calls `_reward_termination()` to compute the termination reward. This reward is then scaled and added to both the reward buffer and the episode sums.

The detailed reward terms and corresponding weights will be presented in 2.1

1.4 MuJoCo Sim-to-sim Code

After training in Isaac Gym, sim-to-sim validation is needed to test the generalization of the policy. Here we employ mujoco as the sim-to-sim simulation engine.

This section describes the implementation of the `MuJoCoSim` class, which is responsible for setting up and running a MuJoCo-based simulation for a robot. The class

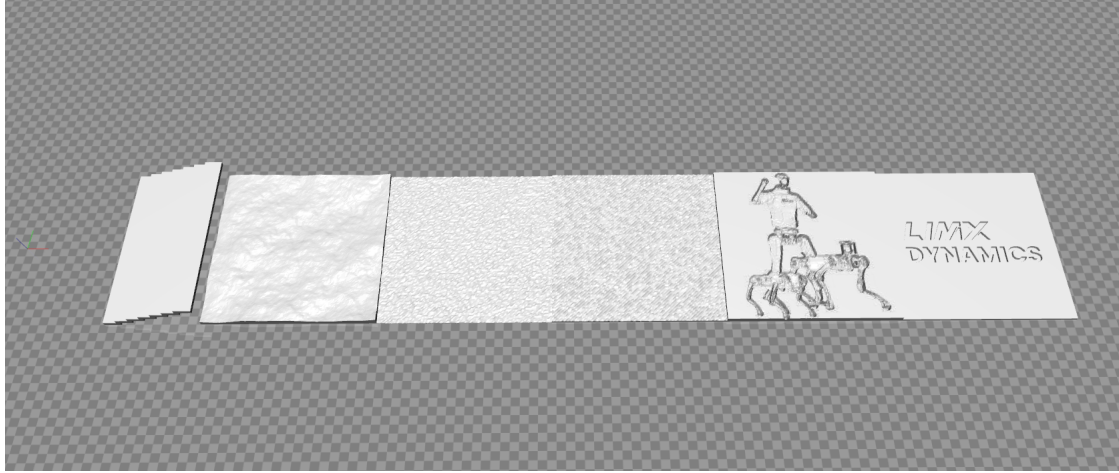


Figure 4: MuJoCo environment we created for validate sim2sim performance

integrates a control policy, simulation parameters, and provides interfaces to interact with the simulation through key commands.

1. Constructor: `__init__`

```

1 def __init__(self):
2     """Class constructor to set up simulation configuration and control policy."""
3     # Load the policy configuration and model
4     config_file = os.path.join(CURRENT_DIR, "policy/PF_TRON1A/config.yaml")
5     self.cfg = utils.load_config(config_file)
6     policy_file = os.path.join(CURRENT_DIR, "policy/PF_TRON1A/policy.pt")
7     self.policy = torch.jit.load(policy_file)
8
9     # Load the MuJoCo model
10    self.model = mujoco.MjModel.from_xml_path( os.path.join(CURRENT_DIR, "robot_description/
11        PF_TRON1A/xml/robot.xml"))
12    self.data = mujoco.MjData(self.model)
13
14    # Control and Simulation parameters
15    self.init_base_height = self.cfg["init_base_height"]
16    self.num_joint = self.cfg["num_action"]
17    self.Kp = self.cfg["kp"] * np.ones(self.num_joint)
18    self.Kd = self.cfg["kd"] * np.ones(self.num_joint)
19    self.joint_pos_default = np.array(list(self.cfg["default_joint_angle"].values()))
20    self.torque_limit = self.cfg["torque_limit"] * np.ones(self.num_joint)
21    self.obs_scales = self.cfg["obs_scales"]
22    self.user_cmd_scale = np.array( [
23        self.obs_scales["lin_vel"],
24        self.obs_scales["lin_vel"],
25        self.obs_scales["ang_vel"] ])
26    self.decimation = self.cfg["decimation"]
27    self.action_scale = self.cfg["action_scale"]
28
29    # Key callback function for mujoco viewer
30    self.paused = False
31    def viewer_key_callback(keycode):
32        if chr(keycode) == "p":
33            self.paused = not self.paused
34
35    # Reset Simulation and Start the simulation viewer
36    self.viewer = mujoco.viewer.launch_passive( self.model, self.data, key_callback=
37        viewer_key_callback, show_left_ui=False, show_right_ui=False)

```

```
self.reset_sim()
```

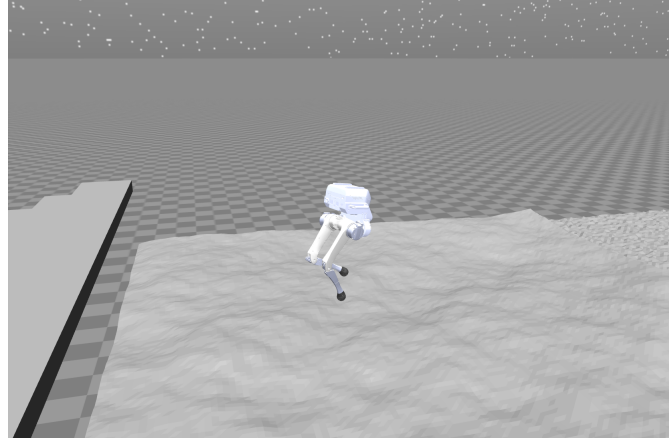


Figure 5: TRON1 walking on rough terrain in MuJoCo

In the constructor, the `MuJoCoSim` class does the following:

- Loads the robot's control policy from a PyTorch model file (`policy.pt`) and configuration (`config.yaml`).
- Initializes the MuJoCo model and data structures using the robot's XML description.
- Sets control and simulation parameters like joint stiffness (`Kp`), damping (`Kd`), joint positions, torque limits, and scaling factors for observations and actions.
- Starts the MuJoCo viewer and attaches a keyboard callback function to pause or resume the simulation when the spacebar is pressed.

2. Method: `reset_sim`

```

1 def reset_sim(self):
2     """Reset Simulation Configuration"""
3     # Reset mujoco data structs
4     mujoco.mj_resetData(self.model, self.data)
5     # Reset robot qpos
6     self.data.qpos[0] = 0.0
7     self.data.qpos[1] = 0.0
8     self.data.qpos[2] = self.init_base_height
9     self.data.qpos[3] = 0.0
10    self.data.qpos[4] = 0.0
11    self.data.qpos[5] = 0.0
12    self.data.qpos[6] = 1.0
13    for i in range(self.num_joint):
14        self.data.qpos[i + 7] = self.joint_pos_default[i]
15    mujoco.mj_step(self.model, self.data)
16    # Reset User commands
17    self.commands = np.zeros((self.cfg['num_user_cmd'], ))
18    self.commands[0] = 0.0 # x velocity
19    self.commands[1] = 0.0 # y velocity
20    self.commands[2] = 0.0 # yaw angular velocity
21    # Reset Policy
22    self.action = np.zeros(self.num_joint) # Policy action: reference joint position
23    self.obs = np.zeros(self.cfg["num_obs"])

```

```
24 self.target_q = np.zeros(self.num_joint) # Joint torque
25 self.iter_ = 0
```

The `reset_sim` method is responsible for resetting the simulation environment. It resets:

- The MuJoCo data structure (`mj_resetData`).
- The robot's position and joint angles to their default values.
- The user commands (`self.commands`), which represent velocity commands for the robot.
- The policy's action values and target joint positions.

3. Method: `get_joint_state`

```
1 def get_joint_state(self):
2     """Retrieve the joint position and velocity states."""
3     q_ = self.data.qpos
4     dq_ = self.data.qvel
5     return q_[7:], dq_[6:]
```

The `get_joint_state` method retrieves the joint positions and velocities from the MuJoCo simulation data (`self.data.qpos` and `self.data.qvel`). It returns the joint position starting from the 7th index (as the first 7 values represent the base and orientation of the robot) and the joint velocity from the 6th index.

4. Method: `compute_obs`

```
1 def compute_obs(self):
2     """Calculate and return the observed states from the policy input."""
3     dof_pos, dof_vel = self.get_joint_state()
4     base_ang_vel, projected_gravity = self.get_base_state()
5     self.obs = np.concatenate([
6         base_ang_vel * self.obs_scales['ang_vel'], # Scaled base angular velocity
7         projected_gravity, # Projected gravity vector in body frame
8         dof_pos * self.obs_scales['dof_pos'], # Scaled joint positions
9         dof_vel * self.obs_scales['dof_vel'], # Scaled joint velocities
10        self.action, # Last actions taken by the robot
11        self.user_cmd_scale * self.commands # Scaled velocity commands from user input
12    ])
```

The `compute_obs` method calculates the observations for the robot. It gathers data from multiple sources:

- The robot's angular velocity and projected gravity.
- Joint positions and velocities, all of which are scaled using predefined scaling factors from the configuration.
- The actions the robot took in the previous step.
- User velocity commands that control the robot.

These values are concatenated to form the complete observation vector (`self.obs`).

5. Method: `compute_actions`

```
1 def compute_actions(self):
2     """Computes the actions based on the current observations using the policy session."""
3     self.compute_obs()
4     input_tensor = np.concatenate([self.obs], axis=0)
5     input_tensor = input_tensor.astype(np.float32)
6     self.action = self.policy(torch.tensor(input_tensor)).detach().numpy()
7     self.action[:] = np.clip(self.action, - self.cfg["clip_action"], self.cfg["clip_action"])
8     self.target_q[:] = self.action_scale * self.action
```

The `compute_actions` method computes the actions for the robot using the control policy. It:

- Computes the current observations (`self.compute_obs`).
- Feeds the observation data into the policy model to compute the next action.
- Clips the resulting action values to be within a specified range (`self.cfg["clip_action"]`).
- Scales the actions as required and stores them in `self.target_q`.

6. Method: `pd_control`

```
1 def pd_control(self, target_q):
2     """Use PD to find target joint torques"""
3     dof_pos, dof_vel = self.get_joint_state()
4     return (target_q + self.joint_pos_default - dof_pos) * self.Kp - self.Kd * dof_vel
```

The `pd_control` method computes the joint torques using a Proportional-Derivative (PD) controller. The controller applies a correction based on the difference between the desired joint positions (`target_q`) and the current joint positions, as well as the joint velocities.

7. Main Loop: `run`

```
1 def run(self):
2     """Main loop of simulation"""
3     listener = keyboard.Listener(on_press=self.keyboard_listener_on_press)
4     listener.start()
5     try:
6         while self.data.time < 100.0 and self.viewer.is_running():
7             if not self.paused:
8                 step_start = time.time()
9                 if self.iter_ % self.decimation == 0:
10                     self.compute_actions()
11                     joint_torque = self.pd_control(self.target_q) # Calc torques
12                     joint_torque = np.clip(joint_torque, - self.torque_limit, self.torque_limit) #
13                                     Clip torques
14                     self.data.ctrl = joint_torque
15                     if not self.paused:
16                         self.viewer.cam.lookat = self.data.body('base_Link').subtree_com
17                         mujoco.mj_step(self.model, self.data)
18                         if self.iter_ % 10 == 0:
19                             self.viewer.sync()
20                         if self.iter_ % 500 == 0:
```

```

20         print("CMD:", self.commands[0], "\n", self.commands[1], "\n", self.commands
21               [2])
22         print("Act:", self.data.qvel[0], "\n", self.data.qvel[1], "\n", self.data.
23               qvel[2])
24         self.iter_ += 1
25         time_until_next_step = self.model.opt.timestep - (time.time() - step_start)
26         if time_until_next_step > 0:
27             time.sleep(time_until_next_step)
28     except KeyboardInterrupt:
29         print("\nMuJoCoSim2SimProgram is Stopped\n")
30     finally:
31         listener.stop()

```

The `run` method contains the main simulation loop. It listens for keyboard input to control the robot's movement and updates the simulation at each time step:

- The loop runs while the simulation time is less than 100 seconds and the viewer is running.
- It computes the next action using `compute_actions`, then applies the PD controller to compute joint torques.
- The simulation is updated with `mujoco.mj_step`, and the camera view is adjusted to follow the robot.
- The loop prints information about the robot's commands and actions at regular intervals.

Conclusion

This file demonstrates how to integrate a control policy with a MuJoCo-based simulation environment. By managing robot states, actions, and keyboard controls, the class allows users to simulate and interact with the robot in real-time, using a model-based approach with Proportional-Derivative control for joint actuation.

2 Details Techniques for Enhancing Robustness

In this chapter, various techniques used in this project will be introduced.

2.1 Reward Terms and Weights

Reward Term	Equation	Weight
Lin. velocity tracking	$\exp(-\frac{\ v_{xy}^{cmd} - v_{xy}\ _2^2}{0.025})$	10.0
Ang. velocity tracking	$\exp(-\frac{(\omega_z^{cmd} - \omega_z)^2}{0.025})$	5.0
Lin. velocity (z)	v_z^2	-0.5
Ang. velocity (xy)	$\ \omega_{xy}\ _2^2$	-0.05
Pen. joint acc.	$\ddot{q}^2$	-2.5×10^{-7}
Pen. joint torque.	$\ \tau\ _2^2$	-2.5×10^{-5}
Joint torque limits	n_{limit}	-0.1
Joint torque limits	n_{limit}	-0.1
Action rate	$\ a_t - a_{t-1}\ _2^2$	-0.01
Base height	$(h_{des} - h)^2$	-1.0
Collision	$n_{collision}$	-1.0
Orientation (xy)	$- g_{xy} _2^2$	-5.0
Stand still	$\ v_{xy}\ _1 + \ \omega_z\ _1$ if cmd i 0.1	-2.0
No fly	$n_{single_foot_contact}$	1.0
Feet regulation	$r^{fr} = \sum_{feet} \ v_{xy}^{foot}\ _2^2 \exp\left(-\frac{p_z^{foot}}{0.025h_{des}}\right)$	-0.001
Feet slip	$\ v_{xy}^{contact_foot}\ _2^2$	-1.5
Feet distance	$\max\left(0, 0.1 - \ p_{xy}^{left} - p_{xy}^{right}\ _2\right)$	-100.0
Feet contact forces	$\ f_{feet}^{contact}\ _2^2$	-0.01
Feet air time	$n_{feet_air_time_below_min} + n_{feet_air_time_above_max}$	60.0
Feet unbalanced air time	$var(t_{feet})$	-300.0
Feet unbalanced height	$var(h_{feet})$	-60.0
Termination	$n_{termination}$	-150.0
Survival	$n_{survived}$	100.0

The above list shows the reward term used in our project. Among them, due to the uncertainty of Monte Carlo sampling, our final policy is trained after several resumes, so the weights of the reward terms are slightly tuned during multiple training process. Here, we just listed the "just fine" weights in the above.

Among the reward terms above, most of the terms are adopted from legged_gym. The regulation term for feet is borrowed from CTS [8], and the slip term for feet is borrowed from RMA [5]. In our practical training, this feet regulation term may cause the agents fail to walk around due to its v_{xy}^{foot} term, causing the agents stand in place, and it takes quite a long time to gain the ability to walk around. So in our experiment, this term was dynamically added and removed for fine tune.

The reward terms that start with feet, are responsible for regulation the gait behavior, which makes the agent learns expected gait transition just like the gait state machine in the traditional optimal control methodology (ie, Convex MPC [1], MPC+WBIC [4]).

Through our training experience, we are deeply aware that it is very complex and tedious to train a control strategy from scratch through the sparse reward functions, which requires a lot of manual parameter tuning process. Therefore, one way to solve this problem is to introduce proper prior into the training process, that is, to make the

strategy converge faster or to add an artificially specified reference motion pattern for mimic (ie, imitation learning).

2.2 Domain Randomization

For domain randomization, we kept randomization code from `legged_gym`. The configurations are shown below.

```
1 class domain_rand:
2     randomize_friction = True
3     friction_range = [0.0, 1.6]
4     randomize_base_mass = True
5     added_mass_range = [-1.0, 2.0]
6     randomize_base_com = True
7     rand_com_vec = [0.05, 0.05, 0.05]
8     push_robots = True
9     push_interval_s = 7
10    max_push_vel_xy = 1.0
```

In our early experiment, we tried to also randomize coefficient of restitution (contact parameter in simulation, which measures the decay of relative speed after two rigid body collide). However, the policy turned to be hard to converge, even after 10,000 iterations, so that we finally remove the randomization configuration for the coefficient of restitution.

3 Special Tasks Training on Flat Ground

3.1 Single Leg Hopping

Inspired by Raibert Hopping Machine [6], we designed a single leg standing tasking RL environment for Limx bipedal robot TRON1.

To achieve this task, we need to re-implement task-related reward terms to regulate the agent’s behavior. We desire the left leg as the supporting leg, and right leg remains still. We kept the gait related reward terms in rough terrain task to regulate support leg’s rhythm, added an additional reward term to penalize right leg’s deviation from its default position, and finally re-implemented the termination condition (termination when base or right foot endure forces).

For the user command, we expected the agent may follow linear velocity in x and y direction, but not angular velocity in z (due to the nature of a single 3-dof leg with pointfoot, it is hard to adjust the reaction moment from the ground actively).

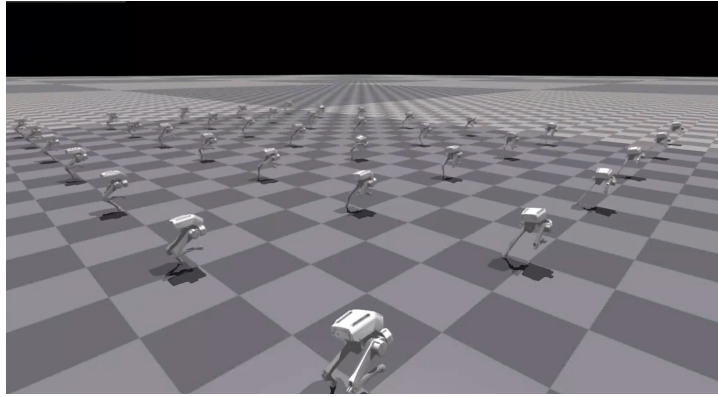


Figure 6: Replay of single leg task in Isaac Gym

Finally, the agent learned a bit of single-leg balancing techniques from scratch, but not that much. For the Raibert hopping machine [6], basically the nature is to using a simplified model and a heuristic term for guessing the next foot step. We deem that if we can add a reference motion, so that the agent may learn and converge more effectively.

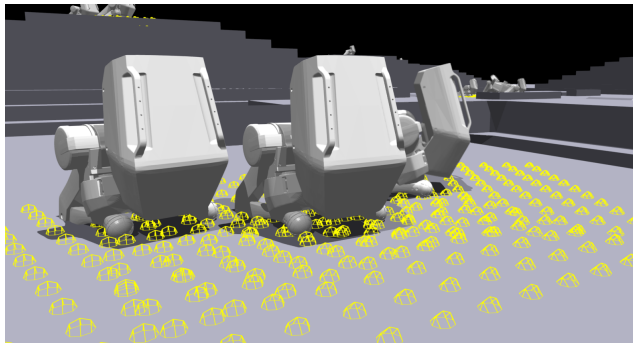


Figure 7: Agent learns to kneel down

3.2 Kneel Down

As mentioned in 7, we made the agent learned to kneel down by chance. But also we can make a few adjustment to make it more specified and directional for the training

of this task. By simply rewarding the contact between thigh links and shank or foot links, rewarding the symmetry between left and right leg, penalizing the joint torques and velocity, penalizing deviation of the hip roll joint from zero, the agent quickly learns to kneel down from its default joint configuration.

REFERENCES

- [1] Jared Di Carlo et al. “Dynamic Locomotion in the MIT Cheetah 3 Through Convex Model-Predictive Control”. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 1–9. DOI: 10.1109/IROS.2018.8594448.
- [2] RSL Lab Eth Zurich. *Legged Gym*. URL: https://github.com/leggedrobotics/legged_gym.
- [3] RSL Lab Eth Zurich. *RSL RL*. URL: https://github.com/leggedrobotics/rsl_rl.
- [4] Donghyun Kim et al. “Highly dynamic quadruped locomotion via whole-body impulse control and model predictive control”. In: *arXiv preprint arXiv:1909.06586* (2019).
- [5] Ashish Kumar et al. “Rma: Rapid motor adaptation for legged robots”. In: *arXiv preprint arXiv:2107.04034* (2021).
- [6] Marc H Raibert. *Legged robots that balance*. MIT press, 1986.
- [7] Nikita Rudin et al. “Learning to walk in minutes using massively parallel deep reinforcement learning”. In: *Conference on Robot Learning*. PMLR. 2022, pp. 91–100.
- [8] Hongxi Wang et al. “CTS: Concurrent Teacher-Student Reinforcement Learning for Legged Locomotion”. In: *IEEE Robotics and Automation Letters* (2024).