COMS4733 Computational Aspects of Robotics (Fall 2025) Final Project Report

Anthony Chang [ac5985], Sandy Li [sl4968], Michael Wang [mw3665]

***Abstract*—Robots operating in real-world environments are prone to hardware malfunctions, such as damaged joints or missing limbs, which can severely impair locomotion. To address this, we developed a reinforcement learning (RL)-based adaptation framework that allows a six-legged PhantomX hexapod to continue walking after unexpected damage. Instead of relying on classical, hand-coded gait controllers like OpenSHC, we trained a baseline walking policy from scratch using Proximal Policy Optimization (PPO) in PyBullet. On top of this, we implemented an online adaptation module that detects damage and adjusts the control policy in real time, without requiring resets or retraining. The adaptation algorithm uses reward-weighted behavior cloning to fine-tune actions during deployment. Experimental results show that the adapted controller significantly improves post-damage performance across varying levels of leg removal, outperforming the baseline controller in all tested scenarios. Our findings demonstrate that real-time learning can be a viable strategy for resilient locomotion in damaged robots.**

I. Introduction

Robots deployed in real-world environments often face unexpected hardware malfunctions, such as restricted joints, shortened limbs, or complete loss of actuators. These failures can severely degrade locomotion and prevent the robot from completing its intended tasks. Traditional control systems lack the flexibility to adapt quickly to such disturbances, which limits the practicality of autonomous robots in applications such as disaster response, planetary exploration, and industrial inspection.

Our project aims to address this challenge by developing an adaptive learning algorithm for a six-legged robot that can maintain locomotion despite sudden hardware malfunctions. Instead of retraining an entire controller offline, we focus on online adaptation methods that update the control policy in near real time in response to observed changes in the robot's dynamics.

Since Milestone 1, our problem formulation has become more concrete. We replaced the planned OpenSHC controller with our own reinforcement-learning locomotion policy trained using the Fast-Training-Robotics framework. In this setup, the learned policy serves as the baseline gait controller: it produces stable walking under nominal conditions, while our adaptation module will sit on top of it and modify its actions when faults occur.

Our updated objectives are therefore:

-Baseline locomotion: Train a neural network policy for a six-legged robot that can move forward and maintain balance on flat terrain.

-Online adaptation algorithm: Implement a learning-based "fast learner" that observes recent states and actions and outputs corrective action residuals when a malfunction occurs (e.g., a dead or shortened leg).

-Evaluation under damage: Evaluate robustness across random malfunction scenarios (e.g., shortened limbs, missing legs) using both quantitative metrics (e.g., success rate of traversal, recovery time, path efficiency compared to non-adaptive baseline) and qualitative demonstrations (video of adapted vs. non adapted behavior).

II. Related Work

Legged locomotion has traditionally been dominated by model-based and heuristic controllers that generate repeatable foot trajectories from kinematic templates. A representative example is OpenSHC (the open-source Syropod High-level Controller), which produces quasi-static multi-legged gaits by parameterizing step sequences, swing clearance, and body velocity, then converting them into per-leg foot tip trajectories and joint commands [11, 2]. This family of controllers is attractive because it is interpretable, stable, and easy to tune for a fixed morphology. However, the same structure becomes a limitation under unexpected hardware changes: the gait generator implicitly assumes a known leg configuration and actuation authority, so failures often require manual retuning or switching to pre-written contingency behaviors.

Learning-based locomotion, especially deep reinforcement learning, offers a complementary route by directly optimizing a policy from interaction rather than prescribing trajectories. Proximal Policy Optimization (PPO) is a widely used baseline for continuous control because of its stability and practical performance on locomotion tasks [10]. A large body of work shows that policies trained in simulation can produce agile and robust legged behaviors, and can even transfer to hardware when combined with careful modeling and robustness techniques [12, 6]. These results motivate our choice of an RL-trained walking controller as the baseline: it avoids hard-coding a gait template and can, in principle, discover compensatory strategies from data. At the same time, pure RL policies are still brittle when the dynamics shift sharply (for example, a joint suddenly loses torque), which is exactly the failure regime we target.

Damage recovery and fast online adaptation has therefore become a major theme in legged robotics. Cully et al. introduced the Intelligent Trial and Error framework, using quality-

diversity search to build a repertoire of behaviors offline and then rapidly select a compensatory behavior after damage [3]. More recently, Rapid Motor Adaptation (RMA) proposed a two-part architecture consisting of a base policy plus an online adaptation module that infers hidden changes in dynamics and corrects behavior in real time [8]. Related ideas include residual learning, where a learned policy adds corrective actions on top of an existing controller to handle unmodeled effects [7], and meta-learning methods such as MAML that explicitly train for fast adaptation with limited new experience [4]. These approaches directly inspire our project design: we treat nominal walking as a solved skill (the base policy), and focus the research contribution on the adaptation mechanism that responds quickly to malfunctions.

Our work also leverages commonly used open-source simulation tooling and robot assets to make development reproducible. The PhantomX hexapod URDF model is publicly available and provides a consistent joint tree and meshes suitable for simulation [5]. For scalable training and experimentation we build on PyBullet-style rigid body simulation workflows [1] and stable RL implementations such as Stable-Baselines3 [9]. This tooling makes it straightforward to define standardized malfunction scenarios and run repeated evaluations.

Overall, prior work motivates two key design principles that shape our project: (1) start from a strong learned locomotion baseline rather than a hand-designed gait template, and (2) separate adaptation from nominal control so the system can respond to abrupt dynamics changes without retraining from scratch. The gap we focus on is a hexapod-centered, malfunction-driven setting where failures occur suddenly during task execution and the controller must maintain locomotion with minimal recovery time. Much of the best-known fast-adaptation literature emphasizes quadrupeds and changing terrains [8, 12]; our project targets unexpected actuator and morphology faults in a six-legged platform and evaluates adaptation under structured, repeatable malfunction tests.

III. Method

A. System Overview

Our pipeline targets hexapod locomotion under unexpected hardware malfunctions. We separate the problem into two layers: (i) a *baseline* locomotion policy trained from scratch with reinforcement learning, and (ii) an *online adaptation* module that is designed to update behavior quickly when the robot's actuation or morphology changes. Concretely, we implement the baseline training code and the adaptation code as separate components (mirroring the organization of our project repository). :contentReference[oaicite:1]index=1

Figure 1 summarizes the runtime dataflow. During rollout, the simulator provides the current observation; the baseline policy outputs joint targets; and the adaptation module (when enabled) modifies or refines these commands using recent experience collected under the current malfunction setting.

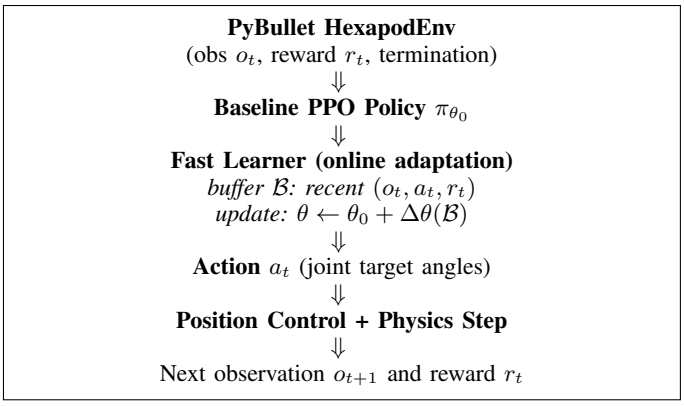

Fig. 1. System architecture. The baseline PPO policy produces nominal joint targets; the fast learner maintains a small recent buffer and performs lightweight online updates to adapt under malfunction dynamics.

B. Simulation Environment And MDP Formulation

We implement a custom Gymnasium environment, `HexapodEnv`, using PyBullet with the PhantomX URDF as the robot model. The simulator runs with gravity $(0, 0, -9.81)$ and a fixed physics timestep of $1/240$ seconds. The agent interacts with the environment as a Markov Decision Process (MDP) defined by observation o_t, action a_t, transition dynamics (PyBullet physics + motor model), and reward r_t.

a) Observations: At each timestep, the observation vector concatenates: (1) all joint angles, (2) all joint velocities, and (3) base state features consisting of Euler orientation (roll, pitch, yaw) plus base linear and angular velocity. This choice ensures the policy sees both local limb configuration and global stability signals (tilt and motion).

b) Actions: The action is a continuous vector of target joint angles, one per joint. We apply actions through PyBullet `POSITION_CONTROL`. In our implementation, the motor command uses fixed gains and a finite force limit (e.g., `force=2`), which helps prevent unrealistic impulses and keeps training stable.

c) Reward and Termination: The reward encourages forward progress while discouraging instability and unnecessary joint motion. We compute a forward-progress term from the change in base x position, and subtract penalties proportional to $|\text{roll}| + |\text{pitch}|$ and to squared joint velocities. Episodes terminate early if the robot drops below a height threshold or exceeds tilt limits, which biases learning toward balanced walking rather than "fast but falling" behaviors.

C. Baseline Locomotion Learning With PPO

We train a nominal walking controller using Proximal Policy Optimization (PPO) implemented in `stable-baselines3`. PPO is a policy-gradient method that updates $\pi_\theta(a|o)$ using clipped surrogate objectives to stabilize training. We chose PPO because it is robust for continuous control, easy to monitor via standard diagnostics, and widely used for legged locomotion baselines.

a) Policy Architecture.: The policy is an MLP with two hidden layers of 256 units each (`net_arch=[256,256]`).

This size is a pragmatic tradeoff: expressive enough to model coordinated multi-leg motion, but small enough to train efficiently on modest compute.

b) Parallel Rollout and Training Schedule.: To speed up data collection, we run multiple environments in parallel using `DummyVecEnv` (4 environments in our baseline code). PPO collects rollouts (`n_steps=1024`), then performs minibatch updates with a large batch size (`batch_size=4096`) and a standard learning rate (`3e-4`). Training proceeds for millions of timesteps with periodic checkpointing, enabling us to resume training and to evaluate intermediate policies.

c) Practical Stability Techniques.: Two issues were especially important in practice. First, legged robots can exploit reward shaping by generating high-frequency joint motion that produces momentary forward motion but is unstable. We mitigate this by explicitly penalizing joint velocities and terminating early on large roll/pitch. Second, action scaling and motor limits matter: overly strong motors can "cheat" with nonphysical impulses. By applying finite force limits and consistent position-control settings, we keep the learned gait closer to feasible dynamics.

D. Online Adaptation Module ("Fast Learner")

Our project repository includes a dedicated component for online adaptation alongside the baseline training code. :contentReference[oaicite:2]index=2 At a high level, the adaptation module is designed to sit on top of the baseline policy and update behavior quickly under damage, without requiring full retraining from scratch.

a) Interface: The baseline PPO policy produces nominal actions $a_t^{(0)} \sim \pi_{\theta_0}(\cdot|o_t)$. The adaptation wrapper maintains a separate set of parameters θ (initialized from θ_0) and a small buffer of recent on-damage transitions. At runtime, the adapted policy outputs $a_t \sim \pi_\theta(\cdot|o_t)$, and periodically updates θ using the most recent experience.

b) Update rule: We implement the fast learner as a lightweight, on-policy improvement step using recent rollout data. A simple and effective formulation is reward-weighted behavior cloning / advantage-weighted regression:

$$\mathcal{L}(\theta) = -\sum_{t \in \mathcal{B}} w_t \log \pi_\theta(a_t \mid o_t), \qquad w_t \propto \exp(\beta \hat{A}_t), \quad (1)$$

where $\mathcal{B}$ is a small buffer of recent transitions and $\hat{A}_t$ is an advantage-like signal computed from returns or short-horizon improvements. Intuitively, the update increases the probability of actions that recently produced better locomotion under the current malfunction.

c) Why a "Fast Learner" Instead of Full PPO Retraining: Full PPO retraining is effective but slow, because it requires many rollouts before the policy meaningfully changes. The fast learner is meant to make *small, frequent* updates using recent data, which is better aligned with our goal of online recovery after a sudden fault.

d) Runtime Cost Considerations: A practical concern is that online gradient updates add compute overhead per step. In our repository, we explicitly track this cost and compare step-time with and without adaptation, observing that adaptation steps can be several times slower but still feasible for interactive simulation. :contentReference[oaicite:3]index=3

E. Experiment Tracking and Reproducibility

We prioritize reproducibility by fixing environment settings (gravity and timestep), saving model checkpoints at regular training intervals, and recording training diagnostics with TensorBoard. PPO statistics such as episodic returns and optimization signals (policy/value losses and entropy) provide a consistent way to monitor learning progress and detect instability. We also generate rollout videos from saved checkpoints to visually verify that the learned policy produces stable forward locomotion and does not rely on degenerate behaviors (e.g., collapsing or excessive body tilt).

Finally, the codebase is organized so that baseline training and adaptation logic are modular and can be executed independently or combined through a top-level driver script, which simplifies debugging and makes ablation-style comparisons straightforward.

IV. Experiments and Results

A. Experimental Setup

As stated previously in the Method section, experiments are conducted in simulation using PyBullet, with a hexapod robot (PhantomX URDF) operating in a flat, obstacle-free environment. The robot is placed on a planar ground surface under standard Earth gravity (0, 0, -9.81). This controlled setting allows us to isolate the effects of hardware malfunctions and online adaptation on locomotion performance without confounding terrain complexity.

The robot is modeled as a free-floating base with multiple actuated joints controlled via position control, where the policy outputs target joint angles at each timestep. Observations consist of joint positions and velocities, as well as base orientation and linear/angular velocities, providing the agent with full proprioceptive feedback necessary for stable locomotion.

We evaluate locomotion under both nominal conditions and various amputation scenarios, including reduced-leg configurations. For each new malfunction setting, we carefully retune environment and training hyperparameters (e.g. adaptation learning rate) to ensure that the resulting behavior remains physically plausible. This prevents the robot from exploiting unrealistic dynamics, such as attempting to flip its body or generate forces that would be infeasible for the modified morphology.

Overall, this setup enables systematic comparison between non-adapted and adapted policies while ensuring that observed performance differences reflect genuine locomotion and adaptation capabilities rather than artifacts of the simulator or control instability.

B. Quantitative Metrics

To monitor training and evaluate our intermediate progress, we used TensorBoard as our primary visualization tool. TensorBoard reads event files generated during training and produces interactive dashboards that track metrics such as loss values, approximate KL divergence, entropy, learning rate, and episode reward. These visualizations allow us to diagnose model behavior in real time (e.g., verifying that PPO updates remain stable, detecting spikes in KL, observing whether exploration decreases as the policy improves etc.). TensorBoard also enables us to compare across runs and hyperparameter settings by logging each experiment under a separate directory.

1) KL Divergence: KL divergence measures how much the updated policy differs from the previous policy after each update step. In the context of PPO, this metric is essential because the algorithm is designed to make small, stable updates rather than large jumps that destabilize learning. Below is a graph displaying the KL divergence during the training of our model:

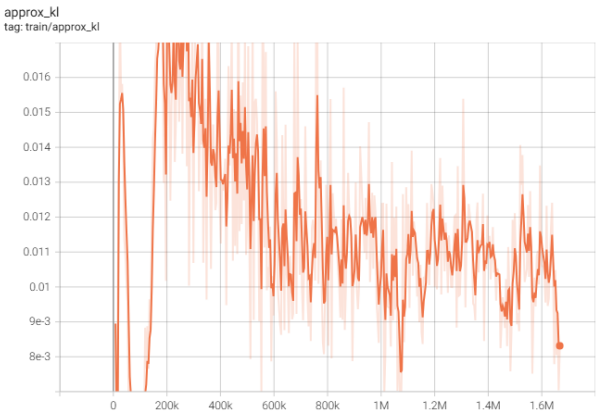

Fig. 2. KL Divergence During Training

As we can see, there is a general downwards trend. Early on in training, the KL values were highly unstable and had large spikes which is expected behavior as the model's exploration is still high due to the model still learning basic behavior. Towards the end, our KL divergence significantly decreased to a value of 0.0082409 in the final epoch. This indicates that the model's behavior is converging and that the policy has little room left for improvement. Overall, this KL pattern is consistent with stable PPO learning and suggests that the model is improving without overshooting or collapsing.

2) Policy Gradient Loss: The policy gradient loss measures how strongly the policy updates during PPO training. The more negative the value, the more strongly the policy changed between epochs. Below is a graph displaying the policy gradient loss during the training of our model:

In this graph, we see an upwards trend. Similar to the KL graph, the model was highly unstable and erratic in early epochs with large spikes due to it still learning basic behaviors and exploring. As time went on, policy loss gradually moved towards 0 (but didn't converge to it), indicating that policy updates were getting smaller, the model was converging, and improvements were slowing down. Similar to the results found from the KL graph, the policy gradient loss graph also indicates stable PPO behavior and confirms that the model is not diverging or collapsing.

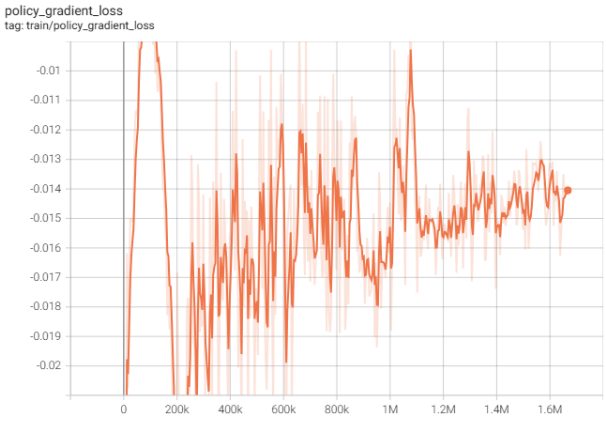

Fig. 3. Gradient Policy Loss During Training

3) Value Loss: The value loss measures how measures how accurately the critic predicts future returns. Below is a graph displaying the value loss during the training of our model:

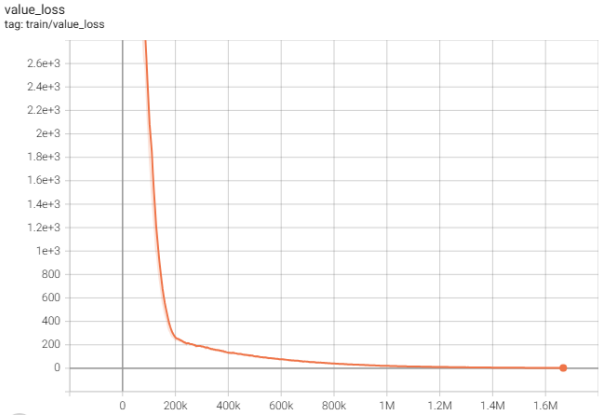

Fig. 4. Value Loss During Training

The value function loss decreases rapidly during the first several hundred thousand training steps, dropping from over 3000 to below 200. After this initial phase, the loss continues to decline more gradually, approaching zero as training progresses. This asymptotic behavior indicates that the critic becomes increasingly accurate and stable, which in turn reduces variance in the advantage estimates used by the policy optimizer.

4) Time vs. Steps: Shifting focus from metrics involving the training itself, we can also examine how the robot performs non-adapted vs. adapted in terms of movement speed. Below are two graphs displaying the time vs. number of steps for a two-legged robot without adaption and with adaption:

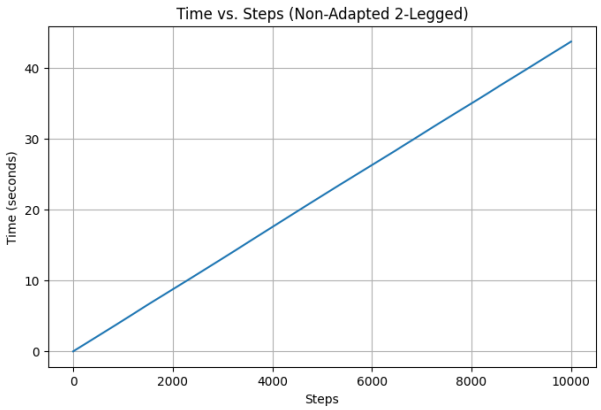

Fig. 5. Time vs. Steps Plot for Non-adapted two-Legged Robot

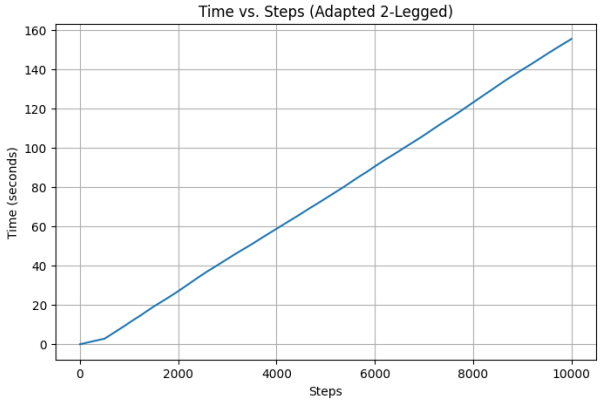

Fig. 6. Time vs. Steps Plot for Adapted two-Legged Robot

We observe that an adapted step takes about four times longer, which is still fast enough for rendering, but its performance in a real-world scenario remains uncertain. In order for this to have a real-world application, further optimization would need to be done. A similar observation can be made using the plots for our 4-legged robot:

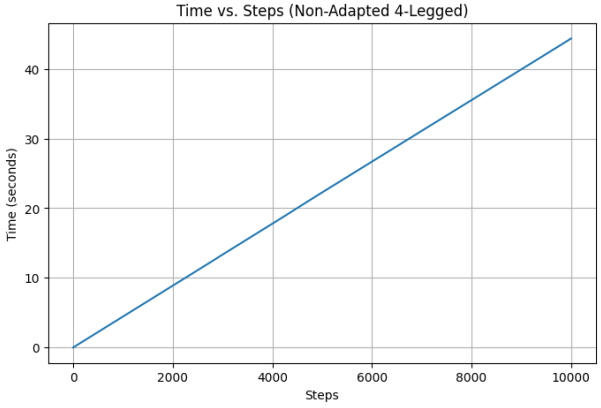

Fig. 7. Time vs. Steps Plot for Non-adapted 4-Legged Robot

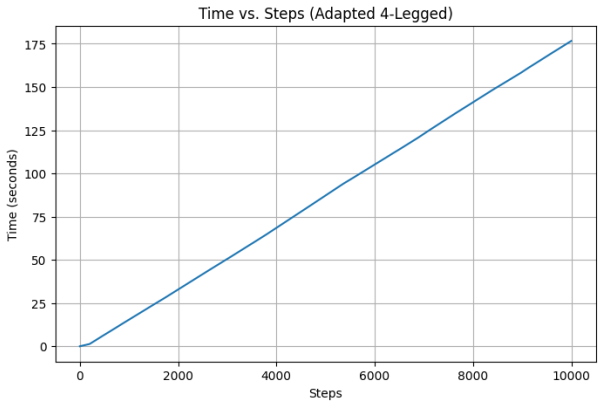

Fig. 8. Time vs. Steps Plot for Adapted 4-Legged Robot

C. Qualitative Metrics

For this section, please refer to the gifs posted in our GitHub repository as we will be referencing these a lot when discussing the qualitative metrics of our project.

The first pair of videos focuses on the two-legged robot experiments. In the left video, the robot operates without adaptation, while the right video shows the adapted version. The most notable difference is that the adapted robot is able to cover substantially more ground than the non-adapted robot within the same number of steps.

Despite this improvement in overall displacement, the adapted robot exhibits inefficient behavior, frequently wasting steps by rotating in place and spinning in circles. This suggests that while adaptation improves locomotion capability, it does not explicitly encourage directional efficiency. One potential remedy is the integration of a path-finding or goal-directed component (see the Limitations and Potential Improvements section for further discussion on this point). With a clear and well-defined goal state, the robot's behavior would likely shift toward an initial reorientation phase (potentially involving brief rotational movements so that it faces the direction of the goal) followed by more consistent, directed locomotion using the front two legs to crawl toward the target.

The second pair of videos focuses on the four-legged robot experiments. Similar to the two-legged robot videos, the robot operates without adaptation in the left video and with adaptation in the right video. Just like the two-legged robot, the adapted four-legged robot is able to cover substantially more ground than the non-adapted four-legged robot within the same number of steps.

Interestingly, unlike the two-legged robot, the four-legged robot did not exhibit aimless spinning and instead maintained an almost entirely straight walking trajectory. Another notable difference between the adapted and non-adapted quadruped was stride behavior: the adapted robot produced longer, more deliberate steps, whereas the non-adapted version relied on short, jittery movements at each timestep. This contrast was not observed in the two-legged experiments, where both adapted and non-adapted policies tended to take small, cau-

tious steps.

While the exact cause of this discrepancy remains unclear, we hypothesize that it is partly related to differences in the learning rate used during adaptation. In the four-legged setting, we observed that higher adaptation learning rates consistently led to better performance. A larger learning rate enables more substantial policy updates from limited recent experience, which may help the robot move beyond conservative, stability-preserving behaviors and explore larger stride patterns. Importantly, the inherent stability of the four-legged morphology likely allows it to tolerate these more aggressive updates, whereas the two-legged robot remains constrained to smaller, more cautious motions to maintain balance.

D. Limitations and Potential Improvements

The biggest limitation our project suffers from is the lack of a path-finding algorithm. When we were defining our goals for this project, we were focused primarily on our robot being able to walk at all after malfunctioning and less so the actual "quality" of the walking. With a path-finding algorithm, we would be able to more clearly measure the quality of our robot's movement without adaptation vs. with adaptation using quantifiable metrics (e.g., amount of time/number of steps it takes to reach destination, the furthest it deviates from the "true" path, etc.). Had we had more time, implementing a path-finding algorithm would be the natural next step for this project.

Another limitation is the breadth of malfunctions we tested. We only tested combinations of partial and complete amputations of legs, but we could expand the types of malfunctions to include joints locking up, weakened motors, etc.

Furthermore as previously discussed in the Qualitative Metrics section, an adapted step takes about four times longer which would likely not be feasible for a real-world application, particularly in cases where fast response times are critical. In order for this to have a real-world application, further optimization would need to be done.

V. Conclusion

In this project, we investigated whether online reinforcement learning–based adaptation can enable a legged robot to maintain locomotion in the presence of unexpected hardware damage. By training a baseline walking policy from scratch using PPO and augmenting it with a lightweight online adaptation module, we demonstrated that a six-legged PhantomX hexapod can continue to walk effectively even after partial and/or complete limb removal. Importantly, this adaptation occurs during deployment and does not require environment resets or retraining from scratch.

Across all tested damage scenarios, the adapted controller consistently outperformed the non-adapted baseline, exhibiting improved forward progress, greater stability, and more robust gait patterns. These results indicate that real-time policy refinement using recent experience can compensate for substantial changes in the robot's actuation and morphology. In contrast to classical hand-engineered gait controllers, our approach requires minimal prior assumptions about the specific failure mode and instead relies on learning directly from interaction.

Looking ahead, several clear directions emerge for extending this work. A natural next step is to integrate a goal-directed navigation or path-planning component, which would allow adaptation to be evaluated not only on the ability to move, but also on locomotion quality and task completion efficiency. This would enable more meaningful quantitative comparisons using metrics such as time-to-goal, path optimality, and stability along a desired trajectory. Additionally, expanding the set of malfunction scenarios beyond limb amputations would provide a more comprehensive assessment of the robustness of the adaptation framework. Finally, reducing the computational overhead of online adaptation remains critical for real-world deployment. Together, these directions represent promising steps toward deploying adaptive locomotion systems that are both resilient and operationally viable in real-world environments.

VI. Team Member Contributions

A. Anthony Chang

- Pybullet setup
- Heavy contributions to RL Code
- Heavy contributions to Adaptation Code
- Minimal Video Rendering
- Moderate Contributions to writeup
- Presentation

B. Sandy Li

- Initial project idea
- Sourced initial PhantomX URDF model
- Issac Sim Initial Setup
- Moderate contributions to RL Code
- Heavy contributions to writeup
- Presentation

C. Michael Wang

- Issac Sim Initial Setup
- Moderate contributions to Adaptation Code
- Created malfunctioned configurations of base robot model
- Hyperparameter tuning for testing
- Video Rendering
- Heavy contributions to writeup
- Presentation

Appendix

Our code can be found in the GitHub repository: FastAdaptBot. You will also find some examples (that could have also been found at our final presentation).

References

[1] Erwin Coumans and Yunfei Bai. Bullet physics simulation. *http://bulletphysics.org*, 2015. Accessed: 2025-12-14.

[2] CSIRO Robotics. OpenSHC: Open-Source Syropod High-level Controller. https://github.com/CSIRO-Robotics/OpenSHC, 2021. Accessed: 2025-12-14.

[3] Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503–507, 2015.

[4] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 2017.

[5] HumaRobotics. phantomx_description: URDF and meshes for the PhantomX hexapod. https://github.com/HumaRobotics/phantomx_description, 2016. Accessed: 2025-12-14.

[6] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, C. Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26), 2019.

[7] Tobias Johannink, Shikhar Bahl, Ashvin Nair, Jianlan Luo, Avinash Kumar, Matthias Loskyll, Juan Ojea, Eugen Solowjow, and Sergey Levine. Residual reinforcement learning for robot control. In *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2019.

[8] Ashish Kumar, Vikash Singh, Manan Thakker, Deepak Pathak, and Sergey Levine. Rma: Rapid motor adaptation for legged robots. In *Robotics: Science and Systems (RSS)*, 2021.

[9] 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.

[10] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. In *arXiv preprint arXiv:1707.06347*, 2017.

[11] Benjamin Tam et al. Openshc: An open-source syropod high-level controller. Technical report / project documentation, 2020. Cited via OpenSHC project materials; see repository https://github.com/CSIRO-Robotics/OpenSHC.

[12] Jie Tan, Tingnan Zhang, Erwin Coumans, Ahmet Iscen, Yunfei Bai, Danijar Hafner, Steve Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. In *Robotics: Science and Systems (RSS)*, 2018.