Language Model Guided Sim-To-Real Transfer

Anonymous CVPR submission

Abstract

Transferring policies learned in simulation to the real world is a promising strategy for acquiring robot skills at scale. However, sim-to-real approaches typically rely on manual design and tuning of the task reward function as well as the simulation physics parameters, rendering the process slow and human-labor intensive. In this paper, we investigate using Large Language Models (LLMs) to automate and accelerate sim-to-real design. Our LLM-guided sim-to-real approach requires only the physics simulation for the target task and automatically constructs suitable reward functions and domain randomization distributions to support real-world transfer. We first demonstrate our approach can discover sim-to-real configurations that are competitive with existing human-designed ones on quadruped locomotion and dexterous manipulation tasks. Then, we showcase that our approach is capable of solving novel robot tasks, such as quadruped balancing and walking atop a yoga ball, without iterative manual design.

1. Introduction

We propose `DrEureka` (**D**omain **R**andomization Eureka), a novel algorithm leveraging Large Language Models (LLMs) to automate the development of reward functions and domain randomization (DR) parameters for sim-to-real transfer. By leveraging the LLM's strong grasp of physical knowledge [1, 2] and effectiveness in generating hypotheses, `DrEureka` simplifies the traditionally manual reward and DR tuning process by efficiently synthesizing reward functions and DR parameters.

We evaluate `DrEureka` on quadruped and dexterous manipulator platforms, demonstrating that our method is general and applicable to diverse robots and tasks. For forward locomotion, `DrEureka`-trained policies outperform human-designed ones by 34% in speed and 20% in distance across different terrains. In dexterous cube rotation, `DrEureka`'s best policy performs nearly 300% more in-hand cube rotations than the human-developed policy. Finally, we apply `DrEureka` to a novel task—balancing a quadruped on a yoga ball, achieving up to 15 seconds of balance in an evaluation setting and over four minutes outdoors with additional controls.

2. Method

`DrEureka` consists of three stages (Figure 1). First, we build on Eureka [3], an algorithm that repeatedly samples reward function candidates from an LLM, trains policies with each reward candidate, and provides the best-performing policy's reward and training statistics as feedback for the LLM. To prevent simulated policies from over-exerting motors or learning unnatural behavior, we directly exploit the strong instruction-following capability of instruction-tuned LLMs [4] and prompt the LLM to explicitly consider safety terms for stability, smoothness, and desirable task-specific attributes. The resulting best reward-policy pair $R_{\text{DrEureka}}, \pi_{\text{initial}}$ is much more suitable for deployment and minimizes the risk of dangerous behavior.

Then, we introduce a simple *reward aware physics prior (RAPP)* mechanism to compute feasible DR parameter bounds. At a high level, RAPP seeks for the maximally diverse range of environment parameters where π_{initial} is still performant. In practice, for each parameter, we search through a general range of potential values at varying magnitudes, and with each value, we set it in simulation (keeping all other parameters at default) and roll out π_{initial} in this modified simulation. If the policy's performance satisfies a pre-defined success criterion, we deem this value as feasible for this parameter. Given the set of all feasible values for each parameter, our lower and upper bounds for a parameter are the minimum and maximum feasible values.

Finally, we use RAPP-defined ranges to guide the LLM in generating domain randomization (DR) configurations, contrasting with automatic domain randomization methods that directly apply these ranges. Concretely, we provide all randomizable parameters and their RAPP ranges in the LLM context and ask the LLM (1) to choose a subset of to randomize and (2) determine their randomization ranges. In this manner, the backbone LLM zero-shot generates several independent DR configuration samples, and we use RL to train policies for each reward and DR combination, resulting in a set of policies. Unlike the reward design component, it is difficult to select the *best* DR configuration and policy in simulation because each policy is trained on its own DR distribution and cannot be easily compared. Hence, we keep all m policies and report both the best and the average performance in the real world.

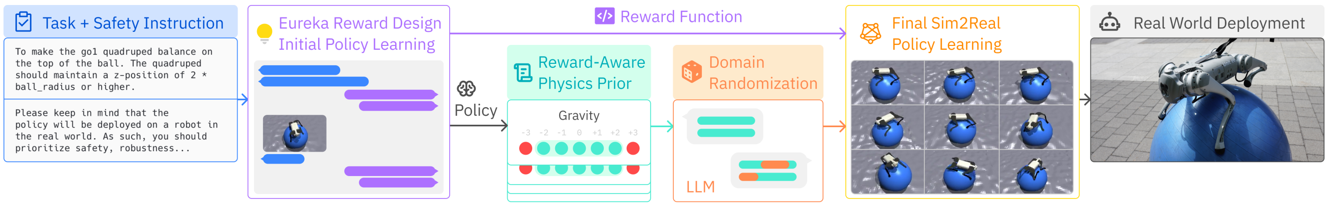

Figure 1. DrEureka uses reward generation, RAPP, and DR generation to produce deployable real-world policies.

Forward Locomotion		
Sim-to-real Configuration	Forward Velocity (m/s)	Meters Traveled (m)
Human-Designed [5]	1.32 ± 0.44	4.17 ± 1.57
Eureka [3]	0.0 ± 0.0	0.0 ± 0.0
Our Method (Best)	**1.83** ± 0.07	**5.0** ± 0.00
Our Method (Average)	1.66 ± 0.25	4.64 ± 0.78
Cube Rotation		
Sim-to-real Configuration	Rotation (rad)	Time-to-Fall (s)
Human-Designed [6]	3.24 ± 1.66	20.00 ± 0.00
Our Method (Best)	**9.39** ± 4.15	**20.00** ± 0.00
Our Method (Average)	4.67 ± 3.55	16.29 ± 6.28

Table 1. **Comparison against baselines.** DrEureka's average and best policies outperform Human-Designed and a prior reward-design baseline.

3. Results and Analysis

We evaluate our method on the Unitree Go1 quadruped for the forward locomotion task, which commands the robot to walk forward at 2 meters-per-second on flat terrains. We also validate DrEureka on the Leaphand [6] for cube rotation, which involves rotating a cube in-hand as many times as possible within a 20-second interval. We compare with policies from Margolis et al. [5] and Shaw et al. [6], which we refer to as Human-Designed, as well as Eureka [3], which does not have safety consideration and domain randomization. More details about our experimental setup and ablations are in the Appendix.

Comparison to Existing Sim-to-Real Configurations. We first compare DrEureka to Human-Designed to assess whether DrEureka is capable of providing sim-to-real training configurations comparable to human-designed ones. For forward locomotion, as shown in Table 1, DrEureka is able to outperform Human-Designed in terms of both forward velocity as well as distance traveled on the track. The performance of DrEureka is robust across its different DR sample outputs; the average performance does not lag too far behind the best DrEureka configuration and still performs on par with or slightly better than Human-Designed. In contrast, the plain Eureka policy fails to walk in the real world (more analysis in Appendix), validating that a reward design algorithm suitable for simulation is not sufficient for sim-to-real transfer.

Similarly, for cube rotation, we see in Table 1 that DrEureka outperforms Human-Designed in terms of rotation while maintaining a competitive time-to-fall duration. We note that this task permits very little room for error; thus, policies generally perform very well or very badly, which is reflected in the relatively larger standard deviation across DrEureka's policies. Nevertheless, the best policy from DrEureka significantly outperforms the baseline by nearly three times the rotation without dropping the cube. These results highlight the effectiveness and versatility of our approach across diverse robotic platforms.

Real-world Robustness. One main appeal of domain randomization is the robustness of the learned policies to real-world environment perturbations. To probe whether DrEureka policies exhibit this capability, we test DrEureka (Best) and Human-Designed on several additional testing environments for forward locomotion. Within the lab environment, we consider an artificial grass turf as well as putting socks on the quadruped legs. For an outdoor environment, we test on an empty pedestrian sidewalk. Numerical results are in the Appendix. We see that across different testing conditions, DrEureka remains performant and consistently matches or outperforms Human-Designed. This validates that DrEureka is capable of producing robust policies in the real world.

The Walking Globe Trick. We employ DrEureka for the novel and challenging globe walking task where the quadruped balances on a yoga ball. The deformable, bouncy surface, which is not accurate in simulation, increases task complexity. Lacking existing sim-to-real configurations, this task offers an ideal test-bed for DrEureka's ability to accelerate robot skill discovery.

In a lab setting that straps the robot to a central support point, we observe the quadruped staying on the ball for an average of 15.43 seconds, many times making recovery actions to stabilize the ball and readjust its pose. When deployed in diverse, uncontrolled outdoor scenes with appropriate controls that limit the robot's speed, the policy operated effectively for over four minutes under various conditions and obstacles. In summary, DrEureka's adeptness at tackling the novel and complex task of quadrupedal globe walking showcases its capacity to push the boundaries of what is achievable in robotic control. This feat, achieved without prior sim-to-real pipelines, highlights DrEureka's potential to accelerate the development of robust and versatile robotic policies in the real world.

References

[1] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022. 1

[2] Yi Ru Wang, Jiafei Duan, Dieter Fox, and Siddhartha Srinivasa. Newton: Are large language models capable of physical reasoning? *arXiv preprint arXiv:2310.07018*, 2023. 1

[3] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023. 1, 2, 4, 5

[4] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022. 1

[5] Gabriel B Margolis, Ge Yang, Kartik Paigwar, Tao Chen, and Pulkit Agrawal. Rapid locomotion via reinforcement learning. *arXiv preprint arXiv:2205.02824*, 2022. 2, 4, 6, 14

[6] Kenneth Shaw, Ananye Agarwal, and Deepak Pathak. Leap hand: Low-cost, efficient, and anthropomorphic hand for robot learning. *arXiv preprint arXiv:2309.06440*, 2023. 2, 4

[7] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014. 4

[8] OpenAI. Gpt-4 technical report, 2023. 4

[9] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019. 4

[10] Ankur Handa, Arthur Allshire, Viktor Makoviychuk, Aleksei Petrenko, Ritvik Singh, Jingzhou Liu, Denys Makoviichuk, Karl Van Wyk, Alexander Zhurkevich, Balakumar Sundaralingam, et al. Dextreme: Transfer of agile in-hand manipulation from simulation to reality. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5977–5984. IEEE, 2023.

[11] Gabriele Tiboni, Pascal Klink, Jan Peters, Tatiana Tommasi, Carlo D'Eramo, and Georgia Chalvatzaki. Domain randomization via entropy maximization. *arXiv preprint arXiv:2311.01885*, 2023. 4

[12] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*, 2023. 5

[13] Michael R Zhang, Nishkrit Desai, Juhan Bae, Jonathan Lorraine, and Jimmy Ba. Using large language models for hyperparameter optimization. *arXiv e-prints*, pages arXiv–2312, 2023.

[14] Anonymous. Large language models to enhance bayesian optimization, 2024. URL https://openreview.net/forum?id=OOxotBmGol.

[15] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, pages 1–3, 2023. 5

[16] Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning quadrupedal locomotion over challenging terrain. *Science robotics*, 5(47):eabc5986, 2020. 6

[17] Nikita Rudin, David Hoeller, Philipp Reist, and Marco Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning. In *Conference on Robot Learning*, pages 91–100. PMLR, 2022.

[18] Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. Rma: Rapid motor adaptation for legged robots. *arXiv preprint arXiv:2107.04034*, 2021.

[19] Gabriel B Margolis and Pulkit Agrawal. Walk these ways: Tuning robot control for generalization with multiplicity of behavior. In *Conference on Robot Learning*, pages 22–31. PMLR, 2023. 6

[20] Yandong Ji, Gabriel B Margolis, and Pulkit Agrawal. Dribblebot: Dynamic legged manipulation in the wild. *arXiv preprint arXiv:2304.01159*, 2023. 14

Appendix

A. Experimental Setup

Robot and Task. For our main experiments on quadrupedal locomotion, we use Unitree Go1. The Go1 is a small quadrupedal robot with 12 degrees of freedom across four legs. Its observations include joint positions, joint velocities, and a gravity vector in the robot's local frame, as well as a history of past observations and actions. We use the simulation environment as well as the real-world controller from Margolis et al. [5].

The task of forward locomotion is to walk forward at 2 meters-per-second on flat terrains; while it is possible for the robot to walk forward at a higher speed, we find 2 m/s to strike a good balance between task difficulty and safety as our goal is not to achieve the highest speed possible on the robot. In the real world, we set up a 5-meter track in the lab (see Figure 4) and measure the forward projected velocity and total meters traveled in the track direction.

Additionally, we conduct experiments on the LEAP Hand [6]. The LEAP hand is a low-cost anthropomorphic robot hand, featuring 16 degrees of freedom distributed among three fingers and a thumb.

The cube rotation task involves rotating a cube in-hand as many times as possible within a 20-second interval. This task is challenging because the policy only receives 16 joint angles and proprioceptive history as observations and does not have access to the position and the pose of the cube. The policy then outputs target joint angles as position commands to the motors.

For the cube rotation task, we follow the training and deployment workflow outlined by the LeapHand authors. For training all the policies, we use the same GRU [7] architecture that receives 16 joint angles as input and outputs 16 target joint angles. We also follow the LeapHand training code to randomize the initial pose of the hand and the size of the cube. When deploying trained policies in the real world, the target joint angles are passed as position commands to a PID controller running at 20 Hz.

In addition to the initial pose of the hand and the size of the cube, the `Human Designed` policy is trained with DR in object mass, object center of mass, hand friction, stiffness and damping. In `DrEureka`, we extend the simulation setup to include additional domain randomization parameters, such as hand restitution, joint friction, armature, object friction and object restitution. These parameters, along with the others, are detailed in Table 4.

Methods. `DrEureka` uses GPT-4 [8] as the backbone LLM. `DrEureka` uses the original Eureka hyperparameters for reward generation before sampling 16 DR configurations. To understand the best and the average performance of `DrEureka`, we train policies for all 16 configurations and evaluate all policies in the real world. Given the lack of a prior baseline in our proposed problem setting, we primarily compare to human-designed reward function as well as domain randomization configuration from the original task implementation from Margolis et al. [5] as reference points; We refer to this baseline as `Human-Designed`. Note that this baseline trains a velocity-conditioned policy and utilizes a reward function with a velocity curriculum that gradually increases as policy training progresses. For our comparison, we train on the whole curriculum but evaluate the policy at 2 m/s. Note that the purpose of comparing to `Human-Designed` is to determine whether `DrEureka` can be *useful* – i.e., enabling sim-to-real transfer on a representative robot task for which robotics researchers have devoted time to designing effective sim-to-real pipelines. The absolute performance ordering is of less importance as LLMs and humans arrive at their respective sim-to-real configurations using vastly different computational and cognitive mechanisms.

To verify that a policy outputted by a reward-design algorithm itself is not effective for real-world deployment, we also compare against Eureka [3], which designs rewards using LLMs without safety consideration and trains policies without domain randomization. In our analysis, we further consider several ablations of `DrEureka` in greater detail.

Sim-to-real Configuration	Forward Velocity (m/s)	Meters Traveled (m)
Our Method (Average)	1.66 ± 0.25	4.64 ± 0.78
Without DR	1.21 ± 0.39	4.17 ± 1.04
With `Human-Designed` DR	1.35 ± 0.16	4.83 ± 0.29
With Prompt DR	1.43 ± 0.45	4.33 ± 0.58
Without Prior	0.09 ± 0.36[1]	0.31 ± 1.25
With Uninformative Prior	0.08 ± 0.33[1]	0.28 ± 1.13
With Random Sampling	0.98 ± 0.45	2.81 ± 1.80

Table 2. **Ablations result.** Ablations of the DR formulation in `DrEureka` all result in decreased performance.

B. Ablation Experiments

Our ablation experiments aim to answer whether `DrEureka` generates effective DR configurations.

Ablation Details. We compare `DrEureka` against two classes of ablations that probe (1) whether some fixed DR configuration can generally outperform `DrEureka` samples, and (2) the importance of `DrEureka`'s reward-aware priors and LLM sampling. In the first class, we first compare to an ablation that does not train with domain randomization (**No DR**). Second, we consider a baseline that trains with the `human-designed` DR (**Human-Designed DR**) in the original implementation. Third, we consider a baseline that directly uses the full ranges of the RAPP parameter priors as the DR configuration (**Prompt DR**); this ablation can viewed as applying domain randomization algorithms [9–11] that seek to prescribe the maximally diverse parameter ranges where the policy performs well as

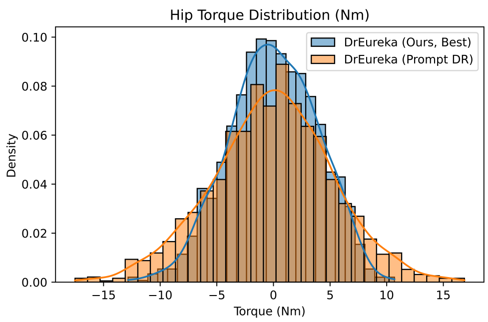

Figure 2. Policies trained on `DrEureka` DR configurations exert less torque in the real world.

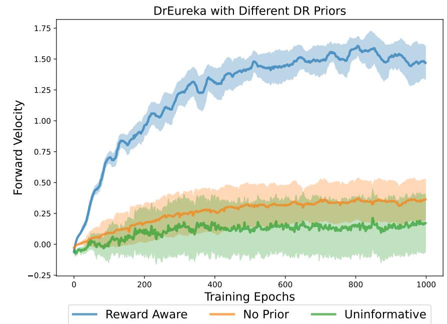

Figure 3. Ablations for different domain randomization priors. Replacing RAPP with other choices makes the LLM generate configurations that are difficult to train in simulation.

the configurations. In the second category of ablations, we consider an ablation that only has access to the set of physics parameters but without the reward-aware priors (**No Prior**). Additionally, we consider an ablation that has only the default search range for RAPP as the parameter priors (**Uninformative Prior**). Finally, we consider a baseline that randomly samples from the RAPP ranges (**Random DR**); this baseline helps show whether LLM-based sampling is a better hypothesis generator. In all ablations, we fix the `DrEureka` reward function for the task and only modify the DR configurations.

`DrEureka` outperforms all DR ablations. The real-world evaluation of these ablations is included in Table 1. We first analyze the group of ablations that fix a single choice of DR configuration or lack thereof. We see that our tasks clearly demand domain randomization as **No DR** is inferior to both `DrEureka` and `Human-Designed`. However, finding a suitable DR is not trivial. **Prompt DR** suggests wide parameter ranges (especially over friction as seen in prompts in Appendix) that forces the robot to over-exert forces; this result is validated in Figure 2 where we visualize the histogram of hip torque readings from real-world deployment of `DrEureka` policies versus **Prompt DR** policies. On the other hand, using **Human-Designed DR** does not match the performance of `DrEureka`, illustrating the importance of reward-aware domain randomization. Onto the sampling-based baselines, the subpar performance of **Random Sampling** suggests the effectiveness of LLMs as hypothesis generators, consistent with prior works that have found LLMs to be effective for suggesting initial samples for optimization problems [3, 12–15]. However, fully utilizing LLM's zero-shot generation capability requires proper grounding of the sampling space. **No Prior** and **Uninformative Prior**, despite using a LLM as sampler, performs very poorly and often results in policies that trigger safety protection power cutoff in the real world. One common concern for LLM-based solutions is data leakage, in which the LLM has seen the problems and solutions for an evaluation task. In our setting, if the LLM has seen the simulations tasks and consequently the `human-designed` ranges in the open-sourced code base, then even if the priors are withheld in the context, it should be possible to output reasonable ranges out of the box. Fortunately, the negative results of **No Prior** confirms that data leakage does not appear in our evaluation. Altogether, these results affirm that both reward-aware parameter priors and LLM as a hypothesis generator in the `DrEureka` framework are necessary for best real-world performance.

Safety Instruction	Velocity (Sim)	Velocity (Real)
Yes (`DrEureka` w.o DR)	1.70 ± 0.11	$\mathbf{1.21} \pm 0.39$
No (Eureka)	$\mathbf{1.83} \pm 0.05$	0.0 ± 0.0

Table 3. **`DrEureka` safety instruction ablation.** Omitting the safety instruction from `DrEureka` results in policies that run quickly in simulation but fail in the real world.

Sampling from `DrEureka` priors enables stable simulation training. Finally, to better understand the drastically different performances of different `DrEureka` prior choices in the real world, we present the simulation training curves in Figure 3. Note that the performances are not directly comparable as each method is trained and evaluated on its own DR distributions. Nevertheless, we observe the stable training progress of `DrEureka`. In contrast, despite using a LLM, the ablations synthesize poor DR ranges, resulting in difficult policy training dynamics.

Safety instruction enables safe reward functions. In addition to comparing against human-written reward functions, we also ablate `DrEureka`'s own reward design procedure. In particular, to verify that `DrEureka`'s safety instruction yields more deployable reward functions, we compare to an ablation of `DrEureka` that does not include custom safety suggestions in the prompt; see Appendix for the functional form of this reward function. Note that this ablation is identical to the original Eureka algorithm in Ta-

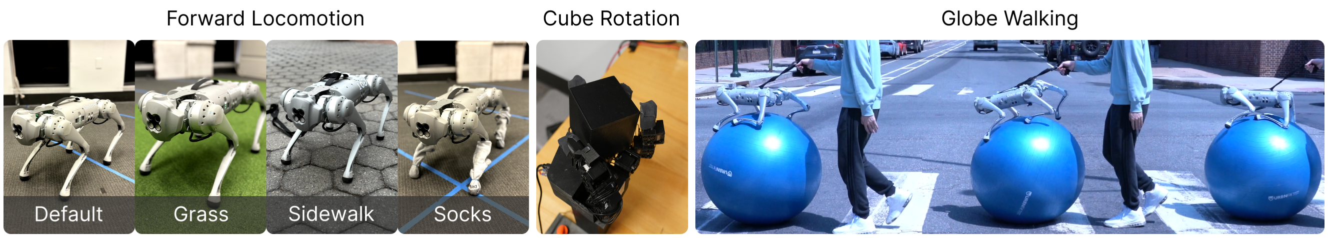

Figure 4. Our forward locomotion, cube rotation, and globe walking tasks.

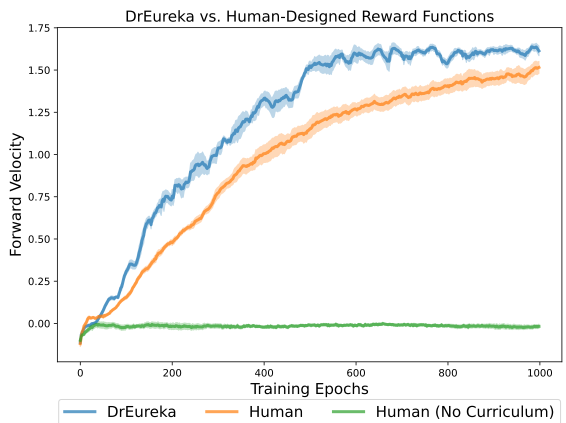

Figure 5. Comparison between `DrEureka` and `Human-Designed` reward functions on the simulation locomotion task. `DrEureka` has higher sample efficiency and asymptotic performance, while `Human-Designed` relies on a velocity curriculum to perform well.

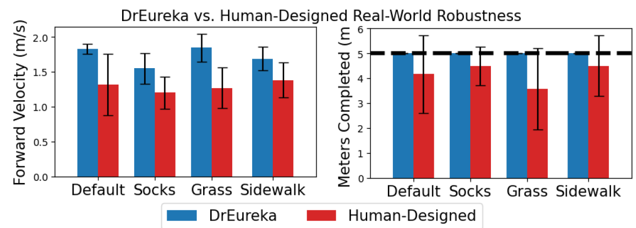

Figure 6. **Real-world robustness evaluation.** `DrEureka` performs consistently across different terrains and maintains advantages over `Human-Designed`.

ble 1, and we compare it to the `DrEureka` (No DR) variant to eliminate the influence of domain randomization in policy performance. As shown in Table 3, removing the safety prompt results in a final reward function that can move faster in simulation than `DrEureka`. However, the robot acquires an unnatural gait with three of its feet and the hip dragging on the ground. Consequently, in the real world, this behavior does not transfer, and the policy directly face-plants at the starting line; this is not surprising as the Eureka reward function contains just a generic action smoothing term for safety, which in itself does not prohibit awkward behaviors. See our supplementary material for a video comparison.

C. Qualitative Reward Analysis

Given the results of our experiments, we qualitatively analyze the `DrEureka` reward function R_{DrEureka} (i.e., the best reward function from the reward design stage). The mathematical expression is shown in Table 6, and the raw programmatic output from the LLM is reproduced in section 3 of the Appendix. We observe that this reward function is *multiplicative* of its components, a clear deviation from established reward functions for quadrupedal locomotion tasks that bear additive rewards [5, 16–19]. The multiplicative nature of `DrEureka` reward also introduces an interesting effect from the `DOF Violations` term, which is a binary function that indicates whether any robot joint exceeds the joint limit. Namely, if any joint violation occurs, then the entire reward for that time step is 0. Intuitively, this reward function encourages the policy to always learn within the space of safe behavior, as any violation is heavily penalized. While prior reward functions on locomotion tasks have considered a binary penalty term on joint limit violation [5], they often incorporate it as an additive penalty, which may not have a large effect on the behavior due to weight scaling. In summary, `DrEureka` reward is simple, *eccentric*, yet effective.

D. Effectiveness of `DrEureka` Sim-to-real Rewards

In this section, we compare `DrEureka`'s reward against baselines and ablations to conclude that `DrEureka` reward is at once effective, safe, and novel; `DrEureka`'s reward expression is captured in Table 6.

`DrEureka` does not need a reward curriculum. To study the effectiveness of the reward functions in isolation, we fix the domain randomization configurations to be `Human-Designed` for both `DrEureka` and `Human-Designed` reward functions and re-train several policies in simulation. Since `Human-Designed` reward utilizes a velocity curriculum, we also evaluate an ablation of the `Human-Designed` reward function that has a fixed velocity target (i.e., 2.0 m/s) to put it on an equal

Table 4. **Domain randomization parameters for cube rotation, along with their valid ranges and RAPP search ranges.**

footing with the Eureka reward function as a standalone reward function. The training curves are shown in Figure 5 in the Appendix. We find that `DrEureka` reward enables more sample-efficient training and reaches higher asymptotic performance. In contrast, the `Human-Designed` reward crucially depends on the explicit curriculum to work comparably; as a stand-alone reward function without curriculum inputs, `Human-Designed` makes little progress.

E. Full Prompts

In this section, we provide all `DrEureka` prompts used for experiments and ablations.

E1. Reward Generation Prompts

This section contains the system and task prompts for generating reward functions for forward locomotion and globe walking tasks using `DrEureka`.

```
You are a reward engineer trying to write reward functions to solve reinforcement learning tasks as effective as
     possible.
Your goal is to write a reward function for the environment that will help the agent learn the task described in text.
Your reward function should use useful variables from the environment as inputs. As an example,
the reward function signature can be: {task_reward_signature_string}
Make sure any new tensor or variable you introduce is on the same device as the input tensors.
```

Prompt 1. `DrEureka` system prompt for reward generation.

```
To make the go1 quadruped run forward with a velocity of exactly 2.0 m/s in the positive x direction of the global
     coordinate frame. The policy will be trained in simulation and deployed in the real world, so the policy should
     be as steady and stable as possible with minimal action rate. Specifically, as it's running, the torso should
     remain near a z position of 0.34, and the orientation should be perpendicular to gravity. Also, the legs should
     move smoothly and avoid the DOF limits.
```

Prompt 2. `DrEureka` forward locomotion task prompt for reward generation.

```
To make the go1 quadruped balance on the top of the ball. The quadruped should maintain a z-position of 2 *
     ball_radius or higher. Please keep in mind that the policy learned using your reward terms will be deployed on a
     robot in the real world. As such, you should prioritize safety, robustness, and feasibility over performance.
     Please generate reward terms that penalize actions that are unsafe or infeasible. Please also penalize jittery or
      fast actions that may burn out the motors. Also, remember to keep the scaling of your regularization terms small
     . If you choose to use env.torques, please keep in mind that this value will be large, so your scaling for this
     term should be near 0.00001.
```

Prompt 3. `DrEureka` globe walking task prompt for reward generation.

E2. Reward Generation Ablation Prompts

This section contains prompts used in ablation studies, specifically for generating reward functions without safety instructions to assess the impact of such instructions on the generated rewards.

```
The Python environment is {environment source code}. Write a reward function for the following task: To make the go1
     quadruped run forward with a velocity of exactly 2.0 m/s in the positive x direction of the global coordinate
     frame.
```

Prompt 4. `DrEureka` forward locomotion task prompt for reward generation, without safety instructions.

E3. Domain Randomization Generation Prompts

This section includes the initial system and user prompts for generating domain randomization configurations, demonstrating how `DrEureka` is applied to different tasks for robust policy training.

````
You are a reinforcement learning engineer. Your goal is to design a set of domain randomization parameters for the
     given task to facilitate successful deployment of the trained policy in the real world.
To do so, you will be given valid parameters as well as a range for each parameter that indicates the maximum and
     minimum values that parameter can take. Please note that your randomization ranges do not need to cover most of
     the range.
Also, you should keep in mind that the more you randomize, the more difficult it will be for the policy to learn the
     task within our fixed compute budget. A good policy should be trained only on randomization ranges that will help
      it adapt to the real world.
You should first reason over each parameter and determine if it's useful for domain randomization.
Then, you should output a range of values for each parameter that you think will be useful for the task in a real-
     world deployment. Please explain your reasoning for each parameter.

Output your response in the form of Python code that sets the parameters as variables, e.g.:
```
friction_range = [0.0, 1.0]
```
Please make your variable names match the parameter names provided. Each variable should be assigned a range formatted
      as a Python list with two elements. Write everything else as Python comments.
````

Prompt 5. `DrEureka` system prompt for DR generation.

```
The task is to train a quadruped robot to run on a variety of terrains indoor and outdoor. The goal of the robot is to
     run forward at 2.0 m/s while remaining steady and safe in the real world.
The robot will be trained in simulation and then deployed in the real world.
Our parameters and valid ranges are the following:
    friction_range = [0.0, 10.0]
    restitution_range = [0.0, 1.0]
    added_mass_range = [-5.0, 5.0]
    com_displacement_range = [-0.1, 0.1]
    motor_strength_range = [0.5, 2.0]
    Kp_factor_range = [0.5, 2.0]
    Kd_factor_range = [0.5, 2.0]
    dof_stiffness_range = [0.0, 1.0]
    dof_damping_range = [0.0, 0.5]
    dof_friction_range = [0.0, 0.01]
    dof_armature_range = [0.0, 0.01] (This is the range of values added onto the diagonal of the joint inertia matrix
     .)
    push_vel_xy_range = [0.0, 1.0]   (This is the range of magnitudes of a vector added onto the robot's xy velocity.)
    gravity_range = [-1.0, 1.0]      (This is the range of values added onto each dimension of [0.0, 0.0, -9.8].
                                      For example, [0.0, 0.0] would keep gravity constant.)
```

Prompt 6. `DrEureka` quadruped prompt with RAPP from `DrEureka` policy. This prompt corresponds to the 'Our Method' configuration in Table 1.

```
The task is to train a quadruped robot to balance on a yoga ball for as long as possible.
The robot will be trained in simulation and then deployed in the real world. Please note that our simulation
     environment models the ball as a solid rigid object, so the robot will not be able to deform the ball in any way.
      However, our real yoga ball is hollow, bouncy, and deformable, so the robot will need to adapt to this
     difference. Please keep this in mind when designing your domain randomization.
Our parameters and valid ranges are the following:
    robot_friction_range = [0.1, 1.0]
    robot_restitution_range = [0.0, 1.0]
    robot_payload_mass_range = [-1.0, 5.0]
    robot_com_displacement_range = [-0.1, 0.1]
    robot_motor_strength_range = [0.9, 1.1]
    robot_motor_offset_range = [-0.01, 0.1]
    ball_mass_range = [0.5, 5.0]
    ball_friction_range = [0.1, 3.0]
    ball_restitution_range = [0.0, 1.0]
    ball_drag_range = [0.0, 1.0]
    terrain_ground_friction_range = [0.0, 1.0]
    terrain_ground_restitution_range = [0.0, 1.0]
    terrain_tile_roughness_range = [0.0, 0.1]
    robot_push_vel_range = [0.0, 0.5]
    ball_push_vel_range = [0.0, 0.5]
    gravity_range = [-0.5, 0.5]
```

Prompt 7. `DrEureka` globe walking prompt with RAPP from `DrEureka` policy.

E4. Domain Randomization Generation Ablation Prompts

This section includes prompts used in ablation experiments that test the importance of RAPP priors in the LLM prompt. Below, we include a prompt with no prior context and a prompt whose context is the entire range tested by the RAPP algorithm.

```
The task is to train a quadruped robot to run on a variety of terrains indoor and outdoor. The goal of the robot is to
     run forward at 2.0 m/s while remaining steady and safe in the real world.
The robot will be trained in simulation and then deployed in the real world.
Our parameters are the following:
    friction_range
    restitution_range
    added_mass_range
    com_displacement_range
    motor_strength_range
    Kp_factor_range
    Kd_factor_range
    dof_stiffness_range
    dof_damping_range
    dof_friction_range
    dof_armature_range        (This is the range of values added onto the diagonal of the joint inertia matrix.)
    push_vel_xy_range         (This is the range of magnitudes of a vector added onto the robot's xy velocity.)
    gravity_range             (This is the range of values added onto each dimension of [0.0, 0.0, -9.8]. For example,
                              [0.0, 0.0] would keep gravity constant.)
```

Prompt 8. Initial quadruped prompt (no context). This prompt corresponds to the 'Without Prior' configuration in Table 1.

```
The task is to train a quadruped robot to run on a variety of terrains indoor and outdoor. The goal of the robot is to
      run forward at 2.0 m/s while remaining steady and safe in the real world.
The robot will be trained in simulation and then deployed in the real world.
Our parameters and valid ranges are the following:
    friction_range = [0.0, 10.0]
    restitution_range = [0.0, 1.0]
    added_mass_range = [-10.0, 10.0]
    com_displacement_range = [-10.0, 10.0]
    motor_strength_range = [0.0, 2.0]
    Kp_factor_range = [0.0, 2.0]
    Kd_factor_range = [0.0, 2.0]
    dof_stiffness_range = [0.0, 10.0]
    dof_damping_range = [0.0, 10.0]
    dof_friction_range = [0.0, 10.0]
    dof_armature_range = [0.0, 10.0]          (This is the range of values added onto the diagonal of the joint
     inertia matrix.)
    push_vel_xy_range = [0.0, 10.0]            (This is the range of magnitudes of a vector added onto the robot's xy
     velocity.)
    gravity_range = [-10.0, 10.0]               (This is the range of values added onto each dimension of [0.0, 0.0,
     -9.8]. For example, [0.0, 0.0] would keep gravity constant.)
```

Prompt 9. Initial quadruped prompt (uninformative context). This prompt corresponds to the 'With Uninformative Prior' configuration in Table 1.

F. `DrEureka` Outputs

In this section, we detail the reward functions generated by `DrEureka` and applied in the training of forward locomotion and globe walking task.

F1. LLM-Generated Rewards

```python
def compute_reward(self):
    env = self.env  # Do not skip this line. Afterwards, use env.{parameter_name} to access parameters of the
     environment.

    # Ideal forward velocity in the x direction
    target_velocity_x = 2.0
    # Ideal height of the robot's torso
    target_height_z = 0.34

    # Compute the velocity reward component
    current_velocity_x = env.root_states[:, 7]  # Linear velocity in x from the root_states tensor
    velocity_error = torch.abs(current_velocity_x - target_velocity_x)
    velocity_reward = torch.exp(-velocity_error)

    # Compute the height reward component
    current_height = env.root_states[:, 2]  # Position in z from the root_states tensor
    height_error = torch.abs(current_height - target_height_z)
    height_reward = torch.exp(-5.0 * height_error)  # More weight to maintain height

    # Compute the orientation reward component
    # Ideal orientation is perpendicular to gravity, i.e., the projected gravity vector should be [0, 0, -1] in the
     robot's frame
    ideal_projected_gravity = torch.tensor([0., 0., -1.], device=env.device).repeat((env.num_envs, 1))
    orientation_error = torch.norm(env.projected_gravity - ideal_projected_gravity, dim=1)
    orientation_reward = torch.exp(-5.0 * orientation_error)  # More weight to maintain orientation

    # Legs movement within DOF limits reward component
    dof_limit_violations = torch.any(
        (env.dof_pos < env.dof_pos_limits[:, 0]) | (env.dof_pos > env.dof_pos_limits[:, 1]),
        dim=-1)
    dof_limit_violations_reward = 1.0 - dof_limit_violations.float()  # Penalize if any DOF limit is violated

    # Smoothness reward component (penalize the change in actions to encourage smooth movements)
    action_difference = torch.norm(env.actions - env.last_actions, dim=1)
    smoothness_reward = torch.exp(-0.1 * action_difference)

    # Combine reward components
    total_reward = velocity_reward * height_reward * orientation_reward * dof_limit_violations_reward *
     smoothness_reward

    # Debug information
    reward_components = {"velocity_reward": velocity_reward,
                         "height_reward": height_reward,
```

```
                        "orientation_reward": orientation_reward,
                        "dof_limit_violations_reward": dof_limit_violations_reward,
                        "smoothness_reward": smoothness_reward}
    return total_reward, reward_components
```

Prompt 10. Final reward for forward locomotion task from DrEureka.

```
def _reward_height(self):
    env = self.env
    height_threshold = 2.0 * env.ball_radius
    height_temperature = 7.0  # Fine-tuned temperature parameter
    height_exp = torch.exp((env.base_pos[:, 2] - height_threshold) / height_temperature)
    height_reward = torch.where(env.base_pos[:, 2] >= height_threshold, height_exp, torch.zeros_like(env.base_pos[:,
     2]))
    return 1.5 * height_reward  # Updated scaling

def _reward_balance(self):
    env = self.env
    balance_temperature = 5.0  # Fine-tuned temperature parameter
    ball_top = env.object_pos_world_frame.clone()
    ball_top[:, 2] += env.ball_radius

    feet_dist_to_ball_top = torch.norm(env.foot_positions - ball_top.unsqueeze(1), dim=-1)
    balance_exp = torch.exp(-feet_dist_to_ball_top / balance_temperature)
    balance_reward = torch.mean(balance_exp, dim=-1)
    return 2.0 * balance_reward  # Updated scaling

def _reward_smooth_actions(self):
    env = self.env
    action_diff = env.actions - env.last_actions
    smooth_actions_reward = -torch.mean(torch.abs(action_diff), dim=-1)
    return 1.0 * smooth_actions_reward  # Increase scale of smooth_actions_reward

def _reward_penalize_large_actions(self):
    env = self.env
    large_action_penalty = -torch.mean(torch.abs(env.actions), dim=-1)
    return 0.3 * large_action_penalty  # Increase scaling for penalize_large_actions
```

Prompt 11. Final reward for globe walking task from DrEureka. Due to a limitation in the original environment's codebase, the Eureka reward format here splits each term into a separate function and computes the final reward as a sum of all terms. Besides a minimal change in the prompt to describe this format, everything else is the same.

F2. LLM-Generated Domain Randomizations

In this section, we provide the examples of domain randomization configurations generated by DrEureka given Reward-Aware Physics Prior.

```
# Friction is important as it affects how the robot interacts with different surfaces (indoor, outdoor).
friction_range = [0.5, 5.0]  # Moderate range to cover various surfaces like tiles, grass, dirt, etc.

# Restitution affects how the robot bounces off surfaces or objects; however, for a running task, this might be less
     critical.
restitution_range = [0.0, 0.5]  # Lower range as we're not focusing on bouncing, but it's still relevant for minor
     impacts.

# Added mass simulates the effect of carrying additional weight, which could influence stability and motor strength
     requirements.
added_mass_range = [-2.0, 2.0]  # A moderate range to simulate carrying light payloads or none.

# Center of mass displacement affects stability and maneuverability.
com_displacement_range = [-0.05, 0.05]  # Slight displacements to ensure the model can handle minor imbalances.

# Motor strength directly affects the robot's ability to move and respond to commands.
motor_strength_range = [0.75, 1.75]  # A moderate range allowing for variability in motor performance.

# Kp_factor (proportional gain) and Kd_factor (derivative gain) influence the control loop's responsiveness and
     stability.
Kp_factor_range = [0.75, 1.5]  # A range that allows for experimenting with control responsiveness without becoming
     unstable.
Kd_factor_range = [0.75, 1.5]  # Similar to Kp to ensure controllability under various conditions.

# Degrees of freedom (DOF) stiffness and damping affect how rigid and shock-absorbing the joints are, respectively.
```

```
dof_stiffness_range = [0.2, 0.8]  # A range that allows for some variation in joint stiffness without being too rigid
     or floppy.
dof_damping_range = [0.1, 0.4]  # Provides variability in how the robot absorbs shocks and returns to equilibrium.

# DOF friction and armature values are very specific parameters that could affect energy efficiency and precision in
     movements.
dof_friction_range = [0.0, 0.005]  # Small range based on the minor impact it has on performance but still considering
      efficiency.
dof_armature_range = [0.0, 0.005]  # Minimally varies the joint inertia to see the effects on stability and precision.

# Push velocity in the xy plane simulates external forces or disturbances, e.g., wind or collisions.
push_vel_xy_range = [0.0, 0.5]  # A range to help the robot learn to stabilize against moderate disturbances.

# Gravity variations can simulate different loading conditions or slight inclines/declines the robot might face.
gravity_range = [-0.2, 0.2]  # Small tweaks to represent minor slopes or variations the robot might need to adapt to.

# Note: Each range is chosen with the thought of exposing the robot to a variety of conditions it might face in the
     real world
# without making the task unnecessarily difficult to learn within our compute budget.
```

Prompt 12. Example DrEureka DR configuration for forward locomotion.

```
# Friction between the robot and the ball is crucial because it affects how well the robot can balance and maneuver on
      the ball. Since the real ball can be less predictable, a wider range should promote adaptability.
robot_friction_range = [0.1, 1.0]

# Restitution, or bounciness, will affect how the robot interacts with surfaces upon collision. Though the simulation
     does not account for ball deformation, varying restitution can simulate the unpredictability of these
     interactions.
robot_restitution_range = [0.2, 0.8]

# Since the payload will directly affect the robot's balance and how it responds to shifts in weight, we allow for
     variability but avoid extreme negative values to maintain realism.
robot_payload_mass_range = [0.0, 3.0]

# Center of mass displacement affects balance and stability. Randomization within a moderate range can prepare the
     robot for shifts in its own weight distribution.
robot_com_displacement_range = [-0.05, 0.05]

# Motor strength is critical for moving and balancing. A narrow range ensures the robot remains capable of movement
     but can adapt to variability in its actuation power.
robot_motor_strength_range = [0.95, 1.05]

# Motor offsets will simulate imperfections in actuator performance. Randomizing this could prepare the robot for real
     -world inaccuracies.
robot_motor_offset_range = [-0.005, 0.05]

# The ball's mass will significantly impact how the robot interacts with it. Since the ball is hollow and can be
     deformed, a middle-range should provide a good balance between too light and too heavy.
ball_mass_range = [1.0, 3.0]

# Ball friction and restitution are critical for preparing the robot to interact with a bouncy and deformable ball.
     These ranges allow for significant variability.
ball_friction_range = [0.5, 2.5]
ball_restitution_range = [0.4, 0.9]

# Ball drag simulates air resistance, which could affect interactions at higher speeds.
ball_drag_range = [0.1, 0.5]

# The robot might not always operate on similar terrains, so simulating a range of frictions can be beneficial.
     However, the restitution of the ground is less critical here.
terrain_ground_friction_range = [0.2, 0.8]
terrain_ground_restitution_range = [0.0, 0.5]

# Terrain roughness could influence balance and traction, so a slight variation can introduce realistic challenges
     without overwhelming the learning process.
terrain_tile_roughness_range = [0.02, 0.08]

# Varying the push velocities can help the robot learn to maintain balance against unexpected forces.
robot_push_vel_range = [0.1, 0.4]
ball_push_vel_range = [0.1, 0.4]

# Considering the task does not involve drastic changes in gravity, we only slightly vary this to simulate minor
     differences in weight sensation.
gravity_range = [-0.1, 0.1]
```

Prompt 13. Example DrEureka DR configuration for globe walking.

G. Mathematical Representation of DrEureka Rewards

In this section, we convert the programmatic human-written and LLM-generated reward functions into mathematical expressions for comparison.

Symbol	Explanation
v_x^t, v_x	Agent's and target's linear velocity along the x-axis.
ω_z^t, ω_z	Agent's and target's angular velocity around the z-axis.
v_z	Velocity along the z-axis.
ω_{xy}	Velocities in the roll and pitch directions.
p_z^t, p_z	Agent's and target's base height.
g_{xy}	Base orientation in the horizontal plane.
j, j_l, j_h	Joint position and lower, upper joint limits.
τ	Applied torques.
$\ddot{j}$	Joint acceleration.
a_t, a_{t-1}	Consecutive actions to measure smoothness and action rate.
t_{air}	Feet airtime during next contact transitions.
$foot\_position, ball\_top\_position$	3D Positions of the robot foot and the top of the ball.

Table 5. Explanation of Symbols Used in Reward Function Tables.

Term	Symbol
`Linear velocity tracking`	$0.02 * \exp\{-(v_x - v_x^t)^2/0.25\}$
`Angular velocity tracking`	$0.01 * \exp\{-(\omega_z - \omega_z^t)^2/0.25\}$
`Z-velocity penalty`	$-0.04 * v_z^2$
`Roll-pitch-velocity penalty`	$-0.001 * \|\omega_{xy}\|^2$
`Base height penalty`	$-0.6 * (p_z - p_z^t)^2$
`Base orientation penalty`	$-0.1 * \|g_{xy}\|^2$
`Collision penalty`	$-0.02 * \mathbf{1}[\text{collision}]$
`Joint limit penalty`	$-0.2 * (\max(0, j_l - j) + \max(0, j - j_h))$
`Torque penalty`	$-2e-6 * \|\tau\|^2$
`Joint acceleration penalty`	$-5e-9 * \|\ddot{j}\|^2$
`Action rate penalty`	$-2e-4 * \|a_t - a_{t-1}\|^2$
`Feet airtime`	$0.02 * \sum t_{air} * \mathbf{1}[\text{next contact}]$

Table 7. **Human-written reward function for forward locomotion.** The total reward is the sum of the components above.

Term	Symbol
`Forward velocity`	$\exp\{-(v_x - v_x^t)^2/2\}$
`Action smoothness`	$-0.25 * \|a_t - a_{t-1}\|$
`Angular velocity`	$-0.25 * \\|\omega_{xyz}\\|_2$
`Eureka reward`	`Forward velocity` + `Action smoothness` + `Angular velocity`

Table 8. **Final reward for forward locomotion from Eureka without safety instruction.**

Term	Symbol
`Velocity`	$\exp\{-(\|v_x - v_x^t\|)\}$
`Height`	$\exp\{-5.0 * \|p_z - p_z^t\|\}$
`Orientation`	$\exp\{-5.0 * \\|g_{xy} - g_{xy}^t\\|_2\}$
`DOF violations`	$1.0 - \mathbf{1}[j < j_l \cup j > j_h]$
`Action smoothness`	$\exp\{-0.1 * \\|a_t - a_{t-1}\\|_2\}$
`DrEureka` reward	`velocity` * `height` * `orientation` * `DOF violations` * `action smoothness`

Table 6. **`DrEureka` reward function for quadruped locomotion.** The cumulative reward is a product of the terms above.

Term	Symbol
`Height`	$1.5 * \mathbb{1}_{\{p_z^t > p_z\}} * \exp\{\frac{p_z^t - p_z}{7}\}$
`Balance`	$2 * \exp\{\frac{-\|foot\_position - ball\_top\_position\|}{5}\}$
`Action smoothness`	$-1 * \|a_t - a_{t-1}\|$
`Large Action Penalty`	$-0.3 * \|a_t\|$
`Eureka reward`	`Height` + `Balance` + `Action smoothness` + `Large Action Penalty`

Table 9. **Final reward for the walking globe task.**

H. Experimental Setup

H1. Forward Locomotion

For the forward locomotion task, our policy takes joint positions, joint velocities, a gravity vector, and a history of past observations and actions as input. It produces joint position commands for a PD controller, which has a proportional gain of 20 and derivative gain of 0.5.

We extend the simulation setup from Margolis et al. [5], and we include additional domain randomization parameters, specifically joint stiffness, damping, friction, and armature that were not in the their work. These parameters, along with the others in Table 10, were randomized during training. We chose these parameters based on IsaacGym's documentation on rigid body, rigid shape, and DOF properties[2].

Property	Valid Range	RAPP Search Range
`friction`	$[0, \infty)$	$[0, 10]$
`restitution`	$[0, 1]$	$[0, 1]$
`payload mass`	$(-\infty, \infty)$	$[-10, 10]$
`center of mass displacement`	$(-\infty, \infty)$	$[-10, 10]$
`motor strength`	$[0, \infty)$	$[0, 2]$
`scaling factors for proportional gain`	$[0, \infty)$	$[0, 2]$
`scaling factors for derivative gain`	$[0, \infty)$	$[0, 2]$
`push velocity`	$[0, \infty)$	$[0, 10]$
`gravity`	$(-\infty, \infty)$	$[-10, 10]$
`dof stiffness`	$[0, \infty)$	$[0, 10]$
`dof damping`	$[0, \infty)$	$[0, 10]$
`dof friction`	$[0, \infty)$	$[0, 10]$
`dof armature`	$[0, \infty)$	$[0, 10]$

Table 10. **Domain randomization parameters for forward locomotion, along with their valid ranges and RAPP search ranges.** Though the scale of these parameters differs, each RAPP range is chosen from one of four general-purpose ranges (`0_to_infty`, `0_to_1`, `centered_0`, `centered_1`).

H2. Globe Walking

For globe walking, we largely extend the framework from forward locomotion, with a few exceptions. First, the policy takes in an additional yaw sensor as input. Second, to account for actuator inaccuracies in the real world, we use an actuator network from Ji et al. [20]; this network is pretrained on log data to predict real robot torques from joint commands, and we use it to compute torques from actions in simulation when training the quadruped. Third, we have additional domain randomization parameters, shown in Table 11.

In the real world, we deploy our quadruped on a 34-inch yoga ball. We did not have a stable pole to tether our quadruped, so we instead resort to a human holding the end of the leash; however, we are careful to hold the leash parallel to the ground to ensure that the human does not provide any upward force that might aid the robot, and our sole purpose is to keep the robot within a safe radius.

[2] Relevant functions in the documentation are `isaacgym.gymapi.RigidBodyProperties`, `isaacgym.gymapi.RigidShapeProperties`, `isaacgym.gymapi.Gym.get_actor_dof_properties()`. Note that among these properties, there are a few fields that we found had no effect in simulation. We discarded them for our domain randomization.

Property	Valid Range	RAPP Search Range
`robot friction`	$[0, \infty)$	$[0, 10]$
`robot restitution`	$[0, 1]$	$[0, 1]$
`robot payload mass`	$(-\infty, \infty)$	$[-10, 10]$
`robot center of mass displacement`	$(-\infty, \infty)$	$[-10, 10]$
`robot motor strength`	$[0, \infty)$	$[0, 2]$
`robot motor offset`	$(-\infty, \infty)$	$[-10, 10]$
`ball mass`	$[0, \infty)$	$[0, 10]$
`ball friction`	$[0, \infty)$	$[0, 10]$
`ball restitution`	$[0, 1]$	$[0, 1]$
`ball drag`	$[0, \infty)$	$[0, 10]$
`terrain friction`	$[0, \infty)$	$[0, 10]$
`terrain restitution`	$[0, 1]$	$[0, 1]$
`terrain roughness`	$[0, \infty)$	$[0, 10]$
`robot push velocity`	$[0, \infty)$	$[0, 10]$
`ball push velocity`	$[0, \infty)$	$[0, 10]$
`gravity`	$(-\infty, \infty)$	$[-10, 10]$

Table 11. **Domain randomization parameters for globe walking, along with their valid ranges and RAPP search ranges.**