Harnessing Heuristics for Deep Reinforcement Learning via Constrained Optimization

Chi-Chang Lee[2]*, Zhang-Wei Hong[1]* , Pulkit Agrawal[1]
Improbable AI Lab at Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Abstract

In many reinforcement learning (RL) applications, incorporating heuristic signals alongside the exact task objective is crucial for achieving desirable performance. However, heuristics can occasionally lead to biased and suboptimal policies for the exact task objective. Common strategies to enhance performance involve modifying the training objective to ensure that the optimal policies for both the heuristic and the exact task objective remain invariant. Despite this, these strategies often underperform in practical scenarios with finite training data. This paper explores alternatives for improving task performance in finite data settings using heuristic signals. Instead of ensuring optimal policy invariance, we aim to train a policy that surpasses one trained solely with heuristics. We propose a constrained optimization procedure that uses the heuristic policy as a reference, ensuring the learned policy always outperforms the heuristic policy on the exact task objective. Our experiments on robotic locomotion, helicopter, and manipulation tasks demonstrate that this method consistently improves performance, regardless of the general effectiveness of the heuristic signals.

1 Introduction

Training a policy using reinforcement learning (RL) with sparse or delayed rewards is often ineffective. Instead of relying solely on sparse task rewards that indicate whether the agent succeeded or failed, it is common practice to create *heuristic* reward functions that provide denser reward supervision to facilitate RL training. Manual design of reward functions has proven particularly useful in solving complex robotic tasks involving object manipulation [1] and locomotion [2–4]. For example, consider the task of door-opening - a heuristic reward function may reward the robot for approaching the door, locating the handle, and pulling the handle, making RL training easier than using rewards only for task completion. However, training with heuristic rewards can be sub-optimal, as the robot might focus on an intermediate sub-task, such as locating the handle, instead of completing the overall task.

The current dominant paradigm of finding good heuristic rewards is to iterate and refine multiple reward functions until a satisfactory one is found. However, this process is time-consuming and requires substantial domain knowledge. Given the prevalence of the tedious but necessary practice of reward design, it is desirable to have an algorithm that frees the reward designer from worrying about when a heuristic will hurt or benefit the agent's performance. It would be ideal if the reward designer could think of "heuristic rewards" simply as "hints" provided to the agent to aid training, and the training algorithm can automatically figure out how to utilize the hint to maximize task performance. Such an algorithm would simplify reward design and help agents achieve high task performance.

*indicates equal contribution. [1] Improbable AI Lab, MIT, Cambridge, USA. [2] National Taiwan University, Taiwan.

One approach to address this problem is to combine the task objective $J(\pi)$ and the heuristic objective $H(\pi)$ into a mixed training objective $J(\pi) + \lambda H(\pi)$, where λ controls the balance between the two objectives for a policy π. This method aims to maximize both objectives with the hope of optimizing the task objective J. However, optimizing this mixed objective requires careful tuning of λ; otherwise, the algorithm might prioritize heuristic rewards while neglecting task objective. To avoid the effort of tuning λ, prior works [5, 6] propose training policies with this mixed objective while ensuring that the optimal policy for $J(\pi) + \lambda H(\pi)$ remains the same as for $J(\pi)$ (i.e., optimal policy invariance [5]). However, both our study and recent works [7] indicate that these approaches [5, 7, 6] empirically perform even worse than policies trained solely with heuristic objectives on complex robotic tasks.

Our key observation is that the guarantee of optimal policy invariance rarely holds true in practice when RL algorithms are trained in a finite data regime. This is because the guarantee relies on the optimal value function [5], which is unattainable in finite data settings. To improve beyond heuristic policies, our key insight is that *ensuring optimal policy invariance is unnecessary and impractical*. Instead, the necessary condition for improving beyond the heuristic policy π_H is performing policy improvement steps on task rewards, i.e., learning a policy π such that $J(\pi) \geq J(\pi_H)$, which deep RL algorithms [8–11] can achieve with finite data. Despite being feasible in practice, this may still be difficult when task rewards are sparse or delayed. Thus, the question is: How do we use this insight to enhance the policy's performance beyond heuristic policies?

The main challenge in improving a policy's performance with heuristic rewards is ensuring that the original task reward is maximized. Intuitively, we aim to use the heuristic reward for training when it enhances task performance and disregard it when it does not. Rather than manually tuning the weight coefficient λ between the two objectives, we translate the policy improvement requirement $(J(\pi) \geq J(\pi_H))$ into a constraint to prevent the policies from exploiting heuristic rewards. This constraint ensures that the learned policy π achieves higher task objectives J than the policy π_H trained solely with heuristic rewards, while still maximizing both rewards during training.

In this paper, we propose to translate this idea to the constrained optimization objective as follows:

$$\max_{\pi} J(\pi) + H(\pi) \quad \text{subject to} \quad J(\pi) \geq J(\pi_H)$$

Optimizing this objective at each iteration allows learning a policy performing better than or equal to policies trained only on heuristic rewards. It prevents capitalizing on heuristic rewards at the expense of task rewards. Moreover, it enables adaptively balancing both rewards over time instead of using a fixed coefficient. Our contribution is an add-on to existing deep RL algorithms to improve RL algorithms trained with heuristic rewards. We evaluated our method on robotic locomotion, helicopter, and manipulation tasks using the IsaacGym simulator [12]. The results show that our method led to superior task rewards and higher task-completion success rates compared to the policies solely trained with heuristic rewards, even when heuristic rewards are ill-designed.

2 Preliminaries: Reinforcement Learning with Heuristic

Reinforcement Learning (RL): RL is a popular paradigm for solving sequential decision-making problems [13] where the problems are modeled as an interaction between an agent and an unknown environment [13]. The agent aims to improve its performance through repeated interactions with the environment. At each round of interaction, the agent starts from the environment's initial state s_0 and samples the corresponding trajectory. At each timestep t within that trajectory, the agent perceives the state s_t, takes an action $a_t \sim \pi(.|s_t)$ according to its policy π, receives a *task* reward $r_t = r(s_t, a_t)$, and transitions to a next state s_{t+1} until reaching terminal states, after which a new trajectory is initialized from s_0, and the cycle repeats. The agent's goal is to learn a policy π that maximizes the expected return $J(\pi)$ in a trajectory as below:

$$J(\pi) = \mathbb{E}_{\pi}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)], \tag{1}$$

where γ denotes a discount factor [13] and $\mathbb{E}_{\pi}[.]$ denotes taking expectation over the trajectories sampled by π. In the following, we term J as the true *task* objective, as it indicates the performance of a policy on the task.

RL with Heuristic: In many tasks, learning a policy to maximize the true objective J is challenging because rewards may be sparse or delayed. This lack of feedback makes policy optimization difficult for RL algorithms. To address this, practitioners often use a heuristic reward function h with denser reward signals to facilitate optimization, aiming to learn a policy that performs better in J. The policy trained to maximize the expected return of heuristic rewards is called the *heuristic* policy π_H. The expected return of heuristic rewards, termed the *heuristic* objective H, is defined as:

$$H(\pi_H) = \mathbb{E}_{\pi_H}\left[\sum_{t=0}^{\infty} \gamma^t h(s_t, a_t)\right], \tag{2}$$

where $h(s_t, a_t)$ is the heuristic reward at timestep t for state s_t and action a_t.

3 Method: Improving Heuristic Policy via Constrained Optimization

Problem statement: Optimizing both task J and heuristic H objectives jointly could lead to better task performance than training solely with J or H, but needs careful tuning on the weight coefficient λ among both objectives in $\max_\pi J(\pi) + \lambda H(\pi)$. Without careful tuning, the policy π may learn to exploit heuristic rewards H and compromise performance of J. The goal of this paper is to mitigate the requirement of tuning this coefficient to balance them in order to improve task performance.

Key insight - Leveraging Heuristic with Constraint: We aim to use the heuristic objective H for training only when it improves task performance J and ignore it otherwise. Rather than manually tuning the weight coefficient λ to balance both rewards, we introduce a key insight: impose a *policy improvement* constraint (i.e., $J(\pi) \geq J(\pi_H)$) during training. This prevents RL algorithms from exploiting heuristic rewards H at the expense of task rewards J. To achieve this goal, we introduce the following constrained optimization objective:

$$\max_\pi J(\pi) + H(\pi) \text{ subject to } J(\pi) \geq J(\pi_H). \tag{3}$$

This constrained objective (Equation 3) results in an improved policy π over the heuristic policy π_H, leading us to call this framework *Heuristic-Enhanced Policy Optimization (HEPO)*. A practical algorithm to optimize this objective is presented in Section 3.1, and its implementation on a widely-used RL algorithm [8] in robotics is detailed in Section 3.2.

3.1 Algorithm: Heuristic-Enhanced Policy Optimization (HEPO)

Finding feasible solutions for the constrained optimization problem in Equation 3 is challenging due to the nonlinearity of the objective function J with respect to π. One practical approach is to convert it into the following unconstrained min-max optimization problem using Lagrangian duality:

$$\min_{\alpha \geq 0} \max_\pi \mathcal{L}(\pi, \alpha), \text{ where } \mathcal{L}(\pi, \alpha) := J(\pi) + H(\pi) + \alpha\left(J(\pi) - J(\pi_{\mathrm{H}})\right), \tag{4}$$

where the Lagrangian multiplier is $\alpha \in \mathbb{R}^+$. We can optimize the policy π and the multiplier α for this min-max problem by a gradient descent-ascent strategy, alternating between optimizing π and α.

Enhanced policy π: The optimization objective for the policy π can be obtained by rearranging Equation 4 as follows:

$$\begin{aligned} &\max_\pi \ (1+\alpha)J(\pi) + H(\pi), \\ &\text{where } (1+\alpha)J(\pi) + H(\pi) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t \big((1+\alpha)r(s_t, a_t)) + h(s_t, a_t)\big)\right]. \end{aligned} \tag{5}$$

This represents an unconstrained regular RL objective with the modified reward at each step as $(1+\alpha)r(s_t, a_t) + h(s_t, a_t)$, which can be optimized using any off-the-shelf deep RL algorithm. In this modified reward, the task reward $r(s_t, a_t)$ is weighted by the Lagrangian multiplier α, reflecting the potential variation in the task reward's importance during training as α evolves. The interaction between the update of the Lagrangian multiplier and the policy will be elaborated upon next.

Lagrangian Multiplier α: The Lagrangian multiplier α is optimized for Equation 4 by stochastic gradient descent, with the gradient defined as:

$$\nabla_{\alpha}\mathcal{L}(\pi, \alpha) = J(\pi) - J(\pi_{\text{H}}). \tag{6}$$

Notably, $\nabla_{\alpha}\mathcal{L}(\pi, \alpha)$ is exactly the performance gain of the policy π over the heuristic policy π_H on the task objective J. By applying gradient descent with $\nabla_{\alpha}\mathcal{L}(\pi, \alpha)$, when $J(\pi) > J(\pi_H)$ and thus $\nabla_{\alpha}\mathcal{L}(\pi, \alpha) > 0$, the Lagrangian multiplier α decreases. As α represents the weight of the task reward in Equation 5, it indicates that when π outperforms π_H, the importance of the task reward diminishes because π already achieves superior performance compared to the heuristic policy π_H regarding the task objective J. Conversely, when $J(\pi) < J(\pi_H)$, α increases, thereby emphasizing the importance of task rewards in optimization. The update procedure for the Lagrangian multiplier α offers an adaptive reconciliation between the heuristic reward h and the task reward r.

3.2 Implementation

We present a practical approach to optimize the min-max problem in Equation 4 using Proximal Policy Optimization (PPO) [8]. We selected PPO because it is widely used in robotic applications involving heuristic rewards, although our HEPO framework is not restricted to PPO. The standard PPO implementation involves iterative stochastic gradient descent updates over numerous iterations, alternating between collecting trajectories with policies and updating those policies. We outline the optimization process for each iteration and provide a summary of our implementation in Algorithm 1.

Training policies π and π_H: Instead of pre-training the heuristic policy π_H, which requires additional data and reduces data efficiency, we concurrently train both the enhanced policy π and the heuristic policy π_H, allowing them to share data. For each iteration i, we gather trajectories τ and τ_H using the enhanced policy π^i and the heuristic policy π^i_H, respectively. Following PPO's implementation, we compute the advantages $A^{\pi^i}_r(s_t, a_t)$, $A^{\pi^i_H}_r(s_t, a_t)$, $A^{\pi^i}_h(s_t, a_t)$, and $A^{\pi^i_H}_h(s_t, a_t)$ for the task reward r and heuristic reward h with respect to π^i and π^i_H. We then weight the advantage with the action probability ratio between the new policies being optimized (i.e., π^{i+1} and π^{i+1}_H) and the policies collecting the trajectories (i.e., π^i or π^i_H). Finally, we optimize the policies at the next iteration $i+1$ for the objectives in Equations 7 and 8:

$$\begin{aligned}
\pi^{i+1} \leftarrow \arg\max_{\pi}\ & \mathbb{E}_{\tau\sim\pi^i}\left[\frac{\pi(a_t|s_t)}{\pi^i(a_t|s_t)}\Big((1+\alpha)A^{\pi^i}_r(s_t,a_t) + A^{\pi^i}_h(s_t,a_t)\Big)\right] + & (7)\\
& \mathbb{E}_{\tau_H\sim\pi^i_H}\left[\frac{\pi(a_t|s_t)}{\pi^i_H(a_t|s_t)}\Big((1+\alpha)A^{\pi^i_H}_r(s_t,a_t) + A^{\pi^i_H}_h(s_t,a_t)+\Big)\right] & \text{(Enhanced policy)}\\
\pi^{i+1}_H \leftarrow \arg\max_{\pi}\ & \mathbb{E}_{\tau_H\sim\pi^i_H}\left[\frac{\pi(a_t|s_t)}{\pi^i_H(a_t|s_t)}A^{\pi^i}_h(s_t,a_t)\right] + & (8)\\
& \mathbb{E}_{\tau\sim\pi^i}\left[\frac{\pi(a_t|s_t)}{\pi^i(a_t|s_t)}A^{\pi^i_H}_h(s_t,a_t)\right] & \text{(Heuristic policy).}
\end{aligned}$$

Maximizing the advantages will result in a policy that maximizes the expected return for a chosen reward function, as demonstrated in PPO [8]. This enables us to maximize the objective J and H. We estimate the advantages $A^{\pi^i}_r(s_t, a_t)$ and $A^{\pi^i}_h(s_t, a_t)$ (or $A^{\pi^i_H}_r(s_t, a_t)$ and $A^{\pi^i_H}_h(s_t, a_t)$) using the standard PPO implementation with different reward functions. Therefore, we omit the details of the advantage's clipped surrogate objective in PPO, and leave them in Appendix A.1.

Although PPO is an on-policy algorithm, the use of off-policy importance ratio correction (i.e., the action probability ratios between two policies) allows us to use states and actions generated by another policy. This enables us to train π using data from π_H and vice versa. Both policies π and π_H are trained using the same data but with different reward functions. Note that collecting trajectories from both policies does not require more data than the standard PPO implementation. We collect half the trajectories with each policy, π and π_H, for a total of B trajectories (see Algorithm 1). Then, we update both π and π_H using all B trajectories.

Optimizing the Lagrangian multiplier α: To update the Lagrangian multiplier α, we need to compute the gradient in Equation 6, which corresponds to the performance gain of the enhanced policy π over the heuristic policy π_H on the task objective J. Utilizing the performance difference lemma [14, 9], we relate this improvement to the expected advantages over trajectories sampled by the enhanced policy π as $J(\pi) - J(\pi_H) = \mathbb{E}_\pi\left[A^{\pi_H} r(s_t, a_t)\right]$. However, this approach only utilizes half of the trajectories at each iteration since it exclusively relies on trajectories from the enhanced policy π. To leverage trajectories from both policies, we also consider the performance gain in the reverse direction as $-(J(\pi_H) - J(\pi)) = -\mathbb{E}_{\pi_H}\left[A^{\pi} r(s_t, a_t)\right]$. Consequently, we can estimate the gradient of α using trajectories from both policies, as illustrated below:

$$\nabla_\alpha \mathcal{L}(\pi, \alpha) = J(\pi) - J(\pi_H) = \mathbb{E}_\pi\left[A_r^{\pi_H}(s_t, a_t)\right] \tag{9}$$

$$= -(J(\pi_H) - J(\pi)) = -\mathbb{E}_{\pi_H}\left[A_r^{\pi}(s_t, a_t)\right]. \tag{10}$$

At each iteration i, we estimate the gradient of α using the advantage $A_r^{\pi_H}(s_t, a_t)$ and $A_r^{\pi}(s_t, a_t)$ on the trajectories sampled from both π^i and π_H^i, and update α with stochastic gradient descent as follows:

$$\alpha \leftarrow \alpha - \frac{\eta}{2}\left(\mathbb{E}_{\tau \sim \pi^i}\left[A_r^{\pi_H^i}(s_t, a_t)\right] - \mathbb{E}_{\tau \sim \pi_H^i}\left[A_r^{\pi^i}(s_t, a_t)\right]\right), \tag{11}$$

where $\eta \in \mathbb{R}^+$ is the step size. The expected advantage in Equation 11 are estimated using the generalized advantage estimator (GAE) [15].

Algorithm 1 Heuristic-Enhanced Policy Optimization (HEPO)

```
1: Input: Number of trajectories per iteration $B$
2: Initialize the enhanced policy $\pi^0$, the heuristic policy $\pi_H^0$, and the Lagrangian multiplier $\alpha$
3: for $i = 0 \cdots$ do                                          ▷ $i$ denotes iteration index
4:     Rollout $B/2$ trajectories $\tau$ by $\pi^i$
5:     Rollout $B/2$ trajectories $\tau_H$ by $\pi_H^i$
6:     $\pi^{i+1} \longleftarrow$ Train the policy $\pi^i$ for optimizing Equation 7 using both $\tau$ and $\tau_H$
7:     $\pi_H^{i+1} \longleftarrow$ Train the policy $\pi_H^i$ for optimizing Equation 8 using both $\tau$ and $\tau_H$
8:     $\alpha \longleftarrow$ Update the Lagrangian multiplier $\alpha$ by gradient descent (Equation 9) using $\tau$ and $\tau_H$
9: end for
```

3.3 Connection to Extrinsic-Intrinsic Policy Optimization [6]

Closely related to our HEPO framework, Chen et al. [6] proposes Extrinsic-Intrinsic Policy Optimization (EIPO), which trains a policy to maximize both task rewards and exploration bonuses [16] subject to the constraint that the learned policy π must outperform the *task* policy π_J trained solely on task rewards. HEPO and EIPO differ in their objective functions and implementation of the constrained optimization problem. Additional information can be found in the Appendix, covering the objective formulation (Appendix A.1), implementation tricks (Appendix A.2), and detailed pseudocode (Appendix A.3).

Exploration bonuses [16] can be viewed as heuristic rewards. The main difference between HEPO and EIPO's optimization objectives lies in constraint design. Both frameworks require the learned policy π to outperform a reference policy π_{ref} (i.e., $J(\pi) \geq J(\pi_{\text{ref}})$) but use a different reference policy. EIPO uses the task policy π_J as the reference policy π_{ref} because they aim for asymptotic optimality in task rewards. If the constraint is satisfied with π_J being the optimal policy for task rewards, the learned policy π will also be optimal for task rewards. In contrast, HEPO uses the heuristic policy π_H trained solely on heuristic rewards since HEPO aims to improve upon it.

HEPO simplifies the implementation. Both HEPO and EIPO train two policies with shared data, but EIPO alternates the policy used for trajectory collection each iteration and has a complex switching rule, which introduces more hyperparameters. HEPO collects trajectories using both policies together at each iteration, simplifying implementation and avoiding extra hyperparameters.

4 Experiments

We evaluate whether HEPO enhances the performance of RL algorithms in maximizing task rewards while training with heuristic rewards. We conduct experiments on 9 tasks from IsaacGym (ISAAC) [12] and 20 tasks from the Bidexterous Manipulation (BI-DEX) benchmark [17]. These tasks rely on heavily engineered reward functions for training RL algorithms. Each task has a task reward function r that defines the task objective J to be maximized, and a heuristic reward function h that defines the heuristic objective H, provided in the benchmarks to facilitate the optimization of task objectives J. We implement HEPO based on PPO [8] and compare it with the following baselines:

- **H-only (heuristic only)**: This is the standard PPO baseline provided in ISAAC. The policy is trained solely using the heuristic reward: $\max_\pi H(\pi)$. The heuristic reward functions in ISAAC and BI-DEX are designed to help RL algorithms maximize the task objective J. This baseline is crucial to determine if an algorithm can surpass a policy trained with highly engineered heuristic rewards.
- **J-only (task only)**: The policy is trained using only the task reward: $\max_\pi J(\pi)$. This baseline demonstrates the performance achievable without heuristics. Ideally, algorithms that incorporate heuristics should outperform this baseline.
- **J+H (mixture of task and heuristic)**: The policy is trained using a mixture of task and heuristic rewards: $\max_\pi J(\pi) + \lambda H(\pi)$, with λ balancing the two rewards. As [6] shows, proper tuning of λ can enhance task performance by balancing both training objectives.
- **Potential-based Reward Shaping (PBRS) [5]**: The policy is trained to maximize $\mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t + \gamma h_{t+1} - h_t]$, where r_t and h_t are the task and heuristic rewards at timestep t. PBRS guarantees that the optimal policy is invariant to the task reward function. We include it as a baseline to examine if these theoretical guarantees hold in practice.
- **HuRL [7]**: The policy is trained to maximize $\mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t + (1 - \beta_i)\gamma h_{t+1}]$, where β_i is a coefficient updated at each iteration to balance heuristic rewards during different training stages. The scheduling mechanism is detailed in [7] and our source code provided in the Supplementary Material.

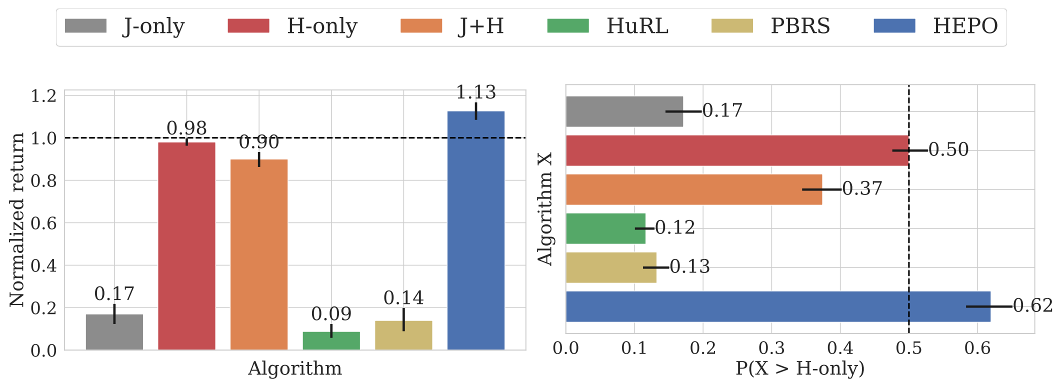
Each method is trained for 5 random seeds and implemented based on the open-sourced implementation [18], where the detailed training hyperparameters can be found in Appendix A.4.

Metrics: Based on the task success criteria in ISAAC and BI-DEX, we consider two types of task reward functions r: (i) Progressing (for locomotion or helicopter robots) and (ii) Goal-reaching (for manipulation). In progressing tasks, robots aim to maximize their traveling distance or velocity from an initial point to a destination. Thus, movement progress is defined as the task reward. In goal-reaching tasks, robots aim to complete assigned goals by reaching specific goal states. Here, task rewards are binary, with a value of 1 indicating successful attainment of the goal and 0 otherwise. Detailed descriptions of our task objectives and total reward definitions are provided in Appendix C.

4.1 Benchmark results

Setup: We aim to determine if HEPO achieves higher task returns and improves upon the policy trained with only heuristic rewards (H-only) in the majority of tasks. In this experiment, we use the heuristic reward functions from the ISAAC and BI-DEX benchmarks. To measure performance improvement over the heuristic policy, we normalize the return of each algorithm X using the formula $(J_X - J_{\text{random}})/(J_{\text{H-only}} - J_{\text{random}})$, where J_X, $J_{\text{H-only}}$, and J_{random} denote the task returns of algorithm X, the heuristic policy, and the random policy, respectively. In Figure 1, we present the interquartile mean (IQM) of the normalized return and the probability of improvement for each method across 29 tasks, following [19]. IQM, also known as the 25% trimmed mean, is a robust estimate against outliers. It discards the bottom and top 25% of runs and calculates the mean score of the remaining 50%. The probability of improvement measures whether an algorithm performs better than another, regardless of the margin of improvement. Both approaches prevent outliers from dominating the performance estimate.

Results: The results in Figure 1 indicate that policies trained with task rewards only (J-only) generally perform worse than those trained with heuristics, both in terms of IQM of normalized return and probability of improvement. PBRS does not improve upon J-only, demonstrating that the optimal policy invariance guarantee rarely holds in practice. HuRL also fails to outperform J-only, likely due to the sensitivity of weight coefficient scheduling to specific tasks, hindering performance with uniform hyperparameters. Similarly, policies trained with both task and heuristic rewards (J+H) perform slightly worse than those trained with heuristics only (H-only), possibly because the weight coefficient balancing both rewards is too task-sensitive to work across all tasks. HEPO, however, outperforms all other methods in both IQM of normalized returns and shows a probability of improvement over the heuristic policy greater than 50%, indicating statistically significant improvements as suggested by Agarwal et al. [19]. Complete learning curves are presented in the Appendix B.1.



(a) IQM over ISAAC + BI-DEX (b) Probability of Improvement over ISAAC + BI-DEX

Figure 1: **(a)** The vertical axis represents the interquartile mean (IQM) [19] of normalized task return across 29 tasks. HEPO outperforms the policy trained solely with a heavily engineered heuristic reward function (H-only) and other methods, demonstrating that HEPO makes better use of heuristic rewards for learning. **(b)** The horizontal axis shows the probability that algorithm X outperforms the policy trained solely with heuristic rewards (H-only). HEPO achieves a 62% probability of improvement over the heuristic policy on average, with the lower bound of the confidence interval above 50%, indicating statistically significant improvements over the heuristic policy.

4.2 Can HEPO be robust to reward functions designed in the wild?

Setup: Our goal is to develop an RL algorithm that can effectively utilize heuristic reward functions, thereby reducing the time costs associated with reward design. We simulate reward design scenarios and evaluate the algorithm's performance when trained with heuristic reward functions created under real-world conditions. Unlike the highly engineered reward functions in ISAAC, participants were asked to design their heuristic reward functions within a short time frame. This approach assesses the algorithm's effectiveness when trained with less refined heuristic reward functions. Participants were asked to iterate on their reward design by writing the reward function, training a policy with a given RL algorithm, reviewing the videos and learning curves, refining the reward, and repeating the process. We selected the *FrankaCabinet* task for this study because its original heuristic reward function is heavily engineered and consisting of many terms. The task of *FrankaCabinet* is training a robot arm to open a cabinet. We recruited 12 participants and divided them into two groups: one group used HEPO to iterate on heuristic rewards, while the other group used PPO. This approach ensures that the designed reward functions are not specialized for one algorithm and ineffective for another. Each participant was instructed to edit the heuristic reward function to help RL algorithms maximize the task return. We used the same task reward metric as in Section 4.1. We then used the final versions of their heuristic reward functions to train HEPO and PPO, and reported the normalized return of the learned policies. Note that we adhered to the normalization scheme outlined in Section 4.1, based on the performance of the policy trained with the original heuristic reward function. This approach allows us to observe any performance drop when training policies with less engineered heuristic reward functions.

Quantitative results: Figure 2 shows that HEPO achieves higher average normalized returns than PPO trained only on heuristic rewards (PPO (H-only)) in 7 out of 12 heuristic reward functions. The improvement is statistically significant for 4 heuristic reward functions ($h3$, $h5$, $h6$, and $h8$). Additionally, Table 1 indicates that across all heuristic functions, HEPO achieves a higher interquartile mean (IQM) of normalized returns and has a statistically significant probability of outperforming PPO (H-only) with low

	Mean (95% CI)
IQM (HEPO)	0.98 (0.73, 1.23)
IQM (PPO)	0.42 (0.31, 0.54)
PI (HEPO > PPO)	0.64 (0.62, 0.65)

Table 1: HEPO outperforms PPO on real-world reward functions, achieving higher normalized returns and statistically significant probability of improvement (PI) greater than 0.5.

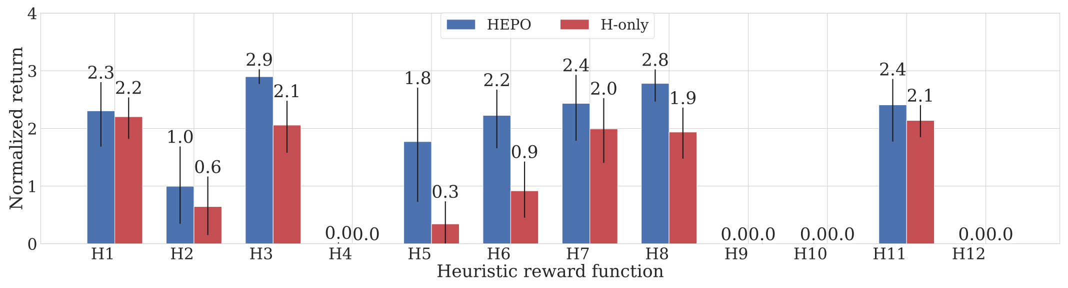


Figure 2: Normalized task return of PPO (H-only) and HEPO that are trained with heuristic reward function $h1$ to $h12$ designed by human subjects in the real world reward design condition. HEPO achieves higher task return than PPO (H-only) in 7 out of 12 tasks with statistically significant improvement in 4 tasks. This shows HEPO is robust to possibly ill-designed heuristic reward functions and can leverage them to improve performance.

er confidence bound greater than 0.5. This suggests that even when trained with poorly designed heuristic reward functions, HEPO performs better than PPO (H-only). Notably, PPO (H-only) that is trained with $h2$, $h5$, and $h6$ achieves normalized returns below 1, while HEPO achieves returns greater than 1. Since returns are normalized using the performance of the PPO policy trained with the well-designed heuristic reward function in ISAAC, a return below 1.0 indicates a performance drop for PPO (H-only) when using potentially ill-designed heuristic rewards. In contrast, HEPO can improve upon policies trained with carefully engineered heuristic reward functions, even when trained with possibly ill-designed heuristic reward functions.

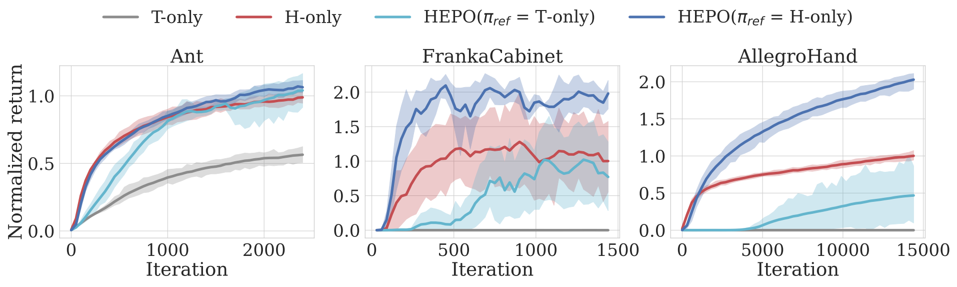
Qualitative observation: We aim to understand why PPO's performance declines when trained with heuristic reward functions $h2$, $h5$, and $h6$. These functions are similar to the original heuristic reward in *FrankaCabinet*, but with different weights for each term. For example, in $h5$, the weight of action penalty is 1, whereas in the original heuristic reward function it is 7.5. This suggests that HEPO might handle poorly scaled heuristic reward terms better than PPO, which is sensitive to these weights. The heuristic reward functions $h12$ and $h9$ had an incorrect sign for the distance component, which caused the policy to be rewarded for moving away from the cabinet instead of toward it, making the learning task more challenging.

4.3 Ablation Studies

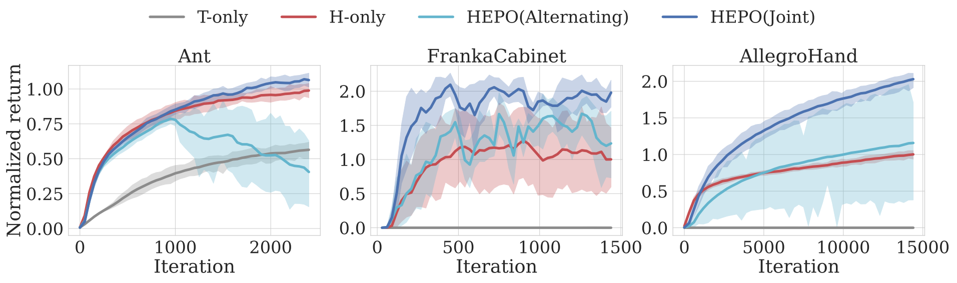
Expanding on the discussion of relation to relevant work EIPO [6] in Section 3.3, our goal is to examine the implementation choices of HEPO and illustrate the efficacy of each modification in this section. HEPO differs from EIPO primarily in two aspects: (1) the selection of a reference policy π_{ref} in the constraint $J(\pi) \geq J(\pi_{\text{ref}})$, and (2) the strategy for utilizing policies to gather trajectories. Both studies are conducted on standard locomotion and manipulation tasks, such as *Ant*, *FrankaCabinet*, and *AllegroHand*. In addition, we provide further studies on the sensitivity to hyperparameters in Appendix B.2.

Selection of reference policy in constraint: HEPO and EIPO both enforce a performance improvement constraint $J(\pi) \geq J(\pi_{\text{ref}})$ during training. HEPO uses a heuristic policy π_H as the reference ($\pi_{\text{ref}} =$ H-only), while EIPO uses a task-only policy ($\pi_{\text{ref}} =$ T-only). However, relying solely on policies trained with task rewards as references may not suffice for complex robotic tasks, as they often perform much worse than those trained with heuristic rewards. We compared the performance of HEPO with different reference policies in Figure 3a. The result shows that setting $\pi_{\text{ref}} =$ T-only (EIPO) improves the performance over the task-only policy *T-only* while notably degrading performance, sometimes even worse than *H-only*, suggesting it's insufficient for surpassing the heuristic policy.

Strategy of collecting trajectories: We use both the enhanced policy π and the heuristic policy π_H simultaneously to sample half of the environment's trajectories (referred to as *Joint*). Conversely, EIPO switches between π and π_H using a specified mechanism, where only one selected policy



(a) Comparison of reference policy π_{ref} choice in HEPO's constraint $J(\pi) \geq J(\pi_{\text{ref}})$



(b) Comparison of trajectory collecting strategies

Figure 3: **(a)** We show that using the policies trained with heuristic rewards is better than using the policies trained with task rewards. **(b)** *HEPO(Joint)* that collects trajectories using both policies leads to better performance than *HEPO(Alternating)* that alternates between two policies to collect trajectories. See Section 4.3 for details

samples trajectories for updating both π and π_H within the same episode (referred to as *Alternating*). This study compares the performance of these two trajectory rollout methods. We modify HEPO to gather trajectories using the *Alternating* strategy and present the results in Figure 3b. The findings indicate that *Alternating* results in a performance drop during mid-training and fails to match the performance of *HEPO(Joint)*. We hypothesize that this occurs because the batch of trajectories collected solely by one policy deviates significantly from those that another policy can generate (i.e., high off-policy error), leading to less effective PPO policy updates. In contrast, *Joint* samples trajectories using both policies, preventing the collected trajectories from deviating too much from each other.

5 Related Works

Reward shaping: Reward shaping has been a significant area, including potential-based reward shaping (PBRS) [5, 20], bilevel optimization approaches [21–23] on reward model learning, and heuristic-guided methods (HuRL) [7] that schedule heuristic rewards. Our method differs as it is a policy optimization method agnostic to the heuristic reward function and can be applied to those shaped or learned rewards.

Constrained policy optimization: Recent work like Extrinsic-Intrinsic Policy Optimization (EIPO) [6] proposes constrained optimization by tuning exploration bonuses to prevent exploiting them at the cost of task rewards. Extensions [24] balance imitating a teacher model and reward maximization. Our work differs in balancing human-designed heuristic rewards and task rewards, improving upon policies trained with engineered heuristic rewards. We also propose implementation enhancements over EIPO [6] (Section 4.3).

6 Discussion & Limitation

HEPO is a tool for RL practitioners: In this paper, we showed that HEPO is robust to the possibly ill-designed heuristic reward function in Section 4.2 and also exhibit high-probability improvement over PPO when training with heavily engineered heuristic rewards in robotic tasks in Section 4.1. Moving forward, when users need to integrate heuristic reward functions into RL algorithms, HEPO can potentially be a useful tool to reduce users' time on designing rewards since it can improve performance even with under-engineered heuristic rewards.

Limitations: While HEPO shows high-probability performance improvement over heuristic policies trained with well-designed heuristic reward, one limitation is that HEPO does not have a guarantee to converge to the optimal policy theoretically. One future work can be incorporating the insight in recent theoretical advances on reward engineering [25] to make a convergence guarantee.

References

[1] Tao Chen, Jie Xu, and Pulkit Agrawal. A system for general in-hand object re-orientation. *Conference on Robot Learning*, 2021.

[2] Gabriel B Margolis, Ge Yang, Kartik Paigwar, Tao Chen, and Pulkit Agrawal. Rapid locomotion via reinforcement learning. *Robotics: Science and Systems*, 2022.

[3] 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.

[4] Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. RMA: rapid motor adaptation for legged robots. In *Robotics: Science and Systems XVII, Virtual Event, July 12-16, 2021*, 2021.

[5] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pages 278–287, 1999.

[6] Eric Chen, Zhang-Wei Hong*, Joni Pajarinen, and Pulkit (* equal contribution) Agrawal. Redeeming intrinsic rewards via constrained optimization. *Advances in Neural Information Processing Systems*, 35:4996–5008, 2022.

[7] Ching-An Cheng, Andrey Kolobov, and Adith Swaminathan. Heuristic-guided reinforcement learning. *Advances in Neural Information Processing Systems*, 34:13550–13563, 2021.

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

[9] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1889–1897, 2015.

[10] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 2015.

[12] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. Isaac gym: High performance gpu-based physics simulation for robot learning, 2021.

[13] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2018.

[14] Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *In Proc. 19th International Conference on Machine Learning*. Citeseer, 2002.

[15] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *International Conference of Representation Learning*, 2015.

[16] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1lJJnR5Ym.

[17] Yuanpei Chen, Yaodong Yang, Tianhao Wu, Shengjie Wang, Xidong Feng, Jiechuan Jiang, Zongqing Lu, Stephen Marcus McAleer, Hao Dong, and Song-Chun Zhu. Towards human-level bimanual dexterous manipulation with reinforcement learning. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022.

[18] Denys Makoviichuk and Viktor Makoviychuk. rl-games: A high-performance framework for reinforcement learning. https://github.com/Denys88/rl_games, May 2021.

[19] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in Neural Information Processing Systems*, 34, 2021.

[20] Sam Devlin and Daniel Kudenko. Dynamic potential-based reward shaping. In *Adaptive Agents and Multi-Agent Systems*, 2012.

[21] Yujing Hu, Weixun Wang, Hangtian Jia, Yixiang Wang, Yingfeng Chen, Jianye Hao, Feng Wu, and Changjie Fan. Learning to utilize shaping rewards: A new approach of reward shaping. *Advances in Neural Information Processing Systems*, 33:15931–15941, 2020.

[22] Dhawal Gupta, Yash Chandak, Scott Jordan, Philip S Thomas, and Bruno C da Silva. Behavior alignment via reward function optimization. *Advances in Neural Information Processing Systems*, 36, 2024.

[23] Zeyu Zheng, Junhyuk Oh, and Satinder Singh. On learning intrinsic rewards for policy gradient methods. *Advances in Neural Information Processing Systems*, 31, 2018.

[24] Idan Shenfeld, Zhang-Wei Hong, Aviv Tamar, and Pulkit Agrawal. TGRL: An algorithm for teacher guided reinforcement learning. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

[25] Abhishek Gupta, Aldo Pacchiano, Yuexiang Zhai, Sham Kakade, and Sergey Levine. Unpacking reward shaping: Understanding the benefits of reward engineering on sample complexity. *Advances in Neural Information Processing Systems*, 35:15281–15295, 2022.

[26] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2014.

A Implementation Details

A.1 Full Derivation

We will detailedly describe the update of the enhanced policy (π in Equation 7) and the heuristic policy (π_H in Equation 8) at each iteration.

A.1.1 Notations

- $V_r^{\pi}(s_t) \coloneqq \mathbb{E}_{(s_t,a_t)\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t,a_t)|s_0=s_t\right]$
- $V_h^{\pi}(s_t) \coloneqq \mathbb{E}_{(s_t,a_t)\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t h(s_t,a_t)|s_0=s_t\right]$
- $A_r^{\pi}(s_t,a_t) \coloneqq r(s_t,a_t)+V_r^{\pi}(s_{t+1})-V_r^{\pi}(s_t)$
- $A_h^{\pi}(s_t,a_t) \coloneqq h(s_t,a_t)+V_h^{\pi}(s_{t+1})-V_h^{\pi}(s_t)$
- B_{HEPO}: the buffer to store samples collected by π^i
- B_H: the buffer to store samples collected π_H^i.

A.1.2 Enhanced Policy π Update

Given a α value, π^{i+1} is derived using the arguments of the maxima in Equation 4, which can be re-written as follows:

$$\begin{aligned}
\pi^{i+1} &= \arg\max_{\pi}\left\{J(\pi)+H(\pi)-\alpha\Big(J(\pi)-J(\pi_H^i)\Big)\right\}\\
&= \arg\max_{\pi}\left\{(1+\alpha)J(\pi)+H(\pi)\right\}\\
&= \arg\max_{\pi}\bigg\{\Big((1+\alpha)J(\pi)+H(\pi)\Big)-\frac{1}{2}\Big((1+\alpha)J(\pi^i)+H(\pi^i)\Big)\\
&\qquad -\frac{1}{2}\Big((1+\alpha)J(\pi_H^i)+H(\pi_H^i)\Big)\bigg\}\\
&= \arg\max_{\pi}\bigg\{\frac{1}{2}\mathbb{E}_{\pi}\Big[(1+\alpha)A_r^{\pi^i}(s_t,a_t)+A_h^{\pi^i}(s_t,a_t)\Big]\\
&\qquad +\frac{1}{2}\mathbb{E}_{\pi}\Big[(1+\alpha)A_r^{\pi_H^i}(s_t,a_t)+A_h^{\pi_H^i}(s_t,a_t)\Big]\\
&= \arg\max_{\pi}\left\{\frac{1}{2}\mathbb{E}_{\pi}\Big[U_\alpha^{\pi^i}(s_t,a_t)\Big]+\frac{1}{2}\mathbb{E}_{\pi}\Big[U_\alpha^{\pi_H^i}(s_t,a_t)\Big]\right\}\\
&= \arg\max_{\pi}\left\{\mathbb{E}_{\pi}\Big[U_\alpha^{\pi^i}(s_t,a_t)\Big]+\mathbb{E}_{\pi}\Big[U_\alpha^{\pi_H^i}(s_t,a_t)\Big]\right\}
\end{aligned} \tag{12}$$

where $U_\alpha^{\pi^i}$ and $U_\alpha^{\pi_H^i}$ are defined as follows:

$$\begin{aligned}
U_\alpha^{\pi^i}(s_t,a_t) &\coloneqq (1+\alpha)A_r^{\pi^i}(s_t,a_t)+A_h^{\pi^i}(s_t,a_t)\\
U_\alpha^{\pi_H^i}(s_t,a_t) &\coloneqq (1+\alpha)A_r^{\pi_H^i}(s_t,a_t)+A_h^{\pi_H^i}(s_t,a_t)
\end{aligned} \tag{13}$$

To efficiently achieve the update process in Equation 12, we aim to utilize previously collected trajectories for optimization, outlined in Equation 7. Here, we refer to [8], using those previously collected trajectories to form a lower bound surrogate objectives, $\hat{J}_\alpha^{\pi^i}(\pi)$ and $\hat{J}_\alpha^{\pi_H^i}(\pi)$, as alternatives of $\mathbb{E}_\pi[U_\alpha^{\pi^i}(s_t,a_t)]$ and $\mathbb{E}_\pi[U_\alpha^{\pi_H^i}(s_t,a_t)]$ to derive π^{i+1}:

$$\begin{aligned}
\hat{J}^{\pi^i}_{\text{HEPO}}(\pi) &:= \frac{1}{|B_{\text{HEPO}}|}\sum_{(s_t,a_t)\in B_{\text{HEPO}}}\Big[\sum_{t=0}^{\infty}\gamma^t \min\Big\{\frac{\pi(a_t|s_t)}{\pi^i(a_t|s_t)}U_\alpha^{\pi^i}(s_t,a_t),\\
&\qquad \text{clip}\left(\frac{\pi(a_t|s_t)}{\pi^i(a_t|s_t)},1-\epsilon,1+\epsilon\right)U_\alpha^{\pi^i}(s_t,a_t)\Big\}\Big]\\
\hat{J}^{\pi_H^i}_{\text{HEPO}}(\pi) &:= \frac{1}{|B_H|}\sum_{(s_t,a_t)\in B_H}\Big[\sum_{t=0}^{\infty}\gamma^t \min\Big\{\frac{\pi(a_t|s_t)}{\pi_H^i(a_t|s_t)}U_\alpha^{\pi_H^i}(s_t,a_t),\\
&\qquad \text{clip}\left(\frac{\pi(a_t|s_t)}{\pi_H^i(a_t|s_t)},1-\epsilon,1+\epsilon\right)U_\alpha^{\pi_H^i}(s_t,a_t)\Big\}\Big],
\end{aligned} \tag{14}$$

where $\mathbb{E}_\pi[U_\alpha^{\pi^i}(s_t,a_t)] \geq \hat{J}^{\pi^i}_{\text{HEPO}}(\pi)$ and $\mathbb{E}_\pi[U_\alpha^{\pi_H^i}(s_t,a_t)] \geq \hat{J}^{\pi_H^i}_{\text{HEPO}}(\pi)$ always hold; $\epsilon \in [0,1]$ denotes a threshold. Intuitively, this clipped objective (Eq. 14) penalizes the policy π that behaves differently from π^i or π_H^i because overly large or small the action probability ratios between two policies are clipped.

A.1.3 Heuristic Policy π_H Update

π_H^{i+1} is derived using the arguments of the maxima of $H(\pi)$, which can be re-written as follows:

$$\begin{aligned}
\pi_H^{i+1} &= \arg\max_\pi \Big\{H(\pi)\Big\}\\
&= \arg\max_\pi \Big\{H(\pi) - \frac{1}{2}H(\pi^i) - \frac{1}{2}H(\pi_H^i)\Big\}\\
&= \arg\max_\pi \Big\{\frac{1}{2}\mathbb{E}_\pi\Big[A_h^{\pi^i}(s_t,a_t)\Big] + \frac{1}{2}\mathbb{E}_\pi\Big[A_h^{\pi_H^i}(s_t,a_t)\Big]\Big\}\\
&= \arg\max_\pi \Big\{\mathbb{E}_\pi\Big[A_h^{\pi^i}(s_t,a_t)\Big] + \mathbb{E}_\pi\Big[A_h^{\pi_H^i}(s_t,a_t)\Big]\Big\}
\end{aligned} \tag{15}$$

Similarly, we again rely on the approximation from [8] to derive a lower bound surrogate objective for both $\mathbb{E}_\pi[A_h^{\pi^i}(s_t,a_t)]$ and $\mathbb{E}_\pi[A_h^{\pi_H^i}(s_t,a_t)]$ as follows:

$$\begin{aligned}
\hat{H}^{\pi^i}(\pi) &:= \frac{1}{|B_{\text{HEPO}}|}\sum_{(s_t,a_t)\in B_{\text{HEPO}}}\Big[\sum_{t=0}^{\infty}\gamma^t \min\Big\{\frac{\pi(a_t|s_t)}{\pi^i(a_t|s_t)}A_h^{\pi^i}(s_t,a_t),\\
&\qquad \text{clip}\left(\frac{\pi(a_t|s_t)}{\pi^i(a_t|s_t)},1-\epsilon,1+\epsilon\right)A_h^{\pi^i}(s_t,a_t)\Big\}\Big],
\end{aligned} \tag{16}$$

$$\begin{aligned}
\hat{H}^{\pi_H^i}(\pi) &:= \frac{1}{|B_H|}\sum_{(s_t,a_t)\in B_H}\Big[\sum_{t=0}^{\infty}\gamma^t \min\Big\{\frac{\pi(a_t|s_t)}{\pi_H^i(a_t|s_t)}A_h^{\pi_H^i}(s_t,a_t),\\
&\qquad \text{clip}\left(\frac{\pi(a_t|s_t)}{\pi_H^i(a_t|s_t)},1-\epsilon,1+\epsilon\right)A_h^{\pi_H^i}(s_t,a_t)\Big\}\Big]
\end{aligned} \tag{17}$$

where $\mathbb{E}_\pi[A_h^{\pi^i}(s_t,a_t)] \geq \hat{H}^{\pi^i}(\pi)$ and $\mathbb{E}_\pi[A_h^{\pi_H^i}(s_t,a_t)] \geq \hat{H}^{\pi_H^i}(\pi)$ always hold. Different from vanilla heuristic training, instead of solely collecting trajectories from π_H^i, we collect trajectories from both π and π_H to enrich sample efficiency.

A.2 Implementation Tricks

A.2.1 Sample Sharing for Value Function Update

In practice, obtaining real value functions for training is not feasible. We estimate the value function using collected trajectories, but this approach tends to fail because the value function becomes biased toward the policy responsible for trajectory collection.

To prevent error information from the estimated value function interfering with the training procedure, we share the trajectory samples within the B_{HEPO} and B_H buffers to update our value functions:

$$V_r^{\pi^{i+1}} \leftarrow \arg\min_V \left\{ \sum_{\left(s_t, r(s_t,a_t), s_{t+1}\right) \in B_{\text{HEPO}} \cup B_H} \frac{|r(s_t,a_t) + \gamma V_r^{\pi^i}(s_{t+1}) - V(s_t)|^2}{|B_{\text{HEPO}}| + |B_H|} \right\} \tag{18}$$

$$V_h^{\pi^{i+1}} \leftarrow \arg\min_V \left\{ \sum_{\left(s_t, h(s_t,a_t), s_{t+1}\right) \in B_{\text{HEPO}} \cup B_H} \frac{|h(s_t,a_t) + \gamma V_h^{\pi^i}(s_{t+1}) - V(s_t)|^2}{|B_{\text{HEPO}}| + |B_H|} \right\} \tag{19}$$

$$V_r^{\pi_H^{i+1}} \leftarrow \arg\min_V \left\{ \sum_{\left(s_t, r(s_t,a_t), s_{t+1}\right) \in B_{\text{HEPO}} \cup B_H} \frac{|r(s_t,a_t) + \gamma V_r^{\pi_H^i}(s_{t+1}) - V(s_t)|^2}{|B_{\text{HEPO}}| + |B_H|} \right\} \tag{20}$$

$$V_h^{\pi_H^{i+1}} \leftarrow \arg\min_V \left\{ \sum_{\left(s_t, h(s_t,a_t), s_{t+1}\right) \in B_{\text{HEPO}} \cup B_H} \frac{|h(s_t,a_t) + \gamma V_h^{\pi_H^i}(s_{t+1}) - V(s_t)|^2}{|B_{\text{HEPO}}| + |B_H|} \right\} \tag{21}$$

A.2.2 Smoothing Lagrangian Multiplier α Update

The Lagrangian multiplier α determines the desired constraint information during training. However, in practice the gradient α tends to become explosive. To stabilize the α update procedure, we accumulate previous gradients and adopt the Adam optimizer [26] as follows:

$$\begin{aligned} g(\alpha) &\leftarrow \text{med}\left\{ \frac{1}{|B_{\text{HEPO}}|} \sum_{(s_t,a_t) \in B_{\text{HEPO}}} \left[A_r^{\pi_H^i}(s_t,a_t) \right] - \frac{1}{|B_H|} \sum_{(s_t,a_t) \in B_H} \left[A_r^{\pi^i}(s_t,a_t) \right] \right\}_{i-K}^{i} \\ \alpha &\leftarrow \text{AdamOpt}\left[g(\alpha) \right] \end{aligned} \tag{22}$$

where K is the number of previous K advantage expectation records that we take into account. To smooth the current α gradient for each update, we calculate the median of the previous K records. In our experiments, we assigned K a value of 8.

A.3 Overall Workflow

Algorithm 2 Detailed Heuristic-Enhanced Policy Optimization (HEPO)

```
1:  Initialize policies $(\pi^1, \pi_H^1)$ and values $(V_r^{\pi^1}, V_h^{\pi^1}, V_r^{\pi_H^1}, V_h^{\pi_H^1})$
2:  for $i = 1 \cdots$ do                                        ▷ $i$ denotes iteration index
3:      # ROLLOUT STAGE
4:      Collect trajectory buffers $(B_{\text{HEPO}}, B_H)$ using $(\pi^i, \pi_H^i)$
5:      Compute $\left(A_r^{\pi^i}(s_t,a_t), A_h^{\pi^i}(s_t,a_t)\right)$ via GAE with $\left(V_r^{\pi^i}, V_h^{\pi^i}\right)$ $\forall (s_t,a_t) \in B_{\text{HEPO}}$
6:      Compute $\left(A_r^{\pi_H^i}(s_t,a_t), A_h^{\pi_H^i}(s_t,a_t)\right)$ via GAE with $\left(V_r^{\pi_H^i}, V_h^{\pi_H^i}\right)$ $\forall (s_t,a_t) \in B_H$
7:      Compute $\left(\hat{J}_{\text{HEPO}}^{\pi^i}, \hat{J}_{\text{HEPO}}^{\pi_H^i}\right)$ based on Equation 14
8:      Compute $\left(\hat{H}^{\pi^i}, \hat{H}^{\pi_H^i}\right)$ based on Equation 16
9:
10:     # UPDATE STAGE
11:     $\pi^{i+1} \leftarrow \arg\max_\pi \left\{ \hat{J}_{\text{HEPO}}^{\pi^i}(\pi) + \hat{J}_{\text{HEPO}}^{\pi_H^i}(\pi) \right\}$
12:     $\pi_H^{i+1} \leftarrow \arg\max_\pi \left\{ \hat{H}^{\pi^i}(\pi) + \hat{H}^{\pi_H^i}(\pi) \right\}$
13:     Update $(V_r^{\pi^i}, V_h^{\pi^i}, V_r^{\pi_H^i}, V_h^{\pi_H^i})$ based on Equation 18
14:     Update $\alpha$ based on Equation 22
15: end for
```

A.4 Training details

Following the PPO framework [8], our experiments are based on a continuous action actor-critic algorithm implemented in `rl_games` [18], using Generalized Advantage Estimation (GAE) [15] to compute advantages for policy optimization. For PPO, we employed the same policy network and value network architecture, and the same hyperparameters used in `IsaacGymEnvs` [12]. We also include our source code in the supplementary material. In HEPO, we use two policies for optimization, with each policy maintaining the same model configurations as those used in PPO. Below we introduce HEPO-specific hyperparameters used in our experiments in Section 4.1. The hyperparameters for updating the Lagrangian multiplier α in HEPO are listed as follows:

Table 2: HEPO Hyperparameters

Name	**Value**
Initial α	0.0
Step size η of α (learning rate)	0.01
Clipping range of $\delta\alpha$ $(-\epsilon_\alpha, \epsilon_\alpha)$	1.0
Range of the α value	$[0, \infty)$

For baselines, we search for hyperparamters λ for $J + H$ in *Ant*, *FrankaCabinet*, and *AllegroHand*, as shown in Section B.2. We set $\lambda = 1$ for all the experiments because it shows better performance on the three chosen environments. For HuRL [7], we follow the scheduling setting provided in their paper.

478 # B Supplementary Experimental Results

479 ## B.1 All Learning Curves on the task objective J

We present all the learning curves in Figure 4.

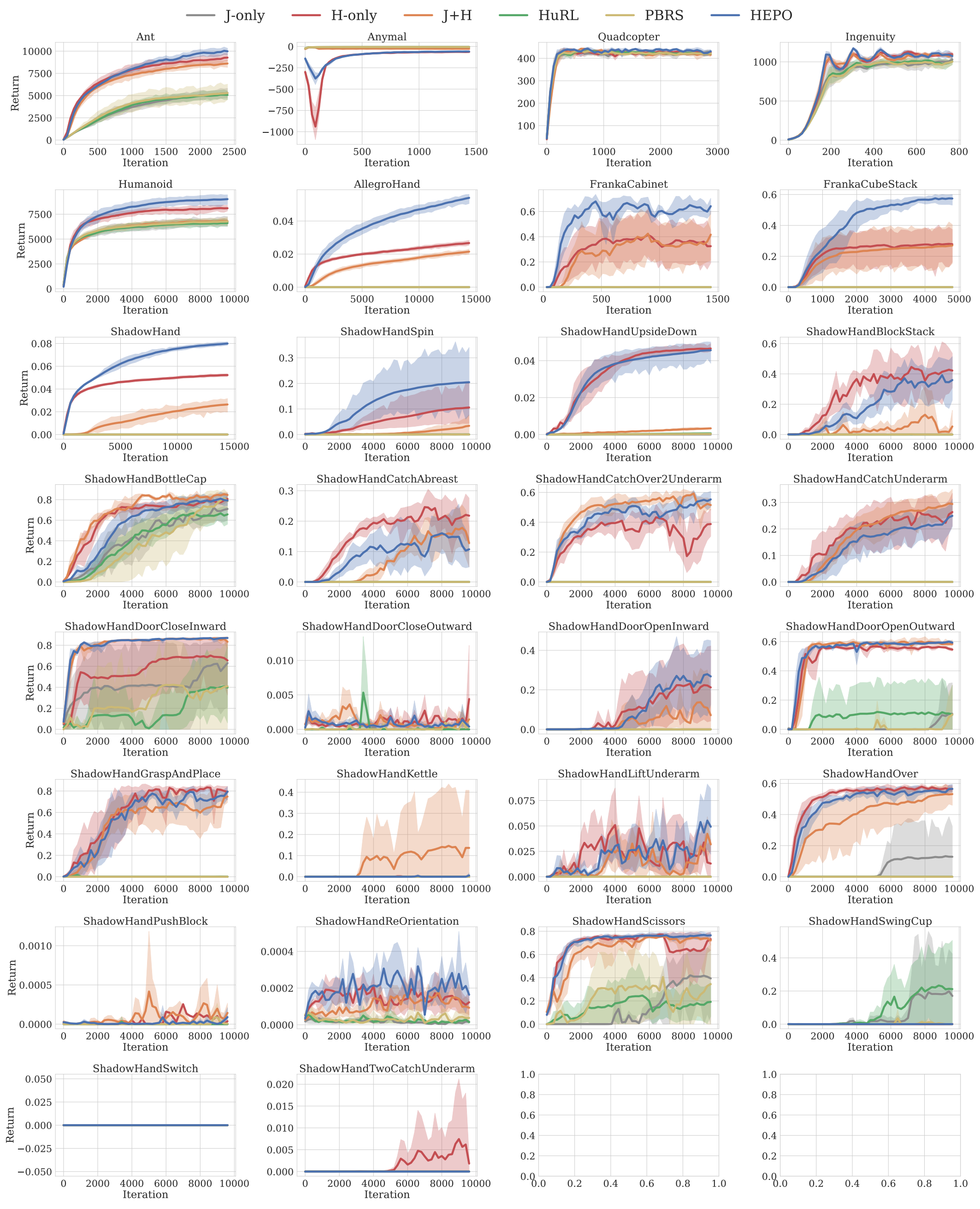


Figure 4: All learning curves in Section 4.1

480

481 ## B.2 Sensitivity to hyperparameters

482 In this section, we aim to verify HEPO's sensitivity to two main types of hyperparameters: (1) the
483 weight of the heuristic reward in optimization (denoted as λ) and (2) the learning rate for updating α.

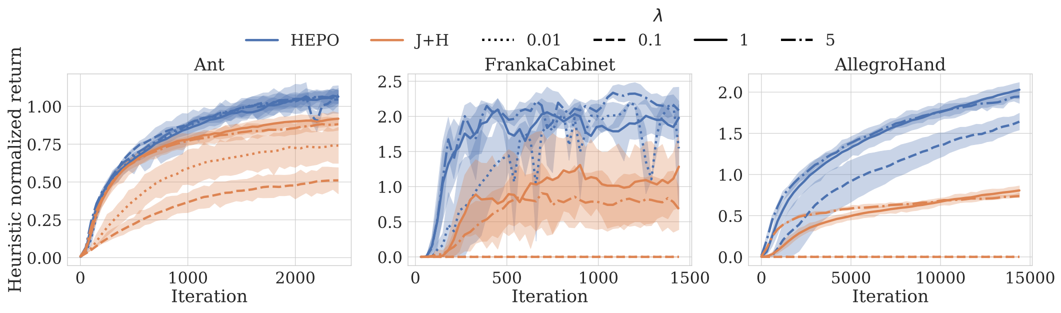


Figure 5: Sensitivity to λ

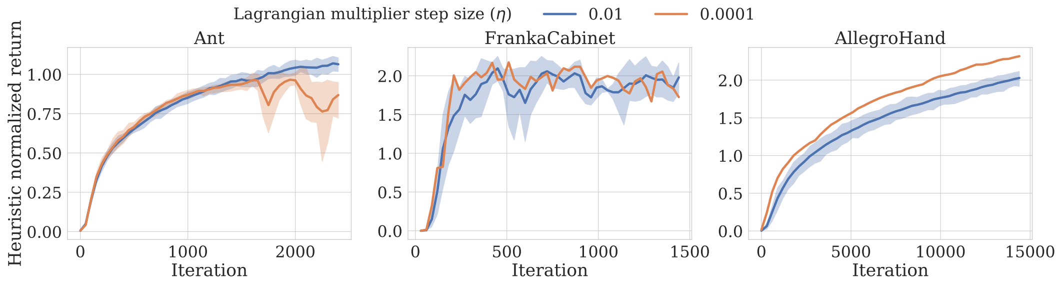


Figure 6: Sensitivity to alpha learning rate

Similar to Section 4.3, we conducted our experiments on the *Ant*, *FrankaCabinet*, and *AllegroHand* tasks.

B.2.1 Sensitivity to the λ Value

Both **HEPO** and **J+H** can set a scaling coefficient to weight the heuristic reward in optimization, such that the objective becomes $J(\pi) + \lambda H(\pi)$. This scaling coefficient can be used to balance both objectives. In this study, we compare **HEPO** and **J+H** on their performance sensitivity to the choice of λ, exhaustively training both **HEPO** and **J+H** with varying λ values. Note that though the formulation of HEPO does not depend on λ, one can still set a λ coefficient to scale the heuristic reward in HEPO. In our experiments, we did not optimize λ for HEPO but for the baselines trained with both rewards (J+H). In Figure 5, we found that **J+H** is sensitive to λ in all selected tasks, while **HEPO** performs well across a wide range of λ values. This indicates that **HEPO** is robust to the choice of λ.

B.2.2 Sensitivity to the Learning Rate for Updating α

HEPO's robustness relies on the α update, as it reflects the necessary constraint information at each iteration. Similar to Appendix B.2.1, setting different initial values of α is equivalent to using different λ values for our estimation, since both can be rewritten as the ratio between $H(\pi)$ and $J(\pi)$. Both of these parameters indicate the necessary constraint information for conducting multi-objective optimization. In this study, we aim to verify whether **HEPO** can yield comparable improvement gaps under different initial values of α, thus providing a more robust optimization procedure.

As shown in Figure 6, we observe that **HEPO** is also robust to the choice of α's initial values, similar to the results in Figure 5.

C Environment Details

As depicted in Section 4, we conducted our experiments based on the Isaac Gym (**ISAAC**) simulator [12] and the Bi-DexHands (**BI-DEX**) benchmark [17]. The selected task classes in ISAAC can be partitioned into 4 groups - Locomotion Tracking (***Anymal***), Locomotion Progressing (***Ant*** and ***Humanoid***), Helicopter Progressing (***Ingenuity*** and ***Quadcopter***), and Manipulation Tasks

(***FrankaCabinet***, ***FrankaCubeStack***, ***ShadowHand***, and ***AllegroHand***). In addition, BI-DEX provides dual dexterous hand manipulation tasks through ISAAC, reaching human-level sophistication of hand dexterity and bimanual coordination. Their tasks include ***ShadowHandOver***, ***ShadowHandCatchUnderarm***, ***ShadowHandCatchOver2Underarm***, ***ShadowHandCatchAbreast***, ***ShadowHandTwoCatchUnderarm***, ***ShadowHandLiftUnderarm***, ***ShadowHandDoorOpenInward***, ***ShadowHandDoorOpenOutward***, ***ShadowHandDoorCloseInward***, ***ShadowHandDoorCloseOutward***, ***ShadowHandSpin***, ***ShadowHandUpsideDown***, ***ShadowHandBlockStack***, ***ShadowHandBottleCap***, ***ShadowHandGraspAndPlace***, ***ShadowHandKettle***, ***ShadowHandPen***, ***ShadowHandPushBlock***, ***ShadowHandReOrientation***, ***ShadowHandScissors***, ***ShadowHandSwingCup***.

For Locomotion Tracking, our emphasis lies in assessing the precision of velocities in linear and angular motions, ensuring that the robot responds closely to the assigned values. To this end, we define tracking errors as our task rewards. For Locomotion Progressing and Helicopter Progressing, our emphasis lies in evaluating the progress made by the robots in reaching the assigned destination from a given start point. To this end, we define movement progress as our task rewards. For Manipulation tasks and all tasks within the BI-DEX benchmark, our emphasis lies in whether and how quickly the robotic hands can successfully complete the assigned missions, reaching the desired goal states. To this end, we define task rewards using a binary label, assigning a value of 1 to indicate successful attainment of the goal state and 0 otherwise.

The following are our reward function definitions, which include heuristic and task reward terms in Python style:

Isaac Gym - Locomotion Tracking Task: ***Anymal***

```
def compute_anymal_reward(
    root_states,
    commands,
    torques,
    contact_forces,
    knee_indices,
    episode_lengths,
    rew_scales,
    base_index,
    max_episode_length
):
    # (reward, reset, feet_in air, feet_air_time, episode sums)
    # type: (Tensor, Tensor, Tensor, Tensor, Tensor, Tensor, Dict[str, float], int, int) -> Tuple[Tensor,
    ↪  Tensor, Tensor]

    # prepare quantities (TODO: return from obs ?)
    base_quat = root_states[:, 3:7]
    base_lin_vel = quat_rotate_inverse(base_quat, root_states[:, 7:10])
    base_ang_vel = quat_rotate_inverse(base_quat, root_states[:, 10:13])

    # velocity tracking reward
    lin_vel_error = torch.sum(torch.square(commands[:, :2] - base_lin_vel[:, :2]), dim=1)
    ang_vel_error = torch.square(commands[:, 2] - base_ang_vel[:, 2])
    rew_lin_vel_xy = torch.exp(-lin_vel_error/0.25) * rew_scales["lin_vel_xy"]
    rew_ang_vel_z = torch.exp(-ang_vel_error/0.25) * rew_scales["ang_vel_z"]

    # torque penalty
    rew_torque = torch.sum(torch.square(torques), dim=1) * rew_scales["torque"]

    total_reward = rew_lin_vel_xy + rew_ang_vel_z + rew_torque
    total_reward = torch.clip(total_reward, 0., None)
    tracking_reward = -(lin_vel_error + ang_vel_error)

    # reset agents
    reset = torch.norm(contact_forces[:, base_index, :], dim=1) > 1.
    reset = reset | torch.any(torch.norm(contact_forces[:, knee_indices, :], dim=2) > 1., dim=1)
    time_out = episode_lengths >= max_episode_length - 1  # no terminal reward for time-outs
    reset = reset | time_out
    heuristic_reward, task_reward = total_reward.detach(), tracking_reward
    return heuristic_reward, task_reward, reset
```

Isaac Gym - Locomotion Progressing Task: ***Ant***

```
def compute_ant_reward(
    obs_buf,
    reset_buf,
    progress_buf,
    actions,
    up_weight,
    heading_weight,
    potentials,
    prev_potentials,
    actions_cost_scale,
    energy_cost_scale,
    joints_at_limit_cost_scale,
    termination_height,
    death_cost,
    max_episode_length
):
    # type: (Tensor, Tensor, Tensor, Tensor, float, float, Tensor, Tensor, float, float, float, float,
    ↪  float, float) -> Tuple[Tensor, Tensor, Tensor]

    # reward from direction headed
    heading_weight_tensor = torch.ones_like(obs_buf[:, 11]) * heading_weight
    heading_reward = torch.where(obs_buf[:, 11] > 0.8, heading_weight_tensor, heading_weight * obs_buf[:,
    ↪  11] / 0.8)

    # aligning up axis of ant and environment
    up_reward = torch.zeros_like(heading_reward)
    up_reward = torch.where(obs_buf[:, 10] > 0.93, up_reward + up_weight, up_reward)

    # energy penalty for movement
    actions_cost = torch.sum(actions ** 2, dim=-1)
    electricity_cost = torch.sum(torch.abs(actions * obs_buf[:, 20:28]), dim=-1)
    dof_at_limit_cost = torch.sum(obs_buf[:, 12:20] > 0.99, dim=-1)

    # reward for duration of staying alive
    alive_reward = torch.ones_like(potentials) * 0.5
    progress_reward = potentials - prev_potentials

    total_reward = progress_reward + alive_reward + up_reward + heading_reward - \
        actions_cost_scale * actions_cost - energy_cost_scale * electricity_cost - dof_at_limit_cost *
        ↪  joints_at_limit_cost_scale

    # adjust reward for fallen agents
    total_reward = torch.where(obs_buf[:, 0] < termination_height, torch.ones_like(total_reward) *
    ↪  death_cost, total_reward)

    # reset agents
    reset = torch.where(obs_buf[:, 0] < termination_height, torch.ones_like(reset_buf), reset_buf)
    reset = torch.where(progress_buf >= max_episode_length - 1, torch.ones_like(reset_buf), reset)
    heuristic_reward, task_reward = total_reward, progress_reward
    return heuristic_reward, task_reward, reset
```

Isaac Gym - Locomotion Progressing Task: ***Humanoid***

```python
def compute_humanoid_reward(
    obs_buf,
    reset_buf,
    progress_buf,
    actions,
    up_weight,
    heading_weight,
    potentials,
    prev_potentials,
    actions_cost_scale,
    energy_cost_scale,
    joints_at_limit_cost_scale,
    max_motor_effort,
    motor_efforts,
    termination_height,
    death_cost,
    max_episode_length
):
    # type: (Tensor, Tensor, Tensor, Tensor, float, float, Tensor, Tensor, float, float, float, float,
    ↪  Tensor, float, float, float) -> Tuple[Tensor, Tensor, Tensor]

    # reward from the direction headed
    heading_weight_tensor = torch.ones_like(obs_buf[:, 11]) * heading_weight
    heading_reward = torch.where(obs_buf[:, 11] > 0.8, heading_weight_tensor, heading_weight * obs_buf[:,
    ↪  11] / 0.8)

    # reward for being upright
    up_reward = torch.zeros_like(heading_reward)
    up_reward = torch.where(obs_buf[:, 10] > 0.93, up_reward + up_weight, up_reward)

    actions_cost = torch.sum(actions ** 2, dim=-1)

    # energy cost reward
    motor_effort_ratio = motor_efforts / max_motor_effort
    scaled_cost = joints_at_limit_cost_scale * (torch.abs(obs_buf[:, 12:33]) - 0.98) / 0.02
    dof_at_limit_cost = torch.sum((torch.abs(obs_buf[:, 12:33]) > 0.98) * scaled_cost *
    ↪  motor_effort_ratio.unsqueeze(0), dim=-1)

    electricity_cost = torch.sum(torch.abs(actions * obs_buf[:, 33:54]) *
    ↪  motor_effort_ratio.unsqueeze(0), dim=-1)

    # reward for duration of being alive
    alive_reward = torch.ones_like(potentials) * 2.0
    progress_reward = potentials - prev_potentials

    total_reward = progress_reward + alive_reward + up_reward + heading_reward - \
        actions_cost_scale * actions_cost - energy_cost_scale * electricity_cost - dof_at_limit_cost

    # adjust reward for fallen agents
    total_reward = torch.where(obs_buf[:, 0] < termination_height, torch.ones_like(total_reward) *
    ↪  death_cost, total_reward)

    # reset agents
    reset = torch.where(obs_buf[:, 0] < termination_height, torch.ones_like(reset_buf), reset_buf)
    reset = torch.where(progress_buf >= max_episode_length - 1, torch.ones_like(reset_buf), reset)
    heuristic_reward, task_reward = total_reward, progress_reward
    return heuristic_reward, task_reward, reset
```

Isaac Gym - Helicopter Progressing Task: ***Ingenuity***

```
def compute_ingenuity_reward(root_positions, target_root_positions, root_quats, root_linvels,
↪  root_angvels, reset_buf, progress_buf, max_episode_length):
    # type: (Tensor, Tensor, Tensor, Tensor, Tensor, Tensor, Tensor, float) -> Tuple[Tensor, Tensor,
    ↪  Tensor]

    # distance to target
    target_dist = torch.sqrt(torch.square(target_root_positions - root_positions).sum(-1))
    pos_reward = 1.0 / (1.0 + target_dist * target_dist)

    # uprightness
    ups = quat_axis(root_quats, 2)
    tiltage = torch.abs(1 - ups[..., 2])
    up_reward = 5.0 / (1.0 + tiltage * tiltage)

    # spinning
    spinnage = torch.abs(root_angvels[..., 2])
    spinnage_reward = 1.0 / (1.0 + spinnage * spinnage)

    # combined reward
    # uprigness and spinning only matter when close to the target
    reward = pos_reward + pos_reward * (up_reward + spinnage_reward)

    # resets due to misbehavior
    ones = torch.ones_like(reset_buf)
    die = torch.zeros_like(reset_buf)
    die = torch.where(target_dist > 8.0, ones, die)
    die = torch.where(root_positions[..., 2] < 0.5, ones, die)

    # resets due to episode length
    reset = torch.where(progress_buf >= max_episode_length - 1, ones, die)
    heuristic_reward, task_reward = reward, pos_reward
    return heuristic_reward, task_reward, reset
```

Isaac Gym - Helicopter Progressing Task: ***Quadcopter***

```
def compute_quadcopter_reward(root_positions, root_quats, root_linvels, root_angvels, reset_buf,
↪  progress_buf, max_episode_length):
    # type: (Tensor, Tensor, Tensor, Tensor, Tensor, Tensor, float) -> Tuple[Tensor, Tensor, Tensor]

    # distance to target
    target_dist = torch.sqrt(root_positions[..., 0] * root_positions[..., 0] +
                             root_positions[..., 1] * root_positions[..., 1] +
                             (1 - root_positions[..., 2]) * (1 - root_positions[..., 2]))
    pos_reward = 1.0 / (1.0 + target_dist * target_dist)

    # uprightness
    ups = quat_axis(root_quats, 2)
    tiltage = torch.abs(1 - ups[..., 2])
    up_reward = 1.0 / (1.0 + tiltage * tiltage)

    # spinning
    spinnage = torch.abs(root_angvels[..., 2])
    spinnage_reward = 1.0 / (1.0 + spinnage * spinnage)

    # combined reward
    # uprigness and spinning only matter when close to the target
    reward = pos_reward + pos_reward * (up_reward + spinnage_reward)

    # resets due to misbehavior
    ones = torch.ones_like(reset_buf)
    die = torch.zeros_like(reset_buf)
    die = torch.where(target_dist > 3.0, ones, die)
    die = torch.where(root_positions[..., 2] < 0.3, ones, die)

    # resets due to episode length
    reset = torch.where(progress_buf >= max_episode_length - 1, ones, die)
    heuristic_reward, task_reward = reward, pos_reward
    return heuristic_reward, task_reward, reset
```

Isaac Gym - Manipulation Task: ***FrankaCabinet***

```
def compute_franka_reward(
    reset_buf, progress_buf, reset_goal_buf, successes, consecutive_successes, actions, cabinet_dof_pos,
    franka_grasp_pos, drawer_grasp_pos, franka_grasp_rot, drawer_grasp_rot,
    franka_lfinger_pos, franka_rfinger_pos,
    gripper_forward_axis, drawer_inward_axis, gripper_up_axis, drawer_up_axis,
    num_envs, dist_reward_scale, rot_reward_scale, around_handle_reward_scale, open_reward_scale,
    finger_dist_reward_scale, action_penalty_scale, distX_offset, max_episode_length
):
    # type: (Tensor, Tensor, Tensor, Tensor, Tensor, Tensor, Tensor, Tensor, Tensor, Tensor, Tensor,
    ↪  Tensor, Tensor, Tensor, Tensor, Tensor, Tensor, int, float, float, float, float, float, float,
    ↪  float, float) -> Tuple[Tensor, Tensor, Tensor, Tensor]

    # distance from hand to the drawer
    d = torch.norm(franka_grasp_pos - drawer_grasp_pos, p=2, dim=-1)
    dist_reward = 1.0 / (1.0 + d ** 2)
    dist_reward *= dist_reward
    dist_reward = torch.where(d <= 0.02, dist_reward * 2, dist_reward)

    axis1 = tf_vector(franka_grasp_rot, gripper_forward_axis)
    axis2 = tf_vector(drawer_grasp_rot, drawer_inward_axis)
    axis3 = tf_vector(franka_grasp_rot, gripper_up_axis)
    axis4 = tf_vector(drawer_grasp_rot, drawer_up_axis)

    dot1 = torch.bmm(axis1.view(num_envs, 1, 3), axis2.view(num_envs, 3, 1)).squeeze(-1).squeeze(-1)  #
    ↪  alignment of forward axis for gripper
    dot2 = torch.bmm(axis3.view(num_envs, 1, 3), axis4.view(num_envs, 3, 1)).squeeze(-1).squeeze(-1)  #
    ↪  alignment of up axis for gripper
    # reward for matching the orientation of the hand to the drawer (fingers wrapped)
    rot_reward = 0.5 * (torch.sign(dot1) * dot1 ** 2 + torch.sign(dot2) * dot2 ** 2)

    # bonus if left finger is above the drawer handle and right below
    around_handle_reward = torch.zeros_like(rot_reward)
    around_handle_reward = torch.where(franka_lfinger_pos[:, 2] > drawer_grasp_pos[:, 2],
                                       torch.where(franka_rfinger_pos[:, 2] < drawer_grasp_pos[:, 2],
                                                   around_handle_reward + 0.5, around_handle_reward),
                                                   ↪  around_handle_reward)
    # reward for distance of each finger from the drawer
    finger_dist_reward = torch.zeros_like(rot_reward)
    lfinger_dist = torch.abs(franka_lfinger_pos[:, 2] - drawer_grasp_pos[:, 2])
    rfinger_dist = torch.abs(franka_rfinger_pos[:, 2] - drawer_grasp_pos[:, 2])
    finger_dist_reward = torch.where(franka_lfinger_pos[:, 2] > drawer_grasp_pos[:, 2],
                                     torch.where(franka_rfinger_pos[:, 2] < drawer_grasp_pos[:, 2],
                                                 (0.04 - lfinger_dist) + (0.04 - rfinger_dist),
                                                 ↪  finger_dist_reward), finger_dist_reward)

    # regularization on the actions (summed for each environment)
    action_penalty = torch.sum(actions ** 2, dim=-1)

    # how far the cabinet has been opened out
    open_reward = cabinet_dof_pos[:, 3] * around_handle_reward + cabinet_dof_pos[:, 3]  #
    ↪  drawer_top_joint

    rewards = dist_reward_scale * dist_reward + rot_reward_scale * rot_reward \
        + around_handle_reward_scale * around_handle_reward + open_reward_scale * open_reward \
        + finger_dist_reward_scale * finger_dist_reward - action_penalty_scale * action_penalty

    # bonus for opening drawer properly
    rewards = torch.where(cabinet_dof_pos[:, 3] > 0.01, rewards + 0.5, rewards)
    rewards = torch.where(cabinet_dof_pos[:, 3] > 0.2, rewards + around_handle_reward, rewards)
    rewards = torch.where(cabinet_dof_pos[:, 3] > 0.39, rewards + (2.0 * around_handle_reward), rewards)

    # prevent bad style in opening drawer
    rewards = torch.where(franka_lfinger_pos[:, 0] < drawer_grasp_pos[:, 0] - distX_offset,
                          torch.ones_like(rewards) * -1, rewards)
    rewards = torch.where(franka_rfinger_pos[:, 0] < drawer_grasp_pos[:, 0] - distX_offset,
                          torch.ones_like(rewards) * -1, rewards)

    # reset if drawer is open or max length reached
    successes = torch.where(cabinet_dof_pos[:, 3] > 0.39, torch.ones_like(successes), successes)
    goal_reach = torch.where(cabinet_dof_pos[:, 3] > 0.39, torch.ones_like(reset_goal_buf),
    ↪  torch.zeros_like(reset_goal_buf))
    reset_buf = torch.where(progress_buf >= max_episode_length - 1, torch.ones_like(reset_buf),
    ↪  reset_buf)

    consecutive_successes = torch.where(reset_buf > 0, successes * reset_buf, consecutive_successes)
    heuristic_reward, task_reward = rewards, goal_reach
    return heuristic_reward, reset_buf, task_reward, consecutive_successes
```

Isaac Gym - Manipulation Task: ***FrankaCubeStack***

```
def compute_franka_reward(
    reset_buf, progress_buf, reset_goal_buf, actions, states, reward_settings, max_episode_length
):
    # type: (Tensor, Tensor, Tensor, Tensor, Dict[str, Tensor], Dict[str, float], float) -> Tuple[Tensor,
    ↪  Tensor, Tensor]

    # Compute per-env physical parameters
    target_height = states["cubeB_size"] + states["cubeA_size"] / 2.0
    cubeA_size = states["cubeA_size"]
    cubeB_size = states["cubeB_size"]

    # distance from hand to the cubeA
    d = torch.norm(states["cubeA_pos_relative"], dim=-1)
    d_lf = torch.norm(states["cubeA_pos"] - states["eef_lf_pos"], dim=-1)
    d_rf = torch.norm(states["cubeA_pos"] - states["eef_rf_pos"], dim=-1)
    dist_reward = 1 - torch.tanh(10.0 * (d + d_lf + d_rf) / 3)

    # reward for lifting cubeA
    cubeA_height = states["cubeA_pos"][:, 2] - reward_settings["table_height"]
    cubeA_lifted = (cubeA_height - cubeA_size) > 0.04
    lift_reward = cubeA_lifted

    # how closely aligned cubeA is to cubeB (only provided if cubeA is lifted)
    offset = torch.zeros_like(states["cubeA_to_cubeB_pos"])
    offset[:, 2] = (cubeA_size + cubeB_size) / 2
    d_ab = torch.norm(states["cubeA_to_cubeB_pos"] + offset, dim=-1)
    align_reward = (1 - torch.tanh(10.0 * d_ab)) * cubeA_lifted

    # Dist reward is maximum of dist and align reward
    dist_reward = torch.max(dist_reward, align_reward)

    # final reward for stacking successfully (only if cubeA is close to target height and corresponding
    ↪  location, and gripper is not grasping)
    cubeA_align_cubeB = (torch.norm(states["cubeA_to_cubeB_pos"][:, :2], dim=-1) < 0.02)
    cubeA_on_cubeB = torch.abs(cubeA_height - target_height) < 0.02
    gripper_away_from_cubeA = (d > 0.04)
    stack_reward = cubeA_align_cubeB & cubeA_on_cubeB & gripper_away_from_cubeA

    # Compose rewards

    # We either provide the stack reward or the align + dist reward
    rewards = torch.where(
        stack_reward,
        reward_settings["r_stack_scale"] * stack_reward,
        reward_settings["r_dist_scale"] * dist_reward + reward_settings["r_lift_scale"] * lift_reward +
        ↪  reward_settings[
            "r_align_scale"] * align_reward,
    )

    # Compute resets
    reset_buf = torch.where((progress_buf >= max_episode_length - 1), torch.ones_like(reset_buf),
    ↪  reset_buf)
    goal_reach = torch.where(stack_reward > 0, torch.ones_like(reset_goal_buf),
    ↪  torch.zeros_like(reset_goal_buf))
    heuristic_reward, task_reward = rewards, goal_reach
    return heuristic_reward, task_reward, reset_buf
```

Isaac Gym - Manipulation Task: ***ShadowHand***

```python
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    goal_dist = torch.norm(object_pos - target_pos, p=2, dim=-1)

    if ignore_z_rot:
        success_tolerance = 2.0 * success_tolerance

    # Orientation alignment for the cube in hand and goal cube
    quat_diff = quat_mul(object_rot, quat_conjugate(target_rot))
    rot_dist = 2.0 * torch.asin(torch.clamp(torch.norm(quat_diff[:, 0:3], p=2, dim=-1), max=1.0))

    dist_rew = goal_dist * dist_reward_scale
    rot_rew = 1.0/(torch.abs(rot_dist) + rot_eps) * rot_reward_scale

    action_penalty = torch.sum(actions ** 2, dim=-1)

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    reward = dist_rew + action_penalty * action_penalty_scale

    # Find out which envs hit the goal and update successes count
    goal_reach = torch.where(torch.abs(rot_dist) <= success_tolerance, torch.ones_like(reset_goal_buf),
    ↪  reset_goal_buf)
    successes = successes + goal_reach

    # Success bonus: orientation is within `success_tolerance` of goal orientation
    reward = torch.where(goal_reach == 1, reward + reach_goal_bonus, reward)

    # Fall penalty: distance to the goal is larger than a threshold
    reward = torch.where(goal_dist >= fall_dist, reward + fall_penalty, reward)

    # Check env termination conditions, including maximum success number
    resets = torch.where(goal_dist >= fall_dist, torch.ones_like(reset_buf), reset_buf)
    if max_consecutive_successes > 0:
        # Reset progress buffer on goal envs if max_consecutive_successes > 0
        progress_buf = torch.where(torch.abs(rot_dist) <= success_tolerance,
        ↪  torch.zeros_like(progress_buf), progress_buf)
        resets = torch.where(successes >= max_consecutive_successes, torch.ones_like(resets), resets)
    resets = torch.where(progress_buf >= max_episode_length - 1, torch.ones_like(resets), resets)

    # Apply penalty for not reaching the goal
    if max_consecutive_successes > 0:
        reward = torch.where(progress_buf >= max_episode_length - 1, reward + 0.5 * fall_penalty, reward)

    num_resets = torch.sum(resets)
    finished_cons_successes = torch.sum(successes * resets.float())

    cons_successes = torch.where(num_resets > 0, av_factor*finished_cons_successes/num_resets + (1.0 -
    ↪  av_factor)*consecutive_successes, consecutive_successes)
    heuristic_reward, task_reward = reward, goal_reach
    return heuristic_reward, resets, task_reward, progress_buf, successes, cons_successes
```

Isaac Gym - Manipulation Task: ***AllegroHand***

```python
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    goal_dist = torch.norm(object_pos - target_pos, p=2, dim=-1)

    if ignore_z_rot:
        success_tolerance = 2.0 * success_tolerance

    # Orientation alignment for the cube in hand and goal cube
    quat_diff = quat_mul(object_rot, quat_conjugate(target_rot))
    rot_dist = 2.0 * torch.asin(torch.clamp(torch.norm(quat_diff[:, 0:3], p=2, dim=-1), max=1.0))

    dist_rew = goal_dist * dist_reward_scale
    rot_rew = 1.0/(torch.abs(rot_dist) + rot_eps) * rot_reward_scale

    action_penalty = torch.sum(actions ** 2, dim=-1)

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    reward = dist_rew + rot_rew + action_penalty * action_penalty_scale

    # Find out which envs hit the goal and update successes count
    goal_reach = torch.where(torch.abs(rot_dist) <= success_tolerance, torch.ones_like(reset_goal_buf),
    ↪  reset_goal_buf)
    successes = successes + goal_reach

    # Success bonus: orientation is within `success_tolerance` of goal orientation
    reward = torch.where(goal_reach == 1, reward + reach_goal_bonus, reward)

    # Fall penalty: distance to the goal is larger than a threshold
    reward = torch.where(goal_dist >= fall_dist, reward + fall_penalty, reward)

    # Check env termination conditions, including maximum success number
    resets = torch.where(goal_dist >= fall_dist, torch.ones_like(reset_buf), reset_buf)
    if max_consecutive_successes > 0:
        # Reset progress buffer on goal envs if max_consecutive_successes > 0
        progress_buf = torch.where(torch.abs(rot_dist) <= success_tolerance,
        ↪  torch.zeros_like(progress_buf), progress_buf)
        resets = torch.where(successes >= max_consecutive_successes, torch.ones_like(resets), resets)

    timed_out = progress_buf >= max_episode_length - 1
    resets = torch.where(timed_out, torch.ones_like(resets), resets)

    # Apply penalty for not reaching the goal
    if max_consecutive_successes > 0:
        reward = torch.where(timed_out, reward + 0.5 * fall_penalty, reward)

    num_resets = torch.sum(resets)
    finished_cons_successes = torch.sum(successes * resets.float())

    cons_successes = torch.where(num_resets > 0, av_factor*finished_cons_successes/num_resets + (1.0 -
    ↪  av_factor)*consecutive_successes, consecutive_successes)
    heuristic_reward, task_reward = reward, goal_reach
    return heuristic_reward, resets, task_reward, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandOver***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    goal_dist = torch.norm(target_pos - object_pos, p=2, dim=-1)
    if ignore_z_rot:
        success_tolerance = 2.0 * success_tolerance

    # Orientation alignment for the cube in hand and goal cube
    quat_diff = quat_mul(object_rot, quat_conjugate(target_rot))
    rot_dist = 2.0 * torch.asin(torch.clamp(torch.norm(quat_diff[:, 0:3], p=2, dim=-1), max=1.0))

    dist_rew = goal_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    reward = torch.exp(-0.2*(dist_rew * dist_reward_scale + rot_dist))

    # Find out which envs hit the goal and update successes count
    goal_resets = torch.where(torch.abs(goal_dist) <= 0, torch.ones_like(reset_goal_buf), reset_goal_buf)
    successes = torch.where(successes == 0,
                    torch.where(goal_dist < 0.03, torch.ones_like(successes), successes), successes)

    # Success bonus: orientation is within `success_tolerance` of goal orientation
    reward = torch.where(goal_resets == 1, reward + reach_goal_bonus, reward)

    # Fall penalty: distance to the goal is larger than a threashold
    reward = torch.where(object_pos[:, 2] <= 0.2, reward + fall_penalty, reward)

    # Check env termination conditions, including maximum success number
    resets = torch.where(object_pos[:, 2] <= 0.2, torch.ones_like(reset_buf), reset_buf)
    if max_consecutive_successes > 0:
        # Reset progress buffer on goal envs if max_consecutive_successes > 0
        progress_buf = torch.where(torch.abs(rot_dist) <= success_tolerance,
        ↪  torch.zeros_like(progress_buf), progress_buf)
        resets = torch.where(successes >= max_consecutive_successes, torch.ones_like(resets), resets)
    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    # Apply penalty for not reaching the goal
    if max_consecutive_successes > 0:
        reward = torch.where(progress_buf >= max_episode_length, reward + 0.5 * fall_penalty, reward)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes).mean()

    goal_reach = torch.where(goal_dist <= 0.03, torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach
    return heuristic_reward, task_reward, resets, goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandCatchUnderarm***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    goal_dist = torch.norm(target_pos - object_pos, p=2, dim=-1)

    if ignore_z_rot:
        success_tolerance = 2.0 * success_tolerance

    # Orientation alignment for the cube in hand and goal cube
    quat_diff = quat_mul(object_rot, quat_conjugate(target_rot))
    rot_dist = 2.0 * torch.asin(torch.clamp(torch.norm(quat_diff[:, 0:3], p=2, dim=-1), max=1.0))

    dist_rew = goal_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    reward = torch.exp(-0.2*(dist_rew * dist_reward_scale + rot_dist))

    # Find out which envs hit the goal and update successes count
    goal_resets = torch.where(torch.abs(goal_dist) <= 0, torch.ones_like(reset_goal_buf), reset_goal_buf)
    successes = torch.where(successes == 0,
                    torch.where(goal_dist < 0.03, torch.ones_like(successes), successes), successes)

    # Fall penalty: distance to the goal is larger than a threashold
    reward = torch.where(object_pos[:, 2] <= 0.1, reward + fall_penalty, reward)

    # Check env termination conditions, including maximum success number
    resets = torch.where(object_pos[:, 2] <= 0.1, torch.ones_like(reset_buf), reset_buf)
    if max_consecutive_successes > 0:
        # Reset progress buffer on goal envs if max_consecutive_successes > 0
        progress_buf = torch.where(torch.abs(rot_dist) <= success_tolerance,
        ↪  torch.zeros_like(progress_buf), progress_buf)
        resets = torch.where(successes >= max_consecutive_successes, torch.ones_like(resets), resets)
    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    # Apply penalty for not reaching the goal
    if max_consecutive_successes > 0:
        reward = torch.where(progress_buf >= max_episode_length, reward + 0.5 * fall_penalty, reward)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes)
    goal_reach = torch.where(goal_dist <= 0.03,
                             torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach
    return heuristic_reward, task_reward, resets, goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandCatchOver2Underarm***

```python
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, left_hand_base_pos,
    ↪  right_hand_base_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool, device:
    ↪  str
):
    # Distance from the hand to the object
    goal_dist = torch.norm(target_pos - object_pos, p=2, dim=-1)
    if ignore_z_rot:
        success_tolerance = 2.0 * success_tolerance

    # Orientation alignment for the cube in hand and goal cube
    quat_diff = quat_mul(object_rot, quat_conjugate(target_rot))
    rot_dist = 2.0 * torch.asin(torch.clamp(torch.norm(quat_diff[:, 0:3], p=2, dim=-1), max=1.0))

    dist_rew = goal_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    reward = torch.exp(-0.2*(dist_rew * dist_reward_scale + rot_dist))

    # Find out which envs hit the goal and update successes count
    goal_resets = torch.where(torch.abs(goal_dist) <= 0, torch.ones_like(reset_goal_buf), reset_goal_buf)
    successes = torch.where(successes == 0,
                    torch.where(goal_dist < 0.03, torch.ones_like(successes), successes), successes)

    # Check env termination conditions, including maximum success number
    right_hand_base_dist = torch.norm(right_hand_base_pos - torch.tensor([0.0, 0.0, 0.5],
    ↪  dtype=torch.float, device=device), p=2, dim=-1)
    left_hand_base_dist = torch.norm(left_hand_base_pos - torch.tensor([0.0, -0.8, 0.5],
    ↪  dtype=torch.float, device=device), p=2, dim=-1)

    resets = torch.where(right_hand_base_dist >= 0.1, torch.ones_like(reset_buf), reset_buf)
    resets = torch.where(left_hand_base_dist >= 0.1, torch.ones_like(resets), resets)
    resets = torch.where(object_pos[:, 2] <= 0.3, torch.ones_like(resets), resets)
    if max_consecutive_successes > 0:
        # Reset progress buffer on goal envs if max_consecutive_successes > 0
        progress_buf = torch.where(torch.abs(rot_dist) <= success_tolerance,
        ↪  torch.zeros_like(progress_buf), progress_buf)
        resets = torch.where(successes >= max_consecutive_successes, torch.ones_like(resets), resets)
    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    # Apply penalty for not reaching the goal
    if max_consecutive_successes > 0:
        reward = torch.where(progress_buf >= max_episode_length, reward + 0.5 * fall_penalty, reward)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes)
    goal_reach = torch.where(goal_dist <= 0.03,
                          torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach
    return heuristic_reward, task_reward, resets, goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandCatchAbreast***

```python
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, left_hand_pos,
    ↪  right_hand_pos, left_hand_base_pos, right_hand_base_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool, device:
    ↪  str
):
    # Distance from the hand to the object
    goal_dist = torch.norm(target_pos - object_pos, p=2, dim=-1)

    if ignore_z_rot:
        success_tolerance = 2.0 * success_tolerance

    # Orientation alignment for the cube in hand and goal cube
    quat_diff = quat_mul(object_rot, quat_conjugate(target_rot))
    rot_dist = 2.0 * torch.asin(torch.clamp(torch.norm(quat_diff[:, 0:3], p=2, dim=-1), max=1.0))

    dist_rew = goal_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    reward = torch.exp(-0.2*(dist_rew * dist_reward_scale + rot_dist))

    # Find out which envs hit the goal and update successes count
    goal_resets = torch.where(torch.abs(goal_dist) <= 0, torch.ones_like(reset_goal_buf), reset_goal_buf)

    successes = torch.where(successes == 0,
                    torch.where(goal_dist < 0.03, torch.ones_like(successes), successes), successes)

    # Fall penalty: distance to the goal is larger than a threashold
    reward = torch.where(object_pos[:, 2] <= 0.2, reward + fall_penalty, reward)

    # Check env termination conditions, including maximum success number
    right_hand_base_dist = torch.norm(right_hand_base_pos - torch.tensor([-0.3, -0.55, 0.5],
    ↪  dtype=torch.float, device=device), p=2, dim=-1)
    left_hand_base_dist = torch.norm(left_hand_base_pos - torch.tensor([-0.3, -1.15, 0.5],
    ↪  dtype=torch.float, device=device), p=2, dim=-1)

    resets = torch.where(right_hand_base_dist >= 0.1, torch.ones_like(reset_buf), reset_buf)
    resets = torch.where(left_hand_base_dist >= 0.1, torch.ones_like(resets), resets)

    resets = torch.where(object_pos[:, 2] <= 0.2, torch.ones_like(resets), resets)
    if max_consecutive_successes > 0:
        # Reset progress buffer on goal envs if max_consecutive_successes > 0
        progress_buf = torch.where(torch.abs(rot_dist) <= success_tolerance,
        ↪  torch.zeros_like(progress_buf), progress_buf)
        resets = torch.where(successes >= max_consecutive_successes, torch.ones_like(resets), resets)
    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    # Apply penalty for not reaching the goal
    if max_consecutive_successes > 0:
        reward = torch.where(progress_buf >= max_episode_length, reward + 0.5 * fall_penalty, reward)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes)
    goal_reach = torch.where(goal_dist <= 0.03,
                             torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach
    return heuristic_reward, task_reward, resets, goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandTwoCatchUnderarm***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, object_another_pos,
    ↪  object_another_rot, target_another_pos, target_another_rot,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    goal_dist = torch.norm(target_pos - object_pos, p=2, dim=-1)
    if ignore_z_rot:
        success_tolerance = 2.0 * success_tolerance

    goal_another_dist = torch.norm(target_another_pos - object_another_pos, p=2, dim=-1)
    if ignore_z_rot:
        success_tolerance = 2.0 * success_tolerance

    # Orientation alignment for the cube in hand and goal cube
    quat_diff = quat_mul(object_rot, quat_conjugate(target_rot))
    rot_dist = 2.0 * torch.asin(torch.clamp(torch.norm(quat_diff[:, 0:3], p=2, dim=-1), max=1.0))

    quat_another_diff = quat_mul(object_another_rot, quat_conjugate(target_another_rot))
    rot_another_dist = 2.0 * torch.asin(torch.clamp(torch.norm(quat_another_diff[:, 0:3], p=2, dim=-1),
    ↪  max=1.0))

    dist_rew = goal_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    reward = torch.exp(-0.2*(dist_rew * dist_reward_scale + rot_dist)) +
    ↪  torch.exp(-0.2*(goal_another_dist * dist_reward_scale + rot_another_dist))

    # Find out which envs hit the goal and update successes count
    goal_resets = torch.where(torch.abs(goal_dist) <= 0, torch.ones_like(reset_goal_buf), reset_goal_buf)
    successes = torch.where(successes == 0,
                    torch.where(goal_dist + goal_another_dist < 0.06, torch.ones_like(successes),
                    ↪  successes), successes)

    # Fall penalty: distance to the goal is larger than a threashold
    reward = torch.where(object_pos[:, 2] <= 0.2, reward + fall_penalty, reward)
    reward = torch.where(object_another_pos[:, 2] <= 0.2, reward + fall_penalty, reward)

    # Check env termination conditions, including maximum success number
    resets = torch.where(object_pos[:, 2] <= 0.2, torch.ones_like(reset_buf), reset_buf)
    resets = torch.where(object_another_pos[:, 2] <= 0.2, torch.ones_like(reset_buf), resets)

    if max_consecutive_successes > 0:
        # Reset progress buffer on goal envs if max_consecutive_successes > 0
        progress_buf = torch.where(torch.abs(rot_dist) <= success_tolerance,
        ↪  torch.zeros_like(progress_buf), progress_buf)
        resets = torch.where(successes >= max_consecutive_successes, torch.ones_like(resets), resets)
    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    # Apply penalty for not reaching the goal
    if max_consecutive_successes > 0:
        reward = torch.where(progress_buf >= max_episode_length, reward + 0.5 * fall_penalty, reward)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes).mean()
    goal_reach = torch.where(goal_dist + goal_another_dist <= 0.06,
                             torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach
    return heuristic_reward, task_reward, resets, goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandLiftUnderarm***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, pot_left_handle_pos,
    ↪  pot_right_handle_pos,
    left_hand_pos, right_hand_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    goal_dist = torch.norm(target_pos - object_pos, p=2, dim=-1)

    # goal_dist = target_pos[:, 2] - object_pos[:, 2]
    right_hand_dist = torch.norm(pot_right_handle_pos - right_hand_pos, p=2, dim=-1)
    left_hand_dist = torch.norm(pot_left_handle_pos - left_hand_pos, p=2, dim=-1)

    right_hand_dist_rew = right_hand_dist
    left_hand_dist_rew = left_hand_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    up_rew = torch.zeros_like(right_hand_dist_rew)
    up_rew = torch.where(right_hand_dist < 0.08,
                        torch.where(left_hand_dist < 0.08,
                                        3*(0.385 - goal_dist), up_rew), up_rew)

    reward = 0.2 - right_hand_dist_rew - left_hand_dist_rew + up_rew

    resets = torch.where(object_pos[:, 2] <= 0.3, torch.ones_like(reset_buf), reset_buf)
    resets = torch.where(right_hand_dist >= 0.2, torch.ones_like(resets), resets)
    resets = torch.where(left_hand_dist >= 0.2, torch.ones_like(resets), resets)

    # Find out which envs hit the goal and update successes count
    successes = torch.where(successes == 0,
                    torch.where(goal_dist < 0.05, torch.ones_like(successes), successes), successes)

    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    goal_resets = torch.zeros_like(resets)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes).mean()
    goal_reach = torch.where(goal_dist <= 0.05,
                            torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandDoorOpenInward***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, door_left_handle_pos,
    ↪  door_right_handle_pos,
    left_hand_pos, right_hand_pos, right_hand_ff_pos, right_hand_mf_pos, right_hand_rf_pos,
    ↪  right_hand_lf_pos, right_hand_th_pos,
    left_hand_ff_pos, left_hand_mf_pos, left_hand_rf_pos, left_hand_lf_pos, left_hand_th_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    goal_dist = torch.norm(target_pos - object_pos, p=2, dim=-1)

    right_hand_finger_dist = (torch.norm(door_right_handle_pos - right_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(door_right_handle_pos - right_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(door_right_handle_pos - right_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(door_right_handle_pos - right_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(door_right_handle_pos - right_hand_th_pos, p=2, dim=-1))
    left_hand_finger_dist = (torch.norm(door_left_handle_pos - left_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(door_left_handle_pos - left_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(door_left_handle_pos - left_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(door_left_handle_pos - left_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(door_left_handle_pos - left_hand_th_pos, p=2, dim=-1))

    right_hand_dist_rew = right_hand_finger_dist
    left_hand_dist_rew = left_hand_finger_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    up_rew = torch.zeros_like(right_hand_dist_rew)
    up_rew = torch.where(right_hand_finger_dist < 0.5,
                    torch.where(left_hand_finger_dist < 0.5,
                                    torch.abs(door_right_handle_pos[:, 1] - door_left_handle_pos[:, 1]) *
                                    ↪  2, up_rew), up_rew)

    reward = 2 - right_hand_dist_rew - left_hand_dist_rew + up_rew

    resets = torch.where(right_hand_finger_dist >= 1.5, torch.ones_like(reset_buf), reset_buf)
    resets = torch.where(left_hand_finger_dist >= 1.5, torch.ones_like(resets), resets)

    # Find out which envs hit the goal and update successes count
    successes = torch.where(successes == 0,
                    torch.where(torch.abs(door_right_handle_pos[:, 1] - door_left_handle_pos[:, 1]) >
                    ↪  0.5, torch.ones_like(successes), successes), successes)

    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    goal_resets = torch.zeros_like(resets)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes).mean()
    goal_reach = torch.where(torch.abs(door_right_handle_pos[:, 1] - door_left_handle_pos[:, 1]) >= 0.5,
                            torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandDoorOpenOutward***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, door_left_handle_pos,
    ↪  door_right_handle_pos,
    left_hand_pos, right_hand_pos, right_hand_ff_pos, right_hand_mf_pos, right_hand_rf_pos,
    ↪  right_hand_lf_pos, right_hand_th_pos,
    left_hand_ff_pos, left_hand_mf_pos, left_hand_rf_pos, left_hand_lf_pos, left_hand_th_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    right_hand_finger_dist = (torch.norm(door_right_handle_pos - right_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(door_right_handle_pos - right_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(door_right_handle_pos - right_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(door_right_handle_pos - right_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(door_right_handle_pos - right_hand_th_pos, p=2, dim=-1))
    left_hand_finger_dist = (torch.norm(door_left_handle_pos - left_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(door_left_handle_pos - left_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(door_left_handle_pos - left_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(door_left_handle_pos - left_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(door_left_handle_pos - left_hand_th_pos, p=2, dim=-1))

    right_hand_dist_rew = right_hand_finger_dist
    left_hand_dist_rew = left_hand_finger_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    up_rew = torch.zeros_like(right_hand_dist_rew)
    up_rew = torch.where(right_hand_finger_dist < 0.5,
                torch.where(left_hand_finger_dist < 0.5,
                                torch.abs(door_right_handle_pos[:, 1] - door_left_handle_pos[:, 1]) *
                                ↪  2, up_rew), up_rew)
    reward = 2 - right_hand_dist_rew - left_hand_dist_rew + up_rew

    resets = torch.where(right_hand_finger_dist >= 1.5, torch.ones_like(reset_buf), reset_buf)
    resets = torch.where(left_hand_finger_dist >= 1.5, torch.ones_like(resets), resets)

    # Find out which envs hit the goal and update successes count
    successes = torch.where(successes == 0,
                torch.where(torch.abs(door_right_handle_pos[:, 1] - door_left_handle_pos[:, 1]) >
                ↪  0.5, torch.ones_like(successes), successes), successes)

    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    goal_resets = torch.zeros_like(resets)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes).mean()
    goal_reach = torch.where(torch.abs(door_right_handle_pos[:, 1] - door_left_handle_pos[:, 1]) >= 0.5,
                            torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandDoorCloseInward***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, door_left_handle_pos,
    ↪  door_right_handle_pos,
    left_hand_pos, right_hand_pos, right_hand_ff_pos, right_hand_mf_pos, right_hand_rf_pos,
    ↪  right_hand_lf_pos, right_hand_th_pos,
    left_hand_ff_pos, left_hand_mf_pos, left_hand_rf_pos, left_hand_lf_pos, left_hand_th_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    right_hand_finger_dist = (torch.norm(door_right_handle_pos - right_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(door_right_handle_pos - right_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(door_right_handle_pos - right_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(door_right_handle_pos - right_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(door_right_handle_pos - right_hand_th_pos, p=2, dim=-1))
    left_hand_finger_dist = (torch.norm(door_left_handle_pos - left_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(door_left_handle_pos - left_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(door_left_handle_pos - left_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(door_left_handle_pos - left_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(door_left_handle_pos - left_hand_th_pos, p=2, dim=-1))

    right_hand_dist_rew = right_hand_finger_dist
    left_hand_dist_rew = left_hand_finger_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    up_rew = torch.zeros_like(right_hand_dist_rew)
    up_rew = torch.where(right_hand_finger_dist < 0.5,
                    torch.where(left_hand_finger_dist < 0.5,
                                    1 - torch.abs(door_right_handle_pos[:, 1] - door_left_handle_pos[:,
                                    ↪  1]) * 2, up_rew), up_rew)

    reward = 2 - right_hand_dist_rew - left_hand_dist_rew + up_rew

    resets = torch.where(right_hand_finger_dist >= 1.5, torch.ones_like(reset_buf), reset_buf)
    resets = torch.where(left_hand_finger_dist >= 1.5, torch.ones_like(resets), resets)

    # Find out which envs hit the goal and update successes count
    successes = torch.where(successes == 0,
                    torch.where(torch.abs(door_right_handle_pos[:, 1] - door_left_handle_pos[:, 1]) <
                    ↪  0.5, torch.ones_like(successes), successes), successes)

    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    goal_resets = torch.zeros_like(resets)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes)
    goal_reach = torch.where(torch.abs(door_right_handle_pos[:, 1] - door_left_handle_pos[:, 1]) <= 0.5,
                            torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach      return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandDoorCloseOutward***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, door_left_handle_pos,
    ↪  door_right_handle_pos,
    left_hand_pos, right_hand_pos, right_hand_ff_pos, right_hand_mf_pos, right_hand_rf_pos,
    ↪  right_hand_lf_pos, right_hand_th_pos,
    left_hand_ff_pos, left_hand_mf_pos, left_hand_rf_pos, left_hand_lf_pos, left_hand_th_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    right_hand_finger_dist = (torch.norm(door_right_handle_pos - right_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(door_right_handle_pos - right_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(door_right_handle_pos - right_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(door_right_handle_pos - right_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(door_right_handle_pos - right_hand_th_pos, p=2, dim=-1))
    left_hand_finger_dist = (torch.norm(door_left_handle_pos - left_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(door_left_handle_pos - left_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(door_left_handle_pos - left_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(door_left_handle_pos - left_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(door_left_handle_pos - left_hand_th_pos, p=2, dim=-1))

    right_hand_dist_rew = right_hand_finger_dist
    left_hand_dist_rew = left_hand_finger_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    up_rew = torch.zeros_like(right_hand_dist_rew)
    up_rew = torch.where(right_hand_finger_dist < 0.5,
                    torch.where(left_hand_finger_dist < 0.5,
                                    1 - torch.abs(door_right_handle_pos[:, 1] - door_left_handle_pos[:,
                                    ↪  1]) * 2, up_rew), up_rew)

    reward = 6 - right_hand_dist_rew - left_hand_dist_rew + up_rew

    resets = torch.where(right_hand_finger_dist >= 3, torch.ones_like(reset_buf), reset_buf)
    resets = torch.where(left_hand_finger_dist >= 3, torch.ones_like(resets), resets)

    # Find out which envs hit the goal and update successes count
    successes = torch.where(successes == 0,
                    torch.where(torch.abs(door_right_handle_pos[:, 1] - door_left_handle_pos[:, 1]) <
                    ↪  0.5, torch.ones_like(successes), successes), successes)

    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    goal_resets = torch.zeros_like(resets)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes).mean()
    goal_reach = torch.where(torch.abs(door_right_handle_pos[:, 1] - door_left_handle_pos[:, 1]) <= 0.5,
                torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandSpin***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    goal_dist = torch.norm(object_pos - target_pos, p=2, dim=-1)

    if ignore_z_rot:
        success_tolerance = 2.0 * success_tolerance

    # Orientation alignment
    # Modified so pen is symmetrical; since we only rotate around the z axis,
    quat_diff_1 = quat_mul(object_rot, quat_conjugate(target_rot))
    rot_dist_1 = 2.0 * torch.asin(torch.clamp(torch.norm(quat_diff_1[:, 0:3], p=2, dim=-1), max=1.0))
    quat_diff_2 = quat_mul(object_rot, quat_conjugate(flip_orientation(target_rot)))
    rot_dist_2 = 2.0 * torch.asin(torch.clamp(torch.norm(quat_diff_2[:, 0:3], p=2, dim=-1), max=1.0))
    rot_dist = torch.min(rot_dist_1, rot_dist_2)

    dist_rew = goal_dist * dist_reward_scale
    rot_rew = 1.0/(torch.abs(rot_dist) + rot_eps) * rot_reward_scale

    action_penalty = torch.sum(actions ** 2, dim=-1)

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    reward = dist_rew + rot_rew + action_penalty * action_penalty_scale

    # Find out which envs hit the goal and update successes count
    goal_resets = torch.where(torch.abs(rot_dist) <= success_tolerance, torch.ones_like(reset_goal_buf),
    ↪  reset_goal_buf)
    successes = successes + goal_resets

    # Success bonus: orientation is within `success_tolerance` of goal orientation
    reward = torch.where(goal_resets == 1, reward + reach_goal_bonus, reward)

    # Fall penalty: distance to the goal is larger than a threshold
    reward = torch.where(goal_dist >= fall_dist, reward + fall_penalty, reward)

    # Check env termination conditions, including maximum success number
    resets = torch.where(goal_dist >= fall_dist, torch.ones_like(reset_buf), reset_buf)
    if max_consecutive_successes > 0:
        # Reset progress buffer on goal envs if max_consecutive_successes > 0
        progress_buf = torch.where(torch.abs(rot_dist) <= success_tolerance,
        ↪  torch.zeros_like(progress_buf), progress_buf)
        resets = torch.where(successes >= max_consecutive_successes, torch.ones_like(resets), resets)
    resets = torch.where(progress_buf >= max_episode_length - 1, torch.ones_like(resets), resets)

    # Apply penalty for not reaching the goal
    if max_consecutive_successes > 0:
        reward = torch.where(progress_buf >= max_episode_length - 1, reward + 0.5 * fall_penalty, reward)

    num_resets = torch.sum(resets)
    finished_cons_successes = torch.sum(successes * resets.float())

    cons_successes = torch.where(num_resets > 0, av_factor*finished_cons_successes/num_resets + (1.0 -
    ↪  av_factor)*consecutive_successes, consecutive_successes)
    goal_reach = torch.where(torch.abs(rot_dist) <= success_tolerance,
                             torch.ones_like(reset_goal_buf), torch.zeros_like(reset_goal_buf))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandUpsideDown***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    goal_dist = torch.norm(object_pos - target_pos, p=2, dim=-1)

    if ignore_z_rot:
        success_tolerance = 2.0 * success_tolerance

    # Orientation alignment for the cube in hand and goal cube
    quat_diff = quat_mul(object_rot, quat_conjugate(target_rot))
    rot_dist = 2.0 * torch.asin(torch.clamp(torch.norm(quat_diff[:, 0:3], p=2, dim=-1), max=1.0))

    dist_rew = goal_dist * dist_reward_scale
    rot_rew = 1.0/(torch.abs(rot_dist) + rot_eps) * rot_reward_scale

    action_penalty = torch.sum(actions ** 2, dim=-1)

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    reward = dist_rew + rot_rew + action_penalty * action_penalty_scale

    # Find out which envs hit the goal and update successes count
    goal_resets = torch.where(torch.abs(rot_dist) <= success_tolerance, torch.ones_like(reset_goal_buf),
    ↪  reset_goal_buf)
    successes = successes + goal_resets

    # Success bonus: orientation is within `success_tolerance` of goal orientation
    reward = torch.where(goal_resets == 1, reward + reach_goal_bonus, reward)

    # Fall penalty: distance to the goal is larger than a threshold
    reward = torch.where(goal_dist >= fall_dist, reward + fall_penalty, reward)

    # Check env termination conditions, including maximum success number
    resets = torch.where(goal_dist >= fall_dist, torch.ones_like(reset_buf), reset_buf)
    if max_consecutive_successes > 0:
        # Reset progress buffer on goal envs if max_consecutive_successes > 0
        progress_buf = torch.where(torch.abs(rot_dist) <= success_tolerance,
        ↪  torch.zeros_like(progress_buf), progress_buf)
        resets = torch.where(successes >= max_consecutive_successes, torch.ones_like(resets), resets)
    resets = torch.where(progress_buf >= max_episode_length - 1, torch.ones_like(resets), resets)

    # Apply penalty for not reaching the goal
    if max_consecutive_successes > 0:
        reward = torch.where(progress_buf >= max_episode_length - 1, reward + 0.5 * fall_penalty, reward)

    num_resets = torch.sum(resets)
    finished_cons_successes = torch.sum(successes * resets.float())

    cons_successes = torch.where(num_resets > 0, av_factor*finished_cons_successes/num_resets + (1.0 -
    ↪  av_factor)*consecutive_successes, consecutive_successes)
    goal_reach = torch.where(torch.abs(rot_dist) <= success_tolerance, torch.ones_like(reset_goal_buf),
    ↪  torch.zeros_like(reset_goal_buf))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandBlockStack***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, block_right_handle_pos,
    ↪  block_left_handle_pos,
    left_hand_pos, right_hand_pos, right_hand_ff_pos, right_hand_mf_pos, right_hand_rf_pos,
    ↪  right_hand_lf_pos, right_hand_th_pos,
    left_hand_ff_pos, left_hand_mf_pos, left_hand_rf_pos, left_hand_lf_pos, left_hand_th_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    stack_pos1 = target_pos.clone()
    stack_pos2 = target_pos.clone()

    stack_pos1[:, 1] -= 0.1
    stack_pos2[:, 1] -= 0.1
    stack_pos1[:, 2] += 0.05

    goal_dist1 = torch.norm(stack_pos1 - block_left_handle_pos, p=2, dim=-1)
    goal_dist2 = torch.norm(stack_pos2 - block_right_handle_pos, p=2, dim=-1)

    right_hand_finger_dist = (torch.norm(block_right_handle_pos - right_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(block_right_handle_pos - right_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(block_right_handle_pos - right_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(block_right_handle_pos - right_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(block_right_handle_pos - right_hand_th_pos, p=2, dim=-1))
    left_hand_finger_dist = (torch.norm(block_left_handle_pos - left_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(block_left_handle_pos - left_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(block_left_handle_pos - left_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(block_left_handle_pos - left_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(block_left_handle_pos - left_hand_th_pos, p=2, dim=-1))

    right_hand_dist_rew = right_hand_finger_dist
    left_hand_dist_rew = left_hand_finger_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    up_rew = torch.zeros_like(right_hand_dist_rew)
    up_rew = torch.where(right_hand_finger_dist < 0.5,
                    torch.where(left_hand_finger_dist < 0.5,
                        (0.24 - goal_dist1 - goal_dist2) * 2, up_rew), up_rew)

    stack_rew = torch.zeros_like(right_hand_dist_rew)
    stack_rew = torch.where(goal_dist2 < 0.07,
                    torch.where(goal_dist1 < 0.07,
                        (0.05-torch.abs(stack_pos1[:, 2] - block_left_handle_pos[:, 2])) * 50
                        ↪  ,stack_rew),stack_rew)

    reward = 1.5 - right_hand_dist_rew - left_hand_dist_rew + up_rew + stack_rew

    resets = torch.where(right_hand_dist_rew <= 0, torch.ones_like(reset_buf), reset_buf)
    resets = torch.where(right_hand_finger_dist >= 0.75, torch.ones_like(resets), resets)
    resets = torch.where(left_hand_finger_dist >= 0.75, torch.ones_like(resets), resets)

    # Find out which envs hit the goal and update successes count
    successes = torch.where(successes == 0,
                    torch.where(stack_rew > 1, torch.ones_like(successes), successes), successes)

    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    goal_resets = torch.zeros_like(resets)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes)
    goal_reach = torch.where(stack_rew >= 1,
                            torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandBottleCap***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, bottle_cap_pos,
    ↪  bottle_pos, bottle_cap_up,
    left_hand_pos, right_hand_pos, right_hand_ff_pos, right_hand_mf_pos, right_hand_rf_pos,
    ↪  right_hand_lf_pos, right_hand_th_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    right_hand_dist = torch.norm(bottle_cap_pos - right_hand_pos, p=2, dim=-1)
    left_hand_dist = torch.norm(bottle_pos - left_hand_pos, p=2, dim=-1)

    right_hand_finger_dist = (torch.norm(bottle_cap_pos - right_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(bottle_cap_pos - right_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(bottle_cap_pos - right_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(bottle_cap_pos - right_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(bottle_cap_pos - right_hand_th_pos, p=2, dim=-1))

    right_hand_dist_rew = right_hand_finger_dist
    left_hand_dist_rew = left_hand_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    up_rew = torch.zeros_like(right_hand_dist_rew)

    up_rew =  torch.where(right_hand_finger_dist <= 0.3, torch.norm(bottle_cap_up - bottle_pos, p=2,
    ↪  dim=-1) * 30, up_rew)

    reward = 2.0 - right_hand_dist_rew - left_hand_dist_rew + up_rew

    resets = torch.where(bottle_cap_pos[:, 2] <= 0.5, torch.ones_like(reset_buf), reset_buf)
    resets = torch.where(right_hand_dist >= 0.5, torch.ones_like(resets), resets)
    resets = torch.where(left_hand_dist >= 0.2, torch.ones_like(resets), resets)

    # Find out which envs hit the goal and update successes count
    successes = torch.where(successes == 0,
                    torch.where(torch.norm(bottle_cap_up - bottle_pos, p=2, dim=-1) > 0.03,
                    ↪  torch.ones_like(successes), successes), successes)

    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    goal_resets = torch.zeros_like(resets)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes)
    goal_reach = torch.where(torch.norm(bottle_cap_up - bottle_pos, p=2, dim=-1) >= 0.03,
                            torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandGraspAndPlace***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, block_right_handle_pos,
    ↪  block_left_handle_pos,
    left_hand_pos, right_hand_pos, right_hand_ff_pos, right_hand_mf_pos, right_hand_rf_pos,
    ↪  right_hand_lf_pos, right_hand_th_pos,
    left_hand_ff_pos, left_hand_mf_pos, left_hand_rf_pos, left_hand_lf_pos, left_hand_th_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    right_hand_finger_dist = (torch.norm(block_right_handle_pos - right_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(block_right_handle_pos - right_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(block_right_handle_pos - right_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(block_right_handle_pos - right_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(block_right_handle_pos - right_hand_th_pos, p=2, dim=-1))
    left_hand_finger_dist = (torch.norm(block_left_handle_pos - left_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(block_left_handle_pos - left_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(block_left_handle_pos - left_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(block_left_handle_pos - left_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(block_left_handle_pos - left_hand_th_pos, p=2, dim=-1))

    right_hand_dist_rew = torch.exp(-10 * right_hand_finger_dist)
    left_hand_dist_rew = torch.exp(-10 * left_hand_finger_dist)

    up_rew = torch.zeros_like(right_hand_dist_rew)
    up_rew = torch.exp(-10 * torch.norm(block_right_handle_pos - block_left_handle_pos, p=2, dim=-1)) * 2

    reward = right_hand_dist_rew + left_hand_dist_rew + up_rew

    resets = torch.where(right_hand_dist_rew <= 0, torch.ones_like(reset_buf), reset_buf)
    resets = torch.where(right_hand_finger_dist >= 1.5, torch.ones_like(resets), resets)
    resets = torch.where(left_hand_finger_dist >= 1.5, torch.ones_like(resets), resets)

    # Find out which envs hit the goal and update successes count
    successes = torch.where(successes == 0,
                    torch.where(torch.norm(block_right_handle_pos - block_left_handle_pos, p=2, dim=-1) <
                    ↪  0.2, torch.ones_like(successes), successes), successes)

    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    goal_resets = torch.zeros_like(resets)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes).mean()
    goal_reach = torch.where(torch.norm(block_right_handle_pos - block_left_handle_pos, p=2, dim=-1) <=
    ↪  0.2,
                            torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandKettle***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, kettle_handle_pos,
    ↪  bucket_handle_pos, kettle_spout_pos,
    left_hand_pos, right_hand_pos, right_hand_ff_pos, right_hand_mf_pos, right_hand_rf_pos,
    ↪  right_hand_lf_pos, right_hand_th_pos,
    left_hand_ff_pos, left_hand_mf_pos, left_hand_rf_pos, left_hand_lf_pos, left_hand_th_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    right_hand_finger_dist = (torch.norm(kettle_handle_pos - right_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(kettle_handle_pos - right_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(kettle_handle_pos - right_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(kettle_handle_pos - right_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(kettle_handle_pos - right_hand_th_pos, p=2, dim=-1))
    left_hand_finger_dist = (torch.norm(bucket_handle_pos - left_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(bucket_handle_pos - left_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(bucket_handle_pos - left_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(bucket_handle_pos - left_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(bucket_handle_pos - left_hand_th_pos, p=2, dim=-1))

    right_hand_dist_rew = right_hand_finger_dist
    left_hand_dist_rew = left_hand_finger_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    up_rew = torch.zeros_like(right_hand_dist_rew)
    up_rew = torch.where(right_hand_finger_dist < 0.7,
                    torch.where(left_hand_finger_dist < 0.7,
                                    0.5 - torch.norm(bucket_handle_pos - kettle_spout_pos, p=2, dim=-1) *
                                    ↪  2, up_rew), up_rew)

    reward = 1 + up_rew - right_hand_dist_rew - left_hand_dist_rew

    resets = torch.where(bucket_handle_pos[:, 2] <= 0.2, torch.ones_like(reset_buf), reset_buf)

    # Find out which envs hit the goal and update successes count
    successes = torch.where(successes == 0,
                    torch.where(torch.norm(bucket_handle_pos - kettle_spout_pos, p=2, dim=-1) < 0.05,
                    ↪  torch.ones_like(successes), successes), successes)

    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    goal_resets = torch.zeros_like(resets)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes).mean()
    goal_reach = torch.where(torch.norm(bucket_handle_pos - kettle_spout_pos, p=2, dim=-1) <= 0.05,
                            torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandPen***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, pen_right_handle_pos,
    ↪  pen_left_handle_pos,
    left_hand_pos, right_hand_pos, right_hand_ff_pos, right_hand_mf_pos, right_hand_rf_pos,
    ↪  right_hand_lf_pos, right_hand_th_pos,
    left_hand_ff_pos, left_hand_mf_pos, left_hand_rf_pos, left_hand_lf_pos, left_hand_th_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    right_hand_finger_dist = (torch.norm(pen_right_handle_pos - right_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(pen_right_handle_pos - right_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(pen_right_handle_pos - right_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(pen_right_handle_pos - right_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(pen_right_handle_pos - right_hand_th_pos, p=2, dim=-1))
    left_hand_finger_dist = (torch.norm(pen_left_handle_pos - left_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(pen_left_handle_pos - left_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(pen_left_handle_pos - left_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(pen_left_handle_pos - left_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(pen_left_handle_pos - left_hand_th_pos, p=2, dim=-1))

    right_hand_dist_rew = torch.exp(-10 * right_hand_finger_dist)
    left_hand_dist_rew = torch.exp(-10 * left_hand_finger_dist)

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    up_rew = torch.zeros_like(right_hand_dist_rew)
    up_rew = torch.where(right_hand_finger_dist < 0.75,
                    torch.where(left_hand_finger_dist < 0.75,
                        torch.norm(pen_right_handle_pos - pen_left_handle_pos, p=2, dim=-1) * 5 - 0.8,
                        ↪  up_rew), up_rew)

    reward = up_rew + right_hand_dist_rew + left_hand_dist_rew

    resets = torch.where(right_hand_dist_rew <= 0, torch.ones_like(reset_buf), reset_buf)
    resets = torch.where(right_hand_finger_dist >= 1.5, torch.ones_like(resets), resets)
    resets = torch.where(left_hand_finger_dist >= 1.5, torch.ones_like(resets), resets)

    # Find out which envs hit the goal and update successes count
    successes = torch.where(successes == 0,
                    torch.where(torch.norm(pen_right_handle_pos - pen_left_handle_pos, p=2, dim=-1) * 5 >
                    ↪  1.5, torch.ones_like(successes), successes), successes)

    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    goal_resets = torch.zeros_like(resets)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes)
    goal_reach = torch.where(torch.norm(pen_right_handle_pos - pen_left_handle_pos, p=2, dim=-1) * 5 >=
    ↪  1.5,
                            torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandPushBlock***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, left_target_pos, left_target_rot,
    ↪  right_target_pos, right_target_rot, block_right_handle_pos, block_left_handle_pos,
    left_hand_pos, right_hand_pos, right_hand_ff_pos, right_hand_mf_pos, right_hand_rf_pos,
    ↪  right_hand_lf_pos, right_hand_th_pos,
    left_hand_ff_pos, left_hand_mf_pos, left_hand_rf_pos, left_hand_lf_pos, left_hand_th_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    left_goal_dist = torch.norm(left_target_pos - block_left_handle_pos, p=2, dim=-1)
    right_goal_dist = torch.norm(right_target_pos - block_right_handle_pos, p=2, dim=-1)

    right_hand_finger_dist = (torch.norm(block_right_handle_pos - right_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(block_right_handle_pos - right_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(block_right_handle_pos - right_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(block_right_handle_pos - right_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(block_right_handle_pos - right_hand_th_pos, p=2, dim=-1))
    left_hand_finger_dist = (torch.norm(block_left_handle_pos - left_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(block_left_handle_pos - left_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(block_left_handle_pos - left_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(block_left_handle_pos - left_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(block_left_handle_pos - left_hand_th_pos, p=2, dim=-1))

    right_hand_dist_rew = 1.2-1*right_hand_finger_dist
    left_hand_dist_rew = 1.2-1*left_hand_finger_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    up_rew = torch.zeros_like(right_hand_dist_rew)
    up_rew = 5 - 5*left_goal_dist - 5*right_goal_dist

    reward = right_hand_dist_rew + left_hand_dist_rew + up_rew

    resets = torch.where(right_hand_finger_dist >= 1.2, torch.ones_like(reset_buf), reset_buf)
    resets = torch.where(left_hand_finger_dist >= 1.2, torch.ones_like(resets), resets)

    # Find out which envs hit the goal and update successes count
    successes = torch.where(successes == 0,
                    torch.where(torch.abs(left_goal_dist) <= 0.1,
                        torch.where(torch.abs(right_goal_dist) <= 0.1, torch.ones_like(successes),
                        ↪  torch.ones_like(successes) * 0.5), successes), successes)

    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    goal_resets = torch.zeros_like(resets)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes).mean()
    goal_reach = 0.5 * (torch.where(torch.abs(left_goal_dist) <= 0.1,
                                torch.ones_like(successes), torch.zeros_like(successes)) \
                        + torch.where(torch.abs(right_goal_dist) <= 0.1,
                                  torch.ones_like(successes), torch.zeros_like(successes)))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandReOrientation***

```python
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, object_another_pos,
    ↪  object_another_rot, target_another_pos, target_another_rot,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    goal_dist = torch.norm(target_pos - object_pos, p=2, dim=-1)
    if ignore_z_rot:
        success_tolerance = 2.0 * success_tolerance

    goal_another_dist = torch.norm(target_another_pos - object_another_pos, p=2, dim=-1)
    if ignore_z_rot:
        success_tolerance = 2.0 * success_tolerance

    # Orientation alignment for the cube in hand and goal cube
    quat_diff = quat_mul(object_rot, quat_conjugate(target_rot))
    rot_dist = 2.0 * torch.asin(torch.clamp(torch.norm(quat_diff[:, 0:3], p=2, dim=-1), max=1.0))

    quat_another_diff = quat_mul(object_another_rot, quat_conjugate(target_another_rot))
    rot_another_dist = 2.0 * torch.asin(torch.clamp(torch.norm(quat_another_diff[:, 0:3], p=2, dim=-1),
    ↪  max=1.0))

    dist_rew = goal_dist * dist_reward_scale + goal_another_dist * dist_reward_scale
    rot_rew = 1.0/(torch.abs(rot_dist) + rot_eps) * rot_reward_scale + 1.0/(torch.abs(rot_another_dist) +
    ↪  rot_eps) * rot_reward_scale

    action_penalty = torch.sum(actions ** 2, dim=-1)

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    reward = dist_rew + rot_rew + action_penalty * action_penalty_scale

    # Find out which envs hit the goal and update successes count
    goal_resets = torch.where(torch.abs(rot_dist) < 0.1, torch.ones_like(reset_goal_buf), reset_goal_buf)
    goal_resets = torch.where(torch.abs(rot_another_dist) < 0.1, torch.ones_like(reset_goal_buf),
    ↪  reset_goal_buf)

    successes = successes + goal_resets

    # Success bonus: orientation is within `success_tolerance` of goal orientation
    reward = torch.where(goal_resets == 1, reward + reach_goal_bonus, reward)

    # Fall penalty: distance to the goal is larger than a threashold
    reward = torch.where(object_pos[:, 2] <= 0.2, reward + fall_penalty, reward)
    reward = torch.where(object_another_pos[:, 2] <= 0.2, reward + fall_penalty, reward)

    # Check env termination conditions, including maximum success number
    resets = torch.where(object_pos[:, 2] <= 0.2, torch.ones_like(reset_buf), reset_buf)
    resets = torch.where(object_another_pos[:, 2] <= 0.2, torch.ones_like(reset_buf), resets)

    if max_consecutive_successes > 0:
        # Reset progress buffer on goal envs if max_consecutive_successes > 0
        progress_buf = torch.where(torch.abs(rot_dist) <= success_tolerance,
        ↪  torch.zeros_like(progress_buf), progress_buf)
        resets = torch.where(successes >= max_consecutive_successes, torch.ones_like(resets), resets)
    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    # Apply penalty for not reaching the goal
    if max_consecutive_successes > 0:
        reward = torch.where(progress_buf >= max_episode_length, reward + 0.5 * fall_penalty, reward)

    num_resets = torch.sum(resets)
    finished_cons_successes = torch.sum(successes * resets.float())

    cons_successes = torch.where(num_resets > 0, av_factor*finished_cons_successes/num_resets + (1.0 -
    ↪  av_factor)*consecutive_successes, consecutive_successes)
    goal_reach = 0.5 * (torch.where(torch.abs(rot_dist) <= 0.1, torch.ones_like(reset_goal_buf),
    ↪  torch.zeros_like(reset_goal_buf)) \
            + torch.where(torch.abs(rot_another_dist) <= 0.1, torch.ones_like(reset_goal_buf),
            ↪  torch.zeros_like(reset_goal_buf)))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandScissors***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, scissors_right_handle_pos,
    ↪  scissors_left_handle_pos, object_dof_pos,
    left_hand_pos, right_hand_pos, right_hand_ff_pos, right_hand_mf_pos, right_hand_rf_pos,
    ↪  right_hand_lf_pos, right_hand_th_pos,
    left_hand_ff_pos, left_hand_mf_pos, left_hand_rf_pos, left_hand_lf_pos, left_hand_th_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    right_hand_finger_dist = (torch.norm(scissors_right_handle_pos - right_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(scissors_right_handle_pos - right_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(scissors_right_handle_pos - right_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(scissors_right_handle_pos - right_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(scissors_right_handle_pos - right_hand_th_pos, p=2, dim=-1))
    left_hand_finger_dist = (torch.norm(scissors_left_handle_pos - left_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(scissors_left_handle_pos - left_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(scissors_left_handle_pos - left_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(scissors_left_handle_pos - left_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(scissors_left_handle_pos - left_hand_th_pos, p=2, dim=-1))

    right_hand_dist_rew = right_hand_finger_dist
    left_hand_dist_rew = left_hand_finger_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    up_rew = torch.zeros_like(right_hand_dist_rew)
    up_rew = torch.where(right_hand_finger_dist < 0.7,
                  torch.where(left_hand_finger_dist < 0.7,
                      (0.59 + object_dof_pos[:, 0]) * 5, up_rew), up_rew)

    reward = 2 + up_rew - right_hand_dist_rew - left_hand_dist_rew

    resets = torch.where(up_rew < -0.5, torch.ones_like(reset_buf), reset_buf)
    resets = torch.where(right_hand_finger_dist >= 1.75, torch.ones_like(resets), resets)
    resets = torch.where(left_hand_finger_dist >= 1.75, torch.ones_like(resets), resets)

    # Find out which envs hit the goal and update successes count
    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    successes = torch.where(successes == 0,
                  torch.where(object_dof_pos[:, 0] > -0.3, torch.ones_like(successes), successes),
                  ↪  successes)

    goal_resets = torch.zeros_like(resets)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes).mean()
    goal_reach = torch.where(object_dof_pos[:, 0] >= -0.3,
                         torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandSwingCup***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, cup_right_handle_pos,
    ↪  cup_left_handle_pos,
    left_hand_pos, right_hand_pos, right_hand_ff_pos, right_hand_mf_pos, right_hand_rf_pos,
    ↪  right_hand_lf_pos, right_hand_th_pos,
    left_hand_ff_pos, left_hand_mf_pos, left_hand_rf_pos, left_hand_lf_pos, left_hand_th_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    right_hand_finger_dist = (torch.norm(cup_right_handle_pos - right_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(cup_right_handle_pos - right_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(cup_right_handle_pos - right_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(cup_right_handle_pos - right_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(cup_right_handle_pos - right_hand_th_pos, p=2, dim=-1))
    left_hand_finger_dist = (torch.norm(cup_left_handle_pos - left_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(cup_left_handle_pos - left_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(cup_left_handle_pos - left_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(cup_left_handle_pos - left_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(cup_left_handle_pos - left_hand_th_pos, p=2, dim=-1))

    # Orientation alignment for the cube in hand and goal cube
    quat_diff = quat_mul(object_rot, quat_conjugate(target_rot))
    rot_dist = 2.0 * torch.asin(torch.clamp(torch.norm(quat_diff[:, 0:3], p=2, dim=-1), max=1.0))

    right_hand_dist_rew = right_hand_finger_dist
    left_hand_dist_rew = left_hand_finger_dist

    rot_rew = 1.0/(torch.abs(rot_dist) + rot_eps) * rot_reward_scale - 1

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    up_rew = torch.zeros_like(rot_rew)
    up_rew = torch.where(right_hand_finger_dist < 0.4,
                    torch.where(left_hand_finger_dist < 0.4,
                                    rot_rew, up_rew), up_rew)

    reward = - right_hand_dist_rew - left_hand_dist_rew + up_rew

    resets = torch.where(object_pos[:, 2] <= 0.3, torch.ones_like(reset_buf), reset_buf)

    # Find out which envs hit the goal and update successes count
    successes = torch.where(successes == 0,
                    torch.where(rot_dist < 0.785, torch.ones_like(successes), successes), successes)


    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    goal_resets = torch.zeros_like(resets)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes).mean()
    goal_reach = torch.where(rot_dist <= 0.785, torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

Bi-DexHands: ***ShadowHandSwitch***

```
def compute_hand_reward(
    rew_buf, reset_buf, reset_goal_buf, progress_buf, successes, consecutive_successes,
    max_episode_length: float, object_pos, object_rot, target_pos, target_rot, switch_right_handle_pos,
    ↪  switch_left_handle_pos,
    left_hand_pos, right_hand_pos, right_hand_ff_pos, right_hand_mf_pos, right_hand_rf_pos,
    ↪  right_hand_lf_pos, right_hand_th_pos,
    left_hand_ff_pos, left_hand_mf_pos, left_hand_rf_pos, left_hand_lf_pos, left_hand_th_pos,
    dist_reward_scale: float, rot_reward_scale: float, rot_eps: float,
    actions, action_penalty_scale: float,
    success_tolerance: float, reach_goal_bonus: float, fall_dist: float,
    fall_penalty: float, max_consecutive_successes: int, av_factor: float, ignore_z_rot: bool
):
    # Distance from the hand to the object
    right_hand_finger_dist = (torch.norm(switch_right_handle_pos - right_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(switch_right_handle_pos - right_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(switch_right_handle_pos - right_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(switch_right_handle_pos - right_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(switch_right_handle_pos - right_hand_th_pos, p=2, dim=-1))
    left_hand_finger_dist = (torch.norm(switch_left_handle_pos - left_hand_ff_pos, p=2, dim=-1) +
    ↪  torch.norm(switch_left_handle_pos - left_hand_mf_pos, p=2, dim=-1)
                            + torch.norm(switch_left_handle_pos - left_hand_rf_pos, p=2, dim=-1) +
                            ↪  torch.norm(switch_left_handle_pos - left_hand_lf_pos, p=2, dim=-1)
                            + torch.norm(switch_left_handle_pos - left_hand_th_pos, p=2, dim=-1))

    right_hand_dist_rew = right_hand_finger_dist
    left_hand_dist_rew = left_hand_finger_dist

    # Total reward is: position distance + orientation alignment + action regularization + success bonus
    ↪  + fall penalty
    up_rew = torch.zeros_like(right_hand_dist_rew)
    up_rew = (1.4-(switch_right_handle_pos[:, 2] + switch_left_handle_pos[:, 2])) * 50

    reward = 2 - right_hand_dist_rew - left_hand_dist_rew + up_rew

    resets = torch.where(right_hand_dist_rew <= 0, torch.ones_like(reset_buf), reset_buf)

    # Find out which envs hit the goal and update successes count
    successes = torch.where(successes == 0,
                    torch.where(1.4-(switch_right_handle_pos[:, 2] + switch_left_handle_pos[:, 2]) >
                    ↪  0.05, torch.ones_like(successes), successes), successes)

    resets = torch.where(progress_buf >= max_episode_length, torch.ones_like(resets), resets)

    goal_resets = torch.zeros_like(resets)

    cons_successes = torch.where(resets > 0, successes * resets, consecutive_successes).mean()
    goal_reach = torch.where(1.4 - (switch_right_handle_pos[:, 2] + switch_left_handle_pos[:, 2]) >=
    ↪  0.05,
                             torch.ones_like(successes), torch.zeros_like(successes))
    heuristic_reward, task_reward = reward, goal_reach     return heuristic_reward, task_reward, resets,
    ↪  goal_resets, progress_buf, successes, cons_successes
```

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: This paper aims to explore alternatives for improving task performance in finite data settings using heuristic signals. Our experiments on robotic locomotion, helicopter, and manipulation tasks demonstrate that this method consistently improves performance, regardless of the general effectiveness of the heuristic signals. We are confident that our abstract and introduction sections accurately reflect the paper's contributions and scope.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The limitations of our approach are illustrated in Section 6.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory Assumptions and Proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [Yes]

Justification: We have shown our derivation details and limitations in Section A.1 and Section 6.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental Result Reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: We have provided detailed derivation and implementation descriptions (including the simulation environments, hyperparameters, and reward definitions for training and evaluations) in our Appendix section. Additionally, we have also provided our source code in our Supplementary Materials.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We have provided detailed implementation descriptions (including the simulation environments, hyperparameters, and reward definitions for training and evaluations) in our Appendix section. Additionally, we have also provided our source code in our Supplementary Materials. The adopted simulation environments are all well-known and available online.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so “No” is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental Setting/Details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: We have provided detailed implementation descriptions (including the simulation environments, hyperparameters, and reward definitions for training and evaluations) in the experiment and Appendix sections. Additionally, we have also provided our source code in our Supplementary Materials, including all the training and environment configurations.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment Statistical Significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: All of our experimental results were obtained by 5 random seeds. We have provided the mean and standard deviation for our experimental results.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments Compute Resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [No]

Justification: But each training procedure can be performed on a single GeForce RTX 2080 Ti device. The required computational resources for all the simulation benchmarks are listed on their respective websites.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code Of Ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

Answer: [Yes]

Justification: We have carefully examined the ethical guidelines and verified that our work fully adheres to all the principles and requirements.

Guidelines:

- The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.

- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: The paper does not discuss both potential positive societal impacts and negative societal impacts of the work performed.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: This paper has no such risks.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

Justification: Throughout this paper, we have provided proper citations and references for all utilized repositories, benchmark simulations, and models/algorithms to uphold transparency and ensure appropriate attribution.

Guidelines:

- The answer NA means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New Assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: We will furnish comprehensive documentation for our released code, elucidating its usage and providing information about the original source. Additionally, we have ensured that any code modified from external sources is subject to licenses that permit modification and redistribution.

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and Research with Human Subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Justification: No, but we engaged 12 participants in devising reward functions as part of the experiments detailed in Section 4.2.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

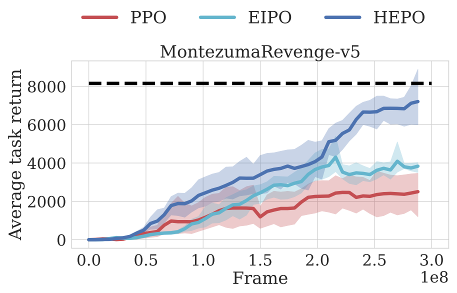
    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

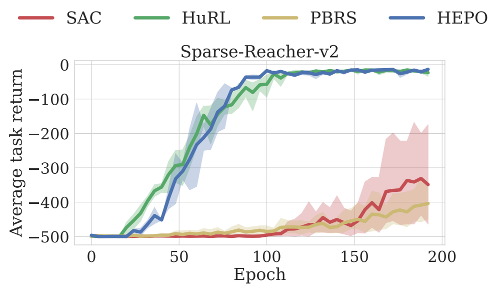
    Justification: This paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

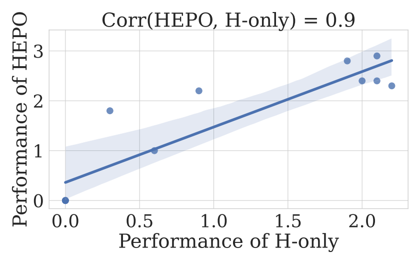
    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.



(a) EIPO v.s. HEPO on PPO

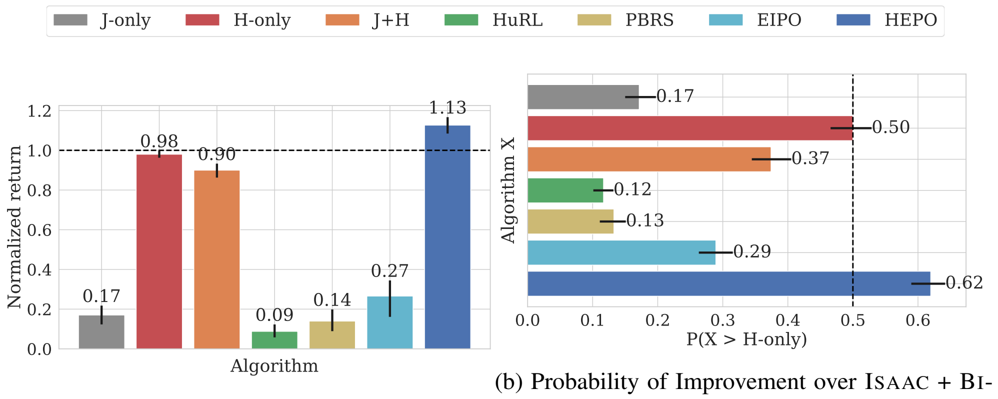


(b) HuRL v.s. HEPO on SAC



(c) Correlation between the performance of HEPO and H-only.

Figure 7: **(a)** Comparison of HEPO and EIPO [6] on the most challenging Atari task, `Montezuma's Revenge`, shown in the EIPO paper [6]. Both are implemented on top of EIPO's PPO codebase using RND exploration bonuses [16] as heuristic rewards H, as suggested in [6]. HEPO outperforms EIPO, achieving the performance (denoted as dashed line) similar to PPO trained with RND at convergence (2 billion frames) reported in [16] in five times fewer frames. **(b)** HEPO matches HuRL's performance on the most challenging `Sparse-Reacher` task using HuRL's SAC codebase [7], despite HuRL being tuned for this task and HEPO using the same hyperparameters from our Section 4. This also highlights HEPO's generality in different RL algorithms. **(c)** HEPO's performance is positively correlated with that of the heuristic policy trained with heuristic rewards only (H-only), suggesting that HEPO's effectiveness will improve as the quality of heuristic rewards increases.



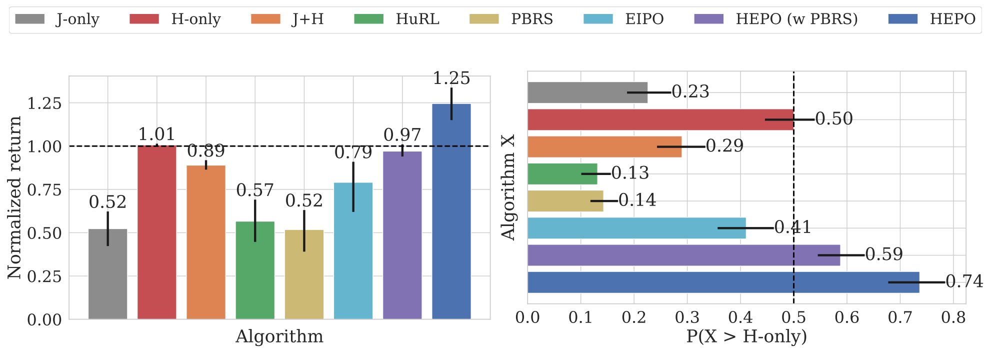
(a) IQM over Isaac + Bi-Dex

(b) Probability of Improvement over Isaac + Bi-Dex

Algorithm X	IQM (95% CI)	P(X > H-only) (95% CI)	Min	Max
H-only	0.42 (0.31, 0.54)	0.50 (0.49, 0.51)	0	2.43
HuRL	0.00 (0.00, 0.00)	0.33 (0.32, 0.33)	0	2.32
PBRS	0.00 (0.00, 0.00)	0.22 (0.21, 0.22)	0	0.23
HEPO	0.98 (0.73, 1.23)	0.64 (0.62, 0.65)	0	3.02

(c) Comparison of additional baselines on heuristic reward functions designed in the wild (Section 4.2)

Figure 8: **(a) & (b)** Our HEPO outperforms EIPO in both IQM of normalized task return and probability of improvements across 29 tasks on `Isaac` [12] + `Bi-Dex` [17]. We implemented EIPO as described in [6], alternating between two policies and training the reference policy with task rewards only. See Section 4.3 for details. **(c)** Evaluated on reward function designed in the wild (Section 4.2), HEPO shows higher IQM of normalized return and probability of improvement compared to policies trained with only heuristic rewards (H-only), HuRL [7], and PBRS [5].



(a) IQM over Isaac

(b) Probability of Improvement over Isaac

Figure 9: **(a) & (b)** Following the setup in Section 4.1, we trained HEPO with PBRS-shaped heuristic rewards (denoted as HEPO (w/ PBRS)) on 9 `Isaac` tasks [12]. PBRS ensures that optimizing these rewards leads to the same optimal policy as optimizing the original task rewards. Despite poor performance when training policies directly with PBRS-shaped rewards, HEPO (w/ PBRS) outperforms most baselines and matches H-only in terms of normalized return IQM. It also shows significant improvement probability over H-only (i.e., lower confidence bound on the probability of improvement is greater than 0.5). These results suggest PBRS and HEPO can be combined to achieve both theoretical guarantees and better empirical performance.