DexDiffuser: Interaction-aware Diffusion Planning for Adaptive Dexterous Manipulation

Zhixuan Liang[1,2] Yao Mu[1,3] Yixiao Wang[2] Fei Ni[4] Tianxing Chen[3]
Wenqi Shao[3] Wei Zhan[2] Masayoshi Tomizuka[2†] Ping Luo[1,3†] Mingyu Ding[2]

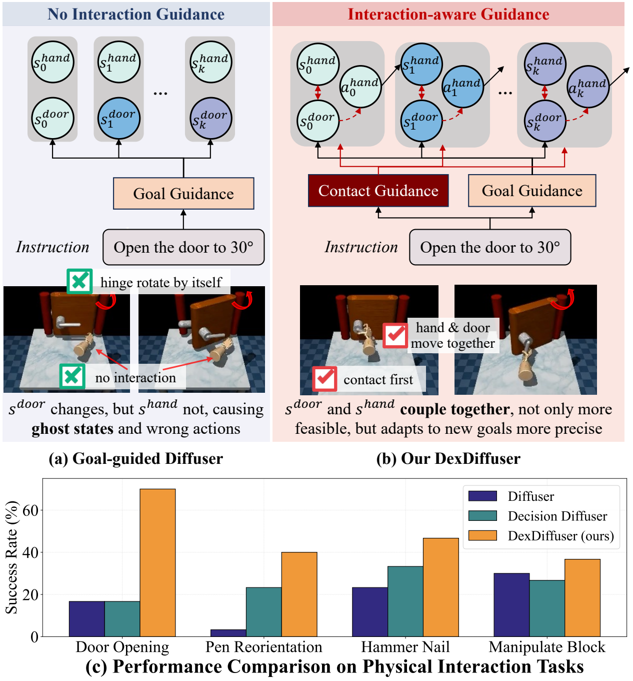
[1]The University of Hong Kong [2]University of California, Berkeley
[3]Shanghai AI Laboratory [4]Tianjin University

{zxliang, ymu, pluo}@cs.hku.hk {yixiao_wang, wzhan, tomizuka, myding}@berkeley.edu

https://dexdiffuser.github.io/

Abstract

Dexterous manipulation with contact-rich interactions is crucial for advanced robotics. While recent diffusion-based planning approaches show promise for simpler manipulation tasks, they often produce unrealistic ghost states (e.g., the object automatically moves without hand contact) or lack adaptability when handling complex sequential interactions. In this work, we introduce DexDiffuser, an interaction-aware diffusion planning framework for adaptive dexterous manipulation. DexDiffuser models joint state-action dynamics through a dual-phase diffusion process which consists of pre-interaction contact alignment and post-contact goal-directed control, enabling goal-adaptive generalizable dexterous manipulation. Additionally, we incorporate dynamics model-based dual guidance and leverage large language models for automated guidance function generation, enhancing generalizability for physical interactions and facilitating diverse goal adaptation through language cues. Experiments on physical interaction tasks such as door opening, pen and block re-orientation, and hammer striking demonstrate DexDiffuser's effectiveness on goals outside training distributions, achieving over twice the average success rate (59.2% vs. 29.5%) compared to existing methods. Our framework achieves 70.0% success on 30-degree door opening, 40.0% and 36.7% on pen and block half-side re-orientation respectively, and 46.7% on hammer nail half drive, highlighting its robustness and flexibility in contact-rich manipulation.



(a) Goal-guided Diffuser **(b) Our DexDiffuser**

(c) Performance Comparison on Physical Interaction Tasks

Figure 1. (a) Previous diffusers directly apply goal guidance to object states, which leads to ghost states where objects move independently leaving hand states unchanged. (b) DexDiffuser introduces contact guidance that jointly influences both hand/object states and hand actions, while maintaining tight state-action coupling. It not only prevents ghost states, but also enables precise goal adaptation through coordinated hand-object motion. (c) Quantitative comparisons with previous methods on goal-adapted interaction tasks.

1. Introduction

Dexterous manipulation, a cornerstone of advanced robotics with applications from service robotics to industrial automation, remains a challenging problem despite advances in reinforcement learning (RL) [2, 4, 7, 44, 46] and imitation learning [22, 31]. Recently, diffusion-based planning [1, 13, 23, 27] has emerged as a promising new representative of imitation learning, capable of learning intricate motion trajectories from demonstration data for smoother

[†]Corresponding authors.

and more adaptable control. However, current diffusion approaches are primarily designed for simpler gripper-based manipulation tasks, focusing on either trajectory completion or action replay by reaching target positions sequentially. They fall short in capturing the staged and contact-rich interactions required for more sophisticated tasks, such as door opening and tool handling, which involve dexterous multi-fingered robotic hands.

Current diffusion-based planning frameworks can be generally divided into two streams based on whether they generate actions or states. Action-based diffusion models [13, 50] excel in well-defined tasks but often lack generalizability in adapting to complex or new tasks with flexible interaction requirements, necessitating continual data collection for new goal configurations even within the same dynamics. This limits their effectiveness in contact-rich interactions. In contrast, state-based diffusion methods [1, 23, 34], including those adapted from video diffusion models for imitation learning [6, 15], tend to produce unrealistic "ghost states". In these cases, objects appear to react independently of physical contact, such as drawers opening on their own before the manipulator reaches them or objects rotating mid-air without direct interaction, as shown in Fig. 1 and Fig. 2. This issue arises because a manipulator's actions must first influence its intermediate states before impacting an object, revealing the importance of modeling state transitions with realistic physics-driven interactions. Addressing these limitations in contact-rich dexterous manipulation requires a model that is both interaction-aware and adaptive to task constraints, while remaining grounded in realistic physical behavior.

In this work, we propose DexDiffuser, an interaction-aware diffusion planning framework tailored for adaptive dexterous manipulation. DexDiffuser models joint state-action dynamics that takes the state output to guide and constrain the action output with explicit physical dynamics. A dynamics model-based dual guide is incorporated to maintain coherence with dynamical patterns observed in training data, addressing the action-state consistency challenge first identified in Diffuser [23], which however prioritized state diffusion over action diffusion, as compared in Fig. 1. Furthermore, to automate guidance function design, DexDiffuser introduces an approach using large language models (LLMs) in a text-to-reward paradigm. Together, these designs allow DexDiffuser to generalize across diverse goals and adapt to novel configurations or even task reversals via language cues in a classifier-guided structure.

Specifically, DexDiffuser introduces a goal-adaptive diffusion mechanism designed to handle complex, multi-contact interactions through a dual-phase process that diffuses across state and action spaces. 1) In the first, pre-contact phase, it guides the manipulator to align with the object's key interaction points, such as a handle or center of mass, ensuring stable alignment before initiating physical interaction. 2) In the subsequent post-contact phase, it introduces joint guidance over both the manipulator and the object states, enabling fine-grained control to achieve the target state for the object. This sequential approach integrates both action diffusion, preventing premature influence on the object's state before contact, and state diffusion, ensuring effective goal alignment throughout. By generating state and action in an interaction-aware manner, DexDiffuser produces more coherent and realistic trajectories suited to complex tasks like tool manipulations.

To evaluate DexDiffuser's effectiveness, we conducted experiments on dexterous manipulation tasks, covering both in-domain and goal-adaptability challenges, *e.g.*, adapting to new goal "door closing" from "90-degree door opening" training data. Results with up to 70.0% success rate on the 30-degree door task (vs. the next best 16.7% for Diffusion Policy) and 46.7% on the hammer nail half-drive task (vs. the next best 33.3% for Decision Diffuser), confirm DexDiffuser's robustness and adaptability in capturing complex hand-object-environment interactions.

In summary, DexDiffuser advances adaptive dexterous manipulation by: 1) We propose the first interaction-aware, goal-adaptive diffusion planner for dexterous manipulation, modeling manipulator-object-environment dependencies to handle sequential tasks with complex state transitions. 2) By jointly modeling state-action behaviors with dynamics-based dual guidance and LLM-based interaction guidance, DexDiffuser sets a new standard for adaptive planning in dexterous manipulation and for the first time extends text-to-reward concepts to diffusers. 3) Experimental validation on diverse dexterous manipulation tasks, demonstrating its robustness and adaptability. DexDiffuser achieves over twice the average success rate of the next best method (59.2% vs. 29.5%) across goal-directed tasks.

2. Related Works

Dexterous Manipulation. Dexterous manipulation [10–12, 18, 19, 29, 37, 39, 42] with multi-fingered hands enables complex tasks in unstructured environments by mimicking human hand flexibility. Initially, traditional methods using trajectory optimization and precise dynamics models [33, 38], struggled with high-dimensional action spaces and contact-rich dynamics. This led to the adoption of reinforcement learning (RL) [8, 38, 46, 52] for handling complex, high-DOF interactions. However, RL requires extensive online exploration and carefully designed reward functions [9, 33] where inadequate reward shaping can significantly slow down learning and limit adaptability [49, 51]. While demonstration-based methods [51] reduce sample complexity, they struggle to generalize across sequential, contact-rich tasks. DexDiffuser addresses these challenges by explicitly modeling hand-object-environment interactions, enabling goal-adaptive planning without intricate reward shaping, thus allowing for more efficient learning in complex, sequential dexterous manipulation tasks.

Diffusion-based Planning Methods. Planning with diffusion models has become prominent in imitation learning for robotic manipulation [13, 23, 27, 28, 34]. Initially, classifier-guided methods [23, 27] used task-specific classifiers to condition policies through reward gradients. Simultaneously, classifier-free diffusion emerged, integrating task variations within the model without external classifiers [1, 14]. While efficient, classifier-free methods lack flexibility for zero-shot explicit conditioning tasks due to reliance on training data configurations.

DexDiffuser addresses this by combining classifier-guided diffusion over both state and action spaces, enabling precise, interaction-aware planning that adapts dynamically to the evolving states of both the manipulator and object for more realistic and adaptable manipulation.

LLM-based Robotics Policy Code Generation. Recent works have demonstrated the potential of LLMs in generating executable code for robotics tasks. Code as Policies [26] and RoboCodeX [32] showed that LLMs can effectively translate high-level task descriptions into functional robot control programs. In reinforcement learning, Eureka [30] pioneered the use of LLMs to determine crucial algorithm parameters and architectures. Text2Reward [48] further advanced this direction by directly generating complete reward functions from natural language descriptions, demonstrating well-structured prompts with comprehensive environment information can enable reliable reward function generation. Our work extends this text-to-code paradigm to imitation learning through diffusion-based planner. DexDiffuser provides a natural interface for LLM-generated guidance functions through its explicit energy function formulation, bridging the gap between natural language task specification and learned behavioral policies.

3. Preliminary

3.1. Diffusion Model as Policy

We formulate the dexterous manipulation planning problem within the Markov Decision Process (MDP) framework [36], defined as $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$. The objective is to find an optimal action sequence $\boldsymbol{a}^*_{0:T}$ that satisfies:

$$\boldsymbol{a}^*_{0:T} = \arg\max_{\boldsymbol{a}_{0:T}} \mathcal{J}(\boldsymbol{s}_0, \boldsymbol{a}_{0:T}) = \arg\max_{\boldsymbol{a}_{0:T}} \sum_{t=0}^{T} \gamma^t R(\boldsymbol{s}_t, \boldsymbol{a}_t),$$

where state transitions follow $\boldsymbol{s}_{t+1} = \mathcal{T}(\boldsymbol{s}_t, \boldsymbol{a}_t)$. (1)

Following [1, 23], we leverage diffusion models to address this planning problem by treating state or action trajectories $\boldsymbol{\tau}$ as sequential data. The reverse process of diffusion learns to denoise trajectories from a standard normal distribution through conditional probability $p_\theta(\boldsymbol{\tau}^{i-1} \mid \boldsymbol{\tau}^i)$. The model is trained to maximize the likelihood:

$$p_\theta\left(\boldsymbol{\tau}^0\right) = \int p\left(\boldsymbol{\tau}^N\right) \prod_{i=1}^{N} p_\theta\left(\boldsymbol{\tau}^{i-1} \mid \boldsymbol{\tau}^i\right) \mathrm{d}\boldsymbol{\tau}^{1:N}, \tag{2}$$

with the optimization objective inspired by ELBO,

$$\theta^* = \arg\min_\theta -\mathbb{E}_{\boldsymbol{\tau}^0}\left[\log p_\theta\left(\boldsymbol{\tau}^0\right)\right], \tag{3}$$

For practical implementation, we adopt the simplified surrogate loss [21] that focuses on predicting the noise term:

$$\mathcal{L}_{\text{denoise}}(\theta) = \mathbb{E}_{i,\boldsymbol{\tau}^0\sim q,\epsilon\sim\mathcal{N}}[||\epsilon - \epsilon_\theta(\boldsymbol{\tau}^i, i)||^2]. \tag{4}$$

3.2. Classifier-free Conditional Policy

To generate high-reward trajectories, classifier-free guidance [14] has been transferred from image to trajectory generation [1]. This approach incorporates guidance signals $\boldsymbol{y}(\boldsymbol{\tau})$ directly in the noise prediction model by:

$$\hat{\epsilon} = \epsilon_\theta(\boldsymbol{\tau}^i, \varnothing, i) + \omega(\epsilon_\theta(\boldsymbol{\tau}^i, \boldsymbol{y}, i) - \epsilon_\theta(\boldsymbol{\tau}^i, \varnothing, i)), \tag{5}$$

where ω controls the guidance strength, and $\varnothing$ denotes the absence of conditioning. During sampling, trajectories are generated with the modified noise $\hat{\epsilon}$, employing reparameterization technique.

3.3. Classifier-guided Conditional Policy

While classifier-free diffusion offers a streamlined approach, its conditioning flexibility relies solely on implicit representations within the training data. Classifier-guided approach, in contrast, enables direct reward or goal conditioning through gradient-based guidance.

For reward maximization, it introduces trajectory optimality $\mathcal{O}_t$ at timestep t, following a Bernoulli distribution where $p(\mathcal{O}_t = 1) = \exp(\gamma^t \mathcal{R}(\boldsymbol{s}_t, \boldsymbol{a}_t))$. The diffusion process can be naturally extended to incorporate conditioning by sampling from perturbed distributions:

$$\tilde{p}_\theta(\boldsymbol{\tau}) = p(\boldsymbol{\tau} \mid \mathcal{O}_{1:T} = 1) \propto p_\theta(\boldsymbol{\tau}) p(\mathcal{O}_{1:T} = 1 \mid \boldsymbol{\tau}) \tag{6}$$

Under Lipschitz conditions on $p(\mathcal{O}_{1:T} \mid \boldsymbol{\tau}^i)$ [16], the reverse diffusion process follows:

$$p_\theta(\boldsymbol{\tau}^{i-1} \mid \boldsymbol{\tau}^i, \mathcal{O}_{1:T}) \approx \mathcal{N}(\boldsymbol{\tau}^{i-1}; \mu_\theta + \alpha\Sigma g, \Sigma), \tag{7}$$

where the guidance gradient g is:

$$\begin{aligned} g &= \nabla_{\boldsymbol{\tau}} \log p(\mathcal{O}_{1:T} \mid \boldsymbol{\tau})|_{\boldsymbol{\tau}=\mu_\theta} \\ &= \sum_{t=0}^{T} \gamma^t \nabla_{\boldsymbol{s}_t, \boldsymbol{a}_t} \mathcal{R}(\boldsymbol{s}_t, \boldsymbol{a}_t)|_{(\boldsymbol{s}_t, \boldsymbol{a}_t)=\mu_t} = \nabla_{\boldsymbol{\tau}} \mathcal{J}(\mu_\theta). \end{aligned} \tag{8}$$

For discrete goal conditioned tasks, the constraint can be simplified by directly substituting conditional values at each diffusion timestep $i \in \{0, 1, ..., N\}$.

4. Analysis of Diffusion-based Planning Methods for Interaction-intensive Tasks

Current diffusion-based methods are widely adopted for robotic manipulation but reveal significant limitations when applied to dexterous, sequential interaction tasks. Table 1 provides an overview of prominent diffusion-based methods (including Diffuser [23], Decision Diffuser [1], Diffusion Policy [13] and ours DexDiffuser), categorizing each by their conditioning approach, action generation method, and goal adaptability. In this section, we analyze these challenges across three key dimensions.

Limitations of Action-only Diffusion in Explicit State Conditioning. Existing diffusion planners, especially action only models like Diffusion Policy [13], excel in providing precise, consistent action control, benefiting from extensive training data. Action diffusion ensures stable action precision despite variations in arm dynamics, and bypasses errors from inverse kinematics. This yields high performance when training data is sufficient and diverse. However, for tasks requiring multi-stage adaptive guidance, action-only diffusion lacks the flexibility needed for explicit state guidance at intermediate stages, like aligning hand and object at pre-grasp and transitioning accurately to post-grasp states. For example, Diffusion Policy [13] trained on data for opening a door to 90 degrees cannot adapt well to opening 30 or 60 degrees.

Ghost States in State-only Diffusion for Sequential Interaction. While state-based diffusion offers the advantage of flexible goal specification, it is most effective in environments where all degrees of freedom are directly controllable. This is suitable for fully actuated tasks, such as MuJoCo Half-Cheetah, Hopper, and Walker [23, 43], and straightforward pick-and-place tasks with manipulators like KUKA or Franka [1, 13] where control is limited to positioning the end-effector at specific points. In such scenarios, the system's complete state can be manipulated directly.

However, in dexterous manipulation tasks that require indirect control—such as striking a nail with a hammer using a dexterous hand—additional uncontrolled degrees of freedom, like the hammer head and nail positions, must be influenced through intermediary states of the hand. In these cases, applying state-only diffusion across all joints, including those of objects beyond the hand, can result in unrealistic "ghost states". This phenomenon, where objects appear to move independently of contact as illustrated in Fig. 1 and Fig. 2, disrupts the realism required for interaction tasks that depend on adaptive, contact-based control adjustments.

Classifier-free vs. Classifier-guided Adaptability. Classifier free diffusion models, valued for bypassing the need for external classifiers, encode task variations directly within the model. This structure is effective for tasks constrained within observed configurations, but limits goal adaptability in zero-shot or new-task scenarios, where goals and conditions differ from training data. For instance, Diffusion Policy [13], in the push-T task, cannot directly modify the target position of the block due to the fixed goal position in training data—a limitation similar to our door experiments, where training data includes only a 90° target angle. In contrast, classifier-guided methods, such as ours, mitigate this limitation by offering adaptable, gradient-based guidance, enabling direct conditioning on new goals or rewards, enhancing flexibility across a range of interactive tasks.

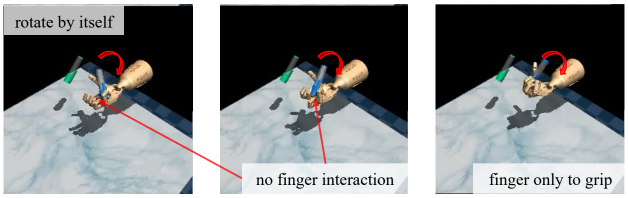


Figure 2. **Demonstration of ghost states on pen reorientation.** The pen autonomously rotates to the desired orientation without any hand manipulation, and finally, the fingers move to grip the pen in the target state.

5. Method

5.1. Interaction-aware Diffusion-based Planning

To address these limitations, we propose DexDiffuser, an interaction-aware diffusion planning framework (Fig. 3), maintaining physical consistency and enabling flexible goal adaptation for dexterous manipulation.

Joint State-Action Diffusion Model. Our approach builds upon classifier-guided diffusion models. But we jointly diffuse over the concatenated state-action space $\boldsymbol{\tau} = [(\boldsymbol{a}_0, \boldsymbol{s}_0), (\boldsymbol{a}_1, \boldsymbol{s}_1), ..., (\boldsymbol{a}_T, \boldsymbol{s}_T)]$. This design choice directly addresses the key limitations identified above: (1) By including states in the diffusion process, we enable explicit state conditioning and goal specification, overcoming the limitations of action-only approaches; (2) Through classifier-guided diffusion, we allow flexible goal adaptation without exhaustive training data; (3) By jointly modeling states and actions, we maintain their physical coupling while preventing ghost states through carefully designed guidance. During execution, we utilize the generated actions with denoised states for guidance, effectively bridging the gap between state conditioning and action precision.

Extended Classifier-guided Diffusion Policy Formulation. Building upon the basic classifier-guided diffusion framework (Sec. 3.3), we extend the formulation to accommodate multiple guidance (or constraints) simultaneously for complex interaction tasks. According to Eq. 6, the standard guided diffusion model follows:

$$\tilde{p}_\theta(\boldsymbol{\tau}) \propto p_\theta(\boldsymbol{\tau}) p(\mathcal{O}_{1:T} = 1 \mid \boldsymbol{\tau}) \propto p_\theta(\boldsymbol{\tau}) h(\boldsymbol{\tau}), \quad (9)$$

where we generalize $p(\mathcal{O}_{1:T} = 1 \mid \boldsymbol{\tau})$ as a behavior model $h(\boldsymbol{\tau})$. Then we further generalize this formulation through

Method	State or Action Diffusion	Classifier Guided or Free	Action Gen Method	Goal Adaptability	No Ghost States	Interaction Aware
Diffuser [23]	State	C-Guided	Inverse Dyn	✓	×	×
Decision Diffuser [1]	State	C-Free	Inverse Dyn	× (if diverse data, then ✓)	×	×
Diffusion Policy [13]	Action	C-Free	Direct	× (if diverse data, then ✓)	✓	×
DexDiffuser (Ours)	State & Action	C-Guided	Direct	✓	✓	✓

Table 1. **Comparison of diffusion-based approaches for robot manipulation.** Quantitative results on door-opening are shown in Sec. 6.

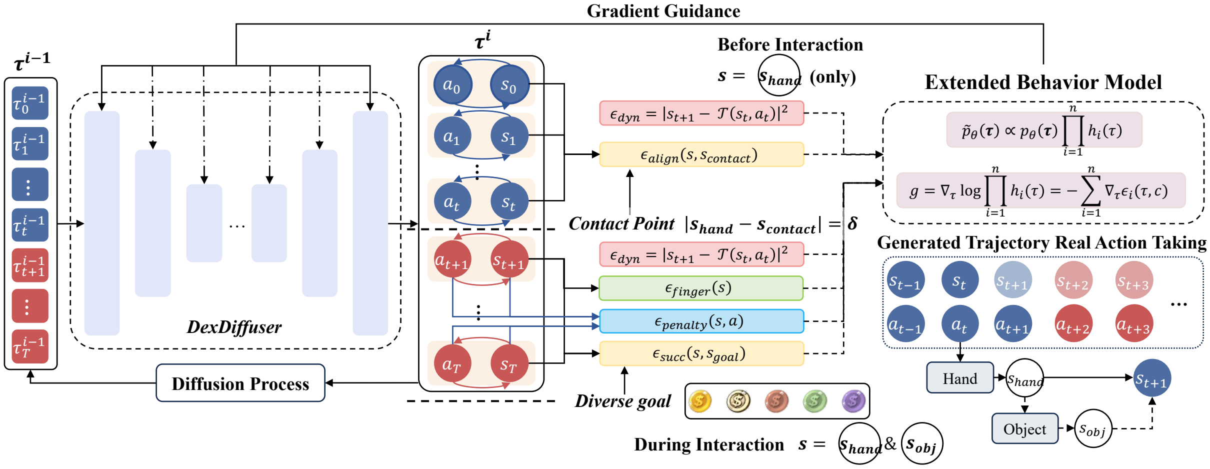

Gradient Guidance
τ^{i-1}
τ_0^{i-1}
τ_1^{i-1}
τ_t^{i-1}
τ_{t+1}^{i-1}
τ_T^{i-1}
DexDiffuser
Diffusion Process
τ^i
a_0
s_0
a_1
s_1
a_t
s_t
a_{t+1}
s_{t+1}
a_T
s_T
Before Interaction
$s = s_{hand}$ (only)
$\epsilon_{dyn} = |s_{t+1} - \mathcal{T}(s_t, a_t)|^2$
$\epsilon_{align}(s, s_{contact})$
Contact Point $|s_{hand} - s_{contact}| = \delta$
$\epsilon_{dyn} = |s_{t+1} - \mathcal{T}(s_t, a_t)|^2$
$\epsilon_{finger}(s)$
$\epsilon_{penalty}(s, a)$
$\epsilon_{succ}(s, s_{goal})$
Diverse goal
During Interaction $s = s_{hand}$ & s_{obj}
Extended Behavior Model
$\tilde{p}_\theta(\tau) \propto p_\theta(\tau) \prod_{i=1}^{n} h_i(\tau)$
$g = \nabla_\tau \log \prod_{i=1}^{n} h_i(\tau) = -\sum_{i=1}^{n} \nabla_\tau \epsilon_i(\tau, c)$
Generated Trajectory Real Action Taking
s_{t-1}
s_t
s_{t+1}
s_{t+2}
s_{t+3}
a_{t-1}
a_t
a_{t+1}
a_{t+2}
a_{t+3}
...
Hand
s_{hand}
s_{t+1}
Object
s_{obj}

Figure 3. **Framework of DexDiffuser.** DexDiffuser employs joint state-action diffusion with interaction-aware guidance. Before interaction (top right), guidance aligns the hand to the object contact point. Upon contact (bottom right), additional guidance steers both hand and object states toward the goal, enforcing physical constraints and avoiding ghost states. A learned dynamics model further ensures consistency between states and actions. This extended behavior model-based framework ensures adaptive, realistic control for manipulation.

a product of experts framework [20], where each expert represents a specific behavior model:

$$\tilde{p}_\theta(\boldsymbol{\tau}) \propto p_\theta(\boldsymbol{\tau}) \prod_{i=1}^{n} h_i(\boldsymbol{\tau}). \tag{10}$$

From an energy-based perspective, each behavior model encoding task-specific objectives or constraints is:

$$h_i(\boldsymbol{\tau}, c) = \frac{1}{\int e^{-\varepsilon_i(\boldsymbol{\tau}, c)} d\boldsymbol{\tau}} e^{-\varepsilon_i(\boldsymbol{\tau}, c)}, \tag{11}$$

where $\varepsilon_i(\boldsymbol{\tau}, c)$ represents the energy function for the i-th guidance objective, with c denoting task-specific conditions. This formulation allows combining multiple objectives (*e.g.*, reaching the target state while maintaining physical consistency) via their respective guidance functions.

Under appropriate smoothness conditions, the guidance gradient g in the reverse diffusion process (Eq. 7) can be decomposed as the sum of individual guidance gradients:

$$g = \nabla_{\boldsymbol{\tau}} \log \prod_{i=1}^{n} h_i(\boldsymbol{\tau}) = \sum_{i=1}^{n} \nabla_{\boldsymbol{\tau}} \log h_i(\boldsymbol{\tau}) = -\sum_{i=1}^{n} \nabla_{\boldsymbol{\tau}} \varepsilon_i(\boldsymbol{\tau}, c).$$

This enables integration of multiple guidance signals, each addressing different aspects of the interaction task, while maintaining a coherent optimization objective.

Contact-based Task Guidance. For contact-based manipulation tasks such as door opening and tool using, DexDiffuser employs a dual-phase interaction approach that acknowledges the fundamentally different nature of interaction before and after contact establishment. The framework automatically determines the phase transition based on the distance between the palm position and the designated contact point on the object, applying a smooth transition mask to blend between phases.

In the pre-grasp phase, our framework focuses on guiding the manipulator to achieve stable alignment with the interaction point while preventing premature object influence. We engineer two primary guidance components: 1) Alignment guidance ϵ_{align} that directs the end-effector towards precise interaction points while maintaining natural approaching trajectories; 2) Dynamics consistency guidance ϵ_{dyn} that leverages a separately trained transition model $\tilde{\mathcal{T}}(\boldsymbol{s}, \boldsymbol{a})$ to ensure physically plausible motion patterns.

Upon establishing contact (determined by palm-object proximity), the post-grasp phase activates additional guidance mechanisms: 1) Goal-directed guidance ϵ_{succ} that steers the coupled hand-object system towards target configurations; 2) Physical constraint guidance $\epsilon_{\text{penalty}}$ that prevents unrealistic state changes (*e.g.*, limiting per-step changes in both door hinge and latch angles); 3) Continued dynamics guidance ϵ_{dyn} to maintain motion feasibility.

Therefore, the guidance energy function follows,

$$\epsilon = \begin{cases} \epsilon_{\text{pre}} = \epsilon_{\text{align}} + \epsilon_{\text{dyn}} & \text{if } |\boldsymbol{s}_{\text{hand}} - \boldsymbol{s}_{\text{contact}}| > \delta \\ \epsilon_{\text{post}} = \epsilon_{\text{succ}} + \epsilon_{\text{dyn}} + \epsilon_{\text{penalty}} & \text{otherwise} \end{cases} \tag{12}$$

where $\boldsymbol{s}_{\text{hand}}$ and $\boldsymbol{s}_{\text{contact}}$ represents the state of dexterous hand and object contact point (*e.g.* door latch, hammer handle) respectively, and δ is a small threshold. The separated design of grasp proposal guidance (ϵ_{align}) and task achieving guidance (ϵ_{succ}) mirrors successful strategies in prior work [44, 46], effective for dexterous manipulation.

In-hand Manipulation Guidance. For tasks primarily involving in-hand manipulation (e.g., pen spinning, object reorientation), where objects are typically already in hand or quickly transition to in-hand states, we employ a simplified single-phase guidance structure: 1) Goal state guidance ϵ_{succ} for achieving target object configurations; 2) Active finger motion guidance to ensure realistic object manipulation; 3) Dynamics consistency guidance ϵ_{dyn} to maintain physical plausibility; 4) Physical constraint guidance $\epsilon_{\text{penalty}}$ that prevents unrealistic state changes.

$$\epsilon = \epsilon_{\text{goal}} + \epsilon_{\text{finger}} + \epsilon_{\text{dyn}} + \epsilon_{\text{penalty}}. \tag{13}$$

Specially, we define the behavior model that encourages active finger involvement as,

$$h_{\text{finger}}(\boldsymbol{\tau}, t) = H(|\boldsymbol{s}^{t+1}_{\text{finger-joints}} - \boldsymbol{s}^{t}_{\text{finger-joints}}| - \delta), \tag{14}$$

where $\boldsymbol{s}^{t}_{\text{finger-joints}}$ is the state vector of all finger joints at planning step t. δ is another small threshold and $H(\cdot)$ is the Heaviside step function [45]. Thus, the energy function ϵ_{finger} is a Dirac delta function that directly sets value when satisfying the constraints. This specialized handling prevents unrealistic "ghost states" where objects appear to move independently of finger actions, as that in Sec. 4.

Dynamics-aware Generation. A key challenge in joint state-action diffusion is maintaining consistency between generated states and actions [23] during the denoising process. Our framework addresses this through a learned dynamics model trained on demonstration data, serving as a crucial guide during trajectory generation.

$$\varepsilon_{\text{dyn}}(\boldsymbol{\tau}) = |\boldsymbol{s}_{t+1} - \mathcal{T}(\boldsymbol{s}_t, \boldsymbol{a}_t)|^2. \tag{15}$$

By penalizing state-action pairs that violate observed physical patterns, this guidance ensures our joint diffusion maintains both state conditioning benefits and action feasibility.

5.2. LLM-Based Guidance Generation

The design of task-specific guidance functions for diffusion policies traditionally requires significant manual effort, particularly for diverse dexterous manipulation tasks. To address this challenge, we leverage large language models for automated guidance generation, adopting text-to-reward paradigm from reinforcement learning literature [30, 48].

Environment Abstraction. Our approach employs a comprehensive *Pythonic* environment representation that captures the complete interaction system. This abstraction encapsulates detailed robot joint configurations, and object-environment specifications, enabling the LLM to generate precise guidance functions that account for the full complexity of dexterous manipulation tasks.

Guidance Generation. Our classifier-guided diffusion framework enables direct translation of natural language descriptions into executable guidance functions. Unlike classifier-free approaches that encode task variations implicitly through training data, our method generates explicit, adaptable guidance without extensive retraining, offering greater flexibility and interpretability in task specification.

Integration. As previous methods, we integrate multiple prompt components—including Instruction, Environment Abstraction, Background Knowledge, and Reducing Error with Code Execution—to create effective LLM-generated guidance functions. Our approach uses Few-shot Knowledge in place of traditional few-shot examples, allowing the model to access relevant functions and best practices without direct examples. Besides, each guidance component is normalized over the trajectory horizon to ensure balanced contributions across objectives while preserving their temporal structure. Detailed examples of our prompts and the resulting guidance functions are provided in Appendix D.

6. Experiments

We evaluate DexDiffuser on four challenging dexterous manipulation tasks from the Adroit hand [38] environment and the Shadow Hand environment [35]. Both environments feature a 24-joint Shadow Hand simulator with up to 30 degrees of freedom, designed to closely match the physical Shadow Dexterous Hand [41]. While we use the expert demonstrations from D4RL [17] collected by teleoperation for Adroit tasks (door opening, hammer striking, and pen reorientation), we collect 5000 expert trajectories using TQC+HER [3, 25] for the block rotate-Z task for training.

The door task represents multi-stage manipulation where the hand must reach and rotate a door handle, then pull or push the door to a target angle. The hammer task tests tool use capabilities, requiring the hand to grasp the hammer and strike a nail, while the pen and the block task evaluates in-hand dexterity, targeting continuous object reorientation.

6.1. Performance Comparisons on Goal Adaptability in Interaction-Aware Tasks

We evaluate DexDiffuser in the Door environment to test its goal adaptability across various target angles. Specifically, the evaluation tasks are opening the door to 30, 50, 70, 90 and 110 degrees, as well as a reversal task (door closing). Note that the training data only includes 90-degree door-opening demonstrations. For some of these tasks, we adjust the environment settings, such as expanding the door's range of motion, to satisfy the evaluation requirements and create distinct challenges of adaptability.

We compare DexDiffuser with five baselines: two classifier-guided methods (Diffuser [23] with Goal Inpainting that sets discrete goal states, and Diffuser with Guided

Method	Condition	Open 30°	Open 50°	Open 70°	Open 90°	Open 110°	Close Door	Average
Diffuser [23]	Goal Inpainting	16.7 ±4.7	16.7 ±12.5	6.7 ±4.7	56.7 ±9.4	10.0 ±8.2	0	17.8
Diffuser [23]	Guided Sampling	10.0 ±8.2	26.7 ±17.0	10.0 ±4.7	63.3 ±18.7	6.7 ±9.4	**60.0** ±8.2	29.5
Decision Diffuser [1]	Embedding	0	3.3 ±4.7	16.7 ±4.7	**100** ±0	**30.0** ±8.2	0	25.0
Diffusion Policy [13]	Embedding	16.7 ±4.7	3.3 ±4.7	13.3 ±12.5	**100** ±0	3.3 ±4.7	0	22.8
DexDiffuser-like	Goal Inpainting	46.7 ±4.7	13.3 ±9.4	**53.3** ±4.7	20.0 ±8.2	6.7 ±4.7	0	23.3
DexDiffuser (Ours)	Guided Sampling	**70.0** ±8.2	**56.7** ±4.7	**53.3** ±8.2	90.0 ±8.2	**26.7** ±14.1	**58.3** ±13.4	**59.2**

Table 2. **Success rates (in %) of different diffusion-based approaches in Adroit Hand [38] environment.** All models were trained on the Open 90° task only, and we test their adaptability to other task goals in Adroit Door environment. All results and standard deviation are calculated over 3 tries for 10 random seeds. Best methods and those within 5% of the best are highlighted in **bold**.

Environment	**Task**	**Diffuser [23]** (Inpaint)	**Decision Diffuser [1]**	**DexDiffuser (Ours)**
Door	Open 90°	56.7 ±9.4	**100** ±0	90.0 ±8.2
Door	Open 30°	16.7 ±4.7	16.7 ±4.7	**70.0** ±8.2
Pen	Full Re-orientation	10.0 ±0	80.0 ±8.2	**93.3** ±4.7
Pen	Half-side Re-orientation	3.3 ±4.7	23.3 ±9.4	**40.0** ±8.2
Hammer	Nail Full Drive	53.3 ±9.4	76.7 ±9.4	**90.0** ±8.2
Hammer	Nail Half Drive	23.3 ±12.5	33.3 ±4.7	**46.7** ±12.5
Manipulate Block	Rotate-Z	36.7 ±12.5	40.0 ±8.2	**50.0** ±8.2
Manipulate Block	Half-side Rotate-Z	30.0 ±0	26.7 ±4.7	**36.7** ±4.7

Table 3. **Overall performance of dexterous manipulation with goal adaptability on multiple environments and tasks.** We compare our method with one classifier-guided baseline and one classifier-free baseline. The results are calculated over 3 tries for 10 random seeds.

Sampling that leverages continuous gradients for fine control), two classifier-free methods (Decision Diffuser [1] and Diffusion Policy [13] that apply diffusion on states and actions, respectively), and a variant of DexDiffuser (denoted DexDiffuser-like) that uses goal inpainting. To enhance learning of goal condition, classifier-free methods uses *the difference between the current door angle and target angle* as the condition, rather than a fixed 90° target.

As shown in Tab. 2, classifier-free methods perform well on the 90° task, consistent with the training data, but their success declines sharply on new target angles, indicating limited adaptability to out-of-distribution targets. Classifier-guided methods demonstrate moderate but consistent performance across goal-adaptive tasks yet their overall success rates remain suboptimal due to imprecise state-action relation modeling in the policy.

In contrast, DexDiffuser, achieves consistently high success rates across nearly all tasks. While it achieves 90.0% success on the training task (90°) compared to 100% of classifier-free methods, this slight performance trade-off enables substantially better generalization. Averaging a 59.2% success rate, over twice that of the next best method (29.5%), DexDiffuser demonstrates robust adaptability and stability across both in-domain and goal-adaptive scenarios.

6.2. Evaluation on Various Dexterous Tasks

To evaluate the cross-task adaptability and goal-oriented performance of DexDiffuser, we test it across multiple dexterous manipulation tasks in the Door, Pen, Hammer and Block environments, as summarized in Table 3. In addition to the Door task (90° and 30° targets), we examine three additional tasks: Pen Re-orientation, Hammer Nail Drive and Block Rotate-Z. The Pen Re-orientation task involves aligning a pen to a specified orientation, with a particularly challenging goal-adaptability variant, Half-side Re-orientation, where training data includes only right-hemisphere orientations while test goals require left-hemisphere rotations. Similarly, the Block Rotate-Z task requires z-axis rotation control, with its Half-side variant trained on positive goal yaw angles but tested on negative ones. The variant Nail Half Drive task requires the hand to drive a nail and stop halfway before retracting, testing control precision for partial completion goals.

We compare DexDiffuser with two baselines: Diffuser [23] (Inpainting), using classifier-guided goal inpainting as in the previous section, and Decision Diffuser (DD) [1], a classifier-free approach modified to use action diffusion for the Pen Re-orientation task, as modeling dynamics for this task is particularly challenging, making direct action generation more effective than state-based diffusion. As shown in Tab. 3, DexDiffuser consistently achieves superior results across both in-domain and goal-adaptive tasks. For instance, DexDiffuser achieves 93.3% success rate on pen full re-orientation (in-domain) compared to DD's 80% and Diffuser's 10%, and 46.7% on nail half drive (goal-adaptive) vs. 23.3% for Diffuser and 33.3% for DD. Although Decision Diffuser demonstrates meaningful performance on the challenging pen half-side re-orientation, leveraging the inherent multi-modality and anisotropy of diffusion models, DexDiffuser still performs

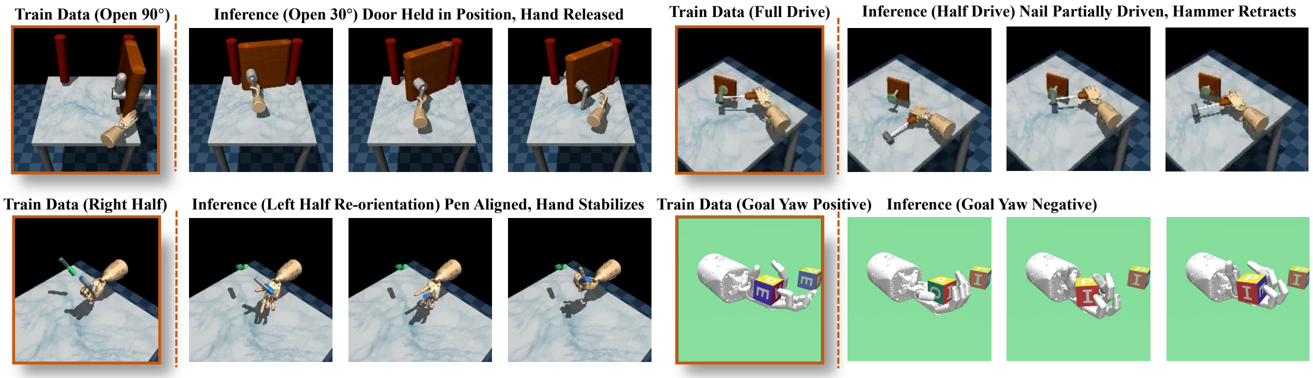

Train Data (Open 90°)
Inference (Open 30°) Door Held in Position, Hand Released
Train Data (Full Drive)
Inference (Half Drive) Nail Partially Driven, Hammer Retracts
Train Data (Right Half)
Inference (Left Half Re-orientation) Pen Aligned, Hand Stabilizes
Train Data (Goal Yaw Positive)
Inference (Goal Yaw Negative)

Figure 4. **Visualization results of goal-adaptive tasks by DexDiffuser.** For each task, a training data sample (with orange stroke) is followed by inference on novel goals beyond the training set. In the Door task, DexDiffuser guides the door to the target angle (30°) and holds it in position as the hand releases *that cannot be attained by simply truncating actions from 90° training data*. Similarly, DexDiffuser re-orients the pen or the block, stabilizes the hand, and drives the nail partially before retracting the hammer, avoiding ghost states and achieving goal adaptability.

Task	Naïve Guide	Human Craft	LLM Gen
Door Open 30°	0	70.0 ±8.2	40.0 ±8.2
Pen Half-side Re-orien	20.0 ±8.2	40.0 ±8.2	26.7 ±4.7
Hammer Half Nail	20.0 ±8.2	46.7 ±12.5	43.3 ±9.4

Table 4. **Ablation study on LLM-based guidance generation.**

better (40.0% vs. 23.3%). These results underscore DexDiffuser's robustness and adaptability across a range of manipulation tasks, demonstrating stability on familiar goals and adaptability to novel, goal-oriented challenges.

6.3. Ablation on LLM-based Guidance Generation

Table 4 presents results for different guidance methods on goal adaptability tasks. All three methods are based on the same joint state-action diffusion model. The Human Craft approach reflects our above results with manually designed, interaction-aware guidance. LLM Gen uses the method described in Sec. 5.2, with guidance functions generated by Claude Sonnet 3.5 [5]. Naive Guide directly guides the object to the goal, corresponding to the ghost state baseline. The results indicate that both Human Craft and LLM Gen significantly outperform Naive Guide across tasks, with Human Craft achieving the highest success rates.

6.4. Ablation Study of DexDiffuser Framework

We analyze the contribution of each component in DexDiffuser through ablation studies (Tab. 5), across multiple door-opening tasks (open 30°, 50°, 70°, and 90°), using the same training checkpoint for fair comparison. The baseline Diffuser[23] uses a basic goal-guidance strategy, while Dyn-guide enhances it with dynamics guidance for better state-action consistency. Joint S&A adopts joint state-action denoising like DexDiffuser but retains naive goal guidance. DexDiffuser incorporates all components and achieves the highest success rate of 67.5%, significantly outperforming the other configurations and demonstrating the effectiveness of our full design.

Method	Goal Guidance	Dynamics Guide	Joint State Action	Interact Mechanism	Overall SR
No-guide	×	×	×	×	24.1
Diffuser [23]	✓	×	×	×	27.5
Dyn-guide	✓	✓	×	×	27.5
Joint S&A	✓	×	✓	×	30.8
Dyn+Joint	✓	✓	✓	×	31.7
DexDiffuser	✓	✓	✓	✓	**67.5**

Table 5. **Ablation study on DexDiffuser framework.** We report the average success rates (overall SR) on Adroit Door environment over open 30°, 50°, 70° and 90° tasks.

6.5. Visualizations

Figure 4 illustrates the interaction-aware behavior of DexDiffuser across various goal-adaptive dexterous tasks. Each task visualization includes a sample from training and corresponding goal-adaptive execution by DexDiffuser. It ensures realistic contact by aligning hands with target objects using joint dynamics modeling, eliminating ghost states.

For instance, in the Door tasks, DexDiffuser guides the hand to grasp the handle before adjusting the door to target angles, holding the door steady as the hand releases, which is unachievable by simply slicing 90° training data. Similarly, in the Pen Re-orientation, Block Rotate-Z and Hammer Nail Drive tasks, DexDiffuser effectively manages large re-orientations and phased control: the hand rotates the pen over a wide arc, and the hammer strikes the nail partially before retracting, ensuring smooth, contact-driven transitions throughout. Results underscore DexDiffuser's ability to maintain physically realistic interactions while adapting to novel goals.

7. Conclusion

This work presents DexDiffuser, an interaction-aware diffusion planning framework for adaptive dexterous manipulation that can generalize to diverse task goals even in contact-rich scenarios. By modeling joint state-action dynamics and incorporating a dual-phase diffusion mechanism, it addresses action-state consistency issues, including the “ghost state” and generalization problems observed in previous diffusion methods. DexDiffuser’s design enables it to handle intricate multi-contact interactions through a pre-contact alignment and a post-contact control, ensuring dynamics-based and physics-realistic interactions for both seen and unseen goal-directed contact-rich manipulation. DexDiffuser sets up a standardized pipeline for interaction-aware and joint state-action diffusion planning. We believe its potential to advance the field toward diverse dexterous tasks while remaining grounded in real physics and dynamics.

References

[1] Anurag Ajay, Yilun Du, Abhi Gupta, Joshua B Tenenbaum, Tommi S Jaakkola, and Pulkit Agrawal. Is conditional generative modeling all you need for decision making? In *The Eleventh International Conference on Learning Representations*, 2023. 1, 2, 3, 4, 5, 7

[2] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik’s cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019. 1

[3] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017. 6

[4] OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020. 1

[5] Anthropic. Claude 3.5 sonnet, 2024. Available at: https://www.anthropic.com/news/claude-3-5-sonnet. 8

[6] Chi-Lam Cheang, Guangzeng Chen, Ya Jing, Tao Kong, Hang Li, Yifeng Li, Yuxiao Liu, Hongtao Wu, Jiafeng Xu, Yichu Yang, et al. Gr-2: A generative video-language-action model with web-scale knowledge for robot manipulation. *arXiv preprint arXiv:2410.06158*, 2024. 2

[7] Tao Chen, Jie Xu, and Pulkit Agrawal. A system for general in-hand object re-orientation. In *Conference on Robot Learning*, pages 297–307. PMLR, 2022. 1

[8] Yuanpei Chen, Tianhao Wu, Shengjie Wang, Xidong Feng, Jiechuan Jiang, Zongqing Lu, Stephen McAleer, Hao Dong, Song-Chun Zhu, and Yaodong Yang. Towards human-level bimanual dexterous manipulation with reinforcement learning. *Advances in Neural Information Processing Systems*, 35:5150–5163, 2022. 2

[9] Yuanpei Chen, Yiran Geng, Fangwei Zhong, Jiaming Ji, Jiechuang Jiang, Zongqing Lu, Hao Dong, and Yaodong Yang. Bi-dexhands: Towards human-level bimanual dexterous manipulation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023. 2

[10] Yuanpei Chen, Chen Wang, Yaodong Yang, and C Karen Liu. Object-centric dexterous manipulation from human motion data. *arXiv preprint arXiv:2411.04005*, 2024. 2

[11] Zerui Chen, Shizhe Chen, Cordelia Schmid, and Ivan Laptev. Vividex: Learning vision-based dexterous manipulation from human videos. *arXiv preprint arXiv:2404.15709*, 2024.

[12] Zoey Qiuyu Chen, Karl Van Wyk, Yu-Wei Chao, Wei Yang, Arsalan Mousavian, Abhishek Gupta, and Dieter Fox. Dextransfer: Real world multi-fingered dexterous grasping with minimal human demonstrations. *arXiv preprint arXiv:2209.14284*, 2022. 2

[13] Cheng Chi, Zhenjia Xu, Siyuan Feng, Eric Cousineau, Yilun Du, Benjamin Burchfiel, Russ Tedrake, and Shuran Song. Diffusion policy: Visuomotor policy learning via action diffusion. *The International Journal of Robotics Research*, page 02783649241273668, 2023. 1, 2, 3, 4, 5, 7

[14] Prafulla Dhariwal and Alexander Nichol. Diffusion models beat gans on image synthesis. *Advances in neural information processing systems*, 34:8780–8794, 2021. 3, 1

[15] Yilun Du, Sherry Yang, Bo Dai, Hanjun Dai, Ofir Nachum, Josh Tenenbaum, Dale Schuurmans, and Pieter Abbeel. Learning universal policies via text-guided video generation. *Advances in Neural Information Processing Systems*, 36, 2024. 2

[16] William Feller. On the theory of stochastic processes, with particular reference to applications. In *Selected Papers I*, pages 769–798. Springer, 2015. 3

[17] Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4rl: Datasets for deep data-driven reinforcement learning. *arXiv preprint arXiv:2004.07219*, 2020. 6

[18] Abhishek Gupta, Justin Yu, Tony Z Zhao, Vikash Kumar, Aaron Rovinsky, Kelvin Xu, Thomas Devlin, and Sergey Levine. Reset-free reinforcement learning via multi-task learning: Learning dexterous manipulation behaviors without human intervention. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6664–6671. IEEE, 2021. 2

[19] Tairan He, Zhengyi Luo, Xialin He, Wenli Xiao, Chong Zhang, Weinan Zhang, Kris Kitani, Changliu Liu, and Guanya Shi. Omnih2o: Universal and dexterous human-to-humanoid whole-body teleoperation and learning. *arXiv preprint arXiv:2406.08858*, 2024. 2

[20] Geoffrey E Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002. 5

[21] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020. 3

[22] Wenlong Huang, Igor Mordatch, Pieter Abbeel, and Deepak Pathak. Generalization in dexterous manipulation via geometry-aware multi-task learning. *arXiv preprint arXiv:2111.03062*, 2021. 1

[23] Michael Janner, Yilun Du, Joshua Tenenbaum, and Sergey Levine. Planning with diffusion for flexible behavior synthesis. In *International Conference on Machine Learning*, pages 9902–9915. PMLR, 2022. 1, 2, 3, 4, 5, 6, 7, 8

[24] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015. 4

[25] Arsenii Kuznetsov, Pavel Shvechikov, Alexander Grishin, and Dmitry Vetrov. Controlling overestimation bias with truncated mixture of continuous distributional quantile critics. In *International Conference on Machine Learning*, pages 5556–5566. PMLR, 2020. 6

[26] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500. IEEE, 2023. 3

[27] Zhixuan Liang, Yao Mu, Mingyu Ding, Fei Ni, Masayoshi Tomizuka, and Ping Luo. Adaptdiffuser: Diffusion models as adaptive self-evolving planners. In *International Conference on Machine Learning*, pages 20725–20745. PMLR, 2023. 1, 3

[28] Zhixuan Liang, Yao Mu, Hengbo Ma, Masayoshi Tomizuka, Mingyu Ding, and Ping Luo. Skilldiffuser: Interpretable hierarchical planning via skill abstractions in diffusion-based task execution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16467–16476, 2024. 3

[29] Zhengyi Luo, Jinkun Cao, Sammy Christen, Alexander Winkler, Kris Kitani, and Weipeng Xu. Grasping diverse objects with simulated humanoids. *arXiv preprint arXiv:2407.11385*, 2024. 2

[30] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023. 3, 6

[31] Priyanka Mandikal and Kristen Grauman. Dexvip: Learning dexterous grasping with human hand pose priors from video. In *Conference on Robot Learning*, pages 651–661. PMLR, 2022. 1

[32] Yao Mu, Junting Chen, Qing-Long Zhang, Shoufa Chen, Qiaojun Yu, GE Chongjian, Runjian Chen, Zhixuan Liang, Mengkang Hu, Chaofan Tao, et al. Robocodex: Multimodal code generation for robotic behavior synthesis. In *Forty-first International Conference on Machine Learning*, 2024. 3

[33] Anusha Nagabandi, Kurt Konolige, Sergey Levine, and Vikash Kumar. Deep dynamics models for learning dexterous manipulation. In *Conference on Robot Learning*, pages 1101–1112. PMLR, 2020. 2

[34] Fei Ni, Jianye Hao, Yao Mu, Yifu Yuan, Yan Zheng, Bin Wang, and Zhixuan Liang. Metadiffuser: Diffusion model as conditional planner for offline meta-rl. In *International Conference on Machine Learning*, pages 26087–26105. PMLR, 2023. 2, 3

[35] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, Vikash Kumar, and Wojciech Zaremba. Multi-goal reinforcement learning: Challenging robotics environments and request for research, 2018. 6

[36] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 1994. 3

[37] Yuzhe Qin, Yueh-Hua Wu, Shaowei Liu, Hanwen Jiang, Ruihan Yang, Yang Fu, and Xiaolong Wang. Dexmv: Imitation learning for dexterous manipulation from human videos. In *European Conference on Computer Vision*, pages 570–587. Springer, 2022. 2

[38] Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. *arXiv preprint arXiv:1709.10087*, 2017. 2, 6, 7

[39] Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. *arXiv preprint arXiv:1709.10087*, 2017. 2

[40] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18*, pages 234–241. Springer, 2015. 4

[41] Shadow Robot Company. Shadow robot. `https://www.shadowrobot.com/`, 2024. Accessed: 2024-11-14. 6

[42] Aravind Sivakumar, Kenneth Shaw, and Deepak Pathak. Robotic telekinesis: Learning a robotic hand imitator by watching humans on youtube. *arXiv preprint arXiv:2202.10448*, 2022. 2

[43] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE, 2012. 4

[44] Weikang Wan, Haoran Geng, Yun Liu, Zikang Shan, Yaodong Yang, Li Yi, and He Wang. Unidexgrasp++: Improving dexterous grasping policy learning via geometry-aware curriculum and iterative generalist-specialist learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3891–3902, 2023. 1, 6

[45] Eric W Weisstein. Heaviside step function. *https://mathworld. wolfram. com/*, 2002. 6

[46] Tianhao Wu, Yunchong Gan, Mingdong Wu, Jingbo Cheng, Yaodong Yang, Yixin Zhu, and Hao Dong. Unidexfpm: Universal dexterous functional pre-grasp manipulation via diffusion policy. *arXiv preprint arXiv:2403.12421*, 2024. 1, 2, 6

[47] Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018. 4

[48] Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2reward: Reward shaping with language models for reinforcement learning. In *The Twelfth International Conference on Learning Representations*, 2024. 3, 6

[49] Chunmiao Yu and Peng Wang. Dexterous manipulation for multi-fingered robotic hands with reinforcement learning: A review. *Frontiers in Neurorobotics*, 16:861825, 2022. 2

[50] Tony Z Zhao, Vikash Kumar, Sergey Levine, and Chelsea Finn. Learning fine-grained bimanual manipulation with low-cost hardware. *arXiv preprint arXiv:2304.13705*, 2023. 2

[51] Bohan Zhou, Haoqi Yuan, Yuhui Fu, and Zongqing Lu. Learning diverse bimanual dexterous manipulation skills from human demonstrations. *arXiv preprint arXiv:2410.02477*, 2024. 2

[52] Henry Zhu, Abhishek Gupta, Aravind Rajeswaran, Sergey Levine, and Vikash Kumar. Dexterous manipulation with deep reinforcement learning: Efficient, general, and low-cost. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 3651–3657. IEEE, 2019. 2

DexDiffuser: Interaction-aware Diffusion Planning for Adaptive Dexterous Manipulation

Supplementary Material

A. Brief Theoretical Review of Gradient Guidance in Classifier-guided Diffusion Model

For a trajectory $\boldsymbol{\tau}$, we define the reverse process of a standard diffusion model as $p_\theta(\boldsymbol{\tau}^i|\boldsymbol{\tau}^{i+1})$. To enable goal-directed generation, we introduce a classifier $p_\phi(\boldsymbol{y}|\boldsymbol{\tau}^i)$ that evaluates whether a noisy trajectory $\boldsymbol{\tau}^i$ satisfies the goal condition $\boldsymbol{y}$. The combined process is denoted as $p_{\theta,\phi}(\boldsymbol{\tau}^i|\boldsymbol{\tau}^{i+1}, \boldsymbol{y})$.

Under property of Markov process in diffusion model illustrated by [14, 27], we can establish:

$$p_{\theta,\phi}\left(\boldsymbol{y} \mid \boldsymbol{\tau}^i, \boldsymbol{\tau}^{i+1}\right) = p_\phi\left(\boldsymbol{y} \mid \boldsymbol{\tau}^i\right). \tag{16}$$

This leads to our first key theorem:

Theorem A.1. *The conditional sampling probability of the reverse diffusion process $p_{\theta,\phi}(\boldsymbol{\tau}^i \mid \boldsymbol{\tau}^{i+1}, \boldsymbol{y})$ can be decomposed into a product of the unconditional transition probability $p_\theta(\boldsymbol{\tau}^i \mid \boldsymbol{\tau}^{i+1})$ and the classifier probability $p_\phi(\boldsymbol{y} \mid \boldsymbol{\tau}^i)$, up to a normalizing constant Z:*

$$p_{\theta,\phi}(\boldsymbol{\tau}^i \mid \boldsymbol{\tau}^{i+1}, \boldsymbol{y}) = Z p_\theta(\boldsymbol{\tau}^i \mid \boldsymbol{\tau}^{i+1}) p_\phi(\boldsymbol{y} \mid \boldsymbol{\tau}^i). \tag{17}$$

Proof. By applying Bayes' theorem:

$$\begin{aligned}
p_{\theta,\phi}(\boldsymbol{\tau}^i\, |\boldsymbol{\tau}^{i+1},\ \boldsymbol{y}) &= \frac{p_{\theta,\phi}\left(\boldsymbol{\tau}^i, \boldsymbol{\tau}^{i+1}, \boldsymbol{y}\right)}{p_{\theta,\phi}\left(\boldsymbol{\tau}^{i+1}, \boldsymbol{y}\right)} \\
&= \frac{p_{\theta,\phi}\left(\boldsymbol{y} \mid \boldsymbol{\tau}^i, \boldsymbol{\tau}^{i+1}\right) p_\theta\left(\boldsymbol{\tau}^i, \boldsymbol{\tau}^{i+1}\right)}{p_\phi\left(\boldsymbol{y} \mid \boldsymbol{\tau}^{i+1}\right) p_\theta\left(\boldsymbol{\tau}^{i+1}\right)} \\
&= \frac{p_{\theta,\phi}\left(\boldsymbol{y} \mid \boldsymbol{\tau}^i, \boldsymbol{\tau}^{i+1}\right) p_\theta\left(\boldsymbol{\tau}^i \mid \boldsymbol{\tau}^{i+1}\right) p_\theta\left(\boldsymbol{\tau}^{i+1}\right)}{p_\phi\left(\boldsymbol{y} \mid \boldsymbol{\tau}^{i+1}\right) p_\theta\left(\boldsymbol{\tau}^{i+1}\right)} \\
&= \frac{p_\phi\left(\boldsymbol{y} \mid \boldsymbol{\tau}^i\right) p_\theta\left(\boldsymbol{\tau}^i \mid \boldsymbol{\tau}^{i+1}\right)}{p_\phi\left(\boldsymbol{y} \mid \boldsymbol{\tau}^{i+1}\right)},
\end{aligned}$$

where $p_\phi\left(\boldsymbol{y} \mid \boldsymbol{\tau}^{i+1}\right)$ becomes the normalizing constant Z as it is independent of $\boldsymbol{\tau}^i$. □

For practical implementation, we derive:

Theorem A.2. *Under the assumption of sufficient reverse diffusion steps, the conditional sampling probability $p_{\theta,\phi}(\boldsymbol{\tau}^i|\boldsymbol{\tau}^{i+1}, \boldsymbol{y})$ can be approximated by a modified Gaussian distribution, where the mean is shifted by the classifier gradient and the variance remains unchanged from the unconditional process:*

$$p_{\theta,\phi}(\boldsymbol{\tau}^i|\boldsymbol{\tau}^{i+1}, \boldsymbol{y}) \approx \mathcal{N}(\boldsymbol{\tau}^i; \mu_\theta + \Sigma\nabla_{\boldsymbol{\tau}} \log p_\phi\left(\boldsymbol{y} \mid \boldsymbol{\tau}^i\right), \Sigma), \tag{18}$$

where μ_θ and Σ denote the mean and variance of the unconditional reverse diffusion process $p_\theta(\boldsymbol{\tau}^i \mid \boldsymbol{\tau}^{i+1})$.

Proof. First, express the unconditional process as:

$$\begin{aligned}
p_\theta(\boldsymbol{\tau}^i \mid \boldsymbol{\tau}^{i+1}) &= \mathcal{N}(\boldsymbol{\tau}^i; \mu_\theta, \Sigma). \\
\log p_\theta(\boldsymbol{\tau}^i \mid \boldsymbol{\tau}^{i+1}) &= -\frac{1}{2}(\boldsymbol{\tau}^i - \mu_\theta)^T \Sigma^{-1} (\boldsymbol{\tau}^i - \mu_\theta) + C.
\end{aligned}$$

Apply Taylor expansion to $\log p_\phi\left(\boldsymbol{y} \mid \boldsymbol{\tau}^i\right)$ around $\boldsymbol{\tau}^i = \mu_\theta$:

$$\begin{aligned}
\log p_\phi\left(\boldsymbol{y} \mid \boldsymbol{\tau}^i\right) &= \log p_\phi\left(\boldsymbol{y} \mid \boldsymbol{\tau}^i\right)|_{\boldsymbol{\tau}^i=\mu_\theta} \\
&\quad + \left(\boldsymbol{\tau}^i - \mu_\theta\right) \nabla_{\boldsymbol{\tau}^i} \log p_\phi\left(\boldsymbol{y} \mid \boldsymbol{\tau}^i\right)\Big|_{\boldsymbol{\tau}^i=\mu_\theta}.
\end{aligned}$$

Applying the logarithm to both sides of Eq. 17:

$$\begin{aligned}
\log p_{\theta,\phi}(\boldsymbol{\tau}^i|\boldsymbol{\tau}^{i+1}, \boldsymbol{y}) &= \log p_\theta(\boldsymbol{\tau}^i|\boldsymbol{\tau}^{i+1}) + \log p_\phi(\boldsymbol{y}|\boldsymbol{\tau}^i) + C_1 \\
&= -\frac{1}{2}\left(\boldsymbol{\tau}^i - \mu_\theta\right)^T \Sigma^{-1} \left(\boldsymbol{\tau}^i - \mu_\theta\right) \\
&\quad + \left(\boldsymbol{\tau}^i - \mu_\theta\right) \nabla \log p_\phi\left(\boldsymbol{y} \mid \boldsymbol{\tau}^i\right) + C_2
\end{aligned}$$

Completing the square yields:

$$\begin{aligned}
RHS = &-\frac{1}{2}\left(\boldsymbol{\tau}^i - \mu_\theta - \Sigma\nabla \log p_\phi\left(\boldsymbol{y} \mid \boldsymbol{\tau}^i\right)\right)^T \Sigma^{-1} \\
&\times \left(\boldsymbol{\tau}^i - \mu_\theta - \Sigma\nabla \log p_\phi\left(\boldsymbol{y} \mid \boldsymbol{\tau}^i\right)\right) + C_3.
\end{aligned}$$

This establishes the Gaussian form of the approximation. □

This theoretical framework underlies our goal-directed diffusion planning approach.

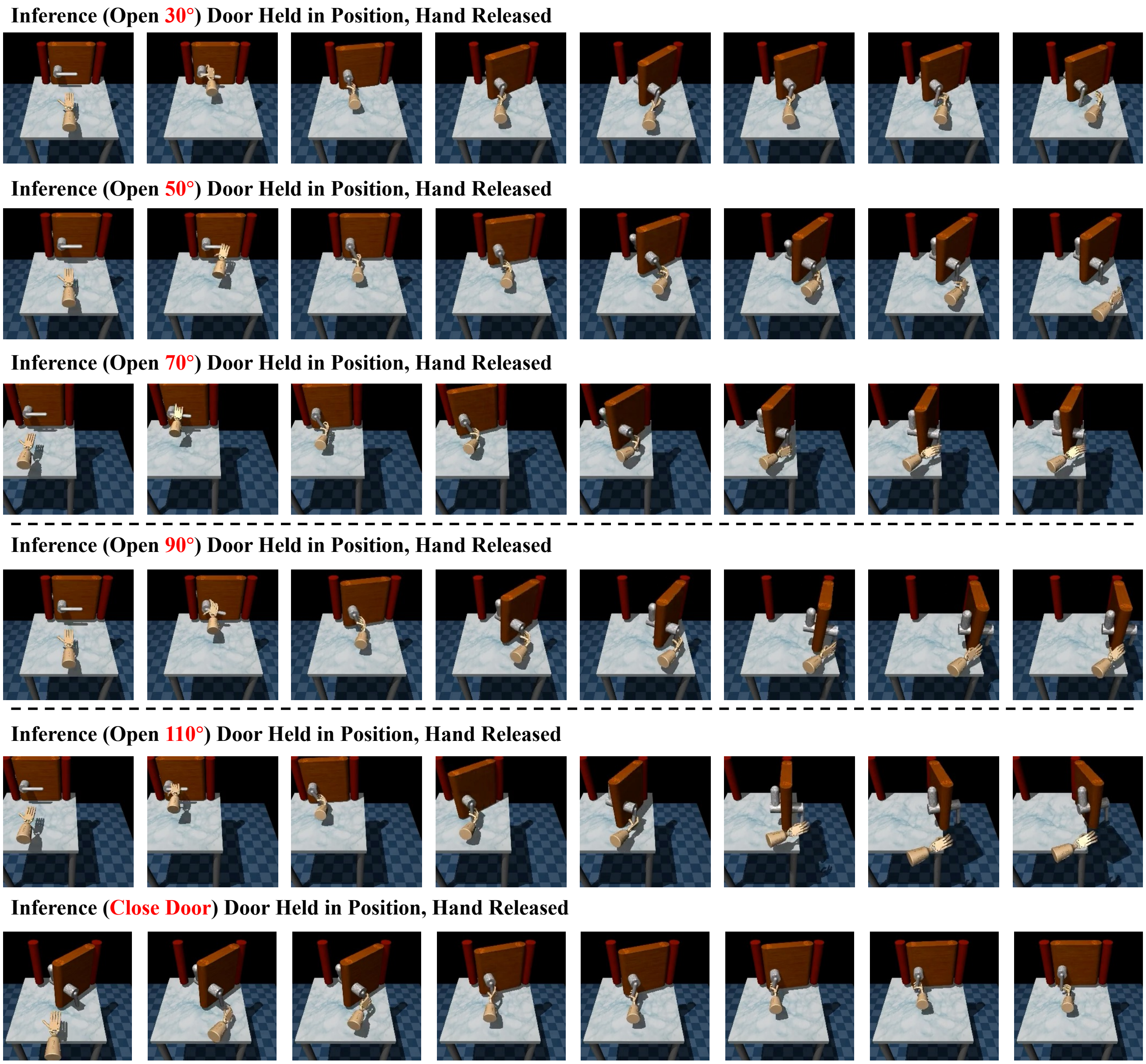
B. More Visualizations

B.1. Goal Adaptive Door Tasks

We present detailed visualizations of DexDiffuser's performance on various door manipulation tasks in Fig. 5, demonstrating its adaptability to different target angles and even task reversal. Each row shows a sequence of eight frames capturing key moments in the manipulation process.

For opening tasks with different target angles, we observe consistent behavior patterns: the hand first approaches and grasps the handle, then rotates it precisely to the specified angle, and finally releases while maintaining the door's position. Notably, even though trained only on 90° demonstrations, DexDiffuser successfully generalizes to both smaller angles (30°, 50°, 70°) and a larger angle (110°), maintaining stable control throughout the motion.

The final row demonstrates the model's capability for task reversal - closing the door. This is particularly challenging as it requires adapting the learned manipulation



Inference (Open 30°) Door Held in Position, Hand Released

Inference (Open 50°) Door Held in Position, Hand Released

Inference (Open 70°) Door Held in Position, Hand Released

Inference (Open 90°) Door Held in Position, Hand Released

Inference (Open 110°) Door Held in Position, Hand Released

Inference (Close Door) Door Held in Position, Hand Released

Figure 5. **Visualization of goal-adaptive door manipulation.** Despite training only on 90° demonstrations, DexDiffuser adapts to various target angles (30°-110°) and door closing, maintaining stable control and physical consistency throughout the motion sequence.

strategy in the opposite direction. The sequence shows the hand approaching the open door, grasping the handle, and smoothly guiding it to the closed position.

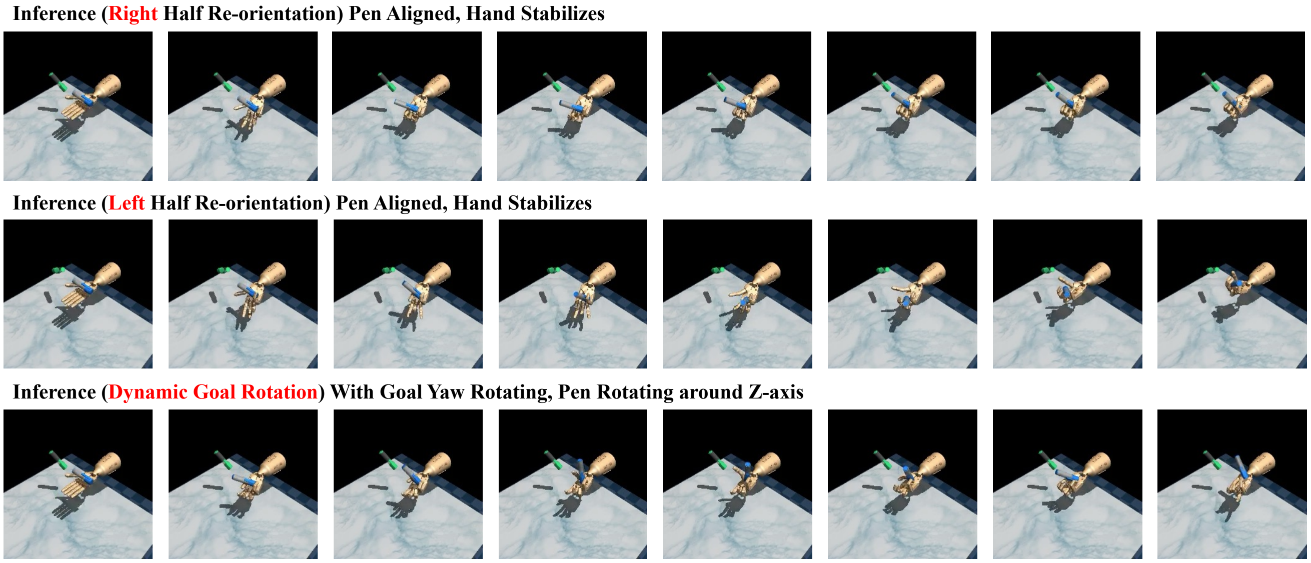
Across all variations, we observe several key characteristics: (1) Consistent contact-rich interaction phases; (2) Precise angle control regardless of target; (3) Stable door holding after reaching the target; (4) Smooth hand retraction while maintaining door position.

These visualizations illustrate DexDiffuser's robust goal adaptation capabilities while maintaining physical realism in the manipulation process.

B.2. Other Dexterous Manipulation Tasks

First, we showcase our model's capabilities on pen manipulation tasks with detailed visualizations, in Fig. 6. The first two rows demonstrate the model's performance on standard re-orientation tasks: right-half and left-half re-orientation. Notably, as the pen starts from a horizontal-right position, the left-half re-orientation (second row) is particularly challenging, requiring a large rotational arc of nearly 180 degrees to reach the target orientation in the left hemisphere.

Beyond these static goal tasks, we further evaluate our model's adaptability through a dynamic goal rotation task

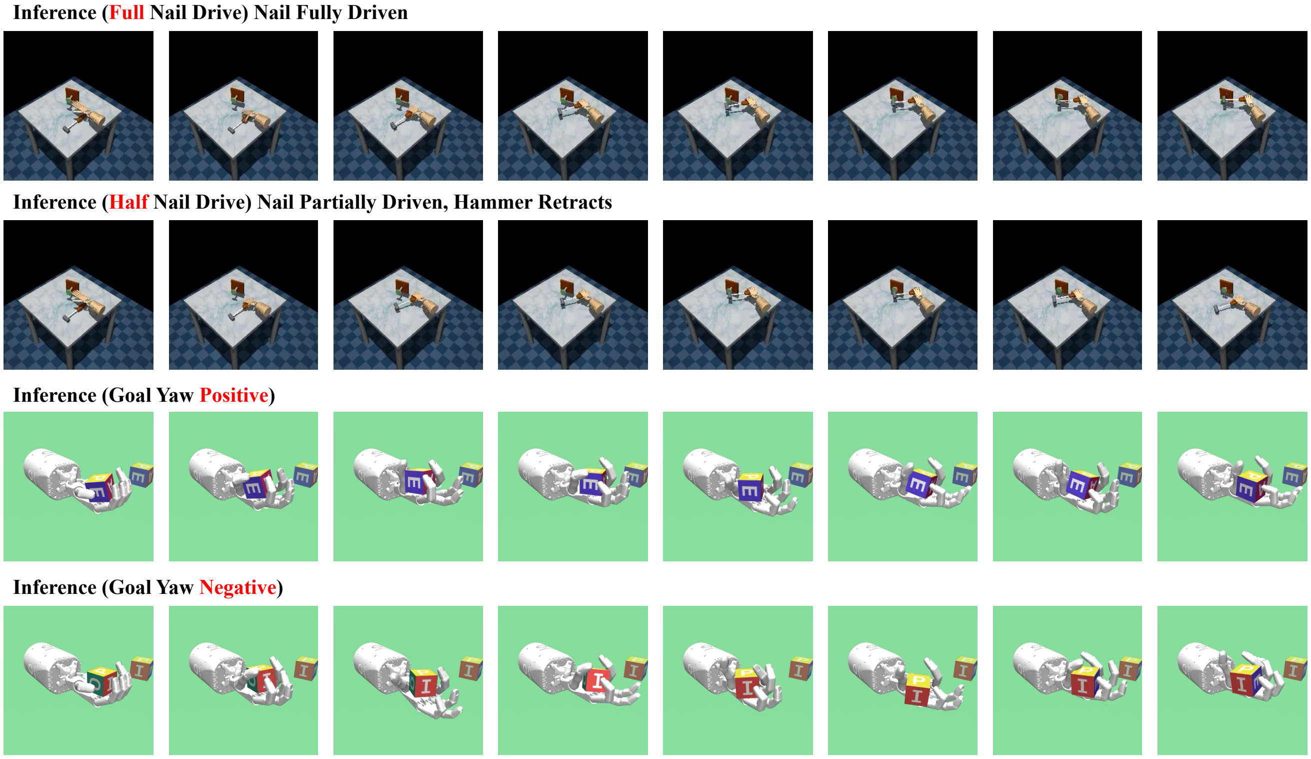


Inference (Right Half Re-orientation) Pen Aligned, Hand Stabilizes

Inference (Left Half Re-orientation) Pen Aligned, Hand Stabilizes

Inference (Dynamic Goal Rotation) With Goal Yaw Rotating, Pen Rotating around Z-axis

Figure 6. **Visualization of pen manipulation tasks.** Top: right-half re-orientation (training distribution). Middle: left-half re-orientation, requiring challenging large-arc rotation from initial horizontal-right position. Bottom: dynamic goal tracking where **target yaw angle rotates uniformly**, demonstrating the model's ability to generalize from static to dynamic goals.



Inference (Full Nail Drive) Nail Fully Driven

Inference (Half Nail Drive) Nail Partially Driven, Hammer Retracts

Inference (Goal Yaw Positive)

Inference (Goal Yaw Negative)

Figure 7. **Visualization of hammer and block manipulation tasks.** Top two rows: full and partial nail-driving tasks, demonstrating precise control over interaction depth. Bottom two rows: block orientation tasks with quaternion-based pose control, showing adaptation to both positive and negative yaw rotations while maintaining multi-angle alignment.

(third row). Using the model trained on full re-orientation data, we design a scenario where the target orientation's *yaw* angle uniformly rotates over time. The visualization demonstrates that our model successfully learns the underlying rotational dynamics *around the z-axis*, smoothly tracking the time-varying target while maintaining stable manipulation.

For the hammer task in Fig. 7, we demonstrate both full and partial nail-driving capabilities. The first row shows the complete nail-driving sequence, where the hand grasps the hammer, positions it precisely, and drives the nail fully into the board. The second row showcases our partial driving task, where the model exhibits precise control by stopping halfway and smoothly retracting the hammer, demonstrating fine-grained control over the manipulation process.

For the block manipulation task also in Fig. 7, we present two scenarios of quaternion-based orientation control. In the first sequence (Goal Yaw Positive), the hand needs to carefully adjust multiple rotational degrees of freedom to achieve the target pose, as the task requires alignment in all three orientation angles. The second sequence (Goal Yaw Negative) presents a more challenging scenario, requiring a larger rotational motion around the z-axis while maintaining control over other orientation angles. This demonstrates our model's capability to handle complex, multi-dimensional orientation targets in quaternion space.

C. Implementation Details

We implement our framework following standard diffusion model settings [23] with several modifications:

Network Architecture. We adopt a temporal U-Net [40] architecture consisting of 6 residual blocks for noise prediction. Each block contains dual temporal convolutions with group normalization [47], followed by a Mish activation [47]. Timestep information is injected through a linear embedding layer and added after the first convolution in each block. The dynamics model uses a 3-layer MLP with batch normalization, ReLU activation, and hidden dimension 512.

Training Configuration. The model is optimized using Adam [24] optimizer with a learning rate of 2×10^{-4} and batch size 256, trained for 5×10^5 steps across all tasks. For both our method and the classifier-free baselines [1, 13], we predict the denoised trajectory $\boldsymbol{\tau}_0$ directly rather than the noise term ϵ, which is incentive to the performance of classifier-free methods.

Task-Specific Parameters. We use different planning horizons during training ($T = 32$) and inference ($T = 8$ for door / block tasks, $T = 32$ for hammer / pen tasks). The diffusion process uses $K = 20$ denoising steps across tasks.

The guidance scale α is task-dependent, selected from $\{500, 1000, 2000\}$ based on empirical performance.

Computational Resources. All models are trained on a single NVIDIA GeForce RTX 3090 GPU, requiring training for approximately 30 hours per task.

D. LLM-based Guidance Generation Prompts

D.1. Overview

We present our structured prompting strategy for generating guidance functions through LLMs, which can be abstracted by the experts who developed the environment. Our prompts comprise several key components:

Expert Role Definition. We begin by defining the LLM's role as an expert in robotics, diffusion models, and code generation, specifically focusing on developing guidance functions for diffusion-based planners.

Environment Abstraction. The environment is represented through a comprehensive class hierarchy:

- BaseEnv: Contains core components (hand, objects) and observation space definition;
- AdroitHand: Detailed 28-DOF joint specification;
- Supporting Classes: Door, Handle, *etc*., with physical properties and state representations.

Technical Context. We provide three essential contexts:

- Interaction Knowledge: Defines dual-phase guidance strategy (pre-interaction and post-interaction);
- Function Call Paradigms: Specifies normalization handling and dynamics model usage through function call;
- Differentiability Requirements: Ensures differentiability, proper tensor operations, and physical consistency.

Generation Hints. We include:

- Task Instruction;
- Task-specific constraints and requirements;
- (Optional) Few-shot examples demonstrating specific techniques like soft interpolation and reward scaling.

From next page, we provide the complete prompt templates used for generating guidance functions.

D.2. Hand Door Task Prompt Example

```
You are an expert in robotics, diffusion model, reinforcement learning, and code generation.
We are going to use an Adroit Shadow Hand to complete given tasks. The action space of the robot is a
     normalized ‘Box(-1.0, 1.0, (28,), float32)‘.

Now I want you to help me write a guidance function for a diffusion-based planner.
1. The guidance function is used to steer the sampling process toward desired outcomes during the reverse
     diffusion process.
2. The guidance function should be differentiable, which computes a scalar reward indicating how well each
     intermediate trajectory aligns with the task objectives.

In manipulation tasks involving interaction with an object, such as opening a door, hammer striking, note
     that we cannot directly control the object’s state. Thus, the guidance function should consider a
     two-phase approach:
Phase 1 (Pre-Interaction Phase): The guidance function should focus solely on guiding the hand’s state to
     align with the object’s handle or interaction point.
Phase 2 (Post-Interaction Phase): Once the hand is in contact with the object, the guidance function should
     aim to move the object towards achieving the task goal. During this phase, the guidance function
     typically include the following components (some part is optional, so only include them if really
     necessary):
1. difference between the current state of the object and its goal state
2. dynamics constraints to ensure the interactions between the hand and the object are physically plausible
3. regularization of the object’s state change (e.g., limiting the hinge state change of a door to avoid
     abrupt movements).
4. [optional] extra constraint of the target object, which is often implied by the task instruction
5. [optional] extra constraint of the robot, which is often implied by the task instruction
...

Environment Description:
class BaseEnv(gym.Env):
    self.hand : AdroitHand     # The Adroit Shadow Hand used in the environment
    self.door : Door           # The Door object in the environment
    self.dt : float            # The time between two actions, in seconds

    def get_obs(self) -> np.ndarray[(30,)]:
        # Returns the observation vector
        obs = np.concatenate([
            self.hand.get_joint_positions(),                       # Indices 0-27
            [self.door.hinge.angle],                               # Index 28
            [self.door.latch.angle],                               # Index 29
            self.hand.palm.get_position()                          # Indices 30-32
            self.door.handle.get_position()                        # Indices 33-35
        ])
        return obs

class AdroitHand:
    self.arm : Arm             # The arm component of the hand
    self.wrist : Wrist         # The wrist component of the hand
    self.fingers : Fingers     # The fingers of the hand
    self.palm : Palm           # The palm of the hand

    def get_joint_positions(self) -> np.ndarray[(28,)]:
        # Returns the angular positions of all joints in the hand and arm
        return np.array([
            self.arm.translation_z.position,      # Index 0: ARTz
            self.arm.rotation_x.angle,            # Index 1: ARRx
            self.arm.rotation_y.angle,            # Index 2: ARRy
            self.arm.rotation_z.angle,            # Index 3: ARRz
            self.wrist.wrist_joint_1.angle,       # Index 4: WRJ1
            self.wrist.wrist_joint_0.angle,       # Index 5: WRJ0
            # Finger joints
            self.fingers.ffj3.angle,              # Index 6: FFJ3
            self.fingers.ffj2.angle,              # Index 7: FFJ2
            self.fingers.ffj1.angle,              # Index 8: FFJ1
            self.fingers.ffj0.angle,              # Index 9: FFJ0
            self.fingers.mfj3.angle,              # Index 10: MFJ3
            self.fingers.mfj2.angle,              # Index 11: MFJ2
            self.fingers.mfj1.angle,              # Index 12: MFJ1
            self.fingers.mfj0.angle,              # Index 13: MFJ0
            self.fingers.rfj3.angle,              # Index 14: RFJ3
            self.fingers.rfj2.angle,              # Index 15: RFJ2
            self.fingers.rfj1.angle,              # Index 16: RFJ1
            self.fingers.rfj0.angle,              # Index 17: RFJ0
            self.fingers.lfj4.angle,              # Index 18: LFJ4
            self.fingers.lfj3.angle,              # Index 19: LFJ3
            self.fingers.lfj2.angle,              # Index 20: LFJ2
            self.fingers.lfj1.angle,              # Index 21: LFJ1
            self.fingers.lfj0.angle,              # Index 22: LFJ0
            self.fingers.thj4.angle,              # Index 23: THJ4
            self.fingers.thj3.angle,              # Index 24: THJ3
            self.fingers.thj2.angle,              # Index 25: THJ2
            self.fingers.thj1.angle,              # Index 26: THJ1
            self.fingers.thj0.angle               # Index 27: THJ0
        ])

class Arm:
    self.translation_z : SlideJoint  # ARTz
    self.rotation_x : HingeJoint     # ARRx
    self.rotation_y : HingeJoint     # ARRy
    self.rotation_z : HingeJoint     # ARRz

class Wrist:
    self.wrist_joint_1 : HingeJoint  # WRJ1
    self.wrist_joint_0 : HingeJoint  # WRJ0

class Fingers:
    # Forefinger joints
    self.ffj3 : HingeJoint  # FFJ3
    self.ffj2 : HingeJoint  # FFJ2
    self.ffj1 : HingeJoint  # FFJ1
    self.ffj0 : HingeJoint  # FFJ0

    # Middle finger joints
    self.mfj3 : HingeJoint  # MFJ3
    self.mfj2 : HingeJoint  # MFJ2
    self.mfj1 : HingeJoint  # MFJ1
    self.mfj0 : HingeJoint  # MFJ0

    # Ring finger joints
    self.rfj3 : HingeJoint  # RFJ3
    self.rfj2 : HingeJoint  # RFJ2
    self.rfj1 : HingeJoint  # RFJ1
    self.rfj0 : HingeJoint  # RFJ0

    # Little finger joints
    self.lfj4 : HingeJoint  # LFJ4
    self.lfj3 : HingeJoint  # LFJ3
    self.lfj2 : HingeJoint  # LFJ2
    self.lfj1 : HingeJoint  # LFJ1
    self.lfj0 : HingeJoint  # LFJ0

    # Thumb joints
    self.thj4 : HingeJoint  # THJ4
    self.thj3 : HingeJoint  # THJ3
    self.thj2 : HingeJoint  # THJ2
    self.thj1 : HingeJoint  # THJ1
    self.thj0 : HingeJoint  # THJ0

class Palm:
    self.pose : ObjectPose         # The 3D position and orientation of the palm

    def get_position(self) -> np.ndarray[(3,)]:
        # Returns the position of the palm in world coordinates
        return self.pose.position

class Door:
    self.latch : HingeJoint        # The latch joint of the door
    self.hinge : HingeJoint        # The hinge joint of the door
    self.handle : Handle           # The handle of the door

class Handle:
    self.pose : ObjectPose         # The 3D position and orientation of the handle

    def get_position(self) -> np.ndarray[(3,)]:
        # Returns the position of the handle in world coordinates
        return self.pose.position

class HingeJoint:
    self.angle : float                 # Joint angle in radians
    self.angular_velocity : float      # Joint angular velocity in radians per second

class SlideJoint:
    self.position : float              # Position along the slide in meters
    self.velocity : float              # Velocity along the slide in meters per second

class ObjectPose:
    self.position : np.ndarray[(3,)]    # 3D position in world coordinates
    self.orientation : np.ndarray[(4,)] # Quaternion orientation (w, x, y, z)
```

Observation Index Mapping:
Index 0: Linear translation of the full arm towards the door (self.hand.arm.translation_z.position);
Index 1-27: Angular positions of the hand and arm joints (as per the joint order above);
Index 28: Angular position of the door hinge (self.door.hinge.angle);
Index 29: Angular position of the door latch (self.door.latch.angle);
Index 30-32: Position of the center of the palm in x, y, z (self.hand.palm.get_position());
Index 33-35: Position of the handle of the door in x, y, z (self.door.handle.get_position()).

Additional knowledge:

1. All angles are expressed in radians.
2. The input ‘normed_obs‘ is a tensor with shape (B, H, obs_dim), ‘normed_actions‘ is a tensor with shape (B, H, act_dim), where B is the batch size, H is the horizon length. The normed_obs is gotten from ‘normed_obs = get_obs()‘.
3. If you need to match the observations or actions to some explicit value and if not without_normalizer, you should unnormalize them using ‘self.unnormalize(normed_obs, is_obs=True)‘.
4. If ‘dyn_model‘ is provided, please call ‘self.cal_dyn_reward(state=normed_obs, action=normed_actions)‘ to calculates the reward for dynamics inconsistency (a scalar value) between generated states and actions. Only consider it in phase 2. Pay attention the input should be normed_obs and normed_actions before unnormalizing them.
5. Use L2 distance via ‘torch.norm(,p=2)‘ to calculate all the difference instead of mse loss or ‘torch.abs‘.
6. The transition between Phase 1 and Phase 2 by using a grasp mask to determine if the hand has successfully grasped the object. Use a condition like ‘mask = torch.norm(palm_pos[:, 0, :] - handle_pos[:, 0, :], p=2, dim=1) < 0.1‘ to switch from guiding only the hand to guiding both the hand and the object.

You are allowed to use any existing Python package if applicable, but only use them when absolutely necessary. Please import the required packages at the beginning of the function.

I want it to fulfill the following task: **{"Write a guidance function for a diffusion-based planner that helps the Adroit Shadow Hand open the door to 30 degrees (pi/6 radians)."}**

1. Please think step by step and explain what it means in the context of this environment;
2. Then write a differentiable guidance function that guides the planner to generate actions smoothly based on the current normed state and action, with the function prototype as ‘def guidance_fn(self, normed_obs, normed_actions, dyn_model=None, without_normalizer=False)‘. The function should return the ‘reward‘ as a torch.Tensor of shape ‘(B,)‘;
3. Make sure the guidance aligns with the two phases: In Phase 1, only calculate a pre-grasp reward to guide the hand closer to the object. In Phase 2, guide both the object toward the final task goal. Ensure object velocity constraints are applied to regulate object state changes.
4. All the reward including the goal achieving reward should be across all horizon steps. For some term, use ‘torch.mean()‘ to accumulate reward over the horizon. For terms where the last dimension is 1 (such as angles), we should use torch.squeeze to remove that dimension before calculating the norm at dimension 1, rather than dimension 2.
5. Use ‘self.scaling_factors‘ as an empty dictionary by default. If the scaling factor for any reward component does not exist, initialize it adaptively to make that first reward term in batch approximately 12 initially, except for the goal-achieving reward (make the reward 30) and the dynamics reward (make it 1.2).
6. Take care of variables’ type, never use functions or variables not provided. Ensure that all operations are compatible with PyTorch tensors and the function is differentiable. Do not use any absolute value operation and inplace operations, e.g. ‘x += 1‘, ‘x[0] = 1‘, using ‘x = x + 1‘ instead.
7. Pay attention to the physical meaning of each dimension in the observation and action data as explained in the environment description above.
8. When you writing code, you can also add some comments as your thought, like this:

```
# Here unnormalize the observations if a normalizer is provided
# Here use ‘torch.norm‘ to compute the L2 distance between the current and target angles for the door hinge
# Here cauculate the grasp mask for the pre-interaction phase
```

Few-shot hint:

1. Ensure that the guidance function uses soft interpolation for targets, e.g., smoothly guiding the door hinge angle towards soft goals over the trajectory horizon like ‘interpolated_angle = (1 - alpha) * current_angle + alpha * target_angle‘.

D.3. Hand Pen Task Prompt Example

```
You are an expert in robotics, diffusion model, reinforcement learning, and code generation.
We are going to use an Adroit Shadow Hand to complete given tasks. The action space of the robot is a
     normalized ‘Box(-1.0, 1.0, (28,), float32)‘.

Now I want you to help me write a guidance function for a diffusion-based planner.
1. The guidance function is used to steer the sampling process toward desired outcomes during the reverse
     diffusion process.
2. The guidance function should be differentiable, which computes a scalar reward indicating how well each
     intermediate trajectory aligns with the task objectives.

In manipulation tasks involving interaction with an object, such as rotating a pen, note that we cannot
     directly control the object’s state. Thus, the guidance function should consider a two-phase approach:
[optional] Phase 1 (Pre-Interaction Phase): The guidance function should focus solely on guiding the hand’s
     state to align with the object’s handle or interaction point.
Phase 2 (Post-Interaction Phase): Once the hand is in contact with the object, the guidance function should
     aim to move the object towards achieving the task goal. During this phase, the guidance function
     typically include the following components (some part is optional, so only include them if really
     necessary):
1. difference between the current state of the object and its goal state
2. dynamics constraints to ensure the interactions between the hand and the object are physically plausible
3. regularization of the object’s state change (e.g., encourage the hand joint movement to enhance
     interaction with the object).
4. [optional] extra constraint of the target object, which is often implied by the task instruction
5. [optional] extra constraint of the robot, which is often implied by the task instruction
...
```

Environment Description:

```
class BaseEnv(gym.Env):
    self.hand : AdroitHand     # The Adroit Shadow Hand used in the environment
    self.pen : Pen             # The Pen object in the environment
    self.target : Target       # The target orientation for the pen
    self.dt : float            # The time between two actions, in seconds

    def get_obs(self) -> np.ndarray[(36,)]:
        # Returns the observation vector
        obs = np.concatenate([
            self.hand.get_joint_positions(),                      # Indices 0-23
            self.pen.get_qpos()                                   # Indices 24-29
            self.pen.get_relative_rotation(),                     # Indices 30-32
            self.target.get_relative_rotation(),                  # Indices 33-35
        ])
        return obs

class AdroitHand:
    self.wrist : Wrist         # The wrist component of the hand
    self.fingers : Fingers     # The fingers of the hand
    self.palm : Palm           # The palm of the hand

    def get_joint_positions(self) -> np.ndarray[(24,)]:
        # Returns the angular positions of all joints in the hand
        return np.array([
            self.wrist.wrist_joint_1.angle,       # Index 0: WRJ1
            self.wrist.wrist_joint_0.angle,       # Index 1: WRJ0
            # Finger joints
            self.fingers.ffj3.angle,              # Index 2: FFJ3
            self.fingers.ffj2.angle,              # Index 3: FFJ2
            self.fingers.ffj1.angle,              # Index 4: FFJ1
            self.fingers.ffj0.angle,              # Index 5: FFJ0
            self.fingers.mfj3.angle,              # Index 6: MFJ3
            self.fingers.mfj2.angle,              # Index 7: MFJ2
            self.fingers.mfj1.angle,              # Index 8: MFJ1
            self.fingers.mfj0.angle,              # Index 9: MFJ0
            self.fingers.rfj3.angle,              # Index 10: RFJ3
            self.fingers.rfj2.angle,              # Index 11: RFJ2
            self.fingers.rfj1.angle,              # Index 12: RFJ1
            self.fingers.rfj0.angle,              # Index 13: RFJ0
            self.fingers.lfj4.angle,              # Index 14: LFJ4
            self.fingers.lfj3.angle,              # Index 15: LFJ3
            self.fingers.lfj2.angle,              # Index 16: LFJ2
            self.fingers.lfj1.angle,              # Index 17: LFJ1
            self.fingers.lfj0.angle,              # Index 18: LFJ0
            self.fingers.thj4.angle,              # Index 19: THJ4
            self.fingers.thj3.angle,              # Index 20: THJ3
            self.fingers.thj2.angle,              # Index 21: THJ2
            self.fingers.thj1.angle,              # Index 22: THJ1
            self.fingers.thj0.angle               # Index 23: THJ0
        ])

class Wrist:
    self.wrist_joint_1 : HingeJoint  # WRJ1
    self.wrist_joint_0 : HingeJoint  # WRJ0

class Fingers:
    # Forefinger joints
    self.ffj3 : HingeJoint  # FFJ3
    self.ffj2 : HingeJoint  # FFJ2
    self.ffj1 : HingeJoint  # FFJ1
    self.ffj0 : HingeJoint  # FFJ0

    # Middle finger joints
    self.mfj3 : HingeJoint  # MFJ3
    self.mfj2 : HingeJoint  # MFJ2
    self.mfj1 : HingeJoint  # MFJ1
    self.mfj0 : HingeJoint  # MFJ0

    # Ring finger joints
    self.rfj3 : HingeJoint  # RFJ3
    self.rfj2 : HingeJoint  # RFJ2
    self.rfj1 : HingeJoint  # RFJ1
    self.rfj0 : HingeJoint  # RFJ0

    # Little finger joints
    self.lfj4 : HingeJoint  # LFJ4
    self.lfj3 : HingeJoint  # LFJ3
    self.lfj2 : HingeJoint  # LFJ2
    self.lfj1 : HingeJoint  # LFJ1
    self.lfj0 : HingeJoint  # LFJ0

    # Thumb joints
    self.thj4 : HingeJoint  # THJ4
    self.thj3 : HingeJoint  # THJ3
    self.thj2 : HingeJoint  # THJ2
    self.thj1 : HingeJoint  # THJ1
    self.thj0 : HingeJoint  # THJ0

class Palm:
    self.pose : ObjectPose         # The 3D position and orientation of the palm

    def get_position(self) -> np.ndarray[(3,)]:
        # Returns the position of the palm in world coordinates
        return self.pose.position

class Pen:
    self.pose : ObjectPose         # The 3D position and orientation of the pen
    self.qpos : np.ndarray[(6,)]   # The qpos values of the pen’s joints

    def get_position(self) -> np.ndarray[(3,)]:
        # Returns the position of the pen in world coordinates
        return self.pose.position

    def get_relative_rotation(self) -> np.ndarray[(3,)]:
        # Returns the relative rotation of the pen
        return self.pose.orientation

    def get_position_to_target(self, target: Target) -> np.ndarray[(3,)]:
        # Returns the position vector from the pen to the target
        return target.pose.position - self.pose.position

    def get_rotation_to_target(self, target: Target) -> np.ndarray[(3,)]:
        # Returns the rotation vector from the pen to the target
        return target.pose.orientation - self.pose.orientation

    def get_qpos(self) -> np.ndarray[(6,)]:
        # Returns the qpos values of the pen’s joints
        return self.qpos

class Target:
    self.pose : ObjectPose         # The 3D position
```

Additional knowledge:

1. All angles are expressed in radians.

~~~~
2. The input `normed_obs` is a tensor with shape (B, H, obs_dim), `normed_actions` is a tensor with shape (B,
     H, act_dim), where B is the batch size, H is the horizon length. The normed_obs is gotten from
     `normed_obs = get_obs()`.
3. If you need to match the observations or actions to some explicit value and if not without_normalizer, you
     should unnormalize them using `self.unnormalize(normed_obs, is_obs=True)`.
4. If `dyn_model` is provided, please call `self.cal_dyn_reward(state=normed_obs, action=normed_actions)` to
     calculates the reward for dynamics inconsistency (a scalar value) between generated states and actions.
     Only consider it in phase 2. Pay attention the input should be normed_obs and normed_actions before
     unnormalizing them.
5. Use L2 distance via `torch.norm(,p=2)` to calculate all the difference instead of mse loss or `torch.abs`.
     For terms where the last dimension is 1 (such as angles), we should use torch.squeeze to remove that
     dimension before calculating the norm at dimension 1, rather than dimension 2.

You are allowed to use any existing Python package if applicable, but only use them when absolutely
     necessary. Please import the required packages at the beginning of the function.

I want it to fulfill the following task:  {"Write a guidance function for a diffusion-based planner that helps
     the Adroit Shadow Hand rotate the pen to the desired target orientation."}
1. Please think step by step and explain what it means in the context of this environment;
2. Then write a differentiable guidance function that guides the planner to generate actions smoothly based
     on the current normed state and action, with the function prototype as `def guidance_fn(self, normed_obs,
     normed_actions, dyn_model=None, without_normalizer=False, desired_pen=None)`. The function should return
     the `reward` as a torch.Tensor of shape `(B,)`;
3. All the reward including the goal achieving reward should be across all horizon steps. For some term, use
     `torch.mean()` to accumulate reward over the horizon.
4. Use input `desired_pen` as the target rotation, but you should reshape it by `target_rotation =
     desired_pen[..., -3:].reshape(batch_size, 1, 3).repeat(1, horizon, 1)`. You should first normalize the
     direction vector and then use inner product to calculate the similarity between two orientations.
5. Don't directly use actions to penalize the reward, but you can use the difference between the current and
     previous hand joint states to penalize the reward. You encourage the hand joint movement to enhance
     interaction with the object.
6. Use `self.scaling_factors` as an empty dictionary by default. If the scaling factor for any reward
     component does not exist, initialize it adaptively to make that first reward term in batch approximately
     1 initially, except for the the dynamics reward (make it 2.).
7. Take care of variables' type, never use functions or variables not provided. Ensure that all operations
     are compatible with PyTorch tensors and the function is differentiable. Do not use any absolute value
     operation and inplace operations, e.g. `x += 1`, `x[0] = 1`, using `x = x + 1` instead.
8. Pay attention to the physical meaning of each dimension in the observation and action data as explained in
     the environment description above.
9. When you writing code, you can also add some comments as your thought, like this:
```
# Here unnormalize the observations if a normalizer is provided
# Here use `torch.norm` to compute the L2 distance between the current and target angles for the door hinge
```

Few-shot hint:
1. Ensure that the guidance function uses soft interpolation for targets, e.g., smoothly guiding the pen
     orientation towards soft goals over the trajectory horizon like `interpolated_angle = (1 - alpha) *
     current_obj_orien + alpha * desired_orien`. If use soft goals, don't calculate another hard goal reward.
2. No smoothness reward for the pen movement. Only consider the smoothness of the hand joint movement.
~~~~

D.4. Hand Hammer Task Prompt Example

~~~~
You are an expert in robotics, diffusion model, reinforcement learning, and code generation.
We are going to use an Adroit Shadow Hand to complete given tasks. The action space of the robot is a
     normalized `Box(-1.0, 1.0, (28,), float32)`.

Now I want you to help me write a guidance function for a diffusion-based planner.
1. The guidance function is used to steer the sampling process toward desired outcomes during the reverse
     diffusion process.
2. The guidance function should be differentiable, which computes a scalar reward indicating how well each
     intermediate trajectory aligns with the task objectives.

In manipulation tasks involving interaction with an object, such as opening a door, hammer striking, note
     that we cannot directly control the object's state. Thus, the guidance function should consider a
     two-phase approach:
Phase 1 (Pre-Interaction Phase): The guidance function should focus solely on guiding the hand's state to
     align with the object's handle or interaction point.
Phase 2 (Post-Interaction Phase): Once the hand is in contact with the object, the guidance function should
     aim to move the object towards achieving the task goal. During this phase, the guidance function
     typically include the following components (some part is optional, so only include them if really
     necessary):
~~~~

```
1. difference between the current state of the object and its goal state
2. dynamics constraints to ensure the interactions between the hand and the object are physically plausible
3. regularization of the object’s state change (e.g., limiting the hinge state change of a door to avoid
     abrupt movements).
4. [optional] extra constraint of the target object, which is often implied by the task instruction
5. [optional] extra constraint of the robot, which is often implied by the task instruction
...
```

Environment Description:

```
class BaseEnv(gym.Env):
    self.hand : AdroitHand     # The Adroit Shadow Hand used in the environment
    self.hammer : Hammer       # The Hammer object in the environment
    self.nail : Nail          # The Nail object in the environment
    self.dt : float           # The time between two actions, in seconds

    def get_obs(self) -> np.ndarray[(46,)]:
        # Returns the observation vector
        obs = np.concatenate([
            self.hand.get_joint_positions(),                        # Indices 0-25
            [self.nail.insertion_displacement],                     # Index 26
            self.hammer.get_qpos(),                                 # Indices 27-32
            self.hand.palm.get_position(),                          # Indices 33-35
            self.hammer.get_position(),                             # Indices 36-38
            self.hammer.get_orientation(),                          # Indices 39-41
            self.nail.get_position(),                               # Indices 42-44
            [self.nail.force]                                       # Index 45
        ])
        return obs

class AdroitHand:
    self.arm : Arm             # The arm component of the hand
    self.wrist : Wrist         # The wrist component of the hand
    self.fingers : Fingers     # The fingers of the hand
    self.palm : Palm           # The palm of the hand

    def get_joint_positions(self) -> np.ndarray[(26,)]:
        # Returns the angular positions of all joints in the hand and arm
        return np.array([
            self.arm.rotation_x.angle,            # Index 0: ARRx
            self.arm.rotation_y.angle,            # Index 1: ARRy
            self.wrist.wrist_joint_1.angle,       # Index 2: WRJ1
            self.wrist.wrist_joint_0.angle,       # Index 3: WRJ0
            # Finger joints
            self.fingers.ffj3.angle,              # Index 4: FFJ3
            self.fingers.ffj2.angle,              # Index 5: FFJ2
            self.fingers.ffj1.angle,              # Index 6: FFJ1
            self.fingers.ffj0.angle,              # Index 7: FFJ0
            self.fingers.mfj3.angle,              # Index 8: MFJ3
            self.fingers.mfj2.angle,              # Index 9: MFJ2
            self.fingers.mfj1.angle,              # Index 10: MFJ1
            self.fingers.mfj0.angle,              # Index 11: MFJ0
            self.fingers.rfj3.angle,              # Index 12: RFJ3
            self.fingers.rfj2.angle,              # Index 13: RFJ2
            self.fingers.rfj1.angle,              # Index 14: RFJ1
            self.fingers.rfj0.angle,              # Index 15: RFJ0
            self.fingers.lfj4.angle,              # Index 16: LFJ4
            self.fingers.lfj3.angle,              # Index 17: LFJ3
            self.fingers.lfj2.angle,              # Index 18: LFJ2
            self.fingers.lfj1.angle,              # Index 19: LFJ1
            self.fingers.lfj0.angle,              # Index 20: LFJ0
            self.fingers.thj4.angle,              # Index 21: THJ4
            self.fingers.thj3.angle,              # Index 22: THJ3
            self.fingers.thj2.angle,              # Index 23: THJ2
            self.fingers.thj1.angle,              # Index 24: THJ1
            self.fingers.thj0.angle               # Index 25: THJ0
        ])

class Hammer:
    self.pose : ObjectPose         # The 3D position and orientation of the hammer
    self.velocity : ObjectVelocity # Linear and angular velocities of the hammer
    self.OBJTx : SlideJoint        # The slide joint along the x-axis
    self.OBJTy : SlideJoint        # The slide joint along the y-axis
    self.OBJTz : SlideJoint        # The slide joint along the z-axis
    self.OBJRx : RevoluteJoint     # The revolute joint around the x-axis
    self.OBJRy : RevoluteJoint     # The revolute joint around the y-axis
    self.OBJRz : RevoluteJoint     # The revolute joint around the z-axis
    def get_position(self) -> np.ndarray[(3,)]:
        # Returns the position of the hammer’s center of mass in world coordinates
        return self.pose.position

    def get_orientation(self) -> np.ndarray[(3,)]:
        # Returns the relative rotation of the hammer with respect to x,y,z axes
        return self.pose.get_euler_angles()

    def get_qpos(self) -> np.ndarray[(6,)]:
        # Returns the joint positions of the hammer
        return np.array([self.OBJTx.position, self.OBJTy.position, self.OBJTz.position,
                         self.OBJRx.angle, self.OBJRy.angle, self.OBJRz.angle])

class Nail:
    self.pose : ObjectPose            # The 3D position of the nail
    self.insertion_displacement : float # Current insertion depth of the nail
    self.force : float                # Linear force exerted on the nail head

    def get_position(self) -> np.ndarray[(3,)]:
        # Returns the position of the nail in world coordinates
        return self.pose.position

class ObjectVelocity:
    self.linear : np.ndarray[(3,)]    # Linear velocity in x,y,z
    self.angular : np.ndarray[(3,)]   # Angular velocity around x,y,z axes

class ObjectPose:
    self.position : np.ndarray[(3,)]    # 3D position in world coordinates
    self.orientation : np.ndarray[(4,)]  # Quaternion orientation (w, x, y, z)

    def get_euler_angles(self) -> np.ndarray[(3,)]:
        # Returns the orientation as Euler angles (roll, pitch, yaw)
        return quaternion_to_euler(self.orientation)
```

```
Observation Index Mapping:
Index 0-25: Angular positions of the hand joints (in radians);
Index 26: Insertion displacement of nail (in meters) range from -0.01 to 0.09;
Index 27-32: Qpos of the hammer joints (in meters and radians);
Index 33-35: Position of the center of the palm in x,y,z (in meters);
Index 36-38: Position of the hammer’s center of mass in x,y,z (in meters);
Index 39-41: Relative rotation of hammer’s center of mass w.r.t x,y,z axes (in radians);
Index 42-44: Position of the nail in x,y,z (in meters);
Index 45: Linear force exerted on the head of the nail (in Newtons) range from -1.0 to 1.0.
```

Additional knowledge:

1. All angles are expressed in radians.
2. The input ‘normed_obs‘ is a tensor with shape (B, H, obs_dim), ‘normed_actions‘ is a tensor with shape (B, H, act_dim), where B is the batch size, H is the horizon length. The normed_obs is gotten from ‘normed_obs = get_obs()‘.
3. If you need to match the observations or actions to some explicit value and if not without_normalizer, you should unnormalize them using ‘self.unnormalize(normed_obs, is_obs=True)‘.
4. If ‘dyn_model‘ is provided, please call ‘self.cal_dyn_reward(state=normed_obs, action=normed_actions)‘ to calculates the reward for dynamics inconsistency (a scalar value) between generated states and actions. Only consider it in phase 2. Pay attention the input should be normed_obs and normed_actions before unnormalizing them.
5. Use L2 distance via ‘torch.norm(,p=2)‘ to calculate all the difference instead of mse loss or ‘torch.abs‘.
6. The transition between Phase 1 and Phase 2 by using a grasp mask to determine if the hand has successfully grasped the object. Use a condition like ‘mask = torch.norm(palm_pos[:, 0, :] - handle_pos[:, 0, :], p=2, dim=1) < 0.1‘ to switch from guiding only the hand to guiding both the hand and the object.

You are allowed to use any existing Python package if applicable, but only use them when absolutely necessary. Please import the required packages at the beginning of the function.

I want it to fulfill the following task: **{"Write a guidance function for a diffusion-based planner that helps the Adroit Shadow Hand grasp the hammer and only drive half nail into the board."}**

1. Please think step by step and explain what it means in the context of this environment;
2. Then write a differentiable guidance function that guides the planner to generate actions smoothly based on the current normed state and action, with the function prototype as ‘def guidance_fn(self, normed_obs, normed_actions, dyn_model=None, without_normalizer=False)‘. The function should return the ‘reward‘ as a torch.Tensor of shape ‘(B,)‘;
3. Make sure the guidance aligns with the two phases: In Phase 1, only calculate a pre-grasp reward to guide the hand closer to the object. In Phase 2, guide both the object toward the final task goal. Ensure object velocity constraints are applied to regulate object state changes.
4. All the reward including the goal achieving reward should be across all horizon steps. For some term, use ‘torch.mean()‘ to accumulate reward over the horizon. For terms where the last dimension is 1 (such as angles), we should use torch.squeeze to remove that dimension before calculating the norm at dimension 1, rather than dimension 2.

~~~
5. Take care of variables’ type, never use functions or variables not provided. Ensure that all operations
     are compatible with PyTorch tensors and the function is differentiable. Do not use any absolute value
     operation and inplace operations, e.g. ‘x += 1‘, ‘x[0] = 1‘, using ‘x = x + 1‘ instead.
6. Pay attention to the physical meaning of each dimension in the observation and action data as explained in
     the environment description above.
7. When you writing code, you can also add some comments as your thought, like this:
```
# Here unnormalize the observations if a normalizer is provided
# Here use ‘torch.norm‘ to compute the L2 distance between the current and target angles for the door hinge
# Here cauculate the grasp mask for the pre-interaction phase
```

Few-shot hint:
1. Use nail insertion displacement (range from -0.01 No Drive to 0.09 Full Drive) as a measure of nail
     insertion degree.
2. Constraint both the hammer qpos changes and the hammer position & rotation changes (pos+rot as 1 item) to
     avoid abrupt movements.
3. Use ‘self.scaling_factors‘ as an empty dictionary by default. If the scaling factor for any reward
     component does not exist, initialize it adaptively to make that first reward term in batch approximately
     6 initially, except for the hammer qpos change constraint reward (make the reward 12) and the dynamics
     reward (make it 0.3).
~~~

Then we can replace the task instruction marked by red with specific requirements according to different tasks. This structured prompting approach enables LLMs to generate physically consistent and task-appropriate guidance functions for goal-adaptive dexterous manipulation.

E. Samples of Generated Guidance Function

E.1. Sample of Guidance Function on Hand Door Task

```
def guidance_fn(self, normed_obs, normed_actions, dyn_model=None, without_normalizer=False):
    import math
    """
    Guidance function for door opening task with the Adroit hand.
    Args:
        normed_obs: Normalized observations tensor of shape (B, H, obs_dim)
        normed_actions: Normalized actions tensor of shape (B, H, act_dim)
        dyn_model: Optional dynamics model for consistency checking
        without_normalizer: Boolean indicating if normalization should be skipped
    Returns:
        reward: Tensor of shape (B,) containing the guidance rewards
    """
    batch_size, horizon, _ = normed_obs.shape

    # Get the unnormalized observations if normalizer is provided
    obs = normed_obs if without_normalizer else self.unnormalize(normed_obs, is_obs=True)

    # Extract relevant state information
    palm_pos = obs[:, :, 30:33]  # Palm position
    handle_pos = obs[:, :, 33:36]  # Door handle position
    door_hinge_angle = obs[:, :, 28:29]  # Door hinge angle

    # Define target door angle (30 degrees = math.pi/6 radians)
    target_door_angle = torch.full_like(door_hinge_angle, math.pi / 6)

    # Calculate grasp mask for phase transition
    # Use distance between palm and handle to determine if hand is close enough
    grasp_threshold = 0.1
    grasp_dist = torch.norm(palm_pos - handle_pos, p=2, dim=2)
    grasp_mask = (grasp_dist < grasp_threshold).float()

    # Phase 1: Pre-interaction rewards
    # Calculate reaching reward - guide hand to door handle
    reaching_dist = torch.norm(palm_pos - handle_pos, p=2, dim=2)
    reaching_reward = -torch.mean(reaching_dist, dim=1)  # Average over horizon

    # Scale reaching reward if not already scaled
    if 'reaching' not in self.scaling_factors:
        # Initialize scaling to make initial reward approximately 12
        with torch.no_grad():
            initial_reaching = -reaching_reward[0].item()
```

```
            self.scaling_factors[’reaching’] = 12.0 / (initial_reaching + 1e-8)
    reaching_reward = reaching_reward * self.scaling_factors[’reaching’]

    # Phase 2: Post-interaction rewards
    # Calculate door angle reward
    door_angle_diff = torch.norm(door_hinge_angle - target_door_angle, p=2, dim=2)
    door_reward = -torch.mean(door_angle_diff, dim=1)  # Average over horizon

    # Scale door reward if not already scaled
    if ’door’ not in self.scaling_factors:
        # Initialize scaling to make initial reward approximately 30
        with torch.no_grad():
            initial_door = -door_reward[0].item()
            self.scaling_factors[’door’] = 30.0 / (initial_door + 1e-8)
    door_reward = door_reward * self.scaling_factors[’door’]

    # Calculate velocity constraint reward to ensure smooth door movement
    door_velocity = (door_hinge_angle[:, 1:, 0] - door_hinge_angle[:, :-1, 0]) / self.dt
    velocity_reward = -torch.norm(door_velocity, p=2, dim=1)

    # Scale velocity reward if not already scaled
    if ’velocity’ not in self.scaling_factors:
        with torch.no_grad():
            initial_velocity = -velocity_reward[0].item()
            self.scaling_factors[’velocity’] = 12.0 / (initial_velocity + 1e-8)
    velocity_reward = velocity_reward * self.scaling_factors[’velocity’]

    # Add dynamics consistency reward if model is provided
    dyn_reward = torch.zeros_like(reaching_reward)
    if dyn_model is not None:
        dyn_reward = self.cal_dyn_reward(state=normed_obs, action=normed_actions)
        # Scale dynamics reward if not already scaled
        if ’dynamics’ not in self.scaling_factors:
            with torch.no_grad():
                initial_dyn = dyn_reward[0].item()
                self.scaling_factors[’dynamics’] = 1.2 / (initial_dyn + 1e-8)
        dyn_reward = dyn_reward * self.scaling_factors[’dynamics’]

    # Combine rewards using the grasp mask
    # Pre-interaction phase: only reaching reward
    # Post-interaction phase: door reward + velocity reward + dynamics reward
    total_reward = (1 - grasp_mask[:, 0]) * reaching_reward + \
                   grasp_mask[:, 0] * (door_reward + velocity_reward + dyn_reward)

    return total_reward
```

E.2. Sample of Guidance Function on Hand Pen Task

```
def guidance_fn(self, normed_obs, normed_actions, dyn_model=None, without_normalizer=False, desired_pen=None):
    # Get batch size and horizon length
    batch_size, horizon, obs_dim = normed_obs.shape

    # Process target rotation
    target_rotation = desired_pen[..., -3:].reshape(batch_size, 1, 3).repeat(1, horizon, 1)

    # Initialize reward components
    reward = torch.zeros(batch_size, device=normed_obs.device)

    # Unnormalize observations if normalizer is provided
    if not without_normalizer:
        obs = self.unnormalize(normed_obs, is_obs=True)
    else:
        obs = normed_obs

    # Extract relevant states
    hand_joints = obs[..., :24]  # Hand joint positions
    pen_rotation = obs[..., 30:33]  # Pen rotation

    # 1. Orientation alignment reward using soft interpolation
    # Normalize vectors before computing similarity
    pen_rotation_norm = pen_rotation / (torch.norm(pen_rotation, p=2, dim=-1, keepdim=True) + 1e-6)
    target_rotation_norm = target_rotation / (torch.norm(target_rotation, p=2, dim=-1, keepdim=True) + 1e-6)
    # Compute similarity using dot product (higher means more aligned)
    orientation_similarity = torch.sum(pen_rotation_norm * target_rotation_norm, dim=-1)
    orientation_reward = torch.mean(orientation_similarity, dim=1)

    # Initialize scaling factor for orientation reward if not exists
    if ’orientation’ not in self.scaling_factors:
        self.scaling_factors[’orientation’] = 1.0 / (orientation_reward[0].abs().item() + 1e-6)

    reward = reward + self.scaling_factors[’orientation’] * orientation_reward

    # 2. Hand joint movement smoothness reward
    # Calculate joint position differences between consecutive timesteps
    joint_diffs = hand_joints[:, 1:] - hand_joints[:, :-1]
    smoothness_penalty = torch.norm(joint_diffs, p=2, dim=-1)  # Shape: (batch_size, horizon-1)
    smoothness_reward = -torch.mean(smoothness_penalty, dim=1)  # Average over horizon

    # Initialize scaling factor for smoothness reward if not exists
    if ’smoothness’ not in self.scaling_factors:
        self.scaling_factors[’smoothness’] = 1.0 / (smoothness_reward[0].abs().item() + 1e-6)

    reward = reward + self.scaling_factors[’smoothness’] * smoothness_reward

    # 3. Dynamic consistency reward (if model provided)
    if dyn_model is not None:
        dyn_reward = self.cal_dyn_reward(state=normed_obs, action=normed_actions)

        # Initialize scaling factor for dynamics reward if not exists
        if ’dynamics’ not in self.scaling_factors:
            self.scaling_factors[’dynamics’] = 2.0 / (dyn_reward[0].abs().item() + 1e-6)

        reward = reward + self.scaling_factors[’dynamics’] * dyn_reward

    return reward
```

E.3. Sample of Guidance Function on Hand Hammer Task

```
def guidance_fn(self, normed_obs, normed_actions, dyn_model=None, without_normalizer=False, tool_pos=None):
    """
    Guidance function for hammer-nail task with Adroit hand.
    Args:
        normed_obs: Normalized observations, shape (B, H, obs_dim)
        normed_actions: Normalized actions, shape (B, H, act_dim)
        dyn_model: Optional dynamics model for consistency checking
        without_normalizer: Boolean indicating if normalization should be skipped
    Returns:
        reward: Total reward tensor of shape (B,)
    """
    batch_size = normed_obs.shape[0]
    horizon_len = normed_obs.shape[1]
    device = normed_obs.device

    # Get unnormalized observations if normalizer is provided
    obs = normed_obs if without_normalizer else self.unnormalize(normed_obs, is_obs=True)

    # Extract relevant observations across all timesteps
    palm_pos = obs[:, :, 33:36]  # Hand palm position
    hammer_pos = obs[:, :, 36:39]  # Hammer position
    nail_pos = obs[:, :, 42:45]  # Nail position
    nail_insertion = obs[:, :, 26]  # Nail insertion depth, keep dim for proper broadcasting
    tool_pos = tool_pos[:, None, :].repeat(1, horizon_len, 1)

    # Calculate grasp mask based on distance between palm and hammer
    # Use first timestep to determine if hand has grasped hammer
    grasp_threshold = 0.1
    grasp_mask = torch.norm(palm_pos[:, 0, :] - hammer_pos[:, 0, :], p=2, dim=1) < grasp_threshold

    # Initialize total reward
    total_reward = torch.zeros(batch_size, device=device)

    # Phase 1: Pre-interaction guidance (hand approaching hammer)
    pre_grasp_reward = -torch.mean(
        torch.norm(palm_pos - hammer_pos, p=2, dim=2),
        dim=1
    )

    # Adaptive scaling for pre-grasp reward
    if ’pre_grasp’ not in self.scaling_factors:
        self.scaling_factors[’pre_grasp’] = 6.0 / (torch.abs(pre_grasp_reward[0]) + 1e-6)

    total_reward = total_reward + self.scaling_factors[’pre_grasp’] * pre_grasp_reward

    # Phase 2: Post-interaction guidance (hammer control and nail insertion)
    # Only apply if hand has grasped hammer
    if torch.any(grasp_mask):
        contact_mask = torch.norm(tool_pos - nail_pos, p=2, dim=2) < 0.1
        # Target nail insertion (halfway = 0.04m)
        target_insertion = 0.04 * torch.ones_like(nail_insertion)
        insertion_reward = \
            -torch.norm(nail_insertion - target_insertion, p=2, dim=1) #* contact_mask[:, 0]

        # Adaptive scaling for insertion reward
        if ’insertion’ not in self.scaling_factors:
            self.scaling_factors[’insertion’] = 6.0 / (torch.abs(insertion_reward[0]) + 1e-6)

        # Constraint on hammer position changes (smooth movement)
        hammer_joint_pos_changes = torch.norm(
            obs[:, 1:, 27:33] - obs[:, :-1, 27:33],
            p=2, dim=2
        )
        hammer_joint_reward = -torch.mean(hammer_joint_pos_changes, dim=1)

        # Adaptive scaling for nail movement constraint
        if ’hammer_joint’ not in self.scaling_factors:
            self.scaling_factors[’hammer_joint’] = 6.0 / (torch.abs(hammer_joint_reward[0]) + 1e-6)

        # Constraint on hammer position changes (smooth movement)
        hammer_pos_changes = torch.norm(
            hammer_pos[:, 1:, :] - hammer_pos[:, :-1, :],
            p=2, dim=2
        )
        hammer_movement_reward = -torch.mean(hammer_pos_changes, dim=1)

        # Adaptive scaling for hammer movement constraint
        if ’hammer_movement’ not in self.scaling_factors:
            self.scaling_factors[’hammer_movement’] = 12.0 / (torch.abs(hammer_movement_reward[0]) + 1e-6)  #
     100.

        # Add dynamics consistency reward if model provided
        if dyn_model is not None:
            dyn_reward = -self.cal_dyn_reward(state=normed_obs, action=normed_actions)

            # Adaptive scaling for dynamics reward
            if ’dynamics’ not in self.scaling_factors:
                self.scaling_factors[’dynamics’] = 0.3 / (torch.abs(dyn_reward[0]) + 1e-6)

            # Apply dynamics reward only to grasped trajectories
            total_reward = total_reward + self.scaling_factors[’dynamics’] * dyn_reward * grasp_mask.float()

        # Add all Phase 2 rewards
        phase2_reward = (self.scaling_factors[’insertion’] * insertion_reward +
                        self.scaling_factors[’hammer_joint’] * hammer_joint_reward +
                         self.scaling_factors[’hammer_movement’] * hammer_movement_reward)

        # Apply Phase 2 rewards only to grasped trajectories
        total_reward = total_reward + phase2_reward * grasp_mask.float()

    return total_reward
```