# DexHandDiff: 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 [15, 28], we can establish:

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

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 $Z$:*

$$p_{\theta,\phi}(\boldsymbol{\tau}^i \mid \boldsymbol{\tau}^{i+1},\boldsymbol{y}) = Z p_\theta(\boldsymbol{\tau}^i \mid \boldsymbol{\tau}^{i+1}) p_\phi(\boldsymbol{y} \mid \boldsymbol{\tau}^i). \tag{18}$$

*Proof.* By applying Bayes' theorem:

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

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:*

$$p_{\theta,\phi}(\boldsymbol{\tau}^i|\boldsymbol{\tau}^{i+1},\boldsymbol{y}) \approx \mathcal{N}(\boldsymbol{\tau}^i; \mu_\theta + \Sigma\nabla_{\boldsymbol{\tau}} \log p_\phi\left(\boldsymbol{y} \mid \boldsymbol{\tau}^i\right), \Sigma), \tag{19}$$

*where $\mu_\theta$ and $\Sigma$ 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 $\boldsymbol{\tau}^i = \mu_\theta$:

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

Applying the logarithm to both sides of Eq. 18:

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

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. Environment Settings

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. And the object relocation task requires to grasp the ball first and then move to the desired position.

## C. More Visualizations

Different from concurrent work [50] that focuses on grasping tasks, we conduct experiments on challenging dexterous manipulation benchmarks including door, pen, hammer, and block manipulation tasks, which require sophisticated contact-rich interactions and precise goal-directed control.

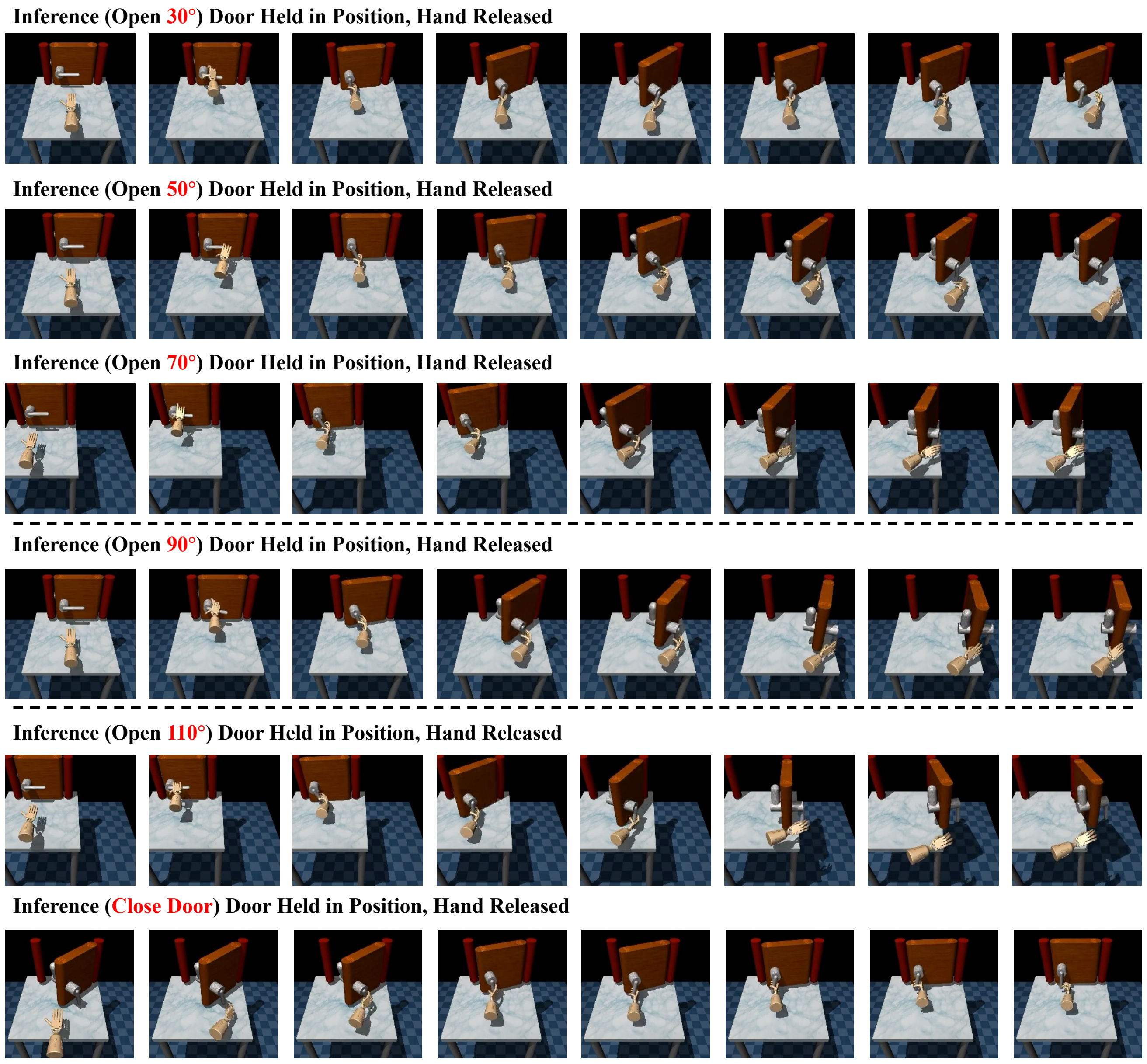
Inference (Open 30°) Door Held in Position, Hand Released
Inference (Open 50°) Door Held in Position, Hand Released
Inference (Open 70°) Door Held in Position, Hand Released
Inference (Open 90°) Door Held in Position, Hand Released
Inference (Open 110°) Door Held in Position, Hand Released
Inference (Close Door) Door Held in Position, Hand Released

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

## C.1. Goal Adaptive Door Tasks

We present detailed visualizations of DexHandDiff'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, DexHandDiff 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 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 DexHandDiff's robust goal adaptation capabilities while maintaining physical realism in the manipulation process.

### C.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 (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.

## D. Implementation Details

We implement our framework following standard diffusion model settings [24] with several modifications:

**Network Architecture.** We adopt a temporal U-Net [43] architecture consisting of 6 residual blocks for noise prediction. Each block contains dual temporal convolutions with group normalization [53], followed by a Mish activation [53]. 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 [25] optimizer with a learning rate of $2 \times 10^{-4}$ and batch size 256, trained for $5 \times 10^5$ steps across all tasks. For both our method and the classifier-free baselines [1, 14], we predict the denoised trajectory $\boldsymbol{\tau}_0$ directly rather than the noise term $\epsilon$, 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 $\alpha$ 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.

## E. LLM-based Guidance Generation Prompts

### E.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.

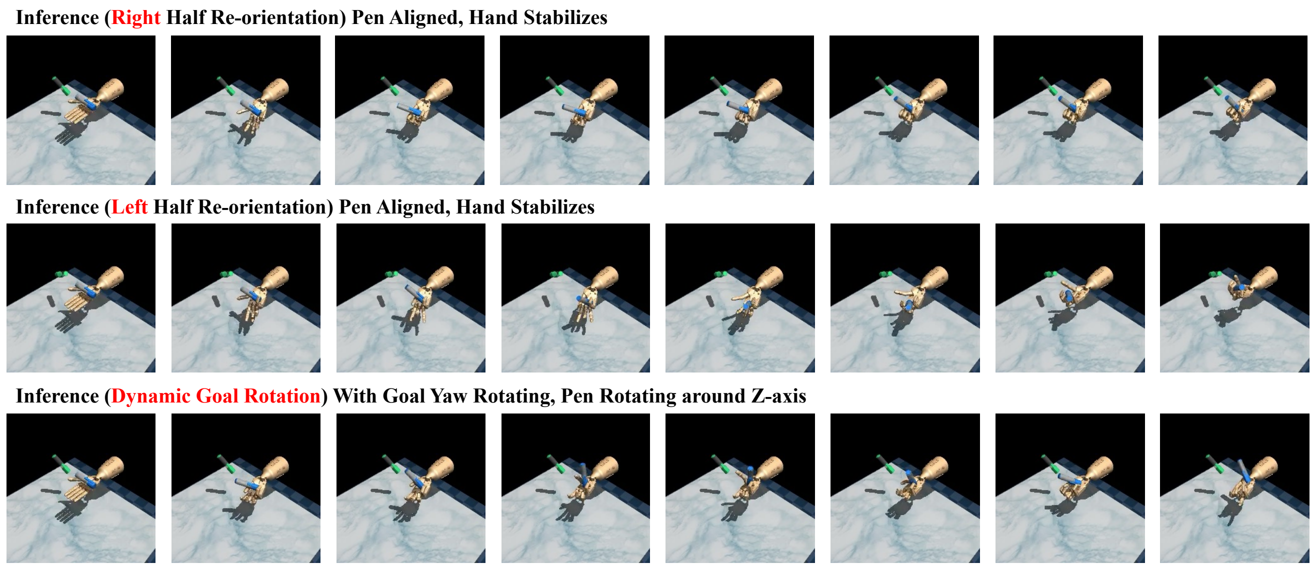
Inference (Right Half Re-orientation) Pen Aligned, Hand Stabilizes
Inference (Left Half Re-orientation) Pen Aligned, Hand Stabilizes
Inference (Dynamic Goal Rotation) With Goal Yaw Rotating, Pen Rotating around Z-axis

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 the 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.

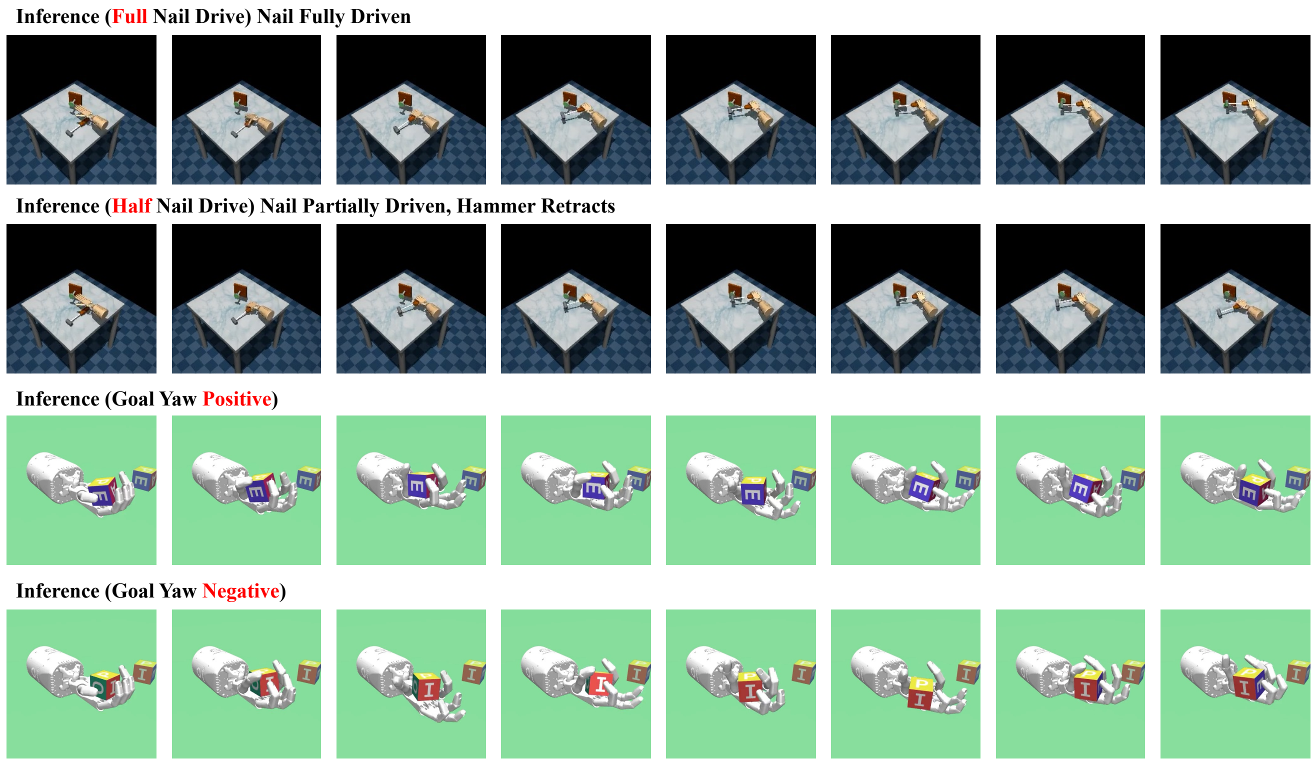
Inference (Full Nail Drive) Nail Fully Driven
Inference (Half Nail Drive) Nail Partially Driven, Hammer Retracts
Inference (Goal Yaw Positive)
Inference (Goal Yaw Negative)

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.

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

## E.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`.

## E.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.

## E.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.

## F. Samples of Generated Guidance Function

### F.1. Sample of Guidance Function on Hand Door Task

```python
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
```

### F.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
```

### F.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
```