Table 1: Controller Hyperparameters

	Parameter	PNGrid	PNRand	Franka
	Horizon (H)	32	64	32
	Temperature (β)	10^{-32}	10^{-32}	10^{-4}
MPPI	Init. cov. (σ^2)	10	100	0.1
	Step size (γ)	0.7	1	1
	Spline knots (n)	None	None	4
FlowMPPI	Init. cov. (σ^2)	10	10	0.1
	Latent cov.	1	1	0.1
	Latent mean penalty (λ)	10^{-4}	10^{-3}	1
NFMPC	Latent cov.	1	1	0.1

A Appendix

A.1 Computational Graph Visualization

We visualize the computational graph of a single episode in Figure 5. The normalizing flow (NF), shift operator (Shift), and loss computation (Loss) are colored blue, indicating that they are differentiable modules. The MPPI update (MPPI) is colored in purple, to indicate that we are approximating the gradient as above, which can be implemented as a custom backwards pass in autodifferentiation software. All of the paths of the graph through which the gradients flow during the backwards pass are colored in purple. The simulator (Sim), colored in green, is solely used to compute the weights for the MPPI update, which are reused in the backwards pass. At each time step, the updated mean $\boldsymbol{\mu}_t$ defines a latent Gaussian, which is used to generate controls applied to the actual environment (Env), denoted by the yellow box. Note that both the latent variables and controls are being passed to the MPPI module. This is because during the forward pass, we can simply take the weighted sum of latent variables. However, during the backward pass, we have to re-run the network with the controls to compute the approximate gradients, as discussed in Section 3.4.

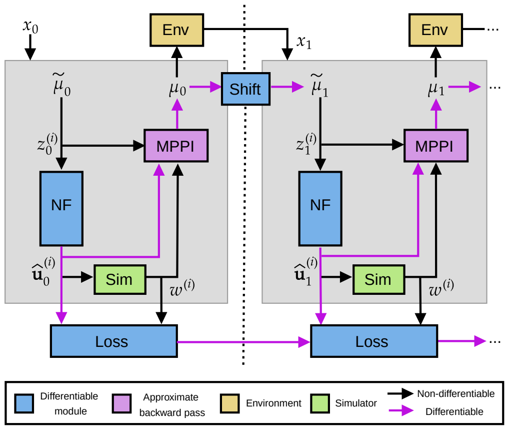

Figure 5: Computational graph of an episode which illustrates the interaction between the normalizing flow (NF), learned latent shift model (Shift), the MPPI update (MPPI) of the latent mean, the simulator (Sim), and the environment (Env).

A.2 Experimental Details

Controller Details. We use a modified version of the MPPI implementation by Bhardwaj et al. [4], which is implemented in PyTorch [28]. For the MPPI baseline, we perform covariance adaptation of the full covariance matrix across the horizon and control dimensions. The initial covariance matrix is always an identity matrix scaled by an initial covariance scalar hyperparameter σ^2. Additionally, this implementation uses Halton sequences [29] for generating control sequence samples and smooths the sampled trajectories with n-degree B-splines in some tasks. When B-splines are used, we sample the Halton sequence once at the beginning of a rollout and then transform it using the mean and covariance of the Gaussian distribution. Additionally, we ensure that the mean of the Gaussian is always in the set of samples. All hyperparameters of the controllers were chosen via a grid search and the final choices for each task are listed in Table 1. When it improves performance, we warm-start the controllers prior to the first time step by running the MPC update for 100 iterations to ensure convergence. Additionally, we normalize the total trajectory costs prior to computing the softmax weights, as discused by Okada and Taniguchi [10].

In general, we use the same settings for **NFMPC** and **FlowMPPI** that were found in this grid search. However, we do not performance covariance adaptation on the latent Gaussian and assume the flow learns how to adjust sample spread as needed. Additionally, we take the previous mean in the control space, shift it forward, and add it to the set of control samples at the next time steps. While the learned latent shift model handles this well in most cases, we found adding this sample sped up training and slightly improved performance. We always use Halton sequences to sample from the latent Gaussian, but never use B-splines. For **FlowMPPI**, we always use half the samples for sampling from the NF and half for the Gaussian perturbations on the current control-space mean. We do perform covariance adaptation on the perturbation Gaussian and re-tune its initial covariance. Finally, we have the additional λ parameter, which penalizes the latent Gaussian samples from deviating too much from the projection of the current control-space mean into the latent space.

Planar Robot Navigation. The planar navigation environment has a state space of $x_t \in \mathbb{R}^4$, which consists of the robot's 2D position, (p_x, p_y), and velocity, (v_x, v_y), and a control space of $u_t \in \mathbb{R}^2$, which are the robot's 2D acceleration commands. The robot has double-integrator dynamics with additive Gaussian noise on the controls, as described by the following equations:

$$\begin{bmatrix} p_x \\ p_y \\ v_x \\ v_y \end{bmatrix}_{t+1} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ v_x \\ v_y \end{bmatrix}_{t} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} (u_t + w_t), \quad w_t \sim \mathcal{N}(0, \sigma \mathrm{I}), \tag{15}$$

where we set $\Delta t = 0.1$ and $\sigma = 1$. Additionally, we added acceleration limits of $\underline{u} = -10$ and $\bar{u} = 10$ for both directions. The cost function consists of the Euclidean distance to the goal, a signed-distance field representation of the obstacles, and a term which encourages the robot to stay within the bounds of the map, and a quadratic control penalty:

$$c(x, u) = w_{goal}||x - x_g||_2^2 + w_{bound}c_{bound}(x_t) + w_{coll}\mathrm{SDF}(p_x, p_y) + w_{ctrl}||u||_2^2, \tag{16}$$

where we define the map bound cost as:

$$c_{bound}(x) = \sum_{i \in (x,y)} \mathbb{I}[(p_i > \bar{p}_i) \; || \; (p_i < \underline{p}_i)] \min\left((p_i - \bar{p}_i)^2, (p_i - \underline{p}_i)^2\right). \tag{17}$$

In the above equations, x_g is the goal state, $\bar{p}_i$ and $\underline{p}_i$ are the upper and lower bounds of the map for each coordinate, and SDF$(\cdot, \cdot)$ indexes an image which represents the signed-distance field. We mainly focus on the PNRandDyn task, which involves steering the robot towards a goal position while avoiding eight dynamic obstacles. While the obstacles drift randomly in the environment with Gaussian steps, their positions are clipped to be within map bounds. We do not update their position if the perturbation would bring it too close to the robot or goal location to prevent collisions which the robot cannot react to in time to avoid. An episode lasts for 200 time steps and is considered successful if the agent reaches the goal without colliding into any obstacles. In Appendix A.3, we also consider a static version of this environment (PNRand), which simply places the eight obstacles randomly in the environment, and a version which arranges the obstacles in a fixed grid (PNGrid).

Franka Panda Arm. The Franka Panda arm environment defines the robot state in joint space with $x_t \in \mathbb{R}^{21}$, consisting of each joint's angle θ_i, angular velocity $\dot{\theta}_i$, and angular acceleration $\ddot{\theta}_i$. Its control space is $u_t \in \mathbb{R}^7$, which are the angular acceleration commands for each joint. The dynamics are deterministic and implemented by the Nvidia Isaac Gym simulator [30]. However, the MPC controllers use a simpler kinematic model defined by Bhardwaj et al. [4], which is implemented in a batch fashion by leveraging its linearity:

$$\ddot{\boldsymbol{\Theta}} = \boldsymbol{u}, \quad \dot{\boldsymbol{\Theta}} = \dot{\Theta}_t + S_l(1)\mathrm{diag}(\Delta t)\ddot{\boldsymbol{\Theta}}, \quad \boldsymbol{\Theta} = \Theta_t + S_l(1)\mathrm{diag}(\Delta t)\dot{\boldsymbol{\Theta}}, \tag{18}$$

where the bold symbols indicate that they consist of values along the entire horizon, $\boldsymbol{\Theta}$ includes the angles for all joints, $S_l(1)$ is a lower triangular matrix filled with 1, and Δt is a vector of time steps across the horizon. We use smaller time steps earlier along the horizon and larger ones for later time steps. By implementing the dynamics in batch, we avoid iteratively unrolling the dynamics, speeding up controller computation significantly. For computing cost, we also require the Cartesian poses X, velocities $\dot{X}$, and accelerations $\ddot{X}$ of the end-effector. These are obtained via:

$$X = \mathrm{FK}(\Theta), \quad \dot{X} = J(\Theta)\dot{\Theta}, \quad \ddot{X} = \dot{J}(\Theta)\dot{\Theta} + J(\Theta)\ddot{\Theta} \tag{19}$$

where FK(Θ) are the forward kinematics and $J(\Theta)$ is the kinematic Jacobian. The cost function is a weighted sum of a number of terms, as defined by Bhardwaj et al. [4], which includes: distance of end-effector to the goal pose c_{pose}, a time varying velocity limit cost c_{stop} that enables stopping within the specified horizon, a joint limit cost c_{joint}, a manipulability cost c_{manip} which encourages the arm to avoid singular configurations, a self-collision cost $c_{self-coll}$, and a obstacle collision cost c_{coll}. The overall final cost function is then:

$$c(x,u) = w_p c_{pose}(x) + w_s c_{stop}(x) + w_j c_{joint}(x) + w_m c_{manip}(x) + w_c(c_{self-coll}(x) + c_{coll}(x)). \quad (20)$$

The self-collision cost is implemented with a neural network that predicts the closest distance between the links of the robot given a configuration. The collision cost is a binary cost which uses a learned collision checking function that operates directly on raw point cloud data and classifies if a robot link is in collision. See Bhardwaj et al. [4] for further details about each of the individual cost terms. For the FRANKAOBSTACLES task, we control the 7 degree-of-freedom (DOF) Franka Panda robot arm and steer it towards a target goal from a fixed starting pose while avoiding a single pole obstacle. The obstacle and goal positions are randomized at the beginning of each episode, which lasts for 600 time steps. An episode is considered successful if the end effector reaches the target position under the time constraints while avoiding the obstacle. In Appendix A.4, we also consider a simplified version which has no obstacles (FRANKA).

Architectural and Training Details. All NFs for both **NFMPC** and **FlowMPPI** were implemented in PyTorch and contain affine coupling layers which use multilayer perceptrons (MLPs) for both the scale and translation terms. The scale and translation networks use Tanh and ReLU activations, respectively. We also employ layer normalization [31] in these networks to help prevent overfitting. Interestingly, we found that adding batch normalization between each layer, as proposed by Dinh et al. [18], actually hurt performance and was therefore excluded. In all environments, we use 5 RealNVP blocks for the NF, and each MLP has a hidden dimensionality of 128 neurons. For the PNGRID, FRANKA, and FRANKAOBSTACLES tasks, the shift model is also an MLP with a single hidden layer of 128 neurons and a ReLU activation function. In PNRAND and PNRANDDYN the shift model is implemented as an LSTM with a hidden dimensionality of 128 neurons. For a fair comparison, the same architecture was used for both **FlowMPPI** and **NFMPC**. We trained all networks with the Adam optimizer [32] using a learning rate of 10^{-4}.

To train the **NFMPC** variants, we following the training procedure in Algorithm 1, which is carried out training over D episodes. In each episode d, we sample an environment from $\mathcal{C}$ and initial state from ρ. We then perform our rollouts by sampling latent controls $\hat{\boldsymbol{z}}_t^{(1:N)}$, passing them through the normalizing flow to get controls $\hat{\boldsymbol{u}}_t^{(1:N)}$, and then applying them to our approximate dynamics model and cost function to get weights $w_t^{(1:N)}$. These variables are used to update the latent distribution of the policy to θ_t. Next, we can either sample a control from the policy or use a control corresponding to the latent mean. We apply this control to the true system and shift the latent distribution parameters forward with the shift model. Finally, we compute the loss for the current time step, accumulate the loss to our running sum, and repeat for T time steps. Once the episode is complete, we update λ_d using the gradient of the loss and carry on to the next episode. Every 100 environments, we test the controller on 10 held out environments and save the current model if it outperforms the previous best on this validation set. While this introduces a variable number of total episodes used to train the models, we generally find convergence between 2000 and 7000 episodes.

Algorithm 1: Training Loop

```
Input: Environment dist. $\mathcal{C}$, initial state dist. $\rho$,
       initial param. $\tilde{\theta}_0$, $\lambda_0$
Parameters: # episodes $D$, episode length $T$, #
            samples $N$
for $d = 1, 2, \ldots, D$ do
    Sample environment $c \sim \mathcal{C}(\cdot)$
    Sample initial state $x_0 \sim \rho(\cdot|c)$
    Initialize episode loss $l_d \leftarrow 0$
    for $t = 0, 1, \ldots, T_k - 1$ do
        $(\hat{\boldsymbol{u}}_t, \hat{\boldsymbol{z}}_t, w_t)^{(1:N)} \leftarrow$ Rollout($x_t$, $c$, $\tilde{\theta}_t$, $\lambda_d$)
        Update $\tilde{\theta}_t$ to $\theta_t$ using latent MPPI update
        Sample $u_t \sim \pi_{\theta_t,\lambda_d}$ or $u_t \leftarrow h^{-1}_{\lambda_d}(\mu_t; c)$
        Apply control to system $x_{t+1} \sim f(x_t, u_t)$
        Shift parameters $\tilde{\theta}_{t+1} \leftarrow \Phi_{\lambda_d}(\theta_t, c)$
        Compute loss $\hat{J}_t(\theta_t, \lambda_d; x_t, c)$
        Accumulate loss $\ell_d \leftarrow \ell_d + \hat{J}_t$
    end
    Update $\lambda_d$ to $\lambda_{d+1}$ with $\nabla_\lambda \ell_d$ using SGD
end
```

In contrast, to train **FlowMPPI**, we do not actually run an episode. Instead, we generate the environment and take a gradient step on the initial distribution of the flow conditioned on the

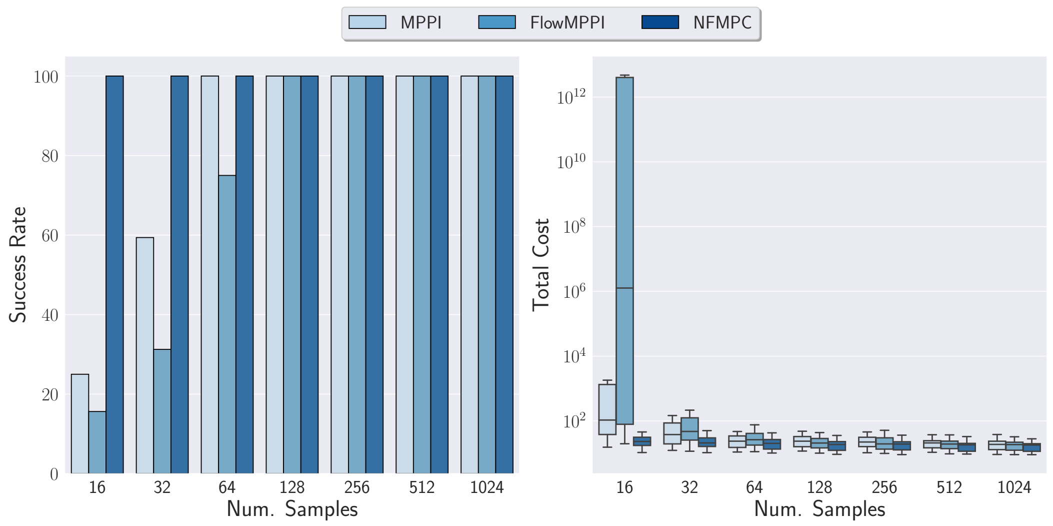

Figure 6: Success rate and cost distribution on the PNGRID task across a different number of samples.

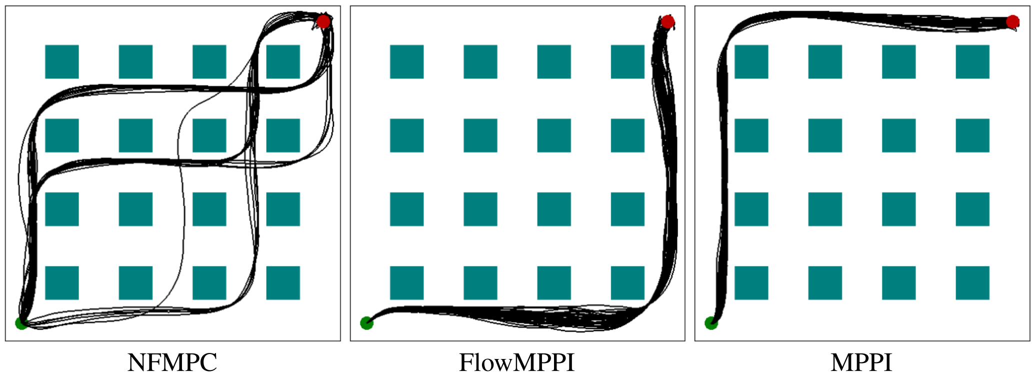

Figure 7: Visualization of trajectories in the PNGRID task across multiple random seeds for a fixed environmental layout.

environmental information. We train **FlowMPPI** over 10000 randomly generated environments for each task. We achieved the best performance by initializing the flow with the **FlowMPPI** solution, and then refining the flow and learning the shift model jointly as above. When we condition the NF on additional information, this is simply appended to the current input of each shift and translation network directly. This conditional information includes start and goal locations for PNGRID and FRANKA and start, goal, and obstacle locations for PNRAND and FRANKAOBSTACLES. For PNRANDDYN we use the current state instead of the initial location, which was found to improve performance for all controllers. For the obstacles, the conditional information is the Cartesian coordinates of each object of interest stacked together in a vector.

A.3 Additional Planar Navigation Experiments

PNGRID. We consider a variant of the planar navigation task that involves static obstacles arranged in a grid (PNGRID), described in Appendix A.2. All other task details are the same as in the PNRANDDYN task, except that we use an unconditional **NFMPC** controller. We quantitatively compare all controllers in Figure 6 and find that **NFMPC** consistently matches or outperforms both **MPPI** and **FlowMPPI** at each sample quantity in terms of both success rate and average trajectory cost of successful trajectories. We also find that **NFMPC** scales more gracefully than **MPPI** as the number of samples is reduced. In fact, we found that while **FlowMPPI** improves over **MPPI** at higher sample counts, it actually performs significantly worse with fewer samples. This is in contrast to the results on more complex environments considered in the main paper, in which **FlowMPPI** generally outperforms **MPPI** at lower sample counts as well. This is potentially because in the standard implementation of **FlowMPPI**, half of the samples come from the NF and the other half

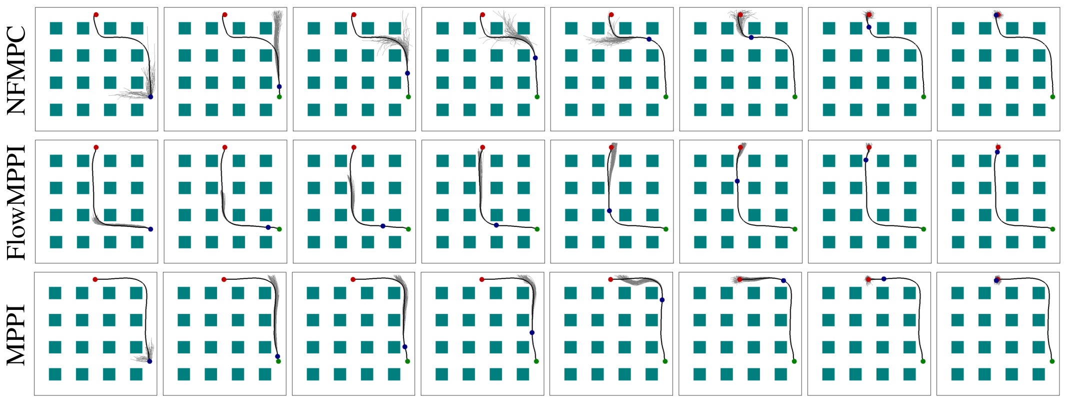

Figure 8: Visualization of a trajectory and top samples from (top) **NFMPC**, (middle) **FlowMPPI**, and (bottom) **MPPI** on the PNGRID task.

are Gaussian perturbations of the current control-space mean. Initially, the samples coming from the NF provide a good initialization for the control distribution mean. However, as the latent Gaussian distribution used by the NF is never updated, half of our samples are always coming from this same distribution. When the environment or task is more complex, the conditioning information provided to the NF is enough to transform the samples in a useful way. However, in this simple environment, our hypothesis is that as the robot moves in the environment, these samples may cease to be as useful or informative. We then have to rely on the other half of samples coming from Gaussian perturbations of the control-space mean to do most of the work. As such, we effectively have half the budget of samples to work with than **MPPI** would, as the samples from the NF potentially do not provide much useful information. Meanwhile, because **NFMPC** is trained recurrently and updates are performed in the latent space, it can better exploit structure in the environment to transform samples.

To better understand what **NFMPC** is doing differently, we superimpose 32 different trajectories with fixed start and goal positions using each controller in Figure 7. We find that both **MPPI** and **FlowMPPI** always select the same path through the environment. Meanwhile, **NFMPC** is able to discover different paths through the environment, allowing it to better react to the stochastic perturbations that knock it off the current plan and improve performance. Additionally, we visualize the resulting trajectories and top samples drawn from the distributions for all controllers on one of the validation environments in Figure 8. As we would expect, the initial trajectory from **FlowMPPI** is better than those of the other two controllers, which basically lay in straight lines in front of the robot. However, **NFMPC** is able to discover a slightly faster route to the goal as it proceeds in the environment. The baseline **MPPI** controller takes a similar, but slightly longer, route to the goal. Additionally, it takes a longer time to ramp up its velocity compared to the other two controllers, contributing to its sub-optimal performance.

PNRAND. Next, we consider a variant of PNRANDDYN in which the eight obstacles are static (PNRAND). We consider the case where we condition the NF on obstacle locations, initial state, and goal position. However, it is important to note that we do not condition the shift model, as we found this consistently hurt performance. In Figure 9, we display the quantitative results and find that **NFMPC** performed about on par with **FlowMPPI**, which outperformed **MPPI**. Unlike on the PNGRID task, both **NFMPC** and **FlowMPPI** scale similarly with a reduction of samples and better than **MPPI**. This can be partly attributed to the fact that the obstacles are more spaced out and there are more ”holes” in the environment than in the grid. Therefore, it is easier to avoid collisions, possibly contributing to the higher success rates when the controller has access to fewer samples. Moreover, conditioning on the obstacle locations provides the NF more information, which can be exploited without updating the latent distribution.

We again visualize the trajectories and top samples for all controllers on a validation environment in Figure 10. First, we note that the samples in **FlowMPPI** appear to be better spread around in the environment to search for good paths towards the goal. Meanwhile, **NFMPC** seems to have all top samples directed in one direction. All models seem to find the same path, with **MPPI** oscillating more near the goal and reaching the goal more slowly than **NFMPC** and **FlowMPPI**. The similar performance of **NFMPC** and **FlowMPPI** can be partially attributed to the fact that all we may really need to succeed in this environment is a good initial trajectory which steers us

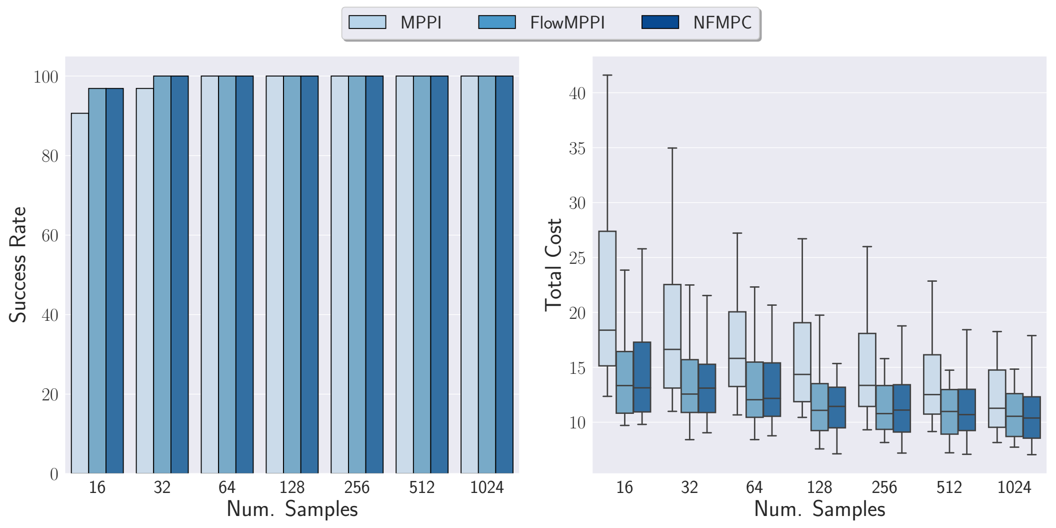

Figure 9: Success rate and cost distribution on the PNRAND environment across a different number of samples.

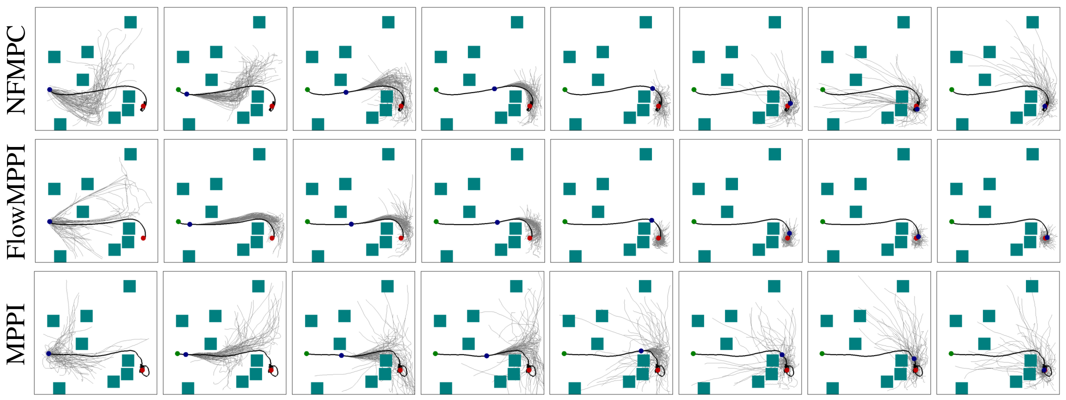

Figure 10: Visualization of a trajectory and top samples from (top) **NFMPC**, (middle) **FlowMPPI**, and (bottom) **MPPI** on the PNRAND task.

towards the goal. Whether we refine this trajectory with a learned latent shift model or with Gaussian perturbations in the control space does not appear to make much difference in performance. However, in PNRANDDYN, we demonstrated that there is a significant advantage to our approach, indicating that dynamic environments may be a good application for **NFMPC**.

Finally, in order to evaluate the benefit of conditioning the NF, we compare the performance of **NFMPC** with and without conditioning the flow on PNRAND in Figure 11. We find that the conditional model consistently outperforms the unconditional model in terms of median cost, with the gap growing at reduced sample counts. This may be because the dynamics in PNRAND are rather simple, and there may not be much general structure for the unconditional model to exploit since the obstacle locations are entirely random. Therefore, while the unconditional model does fairly well, conditioning the flow, and not the shift model, seems to enable further gains, even in this simple task.

PNRANDDYN. In addition to the experiments in the main paper, we also performed additional ablation studies. Specifically, we also considered training an unconditional model, as we did before in the PNRAND task. Moreover, we explored transferring controllers trained in the PNRAND task to this dynamic version of the environment. We plot the quantitative results from these ablation studies in Figure 12. Again, we find that conditional models consistently outperform unconditional ones, with the gap in performance more pronounced. Not only is there a reduction in median cost for unconditional controllers, but it sometimes fails in environments in which its conditional counterpart succeeds. Transferring the models trained in PNRAND works surprisingly well at 1024 samples. However, at lower sample counts, there is a more pronounced difference in the transferred controllers

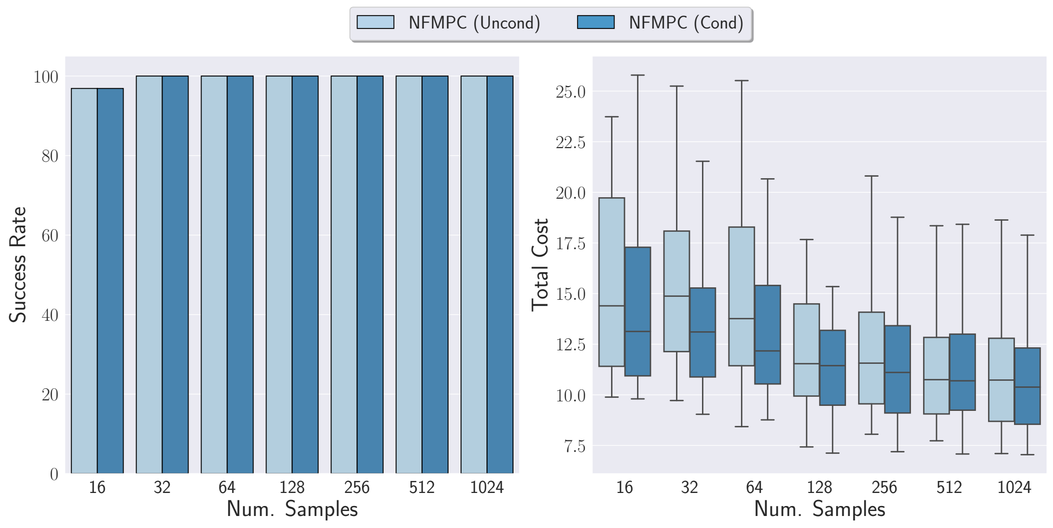

Figure 11: Comparison of unconditional and conditional models on the PNRAND environment across a different number of samples.

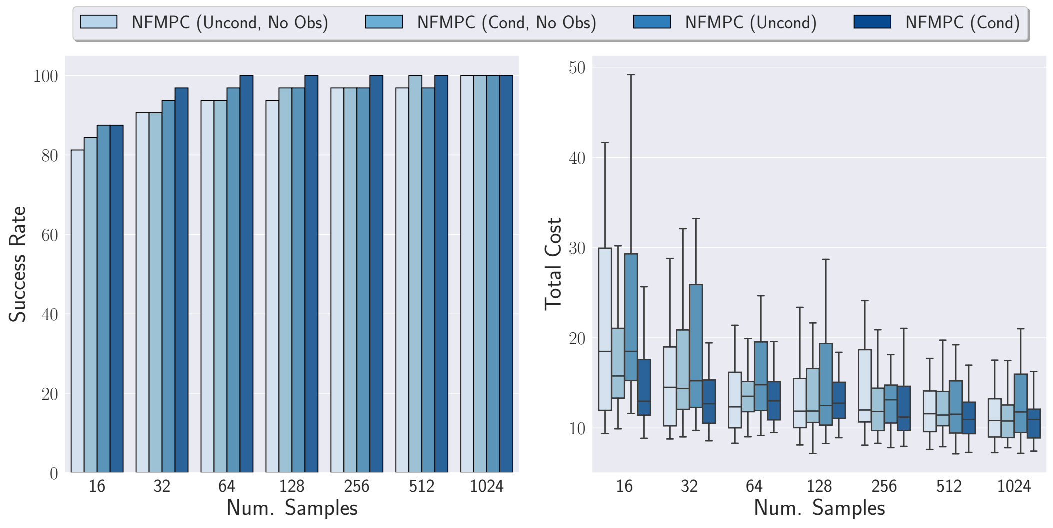

Figure 12: Comparison of unconditional and conditional models on the PNRANDDYN environment across a different number of samples when trained in an environment with no obstacles (PNRAND) and retrained with obstacles present.

to those specifically trained in the dynamic environment. Therefore, it appears that controllers are more efficient at exploring the environments they are trained on, and unsurprisingly, using more samples can partially overcome this gap.

A.4 Additional Franka Experiments

FRANKA. We consider a variant of the FRANKAOBSTACLES task which involves no obstacles, just a target goal, which we call FRANKA. We plot our quantitative results in Figure 14 and find that **NFMPC** again consistently matches or outperforms **MPPI** and **FlowMPPI** at each sample amount. In fact, **NFMPC** always achieves a 100% success rate at all sample counts, while both **MPPI** and **FlowMPPI** significantly drop in performance at lower amounts of samples. Moreover, **FlowMPPI** sometimes actually performed worse than **MPPI**, indicating that conditioning on just goal location does not help as much in this more complex scenario.

FRANKAOBSTACLES. In addition to the results in the main paper, we perform an additional ablation study in which we again compare an unconditional and conditional model on the FRANKAOBSTACLES environment. We also show the performance of transferring the unconditional model

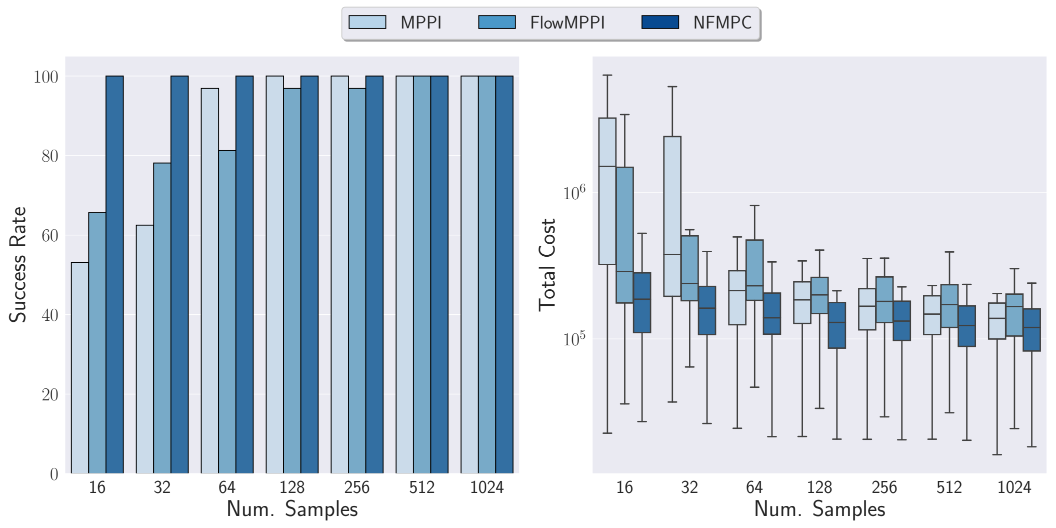

Figure 13: Success rate and cost distribution on the FRANKA environment across a different number of samples.

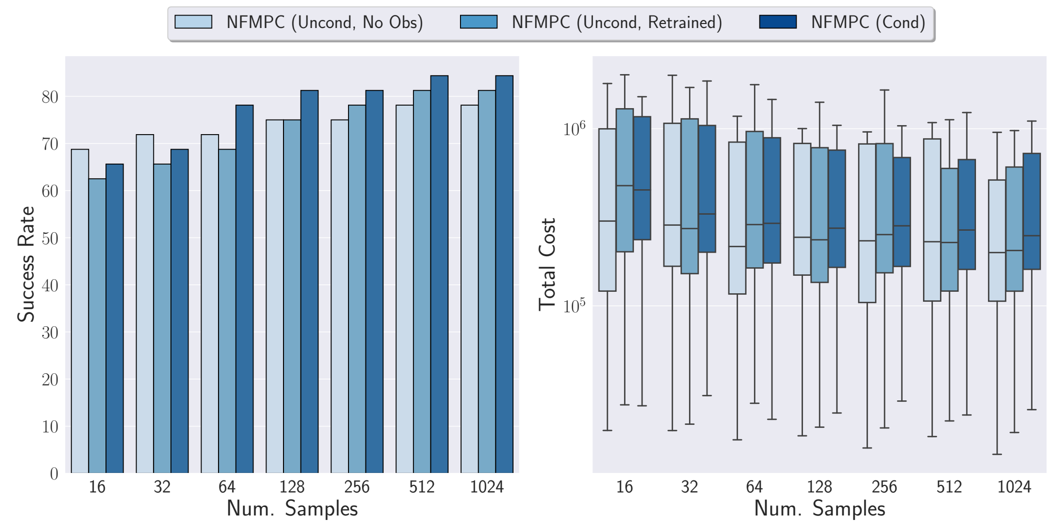

Figure 14: Comparison of an unconditional model trained with and without obstacles to a conditional model in the FRANKAOBSTACLES environment across a different number of samples.

trained on the FRANKA environment without obstacles to an environment which contains the single pole obstacle. At 1024 samples, the conditional model performs the best in terms of success rate. Surprisingly, the unconditional model trained without obstacles performs best in terms of median cost and scales better to fewer samples. Therefore, while the conditional model more often finds a feasible path to the goal, the unconditional model is better able to exploit structure across environments to find lower cost trajectories. One possible explanation is that because the unconditional model is trained without knowing the specific obstacle locations, it has to be more robust to variation in the environment. This also shows that transferring the learned distribution to novel environments is possible. However, since we only have a single static obstacle, it is not clear if these findings would generalize to more complex environments.

Breakdown of Trajectory Cost. We would like to better understand how the learned controllers improve upon the baseline on the FRANKAOBSTACLES task. As discussed in Appendix A.2, the cost function for the Franka tasks is composed of multiple terms. By inspecting the averages for each term, we hope to gain insight into the learned sampling distributions. Briefly, we consider a manipulability cost (Manip), which encourages the arm to avoid singular configurations, a self-collision cost (Self), an obstacle collision cost (Obstacle), and a distance-to-goal cost (Goal).

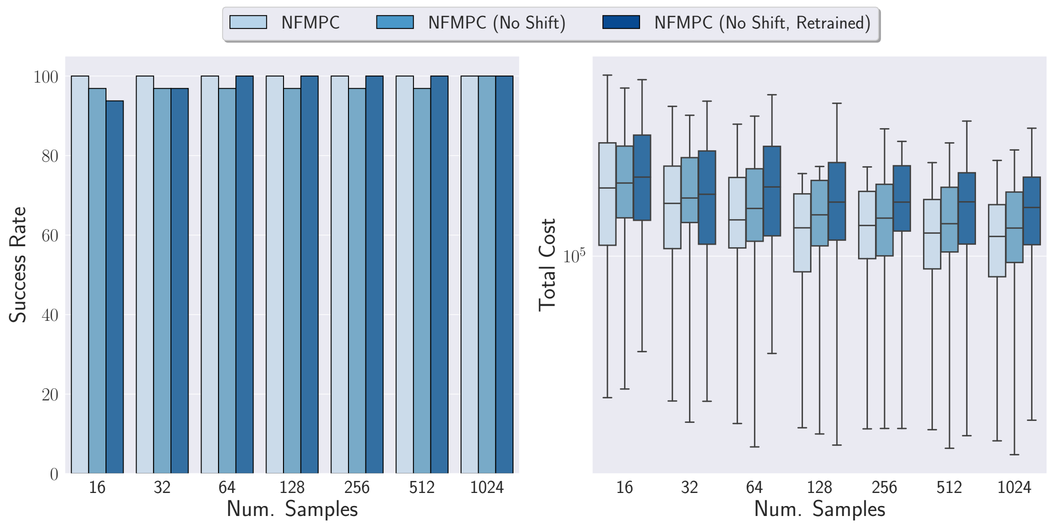

Figure 16: Success rate and cost distribution on the FRANKA task with (**NFMPC**) and without (**NFMPC (No Shift)** and **NFMPC (No Shift, Retrained**) the learned shift model.

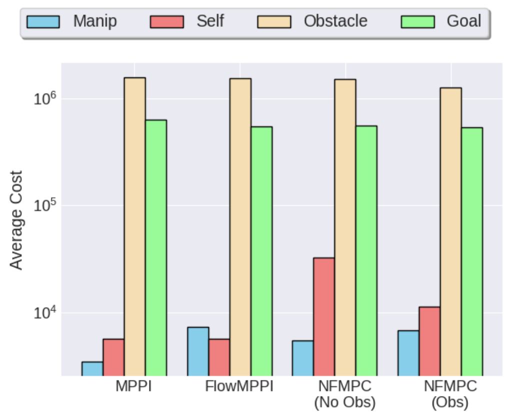

Figure 15: Breakdown of different cost terms for each controller executing the FRANKAOBSTACLES task.

The cost breakdown for these terms in shown in Figure 15. The baseline **MPPI** controller achieves the lowest manipulability cost. Since this term is not weighted as highly, it makes sense that training the normalizing flow would focus on minimizing terms which were more heavily weighted. Meanwhile, **FlowMPPI** achieves the lowest self-collision cost, which may be due to the fact that it starts off with a better initial plan. However, **NFMPC** trained with obstacles (**NFMPC (Obs)**) achieves the lowest obstacle collision and distance-to-goal cost. One possible explanation for this improvement is that, because we train **NFMPC** with BPTT, we are potentially able to account for errors which arise due to the inaccurate model. Additionally, learning the shift model may be an important component to this improvement, which we explore below. Finally, we note that **NFMPC (Obs)** is better than **NFMPC** trained without obstacles (**NFMPC (No Obs)**) in all cost terms except the manipulability cost, as it is in distribution for the task.

Shift Model Ablation. One of our main contributions which sets us apart from prior work is that we learn a latent shift model and train the controller as a recurrent network. To determine the impact of this choice, we consider an alternate scenario where the control sequence is instead shifted in control space. That is, we shift the mean in control space, as in the baseline **MPPI** controller, and then run the flow backwards to infer the corresponding latent mean, which is used to bootstrap the next iteration. We consider two cases: 1) taking a pre-trained controller and removing the shift model (**NFMPC (No Shift)**) and 2) retraining the controller entirely from scratch without the shift model (**NFMPC (No Shift, Retrained)**). In Figure 16, we show the results of this ablation study on the FRANKA task. Removing the shift model strictly hurts performance, even when we retrain the normalizing flow from scratch. Moreover, retraining the flow actually results in worse performance than simply removing the shift model from the pre-trained controller. This implies that training the entire controller with BPTT allows it to discover lower-cost trajectories to the goal.

A.5 Performance Overhead of Normalizing Flow

We measured the average change in wall clock time across different amounts of samples for **NFMPC** and **FlowMPPI** compared to the baseline **MPPI** implementation on our NVIDIA Titan V GPU. For the Planar Robot Navigation and Franka Panda Arm experiments, the average change is $1.61\times$ and $1.91\times$, respectively. Moreover, compared to **MPPI** with 1024 samples, the change in wall clock time for **NFMPC** and **FlowMPPI** with 16 samples is approximately $1.01\times$ and $1.14\times$, respectively. Therefore, the introduction of the normalizing flow (NF) has a notable impact on wall clock time for both methods. However, this overhead is expected and does not prohibit either method's utility in the real world. And there are still clear performance advantages of **NFMPC** over the baselines in terms of success rate and average trajectory cost for a given sample amount.

Furthermore, the performance of **MPPI** reported in the paper and the above timing comparisons are when the optimization is run for a single iteration per time step. However, the performance of **MPPI** generally improves with an increased number of iterations at the cost of an increased runtime. For instance, we compare the performance of **MPPI** run for 3 iterations per time step with **NFMPC** run for a single iteration, both using 1024 samples. In the FRANKA task, we can reduce the average trajectory cost of MPPI to be only 12% worse than **NFMPC**, rather than the previous 19%. When we introduce obstacles for the FRANKAOBSTACLEStask, **MPPI** with 3 iterations can actually match the success rate of **NFMPC**, albeit with a worse average trajectory cost. However, in this case, **NFMPC** actually results in an average reduction of wall clock time by 24.8%. Similarly, for the PNGRID task, **MPPI** with 3 iterations reduces the average trajectory cost to be only 10% worse than **NFMPC**, rather than the previous 40%. However, this again comes at the cost of increased runtime, as **NFMPC** reduces the average wall clock time by 26.8%. Therefore, **NFMPC** allows us to surpass the performance of **MPPI** run with more iterations while reducing the required runtime.

Additionally, **NFMPC** has substantial runtime benefits over **FlowMPPI** due to its improved scaling. For the PNGRIDtask, **NFMPC** with 128 samples outperforms **FlowMPPI** with 1024 samples while reducing average runtime by 22%. Similarly, for the FRANKAtask, **NFMPC** with 64 samples outperforms **FlowMPPI** with 1024 samples while reducing average runtime by 43%. And for both Franka tasks (FRANKA and FRANKAOBSTACLES), **NFMPC** with 16 samples outperforms **FlowMPPI** with 128 samples while reducing average runtime by 8%. As such, there are clear runtime benefits for **NFMPC** over both **FlowMPPI** and **MPPI** run for additional iterations. Finally, it is also important to note that we did not perform a hyperparameter sweep on the NF, and it may be possible to significantly reduce the size of the network while retaining performance benefits.

A.6 Proof for DMD Update of Latent Parameters

Theorem A.1. *Consider the optimization problem*

$$\theta_t = \arg\min_{\theta\in\Theta} \langle \gamma_t g_t, \theta\rangle + D_{KL}(\pi_{\theta,\lambda}||\pi_{\tilde{\theta}_t,\lambda}) \tag{21}$$

where we define

$$\pi_{\theta,\lambda}(\hat{\boldsymbol{u}}|c) = p_\theta(h_\lambda(\hat{\boldsymbol{u}};c))\left|det\,\frac{\partial h_\lambda}{\partial \hat{\boldsymbol{u}}}\right| \tag{22}$$

and h_λ is an invertible, deterministic transformation, p_θ a Gaussian factorized as in Equation (4), and θ is the natural parameters of the Gaussian. Then the corresponding update to the mean of the latent Gaussian is given by:

$$\boldsymbol{\mu}_t = (1-\gamma_t^\mu)\tilde{\boldsymbol{\mu}}_t+\gamma_t^\mu \frac{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}h_\lambda(\hat{\boldsymbol{u}}_t;c)\right]}{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\right]} = (1-\gamma_t^\mu)\tilde{\boldsymbol{\mu}}_t+\gamma_t^\mu \frac{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c))}\hat{\boldsymbol{z}}_t\right]}{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c))}\right]} \tag{23}$$

Proof. First, we note that

$$g_t = \nabla\hat{J}(\tilde{\theta};x_t) = -\frac{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\nabla_{\tilde{\theta}}\log\pi_{\tilde{\theta},\lambda}(\hat{\boldsymbol{u}}_t)\right]}{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\right]}, \tag{24}$$

and that

$$\log\pi_{\tilde{\theta},\lambda}(\hat{\boldsymbol{u}}|c) = \log p_{\tilde{\theta}}(h_\lambda(\hat{\boldsymbol{u}};c)) + \log\left(\left|\det\frac{\partial h_\lambda}{\partial\hat{\boldsymbol{u}}}\right|\right). \tag{25}$$

Therefore, when computing the gradient of Equation (25) with respect to $\tilde{\theta}$, we can drop the log-determinant term as it does not depend on $\tilde{\theta}$. As such, we are left with the gradient of the latent

Gaussian with respect to its natural parameters, or $\nabla_{\tilde{\theta}} \log \pi_{\tilde{\theta},\lambda}(\hat{\boldsymbol{u}}|c) = \nabla_{\tilde{\theta}} \log p_{\tilde{\theta}}(h_\lambda(\hat{\boldsymbol{u}}; c))$. Taking this gradient of the log-likelihood term with respect to the natural parameters, we have:

$$g_t = -\frac{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\left(\phi(h_\lambda(\hat{\boldsymbol{u}}_t;c)) - \tilde{\phi}_t\right)\right]}{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\right]}, \tag{26}$$

where $\tilde{\phi}_t$ is the expectation parameter corresponding to natural parameter $\tilde{\theta}_t$ and $\phi(\cdot)$ is the sufficient statistics of the latent distribution. We can rewrite the expectations in terms of $p_{\tilde{\theta}}(\cdot)$:

$$g_t = -\frac{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c)))}\left(\phi(\hat{\boldsymbol{z}}_t) - \tilde{\phi}_t\right)\right]}{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c))}\right]}, \tag{27}$$

Next, looking at the KL divergence term, we can write:

$$\begin{aligned}
D_{KL}(\pi_{\theta,\lambda}||\pi_{\tilde{\theta}_t,\lambda}) &= \mathbb{E}_{\pi_{\theta,\lambda}}\left[\log\frac{\pi_{\theta,\lambda}(\hat{\boldsymbol{u}})}{\pi_{\tilde{\theta}_t,\lambda}(\hat{\boldsymbol{u}})}\right] \\
&= \mathbb{E}_{\pi_{\theta,\lambda}}\left[\log\frac{p_\theta(h_\lambda(\hat{\boldsymbol{u}};c))}{p_{\tilde{\theta}}(h_\lambda(\hat{\boldsymbol{u}};c))}\frac{\cancel{\left|\det\frac{\partial h_\lambda}{\partial\hat{\boldsymbol{u}}}\right|}}{\cancel{\left|\det\frac{\partial h_\lambda}{\partial\hat{\boldsymbol{u}}}\right|}}\right] \\
&= \mathbb{E}_{p_\theta}\left[\log\frac{p_\theta(\hat{\boldsymbol{z}})}{p_{\tilde{\theta}}(\hat{\boldsymbol{z}})}\right] \\
&= D_{KL}(p_\theta||p_{\tilde{\theta}_t}),
\end{aligned} \tag{28}$$

Then, using Proposition 1 from Wagener et al. [9], we can write the update rule as

$$\phi_t = (1-\gamma_t)\tilde{\phi}_t + \gamma_t\frac{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c)))}\phi(\hat{\boldsymbol{z}}_t)\right]}{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c))}\right]} \tag{29}$$

And when the sufficient statistic is $\phi(\hat{\boldsymbol{z}}_t) = (\hat{\boldsymbol{z}}_t, \hat{\boldsymbol{z}}_t\hat{\boldsymbol{z}}_t^T)$, then we arrive at the usual MPPI update for the mean, but now defined in terms of the latent samples:

$$\boldsymbol{\mu}_t = (1-\gamma_t^\mu)\tilde{\boldsymbol{\mu}}_t + \gamma_t^\mu\frac{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c))}\hat{\boldsymbol{z}}_t\right]}{\mathbb{E}_{p_{\tilde{\theta}},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,h_\lambda^{-1}(\hat{\boldsymbol{z}}_t;c))}\right]} \tag{30}$$

which we can equivalently rewrite in terms of $\pi_{\tilde{\theta},\lambda}$ as:

$$\boldsymbol{\mu}_t = (1-\gamma_t^\mu)\tilde{\boldsymbol{\mu}}_t + \gamma_t^\mu\frac{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}h_\lambda(\hat{\boldsymbol{u}}_t;c)\right]}{\mathbb{E}_{\pi_{\tilde{\theta},\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\right]} \tag{31}$$

□

A.7 Proof of Approximate Gradient Through MPPI Update

Theorem A.2. *Let the MPPI update of the latent mean be given by*

$$\boldsymbol{\mu}_t(\lambda) = (1-\gamma_t^\mu)\tilde{\boldsymbol{\mu}}_t(\lambda) + \gamma_t^\mu\Delta\boldsymbol{\mu}_t, \qquad \Delta\boldsymbol{\mu}_t = \frac{\mathbb{E}_{\pi_{\tilde{\theta}_t(\lambda),\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}h_\lambda(\hat{\boldsymbol{u}}_t;c)\right]}{\mathbb{E}_{\pi_{\tilde{\theta}_t(\lambda),\lambda},\hat{f}}\left[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\right]}. \tag{32}$$

Then the gradient of $\Delta\boldsymbol{\mu}_t$ with respect to λ can be approximated as

$$\frac{\partial\Delta\boldsymbol{\mu}_t}{\partial\lambda} \approx M_1 - M_2M_3 \tag{33}$$

where

$$M_1 = \sum_{i=1}^{N} w_i \Big[\nabla_\lambda h_\lambda(\hat{\boldsymbol{u}}_t^{(i)}; c) + h_\lambda(\hat{\boldsymbol{u}}_t^{(i)}; c)\nabla_\lambda \log \pi_{\tilde{\theta}(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)}|c)\Big],$$
$$M_2 = \sum_{i=1}^{N} w_i h_\lambda(\hat{\boldsymbol{u}}_t^{(i)}; c), \quad M_3 = \sum_{i=1}^{N} w_i \nabla_\lambda \log \pi_{\tilde{\theta}(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)}|c). \tag{34}$$

Proof. First, we rewrite $\Delta\boldsymbol{\mu}_t = \frac{N(\lambda)}{D(\lambda)}$, where

$$N(\lambda) = \mathbb{E}_{\pi_{\tilde{\theta}_t(\lambda),\lambda},\hat{f}}\Big[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)} h_\lambda(\hat{\boldsymbol{u}}_t; c)\Big] \qquad D(\lambda) = \mathbb{E}_{\pi_{\tilde{\theta}_t(\lambda),\lambda},\hat{f}}\Big[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\Big]. \tag{35}$$

Then by the quotient rule of calculus, we have

$$\frac{\partial \Delta\boldsymbol{\mu}_t}{\partial\lambda} = \frac{\nabla_\lambda N(\lambda)}{D(\lambda)} - \frac{N(\lambda)}{D(\lambda)}\frac{\nabla_\lambda D(\lambda)}{D(\lambda)}. \tag{36}$$

We can compute each of these individual terms using the likelihood ratio gradients

$$\nabla_\lambda N(\lambda) = \mathbb{E}_{\pi_{\tilde{\theta}_t(\lambda),\lambda},\hat{f}}\Big[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\Big(\nabla_\lambda h_\lambda(\hat{\boldsymbol{u}}_t; c) + h_\lambda(\hat{\boldsymbol{u}}_t; c)\nabla_\lambda \log \pi_{\tilde{\theta}(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)}|c)\Big)\Big] \tag{37}$$

$$\nabla_\lambda D(\lambda) = \mathbb{E}_{\pi_{\tilde{\theta}_t(\lambda),\lambda},\hat{f}}\Big[e^{-\frac{1}{\beta}C(\hat{\boldsymbol{x}}_t,\hat{\boldsymbol{u}}_t)}\nabla_\lambda \log \pi_{\tilde{\theta}(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)}|c)\Big] \tag{38}$$

Since each of the terms that make up our gradient in Equation (36) are divided by $D(\lambda)$, when we approximate them with Monte Carlo sampling, we can actually write them in terms of the same weights used by MPPI in the forward pass:

$$\frac{\nabla_\lambda N(\lambda)}{D(\lambda)} = \sum_{i=1}^{N} w_i \Big[\nabla_\lambda h_\lambda(\hat{\boldsymbol{u}}_t^{(i)}; c) + h_\lambda(\hat{\boldsymbol{u}}_t^{(i)}; c)\nabla_\lambda \log \pi_{\tilde{\theta}(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)}|c)\Big] = M_1 \tag{39a}$$

$$\frac{N(\lambda)}{D(\lambda)} = \sum_{i=1}^{N} w_i h_\lambda(\hat{\boldsymbol{u}}_t^{(i)}; c) = M_2 \tag{39b}$$

$$\frac{\nabla_\lambda D(\lambda)}{D(\lambda)} = \sum_{i=1}^{N} w_i \nabla_\lambda \log \pi_{\tilde{\theta}(\lambda),\lambda}(\hat{\boldsymbol{u}}_t^{(i)}|c) = M_3 \tag{39c}$$

□

A.8 Sigmoid Flow Layer

We wish to use a sigmoid layer in our normalizing flow to constrain our control sample $\hat{\boldsymbol{u}}$ such that each control along the horizon is between $\underline{u}$ and $\bar{u}$. Since the sigmoid function is invertible, if we append a sigmoid layer at the end of our flow, we have that

$$\hat{\boldsymbol{u}} = w\sigma(\hat{\boldsymbol{y}}_{K-1}) + b \quad \Longleftrightarrow \quad \hat{\boldsymbol{y}}_{K-1} = \sigma^{-1}\Big(\frac{\hat{\boldsymbol{u}} - b}{w}\Big) = \log\Big(\frac{\hat{\boldsymbol{u}} - b}{w - \hat{\boldsymbol{u}} + b}\Big), \tag{40}$$

where $w = \bar{u} - \underline{u}$ and $b = \underline{u}$, the sigmoid and logit functions are applied element-wise, and the scaling and translation are broadcasted to each element of the vector $\hat{\boldsymbol{u}}$. The derivative of the forward transformation is given by:

$$\begin{aligned}\frac{\partial}{\partial x}\Big(w\sigma(x) + b\Big) &= w\sigma(x)(1 - \sigma(x)) \\ &= \frac{w}{1 + e^{-x}}\Big(1 - \frac{1}{1 + e^{-x}}\Big) \\ &= \frac{w}{(1 + e^{-x})(1 + e^{x})}.\end{aligned} \tag{41}$$

Since the sigmoid is applied element-wise, it has a diagonal Jacobian, the log-determinant of which is simply the sum of the log of its diagonal terms:

$$\log\Big|\det\frac{\partial\hat{\boldsymbol{u}}}{\partial\hat{\boldsymbol{y}}_{K-1}}\Big| = \sum_{i=1}^{MH} \log(w) - \log(1 + e^{-\hat{u}_i}) - \log(1 + e^{\hat{u}_i}), \tag{42}$$

where we can implement the last two terms with the Softplus activation function. In the reverse direction, we have that:

$$\frac{\partial}{\partial x}\left(\sigma^{-1}\Big(\frac{x-b}{w}\Big)\right) = \frac{1}{x-b} + \frac{1}{w-x-b} \tag{43}$$

Therefore, the log-determinant is given by:

$$\log\left|\det \frac{\partial \hat{\boldsymbol{y}}_{K-1}}{\partial \hat{\boldsymbol{u}}}\right| = -\sum_{i=1}^{MH}\Big(\log(\hat{u}_i - b) + \log(w - \hat{u}_i - b)\Big), \tag{44}$$

As such, computing the inverse and the log-determinant terms of the Jacobian is efficient and fast in both directions, adding minimal overhead to the flow.