Learning Multi-Body Dynamics by Simulator-Augmented Interaction Networks

Jinjia Guo
jinjaguo@umich.edu
Department of Robotics
University of Michigan
Ann Arbor United States 48109

***Abstract*—This project proposes and implements a hybrid dynamics modeling method for the push-plate operation task. We use the SAIN (Simulator-Augmented Interaction Networks) architecture and introduce a neural network to model the residual based on the prediction of the physics engine Bullet to improve the accuracy of physical modeling. To train the model, we collected about 3,000 high-quality interaction data in an environment with random mass, radius, and friction coefficient. We completed the training of the residual network using a single-step supervision method. On this basis, we further constructed a SAIN-based MPPI (Model Predictive Path Integral) controller, which achieved a 100% task success rate in 40 evaluation environments with random physical properties, significantly better than the baseline method that only uses physics engine predictions. The experiment verified the effectiveness and generalization ability of the fusion of physical priors and data-driven methods in complex contact dynamics modeling and control.To learn more about the Project, visit Github Website**

I. Introduction

In many manipulation tasks, the robot cannot directly act on the target object, but needs to indirectly achieve the target operation with the help of an intermediary object. For example, in the task of pushing a plate, the robot does not directly contact the target plate, but first pushes an intermediate object (such as a large plate) to contact the target plate, and plans a series of actions so that the target plate finally reaches the specified position. This type of problem was first proposed by Ajay et al. [1]. Its essence is an indirect manipulation task with contact and dynamic complexity.

In such tasks, accurate dynamic modeling is crucial for planning and control. Usually, researchers use physics engines (such as Bullet or MuJoCo) to model and simulate the environment. However, contact behavior in reality is often highly uncertain and nonlinear, and the predictions of standard physics engines may produce significant deviations when faced with contact, friction, or changes in object properties. Therefore, in recent years, a class of methods has emerged, which is to learn the prediction error (residual) based on the physics engine and correct it through a neural network to build a simulator-augmented (data-enhanced) dynamic model.

This residual modeling strategy works because it is easier to learn only relatively small residuals than to learn the system dynamics from scratch, and it has higher sample efficiency and stronger generalization ability. As proposed by Kloss et al.[2], combining analytical models with learning models can achieve more stable and accurate prediction of physical behavior, which is better than pure data-driven or analytical methods.

However, existing residual modeling methods usually assume that the number of objects is fixed and the interaction mode is stable, so it is difficult to generalize to scenarios with multiple interacting bodies or changing structures. To this end, this paper introduces a structured dynamic modeling framework: Simulator-Augmented Interaction Network (SAIN). This method introduces physical priors on the basis of Interaction Network, and learns the residual dynamics between the physics engine and the real observation through neural networks, so as to achieve more accurate multi-body interaction modeling.

In this project, we built a push disk experimental environment involving two disk objects based on the Panda robot platform, systematically collected push disk data with changes in mass, radius and friction coefficient, and trained the SAIN model on this basis. In addition, we designed an MPPI controller based on the model and completed the task in 40 environments with different configurations of random physical properties with a success rate 100%, thus verifying the effectiveness and robustness of the proposed method.

II. Method

A. Overall Method

In robotic manipulation tasks, modeling system dynamics is crucial for efficient control and generalization. Existing methods can be broadly classified into four categories, as illustrated in Figure 1:

- **(a) Pure Physics Engine:** Predicts the next state s_{t+1} from the current state s_t and the action a_t purely using physical laws.

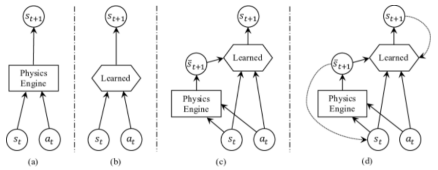

Fig. 1: Four types of dynamics models. (a) Pure physics engine; (b) Fully learned dynamics; (c) Residual learning over physics engine; (d) Recurrent residual learning (used in SAIN).

- **(b) Fully Learned Dynamics:** Uses neural networks to map $(s_t, a_t) \mapsto s_{t+1}$ directly without physical knowledge.
- **(c) Residual Learning:** First applies a physics engine to compute $\bar{s}_{t+1}$, then learns a correction f_θ such that $s_{t+1} = \bar{s}_{t+1} + f_\theta(s_t, a_t)$.
- **(d) Recurrent Residual Learning:** Similar to (c) but with a rollout using the network's own previous predictions, allowing long-term prediction.

In this project, we adopt the Simulator-Augmented Interaction Network (SAIN) framework (Figure 1d), which combines the interpretability of physics simulators with the expressiveness of neural networks.

Our pipeline consists of three main stages: first, We randomly sample object parameters such as mass, radius, and surface friction. For each pushing action, we record: Current state s_t, Action a_t, Real next state s_{t+1} and Physics prediction $\bar{s}_{t+1} = f_{\text{phys}}(s_t, a_t)$. This data set includes rich variations, which help to improve generalization.

Then, instead of predicting the absolute next state, we train the model to predict the residual dynamics (i.e., Δs) between the true state and the prediction of physics.We use the Interaction Network (IN) structure to capture pairwise interactions between objects. The output Δs is added to the current state using a differentiable update function to get the predicted s_{t+1}.

Lastly, the trained SAIN model is deployed within a Model Predictive Path Integral (MPPI) controller. Each sequence is rolled out for 15 steps using SAIN to predict outcomes. The best sequence (lowest cost) is selected, and the first action is executed. This planning strategy takes advantage of the model's ability to simulate long-horizon interaction under uncertainty of dynamics.

Here we show the pseudo-code of the pipeline in Algorithm 1

Algorithm 1 SAIN Training and Control Pipeline

Require: Physics engine f_{phys}, SAIN model f_θ
Require: Randomized environment generator Env()
Require: MPPI horizon H, number of samples K
Require: Training iterations T, batch size B

▷ **Phase I: Data Collection**

for $i = 1$ to N episodes **do**
  Sample object parameters: mass, radius, friction
  Initialize env ← Env.random_reset()
  for $t = 0$ to T_{episode} **do**
    Sample action $a_t \sim$ ActionSampler
    Record current state s_t
    Predict $\bar{s}_{t+1} \leftarrow f_{\text{phys}}(s_t, a_t)$
    Step env: $s_{t+1} \leftarrow$ Env.step(a_t)
    if $\|s_{t+1}^{\text{small}} - s_t^{\text{small}}\| > \epsilon$ **then**
      Store tuple $(s_t, a_t, \bar{s}_{t+1}, s_{t+1})$

▷ **Phase II: SAIN Model Training**

for $epoch = 1$ to T **do**
  for each mini-batch $(s, a, \bar{s}', s')$ of size B **do**
    Compute phys_delta $= \bar{s}' - s$
    Build object features and pairwise relation features
    Predict residual $\Delta\hat{s} = f_\theta(s, a, \bar{s}')$
    Compute predicted next state: $\hat{s}' = s + \Delta\hat{s}$
    Compute loss:
      $\mathcal{L} = \|p - \hat{p}\|^2 + \|v - \hat{v}\|^2 + \|\sin\theta - \sin\hat{\theta}\|^2 + \|\cos\theta - \cos\hat{\theta}\|^2$
    Update model parameters with Adam optimizer

▷ **Phase III: Model Predictive Control with SAIN**

for $episode = 1$ to N_{test} **do**
  Observe current state s_0
  for $t = 0$ to max steps **do**
    for $k = 1$ to K **do**
      Sample action sequence $\{a_0^k, a_1^k, \ldots, a_{H-1}^k\}$
      Simulate trajectory $\hat{s}_{0:H}^k \leftarrow$ rollout$(f_\theta, s_t, \{a^k\}, f_{\text{phys}})$
      Compute trajectory cost C_k
    Select best sequence $k^* = \arg\min C_k$
    Execute first action $a_0^{k^*}$
    Update current state $s_{t+1} \leftarrow$ Env.step$(a_0^{k^*})$
    if goal reached **then**
      break

B. Problem Definition

Let $\mathcal{S}$ denote the state space and $\mathcal{A}$ the action space. A dynamics model is a function $f : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, which

predicts the next state s_{t+1} given the current state $s_t \in \mathcal{S}$ and action $a_t \in \mathcal{A}$:

$$f(s_t, a_t) \approx s_{t+1} \tag{1}$$

There are two general classes of dynamics models:

- **Analytical models:** Based on first principles and physical equations (see Fig. 1a).
- **Data-driven models:** Trained directly from data using neural networks (see Fig. 1b).

Our goal is to learn a hybrid dynamics model that combines the advantages of both (see Fig. 1c). Specifically, let f_r denote the hybrid model, f_p the physics engine, and f_θ the residual model. The formulation is:

$$f_r(s_t, a_t) = f_\theta(f_p(s_t, a_t), s_t, a_t) \approx s_{t+1} \tag{2}$$

Here, f_θ learns the residual between the physics engine's prediction and the ground truth, conditioned on the input.

a) Recurrent Dynamics for Long-Horizon Prediction.: For multi-step rollouts, we use a recurrent residual model $f_\theta^R : \mathcal{S}\times\mathcal{S}\times\mathcal{A} \to \mathcal{S}$. Let $\bar{s}_t$ be the physics engine's prediction at time t and $\hat{s}_t$ the corrected prediction. The recurrent update rule is:

$$\bar{s}_{t+1} = f_p(\bar{s}_t, a_t), \quad \bar{s}_0 = \hat{s}_0 = s_0 \tag{3}$$

$$\hat{s}_{t+1} = f_\theta^R(\bar{s}_{t+1}, \hat{s}_t, a_t) \tag{4}$$

b) Training Objective.: We train the model from observed transitions $\{(s_t, a_t, s_{t+1})\}_{t=0}^{T-1}$ by minimizing the prediction loss:

$$\theta^* = \arg\min_\theta \sum_{t=0}^{T-1} \|\hat{s}_{t+1} - s_{t+1}\|_2^2 + \lambda \|\theta\|_2^2 \tag{5}$$

where λ is the weight of the ℓ_2 regularization term.

c) State Representation.: Each state $s_t \in \mathbb{R}^{16}$ consists of two concatenated object states: a *mid-object* and a *target object*. The state vector of each object is 8-dimensional, including: Position: $[x, y, \theta]$, Velocity: $[\dot{x}, \dot{y}, \omega]$, and object parameters: mass m and radius r. This design allows the model to generalize over varying physical parameters across scenes.

C. Interaction Networks & Simulator-Augmented Interaction Networks (SAIN)

Interaction Networks (INs) [3] were introduced to model complex physical interactions between multiple objects. An IN decomposes the learning problem into two stages: first, it computes pairwise relational features between objects via a learned function f_{rel}; then, it uses these aggregated interactions along with object-specific features to predict future dynamics via another learned function f_{dyn}. This design allows the model to capture object-object interactions in a structured and interpretable way.

However, standard intelligent neural networks are completely data-driven and run independently of any physics engine. While they are highly expressive, they also face two key limitations: First, they typically require a lot of training data to achieve good generalization, especially when object properties (such as mass or radius) change. Second, their predictions are not constrained by physical priors, which can lead to unrealistic or inconsistent behavior in long-term deployments.

To address these challenges, we employ the Simulator Augmented Interaction Network (SAIN) architecture, which augments the original Interaction Network (IN) with a physics engine as a strong prior. Instead of learning the full dynamics from scratch, SAIN learns only the residual between the physics-based simulator predictions and the real-world behavior. The key intuition is that learning this residual is often easier and more sample-efficient than learning the full dynamics.

In SAIN, we first apply a physics engine (e.g., Bullet) to obtain a coarse next-state prediction. This predicted state, along with the original state and action, is fed into the interaction network modules f_{rel} and f_{dyn}. These modules then predict residual updates that are applied on top of the physics prediction to obtain a revised next state. This design enables SAIN to leverage strong physical priors from the simulator to improve generalization to unknown object configurations; more effectively exploit learning capabilities by focusing on model inaccuracies; and achieve more realistic and stable deployments in control.

Compared to the standard Intelligent Input Network (IN), SAIN introduces additional inputs such as the displacement and velocity changes of objects predicted by the simulator. The expanded feature space helps the model learn how the simulator deviates under different object masses, shapes, and surface frictions. This correction information is then fused to the dynamic data learned by the model through residual connections, resulting in a hybrid system that is both physically based and adaptive.

In our implementation, the SAIN model operates in single-step prediction mode during training and supports multi-step deployment during control. The state of each object is represented by pose, velocity, and object parameters (mass and radius), allowing the model to generalize to a wide range of physical variations.

D. Control Algorithm MPPI

To validate the predictive accuracy and control utility of our learned SAIN dynamics model, we integrate it into a model-predictive control (MPC) framework. Specifically, we use the Model Predictive Path Integral (MPPI) control algorithm [4], a sample-based stochastic optimal control method well-suited for continuous control tasks.

At each timestep, MPPI generates K action sequences over a fixed horizon H, evaluates their expected cost using the SAIN dynamics model, and selects the first action from the optimal trajectory. The dynamics model receives the current state and candidate action sequences as input, and predicts future states via single-step rollouts. In our implementation, the predicted dynamics are used in a step-dependent manner, i.e., the model is rolled forward H steps using its own predictions.

The cost function is defined as the weighted squared error between the predicted position and the desired target position of the small disk:

$$c(s_t, a_t) = (p_t - p_{\text{goal}})^\top Q (p_t - p_{\text{goal}}), \tag{6}$$

where p_t is the planar position of the small disk extracted from the predicted state s_t, and Q is a diagonal matrix penalizing x, y, and orientation errors. We empirically set the weights to prioritize planar alignment with the goal location.

To ensure full utilization of the learned model, we replaced the physics engine with the identity function during deployment and relied entirely on SAIN for predictions. The model outputs residual updates for velocity and position, which are applied incrementally at each step.

During each episode, the robot performs up to 15 control steps. At each step, MPPI computes an action using the current environment state and executes it in real simulation. This process repeats until the small disk reaches within 5 cm of the target radius.

This closed-loop control setup allows us to test the model's generalization ability to new object parameters (mass, radius, friction), and we observe that the controller achieves a near-perfect success rate across 40 different evaluation cases, confirming the reliability and generalization ability of the SAIN model.

III. Experiments

The entire algorithm is implemented on Ubuntu 24.04, Python version is 3.12. PyBullet is an open-source framework for simulating physics, controlling robotics, and conducting research in reinforcement learning. It offers an effective physics engine that facilitates rigid body dynamics, collision detection, flexible body simulation, and constraint system modeling. PyBullet is very adept at robotic simulation tasks, capable of loading prevalent robot models (including URDF and SDF formats), executing motion control, path planning, and sensor simulation. The versatile API and visualization features enable users to rapidly construct intricate simulation environments and effortlessly interface with deep learning frameworks and reinforcement learning algorithms, offering an effective testing platform for the development and validation of robotic algorithms. In this experiment, we use pybullet to create a robot environment and verify the algorithm.

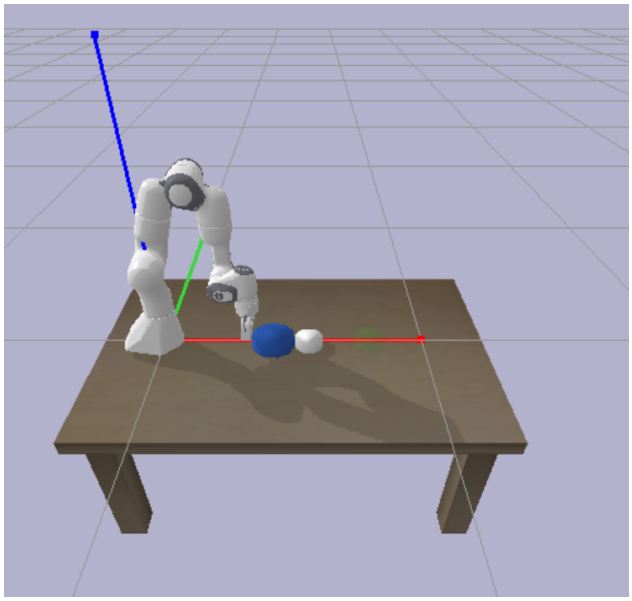

Fig. 2: The Experimental Environment

A. Task and Simulation Setup

Our goal is to evaluate the ability of the SAIN model to predict and control object dynamics in a pushing task, where the robot pushes an intermediate object (middle disk) to indirectly push a target object (small disk) toward a target location. The pushing plane is flat and all objects move in the xy plane (see Figure 2). The middle disk (radius $r = 0.08$) acts as a proxy pusher, while the small disk (radius $r = 0.06$) is the object being pushed. The masses of both objects are initialized to $m = 8$. When the planar position of the small disk is within $0.05\,\text{m}$ of the target, it is considered to have reached the target. Here, we define two environments:

- **Nominal (Non-Randomized) Environment:** used for training the analytical simulator. All physical parameters are fixed:
  - mass_big = 8.0, r_mid_obj = 0.08
  - mass_small = 8.0, r_obj = 0.06
  - friction = 0.12
- **Randomized Environment:** used for collecting training data and evaluating generalization. At the beginning of each episode, object parameters are randomly sampled from:
  - mass_big, mass_small $\sim \mathcal{U}(7.0, 9.0)$
  - r_mid_obj $\sim \mathcal{U}(0.075, 0.085)$, r_obj $\sim \mathcal{U}(0.055, 0.065)$
  - friction $\sim \mathcal{U}(0.08, 0.20)$

  These variations ensure diversity in dynamics and promote generalization.

For state representation, for each object, it has planar position (x, y, θ), planar velocity (v_x, v_y, ω), mass m, and radius r. Full state is $s = [\text{mid-disk}, \text{small-disk}] \in \mathbb{R}^{16}$

B. Data Collection

a) Action Sampling.: We define a discrete action space by discretizing three continuous action parameters: Push location (5 bins), angle (7 bins), and push length (3 bins).

A uniform grid of $5 \times 7 \times 3 = 105$ actions is generated, and one is sampled uniformly at each step.

b) Physics-Augmented Rollout.: To create labeled data tuples of the form $(s_t, a_t, f_p(s_t, a_t), s_{t+1})$, we synchronize two environments:

- **Ground truth environment**: randomized physical parameters, used to generate s_{t+1} via real rollout.
- **Shadow environment**: fixed nominal parameters, used to generate $f_p(s_t, a_t)$ as the physics engine's prediction.

At each step, if the small disk does not move significantly (distance $< 0.002\,\text{m}$), we throw it away.

c) Parallel Collection.: We parallelize data collection across 8 CPU workers. Each worker generates one episode at a time (10 steps per episode), and valid samples are collected. Once 1,000 valid samples have been collected, they are immediately saved to disk in compressed `.npz` format and removed from memory to reduce RAM usage.

d) Dataset Summary.: We collected data from 5,000 randomized episodes, each with up to 10 action steps. After filtering, we obtained approximately 3,000 high-quality transitions.

e) Dataset Format.: Each sample is a 4-tuple:

- `state` $s_t \in \mathbb{R}^{16}$: concatenation of two object states (each with position (x, y, θ), velocity (v_x, v_y, ω), mass m, radius r).
- `action` $a_t \in \mathbb{R}^3$: push action applied to the mid-disk.
- `physics_next` $f_p(s_t, a_t) \in \mathbb{R}^{16}$: prediction from the shadow physics engine.
- `next_state` $s_{t+1} \in \mathbb{R}^{16}$: true next state from ground-truth environment.

f) Batch Construction.: During training, the dataset is randomly split into training and validation subsets (90% / 10%). A batch size of 16 is used to construct minibatches for single-step supervised learning.

C. Model Training

To train the Simulator Augmented Interaction Network (SAIN), we used a dataset of about 3,000 valid single-step transitions collected under randomized object parameters (mass, radius, and friction coefficient). Each transition consists of a tuple $(s, a, fp(s, a), s')$, where s and s' represent the states before and after the action a, respectively, and $fp(s, a)$ is the physics engine's prediction of the next state.

We trained SAIN in a supervised manner, minimizing the mixed loss between the model's prediction $\hat{s}'$ and the true next state s'. Training was performed using the Adam optimizer with an initial learning rate of 0.001 and a learning rate decay factor of 0.5 every 2,500 steps. We trained the model for 8,000 epochs using a batch size of 16.

We recorded the training loss and validation loss during training. The training loss is calculated over all batches of the training set, while the validation loss is evaluated every 500 epochs on a held-out validation set containing 20% of the data. The model checkpoint with the lowest validation loss is saved as the best model.

Figure 3 shows the evolution of training and validation losses over time. We observe that the validation loss remains stable, indicating that the model generalizes well without significant overfitting.

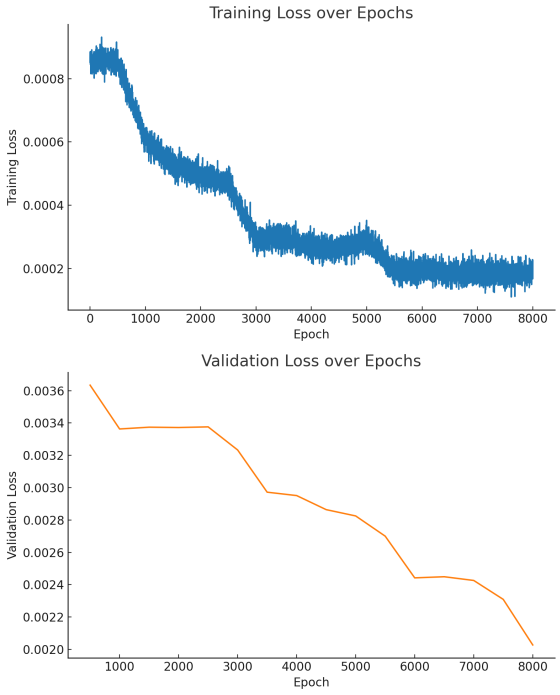

Fig. 3: Training loss (up) and validation loss (down) during SAIN training.

D. Prediction and Control Results in Simulation

To evaluate the learned SAIN dynamics model in closed-loop control, we deploy it with a model predictive

path integral (MPPI) controller in randomized simulation environments. Across **40 randomly sampled test scenes**, the SAIN-MPPI pipeline achieves a **100% success rate**, pushing the small disk to the goal under randomized mass, radius, and friction parameters.

A full set of successful rollouts is available at: DEMOS ,and also you can see Figure 4

TABLE I: Performance of SAIN + MPPI controller on randomized simulation scenes.

Metric	Value
Number of Trials	40
Number of Successes	40
Number of Failures	0
Success Rate	100%

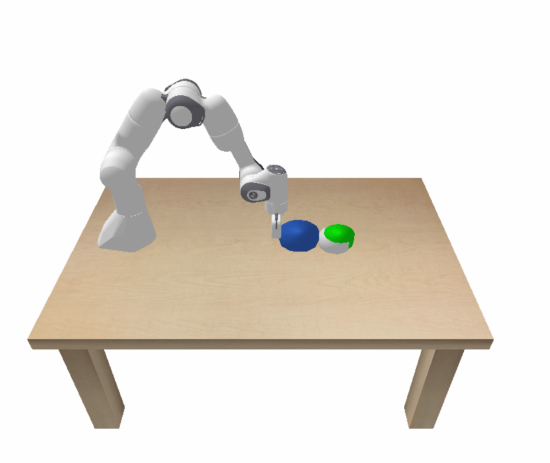

Fig. 4: Example of a successful rollout under randomized dynamics.

Generalization to Unseen Target Positions. Although our SAIN model was trained on a fixed goal configuration, it generalizes remarkably well to perturbed goals. In Figure 5, we shift the goal location by ± 0.02 meters along x and y axes, and the controller still consistently succeeds.

Generalization to Unseen Object Shapes. We also tested shape generalization. While our model was trained with disk-shaped target objects, we replaced the small disk with a cube of similar size and ran the same controller. Surprisingly, the SAIN-MPPI controller still completed the task successfully, as shown in Figure 6.

E. Discuss

Our experiments show that combining an analytical physics engine with learned residual dynamics (via SAIN) can significantly improve prediction and control performance on complex interaction-based tasks. This hybrid approach leverages the strengths of both worlds: the physics-based model provides strong inductive priors and generalization capabilities, while the learned residual corrects systematic errors, especially in the case of domain shifts.

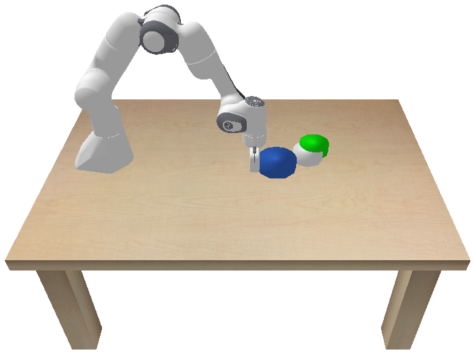

Fig. 5: SAIN generalizes to goal shifts ± 0.02m not seen during training.

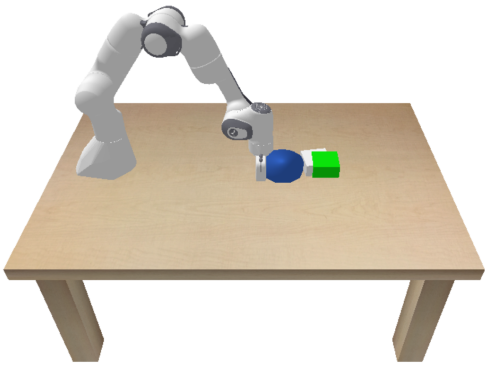

Fig. 6: SAIN generalizes to unseen object shape: replacing disk with a cube.

A key advantage observed in our setting is the generalization to unseen configurations—both in terms of target locations and object shapes. Although the model has never been trained on transformed targets or cube-shaped targets, the SAIN-based controller is still able to solve the task, suggesting that the residual model learns an abstract and transferable understanding of the underlying dynamics.

However, we also note some limitations. The current

formulation assumes a fixed number of objects (e.g., two disks) and relies on carefully curated supervision data from real environments and simulators. Extending this pipeline to handle variable numbers of objects or full image-based state representations remains an exciting direction.

Furthermore, while we use MPPI as our controller, it may be interesting to explore whether more efficient planning methods, such as gradient-based policy optimization or diffusion trajectory planners, can further reduce the computational cost at inference time.

IV. Conclusion

In this study, we propose a hybrid dynamics learning framework based on Simulator-Augmented Interaction Network (SAIN) for driving tasks involving object-to-object interactions. By learning the residual between real-world dynamics and a physics engine, our model achieves higher accuracy and generalization than purely analytical or purely learned alternatives.

We validate SAIN in simulation via a model predictive control pipeline and achieve 100% mission success in 40 random test scenarios. Notably, our approach generalizes to unknown targets and object shapes, highlighting the benefits of combining structured priors with flexible data-driven corrections.

In summary, SAIN provides a practical and sample-efficient approach to learning robot-object interaction dynamics and opens new possibilities for deploying robust policies in partially known physical systems.

References

[1] A. Ajay, M. Bauza, J. Wu, N. Fazeli, J. B. Tenenbaum, A. Rodriguez, and L. P. Kaelbling, "Combining physical simulators and object-based networks for control," 2019. [Online]. Available: https://arxiv.org/abs/1904.06580

[2] A. Kloss, S. Schaal, and J. Bohg, "Combining learned and analytical models for predicting action effects from sensory data," *The International Journal of Robotics Research*, vol. 41, no. 8, p. 778–797, Sep. 2020. [Online]. Available: http://dx.doi.org/10.1177/0278364920954896

[3] P. W. Battaglia, R. Pascanu, M. Lai, D. Rezende, and K. Kavukcuoglu, "Interaction networks for learning about objects, relations and physics," 2016. [Online]. Available: https://arxiv.org/abs/1612.00222

[4] G. Williams, N. Wagener, B. Goldfain, P. Drews, J. M. Rehg, B. Boots, and E. A. Theodorou, "Information theoretic mpc for model-based reinforcement learning," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 1714–1721.