Targeted Object Striking for a 7-DoF Manipulator: A Residual Learning Approach

Priyansh Sinha[*]
Rishin Chakraborty[*]
priyansh.sinha@research.iiit.ac.in
rishin.chakraborty@research.iiit.ac.in
International Institute of Information Technology
Hyderabad, Telangana, India

Samarth Brahmbhatt
Independent Researcher
samarth.robo@gmail.com

Nagamanikandan Govindan
nagamanikandan.g@iiit.ac.in
International Institute of Information Technology
Hyderabad, Telangana, India

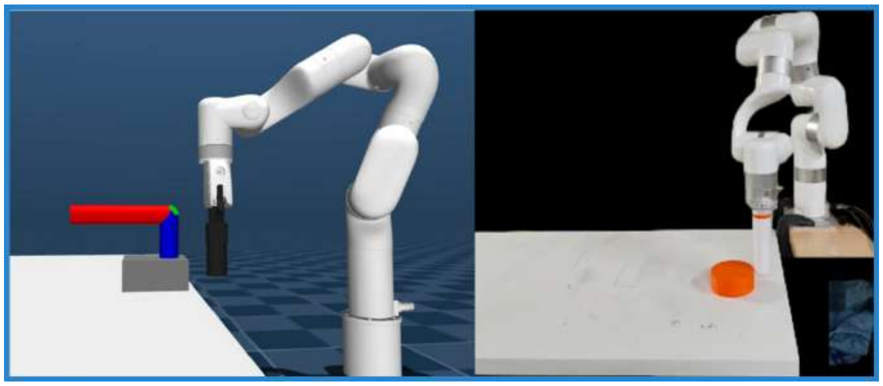


Figure 1: Experimental Setup in Simulation and Real World

Abstract

As robotic manipulators start performing more daily tasks, striking can be a useful method for transporting objects because it significantly increases the reachable workspace. However, striking methods are underexplored compared to pick-and-place because of the difficulty of modeling and executing striking interactions. In this paper, we develop an algorithm for striking objects so that they stop at a target location. We start with an optimizer in simulation that solves for the striking velocity given the relative target position, and perform system identification to set the simulation parameters. However, real-world striking with this model does not have high accuracy because it is unclear which parameters should be considered for identification in practice. Therefore, we finally develop a *residual learning* approach that subsumes all unmodeled differences between the simulation and the real-world environment into action, that is, striking velocity residuals. Our real-world experiments show that the residual learning model results in 81.6% more accurate object strikes.

CCS Concepts

• **Computer systems organization → Robotic control**; **Robotic autonomy**.

Keywords

Striking Manipulation, Non-prehensile Manipulation, Sim-to-Real, Workspace Expansion, Inverse dynamics solver, Residual learning

ACM Reference Format:
Priyansh Sinha, Rishin Chakraborty, Samarth Brahmbhatt, and Nagamanikandan Govindan. 2025. Targeted Object Striking for a 7-DoF Manipulator: A Residual Learning Approach. In *Proceedings of July 02–05, 2025 (AIR '25)*. ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Humans interact with objects through various manipulation strategies, including grasping, pushing, striking, and throwing. While prehensile manipulation, such as grasping, has been the predominant focus in robotic manipulation, it is inherently limited by the robot's reachability and geometries of the object. In contrast, non-prehensile manipulation [9] (NPM, manipulation not involving grasping), particularly striking, extends a robot's effective operational space by significantly enhancing the reachable configurations and degrees of freedom available for object placement. This makes

[*]Both authors contributed equally to this research.

NPM useful when conventional grasping-based manipulation is impractical. Striking finds real-world applications in rapid transport and sorting, especially in cluttered environments without needing multiple arms or complex grippers.

Striking tasks entail high-velocity contacts, complex frictional interactions, and surface-dependent restitution. Traditional system identification cannot capture all these effects simultaneously, and Reinforcement Learning methods are data-hungry and unstable. Our goal is a lightweight, reproducible pipeline that unifies physics-based planning with learned residuals.

In this paper, we develop a robot arm motion controller that can strike objects such that they stop at the target location. Given an ’object’ and an ’environment’, we assume an initial training phase in which strikes can be conducted with the robot, and their execution can be observed. We develop an algorithm that uses data from this training phase to learn a model that can predict the striking velocity given the initial and target position of the object. Striking velocity is defined as the 6 DoF velocity of the robot end-effector at the initial object position. While this model is expected to generalize to unseen initial and target positions, generalizing to unseen objects and surfaces is out of scope of this work. Our experiments are conducted on flat horizontal surfaces, although our algorithm does not require the surface to be either flat or horizontal.

Our key methodological novelties are the following:

(1) Unlike prior work that relies on manual tuning or brute-force approaches, we use machine learning to learn a residual term to absorb all unmodeled effects.
(2) We combine a MuJoCo+NLopt inverse-dynamics solver with a learned residual in a single pipeline, rather than treating sim-to-real and control as separate modules.
(3) Our approach reduces required hardware trials by an order of magnitude compared to a traditional machine learning approach.

We start with a simulation-based inverse dynamics solver that calculates the striking velocity given a pair of initial and final locations of a point mass object. The natural next step is to decide the values of the parameters of the simulator used in the inverse dynamics solver, such that they match the real world. This can be done using data collected from the training phase. This approach is known in the literature as system identification [1].

However, this approach has some limitations that prevent accurate performance in terms of error between the target and achieved object stopping position. This is also seen in our experiments in Section 4.2.2. While robotics simulators are becoming more accurate (we use MuJoCo[13]), the simulator might not expose all the parameters that affect striking in the real world. In addition, it is desirable to be parsimonious while selecting parameters for system identification to limit the amount of training data required, but it is not clear which specific parameters to select. This necessitates a brute force approach that sweeps across all possible sets of simulation parameters (as outlined in Section 3.3), which is computationally expensive.

Therefore, in this paper we explore a *residual learning* approach in which we do not seek to identify specific simulation parameters or their values. Instead, we assume that the net effect of mismatch between simulation and the real world can be captured as an delta i.e. an additive residual term that is added to the striking velocity predicted by the simulator. The residual term has the same units as the action space of the algorithm i.e. 6 DoF velocity. We parameterize the model that predicts the residual striking velocity from the initial and target object positions using a neural network, and learn the neural network from the data collected in the training phase. Finally, we get the striking velocity to be executed on the robot by summing the inverse dynamics solution and the predicted residual. Our real-robot experiments in Section 4.2.2 show that this residual learning approach yields significantly more accurate strikes than the system identification approach.

To summarize, we make the following contributions in this paper:

(1) A MuJoCo simulator-based inverse dynamics solver that calculates the ‘ideal’ (i.e. assuming simulation matches the real world perfectly) striking velocity given a pair of initial and target object positions.
(2) A residual learning approach that predicts an additive term to the solver striking velocity, which compensates for all modeled and unmodeled mismatches between the simulator and the real world.
(3) Experiments with a 7 DoF robot manipulator in the real world, that compare our proposed residual learning approach to baselines.

2 Related Works

Engaging in NPM provides inherent challenges due to the underconstrained nature of the problem, making it challenging to approach using optimization [8], [2]. Recent developments on hybrid trajectory optimization (TO) frameworks have demonstrated efficient methods for planning contact-based manipulation tasks, such as those detailed in [5], [3] and [10]. Studies such as those by [10] have introduced Mathematical Programs with Complementarity Constraints (MPCC) for trajectory optimization in non-prehensile planar manipulation for pushing action. This method is capable of switching between sticking and sliding contact modes, demonstrating improved performance over traditional approaches.

While pushing techniques have been extensively researched, striking remains under-explored in robotics literature. [11]. Striking involves dynamic, high-velocity impacts that pose challenges distinct from those found in static grasping or controlled pushing. Early efforts in dynamic manipulation often relied on model-based controllers tuned for specific tasks; however, these approaches struggled with the inherent uncertainties of impact dynamics.

Zheng et al. introduced TossingBot[15], a system in which a robot learns to throw arbitrary objects by leveraging residual policy learning to account for the complex physics during the throwing (or striking) phase. By learning a corrective residual over a simple baseline controller, TossingBot can adapt to variation in object configurations and dynamic contacts. While TossingBot has demonstrated impressive capabilities in throwing arbitrary objects with high accuracy and speed, there are still many areas where improvements could be made to handle complex and unpredictable environments more effectively:(1) Variability in Object Properties- the system may struggle with objects that have significantly different physical properties (e.g., weight or surface friction) than those it was trained on. (2) Robustness to external factors – Since *throwing* does not require accounting for interactions between the object and the surface, it will influence the performance of other actions like *striking* which stays in constant contact with the surface. (3) It relies on a highly simplified inverse dynamics model that assumes air resistance-free parabolic object trajectories, enabling closed-form solutions. However, this approach is insufficient to account for various environmental disturbances. Employing a full-fledged simulation system like MuJoCo[13] as a model, allows for a more accurate and iterative inverse dynamics solver.

In addition to these approaches, recent advancements have explored reinforcement learning techniques for NPM. For instance, research presented by [6] emphasizes the importance of robust compensation mechanisms in robot actions involving dynamic motions, highlighting how physics simulations can be critical for testing before deploying on real hardware. While this aligns with our proposed Sim-to-Real framework, reinforcement learning introduces several challenges. It is prone to high variance and instability, heavily reliant on large amounts of trial-and-error data, and often sample inefficient and computationally demanding. These factors can hinder deployment in resource-constrained environments. In contrast, a simpler, task-specific method mitigates these limitations by reducing computational overhead and improving practicality.

Finally, integrating learning-based methods with optimization frameworks is gaining traction as a promising way to enhance the reliability and accuracy of robotic manipulation tasks. By leveraging real-world data and simulation outcomes, our approach aims to minimize discrepancies between simulated and actual robot performance, thus addressing the limitations identified in previous research.

3 Methodology

This section outlines the approach used to optimize and execute striking motions with a 7-DOF manipulator, ensuring accurate and reliable real-world performance. Our objective is to determine the optimal striking speed and contact angle (see Figure 3) such that when the object is struck toward its center of mass, it comes to rest precisely at the target position. First, a hardware controller is developed to generate smooth and precise trajectories that achieve the desired striking speed and position. An inverse dynamics solver using Mujoco[13] is then developed to determine optimal striking parameters—speed and angle, given the values of simulation parameters obtained through system identification. Next, a residual learning framework is introduced to account for the unmodeled real-world factors and compensate for the discrepancies between simulation and hardware execution. Finally, the learned residuals are applied to refine hardware commands, improving accuracy in real-world striking tasks. Figure 2 shows the overview of the proposed pipeline.

Problem Statement: Given the initial position of the object i and the desired final position f_d, find the desired cartesian striking speed v and angle of approach $theta$ so that the object comes to rest at f_d after the impact.

The two key challenges with this approach are as follows: (1) Compared to pick-and-place or pushing, where end-effector position is the key consideration, for striking manipulation, we require precise control over end-effector position as well as velocity so that the object can be struck with the desired velocity. To tackle this, we have designed a low-level controller that executes velocity trajectories accurately.
(2) Striking actions are affected by several intrinsic material properties of the object, robot, and environment(including the surface). These are difficult to account for and model accurately in a simulation environment, making it difficult to train a model to predict them. Instead, we introduce a single residual term that accounts for the effects of all these unmodeled parameters, and we train our model to predict this residual term.

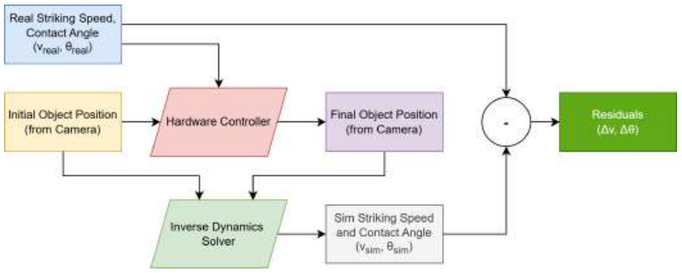

Real Striking Speed, Contact Angle (v_{real}, θ_{real})
Initial Object Position (from Camera)
Hardware Controller
Final Object Position (from Camera)
-
Residuals (Δv, $\Delta\theta$)
Inverse Dynamics Solver
Sim Striking Speed and Contact Angle (v_{sim}, θ_{sim})

Figure 2: Overview of the Residual Learning Pipeline

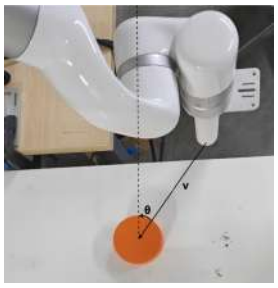


Figure 3: We determine the required striking speed and contact angle for the object to stop at the target location. Here, the contact angle is the angle that the end-effector trajectory makes with the negative x-axis

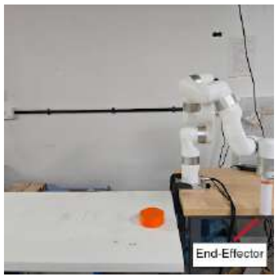
3.1 Low Level Control

For effective striking, the robot's end-effector must strike the object precisely at the desired point of contact with the desired velocity. The primary challenge is generating and executing a trajectory that defines both positions and corresponding velocities as functions of time.

To address this, we utilize the in-built kinematics solver provided by the xArm Python package[14] along with a custom wrapper function. This wrapper computes a smooth velocity profile for a linear trajectory with the following features:

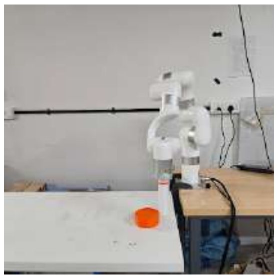
1. **Striking Speed:** The desired speed attained during the motion.
2. **Acceleration:** The Cartesian acceleration of the end-effector.
3. **Striking Position:** The target position where the strike is executed; the end-effector reaches this position at the specified striking speed.

(4) **Pre-strike Position:** A point located 10 mm in front of the striking position. Here, the end-effector achieves the striking speed and maintains it uniformly until reaching the striking position.
(5) **Starting Position:** Based on constant acceleration, this is the point in front of the object from which the end-effector begins accelerating to achieve the striking speed by the pre-strike position.
(6) **Stopping Position:** The point where the end-effector comes to a smooth stop after completing the strike.

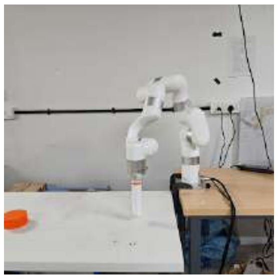


(a) Initial position

(b) Pre-strike position



(c) Strike position



(d) Stopping position

Figure 4: Key Robot poses during the striking action

The wrapper function computes the pre-strike and stopping positions using the given striking speed and acceleration, and then employs the xArm linear trajectory function to execute the entire trajectory.

3.2 Inverse Dynamics Solver

The Inverse dynamics solver is formulated using Mujoco and NLOPT. Its objective is to determine the optimal striking speed (v) and angle of approach (θ) required to move the object from a given initial position to a desired final position (f_d).

To achieve this, an optimization solver utilizing the NLopt package is employed [4]. The solver begins with randomly initialized values for the Striking parameters: striking speed(v_i) and angle (θ_i), executes the motion in MuJoCo to strike the object, and records the resulting final position (f). The error between f and f_d is then computed, and the striking parameters are iteratively refined until the error falls below a predefined threshold. This approach ensures precise determination of striking parameters, improving accuracy and reliability in real-world execution.

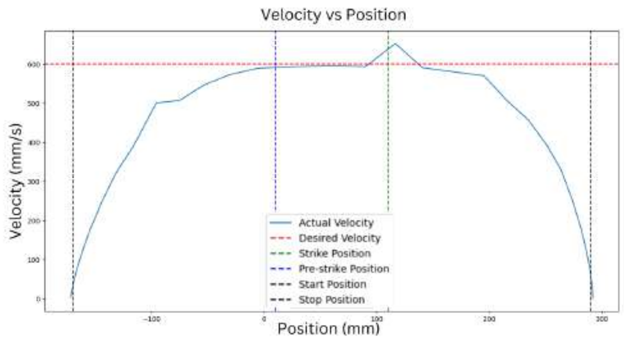


Figure 5: Velocity vs Position plot obtained using our Low Level Controller

Algorithm 1: Optimization of Striking Parameters using NLopt and MuJoCo

```
Input: Desired final position $f_d$, error threshold $\epsilon$
Output: Optimized parameters $(v_{sim}, \theta_{sim})$
Initialize striking speed $v \leftarrow v_0$ and angle $\theta \leftarrow \theta_0$ randomly;
repeat
    Run MuJoCo simulation with parameters $(v, \theta)$ to obtain final position $f$;
    Compute error $E \leftarrow \|f - f_d\|$;
    Update $(v, \theta)$ using NLopt to minimize $E$;
until $E < \epsilon$;
return $(v_{sim}, \theta_{sim})$;
```

3.3 System Identification

A simulation environment is developed in MuJoCo to closely replicate the real-world setup, consisting of a table with a specified coefficient of friction and an object placed on it. To create an accurate model, we use all known physical parameters such as mass and surface area exactly, and then we perform system identification [1] to refine the simulation parameters, focusing on matching the final positions for the given striking velocity and contact angle.

We collect a set of strikes on the hardware, keeping the initial object position constant and noting the striking speed (V), contact angle(θ), and final position(F). Then, we perform strikes with the same striking speed and contact angle in simulation, noting the final position achieved in simulation(F'). We run NLopt to minimize the function $F - F'$. We optimize the following set of values:

- **Contact parameters:**
  - $friction$ (the 3-element friction array: sliding, torsional, rolling)
  - $solref$ (solver reference parameters for contact)
  - $solimp$ (solver impedance parameters for contact)
  - $restitution$ (elasticity of the collision)
- **Fluid coefficients:**
  - $fluid_coeffs$ (contains drag and lift coefficients)
- **Inertia matrix:**
  - $inertia$ (3-element diagonal of the inertia tensor)

Algorithm 2: Optimize MuJoCo Simulation Parameters via NLopt

```
Input: Hardware dataset $\{(V_i, \theta_i, F_i)\}_{i=1}^{N}$, initial parameter
        set $\mathbf{p}_0$, threshold $\epsilon$
Output: Optimized simulation parameters $\mathbf{p}^*$
Define $\mathbf{p}$ = {friction, solref, solimp, restitution,
  fluid_coeffs, inertia};
$\mathbf{p} \leftarrow \mathbf{p}_0$;
repeat
    $E \leftarrow 0$;
    for $i = 1$ to $N$ do
        Run MuJoCo simulation with strike parameters
          $(V_i, \theta_i)$ using current $\mathbf{p}$ to obtain final position $F_i'$;
        $E \leftarrow E + \|F_i - F_i'\|$;
    Update $\mathbf{p}$ with NLopt to minimize $E$;
until $E < \epsilon$;
return $\mathbf{p}$;
```

3.4 Residual Learning

Due to unmodeled parameters in the real world, the final position achieved in hardware may differ from that in simulation for the same striking speed and angle. To bridge this gap, we model the discrepancy between simulation and real-world commands as a residual.

Data is collected on the hardware for 500 randomly selected combinations of initial positions, striking speeds, and striking angles. The hardware measurements for striking speed and angle are recorded as v_{real} and θ_{real}, respectively, while the path length between the initial and final positions is captured as P_x and P_y.

The MuJoCo optimizer computes the simulation-based striking speed and angle, denoted as v_{sim} and θ_s, for the corresponding path lengths. The residuals are then computed as:

$$\Delta v = v_{real} - v_{sim}$$

$$\Delta\theta = \theta_{real} - \theta_{sim}$$

which serve as the ground truth.

A deep learning model is trained on four independent variables: P_x, P_y, v_{sim}, and θ_s, and it predicts the residuals Δv and $\Delta\theta$.

3.4.1 *Model Architecture.*

A multi-layer perceptron (MLP) architecture was employed with three hidden, densely connected layers comprising 128, 64, and 32 neurons, each using ReLU activations. To prevent overfitting, a batch normalization layer and a dropout layer were added. The output layer, with two neurons and a linear activation, predicted the residuals Δv and $\Delta\theta$. The model received four input features: P_x, P_y (path lengths in x and y), v_{sim} (simulation striking speed), and θ_s (simulation striking angle). The dataset was split into training and test sets in a 90:10 ratio, with the training set further divided into training and validation subsets (also 90:10), randomized each epoch. The Mean Squared Error (MSE) loss function was used for its effectiveness in penalizing large errors and accelerating convergence. Standard scaling (zero mean, unit variance) was applied during preprocessing, and batch normalization layers within the network further helped reduce overfitting. The Adam optimizer, well-suited for regression, was used with an initial learning rate of 10^{-3} and a step decay strategy, halving the rate if validation MAE did not improve over five epochs, down to a minimum of 10^{-6}. A batch size of 32 balanced convergence speed and computational efficiency. Training was conducted over 500 epochs, beyond which improvements were minimal. Early stopping was implemented if no validation improvement occurred over 20 epochs. The final model achieved a validation MAE of 0.015, with further hardware validation discussed in Section 4.

3.5 Hardware Striking using the Residual

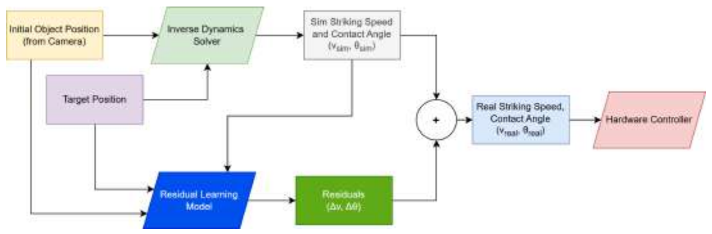

Initial Object Position (from Camera)
Inverse Dynamics Solver
Sim Striking Speed and Contact Angle (v_{sim}, θ_{sim})
Target Position
+
Real Striking Speed, Contact Angle (v_{real}, θ_{real})
Hardware Controller
Residual Learning Model
Residuals (Δv, $\Delta\theta$)

Figure 6: Final execution of the Strike after computation of Δv and $\Delta\theta$ from residual model. Note: (+) denotes the Sim Striking speed is being added with the residuals.

To perform a strike on the hardware targeting a specific final position, the process begins with object detection and determination of its initial position in the robot frame. The initial and final positions are then used to compute the path length parameters P_x and P_y. These parameters are fed into the Mujoco optimizer to predict the simulation-based striking speed (v_{sim}) and angle (θ_{sim}).

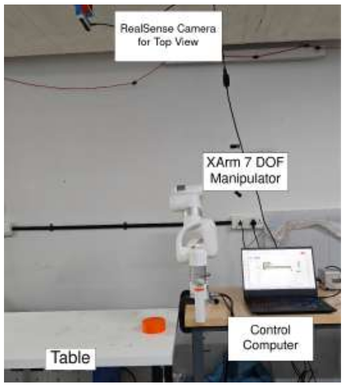
Subsequently, P_x, P_y, v_{sim}, and θ_{sim} are provided as inputs to the residual learning model, which predicts the residuals Δv and $\Delta\theta$. These residuals are added to the optimizer outputs to obtain the required hardware commands:

$$v_{real} = v_{sim} + \Delta v$$

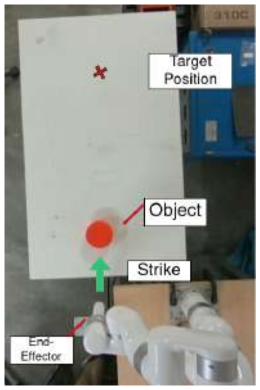
$$\theta_{real} = theta_{sim} + \Delta\theta$$

4 Results

This section describes the setup we used for our experiments, outlines the experiments and discusses the results obtained. In this section we evaluate our proposed residual learning algorithm as well as the system identification approach by conducting strikes in the real world with a 7-DoF xArm manipulator. Our experimental setup consists of the xArm robot mounted at one end of a table, and a top-view RealSense camera that can observe the table, as shown in Figure 7. To detect the object location, we first get an approximate mask in the top-view image using thresholding in the HSV space, and then refine the mask using the OpenCV implementation of GrabCut [12]. The reported object location is the centroid of this refined mask. A script was created to record the initial and final positions of the object, along with the striking speed and contact angle of the striking end-effector by communicating with the robot software library. This script enabled data collection from our robot and camera, which is used in system identification and to train



(a) Side view



(b) Top view

Figure 7: Experimental setup

the residual learning model. The comparison metric for our experiments is the L2 error between the target stopping (x, y) location on the table and the achieved stopping (x, y) location. We report averages and standard deviations calculated over multiple trials.

4.1 Low-level Robot Control

The action output by our algorithm is in terms of end-effector striking velocity in 6-DoF i.e. twist. However, the xArm robot only allows (task-space or joint space) position control. Therefore we first reduced the end-effector 6-DoF velocity output by our algorithm to 2-DoF velocity in the plane of the table. To execute this trajectory on the robot, we initially used the MoveIt Advanced Manipulation Package [7]. However, when we measured the actual position and velocity reported by the robot end-effector, we found that the achieved speed was accurate only up to 500 mm/s, and lacked repeatability at higher speeds.

Therefore, we developed a custom low-level controller that generates a 2D end-effector position trajectory such that its execution results in the end-effector moving at the desired striking velocity at the initial object location. This controller generates a straight-line trajectory centered at the initial object location and oriented by the desired striking angle. The positions of this trajectory follow a trapezoidal velocity profile. We executed it on the robot using using the xArm Python SDK and achieved high accuracy in speeds, and obtained good repeatability across our intended speed range of 400 mm/s - 1000 mm/s.

4.2 Experiments

4.2.1 System Identification. We first assessed the impact of system identification as described in Section 3.3. We conducted 5 strikes on the real robot using the striking velocity calculated by the inverse dynamics solver with and without system identification 1. In the table F is Final Position(cm) and ID is System identification. We found that without system identification, the average error was 23.66 cm, and performing system identification reduced it to 8.74 cm.

4.2.2 Comparing residual learning to system identification. In Figure 8 we compare the real robot striking accuracy results achieved by our proposed residual learning algorithm compared with the system identification approach. We select two pre-defined object initial locations for fair comparison, and report mean and standard deviation in stopping location error for four different target locations. We observe that the residual learning model reduces the mean error in stopping position from an average of 9.12 cm to 1.68 cm, or an 81.6% reduction.

Table 1: Striking accuracy with and without system identification.

Strikes	Real F	Sim F w/o ID	Err w/o ID	Sim F with ID	Err with ID
1	[30, 10]	[53, 27]	27.0	[36, 15]	9.2
2	[25, 15]	[47, 18]	22.8	[32, 18]	7.5
3	[40, 20]	[65, 30]	26.9	[44, 24]	10.1
4	[35, 12]	[55, 28]	25.1	[39, 16]	8.3
5	[20, 8]	[32, 23]	16.5	[27, 13]	8.6
Avg.	–	–	**23.66**	–	**8.74**

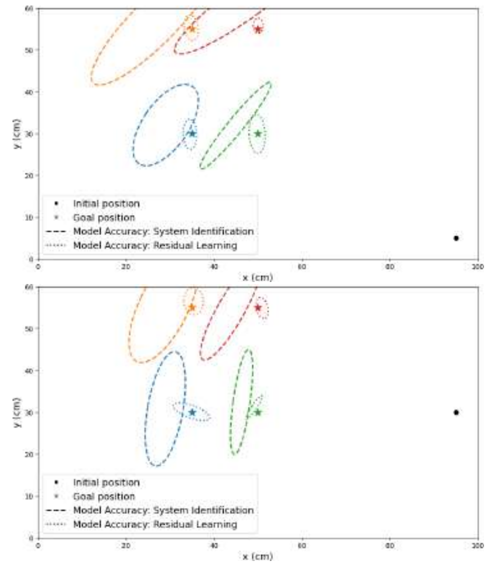


Figure 8: Comparison of real-world object striking accuracy for different models and different initial and target locations. The ellipse centroid indicates the mean stopping position calculated across 5 repetitions, while the length of its axes indicates the standard deviation.

Figure 9 shows snapshots from the robot execution of the strikes in real-world with our residual learning model. It compares the residual model with the system identification. The red dot shows the target position and the green dot represents the current position of the object. Improved error compensation can be observed in the residual learning.

4.2.3 Generalization to a different object. To verify generalization ability of our approach, we performed the same set of experiments, including data collection, model training, and hardware validation,

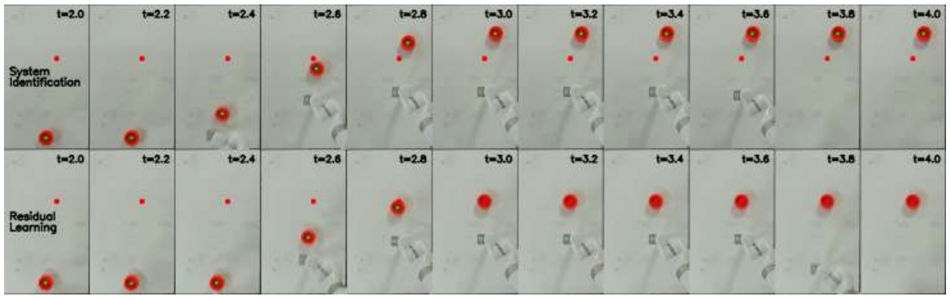


Figure 9: Top view: Hardware experiment snapshots comparing baseline model and residual learning model

for a second object, a cube with different size, mass, and material properties. We obtained similar results for this object as well: the system identification approach results in a mean error of 10.02 cm, while our residual learning approach resulted in a mean error of 4.36 cm.

5 Conclusion

In this paper, we presented a novel framework for non-prehensile striking manipulation that bridges the gap between simulation and real-world execution. Our approach combines physics-based optimization with data-driven residual learning to accurately determine striking parameters, thus expanding the reachable workspace of the robotic manipulator.

The method begins with a high-fidelity MuJoCo simulation whose parameters are iteratively refined using NLopt and system identification techniques. By collecting hardware data on striking speed, contact angle, and final object position, we quantify the discrepancies between simulation and reality. A deep learning model is then trained to predict these residuals that compensate for the effects of unmodeled parameters, thereby enhancing the precision of the striking motion.

Experimental evaluations on a 7-DOF robotic manipulator demonstrate that integrating residual learning into the optimization pipeline significantly improves accuracy and reliability during real-world strikes. Our hybrid framework ensures that the simulation closely replicates the physical environment and adapts to the inherent uncertainties of dynamic interactions. Overall, our results underscore the potential of combining simulation optimization with residual learning to enable robust and efficient sim-to-real transfer in striking manipulation, which may be extended to other complex robotic tasks.

5.1 Future Work

Future work will focus on extending the residual model to account for additional dynamic factors and integrating adaptive control strategies for broader non-prehensile manipulation tasks.

One interesting problem would be dynamically adapting to the object shape and adjusting the contact angle accordingly, which would further improve accuracy and make a single trained model usable for multiple object shapes.

Another problem would be completely eliminating the hardware data collection and developing an adaptive framework that can perform some environment exploration at execution time and tune the residuals based on some parameters measured during the exploration.

References

[1] Adam Allevato, Elaine Schaertl Short, Mitch Pryor, and Andrea Lockerd Thomaz. 2019. TuneNet: One-Shot Residual Tuning for System Identification and Sim-to-Real Robot Task Transfer. *CoRR* abs/1907.11200 (2019). arXiv:1907.11200 http://arxiv.org/abs/1907.11200

[2] Francois R Hogan and Alberto Rodriguez. 2020. Reactive planar non-prehensile manipulation with hybrid model predictive control. *The International Journal of Robotics Research* 39, 7 (2020), 755–773. doi:10.1177/0278364920913938 arXiv:https://doi.org/10.1177/0278364920913938

[3] Taylor A. Howell, Simon Le Cleac'h, Sumeet Singh, Pete Florence, Zachary Manchester, and Vikas Sindhwani. 2022. Trajectory Optimization with Optimization-Based Dynamics. *IEEE Robotics and Automation Letters* 7, 3 (2022), 6750–6757. doi:10.1109/LRA.2022.3152696

[4] Steven G. Johnson. 2007. The NLopt nonlinear-optimization package. https://github.com/stevengj/nlopt.

[5] Prakrut Kotecha, Priyansh Sinha, and Nagamanikandan Govindan. 2024. A Hierarchical Manipulation Planning Framework Combining Striking, Pushing, and Pick & Place Motion Primitives. In *2024 IEEE 20th International Conference on Automation Science and Engineering (CASE)*. 956–961. doi:10.1109/CASE59546.2024.10711645

[6] Kendall Lowrey, Svetoslav Kolev, Jeremy Dao, Aravind Rajeswaran, and Emanuel Todorov. 2018. Reinforcement learning for non-prehensile manipulation: Transfer from simulation to physical system. (03 2018). doi:10.48550/arXiv.1803.10371

[7] Pablo Malvido Fresnillo, Saigopal Vasudevan, Wael M. Mohammed, Jose L. Martinez Lastra, and Jose A. Perez Garcia. 2023. Extending the motion planning framework—MoveIt with advanced manipulation functions for industrial applications. *Robotics and Computer-Integrated Manufacturing* 83 (2023), 102559. doi:10.1016/j.rcim.2023.102559

[8] Matthew T. Mason. 1986. Mechanics and Planning of Manipulator Pushing Operations. *The International Journal of Robotics Research* 5, 3 (1986), 53–71. doi:10.1177/027836498600500303 arXiv:https://doi.org/10.1177/027836498600500303

[9] Matthew T. Mason. 1999. Progress in Nonprehensile Manipulation. *The International Journal of Robotics Research* 18, 11 (1999), 1129–1141. doi:10.1177/02783649922067762 arXiv:https://doi.org/10.1177/02783649922067762

[10] João Moura, Theodoros Stouraitis, and Sethu Vijayakumar. 2022. Non-prehensile Planar Manipulation via Trajectory Optimization with Complementarity Constraints. In *2022 International Conference on Robotics and Automation (ICRA)*. 970–976. doi:10.1109/ICRA46639.2022.9811942

[11] Anuj Pasricha, Yi-Shiuan Tung, Bradley Hayes, and Alessandro Roncone. 2022. PokeRRT: A Kinodynamic Planning Approach for Poking Manipulation. arXiv:2203.04761 [cs.RO] https://arxiv.org/abs/2203.04761

[12] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. 2004. GrabCut: Interactive Foreground Extraction Using Iterated Graph Cuts. In *ACM Transactions on Graphics (TOG)*, Vol. 23. ACM, 309–314.

[13] Emanuel Todorov, Tom Erez, and Yuval Tassa. 2012. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 5026–5033. doi:10.1109/IROS.2012.6386109

[14] UFactory. 2023. xArm-Python-SDK. https://github.com/xArm-Developer/xArm-Python-SDK. [Online; accessed 24-February-2025].

[15] Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. 2020. TossingBot: Learning to Throw Arbitrary Objects With Residual Physics. *IEEE Transactions on Robotics* 36, 4 (2020), 1307–1319. doi:10.1109/TRO.2020.2988642

Received 24 February 2025; accepted 18 April 2025