

427 Appendix

428 A1 Extended Related Work

429 **Sys-ID domain adaptation.** Inspired by classical work in Sys-ID [14, 15], there has been a popular line
 430 of work identifying simulation parameters that match the robot and environment dynamics in the real
 431 environment before task policy training. BayesSim [6] and follow-up work [16, 17] applies Bayesian
 432 inference to iteratively search for a posterior distribution of the simulation parameters based on simulation
 433 and real-world trajectories. The inference problem has also been formulated using RL to minimize
 434 trajectory discrepancies [30]. A different approach [31, 32, 33] learns a residual model of dynamics (often
 435 parameterized with a neural network) to match simulation or an ideal physics model with reality. However,
 436 all these methods consider relatively well-modeled environment parameterizations such as object mass or
 437 friction coefficient during planar contact; Sys-ID approaches have been shown to fail in cases where the
 438 simulation does not closely approximate the real world [13, 18]. There is also work that avoids inferring
 439 the full dynamics but adapts with a low-dimensional latent representation online [34, 35, 36], but the
 440 representation is still trained with regression to match dynamics or simulation parameters. Importantly,
 441 the Sys-ID approaches highlighted above are all task-agnostic; this can lead to poor performance when
 442 trained task policies are sensitive to mismatches in dynamics between simulation and reality. Chi et al.
 443 [18] address the issue by using simulation to predict changes to trajectories from changes in actions as
 444 an implicit policy, but it requires the environment to be resettable, while AdaptSim works with randomly
 445 initialized object states.

446 **Task-driven domain adaptation.** AdaptSim better fits within a different line of work that aims to find
 447 simulation parameters that maximize the task reward in target environments. Muratore et al. [19] apply
 448 Bayesian Optimization (BO) to optimize parameters such as pendulum pole mass and joint damping
 449 coefficient in a real pendulum swing-up task. Other work focus on adapting to simulated domains only
 450 [20, 21, 22]. One major drawback of these methods is that they require a large number of rollouts in target
 451 environments (e.g., 700 in [19]), which is very time-consuming for many tasks requiring human reset.
 452 AdaptSim meta-learns adaptation strategies in simulation and requires only a few real rollouts for inference
 453 (e.g., 20 in our pushing experiments). Liang et al. [37] apply the same task-driven objective to learn an
 454 exploration policy in manipulation tasks, but the task policy is synthesized using estimated simulation
 455 parameters via Sys-ID. Jin et al. [38] applies task-driven reduced-order model for dexterous manipulation
 456 tasks, but again the model is identified with Sys-ID and no vision-based control is involved. Ren et al.
 457 [39] search for adversarial environments (e.g., objects) given the current task performance to robustify the
 458 policy, but unlike AdaptSim, the adversarial metric is measured in simulated domain only without real data.

459 **Learn to search/optimize.** Our work involves learning optimization strategies through meta-learning
 460 across a distribution of relevant problems, allowing for customization to the specific setting and increased
 461 sample efficiency [40, 41]. Chen et al. [42] meta-learns an RNN optimizer for black-box optimization.
 462 Volpp et al. [43] meta-learns the acquisition function in BO with RL; it is able to learn new exploration
 463 strategies for black-box optimization and tuning controller gains in sim-to-real transfer. Meta RL trains the
 464 task policy directly to optimize performance in new environments [44, 45, 46] — AdaptSim applies meta
 465 RL to optimize simulation parameters instead.

466 A2 Additional details on approach

467 A2.1 Sparse adaptation reward

468 In practice, we are only concerned with the reward if it reaches some minimum threshold — a bad task
 469 policy is not useful. Thus we use a sparse-reward version of Eq. (2),

$$\mathbb{E}_{E^s \sim \mathcal{U}_\Omega} \mathbb{E}_{\mathcal{E}_0 \sim \mathcal{U}_P} \sum_{i=0}^I \gamma^i \mathbb{1}(R(\pi_{\mathcal{E}_i}^*; E^s) \geq \bar{R}) R(\pi_{\mathcal{E}_i}^*; E^s), \quad (\text{A1})$$

470 where $\mathbb{1}(\cdot)$ is the indicator function and \bar{R} is the sparse-reward threshold. Using a sparse reward also
 471 discourages the adaptation policy from being myopic and getting trapped at a sub-optimal solution,

especially since we use a relatively small I (e.g., 5-10) in order to minimize the amount of real data, and use a small discount factor γ ($=0.9$).

A2.2 Task policy reuse across parameter distributions

Algorithm 1 requires training the task policy for each \mathcal{E} , which can be expensive with the two manipulation tasks. Our intuition is that we can share the task policy between parameter distributions of close distance, with the following heuristics:

- Record the total budget (i.e., number of trajectories), and j , the number of simulation parameter distributions that a task policy has been trained with.
- Define distance between two parameter distribution $D(\cdot, \cdot)$ such as L2 distance between the mean. If \mathcal{E}_i is within a threshold \bar{D} from a previously seen distribution, re-use the task policy. If the policy is already trained with M_{\max} budget total, do not train again; otherwise train with $\max(M_{\min}, \alpha^{j-1}M)$ budget, where $\alpha < 1$ and M is the budget for training the policy for the first time.
- If the nearby parameter distribution re-uses a task policy, do not re-use the same policy again. This prevents the same task policy being used for too many \mathcal{E} .

Remark 1 *re-using task policies between parameter distributions makes the reward R depend on the adaptation history, as $\pi_{\mathcal{E}}^*$ depends on previous \mathcal{E} that are used for training. We choose not to model this history dependency in f , as the reward should be largely dominated by the current \mathcal{E} .*

A3 Additional details of adaptation policies

Hyperparameters. Table A1 shows the hyperparameters used for the adaptation policy training in Phase 1, including those defining the heuristics for re-using task policies among simulation parameter distributions. We generally use smaller adaptation step δ for smaller dimensional Ω .

Parameter	Task		
	Pendulum	Pushing	Scooping
Total adaptation steps, K	1e4	1e4	1e4
Adaptation horizon, I	10	8	8
Adaptation step size, δ	0.10	0.15	0.15
Adaptation discount factor, γ	0.9	0.9	0.9
Sparse reward threshold, \bar{R}	0.95	0.8	0.5
Task policy reuse threshold, \bar{D}	-	0.16	0.16
Task policy max budget, M_{\max}	-	3e4	4e3
Task policy budget discount, α	-	0.9	0.9
Task policy init budget, M	-	1e4	1.2e3

Table A1: Hyperparameters used in adaptation policy training for the three tasks.

Trajectory observations. We detail the trajectory observation (as input to the adaptation policy) used in the three tasks.

- Pendulum task: each trial is 2.5 seconds long, and we use 12 evenly spaced points along the trajectories of the two joints, and thus each trajectory is 24 dimensional. For AdaptSim-State, SysID-Bayes-State, and SysID-Bayes-Point, again 12 points are used but sampled from the last 0.5 second only. One trajectory is used at each adaptation iteration — the trajectory input to the adaptation policy is 24 dimensional.
- Pushing task: each trial is 1.3 seconds long, and we use 6 evenly spaced points along the X-Y trajectory of the bottle, and thus each trajectory is also 12 dimensional. For AdaptSim-State, SysID-Bayes-State, and SysID-Bayes-Point, only the final X-Y position of the bottle is used. Two trajectories are used at each adaptation iteration — the trajectory input to the adaptation policy is 24 dimensional.

- Scooping task: each trial is 1 second long, and we use X-Y position of the food piece at the time step $[0, 0.2, 0.3, 0.4, 0.5, 0.6, 0.8, 1.0]$ s (more sampling around the initial contact between the spatula and the piece), and thus each trajectory is 16 dimensional. Two trajectories are used at each adaptation iteration — the trajectory input to the adaptation policy is 32 dimensional.

In real experiments, we track the bottle position in the pushing task using 3D point cloud information from a Azure Kinect RGB-D camera, which we find accurate. In the scooping task, the food pieces are too small and thin to be reliably tracked with point cloud, and thus we resort to extracting the contours from the RGB image and then finding the corresponding depth values at the same pixels in the depth image. During fast contact there can be motion blur around the food piece, and thus we add Gaussian noise with 0.2cm mean for X position and zero mean for Y position, and 0.2cm covariance for both, to the points in the ground-truth trajectories in simulation. We use positive mean in X since the motion blur tends to occur in the forward direction.

A4 Additional details of the task setup and task policies

Trajectory observation First, we remove the action sequence from the task-policy trajectory and keep the state sequence only. Since the dynamics in real environments can be OOD, in order to achieve similar high-reward states as in simulated environments, the robot would need to use some actions not seen during training (or not seen for the particular state), hindering the adaptation policy to generalize if action sequence were included in the task policy trajectory. We assume that the task-relevant *state* sequence is covered by \mathcal{T} if the task policy performs reasonably well in the real environment. This choice is also present in the state-only inverse RL literature [47] that addresses train-test dynamics mismatch. See Fig. A4 and related discussions in Sec. 6.3.

A4.1 Dynamic pushing of a bottle

Trajectory parameterization. Here we detail the trajectory of the end-effector pusher designed for the task (Fig. A1). The trajectory is parameterized with two parameters: (1) planar pushing angle, which is the yaw orientation of the pusher relative to the forward direction that controls the direction of the bottle being pushed, and (2) forward speed (of the end-effector), in the direction specified by the pushing angle. The pushing angle varies between -0.3rad and 0.3rad , and the forward speed varies between 0.4m/s and 0.8m/s . We find 0.8m/s roughly the upper speed limit of the Franka Panda arm used. The pusher also pitches upwards during the motion and the speed is fixed to 0.8rad/s . We design such trajectories to maximize the pushing distance at the hardware limit.

Initial and goal states. The bottle is placed at the fixed location ($x=0.56\text{m}, y=0$, relative to the arm base) on the table before the trial starts. The goal location is sampled from a region where the X location is between 0.7 and 1.0m and Y location is at most 10 degrees off from the centerline (Fig. A1 top-right). The patch, a 10cm by 10cm square, is placed at $x = 0.75\text{m}$ with its center (lateral position is varied as one of the simulation parameter).

Task policy parameterization. The task policy is parameterized using a Normalized Advantage Function (NAF) [48] that allows efficient Q Learning with continuous action output by restricting the Q value as a quadratic function of the action, and thus the action that maximizes the Q value can be found exactly without sampling. In this task, it maps the desired 2D goal location of the bottle to the two action parameters, planar pushing angle and forward speed. The policy is open-loop — the actions are determined before the trial starts and there is no feedback using camera observations.

Hardware setup. A 3D-printed, plate-like pusher is mounted at the end-effector instead of the parallel-jaw gripper in both simulation and reality. We also wrap elastic rubber bands around the bottom of the pusher and contact regions of the bottle to induce more elastic collision, which we find increases the sliding distance of the bottle.

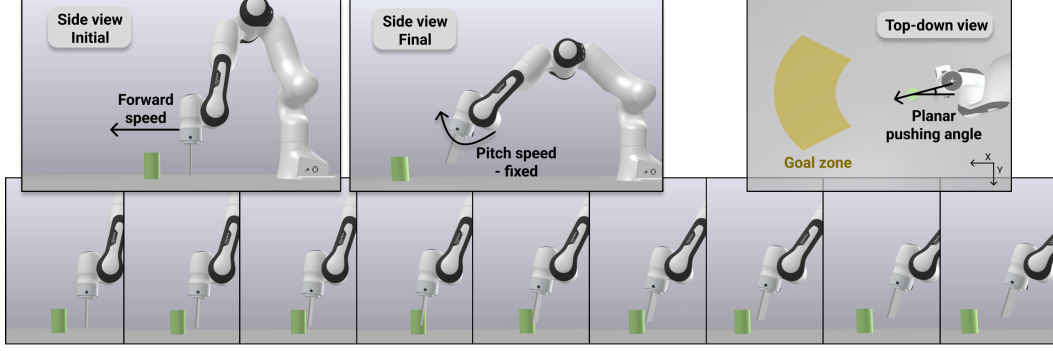


Figure A1: Visualization of the pushing trajectory and goal locations in the Drake simulator. There are two action parameters: (1) forward speed (of the end-effector) and (2) planar pushing angle (*i.e.*, yaw orientation of the end-effector). The patch is not visualized.

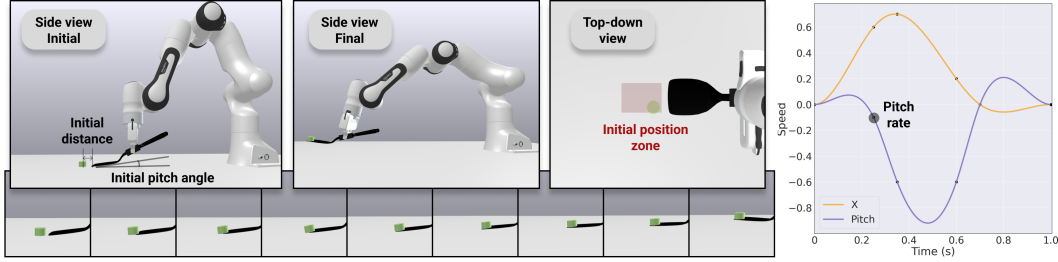


Figure A2: Visualization of the scooping trajectory and initial positions of the food piece in the Drake simulator. There are three action parameters: (1) initial distance (between the spatula and food piece), (2) initial pitch angle (of the spatula from the table), and (3) pitch rate (of the end-effector at time step $t=0.25$).

548 A4.2 Dynamic scooping of food pieces

549 **Trajectory parameterization.** Here we detail the trajectory of the end-effector with the spatula designed
 550 for the task (Fig. A2). The end-effector velocity trajectory is generated using cubic spline with values
 551 clamped at five timesteps. The trajectory only varies in the X and pitch direction (in the world frame),
 552 while remaining zero in the other directions. The only value defining the trajectory that the task policy
 553 learns is the pitch rate, which is the pitch speed at the time $t=0.25$ s and varies between -0.2rad/s and
 554 0.2rad/s . A positive pitch rate means the spatula lifting off the table late, while a negative one means lifting
 555 off early (see the effects in Fig. 8). The other two values that the task policy outputs are the initial pitch
 556 angle of the spatula from the table (varying from 2 to 10 degrees), and the initial distance between the
 557 spatula and the food piece (varying between 0.5cm to 2cm). Generally a higher initial pitch angle can help
 558 scoop under food pieces with flat bottom, and a smaller angle helps scoop under ellipsoidal shapes. We
 559 design such trajectories after extensive testing with food pieces of diverse geometric shapes and physical
 560 properties in both simulation and reality.

561 **Initial states.** The food piece is randomly placed in a box area of 8x6cm in front of the spatula; the initial
 562 distance is relative to the initial food piece location.

563 **Task policy parameterization.** The task policy is parameterized using a NAF again. In this task, it maps
 564 the initial 2D position of the food piece to the three action parameters: pitch rate, initial pitch angle, and
 565 initial distance.

566 **Hardware setup.** We use the commercially available OXO Nylon Square Turner¹ as the spatula used
 567 for scooping. It has a relatively thin edge (about 1.2mm) that helps scoop under thin pieces. A box-like,
 568 3D-printed adapter with high-friction tape is mounted on the handle to help the parallel-jaw gripper grasp
 569 the spatula firmly. The exact 3D model of the spatula with the adapter is designed and used in the Drake
 570 simulator; the deformation effect as it bends against the table is not modeled in simulation.

¹link: <https://www.amazon.com/OXO-11107900LOW-Grips-Square-Turner/dp/B003L000SU>

571 A5 Additional details of experiments

572 A5.1 Simulated adaptation

573 Table A2 shows the simulation parameters used in different simulated target environments for the three
574 tasks (results shown in Table 3).

Task	Parameter	Setting					Range
		WD	OOD-1	OOD-2	OOD-3	OOD-4	
Pendulum	m_1	1.8	1.8	0.5	1.2	0.4	[1,2]
	m_2	1.2	0.3	1.8	1.8	2.6	[1,2]
	b_1	1.5	1.5	1.5	10.0	1.0	[1,2]
	b_2	1.5	1.5	1.5	10.0	2.0	[1,2]
Pushing	μ	0.1	0.25	0.05	0.15	0.30	[0.05,0.2]
	e	1e5	5e4	1e5	5e6	1e5	[1e4,1e6]
	μ_p	0.6	0.1	0.9	0.1	0.15	[0.2,0.8]
	y_p	0.05	-0.1	0.05	-0.15	0.1	[-0.1,0.1]
Scooping	μ	0.30	0.45	0.20	0.30	0.40	[0.25,0.4]
	e	5e4	1e4	5e4	1e6	1e5	[1e4,5e5]
	g	1	0	1	0	2	{0,1}
	h	2.0	1.4	2.2	2.8	1.9	[1.5,2.5]

Table A2: Simulation parameters used in different simulated target environments for the three tasks. OOD parameters (outside the range used in adaptation policy training) are bolded. For g in the scooping task, 0 stands for ellipsoid, 1 for cylinder, and 2 for box.

575 A5.2 Real adaptation

576 In Fig. A7 and Fig. A9 we demonstrate additional visualizations of the pushing and scooping results with
577 AdaptSim.

578 A5.3 Additional studies

579 **Choice of the simulation parameter space.** To answer Q3, we perform a sensitivity analysis by fixing the
580 target environment (OOD-1 in the double pendulum task) and varying the simulation parameter space. In
581 OOD-1, the OOD parameter is $m_2 = 0.3$ while the range in Ω is [1,2]. Fig. A3 shows the results of reward
582 achieved after adaptation for AdaptSim and the two Sys-ID baselines, as the range shifts further away from
583 $m_2 = 0.3$ to [1.1,2.1], [1.2,2.2], and [1.3,2.3]. Sys-ID performance degrades rapidly, while AdaptSim is
584 more robust.

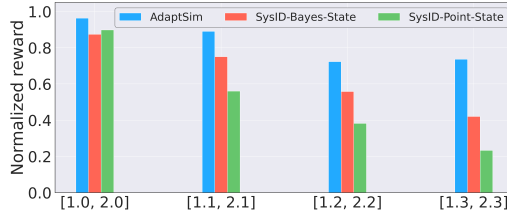


Figure A3: Adaptation results for AdaptSim and Sys-ID baselines in OOD-1 setting of the double pendulum task, with different m_2 ranges in Ω while $m_2 = 0.3$ in the target environment.

585 **Pitfalls of Sys-ID approaches.** Fig. A4 demonstrates the dynamics mismatch between simulation and
586 reality, which illustrates the pitfall of SysID approaches. We plot a set of bottle trajectories from randomly
587 sampled simulation parameters from Ω with a fixed robot action. We also plot the trajectories of Heavy
588 bottle being pushed with the same action in reality. There are segments of real trajectories that are not well
589 matched by the simulated ones, and a slight mismatch can lead to diverging final states (and hence different
590 task rewards).

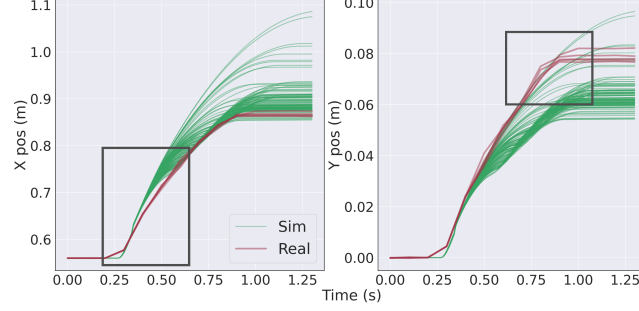


Figure A4: Comparison of trajectories from the simulation domain (green, simulated with randomly sampled simulation parameter settings) and from Heavy bottle in reality (red), with the same robot action applied. The real dynamics can be OOD from simulation (black boxes) while the final position of the bottle can be WD.

Trade-off between real data budget and task performance convergence. In Sec. 4.2 we introduce N , the number of initial simulation parameter distributions that are sampled at the beginning of Phase 2 and then adapt independently. There is a trade-off between the real data budget (linear to N) and convergence of task performance. Adapting more simulation parameter distributions simultaneously can potentially help the task performance converge faster but also require more real data. Fig. A5 shows the effect with the Light bottle in the pushing task. We vary N from 1 to 4 — each simulation parameter distribution takes 2 trajectories at each iteration. $N=1$ shows slow and also worse asymptotic convergence, which shows that the parameter distribution can be trapped in a low-reward regime. $N=2$ performs the best with fastest convergence in terms of number of real trajectories used. Using higher N shows slower convergence. Note that the convergence also depends on the dimension of the simulation parameter space Ω — we expect $N > 2$ is needed for the best convergence rate once the dimension increases from 4 used in the pushing task.

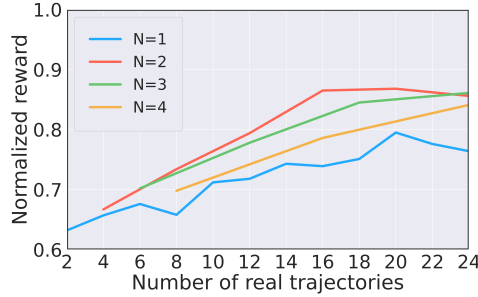


Figure A5: Task performance convergence with respect to the number of real trajectories used with varying N , the number of simulation parameter distributions adapting simultaneously in Phase 2 with the Light bottle in the pushing task.

Sensitivity analysis on adaptation step size. Adaption step size δ can affect the task performance convergence too — δ being too low can cause slow convergence, while δ being too high can prevent convergence since the simulation parameter distribution can “overshoot” the optimal one by a large margin. Fig. A6 shows the effect of adaptation step size ranging from 0.05 to 0.20 in OOD-1 setting of the double pendulum task. $\delta = 0.10$ performs the best while $\delta = 0.05$ shows slower convergence. $\delta = 0.15$ also achieves similar asymptotic performance but the reward is less unstable during adaptation, while with $\delta = 0.20$ the reward does not converge at all.

Comparison of simulation runtime. Compared to Sys-ID baselines, AdaptSim requires significantly longer simulation runtime for training the adaptation policy in Phase 1. For example: SysID-Bayes uses roughly 6 hours of simulation walltime to perform 10 iterations of adaptation in the scooping task while AdaptSim would take 36 hours for Phase 1, and 30 minutes for Phase 2 (i.e., 3 minutes per iteration), using the same computation setup. However, we re-use the same adaptation policy for different food pieces in the scooping task, which amortizes the simulation cost.

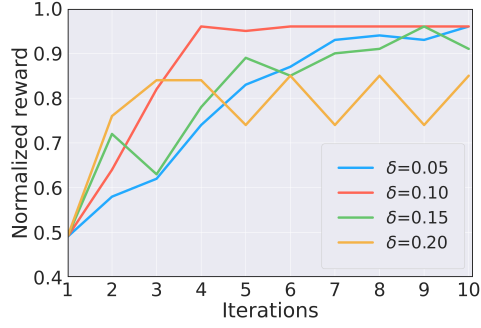


Figure A6: Normalized reward at each adaptation iteration using different adaptation step size δ , in OOD-1 setting of the pendulum task.

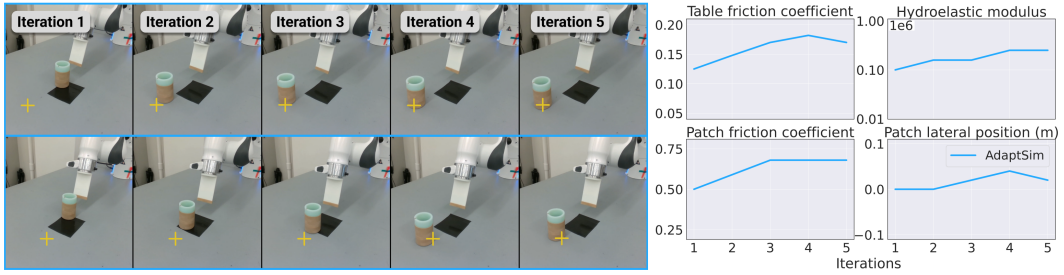


Figure A7: Adaptation results of the pushing task with two different target locations (yellow cross, top and bottom rows) over iterations. The right figure shows the inferred simulation parameter distribution (mean only).

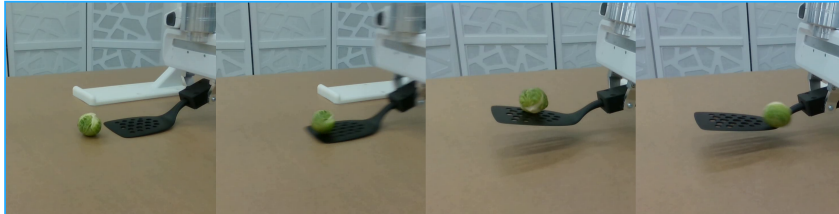


Figure A8: AdaptSim fails to synthesize a task policy for scooping up Brussels sprout. We consider such environment extremely OOD from the simulation domain.

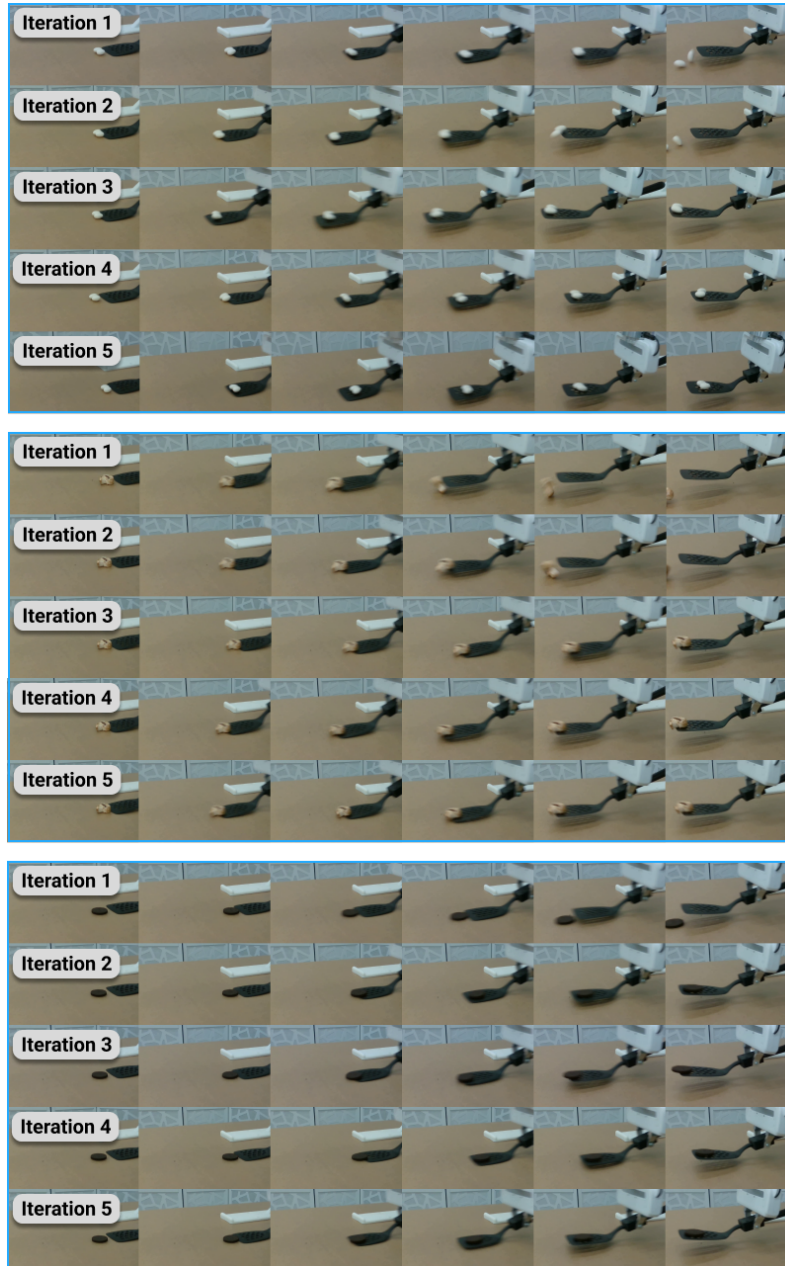


Figure A9: Adaptation results of scooping up (top) chocolate raisins, (middle) mushroom slice, and (bottom) Oreo cookie with AdaptSim.