APPENDIX

VII. ADDITIONAL REAL WORLD RESULTS AND ANALYSIS

A. Detailed Free Hand Motion Planning Results

In this section we perform additional analysis of the free hand motion planning results from the main paper. We include a more detailed version of the main result table (Tab. III). In this table, we additionally include the average (open loop) planning time per method and the average rate of safety violations. Safety violations are defined to occur where there are collisions, the robot hits its joint limits or there are torque limit errors. The open loop planning time for neural methods such as ours or MπNets involves simply measuring the total time taken for rolling out the policy and test time optimization (TTO). We find that sampling-based planners in general never collide when executed. If they produce a safety violation, it is only because they find a trajectory that is infeasible for the robot to execute on the hardware, due to joint or torque limit errors. Neural motion planning methods have much higher collision rates, though Neural MP has a significantly lower collision rate than MπNets, which we attribute to test-time optimization pruning out bad trajectories. We also note that not all collisions are created equal: some are slight, lightly grazing the environment objects while still achieving the goal, while others can be catastrophic, colliding heavily into the environment. In general, we found that our method tends to produce trajectories that may have slight collisions, though most of these are pruned out by TTO. With regards to planning time, MπNets is the fastest method, as our method expends additional compute rolling out 100x more trajectories and then selecting the best one using SDF-based collision checking.

	Bins (↑)	Shelf (↑)	Articulated (↑)	Avg. Success Rate (↑)	Avg. Planning Time (↓)	Avg. Safety Viol. Rate (↓)
Sampling-based Planning:						
`AIT*-80s [10]`	93.75	75	50.0	72.92	80	**0**
`AIT*-10s (fast) [10]`	75.0	37.5	25.0	45.83	10	2.1
Neural:						
`MπNets [42]`	18.75	25.0	6.25	16.67	**1.0**	18.75
`Ours`	**100**	**100**	**87.5**	**95.83**	3.9	4.2

TABLE III: Neural MP performs best across tasks for free-hand motion planning, demonstrating greater improvement as the task complexity grows.

B. Detailed In-hand Motion Planning Results

In this section, we extend the in-hand results shown in the main paper with additional baselines (AIT*-80s, AIT*-10s and MπNets). For this evaluation (see Tab. IV, we consider two of the four in-hand motion planning objects, namely joystick and book. We find sampling-based methods are able to perform in-hand motion planning quite well, matching the performance of our base policy as well as our method without Objaverse data. We also see that MπNets is unable to perform in-hand motion planning on any of the evaluated tasks. This is likely because that network was not trained on data with objects in-hand, demonstrating the importance of including in-hand data when training neural motion planners. Finally, there is a significant gap in performance between our method with and without test-time optimization; pruning out colliding trajectories at test time is crucial for achieving high success rates on motion planning tasks.

	Book (↑)	Joystick (↑)	Avg. Success Rate (↑)	Avg. Planning Time (↓)	Avg. Safety Viol. Rate (↓)
Sampling-based Planning:					
`AIT*-80s [10]`	50	50	50	80	**0**
`AIT*-10s (fast) [10]`	25	50	37.5	10	**0**
Neural:					
`MπNets [42]`	0	0	0	1	37.5
Ours:					
`Ours` (no TTO)	25	**75**	50	**0.9**	50
`Ours` (no Objaverse)	50	50	50	3.9	50
`Ours`	**100**	**75**	**87.5**	3.9	12.5

TABLE IV: Neural MP performs best across tasks for in-hand motion planning, demonstrating greater improvement as the in-hand object becomes more challenging.

C. Test-time Optimization Analysis

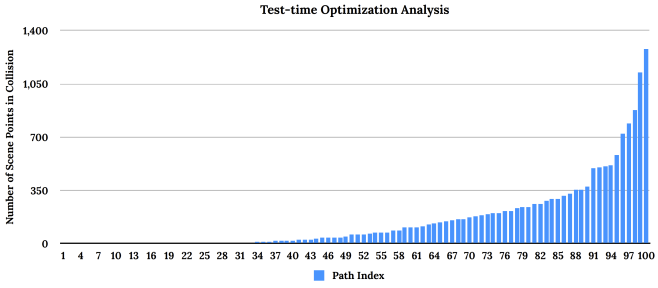


Fig. 5: **Test-time Optimization Analysis** For the Bins Scene 1 task, we plot the number of points in collision across 100 sampled trajectories from the model. 25% of the trajectories are completely collision free and we select a trajectory execute from that subset.

To analyze what the test-time optimization procedure is doing, we first note that the base policy can sometimes produce slight collisions with the environment due to the imprecision of regression. As a result, when sampling from the policy, it is often likely that the policy will lightly graze objects which will count as failures when motion planning. We visualize a set of trajectories sampled from the policy here on our website for the real-world bins task. Observe that for some of the trajectories, the policy slightly intersects with the bin which would cause it to fail when executing in the real world, while for others it simply passes over the bin completely without colliding. We estimate the robot-scene intersection of all of these trajectories by comparing the robot SDF to the scene point-cloud and plot the range of values in Fig. 5. We observe that 25% of trajectories do not collide with the environment, and we select for those. In principle, one could further optimize by selecting the trajectory that is furthest from the scene (using the SDF). In practice, we did not find this necessary and that selecting the first trajectory among those with the fewest expected collisions performed quite well in our experiments.

VIII. Ablations

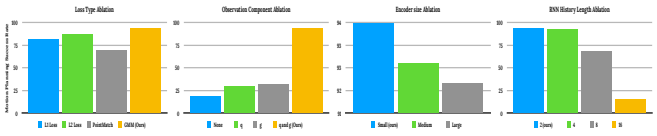


Fig. 6: **Ablation Results** We evaluate four different components of Neural MP, loss type (*left*), observation components (*middle left*), encoder sizes (*middle right*), and RNN history length (*right*). We validate that our design decisions produce measurable improvements in motion planning success rates.

We run additional ablations analyzing components of our method in simulation using a subset of our dataset (100K trajectories) and include additional details for experiments discussed in the main paper.

Loss Types For training objective, we evaluate 4 different options: GMM log likelihood (ours), MSE loss, L1 loss, and PointMatch loss (MπNets). PointMatch loss involves computing the l2 distance between the goal and the predicted end-effector pose using 1024 key-points. We plot the results on held out scenes in Fig. 6. We find that GMM (ours) outperforms L2 loss, L1 loss, and PointMatch Loss (MπNets) by (7%, 12%, and 24%) respectively. One reason this may be the case is that sampling-based motion planners produce highly multi-modal trajectories: they can output entirely different trajectories for the same start and goal pair when sampled multiple times. Since Gaussian Mixture Models are generally more capable of capturing multi-modal distributions, they can hence fit our dataset well. At the same time, the PointMatch [42] loss struggles significantly on our data: it cannot distinguish between 0 and 180 degree flipped end-effector orientations, resulting in many failures due to incorrect end-effector orientations.

Observation Components We evaluate whether our choice of observation components impacts the Neural MP's performance. In theory, the network should be able to learn as well from the point-cloud alone as when the proprioception is included, as the point-cloud contains a densely sampled point-cloud of the current and goal robot configurations. However, in practice, we find that this is not the case. Instead, removing either q or g or both severely harms performance as seen in Fig. 6. We hypothesize that including the proprioception provides a richer signal for the correct delta action to take.

RNN History Length In our experiments, we chose a history length of 2 for the RNN, after sweeping over values of 2, 4, 8, 16 based on performance. From Fig. 6 we see history length 2 achieves the best performance at 94%, while using lengths 4, 8 and 16 achieve progressively decreasing success rates (92.67, 68, 14.67). One possible reason for this is that since point-clouds are already very dense representations that cover the scene quite well, the partial observability during training time is fairly low. A shorter history length also leads to faster training, due to smaller batches and fewer RNN unrolling steps.

Encoder Size Finally, we briefly evaluate whether encoder size is important when training large-scale neural motion planners. We train 3 different size models: small (4M params), medium (8M params) and large (16M params). From the results in Fig. 6, we find that the encoder size does not affect performance by a significant margin (94%, 93%, 92%) respectively and that the smallest model in fact performs best. Based on these results, we opt to use the small, 4M param model in our experiments.

Neural MP-MLP	Neural MP-LSTM	Neural MP-Transformer	Neural MP-ACT
65.0	82.5	**85.0**	47.5

TABLE V: Ablation of different architecture choices for the action decoder. We find that LSTMs and Transformers comparably while LSTMs boast faster inference times.

Architecture Ablation In this experiment, we evaluate how different sequence modelling methods (Transformers and ACT [54]) and simpler action decoders such as MLPs compare against our design choice of using an LSTM. All methods are trained with the same dataset (of 1M trajectories), with the same encoder and GMM output distribution (with the exception of ACT which uses an L1 loss as per the ACT paper). We then evaluate them on held out motion planning tasks (Fig. V which are replicas of our real-world tasks (Bins and Shelf). We note several findings: 1) ACT performs poorly, largely due to its design choice of using an L1 loss which prevents it from handling planner multi-modality effectively, 2) Neural MP with an MLP action decoder also performs significantly worse than LSTMs and Transformers, as it is unable to use history information effectively to reason about the next action 3) Transformers and LSTMs perform similarly, with the Transformer variant performing marginally better, but with significantly slower inference time (2x). Hence we opt to use LSTM policies for our experimental evaluation, but certainly our method is amenable to any choice of sequence modeling architecture that performs well and has fast inference.

Neural MP-MotionBenchMaker	Neural MP-MπNets	Neural MP
0	32.5	82.5

TABLE VI: Comparing different methods for generating datasets for motion planning. We find that policies trained on our data generalize best to held out scenes.

Dataset Ablation Finally, we evaluate the quality of different dataset generation approaches for producing generalist neural motion planners. We do so by training policies on three different datasets (Neural MP, MπNets [42], and MotionBenchMaker [40]) and evaluated on held out motion planning tasks in simulation. We train each model to convergence for 10K epochs and then execute trajectories on two held out tasks that mirror our real world tasks: RealBins and RealShelf. For fairness, we do not include any Objaverse meshes in these tasks, since MPiNets and MotionBenchMaker only have primitive objects. Still, we find that our dataset performs best by a wide margin (Tab. VI). In general,

we found that policies trained on MotionBenchMaker do not generalize well. As mentioned in the related works section, this dataset lacks the realism and diversity necessary to train policies that can generalize to held out motion planning scenes.

IX. Procedural Scene Generation Details

In this section we provide additional details regarding the data generation methods we develop for training large scale neural motion planners.

A. Procedural Scene Generation

We formalize our procedural scene generation as a composition of randomly generated parameteric assets and sampled Objaverse meshes in Alg. 1

Objaverse sampling details The Objaverse are sampled in the task-relevant sampling location of the programmatic asset(s) in the scene, such as between shelf rungs, inside cubbies or within cabinets. Similar to the programmatic assets, these Objaverse assets are also sampled from a category generator $X_{obj}(\mathbf{p})$. Here the parameter p specifies the size, position, orientation of the object as well as task-relevant sampling location of the object in the scene, such as between shelf rungs, inside cubbies or within cabinets. As discussed in the main paper, we propose an approach that iteratively adds assets to a scene by adjusting their position using the effective collision normal vector, computed from the existing assets in the scene. We detail the steps for doing this in Alg. 1.

B. Motion Planner Experts

We use three techniques to improve the data generation throughput when imitating motion planners at scale.

Hindsight Relabeling Tight-space to tight-space problems are the most challenging, particularly for sampling-based planners, often requiring significant planning time (up to 120 seconds) for the planner to find a solution. For some problems, the expert planner is unable to find an exact solution and instead produces approximate solutions. Instead of discarding these, note that we use a goal-conditioned imitation learning framework, where we can simply execute the trajectories in simulation and relabel the observed final state as the new goal.

Reversibility We further improve our data generation throughput by observing that since motion planners inherently produce collision-free paths, the process is reversible, at least in simulation. This allows us to double our data throughput by reversing expert trajectories and re-calculating delta actions accordingly. Additionally, for a neural motion planner to be useful for practical manipulation tasks, it must be able to generate collision free plans for the robot even when it is holding objects. To enable such functionality, we augment our data generation process with trajectories where objects are spawned between the grippers of the robot end effector. There are transformed along with the end-effector during planning in simulation. We consider the object as part of the robot for collision checking and for the sake of our visual observations. In order to handle diverse objects that the robot might have to move with at inference time, we perform significant randomization of the in-hand object that we spawn in simulation. Specifically, we sample this object from the primitive categories of boxes, cylinders or spheres, or even from Objaverse meshes of everyday articles. We randomize the scale of the object between 3 and 30 cm along the longest dimension, and sample random starting locations within a 5cm cube around the end-effector mid-point between grippers.

Smoothing Importantly, we found that naively imitating the output of the planner performs poorly in practice as the planner output is not well suited for learning. Specifically, plans produced by AIT* often result in way-points that are far apart, creating large action jumps and sparse data coverage, making it difficult to for networks to fit the data. To address this issue, we perform smoothing using cubic spline interpolation while enforcing velocity and acceleration limits. The implementation from MπNets performs well in practice, smoothing to a fixed 50 timesteps with a max spacing of 0.1 radians. In general, we found that smoothing is crucial for learning performance as it ensures the maximum action size is small and thus easier for the network to fit to.

C. Data Pipeline Parameters and Compute

In Table VII, we provide a detailed list of all the parameters used in generating the data to train our model.

Compute In order to collect a vast data of motion planning trajectories, we parallelize data collection across a cluster of 2K CPUs. It takes approximately 3.5 days to collect 1M trajectories.

X. Network Training Details

We first describe additional details regarding our neural policy, and then discuss how it is trained. Following the design decisions of MπNets [42], we construct a segmented point-cloud for the robot, consisting of the robot point-cloud, the target goal robot point-cloud and the obstacle point-cloud. Here we note two key differences from MπNets: 1) our network conditioned on the target joint angles, while MπNets only does so through the segmented point-cloud, 2) we condition on the target joint angles, not end-effector pose, decisions that we found improved adherence to the overall target configuration. For in-hand motion planning, we extend this representation by considering the object in-hand as part of the robot for the purpose of segmentation.

We include a hyper-parameter list for our neural motion planner in Table VIII. We train a 20M parameter neural network across our dataset of 1M trajectories. The PointNet++ encoder is 4M parameters and outputs an embedding of dimension 1024. We concatenate this embedding with the encoded q_t and g vectors and pass this into the 16M parameter LSTM decoder. The decoder outputs weights, means, and standard deviations of the 5 GMM modes. We then train the model with negative log likelihood loss for 4.5M gradient steps, which takes 2 days on a 4090 GPU with batch size of 16.

Hyper-parameter	Value
General Motion Planning Parameters	
collision checking distance	1cm
tight space configuration ratio	50%
dataset size	1M trajectories
minimum motion planning time	20s
maximum motion planning time	80s
General Obstacle Parameters	
in hand object ratio	0.5
in hand object size range	[[0.03, 0.03, 0.03], [0.3, 0.3, 0.3]]
in hand object xyz range	[[-0.05, -0.05, 0.], [0.05, 0.05, 0.05]]
min obstacle size	0.1
max obstacle size	0.3
table dim ranges	[[0.6, 1], [1.0, 1.5], [0.05, 0.15]]
table height range	[-0.3, 0.3]
num shelves range	[0, 3]
num open boxes range	[0, 3]
num cubbys range	[0, 1]
num microwaves range	[0, 3]
num dishwashers range	[0, 3]
num cabinets range	[0, 3]
Objaverse Mesh Parameters	
scale range	[0.2, 0.4]
x pos range	[0.2, 0.4]
y pos range	[-0.4, 0.4]
number of mesh objects per programmatic asset	[0, 3]
number of mesh objects on the table	[0, 5]
Table Parameters	
width range	[0.8, 1.2]
depth range	[0.4, 0.6]
height range	[0.35, 0.5]
thickness range	[0.03, 0.07]
leg thickness range	[0.03, 0.07]
leg margin range	[0.05, 0.15]
position range	[[0, 0.8], [-0.6, 0.6]]
z axis rotation range	[0, 3.14]
Shelf Parameters	
width range	[0.5, 1]
depth range	[0.2, 0.5]
height range	[0.5, 1.2]
num boards range	[3, 5]
board thickness range	[0.02, 0.05]
backboard thickness range	[0.0, 0.05]
num vertical boards range	[0, 3]
num side columns range	[0, 4]
column thickness range	[0.02, 0.05]
position range	[[0, 0.8], [-0.6, 0.6]]
z axis rotation range	[-1.57, 0]
Open Box Parameters	
width range	[0.2, 0.7]
depth range	[0.2, 0.7]
height range	[0.3, 0.5]
thickness range	[0.02, 0.06]
front scale range	[0.6, 1]
position range	[[0.0, 0.8], [-0.6, 0.6]]
z axis rotation range	[-1.57, 0.0]

Hyper-parameter	Value
Cubby Parameters	
cubby left range	[0.4, 0.1]
cubby right range	[-0.4, 0.1]
cubby top range	[0.85, 0.35]
cubby bottom range	[0.0, 0.1]
cubby front range	[0.8, 0.1]
cubby width range	[0.35, 0.2]
cubby horizontal middle board z axis shift range	[0.45, 0.1]
cubby vertical middle board y axis shift range	[0.0, 0.1]
board thickness range	[0.02, 0.01]
external rotation range	[0, 1.57]
internal rotation range	[0, 0.5]
num shelves range	[3, 5]
Microwave Parameters	
width range	[0.3, 0.6]
depth range	[0.3, 0.6]
height range	[0.3, 0.6]
thickness range	[0.01, 0.02]
display panel width range	[0.05, 0.15]
distance range	[0.5, 0.8]
external z axis rotation range	[-2.36, -0.79]
internal z axis rotation range	[-0.15, 0.15]
Dishwasher Parameters	
width range	[0.4, 0.6]
depth range	[0.3, 0.4]
height range	[0.5, 0.7]
control panel height range	[0.1, 0.2]
foot panel height range	[0.1, 0.2]
wall thickness range	[0.01, 0.02]
opening angle range	[0.5, 1.57]
distance range	[0.6, 1.0]
external z axis rotation range	[-2.36, -0.79]
internal z axis rotation range	[-0.15, 0.15]
Cabinet Parameters	
width range	[0.5, 0.8]
depth range	[0.25, 0.4]
height range	[0.6, 1.0]
wall thickness range	[0.01, 0.02]
left opening angle range	[0.7, 1.57]
right opening angle range	[0.7, 1.57]
distance range	[0.6, 1.0]
external z axis rotation range	[-2.36, -0.79]
internal z axis rotation range	[-0.15, 0.15]

TABLE VII: **Data Generation Hyper-parameters** We provide a detailed list of hyper-parameters used to procedurally generate a vast variety of scenes in simulation.

Hyper-parameter	Value

PointNet++ Architecture	`PointnetSAModule(`
	`  npoint=128,`
	`  radius=0.05,`
	`  nsample=64,`
	`  mlp=[1, 64, 64, 64],`
	`)`
	`PointnetSAModule(`
	`  npoint=64,`
	`  radius=0.3,`
	`  nsample=64,`
	`  mlp=[64, 128, 128, 256],`
	`)`
	`PointnetSAModule(`
	`  nsample=64,`
	`  mlp=[256, 512, 512],`
	`)`
	`MLP(`
	`  Linear(512, 2048),`
	`  GroupNorm(16, 2048),`
	`  LeakyReLU,`
	`  Linear(2048, 1024),`
	`  GroupNorm(16, 1024),`
	`  LeakyReLU,`
	`  Linear(1024, 1024)`
	`)`

LSTM	1024 hidden dim, 2 layers
Inputs	q_t, g, PCD_t
Batch Size	16
Learning Rate	0.0001
GMM	5 modes
Sequence Length (seq length)	2
Point Cloud Parameters	
Number of Robot / Goal Point-cloud Points	2048
Number of Obstacle Point-cloud Points	4096

TABLE VIII: Hyper-parameters for the model

Algorithm 2 Open-Loop Execution of Neural MP

Input: Neural MP π_θ, segmentor $\mathscr{S}$, initial angles q_0, scene point-cloud PCD_{full}, goal g, horizon H
Output: Executed trajectory on the robot
Initialize: Timestep $t \leftarrow 0$
Initialize: Trajectory $\tau \leftarrow \{\}$
$PCD_0 \leftarrow \mathscr{S}(PCD_{full}) \cup PCD_{q_0} \cup PCD_g$
while goal g not reached & $t < H$ **do**
    $a_t \sim \pi_\theta(q_{t-1}, PCD_{t-1}, g)$
    $q_t \leftarrow q_{t-1} + a_t$
    $PCD_t \leftarrow (PCD_{t_1} \setminus PCD_{q_{t-1}}) \cup PCD_{q_t}$
    $t \leftarrow t+1$
    $\tau \leftarrow \tau + a_t$
end while
Execute the τ open loop on the robot.

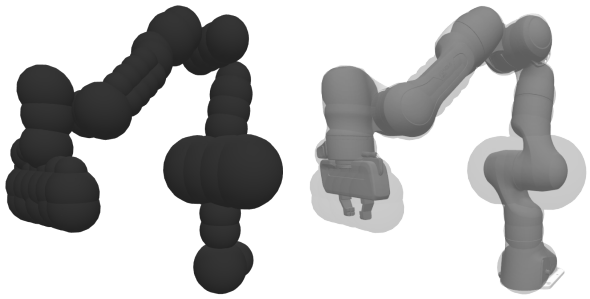


Fig. 7: We visualize the spherical representation on the left and overlay it on the robot mesh on the right.

XI. Real World Setup Details

In this section, we describe our real world robot setup and tasks in detail and perform analysis on the perception used for operating our policies.

A. Real Robot Setup

Hardware For all of our experiments, we use a Franka Emika Panda Robot, which is a 7 degree of freedom manipulator arm. We control the robot using the manimo library (https://github.com/AGI-Labs/manimo) and perform all of experiments using their joint position controller with the default PD gains. The robot is mounted to a fixed base pedestal behind a desk of size .762m by 1.22m with variable height. For sensing, we use four extrinsically calibrated depth cameras, Intel Realsense 435 / 435i, placed around the scene in order to accurately capture the environment. We project the depth maps from each camera into 3D and combine the individual point-clouds into a single scene representation. We then post-process the point-cloud by cropping it to the workspace, filtering outliers and denoising, and sub-sampling a set of 4096 points. This processed point-cloud is then used as input to the policy.

Representation Collision Checking and Segmentation In order to perform real world collision checking and robot point-cloud segmentation, we require a representation of the robot to check intersections with the scene (collision checking) and to filter out robot points from the scene point-cloud (segmentation). While the robot mesh is the ideal candidate for these operations, it is far too slow to run in real time. Instead, we approximate the robot mesh as spheres (visualized in Fig. 7) as we found this performs well in practice while operating an order of magnitude faster. We use 56 spheres in total to approximate the links of the robot as well as the end-effector and gripper. These have radii ranging from 2cm to 10cm and are defined relative to the center of mass of the link. This representation is a conservative one: it encapsulates the robot mesh, which is desirable for segmentation as this helps account for sensing errors which would place robot points outside of the robot mesh.

Robot Segmentation In order to perform robot segmentation in the real world, we use the spherical representation to filter out robot points in the scene, so only the obstacle point-cloud remains. Doing so requires computing the Signed Distance Function (SDF) of the robot representation and then checking the scene point-cloud against it, removing points from the point-cloud in which SDF value is less than threshold ε. For the spherical representation, the SDF computation is efficient: for a sphere with center C and radius r, the SDF of point x is simply $||x-C||_2 - r$. In our experiments, we use a threshold ε of 1cm. We then replace the removed points with points sampled from the robot mesh of the robot. This is done by pre-sampling a robot point-cloud from the robot mesh at the default configuration, then performing forward kinematics using the current joint angles q_t and transforming the robot point-cloud accordingly. Replacing the real robot point-cloud with this sampled point-cloud ensures that the only difference between sim and real is the obstacle point-cloud.

Real-world Collision Checking Given the SDF, collision checking is also straightforward, we denote the robot in collision if any point in the scene point-cloud (this is after robot segmentation) has SDF value less than 1cm. Note this means that first state is by definition collision free. Also, this technique will not hold if performing closed loop planning, in that case this method would always denote the state as collision free as the points with SDF value less than 1cm would be segmented out for each intermediate point-cloud.

Open Loop Deployment For open-loop execution of neural motion planners, we execute the following steps: 1) generate the segmented point-cloud at the first frame, 2) predict the next trajectory way-point by computing a forward pass through the network and sampling an action, 3) update the current robot point-cloud with mesh-sampled point-cloud at the predicted way-point, and 4) repeat until goal reaching success or maximum rollout length is reached. The entire trajectory is then executed on the robot after the rollout. Please see Alg. 2 for a more detailed description of our open-loop deployment method.

B. Tasks

Bins This task requires the neural planner to perform collision avoidance when moving in-between, around and inside two different industrial bins pictured in the first row of

Fig. 9. We randomize the position and orientation of the bins over the table and include the following objects as additional obstacles for the robot to avoid: toaster, doll, basketball, bin cap, and white box. The small bin is of size 70cm x 50cm x 25cm. The larger bin is of size 70cm x 50cm x 37cm. The bins are placed at two sides of the table. Between tasks, we randomize the orientation of the bins between 0 and 45 degrees and we swap the bin ordering (which bin is on the left vs. the right). The bins are placed 45cm in front of the robot, and shifted 60cm left/right.

Shelf This task tests the agent's ability to handle horizontal obstacles (the rungs of the shelf) while maneuvering in tighter spaces (row two in Fig. 9). We randomize the size of the shelf (by changing the number of layers in the shelf from 3 to 2) as well as the position and orientation (anywhere at least .8m away from the robot) with 0 or 30 degrees orientation. The obstacles for this task include the toaster, basketball, baskets, an amazon box and an action figure which increase the difficulty. The shelf obstacle itself is of size 35cm x 80cm x 95cm.

Articulated We extend our evaluation to a more complex primary obstacle, the cabinet, which contains one drawer and two doors and tight internal spaces with small cubby holes (row three of Fig. 9). We randomize the position of the entire cabinet over the table, the joint positions of the drawer and doors and the sizes of the cubby holes. The obstacles for this task are xbox controller box, gpu, action figure, food toy, books and board game box. The size of the cabinet is 40cm x 75cm x 80cm. The size of the top drawer is 30cm x 65cm x 12cm. The size of the cubbies is 35cm x 35cm x 25cm. The drawer has an opening range of 0-30cm and the doors open between 0 and 180 degrees.

In-Hand Motion Planning In this task (shown in row four of Fig. 9), the planner needs to reason about collisions with not only the robot and the environment, but the held object too. We initialize the robot with an object grasped in-hand and run motion planning to reach a target configuration. For this task, we fix the obstacle (shelf) and its position (directly 80cm in front of the robot), instead randomizing across in-hand objects and configurations. We select four objects that vary significantly in size and shape: Xbox controller (18cm x 15cm x 8cm), book (17cm x 23cm x 5cm), toy sword (65cm x 10cm x 2cm), and board game (25cm x 25cm x 6cm). For this evaluation, we assume the object is already grasped by the robot, and the robot must just move with the object in-hand while maintaining its grasp.

C. Perception Visualization and Analysis

We compare point-clouds from simulation and the real world for the Bins and Shelf task and analyze their properties. We replicate Bins Scene 4 and Shelf Scene 1 in simulation: simply measure the dimensions and positions of the real world objects and set those dimensions in simulation using the OpenBox and Shelf procedural assets. As seen in Fig. 8, simulated point-clouds are far cleaner than those in the real world, which are noisy and perhaps more importantly, partial. The real-world point-clouds often have portions missing due to camera coverage as for large objects it is challenging to cover the scene well while remaining within the depth camera operating range. However, we find that our policy is still able to able operate well in these scenes, as PointNet++ is capable of handling partial point-clouds and is trained on a diverse dataset containing many variations of boxes and shelves with different types and number of components as well as sizes, which may enable the policy to generalize to partial boxes and shelves observed in the real world.

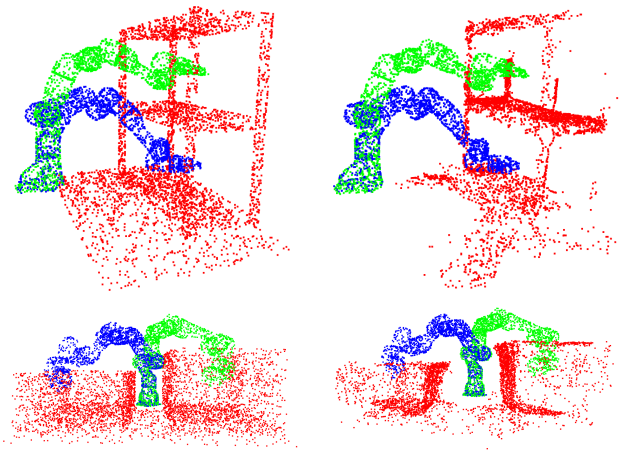
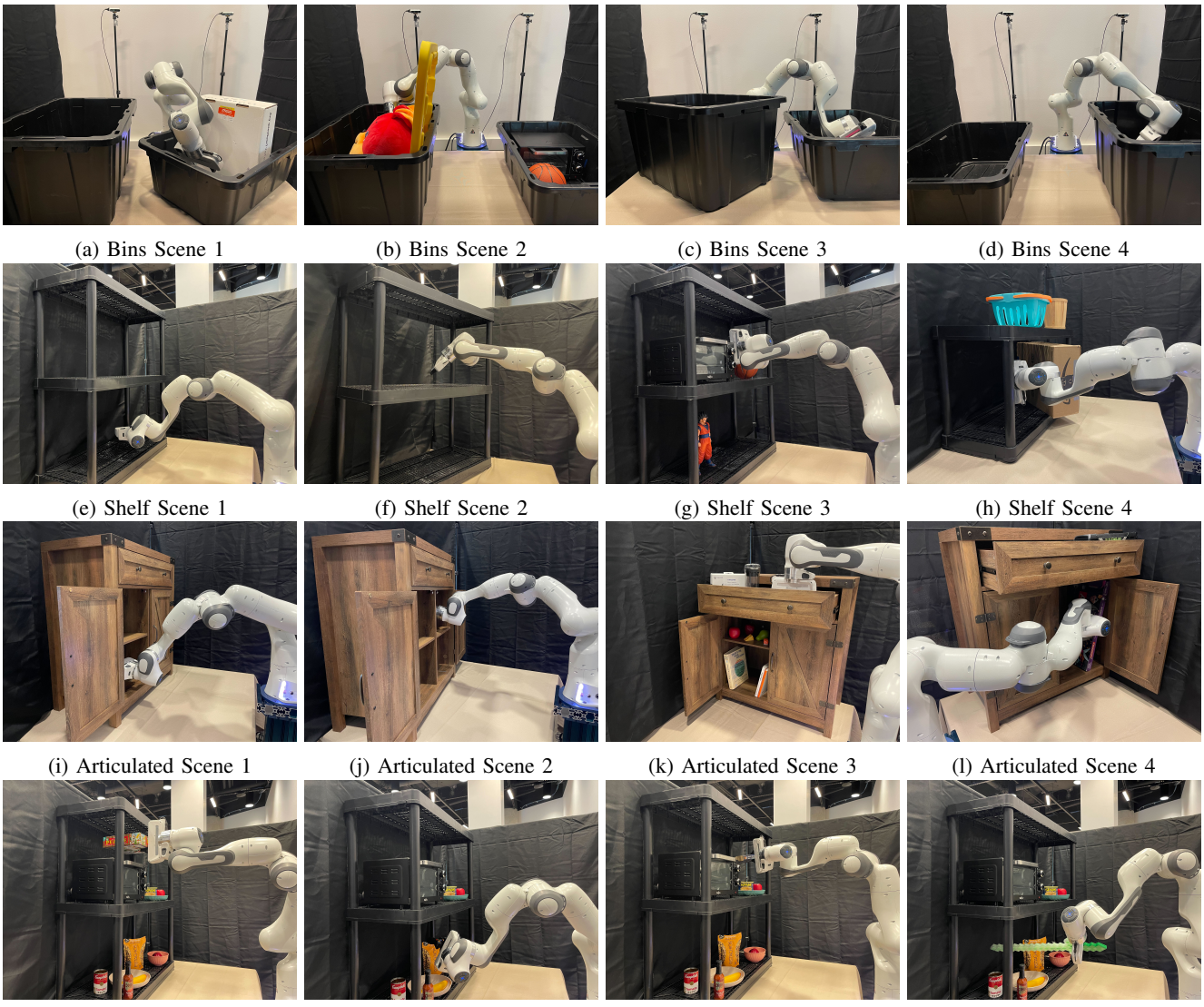


Fig. 8: **Visualization of Sim and Real point-clouds**: We visualize point-clouds of the Bins and Shelf task in sim and real, in the same poses. Due to noise in depth sensing, the real world point-clouds have significantly more deformations, yet our policy generalizes well to these tasks.



(a) Bins Scene 1 (b) Bins Scene 2 (c) Bins Scene 3 (d) Bins Scene 4

(e) Shelf Scene 1 (f) Shelf Scene 2 (g) Shelf Scene 3 (h) Shelf Scene 4

(i) Articulated Scene 1 (j) Articulated Scene 2 (k) Articulated Scene 3 (l) Articulated Scene 4

(m) In Hand Object 1 (n) In Hand Object 2 (o) In Hand Object 3 (p) In Hand Object 4

Fig. 9: Images of our 16 evaluation scenes.