Appendix A Implementation Details

A.1 Sensor Evaluation Details

We evaluate two state-of-the-art dense tactile sensors such as the GelSlim 3.0 [34] and the Soft Bubbles [2]. We compare the sensors in terms of deformations and the transmitted forces. We evaluate the sensors' compliance since it produces tool motion with respect to the robot's end-effector.

Both sensors have dot patterns imprinted on their contact surfaces to enable surface tracking and deformation estimation. The imprinted dots can be detected using segmentation methods. We use Gaussian-weighted adaptive thresholding to binarize the sensor images and then find the dot contours. We use Gunnar-Farneback Dense Optical flow method [35] to track the dots motions, extract the deformation fields, and compute correspondences. We project the dots deformations into their spatial coordinates for the reference and deformed state and use the correspondences to compute each dot's deformation. To estimate the deformation perpendicular to the camera plane, we use the measured depth via their depth camera for the Bubbles sensors. For the GelSlims, we exploit the known rigid object geometry.

We compare the sensors response on two axial motions: Axial motion perpendicular to the grasp frame (Fig. 3 top) and a top-down motion along the sensor's contact plane main axis (Fig. 3 bottom). The robot is controlled in Cartesian impedance mode and commanded to move in increments of 5mm. We measure steady-state states. We use the robot to extract the motion axial force. For the deformation, we report the average of the 5% of the dots that deform the most.

A.2 Membrane Dynamics Model Implementation

We implement our membrane dynamics model in PyTorch. We use PyTorch Lightning for efficient and easy training and logging.

A.2.1 Architecture

Figure 4 illustrates our dynamics model. It is composed by 4 main elements:

- **Tactile Encoder**: The tactile encoder (Fig. A.1 left) embeds a $(2 \times 25 \times 20)$ image into a 15-dimensional vector. It is composed by 3 Convolutional Neural Network (CNN) with (5×5) kernels followed by 1 linear layer. The tactile encoder uses ReLU activations and batch normalizations.
- **Tactile Decoder**: The tactile decoder (Fig. A.1 right) attempts to invert the encoder. It is composed by a linear layer followed by 2 deconvolution layers. It also has ReLU activations and batch normalizations.
- **Membrane Dynamics Model**: As Fig. A.3 A shows, the membrane dynamics model is a 3-layer neural network with ReLU activations and hidden sizes of 200. Its input is a concatenation of the tactile embedding (15-dimensional), wrench (6-dimensional), grasp frame pose (6-dimensional), object embedding (10-dimensional), and robot action (4-dimensional). The grasp frame pose is represented as a position and axis-angle orientation. It predicts the next tactile embedding (15-dimensional), next wrench (6-dimensional), and the correction of the grasp frame pose (6-dimensional).
- **Object Embedding**: This is a PointNet [26] classifier network with last layer replaced by a 10-dimensional linear layer with no output activation to produce the desired embedding.

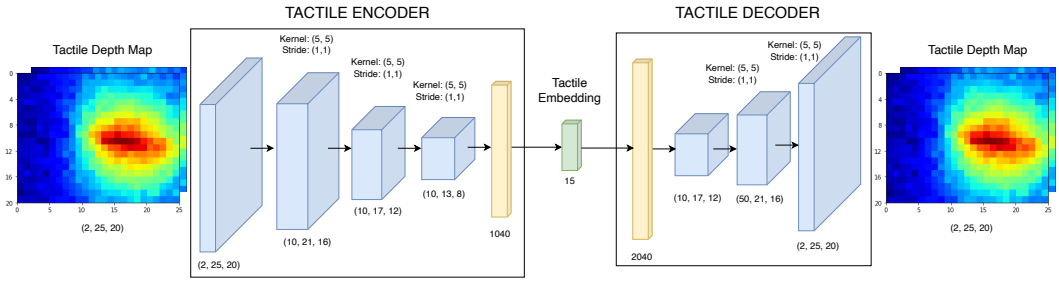

Figure A.1: **Tactile Projection Architecture** An encoder (left) maps the combined tactile depth maps for left and right bubbles into a 15-dimensional embedding. The decoder (right) reconstructs the tactile depth map from the embedding.

A.2.2 Training Details

Given a dataset $\mathcal{D} = \{(\boldsymbol{s}_t, \boldsymbol{z}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1})_i\}_{i=1}^N$, we perform a 80-20 train-validation split. We train our models using the train data until convergence and use the remaining validation data to asses the model performance to unseen data. We use model weights with the lower validation loss. We train our dynamics model in a two-step process. First we train the tactile embedding, using training data from multiple tasks. For the tactile embedding, we use a MSE reconstruction loss on the tactile depth maps.

$$\begin{aligned} \mathcal{L}_{\text{tactile}}(\hat{\boldsymbol{p}}_t, \boldsymbol{p}_t) &= \text{MSE}(\hat{\boldsymbol{p}}_{t+1}, \boldsymbol{p}_{t+1}) \\ \text{where} \quad \hat{\boldsymbol{p}}_{t+1} &= f_{\text{decode}}(f_{\text{encode}}(\boldsymbol{p}_t)) \end{aligned}$$

Once we have trained the tactile embedding, we freeze it and learn the membrane dynamics for each specific task. We use Adam optimization with a supervised loss on the predicted state composed of 3 MSE terms, one for each of the three sensory modalities our model predicts: tactile, wrenches, and poses. The losses are aggregated based on weights α_i to compensate for the discrepancies in units and scale of each sensory contribution. In our case:

$$\begin{aligned} \mathcal{L}_{\text{dyn}}(\hat{\boldsymbol{s}}_t, \boldsymbol{s}_t) = \alpha_1 \text{MSE}(\hat{\boldsymbol{p}}_{t+1}, \boldsymbol{p}_{t+1}) + \alpha_2 \text{MSE}(\hat{\boldsymbol{w}}_{t+1}, \boldsymbol{w}_{t+1}) + \alpha_3 \text{MSE}(\hat{\boldsymbol{r}}_{t+1}, \boldsymbol{r}_{t+1}) \\ \text{where} \quad \hat{\boldsymbol{p}}_{t+1}, \hat{\boldsymbol{w}}_{t+1}, \hat{\boldsymbol{r}}_{t+1} = f(\boldsymbol{p}_t, \boldsymbol{w}_t, \boldsymbol{r}_t, \boldsymbol{z}_t, \boldsymbol{a}_t) \end{aligned}$$

In practice, we found that our model predictions obtained best results with $\alpha_3 = 0$. Since our robot impedance is considerably stiff, the robot motions can be well approximated to be rigid. With this consideration, the only compliance in the system is the sensors' compliance.

Parameter Name	Parameter Symbol	Value
tactile loss weight	α_1	1
wrench loss weight	α_2	0.0001
robot pose loss weight	α_3	0
learning rate	λ	0.001
betas	(β_1, β_2)	$(0.9, 0.999)$

Table 2: **Dynamics Model Hyper-parameters:** Hyper-parameters used for training the dynamics model.

A.3 Observation Model Implementation

The observation model is the responsible to estimate the pose of the grasped object from the membrane state. Since we have a model of the object geometry, we use Iterative Closest Points (ICP) to estimate the best object pose from the membrane state. In particular, we use point-to-point ICP since both the membrane and the object geometry are collections of points.

First, we extract the membrane points that also belong to the object. We refer to such collection of points as imprint. To this end, we filter out all membrane points based on their deformation values. We select the top 10% of the deformations that deform at least 3mm. We also filter out points that are more than 15mm to the imprint point cluster, since they are noise.

Once we have extracted a clean imprint, we apply ICP to iteratively find the best transformation of the object model that minimizing the L2 distance with the imprint. We initialize the translation estimate as the mean of the imprint points. We also initialize the orientation to be randomly within 20° of the imprint main axis. We perform 20 ICP iterations. The resulted estimated tool pose is expressed with respect to the grasp frame, which is also the frame that the membrane points are registered. When in contact, we impose non-penetration to the estimated object pose and we project it to the contact manifold. Intuitively, the object pose is translated to have a shared point with the environment without compromising the ICP score. We detect contact based on the external wrench feedback. When the external wrench perpendicular to the environment surface normal is above a threshold, then the robot assumes that it is in contact with the environment. Typically, we set the threshold to be 1.5N.

Our observation model implementation is able to estimate multiple object poses for multiple membrane states in parallel. We have implemented the imprint detection and the ICP algorithm in PyTorch for fast and easy integration with the controller algorithm. This allows our algorithm to execute all operations in the GPU.

A.4 Controller Implementation

Since our dynamics models are PyTorch models, our controller implementation is based on `pytorch_mppi` [36]. This package exploits batched operations for fast and efficient real-time implementation. Dynamic model queries, cost computations, and sampling are done on a GPU. At each step, the controller samples action sequences around the nominal action sequence based on Gaussian noise. We initialize the nominal action sequence as the mean of the action space. Then, the dynamics model is queried and the membrane states are obtained along the sampled trajectories. Next, the object poses and transmitted forces for each trajectory are estimated using the observation model. The trajectory costs are computed comparing the predicted task states, composed by the object pose and transmitted force, with the desired ones. Finally, the nominal action sequence is updated based on an importance sum where the weight is computed from the trajectory cost. Intuitively, the lower the cost, i.e. the closer the state to the desire one, the higher the contribution to the control sequence update. The optimal action is sent to the robot and then it is discarded.

Table 4 reports average computation times for our controller implementation during task execution. Our experiments are executed in a system equipped with a NVIDIA GeForce RTX 3070 GPU and an AMD Ryzen 9 3900XT CPU.

The individual state cost is formulated in the task-state $\boldsymbol{x} = (\boldsymbol{q}, \boldsymbol{w})$ and its value is computed as:

$$\mathcal{J}(\boldsymbol{x}_t, \boldsymbol{a}_t) = (\boldsymbol{x}_t - \boldsymbol{x}_g)^T \mathbf{Q} (\boldsymbol{x}_t - \boldsymbol{x}_g) + \boldsymbol{a}_t^T \mathbf{R} \boldsymbol{a}_t$$
$$\text{where} \quad \mathbf{Q} = \mathbf{I}, \quad \mathbf{R} = \mathbf{0}$$

Considering the different magnitudes and dimensions in $\boldsymbol{x}_t = (\boldsymbol{q}_t, \boldsymbol{w}_t)$, we further reduce the above expression by splitting it into 2 terms as shown below. We set $\alpha_w = 0.0001$.

$$\mathcal{J}(\boldsymbol{x}_t) = \mathcal{L}_{\text{obj pose}}(\mathbf{q}_t, \mathbf{q}_g) + \alpha_w \text{MSE}(\boldsymbol{w_t}, \boldsymbol{w}_g)$$

For the object pose cost, we combine position and orientation terms in $\boldsymbol{q}$ by exploiting the known object geometry $\boldsymbol{z}$. Since the object geometry is a pointcloud, $\boldsymbol{z} = \{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_M\}$ where $\boldsymbol{z}_i \in \mathbb{R}^3$, we can transform each individual point with the $SE(3)$ transformation given by $\boldsymbol{q}$ and evaluate the distance between the point pose given by $\boldsymbol{q}_t$ and the goal pose $\boldsymbol{q}_g$ using the MSE metric.

$$\begin{aligned} \mathcal{L}_{\text{obj pose}}(\mathbf{q}_t, \mathbf{q}_g) &= \frac{1}{M} \sum_{i=1}^{M} \left[\mathbf{H}(\boldsymbol{q}_t)\tilde{\boldsymbol{z}}_i - \mathbf{H}(\boldsymbol{q}_g)\tilde{\boldsymbol{z}}_i\right]^T \left[\mathbf{H}(\boldsymbol{q}_t)\tilde{\boldsymbol{z}}_i - \mathbf{H}(\boldsymbol{q}_g)\tilde{\boldsymbol{z}}_i\right] - 1 \\ &= \frac{1}{M} \sum_{i=1}^{M} \tilde{\boldsymbol{z}}_i^T \left[\mathbf{H}(\boldsymbol{q}_t) - \mathbf{H}(\boldsymbol{q}_g)\right]^T \left[\mathbf{H}(\boldsymbol{q}_t) - \mathbf{H}(\boldsymbol{q}_g)\right] \tilde{\boldsymbol{z}}_i - 1 \end{aligned} \tag{1}$$

Where $\mathbf{H}(\boldsymbol{q}) \in SE(3)$ is the homogeneous transformation matrix associated to the pose $\boldsymbol{q}$ and $\tilde{\boldsymbol{z}}_i$ is the homogeneous vector $\tilde{\boldsymbol{z}}_i = \begin{bmatrix} \boldsymbol{z}_i^T & 1 \end{bmatrix}^T$.

Parameter Name	Parameter Symbol	Value
lambda	λ	0.01
horizon	T	2
number of samples	N	100
noise sigma	σ_{noise}	30%

Table 3: **Controller Hyper-parameters:** MPPI hyper-parameters used by our implementation.

Process Name	Computation Time [s]
Dynamic Model Queries	$4 \cdot 10^{-5}$
Pose Estimation	0.6
Cost Computation	0.4
Total Control Step	3.0

Table 4: **Computation Performance:** Time costs in seconds for our control implementation during task execution. See table 3 for the evaluated hyper-parameters.

A.5 Baselines Details

- **Bubble Linear Dynamics**: This baseline constrains the membrane pose dynamics so next states are obtained with a linear operation with no biases. (Fig. A.3 B). Therefore, the dynamics are:

$$\begin{bmatrix} \boldsymbol{p}^{\text{emb}}_{t+1} \\ \boldsymbol{w}_{t+1} \end{bmatrix} = \mathbf{A}_{\text{dyn}} \begin{bmatrix} \boldsymbol{p}^{\text{emb}}_{t} \\ \boldsymbol{w}_{t} \\ \boldsymbol{r}_{t} \\ \boldsymbol{z}_{t} \\ \boldsymbol{a}_{t} \end{bmatrix} \quad \text{where } \mathbf{A}_{\text{dyn}} \in \mathbb{R}^{21\times 41}, \quad \boldsymbol{p}^{\text{emb}}_{t} = f_{\text{encode}}(\boldsymbol{p}_t), \quad \boldsymbol{p}_{t+1} = f_{\text{decode}}(\boldsymbol{p}^{\text{emb}}_{t+1})$$

- **Object Pose Dynamics**: This baseline considers the object pose instead of the tactile signature (Fig. A.2). Therefore, the state in this case is $\boldsymbol{s}_{\text{obj},t} = (\boldsymbol{q}_t, \boldsymbol{w}_t, \boldsymbol{r}_t)$. The object pose dynamics are modelled similar to the *Bubble Dynamics* with a 2-hidden layers with 200 units (Fig. A.3 C). Also, similar to the membrane model training, here we train the model with a loss composed by three term, but instead on a MSE over the tactile signatures, we have a loss term on the object pose. This object pose loss is defined in equation 1.

$$\mathcal{L}_{\text{obj dyn}}(\hat{\boldsymbol{s}}_{\text{obj},t}, \boldsymbol{s}_{\text{obj},t}) = \alpha_1 \mathcal{L}_{\text{obj pose}}(\boldsymbol{q}_t, \hat{\boldsymbol{q}}_t) + \alpha_2 \text{MSE}(\hat{\boldsymbol{w}}_{t+1}, \boldsymbol{w}_{t+1}) + \alpha_3 \text{MSE}(\hat{\boldsymbol{r}}_{t+1}, \boldsymbol{r}_{t+1})$$
$$\text{where} \quad \hat{\boldsymbol{q}}_{t+1}, \hat{\boldsymbol{w}}_{t+1}, \hat{\boldsymbol{r}}_{t+1} = f_{\text{obj dyn}}(\boldsymbol{q}_t, \boldsymbol{w}_t, \boldsymbol{r}_t, \boldsymbol{z}_t, \boldsymbol{a}_t)$$

- **Fixed Model**: This baseline assumes that the membrane state remains constant ($\boldsymbol{p}_{t+1} = \boldsymbol{p}_t$), i.e. there is no relative motion between the object and the robot. It also assumes that there is no change in the external sensed forces ($\boldsymbol{w}_{t+1} = \boldsymbol{w}_t$). Future grasp frame poses $\boldsymbol{r}_{t+1}$ are predicted using a deterministic robot action model considering the robot geometry, the current grasp pose $\boldsymbol{r}_t$ and the action $\boldsymbol{a}_t$. This can be expressed as

$$\boldsymbol{s}_{t+1} = (\boldsymbol{p}_{t+1}, \boldsymbol{w}_{t+1}, \boldsymbol{r}_{t+1}) = f_{\text{fixed}}(\boldsymbol{p}_t, \boldsymbol{w}_t, \boldsymbol{r}_t, \boldsymbol{z}_t, \boldsymbol{a}_t) = (\boldsymbol{p}_t, \boldsymbol{w}_t, f_{\text{robot action}}(\boldsymbol{r}_t, \boldsymbol{a}_t))$$

- **Jacobian**: This baseline assumes that the object is rigidly attached to the environment. Therefore its pose w.r.t the environment does not change, i.e. $\boldsymbol{q}_{t+1} = \boldsymbol{q}_t$ and $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t$. As *Fixed Model*, future grasp frame poses $\boldsymbol{r}_{t+1}$ are predicted using a deterministic robot action model.

$$\boldsymbol{s}_{\text{obj},t+1} = (\boldsymbol{q}_{t+1}, \boldsymbol{w}_{t+1}, \boldsymbol{r}_{t+1}) = f_{\text{jacobian}}(\boldsymbol{q}_t, \boldsymbol{w}_t, \boldsymbol{r}_t, \boldsymbol{z}_t, \boldsymbol{a}_t) = (\boldsymbol{q}_t, \boldsymbol{w}_t, f_{\text{robot action}}(\boldsymbol{r}_t, \boldsymbol{a}_t))$$

- **Pseudo-random**: This baseline does not have a model at all. Instead, actions are randomly sampled from each task action space. Although the actions are random, they are constrained towards the goal. For drawing, pseudo-random actions constrain the robot motion to be toward the goal. For pivoting, the pseudo-random actions force motions that push against the table, i.e. actions that produce pivoting motions. This way we give an intuition about the task goal and prevent barren actions. We use rejection sampling to impose the constraints and discard actions that violate them.

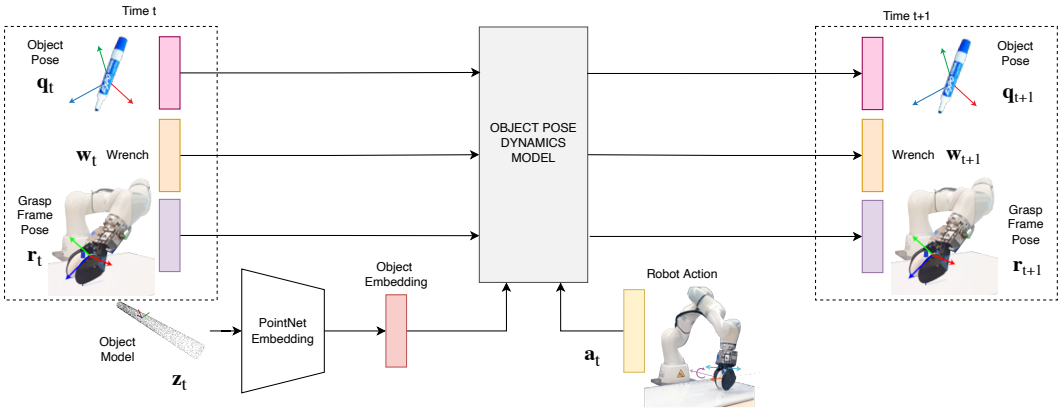

Figure A.2: **Object Pose Dynamics** This architecture replace the tactile depth map for the object pose. As membrane dynamics, object model is still embedded using a PointNet network. However, since here there is not tactile data, we do not need the tactile encoder-decoder networks.

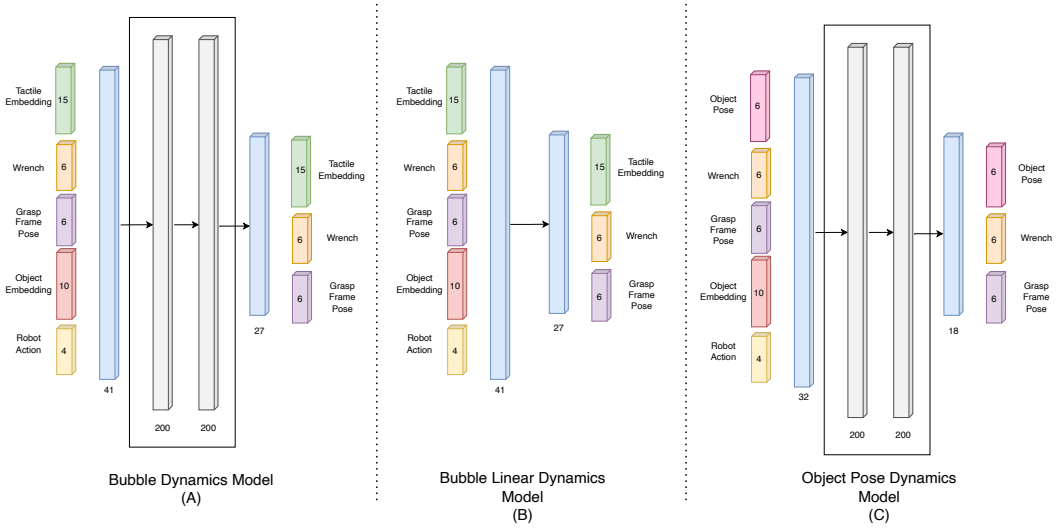

Figure A.3: **Dynamics Architecture Comparison** Bubble Dynamics model (A) and Object Pose Dynamics Model (C) are 3-layer neural networks. They both have 2 hidden-layers with 200 units. Bubble Linear Dynamics (B) predicts the output based on a single linear layer with no biases. (A) and (B) use tactile embeddings, while (C) replaces them with object poses.

A.6 Membrane Dynamics vs Object Pose Dynamics

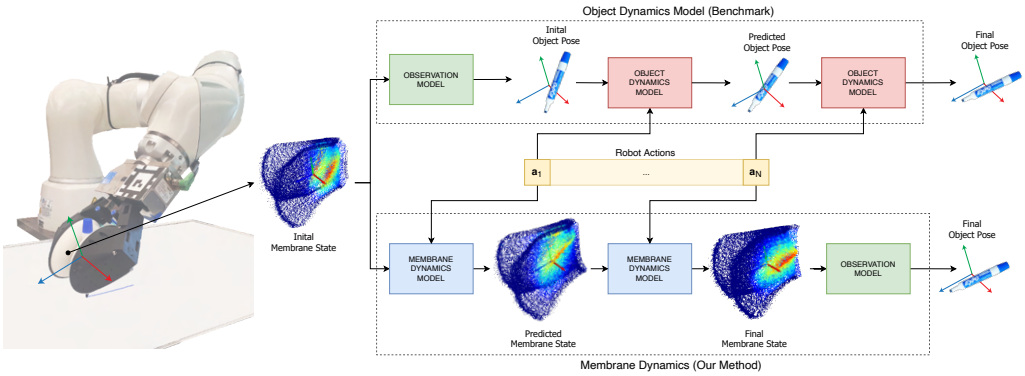

Figure A.4: **Dynamics Model Comparison** As opposed to extracting the current object pose and formulating dynamics in the object pose space (upper), our method (lower) characterizes the task dynamics as experienced by the tactile sensor by explicitly modeling the sensor's membrane deformations. As a result, our method's predicted dynamics are agnostic to the observation model.

A.7 Membrane Signature Processing

The raw tactile signature measured by the PMD PicoFlexx cameras is a (224×171) depth map (Fig. A.5 A). We crop it to remove the borders and the noisy areas, obtaining a (175×140) depth map (Fig. A.5 B). Comparing the current measured signature with a reference signature of the undeformed sensors, we can obtain the deformation map $\boldsymbol{p}_t$ (Fig. A.5 C), which expresses the relative change in camera distance for each pixel. To simplify the model and the data requirements, our model works with a downsampled depth map. Using average-pooling with factors $(7, 7)$, we reduce the deformation map size to (25×20) (Fig. A.5 D). This is the tactile signature size that the membrane dynamics model 4.2 takes as input.

$$\boldsymbol{p}_{t,\{L,R\}} = \boldsymbol{p}^{\text{meas}}_{t,\{L,R\}} - \boldsymbol{p}^{\text{ref}}_{t,\{L,R\}} \quad \text{where} \quad \boldsymbol{p}_{t,\{L,R\}}, \boldsymbol{p}^{\text{meas}}_{t,\{L,R\}}, \boldsymbol{p}^{\text{ref}}_{t,\{L,R\}} \in \mathbb{R}^{175\times140}$$

The observation model expects tactile maps in the original resolution of (175×140). Therefore, predicted tactile state in the downsampled domain $\mathbb{R}^{25\times20}$ (Fig. A.5 E) need to be converted back to the original domain $\mathbb{R}^{175\times140}$. To this end, we use bi-linear interpolation over the predicted deformation maps (Fig. A.5 F).

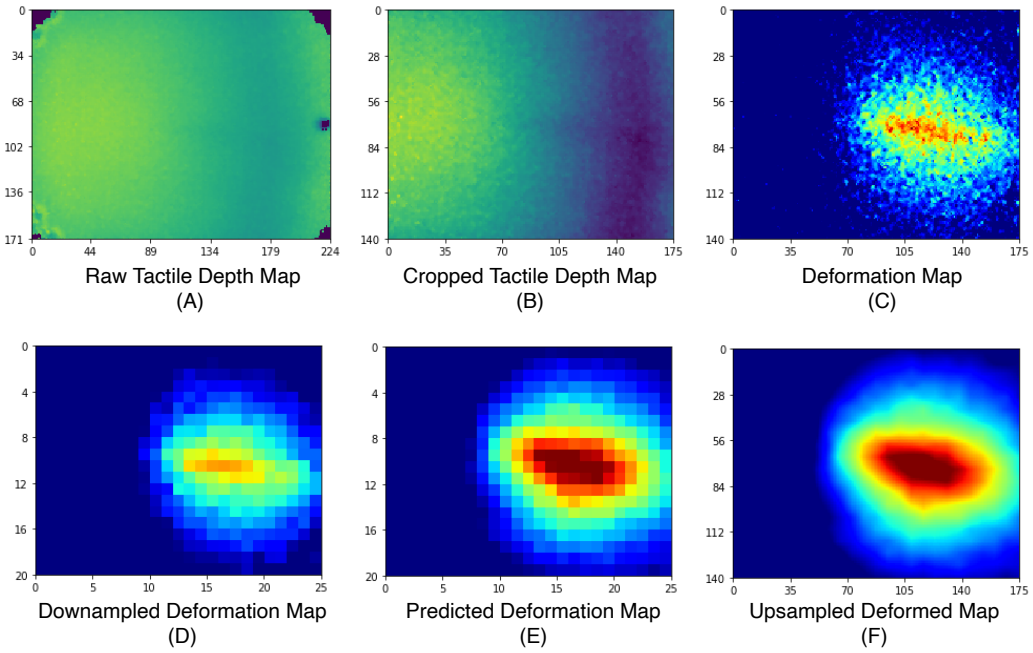

Figure A.5: **Tactile Depth Map Signatures** The PicoFlexx camera's raw measurement (A) is processed (B) and compared with a reference state to extract the depth deformations (C). Our dynamics models operate on a downsampled the depth map (D), obtained using average pooling. Before applying the observation model on model predictions (E), we upsample the deformation map using bi-linear interpolation (F).

Appendix B Experimental Setup

For pivoting, the action space is composed by:

$$\begin{aligned} \boldsymbol{a} = (\text{gw}, \Delta y, \Delta z, \Delta\phi) \in \mathcal{A}_{\text{pivoting}} \\ \text{where gw} \in [5, 40] \text{ mm} \\ \Delta y \in [-40, 40] \text{ mm} \\ \Delta z \in [-d_{\text{env}}, 20] \text{ mm} \\ \Delta\phi \in [-\frac{\pi}{6}, \frac{\pi}{6}] \text{ rad} \end{aligned}$$

For drawing, the action space is composed by:

$$\begin{aligned} \boldsymbol{a} = (\text{gw}, \Delta y, \Delta z, \Delta\phi) \in \mathcal{A}_{\text{drawing}} \\ \text{where gw} \in [10, 40] \text{ mm} \\ \Delta y \in [0, 20] \text{ mm} \\ \Delta z \in [-5, \alpha_{\text{impedance}}] \text{ mm} \\ \Delta\phi \in [-\frac{\pi}{36}, \frac{\pi}{36}] \text{ rad} \end{aligned}$$

B.1 Data Collection

For each task we collect 4000 state-action-state transitions. Each transition sample is also conditioned on the tool geometry $\boldsymbol{z}_i$ of the grasped tool during the transition. As a consequence, each data sample is a quadruplet $(\boldsymbol{s}_t, \boldsymbol{z}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1})$. The data collection process depends on the task:

- **Drawing Data Collection**: For drawing, we collect samples combining random actions with epsilon-greedy samples from a Jacobian controller. We collect 2/3 of the data drawing the evaluation line. The robot starts at a fixed location and a fixed orientation. With probability $p = 0.15$, the controller executes a random action. With probability $1-p$ the controller executes an action given by a Jacobian controller. The Jacobian controller assumes that the marker is rigidly attached to the robot. It outputs the action within the action space that will result in the object closest to the goal configuration of the marker vertically touching the board. This controller serves us as a baseline to bias the dataset towards states that are more likely to be found with the model and controller running. The remaining 1/3 of the data is collected by drawing lines purely random with different directions and starting points. We combine both data sources for a more rich dataset.
- **Pivoting Data Collection**: For pivoting, the data collection pipeline works as follows: First, the user feeds the tool to the robot and this brings it into contact with the environment with a relative orientaiton of $\pm 45^\circ$. The sign is chosen randomly. Then, for a sequence of 5 steps, the robot performs random actions within the action space and records their initial and final states. If the tool falls out of hand, or the execution fails, the data point is discarded and the process is restarted.

B.2 Drawing Evaluation Setup

To evaluate the drawing accuracy, we mount a camera on a top-down view over the drawing board. The board has 3 AprilTag fiducial marks on its corners that track its position (Fig. B.1 A). The tags allow us to unwarp the board (Fig. B.1 B). Processing the unwarpped image and adaptive thresholding we can detect the actual drawing $\mathbf{D}^{\text{meas}}$ (Fig. B.1 C) and compare it with the desired one $\mathbf{D}^{\text{goal}}$. Note that the resolution of the evaluation image is 1 pixel per millimeter. The drawing score ζ_{drawing} is defined as follows:

$$\zeta_{\text{drawing}}(\mathbf{D}^{\text{meas}}, \mathbf{D}^{\text{goal}}) = \frac{\mathbf{D}^{\text{meas}} \cap \mathbf{D}^{\text{goal}}}{\mathbf{D}^{\text{meas}}} \in [0, 1] \quad \text{where } \mathbf{D}^{\text{meas}}, \mathbf{D}^{\text{goal}} \in \mathbb{Z}_2^{565 \times 860}$$

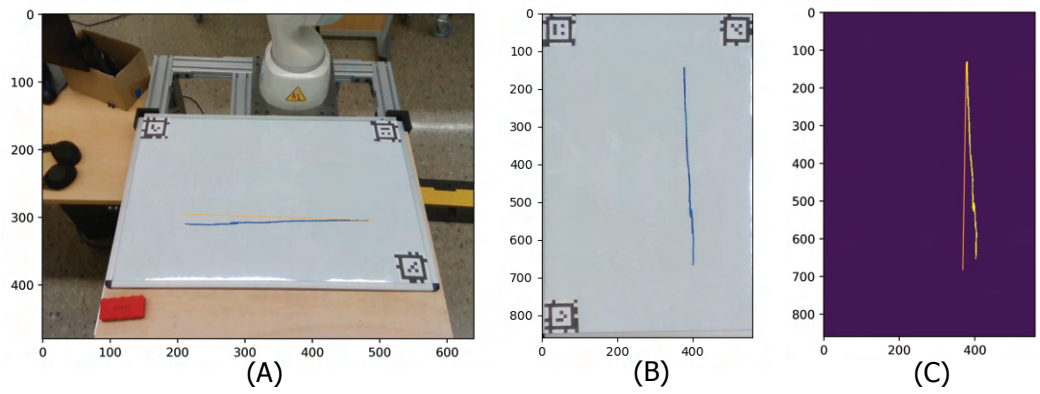

Figure B.1: **Drawing Task Evaluation** (A) Actual drawing (blue) overlayed with the desired one (orange). (B) Unwarped board. (C) Unwarped detected drawing (yellow) compared with the desired drawing (orange).

B.3 Pivoting Evaluation Setup

To asses our algorithm on the pivoting task, we evaluate the accuracy of the achieved tool pose with respect to the desired pose. For simplicity, we only compare the tool orientation defined as the angle between the tool major axis and the z axis of the grasp frame. The pivoting evaluation pipeline is initialized as follows: First, the user feeds the tool to the robot in an arbitrary angle within the range $[-45^\circ, 45^\circ]$. Second, a goal orientation is sampled from a uniform distribution in the range $[-150^\circ, 150^\circ]$. Third, the robot moves so the tool is oriented forming $\pm 45^\circ$ w.r.t. the environment surface. The sign is chosen depending on the direction of the necessary rotation towards the goal angle. L Once the tool is in contact with the environment, the algorithm can execute up to 10 actions to drive the tool to the desired orientation. If the tool has reached the orientation within $\pm 4^\circ$ of the goal orientation, we consider the execution successful. On the other side, if the tool slips out of grasp or it overshoots the goal angle, we consider a failure and the execution is stopped. We record the last final tool orientation. In case the tool has slipped out of hand, we report the last achieved stable pose. For each tool we run 10 executions. The pivoting score is computed as:

$$\zeta_{\text{pivoting}}(\theta^{\text{achieved}}, \theta^{\text{goal}}) = |\theta^{\text{goal}} - \theta^{\text{achieved}}| \in [0, 180^\circ]$$

Appendix C Supplementary Results

C.1 Pivoting Supplementary Results

We evaluate the pivoting performance based on the difference angle between the achieved tool angle and the desired tool angle. The pivoting score is therefore the difference between these two values, which ideally should be 0. We use our observation model to estimate the final reached pose.

The desired tool pose is sampled at random within the graspable range. The controller optimizes the pivoting action sequence to achieve that final pose exploiting the dynamics model.

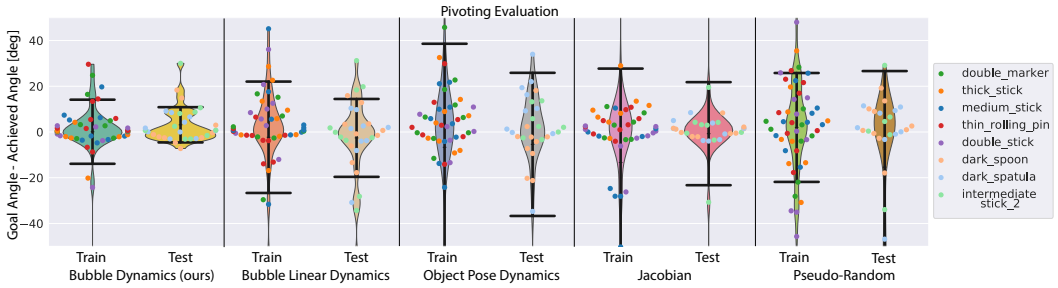

Figure C.1: **Pivoting Evaluation Results** Each column pair represents an evaluated method on both train tools and unseen tools. Colored dots indicate each tool achieved score. Black horizontal lines indicate the sample std around the mean. The pivoting score is the difference in degrees between the desired tool angle and the achieved tool angle. Therefore, the optimal pivoting score is 0.