

624 A Appendix

625 A.1 Pseudo-code for Sequential Manipulation with Relational Keypoint Constraints

Algorithm 1 Relational Keypoint Constraints for Sequential Manipulation

```

1: Initialize current stage  $i \leftarrow 1$ , and current time  $t \leftarrow 1$ 
2: while  $i \leq N$  do
3:   if  $\exists f \in \mathcal{C}_{\text{path}}^{(i)}$  s.t.  $f(k_t) > 0$  then
4:      $i \leftarrow i - 1$ 
5:     continue
6:   end if
7:   if  $\text{distance}(\mathbf{e}_t, \mathbf{e}_{g_i}) < \epsilon$  then
8:      $i \leftarrow i + 1$ 
9:     continue
10:  end if
11:  Solve sub-goal problem for stage  $i$  to obtain  $\mathbf{e}_{g_i}$  (Eq. 2)
12:  Solve path problem for stage  $i$  to obtain  $\mathbf{e}_{t:g_i}$  (Eq. 3)
13:  Execute the next  $m$  actions  $\mathbf{e}_{t+1:t+m}$ 
14:   $t \leftarrow t + m + 1$ 
15: end while

```

626 A.2 Mobile Single-Arm Platform

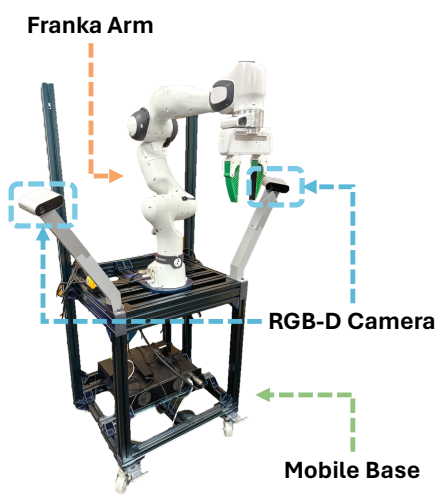


Figure 6: Mobile Single-Arm Platform.

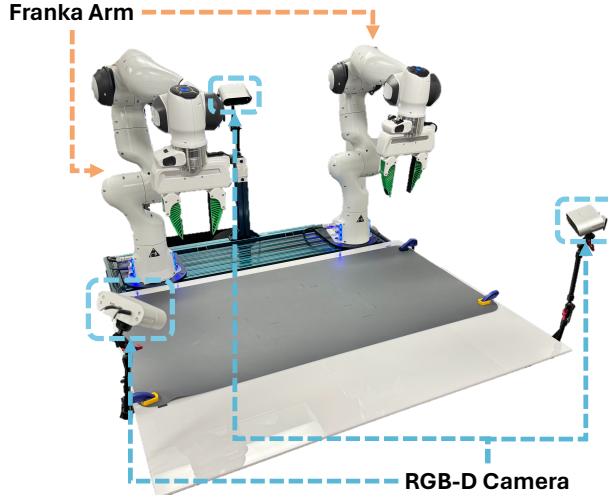


Figure 7: Stationary Dual-Arm Platform.

627 We use a Franka arm mounted on a mobile base built with Vention frames (shown in Figure 6). Note
628 that the base does not have motors and thus cannot move autonomously, but its mobility nevertheless
629 allows us to investigate the proposed method outside of lab environments.

630 Since our pipeline produces a sequence of 6-DoF end-effector poses, we use position control in all
631 experiments, which is running at a fixed frequency of 20 Hz. Specifically, once the robot is given a
632 target end-effector pose in the world frame, we first clip the pose to the pre-defined workspace. Then
633 we linearly interpolate from the current pose to the target pose with a step size of 5mm for position
634 and 1 degree for rotation. To move to each interpolated pose, we first calculate inverse kinematics
635 to obtain the target joint positions based on current joint positions (IK solver from PyBullet [109]).
636 Then we use a joint impedance controller from Deoxys [110] to reach to the target joint positions.

Two RGB-D cameras, Orbbec Femto Bolt, are mounted on each side of the robot facing the workspace center. The cameras capture RGB images and point clouds at a fixed frequency of 20 Hz.

A.3 Stationary Dual-Arm Platform

The stationary dual-arm platform consists of two Franka arms mounted in front of a tabletop workspace (shown in Figure 7). We share the same controller as the mobile single-arm platform with the exception that the two arms are controlled simultaneously at 20 Hz. Specifically, our pipeline produces two 6-DoF end-effector poses at a time, which are sent to the controller together. The controller subsequently calculates IK for both arms and moves the arms using joint impedance control.

Three RGB-D cameras, Orbbec Femto Bolt, are mounted on this platform. Two cameras are mounted on the left and right sides and one camera is mounted in the back. The cameras similarly capture RGB images and point clouds at a fixed frequency of 20 Hz.

A.4 Evaluation Details

Below we discuss the evaluation details for the experiments reported in Section 4.1 and Section 4.2.

A.4.1 Details for In-the-Wild and Bimanual Manipulation (Section 4.1)

For each task, 10 initial different configurations of objects are selected, which cover the full workspace but are manually verified to ensure they are kinematically feasible for the robot. For each trial, a human operator restores the scene to the corresponding configuration and initiates the system. Due to the challenge of developing automatic success criteria for the diverse set of objects and environments investigated in this work, success rates are measured by the operator with the criterion reported under each task description below. For experiments involving external disturbances, the set of disturbances for all trials is pre-selected, and one disturbance is applied to each trial. Specifically, the disturbance is introduced by a human operator using hands to change the object’s pose. Collision checking is disabled for all tasks involving deformable objects.

Pour Tea: The environment consists of a teapot and a cup placed on a counter table in a kitchen setting. The task involves three stages: grasping the handle, aligning the teapot to the top of the cup, and pouring the tea into the cup. The success criterion requires that the teapot remains upright until the pouring stage, and at the end, the spout must be aligned and tilted on top of the cup opening.

Recycle Can: The environment includes one of three types of cans (Coke, Zero Coke, Zero Sprite), a recycle bin with a narrow opening (such that the cans may only go in when they are upright), a landfill bin, and a compost bin, all situated inside an office building. The task involves two stages: grasping the can and reorienting it on top of the recycle bin before dropping it. The success criterion is that the can is successfully thrown into the bin.

Stow Book: The environment consists of a target book placed on a side table and a real-size bookshelf with a 15cm opening among the placed books, all inside an office environment. The task involves two stages: grasping the target book on the side and stowing it inside the opening in the shelf. The success criterion is that the target book is placed steadily after the robot releases the gripper, and the robot must not bump into the shelf or other placed books.

Tape Box: The environment includes a cardboard box, a packaging tape with a dispenser sitting on top of the box that already has one side taped, and a human user collaborating with the robot. The tape has already been unrolled to be enough for taping because unrolling typically requires a large force that exceeds the limit of the robot arm. The task involves two stages: while a human operator is squeezing the box, the robot needs to grasp the tape and align it to the correct side to complete the taping. The success criterion is that the tape must end up in the correct position such that it is aligned with the seam.

682 **(Bimanual) Fold Garment:** The environment consists of a sweater placed flat close to the
683 workspace center, with small deformations on the sleeves, neck, and bottom. The task typically
684 requires four stages: grasping both sleeves, folding them to the middle, grasping the neck, and fold-
685 ing it to the bottom. The success criterion does not enforce consistent stages; as long as the sweater
686 is folded such that it occupies at most half of the original surface size, it is regarded as a success.

687 **(Bimanual) Pack Shoes:** The environment includes an empty shoe box placed close to the
688 workspace center, with two shoes placed on opposite sides of the box in random poses. The task
689 involves two stages: grasping the shoes simultaneously and placing them in the shoe box. The suc-
690 cess criterion does not enforce consistent stages; as long as the shoes are placed into the box without
691 being stacked together or causing bimanual self-collision, it is considered successful.

692 **(Bimanual) Collaborative Folding:** The environment consists of a large blanket (pre-folded to an
693 appropriate size that occupies about 70% of the workspace due to its size exceeding the workspace
694 limit) and a human user collaborating with the robot. The task involves two stages: the robot must
695 grasp the two corners of the blanket opposite to the human user, and the second stage is aligning
696 the two corners with the two corners that the human has grasped. The success criterion is that the
697 robot has grasped the correct corners and can align them with the correct human arms (left-left,
698 right-right).

699 A.4.2 Details for Generalization in Manipulation Strategies (Section 4.2)

700 The dual-arm robot is tasked with folding eight different categories of clothing. We use two metrics
701 for evaluation: "Strategy Success" and "Execution Success," where the former evaluates whether
702 keypoints are proposed and constraints are written appropriately, and the latter evaluates the robotic
703 system's execution given successful strategies.

704 To evaluate "Strategy Success," the garment is initialized close to the center of the workspace. A
705 back-mounted RGB-D camera captures the RGB image. Then, the keypoint proposal module gen-
706 erates keypoint candidates using the captured image, which are then overlaid on top of the original
707 image with numerical marks $\{1, \dots, K\}$. The overlaid image, along with the same generic prompt,
708 is fed into GPT-4 [6] to generate the ReKep constraints. Since folding garments is itself an open-
709 ended problem without ground-truth strategies, we manually judge if the proposed keypoints and
710 the generated constraints are correct. Note that since the constraints are to be executed by a biman-
711 ual robot, and the constraints are almost always connecting (folding) two keypoints such that they
712 are aligned, correctness is measured by whether it is (potentially) executable by the robot without
713 causing self-collision (arms crossing over to opposite sides) and whether the folding strategy can
714 fold the garment to at most half of its original surface area.

715 To evaluate "Execution Success," we take the generated strategies in the previous section that are
716 marked as successful for each garment and execute the sequence on the dual-arm platform, with a
717 total of 10 trials for each garment. Point tracking is disabled as we observe that our point tracker
718 predicts unstable tracks when the garment is potentially folded many times. Success is measured by
719 whether the garment is folded such that its surface area is at most half of its original surface area.

720 A.5 Implementation Details of Keypoint Proposal

721 Herein we describe how keypoint candidates in a scene are generated. For each platform, we use
722 one of the mounted RGB-D cameras to capture an image of size $h \times w \times 3$, depending on which
723 camera has the best holistic view of the environment, as all the keypoints need to be present in the
724 first frame for the proposed method. Given the captured image, we first use DINOv2 with registers
725 (ViT-S14) [5, 111] to extract the patch-wise features $\mathbf{F}_{\text{patch}} \in \mathbb{R}^{h' \times w' \times d}$. Then we perform bilinear
726 interpolation to upsample the features to the original image size, $\mathbf{F}_{\text{interp}} \in \mathbb{R}^{h \times w \times d}$. To ensure the
727 proposal covers all relevant objects in the scene, we extract all masks $\mathbf{M} = \{\mathbf{m}_1, \mathbf{m}_2, \dots, \mathbf{m}_n\}$
728 in the scene using Segment Anything (SAM) [108]. The masks are filtered such that their center
729 3D coordinate (projected with calibrated RGB-D camera) lies within the pre-defined workspace
730 bounds. Within each mask \mathbf{m}_i , we apply PCA to project the features to three dimensions, $\mathbf{F}_{\text{PCA}} =$

PCA($\mathbf{F}_{\text{resized}}[\mathbf{m}_i], 3$). We find that applying PCA improves the clustering as it often removes details and artifacts related to texture that are not useful for our tasks. For each mask j , we cluster the masked features $\mathbf{F}_{\text{interp}}[\mathbf{m}_j]$ using k -means with $k = 5$ with a cosine-similarity metric. The median centroids of the clusters are used as keypoint candidates, which are projected to a world coordinate \mathbb{R}^3 using a calibrated RGB-D camera. Note that we also store which keypoint candidates originate from the same mask, which is later used as part of the rigidity assumption in the optimization loops described in Sec. 3.3. Candidates outside of the workspace bounds are filtered out. To avoid many points cluttered in a small region, we additionally use Mean Shift [112, 113] (with a bandwidth 8cm) to filter out points that are close to each other. Finally, the centroids are taken as final candidates. Alternatively, one may develop a pipeline using only segmentation models [108, 114], but we leave comparisons to future work.

A.6 Querying Vision-Language Model

After we obtain the keypoint candidates, they are overlaid on the captured RGB image with numerical marks $\{1, \dots, K\}$. Then the image and the task instruction are fed into a vision-language model with the prompt described below. The prompt contains only generic instructions with no image-text in-context examples, although a few text-based examples are given to concretely explain the proposed method and the expected output from the model.

For the experiments conducted in this work, we use the latest (likely most capable) GPT-4o [6] at the time of writing. However, due to rapid advancement in this field, the pipeline can directly benefit from newer models that have better vision-language reasoning. Correspondingly, we observe different models exhibit different behaviors when given the same prompt (with the observation that newer models typically require less fine-grained instructions). As a result, instead of developing the best prompt for the suite of tasks in this work, we focus on demonstrating a full-stack pipeline consisting a key component that can be automated and continuously improved by rapid future development.

```

755 ## Instructions
756 Suppose you are controlling a robot to perform manipulation tasks by writing constraint functions in Python.
757 The manipulation task is given as an image of the environment, overlaid with keypoints marked with
758 their indices, along with a text instruction. The instruction starts with a parenthesis indicating
759 whether the robot has a single arm or is bimanual. For each given task, please perform the following
760 steps:
761 - Determine how many stages are involved in the task. Grasping must be an independent stage. Some examples:
762   - "(single-arm) pouring tea from teapot":
763     - 3 stages: "grasp teapot", "align teapot with cup opening", and "pour liquid"
764   - "(single-arm) put red block on top of blue block":
765     - 3 stages: "grasp red block", "align red block on top of blue block", and "release red block"
766   - "(bimanual) fold sleeves to the center":
767     - 2 stages: "left arm grasps left sleeve and right arm grasps right sleeve" and "both arms fold sleeves to
768       the center"
769   - "(bimanual) fold a jacket":
770     - 3 stages: "left arm grasps left sleeve and right arm grasps right sleeve", "both arms fold sleeves to
771       the center", and "grasp the neck with one arm (the other arm stays in place)", and "align the neck
772       with the bottom"
773 - For each stage, write two kinds of constraints, "sub-goal constraints" and "path constraints". The "sub-goal
774   constraints" are constraints that must be satisfied **at the end of the stage**, while the "path
775   constraints" are constraints that must be satisfied **within the stage**. Some examples:
776   - "(single-arm) pouring liquid from teapot":
777     - "grasp teapot" stage:
778       - sub-goal constraints: "align the end-effector with the teapot handle"
779       - path constraints: None
780     - "align teapot with cup opening" stage:
781       - sub-goal constraints: "the teapot spout needs to be 10cm above the cup opening"
782       - path constraints: "robot is grasping the teapot", and "the teapot must stay upright to avoid spilling"
783     - "pour liquid" stage:
784       - sub-goal constraints: "the teapot spout needs to be 5cm above the cup opening", "the teapot spout must
785         be tilted to pour liquid"
786       - path constraints: "the teapot spout is directly above the cup opening"
787   - "(bimanual) fold sleeves to the center":
788     - "left arm grasps left sleeve and right arm grasps right sleeve" stage:
789       - sub-goal constraints: "left arm grasps left sleeve", "right arm grasps right sleeve"
790       - path constraints: None
791     - "both arms fold sleeves to the center" stage:
792       - sub-goal constraints: "left sleeve aligns with the center", "right sleeve aligns with the center"
793       - path constraints: None
794
795 Note:
796 - Each constraint takes a dummy end-effector point and a set of keypoints as input and returns a numerical
797   cost, where the constraint is satisfied if the cost is smaller than or equal to zero.
798 - For each stage, you may write 0 or more sub-goal constraints and 0 or more path constraints.
799 - Avoid using "if" statements in your constraints.
800 - Avoid using path constraints when manipulating deformable objects (e.g., clothing, towels).
801 - You do not need to consider collision avoidance. Focus on what is necessary to complete the task.

```

```

803 - Inputs to the constraints are as follows:
804   - `end_effector`: np.array of shape `(3,)` representing the end-effector position.
805   - `keypoints`: np.array of shape `(K, 3)` representing the keypoint positions.
806 - Inside of each function, you may use native Python functions and NumPy functions.
807 - For grasping stage, you should only write one sub-goal constraint that associates the end-effector with a
808   keypoint. No path constraints are needed.
809 - For non-grasping stage, you should not refer to the end-effector position.
810 - In order to move a keypoint, its associated object must be grasped in one of the previous stages.
811 - The robot can only grasp one object at a time.
812 - Grasping must be an independent stage from other stages.
813 - You may use two keypoints to form a vector, which can be used to specify a rotation (by specifying the angle
814   between the vector and a fixed axis).
815 - You may use multiple keypoints to specify a surface or volume.
816 - You may also use the center of multiple keypoints to specify a position.
817 - A single folding action should consist of two stages: one grasp and one place.
818
819 Structure your output in a single python code block as follows for single-arm robot:
820 ```python
821
822 # Your explanation of how many stages are involved in the task and what each stage is about.
823 # ...
824
825 num_stages = ?
826
827 ### stage 1 sub-goal constraints (if any)
828 def stagel_subgoal_constraint1(end_effector, keypoints):
829     """Put your explanation here."""
830     ...
831     return cost
832 # Add more sub-goal constraints if needed
833
834 ### stage 1 path constraints (if any)
835 def stagel_path_constraint1(end_effector, keypoints):
836     """Put your explanation here."""
837     ...
838     return cost
839 # Add more path constraints if needed
840
841 # repeat for more stages
842 ...
843 ```
844
845 Structure your output in a single python code block as follows for bimanual robot:
846 ```python
847
848 # Your explanation of how many stages are involved in the task and what each stage is about.
849 # ...
850
851 num_stages = ?
852
853 ### left-arm stage 1 sub-goal constraints (if any)
854 def left_stagel_subgoal_constraint1(end_effector, keypoints):
855     """Put your explanation here."""
856     ...
857     return cost
858
859 ### right-arm stage 1 sub-goal constraints (if any)
860 def right_stagel_subgoal_constraint1(end_effector, keypoints):
861     """Put your explanation here."""
862     ...
863     return cost
864 # Add more sub-goal constraints if needed
865
866 ### left stage 1 path constraints (if any)
867 def left_stagel_path_constraint1(end_effector, keypoints):
868     """Put your explanation here."""
869     ...
870     return cost
871 ### right stage 1 path constraints (if any)
872 def right_stagel_path_constraint1(end_effector, keypoints):
873     """Put your explanation here."""
874     ...
875     return cost
876 # Add more path constraints if needed
877
878 # repeat for more stages
879 ...
880 ```
881
882 ## Query
883 Query Task: "[INSTRUCTION]"
884 Query Image: [IMAGE WITH KEYPOINTS]

```

A.7 Implementation Details of Point Tracker

We implement a simple point tracker following [95] based on DINOv2 (ViT-S14) [5] that leverages the fact that multiple RGB-D cameras are present and DINOv2 is efficient to run at a real-time frequency.

At initialization, an array of 3D keypoint positions $\mathbf{k} \in \mathbb{R}^3$ are given. We first take the RGB-D captures from each present camera. For each RGB image, we obtain the pixel-wise DINOv2 features following the same procedure in Section A.5 and record their associated 3D world coordinates using calibrated cameras. For each 3D keypoint positions, we aggregate all the features from points that are within 2cm from all the cameras. The mean of the aggregated features is recorded as the reference feature for each keypoint, which is kept fixed throughout the task.

After initialization, at each time step, we similarly obtain the pixel-wise features from DINOv2 from all cameras with their 3D world coordinates. To track the keypoints, we calculate cosine similarity between features across all pixels and the reference features. The top 100 matches are selected for each keypoint with a cutoff similarity of 0.6. We then reject outliers for the selected matches by calculating median deviation ($m = 2$). Additionally, as the tracked keypoints may oscillate in a small region, we apply a uniform filter with a window size of 10 in the end. The entire procedure runs at a fixed frequency of 30 Hz.

Note that the implemented point tracker is a simplification from [95] for real-time tracking. We refer readers to [95] for more comprehensive discussion on using self-supervised vision models, such as DINOv2, for point tracking. Alternatively, more specialized point trackers can be used [115–119]. We find that our implementation is advantageous in scenarios that involve long-term occlusions as the reference features are kept fixed, but we expect more development from the specialized point trackers that likely would yield better performance in the future.

A.8 Implementation Details of Sub-Goal Solver

The sub-goal problems are implemented and solved using SciPy [101]. The decision variable is a single end-effector pose (position and Euler angles) in \mathbb{R}^6 for single-arm robots and two end-effector poses in \mathbb{R}^{12} for bimanual robot. The bounds for the position terms are the pre-defined workspace bounds, and the bounds for the rotation terms are that the half hemisphere where the end-effector faces down (due to the joint limits of the Franka arm, it is often likely to reach joint limit when an end-effector pose faces up). The decision variables are normalized to $[0, 1]$ based on the bounds. For the first solving iteration, the initial guess is chosen to be the center of the bounds. We use sampling-based global optimization Dual Annealing [102] in the first iteration to quickly search the full space, which is followed by a gradient-based local optimizer SLSQP [103] that refines the solution. The full procedure takes around 1 second for this iteration. In subsequent iterations, we use the solution from previous stage and only use local optimizer as it can quickly adjust to small changes. The optimization is cut off with a fixed time budget represented as number of objective function calls to keep the system running at a high frequency.

We discuss the cost terms in the objective function below.

Constraint Violation: We implement constraints as cost terms in the optimization problem, where the returned costs by the ReKep functions are multiplied with large weights.

Scene Collision Avoidance: We use nvblox [120] with the pytorch wrapper [51] to compute the ESDF of the scene in a separate node that runs at 20 Hz. The ESDF calculation aggregates the depth maps from all available cameras and excludes robot arms using cuRobo and any grasped rigid objects (tracked via a masked tracker model Cutie [121]). A collision voxel grid is then calculated using the ESDF and used by other modules in the system. In the sub-goal solver module, we first downsample the gripper points and the grasped object points to have a maximum of 30 points using farthest point sampling. Then we calculate the collision cost using the ESDF voxel grid with linear interpolation with a threshold of 15cm.

Reachability: Since our decision variables are end-effector poses, which may not be always reachable by the robot arms, especially in confined spaces, we need to add a cost term that encourages finding solutions with valid joint configurations. Therefore, we solve an IK problem in each iteration of the sub-goal solver using PyBullet [109] and use its residual as a proxy for reachability. We find that this takes around 40% of the time of the full objective function. Future works can consider solving the problem in joint space, which would guarantee the solution is within the joint limits by enforcing the bounds. We find that this would be inefficient in our Python-based implementation as we need to calculate forward kinematics for a magnitude of more times in the path solver, because the constraints are calculated in the task space. However, future works can consider using more efficient implementations and solve the problems in joint space [51].

Pose Regularization: We also add a small cost that encourages the sub-goal to be close to the current end-effector pose.

Consistency: Since the solver iteratively solves the problem at a high frequency, we find it useful to include a consistency cost that encourages the solution to be close to the previous solution.

(Dual-Arm only) Self-Collision Avoidance: To avoid two arms collide with each other, we compute the pairwise distance between the two point sets, each including the gripper points and grasped object points.

A.9 Implementation Details of Path Solver

The path problems are implemented and solved using SciPy [101]. The number of decision variables is calculated based on the distance between the current end-effector pose and the target end-effector pose. Specifically, we define a fixed step size (20cm and 45 degree) and linearly approximate the desired number of “intermediate poses”, which are used as decision variables. As in the sub-goal problem, they are similarly represented using position and Euler angles with the same bounds. For the first solving iteration, the initial guess is chosen to be linear interpolation between the start and the target. We similarly use sampling-based global optimization followed by a gradient-based local optimizer in the first iteration and only use local optimizer in subsequent iterations. After we obtain the solution, represented as a number of intermediate poses, we fit a spline using the current pose, the intermediate poses, and the target pose, which are then densely sampled to be executed by the robot.

In the objective function, we first unnormalize the decision variables and use piecewise linear interpolation to obtain dense samples of the path. A spline interpolation would be aligned with how we postprocess the solution, but we find linear interpolation to be computationally more efficient. Below we discuss the individual cost terms in the objective function.

Constraint Violation: Similar to that in the sub-goal problem, we check violation of the ReKep constraints for each dense sample along the path and penalize with large weights.

Scene Collision Avoidance: The calculation is similar to the sub-goal problem, except that it is calculated for each dense sample. We ignore the collision calculation with a 5cm radius near the start and the target poses, as this tends to stabilize the solution when solved at a high frequency due to various real-world noises. We additionally add a table clearance cost that penalizes the path from penetrating the table (or the bottom of the workspace for the mobile single-arm robot).

Path Length: We approximate the path length using the dense samples. Shorter paths are encouraged.

Reachability: We similarly solve an IK problem inside the objective function. See the sub-goal solver section for more details.

Consistency: As in the sub-goal problem, we similarly encourage the solution to be close to the previous one. Specifically, we store the dense samples from the previous iteration. To calculate the far the two dense sample sequences are from each other, we use the pairwise distance between the

981 two sequences (treated as two sets) as an efficient proxy. Alternatively, Hausdorff distance can be
982 used.

983 **(Dual-Arm only) Self-Collision Avoidance:** We similarly compute self-collision avoidance for the
984 dual-arm platform as in the sub-goal problem. We also use pairwise distance between the two
985 sequences to efficiently calculate this cost.