Appendix: Enhancing Dexterity in Robotic Manipulation via Hierarchical Contact Exploration

APPENDIX I
SETTING UP NEW SCENARIOS

In this section, we provide an overview of what are required when setting up new scenarios. Please check our code and Appendix II for the actual implementation.

A. Applicability

This framework can be considered for the tasks of manipulating a single rigid body object in a rigid environment. Environment components must be fixed and not movable. It can also be used when there is no environment component (in-hand manipulation). We need known models of the object, the environment, and the robot.

The robot used to manipulate the object needs to have known collision models, and forward and inverse kinematics. The only parts that can be used to manipulate the object are the defined "fingertips" on the robot.

B. Setup a new robot/hand

Setting up a new robot is the most complicated part. Specifically for implementation in our C++ code, a new class need to be written to inherit a pre-defined abstract class ROBOTTEMPLATE. The user need to fill some specific pure virtual functions that covers the following aspects.

1) Contact force models for fingertips: We use the point contact model for kinematics. However, as the force model for point contact might be too limited, we allow the use of other contact force models. The contact force models that currently exist in our implementation includes:

- Point contact
- Patch contact: we first approximate the fingertips using spheres centered at the point contact locations. The radius of the spheres should approximate the radius of the contact patch for each fingertip. We approximate the patch contact using three point contacts at vertices of an equilateral triangle that is perpendicular to the contact normal and on the sphere.
- Line contact: we approximate the line contact model by two point contacts on twp ends of the line segment.

2) Forward and inverse kinematics for fingertips: The users need to provide the forward and inverse kinematics for the fingertips.

Given the FK and IK model, we precompute the workspace for each fingertip. For general robot hands, we first sample joint angles to get the fingertip points in the workspace through forward kinematics, and then compute the convex hulls. While hands might differ, we estimate this process takes about seconds (with C++ implementation).

3) Robot collision model: The users need to provide the collision model of the robot or the fingertips. If it is unlikely for the robot links to collide with the object or the environment, it is be okay to only provide the collision shape for the fingertips, which will make the computation much faster. Otherwise, the user could simply provide a robot URDF model.

4) Contact relocation planner (optional): A contact relocation planner is required for checking whether a collision-free path exists for a finger to relocate to another contact location.

5) Contact sampling on the object surface (optional): It is best that each fingertip are relatively independent on the kinematic side. If not, our random sampling of robot contacts on the object surface might have a very high rejection rate ($> 90\%$). In this case, we need the user to provide a method for the specific robot in order to more efficiently sample robot contacts on the object surface.

6) Trajectory optimizer (optional): For the robots that are under-actuated (like wheeled robots), the users need to provide Level 3 a trajectory optimizer that finds feasible object states, robot states, and robot controls given Level 2 outputs as warm-start trajectories.

C. Setup a new task type

After setting up the new robot, we need to enable the robot to do a certain type of tasks. Two major things to consider are task mechanics and task parameters for planning.

1) Task Mechanics: Task mechanics include the specific requirements and dynamical model required by the task. Do we have to fully exploit the dynamic property of the system? If yes, we need to have a good trajectory optimization algorithm for the manipulation system in Level 3 to ensure the solutions are feasible. If the task dynamics do not involve the integration of velocity and the robot is fully actuated, it is not necessary to provide a trajectory optimization method in Level 3. In these cases, the user only needs to write a function `task-dynamics(object pose, object velocity, contact info, ...)` that solves a one-step optimization problem. Examples include quasi-static, quasi-dynamic, closure methods, planar pushing, etc.

2) Design choices: A new task type requires several design choices to be made and some search parameters to be tuned. Once the choices are made, changing environments and objects in the same task type should not require more tuning. According to our experience, making designs and tuning parameters are relatively low-effort. We have found

that the planner is not sensitive to specific numerical values for parameters.

The design choices include task features, action probability design for Level 1 and 2, reward design, and value estimation design for Level 1 and 2.

Task features are used in the action probability and reward. Basic features are path length in MCTS, number of robot contact relocations, and object travel distance. Task-dependent features like grasp measures or environment contact changes can be added to encourage specific behaviors like better grasps and less environment contact switches. For generality, it is important to normalize the features by ensuring similar values for desired behaviors across different environments and objects.

There are three action probability functions we need to define: (1) select a contact mode in Level 1, (2) select a child (configuration node) for a mode node in Level 1, and (3) select (time to relocate, contacts to relocate to) in Level 2. For (1), we often encourage the use of the same contact mode as the previous one. For (2), we currently mostly use a uniform distribution. For (3), we would like to encourage relocating when the contacts are not feasible the next timestep. However, the definition of these probabilities is entirely up to the user.

To design the reward function, we use a simple approach that requires no tuning. Given some feature values as data points, we first manually label their reward values between 0 to 1 through human intuition. Next, we fit a logistic function to these data points as the reward function.

Manually value estimation is very flexible. The value estimation in our method is often used to encourage the search to visit a node that has been visited but has not found any positive reward. For example, on the way to the goal pose, if an object pose is reachable through a sequence of contacts (check by Level 2), we can assign 0.1 as its value estimation. Our design principle is to give a small number to any node that is more likely to find a solution than others.

3) Parameters: The search parameters include MCTS exploration rates η_1, η_2 in Level 1 and 2. Adaptive parameter for value estimation λ. In all of our experiments, we let $\eta_1 = 0.1, \eta_2 = 0.1$. We let $\lambda = 0$ if a positive reward has not been found and otherwise $\lambda = 1$. While tuning the parameters may slightly improve the performance for specific tasks, we suspect that most of the time this is not a must. However, λ might need some tuning if one day we have better value estimations, like using learned functions.

D. Setup a new environment

If using our preset robots and tasks, the users can easily setup new environments and objects in one file called *setup.yaml*.

When setting up a model for a new environment, it is usually adequate to use primitive geometries such as cuboids, cylinders, and spheres in the simulation environment. The users need to specify the shape parameters and the locations of the primitive shapes.

E. Setup a new object

For a new object, the users need to provide the object mesh or specify the primitive shape. Surface points will be automatically uniformly sampled on the mesh. Each point (p,n) is represented by its location ($p \in R^3$) and its contact normal ($n \in S^3$) in the object frame. It is usually sufficient to sample about 100 points. The computation is usually in milliseconds.

The user also need to provide the object mass, object inertia, and friction coefficients for robot-object and environment-object contacts.

For each new object and environment, the RRT parameters might need some changes, including the range of object positions, goal biased sampling probability, unit extend length, and the weight for rotation for the distance calculation. The RRT parameters does not require careful tuning, as long as they roughly reflect the task requirement. For example, the weight for rotation is good to be set to 1 if the object bounding box range is between 0.1 - 10 and rotation and translation are roughly of equal importance. If the object orientation is not important at all, the weight is good to be set to 0.01 to 0.1. The unit extend length should be larger if the object start and goal are very far from each other, otherwise the planner will be slow. And it should be smaller if the user expect many different maneuvers required for the task.

Extra note: to avoid numerical issues, we usually scale the whole system such that the average length of the object bounding box is in the range of 1 - 10.

Appendix II
Experiment Details

This section includes the details of the experiments in this paper. The first two are pure planning experiments. The latter two are robot experiments.

A. Manipulation with Environment Interactions

1) Robot model: We consider the robots as free-flying balls, meaning that we do not check for kinematic feasibility but do check for collision of the balls and the environment. For the contact force model, we use the patch contact model, described in Appendix I-B.

2) Task mechanics: We use quasi-static or quasi-dynamic models. For each timestep, we solve a convex programming problem to find if there exists a solution for contact force λ_c to satisfy the force conditions. The problem is formulated as follows:

$$\begin{aligned} &\min_{\lambda} \|\epsilon \lambda^T \lambda\| \\ \text{s.t.}\quad &\text{quasistatic or quasidynamic condition} \end{aligned} \tag{1}$$

where $\epsilon \lambda^T \lambda$ is a regularization term on the contact forces.

The quasi-static condition requires the object to be under static force balance for a selected contact mode

$$\left[G_1 h_1, G_2 h_2, \ldots\right] \cdot \left[\lambda_1, \lambda_2, \ldots\right]^T + F_{\text{external}} = 0 \tag{2}$$

where $\left[\lambda_1, \lambda_2, \ldots\right]^T$ are the magnitudes of forces along active contact force directions $\left[h_1, h_2, \ldots\right]^T$ determined by contact modes. $\left[G_1, G_2, \ldots\right]^T$ are the contact grasp maps. F_{external} includes other forces on the object, such as gravity and other applied forces.

Quasidynamic assumption relaxes the requirement for objects to be in force balance, allowing short periods of dynamic motions. We assume accelerations do not integrate into significant velocities. In numerical integration, the object velocity from the previous timestep is 0. The equations of motions become:

$$M_o \dot{v^o} = \left[G_1 h_1, G_2 h_2, \ldots\right] \cdot \left[\lambda_1, \lambda_2, \ldots\right]^T + F_{\text{external}} \quad (3)$$

In discrete time, the object acceleration $\dot{v^o}$ can be written as $\frac{v^o}{h}$, where h is the step size. The object velocity v^o is computed by solving the constrained velocity from the current pose to the goal pose under a contact mode.

3) Feasibility Checks:

- Task mechanics check: is passed if there exist a solution for Equation 1.
- Finger relocation check: during relocation, the non-relocating robot contacts and environment contacts must also satisfy the task mechanics, assuming the object has zero velocity.
- Collision check: the spheres must not collide with the environment.

4) Features: We manually designed the features, as shown in Table I.

Feature	Description
Path size	node depth in the Level 1 tree
Object travel distance ratio	$\frac{\text{total travel distance}}{dist(x_{\text{start}}, x_{\text{goal}})}$
Robot contact change ratio	$\frac{\text{number of finger contact changes}}{\text{number of fingers}}$
Number of environment contact changes	-
Grasp centroid distance	$dist(c_{contact}, c_{geo})$

TABLE I: Features for Manipulation with Environment Interactions. $c_{contact}$: the centroid of all contact points; c_{geo}: the geometric center of the object.

5) Action Probability: In Level 1, in choosing the next contact mode, we design the action probability to prioritize choosing the contact mode the same as before:

$$p\big(s1 = (x, mode), a\big) = \begin{cases} 0.5 & \text{if} \quad a = \text{previous mode} \\ \frac{0.5}{\#\text{modes}-1} & \text{else} \end{cases} \quad (4)$$

In Level 1, in choosing the next configuration, we let $p\big(s1 = (x, config), a\big)$ be a uniform distribution for all the children and *explore-new*

In Level 2, in choosing a timestep to relocate and the contact points to relocate, the action probability is calculated using a weight function $w(s2, a)$ designed for each action in $\mathcal{A}_{\text{sp}}(s2)$:

$$p(s2, a) = \frac{w(s2, a)}{\sum_{a' \in \mathcal{A}_{\text{sp}}(s2)} w(s2, a')} \quad (5)$$

The manually designed weight function $w(s2, a)$ prefers to let the previous robot contacts stay as long as possible:

$$w(s2, a) = \begin{cases} 0.5 + \frac{0.5}{t_{\max} - t_c + 1} & \text{if} \quad t_c = t_{\max} \\ \frac{0.5}{t_{\max} - t_c + 1} & \text{else} \end{cases} \quad (6)$$

6) Reward Design: We use all the features in Table I and follow the logistic function fitting procedure as described in Appendix I-C.2.

7) Value Estimation: We only use value estimation for Level 1 nodes. Each node has $v_{est} = 0.1$ if any subsequent Level 2 search is able to proceed past that node. For all Level 2 nodes, the value estimation is simply zero.

8) Search Parameters: In both Level 1 and Level 2, we let the exploration rate $\eta_1, \eta_2 = 0.1$. Since we only have value estimation for Level 1, there is only one adaptive parameter λ for Level 1 only. When no reward > 0 has been found, $\lambda = 0$. After any positive reward is observed, $\lambda = 1$.

B. In-hand Manipulation

1) Robot model: The setup is the same as Appendix II-A.1. The only difference is that we now have a workspace limit for each finger.

2) Task mechanics: We use quasi-static models (as described in Appendix II-A.2) or force closure [**?**].

3) Feasibility Check: includes workspace limit check for fingertips, task mechanics check, and finger relocation check.

Features, action probability, reward, value estimation, and search parameters are the same as the Manipulation with Environment Interactions task.

C. Robot Experiment: Dexterous DDHand

1) Dexterous DDHand Overview: Dexterous DDHand is a direct-drive hand with 4 Dofs. It has two fingers and each finger has 2 Dofs for planar translation motions. Each fingertip is a horizontal rod. As a result, we use two endpoints of the rod to approximate the line contact. We provide the planner with the forward and inverse kinematics of the hand. We also provide a contact relocation planner, which follows the object surface (5mm above the object surface) and goes to the new contact location.

2) Feasibility Checks: include inverse kinematics check, collision check, finger relocation force check, finger relocation path check (are there collisions on the relocation path), and task mechanics check.

Task mechanics, features, action probability, reward, value estimation, and search parameters are the same as the Manipulation with Extrinsic Dexterity task.

3) Execution: Given a planned fingertip trajectory, we compute the robot joint trajectory using inverse kinematics and execute it with robot joint position control. In order to ensure some contact force, we shift the end-effector trajectory in the environment contact normal direction for

$$\Delta\text{position} = \frac{\text{Desired contact force}}{\text{Stiffness}} \quad (7)$$

where the stiffness can be tuned due to the direct-drive property.

The execution was conducted in an open-loop manner, meaning that there was no object pose estimation or force control involved. The system was calibrated to ensure that the initial object pose errors are kept within a tolerance of 1 mm. We chose not to provide a formal success rate in our report since this number lacks significance due to its dependency on the accuracy of our manual calibration process. However, as a point of reference, with an initial pose precision of 1 mm, we estimate a success rate of approximately 4 out of 5 attempts.

D. Robot Experiment: Delta Array

1) Delta Array System Overview: The array of soft delta robots is a research platform for the development of multi-robot cooperative dexterous manipulation skills. The system is comprised of 64 soft linear delta robots arranged in an 8x8 hexagonal tessellating grid. Each 3D printed soft delta linkage is actuated using 3 linear actuators to give 3 degrees of translational freedom with a workspace of 3.5cm radius in the X, and Y axes and 10cm in Z-axis. The links are compliant with high elasticity and low hysteresis, with a soft 3D printed fingertip-like end-effector attached to it. We simplify the workspace of each delta robot to be a cylinder with a 2.5cm radius and 6cm height.

We provide the forward and inverse kinematic models to the planner. While running the planner, the IK check is simplified to a workspace limit check (if the contact point is in the cylinder workspace). We only perform collision checks for the fingertips, not the links. While doing the actual execution of the plans, we use inverse kinematics to calculate the robot joint trajectory from the contact point trajectory. In order to ensure some contact force, we shift the end-effector trajectory in the same way as Equation 7, where the stiffness is manually calibrated.

We relocate contacts by letting the delta robot to leave the contact in the contact normal direction, go around the edge of the workspace, and come to the new contact in its normal direction. The entire plan is executed in open-loop. Although delta robots may not offer a high level of accuracy and repeatability, their passive compliance allows for minor deviations from the planned trajectory to be accommodated.

2) Feasibility Check: include workspace limit check, collision check, task mechanics check.

Task mechanics, features, action probability, reward, value estimation, and search parameters are the same as the Manipulation with Extrinsic Dexterity task.

APPENDIX III
RRT FOR ROLLLOUT

The RRT process is summarized in Algorithm 1. The inputs are the current object pose x_{current}, selected contact mode m_{selected}, and the object goal pose x_{goal}. If it can find a solution, it outputs a trajectory from x_{current} to x_{goal}. Every point on the trajectory is (x, m), where $x \in \text{SE}(3)$ is an object pose, m is an environment contact mode.

At each iteration, SAMPLE-RANDOM-OBJECT-POSE sample a new object pose $x_{\text{extend}} \in \text{SE}(3)$. We find the nearest neighbor x_{near} of x_{extend}, and attempt to extend it towards x_{extend} (line 5 - 15, Algorithm 1). Each extension is performed under the guidance of a contact mode. If x_{near} happens to be x_{current}, we let the contact mode be m_{selected} chosen by Level 1 MCTS. Otherwise, the function SELECT-CONTACT-MODE will select the contact mode(s) to perform the extension under. The procedure EXTEND-WITH-CONTACT-MODE extends x_{near} towards x_{extend} under the guidance of a selected contact mode m through projected forward integration.

Algorithm 1 RRT for Expansion and Rollout

1: **procedure** RRT-EXPLORE(x_{current}, m_{selected}, x_{goal})
2: **while** resources left and the goal is not reached **do**
3: $x_{rand} \leftarrow$ SAMPLE-RANDOM-OBJECT-POSE($x_{goal}, p_{\text{sample}}$)
4: $x_{\text{near}} \leftarrow$ NEAREST-NEIGHBOR(x_{rand})
5: **if** $x_{\text{near}} = x_{\text{current}}$ **then**
6: $\mathcal{M} \leftarrow \{m_{\text{selected}}\}$
7: **else**
8: $\mathcal{M} \leftarrow$ SELECT-CONTACT-MODES($x_{\text{near}}, x_{\text{rand}}$)
9: **end if**
10: **for** $m \in \mathcal{M}$ **do**
11: $x_{\text{new}} \leftarrow$ EXTEND-WITH-CONTACT-MODE($x_{\text{near}}, x_{\text{rand}}, m$)
12: **if** $x_{\text{new}} \neq$ null **then**
13: ADD-TO-RRT-TREE(x_{new}, $\mathcal{T}_{\text{rrt}}$)
14: **end if**
15: **end for**
16: **end while**
17: solution-path $\leftarrow$ BACKTRACK($x_{\text{goal}}, \mathcal{T}_{\text{rrt}}$)
18: **return** solution-path
19: **end procedure**
20: **procedure** SELECT-CONTACT-MODE($x_{\text{near}}, x_{\text{rand}}$)
21: $p_{env} \leftarrow$ ENVIRONMENT-CONTACT-POINT-DETECTION(x_{near})
22: $\mathcal{M}_{env} \leftarrow$ ENUMERATE-CONTACT-MODES(p_{env})
23: **for** all $m \in \mathcal{M}_{env}$ **do**
24: **if** EXTEND-FEASIBILITY-CHECK(m, $x_{\text{near}}, x_{\text{rand}}$) **then**
25: $\mathcal{M} \leftarrow \{M, m\}$
26: **end if**
27: **end for**
28: **return** $\mathcal{M}$
29: **end procedure**
30: **procedure** EXTEND-WITH-CONTACT-MODE($x_{\text{near}}, x_{\text{rand}}, m$)
31: $x_{\text{now}} = x_{\text{near}}$
32: **while** true **do**
33: **if** not EXTEND-FEASIBILITY-CHECK(m, $x_{\text{now}}, x_{\text{rand}}$) **then**
34: break
35: **end if**
36: $v \leftarrow$ VELOCITY-UNDER-MODE(m, $x_{\text{now}}, x_{\text{rand}}$)
37: **if** v close to zero **then**
38: break
39: **end if**
40: ▷ Projected forward integration
41: $x_{\text{now}} \leftarrow$ INTEGRATE(x_{now}, v)
42: $x_{\text{now}} \leftarrow$ PROJECT-TO-CONTACTS-MAINTAINED(x_{now}, m)
43: **if** encounter new contacts **then**
44: break
45: **end if**
46: **end while**
47: **return** x_{now}
48: **end procedure**

Next, we explain all the functions in detail.

SAMPLE-RANDOM-OBJECT-POSE sample a new object pose $x_{\text{extend}} \in \text{SE}(3)$. The probability of x_{extend} being the goal pose is p_{sample}, while the probability of it being a random object pose in SE(3) is $1 - p_{\text{sample}}$. We can specify the range limit for the random sample of the object pose.

NEAREST-NEIGHBOR finds the closest object pose to x_{extend} in the tree. The distance between two object poses is computed as $w_t * d_t + w_r * d_r$. w_t and w_r are the weights for translation and rotation. d_t is the Euclidean distance between their locations, and d_r is the angle difference between two rotations. A simple way to compute d_r is to first compute the rotation between two poses $R_{\text{diff}} = R_1 R_2^T$, and convert R_{diff} to axis-angle representation and let d_r be equal to the angle. In general, the users need to adjust the weights according to how important object orientation or position is important in the task. Not much tuning is needed. In our experiment, we scale the object sizes such that the average length of their bounding boxes is about 1 to 10. In this case, one can set the weights using this rule: normal (1), not very important (0.5), not important at all (0.1).

SELECT-CONTACT-MODE first enumerates all contacting-separating contact modes, then filters out infeasible mode through EXTEND-FEASIBILITY-CHECK, and finally returns the set of all feasible modes.

EXTEND-FEASIBILITY-CHECK has two options in implementation. The first option involves storing the robot contacts during the search process. If the current robot contacts pass the feasibility check in Section **??**, the check is deemed successful. However, if they fail, the check can still be considered successful if a sampled set of feasible robot contacts can be generated, ensuring a feasible transition from the current contacts. The current contacts are then updated accordingly. The second option finds if there exists a set of robot contacts that satisfy the feasibility check. Unlike the first option, this method does not retain information about robot contacts. Instead, it is consider successful if it can sample any set of robot contacts, as long as they pass the feasibility check. The second option is more relaxed as it does not take into account previous robot contacts and transitions.

EXTEND-WITH-CONTACT-MODE extends x_{near} towards x_{extend} as much as possible under constraints posted by contact mode m. VELOCITY-UNDER-MODE solves for the object velocity that get x_{near} as close as possible to x_{extend} with respect to the velocity constraints introduced by m. We then integrate the object pose for a small step in the direction of constrained object velocity, and project the new object pose back to the contacts that needed to be maintained.

In PROJECT-TO-CONTACTS-MAINTAINED, the contact mode m needs to be maintained. We first perform contact detection on the object. We then project the object pose back to where the maintaining contacts in m have zero signed distances.