sim-to-real RL to solve dexterous manipulation, this real-to-sim modeling problem is further exacerbated by the necessity to model objects, which have great variability and whose full physical properties cannot be easily quantified. Even when one assumes that the ground-truth physical parameters are known, quantitatively matching the simulation with the real world is hard: due to the limitations of physics engines, the same values for physical constants in simulation and the real world do not necessarily correspond to identical kinematic and dynamic relationships.

Autotuned robot modeling. While robot manufacturers are often able to provide proprietary model files for their robot hardwares, the models mostly serve a starting reference for robot real-to-sim effort rather than ground truth models that can be used without modifications. Empirical solutions to increase modeling accuracy range from hand-tuning the robot model constants and simulatable physical parameters [2] to re-formulating specific kinematic structures (e.g., four-bar linkage) in the simulator of choice [44]. This is a laborious process as there is no "ground truth" pairing between the real world and the simulated world. We propose a practical technique to speed up this real-to-sim modeling process via an "autotune" module. The autotune module enables rapid calibration of simulator parameters to match real robot behavior by automatically searching the parameter space to identify optimal values for both simulator physics and robot model constants in under four minutes (or 2000 simulated steps in 10 Hz). We illustrate the module in Figure 2A and Algorithm 1. The module operates on two parameter types: simulator physics parameters affecting kinematics and dynamics, as well as robot model constants from the URDF file (including link inertia values, joint limits, and joint/link poses). The calibration process begins by initializing multiple simulated environments using randomly sampled parameter combinations from the parameter space, bootstrapped from the manufacturer's robot model file. It then executes N calibration sequences consisting of joint position targets on both the real robot hardware (single run) and all simulated environments in parallel. By comparing the tracking error between each simulated environment and the real robot when following the same joint targets, the module selects the parameter set that minimizes the mean squared error in tracking performance. This approach eliminates iterative manual tuning by requiring only one set of calibration runs on the real robot, automatically optimizing traditionally hard-to-tune URDF parameters, and supporting parallel evaluation of multiple parameter combinations. The method's generality allows it to tune any exposed simulator or robot model parameter that affects kinematic behavior.

Approximate object modeling. As demonstrated in previous works [31, 41], modeling objects as primitive shapes like cylinders with randomized parameters is sufficient for sim-to-real transferrable dexterous manipulation policies to be learned. Our recipe adopts this approach and finds it effective.

Algorithm 1 Real-to-Sim Autotune Module

```
Require:
 1: $E$ : Set of environment parameters to tune
 2: $N$ : Number of calibration action sequences
 3: $R$ : Real robot hardware environment
 4: $M$ : Initial robot model file
 5: procedure AUTOTUNE($E, N, R, M$)
 6:     $P \leftarrow$ InitializeParameterSpace($E, M$)    ▷ Initialize from model
 7:     $S \leftarrow \{\}$    ▷ Set of simulated environments
 8:     for $i \leftarrow 1$ to $K$ do    ▷ $K$ is population size
 9:         $p_i \leftarrow$ RandomSample($P$)
10:         $S_i \leftarrow$ CreateSimEnvironment($p_i$)
11:         $S \leftarrow S \cup \{S_i\}$
12:     end for
13:     $J \leftarrow$ GenerateJointTargets($N$)    ▷ Joint target sequences
14:     $R_{track} \leftarrow$ GetTrackingErrors($R, J$)    ▷ Real tracking
15:     $best\_params \leftarrow$ null
16:     $min\_error \leftarrow \infty$
17:     for $S_i \in S$ do
18:         $S_{track} \leftarrow$ GetTrackingErrors($S_i, J$)
19:         $error \leftarrow$ ComputeMSE($S_{track}, R_{track}$)
20:         if $error < min\_error$ then
21:             $min\_error \leftarrow error$
22:             $best\_params \leftarrow$ GetParameters($S_i$)
23:         end if
24:     end for
        return $best\_params$
25: end procedure
```

B. *Generalizable Reward Design*

In the standard formulation of RL [51], the reward function is a crucial element within the paradigm because it is solely responsible for defining the agent's behavior. Nevertheless, the mainstream of RL research has been preoccupied with the development and analysis of learning algorithms, treating reward signals as given and not subject to change [13]. As tasks of interest become more general, designing reward mechanisms to elicit desired behaviors becomes more important and more difficult [12] — as is the case of applications to robotics. When it comes to dexterous manipulation with multi-fingered hands, reward design becomes even more difficult due to the variety of contact patterns and object geometries.

Manipulation as contact and object goals. From a wide variety of human manipulation activities [15], we observe a general pattern in dexterous manipulation: each motion sequence to execute a task can be defined as a combination of hand-object contact and object states. Building on this intuition, we propose a general reward design scheme for even long-horizon contact-rich manipulation tasks. For each task of interest, we first break it down into an interleaving sequence of contact states and object states. For example, the handover task can be broken down into the following steps: (1) one