

424 Appendix

425 A Task Generation Details

426 A.1 Asset Statistics

427 We summarize the information of the articulated asset library we used in this project in Table 3. To
428 ease the workload of the LLM in understanding the manipulatable parts of an articulated object, we
429 only reserve one unfixed (prismatic or revolute) joint in each object and fix the others. For rigid body
430 objects, we build a dataset adapted from the YCB object dataset in [Maniskill2](#). We have modified
431 the scale and joint limits of the objects to make the scene proper.

432 A.2 Task Generation Statistics

433 We investigate the generated tasks in the task generation process in this section. We have generated
434 100 tasks based on the asset library described in Appendix A.1, including 50 single-step tasks and 50
435 long-horizon tasks. Notice that we consider tasks with the same object class but different instances
436 as the same tasks. Despite the tasks we have generated, these tasks do not include all assets in the
437 asset library and the asset library can also be expanded to contain more object classes and instances.

438 A.3 Demonstration Generation Statistics

439 We study the statistics of the generated demonstrations in detail. We test the success rates of gener-
440 ating demonstrations for 24 single-step and 15 long-horizon tasks, and report them task by task in
441 Table 4. For each task, we run the demo collection process 5 times and each time the kPAM solver
442 is executed for 50 episodes with both spatial and object randomization. We compute the mean and
443 standard deviation of the 5 runs on the success rate. For some difficult long-horizon tasks, in which
444 the articulated object is so small that it may be struggling to place something inside, we fail to gen-
445 erate successful demonstrations with normal randomization, so we reduce the randomization range
446 by half and re-evaluate their success rates.

447 A.4 Prompt Examples

448 In this section, we demonstrate some examples of the prompts we have used to query LLM.
449 The prompt templates are shown as follows:

Prompt: Task Proposal

You are an expert in creating robotic simulation environments and tasks. You are given some artic-
ulated assets for example. Please come up with a creative use of the gripper to manipulate a single
articulated object. Note that the simulation engine does not support deformable objects or accurate
collision models for contacts. Moreover, the robot can only execute a simple trajectory of one mo-
tion.

=====

Here are all the assets. Please try to come up with tasks using only these assets.

...

=====

Here are some examples of good tasks. Try to learn from these structures but avoid overlapping
with them.

...

=====

Here are some bad example task instances with reasons. Try to avoid generating such tasks.

...

reasons: ...

=====

Please describe a NEW task in natural languages and explain its novelty and challenges.

Note:

- Do not use assets that are not in the list above.

450

- Do not repeat the tasks similar to the good examples or the already generated tasks.
- The task needs to obey physics and remain feasible.
- Do not create similar tasks with the same “assets-used” set.
- All the assets are on the table when initialized.
- Do not place objects on small objects.
- Only one articulated object can be loaded.
- The task contains a simple trajectory of one motion.
- The task should have a clear goal, e.g. use “open/close” instead of “adjust position”.

Before the next step, please check if the generated task is a bad task shown in the above examples and meets all the criteria as stated above. Specifically, if the task ****only**** contains a simple trajectory of one motion, and should have a ****clear**** goal. Explain in detail, and get a conclusion. If the task is a bad task, regenerate a new one.

Then, format the answer in a Python dictionary with keys “task-name” and value type string, “task-description” (one short phrase), and value type string with lower-case and separated by hyphens, “assets-used” and value type list of strings, and “success-criteria” (choose from “articulated_open”, “articulated_closed”, “distance_articulated_rigidbody”, “distance_gripper_rigidbody”, and “distance_gripper_articulated”) and value type list of strings. Try to be as creative as possible.

Please remember not to add any extra comments to the Python dictionary.
Let’s think step by step.

451

Prompt: Task Decomposition

You are an expert in creating robotic simulation environments and tasks. A robot arm with a 2-finger gripper is used in all the robotic simulation environments and tasks. In each task, there is exactly one articulated object and one rigid body object that you can manipulate. You will be given long-horizon tasks with each task including at least 2 sub-tasks. Each sub-task can only include one simple motion such as moving the gripper to some object, opening or closing the gripper fingers, or interacting with certain articulated objects by its prismatic/revolute joints.

Please come up with a decomposition of the given long-horizon task to get several sub-tasks. Some rules of such decomposition are listed here:

1. Each long-horizon task can not include over 5 sub-tasks, and usually 3-4 are enough.
2. Each sub-task should only include one simple motion as mentioned.
3. Each sub-task(except “grasp” and “ungrasp”) should be presented in the format of a Python dictionary with keys “task-name” and value type string with lower-case and separated by hyphens, “task-description” (one specific sentence) and value type string, “assets-used” and value type list with necessary asset(s) in the current sub-task, and “success-criteria” (choose from “articulated_open”, “articulated_closed”, “distance_articulated_rigidbody”, “distance_gripper_rigidbody”, and “distance_gripper_articulated”) and value type list of strings.
4. Each sub-task should have only one asset used in the task.
5. If the motion of opening or closing the gripper fingers is included in the whole task, it should be listed as a separate sub-task, whose “task-name” should strictly be “grasp” or “ungrasp” respectively and should be the only key in the dictionary.

=====
Here is an example of the decomposition of the following long-horizon task “...”:

```
...
# Sub-task 1
...
# Sub-task 2
...
# Sub-task 3
...
# Sub-task 4
...
=====
```

Now please start to generate a sub-task decomposition of the following new task:

```
...
```

452

Prompt: Code Generation

Now I will provide you with some reference code and you can write the code for the task “TASK_NAME_TEMPLATE”.

...

=====

The generated code should follow the same structure as the reference and call similar functions. Do not use libraries, extra functions, properties, arguments, or assets that you don't know. Remember to import used functions from the corresponding files as the example task codes. For the articulated object, use “self.articulator” to refer to it, which should be the same as the “assets-used” of the task.

For the objects used, you only have to pass the corresponding parameter (e.g, articulator) and its name (string format, e.g., ‘box’) in the ‘__init__’ function as shown in above codes, and the base class will automatically load them.

Please comment on the code to explain what each piece does and why it's written that way.

Now write the code for the task “TASK_NAME_TEMPLATE” in the Python code block.

453

Prompt: kPAM Solver Stage 1

You are an expert in solving robotic tasks by coding task solution configs. Now please solve the newly generated task by generating the task solution config.

The task solution config contains the necessary positions, parameters, and keypoints for an existing trajectory optimization algorithm to solve a feasible solution. It mainly contains two parts, constraints and pre/post-actuation motions:

- (1) The constraints are used to ensure the gripper is in contact with the object and to implicitly define a certain actuation pose.
- (2) The pre-actuation motions are used to move the gripper to the actuation pose, while the post-actuation motions are used to complete the task after the actuation pose.

=====

Here is the task description.

...

=====

Here are all the available keypoint names for the used manipulator and asset and their descriptions.

...

=====

Here are some examples of the constraint part of some configs.

...

=====

Note that, in the constraint list, you need to define different items of constraint to define an actuation pose for the task. There are some pre-defined types of constraints you can use:

- (1) point2point.constraint: This constraint is used to ensure two keypoints (“keypoint_name” and “target_keypoint_name”, respectively on the tool and object) are in contact.
 - (2) frame_axis_parallel: This constraint is used to ensure two axes (respectively on the tool and object) are parallel. The axis on the tool is defined by a unit vector from “axis_from_keypoint_name” to “axis_to_keypoint_name”, while the axis on the object is defined by “target_axis”([1,0,0] or [0,1,0] or [0,0,1]) which is in the coordinate frame of “target_axis.frame”(world or object).
 - (3) frame_axis_orthogonal: This constraint is used to ensure two axes (respectively on the tool and object) are orthogonal. The axis on the tool is defined by a unit vector from “axis_from_keypoint_name” to “axis_to_keypoint_name”, while the axis on the object is defined by “target_axis”([1,0,0] or [0,1,0] or [0,0,1]) which is in the coordinate frame of “target_axis.frame”(world or object).
 - (4) keypoint_axis_parallel: This constraint is used to ensure two axes (respectively on the tool and object) are parallel. The axis on the tool is defined by a unit vector from “axis_from_keypoint_name” to “axis_to_keypoint_name”, while the axis on the object is defined by another unit vector from “target_axis_from_keypoint_name” to “target_axis_to_keypoint_name”.
 - (5) keypoint_axis_orthogonal: This constraint is used to ensure two axes (respectively on the tool and object) are orthogonal. The axis on the tool is defined by a unit vector from “axis_from_keypoint_name” to “axis_to_keypoint_name”, while the axis on the object is defined by another unit vector from “target_axis_from_keypoint_name” to “target_axis_to_keypoint_name”.
- The tolerance is used to define the tolerance of the constraint.

454

The `target_inner_product` is used to define the inner product between the two axes. For example, if you want to ensure two axes are parallel and in the same direction, you can set the `target_inner_product` to 1.0. If you want to ensure two axes are parallel and of opposite directions, you can set the `target_inner_product` to -1.0. If you want to ensure two axes to be orthogonal, you can set the `target_inner_product` to 0.0.

Usually, you need to define one `point2point_constraint` to ensure contact and several axis constraints to adjust the actuation pose.

=====

Now please first generate the constraint part for task “TASK_NAME_TEMPLATE” in the same config format as the above.

Do not use terms that you have not seen before.

The output should be in the YAML format with no extra text.

455

Prompt: kPAM Solver Stage 2

You are an expert in solving robotic tasks by providing some motion plans and coding task solution configs for a 2-finger robot arm. Now please solve the newly generated task by generating the task solution config.

The task solution config contains the necessary positions, parameters, and keypoints for an existing trajectory optimization algorithm to solve a feasible solution. It mainly contains two parts, constraints and pre/post-actuation motions:

- (1) The constraints are used to ensure the gripper is in contact with the object and to implicitly define a certain actuation pose. An actuation pose means the key frame that the robot arm manipulates the object, usually representing the moment when the gripper gets contact with the object.
- (2) The pre-actuation motions are used to move the gripper to the actuation pose, while the post-actuation motions are used to complete the task after the actuation pose.

=====

Here is the task description.

...

=====

Here are all the available keypoint names for the used manipulator and asset and their descriptions.

...

=====

Here are some examples of the pre/post-actuation part of some task solution configs.

...

=====

Now please generate SOLVER_TRIALS different pre/post-actuation motions for task “TASK_NAME_TEMPLATE” following the same config format shown above based on the constraint part that is generated previously. The pre/post-actuation motions of different solutions can be diverse, but their task names should be the same.

Do not use terms that you have not seen before.

The output should be in the YAML format with no extra text.

The diversity of the motions can be achieved by using different axes for translation.

Notice that the pre-actuation and post-actuation motions are relative to the actuation pose and the translation motions are represented in coordinates relative to the manipulator base.

Let’s think step by step, and try your best to understand the job.

456

Table 3. Statistics of assets used in GenSim2

| Object | Num | Joint Type | Asset Description |
|-----------------|-----|------------|---|
| laptop_rotate | 44 | revolute | A laptop fixed on the table, with a lid connected by a revolute joint that can be opened and closed |
| stapler_move | 3 | prismatic | A stapler that can be moved on the table |
| window_push | 2 | prismatic | A window with a frame connected by a prismatic joint that can be pushed left and right |
| suitcase_move | 3 | prismatic | A suitcase that can be moved on the table |
| oven | 4 | revolute | An oven fixed on the table, with a door connected by a revolute joint that can be opened and closed |
| drawer | 20 | prismatic | A drawer fixed on the table, connected with its body by a prismatic joint that can be opened and closed |
| box_move | 1 | prismatic | A box that can be moved on the table |
| kitchen_pot | 3 | prismatic | A kitchen bot fixed on the table, connecting its lid and body with a prismatic joint |
| bucket_lift | 5 | prismatic | A bucket that can be lifted from the table |
| trash_can | 3 | revolute | A trash can fixed on the table, with a lid connected by a revolute joint that can be opened and closed |
| stapler_press | 3 | revolute | A stapler fixed on the table, with its handle connected to its body by a revolute joint that can be pressed |
| bottle | 2 | revolute | A bottle fixed on the table, with a top connected to its body by a prismatic joint |
| bag_move | 2 | prismatic | A bag that can be moved on the table |
| dishwasher | 3 | revolute | A dishwasher with a door connected by a revolute joint that can be opened and closed |
| suitcase_rotate | 4 | revolute | A suitcase fixed on the table, with a lid connected by a revolute joint that can be opened and closed |
| toaster_move | 10 | prismatic | A toaster that can be moved on the table |
| bag_swing | 2 | revolute | A bag fixed on the table, with a strap connected by a revolute joint |
| toilet | 2 | revolute | A toilet fixed on the table, lid connected by a revolute joint |
| door | 2 | revolute | A door with its frame connected by a revolute joint that can be opened and closed |
| coffee_machine | 2 | prismatic | A coffee machine fixed on the table, with a button connected by a prismatic joint to be pressed |
| faucet | 13 | revolute | A faucet fixed on the table, with a handle connected by a revolute joint that can be turned on and off |
| washing_machine | 2 | revolute | A washing machine fixed on the table, with a door connected by a revolute joint |
| switch | 2 | revolute | A switch with a frame connected by a revolute joint |
| bucket_swing | 26 | revolute | A bucket fixed on the table, with a handle connected by a revolute joint |
| safe_move | 2 | prismatic | A safe that can be moved on the table |
| toaster_press | 10 | prismatic | A toaster fixed on the table, with a button connected by a prismatic joint |
| window_rotate | 2 | revolute | A window with a frame connected by a revolute joint |
| refrigerator | 2 | revolute | A refrigerator fixed on the table, with a door connected by a revolute joint that can be opened and closed |
| laptop_move | 3 | prismatic | A laptop that can be moved on the table |
| safe_rotate | 15 | revolute | A safe fixed on the table, with a door connected by a revolute joint that can be opened and closed |
| bag_lift | 2 | prismatic | A bag that can be lifted from the table |
| microwave | 10 | revolute | A microwave fixed on the table, with a door connected by a revolute joint that can be opened and closed |
| box_rotate | 13 | revolute | A box fixed on the table, with a lid connected by a revolute joint that can be opened and closed |
| bucket_move | 2 | prismatic | A bucket that can be moved on the table |

Table 4. Success Rates of demonstration generation on different tasks

| Task Name | Task Type | Success Rate |
|---------------------------------|-----------------------|-----------------|
| OpenBox | Single-step | 0.84 ± 0.07 |
| CloseBox | Single-step | 0.94 ± 0.03 |
| OpenLaptop | Single-step | 0.76 ± 0.03 |
| CloseLaptop | Single-step | 0.95 ± 0.01 |
| TurnOnFaucet | Single-step | 0.67 ± 0.05 |
| TurnOffFaucet | Single-step | 0.72 ± 0.03 |
| OpenDrawer | Single-step | 0.80 ± 0.02 |
| PushDrawerClose | Single-step | 0.87 ± 0.06 |
| SwingBucketHandle | Single-step | 0.89 ± 0.03 |
| PressToasterLever | Single-step | 0.96 ± 0.03 |
| RotateMicrowaveDoor | Single-step | 0.92 ± 0.01 |
| CloseSafe | Single-step | 0.80 ± 0.04 |
| OpenSafe | Single-step | 0.62 ± 0.03 |
| PushToasterForward | Single-step | 0.99 ± 0.01 |
| CloseSuitcaseLid | Single-step | 0.82 ± 0.05 |
| SwingSuitcaseLidOpen | Single-step | 0.81 ± 0.06 |
| RelocateSuitcase | Single-step | 0.80 ± 0.04 |
| LiftBucketUpright | Single-step | 0.76 ± 0.05 |
| MoveBagForward | Single-step | 0.72 ± 0.05 |
| CloseMicrowave | Single-step | 0.53 ± 0.04 |
| SwingDoorOpen | Single-step | 0.79 ± 0.03 |
| ToggleDoorClose | Single-step | 0.80 ± 0.05 |
| CloseRefrigeratorDoor | Single-step | 0.82 ± 0.03 |
| OpenRefrigeratorDoor | Single-step | 0.75 ± 0.06 |
| PlaceGolfBallIntoDrawer | Long-horizon (simple) | 0.56 ± 0.09 |
| PlaceCrackerBoxIntoDrawer | Long-horizon (simple) | 0.53 ± 0.08 |
| PlaceLemonIntoDrawer | Long-horizon (simple) | 0.53 ± 0.06 |
| PlaceSoftBallIntoDrawer | Long-horizon (simple) | 0.58 ± 0.06 |
| DropAppleIntoDrawer | Long-horizon (simple) | 0.58 ± 0.05 |
| StachCupInBox | Long-horizon (simple) | 0.40 ± 0.09 |
| PlaceGolfBallIntoBox | Long-horizon (simple) | 0.58 ± 0.06 |
| PutCrackerBoxInBox | Long-horizon (simple) | 0.48 ± 0.08 |
| StoreBlockInBox | Long-horizon (simple) | 0.62 ± 0.03 |
| DropAppleIntoBox | Long-horizon (simple) | 0.64 ± 0.04 |
| StoreLemonInRefrigerator | Long-horizon (hard) | 0.48 ± 0.07 |
| PlaceCrackerBoxIntoRefrigerator | Long-horizon (hard) | 0.26 ± 0.09 |
| SecureGoldInSafe | Long-horizon (hard) | 0.21 ± 0.05 |
| FillMugWithWater | Long-horizon (hard) | 0.13 ± 0.04 |

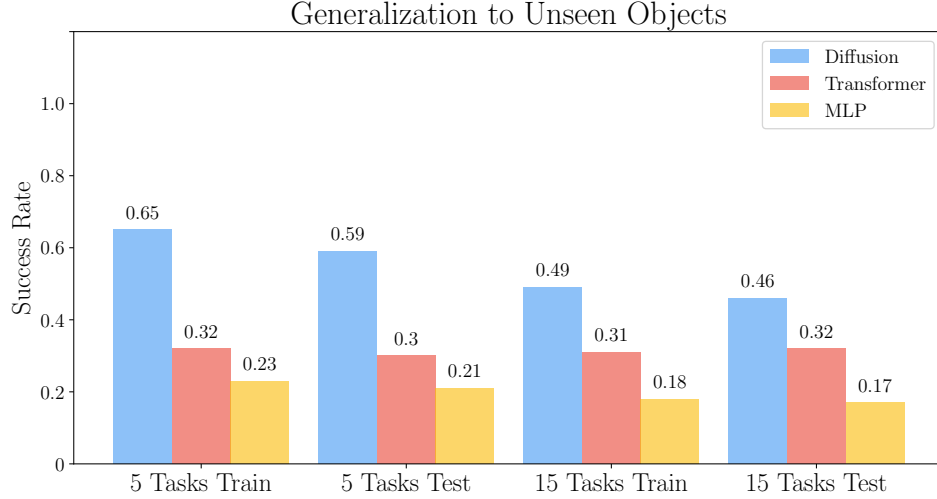


Figure 6. Results for object-level generalization test.

B Multi-Task Training Details

B.1 Architecture Implementation

We implemented a multi-modal policy architecture, as illustrated in Fig. 3, which includes a transformer policy stem, an action head, and different encoders for modeling various types of observations as tokens.

Specifically, we take the CLIP [49] tokenizer to encode the task instruction, which is frozen during training; a pre-trained PointNext [50] to encode the pointcloud, which is fine-tuned during training; an MLP for encoding the proprioception states, which is trained from scratch. The transformer conducts self-attention over all tokens, and then we post-process the tokens for different action heads. For example, we compute the mean pooling of all tokens as the global condition of the diffusion and the MLP head; for the transformer, we compute cross-attention between the modeled token and a set of position embeddings to get the final action sequence.

Regarding the PointNext, we utilize the pre-trained model on ScanObjectNN Classification¹, as it does not include any color information, making it easier for downstream sim-real transfer.

B.2 Additional Experiments for Object-Level Generalization

After multi-task training, we test the generalization ability of our policy to unseen object instances. We choose tasks using assets with more than 20 instances and 10 instances to respectively construct two training sets of 5 tasks and 15 tasks. During training, we only leverage data generated from 90% of the instances and leave the remaining 10% for testing. The initial poses of the objects are randomly initialized in both procedures as mentioned.

We find that the success rates only drop by around 5% on unseen instances, as shown in Fig. 6. These results indicate that by generating data with object-level and spatial-level variance as domain randomization, our policy can acquire generalization in both aspects, which lays the foundation for further sim-to-real transfer.

¹https://drive.google.com/drive/folders/1A584C9x5uAqppbjNNiVq1A_7u0001EII?usp=sharing

C Real-World Details

We selected 7 real world tasks to evaluate the multi-task policy trained on generated data from our pipeline. The *Open Laptop* task required the robot to fully open a partially closed laptop, similarly *Close Laptop* closed the laptop lid. The *Open Safe* and *Close Safe* performed similar opening and closing motions on an articulated door. The *Close Drawer* task required the robot to reach and close an open drawer. The generated *Swing Bucket* task required the robot to push a handle perpendicular to it's current position and the *Open Box* task required it to open a box lid.

As we didn't utilize a discrete action space such as keyframes for our real-world experiments, real-time control was important. We achieved fast inference speeds with our multi-task policy with 0.1s inference latency on an NVIDIA 3080 GPU. We parallelized and synchronized pointcloud processing from 3 Intel Realsense D435 cameras in order to prevent additional latency. Processing of the pointclouds involved a uniform sampling step for time efficiency, farthest point sampling and outlier removal to ensure successful Sim2Real transfer.