

ROBOGEN: TOWARDS UNLEASHING INFINITE DATA FOR AUTOMATED ROBOT LEARNING VIA GENERATIVE SIMULATION

Yufei Wang^{*,1}, Zhou Xian^{*,1}, Feng Chen^{*,2}, Tsun-Hsuan Wang³, Yian Wang⁴, Katerina Fragkiadaki¹, Zackory Erickson¹, David Held¹, Chuang Gan^{4,5}

¹CMU, ²Tsinghua IIIS, ³MIT CSAIL, ⁴UMass Amherst, ⁴MIT-IBM AI Lab

<https://robogen-ai.github.io/>

ABSTRACT

We present *RoboGen*, a *generative robotic agent* that automatically learns diverse robotic skills at scale via *generative simulation*. RoboGen leverages the latest advancements in foundation and generative models. Instead of directly using or adapting these models to produce policies or low-level actions, we advocate for a generative scheme, which uses these models to automatically generate diversified tasks, scenes, and training supervisions, thereby scaling up robotic skill learning with minimal human supervision. Our approach equips a robotic agent with a self-guided *propose-generate-learn* cycle: the agent first proposes interesting tasks and skills to develop, and then generates corresponding simulation environments by populating pertinent objects and assets with proper spatial configurations. Afterwards, the agent decomposes the proposed high-level task into sub-tasks, selects the optimal learning approach (reinforcement learning, motion planning, or trajectory optimization), generates required training supervision, and then learns policies to acquire the proposed skill. Our work attempts to extract the extensive and versatile knowledge embedded in large-scale models and transfer them to the field of robotics. Our fully generative pipeline can be queried repeatedly, producing an *endless* stream of skill demonstrations associated with diverse tasks and environments.

1 INTRODUCTION

This work is motivated by a long-standing and challenging goal in robotics research: empowering robots with a diverse set of skills, enabling them to operate in various non-factory settings and perform a broad range of tasks for humans. Recent years have witnessed impressive progress in teaching robots various complex skills: from deformable object and fluid manipulation (Lin et al., 2022a; Weng et al., 2022; Xu et al., 2023; Xian et al., 2023c; Wang et al., 2023c; Lin et al., 2020), to dynamic and dexterous skills such as object tossing (Zeng et al., 2020), in-hand re-orientation (Chen et al., 2022), soccer playing (Haarnoja et al., 2023) and even robot parkour (Zhuang et al., 2023). However, these skills still remain compartmentalized, have relatively short horizons, and necessitate human-designed task descriptions and training supervision. Notably, due to the expensive and laborious nature of real-world data collection, many of these skills are trained in *simulations* with appropriate domain randomization and then deployed to real-world (Xu et al., 2023; Zhuang et al., 2023; Chen et al., 2022).

Indeed, simulation environments have become a crucial driving force behind diverse robotic skill learning (Lin et al., 2022a; Song et al., 2023; Zhuang et al., 2023). Compared to exploration and data collection in the real-world, skill learning in simulations offers several advantages: 1) simulated environments provide access to privileged low-level states and unlimited exploration opportunities;

*Equal contribution. Contribution Statement: Yufei Wang led the implementation of the whole project. Zhou Xian conceptualized the idea, contributed to the locomotion pipeline, and supervised the direction of the project. Feng Chen was in charge of the soft-body manipulation pipeline.

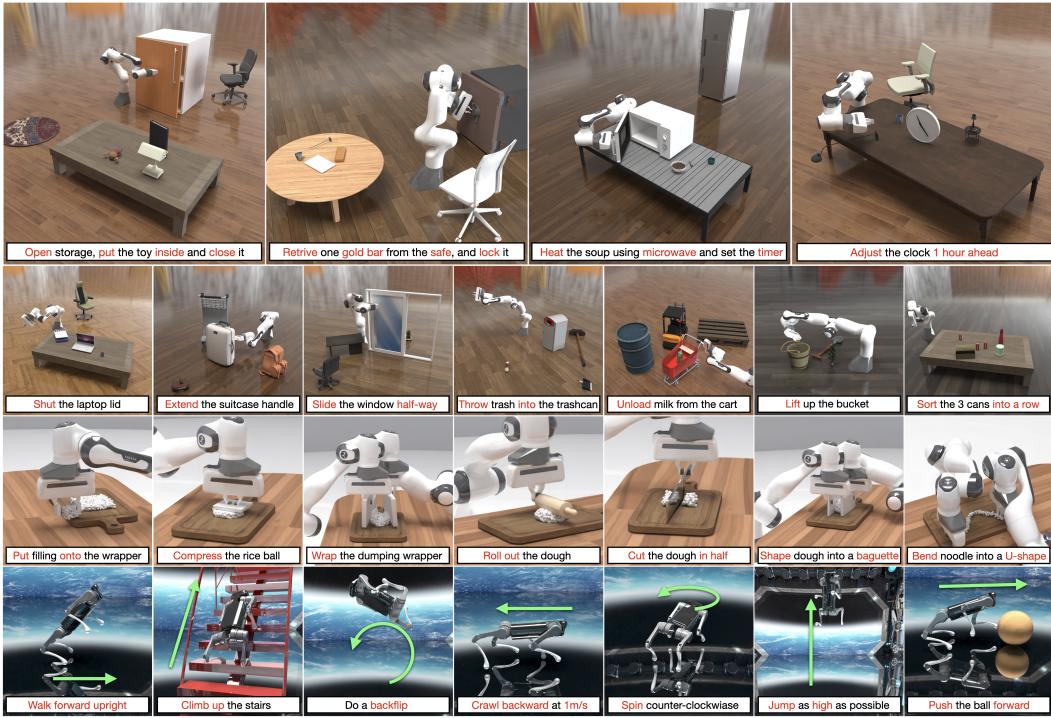


Figure 1: RoboGen is a robotic agent that generates tasks and environments, and acquire skills automatically. Here we show 25 example tasks generated and corresponding skills learned by RoboGen.

2) simulation supports massively parallel computation, enabling significantly faster data collection without reliance on considerable investment in robotic hardware and human labor; 3) exploration in simulation allows robots to develop closed-loop policies and error-recovery capabilities, while real-world demonstrations typically offer only expert trajectories. However, robot learning in simulations also presents its own limitations: while exploration and practicing in simulated environments are cost-effective, constructing these environments requires significant labor effort, demanding tedious steps including designing tasks, selecting relevant and semantically meaningful assets, generating plausible scene layouts and configurations, and crafting training supervisions such as reward or loss functions (James et al., 2020; Srivastava et al., 2022; Gu et al., 2023; Li et al., 2023a). The onerous task of creating these components and constructing individualized simulation settings of each one of the countless tasks encountered in our daily life is an overwhelming challenge, which significantly constrains the scalability of robotic skill learning even in simulated worlds.

In light of this, we present a paradigm termed *Generative Simulation*, marrying the advancements in simulated robotic skill learning and the latest progress in foundation and generative models. Leveraging the generative capabilities of state-of-the-art foundation models, *Generative Simulation* aims to generate information for all the stages needed for diverse robotic skill learning in simulation: from high-level task and skill proposals, to task-dependent scene descriptions, assets selections and generations, policy learning choices, and training supervisions. Thanks to the comprehensive knowledge encoded in latest foundation models, scene and task data generated this way have potentials to closely resemble the distribution of real-world scenarios. In addition, these models can further provide decomposed low-level sub-tasks, which can be seamlessly handled by domain-specific policy learning approaches, thereby producing closed-loop demonstrations for various skills and scenarios.

A distinct advantage of our proposed paradigm lies in the strategic design of what modes of knowledge to extract from contemporary foundation models. These models have demonstrated impressive capabilities across various modalities (Touvron et al., 2023; Driess et al., 2023; OpenAI, 2023; Rombach et al., 2022; Girdhar et al., 2023; Kang et al., 2023), giving rise to autonomous agents capable of using a range of tools and solving a variety of tasks in the *virtual* realm (Surís et al., 2023; Yang et al., 2023; Shen et al., 2023). However, due to the absence of training data pertaining to *dynamics*,

actuations, and *physical interactions*, these models are yet to fully grasp understandings of what’s essential for robots to effectively execute physical actions and interact with the surrounding environments – from discerning the precise joint torque needed for stable locomotion, to high-frequency finger motor commands needed for dexterous manipulation tasks such as rolling a dough. In contrast to recent efforts that employ these foundation models such as Large Language Models (LLMs) for directly yielding policies or low-level actions (Liang et al., 2022; Huang et al., 2023b; Wang et al., 2023b), we advocate for a scheme that extracts information that falls neatly within the capabilities and modalities of these models - object semantics, object affordances, common-sense knowledge to identify valuable learning tasks, etc. We use these knowledge to construct environmental playgrounds, and then resort to additional help from physics-grounded simulations, for robots to develop understandings of physical interactions and acquire diverse skills.

We first described such a paradigm in a recent white paper (Xian et al., 2023a), sketched as a promising pathway towards generating diverse data for generalist robot learning. In this paper, we present *RoboGen*, a comprehensive realization of this paradigm. RoboGen is a generative robotic agent that self-proposes skills to learn, generates scene components and configurations in simulation, labels the tasks with natural language descriptions, and designs proper training supervisions for subsequent skill learning. Our experiments show that RoboGen can deliver a continuous stream of diversified skill demonstrations, spanning tasks including rigid and articulated object manipulation, deformable object manipulation, as well as legged locomotion skills (see Figure 1). The diversity of tasks and skills generated by RoboGen surpasses previous human-created robotic skill learning datasets, with minimal human involvement needed beyond several prompt designs and in-context examples. Our work attempts to extract the extensive and versatile knowledge embedded in large-scale models and transfer them to the field of robotics. When queried endlessly, our system holds the potential for unleashing infinite amount of diversified demonstration data for robot learning, making a step towards fully automated large-scale robotic skill training for generalizable robotic systems.

2 RELATED WORK

Robotic skill learning in simulations Various physics-based simulation platforms have been developed in the past to accelerate robotics research (Liu & Negrut, 2021). These include rigid-body simulators (Coulmans & Bai, 2016; Todorov et al., 2012; Xiang et al., 2020), deformable object simulators (Macklin et al., 2014; Lin et al., 2020; Xu et al., 2023; Heiden et al., 2021), and environments supporting multi-material and their couplings with robots (Xian et al., 2023c; Gan et al., 2021; Gu et al., 2023). Such simulation platforms have been heavily employed in the robotics community for learning diverse skills, including table-top manipulation (Gervet et al., 2023; Tung et al., 2020; Xian et al., 2023b), deformable object manipulation (Lin et al., 2022a; Weng et al., 2022; Wang et al., 2023c; Lin et al., 2022b; Wang et al., 2022; Ha & Song, 2022), object cutting (Heiden et al., 2021; Xu et al., 2023), fluid manipulation (Seita et al., 2023; Xian et al., 2023c), as well as highly dynamic and complex skills such as in-hand re-orientation (Chen et al., 2022; Akkaya et al., 2019), object tossing (Zeng et al., 2020), acrobatic flight (Kaufmann et al., 2020; Loquercio et al., 2021; Song et al., 2023), locomotion for legged robots (Cheng et al., 2023; Zhuang et al., 2023; Radosavovic et al., 2023) and soft robots (Wang et al., 2023a).

Scaling up simulation environments Apart from building physics engines and simulators, a large body of prior work targeted at building large-scale simulation benchmarks, providing platforms for scalable skill learning and standardized benchmarking (Li et al., 2023a; Lin et al., 2020; Xian et al., 2023c; Yu et al., 2020; James et al., 2020; Mu et al., 2021; Gu et al., 2023; Srivastava et al., 2022). Notably, most of these prior simulation benchmarks or skill learning environments are manually built with human labeling. Another line of works attempts to scale up tasks and environments using procedural generation, and generate demonstrations with Task and Motion Planning (TAMP) (Dalal et al., 2023; McDonald & Hadfield-Menell, 2021; Murali et al., 2023; Migimatsu & Bohg, 2019; Toussaint, 2015). These methods primarily build on top of manually-defined rules and planning domains, limiting the diversity of the generated environments and skills to relatively simple pick-and-place and object stacking tasks (Dalal et al., 2023; McDonald & Hadfield-Menell, 2021). Contrary to these works, we extract the common sense knowledge embedded in foundation models such as LLMs, and use them for generating meaningful tasks, relevant scenes, and skill training supervisions, leading to more diverse and plausible skills.

Foundation and generative models for robotics Following the rapid advancement in foundation and generative models in domains of imagery (Poole et al., 2022; Liu et al., 2023c; Melas-Kyriazi

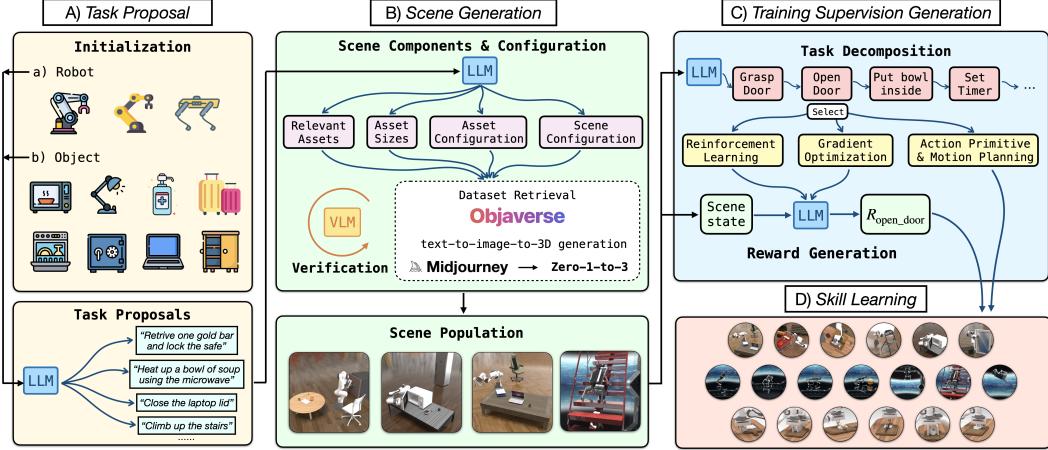


Figure 2: RoboGen consists of the following stages: A) task proposal, B) scene generation, C) training supervision generation, and D) skill learning with generated information.

et al., 2023), language (Touvron et al., 2023; Driess et al., 2023; OpenAI, 2023), and other modalities (Liu et al., 2023a; Girdhar et al., 2023; Huang et al., 2023a), an active line of works investigate how to use these large-scale models for robotics research, via approaches such as code generation (Wu et al., 2023; Liang et al., 2022), data augmentation (Yu et al., 2023a), visual imagination for skill execution (Du et al., 2023), sub-task planning (Ahn et al., 2022; Huang et al., 2022; Lin et al., 2023), concept generalization of learned skills (Brohan et al., 2023), outputting low-level control actions (Wang et al., 2023b), and goal specification (Kapelyukh et al., 2023). More related to ours are recent methods using LLMs for reward generation (Yu et al., 2023b), and sub-task and trajectory generation (Ha et al., 2023). In contrast to them, our proposed system aims at a fully automated pipeline that self-proposes new tasks, generate environments and yields diverse skills.

Generative Simulation We first presented the idea of *generative simulation* in a recent white paper (Xian et al., 2023a). We implemented a proof of concept of generative simulation in (Katara et al., 2023) by generating assets, task descriptions and decompositions for a Franka arm using image generative models and LLMs. In this paper, we extend this line of research to support a wider range of robot types and a more diverse set of tasks with photo-realistic visuals, powered by a more advanced rendering and physics engine.

3 ROBOGEN

RoboGen is an automated pipeline that utilizes the embedded common sense and generative capabilities of the latest foundation models (OpenAI, 2022; Taori et al., 2023) for automatic task, scene, and training supervision generation, leading to diverse robotic skill learning at scale. We illustrate the whole pipeline in Figure 2, composed of several integral stages: *Task Proposal*, *Scene Generation*, *Training Supervision Generation*, and *Skill Learning*. We detail each of them in the following.

3.1 TASK PROPOSAL

RoboGen starts with generating meaningful, diverse, and high-level tasks for robots to learn. Instead of directly querying an LLM for task proposal, we initialize the system with a specific robot type and an object randomly sampled from a pool. The provided robot and sampled object information are then used as input to an LLM to perform task proposal. Such a sampling procedure ensures the diversity of the generated tasks: for instance, a legged robot such as a quadruped is capable of acquiring a variety of locomotion skills, while a robotic arm manipulator has the potential to execute a diverse range of manipulation tasks when paired with different sampled objects. This initialization step serves as a seeding stage, providing a basis upon which the LLM can condition and subsequently reason and extrapolate to generate a variety of tasks, taking into account both robot and object affordances. Apart from object-based initialization, another choice is to employ example-based initialization, where we initialize the query with a provided robot and several example tasks sampled from a list 11 pre-defined tasks.

We use GPT-4 (OpenAI, 2023) as the LLM to query in our current pipeline, but this backend can be upgraded once better models are available. In the following, we explain details of RoboGen in the context of a robotic arm (e.g., Franka) and tasks generated pertain to object manipulation, using object-based initialization. In this case, the objects used for initialization are sampled from a predefined list including common articulated and non-articulated objects in household scenarios such as oven, microwave, dispenser, laptop, dishwasher, etc., extracted from PartNetMobility (Xiang et al., 2020) and RLBench (James et al., 2020). Having been trained on extensive internet-scale datasets, LLMs like GPT-4 are imbued with a rich understanding of the affordances of these objects, knowledge of how to interact with them, and what kind of meaningful tasks they can be associated with. In order to generate tasks that involve interaction and manipulation of sampled articulated objects by the robot, we construct a query comprising the broad category to which the articulated object belongs to, its articulation tree derived from the URDF file, and a semantics file providing annotations about the object’s links, e.g., which link corresponds to the door in a sampled microwave. This information is provided with the PartNetMobility dataset. The category of the articulated object informs what general types of tasks can be performed with this kind of objects, and the articulation tree and the semantics file informs GPT-4 how exactly the object can be articulated, and semantic meanings of each joint and link. The query asks GPT-4 to return a number of tasks that can be performed with the sampled object, where each task consists of the task name, a short description of the task, if there are any other objects that are required for the task besides the provided articulated object, and which joints/links of the articulated object the robot will need to interact with to accomplish the task. Additionally, we include an example input-output pair in the query for GPT-4 to perform in-context learning to improve the quality of its response.

As a concrete example, given a sampled articulated object being a microwave, where `joint_0` is a revolute joint connecting its door, and `joint_1` being another revolute joint controlling a knob of the timer, GPT-4 would return a task named as “heat up a bowl of soup”, with a task description of “The robot arm places a bowl of soup inside the microwave, closes the door and sets the microwave timer for an appropriate heating duration”, additional objects needed for the generated task such as “A bowl of soup”, and task-relevant joints and links including `joint_0` (for opening the microwave door), `joint_1` (for setting the timer), `link_0` (the door), and `link_1` (the timer knob). For the detailed prompt and example responses, please refer to Appendix B. Note that for cases where we sample non-articulated objects or use example-based initialization, the sampled objects and examples are provided only as a hint for task proposal, and the generated tasks will not be tied to them. For articulated objects, since PartNetMobility is the only articulated object dataset with high quality, and already covers diverse range of articulated assets, we will generate tasks dependent on the sampled asset. For locomotion and soft-body manipulation tasks, we use only example-based initialization, and resort to GPT-4 to populate additional required objects. By repeatedly querying with different sampled objects and examples, we can generate a diverse range of manipulation and locomotion tasks, concerning the relevant object affordances when needed.

3.2 SCENE GENERATION

Given a proposed task, we proceed to generate a corresponding simulation scene for learning skills to accomplish the task. As shown in Figure 2, the scene components and configurations are generated conditioned on the task description, and the object assets are retrieved or generated to populate the simulation scene subsequently. Specifically, the scene components and configurations are composed of the following elements: queries for relevant assets to be populated into the scene, their physical parameters such as sizes, configurations such as the initial joint angles, and the overall spatial configurations of the assets.

Obtaining queries for relevant assets Aside from the necessary object assets that are needed for the task generated in the previous step of task proposal, in order to increase the complexity and diversity of the generated scene while resembling object distributions of real-world scenarios, we query GPT-4 to return a number of additional queries for objects that are semantically relevant to the task. See Figure 1 for examples of the additional objects returned by GPT-4 for a task, e.g., for the task “Open storage, put the toy inside and close it”, the generated scene involves additionally a living room mat, a table-top lamp, a book, and an office chair.

Retrieving or generating assets The generated queries for relevant objects (i.e. their language descriptions) will be used to either search in an existing database (Deitke et al., 2023), or to be used as the input to text-to-image (Midjourney, 2022), and then image-to-3d mesh generation models

(Liu et al., 2023b; Poole et al., 2022) to generate the 3d textured mesh of the asset. Specifically, we use Objaverse (Deitke et al., 2023), a large-scale dataset containing over 800k object assets (3d meshes, texture pictures, and etc.) as the main database for the retrieval. For each object in Objaverse, we obtain a list of language descriptions of it by combining the default annotations and a more cleaned version of annotations from (Luo et al., 2023). Given the language description of the asset we want to retrieve, we use Sentence-Bert (Reimers & Gurevych, 2019) to get the embedding of the description, and retrieve k objects from Objaverse whose language embeddings are the most similar to the language embedding of the target asset. Due to noises in the object annotations, there can be significant discrepancies between the actual asset and the intended target, even when the similarity score in the language embedding space is high. To resolve this, we further leverage a vision-language model (VLM) to verify the retrieved assets and filter out the undesired ones. Specifically, we input an image of the retrieved object to the VLM model, and ask the VLM to caption it. The caption, together with the description of the desired asset and the description of the task, are fed back into GPT-4 to verify if the retrieved asset is appropriate to be used in the proposed task. Since the diverse range of assets from Objaverse is inherently challenging for existing pre-trained models, in order to improve the robustness of our system, we use both Bard (Google, 2022) and BLIP-2 (Li et al., 2023b), to cross verify the validity of the retrieved asset, and adopt the asset only when the captions from both of them are deemed suitable for the task by GPT-4. We use $k = 10$ in our pipeline, and if all assets are rejected, we resort to text-to-image followed by image-to-mesh generation models to generate the needed assets from the language description. We use Midjourney (Midjourney, 2022) as our text-to-image generative model, and Zero-1-to-3 (Liu et al., 2023b) as our image-to-mesh generative model. For soft-body manipulation tasks, in order to obtain a more consistent and controllable target shape for the soft-body under manipulation, we ask GPT-4 to come up with desired target shape, and only use this text-to-image-to-mesh pipeline instead of database retrieval.

Asset size Assets from Objaverse (Deitke et al., 2022) or PartNetMobility (Xiang et al., 2020) are usually not of physically plausible sizes. To account for this, we query GPT-4 to generate the sizes of the assets such that: 1) the sizes should match real-world object sizes; 2) the relative sizes between objects allow a plausible solution for solving the task, e.g., for the task of “putting a book into the drawer”, the size of the drawer should be larger than the book.

Initial asset configuration For certain tasks, the articulated object should be initialized with valid states for the robot to learn the skill. For example, for the task of “close the window”, the window should be initialized in an open state; similarly, for the task of opening the door, the door should be initially closed. Again, we query GPT-4 to set the initial configurations of these articulated objects, specified in joint angles. To allow GPT-4 to reason about the task and the articulated object, the query script contains the task description, the articulation tree and semantic descriptions of the articulated object.

Scene configuration Scene configuration specifying the location and relevant poses of each asset in the scene is crucial for both producing plausible environments and allowing valid skill learning. E.g., for the task of “retrieving a document from the safe”, the document needs to be initialized inside the safe; for the task of “removing the knife from the chopping board”, the knife needs to be initially placed on the chopping board. RoboGen queries GPT-4 to generate such special spatial relationships with the task description as the input, and also instructs GPT-4 to place objects in a collision-free manner.

With the generated scene components and configuration, we populate the scene accordingly. See Figure 1 for a collection of example scenes and tasks generated by RoboGen.

3.3 TRAINING SUPERVISION GENERATION

To acquire the skill for solving the proposed task, supervisions for skill learning are needed. To facilitate the learning process, RoboGen first queries GPT-4 to plan and decompose the generated task, which can be long-horizon, into shorter-horizon sub-tasks. Our key hypothesis is, when the task is decomposed into sufficiently short-horizon sub-tasks, each sub-task can be reliably solved by existing algorithms such as reinforcement learning (Schulman et al., 2017; Haarnoja et al., 2018), motion planning (Karaman & Frazzoli, 2011), or trajectory optimization (Kalakrishnan et al., 2011).

After the decomposition, RoboGen then queries GPT-4 to choose a proper algorithm for solving each sub-task. There are three different types of learning algorithms integrated into RoboGen: reinforcement learning, evolutionary strategy, gradient-based trajectory optimization, and action primitive with motion planning. Each of these is suited for different tasks, e.g., gradient-based trajectory optimization is more suitable for learning fine-grained manipulation tasks involving soft bodies such as shaping a dough into a target shape (Xu et al., 2023; Lin et al., 2022a); action primitives coupled with motion planning are more reliable in solving the task such as approaching a target object via a collision-free path; reinforcement learning and evolutionary strategy better suits tasks that are contact rich and involving continuous interaction with other scene components, e.g., legged locomotion, or when the required actions cannot be simply parameterized by discrete end-effector poses, e.g., turning the knob of an oven. We provide examples and let GPT-4 choose online which learning algorithm to use conditioned on the generated sub-task. We consider action primitives including grasping, approaching and releasing a target object. Since parallel jaw gripper can be limited when grasping objects with diverse sizes, we consider a robotic manipulator equipped with a suction cup to simplify object grasping. The grasping and approaching primitives are implemented as follows: we first randomly sample a point on the target object or link, compute a gripper pose that aligns with the normal of the sampled point, and then use motion planning to find a collision-free path to reach the target gripper pose. After the pose is reached, we keep moving along the normal direction until there contact is made with the target object.

For sub-tasks to be learned using RL or trajectory optimization, we prompt GPT-4 to write the corresponding reward functions with a few in-context examples. For object manipulation and locomotion tasks, the reward functions are based on the low-level simulation states which GPT-4 can query via a provided list of API calls. Further, we ask GPT-4 to suggest the action space for the learning algorithm, e.g., delta-translation of the end-effector, or the target location for the end-effector to move to. Delta-translation is more appropriate for tasks that involve local movements, e.g., opening a door after it has been grasped; for tasks that involve transferring an object to a different location, directly specifying the target as the action space makes the learning easier. For soft body manipulation tasks, the reward has a fixed form specified as the earth-mover distance between the current and target shape of the soft body.

3.4 SKILL LEARNING

Once we obtained all the required information for the proposed task, including scene components and configurations, task decompositions, and training supervisions for the decomposed sub-tasks, we are able to construct the scene in simulation for the robot to learn the required skills for completing the task.

As aforementioned, we use a combination of techniques for skill learning, including reinforcement learning, evolutionary strategy, gradient-based trajectory optimization, and action primitive with motion planning, as each are suitable for different types of tasks. For object manipulation tasks, we use SAC (Haarnoja et al., 2018) as the RL algorithm for learning the skill. The observation space is the low-level state of the objects and robot in the task. The action space of the RL policy includes the delta translation or target location (determined by GPT-4) of the robot end-effector, along with its delta rotation. We use BIT* (Gammell et al., 2015) implemented in the Open Motion Planning Library (OMPL) (Sucan et al., 2012) as the underlying motion planning algorithm for the action primitives. For long-horizon tasks that involve multiple sub-tasks, we adopt a simple scheme of learning each sub-task sequentially: for each sub-task, we run RL $N = 8$ times and use the end state with the highest reward as the initial state for the next sub-task. For locomotion tasks, the cross entropy method (CEM (De Boer et al., 2005)) is used for skill learning, which we find to be more stable and efficient than RL. The ground-truth simulator is used as the dynamics model in CEM, and the actions to be optimized are the joint angle values of the robot. For soft body manipulation tasks, we run gradient-based trajectory optimization with Adam (Kingma & Ba, 2014) for learning the skill, where the gradients are provided by the fully differentiable simulator we use. More details on skill learning can be found in Appendix A.

4 EXPERIMENTS

RoboGen is an automated pipeline that can be queried endlessly, and generate a continuous stream of skill demonstrations for diverse tasks. In our experiments, we aim to answer the following questions:

-
- **Task Diversity:** How diverse are the tasks proposed by RoboGen robotic skill learning?
 - **Scene Validity:** Does RoboGen generate valid simulation environments that match the proposed task descriptions?
 - **Training Supervision Validity:** Does RoboGen generate correct task decomposition and training supervisions for the task that will induce intended robot skills?
 - **Skill Learning:** Does integrating different learning algorithms in RoboGen improve the success rate of learning a skill?
 - **System:** Combining all the automated stages, can the whole system produce diverse and meaningful robotic skills?

4.1 EXPERIMENTAL SETUP

Our proposed system is generic and agnostic to specific simulation platforms. However, since we consider a wide range of task categories ranging from rigid dynamics to soft body simulation, and also consider skill learning methods such as gradient-based trajectory optimization which necessitates a differentiable simulation platform, we used Genesis for deploying RoboGen, a simulation platform for robot learning with diverse materials and fully differentiable¹. For skill learning, we use SAC (Haarnoja et al., 2018) as the RL algorithm. The policy and Q networks are both Multi-layer Perceptrons (MLP) of size [256, 256, 256], trained with a learning rate of $3e - 4$. For each sub-task, we train with 1M environment steps. We use BIT* (Gammell et al., 2015) as the motion planning algorithm, and Adam Kingma & Ba (2014) for gradient-based trajectory optimization for soft body manipulation tasks. More implementation details of these learning algorithms can be found in Appendix A.

4.2 EVALUATION METRICS AND BASELINES

We use the following metrics and baselines for evaluating our system:

Task Diversity The diversity of the generated tasks can be measured in many aspects, such as the semantic meanings of the tasks, scene configurations of the generated simulation environments, the appearances and geometries of the retrieved object assets, and the robot actions required to perform the task. For semantic meanings of the tasks, we perform quantitative evaluations by computing the Self-BLEU and the embedding similarity (Zhu et al., 2018) on the generated task descriptions, where lower scores indicate better diversity. We compare to established benchmarks, including RLBench (James et al., 2020), Maniskill2 (Gu et al., 2023), Meta-World (Yu et al., 2020), and Behavior-100 (Srivastava et al., 2022). For object assets and robot actions, we evaluate RoboGen qualitatively using the generated simulation environments and visualizations of learned robot skills.

Scene Validity To verify that the retrieved objects match the requirements of the task, we compute the BLIP-2 scores (Li et al., 2023b) between rendered images of the retrieved objects in the simulation scene, and the text descriptions of the objects. We compare with two ablations of our system. A) *No object verification*: we do not use a VLM to verify the retrieved object, and just retrieve objects based on text matching. B) *No size verification*: we do not use object sizes outputted by GPT-4; instead, we use the default size of the asset as provided in Objaverse or PartNetMobility.

Training Supervision Validity We perform qualitative evaluations on this by presenting images and videos of the learned skills using the generated decomposition and training supervisions.

Skill Learning Performance We compare to an ablation where we remove the options of using motion planning-based primitive and gradient-based trajectory optimization, and rely purely on reinforcement learning to learn the skills. Given a task, we run each method with 4 different seeds, and report the mean and std of the task return (we manually verified that the reward functions generated by GPT-4 are correct for the evaluated tasks).

System We perform qualitative evaluations of the whole system, by providing videos of over 100 learned skills on our websites. In the paper, we show snapshots of representative skills.

¹Genesis is still under development and not yet available to the public. We build our system on top of an internal version, made accessible to us thanks to the courtesy of the development team.

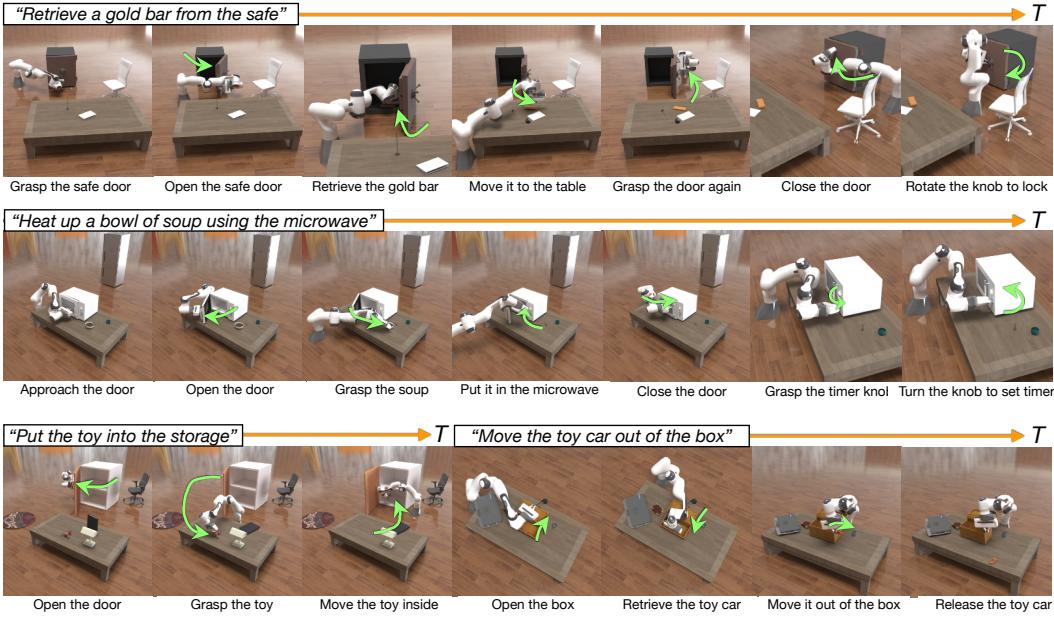


Figure 3: Snapshots of the learned skills on 4 example long-horizon tasks.

4.3 RESULTS

Task Diversity The quantitative evaluation results are presented in Table 1. We compare to a version of RoboGen where a total of 106 tasks are generated. As shown, RoboGen achieves the lowest Self-BLEU and embedding similarity compared to all previous benchmarks, indicating that the diversity of the generated tasks is higher than prior manually established benchmarks under both metrics. This demonstrates that RoboGen can generate a set of tasks whose diversity matches or surpasses prior manually crafted skill learning benchmarks and datasets.

	RoboGen	Behavior-100	RLbench	MetaWorld	Maniskill2
Number of Tasks	106	100	106	50	20
Self-BLEU ↓	0.284	0.299	0.317	0.322	0.674
Embedding Similarity ↓	0.165	0.210	0.200	0.263	0.194

Table 1: **Task diversity comparison.** Note that RoboGen can generate an unbounded number of tasks when queried repeatedly. Here we compare to a version that has a total number of 106 tasks generated, such that number of tasks is similar to those in prior work.

Scene Validity Figure 4 shows the BLIP-2 score of all compared methods on 7 example generated tasks. As shown, removing the size verification leads to drastic decrease in BLIP-2 score. This is expected as the default asset sizes from Objaverse and PartNetMobility can be drastically different from plausible real-world sizes. The ablation “No object verification” also has lower BLIP-2 score, and with a larger variances, indicating our verification step improves object selection validity. The results demonstrate the importance of using both object and size verification in RoboGen.

Training Supervision Validity Figure 3 demonstrates the skills learned with the generated training supervisions from RoboGen, i.e., the task decompositions and reward functions, on 4 example *long-horizon* tasks. As shown, the robot successfully learns skills to complete the corresponding tasks, suggesting that the automatically generated training supervisions are effective in deriving meaningful and useful skills.

Skill Learning We evaluate on 4 tasks that involve interactions with articulated objects. The results are presented in Table 2. We find that allowing learning algorithm selection is beneficial for achieving higher performance for completing the tasks. When only RL is used, the skill learning completely fails for most tasks.

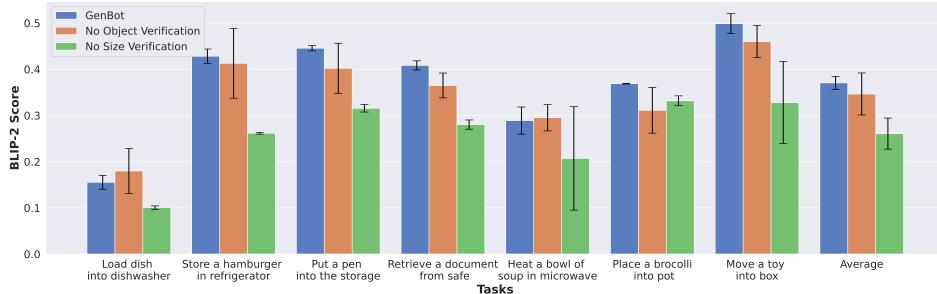


Figure 4: We compare the BLIP-2 score of ablations of RoboGen on 7 tasks. Both ablations achieve lower scores, showing the importance of both object and size verification.

	Slide Window halfway	Open door of washing machine	Open Trashcan lid	Flush the toilet
RoboGen	-31.59 ± 15.05	-484.80 ± 200.15	-551.37 ± 128.10	-12.26 ± 2.15
RoboGen - RL only	-147.75 ± 1.09	-926.68 ± 1.44	-797.46 ± 0.05	-294.83 ± 1.06

Table 2: The return decreases drastically if only RL is used for skill learning.

System Figure 1 and 3 visualize some of the generated tasks and learned skills from RoboGen. As shown in Figure 1, RoboGen can generate diverse tasks for skill learning spanning from rigid/articulated object manipulation, locomotion, and soft body manipulation. Figure 3 further shows that RoboGen is able to deliver long-horizon manipulation skills with reasonable decompositions.

For extensive qualitative results of proposed tasks and learned skills, please refer to our project site.

5 CONCLUSION & LIMITATIONS

We introduced *RoboGen*, a generative agent that automatically proposes and learns diverse robotic skills at scale via generative simulation. RoboGen utilizes the latest advancements in foundation models to automatically generate diverse tasks, scenes, and training supervisions in simulation, making a foundational step towards scalable robotic skill learning in simulation, while requiring minimal human supervision once deployed. Our system is a fully generative pipeline that can be queried endlessly, producing a large number of skill demonstrations associated with diverse tasks and environments. RoboGen is agnostic to the backend foundation models, and can be upgraded continuously using latest models when available.

Our current system still has several limitations: 1) Large-scale verification of learned skills (i.e., whether the resultant skill really solves the corresponding task by text descriptions) is still a challenge in the current pipeline. This could potentially be addressed by using better multi-modal foundation models in the future. Contemporary works (Ma et al., 2023) have also explored using environment feedback for iterative refinement on the generated supervisions (reward functions), which we hope to integrate into our paradigm in the future. 2) Our paradigm is intrinsically constrained by a sim-to-real gap when it comes to real-world deployment. However, with the latest and rapid advancement in physically accurate simulation, as well as techniques such as domain randomization and realistic sensory signal rendering, we expect the sim-to-real gap to be narrowed further in the coming future. 3) Our system assumes that with a right reward function, existing policy learning algorithms are sufficient for learning the proposed skills. For the policy learning algorithms we tested in this paper (RL with SAC and an action space of delta end-effector pose, and gradient-based trajectory optimization), we observe they are still not robust enough and multiple runs are often required to produce successful skill demos for certain generated tasks. We leave integration of more powerful policy learning algorithms into RoboGen, e.g., those with better action parameterizations (Zeng et al., 2021; Seita et al., 2023), as future work.

REFERENCES

- Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.
- Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik’s cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.
- Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. *arXiv preprint arXiv:2307.15818*, 2023.
- Tao Chen, Jie Xu, and Pulkit Agrawal. A system for general in-hand object re-orientation. In *Conference on Robot Learning*, pp. 297–307. PMLR, 2022.
- Xuxin Cheng, Ashish Kumar, and Deepak Pathak. Legs as manipulator: Pushing quadrupedal agility beyond locomotion. *arXiv preprint arXiv:2303.11330*, 2023.
- Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. <http://pybullet.org>, 2016.
- Murtaza Dalal, Ajay Mandlekar, Caelan Garrett, Ankur Handa, Ruslan Salakhutdinov, and Dieter Fox. Imitating task and motion planning with visuomotor transformers, 2023.
- Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134:19–67, 2005.
- Matt Deitke, Dustin Schwenk, Jordi Salvador, Luca Weihs, Oscar Michel, Eli VanderBilt, Ludwig Schmidt, Kiana Ehsani, Aniruddha Kembhavi, and Ali Farhadi. Objaverse: A universe of annotated 3d objects. *arXiv preprint arXiv:2212.08051*, 2022.
- Matt Deitke, Dustin Schwenk, Jordi Salvador, Luca Weihs, Oscar Michel, Eli VanderBilt, Ludwig Schmidt, Kiana Ehsani, Aniruddha Kembhavi, and Ali Farhadi. Objaverse: A universe of annotated 3d objects. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 13142–13153, 2023.
- Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: An embodied multi-modal language model. *arXiv preprint arXiv:2303.03378*, 2023.
- Yilun Du, Mengjiao Yang, Bo Dai, Hanjun Dai, Ofir Nachum, Josh Tenenbaum, Dale Schuurmans, and Pieter Abbeel. Learning universal policies via text-guided video generation. *arXiv preprint arXiv:2302.00111*, 2023.
- Jonathan D Gammell, Siddhartha S Srinivasa, and Timothy D Barfoot. Batch informed trees (bit*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs. In *2015 IEEE international conference on robotics and automation (ICRA)*, pp. 3067–3074. IEEE, 2015.
- Chuang Gan, Jeremy Schwartz, Seth Alter, Damian Mrowca, Martin Schrimpf, James Traer, Julian De Freitas, Jonas Kubilius, Abhishek Bhandwaldar, Nick Haber, Megumi Sano, Kuno Kim, Elias Wang, Michael Lingelbach, Aidan Curtis, Kevin Feigelis, Daniel M. Bear, Dan Gutfreund, David Cox, Antonio Torralba, James J. DiCarlo, Joshua B. Tenenbaum, Josh H. McDermott, and Daniel L. K. Yamins. Threedworld: A platform for interactive multi-modal physical simulation, 2021.
- Theophile Gervet, Zhou Xian, Nikolaos Gkanatsios, and Katerina Fragkiadaki. Act3d: Infinite resolution action detection transformer for robotic manipulation. *arXiv preprint arXiv:2306.17817*, 2023.
- Rohit Girdhar, Alaaeldin El-Nouby, Zhuang Liu, Mannat Singh, Kalyan Vasudev Alwala, Armand Joulin, and Ishan Misra. Imagebind: One embedding space to bind them all. *arXiv preprint arXiv:2305.05665*, 2023.

-
- Google. Bard. <https://bard.google.com/>, 2022.
- Jiayuan Gu, Fanbo Xiang, Xuanlin Li, Zhan Ling, Xiqiang Liu, Tongzhou Mu, Yihe Tang, Stone Tao, Xinyue Wei, Yunchao Yao, et al. Maniskill2: A unified benchmark for generalizable manipulation skills. *arXiv preprint arXiv:2302.04659*, 2023.
- Huy Ha and Shuran Song. Flingbot: The unreasonable effectiveness of dynamic manipulation for cloth unfolding. In *Conference on Robot Learning*, pp. 24–33. PMLR, 2022.
- Huy Ha, Pete Florence, and Shuran Song. Scaling up and distilling down: Language-guided robot skill acquisition. *arXiv preprint arXiv:2307.14535*, 2023.
- Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pp. 1861–1870. PMLR, 2018.
- Tuomas Haarnoja, Ben Moran, Guy Lever, Sandy H Huang, Dhruva Tirumala, Markus Wulfmeier, Jan Humplik, Saran Tunyasuvunakool, Noah Y Siegel, Roland Hafner, et al. Learning agile soccer skills for a bipedal robot with deep reinforcement learning. *arXiv preprint arXiv:2304.13653*, 2023.
- Eric Heiden, Miles Macklin, Yashraj Narang, Dieter Fox, Animesh Garg, and Fabio Ramos. Disect: A differentiable simulation engine for autonomous robotic cutting. *arXiv preprint arXiv:2105.12244*, 2021.
- Rongjie Huang, Jiawei Huang, Dongchao Yang, Yi Ren, Luping Liu, Mingze Li, Zhenhui Ye, Jinglin Liu, Xiang Yin, and Zhou Zhao. Make-an-audio: Text-to-audio generation with prompt-enhanced diffusion models. *arXiv preprint arXiv:2301.12661*, 2023a.
- Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022.
- Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. Voxposer: Composable 3d value maps for robotic manipulation with language models. *arXiv preprint arXiv:2307.05973*, 2023b.
- Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J Davison. Rlbench: The robot learning benchmark & learning environment. *IEEE Robotics and Automation Letters*, 5(2):3019–3026, 2020.
- Mrinal Kalakrishnan, Sachin Chitta, Evangelos Theodorou, Peter Pastor, and Stefan Schaal. Stomp: Stochastic trajectory optimization for motion planning. In *2011 IEEE international conference on robotics and automation*, pp. 4569–4574. IEEE, 2011.
- Minguk Kang, Jun-Yan Zhu, Richard Zhang, Jaesik Park, Eli Shechtman, Sylvain Paris, and Taesung Park. Scaling up gans for text-to-image synthesis. *arXiv preprint arXiv:2303.05511*, 2023.
- Ivan Kapelyukh, Vitalis Vosylius, and Edward Johns. Dall-e-bot: Introducing web-scale diffusion models to robotics. *IEEE Robotics and Automation Letters*, 2023.
- Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.
- Pushkal Katara, Zhou Xian, and Katerina Fragkiadaki. Gen2sim: Scaling up robot learning in simulation with generative models. *arXiv preprint arXiv:2310.18308*, 2023.
- Elia Kaufmann, Antonio Loquercio, René Ranftl, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. Deep drone acrobatics. *arXiv preprint arXiv:2006.05768*, 2020.
- Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

-
- Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabrael Levine, Michael Lingelbach, Jiankai Sun, et al. Behavior-1k: A benchmark for embodied ai with 1,000 everyday activities and realistic simulation. In *Conference on Robot Learning*, pp. 80–93. PMLR, 2023a.
- Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. *arXiv preprint arXiv:2301.12597*, 2023b.
- Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*, 2022.
- Kevin Lin, Christopher Agia, Toki Migimatsu, Marco Pavone, and Jeannette Bohg. Text2motion: From natural language instructions to feasible plans. *arXiv preprint arXiv:2303.12153*, 2023.
- Xingyu Lin, Yufei Wang, Jake Olkin, and David Held. Softgym: Benchmarking deep reinforcement learning for deformable object manipulation. *arXiv preprint arXiv:2011.07215*, 2020.
- Xingyu Lin, Zhiao Huang, Yunzhu Li, Joshua B Tenenbaum, David Held, and Chuang Gan. Diff-skill: Skill abstraction from differentiable physics for deformable object manipulations with tools. *arXiv preprint arXiv:2203.17275*, 2022a.
- Xingyu Lin, Yufei Wang, Zixuan Huang, and David Held. Learning visible connectivity dynamics for cloth smoothing. In *Conference on Robot Learning*, pp. 256–266. PMLR, 2022b.
- C Karen Liu and Dan Negrut. The role of physics-based simulators in robotics. *Annual Review of Control, Robotics, and Autonomous Systems*, 4:35–58, 2021.
- Haohe Liu, Zehua Chen, Yi Yuan, Xinhao Mei, Xubo Liu, Danilo Mandic, Wenwu Wang, and Mark D Plumbley. Audioldm: Text-to-audio generation with latent diffusion models. *arXiv preprint arXiv:2301.12503*, 2023a.
- Ruoshi Liu, Rundi Wu, Basile Van Hoorick, Pavel Tokmakov, Sergey Zakharov, and Carl Vondrick. Zero-1-to-3: Zero-shot one image to 3d object. *arXiv preprint arXiv:2303.11328*, 2023b.
- Zhen Liu, Yao Feng, Michael J Black, Derek Nowrouzezahrai, Liam Paull, and Weiyang Liu. Meshdiffusion: Score-based generative 3d mesh modeling. *arXiv preprint arXiv:2303.08133*, 2023c.
- Antonio Loquercio, Elia Kaufmann, René Ranftl, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. Learning high-speed flight in the wild. *Science Robotics*, 6(59):eabg5810, 2021.
- Tiange Luo, Chris Rockwell, Honglak Lee, and Justin Johnson. Scalable 3d captioning with pre-trained models. *arXiv preprint arXiv:2306.07279*, 2023.
- Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.
- Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Unified particle physics for real-time applications. *ACM Transactions on Graphics (TOG)*, 33(4):1–12, 2014.
- Michael James McDonald and Dylan Hadfield-Menell. Guided imitation of task and motion planning, 2021.
- Luke Melas-Kyriazi, Christian Rupprecht, Iro Laina, and Andrea Vedaldi. Realfusion: 360 $\{\backslash\deg\}$ reconstruction of any object from a single image. *arXiv preprint arXiv:2302.10663*, 2023.
- Midjourney. Midjourney. <https://www.midjourney.com/>, 2022.
- Toki Migimatsu and Jeannette Bohg. Object-centric task and motion planning in dynamic environments. *CoRR*, abs/1911.04679, 2019. URL <http://arxiv.org/abs/1911.04679>.

-
- Tongzhou Mu, Zhan Ling, Fanbo Xiang, Derek Yang, Xuanlin Li, Stone Tao, Zhiao Huang, Zhiwei Jia, and Hao Su. Maniskill: Generalizable manipulation skill benchmark with large-scale demonstrations. *arXiv preprint arXiv:2107.14483*, 2021.
- Adithyavairavan Murali, Arsalan Mousavian, Clemens Eppner, Adam Fishman, and Dieter Fox. Cabinet: Scaling neural collision detection for object rearrangement with procedural scene generation, 2023.
- OpenAI. Chatgpt. <https://openai.com/blog/chatgpt>, 2022.
- OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
- Ben Poole, Ajay Jain, Jonathan T Barron, and Ben Mildenhall. Dreamfusion: Text-to-3d using 2d diffusion. *arXiv preprint arXiv:2209.14988*, 2022.
- Ilija Radosavovic, Tete Xiao, Bike Zhang, Trevor Darrell, Jitendra Malik, and Koushil Sreenath. Learning humanoid locomotion with transformers. *arXiv preprint arXiv:2303.03381*, 2023.
- Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.
- Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10684–10695, 2022.
- John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- Daniel Seita, Yufei Wang, Sarthak J Shetty, Edward Yao Li, Zackory Erickson, and David Held. Toolflownet: Robotic manipulation with tools via predicting tool flow from point clouds. In *Conference on Robot Learning*, pp. 1038–1049. PMLR, 2023.
- Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *arXiv preprint arXiv:2303.17580*, 2023.
- Yunlong Song, Angel Romero, Matthias Müller, Vladlen Koltun, and Davide Scaramuzza. Reaching the limit in autonomous racing: Optimal control versus reinforcement learning. *Science Robotics*, 8(82):eadg1462, 2023.
- Sanjana Srivastava, Chengshu Li, Michael Lingelbach, Roberto Martín-Martín, Fei Xia, Kent Elliott Vainio, Zheng Lian, Cem Gokmen, Shyamal Buch, Karen Liu, et al. Behavior: Benchmark for everyday household activities in virtual, interactive, and ecological environments. In *Conference on Robot Learning*, pp. 477–490. PMLR, 2022.
- Ioan A Sucan, Mark Moll, and Lydia E Kavraki. The open motion planning library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, 2012.
- Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2303.08128*, 2023.
- Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.
- Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pp. 5026–5033. IEEE, 2012.
- Marc Toussaint. Logic-geometric programming: An optimization-based approach to combined task and motion planning. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI’15, pp. 1930–1936. AAAI Press, 2015. ISBN 9781577357384.

-
- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.
- Hsiao-Yu Fish Tung, Zhou Xian, Mihir Prabhudesai, Shamit Lal, and Katerina Fragkiadaki. 3d-oes: Viewpoint-invariant object-factorized environment simulators. *arXiv preprint arXiv:2011.06464*, 2020.
- Tsun-Hsuan Wang, Pingchuan Ma, Andrew Everett Spielberg, Zhou Xian, Hao Zhang, Joshua B Tenenbaum, Daniela Rus, and Chuang Gan. Softzoo: A soft robot co-design benchmark for locomotion in diverse environments. *arXiv preprint arXiv:2303.09555*, 2023a.
- Yen-Jen Wang, Bike Zhang, Jianyu Chen, and Koushil Sreenath. Prompt a robot to walk with large language models. *arXiv preprint arXiv:2309.09969*, 2023b.
- Yufei Wang, David Held, and Zackory Erickson. Visual haptic reasoning: Estimating contact forces by observing deformable object interactions. *IEEE Robotics and Automation Letters*, 7(4):11426–11433, 2022.
- Yufei Wang, Zhanyi Sun, Zackory Erickson, and David Held. One policy to dress them all: Learning to dress people with diverse poses and garments. *arXiv preprint arXiv:2306.12372*, 2023c.
- Thomas Weng, Sujay Man Bajracharya, Yufei Wang, Khush Agrawal, and David Held. Fabricflownet: Bimanual cloth manipulation with a flow-based policy. In *Conference on Robot Learning*, pp. 192–202. PMLR, 2022.
- Jimmy Wu, Rika Antonova, Adam Kan, Marion Lepert, Andy Zeng, Shuran Song, Jeannette Bohg, Szymon Rusinkiewicz, and Thomas Funkhouser. Tidybot: Personalized robot assistance with large language models. *arXiv preprint arXiv:2305.05658*, 2023.
- Zhou Xian, Theophile Gervet, Zhenjia Xu, Yi-Ling Qiao, and Tsun-Hsuan Wang. Towards a foundation model for generalist robots: Diverse skill learning at scale via automated task and scene generation. *arXiv preprint arXiv:2305.10455*, 2023a.
- Zhou Xian, Nikolaos Gkanatsios, Theophile Gervet, and Katerina Fragkiadaki. Unifying diffusion models with action detection transformers for multi-task robotic manipulation. In *7th Annual Conference on Robot Learning*, 2023b.
- Zhou Xian, Bo Zhu, Zhenjia Xu, Hsiao-Yu Tung, Antonio Torralba, Katerina Fragkiadaki, and Chuang Gan. Fluidlab: A differentiable environment for benchmarking complex fluid manipulation. *arXiv preprint arXiv:2303.02346*, 2023c.
- Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, et al. Sapien: A simulated part-based interactive environment. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11097–11107, 2020.
- Zhenjia Xu, Zhou Xian, Xingyu Lin, Cheng Chi, Zhiao Huang, Chuang Gan, and Shuran Song. Roboninja: Learning an adaptive cutting policy for multi-material objects. *arXiv preprint arXiv:2302.11553*, 2023.
- Zhengyuan Yang, Linjie Li, Jianfeng Wang, Kevin Lin, Ehsan Azarnasab, Faisal Ahmed, Zicheng Liu, Ce Liu, Michael Zeng, and Lijuan Wang. Mm-react: Prompting chatgpt for multimodal reasoning and action. *arXiv preprint arXiv:2303.11381*, 2023.
- Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on robot learning*, pp. 1094–1100. PMLR, 2020.
- Tianhe Yu, Ted Xiao, Austin Stone, Jonathan Tompson, Anthony Brohan, Su Wang, Jaspiar Singh, Clayton Tan, Jodilyn Peralta, Brian Ichter, et al. Scaling robot learning with semantically imagined experience. *arXiv preprint arXiv:2302.11550*, 2023a.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, et al. Language to rewards for robotic skill synthesis. *arXiv preprint arXiv:2306.08647*, 2023b.

Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. Tossingbot: Learning to throw arbitrary objects with residual physics. *IEEE Transactions on Robotics*, 36(4): 1307–1319, 2020.

Andy Zeng, Pete Florence, Jonathan Tompson, Stefan Welker, Jonathan Chien, Maria Attarian, Travis Armstrong, Ivan Krasin, Dan Duong, Vikas Sindhwani, et al. Transporter networks: Re-arranging the visual world for robotic manipulation. In *Conference on Robot Learning*, pp. 726–747. PMLR, 2021.

Yaoming Zhu, Sidi Lu, Lei Zheng, Jiaxian Guo, Weinan Zhang, Jun Wang, and Yong Yu. Texxygen: A benchmarking platform for text generation models. In *The 41st international ACM SIGIR conference on research & development in information retrieval*, pp. 1097–1100, 2018.

Ziwen Zhuang, Zipeng Fu, Jianren Wang, Christopher Atkeson, Soeren Schwertfeger, Chelsea Finn, and Hang Zhao. Robot parkour learning. *arXiv preprint arXiv:2309.05665*, 2023.

A IMPLEMENTATION DETAILS

For reinforcement learning, we use SAC (Haarnoja et al., 2018) as the RL algorithm. For both rigid object manipulation and locomotion tasks, the observation space is the low-level state of the objects and robot in the task. The policy and Q networks used in SAC are both Multi-layer Perceptrons (MLP) of size [256, 256, 256]. We use a learning rate of $3e - 4$ for the actor, the critic, and the entropy regularizer. The horizon of all manipulation tasks are 100, with a frameskip of 2, and the horizon for all locomotion tasks are 150, with a frameskip of 4. The action of the RL policy is 6d: where the first 3 elements determines the translation, either as delta translation or target location (suggested by GPT-4), and the second 3 elements determines the delta rotation, expressed as delta-axis angle in the gripper’s local frame. For each sub-task, we train with 1M environment steps.

For action primitives, we use BIT* (Gammell et al., 2015) implemented in the Open Motion Planning Library (OMPL) Sucan et al. (2012) as the motion planning algorithm. For the grasping and the approaching primitive, we first sample a surface point on the target object or link, then compute a gripper pose that aligns the gripper y axis with the normal of the sampled point. The pre-contact gripper pose is set to be 0.03m above the surface point along the normal direction. Motion planning is then used to find a collision-free path to reach the target gripper pose. After the target gripper pose is reached, we keep moving the gripper along the normal until contact is made.

For soft body manipulation tasks, we use Adam Kingma & Ba (2014) for gradient-based trajectory optimization. We run trajectory optimization for 300 gradient steps. We use a learning rate of 0.05 for the optimizer. The horizons of all manipulation tasks are either 150 or 200. We use Earth Mover’s distance between object’s current and target shape as the cost function for trajectory optimization.

For querying GPT-4, we used a temperature between $0.8 - 1.0$ for task proposal to ensure diversity in the generated tasks. For all other stages of RoboGen, we use temperature values between $0 - 0.3$ to ensure more robust responses from GPT-4.

B PROMPTS

Pre-defined tasks for example-based initialization of RoboGen.

```
"""
Task: stack two blocks, with the larger one at the bottom.
Object: A small block, and a large block.
""",

"""
Task: Put the broccoli on the grill pan
Objects: a broccoli, a grill pan
""",

"""
Task: Put 1 mug on the cup holder
Objects: A mug, a mug tree holder
""",

"""
Task: Pick up the hanger and place it on the clothing rack
Objects: a cloth hanger, a clothing rack
""",

"""
Task: Put 1 book into the bookshelf
Objects: a book, a bookshelf
""",

"""
Task: Put the knife on the chopping board
Objects: a kitchen knife, a board
""",

"""
Task: Put a old toy in bin
Objects: A old toy, a rubbish bin
""",

"""
Task: Place the dishes and cutlery on the table in preparation for a meal
Objects: a dish plate, a fork, a spoon, a steak knife
""",

"""
Task: Stack one cup on top of the other
```

```

Objects: Two same cups
"""

"""
Task: Remove the green pepper from the weighing scales and place it on the floor
Objects: A green pepper, a weighing scale
"""

"""
Task: Put the apple on the weighing scale to weigh it
Objects: An apple, a weighing scale
"""

```

In the following, we show all prompts used for generating an articulated object manipulation task using RoboGen.

Task Proposal prompt. We show an example where the sampled object is a trashcan for object manipulation task generation.

```
I will give you an articulated object, with its articulation tree and semantics. Your goal is to imagine some tasks that a robotic arm can perform with this articulated object in household scenarios. You can think of the robotic arm as a Franka Panda robot. The task will be built in a simulator for the robot to learn it.
```

```
Focus on manipulation or interaction with the object itself. Sometimes the object will have functions, e.g., a microwave can be used to heat food, in these cases, feel free to include other objects that are needed for the task.
```

```
Please do not think of tasks that try to assemble or disassemble the object. Do not think of tasks that aim to clean the object or check its functionality.
```

For each task you imagined, please write in the following format:

Task name: the name of the task.

Description: some basic descriptions of the tasks.

Additional Objects: Additional objects other than the provided articulated object required for completing the task.

Links: Links of the articulated objects that are required to perform the task.

- Link 1: reasons why this link is needed for the task
- Link 2: reasons why this link is needed for the task
- ...

Joints: Joints of the articulated objects that are required to perform the task.

- Joint 1: reasons why this joint is needed for the task
- Joint 2: reasons why this joint is needed for the task
- ...

Example Input:

```
'''Oven articulation tree
links:
base
link_0
link_1
link_2
link_3
link_4
link_5
link_6
link_7

joints:
joint_name: joint_0 joint_type: revolute parent_link: link_7 child_link: link_0
joint_name: joint_1 joint_type: continuous parent_link: link_7 child_link: link_1
joint_name: joint_2 joint_type: continuous parent_link: link_7 child_link: link_2
joint_name: joint_3 joint_type: continuous parent_link: link_7 child_link: link_3
joint_name: joint_4 joint_type: continuous parent_link: link_7 child_link: link_4
joint_name: joint_5 joint_type: continuous parent_link: link_7 child_link: link_5
joint_name: joint_6 joint_type: continuous parent_link: link_7 child_link: link_6
joint_name: joint_7 joint_type: fixed parent_link: base child_link: link_7
'''
```

```
'''Oven semantics
link_0 hinge door
link_1 hinge knob
link_2 hinge knob
link_3 hinge knob
link_4 hinge knob
link_5 hinge knob
link_6 hinge knob
link_7 heavy oven_body
'''
```

Example output:

```
Task Name: Open Oven Door
Description: The robotic arm will open the oven door.
Additional Objects: None
Links:
- link_0: from the semantics, this is the door of the oven. The robot needs to approach this door in order to open it.
Joints:
```

```
- joint_0: from the articulation tree, this is the revolute joint that connects link_0. Therefore, the robot needs to actuate this joint for opening the door.
```

```
Task Name: Adjust Oven Temperature
Description: The robotic arm will turn one of the oven's hinge knobs to set a desired temperature.
Additional Objects: None
Links:
- link_1: the robot needs to approach link_1, which is assumed to be the temperature knob, to rotate it to set the temperature.
Joints:
- joint_1: joint_1 connects link_1 from the articulation tree. The robot needs to actuate it to rotate link_1 to the desired temperature.
```

```
Task Name: Heat a hamburger Inside Oven
Description: The robot arm places a hamburger inside the oven, and sets the oven temperature to be appropriate for heating the hamburger.
Additional Objects: hamburger
Links:
- link_0: link_0 is the oven door from the semantics. The robot needs to open the door in order to put the hamburger inside the oven.
link_1: the robot needs to approach link_1, which is the temperature knob, to rotate it to set the desired temperature.
Joints:
- joint_0: from the articulation tree, this is the revolute joint that connects link_0 (the door). Therefore, the robot needs to actuate this joint for opening the door.
- joint_1: from the articulation tree, joint_1 connects link_1, which is the temperature knob. The robot needs to actuate it to rotate link_1 to the desired temperature.
```

```
Task Name: Set Oven Timer
Description: The robot arm turns a timer knob to set cooking time for the food.
Additional Objects: None.
Links:
- link_2: link_2 is assumed to be the knob for controlling the cooking time. The robot needs to approach link_2 to set the cooking time.
Joints:
- joint_2: from the articulation tree, joint_2 connects link_2. The robot needs to actuate joint_2 to rotate link_2 to the desired position, setting the oven timer.
```

Can you do the same for the following object:

```
'''TrashCan articulation tree
links:
base
link_0
link_1

joints:
joint_name: joint_0 joint_type: revolute parent_link: link_1 child_link: link_0
joint_name: joint_1 joint_type: fixed parent_link: base child_link: link_1

```
'''TrashCan semantics
link_0 hinge door
link_1 free trashcan_body
```
'''
```

An example response from GPT-4:

```
Task Name: Open Trash Can
Description: The robotic arm will open the trash can lid.
Additional Objects: None
Links:
- link_0: This link represents the lid of the trash can, based on the semantics. The robot needs to approach this lid to open it.
Joints:
- joint_0: The joint_0 in the articulation tree connects link_0 (the lid). Thus, the robot needs to actuate this joint to open the lid.
```

```
Task Name: Throw Trash Away
Description: The robotic arm places an item of trash inside the trash can.
Additional Objects: A pile of trash
Links:
- link_0: Link_0, as the trash can lid, needs to be approached and opened by the robot in order to throw away the trash.
Joints:
- joint_0: Joint_0, connecting to link_0, needs to be actuated for the robot to open the trash can lid.
```

Scene Generation Prompt. Following the above example, we show the scene generation prompt for the task “Throw Trash Away”.

I need you to describe the initial scene configuration for a given task in the following format, using a yaml file. This yaml file will help build the task in a simulator. The task is for a mobile Franka panda robotic arm to learn a manipulation skill in the simulator. The Franka panda arm is mounted on a floor, at location (1, 1, 0). It can move freely on the floor. The z axis is the gravity axis.

The format is as follows:

```
'''yaml
- use_table: whether the task requires using a table. This should be decided based on common sense. If a table is used, its location will be fixed at (0, 0, 0). The height of the table will be 0.6m. Usually, if the objects involved in the task are usually placed on a table (not directly on the ground), then the task requires using a table.
# for each object involved in the task, we need to specify the following fields for it.
- type: mesh
  name: name of the object, so it can be referred to in the simulator
  size: describe the scale of the object mesh using 1 number in meters. The scale should match real everyday objects. E.g., an apple is of scale 0.08m. You can think of the scale to be the longest dimension of the object.
  lang: this should be a language description of the mesh. The language should be a concise description of the object, such that the language description can be used to search an existing database of objects to find the object.
  path: this can be a string showing the path to the mesh of the object.
  on_table: whether the object needs to be placed on the table (if there is a table needed for the task). This should be based on common sense and the requirement of the task. E.g., a microwave is usually placed on the table.
  center: the location of the object center. If there isn't a table needed for the task or the object does not need to be on the table, this center should be expressed in the world coordinate system. If there is a table in the task and the object needs to be placed on the table, this center should be expressed in terms of the table coordinate, where (0, 0, 0) is the lower corner of the table, and (1, 1, 1) is the higher corner of the table. In either case, you should try to specify a location such that there is no collision between objects.
...
```

```

An example input includes the task names, task descriptions, and objects involved in the task. I will also provide with you the articulation tree and semantics of the articulated object. This can be useful for knowing what parts are already in the articulated object, and thus you do not need to repeat those parts as separate objects in the yaml file.

Your task includes two parts:

1. Output the yaml configuration of the task.
2. Sometimes, the task description / objects involved will refer to generic/placeholder objects, e.g., to place an "item" into the drawer, and to heat "food" in the microwave. In the generated yaml config, you should change these placeholder objects to be concrete objects in the lang field, e.g., change "item" to be a toy or a pencil, and "food" to be a hamburger, a bowl of soup, etc.

Example input:

```
Task Name: Insert Bread Slice
Description: The robotic arm will insert a bread slice into the toaster.
Objects involved: Toaster, bread slice. Only the objects specified here should be included in the yaml file.
```

```
'''Toaster articulation tree
links:
base
link_0
link_1
link_2
link_3
link_4
link_5

joints:
joint_name: joint_0 joint_type: continuous parent_link: link_5 child_link: link_0
joint_name: joint_1 joint_type: prismatic parent_link: link_5 child_link: link_1
joint_name: joint_2 joint_type: prismatic parent_link: link_5 child_link: link_2
joint_name: joint_3 joint_type: prismatic parent_link: link_5 child_link: link_3
joint_name: joint_4 joint_type: prismatic parent_link: link_5 child_link: link_4
joint_name: joint_5 joint_type: fixed parent_link: base child_link: link_5
```

```

```
'''Toaster semantics
link_0 hinge knob
link_1 slider slider
link_2 slider button
link_3 slider button
link_4 slider button
link_5 free toaster_body
```

```

An example output:

```
'''yaml
- use_table: True ### Toaster and bread are usually put on a table.
- type: mesh
 name: "Toaster"
 on_table: True # Toasters are usually put on a table.
 center: (0.1, 0.1, 0) # Remember that when an object is placed on the table, the center is expressed in the table coordinate, where (0, 0, 0) is the lower corner and (1, 1, 1) is the higher corner of the table.
 Here we put the toaster near the lower corner of the table.
 size: 0.35 # the size of a toaster is roughly 0.35m
 lang: "a common toaster"
 path: "toaster.urdf"
- type: mesh
 name: "bread slice"
 on_table: True # Bread is usually placed on the table as well.
 center: (0.8, 0.7, 0) # Remember that when an object is placed on the table, the center is expressed in the table coordinate, where (0, 0, 0) is the lower corner and (1, 1, 1) is the higher corner of the table.
 Here we put the bread slice near the higher corner of the table.
 size: 0.1 # common size of a bread slice
```

```

```

lang: "a slice of bread"
Path: "bread_slice.obj"
```

Another example input:
Task Name: Removing Lid From Pot
Description: The robotic arm will remove the lid from the pot.
Objects involved: KitchenPot. Only the objects specified here should be included in the yaml file.

```KitchenPot articulation tree
links:
base
link_0
link_1

joints:
joint_name: joint_0 joint_type: prismatic parent_link: link_1 child_link: link_0
joint_name: joint_1 joint_type: fixed parent_link: base child_link: link_1
```

```KitchenPot semantics
link_0 slider lid
link_1 free pot_body
```

Output:
```yaml
- use_table: True # A kitchen pot is usually placed on the table.
- type: mesh
  name: "KitchenPot"
  on_table: True # kitchen pots are usually placed on a table.
  center: (0.3, 0.6, 0) # Remember that when an object is placed on the table, the center is expressed in the
    table coordinate, where (0, 0, 0) is the lower corner and (1, 1, 1) is the higher corner of the table.
    Here we put the kitchen pot just at a random location on the table.
  size: 0.28 # the size of a common kitchen pot is roughly 0.28m
  lang: "a common kitchen pot"
  path: "kitchen_pot.urdf"
```

Note in this example, the kitchen pot already has a lid from the semantics file. Therefore, you do not need to
include a separate lid in the yaml file.

One more example input:
Task Name: Heat a hamburger in the oven.
Description: The robotic arm will put a hamburger in the oven and use the oven to heat it.
Objects involved: A hamburger, an oven. Only the objects here should be included in the yaml file.

```Oven articulation tree
links:
base
link_0
link_1
link_2
link_3
link_4
link_5
link_6
link_7

joints:
joint_name: joint_0 joint_type: revolute parent_link: link_7 child_link: link_0
joint_name: joint_1 joint_type: continuous parent_link: link_7 child_link: link_1
joint_name: joint_2 joint_type: continuous parent_link: link_7 child_link: link_2
joint_name: joint_3 joint_type: continuous parent_link: link_7 child_link: link_3
joint_name: joint_4 joint_type: continuous parent_link: link_7 child_link: link_4
joint_name: joint_5 joint_type: continuous parent_link: link_7 child_link: link_5
joint_name: joint_6 joint_type: continuous parent_link: link_7 child_link: link_6
joint_name: joint_7 joint_type: fixed parent_link: base child_link: link_7
```

```Oven semantics
link_0 hinge door
link_1 hinge knob
link_2 hinge knob
link_3 hinge knob
link_4 hinge knob
link_5 hinge knob
link_6 hinge knob
link_7 heavy oven_body
```

Output:
```yaml
- use_table: True # Although an oven is usually just placed on the ground, a hamburger is usually placed on
  the table.
- type: mesh
  name: "Oven"
  on_table: False # An oven is usually just placed on the floor.
  center: (1.0, 0, 0) # Remember that when not on a table, the center is expressed in the world coordinate.
  Since the robot is at (1, 1, 0) and the table is at (0, 0, 0), we place the oven at (1.8, 2, 0) to
  avoid collision with the table and the robot.
  size: 0.9 # the size of an oven is roughly 0.9m
  lang: "a common oven"
```

```

---

```

path: "oven.urdf"
- type: mesh
 name: "hamburger"
 on_table: True # a hamburger is usually placed on the table.
 center: (0.2, 0.4, 0) # Remember that when an object is placed on the table, the center is expressed in the
 table coordinate, where (0, 0, 0) is the lower corner and (1, 1, 1) is the higher corner of the table.
 Here we put the hamburger just at a random location on the table.
 size: 0.1 # common hamburgers' diameters are 10 cm.
 lang: "a common hamburger"
 path: "hamburger.obj"
```

Rules:
- You do not need to include the robot in the yaml file.
- The yaml file should only include the objects listed in "Objects involved".
- Sometimes, the task description / objects involved will refer to generic/placeholder objects, e.g., to place
  an "item" into the drawer, and to heat "food" in the microwave. In the generated yaml config, you
  should change these placeholder objects to be concrete objects in the lang field, e.g., change "item" to
  be a toy or a pencil, and "food" to be a hamburger, a bowl of soup, etc.

Can you do this for the following task:
Task Name: Throw Trash Away
Description: The robotic arm places an item of trash inside the trash can
Objects involved: TrashCan, A pile of trash

```TrashCan articulation tree
links:
base
link_0
link_1

joints:
joint_name: joint_0 joint_type: revolute parent_link: link_1 child_link: link_0
joint_name: joint_1 joint_type: fixed parent_link: base child_link: link_1
```
```TrashCan semantics
link_0 hinge door
link_1 free trashcan_body
```
```

```

### An example response from GPT-4:

```

Output:
```yaml
- use_table: False # Trash cans are usually placed on the floor.
- type: mesh
  name: "TrashCan"
  on_table: False # Trash cans are usually placed on the floor.
  center: (1.5, 1.5, 0) # Remember that when not on a table, the center is expressed in the world coordinate.
    Since the robot is at (1, 1, 0) and the table is at (0, 0, 0), we place the trash can at (1.5, 1.5, 0)
    to avoid collision with the table and the robot.
  size: 0.6 # the size of a common trash can is roughly 0.6m
  lang: "common trash can"
  path: "trashcan.urdf"
- type: mesh
  name: "Trash"
  on_table: True # Trash is usually picked up from a table or the floor. Here we assume it's on the table.
  center: (0.5, 0.5, 0) # Remember that when an object is placed on the table, the center is expressed in the
    table coordinate, where (0, 0, 0) is the lower corner and (1, 1, 1) is the higher corner of the table.
    Here we put the trash in the middle of the table.
  size: 0.1 # common size of a pile of trash
  lang: "pile of paper waste" # Note here, we changed the generic/placeholder "pile of trash" object to be a
    more concrete object: a pile of paper waste.
  path: "paper_waste.obj"
```

```

### More relevant assets prompt.

Given a task, which is for a mobile Franka panda robotic arm to learn a manipulation skill in the simulator, your goal is to add more objects into the task scene such that the scene looks more realistic. The Franka panda arm is mounted on a floor, at location (1, 1, 0). It can move freely on the floor. The z axis is the gravity axis.

The input to you includes the following:  
Task name, task description, the essential objects involved in the task, and a config describing the current task scene, which contains only the essential objects needed for the task. The config is a yaml file in the following format:

```

```yaml
- use_table: whether the task requires using a table. This should be decided based on common sense. If a table
  is used, its location will be fixed at (0, 0, 0). The height of the table will be 0.6m.
# for each object involved in the task, we need to specify the following fields for it.
- type: mesh
  name: name of the object, so it can be referred to in the simulator
  size: describe the scale of the object mesh using 1 number in meters. The scale should match real everyday
    objects. E.g., an apple is of scale 0.08m. You can think of the scale to be the longest dimension of
    the object.
```

```

---

```

lang: this should be a language description of the mesh. The language should be a bit detailed, such that
 the language description can be used to search an existing database of objects to find the object.
path: this can be a string showing the path to the mesh of the object.
on_table: whether the object needs to be placed on the table (if there is a table needed for the task). This
 should be based on common sense and the requirement of the task.
center: the location of the object center. If there isn't a table needed for the task or the object does not
 need to be on the table, this center should be expressed in the world coordinate system. If there is
 a table in the task and the object needs to be placed on the table, this center should be expressed in
 terms of the table coordinate, where (0, 0, 0) is the lower corner of the table, and (1, 1, 1) is the
 higher corner of the table. In either case, you should try to specify a location such that there is
 no collision between objects.
```

```

Your task is to think about what other distractor objects can be added into the scene to make the scene more complex and realistic for the robot to learn the task. These distractor objects are not necessary for the task itself, but their existence makes the scene look more interesting and complex. You should output the distractor objects using the same format as the input yaml file. You should try to put these distractor objects at locations such that they don't collide with objects already in the scene.

Here is one example:

Input:

```

Task name: Heat up a bowl of soup in the microwave
Task description: The robot will grab the soup and move it into the microwave, and then set the temperature to
      heat it.
Objects involved: Microwave, a bowl of soup
Config:
```yaml
- use_table: true
- center: (0.3, 0.7, 0)
 lang: A standard microwave with a turntable and digital timer
 name: Microwave
 on_table: true
 path: microwave.urdf
 size: 0.6
 type: urdf
- center: (0.2, 0.2, 0)
 lang: A ceramic bowl full of soup
 name: Bowl of Soup
 on_table: true
 path: bowl_soup.obj
 size: 0.15
 type: mesh
```

```

Output:

```

```yaml
- name: plate # a plate is a common object placed when there is microwave and bowl of soup, in a kitchen setup
 lang: a common kitchen plate
 on_table: True
 center: (0.8, 0.8, 0)
 type: mesh
 path: "plate.obj"
 size: 0.15 # a plate is usually of scale 0.15m
- name: sponge # a sponge is a common object placed when there is microwave and bowl of soup, in a kitchen
 setup
 lang: a common sponge
 on_table: True
 center: (0.5, 0.2, 0)
 type: mesh
 path: "sponge.obj"
 size: 0.1 # a sponge is usually of scale 0.1m
- name: Oven # an oven is a common object placed when there is microwave and bowl of soup, in a kitchen setup
 lang: a kitchen oven
 on_table: False # an oven is usually a standalone object on the floor
 center: (1.8, 0.5, 0) # remember robot is at (1, 1, 0) and table is at (0, 0, 0). So the oven is placed at
 (1.8, 0.5, 0) in the world coordinate system to avoid collision with other objects.
 type: mesh
 path: "oven.obj"
 size: 0.8 # an oven is usually of scale 0.8m
```

```

Can you do it for the following task:

```

Task name: Throw Trash Away
Task description: The robotic arm places an item of trash inside the trash can
Initial config:
```yaml
- use_table: false
- center: (1.5, 1.5, 0)
 lang: a common trash can
 name: TrashCan
 on_table: false
 path: trashcan.urdf
 size: 0.6
 type: urdf
- center: (0.5, 0.5, 0)
 lang: a pile of paper waste
 name: Trash
 on_table: true
 path: paper_waste.obj
 size: 0.05
```

```

```

type: mesh
```
An example response from GPT-4:

Sure, here is the output for the given task:

```yaml
- name: broom # a broom is a common object found near a trash can
  lang: a common broom
  on_table: False
  center: (2, 1.5, 0)
  type: mesh
  path: "broom.obj"
  size: 1.2 # a broom is usually of scale 1.2m
- name: dustpan # a dustpan is a common object found near a trash can
  lang: a common dustpan
  on_table: False
  center: (2, 2, 0)
  type: mesh
  path: "dustpan.obj"
  size: 0.3 # a dustpan is usually of scale 0.3m
- name: recycling_bin # a recycling bin is a common object found near a trash can
  lang: a common recycling bin
  on_table: False
  center: (1, 2, 0)
  type: mesh
  path: "recycling_bin.obj"
  size: 0.6 # a recycling bin is usually of scale 0.6m
- name: soda_can # a soda can is a common object found near a trash can
  lang: a common soda can
  on_table: True
  center: (0.7, 0.7, 0)
  type: mesh
  path: "soda_can.obj"
  size: 0.1 # a soda can is usually of scale 0.1m
```

```

**Asset size adjustment prompt.** For each object used in the scene, we query GPT-4 to correct its size such that the size matches the real-world size. We continue the above example of throw trash away.

A robotic arm is trying to manipulate some objects to learn corresponding skills in a simulator. However, the size of the objects might be wrong. Your task is to adjust the size of the objects, such that they match each other when interact with each other; and the size should also match what is commonly seen in everyday life, in household scenarios.

Now I will give you the name of the task, the object and their sizes, please correct any unreasonable sizes.

Objects are represented using a mesh file, you can think of size as the longest dimension of the object.

I will write in the following format:

```

```
Task: task description
obj1, mesh, size
obj2, mesh, size
```

```

Please reply in the following format:  
explanations of why some size is not reasonable.

```

```yaml
obj1, mesh, corrected_size
obj2, mesh, corrected_radius
```

```

Here is an example:

Input:

```

```
Task: The robotic arm lowers the toilet seat from an up position to a down position
Toilet, mesh, 0.2
```

```

Output:

A toilet is usually 0.6 - 0.8m in its back height, so the size is not reasonable -- it is a bit too small.

Below is the corrected size.

```

```yaml
Toilet, mesh, 0.7
```

```

Another example:

Input:

```

```
Task: Fill a cup with water under the faucet
Faucet, mesh, 0.25
Cup, mesh, 0.3
```

```

Output:

---

The size of the faucet makes sense. However, the size of the cup is too large for 2 reasons: it does not match the size of the faucet for getting water under the faucet; and it is not a common size of cup in everyday life. Below is the corrected size.

```
'''yaml
Faucet, mesh, 0.25
Cup, mesh, 0.12
'''
```

One more example to show that even if no change is needed, you should still reply with the same size.

Input:

```
'''
```

Task: Open Table Drawer The robotic arm will open a table drawer

```
table, mesh, 0.8
'''
```

Output:

The size of the table is reasonable, so no change is needed.

```
'''yaml
table, mesh, 0.8
'''
```

This is also a good example to show that sometimes, the task description might include two objects, e.g., a table and a drawer, yet there is only one object size provided (here the table). This is not an error, but that the other object is part of the provided object, i.e., here the drawer is part of the table. It's fine, you should then just reply with the corrected size of the object provided, here, the table, in such cases.

```
'''
```

Task: Throw Trash Away The robotic arm places an item of trash inside the trash can

```
trashcan, mesh, 0.6
trash, mesh, 0.1
'''
```

### An example response from GPT-4:

The size of the trashcan is reasonable as it is a common size for a household trashcan. However, the size of the trash is a bit too large. Trash items are usually smaller to fit inside the trashcan. Below is the corrected size.

```
'''yaml
trashcan, mesh, 0.6
trash, mesh, 0.05
'''
```

For scene generation, we also need to specify the asset configuration, and any special scene configuration.

**Adjusting articulated object joint prompt.** We continue the example of throwing the trash away.

Your goal is to set the joint angles of some articulated objects to the right value in the initial state, given a task. The task is for a robot arm to learn the corresponding skills to manipulate the articulated object.

The input to you will include the task name, a short description of the task, the articulation tree of the articulated object, a semantic file of the articulated object, the links and joints of the articulated objects that will be involved in the task, and the substeps for doing the task.

You should output for each joint involved in the task, what joint value it should be set to. You should output a number in the range [0, 1], where 0 corresponds to the lower limit of that joint angle, and 1 corresponds to the upper limit of the joint angle. You can also output a string of "random", which indicates to sample the joint angle within the range.

By default, the joints in an object are set to their lower joint limits. You can assume that the lower joint limit corresponds to the natural state of the articulated object. E.g., for a door's hinge joint, 0 means it is closed, and 1 means it is open. For a lever, 0 means it is unpushed, and 1 means it is pushed to the limit.

Here is an example:

Input:  
Task Name: Close the door  
Description: The robot arm will close the door after it was opened.

```
'''door articulation tree
links:
base
link_0
link_1
link_2

joints:
joint_name: joint_0 joint_type: revolute parent_link: link_1 child_link: link_0
joint_name: joint_1 joint_type: fixed parent_link: base child_link: link_1
joint_name: joint_2 joint_type: revolute parent_link: link_0 child_link: link_2
'''

'''door semantics
link_0 hinge rotation_door
link_1 static door_frame
```

---

```

link_2 hinge rotation_door
```

Links:
- link_0: link_0 is the door. This is the part of the door assembly that the robot needs to interact with.
Joints:
- joint_0: Joint_0 is the revolute joint connecting link_0 (the door) as per the articulation tree. The robot
    needs to actuate this joint cautiously to ensure the door is closed.

substeps:
approach the door
close the door

Output:
The goal is for the robot arm to learn to close the door after it is opened. Therefore, the door needs to be
    initially open, thus, we are setting its value to 1, which corresponds to the upper joint limit.
```joint values
joint_0: 1
```

Another example:
Task Name: Turn Off Faucet
Description: The robotic arm will turn the faucet off by manipulating the switch

```Faucet articulation tree
links:
base
link_0
link_1

joints:
joint_name: joint_0 joint_type: fixed parent_link: base child_link: link_0
joint_name: joint_1 joint_type: revolute parent_link: link_0 child_link: link_1
```

```Faucet semantics
link_0 static faucet_base
link_1 hinge switch
```

Links:
- link_0: link_0 is the door. This is the part of the door assembly that the robot needs to interact with.
Joints:
- joint_0: Joint_0 is the revolute joint connecting link_0 (the door) as per the articulation tree. The robot
    needs to actuate this joint cautiously to ensure the door is closed.

substeps:
grasp the faucet switch
turn off the faucet

Output:
For the robot to learn to turn off the faucet, it cannot be already off initially. Therefore, joint_1 should
    be set to its upper joint limit, or any value that is more than half of the joint range, e.g., 0.8.
```joint value
joint_1: 0.8
```

One more example:
Task Name: Store an item inside the Drawer
Description: The robot arm picks up an item and places it inside the drawer of the storage furniture

```StorageFurniture articulation tree
links:
base
link_0
link_1
link_2

joints:
joint_name: joint_0 joint_type: revolute parent_link: link_1 child_link: link_0
joint_name: joint_1 joint_type: fixed parent_link: base child_link: link_1
joint_name: joint_2 joint_type: prismatic parent_link: link_1 child_link: link_2
```

```StorageFurniture semantics
link_0 hinge rotation_door
link_1 heavy furniture_body
link_2 slider drawer
```

Links:
- link_2: link_2 is the drawer link from the semantics. The robot needs to open this drawer to place the item
    inside.
Joints:
- joint_2: joint_2, from the articulation tree, connects to link_2 (the drawer). Thus, the robot would need to
    actuate this joint to open the drawer to store the item.

substeps:
grasp the drawer
open the drawer
grasp the item

```

```

put the item into the drawer
grasp the drawer again
close the drawer
release the grasp

Output:
This task involves putting one item into the drawer of the storage furniture. As noted in the substeps, the robot needs to first open the drawer, put the item in, and then close it. Since the articulated object is initialized with the lower joint limit, i.e., the drawer is initially closed, it aligns with the task where the robot needs to first learn to open the drawer. Therefore, no particular joint angle needs to be set, and we just output None.

'''joint value
None
'''

Can you do it for the following task:

Task Name: Throw Trash Away
Description: The robotic arm places an item of trash inside the trash can

'''TrashCan articulation tree
links:
base
link_0
link_1

joints:
joint_name: joint_0 joint_type: revolute parent_link: link_1 child_link: link_0
joint_name: joint_1 joint_type: fixed parent_link: base child_link: link_1

'''

'''TrashCan semantics
link_0 hinge door
link_1 free trashcan_body

'''

Links:
link_0: Link_0, as the trash can lid, needs to be approached and opened by the robot in order to throw away the trash.

Joints:
joint_0: Joint_0, connecting to link_0, needs to be actuated for the robot to open the trash can lid.

substeps:
grasp the trash can lid
open the trash can lid
grasp the trash
put the trash into the trash can
release the trash
grasp the trash can lid again
close the trash can lid

```

An example output from GPT-4:

```

Output:
The task involves the robot learning to throw trash away. The robot needs to learn to open the trash can lid, throw the trash in, and then close the lid. Therefore, we need to set the initial joint angle such that the trash can lid is not already open. We can set joint_0 to its lower limit, which corresponds to the trash can lid being closed.

'''joint value
joint_0: 0
'''
```

Special spatial relationship prompt. We keep the “Throw away trash” example.

Your goal is to output any special spatial relationships certain objects should have in the initial state, given a task. The task is for a robot arm to learn the corresponding skills in household scenarios.

The input to you will include
the task name,
a short description of the task,
objects involved in the task,
substeps for performing the task,
If there is an articulated object involved in the task, the articulation tree of the articulated object, the semantic file of the articulated object, and the links and joints of the articulated objects that will be involved in the task.

We have the following spatial relationships:
on, obj_A, obj_B: object A is on top of object B, e.g., a fork on the table.
in, obj_A, obj_B: object A is inside object B, e.g., a gold ring in the safe.
in, obj_A, obj_B, link_name: object A is inside the link with link_name of object B. For example, a table might have two drawers, represented with link_0, and link_1, and in(pen, table, link_0) would be that a pen is inside one of the drawers that corresponds to link_0.

Given the input to you, you should output any needed spatial relationships of the involved objects.

Here are some examples:

Input:

```

Task Name:Fetch Item from Refrigerator
Description: The robotic arm will open a refrigerator door and reach inside to grab an item and then close the
door.
Objects involved: refrigerator, item

```refrigerator articulation tree
links:
base
link_0
link_1
link_2

joints:
joint_name: joint_0 joint_type: fixed parent_link: base child_link: link_0
joint_name: joint_1 joint_type: revolute parent_link: link_0 child_link: link_1
joint_name: joint_2 joint_type: revolute parent_link: link_0 child_link: link_2
```

```refrigerator semantics
link_0 heavy refrigerator_body
link_1 hinge door
link_2 hinge door
```

Links:
link_1: The robot needs to approach and open this link, which represents one of the refrigerator doors, to
reach for the item inside.
Joints:
joint_1: This joint connects link_1, representing one of the doors. The robot needs to actuate this joint to
open the door, reach for the item, and close the door.

Substeps:
grasp the refrigerator door
open the refrigerator door
grasp the item
move the item out of the refrigerator
grasp the refrigerator door again
close the refrigerator door

Output:
The goal is for the robot arm to learn to retrieve an item from the refrigerator. Therefore, the item needs to
be initially inside the refrigerator. From the refrigerator semantics we know that link_0 is the body
of the refrigerator, therefore we should have a spatial relationship as the following:
```spatial relationship
In, item, refrigerator, link_0
```

Another example:
Task Name: Turn Off Faucet
Description: The robotic arm will turn the faucet off by manipulating the switch
Objects involved: faucet

```Faucet articulation tree
links:
base
link_0
link_1

joints:
joint_name: joint_0 joint_type: fixed parent_link: base child_link: link_0
joint_name: joint_1 joint_type: revolute parent_link: link_0 child_link: link_1
```

```Faucet semantics
link_0 static faucet_base
link_1 hinge switch
```

Links:
link_0: link_0 is the door. This is the part of the door assembly that the robot needs to interact with.
Joints:
joint_0: Joint_0 is the revolute joint connecting link_0 (the door) as per the articulation tree. The robot
needs to actuate this joint cautiously to ensure the door is closed.

Substeps:
grasp the faucet switch
turn off the faucet

Output:
There is only 1 object involved in the task, thus no special spatial relationships are required.
```spatial relationship
None
```

```

One more example:

Task Name: Store an item inside the Drawer

Description: The robot arm picks up an item and places it inside the drawer of the storage furniture.

Objects involved: storage furniture, item

```

```StorageFurniture articulation tree
links:
base
link_0
link_1
link_2

joints:
joint_name: joint_0 joint_type: revolute parent_link: link_1 child_link: link_0
joint_name: joint_1 joint_type: fixed parent_link: base child_link: link_1
joint_name: joint_2 joint_type: prismatic parent_link: link_1 child_link: link_2
```

```StorageFurniture semantics
link_0 hinge rotation_door
link_1 heavy furniture_body
link_2 slider drawer
```

Links:
link_2: link_2 is the drawer link from the semantics. The robot needs to open this drawer to place the item inside.

Joints:
joint_2: joint_2, from the articulation tree, connects to link_2 (the drawer). Thus, the robot would need to actuate this joint to open the drawer to store the item.

Substeps:
grasp the drawer
open the drawer
grasp the item
put the item into the drawer
grasp the drawer again
close the drawer
release the grasp

Output:
This task involves putting one item into the drawer of the storage furniture. The item should initially be outside of the drawer, such that the robot can learn to put it into the drawer. Therefore, no special relationships of in or on are needed. Therefore, no special spatial relationships are needed.

```spatial relationship
None
```

```

Can you do it for the following task:

Task Name: Throw Trash Away

Description: The robotic arm places an item of trash inside the trash can

Objects involved: TrashCan, Trash

```

```TrashCan articulation tree
links:
base
link_0
link_1

joints:
joint_name: joint_0 joint_type: revolute parent_link: link_1 child_link: link_0
joint_name: joint_1 joint_type: fixed parent_link: base child_link: link_1
```

```TrashCan semantics
link_0 hinge door
link_1 free trashcan_body
```

Links:
link_0: Link_0, as the trash can lid, needs to be approached and opened by the robot in order to throw away the trash.

Joints:
joint_0: Joint_0, connecting to link_0, needs to be actuated for the robot to open the trash can lid.

Substeps:
grasp the trash can lid
open the trash can lid
grasp the trash
put the trash into the trash can
release the trash
grasp the trash can lid again
close the trash can lid

```

An example output from GPT-4:

```
Output:  
The task involves the robot arm placing an item of trash into the trash can. Initially, the trash should not  
be inside the trash can, so the robot can learn to put it in. Therefore, no special spatial  
relationships are needed.  
'''spatial relationship  
None  
'''
```

Training supervision prompt.

```
A robotic arm is trying to solve some household object manipulation tasks to learn corresponding skills in a  
simulator.
```

```
We will provide you with the task description, the initial scene configurations of the task, which contains  
the objects in the task and certain information about them.  
Your goal is to decompose the task into executable sub-steps for the robot, and for each substep, you should  
either call a primitive action that the robot can execute, or design a reward function for the robot to  
learn, to complete the substep.  
For each substep, you should also write a function that checks whether the substep has been successfully  
completed.
```

```
Common substeps include moving towards a location, grasping an object, and interacting with the joint of an  
articulated object.
```

```
An example task:
```

```
Task Name: Fetch item from refrigerator  
Description: The robotic arm will open a refrigerator door reach inside to grab an item, place it on the table  
, and then close the door  
Initial config:  
'''yaml  
- use_table: true  
- center: (1.2, 0, 0)  
  lang: a common two-door refrigerator  
  name: Refrigerator  
  on_table: false  
  path: refrigerator.urdf  
  size: 1.8  
  type: urdf  
- center: (1.2, 0, 0.5)  
  lang: a can of soda  
  name: Item  
  on_table: false  
  path: soda_can.obj  
  size: 0.2  
  type: mesh  
'''
```

```
I will also give you the articulation tree and semantics file of the articulated object in the task. Such  
information will be useful for writing the reward function/the primitive actions, for example, when the  
reward requires accessing the joint value of a joint in the articulated object, or the position of a  
link in the articulated object, or when the primitive needs to access a name of the object.
```

```
'''Refrigerator articulation tree  
links:  
base  
link_0  
link_1  
link_2  
  
joints:  
joint_name: joint_0 joint_type: fixed parent_link: base child_link: link_0  
joint_name: joint_1 joint_type: revolute parent_link: link_0 child_link: link_1  
joint_name: joint_2 joint_type: revolute parent_link: link_0 child_link: link_2  
'''
```

```
'''Refrigerator semantics  
link_0 heavy refrigerator_body  
link_1 hinge door  
link_2 hinge door  
'''
```

```
I will also give you the links and joints of the articulated object that will be used for completing the task:
```

```
Links:  
link_1: This link is one of the refrigerator doors, which the robot needs to reach for the item inside.  
Joints:  
joint_1: This joint connects link_1, representing one of the doors. The robot needs to actuate this joint to  
open the door, reach for the item, and close the door.
```

```
For each substep, you should decide whether the substep can be achieved by using the provided list of  
primitives. If not, you should then write a reward function for the robot to learn to perform this  
substep.
```

```
If you choose to write a reward function for the substep, you should also specify the action space of the  
robot when learning this reward function.
```

```
There are 2 options for the action space: "delta-translation", where the action is the delta translation of  
the robot end-effector, suited for local movements; and "normalized-direct-translation", where the  
action specifies the target location the robot should move to, suited for moving to a target location.
```

```
For each substep, you should also write a condition that checks whether the substep has been successfully  
completed.
```

Here is a list of primitives the robot can do. The robot is equipped with a suction gripper, which makes it easy for the robot to grasp an object or a link on an object.

`grasp_object(self, object_name):` the robot arm will grasp the object specified by the argument `object_name`.

`grasp_object_link(self, object_name, link_name):` some object like an articulated object is composed of multiple links. The robot will grasp a link with `link_name` on the object with `object_name`.

`release_grasp(self):` the robot will release the grasped object.

`approach_object(self, object_name):` this function is similar to `grasp_object`, except that the robot only approaches the object, without grasping it.

`approach_object_link(self, object_name, link_name):` this function is similar to `grasp_object_link`, except that the robot only approaches the object's link, without grasping it.

Note that all primitives will return a tuple (`rgbs, final_state`) which represents the rgb images of the execution process and the final state of the execution process.

You should always call the primitive in the following format:

```
rgbs, final_state = some_primitive_function(self, arg1, ..., argn)
```

Here is a list of helper functions that you can use for designing the reward function or the success condition :

```
get_position(self, object_name): get the position of center of mass of object with object_name.
get_orientation(self, object_name): get the orientation of an object with object_name.
detect(self, object_name, object_part): detect the position of a part in object. E.g., the opening of a toaster, or the handle of a door.
get_joint_state(self, object_name, joint_name): get the joint angle value of a joint in an object.
get_joint_limit(self, object_name, joint_name): get the lower and upper joint angle limit of a joint in an object, returned as a 2-element tuple.
get_link_state(self, object_name, link_name): get the position of the center of mass of the link of an object.
get_eef_pos(self): returns the position, orientation of the robot end-effector as a list.
get_bounding_box(self, object_name): get the axis-aligned bounding box of an object. It returns the min and max xyz coordinate of the bounding box.
get_bounding_box_link(self, object_name, link_name): get the axis-aligned bounding box of the link of an object. It returns the min and max xyz coordinate of the bounding box.
in_bbox(self, pos, bbox_min, bbox_max): check if pos is within the bounding box with the lowest corner at bbox_min and the highest corner at bbox_max.
get_grasped_object_name(self): return the name of the grasped object. If no object is grasped by the robot, return None. The name is automatically converted to the lower case.
get_grasped_object_and_link_name(self): return a tuple, the first is the name of the grasped object, and the second is the name of the grasped link. If no object is grasped by the robot, return (None, None). The name is automatically converted to the lower case.
gripper_close_to_object(self, object_name): return true if the robot gripper is close enough to the object specified by object_name, otherwise false.
gripper_close_to_object_link(self, object_name, link_name): return true if the robot gripper is close enough to the object link, otherwise false.
```

You can assume that for objects, the lower joint limit corresponds to their natural state, e.g., a box is closed with the lid joint being 0, and a lever is unpushed when the joint angle is 0.

For the above task "Fetch item from refrigerator", it can be decomposed into the following substeps, primitives, and reward functions:

```
substep 1: grasp the refrigerator door
```primitive
 rgbs, final_state = grasp_object_link(self, "Refrigerator", "link_1")
 grasped_object, grasped_link = get_grasped_object_and_link_name(self)
 success = (grasped_object == "Refrigerator".lower() and grasped_link == "link_1".lower())
```

substep 2: open the refrigerator door
```reward
def _compute_reward(self):
 # this reward encourages the end-effector to stay near door to grasp it.
 eef_pos = get_eef_pos(self)[0]
 door_pos = get_link_state(self, "Refrigerator", "link_1")
 reward_near = -np.linalg.norm(eef_pos - door_pos)

 # Get the joint state of the door. We know from the semantics and the articulation tree that joint_1 connects link_1 and is the joint that controls the rotation of the door.
 joint_angle = get_joint_state(self, "Refrigerator", "joint_1")
 # The reward is the negative distance between the current joint angle and the joint angle when the door is fully open (upper limit).
 joint_limit_low, joint_limit_high = get_joint_limit(self, "Refrigerator", "joint_1")
 target_joint_angle = joint_limit_high
 diff = np.abs(joint_angle - target_joint_angle)
 reward_joint = -diff

 reward = reward_near + 5 * reward_joint

 success = diff < 0.1 * (joint_limit_high - joint_limit_low)
```
    return reward, success
```

action space
delta-translation
```
In the last substep the robot already grasps the door, thus only local movements are needed to open it.

substep 3: grasp the item
```primitive
 rgbs, final_state = grasp_object(self, "Item")
 success = get_grasped_object_name(self) == "Item".lower()
```
```

```

---

```

substep 4: move the item out of the refrigerator
````reward
def _compute_reward(self):
    # Get the current item position
    item_position = get_position(self, "Item")

    # The first reward encourages the end-effector to stay near the item
    eef_pos = get_eef_pos(self)[0]
    reward_near = -np.linalg.norm(eef_pos - item_position)

    # The reward is to encourage the robot to grasp the item and move the item to be on the table.
    # The goal is not to just move the soda can to be at a random location out of the refrigerator. Instead,
    # we need to place it somewhere on the table.
    # This is important for moving an object out of a container style of task.
    table_bbox_low, table_bbox_high = get_bounding_box(self, "init_table") # the table is referred to as "
        init_table" in the simulator.
    table_bbox_range = table_bbox_high - table_bbox_low

    # target location is to put the item at a random location on the table
    target_location = np.zeros(3)
    target_location[0] = table_bbox_low[0] + 0.2 * table_bbox_range[0] # 0.2 is a random chosen number, any
        number in [0, 1] should work
    target_location[1] = table_bbox_low[1] + 0.3 * table_bbox_range[1] # 0.3 is a random chosen number, any
        number in [0, 1] should work
    target_location[2] = table_bbox_high[2] # the height should be the table height
    diff = np.linalg.norm(item_position - target_location)
    reward_distance = -diff

    reward = reward_near + 5 * reward_distance

    success = diff < 0.06

    return reward, success
````

```action space
normalized-direct-translation
```
Since this substep requires moving the item to a target location, we use the normalized-direct-translation.

substep 5: grasp the refrigerator door again
```primitive
rgbs, final_state = grasp_object_link(self, "Refrigerator", "link_1")
grasped_object, grasped_link = get_grasped_object_and_link_name(self)
success = (grasped_object == "Refrigerator".lower() and grasped_link == "link_1".lower())
```

substep 6: close the refrigerator door
```reward
def _compute_reward(self):
    # this reward encourages the end-effector to stay near door
    eef_pos = get_eef_pos(self)[0]
    door_pos = get_link_state(self, "Refrigerator", "link_1")
    reward_near = -np.linalg.norm(eef_pos - door_pos)

    # Get the joint state of the door. The semantics and the articulation tree show that joint_1 connects
    # link_1 and is the joint that controls the rotation of the door.
    joint_angle = get_joint_state(self, "Refrigerator", "joint_1")
    # The reward encourages the robot to make joint angle of the door to be the lower limit to close it.
    joint_limit_low, joint_limit_high = get_joint_limit(self, "Refrigerator", "joint_1")
    target_joint_angle = joint_limit_low

    diff = np.abs(target_joint_angle - joint_angle)
    reward_joint = -diff

    reward = reward_near + 5 * reward_joint

    success = diff < 0.1 * (joint_limit_high - joint_limit_low)

    return reward, success
```

```action space
delta-translation
```

```

I will give some more examples of decomposing the task. Reply yes if you understand the goal.

=====

Yes, I understand the goal. Please proceed with the next example.

=====

Another example:

```

Task Name: Set oven temperature
Description: The robotic arm will turn the knob of an oven to set a desired temperature.
Initial config:
```yaml
- use_table: false

```

```

- center: (1, 0, 0) # when an object is not on the table, the center specifies its location in the world
  coordinate.
lang: a freestanding oven
name: oven
on_table: false
path: oven.urdf
size: 0.85
type: urdf
```
```
'''Oven articulation tree:
links:
base
link_0
link_1
link_2
link_3
link_4

joints:
joint_name: joint_0 joint_type: continuous parent_link: link_4 child_link: link_0
joint_name: joint_1 joint_type: continuous parent_link: link_4 child_link: link_1
joint_name: joint_2 joint_type: continuous parent_link: link_4 child_link: link_2
joint_name: joint_3 joint_type: continuous parent_link: link_4 child_link: link_3
joint_name: joint_4 joint_type: fixed parent_link: base child_link: link_4
```
```
'''Oven semantics
link_0 hinge knob
link_1 hinge knob
link_2 hinge knob
link_3 hinge knob
link_4 heavy oven_body
```
Links:
link_0: We know from the semantics that link_0 is a hinge knob. It is assumed to be the knob that controls the temperature of the oven. The robot needs to actuate this knob to set the temperature of the oven.

Joints:
joint_0: from the articulation tree, joint_0 connects link_0 and is a continuous joint. Therefore, the robot needs to actuate joint_0 to turn link_0, which is the knob.

This task can be decomposed as follows:

substep 1: grasp the temperature knob
```primitive
    rgbs, final_state = grasp_object_link(self, "oven", "link_0")
    grasped_object, grasped_link = get_grasped_object_and_link_name(self)
    success = (grasped_object == "oven".lower() and grasped_link == "link_0".lower())
```
```
substep 2: turn the temperature knob to set a desired temperature
```reward
def _compute_reward(self):
 # This reward encourages the end-effector to stay near the knob to grasp it.
 eef_pos = get_eef_pos(self)[0]
 knob_pos = get_link_state(self, "oven", "link_0")
 reward_near = -np.linalg.norm(eef_pos - knob_pos)

 joint_angle = get_joint_state(self, "oven", "joint_0")

 joint_limit_low, joint_limit_high = get_joint_limit(self, "oven", "joint_0")
 desired_temperature = joint_limit_low + (joint_limit_high - joint_limit_low) / 3 # We assume the target
 # desired temperature is one third of the joint angle. It can also be 1/3, or other values between
 # joint_limit_low and joint_limit_high.

 # The reward is the negative distance between the current joint angle and the joint angle of the desired
 # temperature.
 diff = np.abs(joint_angle - desired_temperature)
 reward_joint = -diff

 reward = reward_near + 5 * reward_joint

 success = diff < 0.1 * (joint_limit_high - joint_limit_low)

 return reward, success
```
```
action space
delta-translation
```
I will provide more examples in the following messages. Please reply yes if you understand the goal.

=====
Yes, I understand the goal. Please proceed with the next example.

=====
Here is another example:
```

```

Task Name: Put a toy car inside a box
Description: The robotic arm will open a box, grasp the toy car and put it inside the box.
Initial config:
'''yaml
- use_table: True
- center: (0.2, 0.3, 0)
  on_table: True
  lang: a box
  name: box
  size: 0.25
  type: urdf
- center: (0.1, 0.6, 0)
  on_table: True
  lang: a toy car
  name: toy_car
  size: 0.1
  type: mesh
```
```box articulation tree
links:
base
link_0
link_1
link_2

joints:
joint_name: joint_0 joint_type: revolute parent_link: link_2 child_link: link_0
joint_name: joint_1 joint_type: revolute parent_link: link_2 child_link: link_1
joint_name: joint_2 joint_type: fixed parent_link: base child_link: link_2
```
```box semantics
link_0 hinge rotation_lid
link_1 hinge rotation_lid
link_2 free box_body
```
Links:
link_0: To fully open the box, the robot needs to open both box lids. We know from the semantics that link_0 is one of the lids.
link_1: To fully open the box, the robot needs to open both box lids. We know from the semantics that link_1 is another lid.
Joints:
joint_0: from the articulation tree, joint_0 connects link_0 and is a hinge joint. Thus, the robot needs to actuate joint_0 to open link_0, which is the lid of the box.
joint_1: from the articulation tree, joint_1 connects link_1 and is a hinge joint. Thus, the robot needs to actuate joint_1 to open link_1, which is the lid of the box.

This task can be decomposed as follows:

substep 1: grasp the first lid of the box
```primitive
# The semantics shows that link_0 and link_1 are the lid links.
rgbs, final_state = grasp_object_link(self, "box", "link_0")
grasped_object, grasped_link = get_grasped_object_and_link_name(self)
success = (grasped_object == "box".lower() and grasped_link == "link_0".lower())
```
substep 2: open the first lid of the box
```reward
def _compute_reward(self):
    # This reward encourages the end-effector to stay near the lid to grasp it.
    eef_pos = get_eef_pos(self)[0]
    lid_pos = get_link_state(self, "box", "link_0")
    reward_near = -np.linalg.norm(eef_pos - lid_pos)

    # Get the joint state of the first lid. The semantics and the articulation tree show that joint_0 connects link_0 and is the joint that controls the rotation of the first lid link_0.
    joint_angle = get_joint_state(self, "box", "joint_0")
    # The reward is the negative distance between the current joint angle and the joint angle when the lid is fully open (upper limit).
    joint_limit_low, joint_limit_high = get_joint_limit(self, "box", "joint_0")
    target_joint_angle = joint_limit_high

    diff = np.abs(joint_angle - target_joint_angle)
    reward_joint = -diff

    reward = reward_near + 5 * reward_joint
    success = diff < 0.1 * (joint_limit_high - joint_limit_low)

    return reward, success
```
```action space
delta-translation
```
substep 3: grasp the second lid of the box
```primitive

```

```

        # We know from the semantics that link_0 and link_1 are the lid links.
        rgbs, final_state = grasp_object_link(self, "box", "link_1")
        grasped_object, grasped_link = get_grasped_object_and_link_name(self)
        success = (grasped_object == "box".lower() and grasped_link == "link_1".lower())
```
substep 4: open the second lid of the box
```reward
def _compute_reward(self):
    # This reward encourages the end-effector to stay near the lid to grasp it.
    eef_pos = get_eef_pos(self)[0]
    lid_pos = get_link_state(self, "box", "link_1")
    reward_near = -np.linalg.norm(eef_pos - lid_pos)

    # Get the joint state of the second lid. The semantics and the articulation tree show that joint_1
    # connects link_1 and is the joint that controls the rotation of the second lid link_1.
    joint_angle = get_joint_state(self, "box", "joint_1")
    # The reward is the negative distance between the current joint angle and the joint angle when the lid is
    # fully open (upper limit).
    joint_limit_low, joint_limit_high = get_joint_limit(self, "box", "joint_1")
    target_joint_angle = joint_limit_high

    diff = np.abs(joint_angle - target_joint_angle)
    reward_joint = -diff

    reward = reward_near + 5 * reward_joint

    success = diff < 0.1 * (joint_limit_high - joint_limit_low)
    return reward, success
```
```action space
delta-translation
```
substep 5: grasp the toy car
```primitive
    rgbs, final_state = grasp_object(self, "toy_car")
    success = get_grasped_object_name(self) == "toy_car".lower()
```
substep 6: put the toy car into the box
```reward
def _compute_reward(self):
    # Get the current car position
    car_position = get_position(self, "toy_car")

    # This reward encourages the end-effector to stay near the car to grasp it.
    eef_pos = get_eef_pos(self)[0]
    reward_near = -np.linalg.norm(eef_pos - car_position)

    # Get the box body bounding box
    min_aabb, max_aabb = get_bounding_box_link(self, "box", "link_4") # from the semantics, link_4 is the body
    # of the box.
    diff = np.array(max_aabb) - np.array(min_aabb)
    min_aabb = np.array(min_aabb) + 0.05 * diff # shrink the bounding box a bit
    max_aabb = np.array(max_aabb) - 0.05 * diff
    center = (np.array(max_aabb) + np.array(min_aabb)) / 2

    # another reward is one if the car is inside the box bounding box
    reward_in = 0
    if in_bbox(self, car_position, min_aabb, max_aabb): reward_in += 1

    # another reward is to encourage the robot to move the car to be near the box
    # we need this to give a dense reward signal for the robot to learn to perform this task.
    reward_reaching = -np.linalg.norm(center - car_position)

    # The task is considered to be successful if the car is inside the box bounding box
    success = in_bbox(self, car_position, min_aabb, max_aabb)

    # We give more weight to reward_in, which is the major goal of the task.
    reward = 5 * reward_in + reward_reaching + reward_near
    return reward, success
```
```action space
normalized-direct-translation
```
Since this substep requires moving the item to a target location, we use the normalized-direct-translation.
```

Please decompose the following task into substeps. For each substep, write a primitive/a reward function, write the success checking function, and the action space if the reward is used.

The primitives you can call for the robot to execute:  
`grasp_object(self, object_name)`: the robot arm will grasp the object specified by the argument `object_name`.  
`grasp_object_link(self, object_name, link_name)`: some object like an articulated object is composed of multiple links. The robot will grasp a link with `link_name` on the object with `object_name`.  
`release_grasp(self)`: the robot will release the grasped object.  
`approach_object(self, object_name)`: this function is similar to `grasp_object`, except that the robot only approaches the object, without grasping it.

---

```

approach_object_link(self, object_name, link_name): this function is similar to grasp_object_link, except that
 the robot only approaches the object's link, without grasping it.
Note that all primitives will return a tuple (rgbs, final_state) which represents the rgb images of the
 execution process and the final state of the execution process.
You should always call the primitive in the following format:
rgbs, final_state = some_primitive_function(self, arg1, ..., argn)

The APIs you can use for writing the reward function/success checking function:
get_position(self, object_name): get the position of center of mass of object with object_name.
get_orientation(self, object_name): get the orientation of an object with object_name.
get_joint_state(self, object_name, joint_name): get the joint angle value of a joint in an object.
get_joint_limit(self, object_name, joint_name): get the lower and upper joint angle limit of a joint in an
 object, returned as a 2-element tuple.
get_link_state(self, object_name, link_name): get the position of the center of mass of the link of an object.
get_eef_pos(self): returns the position, orientation of the robot end-effector as a list.
get_bounding_box(self, object_name): get the axis-aligned bounding box of an object. It returns the min and
 max xyz coordinate of the bounding box.
get_bounding_box_link(self, object_name, link_name): get the axis-aligned bounding box of the link of an
 object. It returns the min and max xyz coordinate of the bounding box.
in_bbox(self, pos, bbox_min, bbox_max): check if pos is within the bounding box with the lowest corner at
 bbox_min and the highest corner at bbox_max.
get_grasped_object_name(self): return the name of the grasped object. If no object is grasped by the robot,
 return None. The name is automatically converted to the lower case.
get_grasped_object_and_link_name(self): return a tuple, the first is the name of the grasped object, and the
 second is the name of the grasped link. If no object is grasped by the robot, return (None, None). The
 name is automatically converted to the lower case.
gripper_close_to_object(self, object_name): return true if the robot gripper is close enough to the object
 specified by object_name, otherwise false.
gripper_close_to_object_link(self, object_name, link_name): return true if the robot gripper is close enough
 to the object link, otherwise false.

The action space you can use for learning with the reward: delta-translation is better suited for small
 movements, and normalized-direct-translation is better suited for directly specifying the target
 location of the robot end-effector.
You can assume that for objects, the lower joint limit corresponds to their natural state, e.g., a box is
 closed with the lid joint being 0, and a lever is unpushed when the joint angle is 0.

Task name: Throw Trash Away
Description: The robotic arm places an item of trash inside the trash can
Initial config:
```yaml
- use_table: false
- center: (1.5, 1.5, 0)
  lang: a common trash can
  name: TrashCan
  on_table: false
  path: trashcan.urdf
  reward_asset_path: '102187'
  size: 0.6
  type: urdf
- center: (0.5, 0.5, 0)
  lang: a pile of paper waste
  name: Trash
  on_table: true
  path: paper_waste.obj
  size: 0.05
  type: mesh
```
```
```TrashCan articulation tree
links:
base
link_0
link_1

joints:
joint_name: joint_0 joint_type: revolute parent_link: link_1 child_link: link_0
joint_name: joint_1 joint_type: fixed parent_link: base child_link: link_1
```
```
```TrashCan semantics
link_0 hinge door
link_1 free trashcan_body
```
```
Links:
link_0: Link_0, as the trash can lid, needs to be approached and opened by the robot in order to throw away
    the trash.
Joints:
joint_0: Joint_0, connecting to link_0, needs to be actuated for the robot to open the trash can lid.

```

An example response for the above task “Throw the trash away”:

This task can be decomposed into the following substeps:

substep 1: grasp the trash can lid

```

```primitive
 rgbs, final_state = grasp_object_link(self, "TrashCan", "link_0")
 grasped_object, grasped_link = get_grasped_object_and_link_name(self)
 success = (grasped_object == "TrashCan".lower() and grasped_link == "link_0".lower())
```

substep 2: open the trash can lid
```reward
def _compute_reward(self):
 # This reward encourages the end-effector to stay near the lid to grasp it.
 eef_pos = get_eef_pos(self)[0]
 lid_pos = get_link_state(self, "TrashCan", "link_0")
 reward_near = -np.linalg.norm(eef_pos - lid_pos)

 # Get the joint state of the lid. The semantics and the articulation tree show that joint_0 connects
 # link_0 and is the joint that controls the rotation of the lid.
 joint_angle = get_joint_state(self, "TrashCan", "joint_0")
 # The reward is the negative distance between the current joint angle and the joint angle when the lid is
 # fully open (upper limit).
 joint_limit_low, joint_limit_high = get_joint_limit(self, "TrashCan", "joint_0")
 target_joint_angle = joint_limit_high

 diff = np.abs(joint_angle - target_joint_angle)
 reward_joint = -diff

 reward = reward_near + 5 * reward_joint

 success = diff < 0.1 * (joint_limit_high - joint_limit_low)

 return reward, success
```

```action space
delta-translation
```

substep 3: grasp the trash
```primitive
 rgbs, final_state = grasp_object(self, "Trash")
 success = get_grasped_object_name(self) == "Trash".lower()
```

substep 4: put the trash into the trash can
```reward
def _compute_reward(self):
 # Get the current trash position
 trash_position = get_position(self, "Trash")

 # This reward encourages the end-effector to stay near the trash to grasp it.
 eef_pos = get_eef_pos(self)[0]
 reward_near = -np.linalg.norm(eef_pos - trash_position)

 # Get the trash can body bounding box
 min_aabb, max_aabb = get_bounding_box_link(self, "TrashCan", "link_1") # from the semantics, link_1 is the
 # body of the trash can.
 diff = np.array(max_aabb) - np.array(min_aabb)
 min_aabb = np.array(min_aabb) + 0.05 * diff # shrink the bounding box a bit
 max_aabb = np.array(max_aabb) - 0.05 * diff
 center = (np.array(max_aabb) + np.array(min_aabb)) / 2

 # another reward is one if the trash is inside the trash can bounding box
 reward_in = 0
 if in_bbox(self, trash_position, min_aabb, max_aabb): reward_in += 1

 # another reward is to encourage the robot to move the trash to be near the trash can
 # we need this to give a dense reward signal for the robot to learn to perform this task.
 reward_reaching = -np.linalg.norm(center - trash_position)

 # The task is considered to be successful if the trash is inside the trash can bounding box
 success = in_bbox(self, trash_position, min_aabb, max_aabb)

 # We give more weight to reward_in, which is the major goal of the task.
 reward = 5 * reward_in + reward_reaching + reward_near
 return reward, success
```

```action space
normalized-direct-translation
```

Since this substep requires moving the item to a target location, we use the normalized-direct-translation.

substep 5: release the trash
```primitive
 rgbs, final_state = release_grasp(self)
 success = get_grasped_object_name(self) == None
```

substep 6: grasp the trash can lid again
```primitive
 rgbs, final_state = grasp_object_link(self, "TrashCan", "link_0")
 grasped_object, grasped_link = get_grasped_object_and_link_name(self)
 success = (grasped_object == "TrashCan".lower() and grasped_link == "link_0".lower())
```

```

```

```
substep 7: close the trash can lid
```reward
def _compute_reward(self):
    # This reward encourages the end-effector to stay near the lid
    eef_pos = get_eef_pos(self)[0]
    lid_pos = get_link_state(self, "TrashCan", "link_0")
    reward_near = -np.linalg.norm(eef_pos - lid_pos)

    # Get the joint state of the lid. The semantics and the articulation tree show that joint_0 connects
    # link_0 and is the joint that controls the rotation of the lid.
    joint_angle = get_joint_state(self, "TrashCan", "joint_0")
    # The reward encourages the robot to make joint angle of the lid to be the lower limit to close it.
    joint_limit_low, joint_limit_high = get_joint_limit(self, "TrashCan", "joint_0")
    target_joint_angle = joint_limit_low

    diff = np.abs(target_joint_angle - joint_angle)
    reward_joint = -diff

    reward = reward_near + 5 * reward_joint

    success = diff < 0.1 * (joint_limit_high - joint_limit_low)

    return reward, success
```

```action space
delta-translation
```

```

### Another example of heating up a bowl of soup in the microwave:

This task can be decomposed as follows:

```

substep 1: grasp the microwave door
```primitive
    rgbs, final_state = grasp_object_link(self, "Microwave", "link_0")
    grasped_object, grasped_link = get_grasped_object_and_link_name(self)
    success = (grasped_object == "Microwave".lower() and grasped_link == "link_0".lower())
```

substep 2: open the microwave door
```reward
def _compute_reward(self):
    # Get the joint state of the door. We know from the semantics and the articulation tree that joint_0
    # connects link_0 and is the joint that controls the rotation of the door.
    joint_angle = get_joint_state(self, "Microwave", "joint_0")
    # The reward is the negative distance between the current joint angle and the joint angle when the door is
    # fully open (upper limit).
    joint_limit_low, joint_limit_high = get_joint_limit(self, "Microwave", "joint_0")
    target_joint_angle = joint_limit_high

    diff = np.abs(joint_angle - target_joint_angle)
    reward = -diff

    success = diff < 0.1 * (joint_limit_high - joint_limit_low)

    return reward, success
```

```action space
delta-translation
```
Here from the last substep the robot already grasps the microwave door, thus only local movements are needed
to open the door.

substep 3: grasp the bowl of soup
```primitive
    rgbs, final_state = grasp_object(self, "Bowl of soup")
    success = get_grasped_object_name(self) == "Bowl of soup".lower()
```

substep 4: put the bowl of soup into the microwave
```reward
def _compute_reward(self):
    # Get the current soup position
    soup_position = get_position(self, "Bowl of soup")

    # Get the microwave body bounding box
    min_aabb, max_aabb = get_bounding_box_link(self, "Microwave", "link_3") # from the semantics, link_3 is
    # the body of the microwave.
    diff = np.array(max_aabb) - np.array(min_aabb)
    min_aabb = np.array(min_aabb) + 0.05 * diff # shrink the bounding box a bit
    max_aabb = np.array(max_aabb) - 0.05 * diff
    center = (np.array(max_aabb) + np.array(min_aabb)) / 2

    # First reward is one if the soup is inside the microwave bounding box
    reward_in = 0
    if in_bbox(self, soup_position, min_aabb, max_aabb): reward_in += 1
```

```

---

```

Second reward is to encourage the robot to grasp the soup and move the soup to be near the microwave
we need this to give a dense reward signal for the robot to learn to perform this task.
reward_reaching = -np.linalg.norm(center - soup_position)

The task is considered to be successful if the soup is inside the microwave bounding box
success = in_bbox(self, soup_position, min_aabb, max_aabb)

We give more weight to the first reward which is putting the soup into the microwave.
reward = 5 * reward_in + reward_reaching
return reward, success
```
```
```action space
normalized-direct-translation
```
Since this substep requires moving the item to a target location, we use the normalized-direct-translation.

substep 5: grasp the microwave door again
```primitive
    rgbs, final_state = grasp_object_link(self, "Microwave", "link_0")
    grasped_object, grasped_link = get_grasped_object_and_link_name(self)
    success = (grasped_object == "Microwave".lower() and grasped_link == "link_0".lower())
```
```
substep 6: close the microwave door
```reward
def _compute_reward(self):
 # Get the joint state of the door. We know from the semantics and the articulation tree that joint_0
 # connects link_0 and is the joint that controls the rotation of the door.
 joint_angle = get_joint_state(self, "Microwave", "joint_0")
 # The reward is the negative distance between the current joint angle and the joint angle when the door is
 # fully closed (lower limit).
 joint_limit_low, joint_limit_high = get_joint_limit(self, "Microwave", "joint_0")
 target_joint_angle = joint_limit_low

 diff = np.abs(target_joint_angle - joint_angle)
 reward = -diff

 success = diff < 0.1 * (joint_limit_high - joint_limit_low)
 return reward, success
```
```
```action space
delta-translation
```
Here from the last substep the robot already grasps the microwave door, thus only local movements are needed
to close the door.

substep 7: grasp the microwave timer knob
```primitive
    rgbs, final_state = grasp_object_link(self, "Microwave", "link_1")
    grasped_object, grasped_link = get_grasped_object_and_link_name(self)
    success = (grasped_object == "Microwave".lower() and grasped_link == "link_1".lower())
```
```
substep 8: turn the microwave timer knob to set a desired heating time
```reward
def _compute_reward(self):
 # Get the joint state of the timer knob. We know from the semantics and the articulation tree that joint_1
 # connects link_1 and is the joint that controls the timer knob.
 joint_angle = get_joint_state(self, "Microwave", "joint_1")

 joint_limit_low, joint_limit_high = get_joint_limit(self, "Microwave", "joint_1")
 desired_time = joint_limit_low + (joint_limit_high - joint_limit_low) / 2 # We assume the target desired
 # time is half of the joint angle. It can also be one third, or other values between joint_limit_low
 # and joint_limit_high.

 # The reward is the negative distance between the current joint angle and the joint angle of the desired
 # time.
 diff = np.abs(joint_angle - desired_time)
 reward = -diff

 # if the difference is small enough, we consider it a success. Here the threshold value is determined as a
 # ratio of the joint angle range.
 success = diff < 0.1 * (joint_limit_high - joint_limit_low)
 return reward, success
```
```
```action space
delta-translation
```
Since the robot already gras

```

Here unfortunately we exceeded the 8k token limit of GPT-4. But the main body of the task decompositions have been finished, and the response is still good to be used.