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 GenBot, 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 GenBot.

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

"""
Taks: 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
""",

"""
Taks: 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 GenBot.

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 sematics. 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 invovled 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
         obejct, 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: "a 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: "a 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 d o n 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 # a 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 senes. However, the size of the cup is too large for 2 reasons: it does not match
       the size of tha 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 opened, 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 with you 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 neesd 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 clost 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.
````