Appendices for "RLingua: Improving Reinforcement Learning Sample Efficiency in Robotic Manipulations With Large Language Models"

Liangliang Chen, Yutian Lei, Shiyu Jin, Ying Zhang, *Senior Member, IEEE,* and Liangjun Zhang

APPENDIX A
PSEUDO-CODE OF RLINGUA WITH TD3

The pseudo-code of RLingua with TD3 is shown in Algorithm A-1.

Algorithm A-1 RLingua With TD3

```
Input: The robot task description and optional feedback from a non-expert, a competent LLM, the initial probability $p^{\mathrm{LLM}}$ sampling an action from the LLM-generated policy and its decay rate $\lambda_{\mathrm{annl}}$, two empty replay buffers $\mathcal{R}_{\mathrm{LLM}}$ and $\mathcal{R}_{\mathrm{RL}}$, the data batch size $N$, the reward discount factor $\gamma$, the weight of imitation learning loss $\lambda_{\mathrm{IM}}$, and the small smoothing parameter $\tau$ for updating target networks
Output: A robot controller $\pi_{\boldsymbol{\theta}^\pi}$ for the specified task
    Initialization: The neural network parameters $\boldsymbol{\theta}^\pi$, $\boldsymbol{\theta}_1^Q$, and $\boldsymbol{\theta}_2^Q$, and the target network parameters $\bar{\boldsymbol{\theta}}^\pi \leftarrow \boldsymbol{\theta}^\pi$, $\bar{\boldsymbol{\theta}}_1^Q \leftarrow \boldsymbol{\theta}_1^Q$, and $\bar{\boldsymbol{\theta}}_2^Q \leftarrow \boldsymbol{\theta}_2^Q$
Based on the task description (and human feedback), generate a rule-based controller $\pi_{\mathrm{LLM}}$ from the LLM
while not done do
    while sample collection required do
        $\boldsymbol{a} \sim \alpha \cdot \pi_{\mathrm{LLM}}(\cdot \mid \boldsymbol{s}) + (1-\alpha) \cdot \pi_{\mathrm{LLM}}(\cdot \mid \boldsymbol{s})$ with $\alpha \sim \mathrm{Bernoulli}(p^{\mathrm{LLM}})$
        Observe the next state $\boldsymbol{s}'$ and reward $r$ and store the tuple $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', r)$ to either $\mathcal{R}_{\mathrm{LLM}}$ or $\mathcal{R}_{\mathrm{RL}}$, depending on whether $\boldsymbol{a}$ is sampled from $\pi_{\mathrm{LLM}}$ or $\pi_{\boldsymbol{\theta}^\pi}$
        $p^{\mathrm{LLM}} \leftarrow \lambda_{\mathrm{annl}} \cdot p^{\mathrm{LLM}}$
    end while
    Randomly sample $N$ data from $\mathcal{R}_{\mathrm{RL}}$ and update the critic parameters $\boldsymbol{\theta}_i^Q$, $i = 1, 2$, by minimizing $\frac{1}{N} \sum_{\{\boldsymbol{s}_n, \boldsymbol{a}_n\}_{n=1}^N \in \mathcal{R}_{\mathrm{RL}}} (y_n - Q_{\boldsymbol{\theta}_i^Q}(\boldsymbol{s}_n, \boldsymbol{a}_n))^2$, $i = 1, 2$, respectively, where $y_n \triangleq r_n + \gamma \min_{i \in \{1,2\}} Q_{\bar{\boldsymbol{\theta}}_i^Q}(\boldsymbol{s}'_n, \pi_{\bar{\boldsymbol{\theta}}^\pi}(\boldsymbol{s}'_n) + \boldsymbol{\epsilon}_n)$ with $\boldsymbol{\epsilon}_n$ being a smoothing regularization noise
    if actor update required then
        Randomly sample $N$ data from $\mathcal{R}_{\mathrm{RL}}$ and $\mathcal{R}_{\mathrm{LLM}}$, respectively, and update the actor parameter $\boldsymbol{\theta}^\pi$ by following the gradient $\frac{1}{N} \sum_{\{\boldsymbol{s}_n, \boldsymbol{a}_n\}_{n=1}^N \in \mathcal{R}_{\mathrm{RL}}} \nabla_{\boldsymbol{a}} Q_{\boldsymbol{\theta}_1^Q}(\boldsymbol{s}_n, \boldsymbol{a})\Big|_{\boldsymbol{a}=\pi_{\boldsymbol{\theta}^\pi}(\boldsymbol{s}_n)} \times \nabla_{\boldsymbol{\theta}^\pi} \pi_{\boldsymbol{\theta}^\pi}(\boldsymbol{s}_n) + \frac{2\lambda_{\mathrm{IM}}}{N} \sum_{\{\boldsymbol{s}_n, \boldsymbol{a}_n\}_{n=1}^N \in \mathcal{R}_{\mathrm{LLM}}} (\pi_{\boldsymbol{\theta}^\pi}(\boldsymbol{s}_n) - \boldsymbol{a}_n)^{\mathrm{T}} \nabla_{\boldsymbol{\theta}^\pi} \pi_{\boldsymbol{\theta}^\pi}(\boldsymbol{s}_n)$
        Update target networks $\bar{\boldsymbol{\theta}}_i^Q \leftarrow \tau \boldsymbol{\theta}_i^Q + (1-\tau) \bar{\boldsymbol{\theta}}_i^Q$, $i = 1, 2$, and $\bar{\boldsymbol{\theta}}^\pi \leftarrow \tau \boldsymbol{\theta}^\pi + (1-\tau) \bar{\boldsymbol{\theta}}^\pi$
    end if
end while
```

Liangliang Chen is with the Robotics and Autonomous Driving Lab, Baidu Research, Sunnyvale, CA 94089 USA, and also with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: liangliang.chen@gatech.edu).

Yutian Lei, Shiyu Jin, and Liangjun Zhang are with the Robotics and Autonomous Driving Lab, Baidu Research, Sunnyvale, CA 94089 USA (email: yutianlei@baidu.com, shiyujin@baidu.com, liangjunzhang@baidu.com).

Ying Zhang is with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: yzhang@gatech.edu).

APPENDIX B
EXAMPLES OF PROMPTS AND RESPONSES FOR GPT-4

A. An Example of Prompt Design With Human Feedback

The following dialogues show the prompts and responses of OpenAI GPT-4 for the task `PandaPickAndPlace` in the `panda_gym` environment.

User

Think about you are an expert who would like to finish a task with the Franka Emika Panda Robot. Please provide a step-by-step description of how the end effector should move in order to finish this task.

[Start of the General Task Description]
This task is to use the end effector of the Franka Emika Panda Robot to pick one object from one assigned location on the table and place this object to another location. The object to be moved is a cube with a side of 0.04 m.
[End of the General Task Description]

[Start of the Question]
How many phases can the end effector movement be divided into?
[End of the Question]

[Start of the Template]
Phase [NUM]: The gripper should move [CHOICE: to, around, above] [LOC].
Phase [NUM]: The gripper should be [CHOICE: open, closed].
[End of the Template]

Rules:
1. Please use the above template to answer, but do not include “[Start of the Template]” and “[End of the Template]” in your response.
2. If you see phrases like [NUM], replace the entire phrase with an integer.
3. If you see phrases like [LOC], replace the entire phrase with a location.
4. If you see phrases like [CHOICE: choice1, choice2, ...], you should replace the entire phrase with one of the choices listed.
5. If you see [OPTIONAL], it means you only add the content if necessary.
6. You do not need to ensure that the object is moved to the desired position successfully.

GPT-4

Phase 1: The gripper should move above the assigned location on the table where the object is placed.
Phase 2: The gripper should be open.
Phase 3: The gripper should move to the object.
Phase 4: The gripper should be closed around the object to ensure a secure grip.
Phase 5: The gripper should move above the initial location to clear any possible obstructions on the table.
Phase 6: The gripper should move above the new designated location on the table where the object is to be placed.
Phase 7: The gripper should move to the new location on the table.
Phase 8: The gripper should be open to release the object.
Phase 9: The gripper should move above the new location to clear the object and complete the task.

User

Can you write the detailed codes for the controller of this process? You can write the staged controller in a nested if-else statement if necessary.

The controller inputs are the following variables.
1. gripper_position (dim: 3)
2. gripper_speed (dim:3)
3. gripper_finger_distance (dim: 1)

4. object_position (dim: 3)
5. object_speed (dim: 3)
6. object_orientation (dim: 3)
7. object_rotational_speed (dim: 3)
8. target_position (dim: 3)

You must consider all the following information to make the task less ambiguous:
1. The output of the controller should be a 4-dimensional vector. The first three elements of this vector are the 3-dimensional gripper displacement and the last element is the distance changing value between the fingers.
2. The maximum moving distance of the gripper in each dimension is 0.05 m at one step.
3. When the finger distance is smaller than 0.04 m, the gripper can be thought of as closed. When the finger is larger than 0.06 m, the gripper can be thought of as open. The finger distance ranges from 0 to 0.1 m. You can try to use the largest allowable finger movement when changing its states in order to guarantee the finger state is indeed changed.
4. If you would like to check if the gripper is proximal to the object, you can use the proximal distance of 0.02 m.
5. The initial state of the gripper might be either closed or open. You should check the status and open it if necessary.
6. You do not need to clearly indicate the phases, but you may use them implicitly to write the controller codes.
7. You may use numpy when necessary.
8. You only need to write the detailed controller codes and do not need to provide examples.

GPT-4

Given the complexities involved in robotic control, it's ideal to use a state machine for managing the actions of the robot over time, however, the task asks for a nested if-else approach which can be less clear and more prone to errors. Below is a simplistic approach using if-else control structure. It's recommended to further refine and test this code in a simulated environment before attempting to use it with a real robot.

```python
import numpy as np

def controller(gripper_position, gripper_speed, gripper_finger_distance, object_position, object_speed, object_orientation, object_rotational_speed, target_position):
    # Constants
    MAX_MOVE_DISTANCE = 0.05
    PROXIMAL_DISTANCE = 0.02
    CLOSE_FINGER_DISTANCE = 0.04
    OPEN_FINGER_DISTANCE = 0.06
    FINGER_MAX_MOVE = 0.05

    # Output: [dx, dy, dz, dfinger]
    control_output = np.zeros(4)

    # Calculate the euclidean distances
    distance_to_object = np.linalg.norm(gripper_position - object_position)
    distance_to_target = np.linalg.norm(gripper_position - target_position)

    # Check the initial state of the gripper and open if necessary
    if gripper_finger_distance < OPEN_FINGER_DISTANCE:
        control_output[3] = FINGER_MAX_MOVE  # Open the gripper to the max allowable finger movement
    else:
        # If the gripper is above or near the object
        if distance_to_object <= PROXIMAL_DISTANCE:
            # If the gripper is open, close it to pick the object
            if gripper_finger_distance > CLOSE_FINGER_DISTANCE:
                control_output[3] = -FINGER_MAX_MOVE  # Close the gripper to the max allowable finger movement
            else:
                # If gripper is closed, move towards the target
                direction_to_target = (target_position - gripper_position) / distance_to_target
                move_distance = min(distance_to_target, MAX_MOVE_DISTANCE)
                control_output[:3] = direction_to_target * move_distance
        else:
            # If the gripper is not near the object, move towards the object
            direction_to_object = (object_position - gripper_position) / distance_to_object
            move_distance = min(distance_to_object, MAX_MOVE_DISTANCE)
            control_output[:3] = direction_to_object * move_distance

    # Ensure the finger movement is within the range [0, 0.1]
    control_output[3] = np.clip(control_output[3] + gripper_finger_distance, 0, 0.1) - gripper_finger_distance

    return control_output
```

In this code, we have assumed that the robot should move in a straight line towards the object and then towards the target. The 'controller' function calculates the control output based on the current state of the robot and the object. The control output is a 4D vector where the first three elements are the displacement of the gripper and the last element is the change in finger distance. This code does not handle obstacles, complex trajectories, or unexpected behaviors of the robot or the object. Moreover, this code does not handle the orientation of the gripper or the object, and assumes the object will be in the correct orientation for grasping and placing.

User

After grasping an object, you should hold it firmly and move it to the target position. Can you fine-tune the codes above by incorporating this information?

GPT-4

Certainly! In order to ensure that the gripper holds the object firmly while moving it to the target position, it is important to maintain the gripper's closed state once the object has been grasped. In the updated code below, I added a boolean variable 'object_grasped' to keep track of whether the object has been grasped or not. Once the object is grasped, the 'object_grasped' variable is set to 'True', and the gripper will maintain its closed state as it moves to the target position.

```
import numpy as np

def controller(gripper_position, gripper_speed, gripper_finger_distance, object_position, object_speed, object_orientation, object_rotational_speed, target_position):
    # Constants
    MAX_MOVE_DISTANCE = 0.05
    PROXIMAL_DISTANCE = 0.02
    CLOSE_FINGER_DISTANCE = 0.04
    OPEN_FINGER_DISTANCE = 0.06
    FINGER_MAX_MOVE = 0.05

    # Output: [dx, dy, dz, dfinger]
    control_output = np.zeros(4)

    # Calculate the euclidean distances
    distance_to_object = np.linalg.norm(gripper_position - object_position)
    distance_to_target = np.linalg.norm(gripper_position - target_position)

    # Boolean variable to check if the object has been grasped
    object_grasped = gripper_finger_distance <= CLOSE_FINGER_DISTANCE and distance_to_object <= PROXIMAL_DISTANCE

    # Check the initial state of the gripper and open if necessary
    if gripper_finger_distance < OPEN_FINGER_DISTANCE and not object_grasped:
        control_output[3] = FINGER_MAX_MOVE  # Open the gripper to the max allowable finger movement
    else:
        # If the gripper is above or near the object
        if distance_to_object <= PROXIMAL_DISTANCE and not object_grasped:
            # If the gripper is open, close it to pick the object
            if gripper_finger_distance > CLOSE_FINGER_DISTANCE:
                control_output[3] = -FINGER_MAX_MOVE  # Close the gripper to the max allowable finger movement
            else:
                object_grasped = True  # Set object_grasped to True as the gripper is now closed and near the object
        elif object_grasped:
            # If object is grasped, move towards the target
            direction_to_target = (target_position - gripper_position) / distance_to_target
            move_distance = min(distance_to_target, MAX_MOVE_DISTANCE)
            control_output[:3] = direction_to_target * move_distance
        else:
            # If the gripper is not near the object, move towards the object
            direction_to_object = (object_position - gripper_position) / distance_to_object
            move_distance = min(distance_to_object, MAX_MOVE_DISTANCE)
            control_output[:3] = direction_to_object * move_distance

    # Ensure the finger movement is within the range [0, 0.1]
    control_output[3] = np.clip(control_output[3] + gripper_finger_distance, 0, 0.1) -
    gripper_finger_distance

    return control_output
```

In this updated code, the 'object_grasped' variable is checked in the if-else control structure to decide the next actions. Once the object is grasped, the 'object_grasped' variable ensures that the gripper remains closed and moves towards the target position.

B. An Example of Prompt Design With A Code Template

The following dialogues show the prompts and responses of OpenAI GPT-4 for the task `meat_off_grill` in the `RLBench` environment. We use the code for the task `push_button` as a template and three pre-defined reusable primitive functions `is_close`, `normalize_euler_angle`, and `get_action` are called by the template.

User

Think about you are an expert who would like to finish a task with the Franka Emika Panda Robot. The controlled variables are a 7-dimensional variable of the gripper, which consists of a 3-dimensional gripper displacement, a 3-dimensional gripper orientation change value described by the Euler angle, and a 1-dimensional element indicating if the gripper is open or closed. Please provide a step-by-step description of how the end effector should move in order to finish this task.

[Start of the General Task Description]
This task is to use the end effector of the Franka Emika Panda Robot to grasp a chicken from a grill and then put to a specified location.
[End of the General Task Description]

[Start of the Question]
How many phases the end effector movement can be divided into?
[End of the Question]

[Start of the Template]
Phase [NUM]: The gripper should move [CHOICE: to, around] [LOC] [OPTIONAL: and [CHOICE: open, close] its gripper].
[End of the Template]

Rules:
1. Please use the above template to answer, but do not include "[Start of the Template]" and "[End of the Template]" in your response.
2. If you see phrases like [NUM], replace the entire phrase with an integer.
3. If you see phrases like [LOC], replace the entire phrase with a location.
4. If you see phrases like [CHOICE: choice1, choice2, ...], you should replace the entire phrase with one of the choices listed.
5. If you see [OPTIONAL], it means you only add the content if necessary.
6. You do not need to ensure that the object is moved to the desired position successfully.
7. You should not add redundant phases in your responses.

GPT-4

The end effector movement for the task described can be divided into the following phases:
Phase 1: The gripper should move to above the chicken and open its gripper.

Phase 2: The gripper should move to the chicken.
Phase 3: The gripper should close its gripper.
Phase 4: The gripper should move to above the grill with the chicken.
Phase 5: The gripper should move to the specified location.
Phase 6: The gripper should open its gripper.
Phase 7: The gripper should move away from the specified location.

User

Think about you are a Python programming expert. Can you write the detailed codes for the controller of this process?

The controller inputs include the following variables. These variables are all np.ndarray with shape (6,). The first 3 elements indicate the locations and the last 3 elements indicate Euler angles expressing the orientation.

1. 'gripper_state': The current gripper state
2. 'chicken_state': The current chicken state
3. 'target_chicken_position': The desired position the chicken should be put to
4. 'gripper_open': Indicator if the gripper is open or not

Note that you can use only these 3 variables and should not assume any other additional variances. You should use the following code template for push_button task to generate the codes for this task. For the push_button task, the goal is to push a button with the gripper.

### Start of The Code Template ###

```python
def get_action(button_state, gripper_state, position_above_button, action_limit=0.05):
    # For the task of push_button, we do not care the rotation around the z-axis since this rotation will not influence the push effects.
    # Thus, in this case, we set a euler_mask to mask out the possible rotations around the z-axis.
    euler_mask = np.array([1., 1., 0.])

    # There are two phases for the push_button task, i.e., 1) putting the gripper above the button and 2) pressing the gripper
    # The first step for the controller is to identify the current stages. In order to do this, we create two variables
    # "gripper_above_button" to indicate where the gripper is above the button and "ready_to_press" to indicate whether the gripper
    # is ready to push the button. These variables are initialized to be False's, and the values of them will be determined by
    # following conditions.
    gripper_above_button = False
    ready_to_press = False

    # is_close is function that decides if a three-dimensional np.ndarray is close to another three-dimensional np.ndarray. These
    # np.ndarray's can be either a position and a set of Euler angles.
    # You should directly use the function "normalize_euler_angle" to normalize the Euler angles since some angles may not be within
    # appropriate ranges.
    # Be sure to add the gripper_open condition in the conditions below. This may be necessary for some other tasks.
    if is_close(gripper_state[:3], position_above_button[:3]) \
        and is_close(normalize_euler_angle(gripper_state[3:] - position_above_button[3:]) * euler_mask, np.zeros(3)):
        gripper_above_button = True

    if is_close(gripper_state[:3], button_state[:3]) \
        and is_close(normalize_euler_angle(gripper_state[3:] - button_state[3:]) * euler_mask, np.zeros(3)) and gripper_open is False:
            # In order for ready_to_press to be True, we should have two prerequisite:
            # 1) The gripper should be close to the button.
            # 2) gripper_open should be True.
            ready_to_press = True

    # Based on the phase conditions, we use get_action function to generate the next actions.
    You do not need to care the meanings of
    # action and buffer_action, which are the outputs of the function get_action. The inputs of
    the function get_action include:
    # 1) cur_gripper_pose: The current gripper pose. Default to gripper_state.
    # 2) target_gripper_pose: The target gripper pose in this case, default to be target. Note
    that the target should be carefully
    # assigned for different phases.
    # 3) gripper_open: Indicator where the gripper should be open or not in the next action. You
     should decide if it should be True or False
    # 4) action_limit: The maximum value of the displacement and Euler angles' changing values
    per dimension. Default to action_limit.
    # 5) euler_mask: The mask of Euler angles' changes. Default to euler_mask.
    # NOTE: You should pay attention to the order of the conditions below. A wrong order will
    completely fail the task.
    # NOTE: You should use all the phase indicators above. For example, you should use both "
    ready_to_press" and "gripper_above_button"
    # as conditions
    if ready_to_press:
        # If ready_to_press is True, then we do not need to care if gripper_above_button is True
     or False. In this case, we can directly
        # push the button by setting target_gripper_pose to button_state.
        target = position_below_button
        action, buffer_action = get_action(cur_gripper_pose=gripper_state,
                                           target_gripper_pose=target, gripper_open=False,
                                           action_limit=action_limit, euler_mask=euler_mask)
    elif gripper_above_button:
        # If ready_to_press is False but gripper_above_button is True, then we should close the
    gripper since we should use a closed
        # gripper to push the button.
        target = button_state
        action, buffer_action = get_action(cur_gripper_pose=gripper_state,
                                           target_gripper_pose=target, gripper_open=False,
                                           action_limit=action_limit, euler_mask=euler_mask)
    else:
        # If both ready_to_press and gripper_above_button are False's, then we need to move the
    gripper to position_above_button.
        target = position_above_button
        action, buffer_action = get_action(cur_gripper_pose=gripper_state,
                                           target_gripper_pose=target, gripper_open=True,
                                           action_limit=action_limit, euler_mask=euler_mask)

    return action, buffer_action
```

### End of The Code Template ###

Here are some information you must consider:
1. When the gripper is ready to grasp an object, the gripper must be open and the gripper position should be the same as the object position.
2. For the task meat_off_grill, you do not need to care the roration of Euler angles around the x-axis. That is, you can let euler_mask to be np.array([0., 1., 1.]).
3. You can assume that the functions is_close, normalize_euler_angle, and get_action are available. In order words, you do not need to write the details of these functions.

GPT-4

Certainly! Following the code template and guidelines provided, here's how you might write the controller code for the task of moving a chicken off the grill with the Franka Emika Panda Robot. This script will assume that 'is_close', 'normalize_euler_angle', and 'get_action' are predefined functions available in the context.

```
import numpy as np

def get_action_for_chicken(chicken_state, gripper_state, target_chicken_position, action_limit
    =0.05):
    euler_mask = np.array([0., 1., 1.])  # Mask out rotation around the x-axis

    # Phase indicators
    gripper_above_chicken = False
    gripper_ready_to_grasp = False
    gripper_holding_chicken = False
    chicken_at_target = False

    # Check if the gripper is above the chicken
    if is_close(gripper_state[:3], chicken_state[:3]) and gripper_open:
        gripper_above_chicken = True

    # Check if the gripper is in the right position to grasp the chicken
    if gripper_above_chicken and is_close(normalize_euler_angle(gripper_state[3:] -
    chicken_state[3:]) * euler_mask, np.zeros(3)):
        gripper_ready_to_grasp = True

    # Check if the gripper is holding the chicken (gripper is not open and it’s close to the
    chicken state)
    if not gripper_open and is_close(gripper_state[:3], chicken_state[:3]):
        gripper_holding_chicken = True

    # Check if the chicken has been placed at the target position
    if gripper_holding_chicken and is_close(gripper_state[:3], target_chicken_position[:3]):
        chicken_at_target = True

    # Decision making based on phase indicators
    if chicken_at_target:
        # Open the gripper to release the chicken
        action, buffer_action = get_action(cur_gripper_pose=gripper_state,
                                           target_gripper_pose=target_chicken_position,
                                           gripper_open=True,
                                           action_limit=action_limit,
                                           euler_mask=euler_mask)
    elif gripper_holding_chicken:
        # Move to the target position with the chicken
        action, buffer_action = get_action(cur_gripper_pose=gripper_state,
                                           target_gripper_pose=target_chicken_position,
                                           gripper_open=False,
                                           action_limit=action_limit,
                                           euler_mask=euler_mask)
    elif gripper_ready_to_grasp:
        # Close the gripper to grasp the chicken
        action, buffer_action = get_action(cur_gripper_pose=gripper_state,
                                           target_gripper_pose=chicken_state,
                                           gripper_open=False,
                                           action_limit=action_limit,
                                           euler_mask=euler_mask)
    elif gripper_above_chicken:
        # Descend to the chicken position to be ready to grasp
        action, buffer_action = get_action(cur_gripper_pose=gripper_state,
                                           target_gripper_pose=chicken_state,
                                           gripper_open=True,
                                           action_limit=action_limit,
                                           euler_mask=euler_mask)
    else:
        # Move above the chicken
        action, buffer_action = get_action(cur_gripper_pose=gripper_state,
                                           target_gripper_pose=np.concatenate((chicken_state
    [:3], gripper_state[3:])),
                                           gripper_open=True,
                                           action_limit=action_limit,
                                           euler_mask=euler_mask)

    return action, buffer_action
```

This code defines a function that decides the next action the robot should take based on the current state of the gripper and chicken and the target position for the chicken. The function updates several boolean flags based on the current situation, then makes a decision on the next action to take. It uses the predefined ‘is_close’, ‘normalize_euler_angle’, and ‘get_action’ functions to calculate whether the gripper is in the correct position relative to the chicken and target and to generate the appropriate action commands.

The code of the primitive `get_action`, which is called by `get_action_for_chicken`, can also be generated correctly by GPT-4. The details are given as follows.

User

Think about you are an expert who would like to finish a task with the Franka Emika Panda Robot. The controlled variables are a 7-dimensional variable of the gripper, which consists of a 3-dimensional gripper displacement, a 3-dimensional gripper orientation change value described by the Euler angle, and a 1-dimensional element indicating if the gripper is open or closed.

Please write a primitive function with the name "get_action" that decides the gripper action based on the current gripper pose and target gripper pose. You should use the Python language.

This primitive function should have the following inputs.
1. cur_gripper_pose: (np.ndarray of shape (6,))
The current gripper pose with the first 3 elements being the position and the last 3 elements being the Euler angles describing the orientation
2. target_gripper_pose: (np.ndarray of shape (6,))
The target gripper pose with the first 3 elements being the target position and the last 3 elements being the Euler angles describing the target orientation
3. gripper_open: (bool)
The variable describing whether the gripper should be open or not
4. action_limit: (float, default to 0.05)
The maximum gripper displacement in each of the 3 dimensions. Note that you can multiply the this limit with 'multiplier_for_euler_angle_limit' and use the product to clip the maximum changing values of Euler angles
5. symmetric_z: (bool, default to True)
The indicator of whether the rotation around the z-axis can be regarded symmetric or not when using the normalize_euler_angle function
6. euler_mask (np.ndarray of shape (3,), default to np.ones(3))
For some tasks the rotation around some specific axes is not necessary. In this case, we can use a mask to mask out the unnecessary rotations.

The outputs of this function include the following two variables.
1. A np.ndarray of shape (8,), which should be named "action".
2. A np.ndarray of shape (7,), which should be named "buffer_action".

For the first output with the name "action", we have the following requirements.
1. The first three elements action[:3] should describe the next gripper position subject to the dimension-wise action limit with respect to the displacement.
2. The next four elements action[3:7] should describe the next gripper quaternion subject to the dimension-wise action limit with respect to the Euler angle changes. Note that this quaternion should correspond to the absolute Euler angles after changing rather than the incremental Euler angles.
3. The last element action[-1] describes whether the gripper should be open or not. If the gripper should be open, then this element should be 1, else it should be 0.

For the second output with the name "buffer_action", we have the following requirements.
1. The first three elements buffer_action[:3] should describe the gripper displacement subject to the dimension-wise action limit. However, it should be normalized to the range [-1, 1].
2. The next three elements buffer_action[3:6] should describe the Euler angle changes subject to the dimension-wise action limit. However, it should be normalized to the range [-1, 1].
3. The last element buffer_action[-1] describes whether the gripper should be open or not. If the gripper should be open, then this element should be 1, else it should be -1.

NOTE:
1. You can directly use the function "euler_to_quaternion" to convert the 3-dimensional Euler angles to a 4-dimensional quaternion. You do not need to show this function in your response.
2. You may directly use the function "normalize_euler_angle" to normalize the Euler angles since some angles

may not be within appropriate ranges. For example, some Euler angles may be larger than 2*pi rad. You should implement "normalize_euler_angle" before the dimension-wise action limit. You do not need to show this function in your response.

GPT-4

To create the 'get_action' function as described, we'll need to consider the following steps:

1. **Calculate Gripper Displacement and Euler Angle Changes:** Determine the difference between the current and target poses for both position and Euler angles.
2. **Normalize Euler Angles:** Use the 'normalize_euler_angle' function to ensure the Euler angles are within appropriate ranges, considering the 'symmetric_z' flag and 'euler_mask'.
3. **Apply Dimension-wise Action Limits:** Clip the calculated displacements and Euler angle changes within the specified limits.
4. **Convert Euler Angles to Quaternion:** Use the 'euler_to_quaternion' function to convert the adjusted Euler angles to a quaternion.
5. **Normalize for Buffer Action:** Normalize the displacement and Euler angle changes to the range [-1, 1].
6. **Assemble the 'action' and 'buffer_action' Arrays:** Construct these arrays based on the calculated values and the 'gripper_open' flag.

Here's the implementation of the 'get_action' function:

```python
import numpy as np

def get_action(cur_gripper_pose, target_gripper_pose, gripper_open, action_limit=0.05,
               symmetric_z=True, euler_mask=np.ones(3)):
    # Calculate displacement and Euler angle changes
    displacement = target_gripper_pose[:3] - cur_gripper_pose[:3]
    euler_change = target_gripper_pose[3:] - cur_gripper_pose[3:]

    # Normalize Euler angles
    normalized_euler = normalize_euler_angle(euler_change, symmetric_z)
    normalized_euler *= euler_mask

    # Apply action limits
    limited_displacement = np.clip(displacement, -action_limit, action_limit)
    euler_angle_limit = action_limit * multiplier_for_euler_angle_limit
    limited_euler_change = np.clip(normalized_euler, -euler_angle_limit, euler_angle_limit)

    # Convert normalized and limited Euler angles to quaternion
    next_euler_angles = cur_gripper_pose[3:] + limited_euler_change
    quaternion = euler_to_quaternion(next_euler_angles)

    # Normalize for buffer action
    norm_displacement = limited_displacement / action_limit
    norm_euler_change = limited_euler_change / euler_angle_limit

    # Assemble action array
    action = np.zeros(8)
    action[:3] = cur_gripper_pose[:3] + limited_displacement
    action[3:7] = quaternion
    action[-1] = 1 if gripper_open else 0

    # Assemble buffer action array
    buffer_action = np.zeros(7)
    buffer_action[:3] = norm_displacement
    buffer_action[3:6] = norm_euler_change
    buffer_action[-1] = 1 if gripper_open else -1

    return action, buffer_action
```

Note:

- The 'euler_to_quaternion' and 'normalize_euler_angle' functions are assumed to be defined elsewhere as mentioned in your requirements.
- 'multiplier_for_euler_angle_limit' should be defined or passed as a parameter if it's variable. If it's a constant, you can directly use its value in the code.

APPENDIX C
APPLICABILITY ANALYSIS OF RLINGUA AND SIMULATION TASK SELECTIONS AND DESCRIPTIONS

This appendix section provides an analysis of the practical applicability of the proposed algorithm, RLingua, detailing scenarios where it is effective and where it may not be suitable. Additionally, The criteria used for selecting tasks in `RLBench` are also illustrated.

A. *Applicability Analysis of RLingua*

RLingua leverages the prior knowledge of LLMs and RL to obtain robot controllers with high performance. The success of the robot controller depends on the following two aspects.

1) The capacity of the current LLM (specifically, GPT-4 in our study) to generate a rule-based controller endowed with accurate control logic.
2) The robot task should not be brittle to RL explorations.

With the task examples in `RLBench`, we illustrate these two aspects in detail as follows.

1) RLingua may fail some tasks due to the currently limited ability of GPT-4. Specifically, GPT-4's failure to generate a rule-based controller with correct control logic can be attributed primarily to the following reasons.
    i) GPT-4 is currently unable to generate gripper paths that include arcs. Instead, it is limited to producing paths with polyline shapes, lacking the capability to handle the more complex calculations required for arcs. This limitation becomes apparent in certain tasks within `RLBench`, such as `close_door` and `close_fridge`, where an arc segment in the gripper's trajectory is necessary. Consequently, for these tasks, GPT-4 fails to generate a rule-based controller capable of providing reliable guidance for RL.
    ii) GPT-4 struggles with robot tasks that feature long horizons. The Python-written rule-based controllers it generates rely heavily on multiple, and often nested, `if-else` statements to discern the current task state and determine subsequent control actions. When a task encompasses an extensive horizon with numerous stages, GPT-4's capability to produce a controller with accurate logic is compromised, as it requires the correctness of *all* `if-else` statements. Tasks from `RLBench` falling into this category include, but are not limited to, `block_pyramid`, `stack_blocks`, and `wipe_desk`.
    iii) The controller generated by GPT-4 may sometimes fail to identify the correct grasp pose for specific tasks. In tasks within `RLBench`, such as `tv_on` and `change_channel`, the robot gripper must accurately find a correct grasp pose. During our experiments, we observed instances where the GPT-4 generated controller was unable to reliably grasp an object in the correct pose. For instance, given the symmetric nature of the gripper, grasping an object at joint angles of $90°$ and $-90°$ should be equivalent. However, the current GPT-4 model struggles to effectively integrate this crucial piece of information into the controller code generation process. This oversight can lead to violations of the joint rotation constraints for the Franka Emika Panda robot, resulting in task failure. Consequently, these failures indicate that GPT-4 is unable to generate a controller with the logical reliability necessary to guide reinforcement learning exploration effectively.
2) RLingua is not ideally suited for tasks that are particularly vulnerable to the dynamics of RL exploration.
Effective exploration is crucial for achieving optimal performance in RL. Nonetheless, certain tasks within `RLBench` exhibit a high sensitivity to these exploration processes. Take, for instance, the `stack_cups` task, which demands that the robot sequentially pick two cups and stack them into a third one. For successful execution, the gripper must grasp each cup by its brim. However, during RL exploration phases, a cup may be inadvertently released at any point post-grasp, leading to a high likelihood of the cup tipping over. Subsequent attempts to re-grasp a tipped cup significantly compound the challenge, rendering the task exceedingly difficult due to the sparsity of successful outcomes.

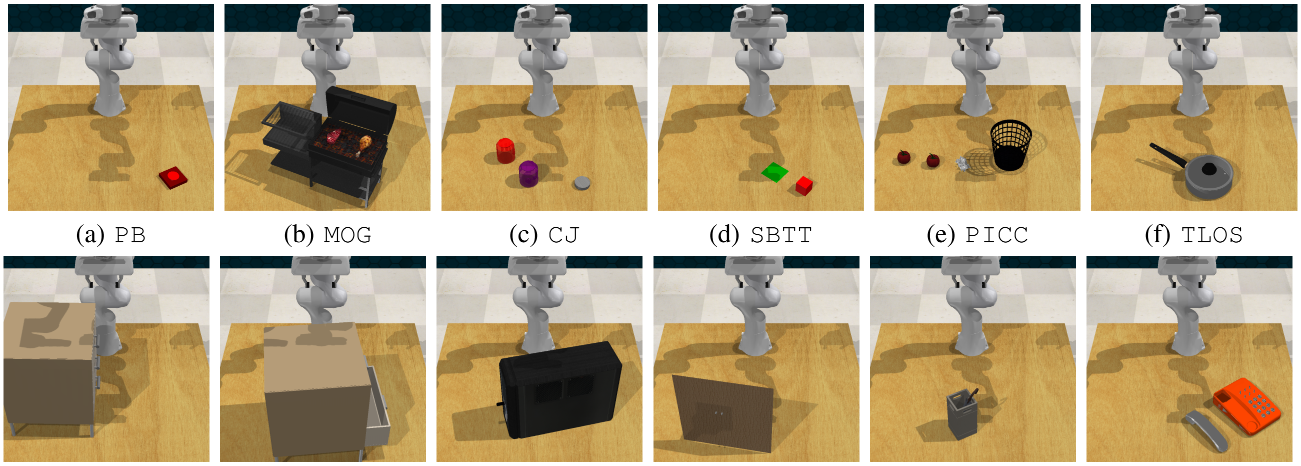
In summary, the effectiveness of RLingua hinges on two primary factors: the capacity of current LLMs to generate rule-based robot controllers, and the susceptibility of tasks to the challenges posed by RL exploration. Regarding the first factor, there is optimism that LLM capabilities will see substantial advancements shortly, thereby extending RLingua's applicability to more complex tasks. As for the task's brittleness to RL explorations, leveraging LLMs' prior knowledge to minimize unnecessary explorations and circumvent actions leading to unrecoverable states presents a viable strategy. An illustrative example is the known prerequisite in the `stack_cups` task within `RLBench`, where a cup should not be released post-grasp until it is positioned above the target. By integrating such knowledge, we can programmatically restrict the gripper from releasing the cup during transit, thereby eliminating certain exploration actions. Looking ahead, our future work aims to refine the prompt design for LLMs, enabling the extraction of such task-specific knowledge directly from LLMs rather than relying on human input. This ambition points towards a deeper integration of LLMs with RL, a direction that promises exciting developments and is earmarked for subsequent investigation.

B. *Simulation Task Selections and Descriptions*

This section outlines the criteria used for selecting tasks in our simulation experiments and provides detailed introductions to the specific robot tasks in our experiments.

TABLE A-I: The classification of robot tasks in `RLBench`

Category	Tasks
Tasks considered by RLingua	`basketball_in_hoop`, **`close_drawer`**, **`close_jar`**, `close_microwave`, `insert_onto_square_peg`, `lamp_off`, `lamp_on`, `lift_numbered_block`, `light_bulb_in`, `light_bulb_out`, **`meat_off_grill`**, `meat_on_grill`, **`open_drawer`**, `open_jar`, `open_wine_bottle`, `pick_and_lift`, `pick_and_lift_small`, `pick_up_cup`, **`phone_on_base`**, `place_shape_in_shape_sorter`, `play_jenga`, `press_switch`, **`push_button`**, `push_buttons`, **`put_item_in_container`**, `reach_target`, **`slide_block_to_target`**, `solve_puzzle`, **`take_lid_off_saucepan`**, `take_off_weighing_scales`, `take_toilet_roll_off_stand`, **`take_umbrella_out_of_umbrella_stand`**, **`take_usb_out_of_computer`**, **`unplug_charger`**
An arc in the gripper path	`close_door`, `close_fridge`, `close_grill`, `close_laptop_lid`, `close_box`, `empty_dishwasher`, `open_box`, `open_door`, `open_fridge`, `open_grill`, `open_microwave`, `open_oven`, `open_washing_machine`, `open_window`, `put_shoes_in_box`, `put_tray_in_oven`, `take_shoes_out_of_box`, `take_tray_out_of_oven`, `toilet_seat_down`, `toilet_seat_up`, `turn_oven_on`
Too long task horizon for LLMs	`block_pyramid`, `empty_container`, `empty_dishwasher`, `get_ice_from_fridge`, `open_window`, `put_item_in_drawer`, `stack_blocks`, `straighten_rope`, `take_item_out_of_drawer`, `wipe_desk`
Correct grasp pose may not be found	`change_channel`, `change_clock`, `hang_frame_on_hanger`, `put_knife_in_knife_block`, `put_knife_on_chopping_board`, `screw_nail`, `slide_cabinet_open`, `slide_cabinet_open_and_place_cups`, `stack_wine`, `take_cup_out_from_cabinet`, `turn_tap`, `tv_on`
Hard for LLMs due to other reasons	`beat_the_buzz`, `scoop_with_spatula`
Brittle to explorations in RL	`hit_ball_with_queue`, `hockey`, `insert_usb_in_computer`, `move_hanger`, `place_cups`, `place_hanger_on_rack`, `plug_charger_in_power_supply`, `pour_from_cup_to_cup`, `place_all_groceries_in_cupboard`, `put_books_on_bookshelf`, `put_bottle_in_fridge`, `put_groceries_in_cupboard`, `put_money_in_safe`, `put_place_in_colored_dish_rack`, `put_toilet_roll_on_stand`, `put_umbrella_in_umbrella_stand`, `reach_and_drag`, `remove_cups`, `set_the_table`, `setup_checkers`, `setup_chess`, `stack_chairs`, `stack_cups`, `sweep_to_dustpan`, `take_frame_off_hanger`, `take_money_out_safe`, `take_plate_off_colored_dish_rack`, `water_plants`, `weighing_scales`



(a) `PB` (b) `MOG` (c) `CJ` (d) `SBTT` (e) `PICC` (f) `TLOS`

(g) `OD` (h) `CD` (i) `TUOOC` (j) `UC` (k) `TUOOUS` (l) `POB`

Fig. A-1: The tasks in the `RLBench` environment: (a) `push_button` (`PB`); (b) `meat_off_grill` (`MOG`); (c) `close_jar` (`CJ`); (d) `slide_block_to_target` (`SBTT`); (e) `put_item_in_container` (`PIIC`); (f) `take_lid_off_saucepan` (`TLOS`); (g) `open_drawer` (`OD`); (h) `close_drawer` (`CD`); (i) `take_usb_out_of_computer` (`TUOOC`); (j) `unplug_charger` (`UC`); (k) `take_umbrella_out_of_umbrella_stand` (`TUOOUS`); (l) `phone_on_base` (`POB`)

1) `RLBench` Simulation Task Selections and Descriptions: Based on the applicability analysis in Section C-A, we classify the 108 robot tasks in `RLBench` [1] into different categories, as shown in TABLE A-I. There are 34 tasks that are considered for RLingua and we randomly sample 12 tasks out of them in the experiment in Section IV-B. To guarantee task diversity, we consider the following criteria when sampling the tasks.

- Reach a target (1 task): `push_button`
- Pick and lift (2 tasks): `take_lid_off_saucepan`, `take_umbrella_out_of_umbrella_stand`
- The task requiring dynamic interactions (1 task): `slide_block_to_target`
- Pick and place (4 tasks): `close_jar`, `meat_off_grill`, `phone_on_base`, `put_item_in_container`
- The tasks requiring complex gripper rotations (4 tasks): `close_drawer`, `open_drawer`, `take_usb_out_of_computer`, `unplug_charger`

These 12 `RLBench` tasks, visualized in Fig. A-1, are introduced as follows.

(a) `push_button`: The robot needs to push a button with its gripper.

(b) `meat_off_grill`: The robot needs to move a chicken off the grill.

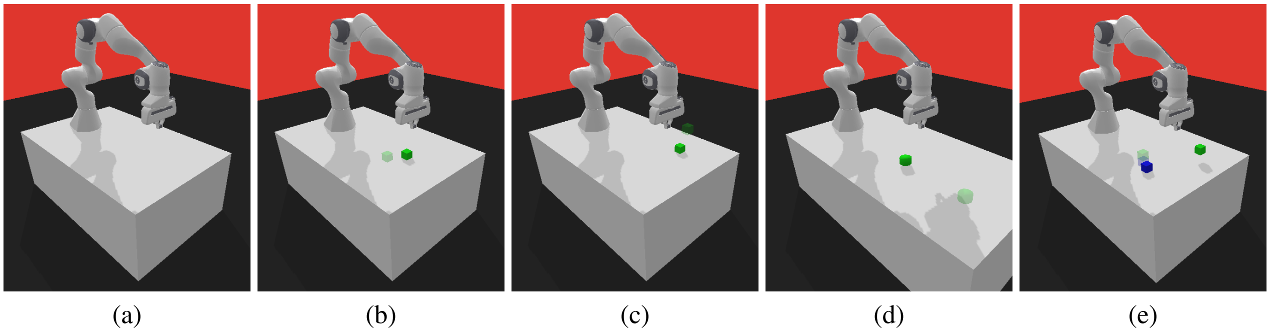

(a) (b) (c) (d) (e)

Fig. A-2: The five tasks in the `panda_gym` environment: (a) `PandaReach`; (b) `PandaPush`; (c) `PandaPickAndPlace`; (d) `PandaSlide`; (e) `PandaStack` (dark color: cube(s); shaded color: the target position(s))

(c) `close_jar`: The robot needs to close a jar with a lid.
(d) `slide_block_to_target`: The robot needs to slide a block to a target position. Compared with `PandaSlide` in `panda_gym`, the friction between the block and table in the `RLBench` environment is larger so that the robot gripper may need to push the block multiple times in order to complete the task successfully.
(e) `put_item_in_container`: The robot needs to put an item in a container with its gripper.
(f) `take_lid_off_saucepan`: The robot needs to take a lid off a saucepan.
(g) `open_drawer`: The robot needs to open a drawer from the handle with its gripper.
(h) `close_drawer`: The robot needs to close a drawer with its gripper.
(i) `take_usb_out_of_computer`: The robot needs to unplug a USB from a computer.
(j) `unplug_charger`: The robot needs to unplug a charger and place it in a certain location with its gripper.
(k) `take_umbrella_out_of_umbrella_stand`: The robot needs to take an umbrella out of an umbrella stand by lifting the umbrella.
(l) `phone_on_base`: The robot needs to pick a phone and place it on a phone base.

2) *`panda_gym` Task Descriptions:*

Five tasks in `panda_gym` [2] are considered in the simulation experiments of this paper.

(a) `PandaReach`: The robot gripper needs to reach a point in the space.
(b) `PandaPush`: The gripper needs to push a cube from one location to another location *within* the reachable region of the gripper.
(c) `PandaPickAndPlace`: The robot needs to pick a cube from one location and place it to another with its gripper.
(d) `PandaSlide`: The gripper needs to slide a cube from one location to another location *beyond* the reachable region of the gripper.
(e) `PandaStack`: The gripper needs to stack two cubes by picking and placing them to assigned locations.

APPENDIX D
EXPERIMENT HYPERPARAMETERS

TABLE A-II shows the hyperparameters of the RLingua with TD3 for the tasks in both `panda_gym` and `RLBench` environments. The meanings of the notations are given in Algorithm A-1. We use the same values of γ for the standard TD3 algorithm in consideration of fair comparisons. The other hyperparameters not listed in TABLE A-II are the same as those in `stable_baselines3` (See https://github.com/DLR-RM/stable-baselines3).

TABLE A-II: The experiment hyperparameters

Environment	γ	p^{LLM}	λ_{annl}	λ_{IM}
`panda_gym`	0.975	0.25	0.99995	1.0
`RLBench`	0.96	0.25	0.999999	1.0

APPENDIX E
DISCUSSIONS

This paper introduces the RLingua framework, which utilizes large language models (LLMs) to assist in reinforcement learning (RL). The main storyline of this paper is that the extracted rule-based controllers from LLMs are helpful in reducing the sample complexity of RL. We take the initial step in this direction, demonstrating the promising aspects of RLingua in various robot tasks. With the rapid development of LLMs, we believe that future LLMs can be prompted to generate reasonable rule-based controllers for more complicated robot tasks, for which the framework of RLingua can also be used to improve the sample efficiency of RL. On the other storyline that we did not emphasize in this paper, we have to admit that there are always some complicated robot tasks that cannot be solved perfectly by only prompting LLMs, even if we hold an optimistic attitude about the rapid development of LLMs. In this perspective, RLingua provides a method to improve the imperfect LLM-generated controllers by RL. We hope that our work in this paper can motivate more relevant research in both storylines.

In the scope of this paper, we assume that the Franka Emika Panda robot can only implement some basic movements, such as displacements and rotations in a small region relative to the current gripper position and orientation. Note that the learned robot primitives in this work can be further extended to more high-level robot tasks [3], [4], [5], [6], [7]. In addition, the framework of RLingua can be combined with the existing methods that use LLMs to assist the reward design [8], [9], [10], [11]. Since both reward design and data efficiency are key practical challenges of RL, this work steps forward to obtain more practical RL agents by leveraging LLMs.

Furthermore, the LLM prompt design described in Section III-A aims to incorporate the prompt sub-parts requiring expert knowledge to a fixed template and leave the generated task descriptions and human feedback, which can be finished by non-experts, to the users. In this sense, we take a step further to achieve a personalized robot controller that can be obtained without expert knowledge.

We would like to note that the framework of RLingua in this paper needs an external perception to extract the state information of robot tasks. The future research direction, stimulated by the development and increased affordability of multi-modal LLMs [12], involves developing an end-to-end RLingua-like framework.

REFERENCES

[1] S. James, Z. Ma, D. R. Arrojo, and A. J. Davison, "Rlbench: The robot learning benchmark & learning environment," *IEEE Robot. Automat. Lett.*, vol. 5, no. 2, pp. 3019–3026, 2020.

[2] Q. Gallouédec, N. Cazin, E. Dellandréa, and L. Chen, "panda-gym: Open-source goal-conditioned environments for robotic learning," *arXiv preprint arXiv:2106.13687*, 2021.

[3] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch, "Language models as zero-shot planners: Extracting actionable knowledge for embodied agents," in *Proc. Int. Conf. Mach. Learn.*, 2022, pp. 9118–9147.

[4] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman *et al.*, "Do as i can, not as i say: Grounding language in robotic affordances," in *Proc. Conf. Robot Learn.*, 2022.

[5] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar *et al.*, "Inner monologue: Embodied reasoning through planning with language models," in *Proc. Conf. Robot Learn.*, 2022.

[6] J. Zhang, J. Zhang, K. Pertsch, Z. Liu, X. Ren, M. Chang, S.-H. Sun, and J. J. Lim, "Bootstrap your own skills: Learning to solve new tasks with large language model guidance," in *Proc. Conf. Robot Learn.*, 2023.

[7] N. Wake, A. Kanehira, K. Sasabuchi, J. Takamatsu, and K. Ikeuchi, "Chatgpt empowered long-step robot control in various environments: A case application," *IEEE Access*, vol. 11, pp. 95 060–95 078, 2023.

[8] M. Kwon, S. M. Xie, K. Bullard, and D. Sadigh, "Reward design with language models," in *Proc. Int. Conf. Learn. Representations*, 2023, pp. 1–18.

[9] W. Yu, N. Gileadi, C. Fu, S. Kirmani, K.-H. Lee, M. G. Arenas, H.-T. L. Chiang, T. Erez, L. Hasenclever, J. Humplik *et al.*, "Language to rewards for robotic skill synthesis," in *Proc. Conf. Robot Learn.*, 2023.

[10] T. Xie, S. Zhao, C. H. Wu, Y. Liu, Q. Luo, V. Zhong, Y. Yang, and T. Yu, "Text2reward: Automated dense reward function generation for reinforcement learning," *arXiv preprint arXiv:2309.11489*, 2023.

[11] Y. J. Ma, W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jayaraman, Y. Zhu, L. Fan, and A. Anandkumar, "Eureka: Human-level reward design via coding large language models," *arXiv preprint arXiv:2310.12931*, 2023.

[12] Z. Yang, L. Li, K. Lin, J. Wang, C.-C. Lin, Z. Liu, and L. Wang, "The dawn of lmms: Preliminary explorations with gpt-4v (ision)," *arXiv preprint arXiv:2309.17421*, vol. 9, p. 1, 2023.