POLITECNICO
DI MILANO

Machine Learning for Mechanical Systems

Large Language Model for high level robot control

Authors:
Enrico Bregni
Francesco Pisacane
Michele Taboni

Date:
July 10, 2025

Contents

1 Introduction

The following reports deals with the use of Large Language Models for high level robot control. This implementation strategy leverages the capability of an LLM to understand human language and uses it to translate sentences into code lines that can then be used in a more traditional robot control framework.

1.1 Large Language Model

LLMs (Large Language Models) are AI models trained on massive text data to understand and generate human-like language.They predict the next word in a sequence, enabling them to answer questions, write code, summarize text, and much more, all by learning patterns in language.

When implemented in an API format, the LLM it is usually addressed with 2 inputs:

- **Prompt:** it is used to set the behavior of the LLM, how it should answer and can seen as a set of rules. Can also include examples in order for the LLM to have additional context on how do behave in specific scenarios.
- **Query:** It is the user request, usually a question or command.

1.2 Zero-Shot LLMs as High-Level Planners

From the previous definition, it is clear that LLMs are designed to be general purpose. In specific fields, such as robotics, it would be ideal to have a LLM specifically trained for the control purpose. However, due to the huge amount of resources necessary for training, this is often not possible for medium to small companies and research groups.

Starting from a pre-trained model, it is possible to guide the LLM behavior by means of careful design of the prompt and/or query. This is often referred as "Prompt engineering".

2 Say-Can model

Say-Can is a framework from Google DeepMind that combines a language model (Say) with a robotic skills model (Can) to let robots understand high-level natural language commands and figure out what they can actually do.

It is able to integrate the use of an LLM with actuation models in order to allow robot control directly from human language.

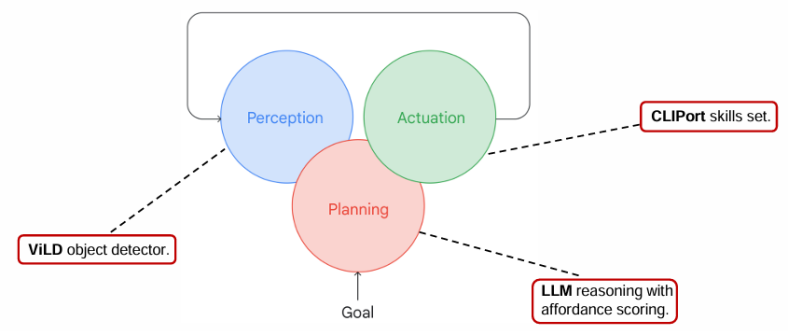

Figure 1: Basic schematic of SayCan implementation

2.1 LLM (Say)

"Say" refers to the reasoning part of the model and its implemented using a Large Language Model. Given a user request, the LLM will output a series or actions. Each action is associated to a score which reflect how much the LLM "thinks" that action satisfies the user request. based only on the lexical interpretation of the query.

2.2 Affordance (Can)

"Can" refers to the actuation model however, in the context of robotics, this includes also perception. Perception and actuation models combined allow to define an "Affordance score", which is a score that evaluates how possible it is to perform the action. Often this translates to binary condition as either the action is possible or it is not.

2.3 Combination Say+Can

The "Say+Can" combination integrates the high-level reasoning capability of the LLM ("Say") with the feasibility assessment by the perception/actuation model ("Can").

"Say" gives the a ranking of actions scored in terms of logical consequentiality with the request while "Can" outputs the action scored in terms of feasibility (affordance).

These two scores (Say and Affordance) are combined (typically multiplied) to yield a final combined score. The action with the highest combined score is then selected for execution. This approach enables the robot to choose actions that are both semantically appropriate to the user's instruction and physically realizable in its environment.

3 LLM for high level robot control

The following scheme shows a possible implementation of the LLM as high level planner. In this implementation the LLM has an affordance feedback from the actuation model however it is not aware of the environment.

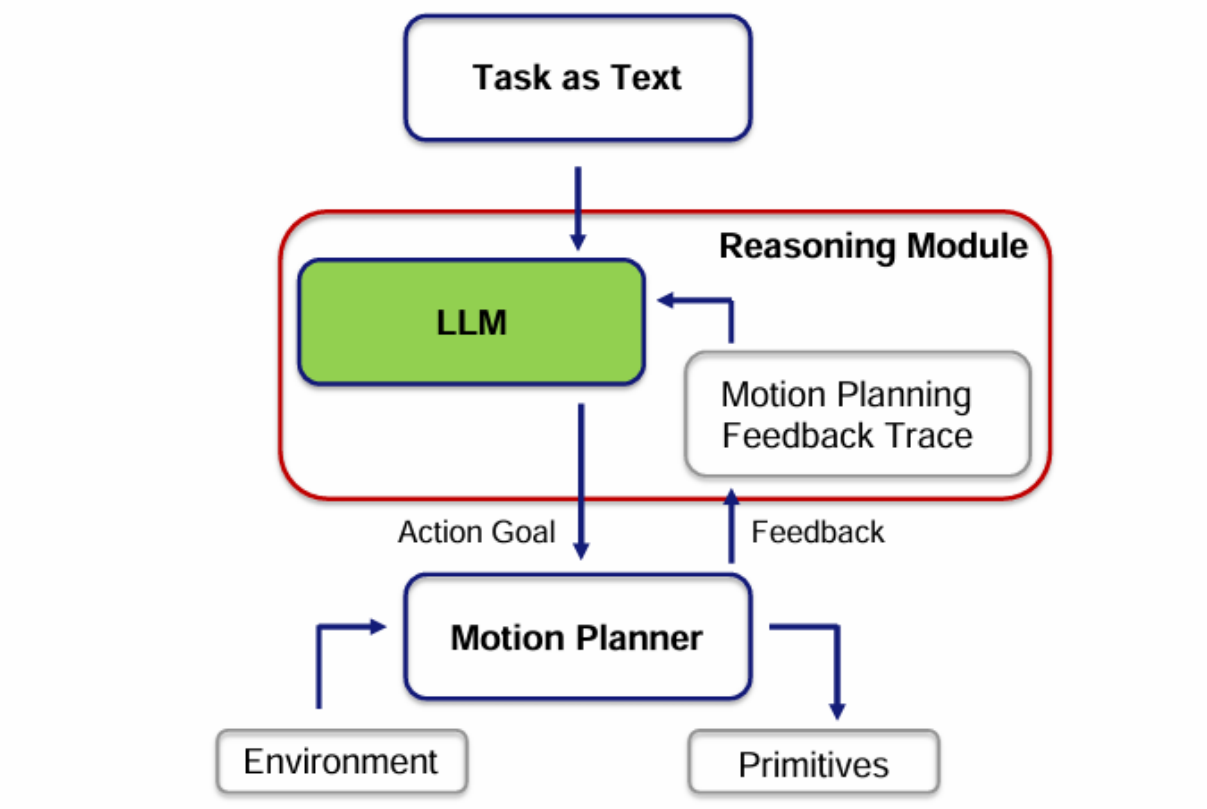

Figure 2: Scheme - LLM as planner

To allow a general purpose LLM to perform such actions, as previously mentioned, the zero-shot approach is used with prompt engineering. The basic prompt used is as follows:

- "After `\\\\'USER INPUT\\\\'` there will be a request to satisfy and there may already be actions trying to satisfy it.
- You must choose from the available options which one you think is next given the previous ones.
- Return only the option you have chosen, without adding or removing anything else!
- IMPORTANT: Answer `\\'done()\\'` when you think the request has been satisfied."

It has the form of a list of rules, however this is not sufficient to obtained the desired result as no enforcing of the format of the output has been done. This could be added with additional rules or with examples from which the LLM can understand how to behave, this has the added bonus of increase robustness.

- ```
  # move all the blocks to the top left corner.
  robot.pick_and_place(blue block, top left corner)
  robot.pick_and_place(red block, top left corner)
  robot.pick_and_place(yellow block, top left corner)
  done()
  ```
- ```
  # put the yellow one the green thing.
  robot.pick_and_place(yellow block, green bowl)
  done()
  ```
- ```
  # move the light colored block to the middle.
  robot.pick_and_place(yellow block, middle)
  done()
  ```
- .....

From the examples the LLM can understand how to behave in some situations and also learns that its output should be a line of code, with a specific format.

In the following report, a similar scheme is implemented to generate the plan for a pick and place robot that manipulates basic object like blocks and bowls. All the following depictions regarding the robot and environment (black plane) are obtained by means PyBullet physics simulations.
  ```

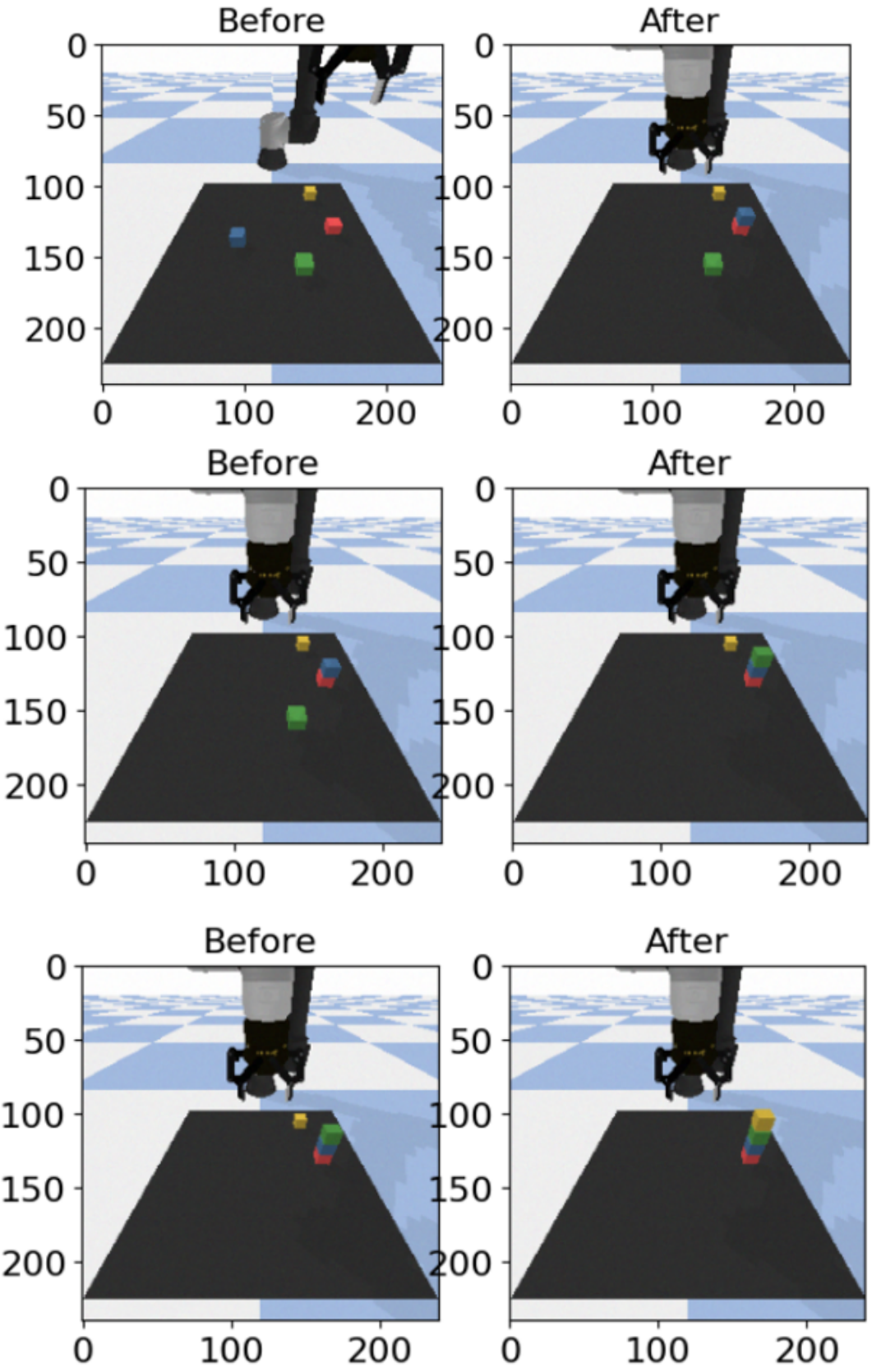

Figure 3: Outcome (Task: "put blue block on red block, then green block on the blue one, then the yellow block on the green one.")

3.1 Visual feedback to LLM

The performance of the LLM can be augmented by granting it access to information about the environment. In robotics this is often done with vision. The LLM is allowed to "see" by using a vision detection model and converting the scene into a text description that can be added to the context of the Large Language Model.

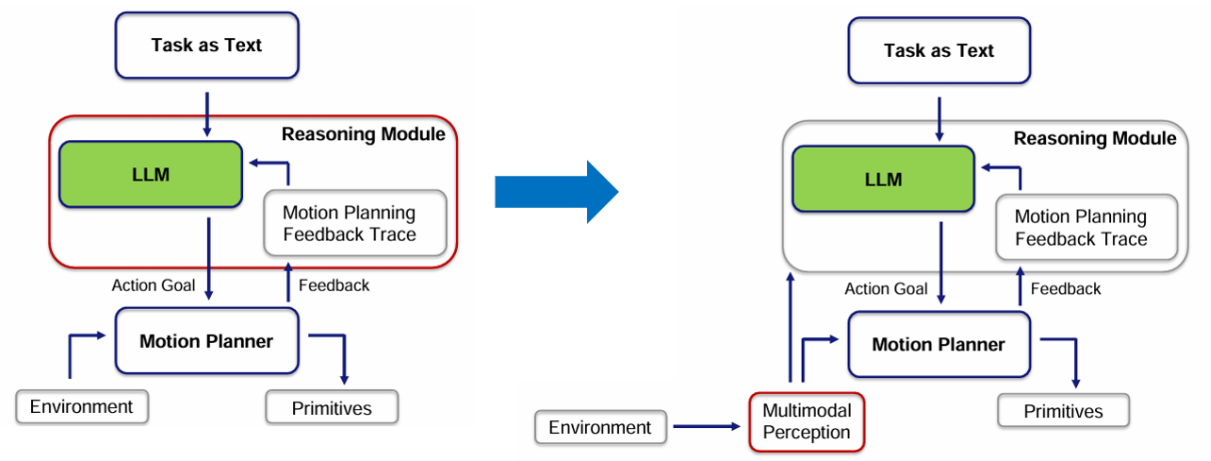

Figure 4: Scheme - LLM as planner with vision

To the prompt, the following line is added:

- You have available also a scene description with the objects observed pickable in the scene, this description is defined within `\\\\'BEGIN SCENE DESCRIPTION\\\\'` through `\\\\'END SCENE DESCRIPTION\\\\'`.

moreover the example now also include a description of the scene in the form of a list of objects:

- ...
- ```
  objects = [red block, yellow block, blue block, green bowl]
  # put the yellow one the green thing.
  robot.pick_and_place(yellow block, green bowl)
  done()
  ```
- ```
  objects = [yellow block, blue block, red block]
  # move the light colored block to the middle.
  robot.pick_and_place(yellow block, middle)
  done()
  ```
- .....

In the following example, the objects present in the scene are (Red Block, Green Block, Blue Block, Yellow Block).

Figure 5: Actions scoring comparison with and without visual feedback

In practical terms, providing a vision feedback to the LLM makes it more "confident" in its proposals. This can be observed as the "LLMSCORE" for the same action is sensibly higher when the LLM has a visual description of the scene in its context.

Furthermore, it can be observed that when visual feedback is available, the LLM assigns low scores to proposals involving objects that are not present in the scene. Conversely, if the feedback it is not implemented the scores are only based on the lexical interpretation done by the LLM, thus the scores are higher.

```
raw_input = "put blue block on red block."
```

With vision feedback in Context

```
Selecting: robot.pick_and_place(blue block, red block)
OPTION:  robot.pick_and_place(red block, blue sphere)
LLMSCORE:  5.299511174334049e-08
AFF SCORE:  0
Selecting: robot.pick_and_place(blue block, red block)
OPTION:  robot.pick_and_place(red block, blue bowl)
LLMSCORE:  5.299511174334049e-08
AFF SCORE:  0
Selecting: robot.pick_and_place(blue block, red block)
OPTION:  robot.pick_and_place(red block, red bowl)
LLMSCORE:  5.299511174334049e-08
AFF SCORE:  0
Selecting: robot.pick_and_place(blue block, red block)
OPTION:  robot.pick_and_place(red block, green bowl)
LLMSCORE:  5.299511174334049e-08
AFF SCORE:  0
Selecting: robot.pick_and_place(blue block, red block)
OPTION:  robot.pick_and_place(red block, yellow bowl)
LLMSCORE:  5.299511174334049e-08
AFF SCORE:  0
Selecting: robot.pick_and_place(blue block, red block)
OPTION:  robot.pick_and_place(red block, orange bowl)
LLMSCORE:  5.299511174334049e-08
AFF SCORE:  0
```

Without vision feedback in Context

```
Selecting: robot.pick_and_place(blue block, red block)
OPTION:  robot.pick_and_place(blue sphere, blue sphere)
LLMSCORE:  0.5713599040170627
AFF SCORE:  0
Selecting: robot.pick_and_place(blue block, red block)
OPTION:  robot.pick_and_place(blue sphere, blue bowl)
LLMSCORE:  0.5713599040170627
AFF SCORE:  0
Selecting: robot.pick_and_place(blue block, red block)
OPTION:  robot.pick_and_place(blue sphere, red bowl)
LLMSCORE:  0.615569417195843
AFF SCORE:  0
Selecting: robot.pick_and_place(blue block, red block)
OPTION:  robot.pick_and_place(blue sphere, green bowl)
LLMSCORE:  0.5682364058033446
AFF SCORE:  0
Selecting: robot.pick_and_place(blue block, red block)
OPTION:  robot.pick_and_place(blue sphere, yellow bowl)
LLMSCORE:  0.5694377512701593
AFF SCORE:  0
Selecting: robot.pick_and_place(blue block, red block)
OPTION:  robot.pick_and_place(blue sphere, orange bowl)
LLMSCORE:  0.42335414250549397
AFF SCORE:  0
Selecting: robot.pick_and_place(blue block, red block)
```

Figure 6: Other actions scoring comparison with and without visual feedback

Specifically, in cases where an attempt is made to pick or place an object that is absent from the scene, the scores with visual feedback are all extremely low and approximately equal. Without vision feedback, the scores for absent objects are considerably higher as the LLM is unaware of them missing.

3.2 Performance in complex environment

In order to test the performance of the vision system and the LLM, a more complex environment can be generated by adding different objects and changing the colors. Moreover, different names to refer to the blocks and bowls (respectively "box" and "circles") have been used. In the following example, it is possible to observe a successful execution of the task, despite the increased complexity and the use of alternative object names.

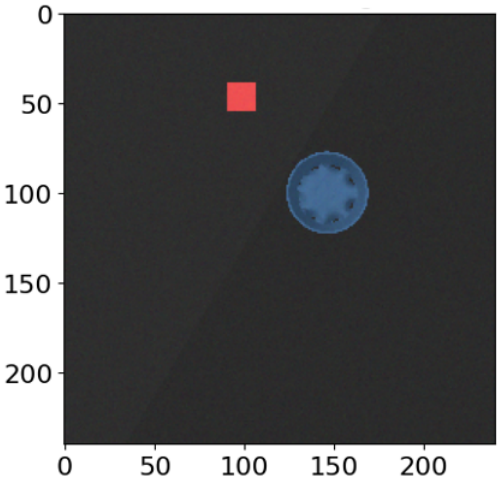

Figure 7: Initial environment

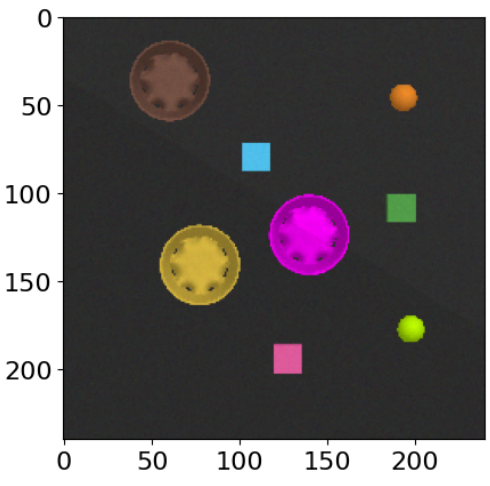

Figure 8: Complex environment

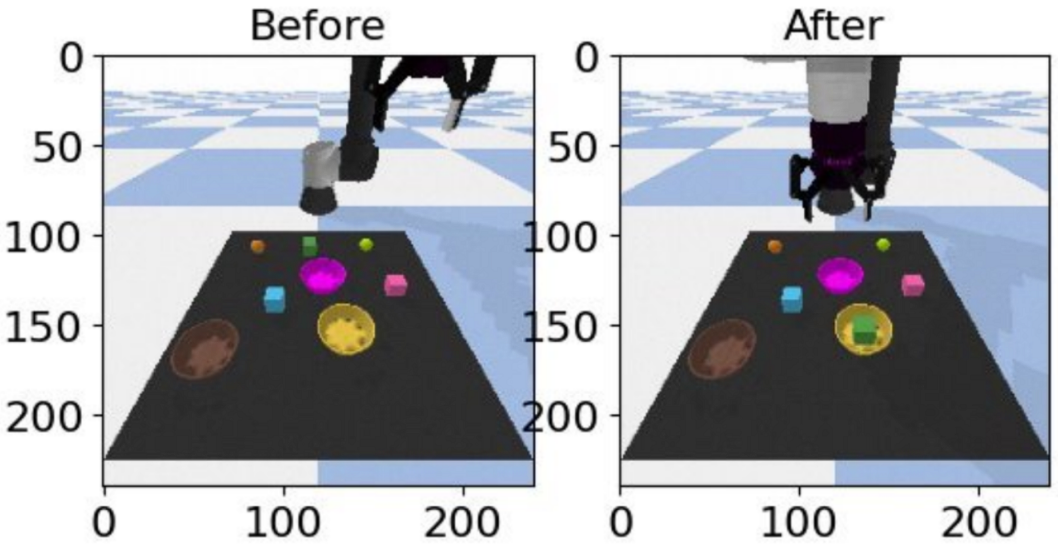

Figure 9: Outcome (task: "put the green box in the yellow circle")

3.3 User feedback to LLM

To enhance the robustness and flexibility of the high-level planning system, a mechanism for user feedback can be implemented. This feature allows the user to modify, refine, or reject a proposed plan before the robot executes it. Hence, the planner is introduced into a Human-Machine feedback loop.

After generating a plan in response to a user command, the system displays the suggested action and awaits user input. The user is then asked to either:

- approve the proposed plan
- provide textual feedback to request a change, correction or rejection of the plan.

The language model reprocesses the plan by considering the user's modification. This is implementing by generating a **new query** with a specific format which include the previous plan and the suggested modifications. This effectively allows the LLM to replan dynamically, with a human-in-the-loop strategy for adaptive robot behavior. The new query is generated as follows:

- `new_query = ('BEGIN PREVIOUS WRONG OUTPUT: ' + previous_plan + 'END PREVIOUS WRONG OUTPUT.' + 'MODIFICATIONS REQUESTS:' + user_feedback )`

Other than a fundamental restructuring of the code to allow for the Human-Machine feedback loop, also several changes in the prompt are necessary. Starting with the addition of the following lines:

- "IMPORTANT: IF after `\\\\'USER INPUT\\\\'` there is `\\\\'BEGIN PREVIOUS WRONG OUTPUT:\\\\'`, propose a change of the plan contained between `\\\\'BEGIN PREVIOUS WRONG OUTPUT:\\\\'` and `\\\\'END PREVIOUS WRONG OUTPUT.\\\\'` according to the request after `\\\\'MODIFICATIONS REQUESTS:\\\\'`,
- IMPORTANT: only apply the required changes, nothing else.
- Once you are done answer `\\'done()\\'`.

Again additional example can be used in order to enrich the context of the LLM:

- ```
  objects = [red block, yellow block, blue block, green block]
  # BEGIN PREVIOUS WRONG OUTPUT:
  robot.pick_and_place(yellow block, top right corner)
  robot.pick_and_place(red block, bottom right corner)
  robot.pick_and_place(yellow block, top right corner)
  robot.pick_and_place(red block, bottom right corner)
  done()
  END PREVIOUS WRONG OUTPUT.
  MODIFICATIONS REQUESTS:
  switch step 1 and step 2
  robot.pick_and_place(yellow block, top right corner)
  robot.pick_and_place(yellow block, top right corner)
  robot.pick_and_place(red block, bottom right corner)
  robot.pick_and_place(red block, bottom right corner)
  done()
  ```
- .....
  ```

Following are some examples:

```
**** Proposed Plan ****
1. Pick the blue block and place it on the red block.
2. Task completed.

Approve this plan? (Type 'approve' to accept, or describe changes): switch the blue block with the red one
Replanning based on feedback: switch the blue block with the red one
QUERY:  BEGIN PREVIOUS OUTPUT:
robot.pick_and_place(blue block, red block)
done()
END PREVIOUS OUTPUT.

MODIFICATIONS REQUESTS.
switch the blue block with the red one


**** Proposed Plan ****
1. Pick the red block and place it on the blue block.
2. Task completed.

Approve this plan? (Type 'approve' to accept, or describe changes): yes
Plan approved. Proceeding to actuation.
```

Figure 10: User feedback example 1

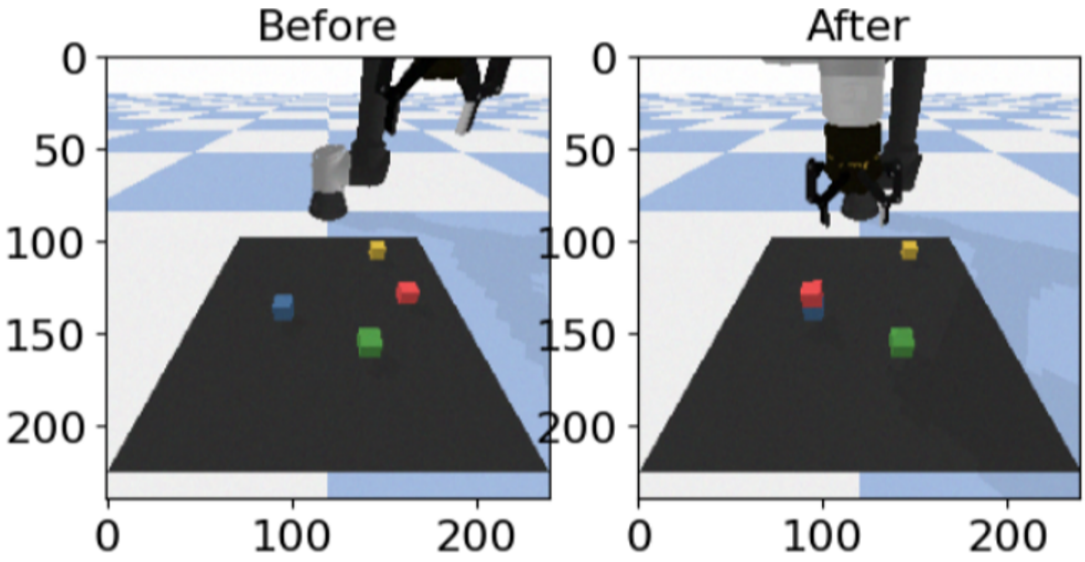

Figure 11: Outcome example 1

```
**** Solution ****
BEGIN SCENE DESCRIPTION:
objects = [red block, green block, blue block, blue sphere, red sphere, yellow block]
END SCENE DESCRIPTION.

# put the green block in the center, then put the red block on the blue block.
Step 0: robot.pick_and_place(green block, middle)
Step 1: robot.pick_and_place(red block, blue block)
**** Proposed Plan ****
0. Pick the green block and place it on the middle.
1. Pick the red block and place it on the blue block.
2. Task completed.

Approve this plan? (Type 'approve' to accept, or describe changes): switch step 0 with step 1
Replanning based on feedback: switch step 0 with step 1
QUERY:  BEGIN PREVIOUS WRONG OUTPUT:
robot.pick_and_place(green block, middle)
robot.pick_and_place(red block, blue block)
done()
END PREVIOUS WRONG OUTPUT.

MODIFICATIONS REQUESTS:.
switch step 0 with step 1

**** Solution ****
BEGIN SCENE DESCRIPTION:
objects = [red block, green block, blue block, blue sphere, red sphere, yellow block]
END SCENE DESCRIPTION.

# BEGIN PREVIOUS WRONG OUTPUT:
robot.pick_and_place(green block, middle)
robot.pick_and_place(red block, blue block)
done()
END PREVIOUS WRONG OUTPUT.

MODIFICATIONS REQUESTS:.
switch step 0 with step 1
Step 0: robot.pick_and_place(red block, blue block)
Step 1: robot.pick_and_place(green block, middle)
**** Proposed Plan ****
0. Pick the red block and place it on the blue block.
1. Pick the green block and place it on the middle.
2. Task completed.

Approve this plan? (Type 'approve' to accept, or describe changes): yes
Plan approved. Proceeding to actuation.
```

Figure 12: User feedback example 2

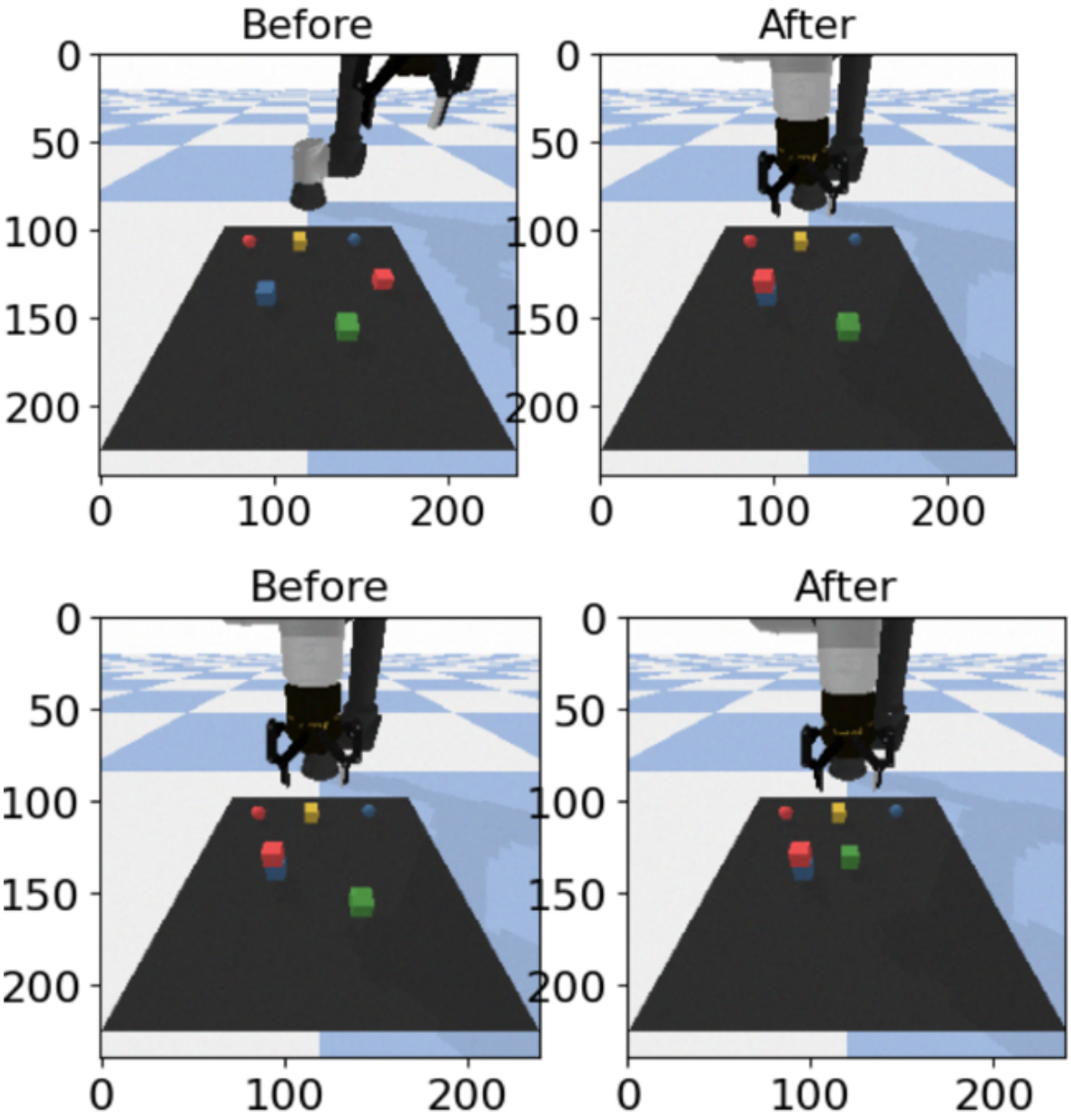

Figure 13: Outcome example 2

```
**** Solution ****
BEGIN SCENE DESCRIPTION:
objects = [red block, green block, blue block, blue sphere, red sphere, yellow block]
END SCENE DESCRIPTION.

# put the green, yellow and red blocks in the top left , top right and bottom right corners following this order.
Step 0: robot.pick_and_place(green block, top left corner)
Step 1: robot.pick_and_place(yellow block, top right corner)
Step 2: robot.pick_and_place(red block, bottom right corner)
**** Proposed Plan ****
0. Pick the green block and place it on the top left corner.
1. Pick the yellow block and place it on the top right corner.
2. Pick the red block and place it on the bottom right corner.
3. Task completed.

Approve this plan? (Type 'approve' to accept, or describe changes): switch step 1 with step 0
Replanning based on feedback: switch step 1 with step 0
QUERY:  BEGIN PREVIOUS WRONG OUTPUT:
robot.pick_and_place(green block, top left corner)
robot.pick_and_place(yellow block, top right corner)
robot.pick_and_place(red block, bottom right corner)
done()
END PREVIOUS WRONG OUTPUT.
MODIFICATIONS REQUESTS:
switch step 1 with step 0
```

```
**** Solution ****
BEGIN SCENE DESCRIPTION:
objects = [red block, green block, blue block, blue sphere, red sphere, yellow block]
END SCENE DESCRIPTION.

# BEGIN PREVIOUS WRONG OUTPUT:
robot.pick_and_place(green block, top left corner)
robot.pick_and_place(yellow block, top right corner)
robot.pick_and_place(red block, bottom right corner)
done()
END PREVIOUS WRONG OUTPUT.
MODIFICATIONS REQUESTS:
switch step 1 with step 0
Step 0: robot.pick_and_place(yellow block, top right corner)
Step 1: robot.pick_and_place(green block, top left corner)
Step 2: robot.pick_and_place(red block, bottom right corner)
**** Proposed Plan ****
0. Pick the yellow block and place it on the top right corner.
1. Pick the green block and place it on the top left corner.
2. Pick the red block and place it on the bottom right corner.
3. Task completed.

Approve this plan? (Type 'approve' to accept, or describe changes): yes
Plan approved. Proceeding to actuation.
Initial state:
GPT-3 says next step: Pick the yellow block and place it on the top right corner.
```

Figure 14: User feedback example 3

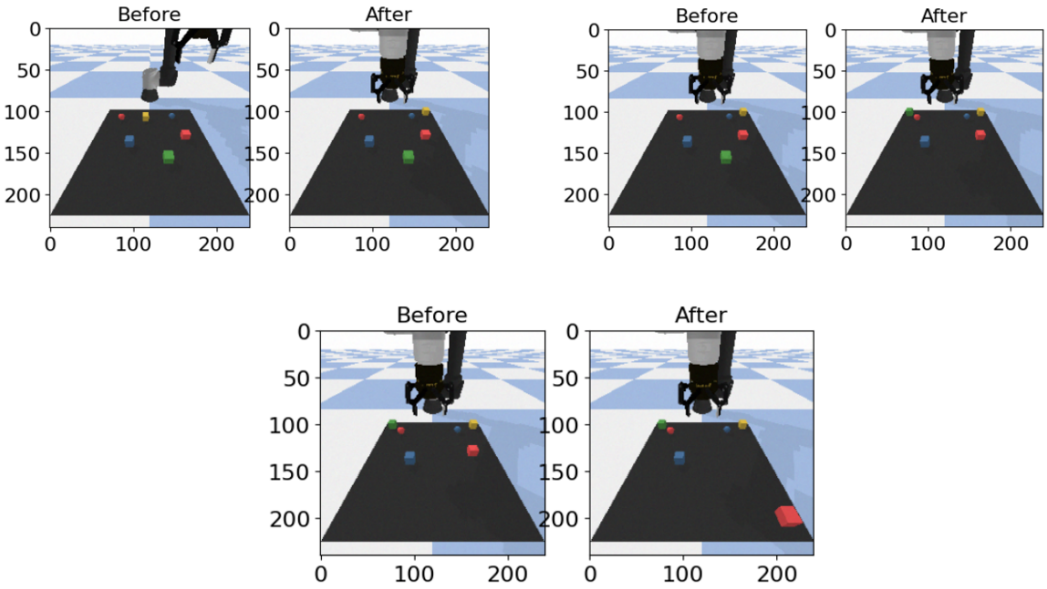

Figure 15: Outcome example 3

3.4 LLM Grounding and Spatial Reasoning

Thus far the LLM has been used as a high-level planner but its information regarding the environment was limited to knowing which objects were in the scene. "Grounding the LLM" consists in giving it lower level information which allow it to perform action closer to the actuation model.

It was chosen to give the LLM name and coordinates of key positions in the environment and to use them to create new place positions on the request of the user. The LLM interprets relative spatial instructions (e.g., "put the blue block between middle and bottom right corner") by mapping them to precise coordinates.

In the prompt, we provided a defined set of rules that the LLM has to follow (role definition, dictionary of predefined named coordinate, strict response format):

- You are a geometry assistant. Your task is to understand the object to be moved and the coordinates of the point between which it should be placed.
- The following named coordinates are available:
  - `"top left corner": (-0.25, -0.25, 0)`
  - `"top right corner": (0.25, -0.25, 0)`
  - `"middle": (0, -0.5, 0)`
  - `"bottom left corner": (-0.25, -0.75, 0)`
  - `"bottom right corner": (0.25, -0.75, 0)`
  - `"top center": (0, -0.25, 0)`
  - `"bottom center": (0, -0.75, 0)`

  - `"left center": (-0.25, -0.5, 0)`
  - `"right center": (0.25, -0.5, 0)`
- You must return only one line, with no explanation.
- `OBJECT` refers to the object mentioned in the user sentence.
- Always return the result in the following format: `[x1, y1, x2, y2] & OBJECT`
- **IMPORTANT**: If no object is detected in the sentence or if no new position is needed, return `ERROR`.

The user query is passed to this model-implementation before it going to the one previously explained. If it detects no necessity to give an output, i.e. it thinks the query does not require to generate a new position, the query will simply be passed to the previous model-implementation without change.

Otherwise, the output `[x1, y1, x2, y2] & OBJECT` is used to generate a new position based on the coordinates given (interpolation). The position is added to the list of available place positions and the a new query is generated with the form:

- query = 'put the ' + OBJECT + ' in new position'

Following an example of the model output:

```
Query: put the blue block between middle and bottom right corner

LLM response:
[0, -0.5, 0.25, -0.75] & blue block
```

Figure 16: LLM grounding example

4 Conclusions

In this report, we explored the integration of Large Language Models (LLMs) into robotic high-level planning systems, showing how natural language commands can be translated into robot-executable actions through prompt engineering and contextual understanding.

We began by analyzing the Say-Can framework, which separates reasoning ("Say") from feasibility ("Can"), and showed how a similar modular structure can be adapted using general-purpose LLMs like GPT in a zero-shot configuration. Despite the absence of domain-specific fine-tuning, robust performance was achieved through careful prompt design and contextual examples.

We enhanced this architecture by integrating visual feedback, allowing the LLM to reason about the current environment. This improved semantic and physical grounding.

The incorporation of user feedback enabled dynamic replanning and correction. This approach improves reliability and trustworthiness, allowing users to steer the robot behavior without reprogramming or retraining.

Crucially, it was also evaluated the model's ability to perform spatial reasoning. Results showed that LLMs can handle basic spatial relationships when well prompted.

Overall, this project indicate that LLMs are powerful tools for intuitive robot control and high-level planning, especially when augmented with perception and feedback mechanisms. Future work should focus on tighter coupling with structured world models, enhanced memory and context handling, and hybrid architectures that combine the flexibility of LLMs with the precision of symbolic reasoning.

List of Figures