Design of a Pick-and-Place Task: An implementation with UR5e and RG2 Gripper

Andrea Cavaliere

December 6, 2023

Contents

1 Introduction

This report documents the implementation of a robotic pick-and-place task in a continuous cycle, implemented within the Gazebo simulation environment. The system consists of a UR5e robotic arm equipped with a GR2 gripper. Following this, the problem is outlined in in Sec. 1.1. Sec. 2, decomposes it into two smaller challenges and includes an analysis of their components along with the implementation choices made to address them. In Sec. 3, possible extensions of the framework are presented.

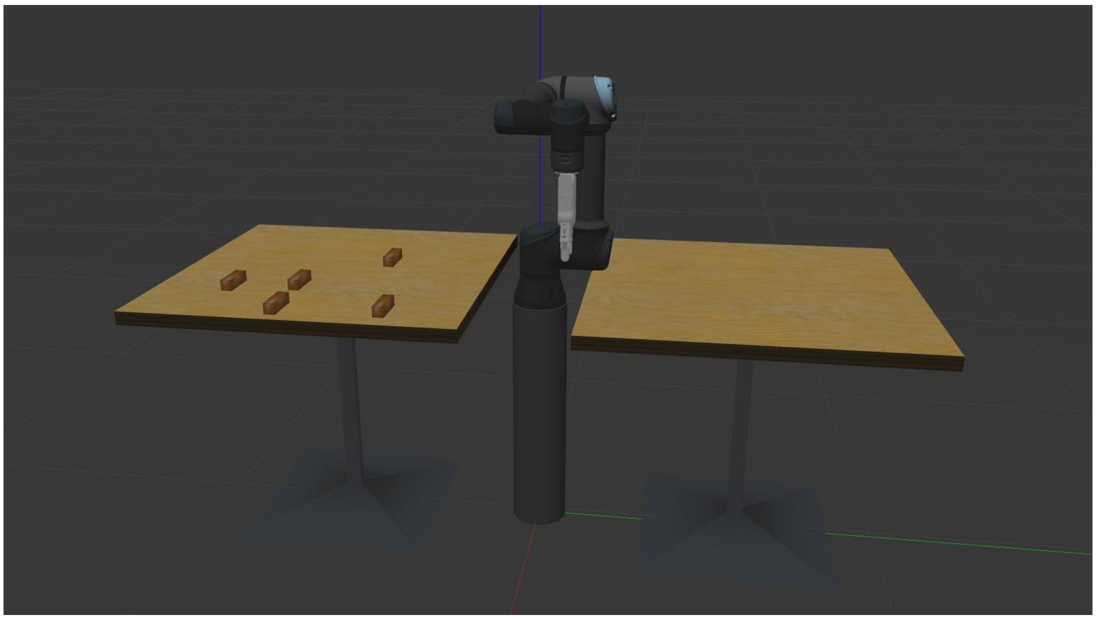


Figure 1: Simulation environment in Gazebo11

1.1 Problem Statement

The robot is placed between two tables (Fig. 1). The task to perform is to transfer blocks from the table on its right to the table on its left.
The control loop is designed to be cyclical, i.e., the robot automatically starts the process when a new block is placed on the starting table.

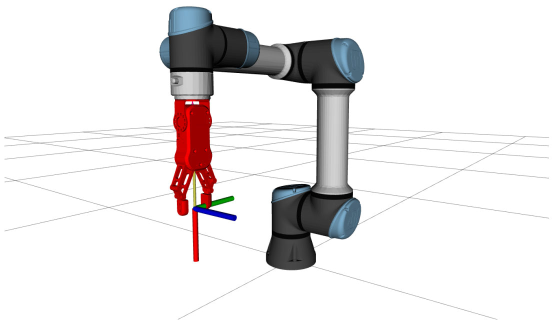
2 Design Choices

While there is no requirement for a specific positioning of the blocks, the chosen design strategy involves organizing them in an equidistant matrix pattern, ensuring a cautious clearance from the table edges.
To perform this task, two requirements should be satisfied:

- Grasping mechanics must be evaluated, focusing on the desired interaction between the gripper and the objects
- Motion planning should be aware of potential collisions with the environment

Additionally, it is necessary consider that the integration of the gripper alters the robot's configuration from Universal Robots' standard models.

2.1 Grasping


(a) manipulator and representation of end-effector frame

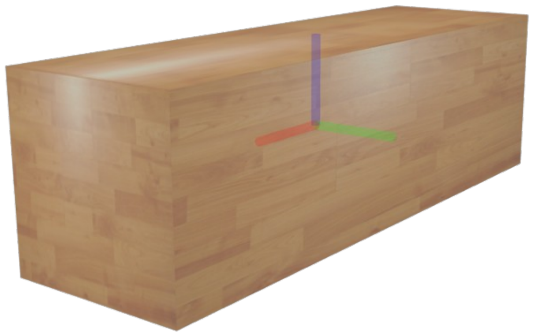

(b) Generic block

Figure 2: Robot and target object

The object to grasp is reported in Fig. 2b. It is a generic box measuring $0.1 \times 0.03 \times 0.03m$. Therefore, it presents two distinct sections: a square section measuring $0.03 \times 0.03m$ and a rectangular section measuring $0.1 \times 0.03m$.
Upon initial analysis, there are two potential strategies to grasp the object: one along the rectangular section and the other along the square section. The square section has been chosen to achieve a robust gripper closure.
Let's define with $\{EE\}$ the frame placed in the middle of the end-effector and $\{T\}$ the one placed in the center of the object to grasp. According to Fig. 2, to achieve the chosen grasping modality, it is sufficient to align z_{EE} with x_T and x_{EE} normal to the rectangular section. This approach ensures robust grasping against potential rotations of the object around the z and y axes.
Notice that, given the object's symmetrical shape, the possibility of rotating around the x-axis has been neglected for simplicity. This simplification is based on the fact that, in real-world situations involving computational vision tools, the system would not be capable of recognizing multiple 90° rotations around this axis and would thus have to rely on its own convention.

2.2 Motion Planning

As mentioned above, the goal is to pick the object from the table on the robot's right and place it on the table on the robot's left. Since the control system can only utilize the a priori knowledge of the pose of the two tables and their geometric properties for collision avoidance, and the knowledge of the robot configuration at each step (no external perception data are provided), an intermediate phase between the pick and the place of the object is introduced. Indeed, the robot is forced to return to its home position. This ensures that the robot moves away and returns to a known and safe configuration before moving into a partially unknown environment. Therefore, the execution of the whole task is composed of four different phases that are reported below according to their execution order:

1. the pick phase;
2. the return in home position phase;
3. the place phase;
4. the return in home position phase.

The use of a specific planner has not been discussed because, in such a simple semi-structured environment, the default planner provided by MoveIt, Lazy Bidirectional KPIECE, was deemed sufficient. For this reason, once the starting point and the endpoint of the desired trajectory are known, only two relevant midway points are required. The first one is the point placed exactly in the middle of trajectory. The second midway point is the one related to an *approach pose* and it is placed exactly above the block with the desired grasp orientation.
Note that the reason for not using motion planner systems with properties of asymptotic stability lies in the fact that such improvement impacts the speed of convergence to a solution.

3 Implementation

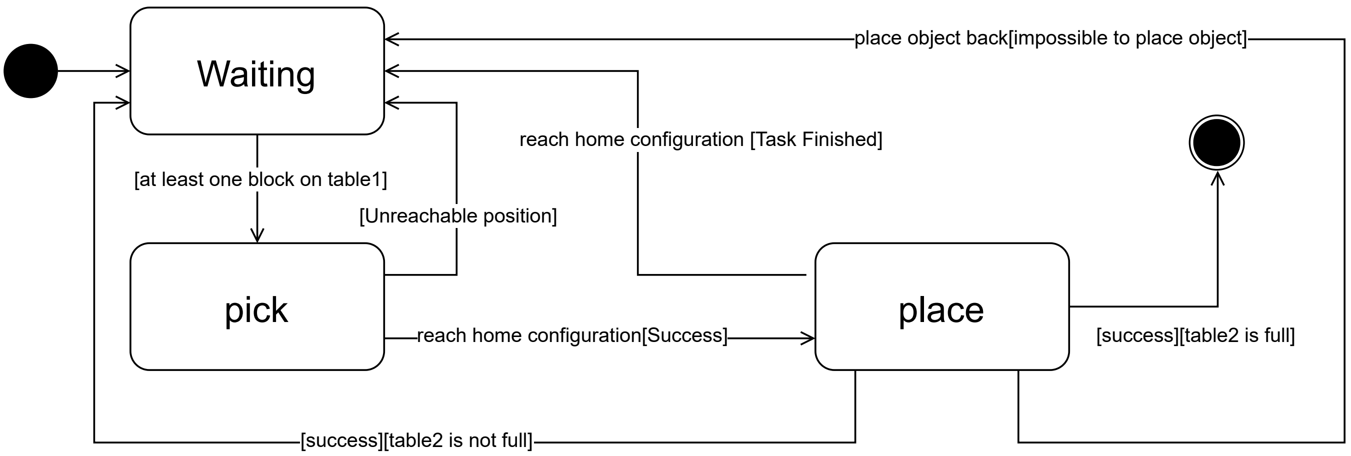

Waiting
place object back[impossible to place object]
reach home configuration [Task Finished]
[at least one block on table1]
[Unreachable position]
pick
reach home configuration[Success]
place
[success][table2 is full]
[success][table2 is not full]

Figure 3: State machine diagram of the robot loop

The simulation runs on Ubuntu 20.04 LTS (Focal Fossa) and requires the ROS Noetic distribution with Gazebo11. Additionally, the MoveIt Framework has been integrated to enhance the starting software.
The control loop system presents three levels: the simulated hardware interface and control, the high level control and the mock perception system. The foundational level consists of low-level control and interface, facilitated by a MoveIt move_group node configured specifically for the UR5e and gripper combination. Building upon this, there is the pick_and_place_node, responsible for executing the task loop for each block.
Fig.3 illustrates the state machine diagram of the robot control loop. While the functions of most states are intuitive, the *waiting* state needs further explanation. In this state, the robot queries a ROS service to obtain a list of objects on Table 1, the one placed on the robot's right, and the task starts only when at least one block is present on the table. Indeed, for this purpose, the ROS service server node, which simulates a vision perception system, is introduced. When queried, this node (from now on, fake_vision node) retrieves a list of all objects in the scene that meet two specific criteria. First, their names must contain the specified substring, in this case, *block*. Secondly, their positions must be confined to the area above the table.
Reviewing the code, it is possible to notice that the y coordinate is not lower-bounded. This detail is a limitation of the current implementation. Consequently, the solution that is proposed in this project as proof of concept tends to offer limited applicability. A more comprehensive and universally applicable solution, based on a broader set of premises, could be developed with additional time and effort. Nonetheless, potential methods to tackle this challenge will be explored in Subsection 4.1.2.

4 Conclusion and Additional Questions

4.1 How would you improve the efficiency and reliability?

Improving the efficiency and reliability of the system involves comprehensive strategies in both areas, each with its specific focus and requirements.

4.1.1 Efficiency

To enhance efficiency, a detailed evaluation of performance metrics and resource utilization is essential. In the existing setup, the robot's limited environmental awareness leads to the execution of trajectories that are safer but less efficient, as exemplified by the robot's return to the home configuration in each cycle. In a fully structured environment with almost all variables known, it's feasible to compute an optimal trajectory. However, this approach demands careful consideration in complex systems that must be adaptable to a variety of tasks and require a significant degree of abstraction in key parameters like object types and grasping strategies.

4.1.2 Reliability

Thoroughly testing the application in a staging environment that closely mimics the production setup is fundamental. Starting from there, a key aspect is the development of robust error handling and recovery mechanisms. The extent and depth of these mechanisms should be aligned with the desired generality of the solution. This means finding a balance between abstract system design and practical error management. For instance, in case of a critical failure like a power outage, the current strategy is for the robot to revert to the home configuration and ensure the gripper is open. This approach is straightforward and safe for the robot and any other entities in the environment, but it does not always benefit the object being manipulated. For example, if a system reboot occurs during the second phase of a Pick subtask, the robot would drop the object and return home.

Various implementation approaches can be considered, starting from whether the robot is aware of holding an object and how it should respond. The most straightforward solution could be to proceed directly with the placement, which necessitates knowledge of the available space and might require additional perception systems. Alternatively, a more encompassing approach would be to use an external database or allocate persistent memory to ensure data persistence.

Addressing the limitation highlighted in Section 3, the current system only employs one type of grasp, and the robot lacks information about the object's physical characteristics. Incorporating a vision system could allow for the evaluation of the most suitable type of grasp for each object, potentially chosen from a library of known grasps in a straightforward and deterministic manner.

Summarizing, the way to improve efficiency and reliability involves both optimizing the system's trajectory planning and enhancing its ability to handle errors and adapt to varying conditions. This will likely require integrating advanced perception capabilities and developing more sophisticated algorithms for trajectory and error management.

4.2 What would you do if we needed multiple robots in a system?

The design currently adopted facilitates the adoption of multiple robots. The structure of the message between the fake_vision module and the robot employs a list of objects, a choice underpinned by the knowledge that, with appropriate configurations, one could develop a multi-robot MoveIt_config package, each arm governed by its move group. This would require some adaptation in the loop of the manager_pipeline.script but would not be difficult. Although this is a viable option, I propose an alternative solution that entails fewer modifications and maintains a high degree of modularity even with the addition of an indefinite number of robots.

Each robot would still be completely autonomous during the task maintaining independent planning

and control pipeline. In the current implementation is already possible to add new robots. The problem is that the robots would not be aware of each other. To solve this problem, it is sufficient to adapt the *get_model_list_service_server* node and its flow. In particular, assuming no editing on the environment, the workspace of the two or more robots would coincide and this could be critical. For this reason, this server should provide only one object to each robot and choose them based on the operational space, ensuring a free collision path. This method streamlines the picking process.
For the placement phase, a second service server, analogous to the *get_model_list_service_server* could be employed. This server would assign to each robot request a free spot on Table 2, following the same idea presented above to avoid collisions.
Implementing such a system would elevate this new node to a pivotal role, introducing an additional prerequisite to the get_model_list state: the availability of at least one free slot before starting any operation.

4.3 How would you deploy the application to multiple physical locations? What is needed to scale it?

In optimizing the deployment strategy for this application, configuration settings will be externalized or kept in an external high-level control loop to cater to the varied needs of different locations and robots. This process will involve optimizing the Docker container by incorporating configuration files as an external volume in the docker-compose.yml file or sourcing them from dedicated cloud services from a command in the entrypoint setup.
Orchestration tools like Kubernetes or Docker Swarm are mainly used for managing deployments, handling scaling, and ensuring that the system is always available. Specifically, Kubernetes is really useful because it has features like auto-scaling and load balancing. These features are important for managing resources well across different locations. Within Kubernetes, the Horizontal Pod Autoscaler (HPA) is the component responsible for this scaling behavior. It continuously monitors the load and performance metrics of an application, such as CPU usage or custom metrics, and adjusts the number of pods accordingly. This ensures that the application always operates with optimal resources, without wasting resources.
The integration of a Continuous Integration/Continuous Deployment (CI/CD) pipeline, potentially utilizing a tool like Travis for its quick setup, will automate testing and deployment processes. This approach ensures that all instances across various locations remain up-to-date and consistent. The CI/CD pipeline will also be tailored to suit different deployment scenarios, with an emphasis on customization to accommodate diverse operational environments. This means the system will be designed to quickly adapt to changes in its environment or operational parameters. For instance, if a new type of robot or sensor is introduced, the system can dynamically adjust its configurations to accommodate these changes without the need for manual intervention. This could be achieved through automated scripts that detect changes in the system and automatically update the configurations.