Intelligent Robotic Manipulation - Final Project


TECHNISCHE UNIVERSITÄT DARMSTADT

Ansh Prakash, David Lang
April 7, 2025

Contents

1 Introduction

The overall Goal of the project is to grasp a YCB-Object and place it in a goal basket while avoiding obstacles. The whole pipeline for the pick-and-place task is divided into five main parts: Perception 2.1, Control 2.2, Grasping 2.3, Tracking 2.4, and Planning 2.5. Each part of our approach is explained in detail, with isolated evaluations where appropriate. Finally, we show our global results in 3 and conclude our work in 4.

Our code can be found here: `https://github.com/AnshPrakash/Franka-pick-and-place`.

2 Method

2.1 Perception

The perception module consists of two components: point cloud extraction and a registration algorithm for pose estimation. We qualitatively evaluate the perception performance through visualizations.

2.1.1 Point Cloud Extraction

For estimating the 6D pose of an object, we need to extract its point cloud based on our sensor data. This data consists of a depth image and its corresponding segmentation map for a static camera and the robots endeffector camera respectively. Additionally, we have access to the view and projection matrices used by the cameras.

The process begins by computing the inverse of the combined projection and view matrix, which allows us to transform 3D points from Normalized Device Coordinates (NDC) back into world coordinates. Since NDC represents positions normalized to the range $[-1, 1]$, we construct the input points by generating a regular grid over the image resolution for the x and y components and linearly mapping the depth buffer values to the NDC z-range. Applying the inverse transformation followed by a perspective division yields the corresponding 3D world positions. Finally, with the provided segmentation map we can extract all points related to the object of interest.

While a static camera view is sufficient for capturing obstacles, the target objects require a local view to generate accurate point clouds. For this reason, we empirically selected five predefined viewpoints — top, front, back, left, and right — to ensure coverage of all relevant surface areas. We explored several strategies to combine the collected sensor data into a single point cloud. Initially, we considered computing a TSDF (Truncated Signed Distance Function) volume directly from the multiple depth images and sampling a point cloud from it afterward. This method is often referenced in literature due to its robustness to noise. However, it requires explicit camera intrinsics and extrinsics, which must be computed from the given projection and view matrices. In our case, this approach did not yield usable results and also failed to isolate the object of interest; therefore, we did not pursue it further.

Instead, we extracted a point cloud from each individual view and refined any misalignments sequentially using the registration algorithm described in the following section. Although this approach produced promising results, the registration occasionally failed, resulting in distorted point clouds that hindered reliable pose estimation. Consequently, we ultimately decided to use only the view that produced the point cloud with the highest number of points.

2.1.2 Registration Algorithm

After extracting the detected object point cloud we utilized a model-based registration algorithm to estimate its 6D pose. Therefore we first extracted the stored ground-truth mesh of our target object, applied the respective scaling and centering and sampled a point cloud from it. Overall we want to estimate the relative pose from our ground-truth pointcloud to our detected pointcloud. The algorithm consists of three steps:

1. **Preprocessing**
   In the preprocessing step we prepare both point clouds for effective registration. After downsampling using a voxel grid filter, surface normals are computed by analyzing local neighborhoods. Subsequently Fast Point Feature Histograms (FPFH) descriptors are computed to capture local geometric properties around a point. These serve as robust features for finding matching points during global registration.

2. **Global Registration – RANSAC**
   In the global registration step using the Random Sample Consensus (RANSAC) algorithm, we determine an initial, coarse alignment between the two point clouds in order to provide a reliable starting point for the refinement during local registration. RANSAC operates by randomly sampling sets of point correspondences from source and target point clouds, estimating candidate transformations, and assessing each transformation based on how many correspondences (inliers) align within a predefined distance threshold. By repeatedly sampling and evaluating potential transformations, RANSAC robustly identifies an initial alignment, even in the presence of noise or outliers.

3. **Local Registration – ICP**
   ICP (Iterative Closest Point) is an algorithm for fine-tuning the alignment between two point clouds. The default Point-to-Point formulation works by iteratively matching points in the source cloud with the closest points in the target cloud and computing a transformation that minimizes the direct Euclidean distance between them. The Point-to-Plane formulation minimizes the distance from points in the source cloud to the tangent planes of the target cloud instead. In many cases where good normal information is available this version is superior, which is why we used it as the initial refinement approach and Point-to-Point as a fallback.

The point cloud alignment process may fail due to factors such as inaccurate point cloud extraction, insufficient overlapping points from bad views, or incorrect correspondences. Because the global registration includes a stochastic component, the algorithm performs multiple attempts in case of failures. As an additional fallback, a Point-to-Point ICP approach is used, which typically offers a higher success rate but slightly lower alignment quality. Furthermore, to simplify the registration task, the source point cloud is initially translated to align its center with that of the target cloud, primarily leaving only orientation adjustments to be estimated.

Open3D is leveraged throughout the perception module as it provides many tools and pre-implemented algorithms for 3D processing.

2.1.3 Visualizations

For all visualizations we use the approach to translate the source point cloud before pose estimation as described above. Our detected object point cloud is displayed in green, the ground-truth point cloud transformed by our estimated pose in blue and the ground-truth point cloud transformed by Pybullet's pose in red.

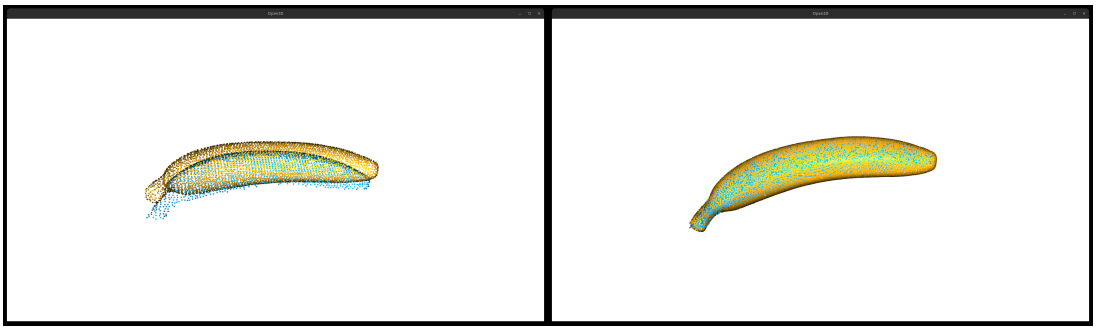

Figure 1: Perception of the YcbBanana object.

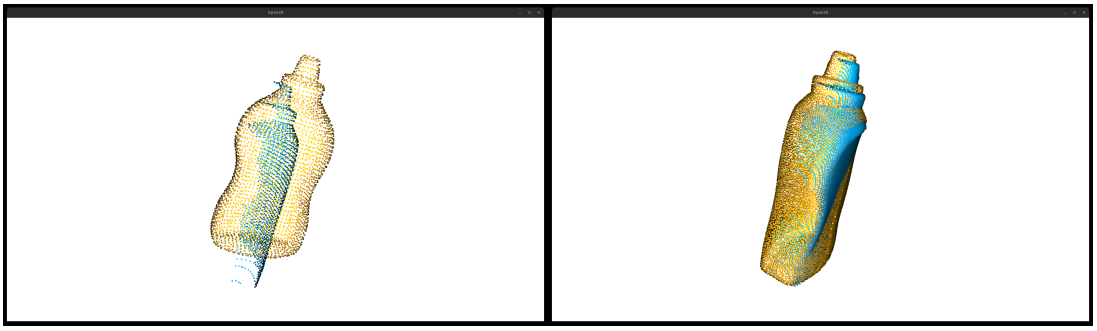

Figure 2: Perception of the YcbMustardBottle object.

While there is a slight offset between the detected point cloud and the transformed ground-truth mesh, this difference is relatively small and therefore negligible for the subsequent grasp generation step.

Figure 3: Visualizating the difference of our pose estimation vs PyBullet's pose applied to the ground-truth point cloud.

As depicted on the right side of Figure 3, our extracted point cloud accurately represents the observed object. However, directly applying PyBullet's ground-truth pose transformation to the ground-truth mesh leads to discrepancies. Consequently, numerically evaluating the accuracy of our point cloud extraction against the ground-truth pose is challenging. This discrepancy may result from the ground-truth mesh not being initially aligned with the origin and identity rotation. Nonetheless, for the scope of our task, this misalignment does not pose an issue, as our primary requirement is an accurate object representation suitable for sampling grasp poses.

2.2 Control

We use the Robotics Library to implement inverse kinematics functionality. The robot model within the library is configured to closely match the setup used in the PyBullet simulation. To solve the inverse kinematics problem, we formulate the task of finding a joint configuration for a given pose as a nonlinear programming (NLP) problem, which we solve using KOMO.

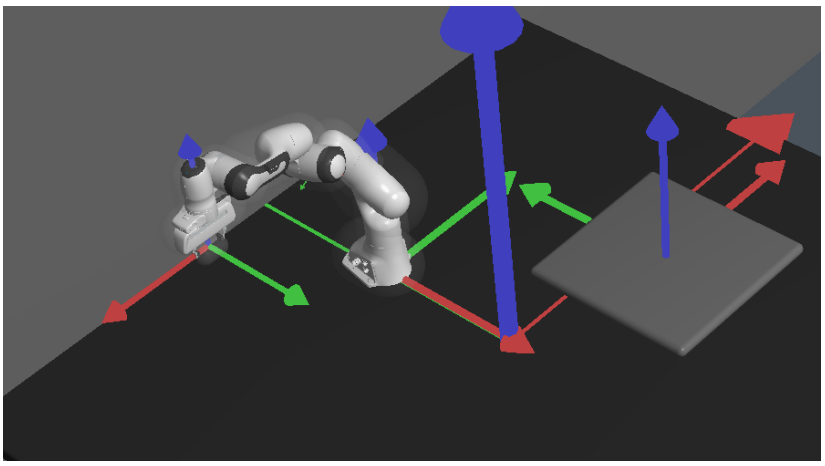

Figure 4: IK finds a feasible solution to a target pose

Algorithm 1 Compute Target Configuration

function ComputeTargetConfig($target\_pos,\ target\_ori$)
  if $target\_ori \neq$ null **then**
    $target\_ori \leftarrow$ ConvertOrientation($target\_ori$)
  end if
  $q_{curr} \leftarrow$ GetJointPositions($robot$)
  UpdateConfiguration(q_{curr})
  ($wall\_pos$, _) $\leftarrow$ GetWallPosition($sim.wall$)
  $q_{home} \leftarrow$ GetCurrentState($configuration$)
  $komo \leftarrow$ InitKOMO($configuration$, 1, 1, 0, False)
  AddObjective(komo, JointState, q_{home}, 10, sos)
  AddObjective(komo, AccumulatedCollisions, 10, sos)
  AddObjective(komo, JointLimits, , ineq)
  AddObjective(komo, Position (l_gripper), $target\_pos$, 10, eq)
  if $target\_ori \neq$ null **then**
    AddObjective(komo, Quaternion (l_gripper), $target\_ori$, 10^3, sos)
  end if
  AddObjective(komo, PositionAboveTable, $[0, 0, 1]$, 10, ineq)
  AddObjective(komo, PositionFromWall, $wall\_pos[0]$, 10, ineq)
  $result \leftarrow$ Solve($komo$)
  if $result$ is feasible **then**
    return GetPath(komo)
  else
    return null
  end if
end function

2.3 Grasping

The grasping module is structured into two steps: grasp pose generation and the subsequent grasp execution. In an additional section, we visually demonstrate the grasping process through several examples. Quantitative metrics evaluating grasp performance are provided in the Results section (3).

2.3.1 Grasp Pose Generation

For grasp pose generation, we utilized the GIGA repository adapted by the PEARL lab `https://github.com/pearl-robot-lab/GIGA`. For both following approaches we used the transformed ground-truth mesh - or the sampled point cloud - of the target object.

GIGA proposed by Z. Jiang et al. in 2021 (`https://arxiv.org/abs/2104.01542`) is a learned model that predicts a 6D grasp pose based on a given TSDF representation. It also directly outputs the respective scores for each predicted grasp. Given its promising capabilities, we initially employed this approach by converting the target mesh into a TSDF representation through scaling and evaluating it on a standardized voxel grid. However, the model failed to produce valid grasp predictions, likely due to uncertainty regarding TSDF quality, appropriate grid resolution, scaling factors, or inherent limitations of the trained model. Due to these challenges and limited time resources, we transitioned from GIGA to a more robust, sampling-based method.

The alternative sampling-based approach, while not related to GIGA, is implemented in the same mentioned repository and references the following paper `http://journals.sagepub.com/doi/10.1177/0278364917735594`. Its strength, particularly suited to our scenario, lies in directly operating on raw point clouds and performing internal validity checks, thereby ensuring generated grasps are practically feasible. To identify the most reliable grasp, we applied a straightforward scoring heuristic based on the assumption that most objects in the YCB dataset are optimally grasped from above. Specifically, grasps were ranked according to their cosine similarity between the grasp's local z-axis and the global negative z-axis direction.

Additionally, we applied different scaling factors (1.0 or 1.5) for target objects, addressing practical grasping constraints; some objects can get too large for any feasible grasp, while others can get too small to allow reliable grasp execution.

2.3.2 Grasp Execution

Assuming feasible grasp poses have already been generated, we model the grasp execution through five sequential stages:

1. **Pre-Grasp Pose**
   To ensure safe and efficient robot motion toward the target object, we first establish a pre-grasp pose. This pose is computed by applying a safe-distance offset along the negative local z-axis of the intended final grasp pose. Utilizing this intermediate pose helps avoid collisions and ensures a controlled approach trajectory.

2. **Open Gripper**
   Before approaching the object, we need to manually open the gripper.

3. **Approach Final Grasp Pose**
   To guarantee that the gripper approaches the final grasp pose along a controlled, linear path, the trajectory between the pre-grasp and final grasp pose is linearly interpolated. This approach minimizes unintended collisions or contacts with the object during the final approach.

4. **Close Gripper**
   After successfully reaching the final pose, we close our gripper and secure the grasp.

5. **Retreat Pose**
   Finally, to ensure the object is safely lifted from the surface and prevent unintended interactions with the environment, a retreat pose is defined. This pose is computed by applying an upward offset along the global z-axis, lifting the object vertically away from the contact surface.

To improve robustness, the grasp execution module includes error handling: if any of the five stages fail due to infeasible poses or if the gripper is found completely closed upon reaching the retreat pose (indicating that the object slipped during grasping), the next-best sampled grasp is attempted. After exceeding a predefined maximum number of retries, the perception module is reactivated to recompute the object's pose. This accounts for potential object displacement caused by collisions or unsuccessful grasp attempts.

2.3.3 Visualizations

The following visualizations show the sampled grasps for two different Ycb objects. On the right the selected grasp based on our scoring heuristic is displayed.

For the meat can shown in Figure 5, grasping from above is beneficial, as lateral grasps would not successfully secure the object. In contrast, the mustard bottle in Figure 6 presents one of the few scenarios in which the selected grasp, while feasible, is suboptimal. Nevertheless, given the retry mechanism available in case of failures, this choice remains acceptable.

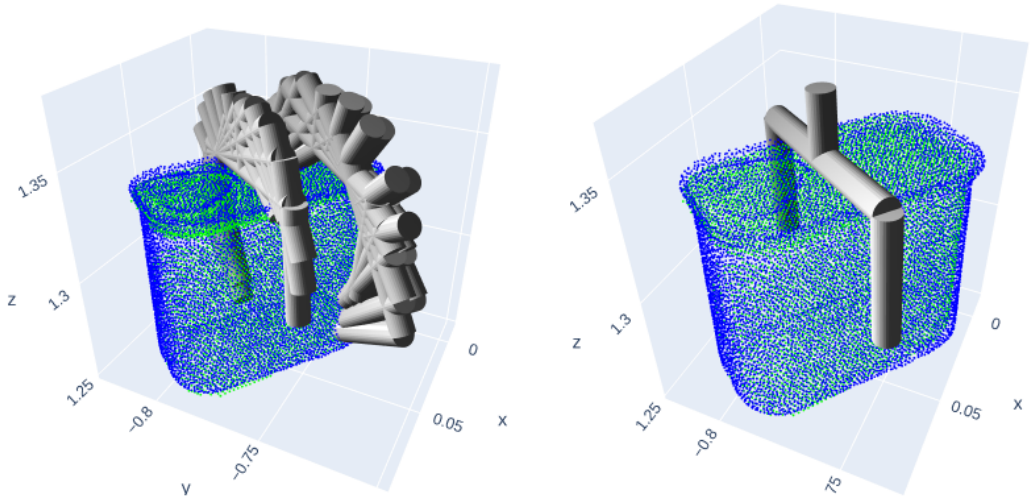

Figure 5: Grasping of the YcbPottedMeatCan object.

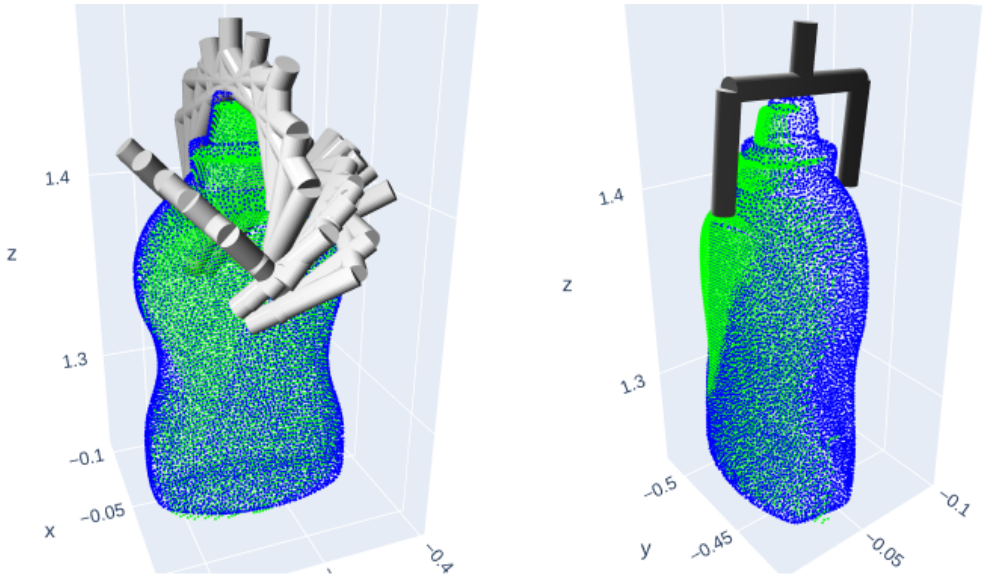

Figure 6: Grasping of the YcbMustardBottle object.

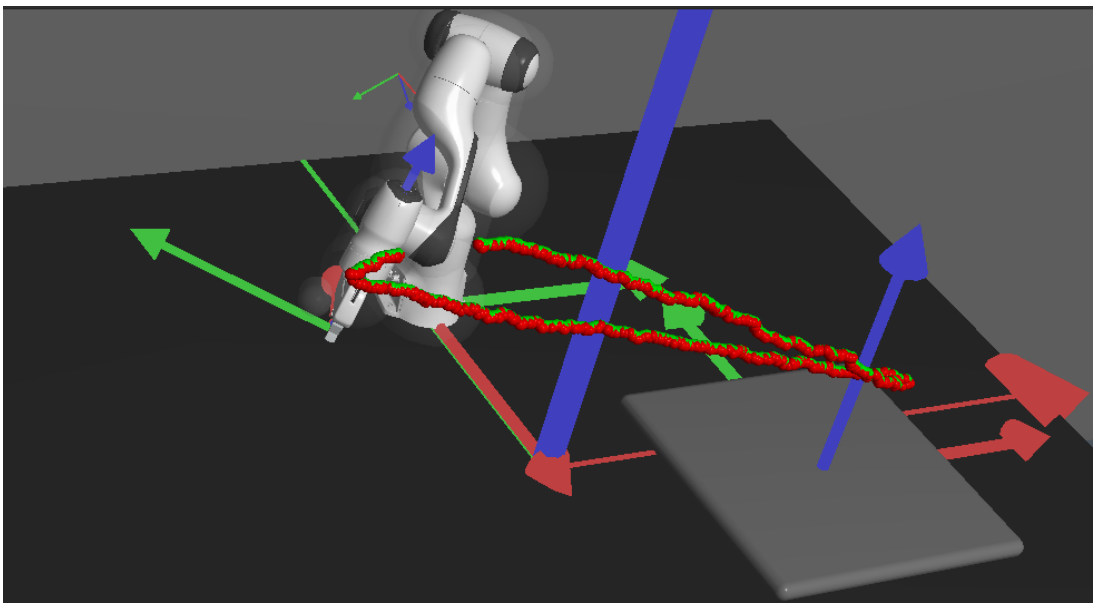

Figure 7: Tracking of a Obstacle 2: Green is the predicted obstacle and Red is the actual position of the obstacle

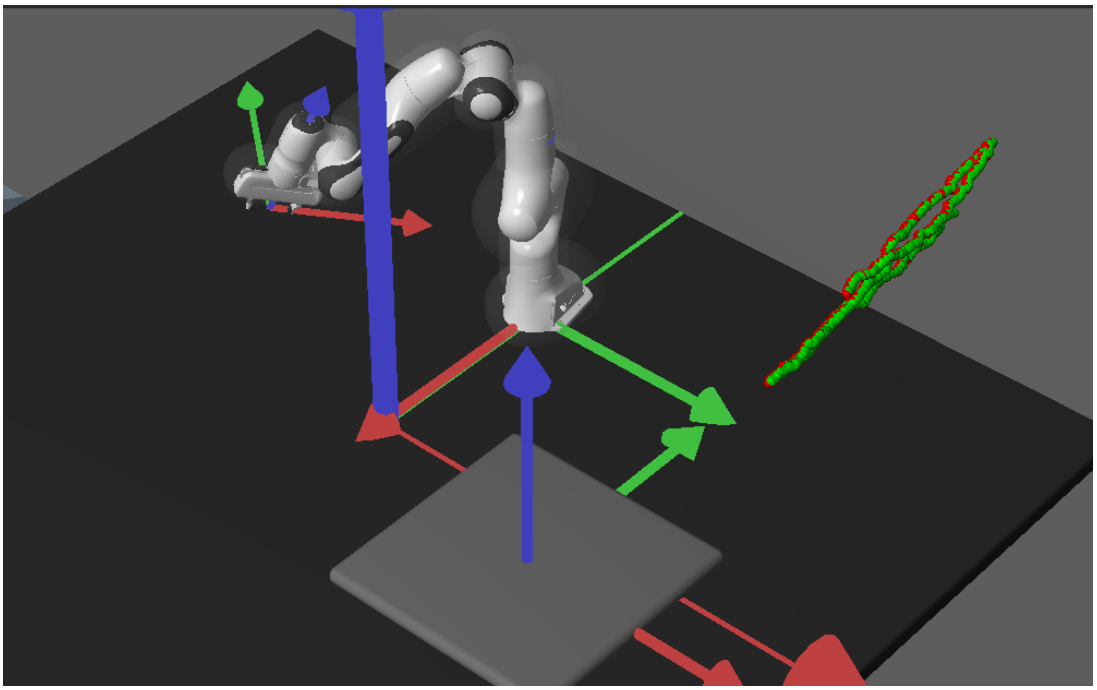

Figure 8: Tracking of a Obstacle 1: Green is the predicted obstacle and Red is the actual position of the obstacle

2.4 Tracking

To have a robust and consistent tracking we are using Kalman filter over our per step estimate from solving a Linear system of equation based on the data we receive from point cloud of perception module using the static camera view.

Here is out dynamics model:

State Transition Model For a constant-velocity model, the discretized equations (using a time step dt) are:

$$\begin{aligned} x_{k+1} &= x_k + v_x \cdot \Delta t \\ v_{x_{k+1}} &= v_{x_k} \\ y_{k+1} &= y_k + v_y \cdot \Delta t \\ v_{y_{k+1}} &= v_{y_k} \\ z_{k+1} &= z_k + v_z \cdot \Delta t \\ v_{z_{k+1}} &= v_{z_k} \end{aligned}$$

We exploit the fact that our obstacles are spherical and points extracted will lie on the surface of sphere and follows its equation. We solve the following Linear system of equations with Least squares to estimate the center and radius of our obstacles

$$(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2. \tag{1}$$

Expanding, we get:

$$x^2 + y^2 + z^2 - 2ax - 2by - 2cz + \left(a^2 + b^2 + c^2 - r^2\right) = 0. \tag{2}$$

Define:

$$d = a^2 + b^2 + c^2 - r^2, \tag{3}$$

so that the sphere equation becomes:

$$x^2 + y^2 + z^2 - 2ax - 2by - 2cz + d = 0. \tag{4}$$

For each point (x_i, y_i, z_i) in the point cloud ($i = 1, \ldots, n$), we have:

$$x_i^2 + y_i^2 + z_i^2 - 2ax_i - 2by_i - 2cz_i + d = 0. \tag{5}$$

Rearranging, we obtain the linear form:

$$-2x_i\, a - 2y_i\, b - 2z_i\, c + d = -\left(x_i^2 + y_i^2 + z_i^2\right). \tag{6}$$

Let

$$\mathbf{X} = \begin{bmatrix} -2x_1 & -2y_1 & -2z_1 & 1 \\ -2x_2 & -2y_2 & -2z_2 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ -2x_n & -2y_n & -2z_n & 1 \end{bmatrix}, \quad \mathbf{p} = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} -\left(x_1^2 + y_1^2 + z_1^2\right) \\ -\left(x_2^2 + y_2^2 + z_2^2\right) \\ \vdots \\ -\left(x_n^2 + y_n^2 + z_n^2\right) \end{bmatrix}.$$

The linear system can then be written as:

$$\mathbf{X}\mathbf{p} = \mathbf{y}. \tag{7}$$

The least-squares solution is found by:

$$\mathbf{p} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}. \tag{8}$$

Finally, the sphere radius is recovered from

$$r = \sqrt{a^2 + b^2 + c^2 - d}. \tag{9}$$

Further, this is feed into Bayes filter for our tracking framework, once the sphere's center is estimated via the least-squares solution from the point cloud, this center measurement is used to update the Kalman filter. At each time step, the filter first performs a prediction based on the previous state and a motion model (such as constant velocity) to obtain the predicted center position. When the new measurement of the center becomes available, we compute the innovation (i.e., the difference between the measured and predicted center) and update the state estimate and its associated error covariance accordingly. This recursive update process allows the Kalman filter to smooth out measurement noise and maintain an accurate and robust estimate of the object's center during tracking, even in the presence of occlusions or sensor inaccuracies.

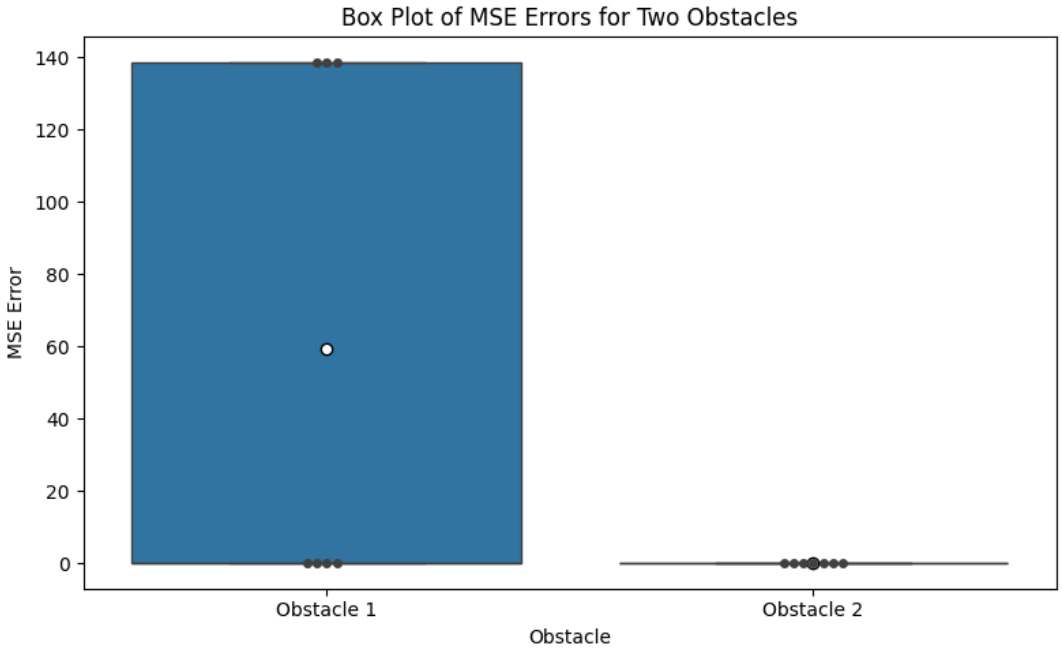

Figure 9: MSE error for both the Obstacles for 10 runs of each 300 sim.step(): Obstacle 1 has high MSE error in cases when it goes behind the wall or occluded by the other obstacle

2.5 Planning

Once the grasp is successful we start the process of putting the object in the tray. We achieve this by sampling a global plan from RRT algorithm and then during movement we replan at a fixed frequency using the same RRT algorithm and then if we detect a possible collision in the future during this re-planning with execute a retreat-replan policy to avoid colliding.

Algorithm 2 Moving the End-Effector (EE) to a Point Above the Tray

$goal \leftarrow$ sample a goal pose above the tray
$replan\_freq \leftarrow f$
for t in time **do**
  if $t \mod replan\_freq = 0$ **then**
    $path \leftarrow$ RRT($goal$, Current position)
    $q \leftarrow$ choose a robot configuration from $path$
    if q is colliding with obstacles **then**
      (q, New goal) $\leftarrow$ retreat-replan policy
      $goal \leftarrow$ New goal
    end if
    set robot to q
  end if
  step()
end for

Local-planning: Retreat-Replan Policy

Algorithm 3 Retreat-Replan Policy

$q \leftarrow$ sample a safe robot configuration
set robot to q
$retry\_freq \leftarrow f$
for t in time **do**
  if robot is colliding **then**
    $q \leftarrow$ sample a safe robot configuration
    set robot to q
  end if
  if $t \mod retry\_freq = 0$ **then**
    $new\_goal \leftarrow$ sample a goal pose above the tray
    $path \leftarrow \text{RRT}(new\_goal)$
    $next\_q \leftarrow$ choose a robot configuration from $path$
    return $(next\_q, new\_goal)$
  end if
end for

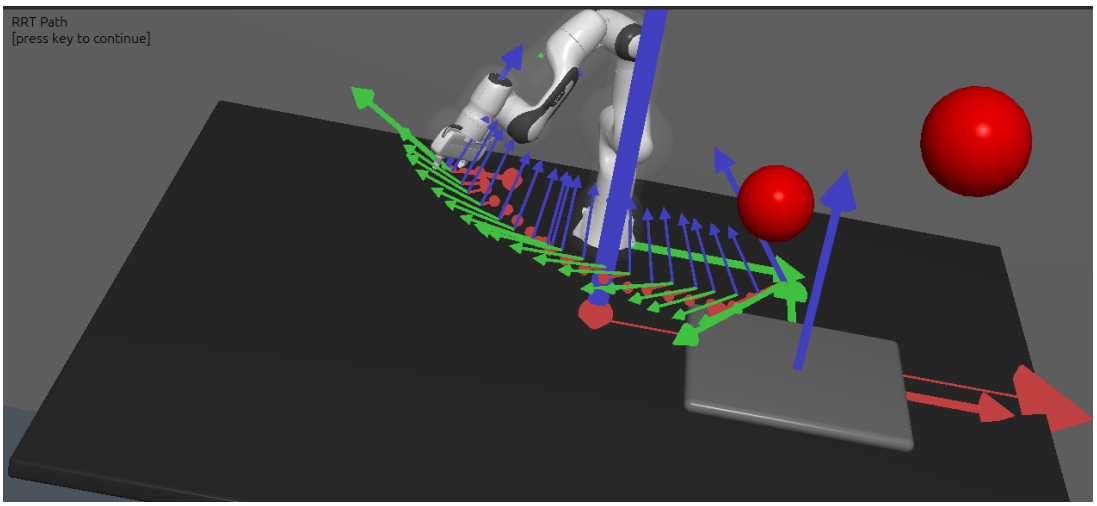

Figure 10: RRT-based path planning

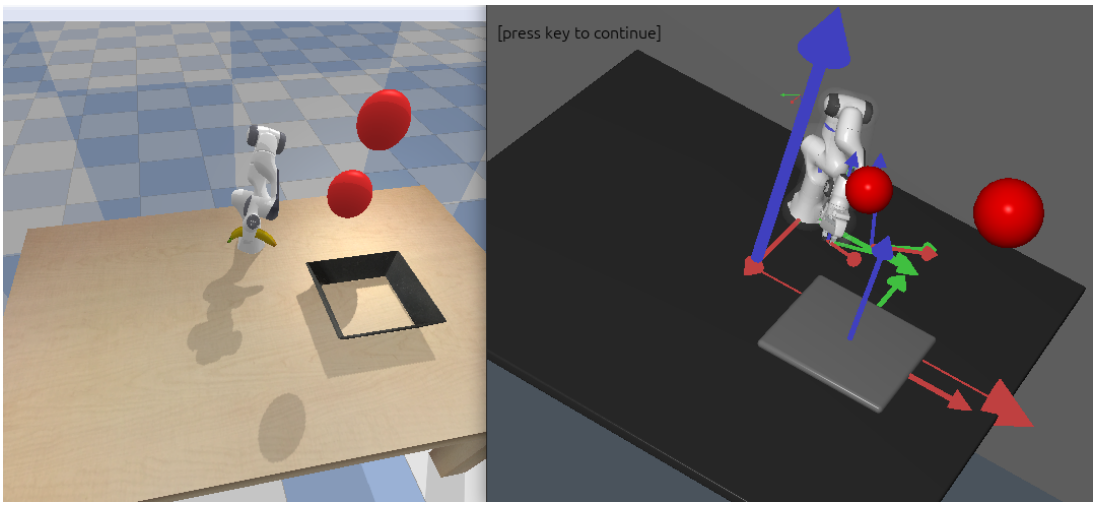

Figure 11: Collision detection : The left image shows the robot in real time, while the right image illustrates a potential configuration that could result in a collision with an object

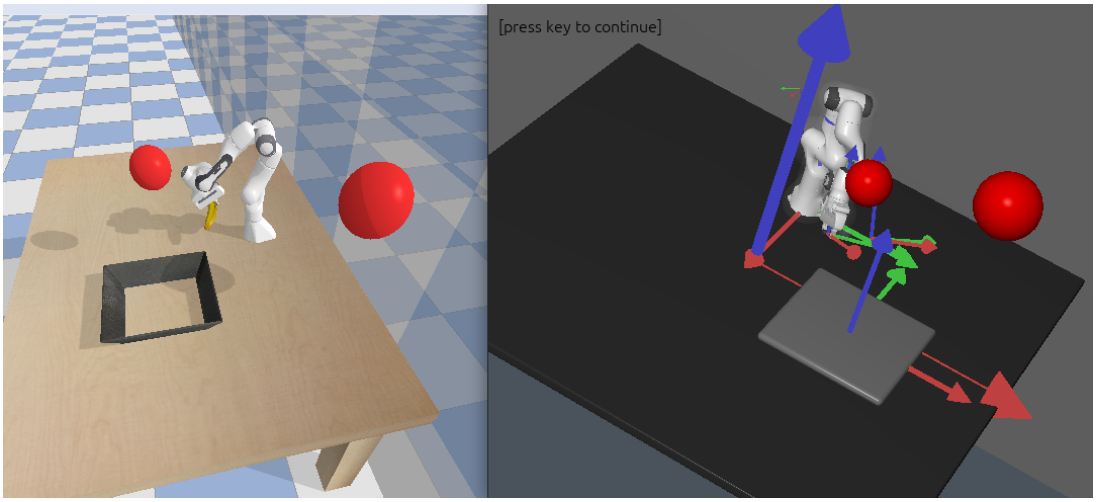

Figure 12: Retreat policy in action: The robot retreats to a sampled, safe configuration to avoid collision.

In many cases this algorithm is able to avoid obstacles and reach the goal position. Failure could happens when the sampled position is still too close to the obstacle and it is unable to detect it in right time after after as replan_freq is high for these scenarios.

3 Results

We divide our results section into two parts: evaluation of all objects with and without obstacles.

Across all experiments, we record three primary metrics: overall success rate, grasp success rate, and exception (failure) rate.

First, a task is classified as successful if the target object is located within a bounding box defined by the tray. Due to the bounding box criteria, some cases might be considered successful even when the object is very close to—but slightly outside—the tray boundaries.

Second, the grasp success rate specifically evaluates whether the gripper successfully grasped the object. Success in this metric is determined by measuring whether the gripper's open width exceeds a defined threshold after retreating. However, it is important to highlight that this measure can be overly optimistic, as it may count suboptimal grasps that do not necessarily secure the object.

Finally, the crash rate tracks program crashes or failures. Crashes can result from various causes, such as collisions with obstacles. Most commonly, these failures manifest through a "Not connected to physics server" error. The exact root causes of this specific error remain unclear and require further investigation.

3.1 No Obstacles

In Table 1 we present the results for all objects over 10 runs each with no obstacles. In this setting, our approach seems to be reliable for most objects, while there are difficulties for some shapes.

Specifically very small and lightweight objects are challenging. These objects are difficult to detect and grasp accurately and can easily shift position. This becomes problematic, for example, when the estimated grasp pose is imperfect, potentially causing the object to be moved out of the workspace. An example is the YcbStrawberry object, indicated by the 0% success rate, only 50% grasp rate, and a notably high 50% crash rate.

In contrast to that, we can reliably solve the task for medium sized objects with a rectangular or cylindric shapes. Examples are the YcbPottedMeatCan or the YcbMasterChefCan with 80% and 100% success rate respectively.

A recurring problem in scenarios without obstacles involves objects slipping from the grasp due to dynamic motions during robot movements. Adjusting the position gain to reduce sudden movements has partially mitigated this issue, but it still occurs frequently. A commonly employed solution is to apply a gear constraint to the gripper along with increased lateral friction. However, since this approach modifies the simulation's dynamics, we opted not to incorporate this adjustment into our final solution.

3.2 Obstacles

In Table 2 we show the results for all objects over 5 runs each with obstacles. Considering the increased replanning and collision checking overhead, we decided to evaluate this setting with fewer iterations due to resource constraints.

Overall it can be seen that obstacles pose a much harder problem and the robot is less reliable in achieving the goal. Still, our approach is also able to achieve this task for many objects.

Object	Grasp Success Rate (%)	Goal Success Rate (%)	Crash Rate (%)
YcbTennisBall	100.00	50.00	0.00
YcbCrackerBox	100.00	60.00	0.00
YcbPottedMeatCan	100.00	80.00	0.00
YcbBanana	100.00	40.00	0.00
YcbPear	90.00	20.00	10.00
YcbPowerDrill	100.00	80.00	0.00
YcbTomatoSoupCan	100.00	40.00	0.00
YcbScissors	100.00	70.00	0.00
YcbMustardBottle	70.00	60.00	30.00
YcbChipsCan	90.00	30.00	10.00
YcbMasterChefCan	100.00	100.00	0.00
YcbMediumClamp	100.00	50.00	0.00
YcbGelatinBox	70.00	60.00	30.00
YcbHammer	100.00	70.00	0.00
YcbStrawberry	50.00	0.00	50.00
YcbFoamBrick	80.00	30.00	20.00

Table 1: Metrics for YCB Objects without Obstacles (10 runs)

Object	Grasp Success Rate (%)	Goal Success Rate (%)	Crash Rate (%)
YcbTennisBall	60.00	0.00	100.00
YcbCrackerBox	60.00	20.00	80.00
YcbPottedMeatCan	60.00	40.00	80.00
YcbBanana	100.00	40.00	20.00
YcbPear	100.00	40.00	20.00
YcbPowerDrill	60.00	20.00	80.00
YcbTomatoSoupCan	80.00	20.00	80.00
YcbScissors	80.00	40.00	40.00
YcbMustardBottle	100.00	60.00	80.00
YcbChipsCan	80.00	20.00	80.00
YcbMasterChefCan	60.00	40.00	60.00
YcbMediumClamp	80.00	0.00	100.00
YcbGelatinBox	80.00	20.00	40.00
YcbHammer	100.00	60.00	40.00
YcbStrawberry	40.00	0.00	60.00
YcbFoamBrick	80.00	0.00	40.00

Table 2: Metrics for YCB Objects with Obstacles (5 runs)

The much higher crash rate can be explained by obstacle collisions and the mentioned physic engine error, which occurred more often than in the setting without obstacles. Note, that a run can still be successful if the program crashes, as obstacle collisions also happen in the resting phase after the gripper already released the target object into the goal.

As the grasping stage is less affected by adding obstacles, the results here are comparable to the previous section.

Finally, the slipping failure is even more problematic here, as the robot needs to avoid fast moving obstacles, which induces more drastic motions.

4 Conclusion

Overall, our pipeline operates in both obstacle-free and dynamic obstacle scenarios. However, we observed some failure cases with certain objects that may slip from the gripper or collide with the tray while reaching the goal.

Our approach leverages nearly all of the techniques introduced in the course. In particular, we use point clouds both to generate grasp candidates and to estimate obstacles. Moreover, our system implements multiple grasp retries in simulation, ensuring that the task can be completed even if the initial grasp attempt fails. Once a grasp is successful, we use a global planner (RRT) to compute a path, and then employ a custom local replanning algorithm to further avoid collisions.

For future work, we plan to improve the speed of local replanning for collision avoidance. Currently, it is the slowest component of our system because it requires continuous tracking and repeated execution of RRT to identify a new safe path. We also aim to reduce the false-positive rate in collision detection by incorporating the obstacles' motion direction, which could further expedite local planning.

Additionally, we encountered failure cases where objects slipped even after a successful grasp. This issue appears to stem from insufficient friction between the object and the gripper, combined with rapid end-effector movement. Although reducing the position gain has shown some improvement, further fine-tuning is required.

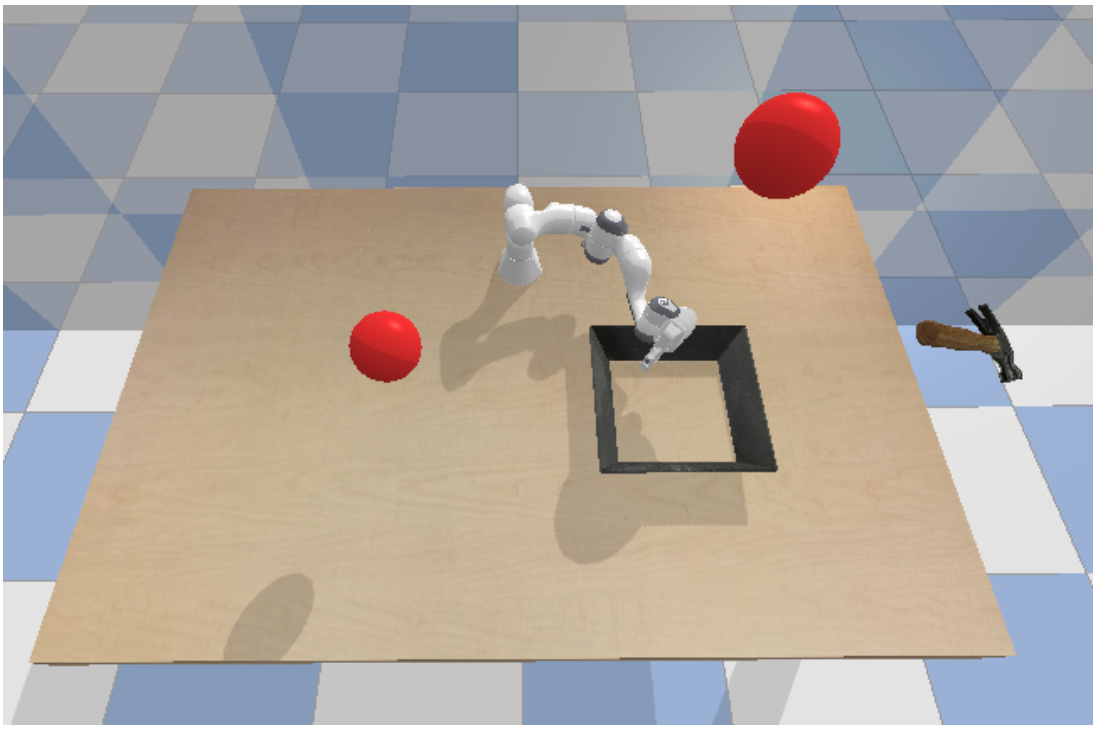

Figure 13: Hammer slipped from the gripper and overshoots

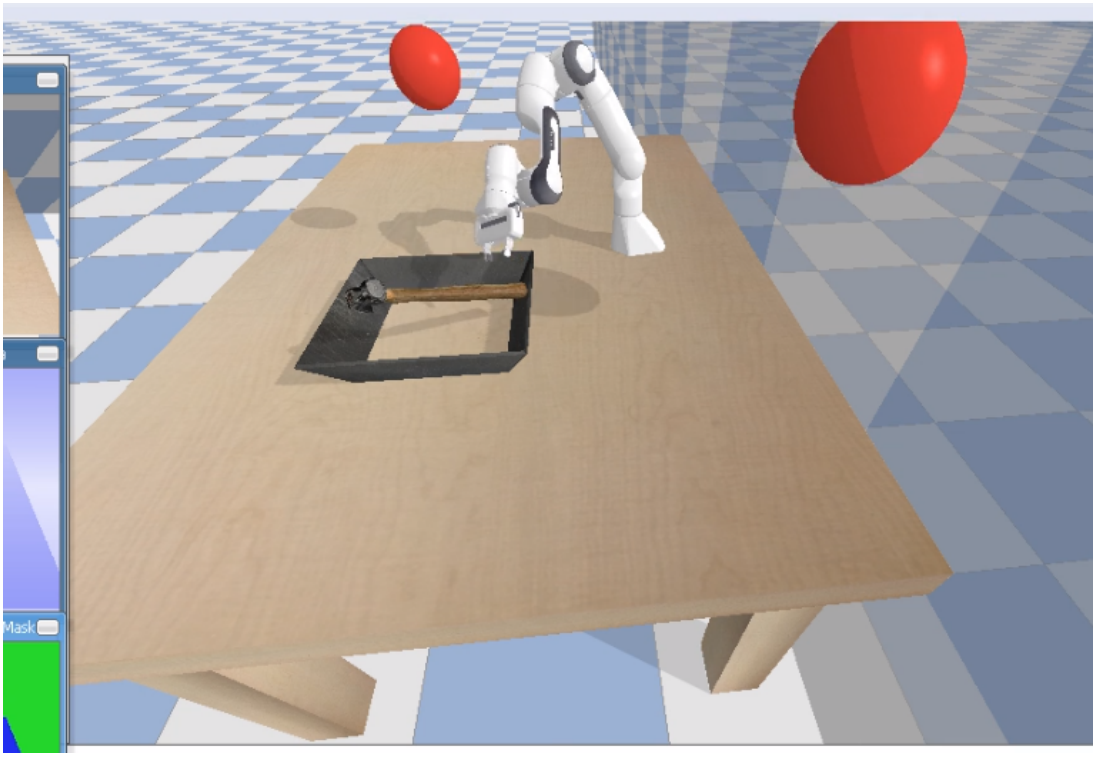

Figure 14: Task completed! End of a successful run

5 Note

Below are several key code sections to note:

Default Object and Goal Positions

```
default_obj_pos: [0.0, -0.65, 1.40]
default_goal_pos: [0.65, 0.55, 1.24]
```

Listing 1: Default positions for the object and goal.

YCBObject Class Constructor

```
class YCBObject:
   ...
   def __init__(self,
             obj_name: str,
             position: Tuple[float, float, float] = (1.0, 0, 1.31),
             orientation: Tuple[float, float, float] = (0, 0, 0),
             scaling: float = 1.5): # NOTE: changed; scale can be set to 1.0
      self.obj_name = obj_name
      ...
      # scaling = 1.0 # 1.5 times normal
```

Listing 2: Constructor of the `YCBObject` class. Note the scaling adjustments.

Table 3: Updated Object Scales

Object	Scale Factor
YcbTennisBall	1.0
YcbCrackerBox	1.0
YcbPottedMeatCan	1.0
YcbBanana	1.5
YcbPear	1.0
YcbPowerDrill	1.0
YcbTomatoSoupCan	1.0
YcbScissors	1.5
YcbMustardBottle	1.0
YcbChipsCan	1.0
YcbMasterChefCan	1.0
YcbMediumClamp	1.5
YcbGelatinBox	1.5
YcbHammer	1.5
YcbStrawberry	1.5
YcbFoamBrick	1.0

Robot Position Control Function

```
def position_control(self, target_positions, pos_gain=None):
   if pos_gain is None:
      p.setJointMotorControlArray(
         self.id,
         jointIndices=self.arm_idx,
         controlMode=p.POSITION_CONTROL,
         targetPositions=target_positions,
      )
   else:
      p.setJointMotorControlArray(
         self.id,
         jointIndices=self.arm_idx,
         controlMode=p.POSITION_CONTROL,
         targetPositions=target_positions,
```

```
        positionGains=[pos_gain] * len(self.arm_idx)
    )
```

Listing 3: Position control function in `Robot.py`. This function adjusts the end-effector's movement using an optional position gain.

6 Milestones Achieved

In this work, we have successfully reached all milestones:

1. **Object detection and pose estimation**
2. **Arm movement to the object**
3. **Successful grasping of the object**
4. **Arm movement with the object to the goal position (without obstacles)**
5. **Detection and tracking of obstacles**
6. **Full pick-and-place execution with obstacles present**

7 Contribution

1. **Perception: David**
2. **Grasping: David, Ansh**
3. **Control: Ansh, David**
4. **Tracking: Ansh**
5. **Planning: Ansh, David**