Report Intelligent Robotic Manipulation

TECHNISCHE UNIVERSITÄT DARMSTADT

Dierking, Magnus and Lippert, Jonathan
March 20, 2024

This is our final report for the voluntary semester project of the Intelligent Robotic Manipulation lecture. Along this written report, we submit screencasts of the simulation in the videos file and documented code.

1 Control

For the first task of the semester project, we develop a control logic for the given robot. The given setting is a $m = 3$-dimensional taskspace and a franka robot with 11 joints, of which 9 control the arm, 2 the gripper and the remaining one represents the end-effector. As a foundation for our control solution, we implemented the inverse kinematic control based on the transpose $\mathbf{J}^T \in \mathbb{R}^{n\times m}$ and the pseudoinverse $\mathbf{J}^\dagger \in \mathbb{R}^{n\times m}$ of the jacobian $\mathbf{J} \in \mathbb{R}^{m\times n}$. While both yield functioning control algorithms, the pseudoinverse proves to be more useful for the project, since it allows us to incorporate secondary objectives.

1.1 Nullspace Control

The Franka robot given in this task is redundant and we want to leverage this to our advantage. Because the redundcany leads to infinitely many possible configurations for any goal position in task space, we can impose additional control commands in the nullspace of the pseudoinverse. For this, we employ an orthogonal projection operator realized via the matrix

$$\Pi := \mathbb{1}^{n\times n} - \mathbf{J}^\dagger\mathbf{J}, \qquad \Pi \in \mathbb{R}^{n\times n}. \tag{1}$$

Given any vector $\mathbf{q} \in \mathbb{R}^n$ in joint space, the product yields $\Pi\mathbf{q} = \mathbf{q} - \mathbf{J}^\dagger\mathbf{J}\mathbf{q} \in \mathbb{R}^m$. This way, the components affected by the pseudoinverse are projected away, leaving only those in its nullspace. We can therefore generate additional control commands by adding the term $\beta(\mathbb{1}^{n\times n} - \mathbf{J}^\dagger\mathbf{J})\mathbf{x}$ with a step width $\beta \in \mathbb{R}$ to the inverse kinematic control based on the pseudoinverse without affecting the primary control objective of following the path. For the additional input $\mathbf{x}$ we choose the gradient $\nabla_\mathbf{q} C = \frac{\partial C}{\partial \mathbf{q}}$ of a cost function

$$C = \sum_j \sum_{i=1}^{n} (f_i(\mathbf{q}) - \mathbf{o}_j)^T (f_i(\mathbf{q}) - \mathbf{o}_j), \tag{2}$$

where $\mathbf{o}_i$ is the position of an obstacles centre and $f_i(\mathbf{q})$ denotes the forward kinematic of a joint i. We reach the control-law

$$\dot{\mathbf{q}} = \alpha\mathbf{J}^\dagger\Delta\mathbf{x} + \beta(\mathbb{1}^{n\times n} - \mathbf{J}^\dagger\mathbf{J}) \cdot \nabla_\mathbf{q} C. \tag{3}$$

1.2 Evasive Maneuvers

The spheres in the given task move at fairly high speeds and change directions randomly. This requires us to dynamically replan the trajectory when facing an imminent collision. When the object moves into the same direction as the gripper, this can lead to situations where dynamic replanning of the objective gets stuck in long loops and keeps triggering a replanning phase. We therefore decided to incorporate an evasion maneuver into our solution. When a collision is imminent, we first try to evade the objects and do not trigger replanning of the path until we are no longer detecting close-by objects.
We implement an `evasive_maneuver` method that simply flips the logic of the nullspace control law we introduced above. Instead of primarily following the path while trying to optimize the cost function in the nullspace, we make the cost function the primary objective. The secondary objective now is to stay close to the goal while evading, without moving the arm too far back to the initial position. Mathematically, we modify the calculation of the velocity control input to

$$\dot{\mathbf{q}} = -\beta\mathbf{J}^\dagger\mathbf{J}\nabla_q C + \alpha(\mathbb{1}^{n\times n} - \mathbf{J}^\dagger\mathbf{J})\mathbf{J}^\dagger\Delta\mathbf{x}. \tag{4}$$

We find that in the given setting a moderate β of $0.2 - 0.4$ works best. For higher values, the sudden movement would often lead to the gripper dropping the object or the simulation breaking down because of the sudden movement.

2 Grasping

In order to grasp the banana, we use the given `sample_grasps` function to generate a set of grasps, see Figure 1. To determine which grasp to execute, we fall back to a heuristic: We choose the grasps whose translational component, i.e. the goal for the end-effector during the grasps is closest to our initial position. Because this would often yield the highest grasp on the banana, we then manually set the z component of the end-effector goal to a position slightly above the table or below the inital graps position in order to generate stable grasps. While this is by no means optimal, it is successful in generating grasps as in Figure 2. To execute the grasps, we

Figure 1: Exemplary ensemble of sampled grasp. We only depict the goal position of the end-effector obtained via the translational component of the grasp.

implement the `approach_grasp` method in the `robot` class, which relies on our controller from Section 1 to move the arm towards the grasp position. To control the pose, we use the end-effector orientation and the pybullet routine `computeDifferenceQuaterion`. We then transform the result to euler angles and compute the joint angles

$$\mathbf{q}^{(k+1)} = \mathbf{q}^{(k+1)} + \kappa \mathbf{J}_{ang}^{\dagger} \chi_{euler} \tag{5}$$

using the pseudoinverse of the angular Jacobian $\mathbf{J}_{ang}$. In order to open and close the gripper, we additionally implement `open_gripper` and `close_gripper` based on velocity control of only joints 9 and 10. The latter method additionally controls the forces applied to the object to make the grip more stable, particularly during the evasion maneuvers.

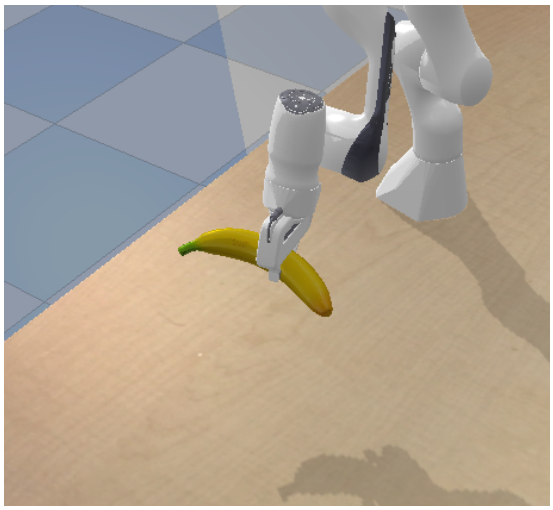
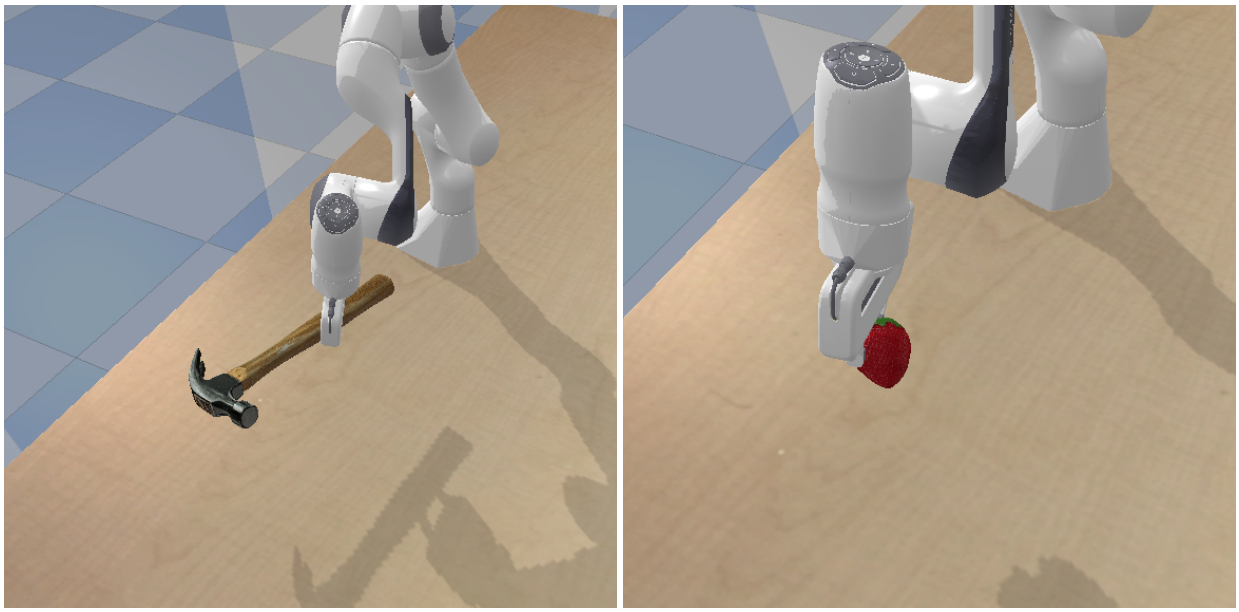


Figure 2: Exemplary successful grasp for the banana object.

2.1 Random Objects

The given banana object proves to be forgiving regarding the grasping task. In our experiments, other objects are significantly harder to grasp. For larger objects such as the hand-drill or the cornflakes, the gripper is physically not able to perform grasping because of

their size, while other such as the spam and tomato tins would often move after being spawned into the scene, which made grasping based on the camera data impossible. Only smaller objects or objects with similar shapes as the banana can be grasped successfully



(a) Exemplary grasp for the hammer object.

(b) Exemplary grasp for the strawberry object.

Figure 3: Collection of successful grasps on other object than the given banana.

with the heuristic. Some examples are depicted in 3.

3 Localization & Tracking

3.1 Method

In order to track the spheres we used the KCF tracker of the open-cv library. We additionally implemented various other trackers, but KCF yielded the best results. The tracker is based on kernels and correlations and must be initialized based on a Region Of Interest (ROI). We define this region using the color of the spheres in HSV color space and create a mask from the RGB image which drops all pixels that do not lie in the HSV color range. From this, we extract the desired contours and consequently the bounding boxes. A visualization of this process is depicted in Figure 4. These boxes are then used to initialize the multitracker instance, which consist of

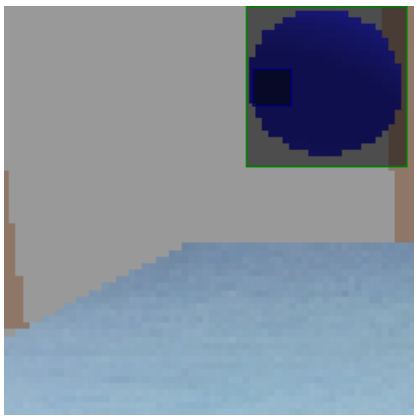


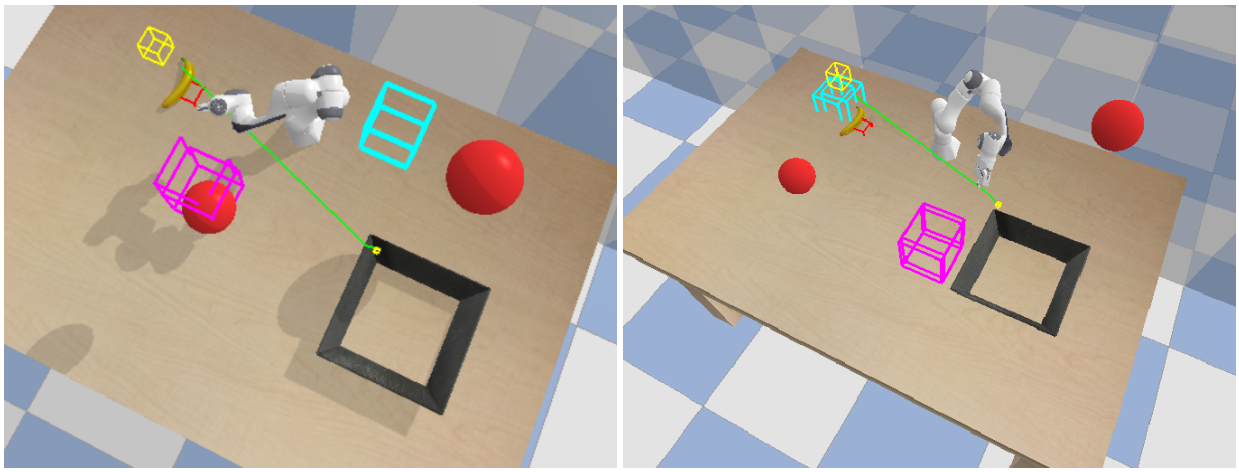
Figure 4: Bounding box based on ROI in HSV color space for one of the spheres.

one tracker for each object. In case the trackers lose their objects, the whole procedure is repeated to reinitialize the multitracker. This implementation is notably effective, especially given that, within the specified environment, one of the spheres may become obscured by the barrier wall, resulting in failures across all tracking methods attempted during our experiments. Consequently, we

opted to reinitialize the tracker at predetermined time intervals.
The position of the tracked objects is calculated by taking the center of the bounding boxes and transforming the position coordinates to the real world coordinates based on the camera intrinsic and extrinsic. For the transformation we have to take into account that openCV and pybullet (which is based on openGL) have different definitions for their camera intrinsic/extrinsic and coordinate systems. However, we found the depth value for spheres to be unreliable. We therefore opted to utilize the minimum depth value within the bounding box area as a more dependable alternative.
In order to improve the tracking we added a second custom cam, which observes the scene from a different point of view.

3.2 Evaluation

The above method leads to satisfactory tracking results for the smaller sphere. For reasons that are not clear to us, the second sphere has a constant offset in its tracked location. The situation is depicted in Fig. 5a. Furthermore, the tracker is not always fully reliable, especially when the spheres are close together and occlude each other in the image. In this case, the tracker only detects one object, as shown in Fig. 6. The open-cv visualizations in Figure 7 also reveal that the tracked bounding boxes are of good quality, which strengthens our suspicion that the depth values are problematic. We also submitted two videos where the constant offset and general



(a) Tracking has constant offset.

(b) Tracking fails completely.

Figure 5: Exemplary situations in which the implemented tracker had problems with constant offsets or failed completely because the larger sphere was occluded behind the wall in the preceding frames.

problem with the depth values becomes apparent.

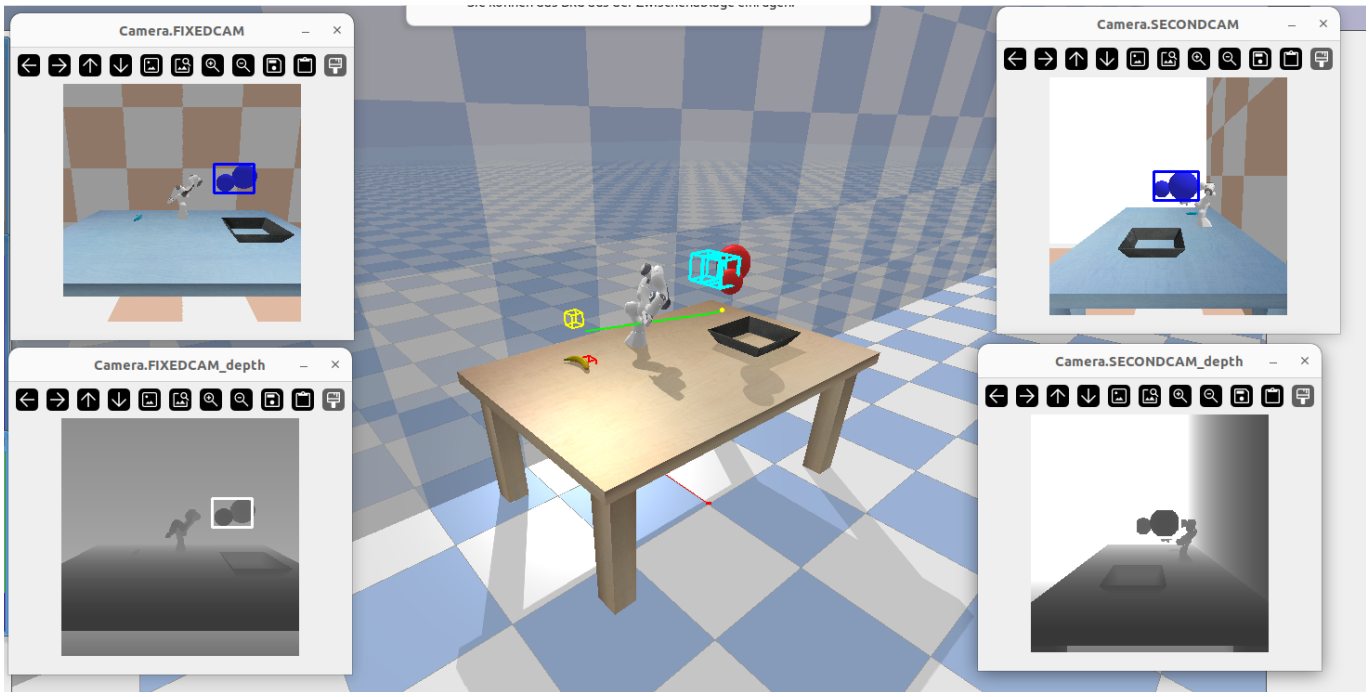


Figure 6: Tracker bounding boxes in camera image with one detected obstacle. The blue and pink bounding boxes depict the estimated 3D location of the tracker, with additional tracking results for the cameras in the smaller images.

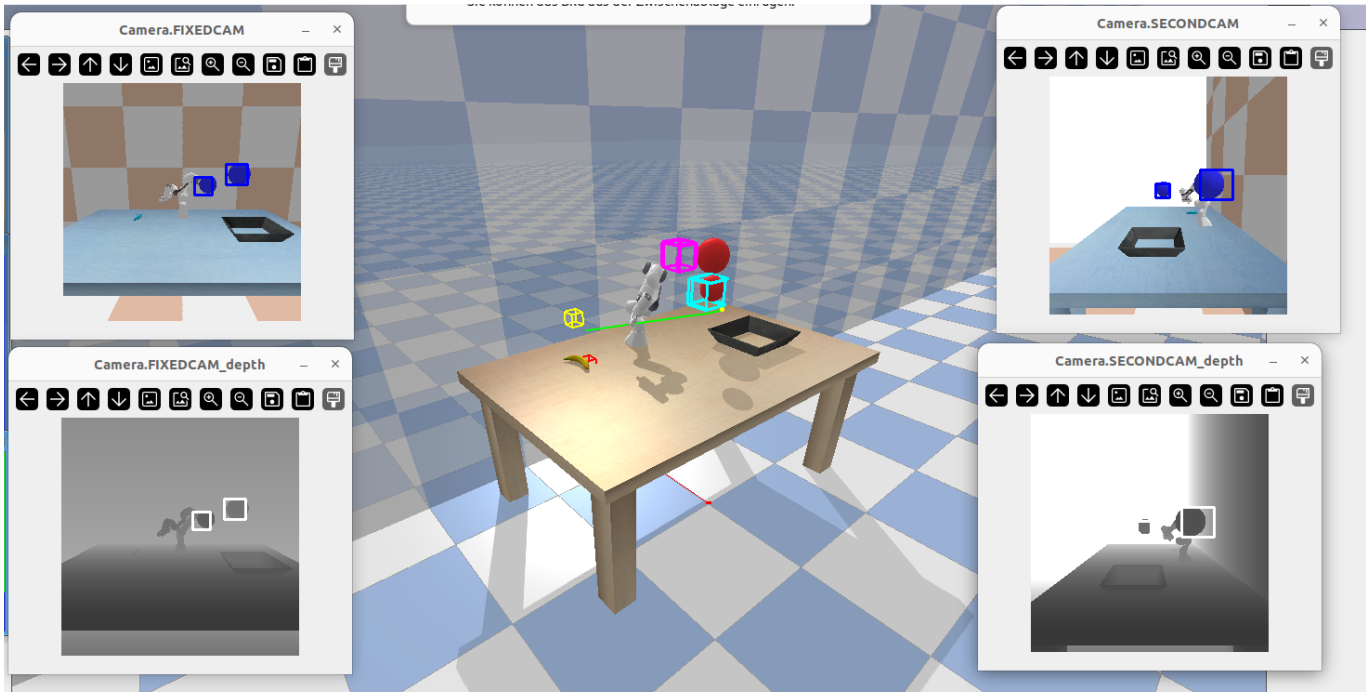


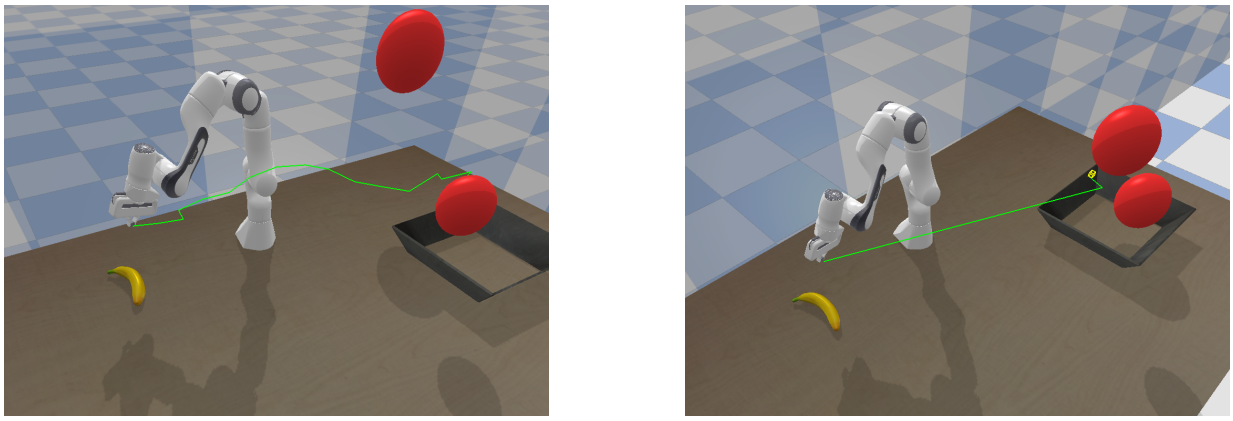
Figure 7: Tracker bounding boxes in camera image with two detected obstacles. The blue and pink bounding boxes depict the estimated 3D location of the tracker, with additional tracking results for the cameras in the smaller images.

4 Planning

We use a seperate `Planner` class to realize all the needed planning logic. The heart of the class is the `plan_trajectory` method that is used to perform planning via the RRT or RRT-Connect algorithms introduced in the lecture.

4.1 The Algorithms

Both algorithms are inspired by the implementation provided in the exercise. In order to speed up the vanilla RRT algorithm, we bias the algorithm towards the goal by using the goal position as the sampled node at every ℓ-th iteration.
For the RRT-Connect algorithm, we have to add a private `__merge_trees` method, because the trees are constructed in opposite directions. When a the paths meet each other, we let the tree starting from the initial position iterate through the other tree and append parent nodes until we reach the goal. This way, we can then extract the path using the same `backtrack` method that we use for the vanilla RRT. While sampling nodes for any algorithms, we exlude points that are inside a cylinder around the robots base to prevent paths that are too close to the arm itself. In order to complete the objective reliably, we need to assess whether any part of our robot is in danger of



(a) Exemplary path found by the RRT algorithm.

(b) Exemplary path found by the RRT-Connect algorithm.

Figure 8: Comparison of the implemented sampling-based planning algorithms in the given pybullet environment.

colliding with obstacles and then trigger a replanning step.

4.2 Checking for Danger

The planning process is solely based on the objects locations at a particular time step. As the objects move, we need to make the planner dynamic by replanning if collisions are imminent. We realize this by checking for an imminent collision of any part of the robot with an obstacles at every timestep and triggering a replanning step in this case. The collision detection is realized in the `check_danger` method of the `Robot` class. For every obstacle we randomly sample vectors in $\mathbb{R}^3$, normalize them and add them to the central position of the object. This way, we create points $\mathbf{s}$ on the spheres surfaces for which we can easily verify if they lie in- or outside of an ellipse via

$$|\mathbf{F}_1 - \mathbf{s}| + |\mathbf{F}_2 - \mathbf{s}| \leq |\mathbf{F}_1 - \mathbf{F}_2| + \epsilon. \tag{6}$$

We can then use the positions of the robots joints as the foci $\mathbf{F}_1, \mathbf{F}_2$ and add a custom margin ϵ to surround every link with an ellipsoid. For visualization purposes, we sample 70 points inside each ellipsoid. These samples are depicted as yellow boxes in Figure 9. Note that some of the boxes are inside the robot. If we have an estimate about an objects velocity, we can incorporate this into our algorithm by additionally checking for collisions with ghost objects that are translated by this velocity vector multiplied with the time step of the simulation. For the end-effector, we compute the euclidean distance to the spheres center point and check if it is smaller than the object radius plus a safety margin that can be set depending on the grasped object.

4.3 Replanning

If a collision is detected via the procedure explained in the section above, we trigger an evasion phase to avoid the impending collision and –once the danger no longer persists– recompute a feasible path to the goal from the current end-effector position. For the initial path that we compute before grasping the object, the RRT-Connect algorithm proves to be the better alternative, because the first part of the path is rarely close to the spheres. In contrast, the RRT can sometimes compute convoluted paths that guide the robot directly into the trajectory of the smaller sphere. However, once we are closer to the tray and perform replanning post-evasion, the bias of

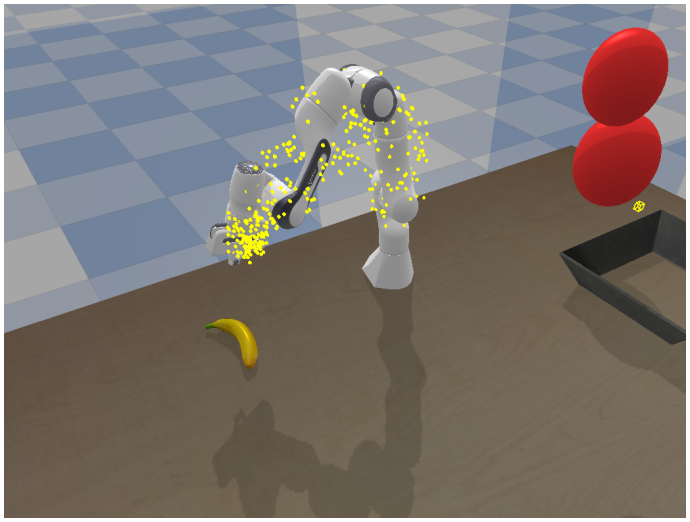


Figure 9: Samples inside the ellipsoids around each of the robots links.

RRT-Connect towards direct routes often yields higher computation times, particularly when the spheres obstruct this path. Although the reason behind RRT's advantage in this context are not clear to us yet, empirical evidence suggests its superior efficacy in this situation.

4.4 Path Smoothing

In order to obtain smoothed trajectories, we use B-spline curves of order d to construct polynomials of order $k-1$ that use the trajectory $\boldsymbol{\tau} = \{\mathbf{p}_0, ..., \mathbf{p}_p\}$ of the RRT step as control points

$$\mathbf{r}(x) = \sum_{i=0}^{p} \mathbf{p}_i B_{i,d}(x), \quad x \in [0,1]. \tag{7}$$

Given a knot vector $\mathbf{t} = (t_0, ..., t_{d+p})$ with elements $t_i \in [0,1]$ that is sorted ascending, the used B-splines are defined for $d = 0$ as

$$B_{i,0}(x) := \begin{cases} 1 & \text{if } t_i \leq t < t_{i+1}, \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

and for any higher order recursively via

$$B_{i,d}(x) := \frac{x - t_i}{t_{i+d} - t_i} B_{i,d-1}(x) + \frac{t_{i+d+1} - x}{t_{i+d+1} - t_{i+1}} B_{i+1,d-1}(x). \tag{9}$$

As we want to interpolate the initial and goal position for any trajectory, the knot vector is chosen such that $t_0 = ... = t_d = 0$ and $t_p = ... = t_{d+p} = 1$. The smoothing is implemented in the `smooth_path` method of the `planner` class. However, we do not use the path

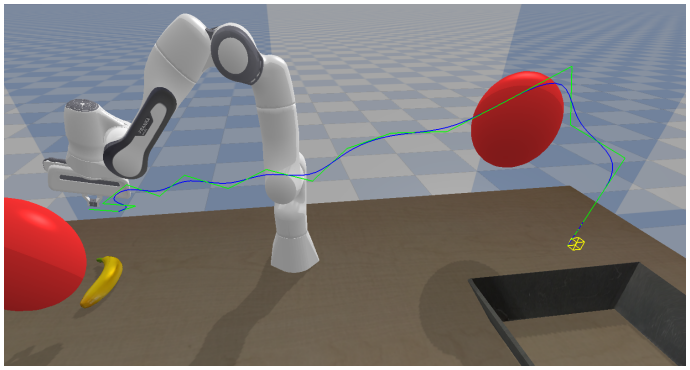


Figure 10: Path smoothing for a path found via the RRT algorithm (green) using a B-spline polynomial of order 4 (blue).

smoothing for the other tasks, as we found the upside to be minimal while the smoothing after any replanning step significantly increases computational demand. Since we update the target point on the robots trajectory when it is a fixed distance away from it, we naturally incorporate a very simple path smoothing even without the B-splines.

5 Objective

Putting everything above together in order to put the banana in the tray, we use an enumeration to implement the main loop. The members encode the stage we are currently trying to complete, namely

- GRASPING - In this mode, we ignnore the objects and the pre-computed path and focus solely on grasping the object. In the gievn scenario, the spheres cannot reach positions at which they could interfere with the robot arm while grasping or apporaching the object. We use our control class based on the pseudoinverse from Section 1 to reach the grasp we extracted as decribed in Section 2. Once the distance between our end-effector and the grasp goal is below a threshold, we close the gripper for a fixed number of steps.
- RETRACTING - Once the gripper is closed, we move back to the initial position to reach the starting point of our pre-computed path to the goal. This approach proved to be advantageous during our project, because starting directly from the grasping position would often lead to the object touching the table and the robot losing its grip.
- NAVIGATING When we arrive back at the initial starting position, we begin to follow the computed path to the tray. During this phase, we employ the control based on the pseudoinverse again. In this stage, we also start to check for collisions at every time-step using the method introduced in Subsection 4.2.
- EVADING If we detect an imminent collision, we switch from the regular control to the evasion maneuver of Subsection 1.2 to reach a safe position and initiate replanning.
- REACHING After replanning, we continue to approach the goal position. Since we are now in reach of the spheres and collisions are possible, we switch to the nullspace control in order to move the arm out of harms way as good as possible. Once we are above the tray, we open the gripper.

An exemplary successful run of the whole pipeline is shown in the video we submitted along with the code and this report.

5.1 Notable Problems and Workarounds

The approach above yielded successful iterations of the simulation; However, several challenges we encountered during this project either impact the success probability of the iteration or required tricks to overcome. The notable ones are listed below.

1 While the given grasping method would always give us around $15 - 20$ possible grasps, some of them were of poor quality. Since we could not reproduce the filtering by grasp quality we used in the exercise, we simply used the grasp whose translation goal was closest to the end-effector. This way, we could at least exclude the grasps on the table surface. Additionally, the object was initiated relatively centered in front of the robot, which is why this method often yielded grasps in the middle of the banana.

2 Another problem occurs when the grasp is successful, but of poor quality. In this case, the robot would sometimes barely hold in to the object and loose its grip during evasion maneuvers or when touching the goal tray with it.

3 The simulation itself would often behave unpredictable. This was most notable during grasping: The gripper would sometimes just go through the banana or get stuck inside of it, which always rendered the iteration unsuccessful. In addition to that, sometimes the physics of the dropped object was not correct: instead of falling straight down or in a slight arc when being dropped during movement, it would sometimes just drop into random directions and miss the tray.

4 The evasion in combination with the tracking implementation was very unreliable because of the issues with the depth image yielding an offset and one sphere being occluded by the wall in certain time steps. In order to demonstrate our full pipeline, we extracted the obstacles position from the simulation, thereby assuming perfect tracking.

5 Although the linear part of the grasp control reduced the error for the translational part of the grasp and the pose control did so for the pose error, we were not able to implement a control scheme that converged to the desired pose. Apart from cases where the initial pose is close to the goal pose, the controls would work against each other when close to the goal, leading to the robot creeping towards the object (the translational control usually had a larger step size). For the grasps displayed in 2, we used the translational part with a very small parameter for the orientation part, resulting in a non-zero orientation error when the gripper closes.

5.2 Further Work

In this section, we provide a brief overview of possible improvements to the submitted work, which we could not integrate due to time constraints.

- The grasping-related problems (1 and 2) could be solved by using the same approach as in the given grasping notebook. For this, we would need to adapt the creation of the `tsdf` objective to our framework.
- The problems we encountered with the tracking seem to be related to the mapping from camera to simulation coordinates. We specifically suspect the depth image to be the reason for the offsets. Spherical object are notoriously hard to track based on depth, because there is only a single infinitely small closest point from any angle.
- We want to add a Kalman Filter to the tracker in order to improve the estimates and to better deal with imperfect bounding boxes. However, until the problem with the constant offset persists, a Kalman filter would not yield a significant improvement.