2D Deterministic Path-based Dynamic Object Grasping

Github repository: 2D_dynamic_object_grasping, Demo Video

Saurav Dosi, Aditya Kulkarni, and Feroz Hatha

***Abstract*— This work addresses the challenge of accurately and efficiently grasping an object moving along a deterministic path using a manipulator robot. The project focuses on estimating the graspable object's future pose to enable successful grasping. A control system is designed to ensure the algorithm accurately determines the interception point based on the object trajectory and execute the grasp within the estimated time.**

I. INTRODUCTION

Grasping dynamic objects accurately and efficiently remains a significant challenge in robotics, particularly when these objects follow deterministic paths. This project aims to address this challenge by developing a control system that enables a manipulator robot to predict and adapt to the future poses of moving objects. By leveraging feedback from the object's trajectory, the robot can plan an intercept with the moving object, ensuring successful grasping operations.

While practical applications of this work lie in industry settings where a robot is required to pick up objects moving on a conveyor belt or in efficient mobile manipulation, the problem is challenging to deal with, especially the strategy to calculate the interception time. Working on this problem would enable us with essential skills in 3 important stages of robotics: Perception, Planning and Control.

The foundation of this project is based on extending existing methodologies, specifically the Grasping Trajectory Optimization with Point Clouds (GraspTrajOpt) technique developed by Xiang et al [1]. This approach, originally designed for static object manipulation, will be adapted for dynamic scenarios. The Fetch robot, known for its compatibility with ROS and the PyBullet simulator, serves as the primary platform for simulating and testing our solutions.

The project is structured into three progressive phases. Initially, we simplify the problem by assuming linear object motion akin to a conveyor belt scenario, focusing on grasping without slipping. Subsequent phases introduce more complex paths such as circular and sinusoidal trajectories and incorporate advanced kinematic adjustments. Finally, we we integrate a perception system that determines the position of the object using an egocentric camera which then can be used to model the motion of the object.

II. RELATED WORK

The paper by Akinola, Xu et al. [2] presents a dynamic grasping framework that is reachability-aware and motion-aware. Specifically, it models the reachability space of the robot using a signed distance field, enabling quick screening of unreachable grasps. Additionally, it trains a neural network to predict grasp quality based on the target's current motion, allowing for real-time filtering of a large grasp database. A seeding approach for arm motion generation utilizes solutions from the previous time step, quickly generating a new arm trajectory that remains close to the previous plan and prevents fluctuation. The authors implement a recurrent neural network (RNN) for modeling and predicting object motion.

Human-robot object handovers have been an actively studied area of robotics over the past decade; however, very few techniques and systems have addressed the challenge of handing over diverse objects with arbitrary appearance, size, shape, and deformability. The paper by Yang et al. [3] presents a vision-based system that enables reactive human-to-robot handovers of unknown objects. The approach combines closed-loop motion planning with real-time, temporally consistent grasp generation to ensure reactivity and motion smoothness. The system is robust to different object positions and orientations and can grasp both rigid and non-rigid objects. The generalizability, usability, and robustness of the approach are demonstrated on a novel benchmark set of 26 diverse household objects, a user study with six participants handing over a subset of 15 objects, and a systematic evaluation examining different ways of handing objects.

Grasping moving objects, such as goods on a belt or living animals, is an important but challenging task in robotics. Conventional approaches rely on a set of manually defined object motion patterns for training, resulting in poor generalization to unseen object trajectories. This work by Wu [4] introduces an adversarial reinforcement learning framework for dynamic grasping, named GraspARL. Specifically, it formulates the dynamic grasping problem as a "move-and-grasp" game, where the robot picks up an object while an adversarial mover attempts to escape. The framework allows for the auto-generation of diverse moving trajectories, resulting in improved generalization capabilities for the robot. Empirical results from simulations and real-world scenarios demonstrate the effectiveness of the proposed method.

III. METHOD

A. Simulation Setup

In this project, we utilized Robot Operating System (ROS) integrated with Gazebo to simulate a robotic manipulation task involving a Fetch robot. The simulation scene consists of a Fetch mobile manipulator robot, a table/conveyor belt, and a cube moving on the table/conveyor belt. The objective of Phase 1 is to intercept and grasp the moving cube using the robot's arm.

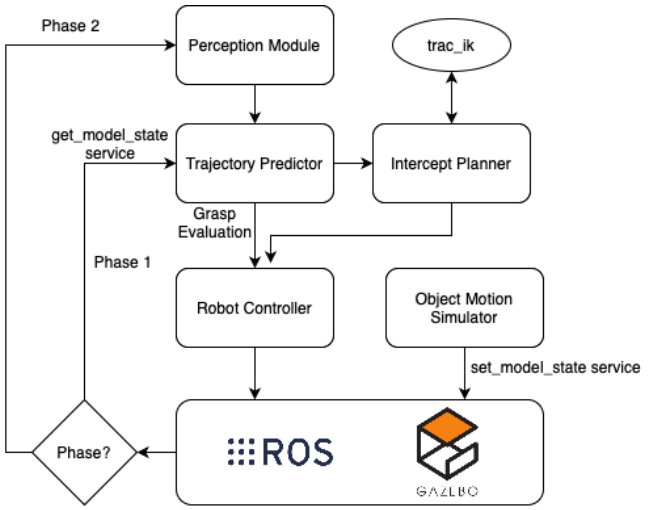

Fig. 1. System Flowchart

We tried two different approaches to simulating uniform rectilinear motion of the cube. The first approach involves moving the cube on a straight line along the surface of a table by applying a fixed velocity. To achieve this, we made use of the pre-existing setup defined by the simple_grasp.launch and simple_pick_place.sdf files that come with fetch_gazebo package. This makes sense because we're able to generate a setting where we have a cafe table on top of which a cube is placed, and the fetch robot is located adjacent to it with its arm positioned ready to grasp the cube. To simulate uniform motion of the cube on the table top, we developed a python script that creates a ROS node which continuously moves the cube in Gazebo in a straight line. Using the rospy library, we create a ROS node and set up a publisher to the '/gazebo/set_model_state' topic. We then configure the initial position of the cube on the table as well as the magnitude of its velocity (set to 0.01 m/s). In the main loop, we repeatedly create a new instance of the ModelState message in each iteration that describes the complete state of the cube model in Gazebo. We then set 3D position of the cube's center in the world coordinate frame (which in the beginning would be the initial position of the cube). Since the motion is only along the y axis and is uniform, we specify that there's no rotation involved. Also, we update only the y coordinate by adding to it the magnitude of velocity. Finally, we reset the position of the cube to it's initial configuration once it approaches the edge of the table.

However, we realized that relative to the time that it takes for the robot to grasp the cube (assuming a reasonable velocity for the cube), the length of the cafe table was too short. We therefore had to replace the cafe table with a longer table. This was realized by developing a URDF file for the table and modifying the launch file to incorporate this table. In the urdf file, we define the necessary materials required to characterize the appearance of the table and add links corresponding to the world, table top, and table legs. We then add fixed joints to connect the world and the table legs to the table top. In the launch file, we load the created urdf file and spawn the setting in Gazebo. Finally, we include the fetch robot's launch file to add it to the setting. To simulate the motion of the cube in this setting, we use the same Python script as before (taking into account the dimensions of the new table). The setting looks as shown in the figure below.

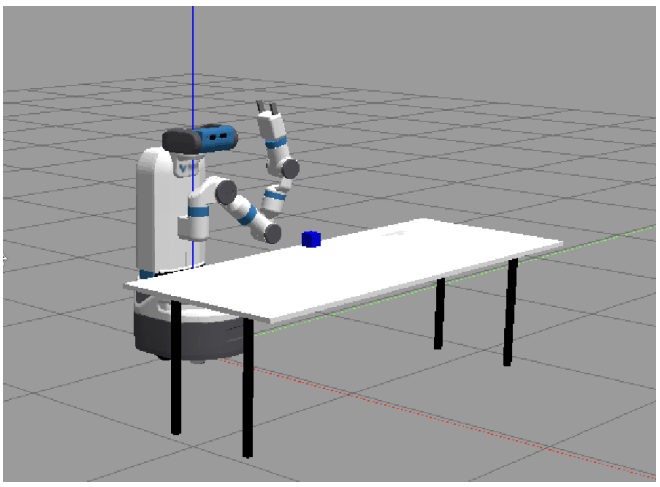

Fig. 2. Simple grasp setup with the cafe table replaced

With this basic approach, we were able to successfully simulate uniform rectilinear motion of the cube. However, we observe that the cube experiences high frequency oscillations in the vertical direction while moving uniformly along the y axis. This apparently is a common issue in Gazebo Physics simulation which could occur due to several reasons. The default contact parameters between the cube and the table surface is a common culprit. The key parameters that affect this include friction coefficients, contact stiffness, and contact damping. Also, the script we've developed uses teleportation (setting absolute positions) rather than physics-based movement, which might explain why we observe oscillations in Gazebo - the physics engine is constantly trying to reconcile the forced position changes with physical constraints like gravity and contact forces. We tried to mitigate this issue by setting the frictions coefficients defined in the URDF of the table to extremely small values (or zero altogether). Although this fixed the issue to some extent, we were not able to get rid of the oscillations completely.

However, a bigger issue we observed with this approach was that the cube continues to be applied the constant velocity even after being grasped by the robot. As a result, it becomes challenging for the gripper to hold on to the cube after being grasped. To address this issue, we realized that a true conveyor belt setup was needed to simulate the motion of the cube (because the cube isn't being applied a velocity - it moves by virtue of being on the surface of the conveyor). Because it is non-trivial to define the geometry of the conveyor belt and the links for the same, we used a pre-existing Gazebo conveyor belt plugin which spawns a (speed) configurable conveyor belt in Gazebo. We then

simply incorporate fetch into this setting. A figure of the same is shown below.

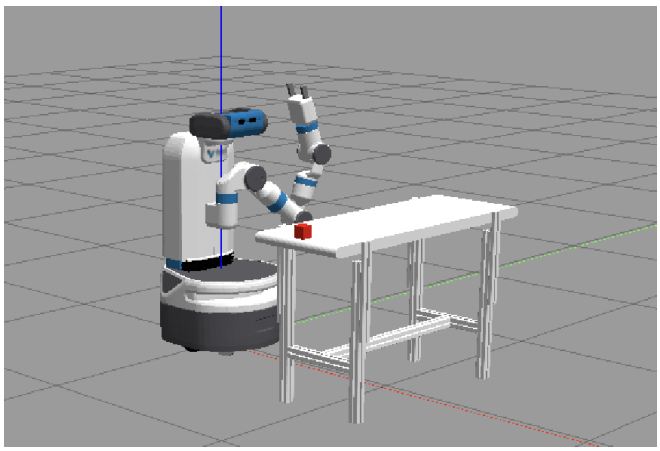

Fig. 3. Conveyor belt setup

With this setup, we solve both of the issues previously mentioned. Also, the speed of the conveyor belt can be adjusted through a rosservice call that sets the power of the conveyor belt. In ROS Gazebo simulations, the power of the conveyor belt often translates to a parameter that governs the velocity of the belt. The conveyor's velocity will be directly proportional to the power parameter as a result of which adjusting the power indirectly adjusts the velocity.

The cube moves along the y-axis in a linear trajectory with a constant velocity. The Fetch robot is initially positioned in a pre-grasp state, defined as being 0.5 units above the edge of the table. The pre-grasp position coordinates are set at (0.6, 0, 0.5), where the x-coordinate represents the horizontal distance from the robot to the table edge, the y-coordinate is aligned with the center of the table, and the z-coordinate indicates the height of 0.5 units above the table's surface. The trajectory planning for intercepting and grasping the cube is divided into two main steps:

1) Positioning Above Intercept Point: The robot first moves horizontally to align itself 0.5 units above the calculated intercept point along the y-axis where it can grasp the object.
2) Vertical Descent for Grasping: Once aligned, the robot moves vertically downwards to grasp and pick up the object.

To verify whether the grasp is successful, we check the position of the cube after attempting to lift it. If successful, the cube's position should approximately match (0.5 - height of the table) in z-coordinates, indicating that it has been lifted off from its original position on the table. For phase-1 evaluation, the cube was moved linearly along y-axis with a constant velocity of 0.1 units. Based on this and manual analysis of time required for robotic arm to move to a position as shown in Figure 4, a fixed inital y intercept coordinate was determined, and it was updated with each successful grasp.

In this project, we employed rospy, the Python client library for ROS, in conjunction with MoveIt for motion planning, time estimation, and trajectory execution during the grasping task. MoveIt was utilized to generate collision-free trajectories for the Fetch robot's arm, ensuring precise movements towards the moving cube. The planning process involved calculating the optimal trajectory from the robot's pre-grasp position to the intercept point, taking into account the cube's velocity and position on the table. Using MoveIt's built-in time parameterization tools, we were able to estimate the time required for each trajectory segment, ensuring that the robot's movements were synchronized with the cube's motion. This integration of rospy and MoveIt provided a robust framework for handling dynamic object manipulation tasks in a simulated environment.

B. Trajectory Prediction

To predict trajectory of the object moving on a deterministic path, we use a method called **Recursive Least Squares**. We take position co-ordinates of the moving object (x, y) as input from Gazebo simulation, update/train the RLS fitter separately for x(t) and y(t) and then predict the position for future time steps T. While our fitter can be a polynomial of any degree, we incorporate sines and cosines for circular and sinusoidal motion of the object for future work.

1) Objective: The goal of RLS is to minimize the weighted sum of the squared errors between the predicted output and the actual output. Given a linear model $y(t) = \mathbf{w}^T\mathbf{x}(t) + \epsilon(t)$, where:
   - $y(t)$ is the observed output at time t.
   - $\mathbf{w}$ is the parameter vector to be estimated.
   - $\mathbf{x}(t)$ is the input vector (features).
   - $\epsilon(t)$ is the error term.
2) Error Definition: The error $e(t)$ at time t is defined as:
$$e(t) = y(t) - \mathbf{w}^T\mathbf{x}(t)$$
   The objective is to minimize the cost function:
$$J(t) = \sum_{k=0}^{t} \lambda^{t-k} e(k)^2$$
   where λ (forgetting factor) controls the weight given to older observations.
3) Recursive Update The key steps in RLS involve:
   - **Initialization**: Start with an initial guess for the parameter vector $\mathbf{w}(0)$ and an initial covariance matrix $\mathbf{P}(0)$, often chosen to be large to allow flexibility in the initial estimates.
   - **Gain Calculation**: For each new observation, compute the gain vector $\mathbf{K}(t)$:
$$\mathbf{K}(t) = \frac{\mathbf{P}(t-1)\mathbf{x}(t)}{\lambda + \mathbf{x}(t)^T\mathbf{P}(t-1)\mathbf{x}(t)}$$
     This gain vector indicates how much to adjust the parameters based on the current error.
   - **Parameter Update**: Update the parameter vector:
$$\mathbf{w}(t) = \mathbf{w}(t-1) + \mathbf{K}(t)e(t)$$

- **Covariance Matrix Update**: Update the covariance matrix:

$$\mathbf{P}(t) = \frac{1}{\lambda}\left(\mathbf{P}(t-1) - \mathbf{K}(t)\mathbf{x}(t)^T\mathbf{P}(t-1)\right)$$

Algorithm 1 HybridRLSFitter Initialization

Class HybridCurveFitter
Input `degree`, `omega`, `lam = 1.0`, `delta = 1e5`
`self.omega` ← `omega`
`self.lam` ← `lam`
`self.num_coeffs` ← `degree` + 1 + 2
`self.theta` ← Array of zeros of size `self.num_coeffs`
`self.P` ← Identity matrix of size `self.num_coeffs` multiplied by `delta`

Algorithm 2 Update Method

Procedure Update(`self, t, y`)
`poly_terms` ← Array of `t` raised to the power of {0, 1, ..., `self.num_coeffs - 3`}
`trig_terms` ← [`cos(self.omega * t)`, `sin(self.omega * t)`]
`phi` ← Concatenate `poly_terms` and `trig_terms`
`P_phi` ← `self.P * phi`
`K` ← `P_phi / (self.lam + phi * P_phi)`
`self.theta` ← `self.theta + K * (y - theta * phi)`
`self.P` ← (`self.P - K * P_phi`^{T})/`self.lam`

Algorithm 3 Predict Method

Function Predict(`self, t`)
return `predicted_value`
`poly_terms` ← Array of `t` raised to the power of {0, 1, ..., `self.num_coeffs - 3`}
`trig_terms` ← [`cos(self.omega * t)`, `sin(self.omega * t)`]
`phi` ← Concatenate `poly_terms` and `trig_terms`
`predicted_value` ← `self.theta * phi`
return `predicted_value`

C. Interception

Intercepting the moving object to effectively grasp it as it moves on a deterministic path is a challenge since this involves taking into consideration different amounts of time required for Inverse Kinematics solution, executing this solution to align the gripper to grasp the object and finally closing the gripper to pick the object. We come up with the following algorithm to effectively intercept the object. The algorithm, for now, considers a cube on the table moving with a linear velocity in the direction of y-axis of the fixed frame.

We also determine the distribution of time required to reach a graspable pose (Fig. 2) in the table area solely in the y direction of the fixed frame. This models the range of t_{pose} vs various y positions on the table.

Algorithm 4 Pseudo Code for Intercept Estimation and Execution

Record current Gazebo time as t_{start}.
while Recursive Least Squares has not converged **do**
  Update RLS fitter.
end while
Create a range of timesteps $T = \{t_0, t_1, \ldots, t_{3000}\}$ with step size 0.01 seconds for a total duration of 30 seconds.
Get predicted positions for all timesteps in T.
Evaluate $t_{\text{gripper}} = \frac{\text{max\_gripper\_distance} - \text{cube\_width}}{\text{gripper\_velocity}}$.
Subtract t_{gripper} from the timestep range T.
Subtract the distribution of t_{pose} from T based on the position-time mapping since t_{pose} is known in y.
Randomly sample k points from the predicted trajectory at a distance d.
Deploy k threads to compute the IK solution using TRAC_IK for each sampled point, with a timeout of m seconds.
Subtract (current Gazebo time $- t_{\text{start}}$) from T.
Select the first step in T which has an IK solution and time left > 0.
Wait for the remaining time for that step.
Deploy the IK solution and then deploy the gripper.

We have now integrated the interception logic into our code. We sample points in the future trajectory of the object and simultaneously plan grasp for every point sampled. Then we discard the points that are unreachable in time. We wait for the first reachable point and then execute the grasp correctly to intercept the object. Even with the offline data for grasp time, the wait time required some fine-tuning. This is because of uncertainty in time required to execute the IK solution.

IV. EXPERIMENTS

A. Pre-grasp Position to Grasp Position

Figure 4 shows the time required to move from pre-grasp position vs the iteration for the grasp. The iteration denotes the position of cube starting along the y-axis, with iteration 0 corresponding to y coordinate 0.4 and iteration 80 to y coordinate -0.4.
This analysis will be helpful to further calculations related to position of intercept position of the cube based on the velocity and path of the cube. Based on the time to intercept, we can choose the appropriate y coordinate for intercept based on the below data.
Key Observations and Asymmetry:

a) Initial Decrease in Time (Iterations 0-30): From iteration 0 to 30, both Grasp Pose Estimate and Grasp Pose Time decrease steadily from

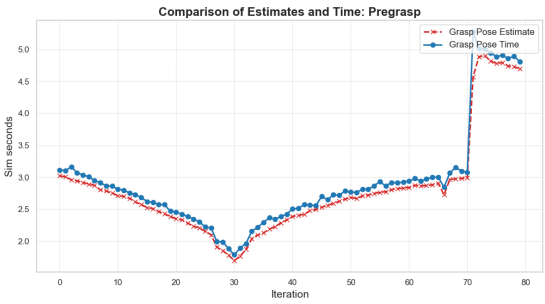

Fig. 4. Time for pre-grasp to grasp movement

around 3.0 seconds to a low of approximately 2.0 seconds. This suggests that as the cube moves from a higher y-coordinate (closer to 0.4) down towards approximately 0.1, the robotic arm becomes more efficient in reaching these positions, requiring less time. The Grasp Pose Estimate remains consistently lower than the Grasp Pose Time, indicating that estimating the grasp pose is slightly faster than executing the movement.

b) Symmetry Break (Iterations 30-70): After reaching the lowest point at iteration 30 (y-coordinate around 0.1), both variables begin to increase gradually as the cube moves further down along the y-axis.
This asymmetry suggests that as the cube moves further down along the y-axis (from around 0.1 to -0.3), it becomes progressively more challenging for the robotic arm to reach these positions, leading to increased time requirements.

c) Sharp Spike at Iteration 70: At iteration 70 (y-coordinate around -0.35), both Grasp Pose Estimate and Grasp Pose Time experience a sudden spike from approximately 3.0 seconds to around 5.0 seconds. This sharp increase indicates that when the cube reaches a lower position close to -0.4 on the y-axis, it becomes significantly more difficult for the robotic arm to move and estimate its grasp pose, possibly due to joint constraints or limitations in maneuverability at these extreme positions.

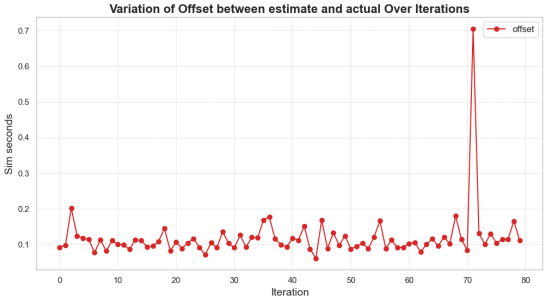

Fig. 5. Difference between estimate and actual time

Figure 5 illustrates the variation of offset between the estimated and actual time taken by a robot to execute a trajectory over 80 iterations. The offset, measured in simulation seconds, remains relatively stable throughout the majority of the iterations, fluctuating between 0.05 and 0.2 seconds. However, there is a significant spike at around iteration 70, where the offset sharply increases to approximately 0.7 seconds before returning to normal levels. This anomaly is due to the robot reconfiguring it's joints completely to grasp the object in the right side of the robot. Overall, aside from this outlier, the system maintains a consistent performance with minor variations in offset.

B. Gazebo Time vs Real World Time

Figure 6 illustrates the comparison of solution calculation times between a simulated environment (represented by the brown solid line) and a real-world environment (represented by the green dashed line) over 80 iterations - each iteration denotes the position of cube starting along the y-axis, with iteration 0 corresponding to y coordinate 0.4 and iteration 80 to y coordinate -0.4. The y-axis represents the time taken to compute joint trajectories in seconds, while the x-axis denotes the iteration number. The context involves calculating joint trajectories for moving from a pregrasp pose to an object's z-coordinate + 0.5 and then from this height to z-coordinate + 0.2 using track_ik.

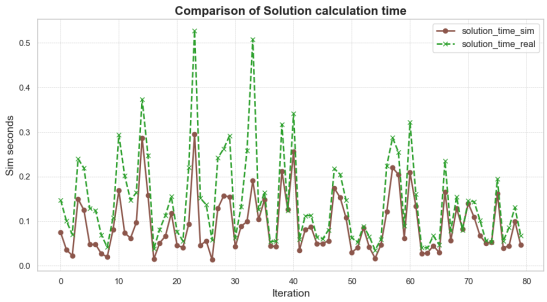

Fig. 6. Solution calculation time trend

The average solution time across both environments is 0.09385 seconds, and the range of solution times is 0.281 seconds. There are several noticeable peaks where both solution times increase significantly. For example, around iteration 20 and iteration 30, both environments experience spikes in calculation time, but the real-world environment consistently takes longer during these peaks. This suggests that certain iterations may involve more complex trajectory calculations or challenging configurations for the solver, causing longer computation times.

C. Evaluation

For phase 1, we configured the simulation environment to simulate the cube moving in a linear path

along y-axis, starting from edge of the table, with a constant velocity. The intercept point was calculated by the intercept calculator algoritm mentioned above. The intercept position was updated based on previous successful grasp. 25 iterations were evaluated for grasp success rate and grasp duration.

a) Grasp Success Rate: During the testing phase, a total of 25 iterations were conducted to evaluate the grasping mechanism. Out of these, one iteration resulted in failure. Thus, the system achieved a grasp rate of 75%.
b) Grasp Time: The grasp time remains constant overall, equal to 7.2 seconds in Gazebo Time, throughout 25 iterations of the cube grasp.

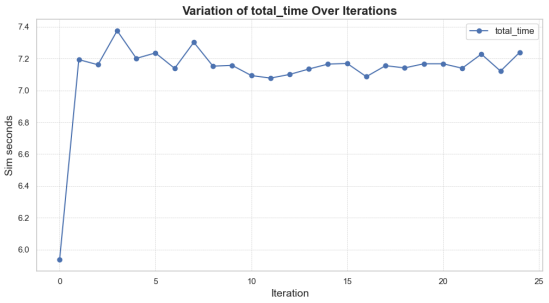

Fig. 7. Total Grasp Time

V. LIMITATIONS AND FUTURE WORK

A. Perception

Currently, we rely solely on the object pose from Gazebo. The perception system, can be developed to accurately perceive the position of the object in real-time using the robot's egocentric camera. The perception module is going to track the movement of that object using sensor data like RGBD data along with object detection/segmentation accompanied by Computational Geometry and make an estimation about the position of the object relative to the depth camera at every time step. We combine this with our current logic. Information relevant for dynamic tasks, like grasping, needs precise timing and positioning for interception. In an actual real-time perception-control feedback loop, this improves accuracy and efficiency in our grasping mechanism, particularly for non-linear paths or objects whose velocity varies.

B. Exploring non-linear Paths

We will extend the system to handle objects moving along deterministic non-linear paths, such as circular or sinusoidal trajectories. By leveraging sensor data and trajectory prediction algorithm- Recursive Least Squares (RLS), we will estimate the object's trajectory in real-time. For these non-linear paths, we will incorporate periodic functions (e.g., sine and cosine) into the prediction model to accurately track and anticipate the object's future positions.

C. Reinforcement Learning

We spent a lot of time in fine-tuning the interception logic. Reinforcement Learning is a great tool for this purpose as we can evaluate the reward based on the grasp quality and interception of the object. This reward function will fine-tune the grasp without additional human effort.

D. Mobile Manipulation

We intend to extend the application to Mobile Manipulation where the robot is steadily approaching an object on the table at a constant velocity and is planning as well as executing the grasp while moving towards the object. This can make the process more efficient.

VI. CONCLUSION

In conclusion, this study successfully demonstrates the feasibility of grasping dynamic objects moving along deterministic paths using a robotic manipulator. The integration of trajectory prediction through Recursive Least Squares and effective control strategies has enabled the Fetch robot to achieve a high grasp success rate of 80% in simulated environments. The project's methodology, which includes leveraging ROS and Gazebo for simulation, provides a robust framework for testing and refining robotic manipulation techniques. The experiments conducted revealed key insights into the efficiency of the robot's movements, particularly highlighting areas where performance can be optimized, such as reducing time spikes during specific iterations. This research holds significant implications for industrial applications, particularly in automated systems for tasks like conveyor belt sorting, where precision and efficiency are paramount. Future work will focus on enhancing the system's adaptability by exploring non-linear object paths and velocities, as well as improving perception techniques to handle more complex and varied environments. These advancements will further enhance the capabilities of robotic systems in dynamic settings, contributing to more sophisticated and versatile automation solutions.

REFERENCES

[1] Y. Xiang, S. H. Allu, R. Peddi, T. Summers, and V. Gogate, "Grasping trajectory optimization with point clouds," *arXiv preprint arXiv:2403.05466*, 2024.
[2] I. Akinola, J. Xu, S. Song, and P. K. Allen, "Dynamic grasping with reachability and motion awareness," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 9422–9429.
[3] W. Yang, C. Paxton, A. Mousavian, Y. Chao, M. Cakmak, and D. Fox, "Reactive human-to-robot handovers of arbitrary objects," *CoRR*, vol. abs/2011.08961, 2020. [Online]. Available: https://arxiv.org/abs/2011.08961
[4] T. Wu, F. Zhong, Y. Geng, H. Wang, Y. Zhu, Y. Wang, and H. Dong, "Grasparl: Dynamic grasping via adversarial reinforcement learning," *arXiv preprint arXiv:2203.02119*, 2022.