UR3 Pick-and-Place Task using Seven-Segment Motion Planning and Simulating in pyBullet

Author: Amin Akhavan Saffar
Date: September 2024

Abstract

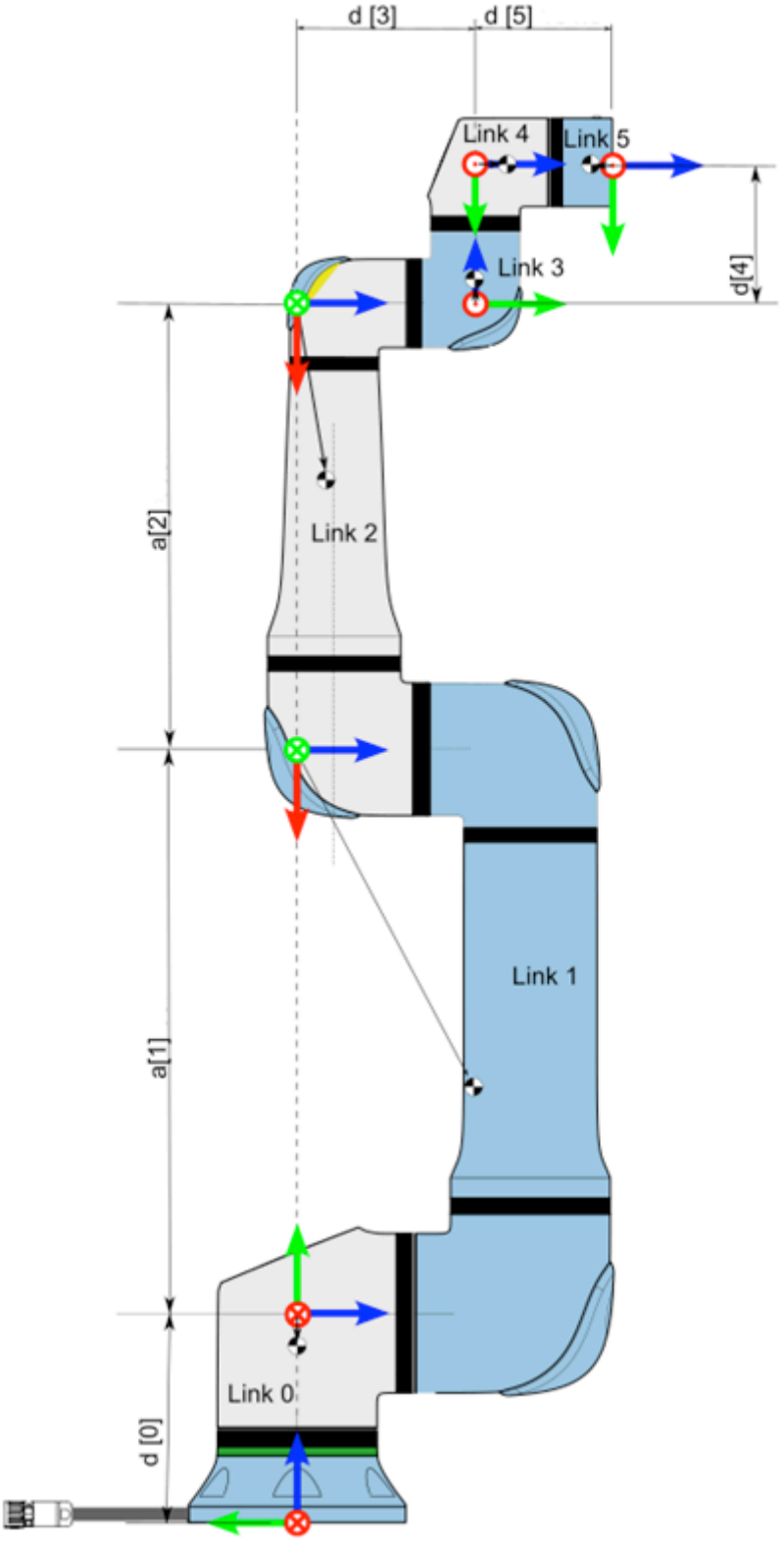
In this project, we developed a trajectory planning and motion control framework for the UR3 robot to perform a pick-and-place task. The trajectory was generated using seven-segment motion profiles in joint space. The results were validated through simulation in PyBullet.

1. Introduction

This document presents the implementation of a pick-and-place task using the UR3 robot. The goal of this project was to plan and execute a trajectory to move an object from an initial location to a target destination. To achieve this, we generated smooth motion profiles for each joint using a seven-segment motion planning method. The trajectory was executed in a simulation environment using PyBullet.

The key tools used in this project were:

- **Robotics Toolbox** for Python: For kinematics and trajectory generation.
- **PyBullet**: For simulation of the robot's motion.
- **Python**: Programming language used for implementation and control.
- **UR3 Robot Model**: Used in both trajectory generation and simulation.



2. Forward and Inverse Kinematics

2.1 Forward Kinematics

Forward kinematics is the process of calculating the position and orientation of the robot's end-effector, given the values of its joint angles. For a serial manipulator like the UR3, the

forward kinematics problem can be solved by multiplying the transformation matrices associated with each joint.

DH Parameters

The most common method to derive the transformation matrices is using **Denavit-Hartenberg (DH) parameters**. Each joint has four DH parameters:

1. (a_i): Link length, the distance between (z_i) and (z_{i+1}) along (x_i).
2. (α_i): Link twist, the angle between (z_i) and (z_{i+1}) around (x_i).
3. (d_i): Link offset, the distance between (x_i) and (x_{i+1}) along (z_i).
4. (θ_i): Joint angle, the rotation around (z_i) that takes (x_i) to (x_{i+1}).

Here's the Denavit-Hartenberg (DH) parameter table for the UR3 robot

Joint	θ **[rad]**	a **[m]**	d **[m]**	$alpha$ **[rad]**
Joint 1	0	0	0.1519	$(\frac{\pi}{2})$
Joint 2	0	-0.24365	0	0
Joint 3	0	-0.21325	0	0
Joint 4	0	0	0.11235	$(\frac{\pi}{2})$
Joint 5	0	0	0.08535	-$(\frac{\pi}{2})$
Joint 6	0	0	0.0819	0

Notes:

- (θ) is the joint angle, which can vary based on the robot's configuration.
- (a) is the link length, representing the distance between the previous joint and the current joint along the x-axis.
- (d) is the link offset, indicating the distance along the z-axis to the common normal.
- (α) is the twist angle, which describes the angle between the z-axes of two consecutive joints.

This table provides a comprehensive overview of the kinematic and dynamic parameters for the UR3 robot, which are essential for modeling its motion and behavior in robotic applications.
Each transformation matrix (A_i) for a given joint is given by:

$$A_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The forward kinematics transformation matrix (T_0^n) from the base frame to the end-effector frame is the product of all individual joint transformation matrices:

$$T_0^n = A_1 \cdot A_2 \cdot \cdots \cdot A_n$$

For the UR3, the forward kinematics involves six transformations since it is a six-degree-of-freedom (6-DOF) robot. The transformation matrix (T_0^6) gives the position and orientation of the end-effector relative to the base.

2.2 Inverse Kinematics

Inverse kinematics (IK) involves calculating the joint angles $(q = [q_1, q_2, \ldots, q_6])$ that will achieve a desired end-effector pose (T_d), defined by a position vector (p) and an orientation matrix (R) in space. For a 6-DOF manipulator like the UR3, solving the inverse kinematics is more complex than forward kinematics due to the nonlinear relationship between joint angles and the end-effector position.

$$q = f^{-1}(T_d) = f^{-1}(p, R)$$

Analytical Methods

For certain robot configurations, like the UR3, closed-form analytical solutions can be derived for the inverse kinematics. The process involves decomposing the desired end-effector pose into translation (position) and rotation (orientation), then solving for the joint angles one by one.

Step-by-Step Analytical IK for UR3

1. **Position of the Wrist Center**:
   The first step in solving IK is determining the position of the wrist center. The wrist center (p_w) is the point where the last three joints intersect and can be found by subtracting the end-effector's orientation from its position:

   $$p_w = p - d_6 \cdot R \cdot [0, 0, 1]^T$$

   Where:
   - (p) is the end-effector's position.
   - (R) is the orientation matrix.
   - (d_6) is the distance from the wrist center to the end-effector along the z-axis.
2. **Solving for (q_1) (Base Joint Angle)**:
   The first joint angle (q_1) can be computed from the projection of the wrist center onto the (x_0)-(y_0) plane:

$$q_1 = \arctan\left(\frac{p_{wy}}{p_{wx}}\right)$$

Where (p_{wx}) and (p_{wy}) are the x and y coordinates of the wrist center.

3. **Solving for (q_2) and (q_3) (Shoulder and Elbow Joint Angles)**:
   Using the known position of the wrist center, we can compute the distances and angles required for the second and third joint angles using the law of cosines and trigonometry:

$$q_2 = \arctan(p_{wz}, r) - \arctan(d_2, d_3)$$

$$q_3 = \arccos\left(\frac{r^2 + z^2 - d_2^2 - d_3^2}{2d_2d_3}\right)$$

4. **Solving for (q_4, q_5, q_6) (Wrist Joint Angles)**:
   Once the first three joint angles are determined, the wrist orientation can be solved by aligning the end-effector's orientation matrix with the desired (R_d). The wrist angles (q_4, q_5, q_6) are computed by extracting the rotational components from the matrix multiplication of the inverse transformation:

$$R_{36} = R_0^3 dot R_d$$

   The wrist angles are then found through standard methods of rotation matrix decomposition.

Numerical Methods

For more complex robots or cases where a closed-form solution doesn't exist, **numerical methods** are used to solve inverse kinematics. One common approach is the **Levenberg-Marquardt (LM) algorithm**, which is a hybrid method combining gradient descent and least-squares minimization. In this method, an initial guess for the joint angles is iteratively improved until the error between the desired and current end-effector pose is minimized.

The error is defined as:

$$e = T_d - T(q)$$

Where (T_d) is the desired transformation and $(T(q))$ is the transformation obtained by applying the current joint angles (q).

This method requires defining a Jacobian matrix (J), which relates changes in joint space (δq) to changes in the end-effector pose (δT):

$$\delta T = J \cdot \delta q$$

Numerical solvers like `IK_LM()` from the `roboticstoolbox` library implement this approach.

2.3 Singularities

One of the significant challenges in inverse kinematics (IK) is dealing with **singularities**. These singularities can lead to a loss of certain degrees of freedom, making the IK problem ill-posed and difficult to solve. Understanding how to monitor the robot's configuration and manage multiple IK solutions is crucial in designing robust robotic systems.

Singularities in Robotics

Singularities occur when the Jacobian matrix $(\mathbf{J})$ becomes rank-deficient, meaning that it does not have full rank. This often happens in configurations where:

- The robot’s joints align in a way that reduces its effective degrees of freedom.
- The robot's end-effector is positioned in a way that any small movement in Cartesian space leads to an undefined or very large change in the joint angles.

At singularity points, the robot may be unable to control its end-effector effectively. For example, consider a robotic arm where two joints are aligned along a straight line; in such cases, the end-effector can move freely in one direction without any corresponding change in joint angles.

Impact of Singularities on IK Solutions

The presence of singularities complicates the IK process in several ways:

1. **Ill-Posed Solutions**: When approaching a singularity, small changes in the desired position or orientation of the end-effector can lead to large and unpredictable changes in the joint angles. This can make the IK solution unstable.
2. **Multiple Solutions**: Many IK problems, especially in redundant robots, can yield multiple valid joint configurations for the same end-effector position. This can create complications when the robot is close to a singularity, as multiple solutions might lead to one being preferred while the other is not achievable.

Managing Robot Configuration

To effectively manage the robot configuration and avoid singularities, several strategies can be employed:

1. **Configuration Monitoring**:
    - Continuously monitor the robot’s configuration in real-time to detect when it is approaching a singularity. Techniques such as analyzing the condition number of the Jacobian matrix can be used. A condition number close to zero indicates that the robot is near a singularity.

- Implement feedback mechanisms to adjust the robot's configuration before reaching a singularity.

2. **Path Re-Planning**:
   - If the desired path leads the robot toward a singularity, it can dynamically adjust its trajectory. Path re-planning can redirect the robot to a different configuration that avoids singularities, ensuring smooth operation.
3. **Singularity Robustness**:
   - Implement control strategies that include redundancy resolution to select among multiple IK solutions, aiming to choose configurations that are least likely to lead to singularities.
   - Consider optimization criteria that minimize the condition number of the Jacobian during motion planning, maintaining stability as the robot moves.

Multiple Inverse Kinematic Solutions

In many robotic systems, particularly those with multiple degrees of freedom, a single desired end-effector position can correspond to several different joint configurations. This is common in robotic arms where there might be several ways to position the end-effector at the same point.

1. **Redundancy Resolution**:
   - For redundant robots, strategies can be applied to choose a configuration that minimizes energy consumption or joint limits. Redundancy can be used to avoid obstacles, maintain a preferred posture, or optimize the robot's operational workspace.
2. **Joint Limits and Constraints**:
   - Implement constraints based on the robot's joint limits to filter out solutions that are infeasible. This helps maintain physical realism and prevents configurations that could lead to mechanical failure.
3. **User-defined Preferences**:
   - In some applications, user-defined criteria can be established to guide the IK solver in selecting preferred configurations, such as maintaining the robot's orientation or avoiding configurations that lead to singularities.

In robotic systems like the UR3, functions like `ur3.config_validate("ldn", ('lr', 'ud', 'nf'))` can be instrumental in assessing the configuration of the robot. This function typically evaluates the robot's current configuration against a set of criteria.

- `"ldn"`: This could represent a specific task or configuration scenario for the robot.
- `('lr', 'ud', 'nf')`: This tuple represents constraints or orientations:
  - **lr**: Left/Right configuration.
  - **ud**: Up/Down orientation.

- **nf**: Forward/Backward positioning.

The configuration validation process helps ensure that the robot is in a suitable state for executing tasks without encountering singularities or invalid configurations. By analyzing the orientation and joint limits, it guides the robot in making decisions that maintain operational stability.

Certainly! Here's an expanded section on trajectory generation, focusing on the key concepts and mathematical formulations related to seven-segment motion planning.

---

3. Trajectory Generation

Key Trajectory Points

The trajectory for the UR3 robot during the pick-and-place task consists of several critical waypoints:

- **Starting Position**: The initial configuration of the robot before executing the task.
- **Pick Position**: The precise location from which the robot will grasp the object.
- **Place Position**: The target destination where the object will be placed.
- **Home Position**: The configuration to which the robot returns after completing the pick-and-place task.

The transformation matrices for these positions were computed in the special Euclidean group $(SE(3))$, which represents the position and orientation of the end-effector in 3D space.

Seven-Segment Motion Planning

To ensure smooth motion transitions between the key trajectory points, we utilized a seven-segment motion planning algorithm. This method is beneficial in robotic applications, as it provides a structured approach to controlling the velocity and acceleration of each joint during movement, thus avoiding abrupt changes that could lead to mechanical stress or inaccuracies.

The seven segments are defined as follows:

1. **Acceleration Phase**:
    - The robot starts from a standstill and accelerates to its maximum speed.
    - The position and velocity profiles can be described by a cubic polynomial function defined over the time interval $([0, t_{\text{accel}}])$:

$$\theta(t) = \theta_0 + v_0 t + \frac{1}{2} a t^2$$

where $(v_0 = 0)$ (initial velocity) and (a) is the acceleration.

2. **Cruise Phase**:
   - The robot maintains its maximum speed.
   - For the duration of this phase, the joint positions can be described linearly:

$$\theta(t) = \theta_{\max} + v_{\max}(t - t_{\text{accel}}) \quad \text{for } t_{\text{accel}} < t < t_{\text{cruise}}$$

3. **Deceleration Phase**:
   - The robot begins to decelerate until it comes to a stop.
   - The polynomial function for this phase can again be represented as:

$$\theta(t) = \theta_{\max} + v_{\max}(t - t_{\text{cruise}}) - \frac{1}{2} a (t - t_{\text{cruise}})^2$$

4. **Additional Segments**:
   - The trajectory may include return to home and possible intermediate waypoints, each governed by similar polynomial functions based on their respective acceleration, cruising, and deceleration phases.

Total Time for the Trajectory

The total time for the trajectory execution is calculated by summing the time spent in each phase:

$$t_{\text{total}} = 2 \cdot t_{\text{accel}} + t_{\text{cruise}}$$

Where:

- (t_{accel}) is the time taken to accelerate to maximum speed,
- (t_{cruise}) is the time spent traveling at maximum speed between the pick and place positions.

Time Parameterization

Given the maximum speed $(v_{\max})$ and maximum acceleration $(a_{\max})$ for each joint, the time spent in each motion phase can be calculated as follows:

1. **Acceleration Time**:
   To compute the time required to reach maximum velocity:

$$t_{\text{accel}} = \frac{v_{\max}}{a_{\max}}$$

2. **Cruise Time**:
   The time spent moving at maximum speed can be derived from the distance covered during this phase. If the total distance is (d):

$$d = v_{\max} \cdot t_{\text{cruise}} \Rightarrow t_{\text{cruise}} = \frac{d}{v_{\max}}$$

3. **Deceleration Time**:
   The deceleration time is identical to the acceleration time since we assume symmetrical acceleration and deceleration:

$$t_{\text{decel}} = t_{\text{accel}}$$

The overall trajectory time should ensure that all joints complete their movements within the same total time. The maximum duration across all joints is selected to avoid synchronization issues during motion.

Generating Trajectory Profiles for Each Joint

To facilitate joint coordination, trajectory profiles for each joint are generated based on their individual parameters. For the joint (j):

- The joint angle profile $(\theta_j(t))$ can be defined using the above polynomial equations for each segment, based on their respective motion characteristics.

For example, if (j) is the first joint and its maximum speed and acceleration are given, the joint profile can be expressed as:

$$\theta_1(t) = \begin{cases} \theta_0 + \frac{1}{2}a_1 t^2, & 0 \leq t < t_{\text{accel}} \\ \theta_{max1} + v_{max1}(t - t_{accel}), & t_{\text{accel}} \leq t < t_{cruise} \\ \theta_{max1} + v_{max1}(t - t_{cruise}) - \frac{1}{2}a_1(t - t_{cruise})^2, & t_{cruise} \leq t < t_{total} \end{cases}$$

This process is repeated for each joint, resulting in a complete trajectory profile for the entire motion sequence.

---

4. Pick-and-Place Motion Senario

Here's an expanded explanation of the pick-and-place motion, incorporating the definition of key points as SE(3) matrices, and detailing each phase of the operation:

Pick-and-Place Motion Description

1. Key Points Definition

The key points in the trajectory are defined as SE(3) transformation matrices, which capture both the position and orientation of the robot's end-effector throughout the pick-and-place task. Below is a breakdown of each key point:

- **Starting Position (T_start)**:

  $$T_{\text{start}} = \text{ur3.fkine}(q_{\text{home}})$$

  This matrix represents the robot's initial configuration, where (q_{home}) denotes the joint angles corresponding to the home position of the UR3 robot. It serves as the reference point for the entire operation.
- **Approach Position (T_approach)**:

  $$T_{\text{approach}} = SE3.\text{Trans}(0.4, 0.0, 0.25) * SE3.\text{RPY}([0, 0, 0])$$

  This matrix positions the end-effector directly above the object to be picked, ensuring a safe distance to avoid collisions.
- **Pick Position (T_pick)**:

  $$T_{\text{pick}} = SE3.\text{Trans}(0.4, 0.0, 0.18) * SE3.\text{RPY}([0, 0, 0])$$

  This transformation moves the end-effector down to the exact location where the object is located, allowing the robot to grasp it securely.
- **Lift Position (T_lift)**:

  $$T_{\text{lift}} = T_{\text{approach}}$$

  After grasping the object, the robot moves back up to the approach height, ensuring the object is safely lifted above any obstacles.
- **Place Approach Position (T_place_approach)**:

  $$T_{\text{place\_approach}} = SE3.\text{Trans}(0.3, 0.1, 0.25) * SE3.\text{RPY}([0, 0, \frac{\pi}{2}])$$

  This position is located directly above the designated placement area. The orientation ensures that the end-effector is correctly aligned for the placement.
- **Place Position (T_place)**:

  $$T_{\text{place}} = SE3.\text{Trans}(0.3, 0.1, 0.18) * SE3.\text{RPY}([0, 0, \frac{\pi}{2}])$$

  This matrix corresponds to the final position where the object will be placed, allowing for a controlled release.

2. Motion Phases

The pick-and-place operation is broken down into distinct phases, each with specific motions and objectives:

a. Movement to Approach Position

- The robot transitions from the **starting position** to the **approach position**. This motion is executed with controlled acceleration to avoid sudden jolts that could destabilize the robot.

b. Descending to Pick Position

- After reaching the approach position, the robot lowers its end-effector to the **pick position**. This descent must be precise to ensure that the gripper makes contact with the object securely.

c. Object Grasping

- Once at the pick position, the gripper closes around the object, executing a grasping function to ensure a secure hold. This phase may involve force sensing to confirm that the object has been grasped correctly.

d. Lifting the Object

- The robot then lifts the object back to the **approach height**. This lifting motion is typically smooth and steady, focusing on maintaining balance and stability.

e. Moving to Place Approach Position

- After lifting, the robot moves horizontally to the **place approach position**. This motion can occur at a constant speed to minimize disturbances during transport.

f. Descending to Place Position

- As the robot approaches the placement area, it decelerates smoothly to reach the **place position**. Proper control during this phase is critical to ensure accurate placement.

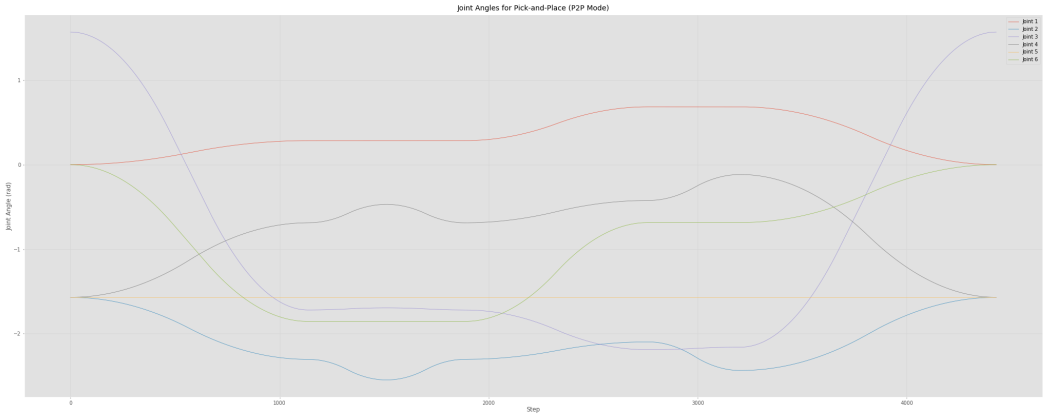
g. Object Placement

- At the place position, the robot opens its gripper to release the object. The release should be gentle to avoid bouncing or sliding, ensuring that the object is placed precisely where intended.

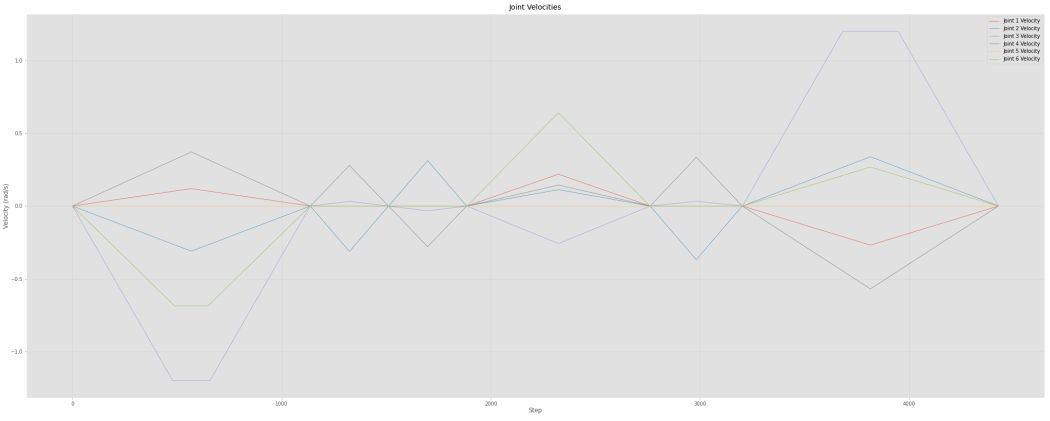
h. Returning to Home Position

- After successfully placing the object, the robot moves back to the **home position**. This final movement can be executed with similar control to the initial movements, ensuring that the robot is ready for the next task.

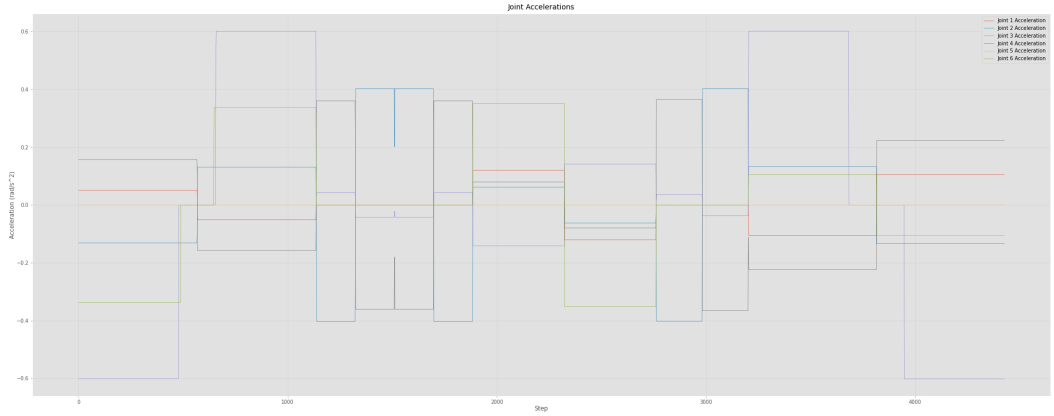
joint angles during pick and place



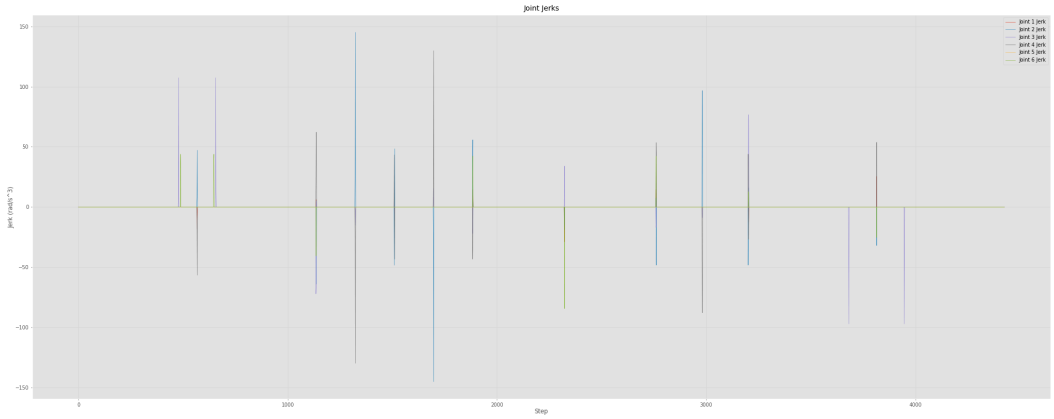
joint velocities during pick and place



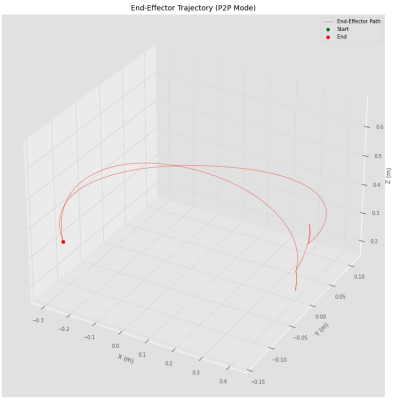
joint acceleration during pick and place



joint jerk during pick and place



End Effector Trajectory during pick and place designed Trajectory



---

5. Simulation with PyBullet

Simulation Setup

To validate the generated trajectory for the pick-and-place task, the PyBullet physics engine was utilized to create a realistic simulation environment. The setup process included the following steps:

1. **Robot Model Configuration**:
    - The UR3 robot model was loaded into the PyBullet environment using URDF (Unified Robot Description Format) files, which defined the robot's physical characteristics, joint configurations, and kinematics. Proper scaling and positioning were ensured to match the real-world dimensions of the UR3.
2. **Workspace Creation**:
    - A simulated workspace was constructed, which included a flat table to represent the surface where the object would be placed. The table dimensions were set to ensure adequate space for the robot's movements.
    - The object intended for manipulation (e.g., a cube or cylinder) was created as a simple geometric shape within the simulation. The initial position of the object was aligned with the pick position defined in the trajectory.
3. **Joint Initialization**:
    - The robot's joints were initialized to their home configuration (denoted by (q_{home})), ensuring that the robot starts from a known state. The joint limits were also set to reflect the physical constraints of the UR3, preventing any unrealistic movements during the simulation.
4. **Physics Parameters**:
    - Various physics parameters, such as gravity, friction, and collision detection, were configured to enhance the realism of the simulation. This included setting appropriate mass and inertia for the object to be manipulated, ensuring accurate interaction with the robot.

Trajectory Execution

With the simulation environment configured, the execution of the planned trajectory was performed as follows:

1. **Trajectory Generation**:
    - The trajectory was generated based on the key points defined earlier (approach, pick, lift, place, and so on). This trajectory included joint position, velocity, and acceleration profiles for each phase of the motion.
2. **Time Step Updates**:
    - As the simulation progressed, joint positions were updated at each discrete time step according to the generated trajectory profiles. The simulation was run for a specified duration, synchronized with the planned motion timeline.
3. **Data Recording**:

- Throughout the motion, the positions, velocities, and accelerations of the robot's joints were recorded. This data collection allowed for a detailed analysis of the robot's performance and motion dynamics. Key parameters recorded included:
  - **Joint Angles**: The current angle of each joint at every time step.
  - **Joint Velocities**: The rate of change of the joint angles, indicating how quickly each joint is moving.
  - **Joint Accelerations**: The rate of change of joint velocities, useful for understanding the forces and torques acting on the joints.

4. **Collision Monitoring**:
   - Collision detection mechanisms were actively employed during the simulation. This ensured that the robot's end-effector did not penetrate the table or the object. If a collision was detected, the simulation would log the event, allowing for adjustments to be made in the trajectory if necessary.
5. **Visual Feedback**:
   - PyBullet provided a visual representation of the robot's movements, allowing for real-time observation of the pick-and-place task. The simulation window displayed the robot, object, and workspace, providing immediate feedback on the trajectory execution and performance.
6. **Post-Execution Analysis**:
   - After the trajectory execution was completed, a comprehensive analysis was performed. This involved comparing the recorded joint positions, velocities, and accelerations against the planned trajectory profiles.
   - Various metrics, such as tracking error (the difference between planned and actual joint positions) and smoothness of motion (evaluated through velocity and acceleration profiles), were calculated to assess the effectiveness of the trajectory.
7. **Adjustment and Optimization**:
   - Based on the results of the post-execution analysis, adjustments to the trajectory could be made to improve performance. This might include refining acceleration profiles, adjusting key points for better alignment, or modifying the approach and place positions to enhance stability.

Conclusion

The simulation in PyBullet provided a robust platform for validating the trajectory generated for the UR3 robot's pick-and-place task. The detailed analysis of the robot's performance during the simulation not only ensured that the planned trajectory was feasible but also identified areas for potential improvement in future implementations. The insights gained from the simulation will inform the real-world application of the trajectory, leading to more reliable and efficient robotic manipulation.

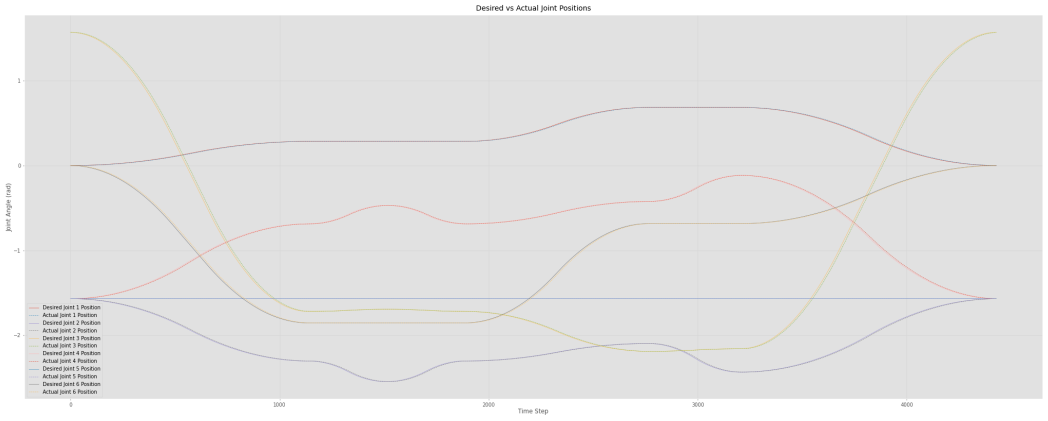
---

6. Results

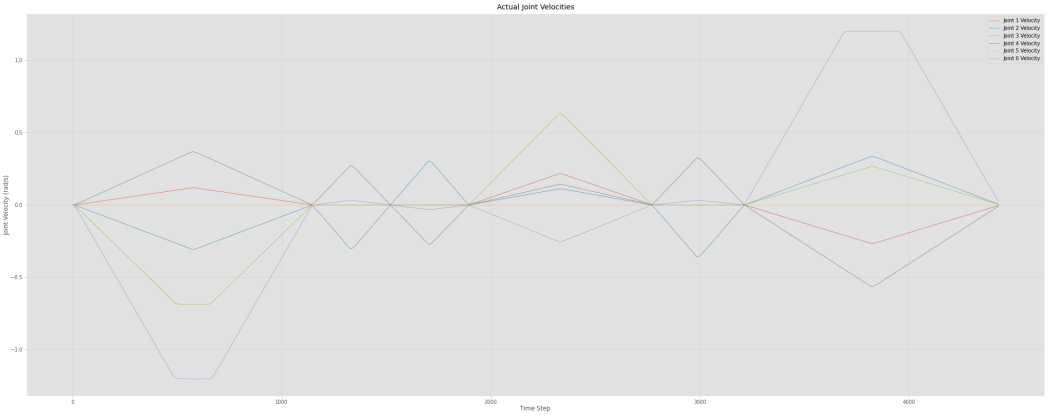
Desired vs Actual Joint Positions

The actual joint positions obtained from the simulation were compared to the desired joint positions from the trajectory plan. Small discrepancies were observed due to simulation dynamics and control imperfections.

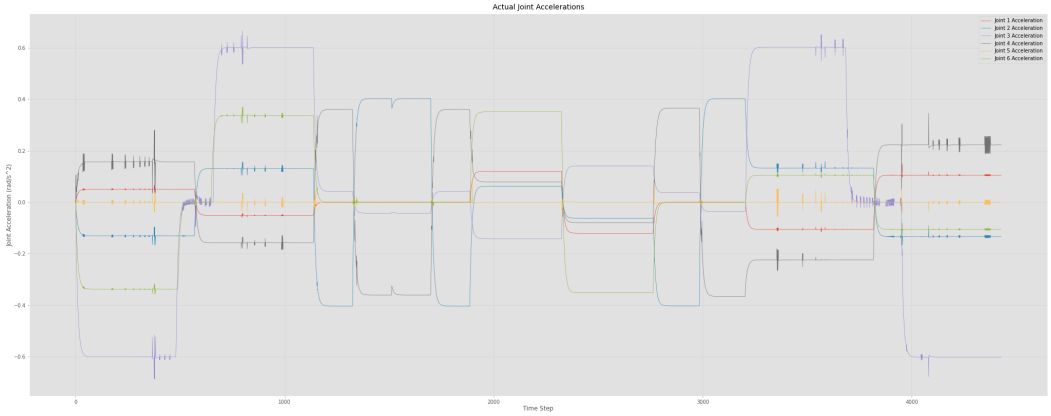
Actual and Desired Joint angles during pick and place, The actual values came from simulation



Actual joint velocities



Actual joint \acceleration



---

7. Conclusion

In this project, we successfully implemented and simulated a pick-and-place task using the UR3 robot. The use of seven-segment motion planning allowed for smooth and controlled motion, which was validated through simulation in PyBullet. Future improvements could involve real-world testing and further optimization of the control system to reduce discrepancies between planned and actual movements.
