Coordinated Dual-Arm Manipulation for Object Transport Using the Nextage Robot

B211116 & B201893

November 2024

Contents

1 Abstract

This report presents an implementation of coordinated dual-arm manipulation for transporting a cube from an initial to a goal configuration. We employ motion planning techniques, using appropropriate dynamic control methods throughout the process. Specifically, we implement Inverse Kinematics (IK) and the Rapidly-exploring Random Tree (RRT) algorithm, followed by an inverse dynamics approach with joint space control.

2 Introduction

This section outlines the research problem, its significance, its challenges, our approach, and its limitations.

2.1 Problem Statement

The task is to create a strategy for the Nextage robot to use both effectors to grasp a box and transport it to a target location. This involves two stages:

1. **Motion Planning**: Generate a path that moves the effectors to a valid grasping configuration and transports the box while avoiding collisions and respecting joint limits.
2. **Dynamics**: Implement a control method in a dynamics simulator to follow the reference path, updating the robot's state and ensuring precise execution at each step.

2.2 Significance

This task addresses key challenges in dual-arm manipulation, a crucial area of robotics with applications in manufacturing and healthcare. Previous research, as noted in [1], has struggled with issues like sampling, path optimisation, and dynamic adaptability in cooperative motion planning. This highlights the need for robust solutions that account for dynamic interactions with the environment. Solving this problem will advance research in robotic motion planning and dynamic control.

2.3 Complexity

This task is especially challenging due to the complex constraints of dual-arm motion planning. Motion planning itself is known to be PSPACE-hard, with computational complexity increasing exponentially as the number of dimensions grows [2]. Naive approaches fail to address this, and when factoring in robot dynamics, the computational requirements become even more demanding [4].

2.4 Approach and Limitations

2.4.1 Approach

Our approach utilises inverse kinematics(IK) and rapidly exploring random trees(RRT) for dual arm motion planning where the oscillation of trajectory is controlled by proportaional-derivative(PD) controller.

1. **IK**: Uses inverse kinematics with Jacobian-based error minimization to adjust joint configurations and reduce positional error, ensuring each step respects joint limits.
2. **RRT**: Randomly samples cube configurations, computes grasp poses, and checks collisions. The algorithm finds the nearest neighbor, extends paths towards the goal, and backtracks to reconstruct a collision-free path once the goal is reached.
3. **Dynamics**:
    - **Bezier Curve**: Smooths trajectory based on computed path by performing linear interpolation between two points in the path.
    - **PD Controller**: Uses the proportional-derivative control method used to find torque for robot manipulation by finding the acceleration given the current position and velocity at the current time step.

2.4.2 Limitations

We encountered limitations with sample efficiency as without proper constraints on the sampling range, RRT needed excessive iterations to find an efficient path. While the planning algorithm produced collision-free paths, trajectory optimisation and torque control were not perfect, leading to occasional undesirable results. The randomness of sampled configurations made it difficult to consistently determine the right forces and gains, affecting the robot's performance in each run. This is discussed further in 3.2.

3 Methods

This section outlines the components used to control our robot, describing their inputs, outputs, objectives, and the rationale behind our choice of algorithms.

3.1 Motion Planning

This section covers the first step of our research: using IK to calculate the grasp pose and planning a path from the initial to goal configuration with RRT.

3.1.1 Inverse Geometry

Objective: This component uses the ‘computegrasppose‘ function in ‘inverse_geometry.py‘ to compute the robot’s grasping configuration and success flag by using joint velocities to adjust the end-effector positions with the pseudoinverse of the Jacobian.

$$v\mathbf{q} = J^{+}\mathbf{e} \tag{1}$$

Where J^{+} refers to the pseudo-inverse Jacobian matrix and $\mathbf{e}$ represents the position and orientation error between the end-effectors and cube hooks(where the robot can pick up the cube).

The steps within our inverse kinematics algorithm include:

- **Error computation:** At each iteration, the function calculates the difference between the robot’s current hand positions (oMleft_hand, oMright_hand) and the target positions on the cube (oMleft_hook, oMright_hook)
- **Jacobian calculation:** The Jacobians for the left and right hands are computed and stacked to form a combined Jacobian.
- **Joint update:** Joint velocities ($v\mathbf{q}$) are derived by multiplying the pseudo-inverse Jacobian with the error vector. This velocity updates the robot’s configuration using integration, moving the robot closer to the target.
- **Convergence check:** he process repeats until the distance between the end-effectors and hooks fall below a small number close to 0 or the iteration limit is reached.

Inputs:

1. **robot:** The robot model object containing information about the robot’s structure and kinematics.
2. **q_current:** The current configuration (joint positions) of the robot.
3. **cube:** The configuration of the cube position.
4. **cubetarget:** The configuration of the target cube position.
5. **viz:** A visualisation parameter to show intermediate states.

Outputs:

1. **q_current:** The configuration of the robot after performing inverse kinematics to get the grasp pose of the end effectors for the cube.
2. **True:** The success flag that always returns true as joint constraints and collision detection are handled within motion planning.

Justification: Inverse kinematics (IK) is used instead of inverse geometry because it provides iterative solutions, accommodating the robot's multiple degrees of freedom. This ensures adaptable hand alignment with the cube's hooks, allowing collision-free paths that respect joint limits without hardcoding placements.

3.1.2 Rapidly Exploring Random Trees

In this component, the 'path.py' file implements an RRT-based path-planning algorithm to compute a valid path for the robot to grasp and move the cube.

Different Methods Used: Initially, we planned to compute the path based on robot configurations alone, but this resulted in the cube slipping at certain angles due to unstable grasps as illustrated in figure 1. To address this, we shifted to computing grasps based on cube configurations, ensuring more stable handling throughout the motion.

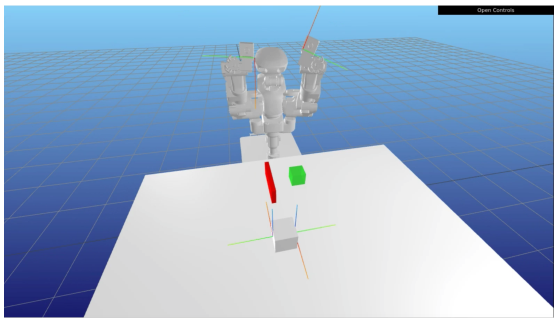


Figure 1: Figure of the cube slipping out of the robot's end-effectors when using robot configurations to do path planning.

Objective: The goal of path.py is to apply the RRT algorithm to find a collision-free path from the initial to the target configurations for both the robot and the cube. The main computepath function initialises the graph G to hold a tuple containing the parent node of the current robot configuration, the cube configuration and the robot configuration, along with a set of sampled_positions to avoid redundant sampling

These are the key steps within our RRT algorithm:

- **Sampling configurations:** Samples random cube configurations, starting by offsetting the cube slightly above its initial position. The helper function sample_cube_higher performs this offset, while a sample_higher flag can enable a wider sampling range if desired.
- **Calculating robot pose:** Inverse kinematics computes the robot's grasp pose, checking for collisions and joint constraints.

- Finding nearest neighbour: Identifies the nearest node in the graph to the sampled cube configuration using Euclidean distance with NEAREST_VERTEX_CUBE_Q. The closest robot pose is found with NEAREST_VERTEX_ROBOT_Q.
- **Generating new configurations:** NEW_CONF_CUBE calculates an incremental path from the nearest cube position to the sampled position. It performs collision checks and, if necessary, adjusts configurations to avoid obstacles. The most important part of this function is that it calculates the grasp pose for the robot for the cube configurations to check if the robot is in collision. If it is, the cube configuration is adjusted more along the path.
- **Growing the graph:** ADD_EDGE_AND_VERTEX expands G with the new configuration, linking it to the prior node and adding the corresponding cube configuration.
- **Validating the edge:** VALID_EDGE checks if the path between two cube configurations is collision-free by interpolating and verifying collision-free robot movement to the target configuration.
- **Path completion or continuation:** If a valid edge reaches the target configuration, computepath confirms a path is found and retrieves the sequence using getpath. If not, the algorithm iterates until either a path is found or the maximum iteration limit (k) is reached.

Functions and Their Inputs and Outputs: The diagram in figure 2 decribes the inputs and outputs of each function used to create the RRT algorithm.

Justification: RRT is well-suited for this problem due to its strong performance in high-dimensional configuration spaces, like ours, and its probabilistic completeness. This means the likelihood of finding a path approaches one as iterations increase [3]. Despite some limitations (see section 2.4.2), RRT was the optimal choice for our needs.

To enhance RRT's accuracy, we modified the NEW_CONF_CUBE function to prioritise configurations where both the robot and the cube were collision-free. This is because some situations arose where the cube was clear but the robot was in collision. Additionally, to address inefficient or failed pathfinding due to an unconstrained sampling space, we implemented a targeted cube-sampling method (sample_cube_higher). This method begins sampling slightly above the initial cube position, helping the path to clear obstacles and reducing sampling time. For a comparison of this optimised sampling approach to the unconstrained method, see section 4.1.2.

3.2 Dynamics

The goal of this task was to enable the robot to execute smooth, realistic motions. By managing dynamics, the robot could follow the planned path with

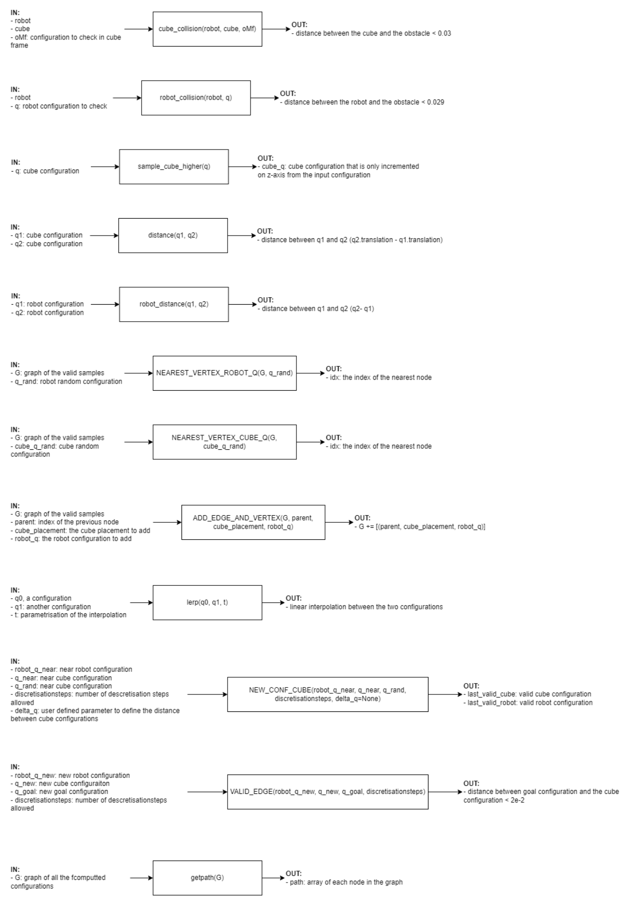

IN:
- robot
- cube
- oMf: configuration to check in cube frame
cube_collision(robot, cube, oMf)
OUT:
- distance between the cube and the obstacle < 0.03
IN:
- robot
- q: robot configuration to check
robot_collision(robot, q)
OUT:
- distance between the robot and the obstacle < 0.029
IN:
- q: cube configuration
sample_cube_higher(q)
OUT:
- cube_q: cube configuration that is only incremented on z-axis from the input configuration
IN:
- q1: cube configuration
- q2: cube configuration
distance(q1, q2)
OUT:
- distance between q1 and q2 (q2.translation - q1.translation)
IN:
- q1: robot configuration
- q2: robot configuration
robot_distance(q1, q2)
OUT:
- distance between q1 and q2 (q2- q1)
IN:
- G: graph of the valid samples
- q_rand: robot random configuration
NEAREST_VERTEX_ROBOT_Q(G, q_rand)
OUT:
- idx: the index of the nearest node
IN:
- G: graph of the valid samples
- cube_q_rand: cube random configuration
NEAREST_VERTEX_CUBE_Q(G, cube_q_rand)
OUT:
- idx: the index of the nearest node
IN:
- G: graph of the valid samples
- parent: index of the previous node
- cube_placement: the cube placement to add
- robot_q: the robot configuration to add
ADD_EDGE_AND_VERTEX(G, parent, cube_placement, robot_q)
OUT:
- G += [(parent, cube_placement, robot_q)]
IN:
- q0, a configuration
- q1: another configuration
- t: parametrisation of the interpolation
lerp(q0, q1, t)
OUT:
- linear interpolation between the two configurations
IN:
- robot_q_near: near robot configuration
- q_near: near cube configuration
- q_rand: near cube configuration
- discretisationsteps: number of descretisation steps allowed
- delta_q: user defined parameter to define the distance between cube configurations
NEW_CONF_CUBE(robot_q_near, q_near, q_rand, discretisationsteps, delta_q=None)
OUT:
- last_valid_cube: valid cube configuration
- last_valid_robot: valid robot configuration
IN:
- robot_q_new: new robot configuration
- q_new: new cube configuraiton
- q_goal: new goal configuration
- discretisationsteps: number of descretisationsteps allowed
VALID_EDGE(robot_q_new, q_new, q_goal, discretisationsteps)
OUT:
- distance between goal configuration and the cube configuration < 2e-2
IN:
- G: graph of all the fcomputted configurations
getpath(G)
OUT:
- path: array of each node in the graph

Figure 2: Description of each of the helper functions in *computepath* function in path.py for motion planning

precise force control. All components required for this task are implemented in ‘control.py‘ to run the robot’s motion using calculated torques.

3.2.1 Trajectory

Objective: The trajectory of the robot is computed from the path found in the motion planning stage. We decided to use Bezier curve to complete the main objective of this function due to the reasons below:

- Bezier curves use interpolation between points in the path, helping create smooth curves given a reference path.
- It’s easy to control the shape of the graph by adjusting the handles on a curve.

- The trajectory generated using the Bezier curve method is in the convex hull of the control points, or the path. This means that when optimising the trajectory, the curve satisfies collision constraints on the path.

Reconstructing the path with duplicates of node minimised the error between the path and the trajectory so that the robot does not collide with the obstacles. The implementation for this is done in *maketraj* function. The figure 3 demonstrates the difference in the trajectory based on the original path and the path with duplicate points. The graph on the left where the curve is generated from the original points has smoother curve. This causes collisions with significantly high possibility. The convex hull of the curve moves closer to the point after adding copies of the reference points, as shown in the graph on the right side of the figure 3.

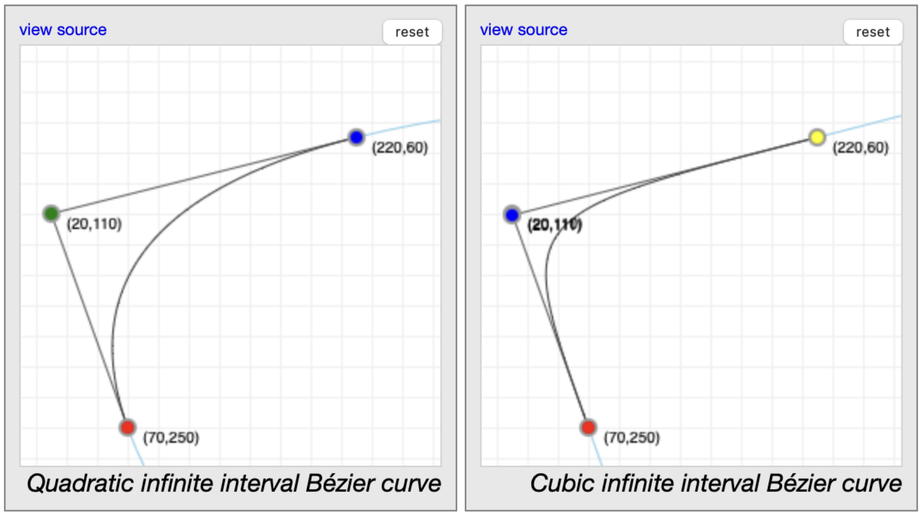


Figure 3: Planned path and Bezier curve (left); Planned path with duplicate of a point and the Bezier curve(right) [5].

Inputs:

1. **q0:** A control point, or the start of the path.
2. **q1:** The other control point, or the end of the path.
3. **T:** Time parameterisation.

Outputs:

1. **q_of_t:** position at time T
2. **vq_of_t:** velocity at time T
3. **vvq_of_t:** acceleration at time T

Justification: In the path reconstructed to generate Bezier curve, the number of copies that is sufficient for the curve to be for fitted to the original path planned was found by sequences of experiments. When the minimum number that Bezier function generates the most desirable motion simulation was reached, we decided to use the number.

Considering that the most important objective of dynamics is to make the robot follow the planned path as smooth as possible, the Bezier curve plays an important role in this task. Using the Bezier curve is beneficial for the fulfillment of the task as it illustrates smooth and precise curve based on the reference path [8].

3.2.2 Control

Objective: The main objective of this component is to make the robot perform the motion with the right force to hold the cube. We decided to use proportional-derivative controller for the control law. K_p and K_v respectively, represent proportional gain and derivative gain. This is the same approach as proportional-integral-derivative (PID) controller, where each of the gain affects as demonstrated in the table below (1): As our robot model does not have force

Parameter	Rise time	Overshoot	Settling time	Steady-state error	Stability
K_p	Decrease	Increase	Small change	Decrease	Degrade
K_i	Decrease	Increase	Increase	Eliminate	Degrade
K_d	Minor change	Decrease	Decrease	No effect	Improve if K_d small

Table 1: A table to show how different gains affects

sensor, we decided to use PD controller for the joint space dynamics control. The PD controller is applied to the pipeline of the motion planning and robot motion control as the figure below (4).

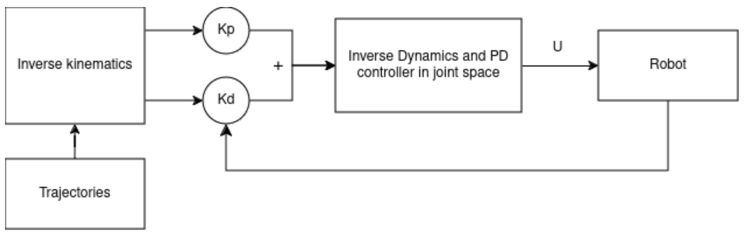


Figure 4: A block diagram of the robot manipulation with PD controller applied

Comprehensive consideration of these components led us to a decision to use inverse dynamics and joint space control. The main objective of this function to find the torque can be expressed in the canonical form below:

$$M(q)\ddot{q} + B(q)[\dot{q}\dot{q}] + C(q)[\dot{q}^2] + G(q) = \mathcal{T}$$

where M, B, C, and G are only configuration dependent, representing inertia matrix, coriolis matrix, centrifugal matrix, and gravity vector, in order. This is an inverse dynamics equation, which can be iteratively done using the Recursive Newton-Euler Algorithm (RNEA).

$$RNEA(q, \dot{q}, \ddot{q}) = \tau$$

Given the position and the velocity of the current time parameter, finding a desired acceleration implied PD-like behaviour around the reference trajectory. This can be computed by the formula below:

$$\ddot{q}_t^* = \ddot{q}_t^{\text{ref}} + K_p(q_t^{\text{ref}} - q_t) + K_d(\dot{q}_t^{\text{ref}} - \dot{q}_t)$$

Now given $q, \dot{q}, \ddot{q} \in \mathbb{R}$ denoting the joint positions, velocities, and accelerations of a 6 degree of freedom configuration, expression of the dynamics, or the joint torques, can use the Lagrangian form:

$$\tau = M\ddot{q} + h$$

where $M \in \mathbb{R}^{6\times6}$ is the inertia matrix and $h \in \mathbb{R}^6$ is the vector of centrifugal, gyroscopic, and Coriolis effects, and generalised gravity torque. Additively to this form of the expression, we used the contact constraint Jacobian J and the vector f_c of desired forces on both effectors in order to control the robot for lifting the cube. Then the dynamics can be expressed as below [6]:

$$\tau = M\ddot{q} + h + J^T f_c$$

Inputs:

- **sim:** This enables simulation of the motion.
- **robot:** This object contains data and information of the robot model.
- **trajs:** Positions, velocities, and accelerations of the current time parameter.
- **tcurrent:** This is the current time parametrisation.
- **cube:** The cube configuration information.

Outputs:

- **torque:** The appropriate torque with the right joint space control.

Justification: The decision to use inverse dynamics instead of forward dynamics is due to the difference in computational cost. As explained above, inverse geometry can be recursively computed using $RNEA$, parameterising it with $q, \dot{q}, \ddot{q}$. In other words, it does not require mass matrix to be given, while forward dynamics does. The complexity of inverse geometry is $O(n)$, which is remarkably faster than forward dynamics whose complexity is $O(n^2)$.

Regarding using joint space control, the reason is that the robot has constraints on the angle of the joints. It is important to use task space control for this task because the joint spaces are constrained to avoid collision with obstacles [7].

4 Results

This section analyses qualitative and quantitative evidence from experiments conducted on each component by varying parameters and observing the outcomes.

4.1 Motion Plannning

4.1.1 Inverse Geometry

The completion of this task can be observed as follows: `https://youtu.be/rCPXcYVG1gA`

The grasp phase intersects the object intentionally, as collision detection and joint limit adherence are incorporated within the path planning phase of our project.

To assess the robustness of our algorithm, we tested the grasp pose computation on randomly generated cube configurations with varied positions and rotations. The table 2 summarises the success rates of these trials, highlighting the cases where the grasp failed and their causes. We can see that the algorithm is accurate in computing grasp pose and only fails twice when the position was unreachable, as it should do.

4.1.2 Rapidly Exploring Random Trees

The intermediate steps and completion of this task can be observed as follows: `https://youtu.be/x75b-BPag6w`

The video illustrates how the algorithm samples and completes the RRT algorithm and in the end, displays the path the robot follows to take the cube from its initial position to its goal position.

To illustrate the robustness of this algorithm, we conducted three tests:

- How often does RRT finds a path, varying max_iterations?
- How often is a path found when varying the initial and target cube configurations?

Initial Cube Placement	**Target Cube Placement**	**Path Found**	**Reason if Path Found is False**
$\begin{bmatrix} 1 & 0 & 0 & 0.44 \\ 0 & 1 & 0 & -0.4 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0.5 \\ 0 & 1 & 0 & 0.12 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True	-
$\begin{bmatrix} -0.83907153 & 0.54402111 & 0 & 0.44 \\ -0.54402111 & -0.83907153 & 0 & -0.4 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -0.83907153 & 0.54402111 & 0 & 0.5 \\ -0.54402111 & -0.83907153 & 0 & 0.12 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True	-
$\begin{bmatrix} 1 & 0 & 0 & 0.2 \\ 0 & 1 & 0 & -0.1 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0.7 \\ 0 & 1 & 0 & 0.16 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True	-
$\begin{bmatrix} -0.14550003 & 0.98935825 & 0 & 0.2 \\ -0.98935825 & -0.14550003 & 0 & -0.1 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -0.9899925 & 0.14112001 & 0 & 0.7 \\ -0.14112001 & -0.9899925 & 0 & 0.16 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True	-
$\begin{bmatrix} 1 & 0 & 0 & 0.6 \\ 0 & 1 & 0 & -0.3 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0.3 \\ 0 & 1 & 0 & 0.2 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True	-
$\begin{bmatrix} -0.65364362 & -0.7568025 & 0 & 0.6 \\ 0.7568025 & -0.65364362 & 0 & -0.3 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -0.9899925 & -0.14112001 & 0 & 0.3 \\ 0.14112001 & -0.9899925 & 0 & 0.2 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	False	The positions for the cubes were unreachable.
$\begin{bmatrix} 1 & 0 & 0 & 0.23 \\ 0 & 1 & 0 & -0.2 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0.65 \\ 0 & 1 & 0 & 0.25 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True	-
$\begin{bmatrix} 0.66031671 & 0.75098725 & 0 & 0.23 \\ -0.75098725 & 0.66031671 & 0 & -0.2 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.40808206 & 0.91294525 & 0 & 0.65 \\ -0.91294525 & 0.40808206 & 0 & 0.25 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True	-
$\begin{bmatrix} 1 & 0 & 0 & 0.53 \\ 0 & 1 & 0 & -0.45 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0.8 \\ 0 & 1 & 0 & 0.19 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	False	The cube configurations were unreachable.
$\begin{bmatrix} 1 & 0 & 0 & 0.53 \\ 0 & 1 & 0 & -0.45 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0.54 \\ 0 & 1 & 0 & 0.19 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True	-
$\begin{bmatrix} 0.28366219 & 0.95892427 & 0 & 0.53 \\ -0.95892427 & 0.28366219 & 0 & -0.45 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -0.65364362 & -0.7568025 & 0 & 0.54 \\ 0.7568025 & -0.65364362 & 0 & 0.19 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True	-

Table 2: A table to show how different initial and target cube positions effect grasp pose.

- How do different ranges for the x, y, z effect the amount of time taken to find a path?

Varying Max Iterations: We tested the robustness of our algorithm by varying the maximum number of iterations allowed for pathfinding. By adjusting the k-values (max iterations), we observed how many iterations were required to either find or fail to find a path. The results are shown in Figure 5:

The graph shows that as iterations increase, the number of iterations taken to find a path also rises. However, RRT consistently finds a path, even with smaller iteration counts. This efficiency is likely due to our sampling strategy, which biases the cube's z-value to ensure it can be lifted over obstacles when needed.

Varying Initial and Target Configurations: To evaluate the robustness of our algorithm, we uniformly sampled initial and target cube configurations within defined ranges of translation and rotation. These configurations were then tested by iterating over a list of initial and final positions to observe whether

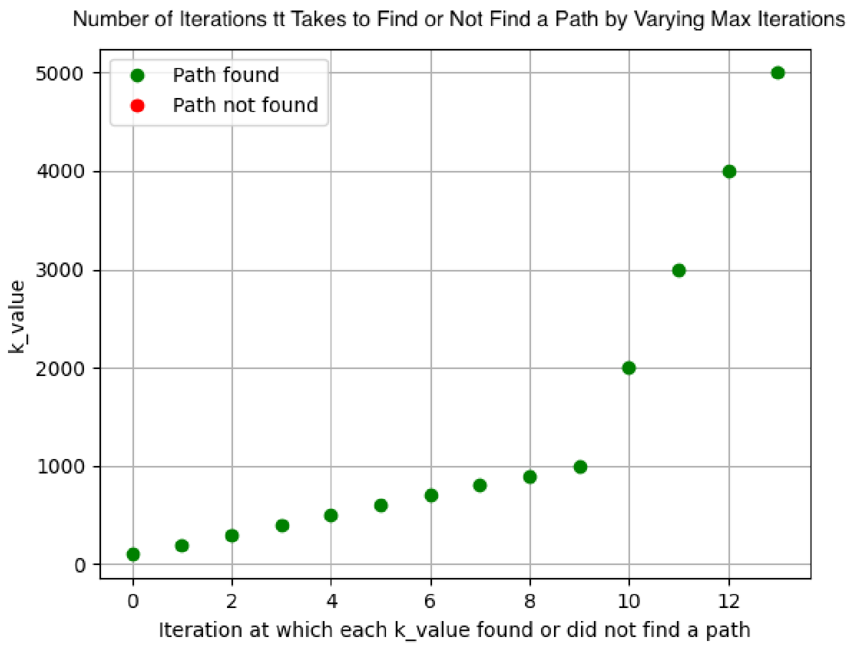


Figure 5: Varying the max iterations to see how many iterations it takes to find or not find a path.

a path was successfully generated. The results are shown in Table 3:

This table demonstrates that the RRT algorithm consistently generates a path for various initial and target cube configurations, highlighting its robustness. However, we believe there is room for improvement in joint configuration optimization, particularly for the elbows and shoulders, as some configurations appeared slightly inaccurate.

Varying Sampling Ranges for the Cube Positions: We tested the robustness of our RRT algorithm by comparing the time taken to find a path with unconstrained and constrained sampling from a higher cube position. The results are shown in the figure 6.

The figure shows that constraining the sampling range significantly reduces the time to find a path. Even at its optimal range multiplier, the unconstrained search takes as long as the constrained search at its highest time, demonstrating the efficiency and robustness of using a constrained sampling space.

Initial Cube Position	Target Cube Position	Path Found
$\begin{bmatrix} 0.79253398 & -0.60982775 & 0 & 0.33 \\ 0.60982775 & 0.79253398 & 0 & -0.4 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.82967677 & 0.55824409 & 0 & 0.4 \\ -0.55824409 & 0.82967677 & 0 & 0.08729505 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True
$\begin{bmatrix} 0.83727424 & -0.54678318 & 0 & 0.33 \\ 0.54678318 & 0.83727424 & 0 & -0.4 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.99865236 & -0.05189867 & 0 & 0.4 \\ 0.05189867 & 0.99865236 & 0 & 0.16241778 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True
$\begin{bmatrix} 0.98509763 & -0.17199613 & 0 & 0.33 \\ 0.17199613 & 0.98509763 & 0 & -0.4 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.73945088 & -0.67321052 & 0 & 0.4 \\ 0.67321052 & 0.73945088 & 0 & 0.14457232 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True
$\begin{bmatrix} 0.99492098 & -0.100659 & 0 & 0.33 \\ 0.100659 & 0.99492098 & 0 & -0.4 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.94858883 & 0.31651104 & 0 & 0.4 \\ -0.31651104 & 0.94858883 & 0 & 0.18707221 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True
$\begin{bmatrix} 0.88183033 & -0.47156683 & 0 & 0.33 \\ 0.47156683 & 0.88183033 & 0 & -0.4 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.78305036 & -0.6219583 & 0 & 0.4 \\ 0.6219583 & 0.78305036 & 0 & 0.20222611 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True
$\begin{bmatrix} 0.7528987 & 0.65813642 & 0 & 0.33 \\ -0.65813642 & 0.7528987 & 0 & -0.4 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.77651292 & -0.63010132 & 0 & 0.4 \\ 0.63010132 & 0.77651292 & 0 & 0.20739531 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True
$\begin{bmatrix} 0.83312943 & -0.55307808 & 0 & 0.33 \\ 0.55307808 & 0.83312943 & 0 & -0.4 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 0.92070169 & 0.39026708 & 0 & 0.4 \\ -0.39026708 & 0.92070169 & 0 & 0.09025268 \\ 0 & 0 & 1 & 0.93 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	True

Table 3: A table to show how different initial and target cube configurations effect whether a path is found.

4.2 Dynamics

This is a demonstration of our dynamics algorithm working within the pybullet simulator: `https://youtu.be/PKFHrZQw76E`.

4.2.1 Trajectory Optimisation:

The graphs below show the path and trajectory in 3D. While both have similar shapes, the direction of the furthest point from the start differs due to a negative force applied on the y and z axes.

Flattening these graphs into 2D to plot the z-values of the path and trajectory verifies that the Bezier curve and path reconstruction with duplicated nodes generated a precise trajectory. This is shown in the graphs below:

The trajectory smooths both the beginning and end of the path while closely following the computed path for most other sections, demonstrating the robustness of this approach.

4.2.2 Control

For the control law, proportional gain (K_p) and force values are user-defined. Several experiments were conducted to determine the optimal values for smooth

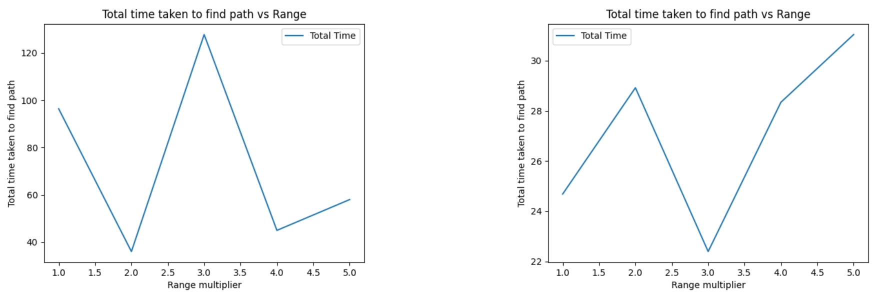


Figure 6: Figure (a) shows the time taken to find a path (in seconds) with an vaired and unconstrained range and figure (b) shows the time taken to find a path (in seconds) with a varied but constrained range.

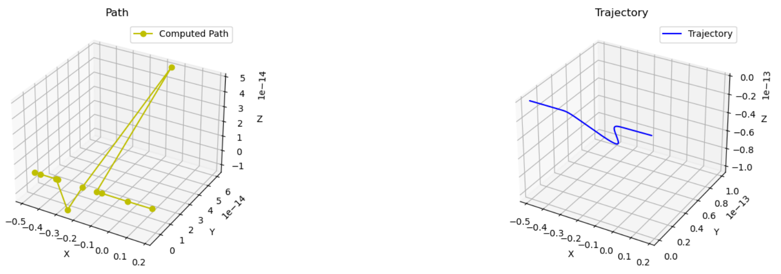
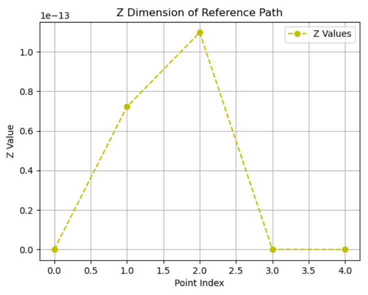


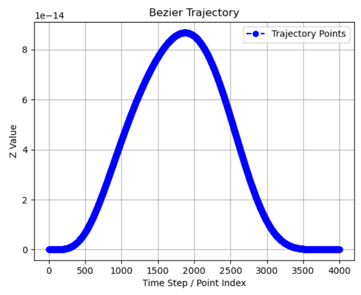
Figure 7: Path(a) and trajectory(b) plotted in 3D space

trajectory manipulation, where the robot securely holds and moves the object. To demonstrate the robustness of this computational framework, the success rate of robot manipulation is shown in the pie chart in Figure 9.

Most experiments resulted in successful manipulation, with the robot following the trajectory smoothly, avoiding obstacles, and securely holding the cube. This demonstrates the robustness of the control law applied.

Overall, dynamics of motion planning is implemented with appropriate control, which is verified in the graphs in Figure 10. Smooth and continuous curves for positions, velocities, and accelerations over time shows that the approach continuously generates the trajectory.





(a) (b)

Figure 8: Path (a) and trajectory (b) plotted in 2D space

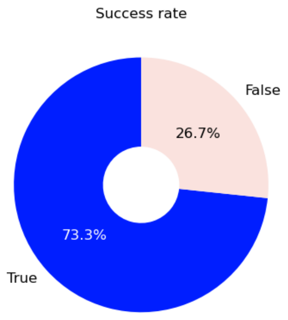


Figure 9: Success rate

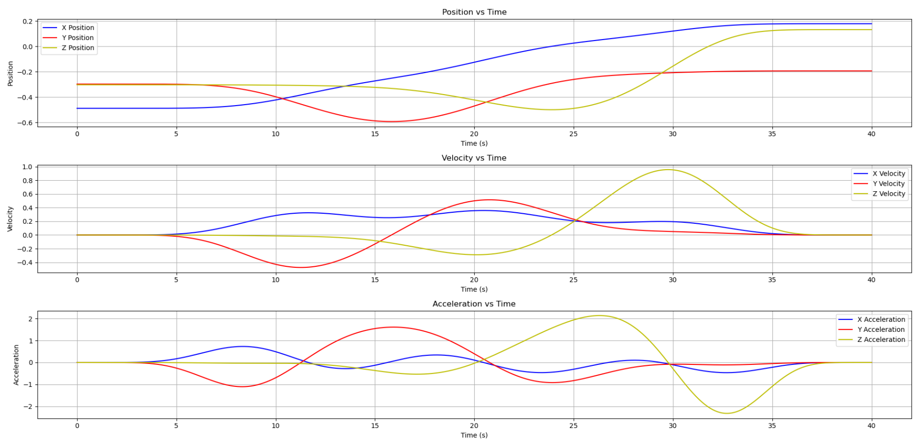


Figure 10: Positions, velocities, accelerations vs time

5 Discussion

In summary, our results showed that while many aspects of our algorithms performed well, there are areas for improvement. The inverse kinematics effectively handled various cube configurations, but adding kinematics for other joints could improve accuracy for more rotated configurations.

The RRT algorithm performed well once we constrained the sampling space, but it may need adjustment for more complex obstacles or tasks beyond simple lifting and placing. Nonetheless, the RRT proved robust under noise, quickly finding paths even with changing initial and target positions, and requiring only a few iterations.

As discussed in section 4.2.2, our dynamics performed well by maintaining close adherence to the planned path and applying sufficient force and gain to lift and move the cube effectively.

To improve accuracy, we could interpolate between path configurations to create sub-paths and iteratively smooth them with Bézier curves, which would be more precise than duplicating nodes. Additionally, because force and gain were sensitive to conditions, future work could focus on dynamically adjusting these based on obstacle proximity and grasping needs rather than experimenting which gave the most successful attempts.

References

[1] Long, H., Li, G., Zhou, F., & Chen, T. (2023). Cooperative Dynamic Motion Planning for Dual Manipulator Arms Based on RRT*Smart-AD Algorithm. Sensors, 23(18), 7759–7759. https://doi.org/10.3390/s23187759

[2] Hopcroft, J. E., Schwartz, J.-C., & Sharir, M. (1984). On the Complexity of Motion Planning for Multiple Independent Objects; PSPACE- Hardness of the "Warehouseman's Problem." 3(4), 76–88. https://doi.org/10.1177/027836498400300405

[3] Ding, J., Zhou, Y., Huang, X., Song, K., Lu, S., & Wang, L. (2023). An improved RRT* algorithm for robot path planning based on path expansion heuristic sampling. Journal of Computational Science, 67, 101937–101937. https://doi.org/10.1016/j.jocs.2022.101937

[4] Gao, K., Ye, Z., Zhang, D., Huang, B., & Yu, J. (2018). Toward Holistic Planning and Control Optimization for Dual-Arm Rearrangement. Retrieved November 13, 2024, from ar5iv website: https://ar5iv.labs.arxiv.org/html/2404.06758

[5] Pomax. *A Primer on Bézier Curves*. Available at: https://pomax.github.io/bezierinfo/#whatis. Accessed: 2024-11-14.

[6] Stéphane Caron, "Joint torques and Jacobian transpose," 2024. (Online). Available: https://scaron.info/blog/joint-torques-and-jacobian-transpose.html. Accessed: 14-Nov-2024.

[7] ScienceDirect, *Task Space*, Available at: https://www.sciencedirect.com/topics/engineering/task-space, Accessed: 2024-11-15.

[8] Lenovo. Bezier Curve. https://www.lenovo.com/us/en/glossary/bezier-curve/?orgRef=https%253A%252F%252Fwww.google.com%252F. Accessed: 2024-11-15.