UNIVERSITY OF EDINBURGH

ADVANCED ROBOTICS

ADVANCED ROBOTICS PRACTICAL

Authors:

Lewis Raeburn & Yu Chen Lee

S1744532 S2449600

Abstract

This report describes the implementation of the control of a robot to perform three main tasks including kinematics of the robot, PD control of the robot, and moving an object to a target position through pushing or grasping and docking. We have developed the robot capable of moving objects to a target position in a confined environment. It implements trajectory planning in the task space to avoid collisions with obstacles. We also achieve PD controls with the movement of the robot controlled by torque generated. The robot is able to achieve the task of moving objects to a target position with error less than 30mm.

THE UNIVERSITY *of* EDINBURGH

Introduction

In this project, we investigate the methods involved in controlling a robot to achieve three main tasks.

The first task is to compute the kinematics of the robot. Forward kinematics of the robot is computed by using the homogenous transformation matrix which includes the rotational matrix and the translation vector of each joint. To deduce the inverse kinematics of the robot, the Jacobian matrix for each joint, including position Jacobian and orientation Jacobian is computed. To find the inverse kinematics of the robot, we compute the Moore-Penrose pseudo-inverse of the Jacobian matrix with the position and orientation error.

The second task in this project is to implement PD controls for the robot for the robot to move joints by generating torque. To achieve this, the PD gains of every single joint of the robot is first tuned individually before fine-tuning them for the robot to move to a desired position with every joint moving together. The tuning of the PD gains is based on the torque equation.

The third task in this project is broken down into two subtasks, which is to push a cube to a target position, and to grasp and dock an object to a target position. To achieve the tasks, we perform task space planning which involves task space interpolation. We generate trajectories using cubic spline interpolation for the motion planning purpose.

Methods

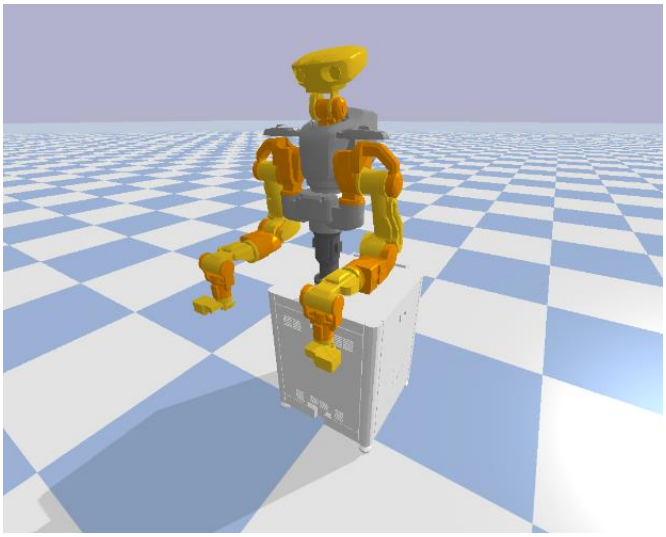

Figure 1: Nextage robot

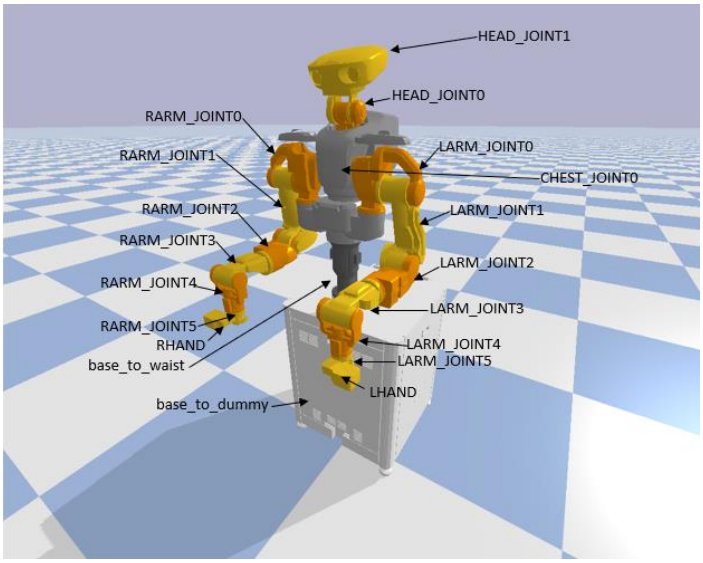

Figure 2: Nextage robot with joints labelled

We work with a Nextage robot as shown in Figure 1 for the tasks. The robot consists of 17 joints in total, including 1 for dummy joint/waist joint/chest joint each, 2 for head joints, and 6 for joints in left/right arm. The joints are labelled and displayed in Figure 2 with the first joint, "base_to_dummy" being a virtual joint and the rest are all revolute joints. The dimensions for each joint are stored in the URDF files.

Task 1: Kinematics

In robot kinematics, forward kinematics is the term used to describe the process of using a robot's kinematic equations to determine the location of the end effector given provided values for the joint parameters.

To determine the forward kinematics for the robot, we compute the Homogenous Transformation Matrix for each joint in the form of:

$$T_{A\to B}(\theta) = \begin{pmatrix} R & p \\ 0 & 1 \end{pmatrix}$$

which R is the rotational matrix and p is the translation with respect to the local frame.

To avoid unwanted exceptions during simulation, we couple the base joint and the waist joint as manipulating the 'fixed' joints (base and waist) will cause the program to crash. The frame directly below the chest joint, the waist is therefore defined as the world frame. The joints of the robot are all positioned at the origin with respect to the local frames.

Therefore, to compute the forward kinematics of a specific joint with respect to the world frame, for example, the 5th joint in left arm (LARM_JOINT5), the Homogenous Transformation Matrix will be:

$$\begin{aligned} &T_{W\to LARM\_JOINT5}(\theta) \\ &= T_{W\to CHEST\_JOINT\_0}(\theta)\cdot T_{CHEST\_JOINT0\to LARM\_JOINT\_0}(\theta)\cdot T_{LARM\_JOINT0\to CHEST\_JOINT1}(\theta) \\ &\cdot T_{LARM\_JOINT1\to LARM\_JOINT2}(\theta)\cdot T_{LARM\_JOINT2\to LARM\_JOINT3}(\theta) \\ &\cdot T_{LARM\_JOINT3\to LARM\_JOINT4}(\theta)\cdot T_{LARM\_JOINT4\to LARM\_JOINT5}(\theta) \end{aligned}$$

$$T_{W\to LARM\_JOINT5}(\theta) = \begin{pmatrix} R & p \\ 0 & 1 \end{pmatrix}$$

Inversely, inverse kinematics uses a robot's kinematic equations to obtain the motion of a robot to reach a target position which can be explained by the equation:

$$dq = J(q)d\emptyset$$

where q is the robot configuration and $\emptyset$ is the kinematic map that maps the robot configuration to the position/orientation of the end effector.

Thus, to compute the inverse kinematics for the robot, we first calculate J(q), the Jacobian Matrix for the robot, which consists of the position Jacobian and orientation Jacobian:

$$J_{pos}(q) = \begin{pmatrix} [a_1 \times (p_{eff} - p_1)] \\ \vdots \\ [a_n \times (p_{eff} - p_n)] \end{pmatrix}^T \in \mathbb{R}^{3\times n}$$

$$J_{ori}(q) = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}^T \in \mathbb{R}^{3\times n}$$

$$J(q) = \begin{bmatrix} a_1 \times (p_{eff} - p_1), & \cdots, & a_n \times (p_{eff} - p_n) \\ a_1, & \cdots, & a_n \end{bmatrix}_{6\times n}$$

where a_n is the orientation axis of the n-th joint, p_{eff} is the location of the end effector, and p_n is the location of the n-th joint. The variable n depends on the number of joints interconnected from the world frame to the end effector. Taking joint 5 in left arm (LARM_JOINT5) as the example, n would be 7.

```
function inverseKinematics(self, endEffector, targetPosition, orientation):
        dy = targetPosition - endEffectorPosition
        thetax = orientation[0]

        thetay = orientation[1]
        thetaz = orientation[2]
        Rx = np.matrix([[1, 0, 0],
                        [0, np.cos(thetax), -np.sin(thetax)],
                        [0, np.sin(thetax), np.cos(thetax)]])
        Ry = np.matrix([[np.cos(thetay), 0, np.sin(thetay)],
                        [0, 1, 0],
                        [-np.sin(thetay), 0, np.cos(thetay)]])
        Rz = np.matrix([[np.cos(thetaz), -np.sin(thetaz), 0],
                        [np.sin(thetaz), np.cos(thetaz), 0],
                        [0, 0, 1]])
        desiredOrientationMatrix = Rz*Ry*Rx
        m = desiredOrientationMatrix * endEffectorRotationalMatrix.T
        param = (orientation[0,0]+orientation[1,1]+orientation[2,2]-1)/2
        if param > 1:
            param = 1
        nue = math.acos(param)
        if nue == 0:
            do = np.array([0, 0, 0])
        else:
            r = 1/(2*math.sin(nue)) * np.array([m[2,1]-m[1,2], m[0,2]-m[2,0],
m[1,0]-m[0,1]])
            do = r * math.sin(nue)
        J = self.jacobianMatrix(endEffector)
        dy_do = np.concatenate((dy, do))
        dTheta = np.linalg.pinv(J) @ dy_do
        return dTheta
```

Figure 3: Simplified code snipper of inverse kinematics algorithm

After obtaining the Jacobian Matrix, we compute the inverse kinematics of the robot. The simplified code snippet of the inverse kinematics algorithm is illustrated in Figure 3. Based on Figure 3, we first calculate the difference between the target position and end effector current position. Then, since the manipulator of the robot does not have spherical wrist, we use equivalent angle representation to represent the difference in angle (orientation error) using the algorithm below [1]:

$$e_O = r\sin\vartheta$$

where r and ϑ can be calculated using matrix multiplication:

$$R(\vartheta, r) = R_d R_e^T(q)$$

where

$$\vartheta = cos^{-1}(\frac{r_{11} + r_{22} + r_{33} - 1}{2})$$

$$r = \frac{1}{2sin\vartheta}\begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix}$$

We then stack the position difference with the orientation error in an array, dy_do. We obtain dTheta, which is the motion of the robot required to reach the target position and orientation by computing the Moore-Penrose pseudo-inverse of the Jacobian Matrix with dy_do.

To move the end effector to a target position, we define the number of interpolation steps required to interpolate the end effector positions and compute the dTheta for each interpolation step.

Task 2: Dynamics

For task 2, we focus on implementing a closed-loop PD controller based on dynamics to control the joints of the robot. Given the current and target joint positions and joint velocities, the controller would generate a torque to move the joints of the robot based on the dynamic system:

$$u(t) = k_p\left(x_{ref} - x(t)\right) - k_d\dot{x}(t)$$

where t is the time, k_p and k_d are PD gains respectively, x_{ref} is the target position, x(t) is the current position, and $\dot{x}(t)$ is the velocity.

By implementing the torque equation in a function, the PD gains are loaded and tuned individually for every single joint and saved to the file 'core/PD_gains.yaml'.

To move a specific joint to a target position using the PD controller, we need the current position and velocity of the joint. We detect the joint revolutional position by using a built-in method from PyBullet, getJointPos(). As for retrieving the joint velocity, we divide the difference between the joint current position and the joint's previous position one step before by the time taken to do so.

Task 3: Nextage Robot Manipulation

In Task 3, there are two subtasks to be completed. The first task is to push an object to a target position. The second task is to grasp and dock an object to a target position.

To complete the tasks in a more efficient manner, we develop a function which performs polynomial interpolation (cubic spline) between a set of given points to generate the trajectories for the joint of the robot to complete the tasks.

The default axis directions are as illustrated in Figure 4.

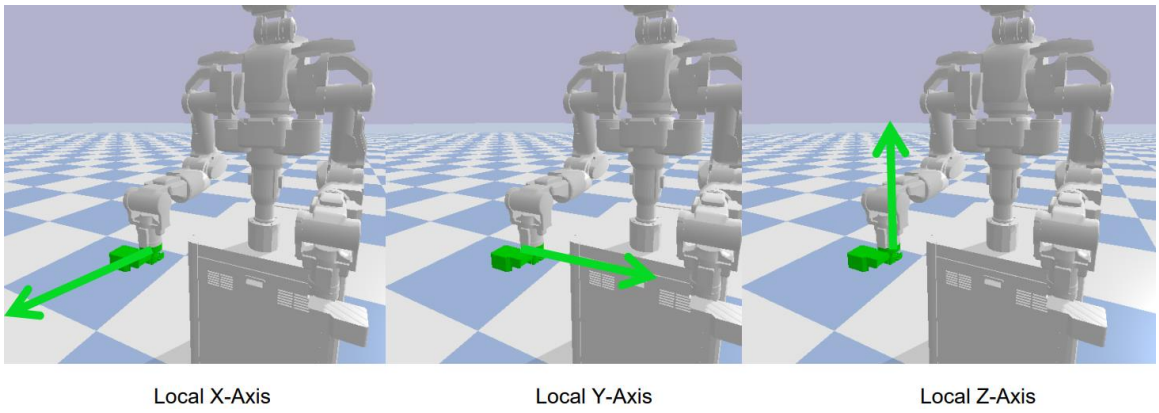

Figure 4: Default axis direction

For the first task, we compute three trajectories for the end effector of the robot to push the cube from its original position to a target position.

The end point for the first trajectory is located at the red dot in Fig. The purpose of placing the point in this location is to avoid the collision between the end effector and the cube. Then, the end point for the second trajectory is situated at the blue dot in Figure 5, which is 8 millimeters away from the x-axis of the cube. The end point for the third trajectory is located at the target.

Trajectories 1 and 2 are cubic spline interpolations whereas Trajectory 3 is a linear interpolation.

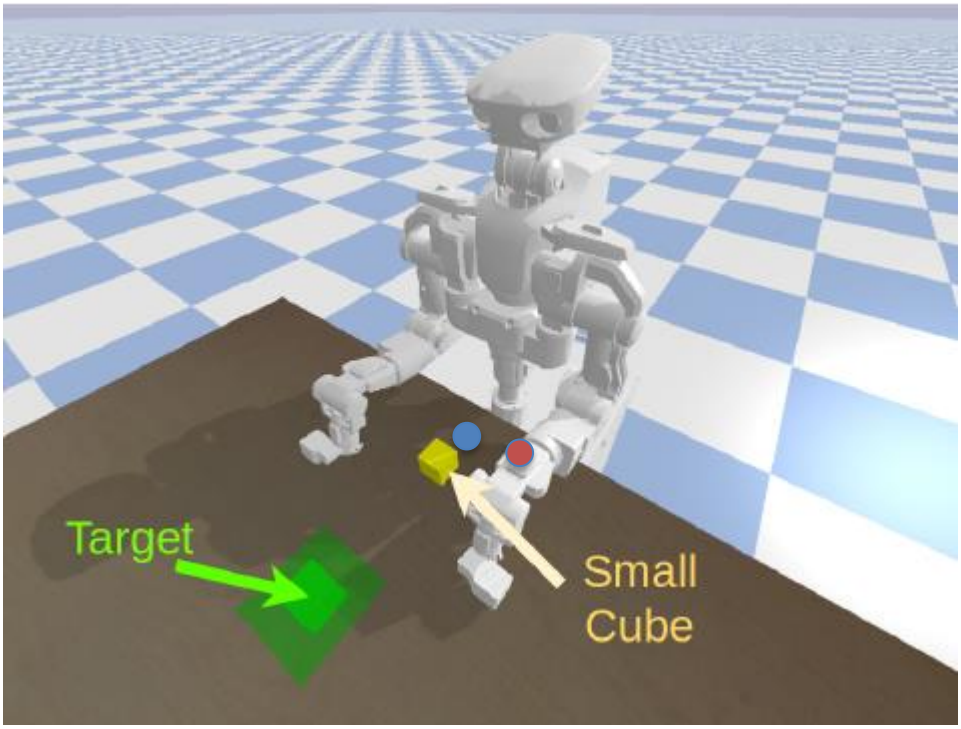

Figure 5: Robot position in first task in Task 3

For the second task, we generate four trajectories for the end effectors of the robot to grasp and dock a dumbbell from its original position to a target position.

For the first trajectory, we define the end points of the trajectory for the left hand at the green dot and right hand at the blue dot in Figure 6. We then apply an orientation of 90° for both end effector to ensure the end effectors could grasp the cube. We generate the second trajectory for the end effectors to grasp the cube to go up along the z-axis till the base of the cube is higher than the obstacle in red. The first two trajectories are cubic spline interpolations and planned in the way that the arms of the robot are moved with respect to the chest frame.

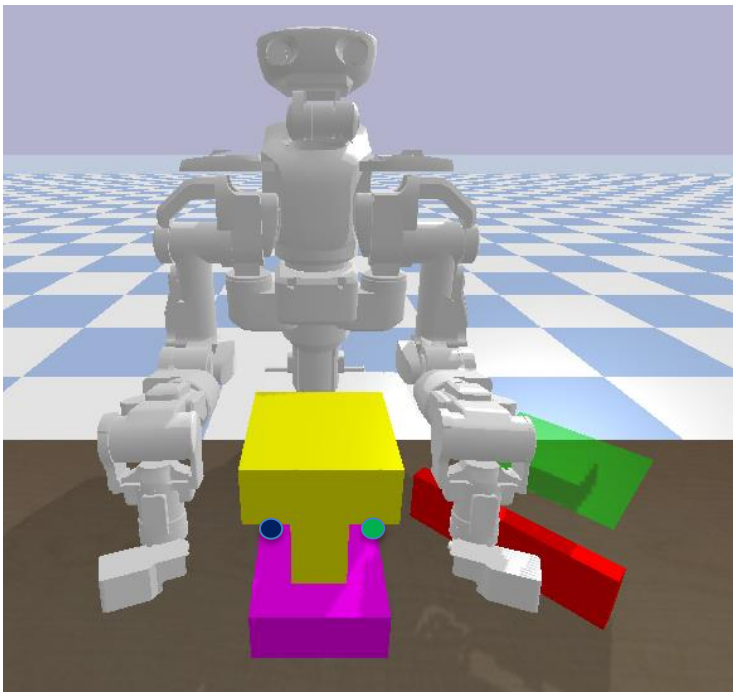

Figure 6: Robot position for second task in Task 3

For the third trajectory, we rotate the chest by 47°, while keeping the arms in fixed joint positions to move the dumbbell to the top of the target position. For the last trajectory, we move the dumbbell downwards by generating cubic spline interpolation.

Results

Task 1: Kinematics

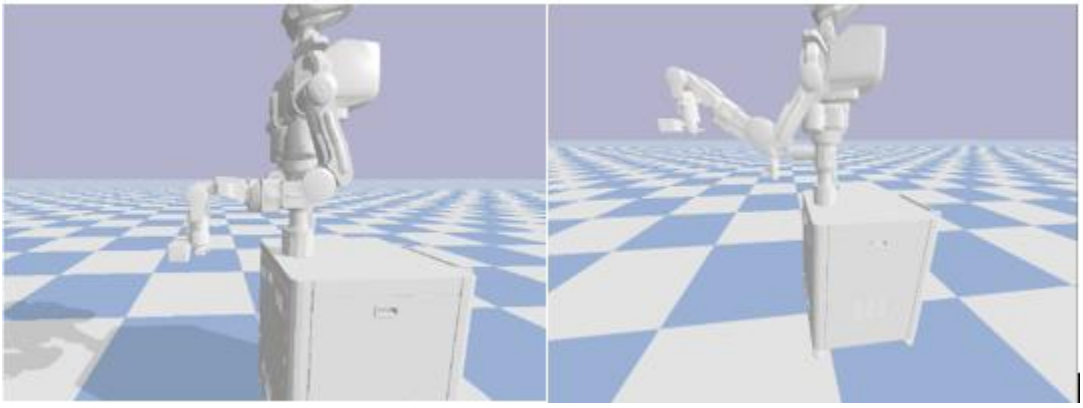

Figure 7: Initial and final position after moving 'LHAND' to position [0.61, 0.16, 0.29]

The final position of the robot after moving 'LHAND' to position [0.61, 0.16, 0.29] are as illustrated in Figure 7. Figure 8 shows the front view of the robot after the movement.

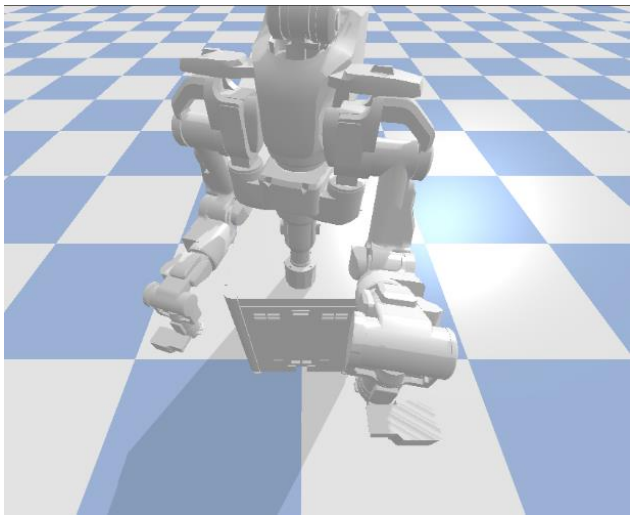

Figure 8: Front view of the robot after moving 'LHAND' to position [0.61, 0.16, 0.29]

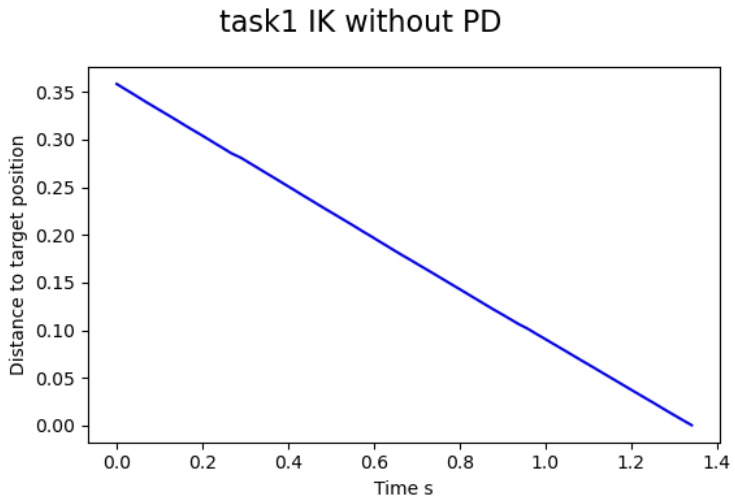

Figure 9: Time taken for 'LHAND' to reach position [0.61, 0.16, 0.29]

Based on Figure 9, we can see that the 'LHAND' of the robot moved to the desired position in less than 1.4 seconds.

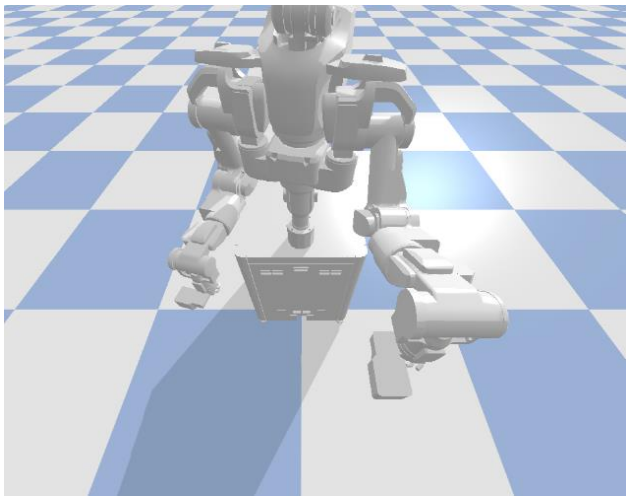

Figure 10: Front view of the robot after moving 'LHAND' to position [0.61, 0.16, 0.29] with orientation 90 degrees

Beside that, other than moving 'LHAND' to position [0.61, 0.16, 0.29], we also applied orientation control to 'LHAND' which we can observe in Figure 10 that the 'LHAND' of the robot rotated 90°around z-axis as well as moving to the intended point.

Task 2: Dynamics

In Task 2, we tuned the PD gains by first tuning the proportional gains before tuning the derivative gains. Initially, we tune the gains for the joints individually to be critically damped at 45°. Then, when we try to move all the joints of the kinematic chain, the expected movement is not ideal. Thus, we tweak the values for the PD

gains slightly in order to achieve the desired motion when trying to move every joint simultaneously.

The optimized PD gains for each joint are as shown in Table.

Table 1: Optimized PD Gains

	Kp	Kd
CHEST_JOINT0	75	20
HEAD_JOINT0	10	1
HEAD_JOINT1	10	1
LARM_JOINT0	50	10
LARM_JOINT1	400	30
LARM_JOINT2	300	30
LARM_JOINT3	500	30
LARM_JOINT4	300	30
LARM_JOINT5	10	1
RARM_JOINT0	50	10
RARM_JOINT1	400	30
RARM_JOINT2	300	30
RARM_JOINT3	500	30
RARM_JOINT4	300	30
RARM_JOINT5	10	1

For the right arm and left arm joints, to prevent instability, the mirror joints have identical values for the PD gains. For example, 'LARM_JOINT0' has the same PD gains as 'RARM_JOINT0'.

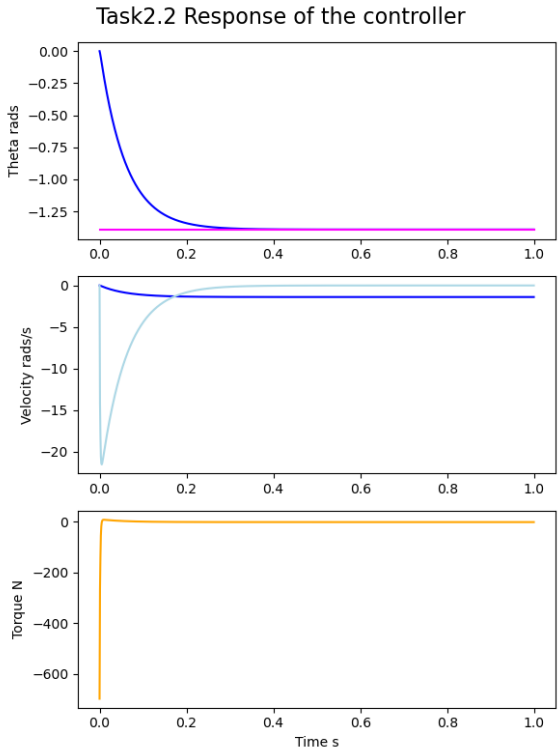

Figure 11: Response of PD controller when tuning 'LARM_JOINT3'

Figure 11 is an example of how one of the joints, 'LARM_JOINT3' behaved when the PD gains are tuned optimally.

After tuning the PD gains, we move the 'LHAND' joint to the same position, [0.61, 0.16, 0.29] as in Task 1 using PD control. Figure shows that by using the PD controller, 'LHAND' reached the target in around 30 seconds. We notice a slight bump at the beginning of the movement of the joint.

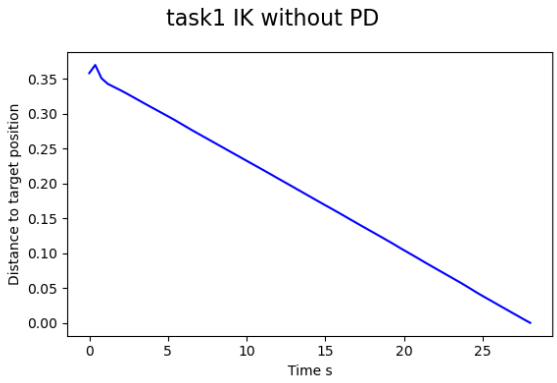

Figure 12: Time taken for 'LHAND' to reach position [0.61, 0.16, 0.29] using PD control

Task 3: Nextage Robot Manipulation

Task 3.1

For the first task in Task 3 which is to push a cube to a target position, the first trajectory the end effector of the robot, 'LHAND' took is as illustrated in the dotted lines in Figure 13 which is a cubic spline interpolation. The goal of computing the first trajectory this way is to avoid the end effector to stumble upon the cube while trying to get behind the cube. It took 25 seconds for 'LHAND' to reach the first trajectory end point from its original position.

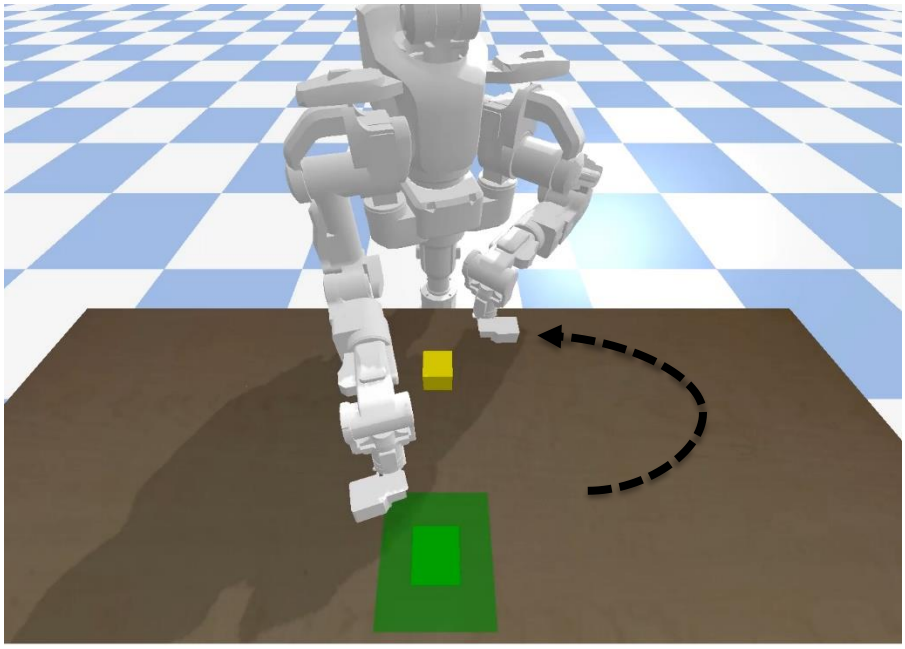

Figure 13: First trajectory for task 3.1

The second trajectory 'LHAND' took is as shown in the dotted line in Figure 14. which is also a cubic spline interpolation. It took 8 seconds for 'LHAND' to reach the second trajectory end point from the first trajectory end point.

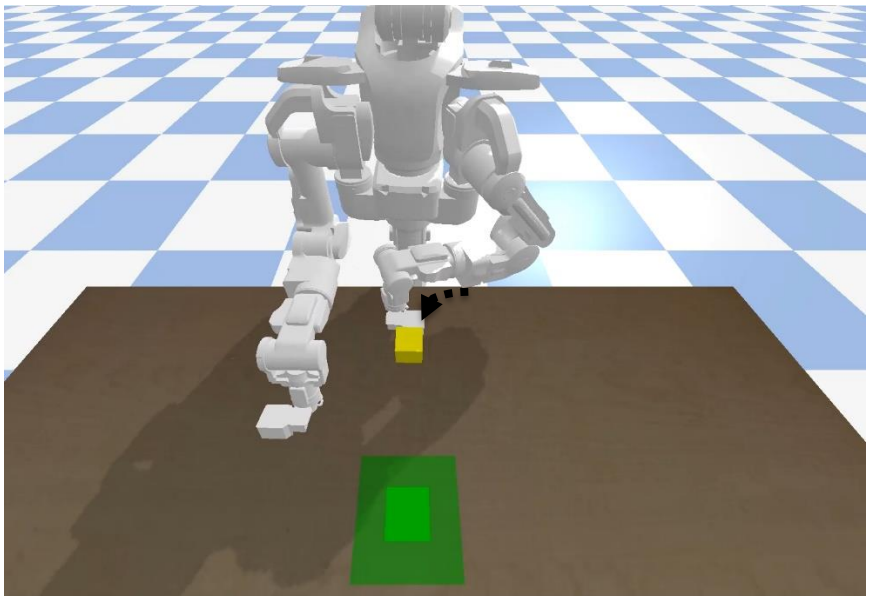

Figure 14: Second trajectory for task 3.1

For the last trajectory, it is a linear interpolation as the cube is in the same y-axis as the target position. Therefore, only a straight push is needed from 'LHAND' as shown in Figure 15. It took 20 seconds for the cube to reach the target position, resulting in 53 seconds in total from the starting position to the target position using the trajectories computed.

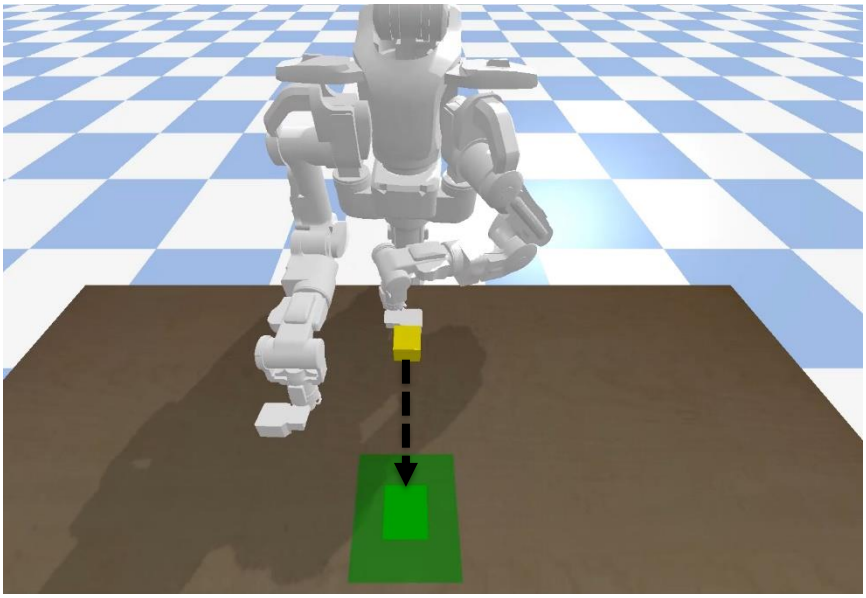

Figure 15: Third trajectory for task 3.1

Task 3.2

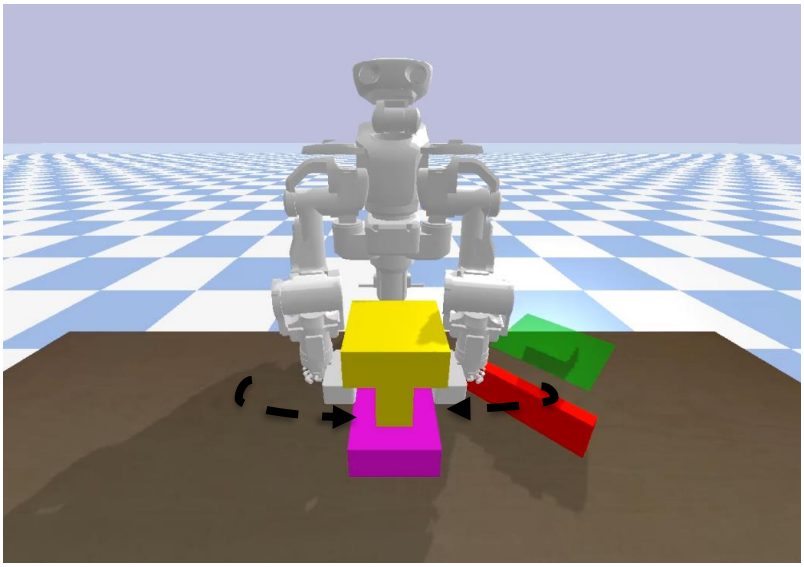

Figure 16: First trajectory for task 3.2

For the second task in Task 3, in the first trajectory which is a cubic spline interpolation as shown in Figure 16, 'LHAND' and 'RHAND' joints of the robot are used simultaneously. Both arms are moved with respect to the chest frame to ensure that the chest joint would not be moving and thus would not cause interference to the movement of the arms. When moving towards the first trajectory end point, the 'LHAND' and 'RHAND' are also rotated by 90° in order to grasp the dumbbell. The first trajectory took 25 seconds.

For the second trajectory, the dumbbell is lifted along the z-axis and forward along the x-axis until the base of the dumbbell is above the obstacle in red. The reason behind the movement along the x-axis is because there were some limitations in which the arms of the robot could not move up along the z-axis while maintaining the x-axis position. The second trajectory took 30 seconds.

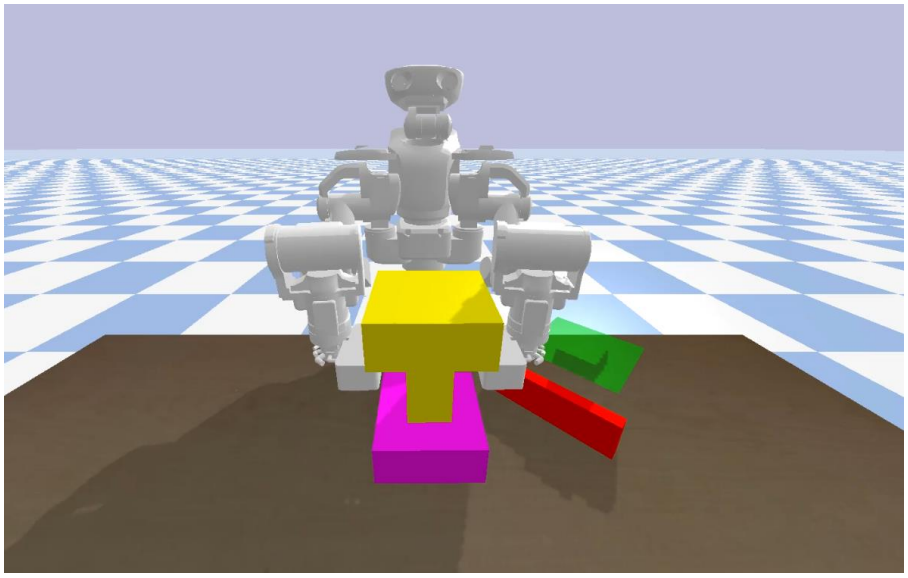

Figure 17: Second trajectory for task 3.2

For the third trajectory, the chest is rotated by 45° while keeping the arms in the same joint positions. This trajectory took 90 seconds.

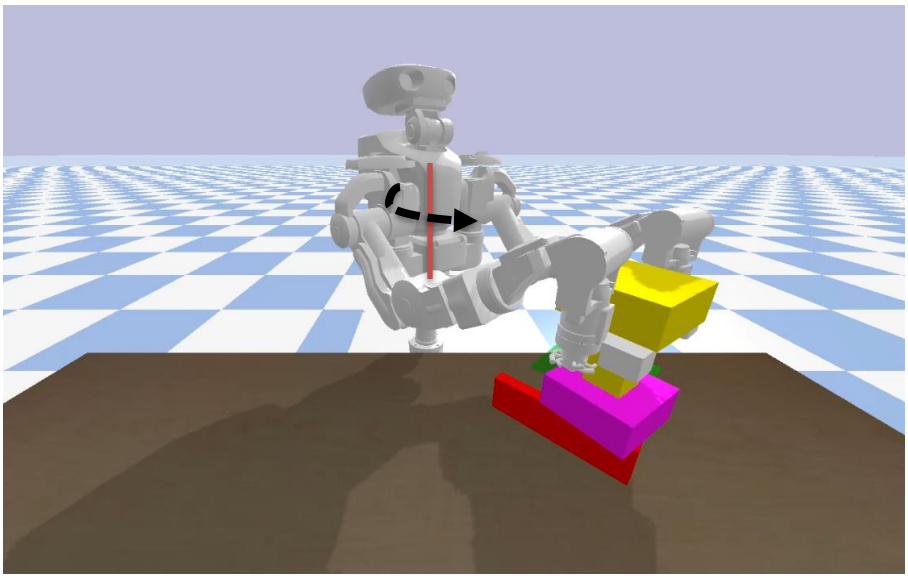

Figure 18: Third trajectory for task 3.2

For the last trajectory, the arms placed the dumbbell to the target position by moving downwards. The x-axis and y-axis target positions for the end effectors are

computed using Pythagoras' Theorem, as the target position are 45° away from the original position. The final trajectory took 30 seconds.

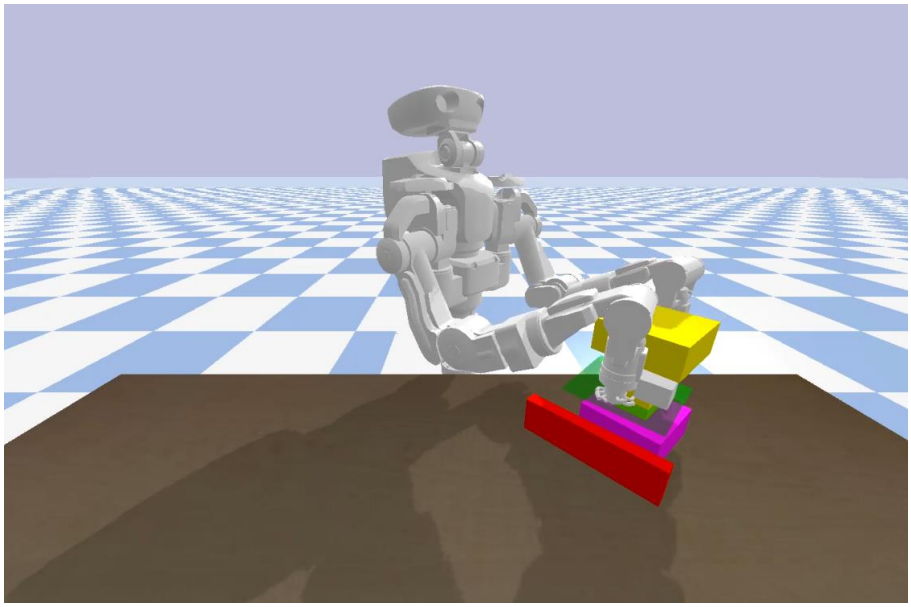

Figure 19: Fourth trajectory for task 3.2

The robot took 2 minutes and 30 seconds to complete the task.

Discussion

Task 1: Kinematics

To make achieving our tasks easier, we implemented orientation control as well as position control for our forward kinematics and inverse kinematics algorithms.

We used equivalent angle representation to represent the orientation error as our joints are revolute joints and do not have spherical wrists.

Task 2: Dynamics

After the process of tuning the PD gains, we found that this approach is not ideal more fine tunings are needed after tuning the joints individually for the joints to be moved simultaneously.

Other than that, a PD controller without the integral gain has the flaw that the error in steady state operation is not minimized. Given scenarios which gravity compensations exist and affect the movements of the robot, without the integral control, errors may occur more often during steady-state.

Furthermore, we suspect that due to the velocity control of the joint is disabled each time we start the movement control using PD, there exists a slight bump in the movement of the end effector.

Task 3: Nextage Robot Manipulation

Regarding the first of the two tasks of task 3, pushing the cube, due to the slight bump in the movement of the end effector upon starting the move_with_PD method, we were forced to concatenate the three trajectories into one array and pass this to move_with_PD as the set trajectory, as the slight bump seemed to ruin the motion. Given this constraint, we had to make sure that the transitions between the three trajectories were smooth, otherwise the movement would become unstable and fail. This was a challenge as we had to have one of the interpolation points be very close to the robot, so that the end effector would be moving in the direction of the target when the final trajectory began.

On the second of the tasks, grasping and docking, we decided to call move_with_PD multiple times and allow the slight bumps to happen. However, the slight bumps result in the dumbbell almost falling off at the end. Perhaps, it would have been better to generate a single trajectory with cubic interpolation.

On the same task, we ran into another problem. We found that when either end effector was too close to the robot and high up, the motion would become unstable and fail. For this reason, we came up with the idea to move the end effectors away from the robot while the dumbbell was being picked up. This way, the motion stayed smooth and completed successfully.

References

[1] B. Siciliano and L. Sciavicco, Robotics: Modelling, Planning and Control, Springer Publishing Company, Incorporated, 2008.

Notes

Link to the videos:
https://drive.google.com/drive/u/6/folders/1lwc4Qkh0zLbcpWm_cZ28gumRwxGMjNo2