INFR11213

Implementation of Humanoid Robot for Simple Tasks

s2455376
s2300928

2022/23 S1

Contents

1 Abstract

This document serves to provide insights into the implementation of basic manipulation tasks using a humanoid robot. It is firstly motivated by an introduction to the robot used, its specifications and use cases. Then we dive into the steps taken to achieve the robot manipulation, including forward kinematics, inverse kinematics, trajectory planning, dynamics controllers and testing. Then, the results of the manipulation tasks are supplied with evaluation. Finally, the report is concluded with a discussion on the best elements of the implementation, as well as its limitations and areas for further research.

2 Introduction

We are given a 3D-model of a Nextage Robot, which is an industrial humanoid robot developed by Kawada Robotics. More specifically, it is an upper-body humanoid robot and is fixed from the waist down, limiting the mobility of the robot. The robot has 17 degrees of freedom in its upper body, including 6 joints for each arm, 2 joints for the head, and 1 for the chest. Its flexibility makes it perfect to work with humans for assembly tasks.

Figure 1: Image of the Nextage Robot

The end goal of this practical experience is to manipulate the Nextage Robot to complete the following tasks:

- Push a cube to a target location
- Grasp object and place in target location

Both of which are typical activities a robot performs inside a factory to assemble a product.

We approached the end goal by breaking it down into sub-tasks and functions that will be needed, using knowledge about forward kinematics, inverse kinematics, controllers and so on. This approach is guided by the material 'ARO Practical Guidebook 2022'. As we complete each function, we perform unit testing and integration testing.

The way we have modularise the tasks can be seen in the following figure:

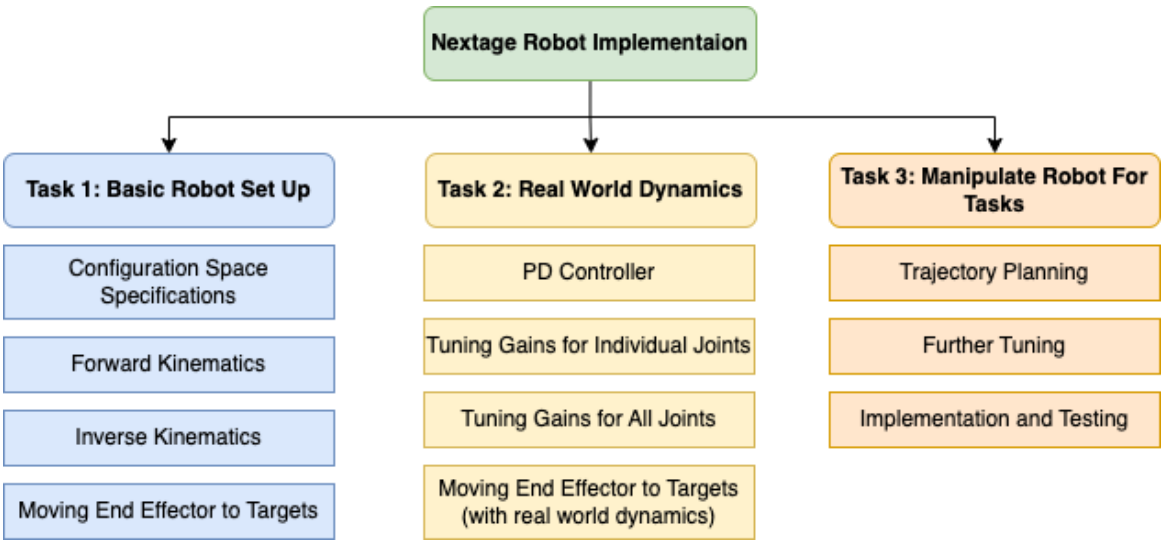

Figure 2: Breakdown of approach to manipulate the Nextage Robot for Task Completion

Finally, we piece the modules together to complete the simple tasks. We performed many testing until the task is completed at the requirements specified:

- Difference between target and final angles is less than 0.035 rads ($\sim 2°$)
- Object is less than 150mm away from target

3 Methods

3.1 Task 1: Basic Robot Set Up

This section describes the steps in order to manipulate the robot to perform a simple movement, without consideration of dynamics. The goal is to build the basics that can move the specified end-effector to a location in the world frame.

3.1.1 Configuration Space Specifications

The first step is to ensure the configuration space specifications are accurate. The translation and rotation axis of the joints need to be defined to reflect the dimensions of the Nextage Robot. This is important because these values would directly affect the homogeneous transformation matrix calculations. Note that these values are only relative to their parent joint, and not in world co-ordinates.

Therefore, it is important to document the relationships between the joints, so that we can multiply the transformation matrices in the kinematics chain to find each joint in the world co-ordinate. We decided that we needed a tree structure for this to happen. We expect each joint on the Nextage Robot to have one single parent, but it is possible for a node to have more than one child (for example, the chest joint is connected to the head joint, left arm and right arm). We are able to determine the relationship between the joints based on the topology diagram of the Nextage Robot from the 'ARO Practical Guidebook 2022'.

After defining these 3 variables, we have fully defined the Nextage Robot's configuration space. We have all we need to find the location of each joint, it just needs to be calculated using forward kinematics, which will be implemented in the next step using Python.

3.1.2 Forward Kinematics

Forward kinematics describes the procedures needed to calculate the position and orientation of every joint, link and the end-effector. This was necessary as we need to know where the end-effector is relative to an absolute co-ordinate system, which we define as the world frame. To do this, the following steps are required in order:

- Obtain homogeneous transformation matrix for each joint,
- Multiply homogeneous matrices in the desired end-effector's kinematics chain,
- Extract the rotation matrix and translation vector from the product.

To obtain the homogeneous transformation matrix for each joint, we first calculate the rotational matrix of each joint using information from the joint rotation axis dictionary in the `getJointRotationalMatrix()` function. If the corresponding axis value is 1, we calculate the rotational matrix about the vector using the formula in Figure 3, else it is left as an identity matrix to represent the lack of rotational freedom in that axis. The rotation matrix R_x, R_y, R_z about the x, y and z is then multiplied together to find R ($R = R_z R_y R_x$), the rotational matrix for the joint to be inputted into the transformation matrix. The joint angle, θ of each joint is obtained through the built-in function `getJointPos()`.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \quad R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \quad R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 3: Calculation of Rotatational Axis about each Axis.

Now that the rotational matrix for each joint is calculated, the transformation matrix can be calculated using the equation below, where the translation vector is appended after the rotational matrix and augmented to be a square matrix.

$$T = \begin{bmatrix} R & p_{translation} \\ 0_{1x3} & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & x \\ r_{21} & r_{22} & r_{23} & y \\ r_{31} & r_{32} & r_{33} & z \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

The homogeneous transformation matrix for each joint is calculated, but it is only relative to its parent joint. To calculate the joint's position relative to the world base frame, the kinematics chain is used. All the transformation matrices within the kinematic chain up to the joint in question are multiplied. Then from the resulting transformation matrix, the rotational matrix and positional vectors relative to the world frame can be extracted. For the Nextage Robot, There are 3 major kinematics chains. One for the left arm, one for the right arm and one for the head.

3.1.3 Inverse Kinematics

After finding out what the current joint angle configuration maps to in terms of the end-effectors Cartesian co-ordinates in the world frame, we need to think in reverse to answer the following question - 'What should my joint angles be for my end-effector to be in position $[x, y, z]$?'. To answer this question, we implement the `inverseKinematics()` function. This function takes a specified joint, and the desired target position and orientation, and returns a configuration of angles for each joint in the kinematic chain for the end-effector to be in the target co-ordinates.

A Jacobian Matrix is calculated using the robot's current configuration, and it helps approximate the direction the robot will move to given small increments. We have decided for the Jacobian Matrix to be a $6 \times N$ matrix to include both positions and orientations. The formula to calculate the Jacobian

is in figure 4. The number of columns 'N' changes based on the end-effector manipulated, but always equal to the length of the kinematic chain associated with the given joint. We chose to implement the $6 \times N$ Jacobian matrix because it was crucial for the orientation of the joints to be aligned.

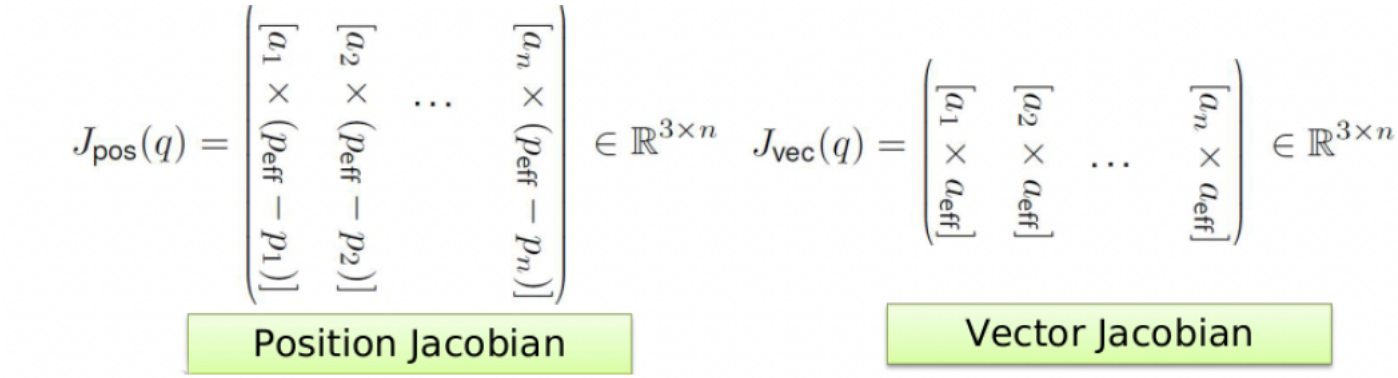

Figure 4: Caption

Finally, the change in configuration of joints to reach the next step is given by the dot product of the Jacobian matrix and the new target pose. This change is added back to the current configuration, thus giving the new configuration the robot needs to reach its target.

3.1.4 Moving End-effector to Targets

Lastly, we tie up the forward kinematics and inverse kinematics functions implemented into the method `move_without_PD()`. This function interpolates the trajectory of sub-targets that the end-effector should follow, and for each sub-target, it calls `inverseKinematics()` to store the next configuration of the robot into a shared dictionary and ticks the simulation one step forward in the `tick_without_PD()` function by directly resetting the joint states with the `resetJointState()` function, ignoring any dynamics. We have made the design choice to ignore performing any iterations within an interpolation step, thus this requires the number of interpolation steps to be greater than at least 1000, so that each sub-target is small enough for a Jacobian Matrix to be a good approximation method. The pseudocode for the `move_without_PD()` function is listed below:

Algorithm 1 Pseudocode to manipulate the Nextage Robot's desired end-effector to a target cartesian co-ordinate

function MOVE_WITHOUT_PD(self, endEffector, targetPosition, interpolationSteps = 1000)
  path ← Kinematics chain list from dictionary jointPathDict
  subtargets ← Linearly interpolate the trajectory using current end-effector position and target end-effector position over the number of interpolation steps

  distanceToTarget ← track the difference between current end-effector position and target position

  for interpolationSteps **do**
    $q*$ ← INVERSEKINEMATICS()
    Update the target positions for each joint in the shared jointTargetPos Dictionary to $q*$
    TICK_WITHOUT_PD( ) ▷ This resets the joint states
  end for

3.2 Task 2: Real World Dynamics

In the next stage, we make the simulation more realistic by considering the addition of gravity, and the fact that the robot's motion needs to be smoothed out using PID controllers to determine how much force is applied to each joint.

3.2.1 PD Controller

A closed-loop controller is implemented for every single joint on the robot. We use a closed loop controller instead of an open loop controller because it gives the robot feedback. To implement this, we need both the real and target joint positions and velocities to calculate the error term, and therefore the torque needed to be applied to each joint. This torque is calculated in the following equation.

$$u_t = k_p(x_{ref} - x_t) + k_d(\dot{x}_{ref} - \dot{x}_t)$$

Where k_p, k_d are the proportional and derivative gains that will be tuned. x_{ref} and $\dot{x}_{ref}$ are the target joint angle and velocity. x_t and $\dot{x}_t$ describes the current joint angle and joint velocity. We have chose to only use the PD controller, compared to a full PID controller because is usually sufficient and easier to tune 2 gains than 3. However, the lack of integral controller could lead to a steady state error, but was a trade-off we were willing to accept as long as the end-effector could reach the target within threshold.

The current joint velocity was approximated. We stored the angles of all the relevant joint in its last position. Then, we find the difference between the joints last angular position and the current joint angle, and divide this difference by the time increment in the simulation, `self.dt`.

3.2.2 Tuning Gains for Individual Joints

Tuning the gains for the joints was an iterative process, which mainly involved setting single target angles for each joint and observing the outputs to see if the movement of the joint was smooth, and reached the target. The outputs used for tuning were plots of the current joint angle and target angle against time, and a plot of the joint's current velocity against the target velocity against time. The closer the current angles followed the target the better. We followed the principles in the table below for tuning the gain values.

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

After tuning each joint to a satisfactory level with a single target, we also wanted to make sure that the joints can also follow a trajectory. Thus we implemented cubic hermit spline as a way to generate an array of targets and observed if the tuned PD gain values still allowed the joints to reach the target.

3.3 Task 3: Manipulate Robot for Tasks

3.3.1 Trajectory Planning with Way-points

Finally, the last task involves using basic manipulations to perform complex tasks. For both the cube pushing and dumbbell lifting tasks, this involves determining a route the end-effector will follow, and mapping out the key way-points the robot needs to reach before a change of state (i.e. change joint orientation or direction of motion).

Once the way-points for each task was selected, the next process was a repetitive tuning process to ensure the object reaches its targets.

4 Results

4.1 Task 1

In task 1, the end-effector 'LARM_JOINT5' was tasked to move to a position of $[0.37, 0.23, 1.06385]$ from an initial position of $[0.37, 0.23, 0.871]$. Figure 5 below shows the joint's distance to its target plotted against time in simulation, and the end-effector stopped at a precise co-ordinate of $[0.36999648, 0.23000528, 1.063077]$. This is accurate to the precision of 1mm. It reached the target in approximately 2 seconds, with a smooth motion that avoided any unnecessary motions. A video of the simulation can be found linked here. Figure 6 provides a visual aid of our implementation of the robot at the target position. We also observe that the orientation settings of the robot is working correctly, with the joint aligned with the z-axis. Overall, the robot's performance in task 1 is excellent, and shows that the basic components of the robot are working.

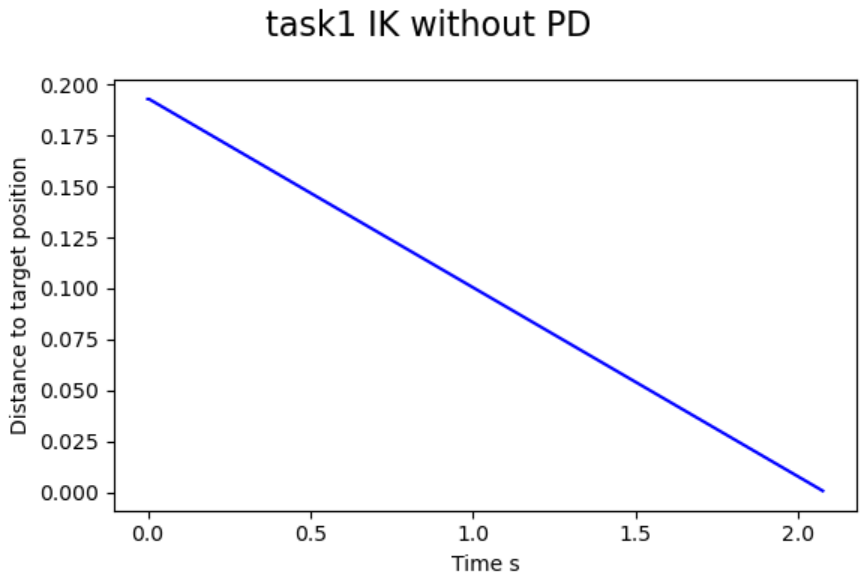

Figure 5: End-Effector LARM_JOINT5 Distance to Target against Time

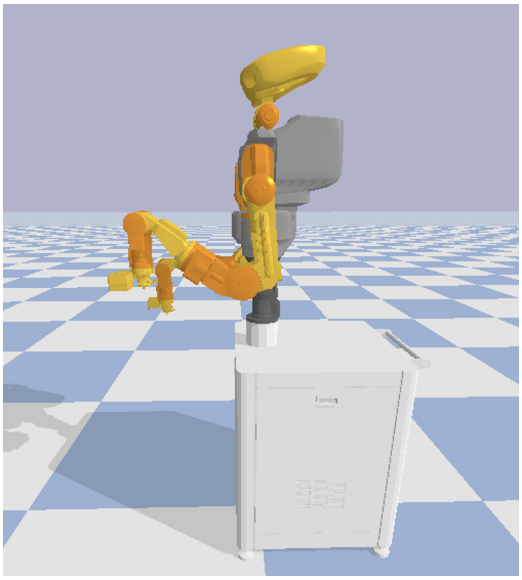

Figure 6: Nextage Robot Simulation Result for Task 1

4.2 Task 2

Although all the gains for the joints have been tuned, we will focus on the joint LARM_JOINT2.

LARM_JOINT2, is a large joint. The goal is to change the angle from its initial $0°$ to $-45°$.

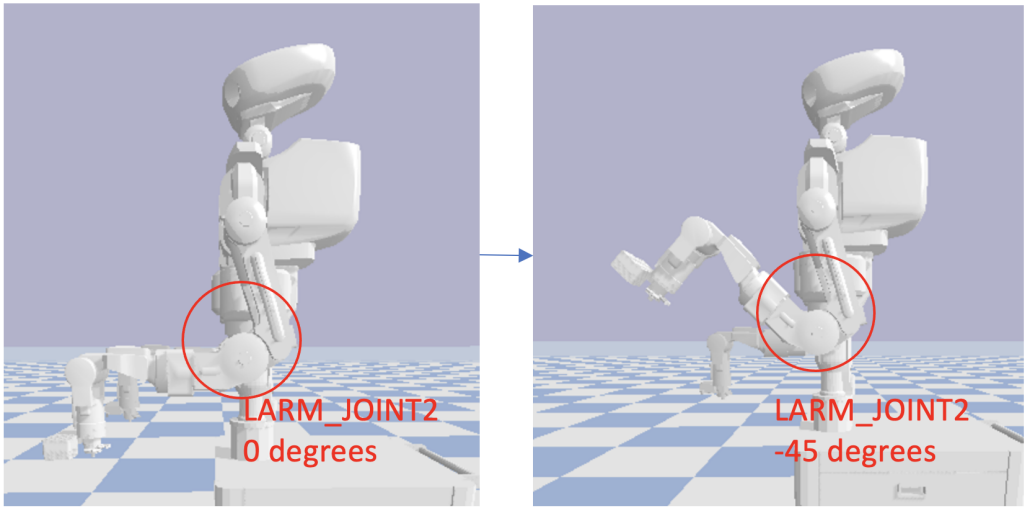

Figure 7: Task 2 Objective Diagram

The tuning output responses can be seen below in the figures below with a final $K_p = 450$, $K_d = 10$. The video of this process can be viewed here.

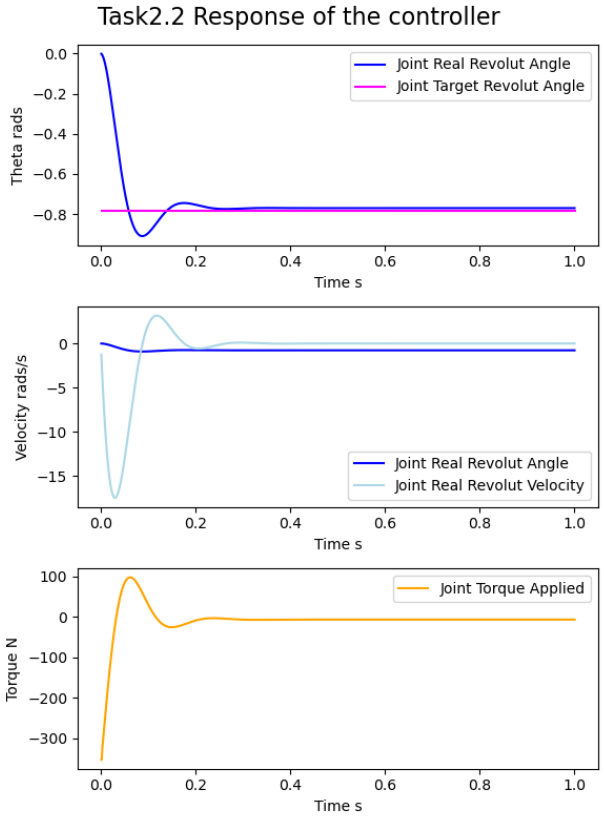

Figure 8: LARM_JOINT_5 Target Tuning

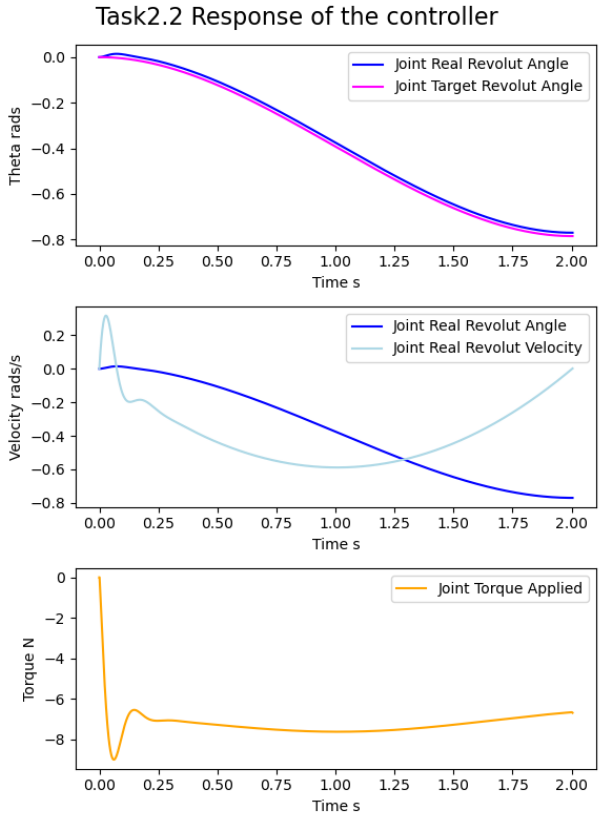

Figure 9: LARM_JOINT_5 Trajectory Tuning

In Figure 8, the single target of $-45°$ was given. In the top graph, we can observe a slight overshoot, and the joint quickly passes its transient state into a steady state in 0.25 seconds. The shape of the response shows little oscillation. In the middle graph, we notice the velocity becomes stationary at about 0.2 seconds. A great torque is applied in the rising period ($\sim 0.1s$) to -300Nm. This response was deemed as good because all targets were reached within threshold, with little oscillation in a short time.

However, we also wanted to test if the gain settings would allow the joint to follow smaller targets, such as targets it would be given in a trajectory from interpolation. Hence we tuned the joints further by providing it with small targets interpolated using a cubic hermit spline function. The result can be seen in Figure 9. In the top graph of Figure 9, the joint angle has the biggest change in motion in the middle, creating an s-shaped curve. Through the trajectory, the joint angle tracks the target and reaching the target smoothly in a more natural motion, although slower compared to Figure 8, after 2 seconds. The joint velocity and torque applied also becomes smaller as the error term used to calculate the torque to be applied has become smaller.

A similar tuning process is carried out for the other joints of the robot. However, we do note that LARM_JOINT2 is a relatively large joint. For smaller joints, the gain values needed to be scaled down significantly to avoid oscillation.

4.3 Task 3

4.3.1 Pushing the Cube

The results for the cube pushing task is displayed below in Figure 10. The robot's left arm, more specifically LARM_JOINT5 was manipulated to a different set of way points that were manually tuned to push the cube forward. These sets of way-points were derived through countless amounts of testing to determine the right fit. We can see in the second frame, the left arm is first pulled back directly. Then it is rotated and placed behind the cube in frame 3 and 4. Finally, after aligning itself with the cube, the left arm tries to maintain a straight forward motion that eventually stops when the the cube reaches its target, depicted in frame 6.

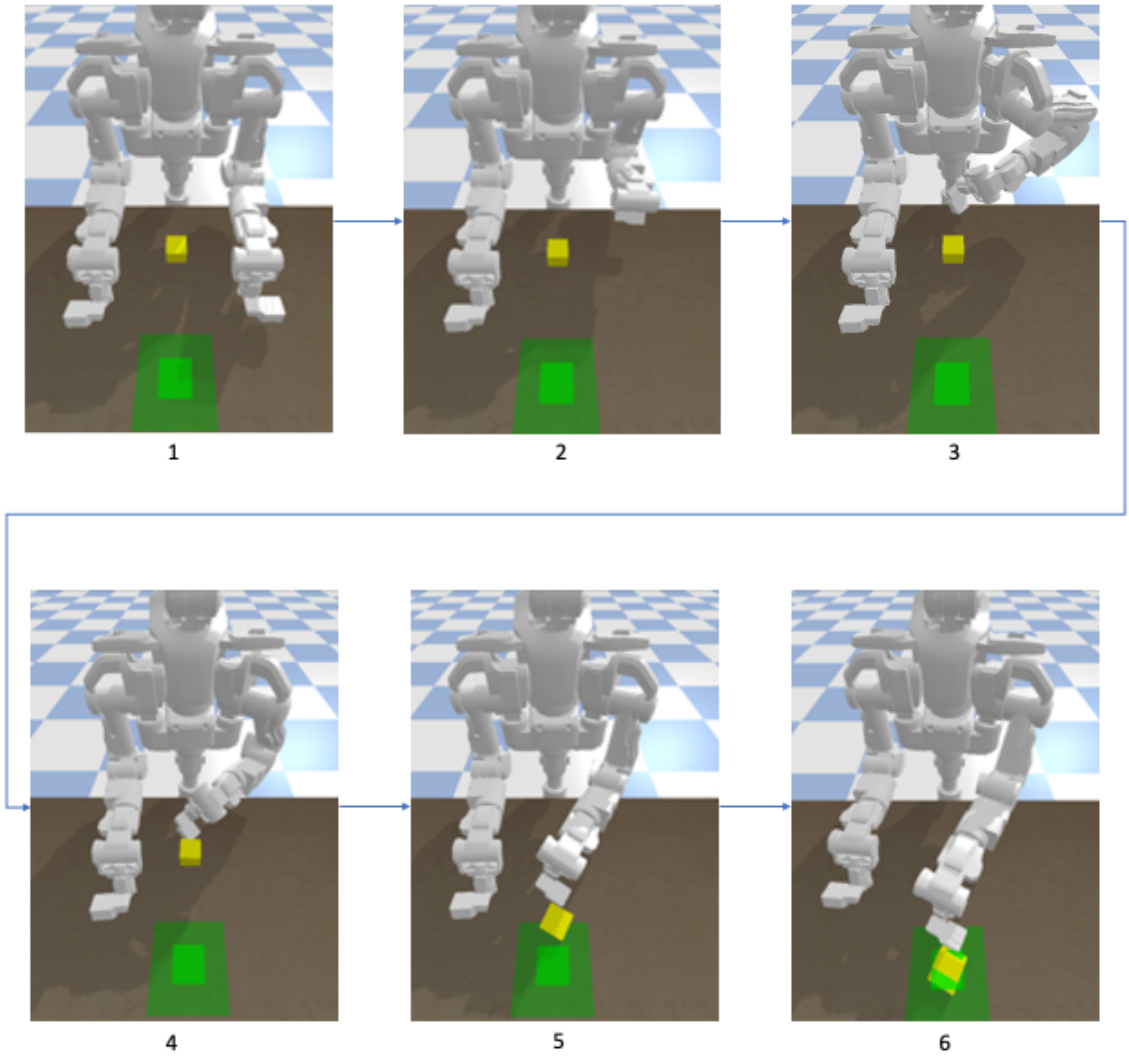

Figure 10: Snapshots of Cube Pushing Task Simulation Indicating the Motion

A video demonstrating task 3.1 can be found here. We use three simple way-points and end-effector orientations to get the job done. We start with drawing the end-effector back with a closed vertical orientation. Then we bring the end-effector behind the cube with a closed orientation slanted between the negative z and y axes. From there, we slowly orient the end-effector in parallel to the downward z axis and slowly push the cube towards the target. There are no excess or unnecessary motions involved. The cube's CoM reaches a final position of $[0.69589, 0.00127, 0.91904](5s.f.)$ compared to the target of $[0.7, 0.0, 0.91]$. The distance between is calculated to be 0.010011m or 10.011mm, which is well below the 150mm threshold to be met. The process was efficient and accurate, and we believe this is a great solution to the task.

4.3.2 Moving the Dumbbell

Finally, the last task involves docking a dumbbell and moving it above the obstacle and placed in a target location. This is achieved in a few way-points, which have the following sub-goals:

- Fix the orientation of the end-effectors most suitable for clamping and translating the cube to the target
- Move the end-effectors together to clamp the stem of the dumbbell
- Translate the dumbbell upwards till the base of the dumbbell is well clear of the obstacle in terms of the z-axis co-ordinate

- Translate the dumbbell is steady steps right to the x and y axes co-ordinates of the target, and translated upwards in terms of the z-axis coordinates
- steadily lower the dumbbell to dock into the target position

Snapshots of the motion can be seen in Figure 11, 12 and 13. It shows the dumbbell being clamped, lifted, docked and placed. The dumbbell's CoM at the end of simulation is located at TargetPositionDumbbell: $[0.326, 0.393, 1.018]$, while the target was: $[0.35, 0.38, 1.0]$. The difference in distance calculated to be 0.0327m or 32.7mm, which is again well below the threshold of 150mm. A video to this simulation can be found here.

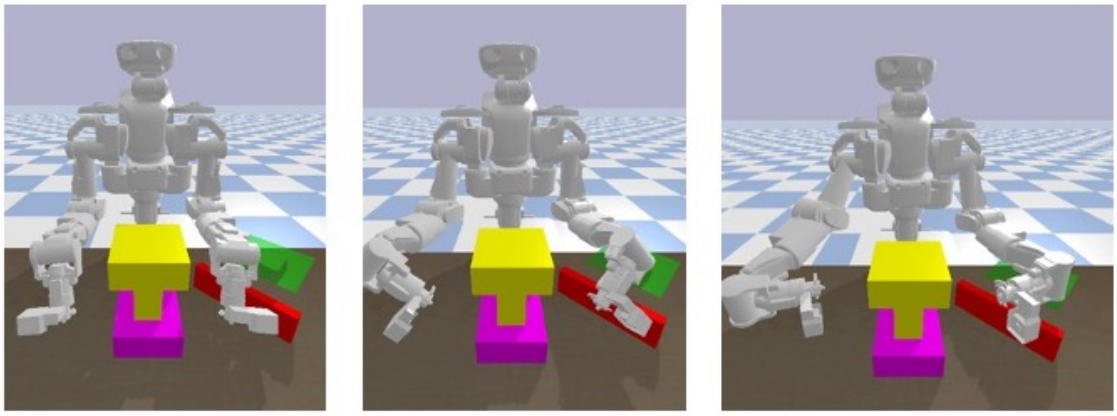

Figure 11: Robot Changing End-Effector Orientation to Clamp the Dumbbell.

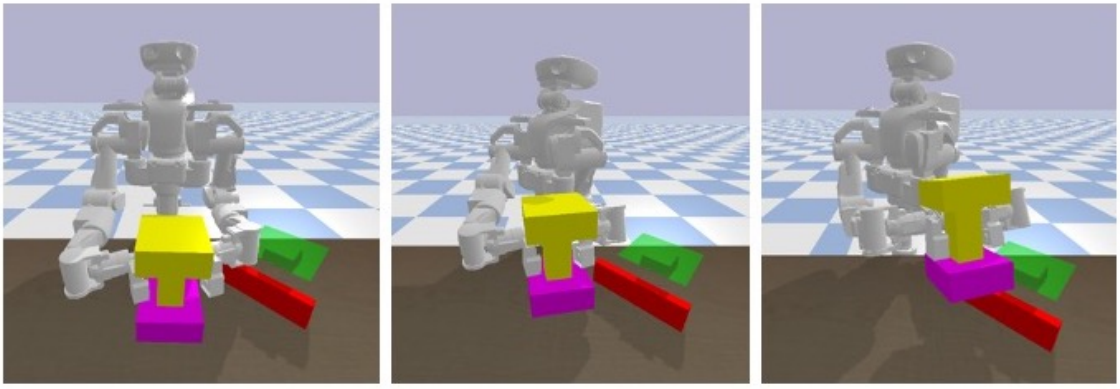

Figure 12: Robot Lifting Dumbbell Over the Obstacle.

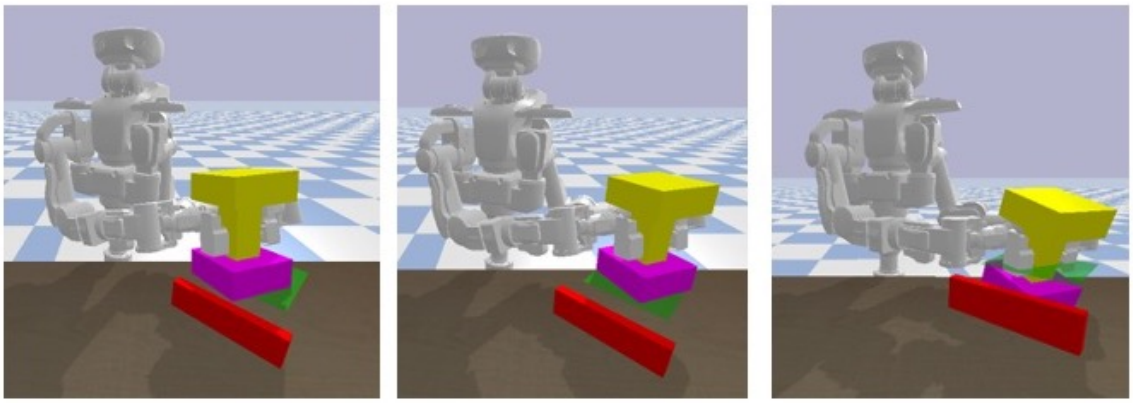

Figure 13: Robot Docking the Dumbbell to Target Position

For this task, we prefer stability over speed. It is easily possible to alter the number of iterations to less than half the value in the provided setup (code) and achieve almost similar results. But we prefer to have a steady, oscillation-less motion instead. After all, with the control frequency, it only translates to 4 seconds in real time, which we believe to be quite acceptable. Overall, we are happy that the object does reach the target well within threshold.

5 Discussion

In conclusion, we were able to successfully manipulate the Nextage humanoid robot to perform simple tasks such as pushing a cube, and moving an object to a target position up to a satisfactory precision. Both of which are high frequency basic tasks the robot is expected to perform as an assembly robot in a manufacturing factory. The most successful elements of our approach was the usage of the cubic hermit spline function, which gave smooth and accurate motions. However, the main limitations of the system was the amount of hard coding that was performed in task 3, where way points were manually calculated through an iterative process. If the target object was located in a slightly different position, it would make our code fail, hence it is not a robust solution. To improve this, it would be good to track the object's location relative to the robot through sensors such as cameras, IR or Lidars. For example, the camera sensor on the Nextage robot was not used in this case, but it would help bring the robot to the next level.

This project has helped us consolidate our knowledge about basic robot manipulation using forward and inverse kinematics, dynamics and controllers, trajectory planning and optimisation. We now move forward feeling more confident about working with humanoid robots in the future.