Kinematics and Dynamics of the Nextage Robot

Lars Thalian Morstad and Wassim Jabrane

November 2022

1 Abstract

Kinematics, control and object manipulation are some of the most central tasks in robotics. In this report, we demonstrate these tasks on a humanoid robot. Firstly, we have developed forward kinematics that allows the calculation of the location of the robot's joints from its joint angles. Building on this, we have created an inverse kinematics solver that gives the joint angles needed for a certain joint to reach a certain position. With this, we developed a closed-loop Proportional-Derivative (PD) controller which considers the dynamics of the robot and gives the torques needed for the robot to reach the locations it is given. Finally, these approaches were tested, first by moving singular joints and later in a pushing trial and a lifting trial. The robot was able to quickly move its joints to a goal position with high accuracy. With a set of given points to move, the robot was able to lift and push objects to a goal location with high precision.

2 Introduction

Our work concerns making the robot's joints move correctly, with the end goal of pushing and lifting objects with high precision. This was decomposed into three tasks that build on each other: Kinematics, Dynamics, and Manipulation. For simplicity, we constrained the tasks to only include moving and controlling the robot. We do not concern ourselves with sensors, and object positions are assumed to be known.
Task 1 involved implementing kinematics, with the goal moving the joints so that the end effector reaches a desired target position without considering forces/dynamics . This was done in two parts:

- Task 1.1 - Forward Kinematics (FK). This refers to the process of calculating the position and orientation of any joint, given the current joint angles of the robot. The joint of interest is usually the end-effector. FK translates joint-space coordinates to world-space coordinates.
- Task 1.2 - Inverse Kinematics (IK). IK is the complimentary to FK, in the sense that it is the process of calculating joint angles that would move joints so the end-effector is in a target position, with a target orientation. IK takes world space coordinates and attempts to translate them into joint space.

In **Task 2**, we implemented a Proportional-Derivative (PD) controller, to tackle introducing dynamics into the process of moving an end-effector to a target position and orientation, finishing with a target velocity. The PD controller is closed-loop, and uses a combination of joint angles produced by IK, points produced by FK, and *gains* - a collection of joint-specific constants. With these, the controller produces torques to move the joints.
Finally, in **Task 3** we combine the previous two tasks to achieve two types of object manipulation.

- Task 3.1 - Pushing cube - using its arms to push an object to a specific location.
- Task 3.2 - Docking dumbbell - picking up and then dropping an object to a certain location while avoiding an obstacle.

In section 3, we describe the robot, its specifications, and the methodology for each task. Section 4 presents the results of the tasks with plots and snapshots of the simulations. At the end of the report in section 5, we summarise, evaluate our results, and discuss the advantages and limitations of our approaches. Finally, we present further ideas on how we could improve the system.

3 Methods

3.1 The Nextage Robot

The tasks above were implemented on the humanoid Nextage Robot, produced by Kawanda Robotics. The robot has the 17 links and 17 joints, and is a torso placed on a base, meaning

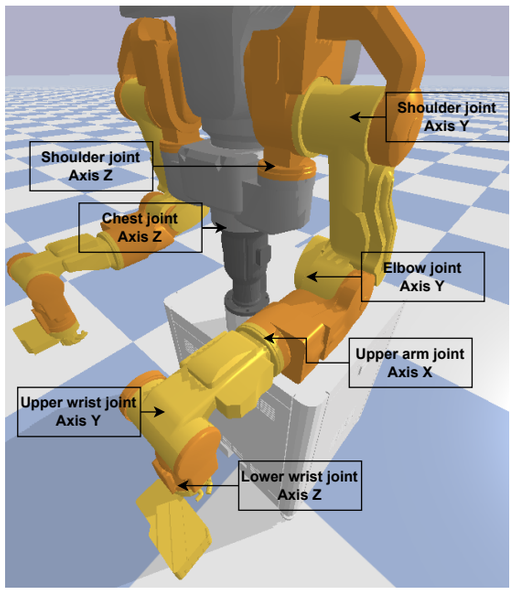


Figure 2: Relevant joints for the tasks, with rotation axes.

it has no legs. Each arm has six joints: two for each of the shoulder, arm and wrist. In addition, there is one in the base and two in the head. All joints are revolute. Its sensing capabilities include two cameras in the head, and one in each end-effector. For testing purposes, development and trials were conducted in a PyBullet[1] simulation in Python 3.

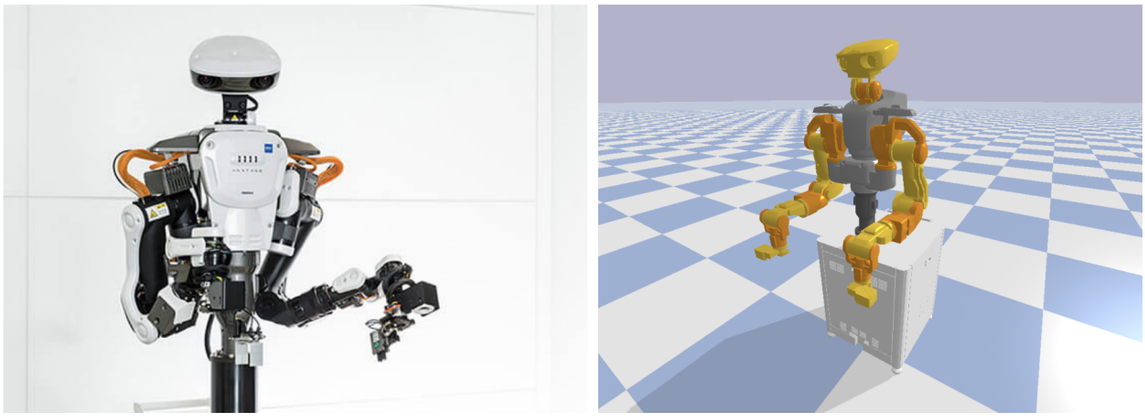


Figure 1: Picture 1 (Left): The Nextage Robot (Image credit: Kawada) Picture 2 (Right): The 3D-model simulation of Nextage Robot

Degrees of freedom	15 axes
Payload (per arm)	1.5 kg
Weight (with platform)	80 kg
Height (with platform)	174 cm
Position Repeatability	Within 0.5 sec

Table 1: Nextage robot specifications

3.2 Task 1

3.2.1 Forward Kinematics

FK is computed by multiplying a chain of transformation matrices known as the *kinematic chain*. In our implementation, the chain goes from a base joint, such as the chest, to an end-effector. To determine the kinematic chain we constructed a map of the robot's topology. The map described what the next joint in the chain would be based on what the given end-effector is. This was needed because the chest joint is connected to both arms and the head, so its next joint is not obvious. Each matrix in the chain corresponds to a coordinate frame that is attached to each of the links of the robot. A transformation matrix is defined as $\begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$, where R is a 3x3 rotation matrix in the axis-angle representation (Equation 1), and t is a 3x1 translation vector that represents the

[1] https://pybullet.org/wordpress/

displacement to the next joint, that is, the link that is attached to the current one. The general axis-angle rotation matrix is shown below, with $[n_0, n_1, n_2]$ being the rotation axis as a unit vector and θ being the rotation angle. In practice the matrix becomes much simpler since each joint rotates around only one axis. See [1] for a detailed description of kinematics algorithms.

$$\begin{bmatrix} n_0^2 \cdot (1 - \cos(\theta)) + \cos(\theta) & n_0 n_1 \cdot (1 - \cos(\theta)) - n_2 \sin(\theta) & n_0 n_2 \cdot (1 - \cos(\theta)) + n_1 \sin(\theta) \\ n_0 n_1 \cdot (1 - \cos(\theta)) + n_2 \sin(\theta) & n_1^2 \cdot (1 - \cos(\theta)) + \cos(\theta) & -n_0 \sin(\theta) + n_1 n_2 \cdot (1 - \cos(\theta)) \\ n_0 n_2 \cdot (1 - \cos(\theta)) - n_1 \sin(\theta) & n_0 \sin(\theta) + n_1 n_2 \cdot (1 - \cos(\theta)) & n_2^2 \cdot (1 - \cos(\theta)) + \cos(\theta) \end{bmatrix} \quad (1)$$

Using the rotation matrix and the translation vectors for each joint, we could compute the forward kinematics of the end-effector. Below is an example kinematic chain, from the base to left end-effector.

$$\begin{aligned} T_{base\_to\_waist->LARM\_5} = {} & T_{base\_to\_waist->CHEST\_0\_LEFT} @ T_{CHEST\_0\_LEFT->LARM\_0} \\ & @ T_{LARM\_0->LARM\_1} \\ & @ T_{LARM\_1->LARM\_2} \\ & @ T_{LARM\_2->LARM\_3} \\ & @ T_{LARM\_3->LARM\_4} \\ & @ T_{LARM\_4->LARM\_5} \end{aligned} \quad (2)$$

When defining our coordinates for all points in the world frame, we always offset by a certain amount. This is because we represent the start of our kinematic chain from the waist fixed joint, which is a translation of 0.85 m in the z-axis from the origin. To represent the coordinates in the world frame accordingly, this means subtracting that offset from the desired goal positions used in IK (Section 3.2.2).

3.2.2 Inverse Kinematics

The IK solver translates a target position and orientation in the world space. A simplified verison of the solver is presented in Algorithm 1. Creating an IK solver requires the computation of the jacobian matrix, J. The dimensions of J depend on the number of joints in the kinematic chain, n, and amount of tasks to complete i.e) position and orientation. In our solver that incorporates orientation, the jacobian is composed of two horizontally stacked nx3 matrices. The left stack is defined as the Position Jacobian J_{pos} which represents the position increment, and the other as Vector Jacobian J_{vec} which represents the orientation increment. Hence, the total dimension of the jacobian matrix would be nx6. The Moore-Penrose pseudoinverse of J is used to map a change in world position to a change in joint angles.

Algorithm 1 inverseKinematics(initQ, targetPos, targetOrientation, nSteps)

```
efPos, efRot ← FK(initQ)
curQ ← initQ
subgoals ← linspace(efPos, targetPos, nSteps)        ▷ nsteps equally spaced points
for each subgoal in subgoals do
    dPos = subgoal - efPos
    dOrientation = targetOrientation - efRot@a_eff    ▷ a_eff, rotation axis of the end effector
    δy = hstack(dPos, dOprientation)
    δq = J^+ δy                                       ▷ Map change in position to change in joint angles
    curQ += δq
    efPos, efRot ← FK(curQ)
end for
return curQ
```

This task could have been completed without including orientation. However, we chose to incorporate this, since it increased the finesse of the robot and was relatively simple to implement.

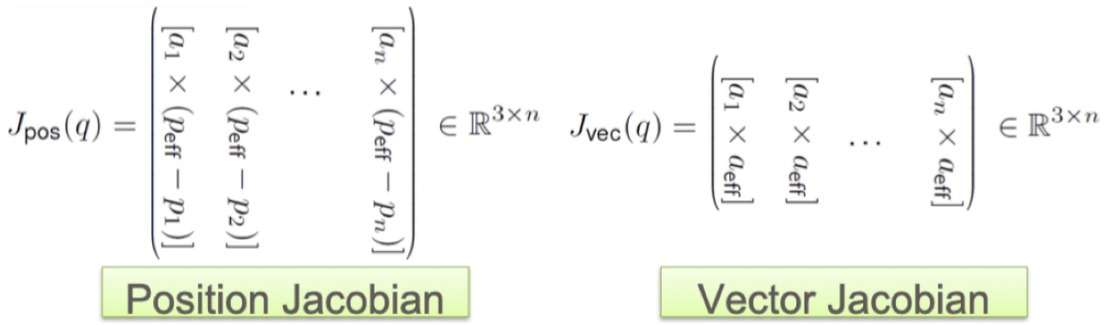


Figure 3: Jacobian matrix, which would be composed of the two matricies J_{pos} and J_{vec}. p_i represents the translation vector and a_i represents the rotation axis of a joint i.

3.3 Task 2

3.3.1 PD Controller

For simplicity, we built a controller that considered the Proportional (Distance) and Derivative (Speed) of different joints to control the robot. The controller can be extended to add a Integral term, but this was not required for adequate precision.
The control loop for the robot can be seen in Figure 4 and takes in the end-effector to move, target location and target velocity as input. It then used IK to determine the joint angles needed to reach this target. These target angles (x_{ref}) are then fed into the PD controller equation (3) together with the gains (k_p, k_d) and the current joint angles of the robot, $x(t)$. The equation outputs joint torques. The joint velocity $\dot{x}(t)$ is estimated by dividing the difference between the current joint angles and angles from the previous iteration, by the simulation time step length.

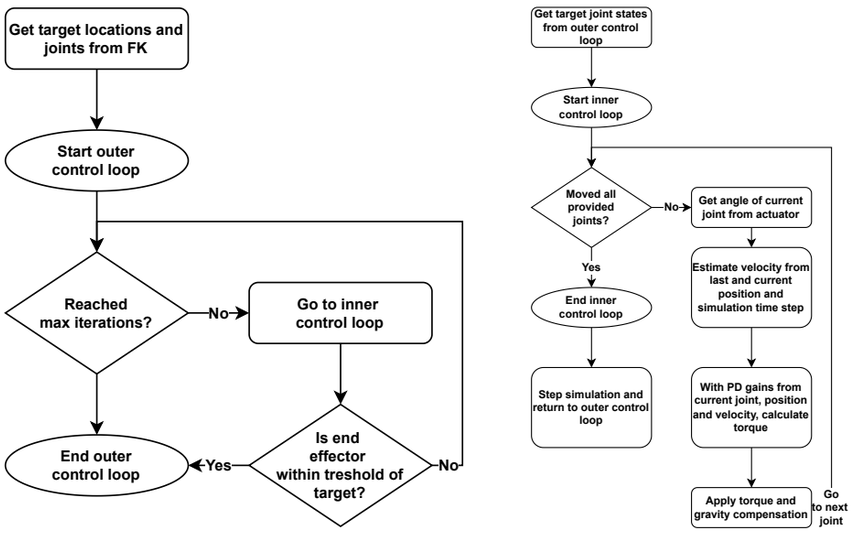
$$\mathbf{u}(t) = k_p(x_{ref} - x(t)) - k_d\dot{x}(t) \tag{3}$$

The controller has two modes, moving one or two arm end-effectors. When moving two, we chose to simply calculate the IK of the two arms separately and then feed all the angles into the controller. They are independent except for movement in the chest joint. The issue that arises from this is that the two IK outputs would contain a common chest joint with potentially differing joint angles. This is because the individual IK calls do not account for each other moving the chest. To resolve potentially conflicting instructions to the chest, we took the average of the two values. This was found to give satisfactory results. We also considered ignoring the chest and starting the kinematic chain at the shoulders to remove any conflict. However, this was discarded as it reduced the robot's range of motion too much. Another option was using the computed chest joints for one arm instead of both. However, this provided a biased outcome towards one of the end effectors and sometimes unstable behaviour when rotating the chest.

The bulk of the time spent on this task was on tuning the individual gains for the robot joints. For this we simply iterated manually until we found values that produces satisfactory results. Starting with the chest and working our way through the arms, each joint was commanded to move to various positions and velocities. Then, the gains were tuned according to error. This can be done automatically but it was deemed as not needed for the task.

Joint	**P Gain**	**D Gain**
Chest	30	15
Shoulder Z	30	15
Shoulder Y	350	40
Elbow Y	215	25
Upper Arm X	80	5
Upper Wrist Y	60	2.5
Lower Wrist Z	1	1

Table 2: Proportional and derivative gains for controller, with head excluded. These should not be considered fully optimal but did fulfill our objectives.



(a) Outer control loop (b) Inner control loop

Figure 4: Flow diagrams for the control of the robot's joints. The outer loop calls the inner loop until the end-effector is within a set threshold distance of the goal, or until too many iterations have passed. The inner loop passes through all the joints in the kinematic chain and uses the PD controller to calculate their torques.

3.4 Task 3

3.4.1 Pushing cube - Task 3.1

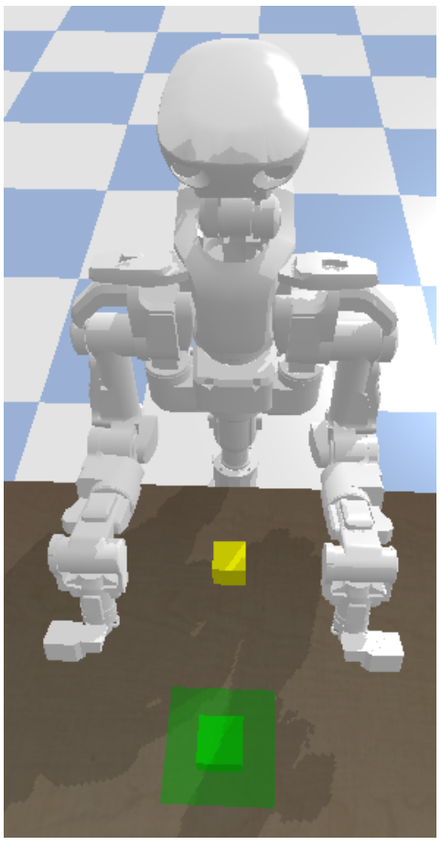
The goal of this task was to move a 7cm, 0.1 kg cube to within 150mm of a specified goal on a flat surface (see Fig. 5). For this simple task, we opted to use a set of five predetermined points for the robot to move to, with one arm. The points cause the left arm to move behind the cube and push it forward, while slightly rotating the end-effector. For this task, we also developed a cubic spline interpolation routine to aid in smooth motion. The pushing was done without interpolation, simply because it was not needed. We explored using two arms to pick and lift the cube. However, it was found that using only one arm was simpler and yielded better results.

3.4.2 Docking dumbbell - Task 3.2

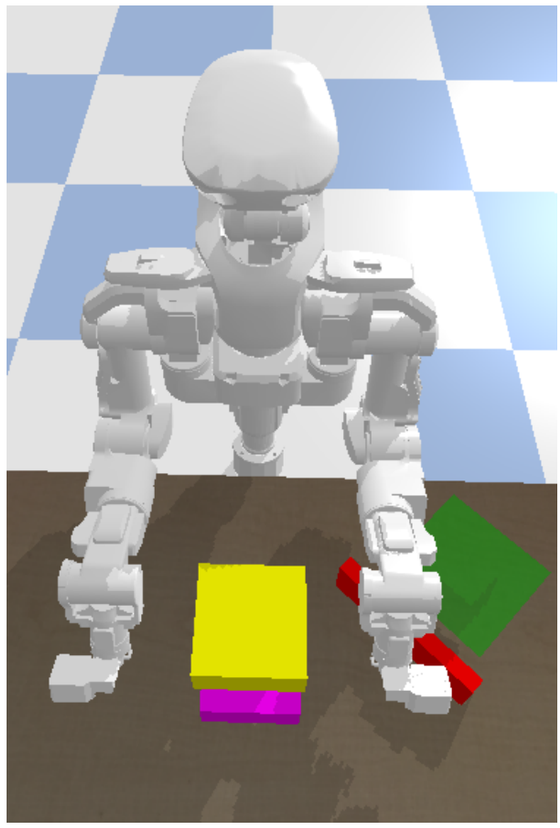
The goal for this task was to lift a 28cm tall, 0.3 kg dumbbell standing upright, over a 12 cm tall rectangular obstacle without colliding. This task motivated the need to move both arms at the same time, as described in section 3.3.1. Again, the goal was to place the object within 150mm of the target (see Fig. 5). This involved three stages:

- Phase 1: Orient the arms to grab the dumbbell.
- Phase 2: Move the arms along with the dumbbell over the red obstacle and lower it towards the target indicated by the green rectangle.
- Phase 3: Move the arms away from the dumbbell to prepare the robot for another task.

For this task, we used a combination of pre-specified points to grasp and lift the dumbbell and interpolated intermediate points to make sure it was put down in a slow and controlled fashion. Additionally, changing the orientation of the end-effectors throughout the lift ensured that the robot maintained a good grip.



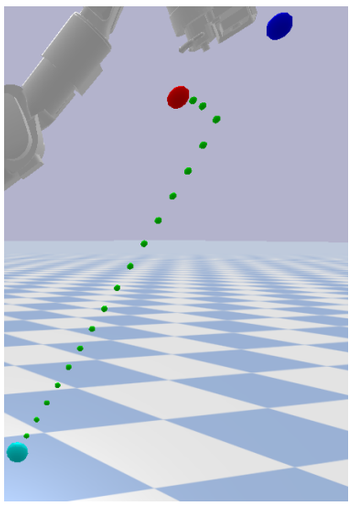
(a) Task 3.1 - Moving the cube without obstacles



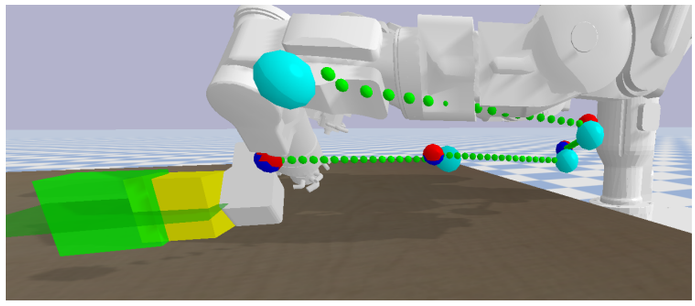
(b) Task 3.2 - Moving the cube with an obstacle

Figure 5: Starting positions for the robot for two tasks: (a) to move the cube (yellow) to from its starting position to within a threshold of 150mm of the target (light green cube) and (b) to lift the cube over an obstacle (red) with a similar goal.

3.5 Visualisation - Debugging



(a)



(b)

Figure 6: Visualising the motion of the end effector using *pybullet*'s tools for debugging purposes.

For debugging, we use *pybullet*'s rendering functions to visualise the results of IK and FK, to see if they worked as intended (see Figure 6). We render up to 4 different spheres in different colours:

- Dark blue: target position
- Cyan: end-effector start position
- Red: end-effector final position. It should end up as close as possible to the dark blue sphere.
- Green: IK subgoals

While not needed for the actual tasks, this visualisation allowed us to quickly iterate on our code since it was far easier to see where something went wrong.

4 Results

4.1 Task 1

The goal of this task was to assess the correctness of the forward and inverse kinematics computations, without considering the dynamics of the robot. This is because correct FK and IK are prerequisites for correct dynamic control. Therefore we used direct joint position commands in PyBullet (the *resetJointStates* function), instead of torque commands. We found that the inverse kinematics output rapidly converges to the target position, as exemplified in Figure 7. Still, when reaching for points outside the robot's workspace, the solver was at times unstable.

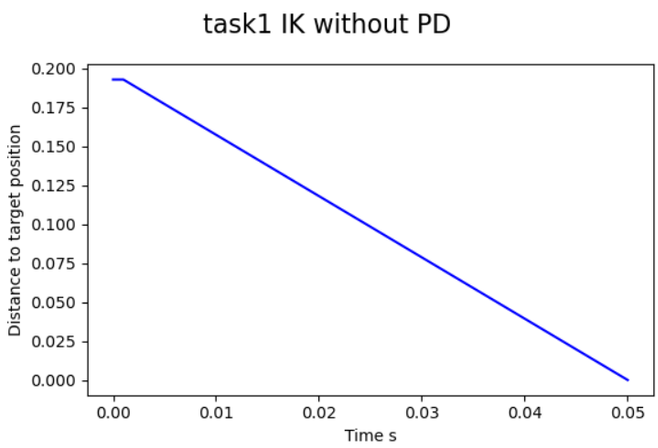
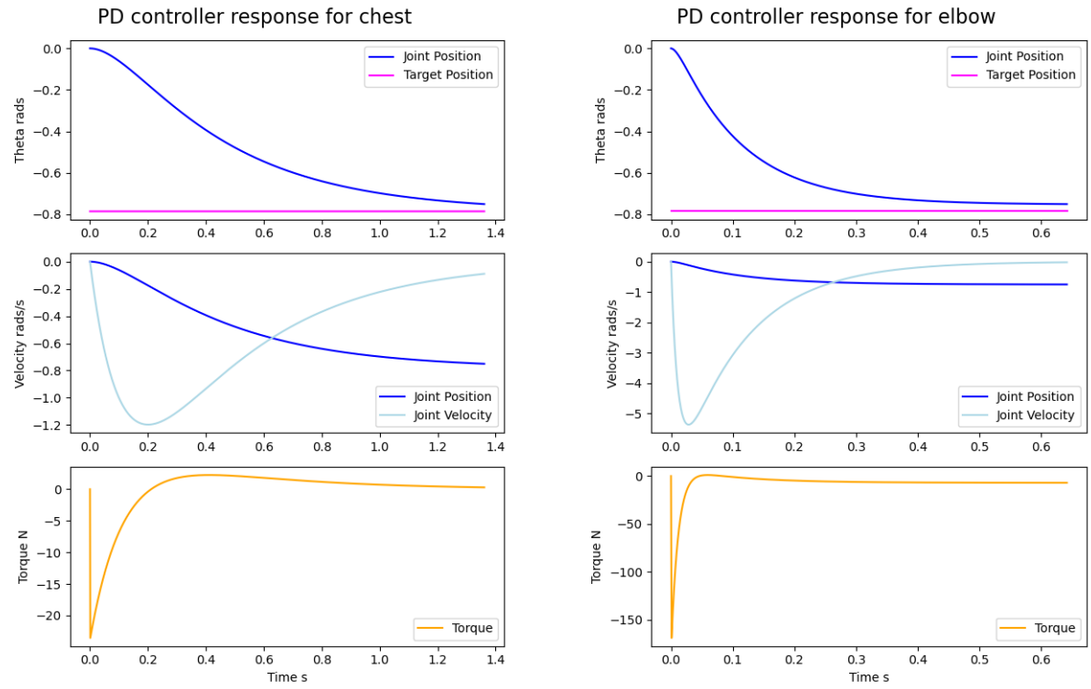


Figure 7: Time taken for the lower left wrist end effector to reach a target position, when the distance becomes (close to) 0. This does not use PD control, and simply sets the joints in the simulation to the planner generated by Inverse Kinematics

4.2 Task 2

Since the goal of this task was to develop a functioning PD controller with appropriate gains, we will examine the response of said controller for various joints. For these tests, we set out goal position to be $45°(\pi/4$ radians), and the goal velocity to be 0. The controller used and produced the gains presented in Table 2. The task was deemed a success if the measured joint angle was within 0.035 radians (2.0054°) of the target. Figure 8 shows the response for the chest and elbow joints. As can be seen from the figure, the robot quickly converges to the goal position quickly and has little overshoot. Although the chest has a slight velocity on the graph, it quickly went to zero like the elbow did. The chest has a slower response than the elbow, so its gains may benefit from being increased. The torques are characterised by sharp troughs to begin moving, followed by a rapid reduction and a small corrective torque in the other direction before returning to zero. All in all, the controller responses were found to be quite stable and did not oscillate much. The largest gains are for the shoulder and elbow Y-axis joints, which have to lift and hold up the rest of the arm. The gains for both arms are identical.



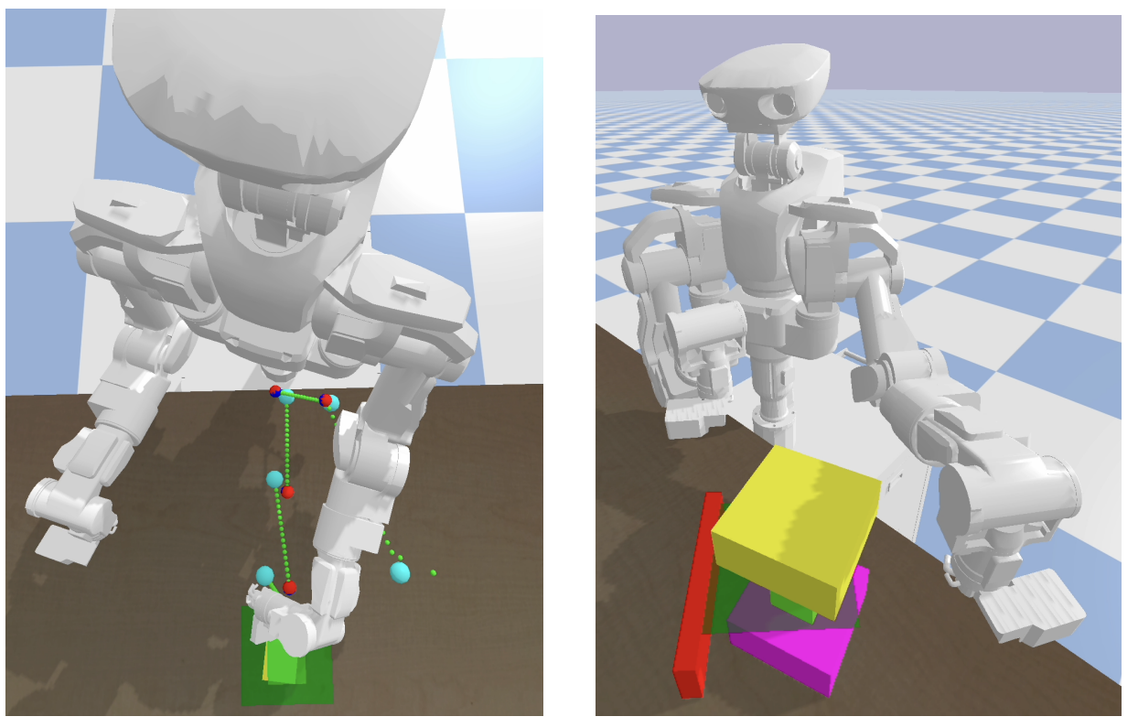
(a) PD response from chest joint

(b) PD response from elbow joint

Figure 8: Controller responses for the chest and elbow joints of the robot, where the goal rotation was 45°and the goal velocity was 0. The top graph depicts joint position in radians, the middle shows joint velocity and positions and the bottom graph shows joint torques. Note how the controller responds to high distances and velocities and applies corrective measures.

4.3 Task 3

Finally, we assess the whole system through two specific object manipulation tasks: pushing a cube and docking a dumbbell. We evaluate the performance of the tasks by reporting the final distance from the object's center of mass (CoM) to their targets. The success criterion for both is reaching a distance of less than 150 mm. Figure 9 shows the simulation upon completion of the tasks, and section 4.3.3 contains a link to the video demonstrations for both tasks.



(a) Task 3.1 - Cube successfully reaching target area, with debugging visualisation turned on.

(b) Task 3.2 - Dumbbell successfully placed on the target area.

Figure 9: Snapshots of the final placements of the objects in their respective targets

4.3.1 Task3.1

Overall, the robot manages to orient and position its arms and successfully accomplishes the task. It gradually pushes the cube to each of the specified coordinates in order, coming to a stop at each of them, and then repeating until the end point. At the end of the execution, the final distance between the cube's CoM and and the target position is below 150 mm, around $1.3 * 10^{-2}$ m.
We found that the robot joint does not always reach the specified goal coordinate if the algorithm reaches maximum iterations. This can be alleviated by increasing the maximum number of iterations of our algorithms or lower thresholds to allow for more precision at the cost of performance.

4.3.2 Task3.2

In this task, the robot manages to grasp the dumbbell in order to lift it above the obstacle. Then, together with cubic spline interpolations, it provides a soft landing for the robot to the target area. Finally, it successfully moves away from the placed dumbbell.
Similarly to in task 3.1, the final distance between the dumbbell's CoM and the target position is below 150 mm, around $4.3 * 10^{-2}$ m. When evaluating this task, we found that the robot was quite sensitive to the provided points. A small change could sometimes make the task fail or make the robot unstable. This occurred mainly if the robot exerted too much force on the environment.

4.3.3 Demo

The video demonstrations of the robot performing tasks 3.1 and 3.2 can be found in the OneDrive folder link here.

5 Discussion

In this report, we have described the kinematics and dynamics of the Nextage robot in a simulation. We have shown that our inverse kinematics solver produces precise joint angles, and that the PD controller gives appropriate torque values, both with a quick response time. Additionally, we have demonstrated two instances of successful object manipulation: pushing a cube and docking a dumbbell.

One of the main limitations is that these methods required the use of predetermined coordinates. Hence, the solution to the object manipulation tasks do not generalize to other tasks. To improve this, we could incorporate motion planning to avoid obstacles and compute goal points dynamically. We could also use the robot's cameras to detect obstacles and objects to manipulate. Not using sensors has simplified the system significantly to a point where it may not be realistic. In reality, we would need to use Kalman filters or similar techniques to estimate an object's location [2]. Another improvement is to deploy techniques that estimate the objects' poses to move objects of different shapes [3]. The use of force sensor on the robot could provide additional feedback to the controller to improve the stability of the system when in contact with objects.

References

[1] Serdar Kucuk and Zafer Bingul. *Robot kinematics: Forward and inverse kinematics*. INTECH Open Access Publisher London, UK, 2006.

[2] Qiang Li et al. "Kalman Filter and Its Application". In: *2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*. 2015, pp. 74–77. DOI: 10.1109/ICINIS.2015.35.

[3] Lucas Manuelli et al. "kPAM: KeyPoint Affordances for Category-Level Robotic Manipulation". In: *CoRR* abs/1903.06684 (2019). arXiv: 1903.06684. URL: http://arxiv.org/abs/1903.06684.

Appendix

Joint	Start	End
LARM_JOINT5	[0.505 0.230 0.182]	[0.583 -0.041 0.112]

Table 3: The starting and final end effector's coordinates at the end of the simulation for task3.1, rounded to three decimal places. LARM_JOINT5 signifies the left wrist end-effector.

Joint	Start	End
LARM_JOINT5	[0.505, 0.230, 0.182]	[0.261, 0.626, 0.292]
RARM_JOINT5	[0.505, -0.230, 0.182]	[0.262, 0.128, 0.303]

Table 4: The starting and final end effectors' coordinates at the end of the simulation for task3.2, rounded to 3 decimal points. LARM_JOINT5 signifies the left wrist end-effector, while RARM_JOINT5 signifies the right wrist end-effector.