

## Robotics 2 Coursework 3 Interaction

### Task A: End-effector orientation calculation

#### Task A Part i

Euler angles, represented as roll, pitch, and yaw, describe the orientation of an end effector relative to the base frame. These angles correspond to rotations around the X, Y, and Z axes, respectively. According to Euler's rotation theorem: "Any rotation can be described by three successive rotations around linearly independent axes."

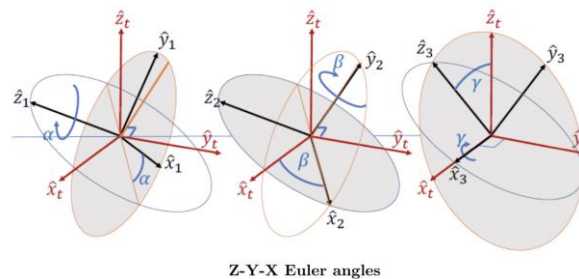


Figure 1: Euler angles

Following the ROS convention, the z-y-x sequence of Euler angles is applied, meaning the first rotation is about the Z-axis (yaw), followed by the Y-axis (pitch), and finally the X-axis (roll). This sequence is essential for defining the orientation of De Niro's arms and ensuring accurate motion execution in the Gazebo simulation environment.

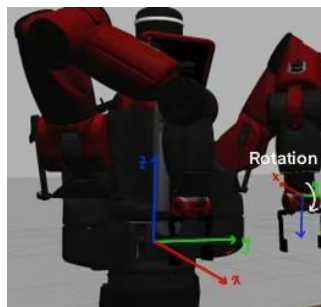


Figure 2: Base frame and End-effector frame

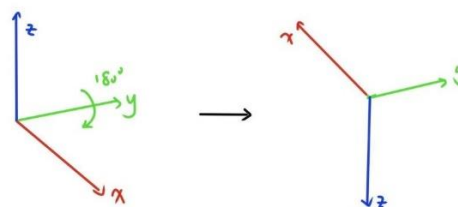


Figure 3: Illustration of the end-effector frame rotation

By comparing the coordinates of the robot's hands with the coordinates of its base frame, the initial orientation can be achieved through a rotation of  $\pi$  radians about the Y-axis. The direction of rotation follows the right-hand rule. This results in roll = 0, pitch =  $\pi$ , and yaw = 0, as shown in Figure 3.

#### Task A Part ii

To directly transfer the brick from one hand to the other without requiring an intermediate placement, the end effectors must adopt alternate orientations. The key requirement for this motion is that both Z-axes of left and right grippers must be aligned and facing the brick.

After achieving the initial orientations defined in the previous step, the left arm must rotate  $\pi/2$  radians about the X-axis to position the brick horizontally toward the right hand. This results in updated Euler angles of roll =  $\pi/2$ , pitch =  $\pi$ , and yaw = 0.

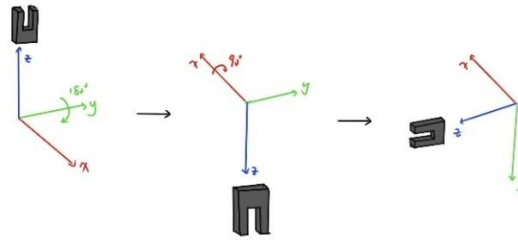


Figure 4: Illustration of the left-hand frame rotation for horizontal alignment

To receive the brick, the right hand must rotate by the same amount but in the opposite direction, meaning a  $-\pi/2$  radians rotation about the X-axis. This leads to final Euler angles for the right hand of roll =  $-\pi/2$ , pitch =  $\pi$ , and yaw = 0.

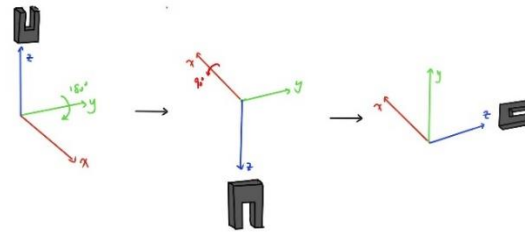


Figure 5: Illustration of the right-hand frame rotation for horizontal alignment

## Task B: Position control to perform pick-and-place tasks

### Task B Part i

In the `position_controller.py` file, the initial rotation of the grippers were implemented in Task B as calculated in Task A. The desired orientations are updated using the roll, pitch, and yaw (RPY) value, and the desired positions are updated using the x, y, z coordinates (XYZ). These values were integrated into the initialization function, which updates the poses of both arms by calling `[arm_name].servo_to_pose(xyz, rpy)` where xyz and rpy correspond to the respective positions and orientations of each arm.

The left and right arm positions (XYZ) were set to `[0.5, 0.5, 0.25]` and `[0.5, -0.5, 0.25]`, respectively, while both arms were assigned an orientation of `[0.0,  $\pi$ , 0.0]` in the RPY format.

```
# Go to the starting Pose for left arm
left_xyz = [0.5, 0.5, 0.25] # Linear
left_rpy = [0.0, np.pi, 0.0] # Angular: The roll and yaw values are 0, and the roll value is pi
left_arm.servo_to_pose(left_xyz, left_rpy)

# Go to the starting Pose for right arm
right_xyz = [0.5, -0.5, 0.25] # Linear: Symmetrical to the left arm about y-axis
right_rpy = [0.0, np.pi, 0.0] # Angular: The roll and yaw values are 0, and the roll value is pi
right_arm.servo_to_pose(right_xyz, right_rpy)
```

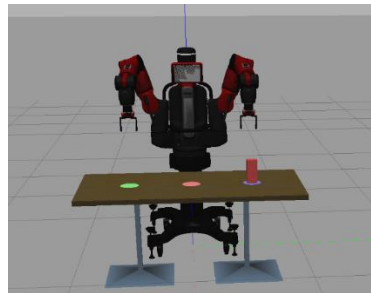


Figure 6: Initial positions of the left and right arms of DE NIRO

## Task B Part ii

In this task, DE NIRO moves the brick from the initial position (XYZ) [0.75, 0.5, -0.05], and the end position (XYZ) [0.75, -0.5, -0.05]. To do this, DE NIRO picks up the brick with its left hand from the blue circle at right, places it on the red circle in the middle, then picked up by the right hand, which finally places it on the green circle. The orientations of the end-effectors remain unchanged during the operation because no hand-over was required, therefore all rpy values are replaced by left\_rpy and right\_rpy which equal to [0.0,  $\pi$ , 0.0].

To update the positions of the end-effectors, several waypoints were defined along the trajectory. These waypoints were chosen to ensure smooth motion between the brick, the table, and target locations while still ensuring a successful transfer.

The robot's grippers were controlled using the .gripper\_close() and .gripper\_open() functions to grasp and release the brick as needed. After releasing the brick, the end-effector grip move up vertically by 0.1 before going to the next orientation. This is because if without the vertical movement before transferring to initial position, the brick will be on the route from position to initial position which results in the brick being hit during the transfer.

```
# Position 1: Left arm grabs the brick from the blue dot
left_arm.servo_to_pose([0.75, 0.5, -0.05], left_rpy)
left_arm.gripper_close() # Close the gripper to grab the brick

# Position 2: Left arm moves the brick to the red dot
left_arm.servo_to_pose([0.75, 0.5, 0.05], left_rpy)
left_arm.servo_to_pose([0.75, 0.0, 0.05], left_rpy)
left_arm.servo_to_pose([0.75, 0.0, -0.05], left_rpy)
left_arm.gripper_open() # Open the gripper to release the brick

# Position 3: Left arm goes back to initial position
left_arm.servo_to_pose([0.75, 0.0, 0.1], left_rpy)
left_arm.servo_to_pose([0.5, 0.5, 0.25], left_rpy)




# Position 4: Right arm grabs the brick from the red dot
right_arm.servo_to_pose([0.75, 0.0, 0.1], right_rpy)
right_arm.servo_to_pose([0.75, 0.0, -0.05], right_rpy)
right_arm.gripper_close() # Close the gripper to grab the brick




# Position 5: Right arm moves the brick to the green dot
right_arm.servo_to_pose([0.75, 0.0, 0.1], right_rpy)
right_arm.servo_to_pose([0.75, -0.5, 0.1], right_rpy)
right_arm.servo_to_pose([0.75, -0.5, -0.05], right_rpy)
right_arm.gripper_open() # Open the gripper to release the brick

# Position 6: Right arm goes back to the initial position
right_arm.servo_to_pose([0.75, -0.5, 0.1], right_rpy)
right_arm.servo_to_pose([0.5, -0.5, 0.25], right_rpy)

return
```

Table 1: Operation descriptions of Task B ii

Step	Arm	XYZ	Description	Image
1	Left	[0.75, 0.5, -0.05]	Left arm moves to initial position of the brick, gripper closes and grabs the brick.	
2	left	[0.75, 0.5, 0.05] [0.75, 0.0, 0.05] [0.75, 0.0, -0.05]	Left gripper moves vertically upward by 0.1, transfer horizontally to middle point, and move downward to open the gripper and let the brick release on mid point.	
3	left	[0.75, 0.0, 0.1] [0.5, 0.5, 0.25]	The left gripper opens, move upward by 0.1 to avoid hitting the brick, then move directly to initial position.	

4	right	[0.75, 0.0, 0.1] [0.75, 0.0, -0.05]	Right hand moves to above the mid-point to avoid knocking the brick over. Right hand then move downward and gripper close to grab the brick.	
5	right	[0.75, 0.0, 0.1] [0.75, -0.5, 0.1] [0.75, -0.5, -0.05]	Right hand moves the brick upward, transfer horizontally to above final position, move vertically downward, then release the brick on the final position by opening the brick.	
6	right	[0.75, -0.5, 0.1] [0.5, -0.5, 0.25]	The right gripper opens, move upward by 0.1 to avoid hitting the brick, then move directly to initial position.	

### Task B Part iii

In this task, the brick was handed over directly from the left arm to the right arm. Using the previously defined waypoints, the left end effector first picked up the brick. It was then rotated  $\pi/2$  radians about the X-axis, as calculated in Part A ii. Similarly, the right end effector rotated  $-\pi/2$  radians about the X-axis to align horizontally with the left hand and complete the transfer.

The problem observed is when transferring the brick, the left and right hand should not be in the same position, or the right hand will be pressing the brick firmly to the left hand, resulting in stiff and troublesome operations. A gap of 0.04 is finalised so that an adequate gap is reserved for the brick transfer.

```

# Position 1: Left arm grabs the brick from the blue dot
left_arm.servo_to_pose([0.75, 0.5, -0.05], left_rpy)
left_arm.gripper_close() # Close the gripper to grab the brick

# Position 2: Left arm moves to the middle of the table and changes its direction to horizontal
left_arm.servo_to_pose([0.75, 0.5, 0.2], left_rpy)
left_arm.servo_to_pose([0.75, 0.02, 0.2], [(np.pi)/2], np.pi, 0.0])

# Position 3: Right arm moves to the middle of the table and changes its direction to horizontal. The brick is passed to the right arm
right_arm.servo_to_pose([0.75, -0.1, 0.2], [-(np.pi)/2], np.pi, 0.0])
right_arm.servo_to_pose([0.75, -0.02, 0.2], [-(np.pi)/2], np.pi, 0.0])
right_arm.gripper_close() # Right arm closes the gripper to grab the brick
left_arm.gripper_open() # Left arm opens the gripper to release the brick

# Position 4: Left arm moves to the initial position
left_arm.servo_to_pose([0.75, 0.2, 0.2], [(np.pi)/2], np.pi, 0.0])
left_arm.servo_to_pose([0.5, 0.5, 0.25], left_rpy)


# Position 5: Right arm moves the brick to the green dot
right_arm.servo_to_pose([0.75, -0.5, 0.1], right_rpy)
right_arm.servo_to_pose([0.75, -0.5, -0.02], right_rpy)
right_arm.gripper_open() # Open the gripper to release the brick






# Position 6: Right arm goes back to the initial position
right_arm.servo_to_pose([0.75, -0.5, 0.1], right_rpy)
right_arm.servo_to_pose([0.5, -0.5, 0.25], right_rpy)

return

```

Table 2: Operation descriptions of Task B iii

Step	Arm	XYZ and RPY	Description	Image
1	Left	[0.75, 0.5, -0.05], [0.0, $\pi$ , 0.0]	Moving left arm to initial position of the brick, gripper closes and grabs the brick.	

2	left	$[0.75, 0.5, 0.2], [0.0, \pi, 0.0]$ $[0.75, 0.02, 0.2], [\pi/2, \pi, 0.0]$	Left hand moves the brick upward, then rotate and , move to the middle position, awaiting transfer.		
3	right	$[0.75, -0.1, 0.2], [\pi/2, \pi, 0.0]$ $[0.75, -0.02, 0.2], [\pi/2, \pi, 0.0]$	Right hand rotate to horizontal and moves to middle position , then moves horizontally to close the gripper. The left hand then opens and the transfer is concluded.		
4	left	$[0.75, 0.2, 0.2], [\pi/2, \pi, 0.0]$ $[0.5, 0.5, 0.25], [0.0, \pi, 0.0]$	The left hand move horizontally by 0.2 to avoid touching the brick, then back to initial position.		
5	right	$[0.75, -0.5, 0.1], [0.0, \pi, 0.0]$ $[0.75, -0.5, -0.02], [0.0, \pi, 0.0]$	The right hand rotates to vertical and moves to above final position, move vertically downward, then release the brick on the final position by opening the brick		
6	right	$[0.75, -0.5, 0.1], [0.0, \pi, 0.0]$ $[0.5, -0.5, 0.25], [0.0, \pi, 0.0]$	The right gripper opens, move upward by 0.1 to avoid hitting the brick, then move directly to initial position.		

## Task C: Position control to perform pick-and-place tasks

### Task C Part i

In Task C, DE NIRO's arms are controlled using inverse kinematics at a velocity level. To achieve smooth and precise motion, DE NIRO's arms are controlled using velocity-based inverse kinematics. Instead of directly specifying joint angles, this approach calculates the required joint velocities to follow a desired trajectory in Cartesian space. By continuously updating joint velocities based on the robot's kinematic model, the system ensures that the end effector moves along the intended path while maintaining stability and responsiveness. Joint movements are determined by calculating instantaneous joint velocities using the following equation:

$$\dot{q} = J^{\#}(q)T \quad (1)$$

where  $\dot{q}$  represents the joint velocity vector,  $J^{\#}(q)$  is the pseudo-inverse of the end effector's Jacobian, and  $T$  is the Cartesian twist matrix, which is defined as:

$$T = \begin{bmatrix} \dot{P} \\ \omega \end{bmatrix} \quad (2)$$

Where  $\dot{P}$  represents the desired linear velocity of the end effector, and  $\omega$  represents the desired angular velocity. By specifying a Cartesian twist  $T$  at each time. The corresponding joint velocities is computed and it enables smooth motion along the desired circular trajectory without changing the orientation. The desired positions provided were assigned to the variables `xyz_des_pick` and `xyz_des_circle` in the main function.

```
#####
## Task C:
# fill in the desired poses
xyz_des_pick = [0.75, 0, 0.93]      # your code here!
xyz_des_circle = [0.75, 0.1, 1.23]  # your code here!
```

## Safety of Operation

In the reachPose function, the robot arm positions itself 0.18m above the object to avoid collisions and ensure a safe and stable approach by preventing collisions with the object or the surrounding environment. This approach is particularly useful when dealing with vertically taller objects, for example a cylinder. By positioning the arm above, the risk of the robot knocking over or disturbing the object is significantly reduced, which is beneficial because it allows the robot to gently lower its end effector onto the object without disturbing its position. This is especially important for ensuring the object remains stable throughout the process. Additionally, the controlled descent and ascent ensure that the robot moves with more precise control over the forces applied. This results in more stable operation with less shaking of the arm which reduces accidental damage to the environment and human by moving too abruptly during the manipulation.

## Task C Part ii

In this task, the brick is grasped from behind rather than from above. The desired orientation of the end effector was defined using Roll, Pitch, and Yaw angles. By considering the orientation of the end-effector frame in relation to the base frame, a  $\pi/2$  rotation about the Y-axis was required to achieve the correct orientation for grasping the brick from behind. This is applied in the code below:

```
rpy_des = [0, np.pi/2, 0] # gripping from behind
```

## Advantages of Quaternions

Quaternions are widely preferred over Euler angles (Roll, Pitch, and Yaw) in robotics for orientation control due to their stability and efficiency in representing 3D rotations. A key advantage is their ability to prevent gimbal lock which is a demerit of Euler angles that occurs when two rotational axes align that reduces freedom of rotation and leads to loss of control. In contrast, quaternions encode rotation in four dimensions using a single unit quaternion (a combination of scalar and vector components). This prevents any loss of rotational freedom since quaternions do not rely on intermediate axis-aligned rotations, and ensures precise and stable motion control, even in complex navigation tasks.

Quaternions also offer computational benefits by requiring fewer calculations for rotations and interpolations compared to Euler angles. The non-linearity of Euler angles can introduce inconsistencies, as the same orientation can be represented by different angle sets. For example,  $(0^\circ, 180^\circ, 0^\circ)$  and  $(180^\circ, 0^\circ, 180^\circ)$  can represent the same orientation and can lead to unexpected flips during interpolation. Quaternions, however, provide consistent orientation representation, using Spherical Linear Interpolation (SLERP) [1] which operates in a 4D unit sphere rather than 3D Cartesian space, allowing for continuous, uniform motion. simplifying interpolation and minimizing unexpected movements. Mathematically, quaternions are defined as:

$$q = e^{\frac{\theta}{2}u} \quad (3)$$

where  $\theta$  is the angle of rotation and  $u$  is the unit vector along the axis of rotation. This representation allows a full 3D rotation to be executed in a single operation, unlike Euler angles, which require multiple steps. The conjugate of a quaternion, denoted as  $q^*$ , is used to reverse the rotation, and the vector is rotated by applying the quaternion operation:

$$p' = qpq^* \quad (4)$$

This ensures efficient and stable rotational transformations and makes quaternions a powerful tool for real-time robotic control systems that demand precision and reliability.

## Task D - Twist computation

### Part i

The aim of this task is to compute the linear ‘velocity’  $\dot{p}$  in ‘velocity\_controller.py’, given the desired end-effector position ‘P\_des’, actual position P and sample time  $\Delta t$  ‘dt’.

We apply the sample formulation of change in displacement over change in time:

$$\text{linearvelocity} = \frac{\Delta \text{displacement}}{\Delta \text{time}} = \frac{P_{\text{final}} - P_{\text{initial}}}{t_{\text{final}} - t_{\text{initial}}} \quad (5)$$

The linear velocity is estimated by computing the difference between the desired position (to be reached) and the actual position (path to track), then dividing it by the sample time.

```
# compute linear and angular velocities given P_des, P, delta angle, r, and dt
# linear displacement. Note that P_des is the desired end-effector position at the next time instant.
dP = (P_des - P)/dt
```

Estimating the end-effector’s velocity based purely on positional changes is a simple and efficient approach. By measuring the difference between the desired and current positions and dividing by the time interval, the velocity can be obtained without additional sensors or differentiation. Its low computational cost makes it suitable for real-time applications.

However, this approach has several inherent limitations. Since it does not consider the robot’s dynamics—such as inertia, acceleration, and external forces—it may lead to inaccuracies in velocity estimation. For example, during rapid or abrupt movements, the estimated velocity may not fully reflect the instantaneous velocity of the end-effector, leading to overshoot, oscillations, or instability in the motion. Additionally, any noise from the position sensors can lead to larger errors. Therefore, position-based control may not be the optimal choice for applications that demand high accuracy and precision.

Additionally, since velocity is derived from discrete position changes rather than predicted. This makes it unable to respond proactively to sudden environmental shifts, posing challenges in dynamic settings. Low sampling rates or system disturbances can also lead to accumulated errors, causing deviations from the intended path.

### Part ii

The angular velocity to reach the desired position also needs to be calculated. To avoid the mathematical singularities when computing, the quaternions are used to express the orientation. The current orientation is calculated using forward kinematics. The inverse of this quaternion is then computed to determine the relative rotation needed to reach the target orientation. By multiplying the desired quaternion with the inverse of the current quaternion, the quaternion error is obtained, which represents the rotational difference.

```
# current pose calculated using forward position kinematics
Pose_vect = self.forward_position_kinematics(joint_values)
P = Pose_vect[0:3].reshape(-1) # position
quat = Pose_vect[3:].reshape(-1) # orientation
```



```
# first take the inverse of the initial orientation quaternion
quat_inv = np.copy(quat)
quat_inv[0:3] = -quat_inv[0:3]

# then compute the quaternion product of the inverse initial orientation
# quaternion with the final orientation quaternion
quat_err = quaternionProduct(quat_des, quat_inv)
```

This is then converted to an axis and rotated around that axis, by doing this the rotation required to move from the current to the final desired orientation can be determined.

```
# angle and axis of error
delta_angle, r = quat2angax(quat_err)
```

Finally, the angular velocity is determined by dividing the rotation axis by the sample time, which gives the rate of change of orientation. This is implemented in the code by computing ‘dw’ using the function  $\theta$  (‘delta\_angle’), the rotation axis ‘r’, and the sample time  $\Delta t$  ‘dt’, following the formulation in Equation 6.

$$w = \frac{\Delta\theta \cdot r}{\Delta t} \quad (6)$$

```
dw = (r*delta_angle)/dt
```

The computed angular and linear velocities are then combined into a twist vector using ‘np.hstack((dP, dw))’.

```
# twist is [linear velocity, angular velocity]
twist = np.hstack((dP, dw))
```

## Task E - Joint velocities computation

The velocity of the end-effector can be determined using its Jacobian and its inverse. However, to handle redundancy or singularities, particularly when the Jacobian is square or non-invertible, the pseudo-inverse is utilized. Since the robot has seven joints while the end-effector moves in three-dimensional space, the Jacobian matrix should take the following form:

$$\tau = \begin{bmatrix} \dot{P} \\ \omega \end{bmatrix} = \begin{bmatrix} v_x \\ x_y \\ v_z \\ \omega_z \\ \omega_y \\ \omega_x \end{bmatrix} = \begin{bmatrix} J_{1,1} & J_{1,2} & \dots & J_{1,7} \\ J_{2,1} & J_{2,2} & \dots & J_{2,7} \\ \vdots & \vdots & \ddots & \vdots \\ J_{6,1} & J_{6,2} & \dots & J_{6,7} \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \vdots \\ \dot{q}_6 \end{bmatrix} \quad (7)$$

The Jacobian of the end-effector ‘J\_ee’ is obtained by passing the joint values ‘joint\_values’ into the self.jacobian method, which is then transformed into an array. Converting it to an array facilitates subsequent mathematical computations. The calculation is performed using the ‘PyKDL’ library, a widely used kinematics and dynamics library in ROS.

```
# compute Jacobian of the end-effector and solve velocity IK with pseudoinverse.
# Use self.jacobian() with joint_values as inputs
J_ee = self.jacobian(joint_values)
J_ee = np.asarray(J_ee) # convert to np.array
```

Then the array subsequently determined using the pseudo-inverse of the Jacobian, as described in Equation 11, in which  $\dot{q}$  represents the joint velocities,  $J^\#(q)$  represents the pseudo-inverse of the Jacobian, and  $\tau$  refers to the previously defined twist vector, which encompasses both linear and angular velocity components.

$$\dot{q} = J^\#(q)\tau \quad (8)$$

```
J_pinv = DPinv(J_ee, eps=1e-10) # compute the pseudoinverse of J_ee
qd = np.matmul(J_pinv, twist) # compute joint velocities from twist
```



Velocity control provides several advantages over position control, particularly in scenarios involving interaction tasks and robots with high degrees of freedom. Position control requires solving inverse kinematics to determine the specific joint angles that place the end-effector in the desired position and orientation. However, this approach can be computationally expensive and introduces challenges such as redundancy, where multiple joint configurations can achieve the same end-effector position, and singularities, which can lead to instability or make certain target positions unreachable.

In contrast, velocity control directly computes the required joint velocities to achieve the desired end-effector motion, eliminating the need for solving inverse kinematics explicitly. This method is computationally more efficient and avoids the issues associated with redundancy and singularities. When an exact solution does not exist, the pseudo-inverse of the Jacobian provides the best approximation in the least-squares sense, minimizing velocity errors. Additionally, by incorporating a small damping factor, velocity control can mitigate singularity-related instabilities, ensuring smoother and more reliable motion. Another key advantage of velocity control is its ability to maintain smooth, continuous motion. Unlike position control, which relies on predefined waypoints and may cause discontinuities, velocity control efficiently interpolates between fewer waypoints, improving responsiveness. This makes it ideal for dynamic tasks requiring real-time adaptation and makes a more intuitive motion. By leveraging differential kinematics and the Jacobian matrix, velocity control provides a scalable and adaptive approach, particularly beneficial for complex robotic systems like the Baxter arm, where precise, real-time interaction is essential for stable performance.

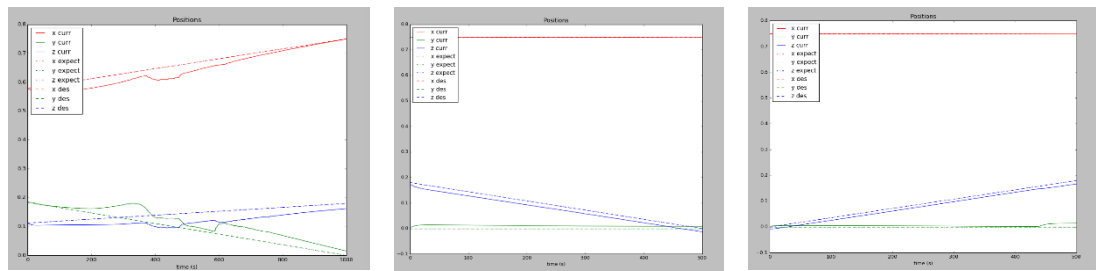


Figure 7 & 8 & 9: Tracking accuracy of the velocity controller

## Task F – Null Space Projector

The robotic arm has 7 degrees of freedom (DOF), whereas task space motion requires a maximum of 6 DOF. This discrepancy introduces redundancy, which can be effectively managed by assigning the robot a secondary motion task that does not interfere with the primary task. A widely used approach to achieve this is the null space projection method.

In this method, the secondary joint velocity lies within the null space of the primary task, i.e. its projection onto the final task motion is zero. This allows the robot to perform the secondary task without affecting the primary motion. Common objectives for these secondary tasks include obstacle avoidance, minimising energy consumption and optimising force distribution for improved efficiency and performance.

The following section provides analytical proof of how the Null Space Projection Method works, beginning with its mathematical formulation presented below:

$$\dot{q} = \dot{q}_1 + N\dot{q}_2 \quad (9)$$

$\dot{q}$  is the overall joint velocity,  $\dot{q}_1$  is the joint velocity for primary task,  $\dot{q}_2$  is the joint velocity for secondary task,  $N$  is the null space projector

The expression of the null space projector is:

$$N = I - J^\dagger J \quad (10)$$

$I$  is identity matrix,  $J$  is the jacobian matrix of the task,  $J^\dagger$  is the pseudo-inverse of  $J$

It should be noted that  $J^\dagger$  has one important property:

$$JJ^\dagger J = J \quad (11)$$

The task-space velocity  $\dot{x}$  is:

$$\dot{x} = J\dot{q} \quad (12)$$

Substitute Equation 9 into it,  $\dot{x}$  can be expressed as:

$$\dot{x} = J(\dot{q}_1 + N\dot{q}_2) \quad (13)$$

$$\dot{x} = J\dot{q}_1 + JN\dot{q}_2 \quad (14)$$

Substitute Equation 10 into it,  $\dot{x}$  can be further expressed as:

$$\dot{x} = J\dot{q}_1 + J(I - J^\dagger J)\dot{q}_2 \quad (15)$$

$$\dot{x} = J\dot{q}_1 - J\dot{q}_2 + JJ^\dagger J\dot{q}_2 \quad (16)$$

Substitute Equation 11 into it,  $\dot{x}$  is simplified as:

$$\dot{x} = J\dot{q}_1 + J\dot{q}_2 - J\dot{q}_2 \quad (17)$$

$$\dot{x} = J\dot{q}_1 \quad (18)$$

As shown in the final equation, the secondary motion does not affect the task space velocity  $\dot{x}$ . The secondary task motion will not interfere with the primary motion. The following code is the expression of the null space projector in the velocity\_controller.py file:

```
#####Task F
# Compute projector into the ee Jacobian null space and joint velocity to reach desired configuration
I = np.eye(nj) # I is the identity matrix
Proj = I - np.matmul(J_pinv, J_ee) # this is the null space projector: N = I - J^†*J
```

## Task G – Redundancy Resolution

Based on the Null Space Projection Method defined in Task F, the redundancy of the DE NIRO robot can be effectively used by introducing a secondary task within the null space of the primary task. The secondary task serves multiple objectives. The following section discusses two common objectives in detail.

### Part i

In Part I, the goal of the secondary task is to avoid contacting the obstacle while still successfully executing the primary task of grabbing the white brick. A desired joint position is specified, and the corresponding secondary joint velocity is derived as follows:

$$\dot{q}_2 = \frac{q_{2desired} - q_{2current}}{t} \quad (19)$$

$q_{2desired}$  is the final joint position,  $q_{2current}$  is the current position,  $t$  is the sampled time

As discussed in Task F, the final joint velocity is the sum of the primary joint velocity and the secondary joint velocity, expressed as:

$$\dot{q} = \dot{q}_1 + N\dot{q}_2 \quad (20)$$

The following code provides the Python implementation for this calculation. The  $q_{desired}$  represents an extended period, the calculated secondary joint velocity is scaled down by multiplying it by 0.01.

```
#####Task G part i
# Compute secondary joint velocities to reach desired configuration q_desired, given the current joint values joint_values
# and sampling time dt
q_desired = self.joint_des
# secondary joint velocities is calculated from difference between q_desired and joint_values divided by sampling time dt.
qd2 = (q_desired - joint_values)/dt
# note that q_desired is the final joint configuration, so we artificially slow the speed down here
# (as if it is interpolating over a longer time period).
qd2 = 0.01 * qd2

# final task motion is sum of primary motion and secondary motion
qd = qd + np.matmul(Proj, qd2) # projection in Null space (equation from task F)
```

As shown in Figure 10 and 11, the robot arm successfully completed the primary task while concurrently achieving the secondary task of obstacle (red sphere) avoidance.

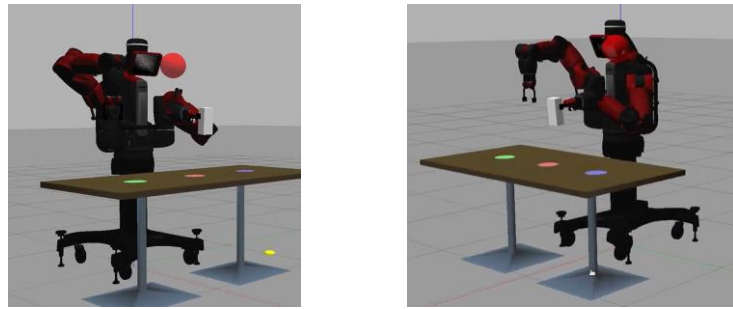


Figure 10 & 11: DE NIRO executes both primary and secondary tasks

### Effects of Null Space Projection

There are several advantages to projecting joint velocities into the null space of the primary task. This technique exploits system redundancy to optimise secondary objectives while ensuring that the primary task remains unaffected. The section below discusses its two major benefits:

#### Improved Manipulability

Manipulability refers to the ability of the robot arm to move efficiently in different directions within its workspace. By utilising redundancy, the system can adopt articulated configurations that increase manipulability. High manipulability helps efficient motion planning as it ensures smooth and precise movements. It also enables the robot to perform internal tasks, such as avoiding obstacles, without changing the end-effector position.

Additionally, maximising manipulability helps to avoid singularity. Singularity occurs when the determinant of the Jacobian matrix becomes zero. The arm loses motion in certain directions. This can have a significant impact on the functionality of the robot and needs to be mitigated. By maximising manipulability, the system ensures that the robot maintains configurations that allow smooth motion, thereby avoiding singular positions.

#### Increased Energy Efficiency

Using null space motion helps optimise motion planning by preventing unnecessary large joint movements. As a result, the torque required to achieve the desired end-effector position is minimised. This optimisation results in reduced energy consumption and actuator effort. The overall efficiency of the robotic system is improved.

### Other Strategies

It is important to note that, in addition to null space projection, several other local resolution methods can be used to exploit redundancy in robotic systems.

One commonly used approach is task-space augmentation. This method extends direct kinematics by incorporating functional constraints that define the required level of dexterity for the redundant structure [2]. While this method effectively manages multiple tasks and performs well in real-time control, it is generally more computationally expensive than the Null Space Projection method.

The Jacobian-based method is also used to address redundancy by selecting a path that optimises a specific desired outcome such as energy efficiency among all available paths. However, the resulting joint-space motion is non-repeatable and singularities may not be globally avoided [3].

## Part ii

With the null space projection introduced, we can now adjust the secondary task without affecting the primary task. The goal of this secondary task is to control the elbow movement of the robot arm along a circular path, avoiding contact with obstacles.

To do this, the Jacobian matrix of the link must be calculated. The `velocity_controller.py` file uses the `link_jacobian()` function to do this. It is important to note that the Jacobian matrix has a size of  $6 \times 7$ . The rows represent both linear and angular velocity in the XYZ directions within the task space, while the columns correspond to the 7 DOF of the robot arm. The inclusion of all seven joints ensures full control of the arm's motion.

However, since only the elbow motion is considered, only the values for the first four joints (up to the elbow) are stored for Jacobian calculation. The remaining joints are assigned zero. The output Jacobian matrix contains only the first three rows of the Jacobian matrix, which is the linear motion in XYZ direction.

Below is the code snippet used for the implementation.

```
def link_jacobian(self, joint_values=None):
    """ computes the Jacobian of the secondary link (in this case, the elbow) """
    # take only initial 4 joints (up to elbow)
    q_elbow = np.copy(joint_values[0:4])

    nj = q_elbow.shape[0]          # number of joints up to the secondary link
    nj_tot = self.num_jnts         # total number of joints in the robot arm
    jacobian = PyKDL.Jacobian(nj)  # set up a PyKDL Jacobian of the right size for the secondary link
    q_kdl = PyKDL.JntArray(nj)    # set up a PyKDL joint array to calculate the Jacobian
    for i in range(nj):           # loop through the joints up to the secondary link
        q_kdl[i] = q_elbow[i]     # fill in the PyKDL joint array

    #####
    # Task E:
    # Fill in the function to compute elbow Jacobian. Inputs are the joint values in KDL and the jacobian matrix
    self.jac_kdl_link.JntToJac(q_kdl, jacobian)

    J_link = self.kdl_to_mat(jacobian) # convert the jacobian from PyKDL format to numpy array

    # after computing the jacobian for the elbow, we need to convert it to a full size jacobian
    J = np.zeros((6, nj_tot)) # Total Jacobian matrix. Fill in with J_link of elbow
    J[:, :J_link.shape[1]] = J_link

    return J[0:3,:] # take only linear part of the jacobian
```

The elbow Jacobian matrix  $J_{\text{elbow}}$  is computed using the above function. Similar to Task E, the pseudo-inverse of elbow Jacobian  $J_{\text{pinv\_elbow}}$  is then calculated. Given the task space velocity  $\text{vel\_elbow}$ , the secondary joint velocity  $\text{qd2}$  can be determined as follows:

$$\dot{q}_2 = J_{\text{elbow}}^+ \dot{x}_{\text{elbow}} \quad (21)$$

It should be noted that since only the elbow velocity and the Jacobian for the elbow joints are considered, the secondary joint velocities for the remaining joints are assigned zero. In other words, their motion remains unchanged.

Finally, the calculated secondary joint velocity is projected into null space and added to the final task velocity. As explained in Task F, this ensures that the secondary task of obstacle avoidance does not interfere with the primary task.

Below is the code snippet used for the implementation:

```
#####Task G part ii
J_elbow = self.link_jacobian(joint_values) # the elbow Jacobian

# pseudo-inverse of the elbow Jacobian
J_pinv_elbow = DPinv(J_elbow, 1e-6)
# secondary joint velocities are the product of pseudo-inverse of the elbow Jacobian and given vel_elbow
qd2 = np.matmul(J_pinv_elbow, vel_elbow)
# final task motion is sum of primary motion and secondary motion
qd = qd + np.matmul(Proj, qd2) # projection in Null space (equation from task F)
```

## Task H– Demolishing the Wall

We looked at how we can control DE NIRO to interact with its environment using forces and torque. The primary equation for torque control in robotics is as follows:

$$\tau = H(q)\ddot{q} + C(q, \dot{q})\dot{q} + \tau_g(q) + \tau_{ext} \quad (22)$$

$\tau$  (Joint Torque) is the torque applied to the robot's joints.  $H(q)$  is the inertia matrix of the robot and depends on the robot's configuration  $q$ .  $C(q, \dot{q})$  represents the centripetal and Coriolis effects, which arise when the robot is in motion.  $\tau_g(q)$  represents the gravitational torque acting on the robot's joints.  $\tau_{ext}$  represents any external torque applied to the system.

To simplify it, we adjusted the gravity vector in simulation platforms like Gazebo. We changed the gravity vector from the real-world value of  $g = -9.8$  to nearly negligible levels ( $g = -0.01$ ). Therefore, the gravity term  $\tau_g(q)$  was dropped. Our aim was to make DE NIRO demonstrate a more precise force on a brick to destroy the wall. To convert the forces at the robot's end effector into joint torques, we used the transpose of the Jacobian matrix:

$$\tau_{ext} = J(q)^T \tau_{ext} F_{ext} \quad (23)$$

Then, we used a PD controller.

Proportional term (P): The proportional term adjusts the control input based on the current position error.

Derivative term (D): The derivative term adjusts the control input based on the rate of change of the error.

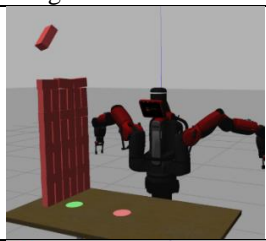

By adjusting the proportional and derivative terms, the controller can directly control the robot's motion direction based on the given error and error change rate, thereby simplifying the process of specifying the force direction.


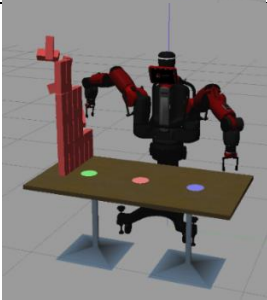

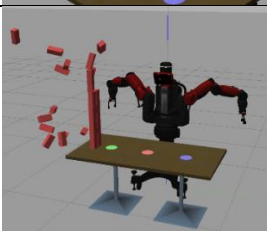
### Task H Part i: Throwing the Brick

```
# change external_force to make DE NIRO throw the brick into the wall (your code here)
external_force = np.array([0, -15.0, 0.0])
# force to accelerate the end effector toward the wall
```

The gravity has been set very low; therefore, we only need to tune the x and y components. The goal is in the negative y direction, so a negative force was added. We tune the values of y and x in the following form: We first tune y because the y component leads the arm to move toward the wall, and then we tune x, which has a slighter effect.

Table x: Tune parameters for throwing brick

Parameter	Descriptions	Image
[0,-5,0]	We first tried -5 as our initial value. However, we found that DE NIRO threw the brick very high. Therefore, we planned to make the negative y larger to hit the brick more centrally.	
[0, -10, 0]	This time, we tuned the y component to -10. In the image, we can see that DE NIRO hit more centrally, so we thought we had found the right direction.	

[0, -20, 0]	To hit the brick more accurately, we tuned the y component force to be larger. The effect went well, so then we thought about tuning the x component direction.	
[-2, -20, 0]	To start, we tuned the x component to a value of -2, but it didn't show much difference. The reason for that is we are using Gazebo for simulation. Although the parameters are the same, the results can vary, and this variation is larger than the change in the x parameter.	
[2, -20, 0]	The way we tuned the x component to negative values didn't affect much, so we then turned it to positive 2. However, it collided worse. Therefore, we think keeping the x component at 0 might be a good option.	
[0, -15, 0]	We slightly tuned the parameter of the y component between -10 and -20. After several attempts, we found that [0, -15, 0] might perform best. Only a few bricks are left.	

## Task H part ii Build up speed

To achieve greater speed, the robot arm should gain greater momentum. According to Newton's law, acceleration is set by specifying torque. The robot arm should use a larger distance and more time to build up speed based on the acceleration. As seen in part I, the arm directly throws forward. In part II, we want DE NIRO to move its arm away from the goal, giving it more time and distance to build up speed.

```
back = False
while not rospy.is_shutdown():
```

We define back = False before the whole while loop.

```
external_force = np.array([-1.5, 5.0, 0.0]) # Initial pull-back force

if not back and y >= y0+0.2:
    #We set back= flase before the while loop in line 490
    #Keep using inital pull-back force, until reach the distance we set
    external_force = np.array([2.0, -10.0, 0.0])
    back = True # Prevent re-triggering point # Apply final force
elif back:
    external_force = np.array([-1.5, -10.0, 0.0]) # Apply final force



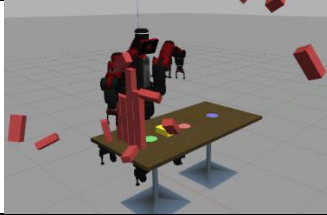

position_error = np.array([x0 - x, y0 - y, z0 - z]) # position error
velocity_error = np.array([- x_dot, - y_dot, - z_dot]) # velocity error
```



```
# if the arm moves past the y point, open the gripper
if y < y_stop + 0.2 and gripper_closed:
    left_arm.gripper_open()
    gripper_closed = False
```

In the code, we set a force that makes the arm go back. Then, we use an if condition to set the distance we need the arm to move away. After that, we set it back to be sure and go to the elif condition to keep moving forward. We then found that the gripper opened too late. When the release point is the same as in part I, the acceleration had already decreased. So, we tuned the parameters regarding when to open the gripper to make it smoother.

Table x: Tune parameters for throwing brick with acceleration

Parameter	y >= y0+0.4		y >= y0+0.3	
Images				
Parameter	y >= y0+0.2		y >= y0+0.1	
Images				

Compared to the parameter for distance,  $y \geq y_0 + 0.2$  works best. Finally, we found that because of the release point change, the arm's movement direction also changed compared to part I due to the timing of when the gripper opens. Therefore, we can no longer use the parameters we tuned in part I. Instead, we tuned a different value, as shown in the code.

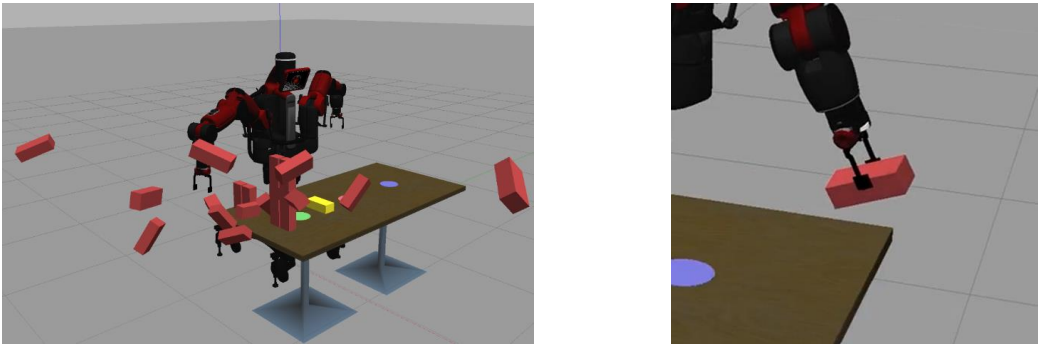


Figure 12: Final result for throwing brick

Task I: Cleaning up Part i: Wiping the Table

After DE NIRO demolished the wall, the robot needed to clean up the table. A sponge was provided for it to carry and clean the target mats off the table.

```
external_force = np.array([0.0, -5.0, -15.0, 0, 0, 0]).reshape((6, 1))
# Let left arm work toward left direction
# force to push down and across on the table
external_force = np.array([0.0, 5.0, -15.0, 0, 0, 0]).reshape((6, 1))
# Let right arm work toward left direction
```



We faced singularity the first couple times. After 20 tuning attempts, we found that a linear force of 15N downwards fit well, and -5N in the y direction to move left, followed by 5N in the y direction to move right. To prevent DE NIRO from using only one line of the sponge when cleaning the table, it is better to include angular velocity terms in the torque control algorithm. This way, the robot will apply rotational forces to the sponge, allowing it to use the entire surface.

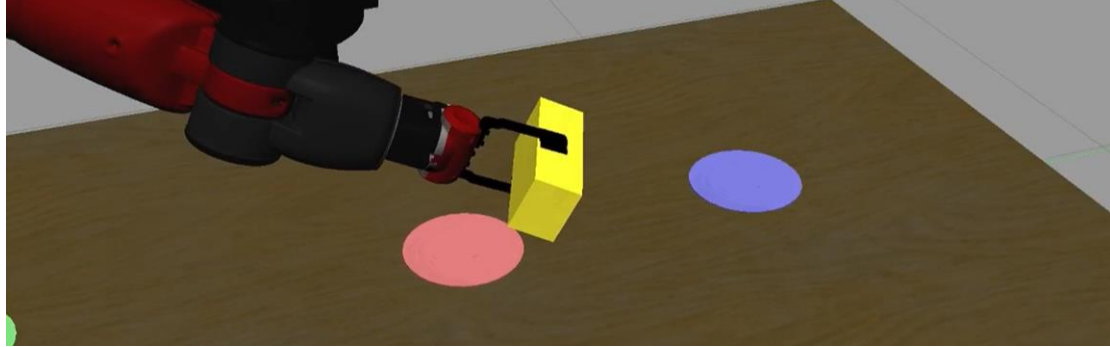


Figure 13: Cleaning Table

## Part ii : Improving Sponge Contact

```
external_force += 20.0 * np.array([- x_dot, - y_dot * 0.2, - z_dot, x_omega, y_omega,
z_omega]).reshape((6, 1)) # derivative feedback
```

To solve the control issues and make it work more smoothly, rotational components were added to the derivative terms to compensate for the twist. After several attempts at tuning  $[-1, 0.5, 1]$  and  $[1, 0.5, 0]$ , we found that  $[1, 1, 1]$  worked best. Finally, we successfully enabled the robot to maintain the sponge's orientation during the cleaning process. After the tuning adjustments, the sponge had more surface contact with the table, which improved the efficiency.



Figure 14: Cleaning Table with rotation force in derivative control

## Bibliography

1. Wang D, Cao W, Takanishi A. Dual-Quaternion-Based SLERP MPC Local Controller for Safe Self-Driving of Robotic Wheelchairs. *Robotics*. 2023 Nov 13;12(6):153.
2. L. Sciavicco, Siciliano B. The Augmented Task Space Approach for Redundant Manipulator Control. *IFAC Proceedings Volumes*. 1988 Oct 1;21(16):125–9.
3. De Luca A. Robotics 2 Robots with kinematic redundancy [Internet]. [cited 2025 Mar 12]. Available from: [http://www.diag.uniroma1.it/deluca/rob2\\_en/02\\_KinematicRedundancy\\_1.pdf](http://www.diag.uniroma1.it/deluca/rob2_en/02_KinematicRedundancy_1.pdf)