Applied Robotics Coursework 1 – Model

James Skinner (01700960), Madelaine Wood (01762829),
Harry Schlote (01746509), Fernanda Espinoza (01783661)

Table of Contents

1. Introduction

This report documents the development of 'Coursework 1 – Model', as part of the Applied Robotics module. Accurate robot modelling is arguably the most important part of robot design and control, and the following tasks detail the key modelling processes, involving; Forward, Inverse, and Differential Kinematics, as well as Robot Control (tuning a proportional-integral-derivative controller).

The robot consists of five links, including the base link and end effector, connected by three revolute joints. For consistency, the lengths of the three long links are denoted as l_0, l_1 and l_2, with respective joint angles q_0, q_1 and q_2.

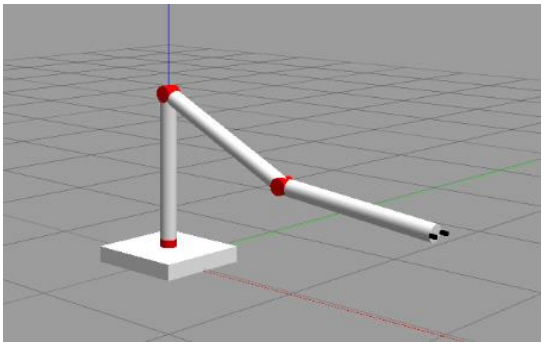


Figure 1: The robot arm, modelled in the Gazebo simulator

2. Forward Kinematics

2.1. Task A: Compute the initial D-H table of the robot arm

[1] The D-H parameters of each link of the robot arm were calculated to create the D-H table of the robot arm in its default configuration, as seen in Figure 2. This would be used to compute the transformation matrices, to obtain the position vector of the end-effector with respect to the base frame (x_0, y_0, z_0), in later tasks.

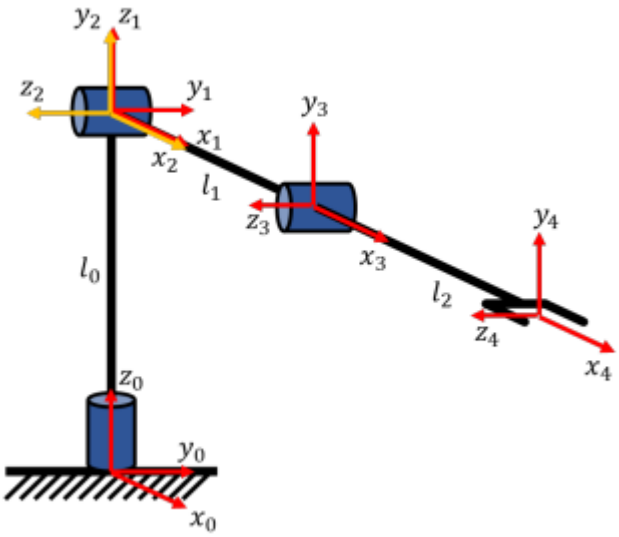


Figure 2: The robot arm in its default configuration

Where:

α_i = The angle between z_{i-1} to z_i measured about x_{i-1}
a_i = The distance between z_{i-1} to z_i measured about x_{i-1}
d_i = The distance between x_{i-1} to x_i measured about z_{i-1}
Θ_i = The angle between x_{i-1} to x_i measured about z_{i-1}

Table 1: D-H table for robot in its default configuration

i	d_i	Θ_i	a_i	α_i
1	l_0	0	0	0
2	0	0	0	$\pi/2$
3	0	0	l_1	0
4	0	0	l_2	0

The code in which this D-H table was implemented is as follows:

```
1.          #DH Table based on diagram in tutorial sheet (See report)
2.          self.DH_tab = np.array([[self.links[0], 0, 0, 0],
3.                                  [0, 0, 0, np.pi/2],
4.                                  [0, 0, self.links[1], 0],
5.                                  [0, 0, self.links[2], 0]])
6.
```

With l_0 being stored in self.links[0], likewise for l_1 and l_2. It should be noted that the Theta values (aka joint angles) were set to zero because, as seen from the default configuration, there is currently zero rotation about the z-axis for each joint. The D-H table is then updated with these values later in the code, as the joints rotate.

2.2. Task B: Complete the code for the D-H matrix

[2] The calculated D-H parameters can be used to compute the transformation matrix from joint i-1 to joint i, using the following matrix:

$$ {}^{i-1}T_i = \begin{bmatrix} cos\theta_i & -sin\theta_i & 0 & \alpha_i \\ sin\theta_i cos\alpha_i & cos\theta_i cos\alpha_i & -sin\alpha_i & -sin\alpha_i d_i \\ sin\theta_i sin\alpha_i & cos\theta_i cos\alpha_i & cos\alpha_i & cos\alpha_i d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

This was implemented in the code, as follows:

```
1.      # Based on DH transformation matrix (from i to i-1) given in tutorial sheet
2.       DH_matrix = np.array([[np.cos(theta), -np.sin(theta), 0, a],
3.                             [np.sin(theta)*np.cos(alpha), np.cos(theta)*np.cos(alpha), -
    np.sin(alpha), -np.sin(alpha)*d],
4.                             [np.sin(theta)*np.sin(alpha), np.cos(theta)*np.sin(alpha),
    np.cos(alpha), np.cos(alpha)*d],
5.                             [0, 0, 0, 1]])
```

It is important to note that this is a generic matrix assuming translation and rotation occur in the positive direction. Therefore, the sign of the D-H parameters is only considered when substituting values.

2.3. Task C: Complete the code to compute forward kinematics

To calculate the pose of the robot's end effector in terms of the base frame, a transformation matrix must be calculated for each joint, transforming from the that joint's frame (i) to the previous joint's frame (i-1). Once these are computed, they can be multiplied together to give a compound transformation matrix of the overall robot, giving the position and orientation of the end effector with respect to the base, as follows:

$$^{0}T_{N} = {}^{0}T_{1}\,{}^{1}T_{2} \ldots {}^{N-1}T_{N}$$

This can be coded iteratively, using a loop. At each iteration, the following calculation is performed:

$$^{0}T_{i} = {}^{0}T_{i-1}\,{}^{i-1}T_{i}$$

The transformation from the i-1th to 0th frame (T_0_i_1) was initialised as the identity matrix, so the first iteration of the loop wouldn't fail. Then, the transformation matrix of joint i with respect to joint i-1 (T_i_1_i) was calculated, using the code from tasks A and B. At the first iteration of the loop, the transformation of joint 1 (current i) with respect to joint 0 (current i-1) was calculated and multiplied with T_0_i_1, to calculate T_0_i - the transformation of the ith joint with respect to the base frame. This result was then stored in T_0_i_1, such that the next iteration of the loop uses this updated value to calculate the transformation matrix up to the next joint. This occurs until all the joints have been accounted for within the loop (for i in range (self.nj)), implemented in code as follows:

```
1.  # Declare the identity matrix, to initialise T_0_i_1 (so first iteration does not fail)
2.          T_0_i_1 = np.identity(4)
3.
4.          # for i in range() --> 0, 1, 2... self.nj
5.          for i in range(self.nj):
6.
7.              # where [i,:] = ith row, all columns (i.e. i = 0, select first row vector)
8.              DH_params = np.copy(self.DH_tab[i,:])
9.              #print('q',q)
10.             #print(DH_params)
11.
12.             # Based on current joint angles (q values), update DH table...
13.             # For revolute joints (rotating motion)...
14.             if self.joint_types[i] == 'r':
15.                 DH_params[1] = DH_params[1]+q[i]
16.             # For prismatic joints (linear motion)
17.             elif self.joint_types[i] == 'p':
18.                 DH_params[0] = DH_params[0]+q[i]
19.
20.             T_i_1_i = DH_matrix(DH_params) #Pose of joint i wrt i-1
21.             # ...(so, first iteration of loop converts joint 1 to joint 0 (base))
22.
23.             ########################## TASK 3 (replace np.eye(4) with correct matrices)
24.             # Original code...
25.             #T_0_i = np.matmul(np.eye(4), np.eye(4)) #Pose of joint i wrt base
26.             T_0_i = np.matmul(T_0_i_1, T_i_1_i) #(see report for formula explanation)
27.
28.             T_0_i_1 = T_0_i # update for next iteration of loop (updates i-1 w.r.t base
29.             # ... which, at next iteration, is equal to current i w.r.t base)
30.
31.         T_0_n_1 = T_0_i
32.         DH_params = np.copy(self.DH_tab[self.nj, :])
33.         T_n_1_n = DH_matrix(DH_params)
34.         T_0_n = np.matmul(T_0_n_1, T_n_1_n)
```

3. Inverse Kinematics

3.1. Task D: Checking if a point is in the workspace of the robot arm

Before the inverse kinematics of the robot arm is computed, a check should be done to make sure that the desired point is inside the robot's reachable workspace. This workspace, which can be seen in Figure 3, is defined by a sphere that is centered on the end of the first link, with a smaller sphere subtracted from it due to this area also being unreachable. Thus, the end effector cannot reach beyond rmax (radius of outer sphere) or within rmin (radius of inner sphere).

[1] The following equations are used to determine whether a point is within the reachable workspace:

$$val = x_p^2 + y_p^2 + (z_p - l_0)^2$$

$$rmax = (l_1 + l_2)$$

$$rmin = (l_1 - l_2)$$

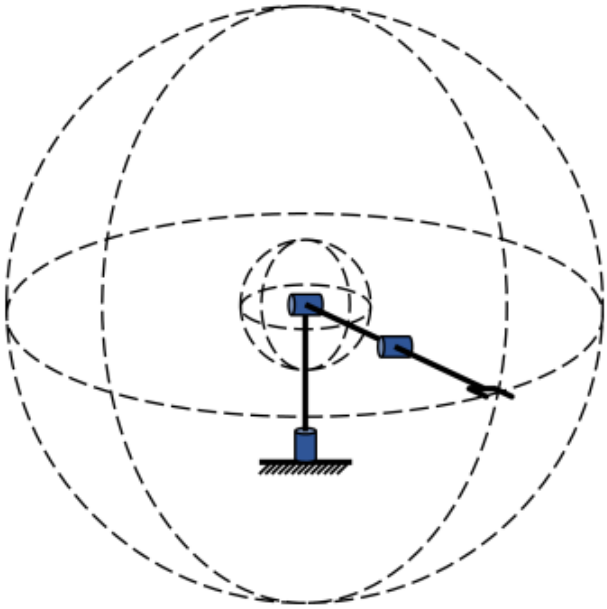


Figure 3: Visualisation of the robot's workspace

'val' is the square of the distance from the end of the first link to the end effector, which is checked to see if the desired point is within the robot's reachable workspace. If 'val' is within the robot's reachable workspace it will abide by these constraints:

$$val \leq rmax^2$$

$$val \geq rmin^2$$

The radius of the outer sphere, 'rmax', is calculated from when the links are fully extended, at $\Theta_2 = 0$. Whereas, the radius of the inner sphere, 'rmin', is calculated from when the links are fully folded, at $\Theta_2 = \pi$.

The code to calculate rmax, rmin and val can be seen below:

```
# Following definitions of val, r_max and r_min in tutorial sheet...
        val = np.power(xP, 2) + np.power(yP, 2) + np.power((zP - l0), 2)
        r_max = l1 + l2
        r_min = l1 - l2
```

To perform the exponent in python, the numpy function 'np.power()' was used.

3.2. Task E: Calculating inverse kinematics

Having added the ability to check if the desired position is inside the workspace, the geometric inverse kinematics could then be calculated to find the joint angles required for the robot arm to reach these desired positions. Calculating q_0 is fairly straight forward:

$$\frac{y_P}{x_P} = \frac{(l_1 \cos(q_1) + l_2 \cos(q_1 + q_2)) \sin(q_0)}{(l_1 \cos(q_1) + l_2 \cos(q_1 + q_2)) \cos(q_0)}$$

$$\frac{y_P}{x_P} = \frac{sin(q_0)}{cos(q_0)}$$

$$\frac{y_P}{x_P} = tan(q_0)$$

$$q_0 = arctan2(y_P, x_P)$$

To calculate q_1 and q_2, the forward kinematic equations can be simplified, ignoring joint 0 (q_0) as this only rotates about the z-axis. This allows the robot to then be treated as planar.

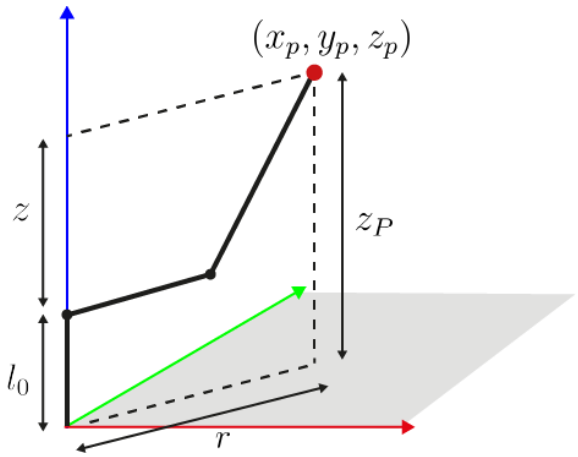


Figure 4: Diagram to outline how the robot can be simplified as a planar two-link manipulator

Looking in this new plane, denoted by the dashed lines in Figure 4, *r* and *z* can be calculated as follows:

$$r = \sqrt{{x_p}^2 + {y_p}^2} = l_1 cos(q_1) + l_2 cos(q_1 + q_2)$$

$$z = z_P - l_0 = l_1 sin(q_1) + l_2 sin(q_1 + q_2)$$

Having computed *r* and *z*, q_1 and q_2 can then be calculated, as follows:

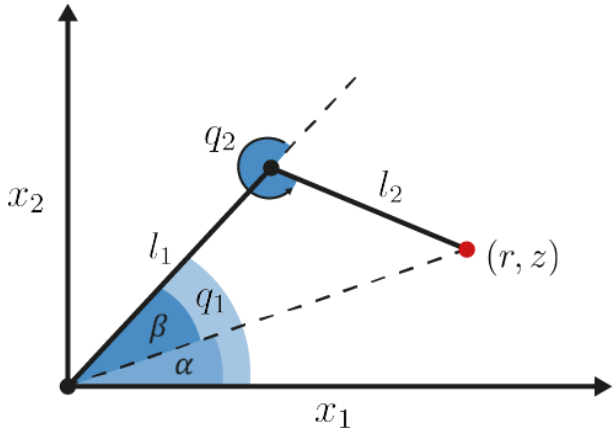


Figure 5: A planar view of links 1 and 2, showing the joint angles (q1 and q2) and end effector position (r and z)

To calculate q_1 of the robot arm:

$$q_1 = \alpha + \beta$$

$$\alpha = arctan2(z, r)$$

Using the cosine rule:

$$l_2{}^2 = r^2 + z^2 + l_1{}^2 - \left(2 * l_2{}^2 * \sqrt{r^2 + z^2} * \cos(\beta)\right)$$

$$\beta = cos^{-1}\left(\frac{r^2 + z^2 + l_1{}^2 - l_2{}^2}{2 * l_1 * \sqrt{r^2 + z^2}}\right)$$

$$q_1 = arctan2(z, r) + cos^{-1}\left(\frac{r^2 + z^2 + l_1{}^2 - l_2{}^2}{2 * l_1 * \sqrt{r^2 + z^2}}\right)$$

The cosine rule can also be used to calculate q_2:

$$r^2 + z^2 = l_1{}^2 + l_2{}^2 - 2 * l_1 * l_2 \cos(q_2 - \pi)$$

$$\cos(q_2 - \pi) = \frac{r^2 + z^2 - l_1{}^2 - l_2{}^2}{-2 * l_1 * l_2}$$

$$-\cos(q_2) = \frac{r^2 + z^2 - l_1{}^2 - l_2{}^2}{-2 * l_1 * l_2}$$

$$q_2 = -cos^{-1}\left(\frac{r^2 + z^2 - l_1{}^2 - l_2{}^2}{-2 * l_1 * l_2}\right)$$

3.3. Task F: Calculating inverse kinematics in Python

Whilst still treating links 1 and 2 as a planar two-link manipulator, there are two possible solutions to the inverse kinematics for all points in the workspace, as highlighted in Figure 6 (with the exception of those lying on the boundaries of the inner and outer spheres of the workspace).

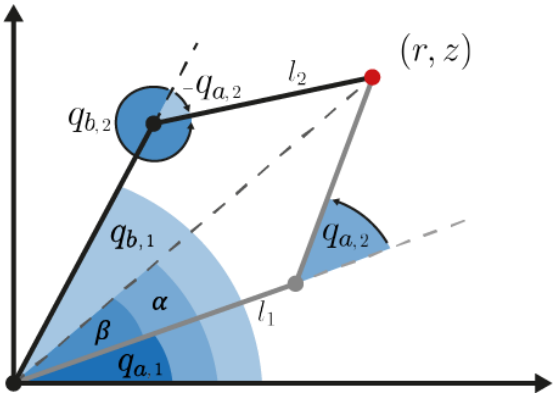


Figure 6: Diagram highlighting the two possible configurations of the robot

The equations derived in task E are used to calculate the angles $q_{b,1}$ and $q_{b,2}$. It should be noted that as the $q_{b,2}$ expression involves -arccos (i.e., a negative value), this actually calculates the acute angle $-q_{a,2}$, which is equivalent to $q_{b,2}$ in the opposite direction. Having calculated expressions for these angles, it is then simple to calculate the angles for the alternate configuration, $q_{a,1}$ and $q_{a,2}$. With reference to the diagram in Figure 6, $q_{a,2}$ is simply the negative value of $q_{b,2}$ (aka $-q_{a,2}$). Similarly, $q_{a,1}$ is calculated with $\alpha - \beta$, as opposed to $\alpha + \beta$.

These calculations were implemented in the code, as follows:

```
1.          # Declare empty arrays (to put values into)
2.          q_a = np.zeros(3)
3.          q_b = np.zeros(3)
4.
5.          # Calculate q0 values (the same for both configurations)
6.          q_a[0] = np.arctan2(yP, xP)
7.          q_b[0] = q_a[0]
8.
9.          # Declare terms r and z (for neat code)
10.         r = np.power((np.power(xP, 2) + np.power(yP, 2)), 0.5)
11.         z = zP - l0
12.         # Declare z squared + r squared (for neat code)
13.         z_2_r_2 = np.power(r, 2) + np.power(z, 2)
14.
15.         # Calculate q2 values
16.         # *Note: We can have +ve or -ve values...
17.         q_a[2] = np.arccos((z_2_r_2 - np.power(l1, 2) - np.power(l2, 2))/(2*l1*l2))
18.         q_b[2] = -np.arccos((z_2_r_2 - np.power(l1, 2) - np.power(l2, 2))/(2*l1*l2))
19.
20.         # Calculate q1 values
21.         # *Note: For corresponding -ve q2 value, q1 = alpha + beta (not alpha - beta)
22.         q_a[1] = np.arctan2(z, r) - np.arccos((z_2_r_2 + np.power(l1, 2) - np.power(l2,
    2))/(2*l1*np.power(z_2_r_2, 0.5)))
23.         q_b[1] = np.arctan2(z, r) + np.arccos((z_2_r_2 + np.power(l1, 2) - np.power(l2,
    2))/(2*l1*np.power(z_2_r_2, 0.5)))
24.
25.         q = [q_a, q_b]
```

It should be noted that as the robot consists of three joints, there is technically four different possible configurations of joint angles, where joint 0 (q_0) could be rotated by 180 degrees (π radians) and the end effector could still reach the desired position. This would provide two more solutions which would visually appear identical to the aforementioned configurations, but with differing values of q_0, q_1 and q_2.

4. Differential Kinematics

4.1. Task G: Computing the Jacobian

The forward kinematics equation for a planar 2 degrees-of-freedom robot are as follows:

$$x_p = (l_2 \cos(q_1) + l_2 \cos(q_1 + q_2))cos(q_0)$$

$$y_p = (l_1 \cos(q_1) + l_2 \cos(q_1 + q_2))sin(q_0)$$

To calculate the Jacobian, the forward kinematics equations should be partially differentiated:

$$\frac{\delta x_p}{\delta q_0} = -(l_1 \cos(q_1) + l_2 \cos(q_1 + q_2)) \sin(q_0) = J_{1,1}$$

$$\frac{\delta x_p}{\delta q_1} = -(l_1 \sin(q_1) + l_2 \sin(q_1 + q_2))\cos(q_0) = J_{1,2}$$

$$\frac{\delta x_p}{\delta q_2} = -l_2 \sin(q_1 + q_2) \cos(q_0) = J_{1,3}$$

$$\frac{\delta y_p}{\delta q_0} = -(l_1 \cos(q_1) + l_2 \cos(q_1 + q_2))\cos(q_0) = J_{2,1}$$

$$\frac{\delta y_p}{\delta q_1} = -(l_1 \sin(q_1) + l_2 \sin(q_1 + q_2))\sin(q_0) = J_{2,2}$$

$$\frac{\delta y_p}{\delta q_2} = -l_2 \sin(q_1 + q_2) \sin(q_0) = J_{2,3}$$

$$\frac{\delta z_p}{\delta q_0} = 0 = J_{3,1}$$

$$\frac{\delta z_p}{\delta q_1} = l_1 \cos(q_1) + l_2 \cos(q_1 + q_2) = J_{3,2}$$

$$\frac{\delta z_p}{\delta q_2} = l_2 \cos(q_1 + q_2) = J_{3,3}$$

These can be inserted into the Jacobian matrix:

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

$$\begin{bmatrix} \dot{x}_p \\ \dot{y}_p \\ \dot{z}_p \end{bmatrix} = \begin{bmatrix} \frac{\delta x_p}{\delta q_0} & \frac{\delta x_p}{\delta q_1} & \frac{\delta x_p}{\delta q_2} \\ \frac{\delta y_p}{\delta q_0} & \frac{\delta y_p}{\delta q_1} & \frac{\delta y_p}{\delta q_2} \\ \frac{\delta z_p}{\delta q_0} & \frac{\delta z_p}{\delta q_1} & \frac{\delta z_p}{\delta q_2} \end{bmatrix} \begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} = \begin{bmatrix} J_{1,1} & J_{1,2} & J_{1,3} \\ J_{2,1} & J_{2,2} & J_{2,3} \\ J_{3,1} & J_{3,2} & J_{3,3} \end{bmatrix} \begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \end{bmatrix}$$

This matrix was implemented in the python file as follows:

```
1.  # Define shorthand representation (for neat code)...
2.          c0 = np.cos(q0)
3.          c1 = np.cos(q1)
4.          c12 = np.cos(q1 + q2)
5.          s0 = np.sin(q0)
6.          s1 = np.sin(q1)
7.          s12 = np.sin(q1 + q2)
8.
9.          # Define the Jacobian (as following notes)...
10.         self.Jacobian = np.array([[-(l1*c1 + l2*c12)*s0, -(l1*s1 + l2*s12)*c0, -l2*s12*c0],
11.                                   [(l1*c1 + l2*c12)*c0, -(l1*s1 + l2*s12)*s0, -l2*s12*s0],
12.                                   [0, l1*c1 + l2*c12, l2*c12]])
```

5. Robot Control

5.1. Task H: Tuning Controller Gains

Robot control consists of using a controller to make the robot follow a desired trajectory. The robot arm in this report uses a simple position controller that ensures it reaches desired joint positions. This controller does not consider the dynamics of the robot, and only uses feedback control, based on the error of the desired positions of the joints. It is a proportional-integral-derivative controller, so it has three terms: the proportional term, the integral term, and the derivative term, abbreviated as PID. This provides a torque, τ_i, on each joint, j_i, equal to:

$$\tau_i = K_p(q_{i,d} - q_i) + K_d\left(\dot{q}_{i,d} - \dot{q}_i\right) + K_i \int_0^t (q_{i,d} - q_i)d\tau$$

Where $q_{i,d}$ and $\dot{q_{i,d}}$are the desired position and velocity of joint i, respectively. The proportional gain, K_p, is also referred to as P, the integral gain, K_i, as I, and the derivate gain, K_d, as D.

The proportional term depends directly on the difference between the desired point and the measured point, often called the error term. Usually, increasing P increases the speed of response of the control system, but when it is too large, it causes unwanted oscillations. If P keeps increasing, then so will the oscillations, causing the system to become unstable. The integral term depends on the cumulative value of the error over time, which means that even a small error term will cause the integral term to increase slowly. The purpose of this term is to reduce the steady-state error, which is the difference between the desired point and the measured point once the system has become stable. Finally, the derivative term depends on the derivative of the error with respect to time. This term is used to decrease the overshoot caused by the proportional or integral terms. Table 2, below, summarises the impact of adjusting these gains on the system.

Table 2: Comparison of impact of PID gains [3]

PID Gain	Percent Overshoot	Settling Time	Steady-State Error
Increasing P	Increases	Minimal Impact	Decreases
Increasing I	Increases	Increases	Zero steady-state error
Increasing D	Decreases	Decreases	No Impact

Tuning was performed to find the optimal PID values. This method consisted of using two techniques to provide sufficient insight into the trajectory of the robot: the first based on visual observation of the Gazebo simulation, and the second relying on plotting the error graphically using matplotlib's 'rqt_plot' function. For this task, since there was no end-effector mass acting on the robot arm, it was assumed that all three joints would require similar torque values and therefore, have the same optimal gains to follow the desired trajectory. Thus, during this tuning process, all three joints were given the same PID gain values.

To begin the tuning process, the I and D values were fixed at 0, while the P values were first increased until oscillations were observed. The P values were first changed from 0 to 10, to observe the movement of the robot with a slight increase in the gain. The robot moved to the desired positions but extremely slowly, so this gain was noted to be insufficient. This led to increasing the P values to 100, where the robot was then able to quickly reach the desired positions, now with too much overshoot, shown by large oscillations. These oscillations were also validated by the 2D error plot, seen in Figure 7.

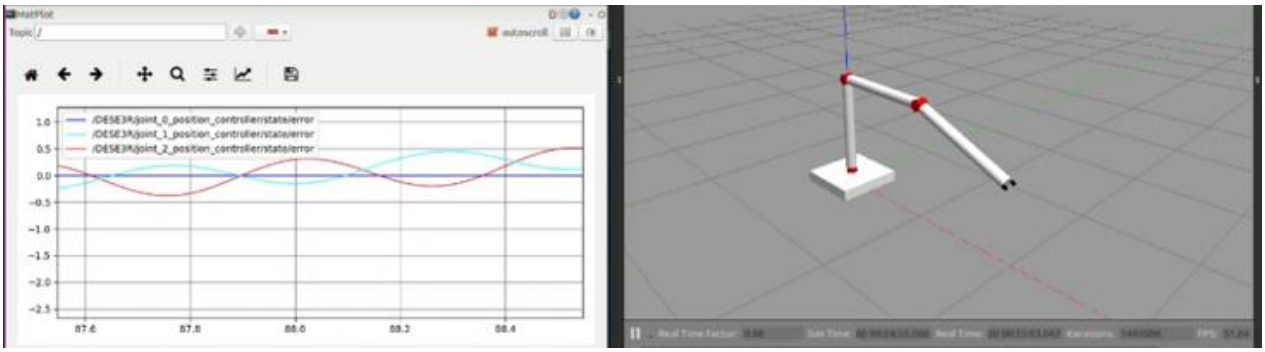


Figure 7: Capture of the robot's behaviour and the corresponding error plot, when P = 100, I = D = 0 (screenshot shows step 1 – desired position of [2,0,1])

To reduce this overshoot, the P values were halved to 50. Oscillations were still observed in the simulation and in the error plot, but further decreasing the P values to 25 led to insufficient gain, shown by the robot's slow reaction when changing positions. Thus, the P values were set at 50.

Next, the D values were tuned to reduce the oscillations and overshoot. The D values were increased from 0 to 10, which caused a slight reduction in the oscillations. These values were then gradually increased further until the overshoot was difficult to notice visually in the simulation, at around D = 50. At these values, the error plot graph still showed small oscillations in the robot's movement, and so the D values were further increased to 100. At this point, it was apparent that there was now too much differential gain, as the robot's motion was being dampened such that the arm could not reach the desired points quickly enough. Therefore, the D values were decreased back down to 75, where the robot's movement was then fast enough to reach the desired target, with very little overshoot noticeable in the simulation or the error plot. This result can be seen in Figure 8, below.

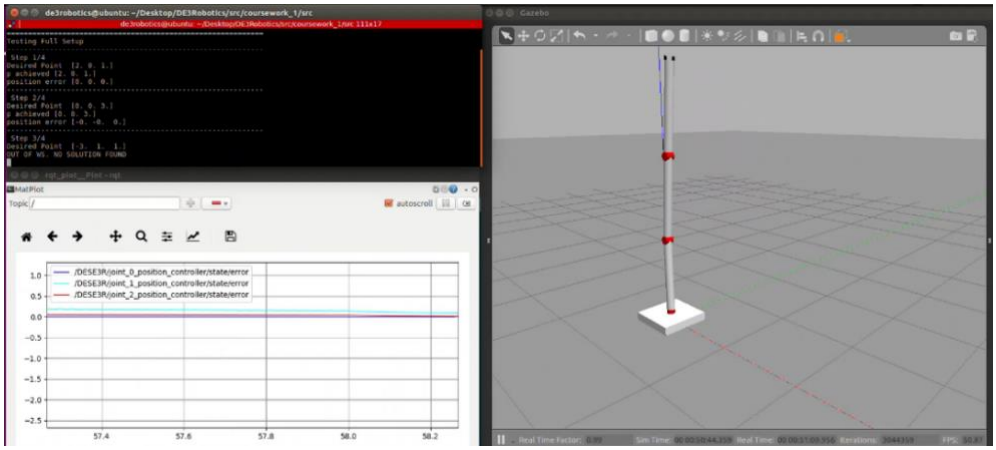


Figure 8: Capture of the robot's behaviour and the corresponding error plot, when P = 50, I = 0, D = 75 (screenshot shows step 2 – desired position of [0,0,3])

Having tuned these values, it was observed that the tracking of the robot's trajectory was still considerably slow. Therefore, the PD values were multiplied by 10, attempting to increase the reaction time. With P values of 500 and D values of 750, the robot's movement became very responsive, with no overshoot visible in the simulation and negligible overshoot observed in the graph. After trialling different values to analyse their effects (for example, doubling P to 1000, which caused the system to oscillate uncontrollably) the final values were accepted as P = 500 and D = 750. This result can be seen in Figure 9, below.

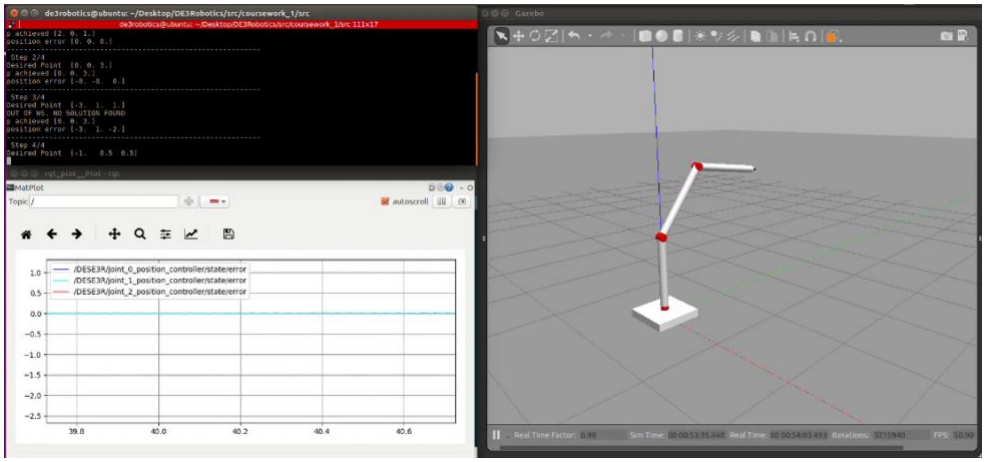


Figure 9: Capture of the robot's behaviour and the corresponding error plot, when P = 500, I = 0, D = 750 (screenshot shows movement between steps 3 and 4 – from desired position of [0,0,3] to [-1, 0.5, 0.5])

For this specific PID controller, the I values (integral gain) were irrelevant to the robot's movement because there was no steady-state error present in the model. This was due to the absence of noise or other external disturbances to the system, since this was purely a simulation, as opposed to a real-life model. This was validated by comparing the robot's behaviour when I = 0, to when I = 500, which yielded no change in the robot's movement visually or in the error plot. Therefore, the optimal gains were accepted as P values of 500, D values of 750 and I values of 0. See Table 3 for a summary of how these values changed during the tuning process, and the effects this had.

Table 3: Overview of the tuning process for task H

Kp	**K_i**	**K_d**	performance
10	0	0	insufficient torque
100	0	0	highly oscillatory
50	0	0	sufficient torque, slightly oscillatory
50	0	25	slightly oscillatory
50	0	50	less oscillatory
50	0	100	stable, but too much damping
50	0	75	stable, but slow-to-react motion
500	0	750	stable, accurate, and responsive motion

5.2. Task I: Adapting the Robot Arm

The end effector mass was edited to represent the robot arm picking up a heavy object, of mass 30 kg. This was completed by editing the file which stored the simulator's description of the robot arm, as follows:

```
1.  <!--Originally, End Effector mass = 0.01 (changed for task I)-->
2.  <xacro:property name="ee_mass" value="30"/>
```

When rerunning the Python code, this adjustment affected the behaviour of joints 1 and 2, whilst leaving joint 0 unaffected, as seen in Figure 10, below.

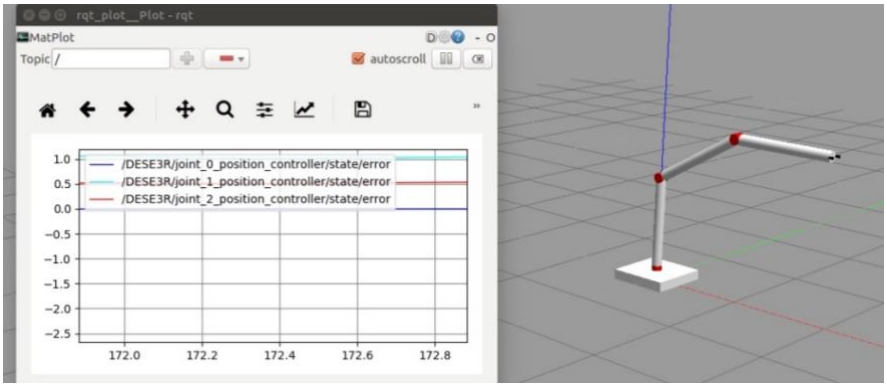


Figure 10: Capture of the robot's behaviour and the corresponding error plot, after having added the 30 kg end effector mass (screenshot shows step 2 – desired position of [0,0,3])

Given that the PID values had been tuned for the robot with an end effector (EE) mass of 0.01 kg, after adding the 30 kg mass (3000 times greater than before), it was apparent that there was now insufficient torque being provided to joints 1 and 2. This was shown by the arm not being able to reach the desired upright position, and the error plot being non-zero for joints 1 and 2.

Under gravity, the additional mass imposes a force on the end effector which acts vertically downwards only. This explains why joint 0 was unaffected, as it rotates about the vertical axis, and thus no extra torque was required, given that there was no additional force acting perpendicular to its motion. Therefore, when retuning the PID values, the controller for joint 0 would not require adjustment.

Furthermore, as the end effector is twice the distance from joint 1 as it is from joint 2 (when joint 2 is fully extended i.e., q2 = 0), it was sensible to assume that joint 1 would require twice the torque of joint 2, given that

$Moment = Force * Distance$. This theory was supported by the error plot, which showed that the error on joint 1 was double that of joint 2. Therefore, when retuning, the PID values of joint 1 would be set to double that of joint 2.

5.3. Task J: Adapting Controller Gains

Using the supporting theory from task I, and following a similar method to task H, the PID values were re-tuned to allow the robot to behave as necessary whilst supporting the 30 kg mass. For efficiency, instead of experimenting with random P values, a simple calculation was first carried out to determine the initial P values…

The simulator's description file was inspected to obtain the link lengths and masses, equal to 1.0 m and 0.5 kg respectively, for each link. Without adding the 30 kg EE mass, to hold the arms in position, as shown in Figure 11, joint 1 would require the following torque:

$$\tau = 0.5(0.5g) + 1.5(0.5g) = g$$

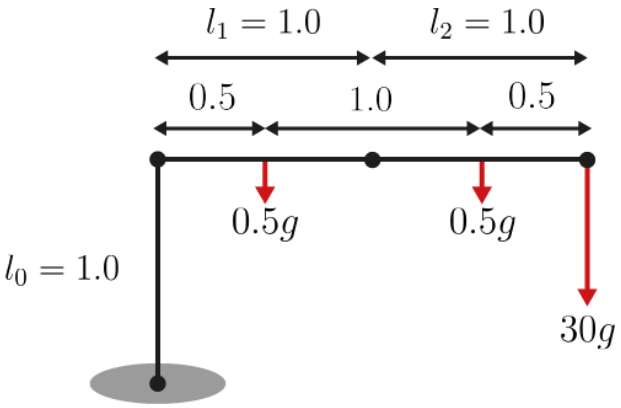


Figure 11: Diagram to support torque calculations (diagram of robot at step 1 – desired position [2, 0, 1])

With the additional 30 kg mass, the torque increases as follows:

$$\tau = 0.5(0.5g) + 1.5(0.5g) + 2(30g) = 61g$$

Thus, the required torque was estimated to be 61 times greater after adding the mass. Accordingly, the first P value to be tested was 30,500 for joint 1 (61 * previous P value, of 500), 15,000 for joint 2 (half of joint 1), and 500 for joint 0 (unchanged). At this stage, I and D values were set to 0 for joints 1 and 2.

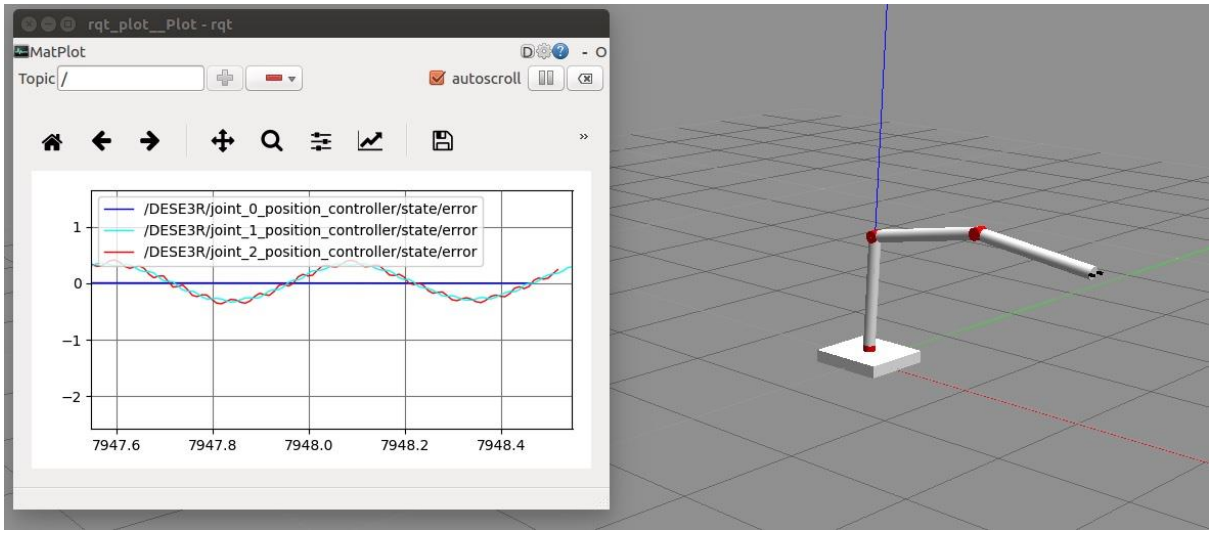


Figure 12: Capture of the robot's behaviour after only tuning the P values, for joints 1 and 2 (Screenshot shows step 1 – desired position of [2,0,1])

Having updated the P values, it appeared as though the joints had sufficient torque to reach the desired positions. However, as shown in the error plot, the joints were highly oscillatory. Alternative P values were also trialled: doubling the P values caused more severe oscillations, which yielded no benefits as the arms could already reach desired position; whilst halving the P values reduced oscillations but meant that the arms had insufficient torque to reach desired position. Therefore, the initial values were accepted, and D values were then adjusted to eliminate oscillations and overshoot.

The D values were first trialled with the same ratio as seen in task H (1.5 times the P values). However, owing to the much larger torque values involved with this task, these values caused the system to become extremely unstable. Therefore, the D values were then tuned using the same method as seen in task H: using small initial values and doubling these values until the system became stable. Given that the torque on joint 1 was double that

of joint 2, the D value for joint 1 was also set to twice that of joint 2. The final values for D were 2600 for joint 1, and 1300 for joint 2, which provided very stable motion, with little overshoot, as seen in Figure 13, below.

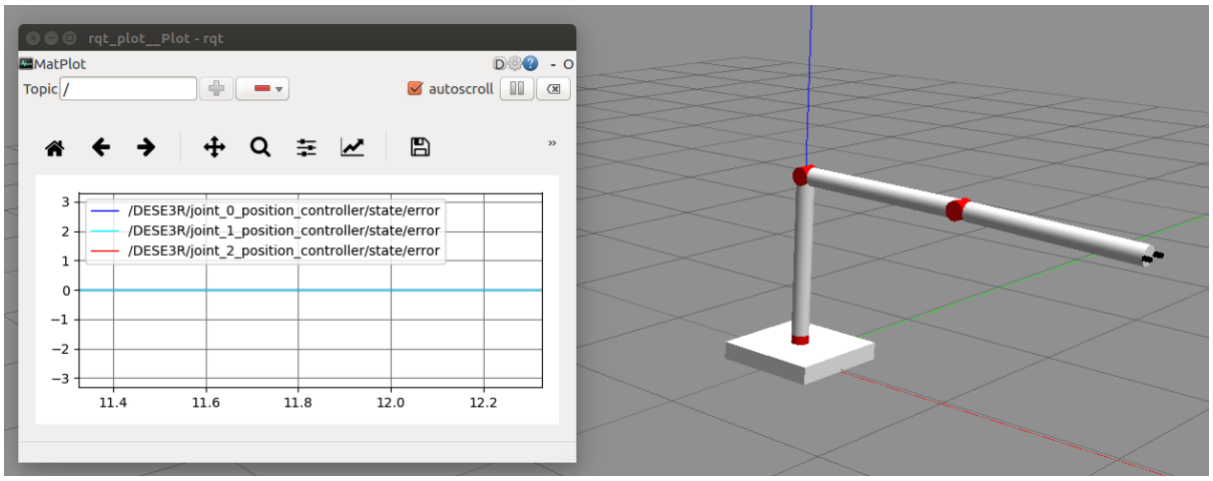


Figure 13: Capture of the robot's behaviour after final tuning (screenshot shows step 1 – desired position of [2,0,1])

Similar to task H, tuning the I values had no effect on the system's performance. This was due to the robot being purely a simulation, and thus, no unexpected external forces or perturbation were acting, which could have caused steady-state (offset) errors. Therefore, the I values were set to 0 for all joints, as before. See Table 4 for a summary of how these values changed during the tuning process, and the effects this had.

Table 4: Overview of the tuning process for task J

K_{p0}	K_{i0}	K_{d0}	K_{p1}	K_{i1}	K_{d1}	K_{p2}	K_{i2}	K_{d2}	**Performance**
500	0	750	500	0	750	500	0	750	Insufficient torque to reach desired positions
500	0	750	30,500	0	0	15,000	0	0	Sufficient torque but highly oscillatory
500	0	750	30,500	0	45,000	15,000	0	22,500	Extremely unstable
500	0	750	30,500	0	2,600	15,000	0	1,300	Stable and accurate motion

It should be noted that after completing the tuning process, the motion was very stable with little error, but was not perfect. For this task, the additional weight on the EE meant that the PID tuning was a lot more sensitive, and thus, more difficult to adjust. Furthermore, as stated on the tutorial sheet, the dynamics of the robot have not been considered in the control method (i.e., no feedforward control) and instead relies on linear feedback control only. Therefore, at the beginning of the trajectory and at discontinuities, such as when the arms stop or change direction, slightly greater error is noticeable. This was likely the case in task H also but, given that there was less weight acting on the arms, this error was not as apparent. To handle these errors, the dynamics of the robot should be modelled, and feedforward control should be implemented into the control code.

References:

[1] *Petar Kormushev. COURSEWORK 1: MODEL. Imperial College London, Dyson School of Design Engineering.*

[2] *Thrishantha Nanayakkara. ROBOTICS: 3D Kinematics Lecture Notes. Imperial College London, Dyson School of Design Engineering.*

[3] *Peter Cheung. ELECTRONICS II: PID Controller. Imperial College London, Dyson School of Design Engineering*

Applied Robotics Coursework 2 – Plan

James Skinner (01700960), Madelaine Wood (01762829),
Harry Schlote (01746509), Fernanda Espinoza (01783661)

Table of Contents

1. Introduction

This report documents the development of 'Coursework 2 – Plan', as part of the Applied Robotics module. A well-implemented motion planning algorithm is essential for a robot to behave intelligently, and this report covers three different implementations: adding waypoints by hand, the potential field algorithm, and the probabilistic road map algorithm.

The motion planning methods will be applied to a virtual representation of Design Engineering's Natural Interaction Robot (Robot DE NIRO), which consists of a dual-arm robot (each with seven degrees of freedom) mounted onto a mobile wheelbase, see Figure 1. Motion planning will be applied to operate the wheels, whilst the arms will be tucked in to avoid unnecessary collisions with the environment.

Figure 1: Real-life view of Robot DE NIRO

2. C-Space Map

2.1. Task A: C-Space dilation

Before motion planning algorithms can be applied, the configuration space (C-space) needs to be setup to accurately represent the workspace. The C-space is an imaginary co-ordinate system where the agent's configuration can be represented as a single point and is useful as it simplifies calculations required for motion planning (easier and quicker to check for collisions).

C-space dilation involves expanding the original map, see Figure 2, based on a mask of the robot. This, effectively, inflates the obstacles to account for the dimensions of the robot, allowing the agent to be represented as a single point. It should be noted that the map is scaled such that 1 m is 16 px wide, and the centre of the world, co-ordinates (0, 0) m, are located at the centre of the map.

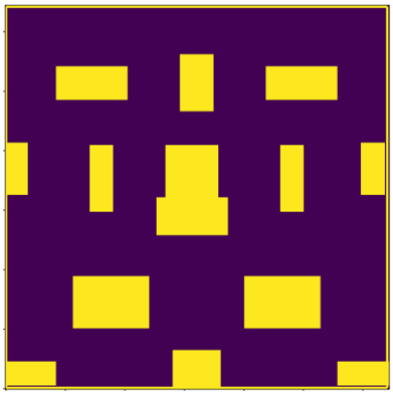


Figure 2: The original map of the workspace

For part i, a simple square mask was used to represent DE NIRO. This robot is 1.0 m wide, and thus the pixel mask consisted of a 16x16 array of ones. The code to create this mask was trivial:

```
1.  ################################################################ TASK A, part i
2.      # SQUARE MASK
3.      # Create the 16x16 matrix of 1s...
4.      robot_mask = np.ones((16, 16))
```

This mask was then applied to generate the dilated C-space, see Figure 3. As can be seen, the obstacles and borders were inflated in both x and y directions by a distance of 8 pixels. This was because the shape of the inflated obstacles is dependent on both the agent's origin location as well as its shape, and given that the agent's origin was at the centre of the square mask, the obstacles were inflated by half the length of each side of the mask. Note that the corners of the dilated C-space were still squared, owing to the use of the square mask.

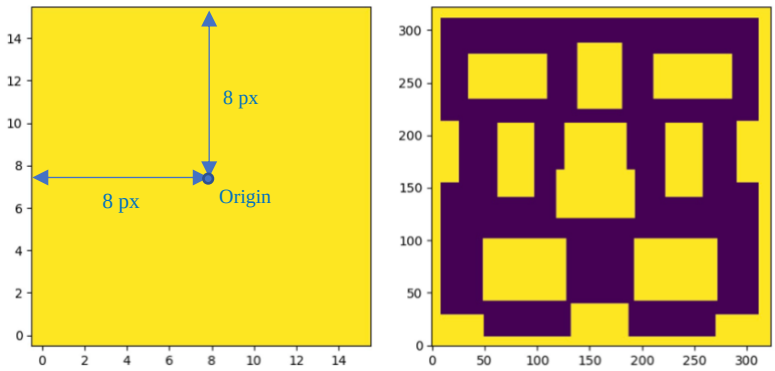


Figure 3: The square mask (left) and its effect on the C-space (right) – units in terms of pixels

For part ii, a circular mask was used to provide a slightly more accurate representation of DE NIRO. Given the width of the robot, this circular mask had a radius of 8 pixels. To create this mask, a square mask of ones was first created, before a 'meta-mask' (containing True or False, depending on whether the given pixel was outside or inside the circle) was applied to it, transforming it into a circular mask. See the commented code, below, for further details.

```
################################################################ TASK A, part ii
    # CIRCULAR MASK - optional for individual students
    # create a square array of the size of the robot
    # ...where a circle the size of the robot is filled with ones

    # Create the 16x16 matrix of 1s...
    robot_mask = np.ones((16, 16))

    # We will create a 'meta-mask' (to apply to our 16x16 matrix mask), using algebra...
    # Generate values that represent co-ordinates (-7.5 to 7.5 in both x and y)
    # Note: 7.5+1 because of non-inclusive upper indexing
    y, x = np.ogrid[-7.5: 7.5+1, -7.5: 7.5+1]
    # Create a mask matrix, using equation of a circle, and checking whether co-ordinates...
    # ...are inside (<= radius^2) or outside (>=) the circle (centre at (0, 0))
    # If outside the circle, value within the mask is set to true
    mask = x**2+y**2 > 8**2

    # Apply mask to square matrix - positions outside the cicle set to 0
    robot_mask[mask] = 0

    # Show the new robot mask
    plt.imshow(robot_mask, vmin=0, vmax=1, origin='lower')
    plt.show()
    # Dilate the C-space (obstacle inflation)
    expanded_img = binary_dilation(img, robot_mask)
```

After applying the mask, see Figure 4, the obstacles were again inflated by 8 pixels in all directions. However, the advantage now was that the corners of the obstacles also had a curved radius, as opposed to being squared-off. This allows the agent to get closer to the corners of obstacles, where the algorithms would otherwise (when using the square mask dilation) 'detect' a collision, even though the agent was not colliding with an obstacle in the workspace. Therefore, this provided a slightly more accurate C-space representation.

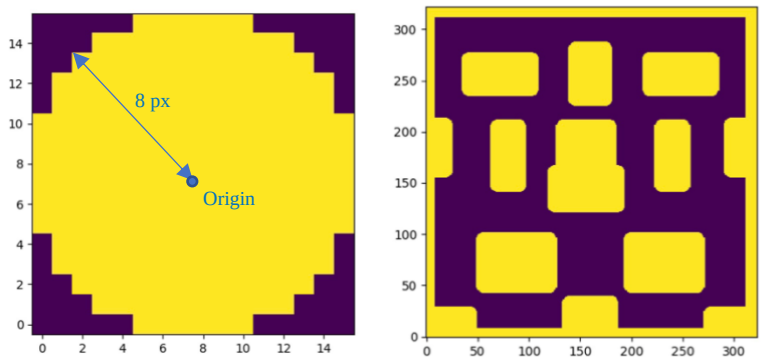


Figure 4: The circular mask (left) and its effect on the C-space (right) – units in terms of pixels

3. Waypoint Navigation

3.1. Task B: Adding waypoints by hand

For this report, all motion planning algorithms should enable DE NIRO to travel from a start point of (0.0, -6.0) m to the goal position of (8.0, 8.0) m, as seen in Figure 5.

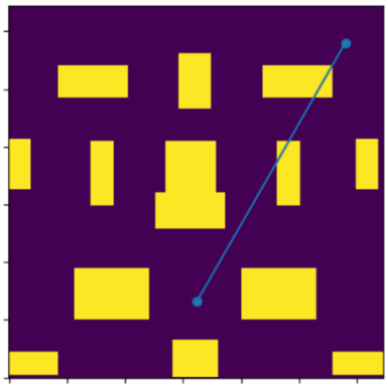


Figure 5: The desired start and end co-ordinates of DE NIRO and the straight-line path between them

The use of manually inputting waypoints is a simplistic and non-optimal motion planning method. Due to the obstacles present in the environment, a straight-line path was not possible, and therefore extra waypoints were required. As seen in the code below, these waypoints were added in terms of world coordinates, as opposed to pixel coordinates.

```
1.  ################################################################ TASK B
2.          # Create an array of waypoints for the robot to navigate via to reach the goal
3.          # *Note: Start/End co-ords are not required - they are added later
4.          waypoints = np.array([[1.2, -3.9],
5.                                [7.4, -0.5]])  # fill this in with your waypoints
6.
```

The coordinates were chosen by examining the dilated map from task A, using the given pixel scale (from 0 to 323.2 pixels on each axis), to select points that appeared most direct to the goal at (8.0, 8.0). Given that the map's scales were in terms of pixels, where 1 m = 16 pixels, pixel coordinates were scaled to world coordinates via:

$$D_W = D_P * 0.0625$$

In terms of world coordinates, the map ranged from -10.1 m to 10.1 m in both x and y directions, with (0, 0) at the centre of the map. In pixel coordinates, this corresponded to 323.2 pixels wide in both x and y directions, with (0, 0) at the bottom left of the map. Thus, to convert from pixel space to world space, the values were translated and scaled using the formula:

$$P_W = (P_P * 0.0625) - 10.1$$

The initial set of coordinates, see Figure 6, used only horizontal and vertical movements (i.e., no diagonal movement), providing a simple route to goal without collision. After inspecting the map and applying the pixel-to-world transformation, the world coordinates input into the code were: [[0.0, -3.37], [2.72, -3.37], [2.72, 8]]. It should be noted that the start and end coordinates did not need inputting into this array, as these were appended later in the code.

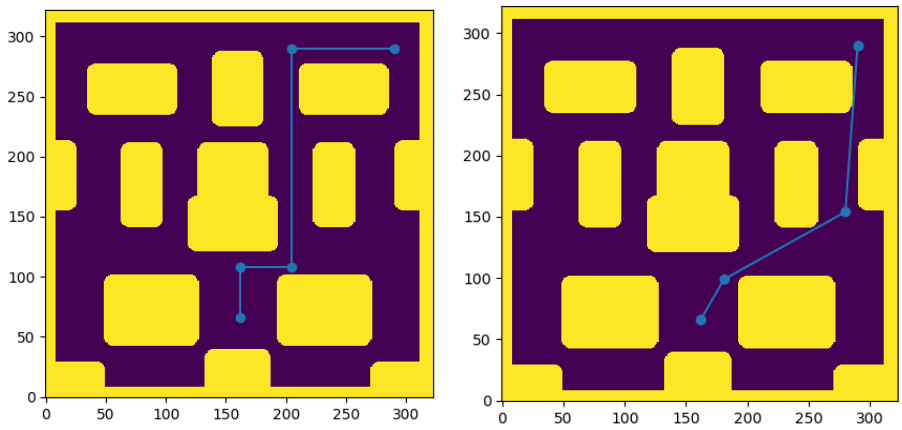


Figure 6: A comparison of two different sets of waypoints, with the diagonal lines (right) providing a more direct route to goal

This provided a total path length for the initial path equal to 22 m. The strategy to then find the shortest path was simply by eyeballing the map and attempting to identify the 'most straight line' solution from the start to the goal, with the least severe changes in direction. Furthermore, by minimising the amount and size of turns, this improved the time to reach the goal, as the robot was typically very slow in changing directions. This yielded a shorter, but still non-optimal distance, with a world coordinate set of [[1.2, -3.9], [7.4, -0.5]], as seen in the right of Figure 6. To calculate this total path length, D, Pythagoras' theorem was used to calculate the distance between each waypoint, before summing these distances to give the total length, as follows:

$$D = \sum_{N=1}^{N} \sqrt{(X_N - X_{N-1})^2 + (Y_N - Y_{N-1})^2}$$

This gave a more optimal path length of:

$$D = \left(\sqrt{(1.2-0)^2+(-3.9--6)^2}\right) + \left(\sqrt{(7.4-1.2)^2+(-0.5--3.9)^2}\right) + \left(\sqrt{(8-7.4)^2+(8--0.5)^2}\right) = 18.01\,m$$

4. Potential Field Algorithm

4.1. Task C: Implementing the potential field algorithm

The potential field algorithm is a reactive motion planning method that works by simulating the agent as a charged particle, which is attracted to the goal and repelled by obstacles, as visualised in Figure 7. In this case, the agent was a negatively charged particle, with the goal positively charged and the obstacles negatively charged.

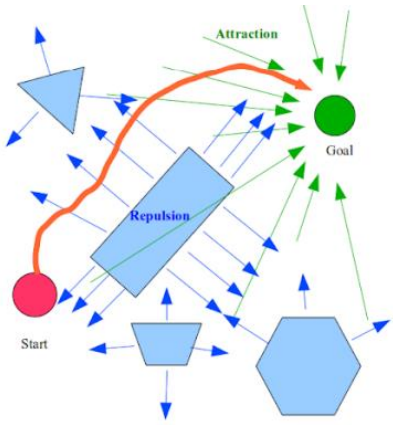


Figure 7: Potential field algorithm visualisation

There are many strategies to calculate the attractive force acting on the agent, one simple method being to make it inversely proportional to the agent's distance to goal, where a similar method can be used to calculate the repulsive forces from each obstacle. However, this is difficult for DE NIRO as it starts quite far away from the goal with many obstacles in between, making this strategy problematic and not as effective. In this case, it can be beneficial to instead impose a constant attractive potential force towards the goal, as follows:

$$f_{att} = K_{att}\hat{x}_g \quad (1)$$

$$f_{rep} = -K_{rep}\frac{1}{N}\sum_{i=1}^{N}\hat{x}_i / d_i \quad (2)$$

Where K represents the constant of attraction/repulsion to be tuned, $\hat{x}$ is the unit vector from the robot to each obstacle/goal, and d is the distance from the robot to each obstacle. For equation 1, the force magnitude was set to 1, which provided the constant attractive force. This was implemented in the code as follows:

```
1.          pos_force_magnitude = 1     # Task asks for constant attractive potential (thus = 1)
2.          # positive force
3.          positive_force = K_att * pos_force_direction * pos_force_magnitude  # normalised
    positive force
4.
```

For equation 2, the force_magnitude was set as negative, inversely proportional to the distance. This ensured that it was repulsive and imposed a force in the opposite direction to move the robot away from the obstacle. Furthermore, as the robot comes closer to the obstacle, the repulsive force becomes more intense. This was implemented in the code as follows:

```
1.  force_magnitude = -1/distance_to_obstacle     # Task asks for inversely proportional
    repulsive potential (= 1/d)
2.  # total negative force on DE NIRO
3.          negative_force = K_rep * np.sum(obstacle_force, axis=0) /
    obstacle_pixel_locations.shape[0]
```

Several different values for the repulsive and attractive forces were trialed, and the resulting path which the robot followed was then plotted on the map, a few examples of which can be seen in Figure 8. However, even after extensive tuning, DE NIRO was not able to successfully reach the end goal, using the constant attractive force and inversely proportional repulsive forces. This was due to the use of linear equations only, meaning that the repulsive forces from all points within the space had similar weights. As such, there were insufficient repulsive forces when the agent was near obstacles, to prevent collisions.

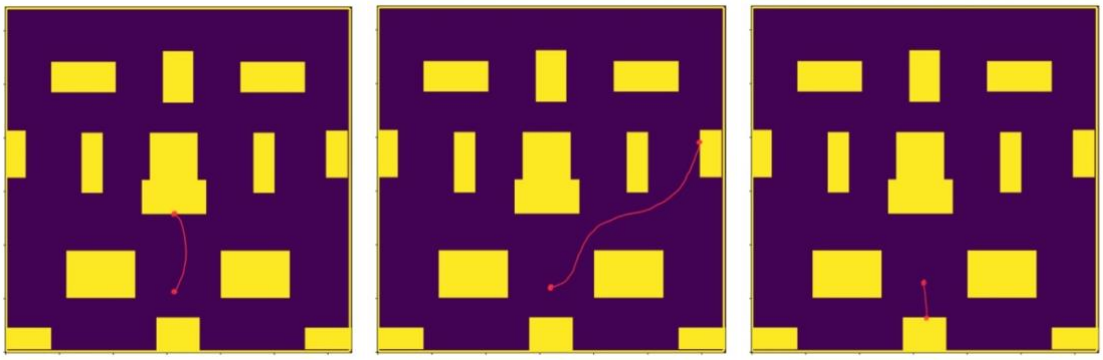


Figure 8: Paths followed during tuning – att=1,rep=100 (left), att=1,rep=15 (middle), att=1,rep=50 (right)

Using the default parameter values (left of Figure 8), since the attractive force was constant and the goal was in the top right corner, the path was seen to initially curve to the right. However, there was not enough attraction to the goal and too much repulsion from the obstacle on the right-hand side, causing DE NIRO to curve away from the goal and into another obstacle. The repulsion was therefore lowered to 15 (middle of Figure 8), which was sufficient to improve upon the original trial, but did not provide enough repulsion to prevent a collision with the obstacle on the right edge of the map. The repulsion was thus increased to 50, as seen on the right of Figure 8. However, this again provided too much repulsion, shown by the agent being pushed backwards, with no attractive movement towards the goal detectable. Furthermore, at certain parameter values, the agent often became stuck at local minima, where the attractive and repulsive forces cancelled out one another, yielding no resultant motion.

As a result of initial trials, it was concluded that the linear methods used to calculate the forces were too simple, and the algorithm would require more complex potential field equations to achieve better performance. Potential fields in nature are often quadratic, such as Electric Fields, which follow the equation: $F = K * Q_1 Q_2 / r^2$. By making the force inversely proportional to the square of the distance, this amplifies the forces being imposed upon the agent, see Figure 9, greatly reducing the chance that the agent will collide with an obstacle. When the agent gets close to an obstacle (< 1 m), the repulsive forces become much greater. Similarly, when the agent is further away from the obstacles (> 1 m), the repulsive forces become much smaller. These quadratic potential functions follow equation 3, and were implemented in the code as follows:

```
1.  # For Part ii, trialling a quadratic potential function...
2.          pos_force_magnitude = 1/(distance_to_goal**2)
```

$$f_{rep} = -K_{rep} \frac{1}{N} \sum_{i=1}^{N} \hat{x}_i / {d_i}^2 \quad (3)$$

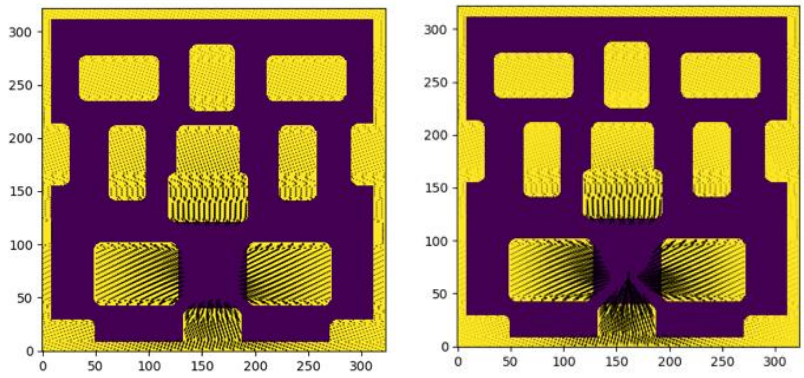


Figure 9: A visual comparison of the imposed forces using linear (left) vs quadratic (right) potential fields (default tuning used)

The constants of attraction/repulsion then had to be tuned again. The default parameters used a high repulsive force, and as such there was insufficient attraction to goal, causing DE NIRO to get stuck within a local minimum, as seen in the left of Figure 10. To deal with this, the repulsion was decreased to 15. However, this yielded little improvement and a very similar movement occurred. Thus, the attractive force was increased to 50, to 'pull' the agent away from this minimum. As can be seen, there was then slightly too much attraction to the goal when the robot was near it, causing the agent to collide with the obstacle that was closest to the goal. This highlighted that a little more repulsion was needed to stop this from happening. Finally, the repulsion was increased to a value of 35, with the attractive force remaining at 50. This allowed DE NIRO to successfully reach goal.

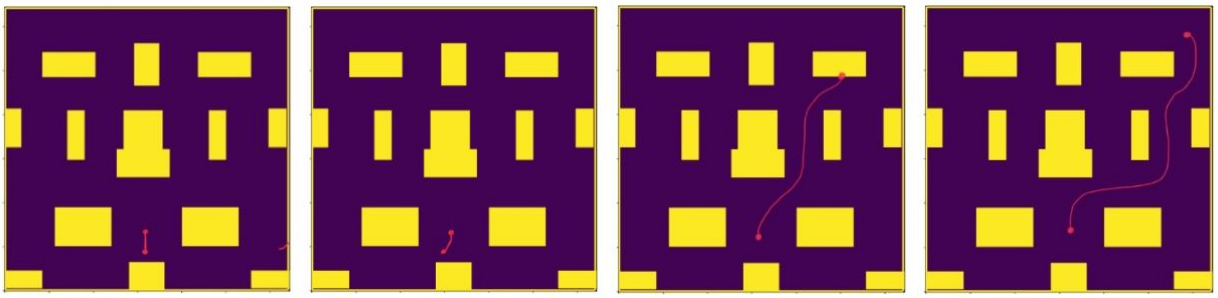


Figure 10: Paths followed during tuning – att=1,rep=100 (1st), att=1,rep=15 (2nd), att=50,rep=15 (3rd), att=50,rep=35 (4th)

It should be noted that, as highlighted in the right of Figure 10, this final path was not the optimal path that could be taken to reach the goal. This is because the potential field algorithm path tends to oscillate and change direction frequently, meaning that the agent moves very indirectly, and thus an optimal solution can never be obtained. Owing to this, when this path is compared to the manually plotted path from task B, it is clear that the oscillations caused by the potential field algorithm produce a worse path, in terms of length.

Although the potential field algorithm cannot produce optimal paths, it does provide several advantages because it is reactive. The main benefit of reactive motion planners is that they do not need to know the full environment or calculate a complete path before the agent begins to move. Therefore, if obstacles are added during the agent's motion, it can react accordingly and dynamically adjust its path on the go. Offline algorithms, such as the probabilistic roadmap (PRM), cannot do this and instead requires all the information of the environment beforehand. Thus, if an obstacle is added during motion, the PRM method must be restarted, and a new graph should be constructed. On the other hand, disadvantages of reactive motion planning (as well as yielding non-optimal solutions) are that they often require a lot of time and effort to properly tune, and even when tuned they may not necessarily be 'complete', in the sense that a solution may exist which they fail to find.

5. Probabilistic Road Map

5.1. Task D: Randomly sampling from the map

Task D aims to implement the first step of the PRM algorithm, which involved randomly sampling points on a map of the robot's environment, generating N_{points} samples that do not collide with obstacles. This was implemented in code as follows:

```
            # Loop through the generated points and check if their pixel location corresponds to an obstacle in self.pixel_map…

            # Iterate through all newly generated points…
            # Initialise index (Stores index of current point being iterated)
            idx = 0
            # Loop through all points generated (in pixel co-ords) - extract x and y values
            for x_point, y_point in pixel_points:
                # Check if point inside object (== 1) - note that indices are y, x (row, col), and make sure to convert to integer values (floating point not accepted)
                if self.pixel_map[int(y_point), int(x_point)] == 1:
                    # If inside object, set flag of that point to rejected
                    rejected[idx] = 1
                # Increment index
                idx += 1
```

The points in the map had already been randomly generated, providing a uniform sampling distribution. Thus, the above code simply iterated through these points, and checked whether each point corresponded to a pixel in the pixel_map of 1 (indicating an obstacle) or a 0 (indicating free space). If a point was within an obstacle, its corresponding entry in the 'rejected' array was set to 1, and these points were then deleted. Further random points were then generated and checked, and this cycle continued until N_{points} samples had been accepted. A plot showing the accepted and rejected points has been included, see Figure 11.

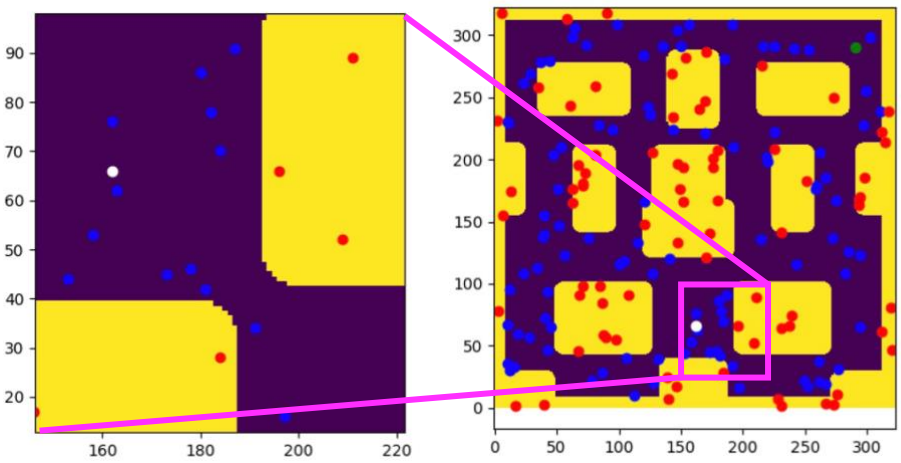


Figure 11: A visualisation of the uniform point sampling (with zoomed-in view) - rejected points in red, accepted in blue

Inspecting the code for generating samples showed that uniform sampling has been used to generate the initial random points. However, this might not be the best strategy to deal with regions of high obstacle density, and often, non-uniform methods are superior. One useful method could be to implement 'obstacle based' sampling, such that if the regions of obstacles are known in advance, sampling can just avoid these regions. Alternatively, samples could be chosen at random as before, but instead of rejecting those that lie within an obstacle, simply pick a random direction, and move the sample in that direction until it becomes free space. This will help to add more samples in close proximity to obstacles, which is very useful for regions of high obstacle density.

Another strategy could be to use Gaussian sampling, which has been shown to be beneficial for gathering more samples closer to obstacles, and less samples in wide-open space, leading to a favourable sampling distribution. This is implemented by generating two random samples, where the distance between them is chosen according to a Gaussian distribution, and only adding a sample if one of the configurations is in free space, while the other is within an obstacle. A pseudo-code representation [1] of how this might be implemented, where Cfree is a coordinate in free space (no obstacle) and Cforb in forbidden space (within an obstacle), is as follows:

```
loop
    c1 ← a random configuration
    d ← a distance chosen according to gaussian distribution
    c2 ← a random configuration at distance d from c1
    if c1 in Cfree and c2 in Cforb:
        add c1 to the graph
    else if c2 in Cfree and c1 in Cforb:
        add c2 to the graph
    else:
        discard both
```

5.2. Task E: Creating the graph

Having sampled the points in task D, a graph then had to be generated to connect these points as nodes. As opposed to connecting each point to every other point, which would have been computationally expensive and time-consuming, edges (connections between nodes) were only added if they were between a minimum and maximum distance. If nodes are far apart, it is likely that there would be an obstacle between them. Conversely, if nodes are very close, then there's no benefit in travelling between them.

The algorithm was first run using default values, with mindist = 0 m and maxdist = 20 m. This took a long time to compute and yielded an extremely dense graph, highlighting that maxdist was too high and mindist was too low. The value of maxdist was then continually decreased until many of the edges passing through obstacles were removed, but with enough connected edges remaining elsewhere to allow multiple routes for the agent to traverse, at around maxdist = 5 m. Similarly, mindist was gradually increased until about 1 m, whereby an overly dense graph was avoided, and the distance remained small enough to avoid unnecessarily disconnected nodes. See Figure 12, below, for a comparison of the graphs created using the default and final parameters.

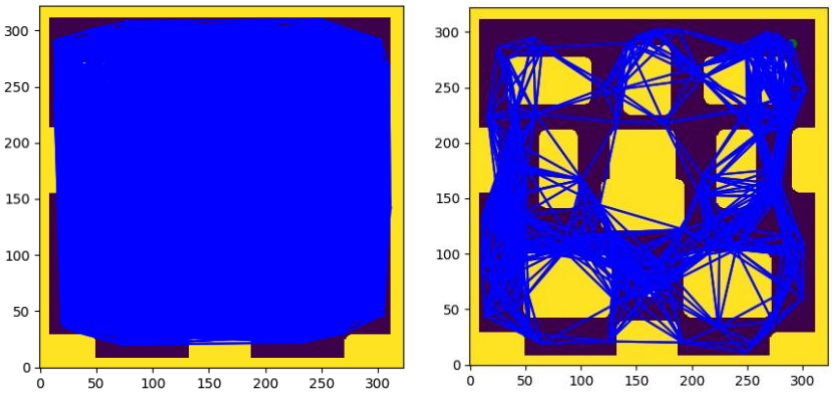


Figure 12: The graphs created using default (left) and final (right) parameters

Maxdist was further reduced, but this caused too many disconnections within the graph, which would have led to poor solutions (or even no solution) when attempting to find an optimal path. Similarly, mindist was further increased, and this also caused unnecessary disconnections between nearby nodes, again leading to worse solutions. See Figure 13, below, for a visualisation of these results.

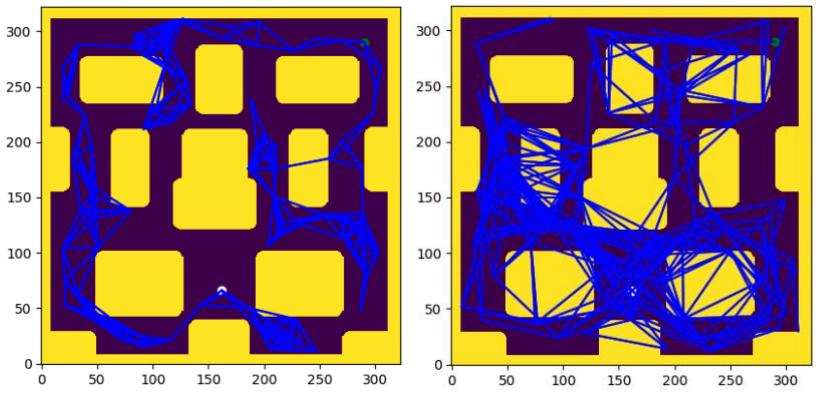


Figure 13: Graphs highlighting the effects of maxdist set too small (left), and mindist set too high (right)

Each created edge then had to be checked for collisions. For this, points along each edge were sampled at regular intervals (according to the resolution), checking whether any point along the given edge was inside an obstacle. To calculate this, the unit direction vector from A to B was required, such that the first point checked would be at node A, and subsequent points could then be computed by adding the product of the unit vector and the resolution. The calculation of the unit vector was carried out using the following code:

```
1.  ################################################################ TASK E ii
2.          # Calculate horizontal and vertical distance between points (B - A gives direction
    from A to B)
3.          dx = pointB[0] - pointA[0]
4.          dy = pointB[1] - pointA[1]
5.          # Apply pythag theorem to find magnitude
6.          distance = np.power((np.power(dx, 2) + np.power(dy, 2)), 0.5)
7.          # Calculate the UNIT direction vector pointing from pointA to pointB
8.          # Unit vector is calculated by dividing each value by magnitude...
9.          direction = np.array([dx/distance, dy/distance])
```

The resolution then had to be tuned; fine (low value) enough to ensure that no accepted edges were in collision with obstacles, but not too fine that it would lead to unnecessarily expensive computation. Given that each pixel on the map represented 1/16 = 0.0625 m, the resolution was set to 0.0625 m. This resulted in successful collision detection and edge removal, and further refining the mesh to less than a pixel's width would not have improved the results, wasting computational time and power. See Figure 14 for partial refinement, and Figure 15 for the results of the final tuned resolution.

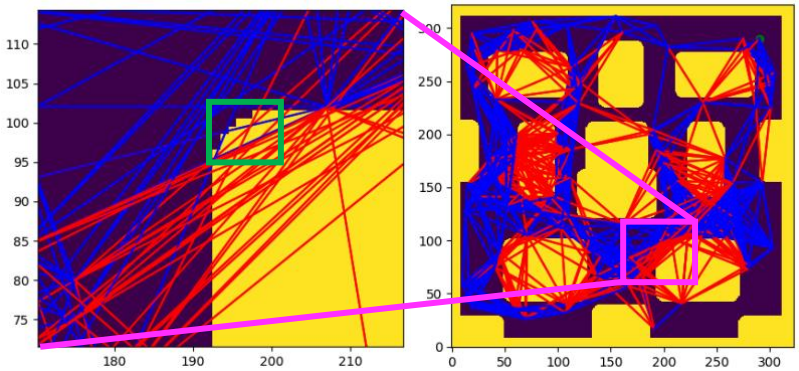


Figure 14: Resolution at 1 m – insufficiently refined and non-removed colliding edges near corners (highlighted by green box)

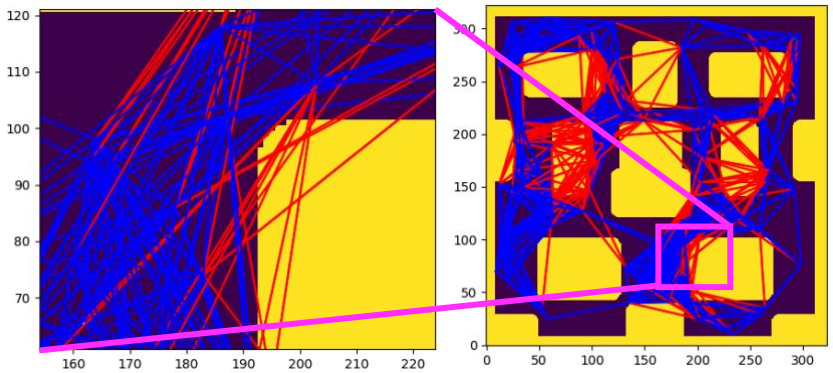


Figure 15: Resolution refined to 0.0625 m – sufficiently refined and all colliding edges successfully removed

A problem occurs when there aren't enough edges to make a complete graph from the initial node to the goal node. To avoid the chances of this occurring, a few different methods could be implemented.

Firstly, 'two-phase connectivity expansion' could be used. This would involve uniformly sampling the design space as before, but when connecting nodes to form edges, store a count of how many times each node fails to connect (i.e., its edge lies within an obstacle). This can then be used to identify the nodes which most frequently fail to connect to other nodes. A second phase of sampling can then be carried out, such that more configurations within the design space can be sampled (non-uniformly) in close proximity to these failing nodes, allowing more useful edges to be formed.

Another similar method would involve identifying isolated nodes which only connect to one other node, as these form edges that would be pointless to traverse. Again, a second phase of sampling nearby these nodes could then be undertaken, allowing more useful edges to be formed and achieving a better-connected graph.

5.3. Task F: Dijkstra's algorithm

Once the graph had been created, the optimal (shortest) path could then be calculated. This was achieved by implementing Dijkstra's algorithm, an overview of which can be seen in Appendix A [2], followed by the main code loop in Appendix B. Once Dijkstra's algorithm has been executed, the goal node row should correspond to the optimal distance and path that DE NIRO will follow. These points can then be used for waypoint navigation.

For this task, first, a suitable 'intial_cost' had to be chosen. This was set to 1000 m, which is much larger than the maximum straight-line distance possible for the given map (28.3 m). It was set significantly higher than this 28.3 m distance as the robot could take a very indirect route due to obstacles, as seen in Figure 16. This large initial cost ensured that Dijkstra's algorithm would work correctly, such that when comparing a newly calculated path to a previous path, the initial cost would not be accepted as the shortest route to the given node.

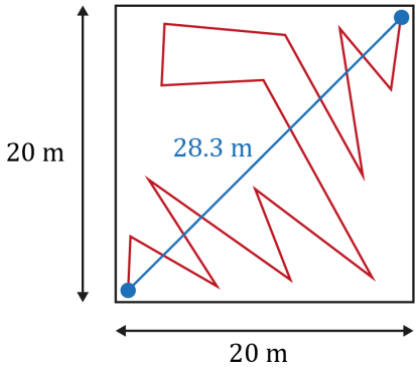


Figure 16: Visualisation of the maximum straight-line distance possible (in blue), and an even longer indirect route (in red)

Before running the algorithm, the final requirement was to calculate 'next_cost_trial'. This is the sum of the current_cost and edge_cost, as it is simply the distance from the starting note to the current node, plus the distance from that current node to the next potential node. This was implemented in the code as follows:

```
1.  #next_cost_trial = 1234  # set this to calculate the cost of going from the initial node to
    the next node via the current node
2.                      next_cost_trial = current_cost + edge_cost
```

When the complete PRM algorithm was executed for the first time, the calculated path appeared far from optimal, with sharp corners and zigzagging present. As highlighted in Figure 17, it could have been straightened to produce a more direct, shorter route to goal. While Dijkstra's algorithm will find the shortest (most optimal) path for its specified graph, PRM itself is only 'probabilistically optimal'. This means that a truly optimal solution can only be obtained if the sampling and graph building algorithms are run for long enough, such that there are nodes and edges that form the perfectly optimal path. This would require an extremely high number of samples, and is not realistically feasible, as it would require too much computational time and power.

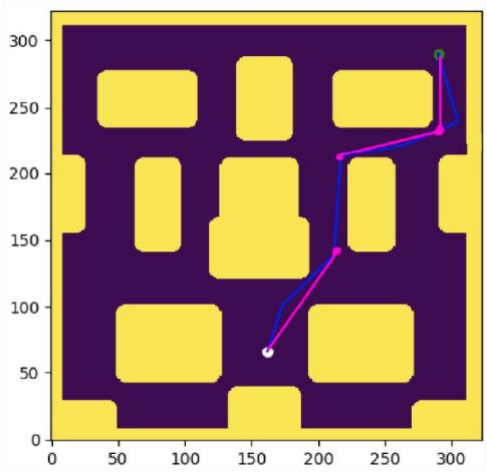


Figure 17: The initial path calculated from PRM, where the pink lines highlight how the path could be made more optimal

The second time the algorithm was run, it produced a path that looked very similar to the manually inputted waypoints from task B. Owing to PRM's probabilistic nature, it was concluded that the more sample points taken, the more optimal the path is likely to be, although this comes at the expense of computation time. PRM is clearly advantageous though over manually inputting waypoints, as once set up, any goal point can be selected, and a close-to-optimal route will be automatically calculated in a reasonable time.

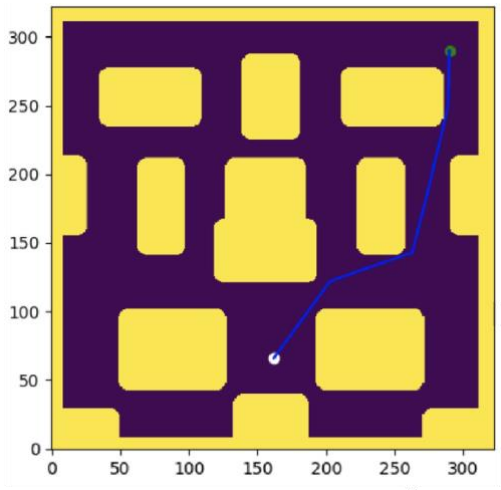


Figure 18: The second path calculated from PRM

Dijkstra's algorithm is known as a one-to-many planning algorithm, which means that it finds the optimal route from a starting node to all other nodes in a graph (if run until completion). This is advantageous as to find the route from the starting node to a new goal, it is not necessary to recalculate all the shortest paths. Instead, the distance of the new goal to its nearest node can be calculated, and this can then be added to the shortest path distance to that node, as previously calculated when running Dijkstra's algorithm to completion.

PRM itself is suitable for multi-query use on the same planning problem. This is because once the graph has been formed, it can be used over and over again. To perform another query, we simple remove the old start and end goal, and add the new ones, connecting them to the neighbouring nodes in the graph. This saves computation power and time, as we don't have to build a new graph for every query. As previously explained, Dijkstra's algorithm would not need to be re-run either, as long as the starting point remains the same, for a different goal node. However, if the starting point is changed, then Dijkstra's algorithm would have to be recalculated. Therefore, this is a key drawback of one-to-many algorithms, as it is computationally wasteful to calculate optimal distances to all nodes, if only the distance from one node to another is desired.

An algorithm that is more suitable for one-to-one planning is the RRT-connect algorithm (where RRT stands for rapidly exploring random tree). This is one-to-one as it builds a tree outward from the goal as well as the start point, finding just a single route between these two points. Normal RRT (without connect) is more similar to Dijkstra's, such that as long as the start node is not changed, it can be reused to find a new goal. However, this is not the case when implementing RRT-connect, and the advantage of using this one-to-one version is that it is faster and more efficient than one-to-many algorithms if we only want to find the path to one single point. This is because it searches more directly towards the desired goal, as opposed to wasting computation and time in searching other directions that we are less interested in. The drawback of this is that if we want to find the path to a new goal, the entire algorithm must be re-executed.

References:

[1] *Boor V, Overmars M, Van der Stappen F. The Gaussian Sampling Strategy for Probabilistic Roadmap Planners. In 1999.*

[2] *Petar Kormushev. COURSEWORK 2: PLAN. Imperial College London, Dyson School of Design Engineering.*

Appendix

A: Dijkstra's algorithm overview [2]

1. Firstly, in the motion planning code, a dataframe should be created which contains every node in the graph. This should include a cost heading, which tells us the total distance travelled to reach this node, and a 'previous' heading, indicating the path taken to reach that node, as seen in Table 1, below.

Table 1: The initial unvisited dataframe

Node	Cost	Previous
$[x_{init}, y_{init}]$	0.0	''
$[x_1, y_1]$	$1e6$	''
$[x_2, y_2]$	$1e6$	''
...	...	...
$[x_N, y_N]$	$1e6$	''

The cost of every node should initially be set to a very high value, except the initial node, which is given a cost of 0.

A second 'visited' dataframe should also be created that is initially empty, and as the algorithm runs, visited nodes are put into this data frame. This can be seen in Table 2, below.

Table 2: The initial visited dataframe

Node	Cost	Previous
''	0.0	''

2. After doing this, the node with the lowest unvisited cost should be set to be the current node.

3. Every unvisited node connected to the current node should then be looped through. The cost of travelling from the current node to the connected node should be checked to see whether it is smaller than the cost already recorded for that connected node. If it is, then the connected node's cost can be updated, and its previous path should be set to the previous path of the current node + the current node.

4. Then, the current node can be removed from the unvisited dataframe and can be placed into the visited dataframe.

5. Steps 2-4 should be repeated until the goal node is in the visited list.

B: Code loop to execute Dijkstra's algorithm

```
3.  # Dijkstra's algorithm!
4.          # Keep running until we get to the goal node
5.          while str(goal_node) not in visited.index.values:
6.
7.              # Go to the node that is the minimum distance from the starting node
8.              current_node = unvisited[unvisited['Cost']==unvisited['Cost'].min()]
9.              current_node_name = current_node.index.values[0]    # the node's name (string)
10.             current_cost = current_node['Cost'].values[0]       # the distance from the
    starting node to this node (float)
11.             current_tree = current_node['Previous'].values[0]   # a list of the nodes
    visited on the way to this one (string)
12.
13.             connected_nodes = graph[current_node.index.values[0]]   # get all of the
    connected nodes to the current node (array)
14.             connected_edges = edges[current_node.index.values[0]]   # get the distance from
    each connected node to the current node
15.
16.             # Loop through all of the nodes connected to the current node
17.             for next_node_name, edge_cost in zip(connected_nodes, connected_edges):
18.                 next_node_name = str(next_node_name)    # the next node's name (string)
19.
20.                 if next_node_name not in visited.index.values:  # if we haven't visited this
    node before
21.
22.                     # update this to calculate the cost of going from the initial node to
    the next node via the current node
23.                     # Original code...
24.                     #next_cost_trial = 1234  # set this to calculate the cost of going from
    the initial node to the next node via the current node
25.                     next_cost_trial = current_cost + edge_cost
26.                     next_cost = unvisited.loc[[next_node_name], ['Cost']].values[0] # the
    previous best cost we've seen going to the next node
27.
28.                     # if it costs less to go the next node from the current node, update
    then next node's cost and the path to get there
29.                     if next_cost_trial < next_cost:
30.                         unvisited.loc[[next_node_name], ['Cost']] = next_cost_trial
31.                         unvisited.loc[[next_node_name], ['Previous']] = current_tree +
    current_node_name    # update the path to get to that node
32.
33.             unvisited.drop(current_node_name, axis=0, inplace=True)     # remove current
    node from the unvisited list
34.
35.             visited.loc[current_node_name] = [current_cost, current_tree]   # add current
    node to the visited list
36.
37.
```

Applied Robotics Coursework 3 – Interact

James Skinner (01700960), Madelaine Wood (01762829),
Harry Schlote (01746509), Fernanda Espinoza (01783661)

Table of Contents

1. Introduction

This report documents the development of 'Coursework 3 – Interact', as part of the Applied Robotics module. The ability for robots to physically interact with their environment is very exciting, and this report covers robot manipulation tasks such as picking and placing objects, as well as applying forces and torques to perform tasks and interact with objects in the environment.

These interaction methods will be applied to a stripped down, virtual representation of Design Engineering's Natural Interaction Robot (Robot DE NIRO), which consists of a dual-arm robot (each with seven degrees of freedom) mounted onto a stationary base. The joints have been denoted as the shoulder (S0 and S1), the elbow (E0 and E1), and the wrist (W0, W1, and W2), as seen in Figure 1.

Figure 1: The simplified version of DE NIRO (left), and the locations of its joints (right)

2. Interaction Using Position Control

2.1. Task A: End-effector orientation calculation

Euler angles are used in this task to help describe the orientation of a rigid body (DE NIRO's hands) with respect to a fixed coordinate system (base frame). The rotation order for this course is z-y-x sequence, with this being yaw, pitch, and roll, respectively. Yaw is the rotation around the z-axis, pitch is the rotation around the y-axis and roll is the rotation around the x-axis. These rotations can be described by three successive rotations around linearly independent axes, as seen in Figure 2.

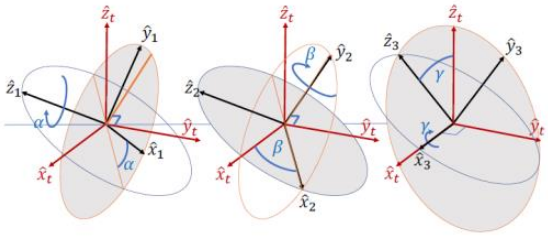


Figure 2: Z-Y-X Euler Angles Rotations

For the first task, the hands should have the end effectors (EE) facing downwards. Given that the z-axis of the EE always points in the forward direction of the end effectors, and the y-axis is always parallel to the grippers' opening/closing movements, this means that the z-axis should be pointing downwards. To achieve this, the coordinate system must rotate 180° about the y-axis (pitch). This rotation can be seen in Figure 3.

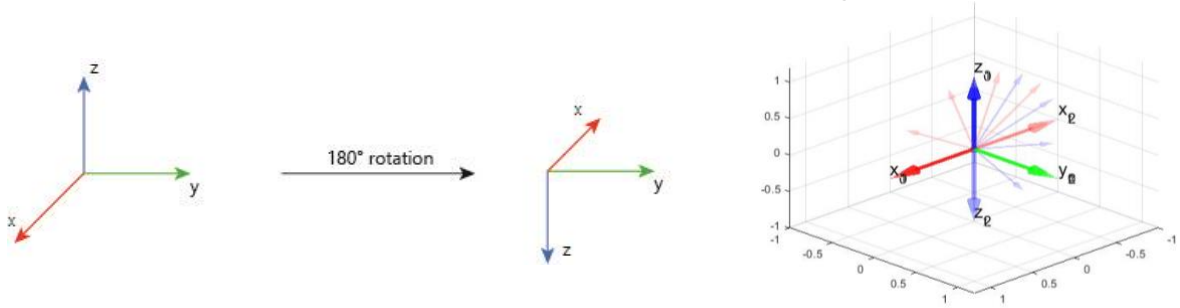


Figure 3: End effector ZYX Euler rotation steps, and visualisation in MATLAB

For this coursework, Z-Y-X moving frame rotations have been implemented. This means that the initial rotation about the z axis is followed by a rotation about the new 'y1' axis, and finally by the new 'x2' axis. The rotation matrix for this transformation consists of three rotations around each of these axes. These rotations are split up according to their axes and are defined as:

Rotation of α radian around the z0-axis:
$$R_z(\alpha) = \begin{bmatrix} cos(\alpha) & -sin(\alpha) & 0 \\ sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation of β radians around the y1-axis:
$$R_y(\beta) = \begin{bmatrix} cos(\beta) & 0 & sin(\beta) \\ 0 & 1 & 0 \\ -sin(\beta) & 0 & cos(\beta) \end{bmatrix}$$

Rotation of γ radians around the x2-axis:
$$R_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\gamma) & -sin(\gamma) \\ 0 & sin(\gamma) & cos(\gamma) \end{bmatrix}$$

Since the rotations occur first around the z-axis, then the y1-axis, and finally the x2-axis, and moving frame rotations are being used, this sequence of rotations can be represented by the following matrix product:

$$R_z(\gamma)R_y(\beta)R_x(\gamma) = \begin{bmatrix} c(\alpha)\,c(\beta) & -c(\gamma)\,s(\alpha) + c(\alpha)\,s(\beta)\,s(\gamma) & s(\alpha)\,s(\gamma) + c(\alpha)\,c(\gamma)\,s(\beta) \\ c(\beta)\,s(\alpha) & c(\alpha)\,c(\gamma) + s(\alpha)\,s(\beta)\,s(\gamma) & -c(\alpha)s(\gamma) + c(\gamma)\,s(\alpha)\,s(\beta) \\ -s(\beta) & c(\beta)s(\gamma) & c(\beta)c(\gamma) \end{bmatrix}$$

For this rotation, the Euler anglers were converted to the following compound rotation matrix:

$$\begin{matrix} yaw\ (\alpha) = 0 \\ pitch\ (\beta) = pi \\ roll\ (\gamma) = 0 \end{matrix} \quad \rightarrow \quad \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

For part ii, to pass the brick between the hands, each end effector must have the correct coordinate orientation, as seen in Figure 4. Each end effector must have their z-axis pointing in towards the brick. This will allow the pincers to grasp the brick with both hands, allowing DE NIRO to pass the brick successfully from one hand to the other. To ensure that the correct direction of rotation was specified, the right-hand grip rule was used, indicating that positive rotation values yield anti-clockwise rotation. An illustration of this can be seen in Figure 5.

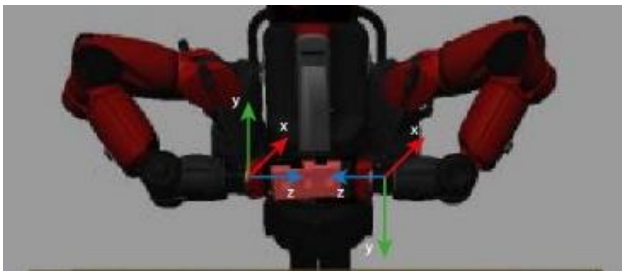


Figure 4: End effectors' coordinate systems

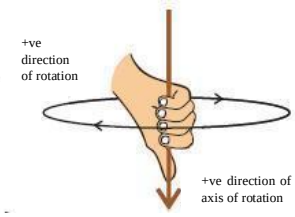


Figure 5: Right-hand grip rule

For De NIRO's left hand, the set of rotations needed to achieve the desired orientation can be seen in Figure 6. For a rotation that is performed in the clockwise direction (when the axis points towards you), a negative angle is used. This means that the 90° rotation around the x-axis should be negated.

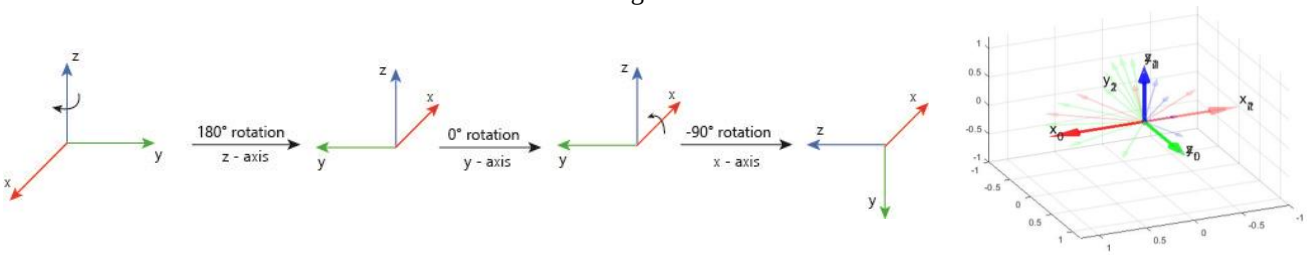


Figure 6: Left hand ZYX Euler rotation steps, and visualisation in MATLAB

These ZYX Euler angles were converted to the following rotation matrix:

$$\begin{matrix} yaw = \pi \\ pitch = 0 \\ roll = -\pi/2 \end{matrix} \quad \rightarrow \quad \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

With regards to DE NIRO's right hand, the sequence of rotations that should occur for the end effector to have the appropriate orientation are seen in Figure 7. The rotations were almost identical, apart from the fact that it rotates 90° anti-clockwise around the x-axis (when the axis points towards you), and hence a positive angle was used.

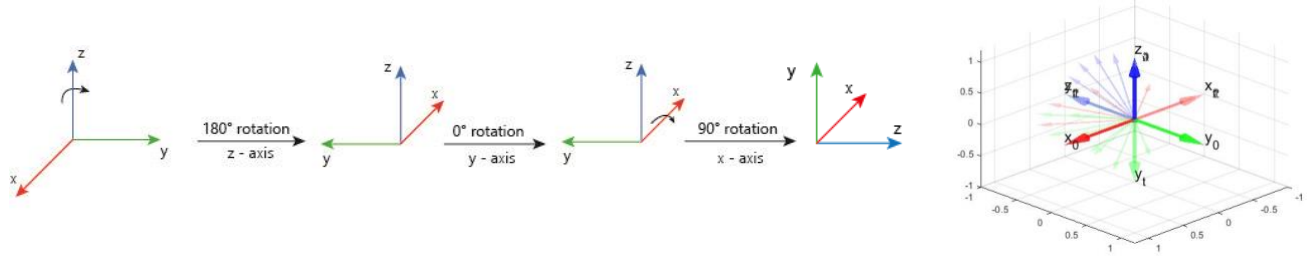


Figure 7: Right hand ZYX Euler rotation steps, and visualisation in MATLAB

These ZYX Euler angles were converted to the following rotation matrix:

$$\begin{matrix} yaw = \pi \\ pitch = 0 \\ roll = \pi/2 \end{matrix} \quad \rightarrow \quad \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

2.1. Task B: Position Control to perform pick-and-place tasks

With regards to the code, for task A, the rotation was defined in the sequence of roll, pitch, and yaw, rather than yaw, pitch, and roll, as before. This meant that the values had to simply be rearranged on input. The code for part i can be seen below, with the values of (0, π, 0) input for each end effector, using numpy to import pi as 'np.pi':

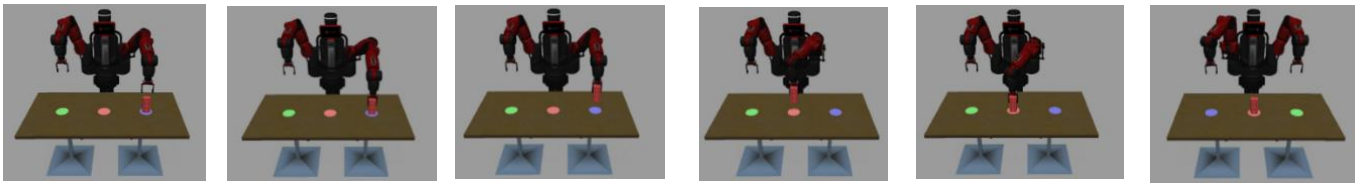
```
1.      # Find the correct orientation for the starting pose
2.      # ========= ========= ============= ========= =========
3.      # Example command to move the arms to a target pose with position control
4.      # Go to the starting Pose for left arm
5.      left_zyx = [0.5, 0.5, 0.25]
6.      left_rpy = [0, np.pi, 0]     # Rotation about y axis (arm oriented vertically downwards)
7.      left_arm.servo_to_pose(left_zyx, left_rpy)
8.
9.      # Go to the starting Pose for right arm
10.     right_zyx = [0.5, -0.5, 0.25]
11.     right_rpy = [0, np.pi, 0]
12.     right_arm.servo_to_pose(right_zyx, right_rpy)
```

Finding the coordinates of each element within the space was done by inspecting the pose window within gazebo. This allowed for the exact coordinate positions of each element to be taken, therefore resulting in a more accurate movement of the brick from one circle to the other. The position of each circle can be seen in Table 1.

Table 1: The positions of each circle, obtained from Gazebo

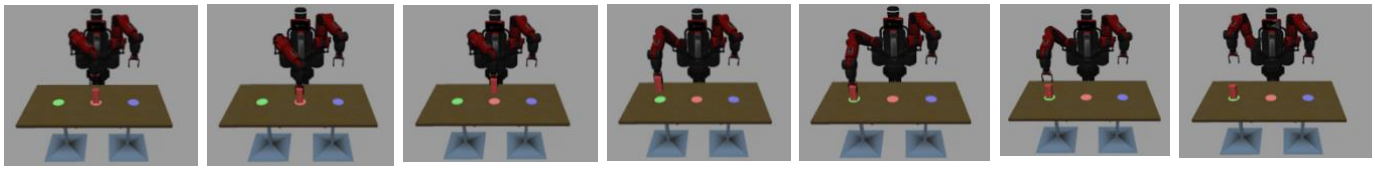
	X coordinate	Y coordinate	Z coordinate
Blue circle	0.75	0.5	0.775
Red circle	0.75	0	0.775
Green circle	0.75	-0.5	0.775

Knowing these coordinates for each circle, the following x, y, z positions were used to perform the pick and place task:



(0.75, 0.5, 0.25) *(0.75, 0.5, 0)* *(0.75, 0.5, 0.25)* *(0.75, 0, 0.25)* *(0.75, 0, 0)* *(0.75, 0, 0.25)*

Figure 8: Left arm motion (and co-ordinates of end effector) for pick and place task



(0.75, 0, 0.25) *(0.75, 0, 0)* *(0.75, 0, 0.25)* *(0.25, -0.5, 0.25)* *(0.75, -0.5, 0)* *(0.75, -0.5, 0.25)* *(0.5, -0.5, 0.25)*

Figure 9: Right arm motion (and co-ordinates of end effector) for pick and place task

The waypoints were carefully considered at every step. The first step was chosen to move directly above the brick, before moving the gripper directly downwards to grip onto the brick and lift it off the table. This was necessary as if the arm moved directly towards the brick (i.e., not from above), it may have knocked the brick over before it could grab it. Similarly, after letting go of the brick (e.g. when placed on the red circle) coordinates were chosen to avoid knocking over the brick before moving the arm back to the initial coordinates, making sure the arm first moved directly upwards so to not knock the brick sideways. When operating the right arm, similar techniques were used such that the end effector was in the correct positions, before moving towards or away from the brick, ensuring that it would not be knocked over. These commands were specified in the code as follows:

```
1.          left_xyz = [0.75, 0.5, 0.25]     # Move arm above brick
2.          left_arm.servo_to_pose(left_xyz, left_rpy)
3.          left_xyz = [0.75, 0.5, 0]        # Move arm around brick
4.          left_arm.servo_to_pose(left_xyz, left_rpy)
```

```
        left_arm.gripper_close()         # Grab brick
        left_xyz = [0.75, 0.5, 0.25]     # Lift up
        left_arm.servo_to_pose(left_xyz, left_rpy)
        left_xyz = [0.75, 0, 0.25]       # Move across
        left_arm.servo_to_pose(left_xyz, left_rpy)
        left_xyz = [0.75, 0, 0]        # Put down
        left_arm.servo_to_pose(left_xyz, left_rpy)
        left_arm.gripper_open()          # Let go of brick
        left_xyz = [0.75, 0, 0.25]       # Move arm up (before moving out the way)
        left_arm.servo_to_pose(left_xyz, left_rpy)
        left_xyz = [0.5, 0.5, 0.25]      # Move left arm back to initial position
        left_arm.servo_to_pose(left_xyz, left_rpy)
        right_xyz = [0.75, 0, 0.25]      # Mover right arm above brick
        right_arm.servo_to_pose(right_xyz, right_rpy)
        right_xyz = [0.75, 0, 0]         # Move arm around brick
        right_arm.servo_to_pose(right_xyz, right_rpy)
        right_arm.gripper_close()        # Grab brick
        right_xyz = [0.75, 0, 0.25]      # Lift up
        right_arm.servo_to_pose(right_xyz, right_rpy)
        right_xyz = [0.75, -0.5, 0.25]   # Move across
        right_arm.servo_to_pose(right_xyz, right_rpy)
        right_xyz = [0.75, -0.5, 0]      # Put down
        right_arm.servo_to_pose(right_xyz, right_rpy)
        right_arm.gripper_open()         # Let go of brick
        right_xyz = [0.75, -0.5, 0.25]   # Move arm up (before moving out the way)
        right_arm.servo_to_pose(right_xyz, right_rpy)
        right_xyz = [0.5, -0.5, 0.25]    # Move arm back to init position
        right_arm.servo_to_pose(right_xyz, right_rpy)
```

The second part of task B was similar to the first, but with the brick being passed between the hands instead. To do this, the brick had to be perfectly horizontal and the gripper on the brick had to be exactly in the right position to help the other hand attach onto the brick. The waypoints used for this task can be seen in the code, as follows:

```
        left_xyz = [0.75, 0.5, 0.25]             # Move left arm above brick
        left_arm.servo_to_pose(left_xyz, left_rpy)
        left_xyz = [0.75, 0.5, 0]                # Move arm around brick
        left_arm.servo_to_pose(left_xyz, left_rpy)
        left_arm.gripper_close()                 # Grab brick
        left_xyz = [0.75, 0.5, 0.25]             # Lift up
        left_arm.servo_to_pose(left_xyz, left_rpy)
        left_rpy = [-np.pi/2, 0, np.pi]          # Rotate brick for passover
        left_xyz = [0.75, 0.05, 0.25]            # Pass brick into centre of the space
        #left_xyz = [0.75, 0.5, 0.25]            # Originally, had this - but this is outside
    of the workspace (can't be in this orientation at this position)
        left_arm.servo_to_pose(left_xyz, left_rpy)
        right_rpy = [np.pi/2, 0, np.pi]          # Rotate right arm for passover
        right_xyz = [0.6, -0.05, 0.25]           # Move right arm to be close to almost grab
    the brick (but closer to the robot in x direction to not knock brick when rotating)
        right_arm.servo_to_pose(right_xyz, right_rpy)
        right_xyz = [0.75, -0.05, 0.25]          # Move right gripper aound the brick
        right_arm.servo_to_pose(right_xyz, right_rpy)
        right_arm.gripper_close()                # Grab brick with right arm
        left_arm.gripper_open()                  # Let go of brick from left arm
        left_xyz = [0.75, 0.075, 0.25]           # Move left arm away
        left_arm.servo_to_pose(left_xyz, left_rpy)
        right_xyz = [0.75, -0.075, 0.25]         # Move right arm away
        right_arm.servo_to_pose(right_xyz, right_rpy)
        left_xyz = [0.5, 0.5, 0.25]              # Left arm back to init position
        left_rpy = [0, np.pi, 0]                 # Left arm back to init rotation
        left_arm.servo_to_pose(left_xyz, left_rpy)
        right_rpy = [0, np.pi, 0]                # Rotate so that brick is vertical again
        right_xyz = [0.75, -0.5, 0.25]           # Move brick above final position
        right_arm.servo_to_pose(right_xyz, right_rpy)
        right_xyz = [0.75, -0.5, 0]              # Put down
        right_arm.servo_to_pose(right_xyz, right_rpy)
        right_arm.gripper_open()                 # Let go of brick
        right_xyz = [0.75, -0.5, 0.25]           # Move arm up (before moving out the way)
        right_arm.servo_to_pose(right_xyz, right_rpy)
        right_xyz = [0.5, -0.5, 0.25]            # Move arm back to init position
        right_arm.servo_to_pose(right_xyz, right_rpy)
```

These waypoints yielded successful motion to pass the brick between the hands, as seen in Figure 10, below.

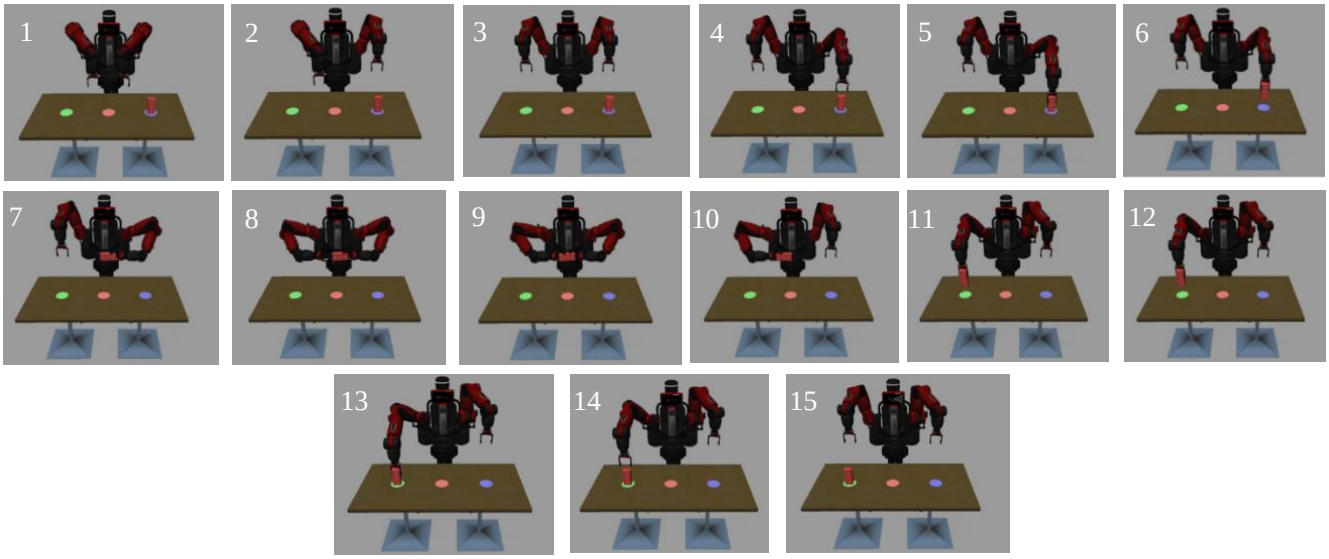


Figure 10: Step-by-step motion for pick and place task, passing between the two hands

The waypoints were carefully considered at every step. A lot of the same steps from part ii were carried through, but now on the 7th step, the orientation of the left end effector was changed to be horizontal instead, with the z-axis pointing in towards the brick, as calculated in Task A. The brick was found to be 0.25m tall, so it was calculated that to pass the brick between the hands, the end effectors should be 0.125m horizontally either side of the brick's centre. The right gripper was opened, and the left gripper was closed at the same step, this allowed for a smoother transition of the brick between the end effectors. Having carefully moved the left arm out of the way, the right arm was then orientated back to its vertical position, allowing DE NIRO to easily place the brick back down on the table to finish the task.

There were many problems encountered while undertaking this task. Firstly, when conducting inverse kinematics on the end effectors, some positions were not possible, such as the left arm at desired pose of zyx (0.75, 0.5, 0.25) and rpy ($-\pi/2$, 0, π). The identified reason for this problem was that this waypoint was not within DE NIRO's reachable workspace (i.e., the brick could not be held horizontally at this position), thus meaning that there was no solution for the inverse kinematics at this point. With that in mind, the points were adjusted, with the desired points being brought closer to DeNiro's body, so that all the points on the trajectory were feasible and withing the reachable workspace.

Another issue that arose was that changing the orientation of the end effector whilst the brick was held, from vertical to horizontal, meant that often, the brick would hit the surface of the table or would hit the other arm, and so the brick was dropped. Therefore, extra intermediate steps were required. One example of a necessary intermediate step can be seen in steps 8 and 9 from figure 10, where the right arm was first brought just behind the brick, before moving around the brick to grasp it. Without this addition, the right arm would try to orientate and grab the brick in one go, knocking the brick out of the left hand. This can also be seen in the supporting video. Furthermore, many of the waypoints were vertically raised to allow for extra insurance, meaning that the brick would be less likely to collide with the table unexpectedly. This ensured that the task could be successfully completed, without dropping the brick.

3. Interaction using Velocity Control

3.1. Task C: Pose definition

Having previously controlled the motion of the robot arms by specifying a trajectory in terms of positions and orientations, the robot will now be controlled by specifying the trajectory in terms of velocity instead. The aim of this section is to use velocity control to perform three separate motions: approaching and picking an object from the table, moving the object in a circular path whilst maintaining a constant end effector orientation, and adjusting the arm to avoid an obstacle whilst ensuring a constant end effector pose (stationary position and orientation).

The first requirement was to fill in the code to determine the desired positions that the robot must reach for each of the motions previously described. These positions were: [0.75, 0, 0.93] for the approach and pick task (co-ordinates specify the location of the brick), and [0.75, 0.1, 1.23] for both the circular path task (starting position of the circular motion) and obstacle avoidance task (position to maintain brick at whilst avoiding the obstacle). The code for this was trivial.

When working with robotic arms, safety is an imperative factor to consider. An interesting implementation for Robot DE NIRO is that when changing positions, the arm first moves to a position 0.18 m above the desired point, before slowly moving down to grasp an object, and then slowly moving back up. This can be seen in the following code from the 'reachPose' function:

```
1.  # approach from 0.18 m above
2.      xyz_approach = [xyz_des[0], xyz_des[1], xyz_des[2] + 0.18]
```

This steady approach method is beneficial as it gives users near the robot arm sufficient time to react if unexpected motion begins to occur (users like operators may be close to the robot arms during programming and troubleshooting). For example, if the wrong pose has been specified in the code, or other bugs are present, the arm may move closer towards the user than intended, as opposed to moving towards the desired target position. Without the slow speed safety precaution, this could either lead to the arm rapidly moving towards the user and causing injury through fast impact, or alternatively, the person also risks being injured through the robot trapping them against a fixed object. By instead ensuring that the robot approaches objects from a reasonable distance above, and slowly moves to pick them up, this affords any user sufficient time to either get out of the way, or actuate an emergency stop device before the robot arm can reach the person at risk, minimising the chance of injury.

A further requirement was to specify the desired orientation to grab the object from behind, rather than from above. This was to be defined in terms of roll, pitch, and yaw angles, before they were converted to quaternions later on. This was calculated visually, using the same method as seen in task A. When using Euler rotations, the aim of the first two rotations is to get one axis to coincide with the corresponding axis of the target frame, before a final third rotation is used to align the other two axes. Therefore, with reference to Figure 11, given that the y axis for the desired orientation to grab from behind already coincides with that of the original base frame, only one rotation around the y axis was required for this transformation. By simply rotating 90 degrees (equivalent to pi/2) anticlockwise around the y axis, using the right-hand grip rule to ensure that the correct direction is taken (anticlockwise being positive rotation), this yielded a desired orientation of [roll, pitch, yaw] = [0, pi/2, 0].

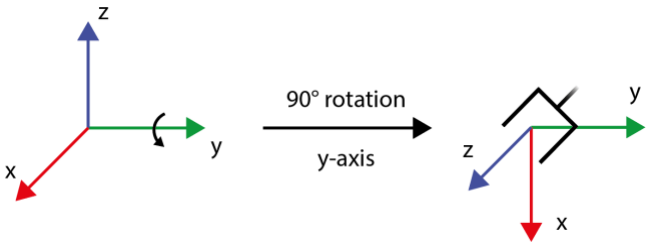


Figure 11: Robot arm 'grabbing from behind' ZYX Euler rotations

It should be noted that the desired orientation is later converted to a quaternion representation, as opposed to using the Euler angles to produce three separate rotation matrices. This is a compact way of representing desired rotations, where any three rotations about three independent axes (Euler rotations) can be represented instead as a single rotation (of angle theta) around a unique axis of rotation (a unit vector, r) – this will be discussed further in task D. There are several benefits of implementing quaternions over roll, pitch, and yaw, for measuring and controlling orientation: the first being computational efficiency.

When combining multiple rotations, otherwise known as concatenating rotations, quaternion multiplication is much faster than 3x3 matrix multiplication. If Euler angles were used, first, the 3x3 rotation matrices would need to be computed from given roll, pitch, and yaw angles, and then these 3x3 rotation matrices would be multiplied together to give the compound rotation. Multiplying two 3x3 matrices together requires $3^3 = 27$ multiplication calculations, whereas multiplying two quaternions together requires only 16 multiplication calculations. Thus, by implementing quaternions, fewer calculations are required, improving the efficiency, which is crucial given that the robot must compute a large number of rotations in real time.

The second advantage is that quaternions have easier and more well-defined interpolation between two different orientations (calculating points along the trajectory from one orientation to another), where a resulting movement that has a constant angular velocity around a single axis can be achieved. To interpolate between the orientations, the angle of rotation (theta) can simply be divided into even steps. However, if ZYX Euler rotations were used instead, it would be much harder to calculate interpolated points to provide a smooth, direct motion from one orientation to another. Given that velocity control is being implemented, and calculating these interpolated points is required, implementing quaternions is essential for this need.

The final advantage is that quaternions avoid the phenomenon known as ‘gimbal lock’. This is when two rotational axes become aligned, and so a degree of freedom is lost, meaning that the arm cannot be instantaneously rotated in a certain direction. In gimbal lock, a rotation can still be achieved to move from any orientation to another, but this orientation will be non-smooth, such that it cannot change orientation in just a single, linear movement. This would be problematic for the robot arm, given that we want to achieve smooth, direct motion. Gimbal lock occurs when the Euler angle representation for a given rotation matrix is not unique. It can be avoided by storing the complete 3x3 rotation matrix instead of the three Euler angles, but this is then a far less compact representation than quaternions, storing 9 values instead of 4 for each rotation. This is therefore less storage efficient, and hence, quaternions are beneficial.

It should be noted that one drawback of using quaternions is that they are much less intuitive to the user. Trying to visualise a single rotation around an arbitrary axis is much harder than considering three separate rotations around independent axes. This is the reason for specifying the roll, pitch, and yaw angles in code, before they are later converted to quaternion representation. The code for both parts of this task is as follows:

```
1.  ## Task C:
2.      # fill in the desired poses
3.      xyz_des_pick = [0.75, 0, 0.93]          # Co-ord from tutorial sheet
4.      xyz_des_circle = [0.75, 0.1, 1.23]      # Co-ord from tutorial sheet
5.      #rpy_des = [-np.pi, 0, np.pi]    # ZYX Euler rotation to give orientation to grab from
    above
6.      rpy_des = [0, np.pi/2, 0]       # Grab brick from behind instead (alternate
    orientation)
7.
```

3.2. Task D: Twist Computation

For this task, a twist needs to be computed that specifies both linear velocities (along x, y, and z directions) and angular velocities (about x, y, and z axes) of the end effector, such that it can move from one pose to the next. This is done by specifying desired poses (positions and orientations) along a trajectory (e.g. positions around a circle with radius = 0.05 m, as in the code), before calculating the velocities to move from each pose of the trajectory to the next. First, we will just compute the linear velocities.

Linear velocity is calculated using the formula: velocity = linear displacement / time. The desired end effector position (in terms of x, y, and z) is passed into the function - this is the next position to reach along the trajectory, a small time into the future ($\Delta t = 1/100$ s). The current end effector position at each time instant (also in terms of x, y, and z) is then calculated using forward kinematics, where the values of each joint angle are input into the predefined DH table and are used to compute the transformation matrices that are multiplied together to calculate the pose of the end effector, with respect to the base frame of the robot. To calculate linear displacement, we simply calculate the direction vector from the current position to desired position, using the formula ΔP = [x_des, y_des, z_des] – [x_current, y_current, z_current]. Each of these values are then divided by the time interval between the two positions, Δt, to calculate the linear velocities. This was implemented in the code as follows:

```
1.          #Original Code...
2.          #dP = []  # linear displacement. Note that P_des is the desired end-effector position
    at the next time instant. P is the current position (calculated from FK)
3.          dP = (P_des - P)/dt      # compute linear velocity, dP -> velocity = displacement
    (difference between positions) / time
```

The main benefit to estimating velocity based on position, as implemented, is that it is much easier for the sensors to measure position, rather than to directly measure velocity. The current position can be calculated simply by measuring the position of each joint angle, and applying forwards kinematics, before differentiating to obtain velocity. Measuring the velocity of each joint is much trickier for the actuator sensors to do, and hence, is avoided. The key drawback to this method, however, is that it is more susceptible to noise errors. When estimating the position of the end effector, non-perfect, noisy readings are obtained. The effect of this noise becomes greater when computing the velocity, where subtracting another position (to calculate the displacement) and discretising the values (dividing by a small interval, Δt) amplifies this noise error, relative to its original value. Thus, we sacrifice some tracking accuracy, in order to calculate the velocity more easily.

To calculate the angular velocity, we use a similar equation to before: angular velocity = angular displacement / time. However, calculating the angular displacement is slightly more complicated than the linear displacement, and requires the use of quaternions. Quaternions can be used to effectively represent a single rotation, theta (called 'delta_angle' in the code), around a unit axis, r (in terms of directions along the x, y, and z axes), as can be seen in Figure 12. This axis-angle representation is converted to and stored as a 'rotation quaternion', q:

$$q = (q0, q1, q2, q3), \text{ where:}$$

$$q0 = \cos(\text{theta}/2),\ q1 = \hat{x}\sin(\text{theta}/2),\ q2 = \hat{y}\sin(\text{theta}/2),\ q3 = \hat{z}\sin(\text{theta}/2)$$

Similarly, the quaternion difference between two quaternions is: $Q_{err} = \cos(\Delta\theta/2) + r\sin(\Delta\theta/2)$ **(1)**

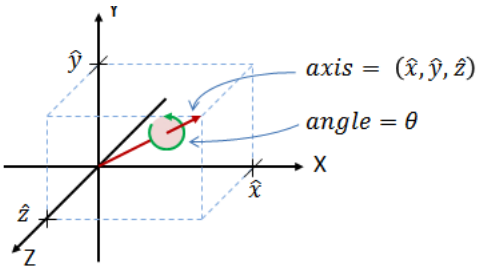


Figure 12: Any 3D rotation can be specified by an axis of rotation and a rotation angle about that axis

Like before, the desired orientation is passed into the function, specified as a quaternion, let's call it q2. Also as before, the current orientation of the end effector is calculated by applying forward kinematics based on the current joint angles, also calculated and stored as a quaternion, let's call it q1. Say we have any two quaternions: q1 and q2, and the difference in the two orientations is Δq. This yields the equation: $q2 = \Delta q * q1$ (note that the multiplication is the Hamilton product, calculated based on various dot products and cross products between the two quaternions). Therefore, to obtain the difference between the quaternions, the equation is as follows:

$$\Delta q = q2 * q1^{-1} \qquad \textbf{(2)}$$

Thus, to obtain the angular displacement between the two quaternions for our robot arm, we must first calculate the inverse (or reciprocal) of the current quaternion (q1). For unit quaternions, the inverse is the same as the conjugate, or in other words, we maintain the scalar value (q0), and negate the vector components (q1, q2 and q3). This is implemented in the code as follows:

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

Having calculated the inverse, we can then calculate the quaternion displacement (Δq), otherwise known as the quaternion error, using equation (2). Where the 'quaternionProduct' function calculates the Hamilton product between two quaternions, this was implemented in the code as follows:

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

Finally, we can convert the quaternion, q, back to a more intuitive axis-angle representation, to obtain theta and r. It should be noted that there are several ways to convert from quaternion to axis-angle, and this was implemented in the code as follows:

```
def quat2angax(quat):
    """ Convert quaternion to angle and axis"""
    v = sqrt(np.sum(np.power(quat[0:3], 2)))  # get the magnitude of the axis components of the quaternion

    if v > 1e-10:
        r = quat[0:3] / v   # convert axis components of the quaternion to unit axis
        angle = 2 * atan2(v, quat[3])   # calculate angle
    else:   # if the axis component has no magnitude, it is pointing in Z direction with zero angle
        r = np.array([0, 0, 1])
        angle = 0.

    angle = atan2(sin(angle), cos(angle))
    return angle,r


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

The values of theta (delta_angle) and r can then be used to compute the angular velocity. To compute the angular displacement around the x, y, and z axes, we simply multiply the angle of rotation, theta, by each component in the unit vector of rotation, r. This is then divided by dt to calculate the angular velocities about each axis, and was implemented in the code, as follows:

```
dw = (r*delta_angle)/dt    # Part ii - multiply the angle of rotation error by the axis of rotation (to give angular displacement), then divide by time (to calculate angular velocity)

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

```

With both the linear and angular velocities calculated for a given time interval, dt, the Cartesian twist (T) can then be used to calculate the instantaneous joint velocities at each time interval to allow the robot arm to complete the desired motion trajectory. This will be discussed in task E.

3.3. Task E: Joint Velocities computation

The instantaneous Cartesian velocity, otherwise known as 'Twist', computed in task D, can be expressed as:

$$T = \begin{bmatrix} \dot{P} \\ \omega \end{bmatrix} = J(q)\dot{q} \quad \textbf{(3)}$$

Where: T is the desired Cartesian twist, J is the arm end-effector's Jacobian, $\dot{q}$ is the joint velocities

This can be rearranged to give an equation in terms of the joint velocities:

$$\dot{q} = J^{\#}(q)T \qquad \textbf{(4)}$$

Where: $^{\#}$ is the pseudoinverse operator

Therefore, to calculate the instantaneous joint velocities, we first need to calculate the Jacobian of the end effector, which specifies the derivative of each joint (7 joints total) with respect to each degree of freedom (6 DOF – x, y, z, θ_x, θ_y, θ_z), producing a 6x7 matrix. This is computed by the pre-written 'Jacobian' Python function, which uses the PyKDL library. The pseudoinverse of this Jacobian is then calculated, using the pre-written 'DPinv()' function. It should be noted that the pseudoinverse must be used because the Jacobian matrix is non-square (6x7, as noted previously), and

non-square matrices are not directly invertible. A benefit of implementing the pseudoinverse is that it yields a 'least squared error, minimum-norm' solution, and thus avoids issues of singularities, which would otherwise cause some instances where the inverse cannot be calculated. The pseudoinverse is calculated using the following equation:

$$J^{\#} = J^{T}(JJ^{T})^{-1} \quad \textbf{(5)}$$

Where: ^{T} is the transpose operator, $^{-1}$ is the inverse operator

The pseudoinverse of the Jacobian yields a 7x6 matrix, which is the correct dimensions to then be multiplied by the twist vector (a single vector storing 6 values). By multiplying these together, as in equation (4), this calculates the joint velocities, $\dot{q}$. This was implemented in the code as follows:

```
1.  ##### Task E
2.          # compute Jacobian of the end-effector and solve velocity IK with pseudoinverse. Use
    self.jacobian() with joint_values as inputs
3.          J_ee = self.jacobian(joint_values)  # Use joint angles to calculate the Jacobian up
    to the end effector
4.          J_ee = np.asarray(J_ee) # convert to np.array
5.          J_pinv = DPinv(J_ee, eps=1e-10)     # Compute the pseudoinverse of the jacobian
6.          qd = np.matmul(J_pinv, twist)       # Compute joint velocities - Following equation
    from tutorial sheet: q_dot = pseudoinv(J)*twist
7.
```

The main advantage of using velocity control is that when working with robot arms of more than three degrees of freedom, it is much easier to calculate inverse differential kinematics (which calculates joint velocities, as seen in task E) as opposed to calculating inverse positional kinematics (to obtain pure joint angle positions). To calculate inverse positional kinematics, an analytical or geometric approach is used to derive the joint angles using trigonometry. This approach was implemented in coursework 1 and is suitable for robotic arms with few joints. However, as the number of joints (or DOF) increases, computing the joints via trigonometric analysis becomes extremely complicated. Comparatively, a numerical approach, where we split the motion between positions into small time steps and continually calculate the required velocity at each timestep, is much more feasible when working with many (> 3) degrees of freedom. Using this approach (velocity control) simply requires us to calculate the Jacobian, take the pseudoinverse of this Jacobian, and multiply this with the desired velocity vector.

Another advantage is that velocity control typically yields smoother motion than positional control. Using positional control, each joint moves directly to the desired joint angle, potentially causing a jittery motion in the end effector. This non-smooth motion is perceivable when there is a delay in the control signal, where positions along a trajectory are not updated frequently enough. However, when implementing velocity control, as the joints move to cause a specified end-effector velocity, it is more robust to variations in control signal delay. This is illustrated clearly in Figure 13 [1]. Furthermore, the tracking accuracy of DE NIRO using velocity control was plotted, as seen in Figure 14.

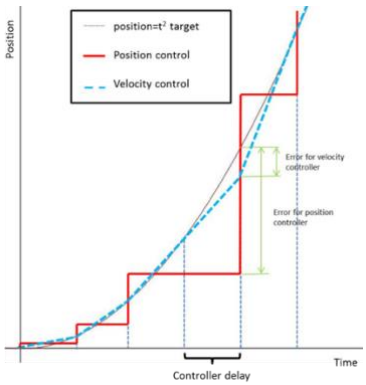


Figure 13: Graphical comparison of position/velocity control [1]

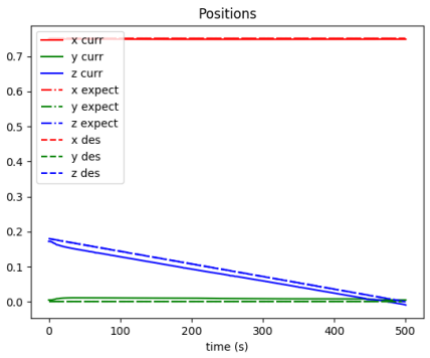


Figure 14: Example of tracking accuracy for velocity control

The main drawback of velocity control is that there is a greater safety risk when errors occur in the code. When bugs are present which cause the code to crash, updated velocity control commands will no longer be sent to the joints. The joints will thus continue to move at the previously specified velocity, which could cause the robot arm to rapidly move in undesired directions. This effect can be seen in the Robot DE NIRO simulation when running the velocity control commands and applying a keyboard interrupt (ctrl + c). Even though the code loop stops running, the robot arm continues to move with constant joint velocities. Positional control avoids this issue, as once the joints have been updated to reach the desired positions, no further motion will occur.

3.4. Task F: Null Space Projector

The following two tasks, F and G, consist in exploring the advantages of the redundancy and null space of DE NIRO. Null space is a set of movements that reconfigure the manipulator structure without changing the end effector position and orientation, and redundancy is where there is more than one configuration that satisfies the posed problem.

Since any task-space motion can be at most 6-dimensional (3 positions and 3 rotations) and each arm of DE NIRO has 7 degrees of freedom, redundancy is present. One advantage of this redundancy is that the robot can be made to perform a secondary motion task while performing the primary task, since the secondary one would not interfere with the primary one. A method typically used for this is the Null Space Projection Method which was computed in task F. The method consists in finding joint velocities q_1 to perform the primary task, and project additional joint velocities q_2 to perform a secondary task into the null space of the primary task, such that the primary task is not interrupted. Based on this, the total joint velocities to be commanded to the robot are computed as:

$$\dot{q} = \dot{q}_1 + N\dot{q}_2$$

Where N is the Null Space Projector which is defined as:

$$N = I - J^{\dagger}J$$

The projection was computed in Task F, given the robot's end-effector Jacobian, J_ee, and the identity matrix, with a dimensionality equal to the number of joints (Line 3) as shown in the following code (Line 4):

```
1.    #####Task F
2.              # Compute projector into the ee Jacobian null space and joint velocity to reach
    desired configuration
3.              I = np.eye(nj)
4.              Proj = I - np.matmul(J_pinv, J_ee)      # Following equation from tutorial sheet
    -> N = I - Pinv(J)*J
5.
```

In the code, the numpy.matmul() function was used to compute the matrix multiplication of two matrix arrays. In this case, used for the multiplication of the Jacobian pseudo-inverse with the Jacobian of the end-effector.

In this section, the primary motion task is for the robot's end-effector to be at a fixed pose $\dot{q}_1 = J^{\#}T_1$ while the secondary task is to avoid a red sphere that suddenly appears beside the elbow of the arm. The secondary task is guaranteed to not interfere with the primary motion, because the final linear velocities are independent from the additional joint velocities $\dot{q}_2$. This can be proven analytically by using the linear velocity equation and the Null Space Projector equation:

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

$$J\dot{q} = J\dot{q}_1 + JN\dot{q}_2$$

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

$$J\dot{q} = J\dot{q}_1 + JI\dot{q}_2 - JJ^{\dagger}J\dot{q}_2$$

Since $JJ^{\dagger}J = J$ and $JI = J$, then this simplifies into:

$$J\dot{q} = J\dot{q}_1 + J\dot{q}_2 - J\dot{q}_2$$

$$\dot{\mathbf{x}} = J\dot{q} = J\dot{q}_1$$

The result is that the linear velocities are equal to that of the primary task, $\dot{q} = \dot{q}_1$, therefore $\dot{q}_2$ will not interfere with this motion.

3.5. Task G: Redundancy Resolution

Task G consists of computing the desired secondary joint velocity $\dot{q}_2$ to drive the robot to the specified configuration q_{des}, defined in the code as q_desired, given the actual joint values of the robot. Joint velocity can be calculated by finding the difference between joint positions, in this case the difference between the joint values and the desired configuration, and then dividing that by the sampling time, dt. This is shown in the code below:

```
1.   #####Task G part i
2.              # Compute secondary joint velocities to reach desired configuration q_desired,
    given the current joint values joint_values
3.              # and sampling time dt
4.              q_desired = self.joint_des
5.
6.              #Original code
7.              #qd2 = []    # replace [] with secondary joint velocities calculated from
    q_desired, joint_values, and sampling time dt.
8.              qd2 = (q_desired - joint_values)/dt
9.              qd2 = 0.01 * qd2 # 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).
10.
```

Projecting some joint velocities in the null space of the primary task means that the manipulator structure, in this case the robot's arm, can be reconfigured without changing the end-effector position or orientation. This is also defined as using 'internal motions' inside the null space. There are other strategies that can be used to exploit the arm's redundancy. One of them is to use a gradient descent method to find the most efficient way to perform a task, since it can be used to reduce some cost which could be time or energy. A standard iterative application of this method is one where at each step the Jacobian, the pseudo-inverse Jacobian, and the Euclidean distance between the end-effector and the target are computed. From these values the next joint angles are computed by following the gradient with respect to the end-effector distance. [2] Optimisation methods resolve redundancy to minimise energy, maximise mechanical advantage and minimise joint velocity.

As a result of this redundancy, the robot is capable to move its elbow away from the obstacle (the red sphere) to avoid collision, without affecting the end-effector's pose. Additional advantages of redundancy include potentially increasing the reachable workspace and improving dexterity [3].

The second part of this task involved commanding a desired motion to the elbow of the robot to avoid the obstacle. To do this, the Jacobian of the elbow had to be computed first by completing the function 'self.link_jacobian'. Inside this function, another function 'self._jac_kdl_link.JntToJac' is used to extract the Jacobian up to the elbow of the robot, where the inputs are the joint values in KDL and the Jacobian matrix (Line 3). Since the elbow is the 4th joint of the robot, only the initial four joints are passed to the function, so the returned Jacobian will be a 6x4 matrix. However, the Jacobian needs to be a 6x7 matrix because only then its pseudo-inverse will result in a 7x6 matrix, which allows the matrix multiplication with the elbow's joint velocities - a vector containing 6 velocity values (x, y, z, θ_x, θ_y, θ_z). So, to convert the returned Jacobian into to a full-size Jacobian, three additional columns of zeros were added. This was done by creating a 6x3 array of zeros (Line 7), where the number of columns was calculated by finding the difference between the total number of joints and the number of joints passed through the function. Then, the 'numpy.concatenate' function was used to append the array of zeros to the elbow Jacobian (Line 8). Lastly, the function returned the linear part of the Jacobian (Line 8). This code can be seen as follows:

```
1.  # Task G part ii – jacobian up to elbow function:
2.          # Fill in the function to compute elbow Jacobian. Inputs are the joint values in KDL
    and the jacobian matrix
3.          self._jac_kdl_link.JntToJac(q_kdl, jacobian)
4.
5.          J_link = self.kdl_to_mat(jacobian) # convert jacobian from PyKDL format to numpy
    array
6.          # after computing jacobian for the elbow, need to convert it to full size jacobian
7.          J_zeros = np.zeros((6, nj_tot-nj))
8.          J = np.concatenate((J_link, J_zeros), axis=1)   # Append extra columns of zero onto
    the end - needed for later multiplication
9.
10.         return J[0:3,:] # take only linear part of the jacobian
```

After the function above was completed, the elbow Jacobian could be used to compute the motion of the elbow. This involved passing 'joint_values' through the function, assigning the result to 'J_elbow' (Line 2) and then converting 'J_elbow' into an array (Line 3) which was critical for the completion of the task, since it allowed the application of numpy functions. Then, the pseudo-inverse was computed with the function 'DPinv' (Line 5) and assigned to the variable 'J_pinv_elbow'. The equation $\dot{q} = J^{-1}v_e$ was used to calculate the secondary motion of the elbow since $\dot{q}_2 = J^{\dagger}v_e$. So, the 'numpy.matmul' function was used to compute the matrix multiplication between the pseudoinverse of the elbow Jacobian and the velocity of the elbow, to find the joint velocities for the secondary motion (Line 6). Finally,

using the Null Space Projection calculated in task F, the final joint velocities were computed (Line 8). The code was implemented as follows:

```
1.  #####Task G part ii
2.              J_elbow = self.link_jacobian(joint_values)      # compute the elbow Jacobian
3.              J_elbow = np.asarray(J_elbow) # convert to np.array - originally this wasn't
    here - really important to add!!!
4.
5.              J_pinv_elbow = DPinv(J_elbow, 1e-6)             # convert pseudoinverse
6.              qd2 = np.matmul(J_pinv_elbow, vel_elbow)        # calculate the joint velocities
7.
8.              qd = qd + np.matmul(Proj, qd2)  # projection in Null space
```

It should be noted that the secondary elbow motion is defined by the movement of the first four joints. However, it still affects the motion of the subsequent joints because they need to compensate the secondary motion to ensure it does not change the end-effector's pose.

4. Interaction using Torque Control

In this section, the way in which DE NIRO can be controlled to interact with its environment using forces and torques will be studied. An advantage of torque control is that in tasks that don't require rigidity, the robot can be made more compliant such that it will be much safer for people around it. Another benefit is that you don't need very high precision in perception or position of the objects that the robot will interact with, because the torque control allows you to adapt by applying force instead. A disadvantage, however, is that it's difficult to apply safety constraints. For example, the self-collision detection between the two arms is difficult while using torque control, and so this gets disabled. Therefore, in these tasks only one arm will move at a time.

4.1. Task H: Demolishing the wall

The feed-forward joint torques on a robot can be calculated from its dynamics:

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

This complex equation used to calculate the full dynamic model of one of DeNiro's arms can be simplified, by assuming that gravity is negligible (changing it from -9.81 in the z-axis to -0.01). The gravity term $\tau_g(q)$ can therefore be ignored, which would normally make up a significant amount of torque on each joint when the arm is stationary. A PD controller was also implemented to simplify the equation such that the remaining dynamic terms to account for inertia and Coriolis forces were not needed. This kept DeNiro's end effector moving in the direction of the external force applied to it.

The Jacobian can be used to convert end-effector forces to joint torques:

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

In order to demolish the wall, the left arm of DeNiro should pick up the brick and throw it towards the wall. This requires a force in the Y-axis. Due to gravity being negligible, a z-axis component of force was not required.

Table 2: Trialling different forces, allowing the brick to knock down the wall

X	Y	Z	Effect
0	-5	0	Force too small resulting in arm deviating from straight line and brick not hitting the wall
0	-25	0	Force too large resulting in arm being uncontrollable and swinging brick up, hitting top of wall
0	-15	0	Could hit the wall more centrally as the launch point is not central, also more force
4	-20	0	Good speed and hits wall centrally to knock most bricks

After tuning the force parameters, DE NIRO successfully destroyed most of the wall. The benefit of controlling the agent by specifying an end effector force as opposed to calculating torques at joint level, is that once the 'EE force to joint torque' code has been written, the calculation of joint torques is done automatically. This means that fewer values need to be adjusted (just force in x, y, and z, as opposed to adjusting 7 different joint torques), and conceptually it is much easier to understand (i.e., it is difficult to know how changing one joint torque will affect the motion of the end effector). Furthermore, there could be many different combinations of joint torques that will yield a single EE force.

In the code, there is a y_stop variable, which specifies the position in the y-axis that the arm is in when it releases the brick. This means that the arm applies the force in the distance between the y position that it picks up the brick in until the position it releases it. There is the potential to build up more acceleration by applying the force over a longer y distance. This can be achieved through moving the brick away from the wall after it is picked up, and then accelerating towards it. To implement this, the following lines of code were added before the while loop which contains the force application.

```
#after picking up the brick, the left arm moves back to this position
xyz = xyz + np.array([0, 0.3, 0])
left_arm.servo_to_pose(xyz, rpy)
```

This moves the brick backwards by 0.3m in the y direction. The y_stop can also be moved closer to the wall to give even more distance for acceleration. This strategy is displayed in

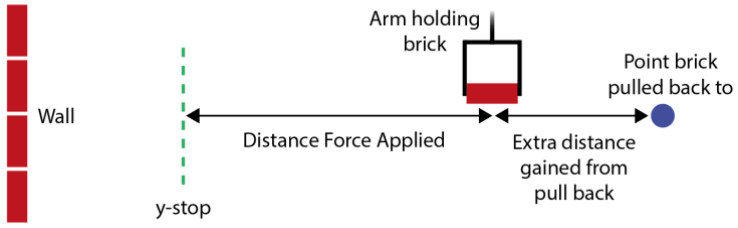


Figure 15: Illustration of pulling the brick back, before acceleration towards the wall

Another method for this was to create two external forces, one to bring the brick backwards, away from the wall, and the other to accelerate it towards the wall. For this method, two y_stops were needed, with the first indicating how far to draw the brick back, before switching to the second force, where the second stop then instructs DE NIRO to release the brick. As seen in the following code, conditional statements were used to implement this method. The reference end effector position (y0) was also updated during the motion. When the brick was pulled back, y0 was set to be at the y_stop1 position, and when it was thrown forward, the reference was set at the point of release (y_stop2). This ensured that the PD controller and the applied forces were not causing contradictory motion.

```
############################ TASK H
    y_stop = -0.0 # the y position to release the brick
    y_stop1 = 0.8
    y_stop2 = -0.0
    Flag = 0
    # change external_force to make DE NIRO throw the brick into the wall
    external_force = np.array([4, -20.0, 0]) # force to accelerate end effector at wall
    external_force1 = np.array([0, 10, 0]) # force during the pull back of brick
    external_force2 = np.array([4, -20, 0]) # force to throw brick after pullback

    if y < y_stop1 and flag == 0: #loop to determine which force to use
        y0 = 0.8 #updating the y-axis reference position
        external_force = external_force1
    elif y >= y_stop1 and flag == 0:
        external_force = external_force2
        y0 = 0.0 #updating the y-axis reference position
        flag = 1
    else:
        external_force = external_force2
```

4.2. Task I: Cleaning the table

Now that the wall has been broken, there are items remaining on the table to be cleared. DE NIRO should be programmed to pick up the yellow sponge and clean the table. External linear force can be applied so that DE NIRO can push the sponge down onto the table as well as moving it horizontally to cover the table area.

A z-axis force of -15 N was applied when wiping in either direction, to ensure that the sponge was constantly pushed down and in contact with the table. A force of -7 N in the y direction was also required to wipe right, and a force of 7 N in the y direction was needed when wiping left, to simulate the wiping motion.

This allowed DE NIRO to achieve the wiping motion. However, it was observed that the sponge was often not flush against the table. The issue with only applying linear force is that it tips the sponge over, because of the contact force between the sponge and the table (the friction). Thus, when the hand moves sideways whilst under this friction, this causes a rotational force that makes the sponge tip. Therefore, counteractive rotational forces or velocities were required to deal with this.

Angular velocity in the derivative control section was added to allow DE NIRO to have better control over the sponge, as illustrated in Figure 16. The addition of x_omega was the most effective term in reducing the rotation, given that tipping most commonly occurred about the x-axis. This was successful in ensuring that the sponge didn't fully flip during the simulation, but it should be noted that there was still some rotation perceivable (non-perfect motion).

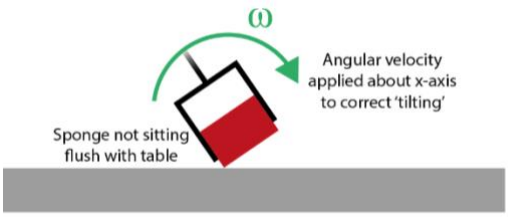


Figure 16: Corrective angular velocity applied to prevent 'sponge tipping'

To achieve this, the derivative control section of the code was edited. Given that there was a large tracking error in the rotation around the x-axis (x_omega), this value was negated and amplified (multiplied by 6), to provide additional external forces to correct this error. The following code shows how the forces and angular velocities were applied:

```
1.          if direction == 'right':
2.          ############ TASK I
3.              external_force = np.array([0.0, -7.0, -15.0, 0, 0, 0]).reshape((6, 1)) # force to
    push down and across on the table
4.
5.              # add proportional and derivative control to keep end effector from deviating
6.              external_force += 400.0 * np.array([0.75 - x, 0.0, 0.0, 0.0, 0.0, 0.0]).reshape((6,
    1))                 # proportional feedback
7.              external_force += 20.0 * np.array([- x_dot, - y_dot * 0.2, - z_dot, -6*x_omega, 0.0,
    0.0]).reshape((6, 1))     # derivative feedback
8.
9.              # convert external force that end effector is applying to joint torques
10.             external_torque = np.matmul(J.T, external_force)
11.
12.             # if the y position goes behond the right hand limit, change direction
13.             if y < y_right_stop or t > 5:
14.                 direction = 'left'
15.                 t = 0
16.                 print 'Wiping left'
17.
18.         if direction == 'left':
19.             ############ TASK I (continued)
20.             external_force = np.array([0.0, 7.0, -15.0, 0, 0, 0]).reshape((6, 1))  # force to
    push down and across on the table
21.
22.             # add proportional and derivative control to keep the end effector from deviating in
    the x direction
23.             external_force += 400.0 * np.array([0.75 - x, 0.0, 0.0, 0.0, 0.0, 0.0]).reshape((6,
    1))                 # proportional feedback
24.             external_force += 20.0 * np.array([- x_dot, - y_dot * 0.2, - z_dot, -6*x_omega, 0.0,
    0.0]).reshape((6, 1))     # derivative feedback
25.
26.             # convert the external force that the end effector is applying to joint torques
27.             external_torque = np.matmul(J.T, external_force)
28.
29.             # if the y position goes beyond the left hand limit, change direction
30.             if y > y_left_stop or t > 5:
31.                 direction = 'right'
32.                 print 'Wiping right'
33.                     t = 0
```

References

1. Zelenak A, Peterson C, Thompson J, Pryor M. The Advantages of Velocity Control for Reactive Robot Motion. In 2015. p. V003T43A003.
2. robotic arm - Solving Inverse Kinematics with Gradient Descent - Robotics Stack Exchange [Internet]. [cited 2022 Mar 17]. Available from: https://robotics.stackexchange.com/questions/9904/solving-inverse-kinematics-with-gradient-descent
3. Bai Y, Gao F, Guo W. Design of mechanical presses driven by multi-servomotor. J Mech Sci Technol. 2011 Sep;25(9):2323–34.

Appendix

A: Additional plots to show the tracking accuracy of velocity control