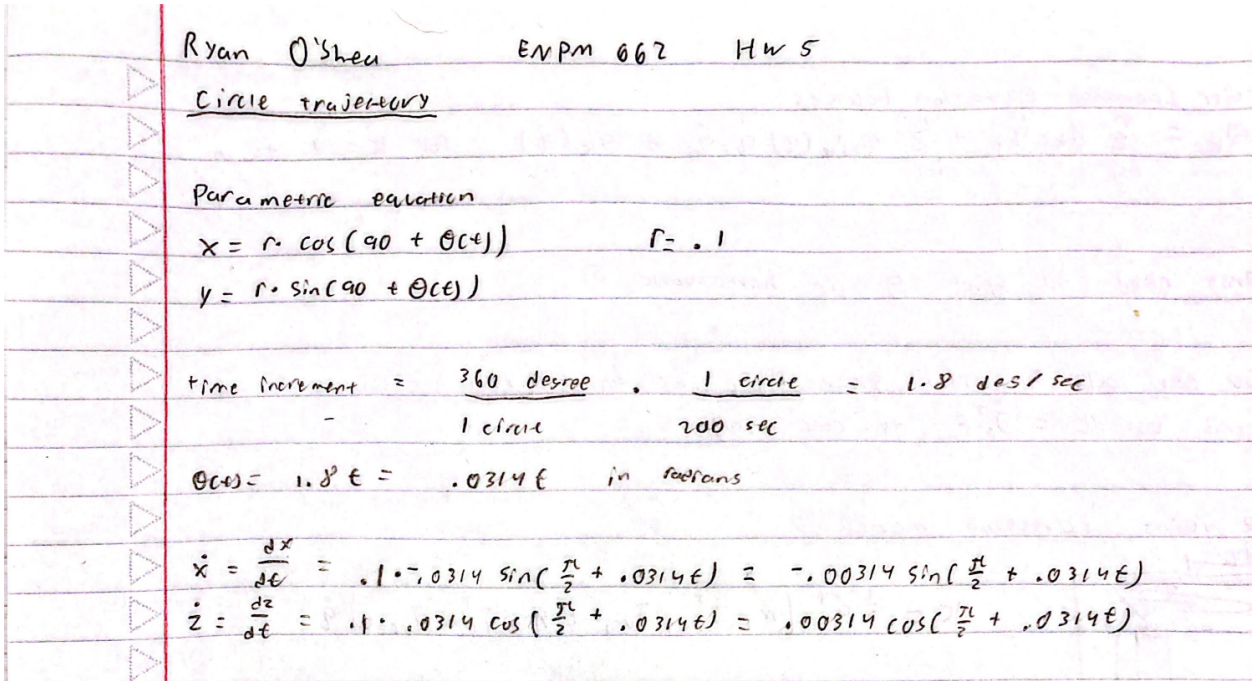
Ryan O'Shea
Roshea
ENPM662

Homework 5 Report

Problem 1

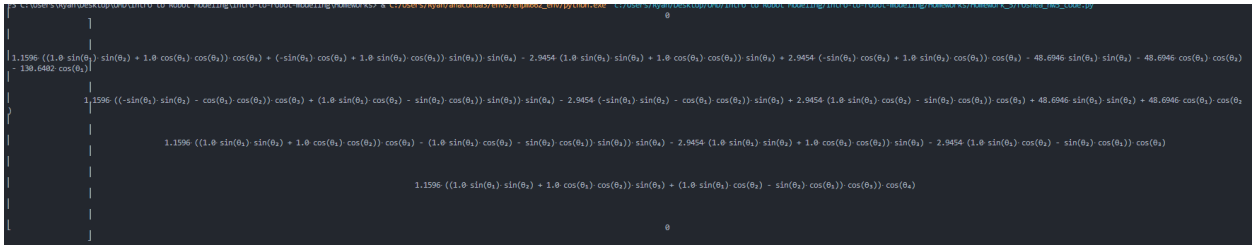
Updated Trajectory

The same parametric circle equations were used to define the path of the end effector over time. The time to complete the trajectory was updated to 200 seconds to find the new version of the equation.



Torque Calculations

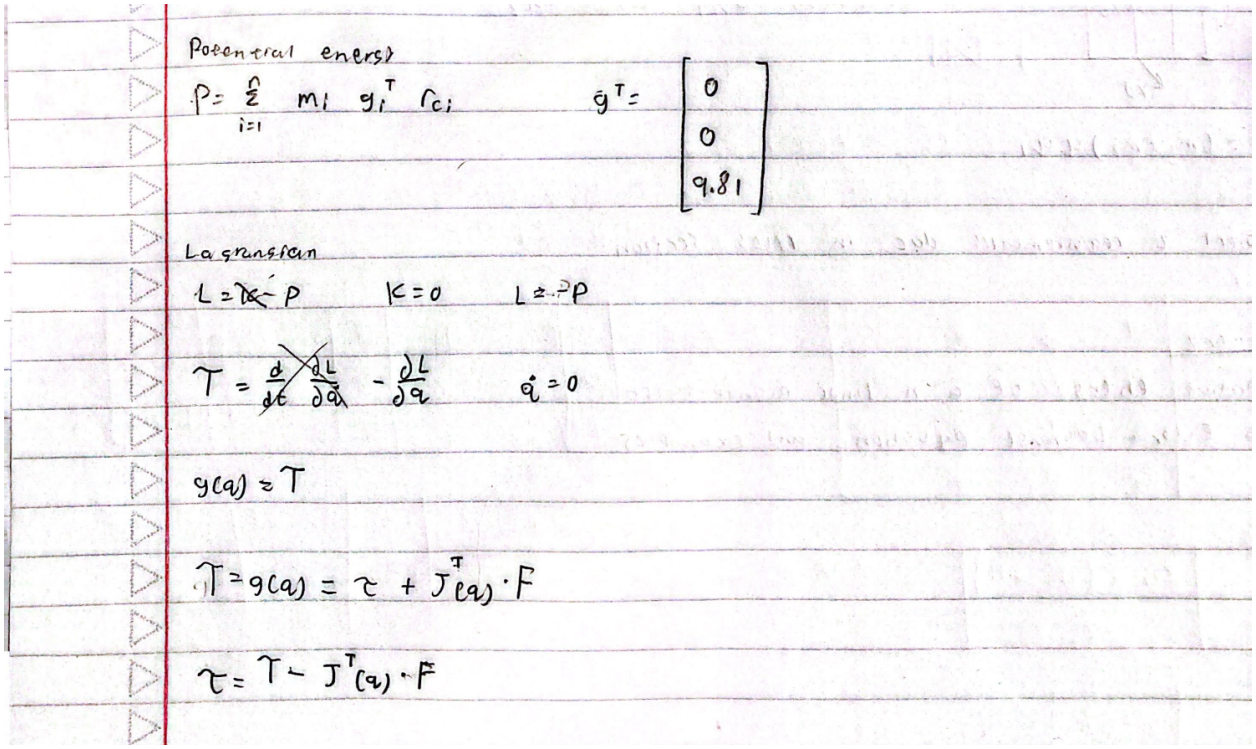
The robot dynamics equation provided on the homework was used with everything but the g(q) term zeroed out because we are assuming a quasi-static system. This also allows us to assume that the kinetic energy of the system at any given moment is also 0. This zeros out K in the Lagrangian equation L = K - P so L = -P. This also simplified the calculation of the Euler-Lagrange equation to $\boldsymbol{T}$ = -partial derivative of L with respect to q. The gravity matrix is then represented by $\boldsymbol{T}$. A printout of the parametric gravity matrix can be found below.



The gravity matrix is a bit difficult to read but it's a 6x1 matrix. The gravity matrix evaluated at the home joint angles can be seen below.

$$\begin{bmatrix} 0 \\ 0.00703207189181691 \\ -0.00311956606529861 \\ 0.000309721208595473 \\ -0.000104669930256063 \\ 0 \end{bmatrix}$$

The joint torques could then be calculated by subtracting the Jacobian transpose * the force vector (-5.0 newtons in the y direction of the origin) frame from $\boldsymbol{T}$. The calculations for the potential energy, Euler-lagrange equation, and joint torque can be seen below.



The code for calculating the potential energy and subsequently the gravity matrix, can be found below. The rci value at each step was calculated by adding translation from from the base to link i and then adding the relative center of mass values to that translation. The sympy diff function is used to calculate the partial derivative of the expression for potential energy for each of the joint angle variables theta 0-5. The theta

values calculated at each timestep are substituted in for the theta variables to evaluate the gravity matrix at a specific joint configuration.

```python
# Calculates the potential energy for each link of the system and then converts it the gravity matrix
def calcPotEnergy(self):
    total_pot_energy = 0
    # Define the gravity vector
    g_vec = np.array([0, 0, 9.81]).transpose()
    # Zip the link masses, center of masses, and the 0 to i transformation matrices and iterate through them
    for link_mass, link_cm, trans_mat in zip(self.link_masses, self.link_cms, self.var_successive_trans_mats):
        # Extract the translation vector
        t_vec = [trans_mat.row(i)[3] for i in range(3)]

        # Calculate rci by adding the link center of mass location to the translation vector
        rci = np.array([link_cm[i] + t_vec[i] for i in range(len(link_cm))]).transpose()

        # Calculate potential energy for the link and add it to the total
        total_pot_energy += link_mass*(np.dot(g_vec, rci))

    # Calculate the partial derivative of the potential enerty wrt each of the joint angles
    pot_energy_partials = sympy.Matrix([sympy.diff(total_pot_energy, var) for var in self.theta_list])


    if self.first_run:
        sympy.pprint(roundExpr(pot_energy_partials, 4))
        self.first_run = False

    # Create a list of substitution pairs of the theta variable and the actual theta value
    substitutions = [(theta_var, theta_val) for theta_var, theta_val in zip(self.theta_list, self.theta_val_list)]

    # Substitute the values in to get the actualy value of the potential energy partial derivative matrix
    # Take its negative since the Lagrangian is K - P
    self.gravity_mat = -pot_energy_partials.subs(substitutions)
```

The link masses, lengths, and center of mass coordinates were set up as a list and list of lists as shown below. The values were taken directly from the universal robotics page here: https://www.universal-robots.com/articles/ur/application-installation/dh-parameters-for-calculations-of-kinematics-and-dynamics/ . The z value of joint 2, table entry 3, was set to negative because they have their Z axis facing the opposite direction from what I set mine as.

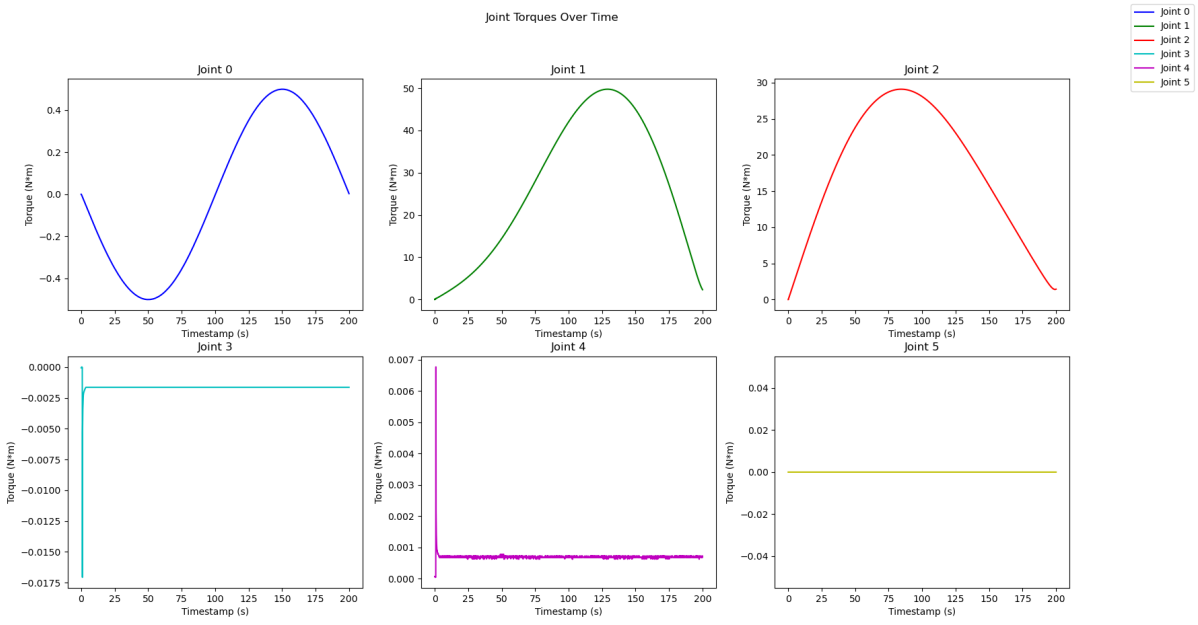
```python
# Physical properties for dynamics
self.link_masses = [7.369, 13.051, 3.989, 2.1, 1.98, 0.615]
# These are the relative center of masses for a link. Their
self.link_cms = [[0.021, 0.0, 0.027],
                 [0.38, 0.0, 0.158],
                 [0.24, 0.0, -0.068], # The Z value is negat
                 [0.0, 0.007,0.018],
                 [0.0, 0.007,0.018],
                 [0.0, 0.0, -0.026]]
```

Finally, the joint torques are calculated using the code below. The joint torques are produced as a 6x1 sympy matrix and recorded at each timestep to be graphed later.

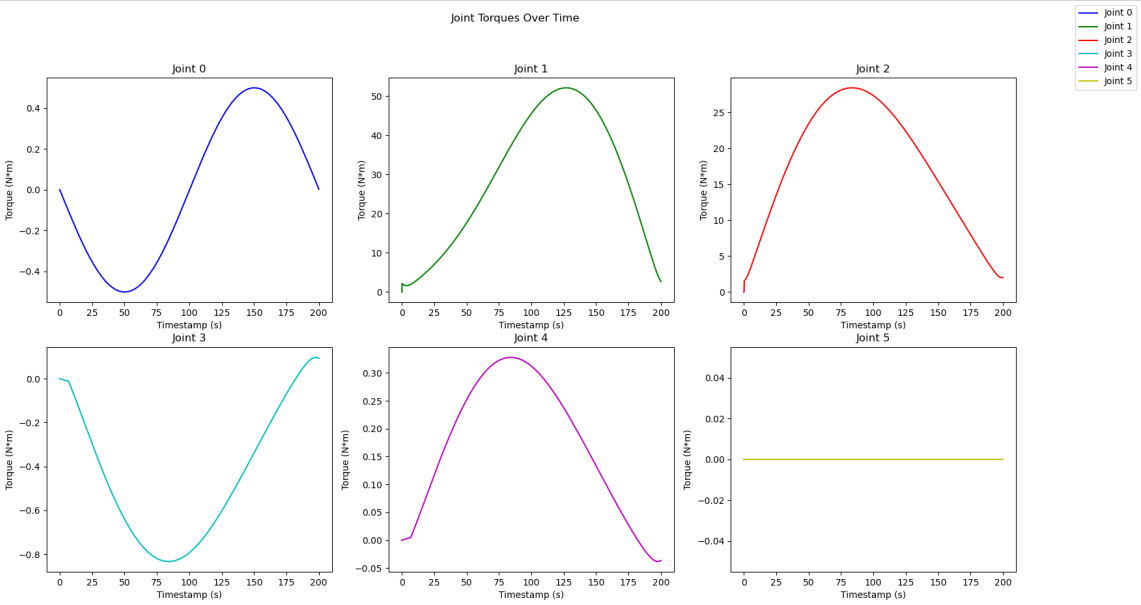
```python
# Calculates the torque at each joint based on the gravity matrix, jacobian, and external force vector at the end effector
def calcJointTorques(self):
    force_vec = sympy.Matrix(np.array([0.0, -5.0, 0.0, 0.0, 0.0, 0.0]).transpose())
    joint_torques = self.gravity_mat - self.jacobian_T*force_vec

    return joint_torques
```

The joint torques are stored at every timestamp similar to the end effector position and the joint angles. The torques are plotted against the timestamps displayed on 6 different graphs as shown below. The plots match the expected behavior of the system as joints 1 and 2 and likely the main movers of the system to produce the circular trajectory due to their axis of rotation being perpendicular to the drawing plane. The torque on these joints first increases as they move towards being horizontal which creates a longer lever arm for the mass of the subsequent joints to act upon which in turn increases the torque experienced by the joint. The torques on joints 1 and 2 then decrease for the remainder of the timesteps as the links become more vertical and their lever arms become shorter. Joint 0 seems to experience a perfectly symmetrical torque during the trajectory which is likely due to it being the main joint that's pushing the pen into the wall. The torques experienced by joint 3 and 4 seem to mostly be impulses in the very beginning of the trajectory followed by sustained very small values. This could be due to the minor epsilon that was added to all the initial joint angles of the system to smooth out the trajectory of the end effector. Joint 5 is the end effector joint and experiences no torque as it doesn't move at all or support any weight above it.

2000 timestamps with initial joint angle epsilon addition

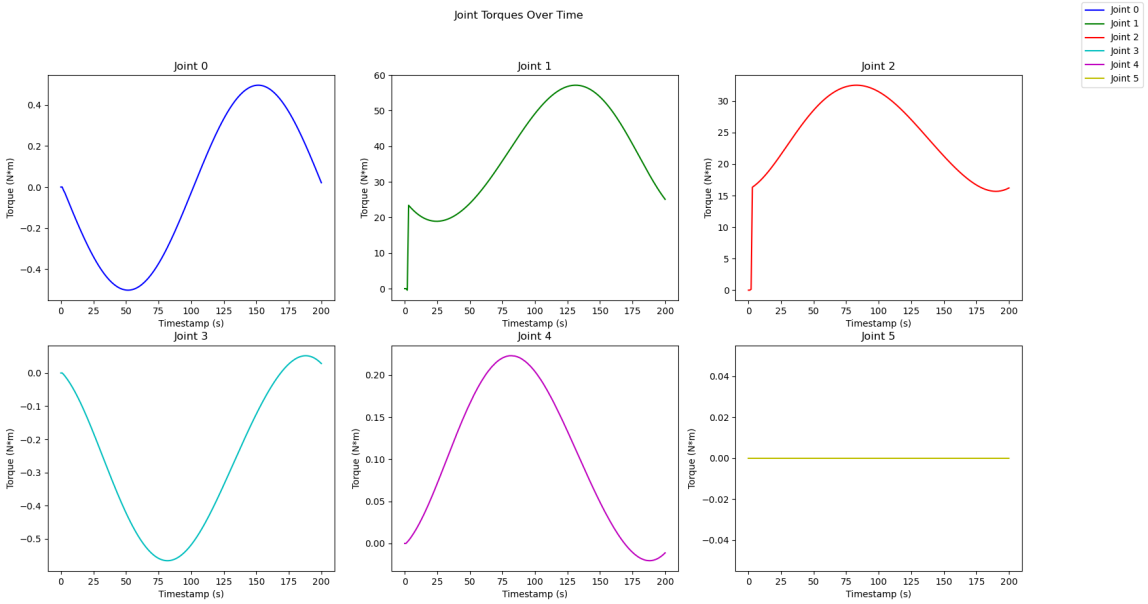


2000 timestamps without initial joint angle epsilon addition

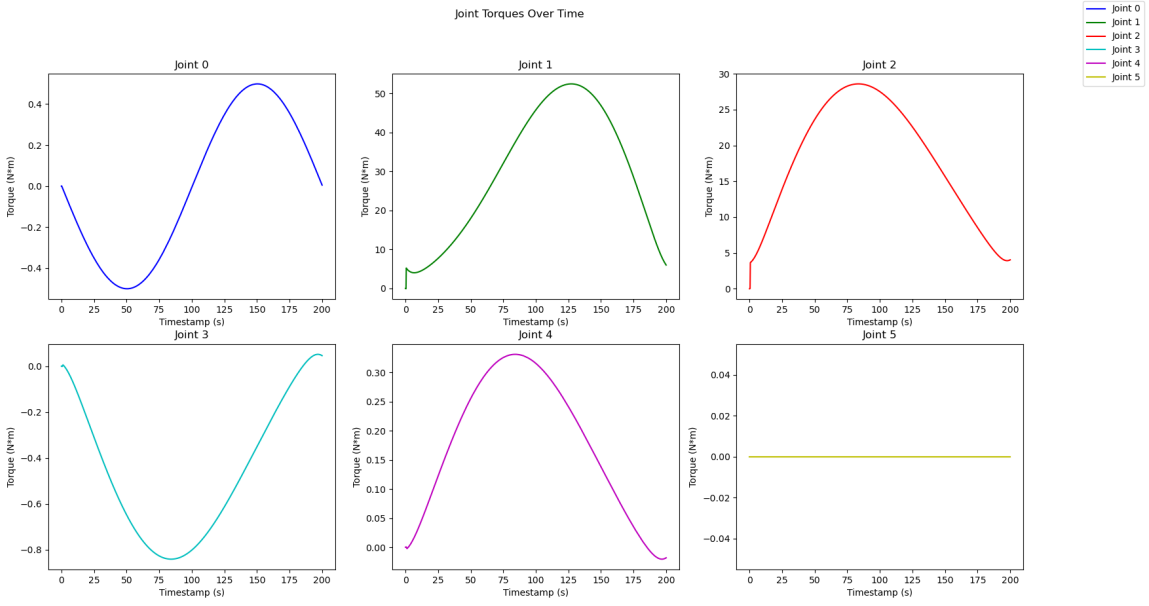


Similar to the trajectory generation, the quality of the joint torque calculations over time increases with the number of timestamps used because it limits the effect larger instantaneous velocities can have on calculations. Examples of torques plots at lower amounts of timestamps can be seen below. The graphs are generally less smooth and produce slightly different results than the one shown above with 2000 timestamps. The magnitude of the torques will also change between runs which I believe can also be attributed to the small random epsilon that is being applied to the starting joint angles during each run of the code. When the code for the random epsilon addition was removed all produced graphs came out with the same torque magnitudes.**The random epsilon addition to the starting angles has been turned off for the code submission as it causes an incredibly rare bug that throws off the trajectory**.

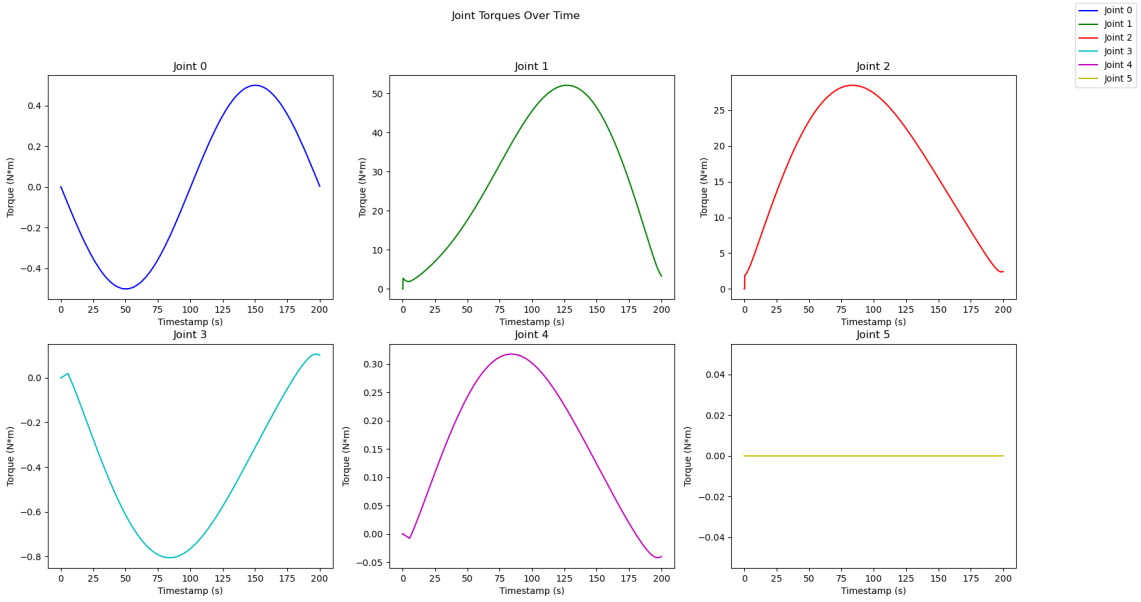
200 timestamps



1000 timestamps



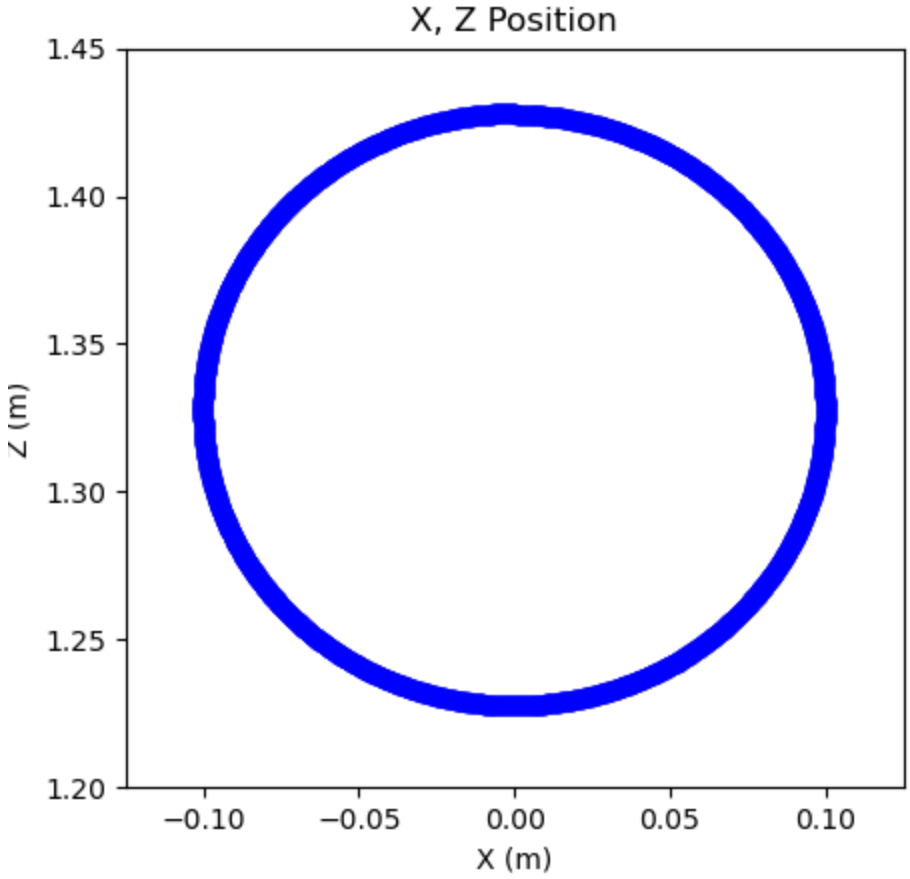
1500 timestamps



The plotting code for the torque graphs is shown below. There is an option to plot all 6 joint torques over time on a single graph by setting the plot_single_torque_graph to true. This produces a clean plot but due to the large scale differences between the torques of the different joints it is not recommended and was mainly created for debugging purposes. The variable is set to false by default but the code is kept for future debugging purposes. The 6 graph plot uses the matplotlib subplots feature and assigns a unique color to each of the plots to make them easier to distinguish between.

```python
joint_names = ["Joint 0", "Joint 1", "Joint 2", "Joint 3", "Joint 4", "Joint 5",]
# Plot all the torques on a single graph
if plot_single_torque_graph:
    for single_torque_list, joint_name in zip(torque_list, joint_names):
        plt.plot(timestamps, single_torque_list, label=joint_name)

    plt.title("Joint Torques Over Time")
    plt.xlabel("Timestamp (s)")
    plt.ylabel("Torque (N*m)")
    plt.legend()

    plt.show()
# Otherwise make six subplots and plot the torques separately
else:
    # Iterable list of plot colors
    plot_colors = iter(['b', 'g', 'r', 'c', 'm', 'y'])

    n_rows = 2
    n_cols = 3
    fig, ax = plt.subplots(nrows=2, ncols=3)
    for idx, (single_torque_list, joint_name) in enumerate(zip(torque_list, joint_names)):
        # Calculates row and col number for each plot based on number of rows and cols
        row_num = 0 if idx <= n_rows else 1
        col_num = idx - n_cols*row_num
        ax[row_num, col_num].plot(timestamps, single_torque_list, label=joint_name, color=next(plot_colors))
        ax[row_num, col_num].set_xlabel("Timestamp (s)")
        ax[row_num, col_num].set_ylabel("Torque (N*m)")
        ax[row_num, col_num].set_title(joint_name)

    fig.suptitle("Joint Torques Over Time")
    fig.legend()
    plt.show()
```

During standard running of the code the circular trajectory is still plotted first before the joint torques graph. This graph can just be closed to then bring up the plot of the torques. An example image of the trajectory can be seen below.


X, Z Position
1.45
1.40
1.35
1.30
1.25
1.20
Z (m)
−0.10
−0.05
0.00
0.05
0.10
X (m)