Chess Robot

Introduction

The goal of this project is to design a control system for a pick and place robot. The specific system that was designed was a four degree of freedom robot arm capable of playing chess. The arm is designed to move chess pieces precisely from one space on the board to any other. A 3D model of this robot can be seen in Figure 1. To do this, a predefined trajectory was acquired for the end effector and the individual joint trajectories were calculated using inverse kinematics. The main challenge involved in designing a controller for such a system was the fact that it was nonlinear. This was overcome through the implementation of the computed-torque linearization law. Once the system was linear it was simple to apply a controller. For this system both a PD and PID controller were investigated and compared.

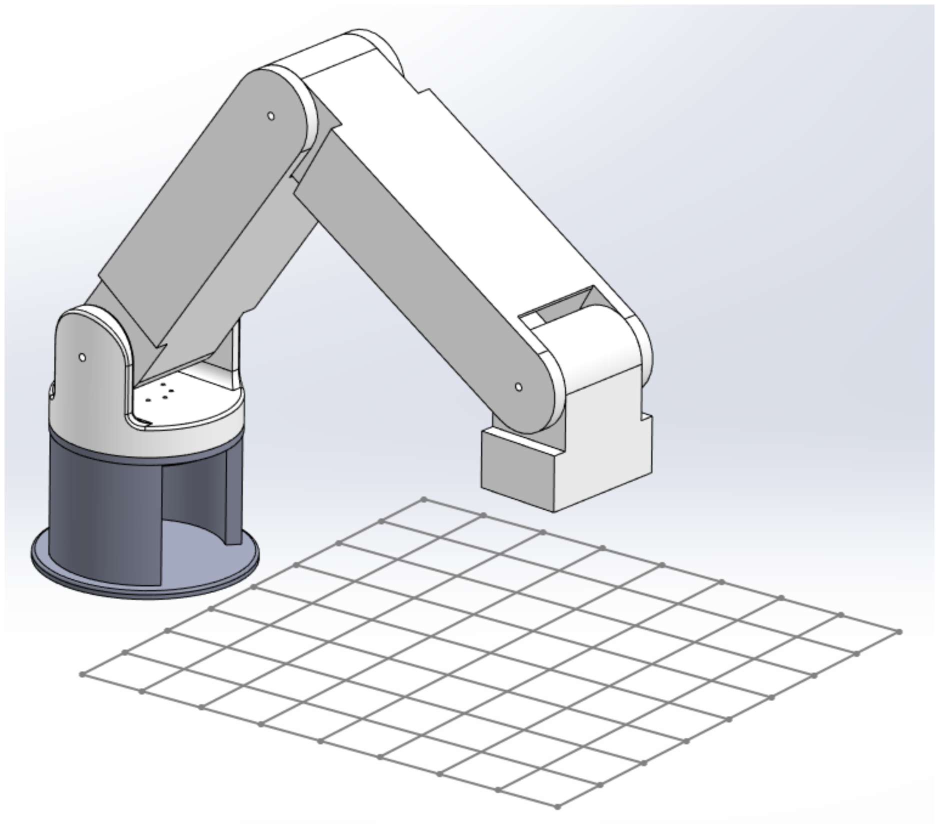

Figure 1: Example 3D model for chess robot.

Background

When designing a controller for a robotic arm there are two main approaches. The first is operational space control which looks at the end effector pose as the input to the system. The second is joint space control which relies on joint states to act as the input to the controller. This is the type of controller that was implemented for this system. One of the major similarities between robotic arms that implement these types of controllers is the need to linearize the system. This is commonly done through the implementation of the computed-torque law. This type of feedback linearization allows for the dynamics of the robotic arm to be represented as a linear system and simplifies the process of applying a controller.

System Dynamics

The system dynamics for our robotic arm were assumed to follow the general robotic arm dynamical model, not considering the effects of friction on the system. The states are defined as the four control angles and their angular velocities, and the inputs to the system are the four torques applied by each motor at their respective joints.

$$\theta = \begin{bmatrix}\theta_1\\ \theta_2\\ \theta_3\\ \theta_4\end{bmatrix}, \quad \dot{\theta} = \begin{bmatrix}\dot{\theta}_1\\ \dot{\theta}_2\\ \dot{\theta}_3\\ \dot{\theta}_4\end{bmatrix}, \quad \tau = \begin{bmatrix}\tau_1\\ \tau_2\\ \tau_3\\ \tau_4\end{bmatrix}$$

$$\tau = H(\theta)\ddot{\theta} + C(\theta,\dot{\theta})\dot{\theta} + g(\theta) = M(\theta)\ddot{\theta} + N(\theta,\dot{\theta})$$

In the above nonlinear dynamics equation, *H* corresponds to the inertia of the system, *C* corresponds to the centrifugal and Coriolis effects of non-Newtonian reference frames at the moving joints, and *g* corresponds to the effects of gravity acting on the system. After formulation, the matrices were combined as *H* = *M* and $C\dot{\theta} + g$ = *N* so that the dynamics would take a form similar to that found in our research. *H*, *C*, and *g* were calculated using the method taught in MAE547 whereby the system is broken up into a collection of point masses at the center of each link and motor, with a known moment of inertia for each component. The method involves several Jacobian matrices and partial derivatives, and the general guiding equations are shown below. See the attached MATLAB code for the full derivation.

$$H(q) = \sum_{i=1}^{n} \left(m_{li} J_p^{(l_i)T} J_p^{l_i} + J_o^{(l_i)T}(R_i I_{l_i}^i R_i^T) J_o^{l_i} + m_{mi} J_p^{(m_i)T} J_p^{m_i} + J_o^{(m_i)T}(R_i I_{m_i}^i R_i^T) J_o^{m_i} \right)$$

$$C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \qquad c_{ij} = \sum_{k=1}^{n} c_{ijk}\dot{q}_k \qquad c_{ijk} = \frac{1}{2}\left(\frac{\delta b_{ij}}{\delta q_k} + \frac{\delta b_{ik}}{\delta q_j} + \frac{\delta b_{jk}}{\delta q_i} \right)$$

$$g_i(q) = -\sum_{j=1}^{n} \left(m_{lj} g_0^T j_{p_i}^{l_j}(q) + m_{mj} g_0^T j_{p_i}^{m_j}(q) \right)$$

Trajectory Planning

In order for the problem to be as realistic as possible, real-world inputs needed to be defined based on the desired end-effector motion in cartesian space. This was achieved by positioning the base of our virtual robotic system centered at one end of the chess board. The robot base is at the origin of a cartesian coordinate frame, with the board columns oriented in the y direction and the rows oriented in the x direction. The centers of each square of the board can be represented as an array of points spaced 6cm apart (this value is very close to the imperial equivalent to the standard chess square size), and the origin is located 2 squares below the first row of the board.

Cartesian waypoints in time can then be established based on an assumed starting position and movements between pieces on the board. All motions are carried out first in the z-axis, and then simultaneously in the x and y axes. The chosen waypoints represent starting in an initial condition, moving up to a clearance height of z = 15cm, moving over to the d2 pawn, moving down to pick up the pawn at an interface height of 5cm (the standard height for a pawn), raising the pawn up to the clearance height, translating it to the space above the d4 square, lowering it to the square, and then raising back up to the clearance height (this is the start of the ever-popular

Queen's Gambit opening). In order to create a twice-differentiable path from one waypoint to another, each move is assigned a duration, and the gaps between the points are filled with linear segments with parabolic blends. 0.5s pauses are also employed between motions in an attempt to reduce jerk. The resulting Cartesian input is shown below in Figure 2.

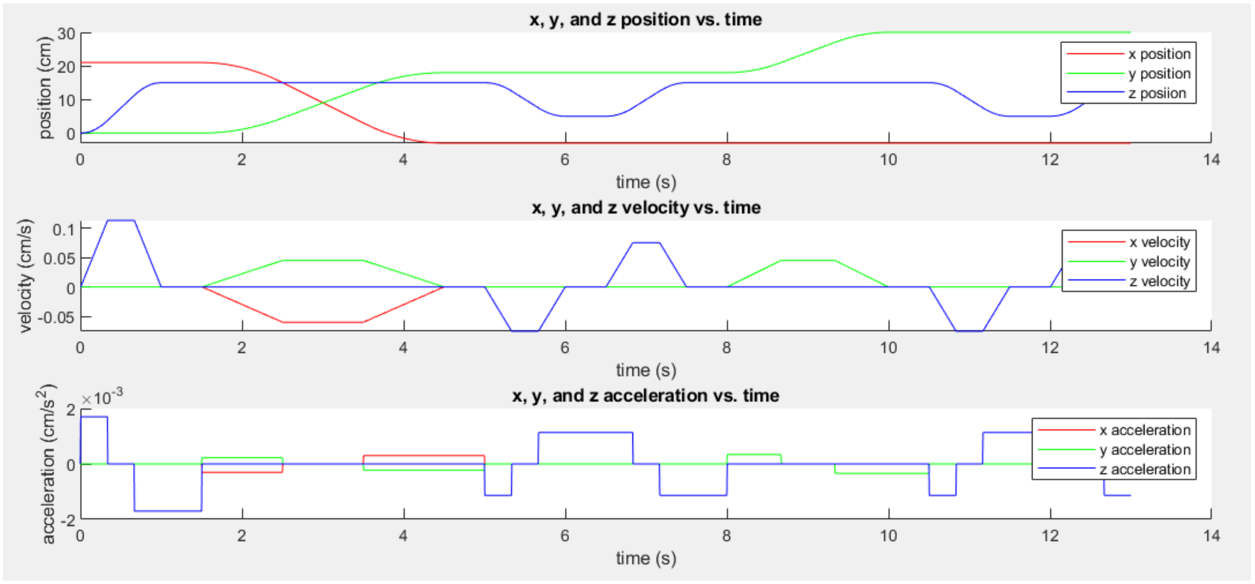

Figure 2: Cartesian position, velocity, and acceleration inputs

Part of what makes this approach novel is the use of a closed-form inverse kinematics (IK) solution to precompute joint-space trajectories from Cartesian paths. Many higher-DOF serial chain robots do not have closed-form IK solutions, but it is possible for our system because coordinate transformations and end effector angle restrictions allow for some simplifications. Because of this, the IK does not need to be computed online by inverting the Jacobian and integrating, which increases accuracy and saves on computation.

To start, the cartesian trajectories (x,y,z) are converted into cylindrical coordinates (θ,p,z) through the following equations:

$$\theta = atan2(y, x)$$
$$p = \sqrt{x^2 + y^2}$$

Once in this form, a few helpful, intermediate geometrical parameters can be described based on the Cartesian path and link lengths l_1 through l_4:

$$z' = z + l_4,\ d = \sqrt{p^2 + (z' - l_1)^2},\ w = atan2((z' - l_1), p)$$

Using these quantities, the inverse kinematics can be directly expressed:

$$\theta_1 = \theta$$
$$\theta_2 = -w + cos\left(\frac{l_2^2 + d^2 - l_3^2}{2l_2d}\right)$$
$$\theta_3 = -\theta_2 - cos^{-1}\left(\frac{p - l_2cos(\theta_2)}{l_3}\right)$$
$$\theta_4 = -\frac{\pi}{2} - \theta_2 - \theta_3$$

These calculations can then be numerically differentiated to fully describe the angular positions, velocities, and accelerations of the joint-space trajectory, which is shown below in Figure 3.

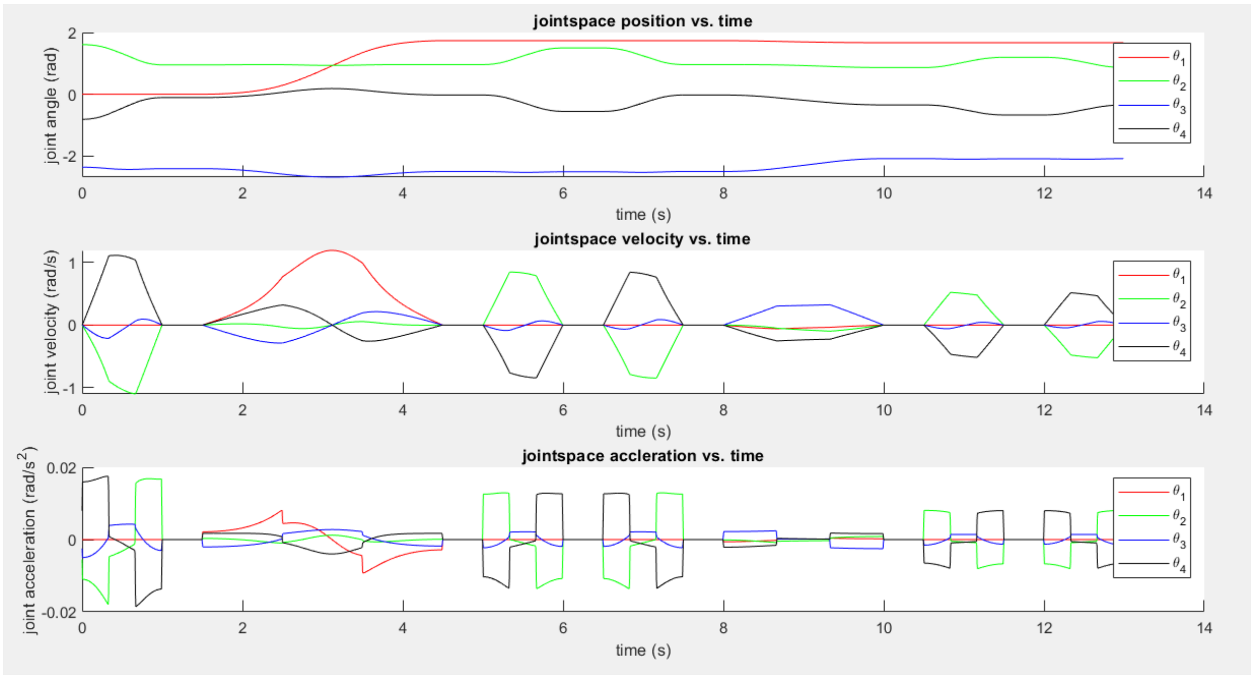

Figure 3: Joint-space trajectory that will serve as the input to the controller

Computed-Torque Linearization

In order to simplify this nonlinear system a feedback linearization technique named computer-torque linearization was implemented. This technique involves utilizing the robot dynamics in a feedforward loop to eliminate nonlinearities in the system. This makes the system much easier to design a controller for and opens up many more options for controllers that only work for linear systems.

To implement this linearization the system the tracking error is defined.

$$e(t) = \theta_d(t) - \theta(t), \qquad \dot{e}(t) = \dot{\theta}_d(t) - \dot{\theta}(t), \qquad \ddot{e}(t) = \ddot{\theta}_d(t) - \ddot{\theta}(t)$$

Where θ represents the true angle of a joint, θ_d represents the desired joint angle, and e is the tracking error. This equation is differentiated to find both the tracking error in the joint velocity and acceleration. This is then used to define the Computed-Torque Law.

$$M\left(\ddot{\theta}_d - u\right) + N = \tau$$

Where M and N are matrices that represent the dynamics of the arm, τ is the torque computed for the system, and u is the feedback control input for the outer loop. This equation along with the robot dynamics equation give us the following equality for the joint acceleration tracking error.

$$\ddot{e} = u$$

Because of this equality u simply needs to be designed to go to zero in order for the tracking error to go to zero. For this project we used a PD and PID controller to define u.
The results of this linearization can be seen in Figure 4 where the nonlinear inner loop can be seen to be a result of the combination of the M and N matrices to calculate the torque. This results in the linear system that the controller is applied to.

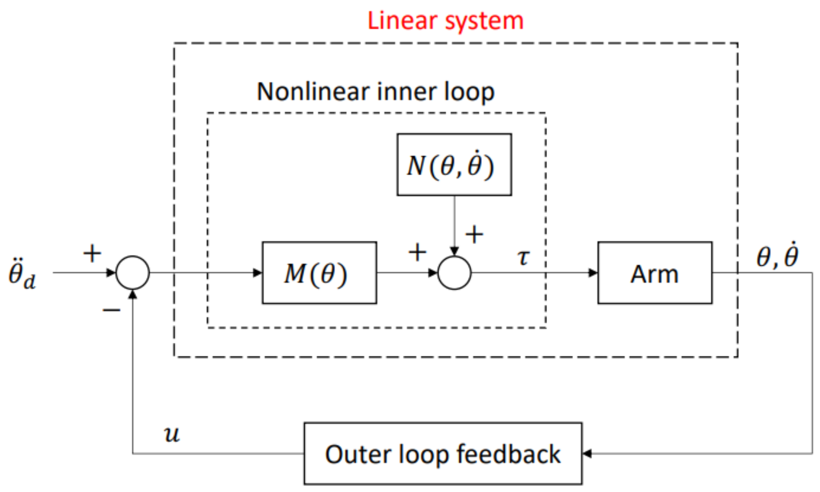

Figure 4: Outline of the computed-torque linearization inside of the nonlinear inner loop box. [1]

Controller Design

After the linearization of the system is established, nearly any controller can be applied. For this system both a PD and PID controller were implemented. For the PD controller, the input u was defined as follows.

$$u = -k_d\dot{e} - k_p e$$

Where k_d and k_p are controller gains used to tune the response of the system. This definition of u is then substituted into the computed-torque linearization.

$$M\left(\ddot{\theta}_d + k_d\dot{e} + k_p e\right) + N = \tau$$

The gains for the PD controller were selected to be 20 for k_d and 400 for k_p. These gains were selected through trial and error comparison. All of this is represented in the diagram in Figure 5.

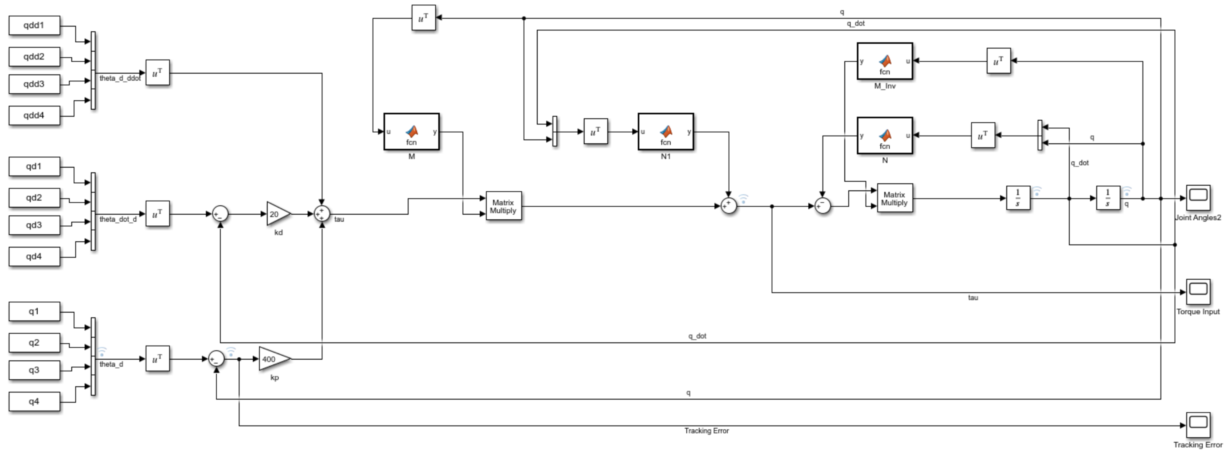

Figure 5: Diagram of PD controller alongside computed torque linearization applied to a 4 DoF robotic arm.

In this figure the twelve desired joint characteristic inputs are seen on the left. They then pass through the PD feedback loop and inside of that the computed-torque feedforward loop.

The PID controller was designed in a very similar way. For the controller u was defined below.

$$u = -k_d\dot{e} - k_p e - k_i \int e$$

Where k_i was added to the PD definition to account for the integrating term. This was then substituted into the computed-torque equation.

$$M\left(\ddot{\theta}_d + k_d\dot{e} + k_p e + k_i \int e\right) + N = \tau$$

The gains for the PID controller were selected to be 25 for k_d, 600 for k_p, and 75 for k_i. As with the PID controller, these gains were selected through trial and error comparison of the tracking error. Using this definition the PID controller and linearization were laid out in the diagram in Figure 6.

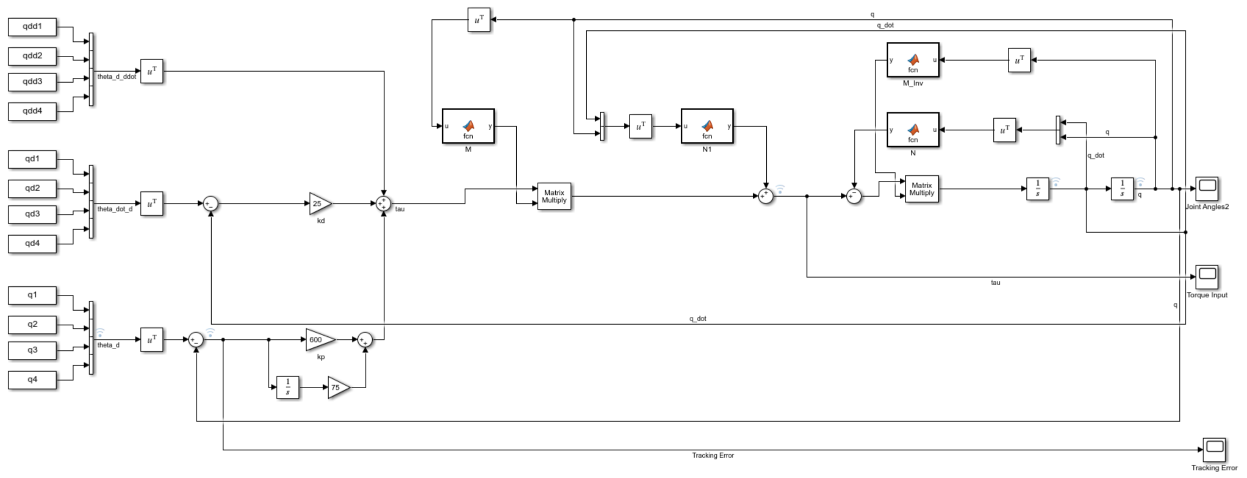

Figure 6: Diagram of PID controller alongside computed torque linearization applied to a 4 DoF robotic arm.

This model functions very similarly to the PD controller with the same impetus being passed in on the left and the linearization taking place directly after the PID feedback loop. The only difference between the two is the addition of the integrating term in the bottom left of the diagram.

Analysis

After running the simulink simulation the outputs of the different controllers were compared. The results of the PD controller can be seen in Figure 7.

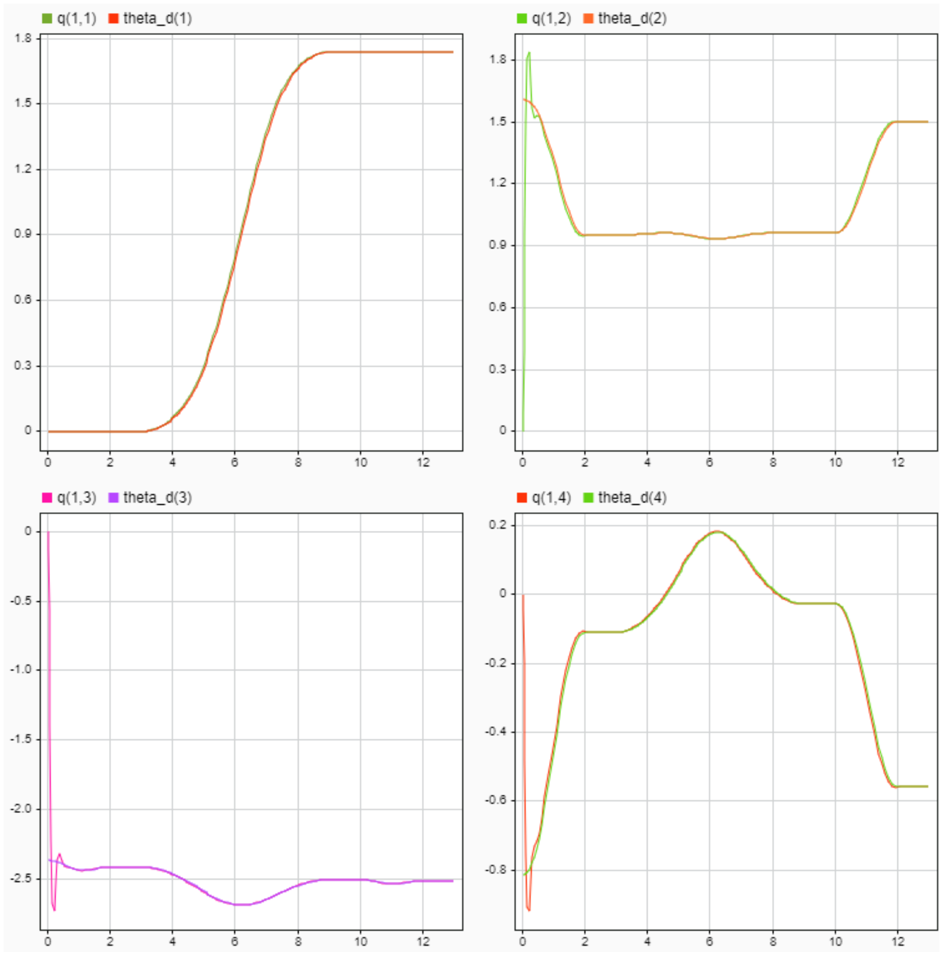

Figure 7: All four joint angle outputs plotted alongside desired joint angle trajectories for PD controller.

Here the desired joint angle is plotted alongside the actual joint angle. It can be seen that the controller quickly matches the desired trajectories. Figure 8 shows the torque that each joint would require to obtain the desired trajectory.

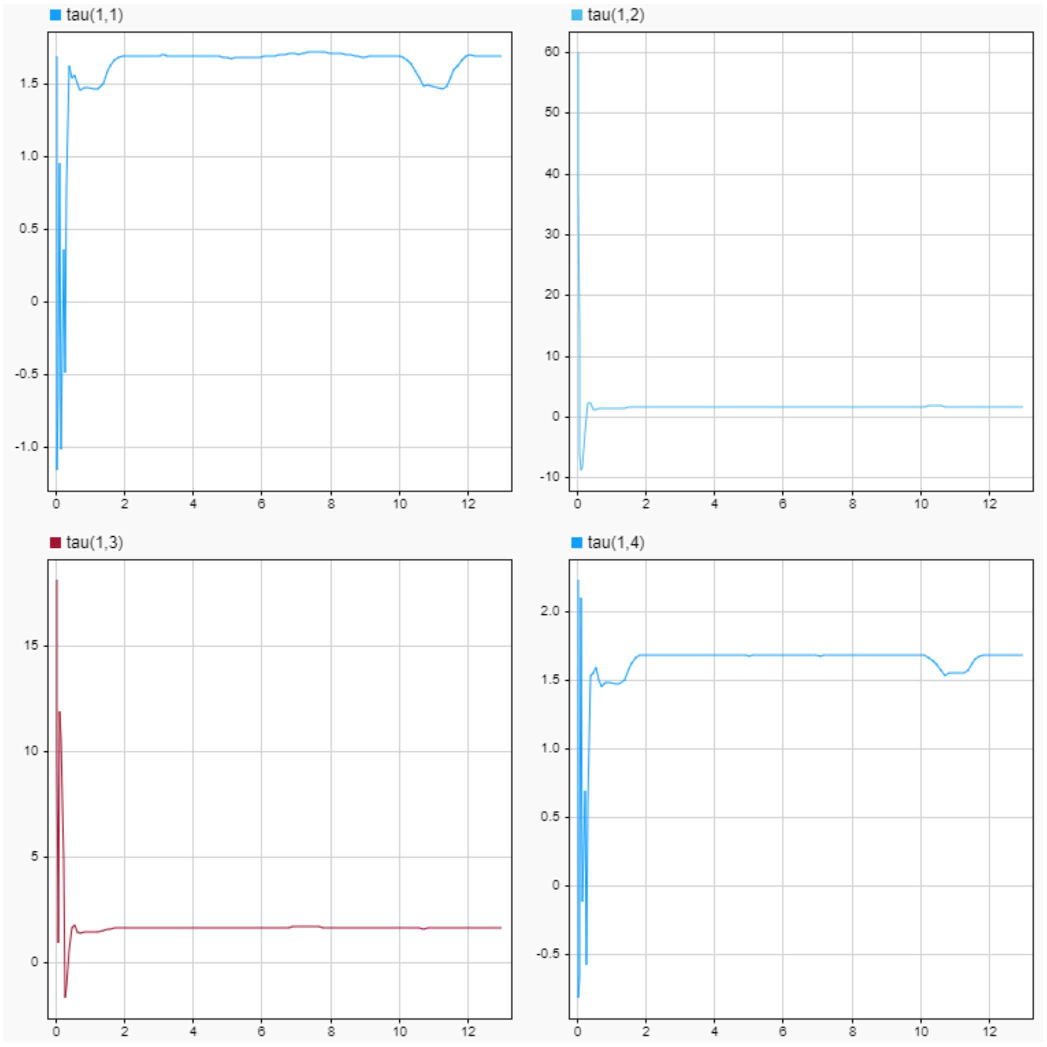

Figure 8: All four joint torques plotted for PD controller.

After some initial noise at the start where the joints are jumping to their starting position, the torque has very little variation. Additionally, all of the values are within reason for the size of the system. The results for the PID controller were similar. The plots for this controller can be seen in Figure 9.

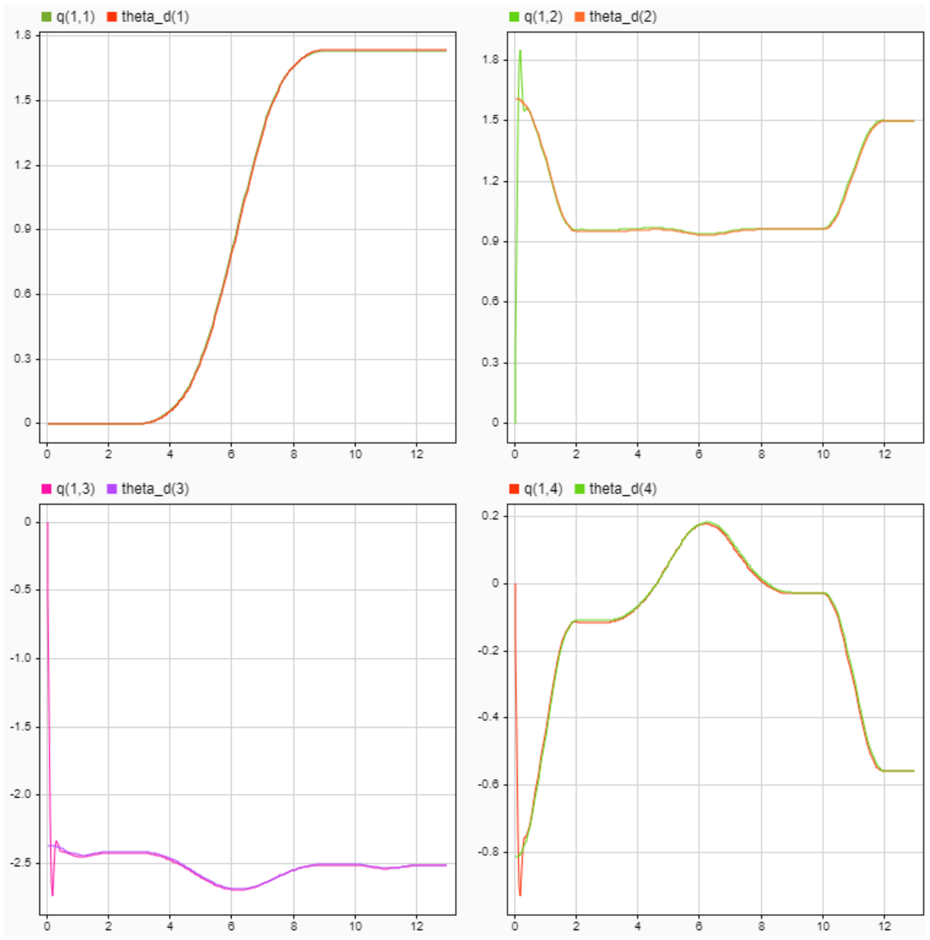

Figure 9: All four joint angle outputs plotted alongside desired joint angle trajectories for PID controller.

Similarly to the PD controller, the PID controller demonstrated a very quick response and very accurate tracking of the desired joint angles. Also similar to the PD controller the torque for each joint was plotted in Figure 10.

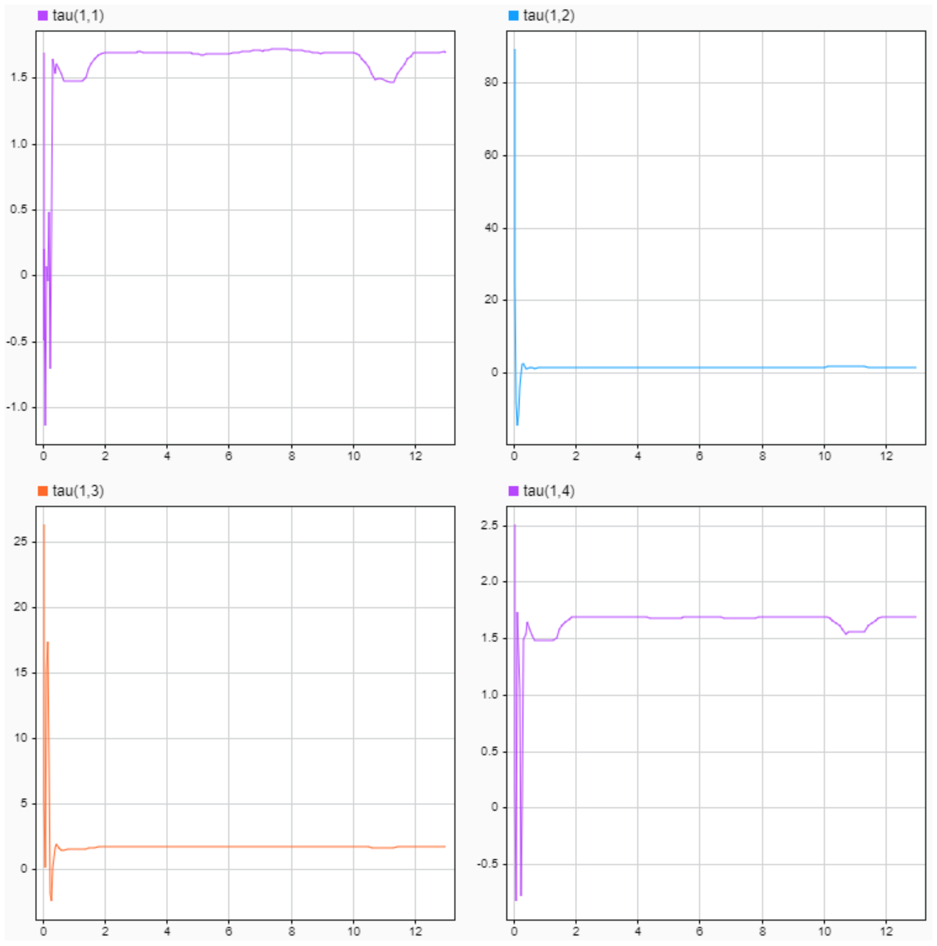

Figure 10: All four joint torques plotted for PID controller.

In these plots the torque can be seen to converge in a similar way to the PD controller. The noise at the start from the jump to the initial values quickly shrinks to a torque with very low variation. In order to compare the results of the two systems the tracking error from one of the joints was plotted in Figure 11.

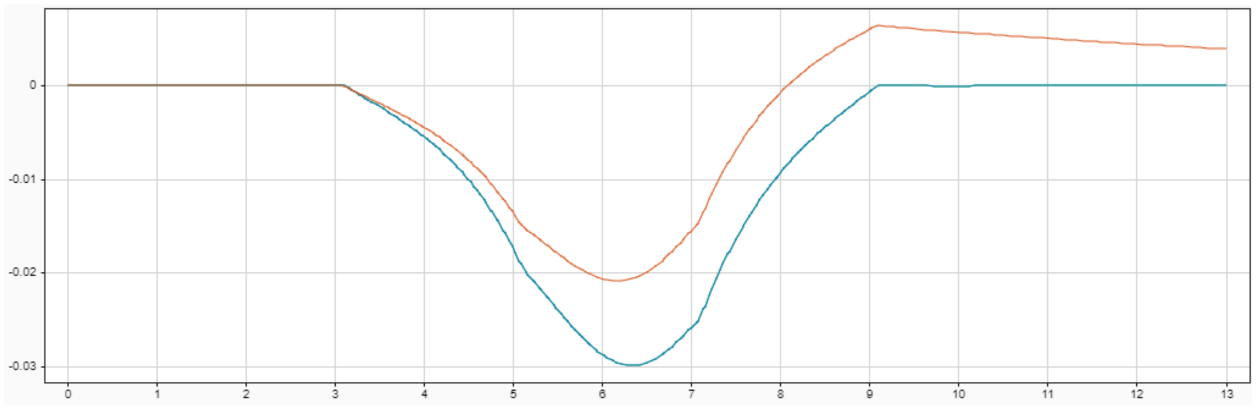

Figure 11: Tracking error for PD (blue) and PID (red) controllers.

From this plot, the PID can be seen to have a larger overall tracking error than the PD which reaches a stable value much faster. Based on this the PD controller will most likely be more accurate for moving something like a chess piece precisely.

Conclusions and Next Steps

The main goal when designing a controller for this 4 degree of freedom robotic arm was first to focus on linearizing the system and then apply a controller to that linear system. The PD and PID controllers that were selected were simple because of this but yielded excellent results. The most difficult part of implementing these controllers was selecting the gains. Since this was done by trial and error comparison and the simulink simulations required a long time to run this took a fair bit of time. This could be improved through the implementation of some desired response characteristics to solidify the gain values.

The biggest lesson learned from this project was how to linearize a nonlinear system through the use of feedback linearization, specifically computed-torque linearization for robotic arms. Additionally, throughout this project the team learned to use Simulink for the first time. This tool was extremely helpful when simulating this system and will be a great skill to have in the future.

Moving forward there are a few improvements we would like to make to the design of this system. First, we would like to apply a controller that can deal with the varying dynamics of the robotic arm. When a chess piece is picked up by the arm the mass in that link changes and thus the dynamics equations currently used to control the arm become invalid. To compensate for this we would like to implement some form of robust controller scheme. It might also be beneficial to compare other methods of system linearization, possibly through an SAA controller. Another area that should be looked into before building a physical system would be how the system handles sensor noise. To solve this problem we would implement a Kalman filter. These are the major areas of study that would be looked into moving forward with this system.

References

[1] Jantsch, Michael, et al. A Scalable Joint-Space Controller for Musculoskeletal Robots with … from https://mediatum.ub.tum.de/doc/1287164/676273.pdf.

[2] Park, S. (2018). Advanced Robotics - Computed-Torque Control [pdf]. from https://www.daslhub.org/unlv/courses/me729-sp/week13/lecture/Note_09_Computed_Torque_Control.pdf.

[3] Tumeh, Z. S. “CSDL | IEEE Computer Society.” Digital Library, 1988, from www.computer.org/csdl/proceedings-article/isic/1988/00065473/12OmNvpw7eg.

[4] Uebel, M., et al. “CSDL | IEEE Computer Society.” Digital Library, 1992, from https://www.computer.org/csdl/proceedings-article/robot/1992/00220238/12OmNyqRnlz.

Appendix

Team Member Contribution

Connor Nail: PD and PID controller design, Computed-torque linearization, System dynamics definition
Matthew Nolan: System dynamics definition, Trajectory planning / Inverse Kinematics

Dynamics Calculator MATLAB Code

```
syms t1 t2 t3 t4 td1 td2 td3 td4

a1 = 0.0635; a2 = 0.3048; a3 = 0.3048; a4 = 0.09525;
l1 = a1/2; l2 = a2/2; l3 = a3/2; l4 = a4/2;
ml1 = 0.3447302; ml2 = 0.5261671; ml3 = 0.5261671; ml4 = 0.2585477;
Il1 = [0.000708188,0,0;0,0.001290541,0;0,0,0.001205675]; Il2 =
[0.007064321,0,0;0,0.001363701,0;0,0,0.006558055]; Il3 =
[0.007064321,0,0;0,0.001363701,0;0,0,0.006558055]; Il4 =
[0.00072282,0,0;0,0.000526751,0;0,0,0.000579427];

q = [t1; t2; t3; t4];
qd = [td1; td2; td3; td4];

%%
A01 = [cos(q(1)),0,sin(q(1)),0;sin(q(1)),0,-cos(q(1)),0;0,1,0,a1;0,0,0,1];
A0l1 = [cos(q(1)),0,sin(q(1)),0;sin(q(1)),0,-cos(q(1)),0;0,1,0,l1;0,0,0,1];
A12 =
[cos(q(2)),-sin(q(2)),0,a2*cos(q(2));sin(q(2)),cos(q(2)),0,a2*sin(q(2));0,0,1,0;0,0,0,1];
A1l2 =
[cos(q(2)),-sin(q(2)),0,l2*cos(q(2));sin(q(2)),cos(q(2)),0,l2*sin(q(2));0,0,1,0;0,0,0,1];
A23 =
[cos(q(3)),-sin(q(3)),0,a3*cos(q(3));sin(q(3)),cos(q(3)),0,a3*sin(q(3));0,0,1,0;0,0,0,1];
A2l3 =
[cos(q(3)),-sin(q(3)),0,l3*cos(q(3));sin(q(3)),cos(q(3)),0,l3*sin(q(3));0,0,1,0;0,0,0,1];
A34 =
[cos(q(4)),-sin(q(4)),0,a4*cos(q(4));sin(q(4)),cos(q(4)),0,a4*sin(q(4));0,0,1,0;0,0,0,1];
A3l4 =
[cos(q(4)),-sin(q(4)),0,l4*cos(q(4));sin(q(4)),cos(q(4)),0,l4*sin(q(4));0,0,1,0;0,0,0,1];

%%
z0 = [0;0;1];
P0 = [0;0;0];

z1 = A01(1:3,3);
P1 = A01(1:3,4);
Pl1 = A0l1(1:3,4);

z2 = A01*A12; z2 = z2(1:3,3);
P2 = A01*A12; P2 = P2(1:3,4);
```

```matlab
Pl2 = A01*A1l2; Pl2 = Pl2(1:3,4);

z3 = A01*A12*A23; z3 = z3(1:3,3);
P3 = A01*A12*A23; P3 = P3(1:3,4);
Pl3 = A01*A12*A2l3; Pl3 = Pl3(1:3,4);

z4 = A01*A12*A23*A34; z4 = z4(1:3,3);
P4 = A01*A12*A23*A34; P4 = P4(1:3,4);
Pl4 = A01*A12*A23*A3l4; Pl4 = Pl4(1:3,4);

%%
JPl1 = [cross(z0,(Pl1-P0)),[0;0;0],[0;0;0],[0;0;0]];
JPl2 = [cross(z0,(Pl2-P0)),cross(z1,(Pl2-P1)),[0;0;0],[0;0;0]];
JPl3 = [cross(z0,(Pl3-P0)),cross(z1,(Pl3-P1)),cross(z2,(Pl3-P2)),[0;0;0]];
JPl4 = [cross(z0,(Pl4-P0)),cross(z1,(Pl4-P1)),cross(z2,(Pl4-P2)),cross(z3,(Pl4-P3))];

JOl1 = [z0,[0;0;0],[0;0;0],[0;0;0]];
JOl2 = [z0,z1,[0;0;0],[0;0;0]];
JOl3 = [z0,z2,z3,[0;0;0]];
JOl4 = [z0,z2,z3,z4];

R1 = A01(1:3,1:3);
R2 = A01*A12; R2 = R2(1:3,1:3);
R3 = A01*A12*A23; R3 = R3(1:3,1:3);
R4 = A01*A12*A23*A34; R4 = R4(1:3,1:3);


%%
B = simplify((ml1*(JPl1.'*JPl1) + JOl1.'*R1*Il1*R1.'*JOl1) + (ml2*(JPl2.'*JPl2) +
JOl2.'*R2*Il2*R2.'*JOl2) + (ml3*(JPl3.'*JPl3) + JOl3.'*R3*Il3*R3.'*JOl3) +
(ml4*(JPl4.'*JPl4) + JOl4.'*R4*Il4*R4.'*JOl4));
fid = fopen('B.txt', 'wt');
fprintf(fid, '%s\n', char(B));
fclose(fid);

%%
for k = 1:4
    for i = 1:4
        for j = 1:4
            C(i,j,k) = 0.5*(diff(B(i,j),q(k)) + diff(B(i,k),q(j)) - diff(B(j,k),q(i)));
        end
    end
end

C = simplify(C(:,:,1)*qd(1)+C(:,:,2)*qd(2)+C(:,:,3)*qd(3)+C(:,:,4)*qd(4));
fid = fopen('C.txt', 'wt');
fprintf(fid, '%s\n', char(C));
fclose(fid);

%%
g0 = [0;0;-9.8];
```

```
g = simplify(-(ml1*g0.'*cross(z0,(Pl1-P0)) + ml2*g0.'*cross(z1,(Pl2-P1)) +
ml3*g0.'*cross(z2,(Pl3-P2)) + ml4*g0.'*cross(z3,(Pl4-P3))));
fid = fopen('g.txt', 'wt');
fprintf(fid, '%s\n', char(g));
fclose(fid);
```

Trajectory Planner / Inverse Kinematics MATLAB Code

```
%% MAE 507 Final Project
% Inverse Kinematics and Trajectory Generation
% Matthew Nolan

clear;  close all; clc;

%% Cartesian Path
%  robot base is located at the origin
% chess board is cenetered on the y-axis, and
spacing = 6; % spacing between chess board square centers, in cm
clearance = 15; % safe clearance height
grab_h = 5; % height to grab a pawn

xsquares = linspace(-3.5*spacing, 3.5*spacing, 8); % x positions of square centers
ysquares = linspace(0, 7*spacing,8) + 2*spacing; % y positions of square centers

ini = [xsquares(8),0,0]; % initial position of end effector

% define waypoints along the trajectory
wpts = [ini;
        ini + [0,0, clearance];
        xsquares(4), ysquares(2), clearance;
        xsquares(4), ysquares(2), grab_h;
        xsquares(4), ysquares(2), clearance;
        xsquares(4), ysquares(4), clearance;
        xsquares(4), ysquares(4), grab_h;
        xsquares(4), ysquares(4), clearance;];

dt = 0.005; % length of time step
pause = 0.5; % length of pauses between moves

t_i = [1, 3, 1, 1, 2, 1, 1]; % time it takes to complete each move (s)

t_end = sum(t_i) + pause*(length(t_i)-1); % final time value, accounting for pauses between
moves
total_moves = 2*length(t_i)-1; % total number of moves taken, including pauses

t = (0:dt:t_end)'; % time vector
```

```matlab
% filler matrices, rows are values in time, columns are individual movenents in the routine
x_i = zeros(max(t_i)/dt, total_moves);
x_di = x_i;
x_ddi = x_i;

y_i = x_i;
y_di = x_i;
y_ddi = x_i;

z_i = x_i;
z_di = x_i;
z_ddi = x_i;

for i = 1:total_moves
    if mod(i,2) == 0 % if the index is currently even, indicating the move is a pause
        Lt = pause/dt;
        last = t_i(i/2)/dt;
        x_i(1:Lt,i) = x_i(last,i-1)*ones(Lt,1); % grab the most recent position and hold it
for the duration of the pause
        x_di(1:Lt,i) = x_di(last,i-1)*ones(Lt,1);
        x_ddi(1:Lt,i) = x_ddi(last,i-1)*ones(Lt,1);

        y_i(1:Lt,i) = y_i(last,i-1)*ones(Lt,1);
        y_di(1:Lt,i) = y_di(last,i-1)*ones(Lt,1);
        y_ddi(1:Lt,i) = y_ddi(last,i-1)*ones(Lt,1);

        z_i(1:Lt,i) = z_i(last,i-1)*ones(Lt,1);
        z_di(1:Lt,i) = z_di(last,i-1)*ones(Lt,1);
        z_ddi(1:Lt,i) = z_ddi(last,i-1)*ones(Lt,1);

    else  % if the index is currently odd, indicating the move is a normal movement
        m = (i+1)/2; % current normal move number
        Lt = t_i(m)/dt; % length of time vector for current move
        [x_i(1:Lt,i), x_di(1:Lt,i), x_ddi(1:Lt,i)] = lspb(wpts(m,1), wpts(m+1,1), Lt);
        [y_i(1:Lt,i), y_di(1:Lt,i), y_ddi(1:Lt,i)] = lspb(wpts(m,2), wpts(m+1,2), Lt);
        [z_i(1:Lt,i), z_di(1:Lt,i), z_ddi(1:Lt,i)] = lspb(wpts(m,3), wpts(m+1,3), Lt);
    end
end

x = ini(1);
dx = 0;
ddx = 0;

y = ini(2);
dy = 0;
ddy = 0;

z = ini(3);
dz = 0;
ddz = 0;

for i = 1:total_moves
```

```
    if mod(i,2) == 0 % if the index is currently even, indicating the move is a pause
        L = pause/dt; % length of time vector for pauses
    else % if the index is currently odd, indicating the move is a normal movement
        m = (i+1)/2; % current normal move number
        L = t_i(m)/dt; % length of time vector for current move
    end

    x = [x; x_i(1:L,i)]; % append relevant values from the x position matrix for the current
move
    dx = [dx; x_di(1:L,i)];
    ddx = [ddx; x_ddi(1:L,i)];

    y = [y; y_i(1:L,i)];
    dy = [dy; y_di(1:L,i)];
    ddy = [ddy; y_ddi(1:L,i)];

    z = [z; z_i(1:L,i)];
    dz = [dz; z_di(1:L,i)];
    ddz = [ddz; z_ddi(1:L,i)];
end

% Plot x, y, and z positions, velocities, and accelerations
figure()
subplot(3,1,1)
title("x, y, and z position vs. time")
xlabel("time (s)")
ylabel("position (cm)")
hold on
plot(t,x,'r')
plot(t,y,'g')
plot(t,z,'b')
legend(["x position","y position","z posiion"])

subplot(3,1,2)
title("x, y, and z velocity vs. time")
xlabel("time (s)")
ylabel("velocity (cm/s)")
hold on
plot(t,dx,'r')
plot(t,dy,'g')
plot(t,dz,'b')
legend(["x velocity","y velocity","z velocity"])

subplot(3,1,3)
title("x, y, and z acceleration vs. time")
xlabel("time (s)")
ylabel("acceleration (cm/s^2)")
hold on
plot(t,ddx,'r')
plot(t,ddy,'g')
plot(t,ddz,'b')
legend(["x acceleration","y acceleration","z acceleration"])
```

```matlab
%% Convert from Cartesian to Cylindrical Coordinates

theta = atan2(y,x);
p = sqrt(x.^2+y.^2);

%% Inverse Kinematics

l1 = 19.05; % distance from the floor up to J2
l2 = 30.48;
l3 = 30.48;
l4 = 9.525;

theta1 = theta; % angle of the base rotation

zprime = z + l4; % desired height of joint 4 in order to get the tip of link 4 to be at z
%w =  acos(p./(sqrt(p.^2+(zprime-l1).^2))); % theta2 plus theta3
%w = asin((zprime - l1)./(sqrt(p.^2+(zprime-l1).^2)));
w = atan2(zprime-l1, p); % angle between the xy plane and a line from the origin to zprime
d = sqrt(p.^2+(zprime-l1).^2); % length of the line mentioned above

theta2 = -w + acos((l2^2+d.^2-l3^2)./(2*l2*d));

%theta3 = asin((zprime-l1-l2*sin(theta2))/l3)-theta2;
theta3 = -(acos((p-l2*cos(theta2))/l3)+theta2);
theta4 = -pi/2-theta2-theta3;

% Numerical Differentiation to Get Velocity and Acceleration

dt = diff(t);
dtheta1 = diff(theta1);
dtheta2 = diff(theta2);
dtheta3 = diff(theta3);
dtheta4 = diff(theta4);

ddtheta1 = diff(dtheta1);
ddtheta2 = diff(dtheta2);
ddtheta3 = diff(dtheta3);
ddtheta4 = diff(dtheta4);

d1 = dtheta1./dt;
d2 = dtheta2./dt;
d3 = dtheta3./dt;
d4 = dtheta4./dt;

dd1 = ddtheta1./dt(1:end-1);
dd2 = ddtheta2./dt(1:end-1);
dd3 = ddtheta3./dt(1:end-1);
dd4 = ddtheta4./dt(1:end-1);

% Trim the final values so all the vectors are the same length
t = t(1:end-2);
```

```
theta1 = theta1(1:length(t));
theta2 = theta2(1:length(t));
theta3 = theta3(1:length(t));
theta4 = theta4(1:length(t));

d1 = d1(1:length(t));
d2 = d2(1:length(t));
d3 = d3(1:length(t));
d4 = d4(1:length(t));

dd1 = dd1(1:length(t));
dd2 = dd2(1:length(t));
dd3 = dd3(1:length(t));
dd4 = dd4(1:length(t));

figure()
subplot(3,1,1)
title("jointspace position vs. time")
xlabel("time (s)")
ylabel("joint angle (rad)")
hold on
plot(t,theta1,'r')
plot(t,theta2,'g')
plot(t,theta3,'b')
plot(t,theta4,'k')
legend(["\theta_1","\theta_2","\theta_3","\theta_4"])

subplot(3,1,2)
title("jointspace velocity vs. time")
xlabel("time (s)")
ylabel("joint velocity (rad/s)")
hold on
plot(t,d1,'r')
plot(t,d2,'g')
plot(t,d3,'b')
plot(t,d4,'k')
legend(["\theta_1","\theta_2","\theta_3","\theta_4"])

subplot(3,1,3)
title("jointspace accleration vs. time")
xlabel("time (s)")
ylabel("joint acceleration (rad/s^2)")
hold on
plot(t,dd1,'r')
plot(t,dd2,'g')
plot(t,dd3,'b')
plot(t,dd4,'k')
legend(["\theta_1","\theta_2","\theta_3","\theta_4"])

export = [t, theta1, theta2, theta3, theta4, d1, d2, d3, d4, dd1, dd2, dd3, dd4];
filename = "jointspace_trajectory.xlsx";
writematrix(export,filename,'Sheet',1)
```