Final Project Report

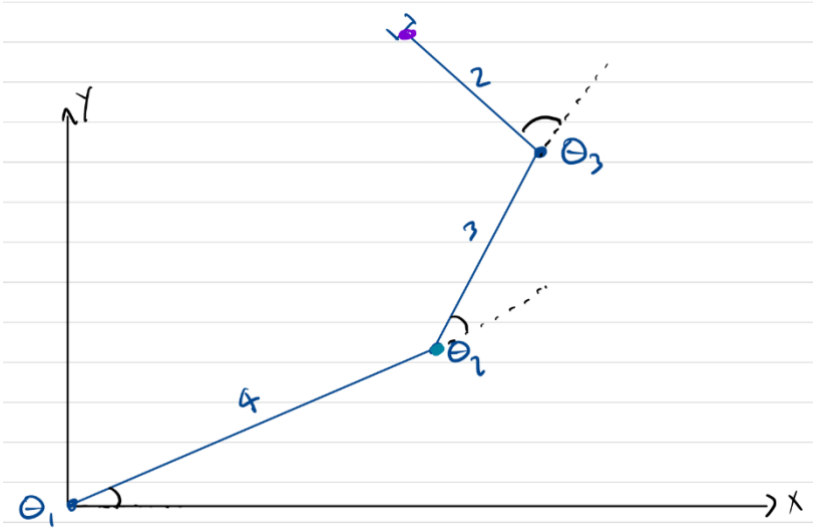
Full Description of Robotic Manipulator:

This is a planar planer robot with 3 rotary links. From now on the robot will be referred to as the RRR Robot. In our modeling the joints rotate about the Z-axis.

Link Specifications:

	Length (m)	Masses (Kg)	Inertial Values (kg-m2)
Link 1	4	20	0.5
Link 2	3	15	0.2
Link 3	2	10	0.1

Manipulator drawings:



DH table:

Joint i	a_{i-1}	α_{i-1}	d_i	Θ_i
1	0	0	0	Θ_1
2	L1 = 4	0	0	Θ_2
3	L2 = 3	0	0	Θ_3
4	L3 = 2	0	0	0

Task 1: Creating the Robot

The first step here is to fully define our 3R planar robot. Using given values from the problem, we derived our transformation matrices as described below. With that, some symbolic variables were initialized such that we could later derive dynamics of our system.

```
syms theta1 theta2 theta3  theta_dot_1 theta_dot_2 theta_dot_3 m1 m2 m3 M grav
deg2rad = pi/180;
theta = [theta1;theta2;theta3];
theta_dot = [theta_dot_1;theta_dot_2;theta_dot_3];
```

Here, we defined these exact variables from the problem/common sense.

```
% masses of links
m1 = 20;
m2 = 15;
m3 = 10;

g = [0 ;-grav; 0]; % gravity only impacts y-axis

% lengths of links
L1 = 4;
L2 = 3;
L3 = 2;

% trabsformation matricies
T01 = [cos(theta1) -sin(theta1) 0 0;...
       sin(theta1)  cos(theta1) 0 0;...
             0             0    1 0;...
             0             0    0 1];
T12 = [cos(theta2) -sin(theta2) 0 L1;...
       sin(theta2)  cos(theta2) 0 0;...
             0             0    1 0;...
             0             0    0 1];

T23 = [cos(theta3) -sin(theta3) 0 L2;...
       sin(theta3)  cos(theta3) 0 0;...
             0             0    1 0;...
             0             0    0 1];
T3F = [sym(1) sym(0) sym(0) sym(L3);...
       sym(0) sym(1) sym(0) sym(0);...
       sym(0) sym(0) sym(1) sym(0);...
       sym(0) sym(0) sym(0) sym(1)];

% rotation matricies
R01 = [cos(theta1) -sin(theta1) 0;...
       sin(theta1)  cos(theta1) 0;...
```

```
             0             0    1];
R12 = [cos(theta2) -sin(theta2) 0;...
       sin(theta2)  cos(theta2) 0;...
             0             0    1];

R23 = [cos(theta3) -sin(theta3) 0;...
       sin(theta3)  cos(theta3) 0;...
             0             0    1];
R3F = [sym(1) sym(0) sym(0);...
       sym(0) sym(1) sym(0);...
       sym(0) sym(0) sym(1)];
```

The following is the transformation from the base to the end-effector. Other matrices are also defined below. as they were needed for calculations later.

```
T0F = simplify(T01*T12*T23*T3F)
```

T0F =

$$\begin{pmatrix} \sigma_2 & -\sigma_1 & 0 & 2\sigma_2 + 3\cos(\theta_1+\theta_2) + 4\cos(\theta_1) \\ \sigma_1 & \sigma_2 & 0 & 2\sigma_1 + 3\sin(\theta_1+\theta_2) + 4\sin(\theta_1) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where

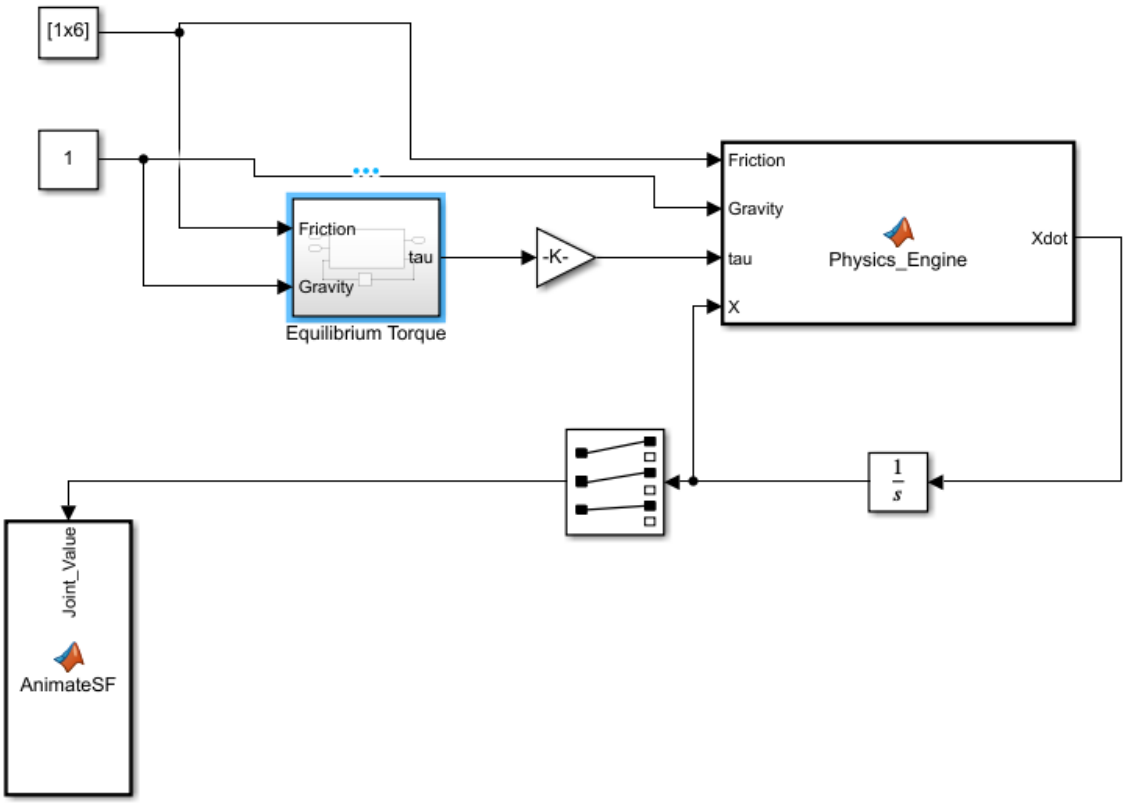
$$\sigma_1 = \sin(\theta_1+\theta_2+\theta_3)$$

$$\sigma_2 = \cos(\theta_1+\theta_2+\theta_3)$$

```
T03 = simplify(T01*T12*T23);
T02 = simplify(T01*T12);

R10 = simplify(transpose(R01));
R21 = simplify(transpose(R12));
R32 = simplify(transpose(R23));
RF3 = simplify(transpose(R3F));
R0F = simplify(R01*R12*R23*R3F);
R02 = simplify(R01*R12);
R03 = simplify(R01*R12*R23);

T10 = simplify(transpose(T01));
T21 = simplify(transpose(T12));
T32 = simplify(transpose(T23));
TF3 = simplify(transpose(T3F));

TF0 = simplify(transpose(T0F));
```

```
T30 = simplify(transpose(T03));
T20 = simplify(transpose(T02));
```

Task 1: Dynamics



The linear and angular velocities were calculated below. Equations from the slideshows were utilized to propagate the velocities through each joint. v11 refers to the linear velocity of link 1, in the coordinates of link 1, and omega11 represents the angular velocity of link 1, in the coordinates of link 1.

```
v11 = R10*[0;0;0];
omega11 = R10*[0;0;0] + theta_dot_1*[0;0;1];
v22 = R21*(v11 + [ 0 -theta_dot_1 0;theta_dot_1 0 0; 0 0 0]*[L1;0;0]);
omega22 = R21*(omega11) + (theta_dot_2)*[0;0;1];
v33 = R32*(v22 + [ 0 -(theta_dot_1 +theta_dot_2) 0;theta_dot_1 + theta_dot_2 0
0; 0 0 0]*[L2;0;0]);
omega33 = R32*(omega22) + (theta_dot_3)*[0;0;1];
vFF = RF3*(v33 + [ 0 -(theta_dot_1 + theta_dot_2 + theta_dot_3) 0;theta_dot_1 +
theta_dot_2 + theta_dot_3 0 0; 0 0 0]*[L3;0;0]);
omegaFF = RF3*(omega33);
```

Next, v0F and omega0F were, calculated. This are the velocities of the end effector in the base coordinate frame.

```
v0F = simplify(R0F * vFF)
```

v0F =

$$\begin{pmatrix} -2\,\dot{\theta}_1\,\sigma_1 - 2\,\dot{\theta}_2\,\sigma_1 - 2\,\dot{\theta}_3\,\sigma_1 - 3\,\dot{\theta}_1\sin(\theta_1+\theta_2) - 3\,\dot{\theta}_2\sin(\theta_1+\theta_2) - 4\,\dot{\theta}_1\sin(\theta_1) \\ 3\,\dot{\theta}_1\cos(\theta_1+\theta_2) + 3\,\dot{\theta}_2\cos(\theta_1+\theta_2) + 4\,\dot{\theta}_1\cos(\theta_1) + 2\,\dot{\theta}_1\,\sigma_2 + 2\,\dot{\theta}_2\,\sigma_2 + 2\,\dot{\theta}_3\,\sigma_2 \\ 0 \end{pmatrix}$$

where

$$\sigma_1 = \sin(\theta_1+\theta_2+\theta_3)$$

$$\sigma_2 = \cos(\theta_1+\theta_2+\theta_3)$$

```
omega0F = simplify(R0F * omegaFF)
```

omega0F =

$$\begin{pmatrix} 0 \\ 0 \\ \dot{\theta}_1+\dot{\theta}_2+\dot{\theta}_3 \end{pmatrix}$$

We now calculate the Jacobian of the system as follows:

```
J0 = simplify([v0F + omega0F]);
```

Here, we are defining more variables given from the problem statement. The objective is to formulate the different matrices that define our dynamics. We are trying to solve for M, B, C, V, and G . From our known as functions of theta and theta dot.

```
% moments of inertia z-copmonent
Izz1 = 0.5;
Izz2 = 0.2;
Izz3 = 0.1;

% moments of inertia of the entire link
IC1 = [0 0   0;...
       0 0  0;...
```

```
       0 0  Izz1];
IC2 = [0 0   0;...
       0 0  0;...
       0 0  Izz2];
IC3 = [0 0   0;...
       0 0  0;...
       0 0  Izz3];

% link masses
m1 = 20;
m2 = 15;
m3 = 10;

% link lengths
L1 = 4;
L2 = 3;
L3 = 2;

% center of mass positions of each link
P_C_1 = T01*[L1/2; 0; 0; 1];
P_C_2 = T02*[L2/2; 0; 0; 1];
P_C_3 = T03*[L3/2; 0; 0; 1];

P_C_1 = P_C_1(1:3);
P_C_2 = P_C_2(1:3);
P_C_3 = P_C_3(1:3);
```

Now, we take the Jacobian of the positions with respect to thetas 1,2,3. Now we have the formulas to generate our Mass Matrix.

```
Jv1 = jacobian(P_C_1,theta);
Jv2 = jacobian(P_C_2,theta);
Jv3 = jacobian(P_C_3,theta);
Jw1 = [0 0 0;...
       0 0 0;...
       1 0 0];
Jw2 = [0 0 0;...
       0 0 0;...
       1 1 0];
Jw3 = [0 0 0;...
       0 0 0;...
       1 1 1];
```

$$M = \sum_{i=1}^{n} (m_i J_{V_i}^T J_{V_i} + J_{V_i}^T I_{C_i} J_{V_i})$$

```
 M = simplify(m1*transpose(Jv1)*Jv1 + transpose(Jw1)*IC1*Jw1) +
(m2*transpose(Jv2)*Jv2 + transpose(Jw2)*IC2*Jw2) + (m3*transpose(Jv3)*Jv3 +
transpose(Jw3)*IC3*Jw3)
```

m111 is the derivative of M(1,1) with respect to theta1. These variables are used to simplify calculation later.

```
 m111 = simplify(diff(M(1,1),theta1));
 m112 = simplify(diff(M(1,1),theta2));
 m113 = simplify(diff(M(1,1),theta3));
 m121 = simplify(diff(M(1,2),theta1));
 m122 = simplify(diff(M(1,2),theta2));
 m123 = simplify(diff(M(1,2),theta3));
 m131 = simplify(diff(M(1,3),theta1));
 m132 = simplify(diff(M(1,3),theta2));
 m133 = simplify(diff(M(1,3),theta3));
 m211 = simplify(diff(M(2,1),theta1));
 m212 = simplify(diff(M(2,1),theta2));
 m213 = simplify(diff(M(2,1),theta3));
 m221 = simplify(diff(M(2,2),theta1));
 m222 = simplify(diff(M(2,2),theta2));
 m223 = simplify(diff(M(2,2),theta3));
 m231 = simplify(diff(M(2,3),theta1));
 m232 = simplify(diff(M(2,3),theta2));
 m233 = simplify(diff(M(2,3),theta3));
 m311 = simplify(diff(M(3,1),theta1));
 m312 = simplify(diff(M(3,1),theta2));
 m313 = simplify(diff(M(3,1),theta3));
 m321 = simplify(diff(M(3,2),theta1));
 m322 = simplify(diff(M(3,2),theta2));
 m323 = simplify(diff(M(3,2),theta3));
 m331 = simplify(diff(M(3,3),theta1));
 m332 = simplify(diff(M(3,3),theta2));
 m333 = simplify(diff(M(3,3),theta3));
```

We use these derivatives to calculate our Christoffel symbols. This is done in the for loop below. From these variable, we can calculate our C, B, V, and G matrices.

```
 for i = 1 : 3
     for j = 1 : 3
         for k = 1 : 3
             temp_var = 0.5 *(eval(strcat('m',num2str(i),num2str(j),num2str(k)))
+ eval(strcat('m',num2str(i),num2str(k),num2str(j))) -
eval(strcat('m',num2str(j),num2str(k),num2str(i))));
             variable.(strcat('b',num2str(i),num2str(j),num2str(k))) = temp_var;
         end
     end
 end
```

$$B = 2\begin{bmatrix} b_{122} & b_{113} & b_{123} \\ b_{212} & b_{213} & b_{223} \\ b_{312} & b_{313} & b_{323} \end{bmatrix}$$

$$C = \begin{bmatrix} b_{111} & b_{122} & b_{133} \\ b_{211} & b_{222} & b_{233} \\ b_{311} & b_{322} & b_{333} \end{bmatrix}$$

```
 C = [variable.b111 variable.b122 variable.b133; variable.b211 variable.b222
variable.b233; variable.b311 variable.b322 variable.b333]
```

C =

$$\begin{pmatrix} 0 & -40\sin(\theta_2+\theta_3)-210\sin(\theta_2) & -40\sin(\theta_2+\theta_3)-30\sin(\theta_3) \\ 40\sin(\theta_2+\theta_3)+210\sin(\theta_2) & 0 & -30\sin(\theta_3) \\ 40\sin(\theta_2+\theta_3)+30\sin(\theta_3) & 30\sin(\theta_3) & 0 \end{pmatrix}$$

```
 B = 2*[variable.b112 variable.b113 variable.b123; variable.b212 variable.b213
variable.b223; variable.b312 variable.b313 variable.b323;]
```

B =

$$\begin{pmatrix} -80\sin(\theta_2+\theta_3)-420\sin(\theta_2) & -80\sin(\theta_2+\theta_3)-60\sin(\theta_3) & -80\sin(\theta_2+\theta_3)-60\sin(\theta_3) \\ 0 & -60\sin(\theta_3) & -60\sin(\theta_3) \\ 60\sin(\theta_3) & 0 & 0 \end{pmatrix}$$

```
 G = -[transpose(Jv1) transpose(Jv2) transpose(Jv3)]*[m1*g;m2*g;m3*g]
```

G =

$$\begin{pmatrix} 15\,\text{grav}\left(\frac{\sigma_1}{2}+4\cos(\theta_1)\right)+10\,\text{grav}\,(\cos(\theta_1+\theta_2+\theta_3)+\sigma_1+4\cos(\theta_1))+40\,\text{grav}\cos(\theta_1) \\ 10\,\text{grav}\,(\cos(\theta_1+\theta_2+\theta_3)+\sigma_1)+\frac{45\,\text{grav}\cos(\theta_1+\theta_2)}{2} \\ 10\,\text{grav}\cos(\theta_1+\theta_2+\theta_3) \end{pmatrix}$$

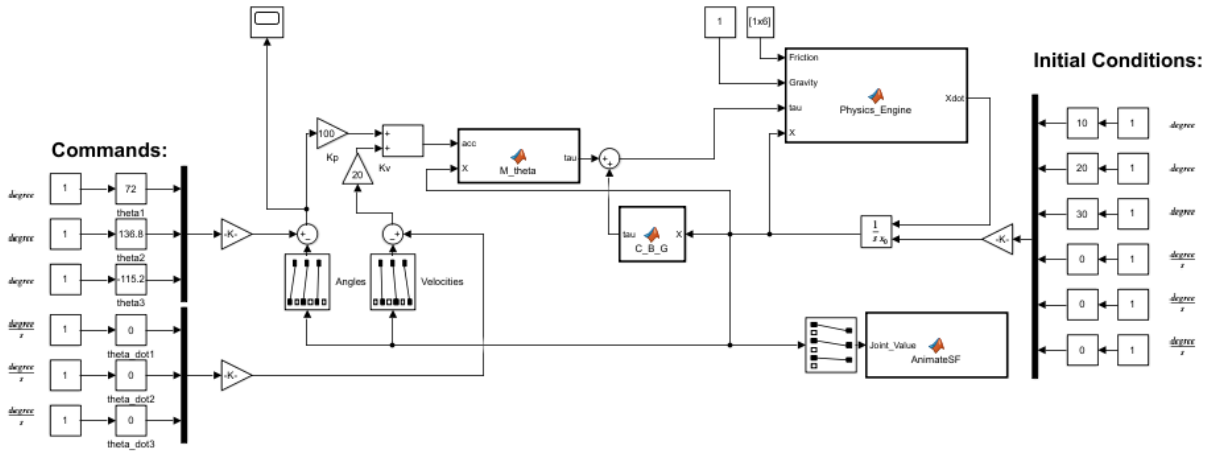
where

$$\sigma_1 = 3\cos(\theta_1+\theta_2)$$

Task 1: Torque to hold a point

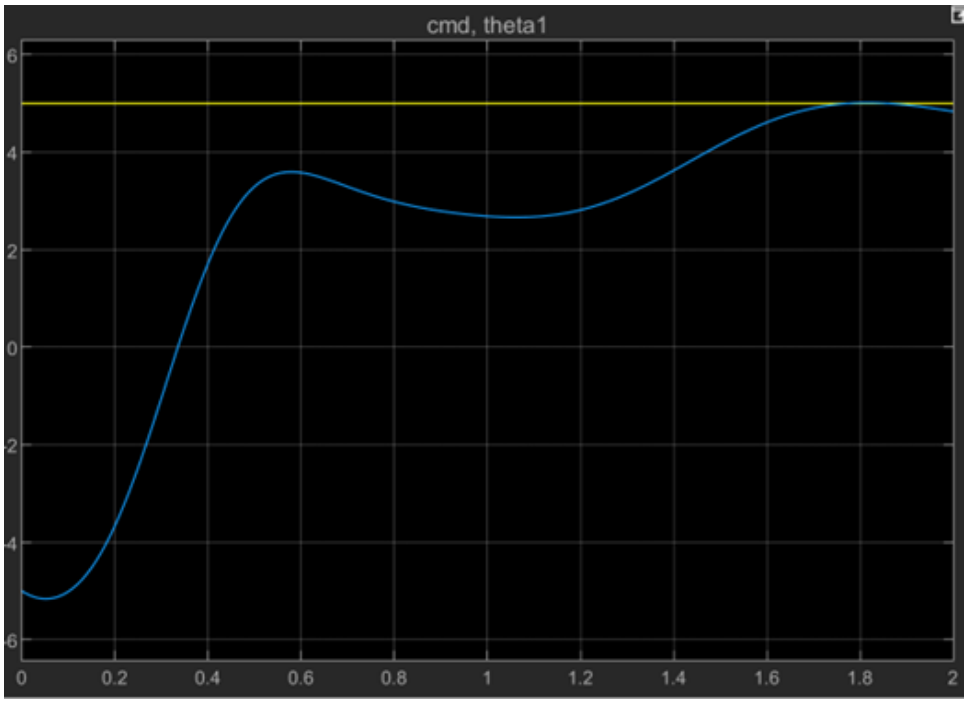
Since here we are only trying to hold the initial condition, we easily calculate the torque by setting $\tau = G$. This will cause the system to hold its initial condition. As the toques will counteract the effect of gravity based on the joint values.

Task 2: Commanding desired Positions

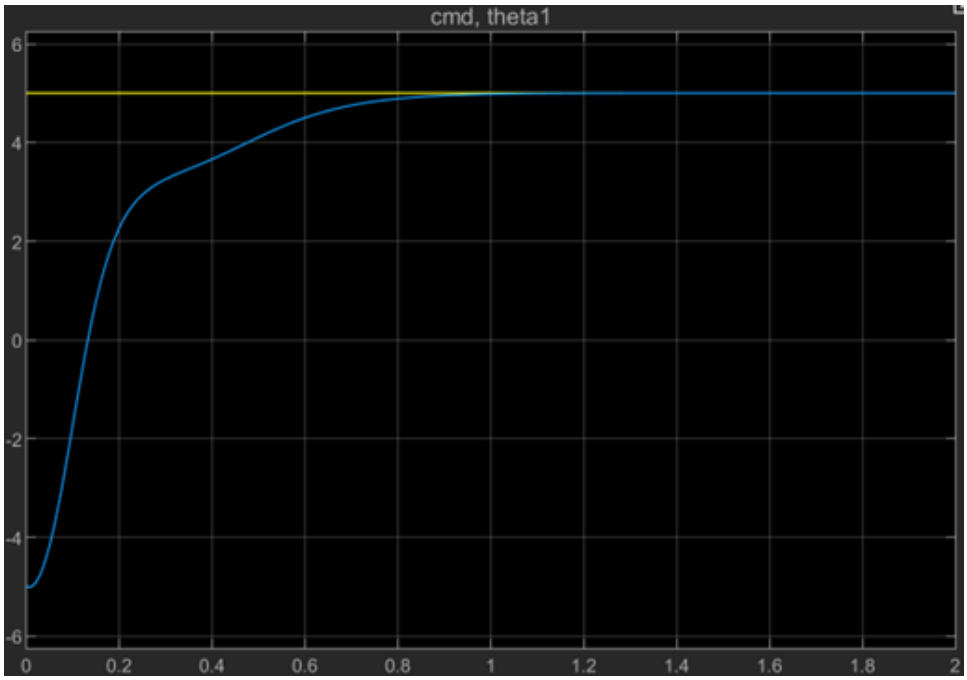


To send the robot to a particular location, we need to feed back the current position. The best way to do this is to partition the dynamics.

Because the mass of the robot has the biggest implication when commanding a particular angle, this is what we use to determine how much torque we will apply. We used a proportional controller that multiplies the error between the desired position by a gain. Likewise, it does the same with desired velocities. These to gains are added into a term that is the acceleration. We multiply this term by the Mass matrix, as $F = ma$. This generated toque is added to the torque generated from C, B, and G matrices. A Simulink diagram showing this system is shown above. The gains on this controller were found experimentally. First, we started with a Kp gain of 10 and a Kv gain of 0. This response was slow and had a steady state error that ocillated around the desired point. To account for the steady state error, the Kv term was increases to 2. We did this as we were assuming that we care more about the positions than the velocity commands. This result is shown below.



Although this was better as eventually the position converged to the command, we were hoping to decrease this oscillation and reduce response time. After some tuning, the values 100 and 20 were selected for Kp and Kv respectively. These results are shown below.

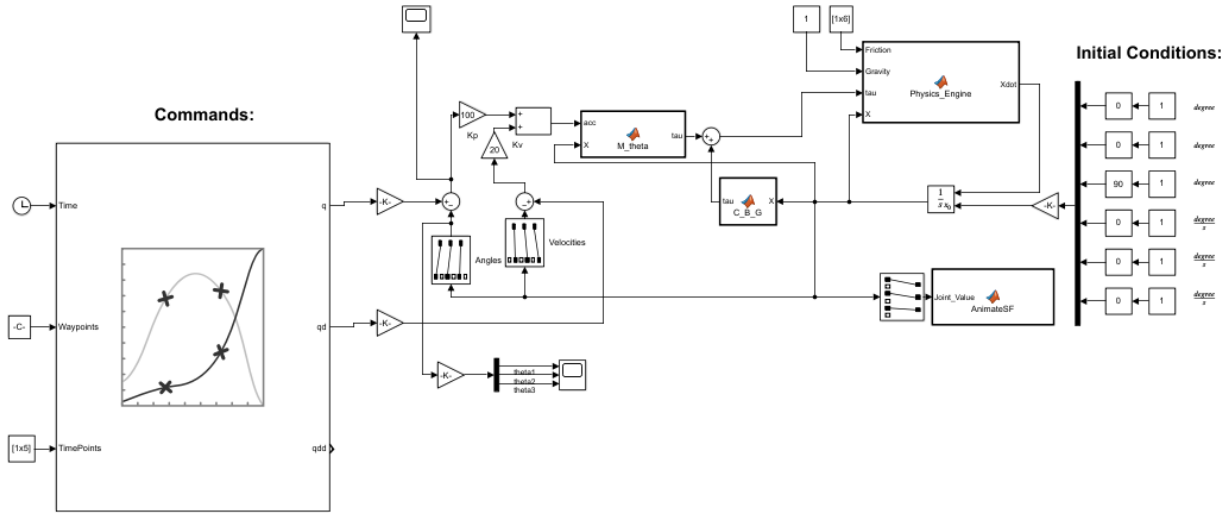


Some testing was done by adding different gains to each joint value, however, we found no improvement in the results, so we kept the gains consistent between all the positions, and all the velocities. A demo sending the robot to multiple desired angles can be seen in the video below.

This is a Clickable Link → Task 2 Video.mp4

Task 3:

To reach different waypoints are specified times, we use spline curve fitting techniques. Below are the initialized times, waypoints.

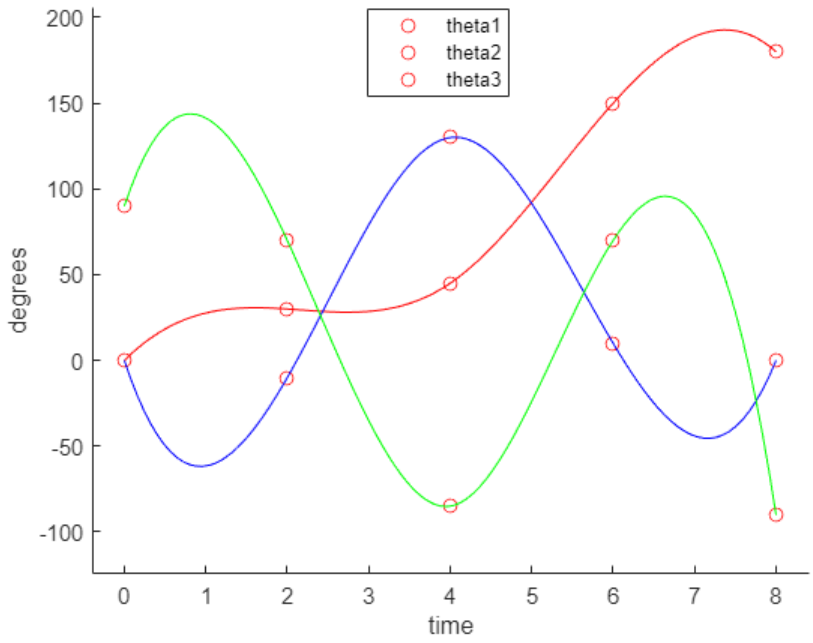



```
% time vector
xx = linspace(0,8,100);
% location at time
times = [0 2 4 6 8];
% waypoints
via_points = [  0   0  90;
               30 -10  70;
               45 130 -85;
              150  10  70;
              180   0 -90;];
path = spline(times,via_points');
```

Here, we plot the curves that we wish to follow.

```
path = spline(times,via_points', linspace(0,8,100));
hold on;
plot(times,via_points,'ro')
plot(linspace(0,8,100),path(1,:), 'r-')
plot(linspace(0,8,100),path(2,:), 'b-')
plot(linspace(0,8,100),path(3,:), 'g-')
legend('theta1', 'theta2', 'theta3')
```

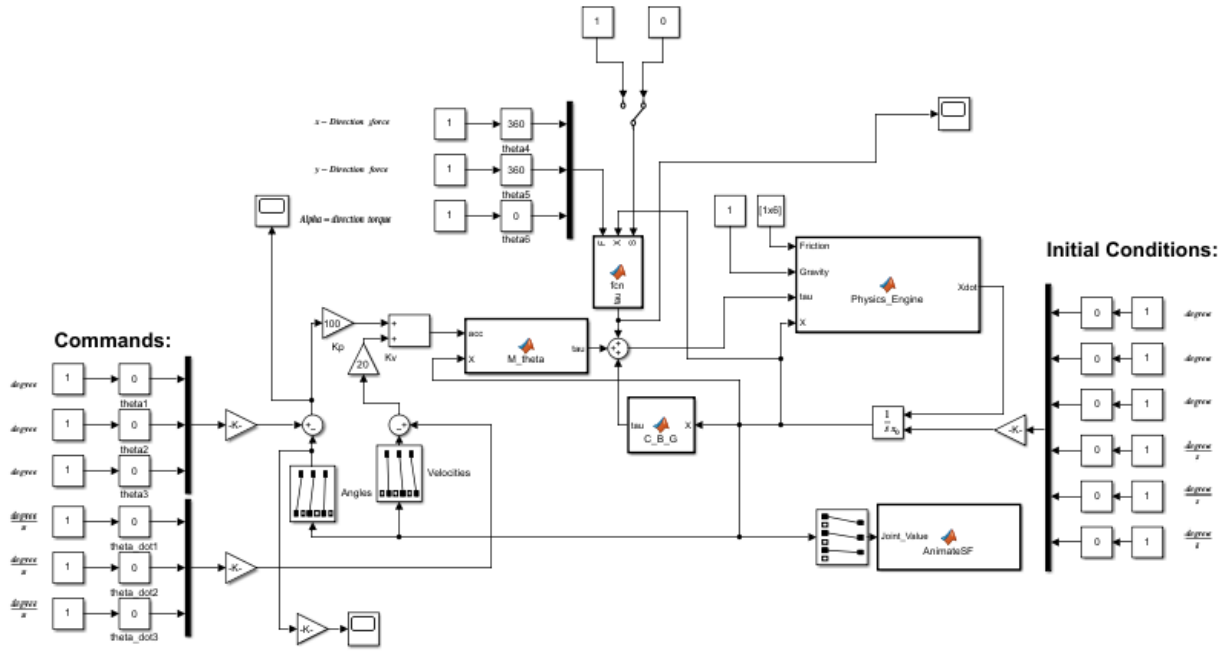
```
xlabel('time')
ylabel('degrees')

xlim([-0.39 8.41])
ylim([-124 206])
legend('Position',[0.42771,0.80123,0.15179,0.11905])
```



Now we use the trajectory block in Simulink to replicate this function on our robot. This diagram is shown below.

This is a Clickable Link → Task 3 Video.mp4

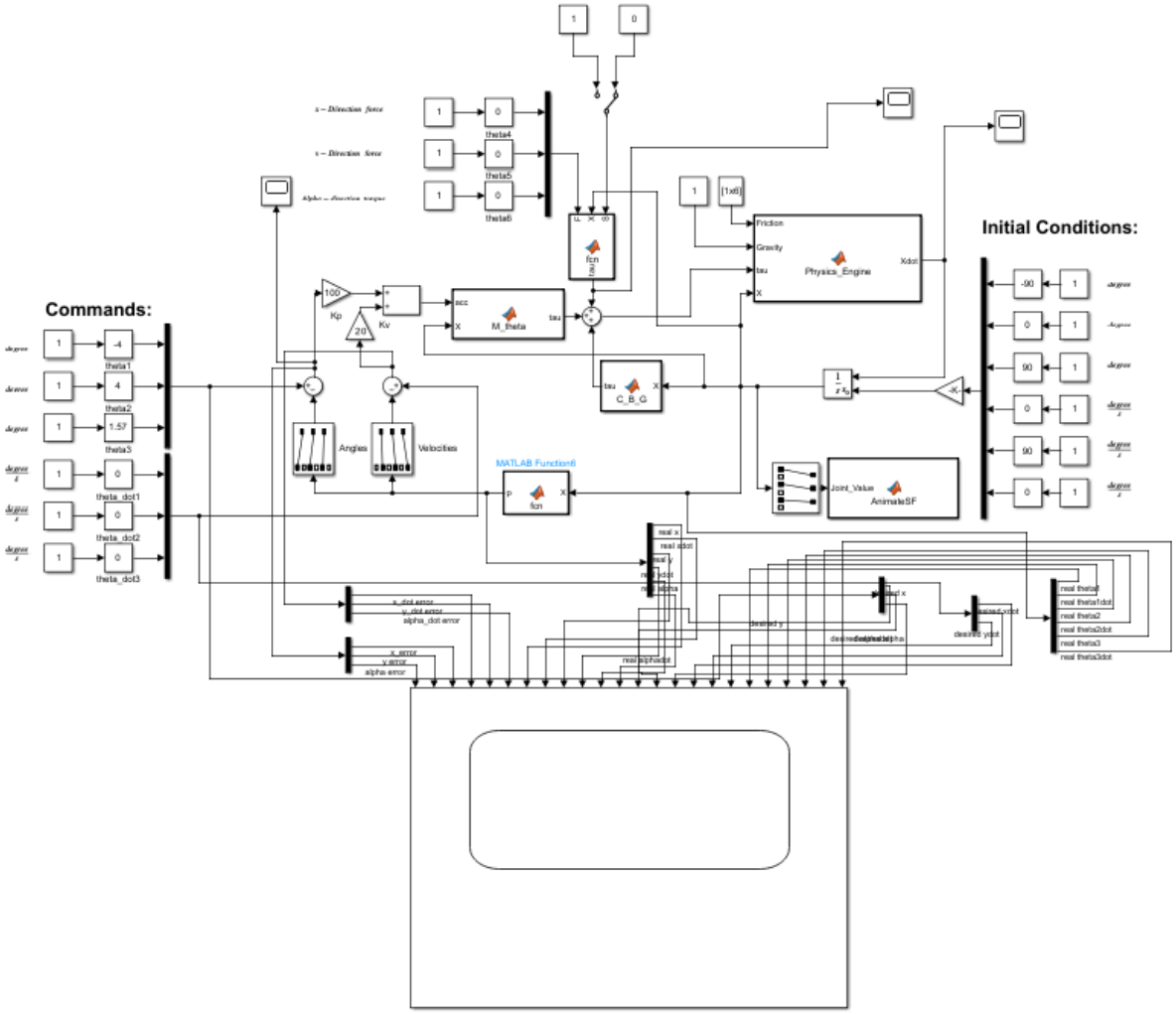
Task 4: Disturbances



To create an exterior disturbance, we simply created an impulse of force in either the x, y, or z direction. We then multiplied these forces by J0 to transform the force into the robots frame. To check to see of our design was correct, we applied a x-force when the robot is set at 0,0,0. As expected, the robot did not move, because e the force was directly into the arm. A Simulink diagram/video demo are shown below when a force is applied to create a noticed disturbance. As expected, the controller send the angles back to the desired positions.

This is a Clickable Link → Task 4 Video.mp4

Task 5: Inverse Kinematics



For finding the inverse kinematics for the RRR Robot, we changed the controller to work in terms of the cartesian system. Instead of finding the exact x, y, and alpha components in terms of the joint values (theta1, theta2, theta3), we kept transforming the velocities from the cartesian space into the joint space using the Jacobian matrix. When transforming from the cartesian space to the join space, the linear velocities are multiplied by the inverse of the Jacobian.

In our MATLAB function where the torque is calculated, the inverse of the Jacobian was multiplied by the current acceleration which is the sum of the velocities and angles that have already been passed through their respective gain blocks. The output of this multiplication is our new acceleration vector in terms of the cartesian space which is then multiplied by the mass matrix to output the torque which is the desired output.

```
accq = J^-1*acc;

MassMatrix=zeros(3);
MassMatrix(1,1)=80*cos(q2 + q3) + 420*cos(q2) + 60*cos(q3) + 12291/20;
MassMatrix(1,2)=40*cos(q2 + q3) + 210*cos(q2) + 60*cos(q3) + 2681/20;
MassMatrix(1,3)=40*cos(q2 + q3) + 30*cos(q3) + 101/10;
MassMatrix(2,1)=40*cos(q2 + q3) + 210*cos(q2) + 60*cos(q3) + 2681/20;
MassMatrix(2,2)=60*cos(q3) + 2681/20;
MassMatrix(2,3)=30*cos(q3) + 101/10;
MassMatrix(3,1)=40*cos(q2 + q3) + 30*cos(q3) + 101/10;
MassMatrix(3,2)=30*cos(q3) + 101/10;
MassMatrix(3,3)=101/10;

tau = MassMatrix*accq;
end
```

In this way, the controller, physics engine, and PID controller all work in unison to push the system towards the desired states since the space has been converted into the cartesian space where it matters.

Below shows the calculations for how to get x, y, alpha when all the theta angles are known.

```
x = 2*cos(theta1 + theta2 + theta3) + 3*cos(theta1 + theta2) + 4*cos(theta1);
```

$$x = 2\cos(\theta_1+\theta_2+\theta_3)+3\cos(\theta_1+\theta_2)+4\cos(\theta_1)$$

```
y = 2*sin(theta1 + theta2 + theta3) + 3*sin(theta1 + theta2) + 4*sin(theta1);
```

$$y = 2\sin(\theta_1+\theta_2+\theta_3)+3\sin(\theta_1+\theta_2)+4\sin(\theta_1)$$

```
alpha = atan2(sin(theta1 + theta2 + theta3),cos(theta1 + theta2 + theta3));
```

$$alpha = \operatorname{atan2}(2\cos(\theta_1+\theta_2+\theta_3)+3\cos(\theta_1+\theta_2)+4\cos(\theta_1),-2\sin(\theta_1+\theta_2+\theta_3)-3\sin(\theta_1+\theta_2)-4\sin(\theta_1))$$

This is a Clickable Link → Task 5 Video.mp4