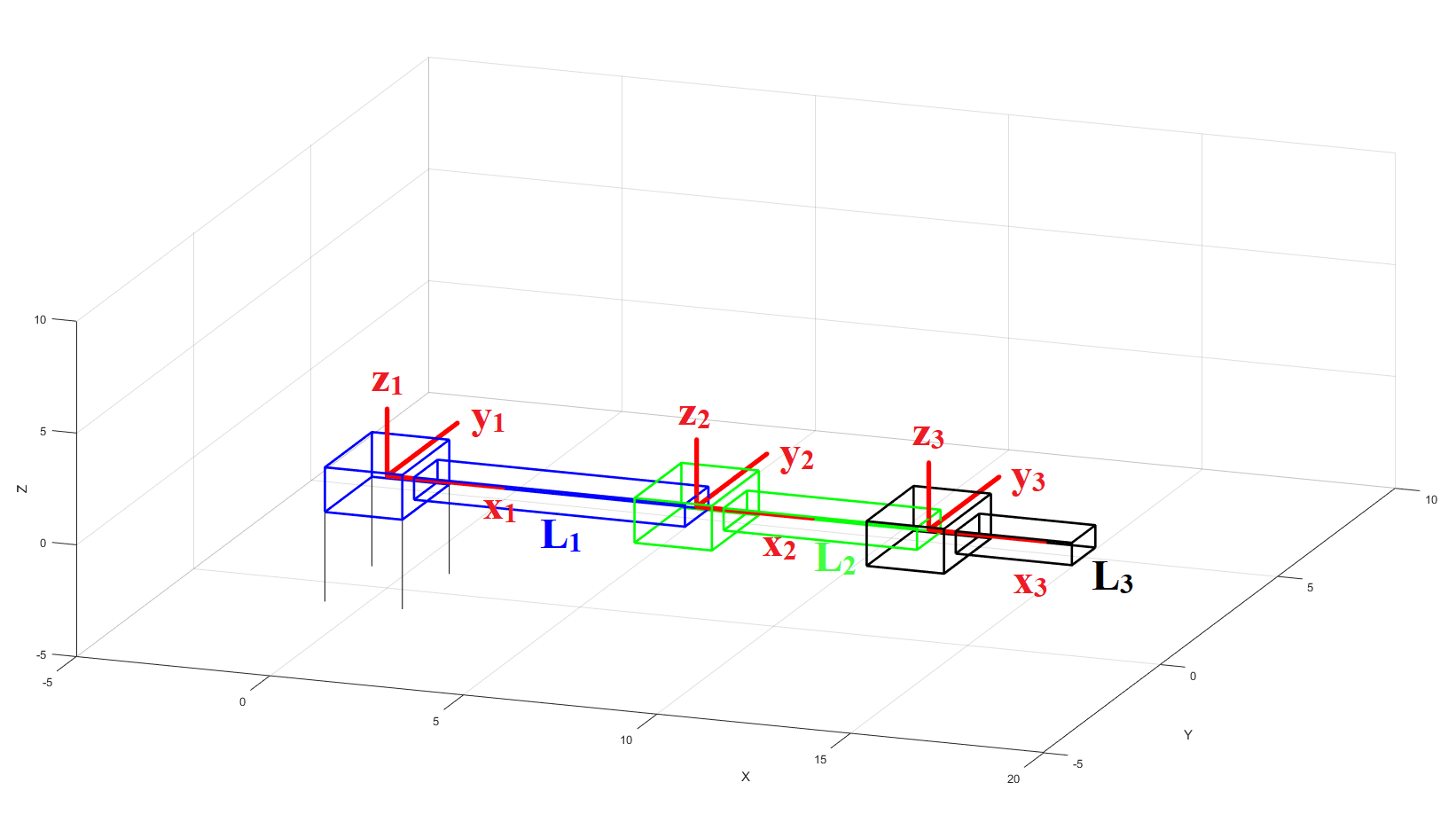
ESE 446 **Final** Report: Planar RRR Manipulator

The goal of this assignment is to create a closed loop controller for the Planar RRR Manipulator using System Partitioning. This is separated into three parts, the preliminary task of defining the robot, the task of modeling the dynamics and an open loop controller and the final task of designing a partitioned controller.

**Task 0: Robot Description**

Before the first task to define a dynamic system for a robot can be done, we need to describe its kinematics. The goal is to create a description of the robot through a set of transformation matrices that we can then use to help describe the dynamics. The kinematics are also how we create a visual animation of the robot.

We start by defining and sketching the robot and the joint axes. I chose to define the axes as shown in Figure 0.1. The important part of this description is that we have the axes along the axes of rotation of each joint, and pointed in the direction of the next joint. This description makes the math easier since it makes all the angles between joints 0 and the lengths positive, as we will see in the next step. We also define the base axis, to be in the same frame as the first axis.



*Figure 0.1: Joint Axes for Planar RRR Manipulator*

After we have defined our joint axes, we want to create a DH table for each joint. We define four parameters for each joint as follows:

distance from   to   along

angle from   to   about

distance from   to   along

angle from   to   about

In the case of the Planar RRR Manipulator, there is no angle between the axes, and the distance along the axes is just , and there is also no distance along the axes, so is zero as well. Therefore, we can create the DH table in Figure 0.2.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | 1 | 0 | 0 | 0 |  |
|  | 2 | 0 |  | 0 |  |
|  | 3 | 0 |  | 0 |  |
|  | 4 | 0 |  | 0 | 0 |

*Figure 0.2: DH Table for Planar RRR Manipulator*

Next, we can use the DH table to calculate transformation matrices between each and joint:

by substituting in the values from the table in Figure 2 for to calculate the matrices, where and . This gives us the following matrices:

These matrices take a vector from the frame (concatenated with a 1) and describes it in the frame (concatenated with a 1). Therefore, we can multiply them together to get a transformation matrix that goes from any a frame to any future frame. The one we are most interested in is

where . This is because it is important in calculating our dynamics. Lastly, we can state that to go from a future frame to a previous one, we can use the inverse of the transformation matrix.

The MATLAB code for the transformation matrices, and using them to create the animation in Figure 0.1 is included in Appendix A.

**Task 1: Dynamics Simulation and Open Loop Controller**

In this task we will describe the dynamics of the Planar RRR Manipulator, assuming it is floating in free space, interacting with the XY-plane, and with gravity pointing in the direction. We start by calculating the Basic Jacobian, and the Jacobian for each frame, and then using the Lagrangian to solve for the angular acceleration of each joint. All the calculations were done symbolically in MATLAB, with the actual values substituted in at the end.

We start by defining the EOAT position in the base frame,

We then can define the position Basic Jacobian is the derivative of the *x* and *y* positions relative to each joint,

The rotation component of the Basic Jacobian is just each one for each joint:

The Jacobian for the system is actually a matrix but, for the sake of writing, I will only show the non-zero components, which are the and components. This is because the robot only moves along the XY-plane, so there is no *z* component, or rotations along the *x, y* axes.

Next we will define the Jacobian in each frame. We can use velocity propagation through the Jacobian to calculate them, and we find that they can be defined to be the derivative of the distance to the center of mass of link of that frame. This gives us

We can do the same for the rotational Jacobians, describing what joints have an effect on each frame, so only has an effect on , and have an effect on and is just the same as

Next, we want to define an equation for the dynamics of the system. We will use a simple second order differential equation in joint space describing the forces on the system

Where is the mass matrix of the system, described by the current position of the joints, is a vector equation describing the forces from the Coriolis and Centrifugal effects, is a vector describing the force of gravity on the system based on the current position, and is input torque vector of torques applied at the joints.

If we can solve for each of these parameters, we have defined a dynamics model of the system by solving the above equation for and setting that to be the acceleration,

Moreover, we can also use this solution to the open loop control problem. If we want acceleration to be zero, we can set the control input to make the inside of the parenthetical to be 0

It also gives us the basis of solving the closed loop system, by letting us define a desired angular acceleration, , and

We will use the Lagrangian, to formulate an equation for to help us describe the system, where the Lagrangian is

With as the Kinetic Energy of the system and as the potential energy of the system. We know that

and we can further break it down into its parts. We know that the potential energy of the system is only in gravity, which doesn’t have , so it will not be in the first term. So we can rewrite the above equation as

where can be defined to be , the forces related to gravity, and to be the inertial forces on the system, .

We can solve for by rewriting the Kinetic energy equation in terms of its base parts

where the inside sum is , is the position Jacobian in the *i*th frame, is the rotation Jacobian in the *i*th frame both of which we found in Part 0, is the mass of the *i*th link which is given and is the Inertial Tensor of the center of mass of the *i*th link. To make this calculation simpler, we define all parts of the inertial tensor to be 0 except . is also given for each link. We then write

Next we use the known the kinetic energy of a system

and take its partial derivative with respect to and to get the remaining parts of our Lagrangian formulation,

The second half of that last equation is directly one of the terms we are looking for, and we already have from above, therefore the remaining terms are

The last piece we need to solve for this symbolically is to be able to rewrite in terms we can solve for. We can formulate the time derivative of to be

using the chain rule, and get

Lastly, we need to solve for . We use the fact that the torque caused by gravity is just

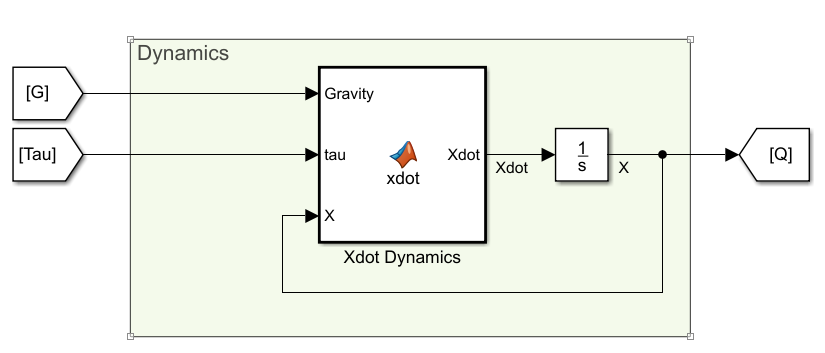
where is the concatenated position Jacobians transposed for each frame, , and is the stacked force of gravity applied to each individual frame, . We get to be the negative of that,

We can then write our acceleration of the system as described above,

and use the fact that is the time integral of   to get a full state space formulation

where . The symbolic calculations in MATLAB for this task are shown in Appendix B, MATLAB B1.

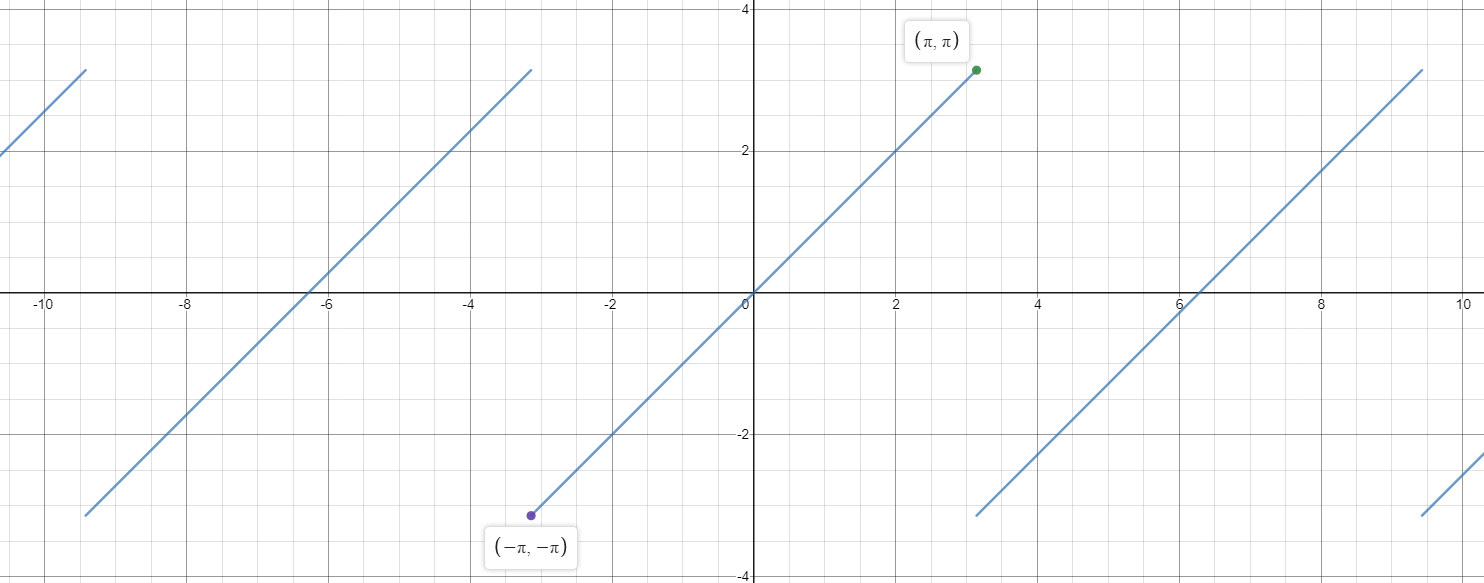
We can now use Simulink to create this model, shown in Figure 1.1. The code for Xdot is in Appendix B, MATLAB B2, is just as described in the equation above.



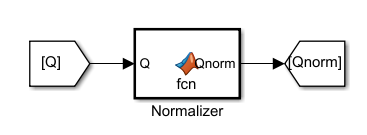
*Figure 1.1: Dynamics System*

The next part of the model is normalizing for the purpose of control. This is simply done for each joint by applying the equation

the result of which is shown below in a Desmos graph in Figure 1.2. The Simulink implementation is shown in Figure 1.3, with MATLAB code in Appendix B, MATLAB B3.

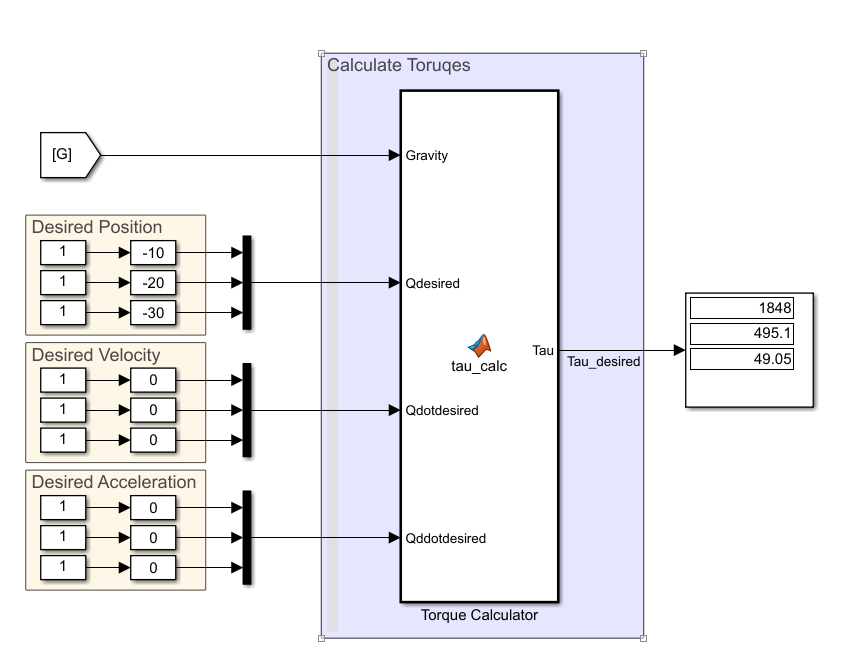


*Figure 1.2: Normalized Joint Angles*



*Figure 1.3: Joint Angle Normalizer*

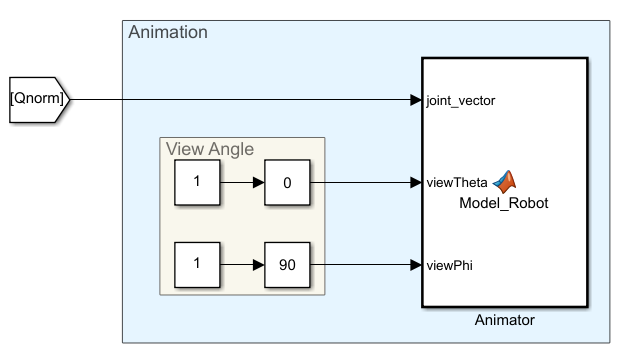
The last part of this task is to implement an open loop controller. This is simply calculating the torque to put maintain the system in a certain state, or to make acceleration 0. The Simulink implementation is shown below in Figure 1.4, with Associated MATLAB code in Appendix B, MATLAB B4. The subsystem takes in the desired parameters and prints on the display the required torque to achieve that.



*Figure 1.4: Open Loop Torque Calculator*

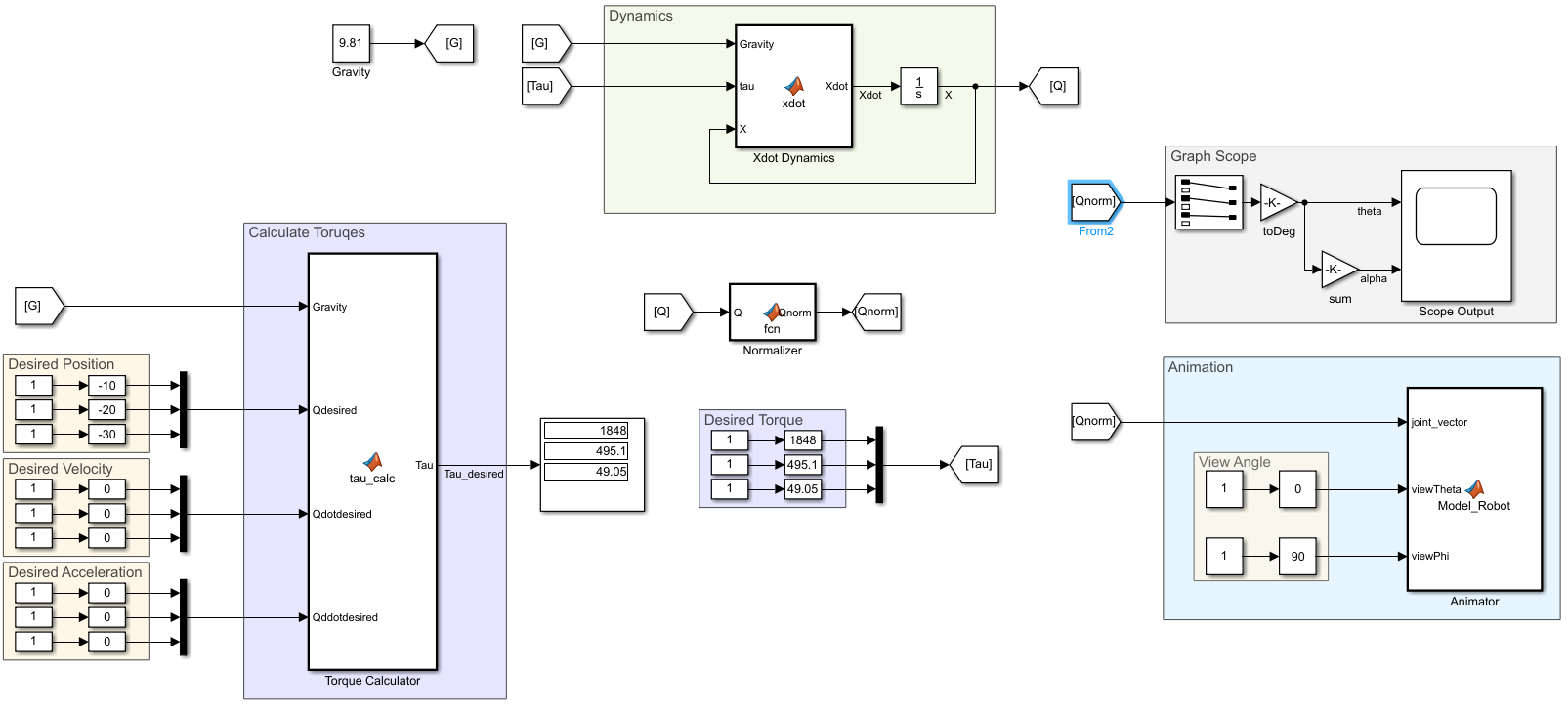
The values printed on the display can then be into the system manually. However, this can be an incredibly unstable controller, as slight variations in the state can cause the system to diverge since there is no feedback. Some states will be stable, for example asking each joint to stay straight down will require 0 torque and friction will cause the system to stably converge to that point, and many states close to there will also be stable. In general, instability occurs for positions above horizontal for any given joint. Note that the same torques are calculated for a positive joint angle as for a negative one.

The last part of the dynamics model is the animator. This is simply using the code that was developed in Task 0 in a MATLAB block and feeding in the joint state and desired view angles (0, 90). The code is in Appendix A, MATLAB A3, and the subsystem is shown below in Figure 1.5.



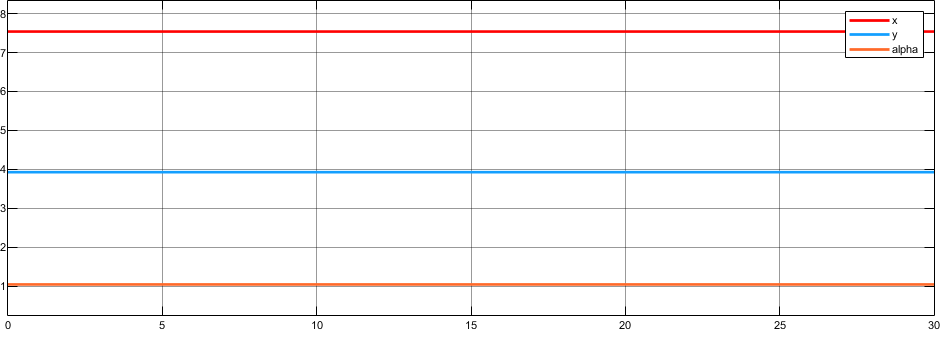
*Figure 1.5: RRR Manipulator Animation*

The complete Simulink system is shown below in Figure 1.6. To complete this task, I will now show demonstrations of the system working in conditions. For all tests, we will set the initial position to be (10, 20, 30) for the joint angles in degrees with an initial velocity of 0 for all joints. ***Since video embeds don’t seem to work in Word, I will attach them separately, named by the Figure they are showing an animation of.*** (The videos are very slow, approximately 1minute/3seconds of animation, so I would recommend skipping through them.)



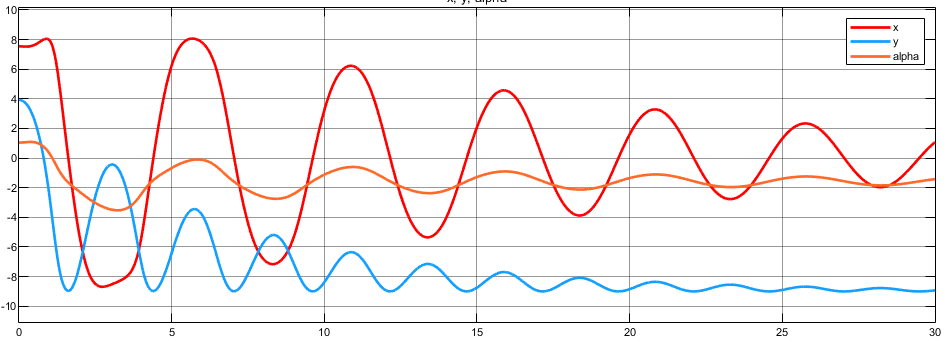
*Figure 1.6: Full Open Loop Simulink Model*

First, I will show the system in with no gravity or torques. We should expect nothing to happen, since there are no forces applied. This is indeed what happens, shown in a graph in Figure 1.7. This does not require an animation to shown what happened.



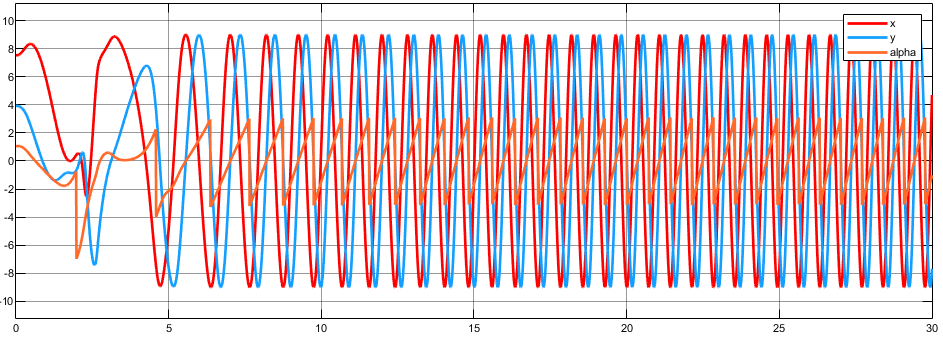
*Figure 1.7: No Torque, No Gravity*

Next, we turn on gravity, but leave off input torques. For this simulation, we expect the system to decay until we have joint angles (90,0,0), or straight down, because friction will dampen the system.



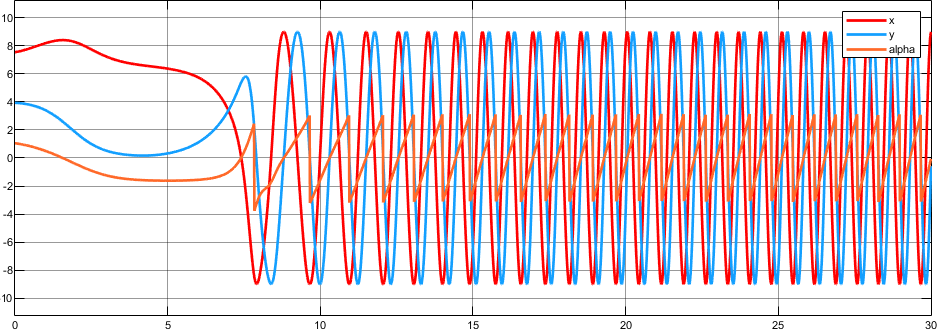
*Figure 1.8: No Torque, With Gravity*

We then turn on the first joint’s actuator, but leave the others out. This should cause the system to destabilize, because the outer two joints will want to hang down, which reduces the effective mass of the system, meaning the stabilizing torque for joint 1 will be too large until its built up enough momentum to continually spin in circles.



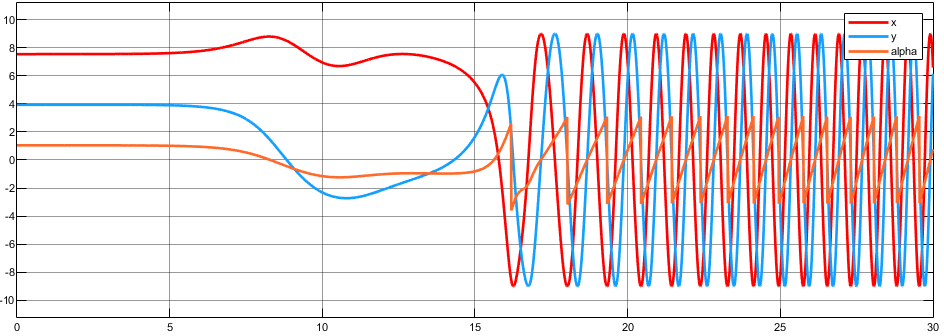
*Figure 1.9: Joint 1 Torque, With Gravity (Unstable)*

We then turn on the first two actuators. We expect the same thing to happen as in the case of one actuator, but slower as the effect is less strong. This is what happens.



*Figure 1.10: First Two Joints’ Actuators, With Gravity (Unstable)*

Lastly, we turn on all three actuators with gravity. We expect this to be an unstable fixed point, because we calculated the acceleration to be 0 with these torques applied but a slight change could cause the system to tend to instability because there is no feedback. In the figure below, we can see it looks stable for the first 5 seconds but then tends toward instability. There is no animation attached because it would take too long and look just like the previous two animations, only slower.



*Figure 1.11: All actuators in Open Loop Controller, (Unstable Fixed Point)*

Note that the instability in these simulations is due in part to the fact that the arm is starting above horizontal (positive *y*), if it was below the simulation in Figure 1.11 would be a stable fixed point. In the next task, we will fix the problem of instability in the upper half using a closed loop controller (and making it possible to control the position instead of just set it somewhere and hoping no one bumps it).

**Task 2: Control Partitioning and Closed Loop Controller**

In this task we will design a closed loop controller for the nonlinear system that works for any pose of the robot, and is not just linearized to a single position. We do this by designing two separate controllers and adding them together. The first controller is the open loop controller we made in Task 1 applied to a closed loop: it reads the state and calculates the torque to keep the acceleration of the system 0; the second is the controller that actually positions the manipulator: it is a PV controller that measures the error in position and velocity states from a desired state, and multiplies them by a gain and the Mass Matrix to get the commanded acceleration.

From Task 1, we defined a closed loop control scheme that gave us

where is the first part of our controller and is the second. We can see that the dynamics of our system are now

where is our commanded acceleration.

Since the first half of the controller essentially cancels the dynamics of the RRR Manipulator, the second controller can assign dynamics of our own. We set the dynamics to be similar to spring-damper system, since we are using a PV controller.

If we let

where , we now have a spring-damper system centered at the desired position and velocity:

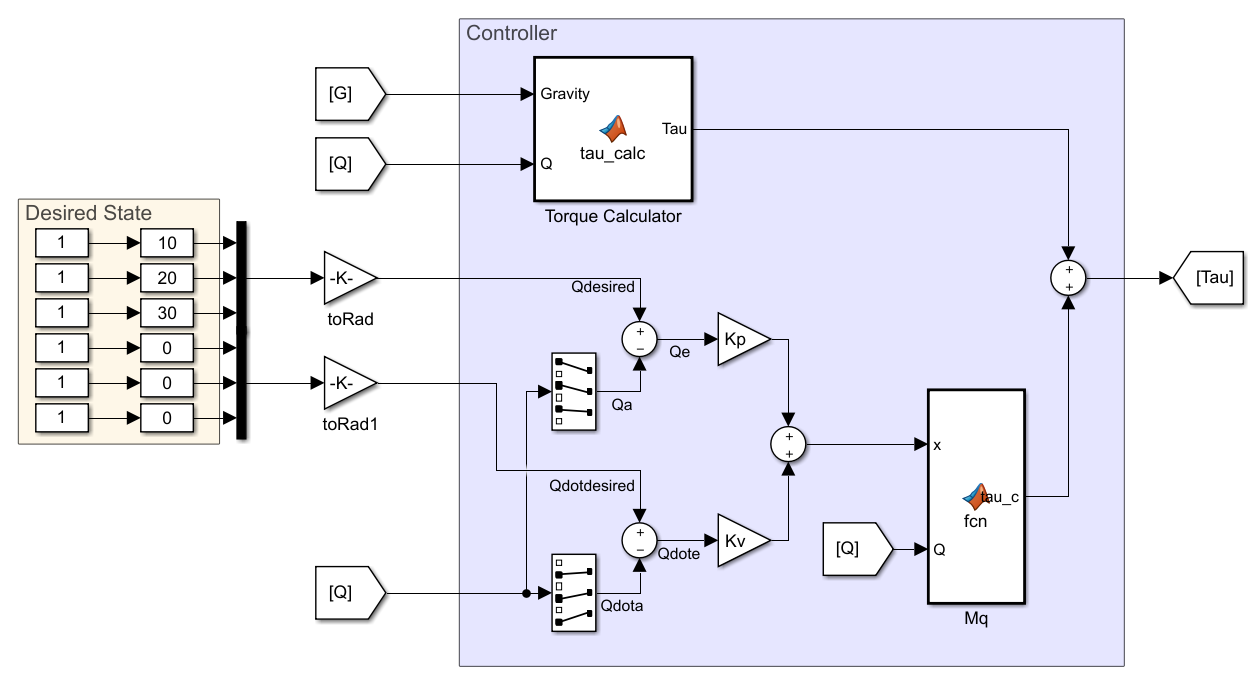
and can use simple state feedback to command it to zero error. We can also use the fact that a critically damped spring-damper system has

to make our system critically damped by keeping the control constants satisfying that equation.

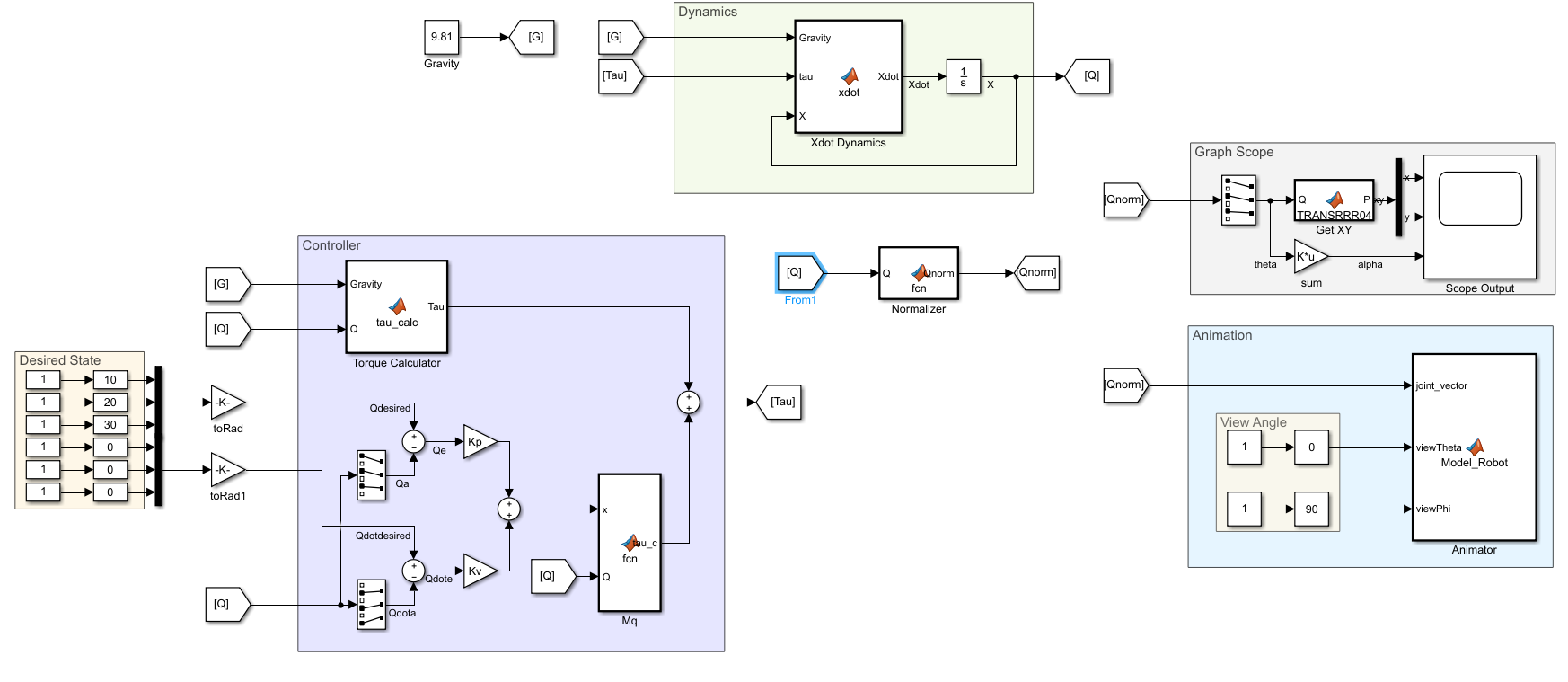
Because each joint system is now independent, we do not have to set different controller gains for each joint, though we could and that would just mean each joint would have different time constants. We can see in each experiment below that each joint has approximately the same time constant.

Note that in order to cause overshooting from underdamping, all we have to do is increase relative to . An example of this is included later (Figure 2.6) for fun.

The only difference to our design is that we now have a controller section that uses the open loop torque calculator we defined in the previous task. The new control-law is shown below in Figure 2.1, and the complete model in Figure 2.2. The code of the two MATLAB function blocks are shown in Appendix C, MATLAB C1 for the open loop controller (top) and MATLAB C2 for the calculation of (bottom).



*Figure 2.1: Closed Loop Partitioning Control-Law*

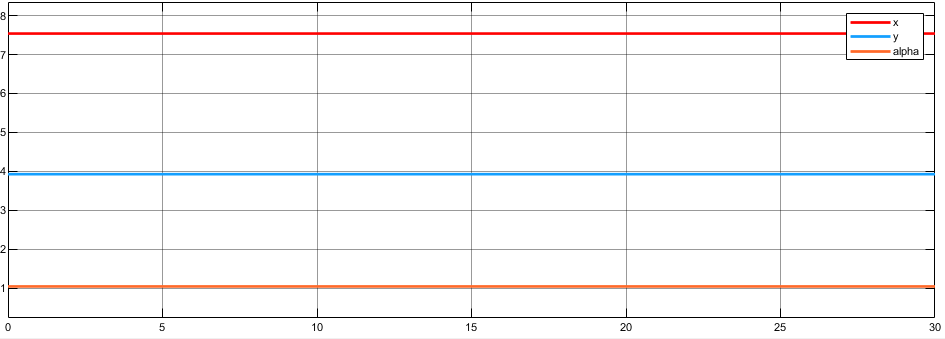
**

*Figure 2.2: Full Closed Loop Controller Model*

We can now test the simulation to show the control law works, and optimize the values of .

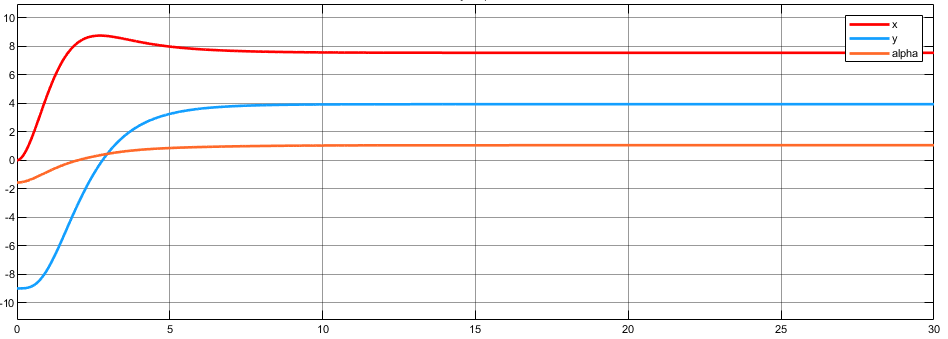
The first simulation is simply inputting the desired joint angles as the initial positions and checking for stability. If the control-law works, the system should be stable (spoiler: it worked). The stable system is shown in Figure 2.3. There is no animation for this because it didn’t move. For every test after this, we will set the system to start hanging straight down and command it to joint angles (10 20 30) degrees.

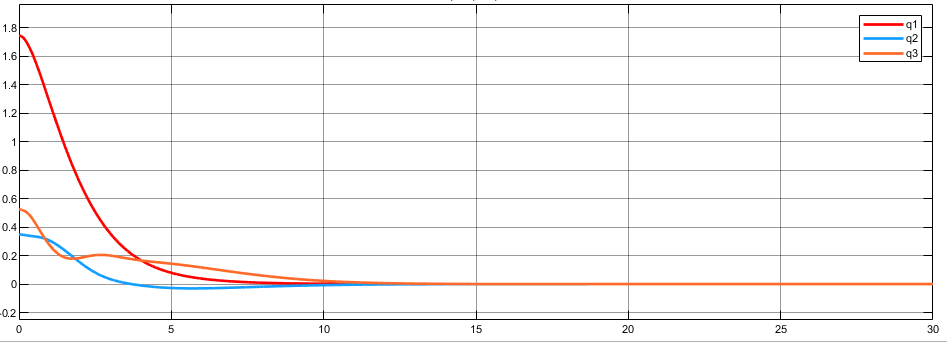
Note that because of the normalizer, the system would have trouble controlling to points near for any of the joints. However, the normalizer could be changed, or removed to allow for larger range.



*Figure 2.3: Closed Loop Stability at Initial Conditions.*

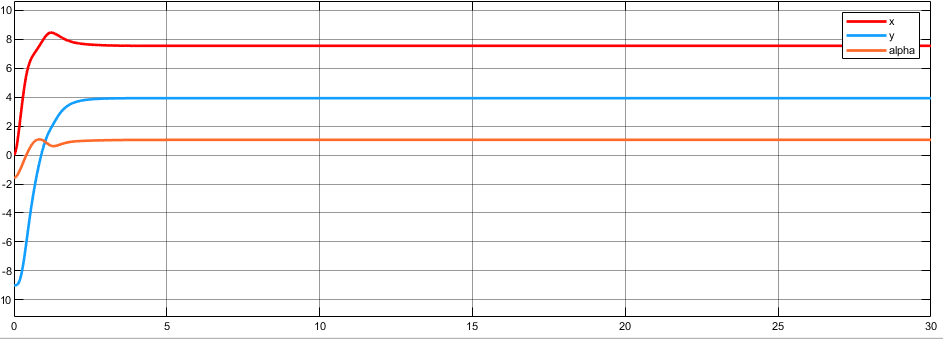
For this next test, we will use . This creates a small damping which means the system will take a long time to converge (time constant of the system is ). This can be seen in Figure 2.4 (top), where it takes around 15 seconds to converge to a steady state. Since we don’t have an integral part to our controller, we will always have a small steady state error, which in this case appears to be negligible. Also seen in Figure 2.4 (bottom) is the convergence of the individual joint angles to zero error. These also take approximately 15 seconds to converge. The time constant convergence is not perfectly accurate (expected error at sec), because as seen by the nonlinearity of the second graph, this is not a true spring-damper system.

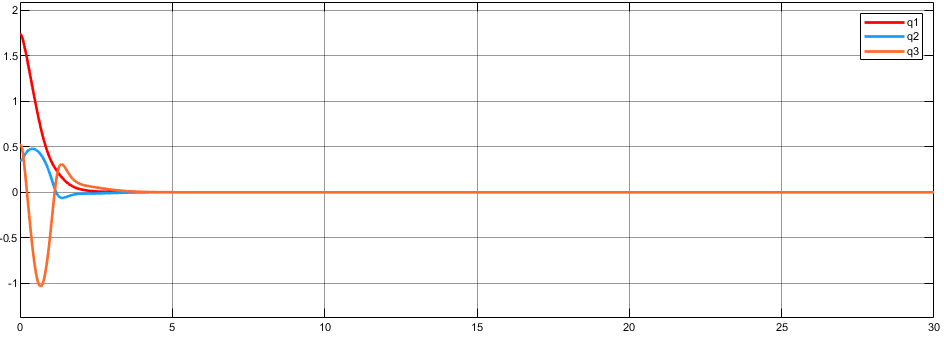




*Figure 2.4: Closed Loop Convergence with*

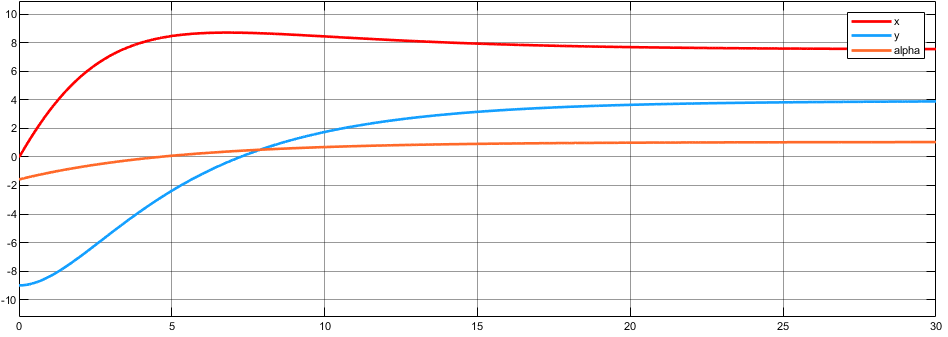
Next, we increase the controller gains, to make the time constant more negative. We set  
, for a time constant of . The main difference is that it converges much quicker, in around 3 seconds, and there is a slightly large overshoot. Again, the second graph is the errors in the joint angles. While there is an apparently large overshoot in I expect it is due to the nonlinearities, rather than the system being underdamped, since it occurs in the same place as it did in Figure 2.4, which certainly isn’t underdamped because it didn’t cross the *x*-axis. It is interesting to note that the nonlinearities did increase with the larger time constant though.

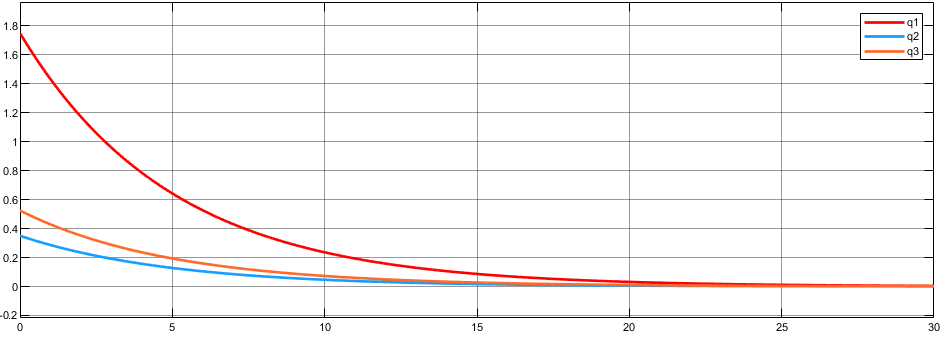




*Figure 2.5: Closed Loop Convergence with*

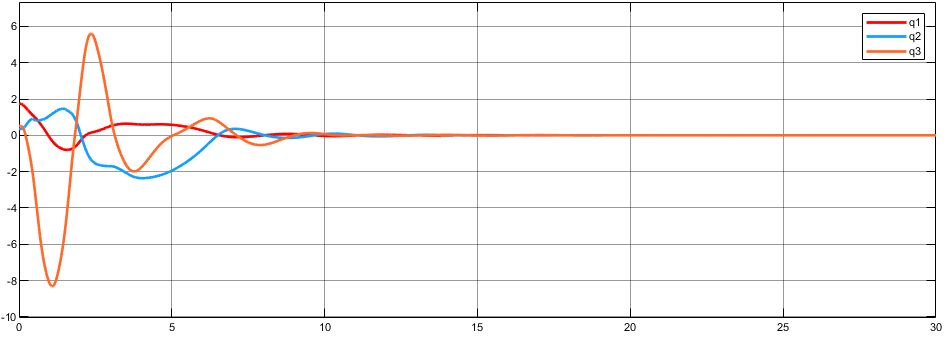
Lastly, we test another order of magnitude increase in . I expected even faster convergence, but with larger nonlinearities. Instead, the rising time seemed to increase, but the nonlinearities almost completely vanished. This is shown in Figure 2.5. While there is a slight overshoot to *X* in the top graph, the actual errors have almost 0 overshoot. Note that I tried increasing the order of magnitude of one more time, but the simulator was not able to handle it.





*Figure 2.6: Closed Loop Convergence with*

For fun, I tried to cause a large overshoot by having a larger relative to the critically damped ratio described earlier. I used and got the graph in Figure 2.6. Note that the underdamped oscillations are visible even through the nonlinearities in this case.



*Figure 2.7: Underdamped Closed Loop Convergence with*

Each of the simulations done in Figures 2.4-2.7 are essentially step responses: the system started at rest, (-90, 0, 0) degree joint angles, and commanded a different position at . The responses look similar to a critically damped second order system, but had some nonlinearities, which are most evident in Figures 2.4 and 2.7. I expect that if we were to command the robot to a larger (further away) angle, it would show more nonlinearities.

They are not shown, but I want to discuss the effects of applying an external torque to the system to simulate someone bumping the robot arm or continually pushing against it. Because this control law does not have an integral term, there will be a steady state error associated with someone continually pushing against it, but it will be able to still recover from a bump.

**Conclusions**

The control partitioning was able to control the robot arm reasonably by separating the dynamics of the system and then remaking them using a PV controller. The best controller gain seemed to be , which had a very fast rise time, but only small nonlinearities and overshoot.

The main advantage to this control law, as noted earlier in the paper is that it turns a nonlinear system into a linear one by removing most of the nonlinear dynamics. It did not remove all of the nonlinear dynamics, and this type of control law is very sensitive to actuator dynamics since it requires accurately modeling a nonlinear system, a small time delay or change in gain can potentially cause the system to be unstable. I did not have time to test the effects of this, but it would be an interesting experiment to apply a time delay and low pass filter to the control input to simulate an actual actuator.

Appendix A: MATLAB of Transformation Matrices and Animation

**Transformation Matrices Function:** Since the transformation matrices are all the same with only a different joint variable for and length , I defined a singular transformation matrix function, TRANSRRR with two parameters, *theta*, *L* and an input vector or set of vectors to be transformed *P*:

function [ Pnew ] = TRANSRRR(P, theta, L)

[~,N] = size(P);

T = [...

cos(theta) -sin(theta) 0 L ;...

sin(theta) cos(theta) 0 0 ;...

0 0 1 0 ;...

0 0 0 1 ;...

];

Pnew = [P; ones(1,N)];

Pnew = T\*Pnew;

Pnew = Pnew(1:3,:);

end

*MATLAB A1: Transformation Matrix Function*

**T03 Function**: As described in Part 0, the matrix is useful, so I created a function for that as well, TRANSRRR03, which takes in a vector, or set of vectors, *P0* and two parameters, a vector of *theta* consisting of *theta1, theta2, theta3* and vector *L* consisting of *L1, L2, L3*. For this robot, *L1* should always be 0, and *L2, L3* correspond to *L1* and *L2* of the robot.

function [ P3 ] = TRANSRRR03(P0, theta, L)

P1 = TRANSRRR(P0, theta(1), L(1));

P2 = TRANSRRR(P1, theta(2), L(2));

P3 = TRANSRRR(P2, theta(3), L(3));

end

*MATLAB A2: T03 Transformation Matrix Function*

**Animation Function:** This function plots the joints in a 3D plot, and is designed to be used in a MATLAB block in Simulink. It takes in three parameters, *joint\_vector*, the angles of the three joints, and *viewTheta, viewPhi*, which rotate the plot to view from the desired angle. To view in the XY plane, *viewTheta, viewPhi* are set to 0, 90. Figure 0.1 was created by adjusting the viewing angles of the animation (and the axes view). An important note is all the lengths of the arms are doubled to make the animation easier to view.

function Model\_Robot(joint\_vector, viewTheta, viewPhi)

%% Unpack Joint Vector

theta = joint\_vector;

L1 = 8;

L2 = 6;

EOA = 4;

L = [0 L1 L2 EOA];

%% Define Links/ Robot Shapes

Base = [...

-1 -1 -1 -1 -1 1 1 -1 1 1 -1 1 1 -1 1 1;...

1 -1 -1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 1;...

-15 -15 -1 -1 -15 -15 -1 -1 -1 -1 -1 -1 -15 -15 -15 -15;...

];

Link1 = [...

1 -1 -1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 1;...

1 1 -1 -1 1 1 -1 -1 -1 -1 -1 -1 1 1 1 1;...

-1 -1 -1 -1 -1 1 1 -1 1 1 -1 1 1 -1 1 1;...

];

L1rod = [...

1 L1 L1 1 1 1 1 1 1 L1 L1 L1 L1 L1 L1 1;...

.5 .5 -.5 -.5 .5 .5 -.5 -.5 -.5 -.5 -.5 -.5 .5 .5 .5 .5;...

-.5 -.5 -.5 -.5 -.5 .5 .5 -.5 .5 .5 -.5 .5 .5 -.5 .5 .5;...

];

Link2 = [...

1 -1 -1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 1;...

1 1 -1 -1 1 1 -1 -1 -1 -1 -1 -1 1 1 1 1;...

-1 -1 -1 -1 -1 1 1 -1 1 1 -1 1 1 -1 1 1;...

];

L2rod = [...

1 L2 L2 1 1 1 1 1 1 L2 L2 L2 L2 L2 L2 1;...

.5 .5 -.5 -.5 .5 .5 -.5 -.5 -.5 -.5 -.5 -.5 .5 .5 .5 .5;...

-.5 -.5 -.5 -.5 -.5 .5 .5 -.5 .5 .5 -.5 .5 .5 -.5 .5 .5;...

];

Link3 = [...

1 -1 -1 1 1 1 1 1 1 -1 -1 -1 -1 -1 -1 1;...

1 1 -1 -1 1 1 -1 -1 -1 -1 -1 -1 1 1 1 1;...

-1 -1 -1 -1 -1 1 1 -1 1 1 -1 1 1 -1 1 1;...

];

L3rod = [...

1 EOA EOA 1 1 1 1 1 1 EOA EOA EOA EOA EOA EOA 1;...

.5 .5 -.5 -.5 .5 .5 -.5 -.5 -.5 -.5 -.5 -.5 .5 .5 .5 .5;...

-.5 -.5 -.5 -.5 -.5 .5 .5 -.5 .5 .5 -.5 .5 .5 -.5 .5 .5;...

];

%% Transform Link 1

BaseLink1= TRANSRRR(Link1 ,theta(1), L(1));

Link1rod = TRANSRRR(L1rod ,theta(1), L(1));

%% Transform Link 2

Link12 = TRANSRRR(Link2 ,theta(2), L(2));

BaseLink2= TRANSRRR(Link12,theta(1), L(1));

L2rod = TRANSRRR(L2rod ,theta(2), L(2));

Link2rod = TRANSRRR(L2rod ,theta(1), L(1));

%% Transform Link 3

Link23 = TRANSRRR(Link3 ,theta(3), L(3));

Link13 = TRANSRRR(Link23,theta(2), L(2));

BaseLink3= TRANSRRR(Link13,theta(1), L(1));

L3rod = TRANSRRR(L3rod ,theta(3), L(3));

L3rod = TRANSRRR(L3rod ,theta(2), L(2));

Link3rod = TRANSRRR(L3rod ,theta(1), L(1));

%% Plotting

plot3(Base(1,:),Base(2,:),Base(3,:),'k-')

axis([-20 20 -20 20 -20 20]);

grid on

hold on

plot3(BaseLink1(1,:),BaseLink1(2,:),BaseLink1(3,:),'b-', 'LineWidth', 2)

plot3(Link1rod(1,:), Link1rod(2,:), Link1rod(3,:), 'b-', 'LineWidth', 2)

plot3(BaseLink2(1,:),BaseLink2(2,:),BaseLink2(3,:),'g-', 'LineWidth', 2)

plot3(Link2rod(1,:), Link2rod(2,:), Link2rod(3,:), 'g-', 'LineWidth', 2)

plot3(BaseLink3(1,:),BaseLink3(2,:),BaseLink3(3,:),'k-', 'LineWidth', 2)

plot3(Link3rod(1,:), Link3rod(2,:), Link3rod(3,:), 'k-', 'LineWidth', 2)

xlabel('X');

ylabel('Y');

zlabel('Z');

view(viewTheta,viewPhi);

hold off

end

*MATLAB A3: Animation Function for use in Simulink*

Appendix B: MATLAB of Dynamics and Open Loop Controller

**Symbolic Dynamics Calculator:** This script calculates the required matrices and vectors to create the dynamics of the RRR system symbolically. The symbolic variables we then copy-pasted into the simulation dynamics (MATLAB B2) and torque calculator (MATLAB B4).

%% Declar Symbolics

syms q1 q2 q3 real

syms q1dot q2dot q3dot real

syms L1 L2 L3 real

syms m1 m2 m3 real

syms Ixx1 Ixx2 Ixx3 real

syms Iyy1 Iyy2 Iyy3 real

syms Izz1 Izz2 Izz3 real

% syms m1 m2 m3 real

q = [q1; q2; q3];

qdot = [q1dot; q2dot; q3dot];

mm = [m1 m2 m3];

LL = [L1 L2 L3];

mm\_actual = [20 15 10]; %kg

LL\_actual = [4 3 2]; %meters

Ic1 = diag([Ixx1,Iyy1,Izz1]);

Ic2 = diag([Ixx2,Iyy2,Izz2]);

Ic3 = diag([Ixx3,Iyy3,Izz3]);

Izz\_all = [Izz1 Izz2 Izz3];

Izz\_actual = [0.5 0.2 0.1];

%% Create Transformation Matrices

T01 = [...

cos(q1) -sin(q1) 0 0 ;...

sin(q1) cos(q1) 0 0 ;...

0 0 1 0 ;...

0 0 0 1 ;...

];

T12 = [...

cos(q2) -sin(q2) 0 L1;...

sin(q2) cos(q2) 0 0 ;...

0 0 1 0 ;...

0 0 0 1 ;...

];

T23 = [...

cos(q3) -sin(q3) 0 L2;...

sin(q3) cos(q3) 0 0 ;...

0 0 1 0 ;...

0 0 0 1 ;...

];

T34 = [...

1 0 0 L3;...

0 1 0 0 ;...

0 0 1 0 ;...

0 0 0 1 ;...

];

T04 = T01\*T12\*T23\*T34;

%% Direct Differentiation Jacobian

% X\_direct = [...

% T04(1:3,4) ;...

% T04(1:3,1) ;...

% T04(1:3,2) ;...

% T04(1:3,3) ;...

% ];

x = L1\*cos(q1) + L2\*cos(q1+q2) + L3\*cos(q1+q2+q3);

y = L1\*sin(q1) + L2\*sin(q1+q2) + L3\*sin(q1+q2+q3);

% r = atan2(y,x);

r = q1+q2+q3;

X\_p = jacobian([x y 0],[q]);

X\_r = jacobian([0 0 r],[q]);

J = [X\_p; X\_r];

%% Calculate Mq

Jv1 = [...

-L1\*sin(q1)/2 0 0;...

L1\*cos(q1)/2 0 0;...

0 0 0;...

];

Jv2 = [...

-(L2\*sin(q1 + q2))/2 - L1\*sin(q1), -(L2\*sin(q1 + q2))/2, 0;...

(L2\*cos(q1 + q2))/2 + L1\*cos(q1), (L2\*cos(q1 + q2))/2, 0;...

0, 0, 0;...

];

Jv3 = [...

-L2\*sin(q1 + q2) - L1\*sin(q1) - (L3\*sin(q1 + q2 + q3))/2, - L2\*sin(q1 + q2) - (L3\*sin(q1 + q2 + q3))/2, -(L3\*sin(q1 + q2 + q3))/2;...

L2\*cos(q1 + q2) + L1\*cos(q1) + (L3\*cos(q1 + q2 + q3))/2, L2\*cos(q1 + q2) + (L3\*cos(q1 + q2 + q3))/2, (L3\*cos(q1 + q2 + q3))/2;...

0, 0, 0;...

];

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];

Mq = (m1\*Jv1'\*Jv1 + Jw1'\*Ic1\*Jw1) +...

(m2\*Jv2'\*Jv2 + Jw2'\*Ic2\*Jw2) +...

(m3\*Jv3'\*Jv3 + Jw3'\*Ic3\*Jw3) ;

Mq = simplify(subs(Mq, [mm LL Izz\_all],[mm\_actual, LL\_actual, Izz\_actual]));

%% Calculate Vqqdot

dMq1 = diff(Mq, q1);

dMq2 = diff(Mq, q2);

dMq3 = diff(Mq, q3);

Mdot = dMq1\*q1dot + dMq2\*q2dot + dMq3\*q3dot;

Vq = Mdot\*qdot -1/2\*[qdot'\*dMq1\*qdot; qdot'\*dMq2\*qdot; qdot'\*dMq3\*qdot];

Vq = simplify(Vq);

%% Calculate Gq

syms grav real

g = [0;-grav;0];

Gq = -[Jv1' Jv2' Jv3']\*[m1\*g;m2\*g;m3\*g];

Gq = simplify(subs(Gq, [mm LL],[mm\_actual LL\_actual]));

% syms tau1 tau2 tau3 real

% tau = [tau1; tau2; tau3];

accel = inv(Mq)\*(- Vq - Gq);

accel = simplify(accel);

%% Position Calculator

syms q1ddot q2ddot q3ddot real

qddot = [q1ddot;q2ddot;q3ddot];

tau = (Vq+Gq)+Mq\*qddot;

tau = simplify(tau);

*MATLAB B1: Symbolic Dynamics Calculations*

**Dynamics Model Function:**  This function is designed for a Simulink block to read in the previous joint space state, *X* and the input torque , and calculate the state derivative. Its output feeds into an integrator to get the state in the next time step.

function Xdot = xdot(Gravity,tau,X)

%% Unpack Variables

SFric1=0;

DFric1=200;

SFric2=0;

DFric2=150;

SFric3=0;

DFric3=100;

grav=Gravity;

q1=X(1);

q1dot=X(2);

q2=X(3);

q2dot=X(4);

q3=X(5);

q3dot=X(6);

%% Calculate Dynamics Matrices

Mq = [...

80\*cos(q2 + q3) + 420\*cos(q2) + 60\*cos(q3) + 12291/20, 40\*cos(q2 + q3) + 210\*cos(q2) + 60\*cos(q3) + 2681/20, 40\*cos(q2 + q3) + 30\*cos(q3) + 101/10;...

40\*cos(q2 + q3) + 210\*cos(q2) + 60\*cos(q3) + 2681/20, 60\*cos(q3) + 2681/20, 30\*cos(q3) + 101/10;...

40\*cos(q2 + q3) + 30\*cos(q3) + 101/10, 30\*cos(q3) + 101/10, 101/10;...

];

Vq = [...

- q3dot\*(40\*q2dot\*sin(q2 + q3) + q3dot\*(40\*sin(q2 + q3) + 30\*sin(q3))) - q2dot\*(q3dot\*(40\*sin(q2 + q3) + 60\*sin(q3)) + q2dot\*(40\*sin(q2 + q3) + 210\*sin(q2))) - q1dot\*(q3dot\*(80\*sin(q2 + q3) + 60\*sin(q3)) + q2dot\*(80\*sin(q2 + q3) + 420\*sin(q2)));...

210\*q1dot^2\*sin(q2) - 30\*q3dot^2\*sin(q3) + 40\*q1dot^2\*sin(q2 + q3) - 60\*q1dot\*q3dot\*sin(q3) - 60\*q2dot\*q3dot\*sin(q3);...

30\*q1dot^2\*sin(q3) + 30\*q2dot^2\*sin(q3) + 40\*q1dot^2\*sin(q2 + q3) + 60\*q1dot\*q2dot\*sin(q3);...

];

Gq = [...

(5\*grav\*(4\*cos(q1 + q2 + q3) + 21\*cos(q1 + q2) + 56\*cos(q1)))/2;...

(105\*grav\*cos(q1 + q2))/2 + 10\*grav\*cos(q1 + q2 + q3);...

10\*grav\*cos(q1 + q2 + q3);...

];

F1=SFric1\*sign(q1dot) + DFric1\*q1dot;

F2=SFric2\*sign(q2dot) + DFric2\*q2dot;

F3=SFric3\*sign(q3dot) + DFric3\*q3dot;

Fq=[F1;F2;F3];

%% Get Acceleration and Xdot

accel = Mq^(-1)\*(tau - Vq - Gq - Fq);

Xdot=[q1dot;accel(1);q2dot;accel(2);q3dot;accel(3)];

end

*MATLAB B2: State Space Dynamics*

**Joint Angle Normalizer:** this function is used in a Simulink block to normalize the joint angles between . It reads in the state and returns a normalized state.

function Qnorm = fcn(Q)

q = Q(1:3);

qdot = Q(4:6);

qnorm = 2.\*atan(tan(q./2));

Qnorm = [qnorm;qdot];

*MATLAB B3: Trig Normalizer*

**Open Loop Torque Calculator**: This function is used in Simulink to calculate the torque required to achieve a desired acceleration at a desired state.

function Tau = tau\_calc(Gravity,Qdesired,Qdotdesired,Qddotdesired)

SFric1=0;

DFric1=200;

SFric2=0;

DFric2=150;

SFric3=0;

DFric3=100;

grav=Gravity;

q1 = Qdesired(1)\*pi/180;

q2 = Qdesired(2)\*pi/180;

q3 = Qdesired(3)\*pi/180;

q1dot = Qdotdesired(1);

q2dot = Qdotdesired(2);

q3dot = Qdotdesired(3);

q1ddot = Qddotdesired(1);

q2ddot = Qddotdesired(2);

q3ddot = Qddotdesired(3);

Mq = [...

80\*cos(q2 + q3) + 420\*cos(q2) + 60\*cos(q3) + 12291/20, 40\*cos(q2 + q3) + 210\*cos(q2) + 60\*cos(q3) + 2681/20, 40\*cos(q2 + q3) + 30\*cos(q3) + 101/10;...

40\*cos(q2 + q3) + 210\*cos(q2) + 60\*cos(q3) + 2681/20, 60\*cos(q3) + 2681/20, 30\*cos(q3) + 101/10;...

40\*cos(q2 + q3) + 30\*cos(q3) + 101/10, 30\*cos(q3) + 101/10, 101/10;...

];

Vq = [...

- q3dot\*(40\*q2dot\*sin(q2 + q3) + q3dot\*(40\*sin(q2 + q3) + 30\*sin(q3))) - q2dot\*(q3dot\*(40\*sin(q2 + q3) + 60\*sin(q3)) + q2dot\*(40\*sin(q2 + q3) + 210\*sin(q2))) - q1dot\*(q3dot\*(80\*sin(q2 + q3) + 60\*sin(q3)) + q2dot\*(80\*sin(q2 + q3) + 420\*sin(q2)));...

210\*q1dot^2\*sin(q2) - 30\*q3dot^2\*sin(q3) + 40\*q1dot^2\*sin(q2 + q3) - 60\*q1dot\*q3dot\*sin(q3) - 60\*q2dot\*q3dot\*sin(q3);...

30\*q1dot^2\*sin(q3) + 30\*q2dot^2\*sin(q3) + 40\*q1dot^2\*sin(q2 + q3) + 60\*q1dot\*q2dot\*sin(q3);...

];

Gq = [...

(5\*grav\*(4\*cos(q1 + q2 + q3) + 21\*cos(q1 + q2) + 56\*cos(q1)))/2;...

(105\*grav\*cos(q1 + q2))/2 + 10\*grav\*cos(q1 + q2 + q3);...

10\*grav\*cos(q1 + q2 + q3);...

];

F1=SFric1\*sign(q1dot) + DFric1\*q1dot;

F2=SFric2\*sign(q2dot) + DFric2\*q2dot;

F3=SFric3\*sign(q3dot) + DFric3\*q3dot;

Fq=[F1;F2;F3];

Tau = Vq + Gq + Fq + Mq\*[q1ddot;q2ddot;q3ddot];

end

*MATLAB 4: Open Loop Torque Calculator*

Appendix C: MATLAB of Partitioning and Closed Loop Controller

**Cancelation Torque Calculator:** This Simulink function takes in the current state and calculates the required torque to set acceleration to 0.

function Tau = tau\_calc(Gravity,Q)

SFric1=0;

DFric1=200;

SFric2=0;

DFric2=150;

SFric3=0;

DFric3=100;

grav=Gravity;

q1 = Q(1);

q2 = Q(3);

q3 = Q(5);

q1dot = Q(2);

q2dot = Q(4);

q3dot = Q(6);

Vq = [...

- q3dot\*(40\*q2dot\*sin(q2 + q3) + q3dot\*(40\*sin(q2 + q3) + 30\*sin(q3))) - q2dot\*(q3dot\*(40\*sin(q2 + q3) + 60\*sin(q3)) + q2dot\*(40\*sin(q2 + q3) + 210\*sin(q2))) - q1dot\*(q3dot\*(80\*sin(q2 + q3) + 60\*sin(q3)) + q2dot\*(80\*sin(q2 + q3) + 420\*sin(q2)));...

210\*q1dot^2\*sin(q2) - 30\*q3dot^2\*sin(q3) + 40\*q1dot^2\*sin(q2 + q3) - 60\*q1dot\*q3dot\*sin(q3) - 60\*q2dot\*q3dot\*sin(q3);...

30\*q1dot^2\*sin(q3) + 30\*q2dot^2\*sin(q3) + 40\*q1dot^2\*sin(q2 + q3) + 60\*q1dot\*q2dot\*sin(q3);...

];

Gq = [...

(5\*grav\*(4\*cos(q1 + q2 + q3) + 21\*cos(q1 + q2) + 56\*cos(q1)))/2;...

(105\*grav\*cos(q1 + q2))/2 + 10\*grav\*cos(q1 + q2 + q3);...

10\*grav\*cos(q1 + q2 + q3);...

];

F1=SFric1\*sign(q1dot) + DFric1\*q1dot;

F2=SFric2\*sign(q2dot) + DFric2\*q2dot;

F3=SFric3\*sign(q3dot) + DFric3\*q3dot;

Fq=[F1;F2;F3];

Tau = Vq + Gq + Fq;

end

*MATLAB C1: Cancelation Torque*

**Mass Matrix Multiplier:** This function takes in a vector (the desired acceleration, ) and multiplies it by the Mass Matrix for the current time to cancel out the effect of the inverse of the Mass Matrix in the dynamics.

function tau\_c = fcn(x,Q)

q1=Q(1);

q2=Q(2);

q3=Q(3);

Mq = [...

80\*cos(q2 + q3) + 420\*cos(q2) + 60\*cos(q3) + 12291/20, 40\*cos(q2 + q3) + 210\*cos(q2) + 60\*cos(q3) + 2681/20, 40\*cos(q2 + q3) + 30\*cos(q3) + 101/10;...

40\*cos(q2 + q3) + 210\*cos(q2) + 60\*cos(q3) + 2681/20, 60\*cos(q3) + 2681/20, 30\*cos(q3) + 101/10;...

40\*cos(q2 + q3) + 30\*cos(q3) + 101/10, 30\*cos(q3) + 101/10, 101/10;...

];

tau\_c = Mq\*x;

*MATLAB C2: Mass Matrix Multiplier of Alpha*