Robot Dynamics and Control

Lab 3 Report

Inverse Dynamics Joint Control

Amith Ramdas Achari
amithr3
Peng Han
penghan2
Section: Friday 9AM
Submitted on April 11, 2022

1 Introduction

Inverse dynamics is a nonlinear control technique that calculates the joint actuator torques generated by the motors in order to achieve a given trajectory. Inverse dynamics control can produce excellent results if we know the dynamics of the system accurately and they are simple to compute. This lab compares inverse dynamics control with the PD control we designed in the previous lab. In this lab, we'll also look at how the system performs when the masses change, as well as how friction compensation improves the system's performance.

2 Tasks

• Apply Friction Compensation to all three joints and see how well it works to eliminate friction in the arm.

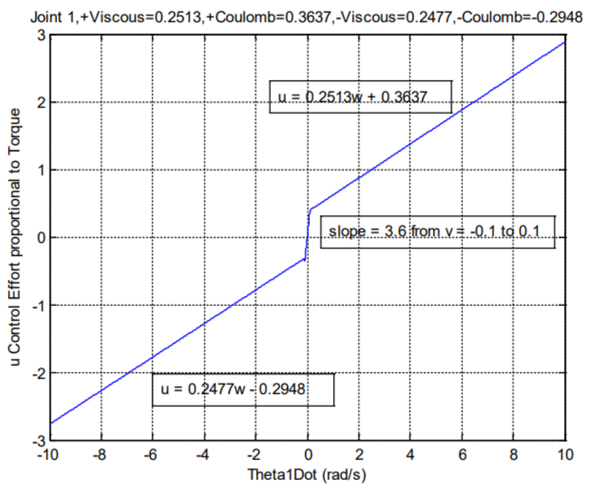
• Design and implement an inverse dynamics control algorithm and compare its performance to the PD plus feedforward control designed in lab 2.

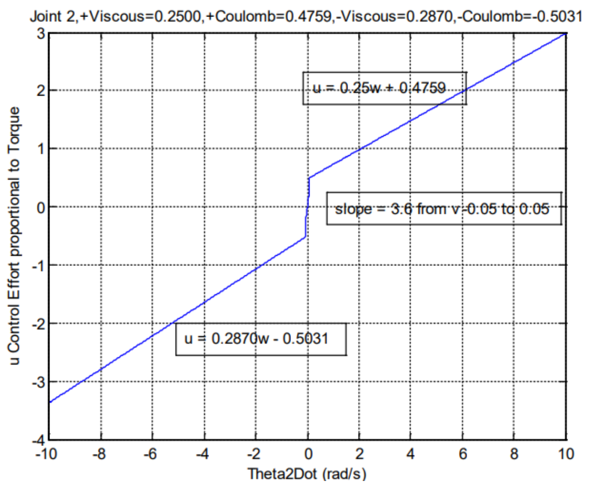
• Examine how well the CRS robot's parameters have been identified and see if better values for the system's parameters can be found.

3 Implement Friction Compensation

3.1 Identifying Friction Curves

To implement friction compensation for each joint of your robot arm, we used straight-line equations given in Manual as shown in Figure. This is based on experimental friction plots on one CRS robot arm. These plots show the best fit lines that were fit to the collected data at constant velocity and control effort (proportional to torque). After friction compensation was implemented, we tweaked the friction coefficients of each joint until we observed that your robot arm's moved without any friction if given a small push.





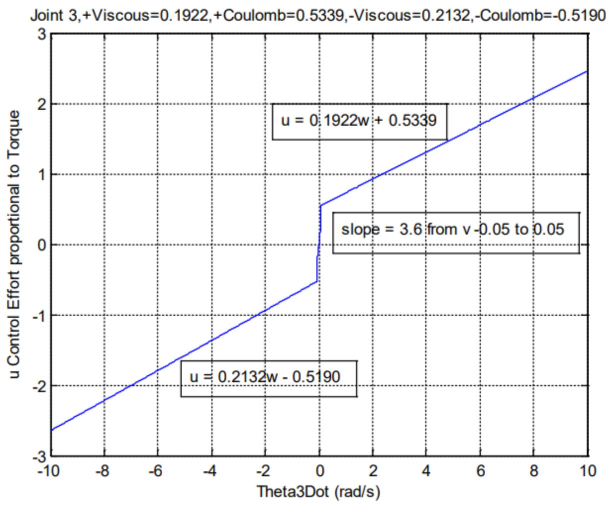
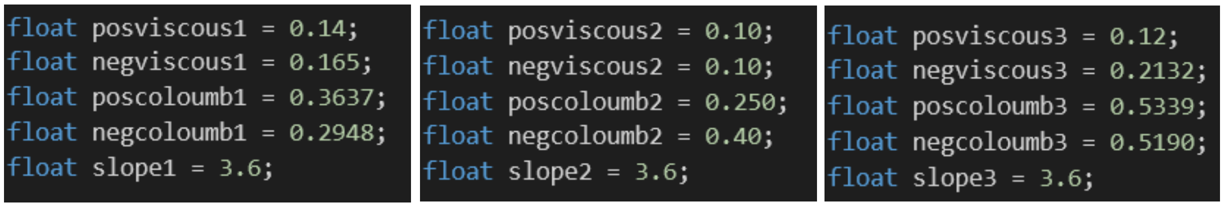


Figure 1. Friction Curves (Top Left: Joint 1, Top Right: Joint 2, Bottom: Joint 3

We added u_fric to our control effort after the full control effort was implemented. When tuning the coefficients, the control effort was set to just friction compensation which is u_fric. The friction compensation for each joint was modeled using a straight-line equation for different values of omega. The friction term has two terms namely viscous and Coulomb friction term. The experimental plot is a combination of these two terms. Coulomb friction accounts for static friction and viscous friction accounts for velocity-dependent friction. However, we should note that even with complex friction models it is extremely difficult to remove friction.

The friction coefficients that have been fine-tuned are:



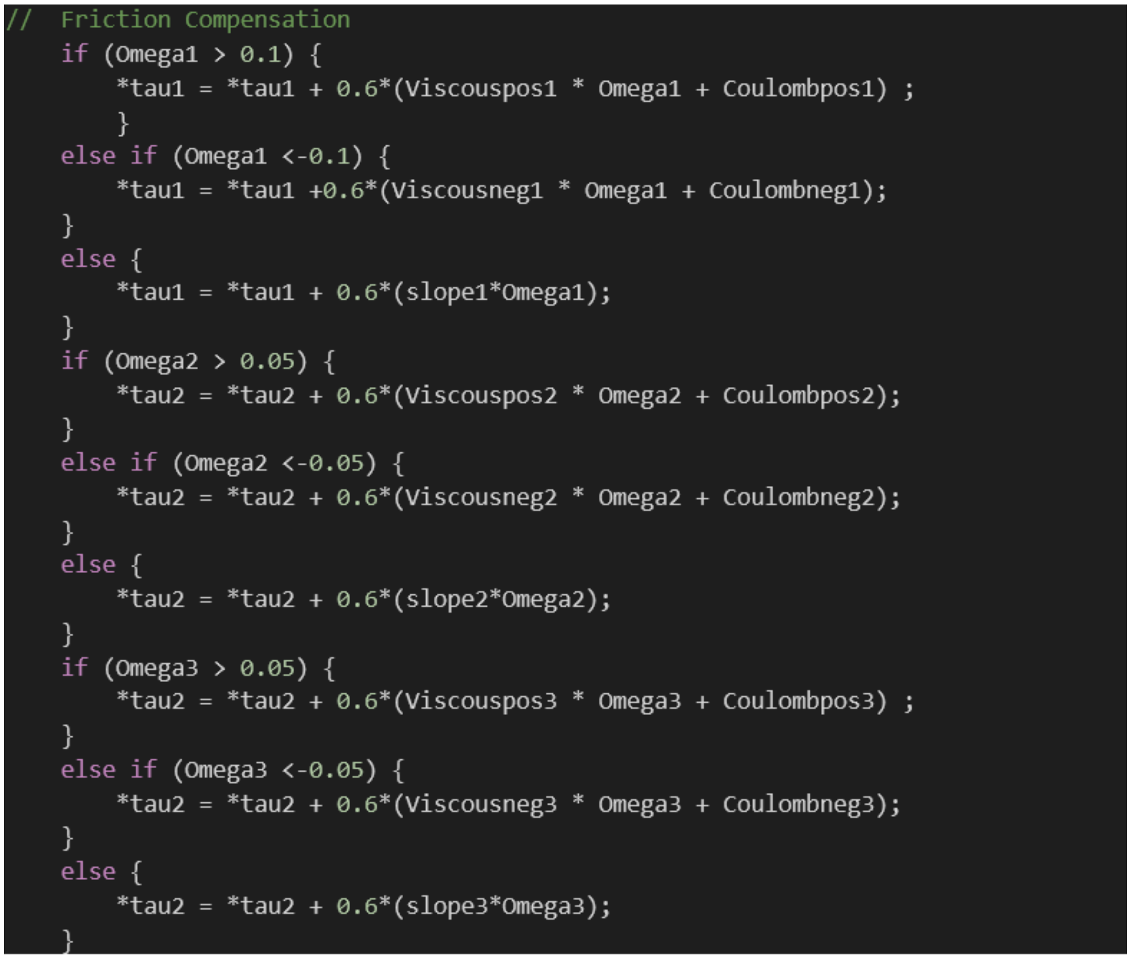
```
float posviscous1 = 0.14;
float negviscous1 = 0.165;
float poscoloumb1 = 0.3637;
float negcoloumb1 = 0.2948;
float slope1 = 3.6;
```

```
float posviscous2 = 0.10;
float negviscous2 = 0.10;
float poscoloumb2 = 0.250;
float negcoloumb2 = 0.40;
float slope2 = 3.6;
```

```
float posviscous3 = 0.12;
float negviscous3 = 0.2132;
float poscoloumb3 = 0.5339;
float negcoloumb3 = 0.5190;
float slope3 = 3.6;
```

Figure 2. Friction Coefficients for Joint 1, Joint 2, Joint 3 respectively

3.2 Implementation of Friction Compensation – C Code



```
//  Friction Compensation
    if (Omega1 > 0.1) {
        *tau1 = *tau1 + 0.6*(Viscouspos1 * Omega1 + Coulombpos1) ;
        }
    else if (Omega1 <-0.1) {
        *tau1 = *tau1 +0.6*(Viscousneg1 * Omega1 + Coulombneg1);
    }
    else {
        *tau1 = *tau1 + 0.6*(slope1*Omega1);
    }
    if (Omega2 > 0.05) {
        *tau2 = *tau2 + 0.6*(Viscouspos2 * Omega2 + Coulombpos2);
    }
    else if (Omega2 <-0.05) {
        *tau2 = *tau2 + 0.6*(Viscousneg2 * Omega2 + Coulombneg2);
    }
    else {
        *tau2 = *tau2 + 0.6*(slope2*Omega2);
    }
    if (Omega3 > 0.05) {
        *tau2 = *tau2 + 0.6*(Viscouspos3 * Omega3 + Coulombpos3) ;
    }
    else if (Omega3 <-0.05) {
        *tau2 = *tau2 + 0.6*(Viscousneg3 * Omega3 + Coulombneg3);
    }
    else {
        *tau2 = *tau2 + 0.6*(slope3*Omega3);
    }
```

Figure 3. Implementation of Friction Compensation for all three joints.

4 Implement the Inverse Dynamics Control Law on Joints Two and Three

A Simulink model was developed for a nonlinear simulation of equations with the parameters given in the lab manual. We should also note that to implement inverse dynamics control it is required to know the model very well. For joint 1, the same feedforward control implemented in lab2 was followed. To track the cubic polynomial reference trajectory, we implemented Inverse dynamics control using the following equations.

$$\tau = D(\theta)a_\theta + C(\theta,\dot{\theta})\dot{\theta} + g(\theta)$$

$$a_{\theta_2} = \ddot{\theta}_2^d + K_{P2} * \left(\theta_2^d - \theta_2\right) + K_{D2} * \left(\dot{\theta}_2^d - \dot{\theta}_2\right)$$

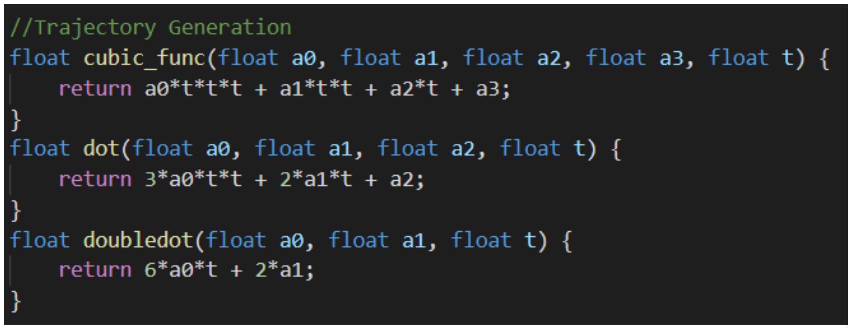
$$a_{\theta_3} = \ddot{\theta}_3^d + K_{P3} * \left(\theta_3^d - \theta_3\right) + K_{D3} * (\dot{\theta}_3^d - \dot{\theta}_3).$$

4.1 Inverse Dynamics Control

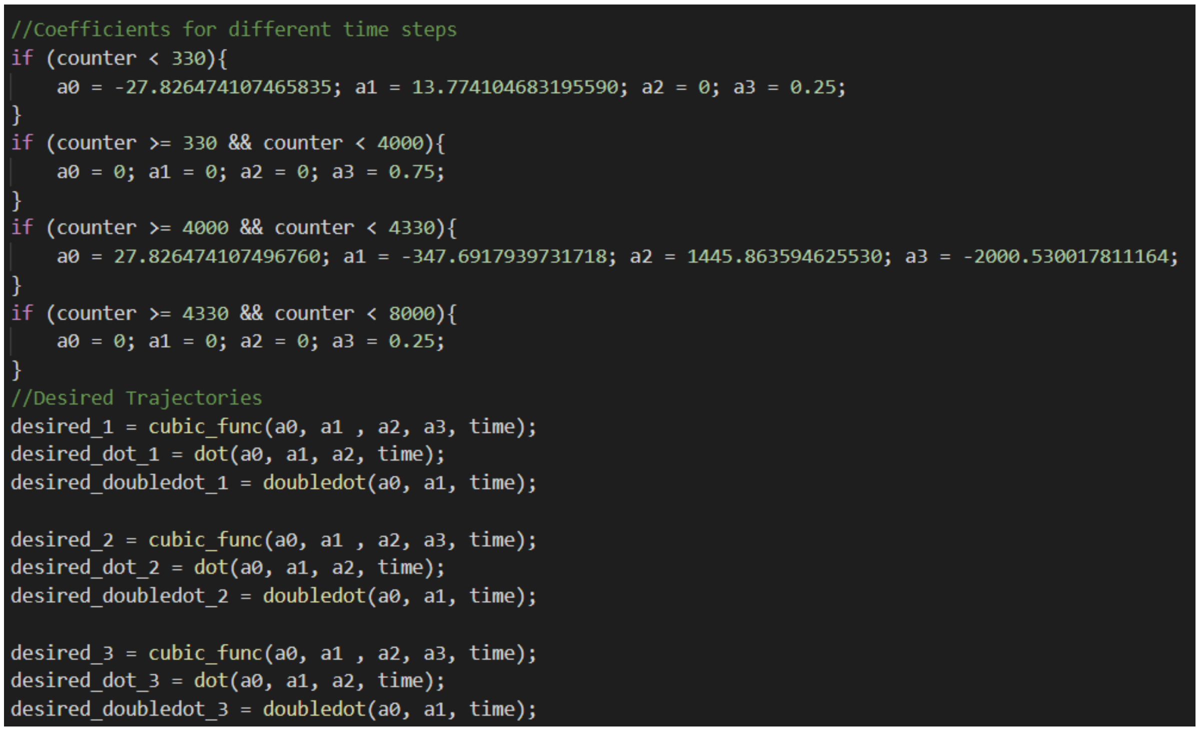
4.1.1 Implementation of the desired trajectory

The desired trajectory in the lab was to start at t = 0.0 at 0.25radians. And follow a cubic path for 0.33seconds to 0.75 radians. Then have the joint stay at 0.75 radians for 4 seconds. From 4 seconds to 4.33 seconds following a cubic trajectory back to 0.25 radians. This was repeated every 8 seconds so that there is a pause between each cubic path to the new joint angle.

We implemented a cubic trajectory function which determined the trajectory at t seconds. Having a cubic trajectory also allowed us to calculate the desired velocity and acceleration for each joint. For each time step, the coefficients of cubic trajectory differed. The determined coefficients for the mentioned trajectories are mentioned below, for each time interval. The intervals are 0-330ms, 330ms to 4000ms, 4000ms to 4330ms and 4330ms to 8000ms. These coefficients determined the q, q_dot, q_dotdot for each joint as shown in Figure 4.



```
//Trajectory Generation
float cubic_func(float a0, float a1, float a2, float a3, float t) {
    return a0*t*t*t + a1*t*t + a2*t + a3;
}
float dot(float a0, float a1, float a2, float t) {
    return 3*a0*t*t + 2*a1*t + a2;
}
float doubledot(float a0, float a1, float t) {
    return 6*a0*t + 2*a1;
}
```



```
//Coefficients for different time steps
if (counter < 330){
    a0 = -27.826474107465835; a1 = 13.774104683195590; a2 = 0; a3 = 0.25;
}
if (counter >= 330 && counter < 4000){
    a0 = 0; a1 = 0; a2 = 0; a3 = 0.75;
}
if (counter >= 4000 && counter < 4330){
    a0 = 27.826474107496760; a1 = -347.6917939731718; a2 = 1445.863594625530; a3 = -2000.530017811164;
}
if (counter >= 4330 && counter < 8000){
    a0 = 0; a1 = 0; a2 = 0; a3 = 0.25;
}
//Desired Trajectories
desired_1 = cubic_func(a0, a1 , a2, a3, time);
desired_dot_1 = dot(a0, a1, a2, time);
desired_doubledot_1 = doubledot(a0, a1, time);

desired_2 = cubic_func(a0, a1 , a2, a3, time);
desired_dot_2 = dot(a0, a1, a2, time);
desired_doubledot_2 = doubledot(a0, a1, time);

desired_3 = cubic_func(a0, a1 , a2, a3, time);
desired_dot_3 = dot(a0, a1, a2, time);
desired_doubledot_3 = doubledot(a0, a1, time);
```

Figure 4. Top: Desired Trajectory functions, Bottom: Desired trajectory calculation for each time interval

4.1.2 *Determining the values of $a_{\theta2}$ and $a_{\theta3}$*

Given the values of measured theta for each joint, desired states, theta_dot, error, and error_dot we determined $a_{\theta2}$ and $a_{\theta3}$ using the following equation

$$a_{\theta_2} = \ddot{\theta}_2^d + K_{P2} * (\theta_2^d - \theta_2) + K_{D2} * (\dot{\theta}_2^d - \dot{\theta}_2)$$
$$a_{\theta_3} = \ddot{\theta}_3^d + K_{P3} * (\theta_3^d - \theta_3) + K_{D3} * (\dot{\theta}_3^d - \dot{\theta}_3).$$

This equation is then used to determine the control effort for joint 2 and joint 3 respectively. The control effort was determined using the following equation for joint 2 and joint 3.

$$\tau = D(\theta)a_\theta + C(\theta,\dot{\theta})\dot{\theta} + g(\theta)$$

We also add the friction compensation u_fric which was determined in part 3 to the control effort to make the system act as if there was no friction.

We also added blocks to our Simulink model to track the error for the inverse dynamics control. The plot generated the error response for each joints. The updated Simulink block diagram is shown in Figure 5.

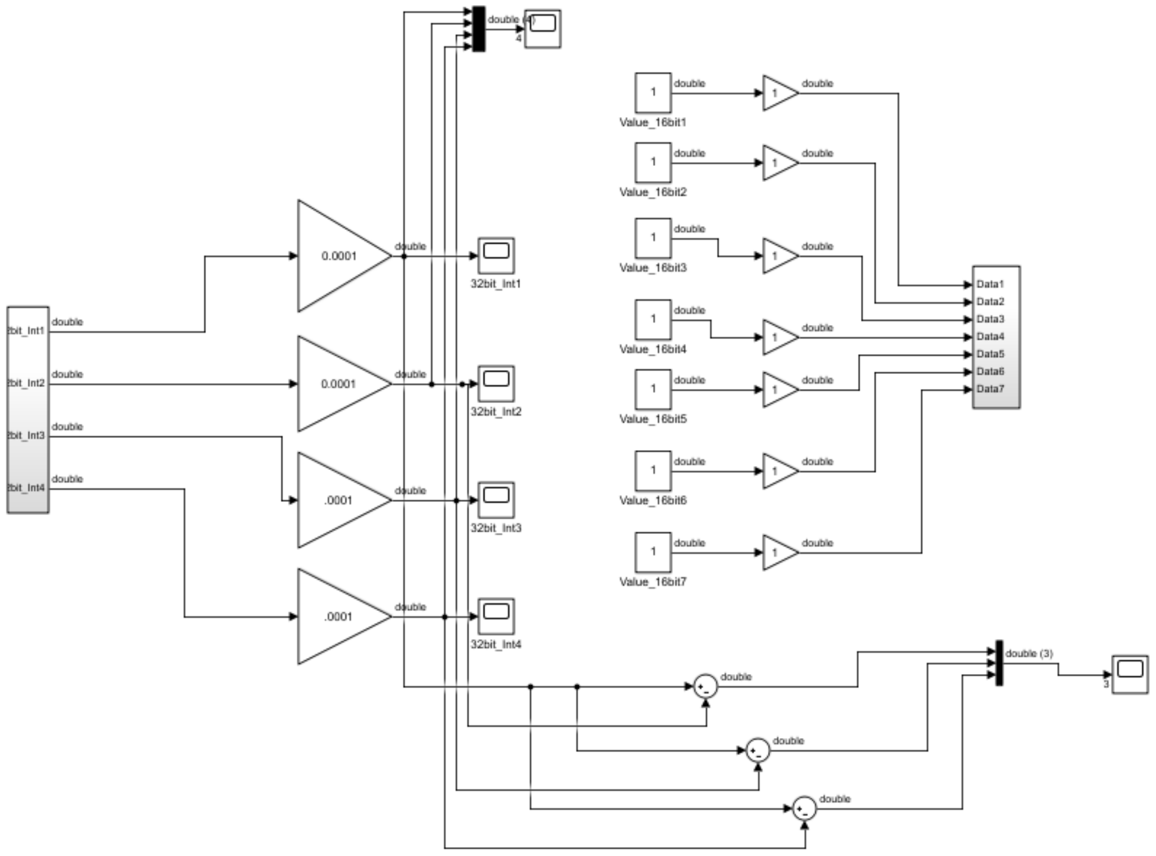
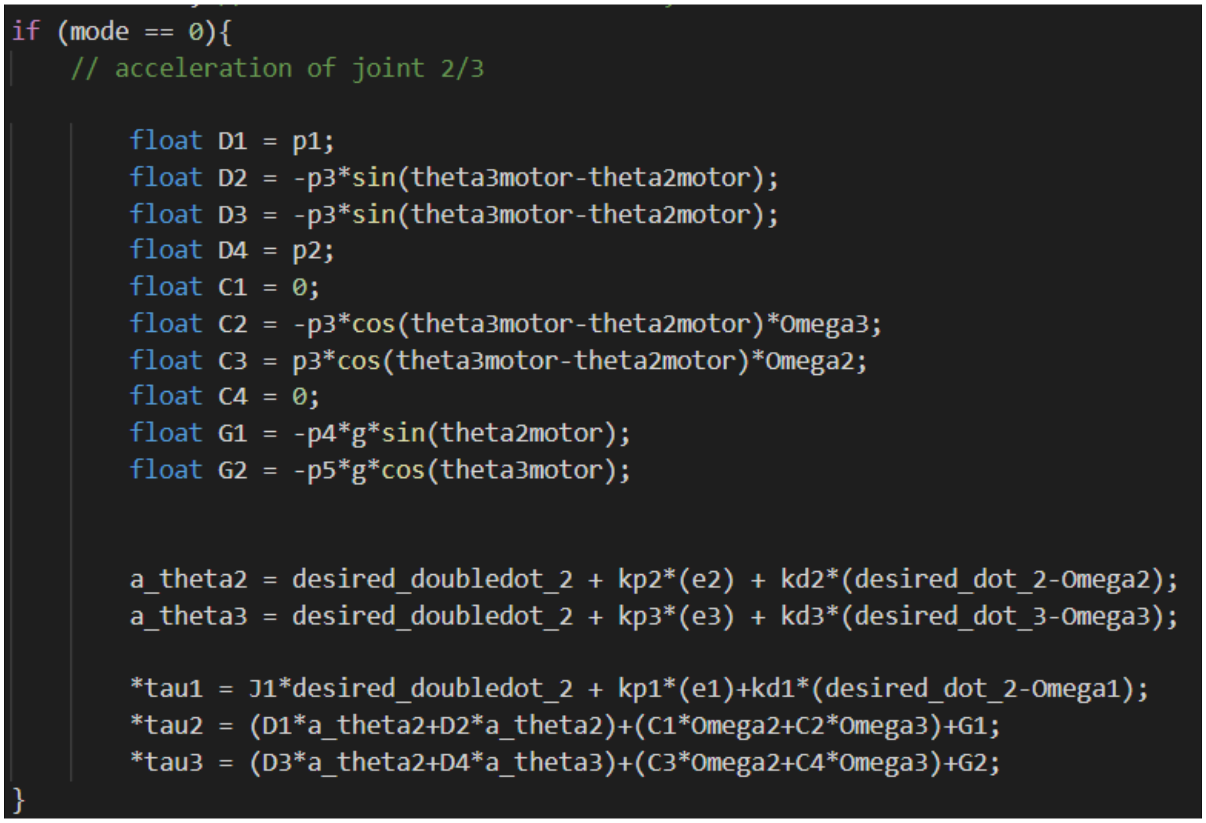


Figure 5. Updated Simulink Model to track errors

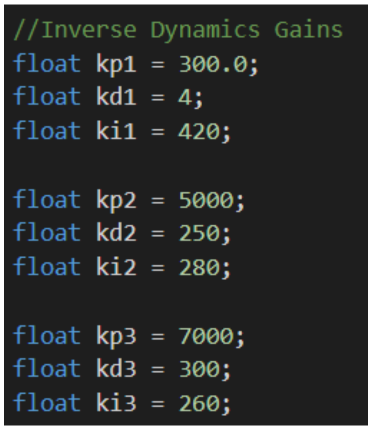
4.1.3 Implementation of Inverse Dynamics – C Code



```
if (mode == 0){
    // acceleration of joint 2/3

        float D1 = p1;
        float D2 = -p3*sin(theta3motor-theta2motor);
        float D3 = -p3*sin(theta3motor-theta2motor);
        float D4 = p2;
        float C1 = 0;
        float C2 = -p3*cos(theta3motor-theta2motor)*Omega3;
        float C3 = p3*cos(theta3motor-theta2motor)*Omega2;
        float C4 = 0;
        float G1 = -p4*g*sin(theta2motor);
        float G2 = -p5*g*cos(theta3motor);


        a_theta2 = desired_doubledot_2 + kp2*(e2) + kd2*(desired_dot_2-Omega2);
        a_theta3 = desired_doubledot_2 + kp3*(e3) + kd3*(desired_dot_3-Omega3);

        *tau1 = J1*desired_doubledot_2 + kp1*(e1)+kd1*(desired_dot_2-Omega1);
        *tau2 = (D1*a_theta2+D2*a_theta2)+(C1*Omega2+C2*Omega3)+G1;
        *tau3 = (D3*a_theta2+D4*a_theta3)+(C3*Omega2+C4*Omega3)+G2;
}
```

Figure 6. Implementation of Inverse Dynamics in C

The Kp and Kd gains for Inverse Dynamics control was different compared to the Feed Forward gain which we found in lab 2. The gains were tuned until the desired requirements were met, which were the same requirements as lab 2. We also observed that smaller gains made the system act as if it was more soft and back drivable. However, we wanted the error to be minimal and the rise time to be less than 300ms, so we increased the gains. This made the entire system not back drivable and made the errors very small, and we were able to reach the desired requirements using the following gains.



```
//Inverse Dynamics Gains
float kp1 = 300.0;
float kd1 = 4;
float ki1 = 420;

float kp2 = 5000;
float kd2 = 250;
float ki2 = 280;

float kp3 = 7000;
float kd3 = 300;
float ki3 = 260;
```

Figure 7. Gains of Inverse Dynamics Control

4.1.4 *Trajectory response and error plots without Mass Inverse Dynamics Control*

The errors for the plots were significantly low of the order of 4*10^{-3}. We decreased the error by increasing the gains. The trajectory followed by the robot is very close to the desired trajectory.

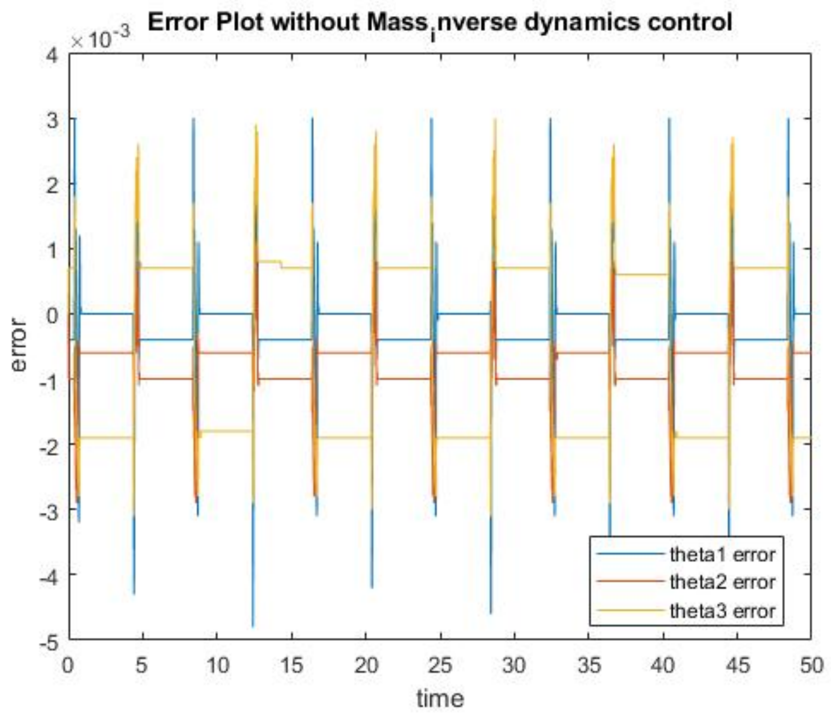


Figure 8. Error plot without Mass Inverse Dynamics Control

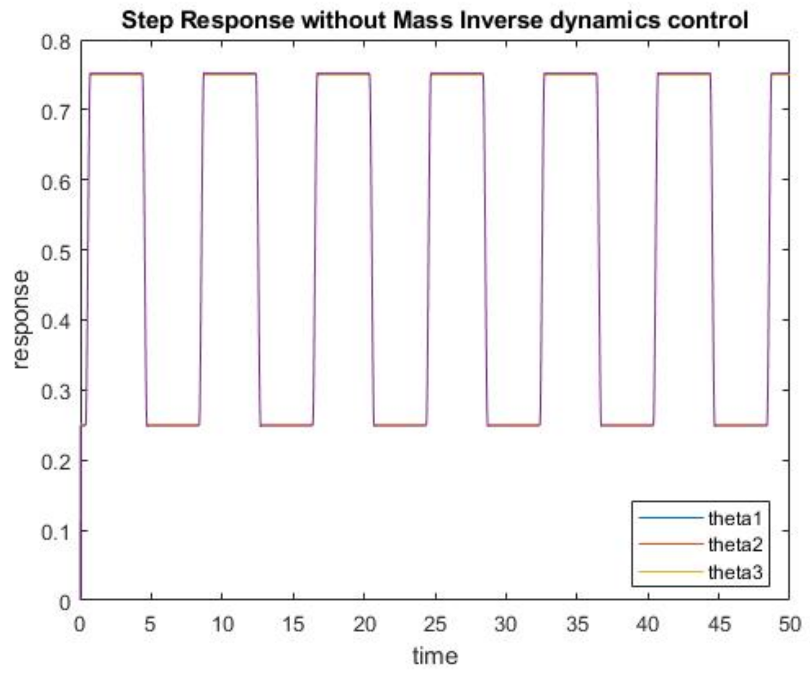
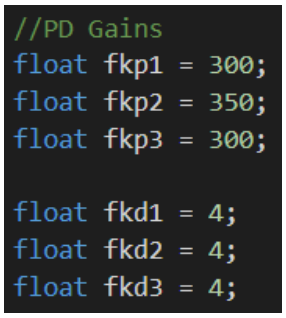


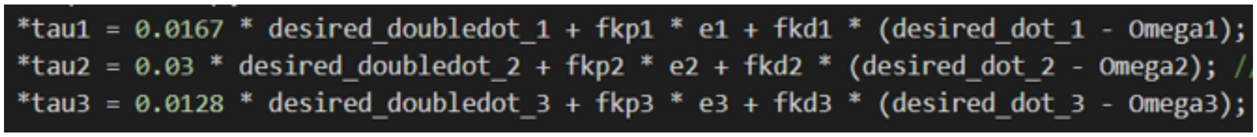
Figure 9. Figure 7: Trajectory Response without Mass Inverse Dynamics Control

4.2 PD Control

PD Control implementation was done in a similar fashion as we did in lab 2, but the only difference here was to add friction compensation. This is done for both the controllers, so once the torques were found they are updated by adding u_fric as shown in figure 3. The gains for PD control were different when compared to Inverse dynamics control and they are as follows:



```
//PD Gains
float fkp1 = 300;
float fkp2 = 350;
float fkp3 = 300;

float fkd1 = 4;
float fkd2 = 4;
float fkd3 = 4;
```



```
*tau1 = 0.0167 * desired_doubledot_1 + fkp1 * e1 + fkd1 * (desired_dot_1 - Omega1);
*tau2 = 0.03 * desired_doubledot_2 + fkp2 * e2 + fkd2 * (desired_dot_2 - Omega2); /
*tau3 = 0.0128 * desired_doubledot_3 + fkp3 * e3 + fkd3 * (desired_dot_3 - Omega3);
```

Figure 10. Top: PD Gains, Bottom: PD Implementation in C

4.2.5 Trajectory response and error plots

We were able to achieve similar errors for each joint by increasing the Kp and Kd gains compared to lab 2. The error was of the order of $4*10^{-3}$. We were not able to see significant differences between the controllers unless we switched between the controllers during the motion.

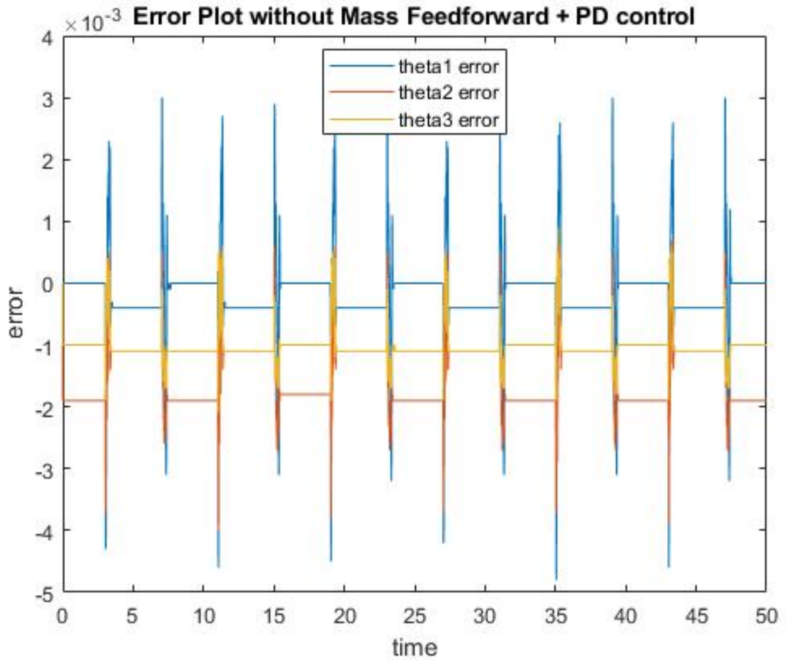


Figure 11. Error plot without Mass PD Control

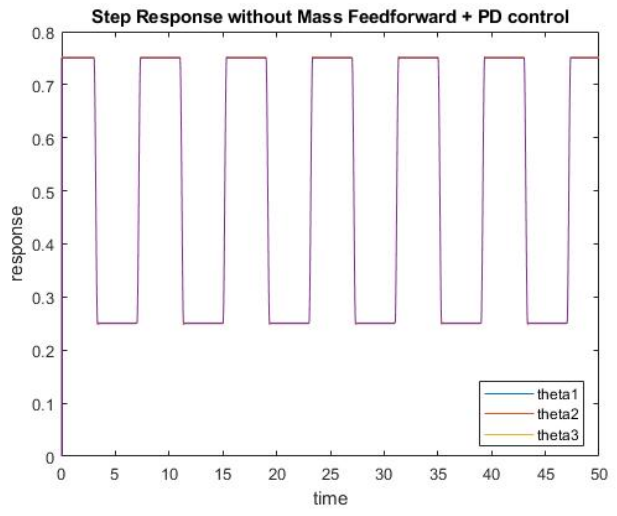
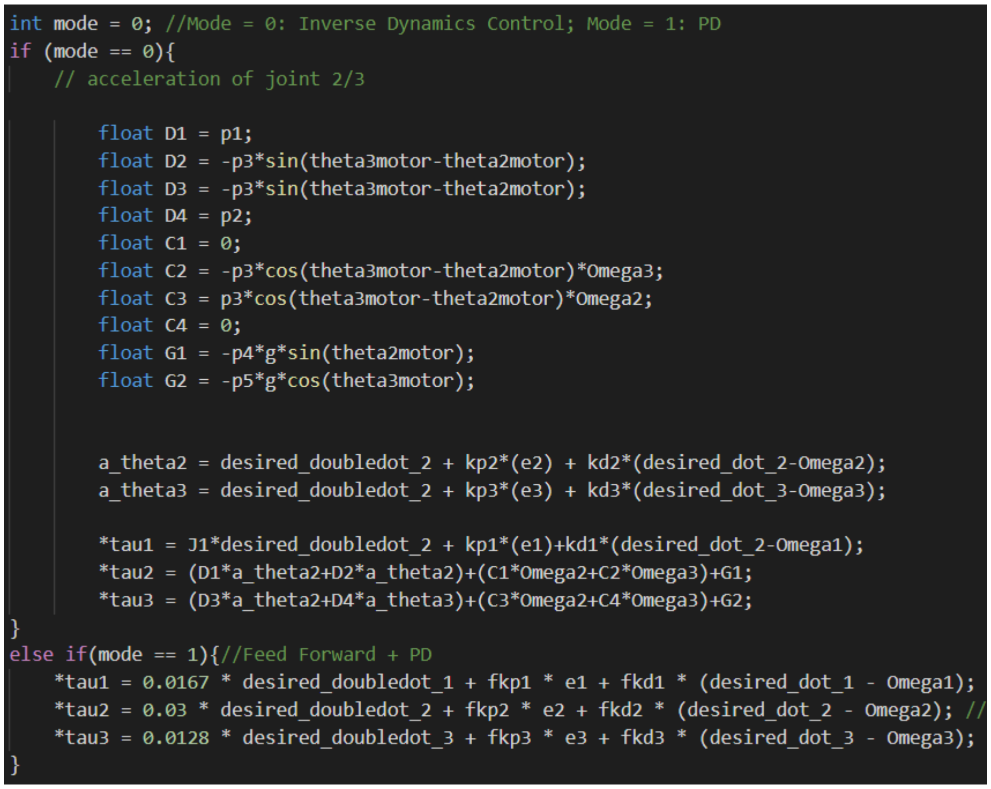


Figure 12. Trajectory Response without Mass PD Control

4.3 Compare Inverse Dynamics Controller to PD Controller

Once the inverse dynamics control was implemented, we created 2 modes for each controller to switch between them. This was done to observe how each controller behaved. The PD control from lab 2 was modified to have friction compensation in the equation. The PD control had to follow the same trajectory but quicker compared to lab 2. Once the modes were set up, we compared them using the watch expression window in Code Composer by switching between 0 and 1.



```
int mode = 0; //Mode = 0: Inverse Dynamics Control; Mode = 1: PD
if (mode == 0){
    // acceleration of joint 2/3

        float D1 = p1;
        float D2 = -p3*sin(theta3motor-theta2motor);
        float D3 = -p3*sin(theta3motor-theta2motor);
        float D4 = p2;
        float C1 = 0;
        float C2 = -p3*cos(theta3motor-theta2motor)*Omega3;
        float C3 = p3*cos(theta3motor-theta2motor)*Omega2;
        float C4 = 0;
        float G1 = -p4*g*sin(theta2motor);
        float G2 = -p5*g*cos(theta3motor);


        a_theta2 = desired_doubledot_2 + kp2*(e2) + kd2*(desired_dot_2-Omega2);
        a_theta3 = desired_doubledot_2 + kp3*(e3) + kd3*(desired_dot_3-Omega3);

        *tau1 = J1*desired_doubledot_2 + kp1*(e1)+kd1*(desired_dot_2-Omega1);
        *tau2 = (D1*a_theta2+D2*a_theta2)+(C1*Omega2+C2*Omega3)+G1;
        *tau3 = (D3*a_theta2+D4*a_theta3)+(C3*Omega2+C4*Omega3)+G2;
}
else if(mode == 1){//Feed Forward + PD
    *tau1 = 0.0167 * desired_doubledot_1 + fkp1 * e1 + fkd1 * (desired_dot_1 - Omega1);
    *tau2 = 0.03 * desired_doubledot_2 + fkp2 * e2 + fkd2 * (desired_dot_2 - Omega2); //
    *tau3 = 0.0128 * desired_doubledot_3 + fkp3 * e3 + fkd3 * (desired_dot_3 - Omega3);
}
```

Figure 13. Modes for Inverse Dynamics Control (Mode 0) and PD Control (Mode 1)

4.3.6 Trajectory response and error plots

It can be observed that at 15 seconds the error increases because of a switch in controller. So, we collected the results of the inverse dynamics controller till 15 seconds and switched to the PD controller till 35 seconds, and then again changed back to the inverse dynamics controller. The abrupt difference in error plots for each joint can be observed in Figure 14. Hence, we can comment that the inverse dynamics controller performs slightly better than the PD controller.

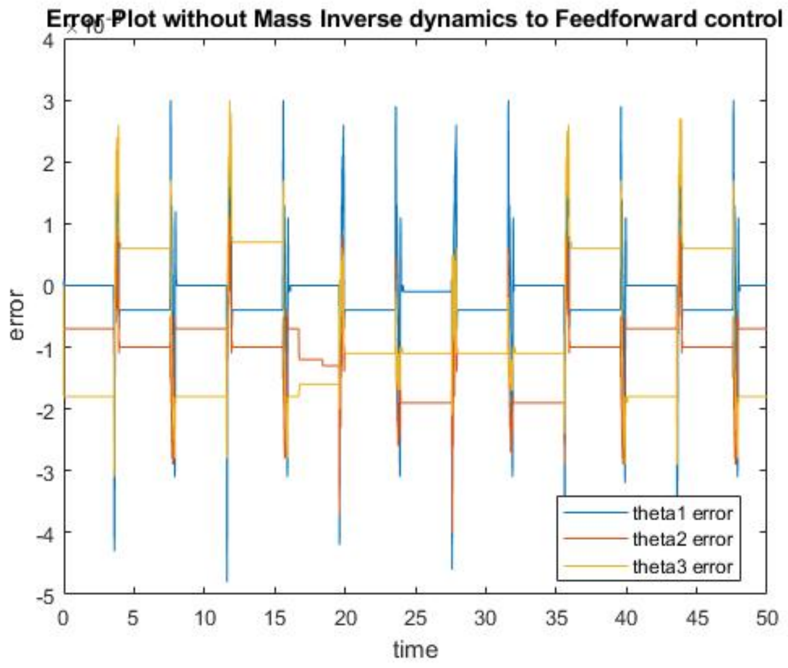


Figure 14. Error Plot without Mass from Inverse Dynamics Control to PD Control

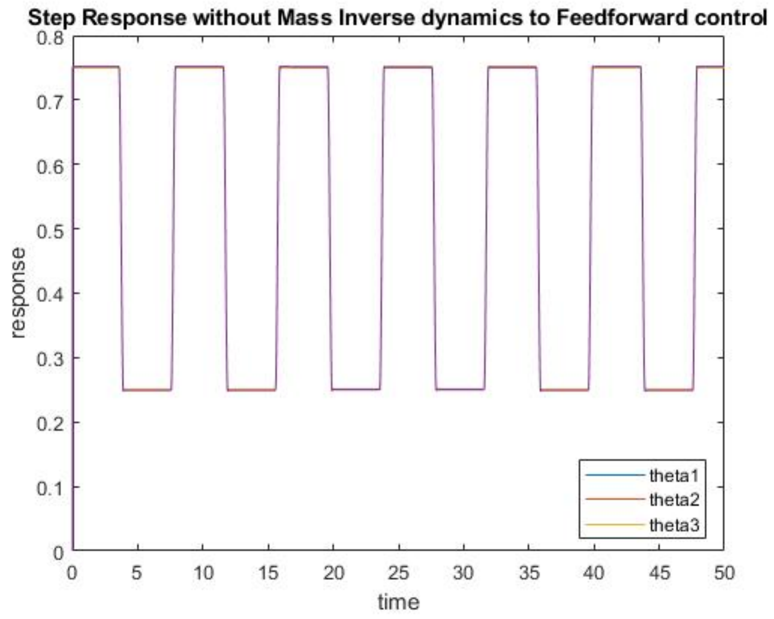
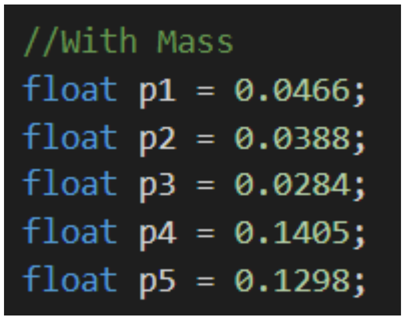


Figure 15. Trajectory Response without Mass from Inverse Dynamics Control to PD Control

5 Control Implementation with Mass

Firstly, we had to change the parameters p1, p2, p3, p4, p5 from the updated MATLAB file mentioned in the manual. We also noticed that the updated parameters were found using the parallel axis theorem to add in the mass. This gave us updated values for these parameters which determine most of the dynamics of the system. The updated values are shown in Figure 16:



```
//With Mass
float p1 = 0.0466;
float p2 = 0.0388;
float p3 = 0.0284;
float p4 = 0.1405;
float p5 = 0.1298;
```

Figure 16. Updated parameters with the disc

Once the parameters we tested our control implementation for both the control strategies implemented. And the results can be seen in the next sections.

5.1 Inverse Dynamics Control

We observed that with mass, the inverse dynamics control was able get the error of order 10^{-2} which can be seen in Figure 17. It can be noticed that the errors were quite small even with an increase in mass. From this, we can reasonably argue that the parameters used to determine the dynamics of the system are quite accurate. Even with the mass the robot was able to closely follow the desired trajectory.

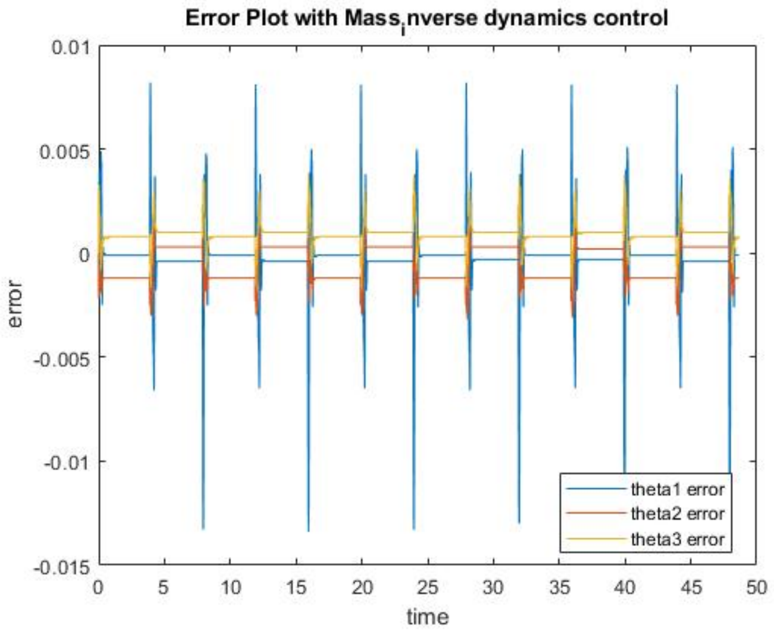


Figure 17. Error plot with Mass Inverse Dynamics Control

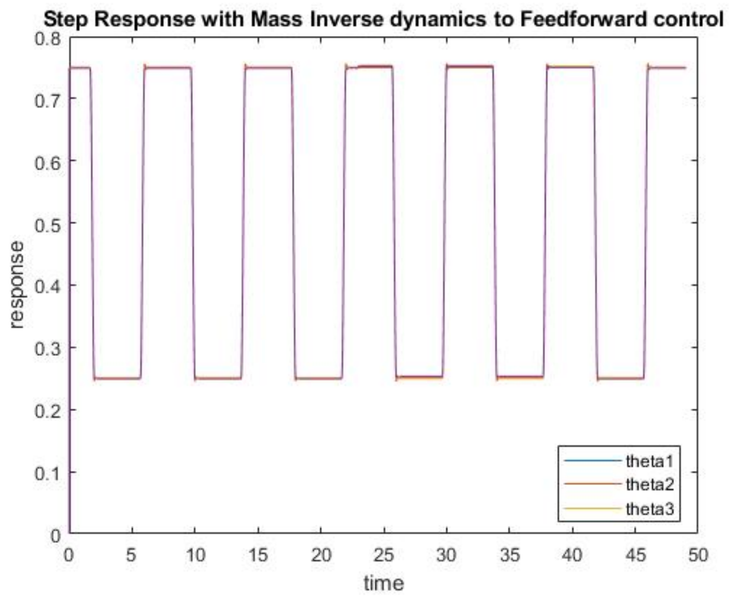


Figure 18. Trajectory Response with Mass Inverse Dynamics Control

5.2 PD Control

5.2.7 Trajectory response and error plots

With PD control we were able to get a similar result as that of an inverse dynamics control, we got the error of the order of 10^{-2} which can be observed in Figure 19. This setting with increased gains was able to track desired closely, however, we can observe from the plots that inverse dynamics control outperformed PD control.

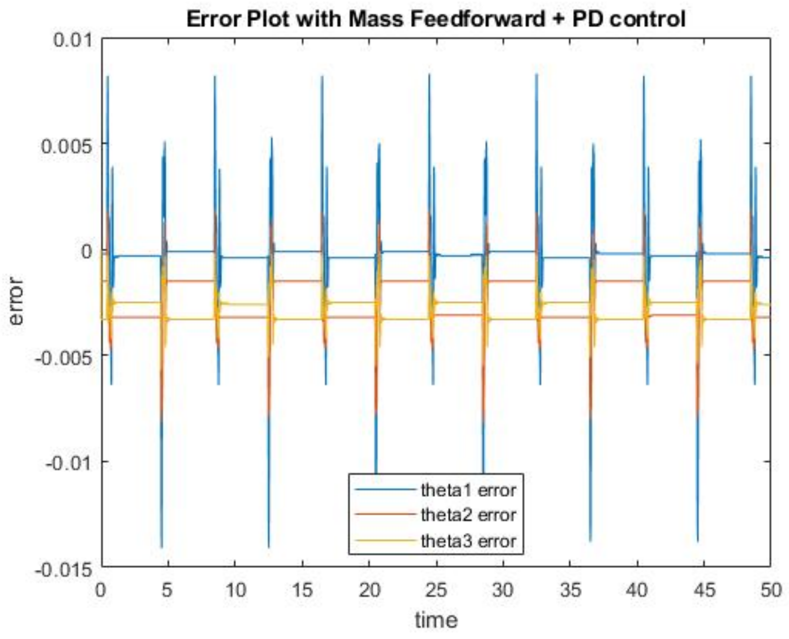


Figure 19. Error plot with Mass PD Control

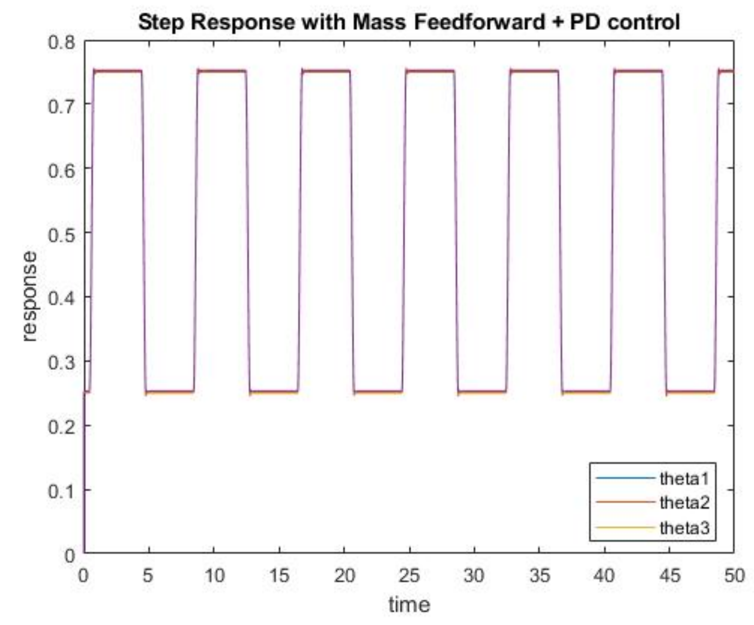


Figure 20. Trajectory Response with Mass PD Control

5.3 Compare Inverse Dynamics Controller to PD Controller

Finally, we compared both the controllers by switching between the two when the mass was added. We can observe that from t = 25seconds to t = 38 seconds the errors are larger compared to otherwise. This is because in this region we operated mode was PD control. And in the remaining part, it was inverse dynamics control. From this error plot, we can say that it performed better for both with mass and without mass control.

5.3.8 Trajectory response and error plots

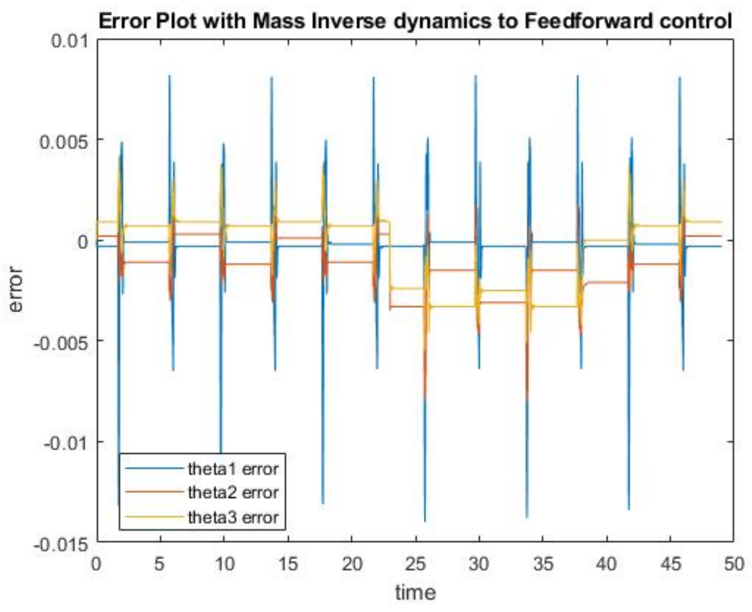


Figure 21. Error Plot with Mass from Inverse Dynamics Control to PD Control

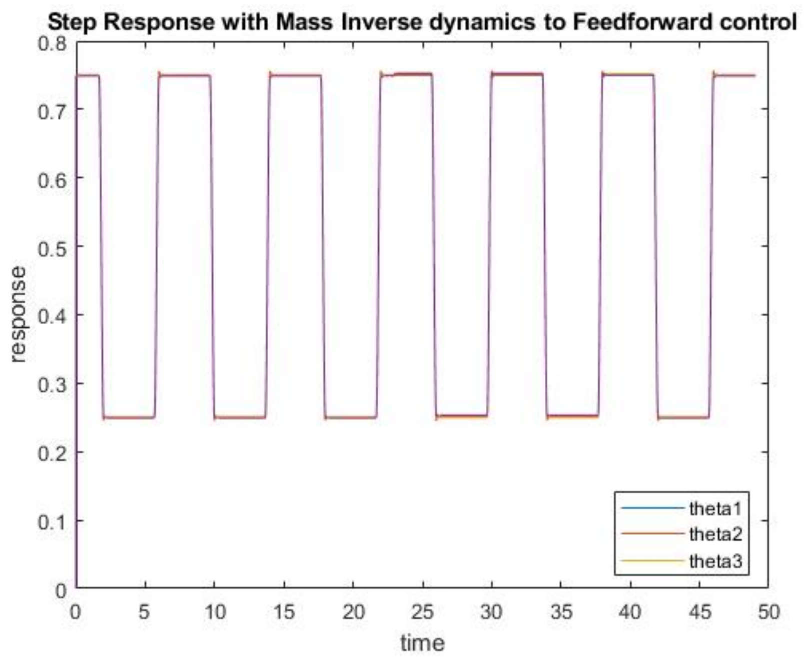


Figure 22. Trajectory Response with Mass from Inverse Dynamics Control to PD Control

6 Conclusion

During the lab, we implemented two control strategies namely Inverse Dynamics control and PD control. Both controls had friction compensation which made them act as if there was no friction. Both controllers were tuned to meet the desired requirements of a rise time less than 300ms and a percent overshoot of less than 1% with minimal steady state error. Then to make the trajectory smooth we also implemented a cubic trajectory with reference to the trajectory mentioned in the report. From the plots, we can say that the parameters for with and without mass are reasonably accurate. However, we think that for with mass situation the accuracy of calculating the dynamics of the system can be further refined. Moreover, we can also improve on the friction modeling, but unless we need super high accuracy, we need not go with refining the friction coefficients. Finally, for the lab and the desired level of requirements we can conclude that the parameters are good enough. Finally, we compared both Inverse Dynamics control and PD control and observed that Inverse Dynamics control performs slightly better than PD control.

7 C Code

```c
#include <tistdtypes.h>
#include <coecsl.h>
#include "user_includes.h"
#include "math.h"

// These two offsets are only used in the main file user_CRSRobot.c  You just need to
create them here and find the correct offset and then these offset will adjust the
encoder readings
float offset_Enc2_rad = -0.415; //-0.37;
float offset_Enc3_rad = 0.233; //0.27;

// Your global variables.
float px = 0; // x coordinate of end effector
float py = 0; // y coordinate of end effector
float pz = 0; // z coordinate of end effector
float theta1m_IK = 0; // theta1 motor from inverse kinematics
float theta2m_IK = 0; // theta2 motor from inverse kinematics
float theta3m_IK = 0; // theta3 motor from inverse kinematics
long mycount = 0;
long counter = 0;
int cycle = 0;

// Omega calculations
float Theta1_old = 0;
float Omega1_old1 = 0;
float Omega1_old2 = 0;
float Omega1 = 0;
float e1_old = 0;
float integral1_old = 0;

float Theta2_old = 0;
float Omega2_old1 = 0;
float Omega2_old2 = 0;
float Omega2 = 0;
float e2_old = 0;
float integral2_old = 0;

float Theta3_old = 0;
float Omega3_old1 = 0;
float Omega3_old2 = 0;
float Omega3 = 0;
float e3_old = 0;
float integral3_old = 0;
```

```
// Constants
float dt = 0.001;
float threshold1 = 0.01;
float threshold2 = 0.06;
float threshold3 = 0.02;

//Inverse Dynamics Gains
float kp1 = 300.0;
float kd1 = 4;
float ki1 = 420;

float kp2 = 5000;
float kd2 = 250;
float ki2 = 280;

float kp3 = 7000;
float kd3 = 300;
float ki3 = 260;

//float kp1 = 85.0;
//float kd1 = 5;
//float ki1 = 420;
//
//float kp2 = 200;
//float kd2 = 20;
//float ki2 = 280;
//
//float kp3 = 200;
//float kd3 = 20;
//float ki3 = 260;

//PD Gains
float fkp1 = 300;
float fkp2 = 350;
float fkp3 = 300;

float fkd1 = 4;
float fkd2 = 4;
float fkd3 = 4;


float theta_dotdot = 0.0;
float e1 = 0;
float e2 = 0;
float e3 = 0;

float x_desired = 0;
float y_desired = 0;
float z_desired = 0;

//Friction Coefficients
float posviscous1 = 0.14;
float negviscous1 = 0.165;
float poscoloumb1 = 0.3637;
float negcoloumb1 = 0.2948;
float slope1 = 3.6;

float posviscous2 = 0.10;
float negviscous2 = 0.10;
float poscoloumb2 = 0.250;
float negcoloumb2 = 0.40;
float slope2 = 3.6;

float posviscous3 = 0.12;
float negviscous3 = 0.2132;
float poscoloumb3 = 0.5339;
float negcoloumb3 = 0.5190;
float slope3 = 3.6;

//Without Mass
float p1 = 0.0300;
float p2 = 0.0128;
float p3 = 0.0076;
float p4 = 0.0753;
float p5 = 0.0298;

//With Mass
//float p1 = 0.0466;
//float p2 = 0.0388;
//float p3 = 0.0284;
//float p4 = 0.1405;
//float p5 = 0.1298;

float a_theta2;
float a_theta3;

float sintheta2;
```

```c
float costheta2;
float sintheta3;
float costheta3;
float g = 9.81;

float J1 = 0.0167;
float J2 = 0.03;
float J3 = 0.0128;

float a0,a1,a2,a3;
float tau1cur,tau2cur,tau3cur;

float desired_1, desired_2, desired_3;
float desired_dot_1, desired_dot_2, desired_dot_3;
float desired_doubledot_1, desired_doubledot_2, desired_doubledot_3;

#pragma DATA_SECTION(whattoprint, ".my_vars")
float whattoprint = 0.0;

#pragma DATA_SECTION(whatnottoprint, ".my_vars")
float whatnottoprint = 0.0;

#pragma DATA_SECTION(theta1array, ".my_arrs")
float theta1array[100];

#pragma DATA_SECTION(theta2array, ".my_arrs")
float theta2array[100];

long arrayindex = 0;

float printtheta1motor = 0;
float printtheta2motor = 0;
float printtheta3motor = 0;

// Assign these float to the values you would like to plot in Simulink
float Simulink_PlotVar1 = 0;
float Simulink_PlotVar2 = 0;
float Simulink_PlotVar3 = 0;
float Simulink_PlotVar4 = 0;

void inverseKinematics(float px, float py, float pz) {
    // Calculate DH angles from end effector position
    float theta1 = atan2(py, px); // DH theta 1
    float z = pz - 10;
    float beta = sqrt(px*px + py*py);
    float L = sqrt(z*z + beta*beta);
    float theta3 = acos((L*L - 200)/200); // DH theta 3
    float theta2 = -theta3/2 - atan2(z, beta); // DH theta 2


    // Convert DH angles to motor angles
    theta1m_IK = theta1;
    theta2m_IK = (theta2 + PI/2);
    theta3m_IK = (theta3 + theta2m_IK - PI/2);
}

// Velocity estimation
void omega(float theta1motor, float theta2motor, float theta3motor) {
    Omega1 = (theta1motor - Theta1_old)/0.001;
    Omega1 = (Omega1 + Omega1_old1 + Omega1_old2)/3.0;

    Theta1_old = theta1motor;

    Omega1_old2 = Omega1_old1;
    Omega1_old1 = Omega1;

    Omega2 = (theta2motor - Theta2_old)/0.001;
    Omega2 = (Omega2 + Omega2_old1 + Omega2_old2)/3.0;

    Theta2_old = theta2motor;

    Omega2_old2 = Omega2_old1;
    Omega2_old1 = Omega2;

    Omega3 = (theta3motor - Theta3_old)/0.001;
    Omega3 = (Omega3 + Omega3_old1 + Omega3_old2)/3.0;

    Theta3_old = theta3motor;

    Omega3_old2 = Omega3_old1;
    Omega3_old1 = Omega3;
}


//Trajectory Generation
float cubic_func(float a0, float a1, float a2, float a3, float t) {
    return a0*t*t*t + a1*t*t + a2*t + a3;
}
float dot(float a0, float a1, float a2, float t) {
    return 3*a0*t*t + 2*a1*t + a2;
```

```
}
float doubledot(float a0, float a1, float t) {
    return 6*a0*t + 2*a1;
}

void  lab(float  theta1motor,float  theta2motor,float  theta3motor,float  *tau1,float
*tau2,float *tau3, int error) {

    //
    if (counter == 8000)
        {
            counter = 0;
        }

    float time = counter*0.001;

    //Coefficients for different time steps
    if (counter < 330){
        a0 = -27.826474107465835; a1 = 13.774104683195590; a2 = 0; a3 = 0.25;
    }
    if (counter >= 330 && counter < 4000){
        a0 = 0; a1 = 0; a2 = 0; a3 = 0.75;
    }
    if (counter >= 4000 && counter < 4330){
        a0 = 27.826474107496760; a1 = -347.6917939731718; a2 = 1445.863594625530; a3
= -2000.530017811164;
    }
    if (counter >= 4330 && counter < 8000){
        a0 = 0; a1 = 0; a2 = 0; a3 = 0.25;
    }
    //Desired Trajectories
    desired_1 = cubic_func(a0, a1 , a2, a3, time);
    desired_dot_1 = dot(a0, a1, a2, time);
    desired_doubledot_1 = doubledot(a0, a1, time);

    desired_2 = cubic_func(a0, a1 , a2, a3, time);
    desired_dot_2 = dot(a0, a1, a2, time);
    desired_doubledot_2 = doubledot(a0, a1, time);

    desired_3 = cubic_func(a0, a1 , a2, a3, time);
    desired_dot_3 = dot(a0, a1, a2, time);
    desired_doubledot_3 = doubledot(a0, a1, time);

    // Calculate/Update omegas
    omega(theta1motor, theta2motor, theta3motor);
```

```
//   Desired theta - current theta
    float e1 = desired_1 - theta1motor;
    float e2 = desired_2 - theta2motor;
    float e3 = desired_3 - theta3motor;


    // Integral approximation
    float integral1 = integral1_old + (e1 + e1_old) * dt / 2;
    float integral2 = integral2_old + (e2 + e2_old) * dt / 2;
    float integral3 = integral3_old + (e3 + e3_old) * dt / 2;

    // Prevents integral windup
    if (fabs(e1) > threshold1) {
        integral1 = 0;
        integral1_old = 0;
    }

    if (fabs(e2) > threshold2) {
            integral2 = 0;
            integral2_old  = 0;
    }

    if (fabs(e3) > threshold3) {
            integral3 = 0;
            integral3_old = 0;
    }

    int mode = 0; //Mode = 0: Inverse Dynamics Control; Mode = 1: PD
    if (mode == 0){
        // acceleration of joint 2/3
//
            float D1 = p1;
            float D2 = -p3*sin(theta3motor-theta2motor);
            float D3 = -p3*sin(theta3motor-theta2motor);
            float D4 = p2;
            float C1 = 0;
            float C2 = -p3*cos(theta3motor-theta2motor)*Omega3;
            float C3 = p3*cos(theta3motor-theta2motor)*Omega2;
            float C4 = 0;
            float G1 = -p4*g*sin(theta2motor);
            float G2 = -p5*g*cos(theta3motor);


            a_theta2 = desired_doubledot_2 + kp2*(e2) + kd2*(desired_dot_2-Omega2);
            a_theta3 = desired_doubledot_2 + kp3*(e3) + kd3*(desired_dot_3-Omega3);

            *tau1 = J1*desired_doubledot_2 + kp1*(e1)+kd1*(desired_dot_2-Omega1);
            *tau2 = (D1*a_theta2+D2*a_theta2)+(C1*Omega2+C2*Omega3)+G1;
            *tau3 = (D3*a_theta2+D4*a_theta3)+(C3*Omega2+C4*Omega3)+G2;
    }
    else if(mode == 1){//Feed Forward + PD
         *tau1 = 0.0167 * desired_doubledot_1 + fkp1 * e1 + fkd1 * (desired_dot_1 -
Omega1); // + ki1 * integral1;
         *tau2 = 0.03 * desired_doubledot_2 + fkp2 * e2 + fkd2 * (desired_dot_2 -
Omega2); // + ki2 * integral2;
         *tau3 = 0.0128 * desired_doubledot_3 + fkp3 * e3 + fkd3 * (desired_dot_3 -
Omega3);
    }

//
     //  Friction Compensation
    //If omega greater than 0.1 (Max Velocity) for joint 1
    if (Omega1 > 0.1) {
        *tau1 = *tau1 + 0.6*(Viscouspos1 * Omega1 + Coulombpos1) ;
        }
    //If omega greater than -0.1 (Min Velocity) for joint 1
    else if (Omega1 <-0.1) {
        *tau1 = *tau1 +0.6*(Viscousneg1 * Omega1 + Coulombneg1);
    }
    //If omega between min and max velocity for joint 1
    else {
        *tau1 = *tau1 + 0.6*(slope1*Omega1);
    }

    //If omega greater than 0.05(Max Velocity) for joint 2
    if (Omega2 > 0.05) {
        *tau2 = *tau2 + 0.6*(Viscouspos2 * Omega2 + Coulombpos2);
    }
    //If omega greater than -0.05 (Min Velocity) for joint 2
    else if (Omega2 <-0.05) {
        *tau2 = *tau2 + 0.6*(Viscousneg2 * Omega2 + Coulombneg2);
    }
    //If omega between min and max velocity for joint 2
    else {
        *tau2 = *tau2 + 0.6*(slope2*Omega2);
    }
    //If omega greater than -0.05 (Max Velocity) for joint 3
    if (Omega3 > 0.05) {
        *tau2 = *tau2 + 0.6*(Viscouspos3 * Omega3 + Coulombpos3) ;
    }
    //If omega greater than -0.05 (Min Velocity) for joint 3
    else if (Omega3 <-0.05) {
        *tau2 = *tau2 + 0.6*(Viscousneg3 * Omega3 + Coulombneg3);
    }
    //If omega between min and max velocity for joint 3
    else {
        *tau2 = *tau2 + 0.6*(slope3*Omega3);
    }

    //Prevents integral windup
    if (*tau1 >= 5) {
        *tau1 = 5;
        integral1 = integral1_old;
    } else if (*tau1 < -5) {
        *tau1 = -5;
        integral1 = integral1_old;
    }

    if (*tau2 >= 5) {
        *tau2 = 5;
        integral2 = integral2_old;
    } else if (*tau2 < -5) {
        *tau2 = -5;
        integral2 = integral2_old;
    }

    if (*tau3 >= 5) {
        *tau3 = 5;
        integral3 = integral3_old;
    } else if (*tau3 < -5) {
        *tau3 = -5;
        integral3 = integral3_old;
    }

    e1_old = e1;
    e2_old = e2;
    e3_old = e3;
    integral1_old = integral1;
    integral2_old = integral2;
    integral3_old = integral3;


    // save past states
    if ((mycount%50)==0) {
        theta1array[arrayindex] = theta1motor;
        theta2array[arrayindex] = theta2motor;

        if (arrayindex >= 100) {
            arrayindex = 0;
        } else {
            arrayindex++;
        }

    }

    if ((mycount%500)==0) {
        if (whattoprint > 0.5) {
            serial_printf(&SerialA, "I love robotics\n\r");
        } else {
            printtheta1motor = theta1motor;
            printtheta2motor = theta2motor;
            printtheta3motor = theta3motor;
             SWI_post(&SWI_printf); //Using a SWI to fix SPI issue from sending too
many floats.
        }

        GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1; // Blink LED on Control Card
        GpioDataRegs.GPBTOGGLE.bit.GPIO60 = 1; // Blink LED on Emergency Stop Box
    }



    Simulink_PlotVar1 = desired_1;  //yellow
    Simulink_PlotVar2 = theta1motor; //blue
    Simulink_PlotVar3 = theta2motor; //orange
    Simulink_PlotVar4 = theta3motor; //green
    mycount++;
    counter++;
}

void printing(void){
    // Printing (theta1, theta2, theta3, px, py, pz) and then on a new line (theta1_IK,
theta2_IK, theta3_IK)
    serial_printf(&SerialA, "px: %.2f, py: %.2f, pz: %.2f \n\r", px, py, pz);
}
```