ME 446
Robot Dynamics and Control
Lab 3 Report

Diyu Yang (dyang37)

Dongbo Wang (dwang49)

Instructor: Daniel Block

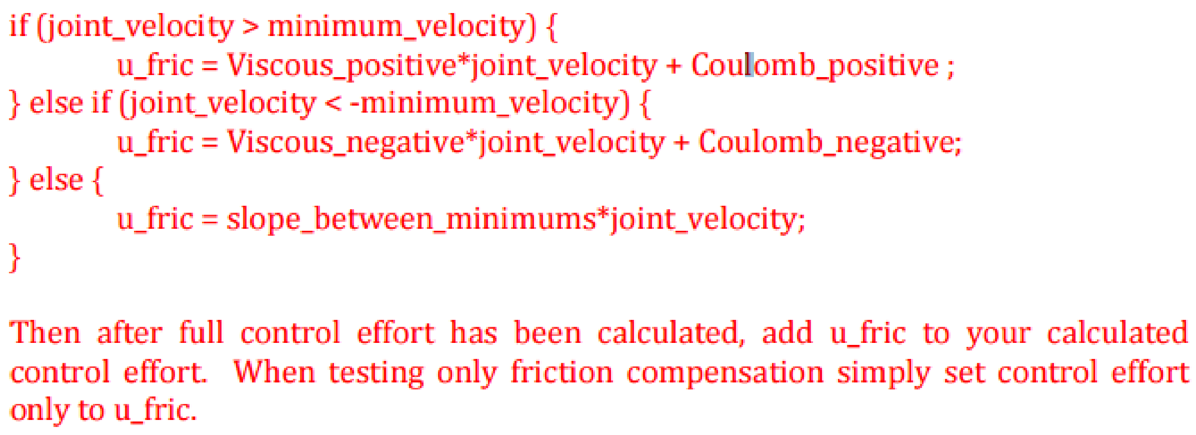
Section: Wed. 2-5pm

Table of Content

Part 1: Implement Friction Compensation

1.1 Identified Friction Curves

To implement the friction compensation, we followed the pseudocode shown in **Figure 1** below.



```
if (joint_velocity > minimum_velocity) {
        u_fric = Viscous_positive*joint_velocity + Coulomb_positive ;
} else if (joint_velocity < -minimum_velocity) {
        u_fric = Viscous_negative*joint_velocity + Coulomb_negative;
} else {
        u_fric = slope_between_minimums*joint_velocity;
}
```

Then after full control effort has been calculated, add u_fric to your calculated control effort. When testing only friction compensation simply set control effort only to u_fric.

Figure 1. Pseudocode for Friction Compensation

So basically we are assuming that the frictional force is linearly related to joint velocity. So we estimate the frictional force using the equations below:

$$u = \begin{cases} v_p w + c_p; & w > w_{min} \\ sw; & -w_{min} \le w \le w_{min} \\ v_n w + c_n; & w < -w_{min} \end{cases}$$

Where v_p = Viscous_positive, v_n = Viscous_negative, c_p =Coulombs_positive, c_n=Coulombs_negative, s = slope_between_minimums.

We tune these parameters by pushing the links with an initial force and looking at the free motion of the robot in both forward (velocity > 0) and reverse (velocity < 0) directions.

By pushing the robot links with large initial force, we were able to obtain a free motion with a high speed. At high speed (for both directions), the Coulumbs term become negligible, so a motion with increasing speed indicates that the magnitude of our viscous term is too large and vice versa.

Similarly, at low speed, the Coulombs force plays a bigger role in the motion of the link. Therefore, at low speed (this speed should still be larger than the minimum velocity term above), an accelerating motion indicates that the magnitude of Coulombs term is too large, and vice versa.

Using the procedures described above, we were able to tune the friction compensation for each of the three joints in both forward and reverse directions.

Our tuned values for three joints:

Viscous_positive[3] = {0.17, 0.23, 0.17} for joint 1, 2, 3.

Viscous_negative[3] = {0.2, 0.25, 0.2} for joint 1, 2, 3.

Coulombs_positive[3] = {0.36, 0.4759, 0.5339} for joint 1, 2, 3.

Coulombs_negative[3] = {-0.2948, -0.5031, -0.5190} for joint 1, 2, 3.

slope_between_minimums[3] = {3.6, 3.6, 3.6} for joint 1, 2, 3.

Note that the linear relationship between joint velocity and frictional force is still a rough approximation. There does exist inaccuracy for this approximation method since we noticed that even our tuned friction compensation becomes somewhat inaccurate at extreme (both high and low) speeds.

Part 2: Implement the Inverse Dynamics Control Law on Joints Two and Three.

2.1 Inverse Dynamics

In this part, we implemented inverse dynamics control for joint 2 and 3, but kept joint 1 the same feed forward control as in lab 2. The inner loop control is given by the following equation:

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

Where a_θ values are given by the outer loop control equations as follows:

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

The basic flow of our code each sample period is given as follows:

1) Calculate the desired trajectory

2) Given measured thetas, calculate actual states, error, error_dot, theta_dot.

3) Calculate the outer loop control to come up values for $a\theta 2$ and $a\theta 3$.

4) Calculate the inner loop control to find control effort to apply to joint 2 and 3.

5) Calculate Lab 2 feed forward control for joint 1 to find control effort to apply.

6) Calculate friction compensation control effort given the velocities of joint 1, 2 and 3.

7) Add the friction compensation to the control efforts calculated in 3 and 4 above.

8) Write control efforts to PWM outputs to drive each joint.

Actual implementation is given in **Appendix A**.

In our own words, the inverse dynamic control algorithm uses the knowledge of the system dynamic equations to calculate appropriate torques to better control the joint angles. The dynamic equations we obtained in lab 2, provided us the linear equation we need to calculate the appropriate torques. We only need to plug in the theta and derivatives into the equation to calculate the final torque. For our robot arm, the joint angle θ could be obtained directly from the motors and the first derivative of the joint angles $\dot{\theta}$ could be approximately calculated using the joint angle change and filtering at an acceptable precision level. However, the second derivative of the joint angles $\ddot{\theta}$ need to be obtained and using already biased $\dot{\theta}$ to calculate them may not the best practice and that's why we have this inner loop that use the desired joint angle and its derivatives to calculate the $\ddot{\theta}$ in a better precision. To sum up the inverse dynamics algorithm, we obtained the torque calculation equations first from the dynamic model of the robot arm. At each time instance, we obtain the actual joint angles and calculate the first derivative of the filtered joint angles. With these information, we are able to compare to the desired trajectory and calculate the errors between desired and actual values. Then, we use the inner loop to calculate the second derivative of joint angles. Finally, we plug in all the values we have into the inverse dynamic equations to calculate the final output torque.

2.2 Tuning Kp Kd gains to minimize error

The objective is to tune Kp and Kd terms for joint 2 and 3 such that the steady state error is less that 0.01.

Basically for each joint, we tune Kp and Kd terms by holding 1 term fixed and change the other one. We look at the error plot and find the value for minimum error. The final tuned values for Kp and Kd as well as error plots are given at the end of section 2.2.

2.3 Creating a faster moving trajectory

In order to better tune our Kp and Kd gains, we created a trajectory that moves faster. Specifically, it follows a cubic path for 0.33 seconds to 0.75 radians. Then the joints stay at 0.75 radians until 4 seconds have elapsed. From 4 seconds to 4.33 seconds it follows a cubic trajectory back to 0.25. The motion repeats every 8 seconds so that there is a pause between each cubic path to the new joint angle.

From the conditions given above, we established the following equations:

$$\theta^d(0) = 0 \quad ; \quad \dot{\theta}^d(0) = 0$$
$$\theta^d(0.33) = 0.75 \quad ; \quad \dot{\theta}^d(0.33) = 0$$
$$\theta^d(4) = 0.75 \quad ; \quad \dot{\theta}^d(4) = 0$$
$$\theta^d(4.33) = 0 \quad ; \quad \dot{\theta}^d(4.33) = 0$$

Furthermore, we assume that our trajectory is in quadratic forms given as

$$\theta_1{}^d(t) = a_1 t^3 + b_1 t^2 + c_1 t + d_1; \quad 0 \le t \le 0.33$$

$$\dot{\theta_1}{}^d(t) = 3a_1 t^2 + 2b_1 t + c_1; \quad 0 \le t \le 0.33$$

$$\theta_2{}^d(t) = a_2 t^3 + b_2 t^2 + c_2 t + d_2; \quad 4 \le t \le 4.33$$

$$\dot{\theta_2}{}^d(t) = 3a_2 t^2 + 2b_2 t + c_2; \quad 4 \le t \le 4.33$$

We have the following matrix equation after plugging in the conditions in the previous page:

$$\begin{bmatrix} \theta_1{}^d(0) \\ \dot{\theta_1}{}^d(0) \\ \theta_1{}^d(0.33) \\ \dot{\theta_1}{}^d(0.33) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0.75 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \end{bmatrix}$$

$$\text{take A} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix}$$

$$\text{we have } \begin{bmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \end{bmatrix} = A^{-1} \begin{bmatrix} 0 \\ 0 \\ 0.75 \\ 0 \end{bmatrix} = \begin{bmatrix} -27.8265 \\ 13.7741 \\ 0 \\ 0.25 \end{bmatrix}$$

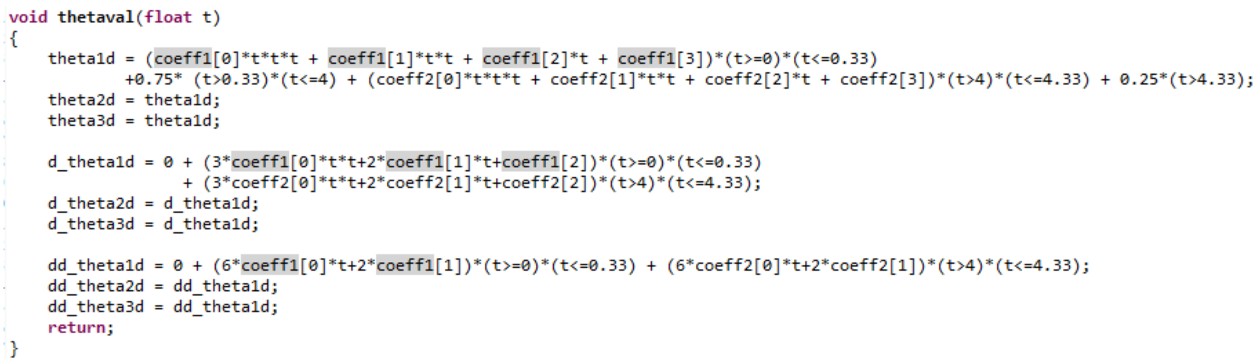
Therefore, ${\theta_1}^d(t) = -27.8265t^3 + 13.7741t^2 + 0.25; 0 \leq t \leq 0.33$

Similarly, the final expression for trajectory from 4 to 4.33s is as follows:

$${\theta_1}^d(t) = 27.83t^3 - 347.691t^2 + 1445.86t - 2000.53; 4 \leq t \leq 4.33$$

For $0.33 < t < 4$, the trajectory ${\theta_1}^d(t) = 0.75$ constant. We also specify the trajectory for joint 2 and joint 3 is the same with joint 1.

Our trajectory implementation in Code Composer is shown in **Figure 2**.



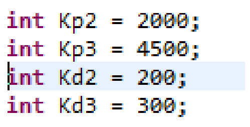
```
void thetaval(float t)
{
    theta1d = (coeff1[0]*t*t*t + coeff1[1]*t*t + coeff1[2]*t + coeff1[3])*(t>=0)*(t<=0.33)
            +0.75* (t>0.33)*(t<=4) + (coeff2[0]*t*t*t + coeff2[1]*t*t + coeff2[2]*t + coeff2[3])*(t>4)*(t<=4.33) + 0.25*(t>4.33);
    theta2d = theta1d;
    theta3d = theta1d;

    d_theta1d = 0 + (3*coeff1[0]*t*t+2*coeff1[1]*t+coeff1[2])*(t>=0)*(t<=0.33)
                  + (3*coeff2[0]*t*t+2*coeff2[1]*t+coeff2[2])*(t>4)*(t<=4.33);
    d_theta2d = d_theta1d;
    d_theta3d = d_theta1d;

    dd_theta1d = 0 + (6*coeff1[0]*t+2*coeff1[1])*(t>=0)*(t<=0.33) + (6*coeff2[0]*t+2*coeff2[1])*(t>4)*(t<=4.33);
    dd_theta2d = dd_theta1d;
    dd_theta3d = dd_theta1d;
    return;
}
```

Figure 2. Trajectory Implementation

The final tuned values for Kd and Kp terms are given as follows:



```
int Kp2 = 2000;
int Kp3 = 4500;
int Kd2 = 200;
int Kd3 = 300;
```

Figure 3. Tuned Gain Parameters

The error plot for three joints (we used PD control from lab 2 for joint 1) are given in **Figure 4** below.

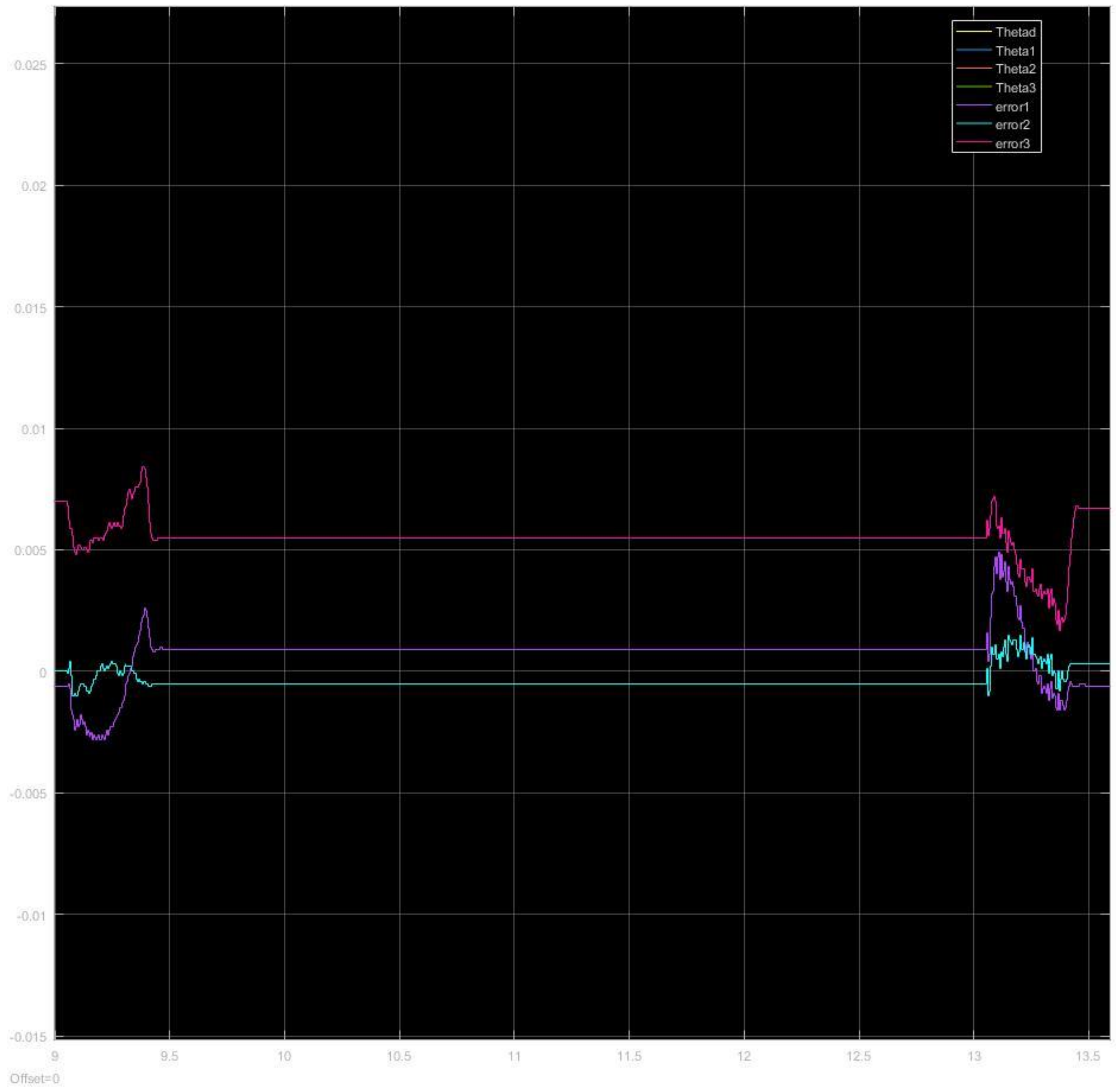


Figure 4. Joint Angle Error

Note that the error plot is implemented directly in Simulink.

According to our error plot, the steady state error for both joint 2 and 3 are way less than 0.01, so our objective was achieved.

Part 3: Compare Inverse Dynamics Controller to PD Controller.

3.1 Compare PD plus feedforward to Inverse Dynamics

For this part, we have both the PD plus Feedforward and the Inverse Dynamics controllers running. We obtained the results for each of the controller applying to the robot arm both with and without mass. When the robot arm is carrying the mass, we need to adjust the values in the dynamic equations to accommodate to this change.

Robot Arm Without Mass

Inverse Dynamics Controller

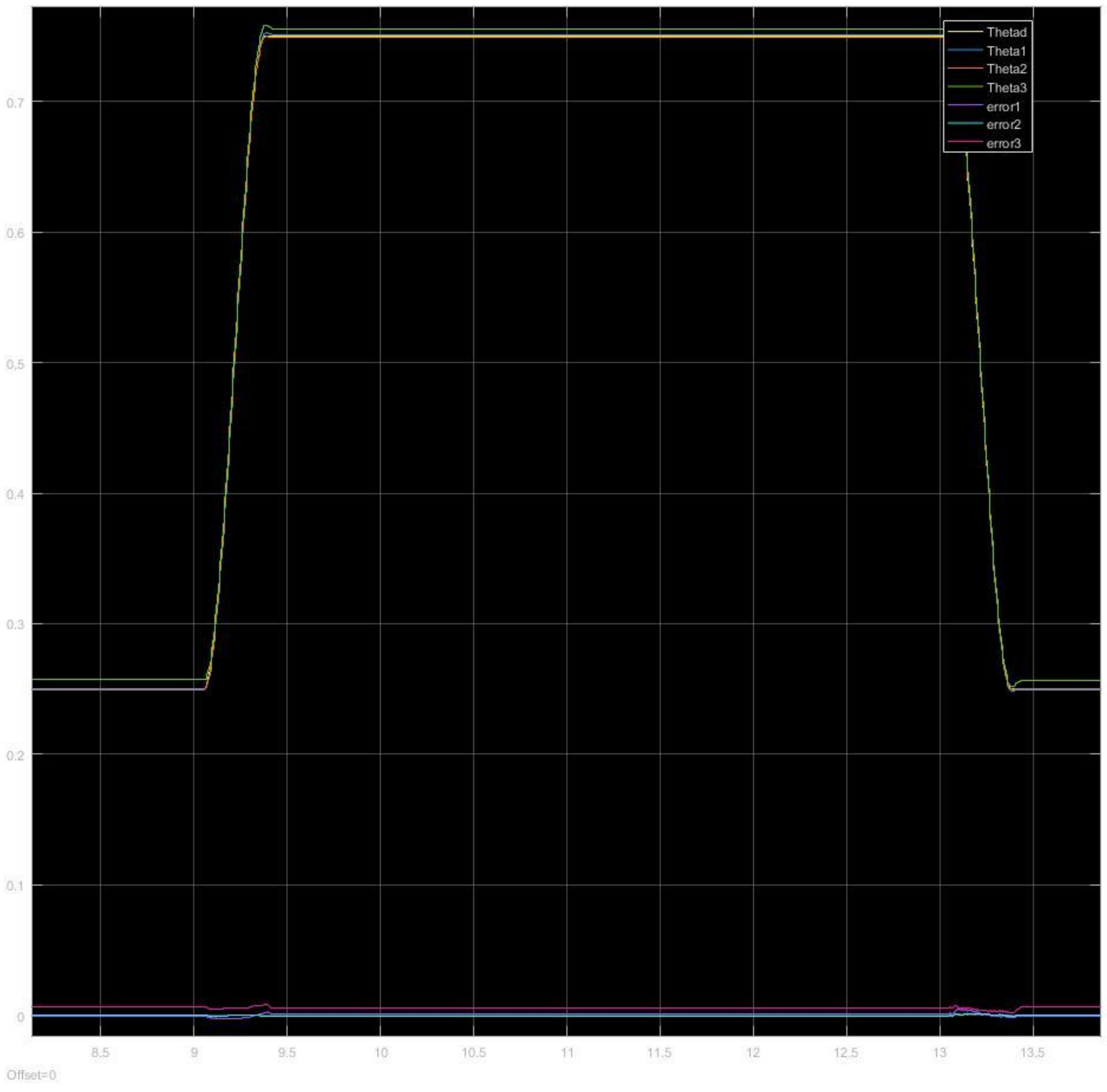


Figure 5. Theta under Inverse Dynamics Controller

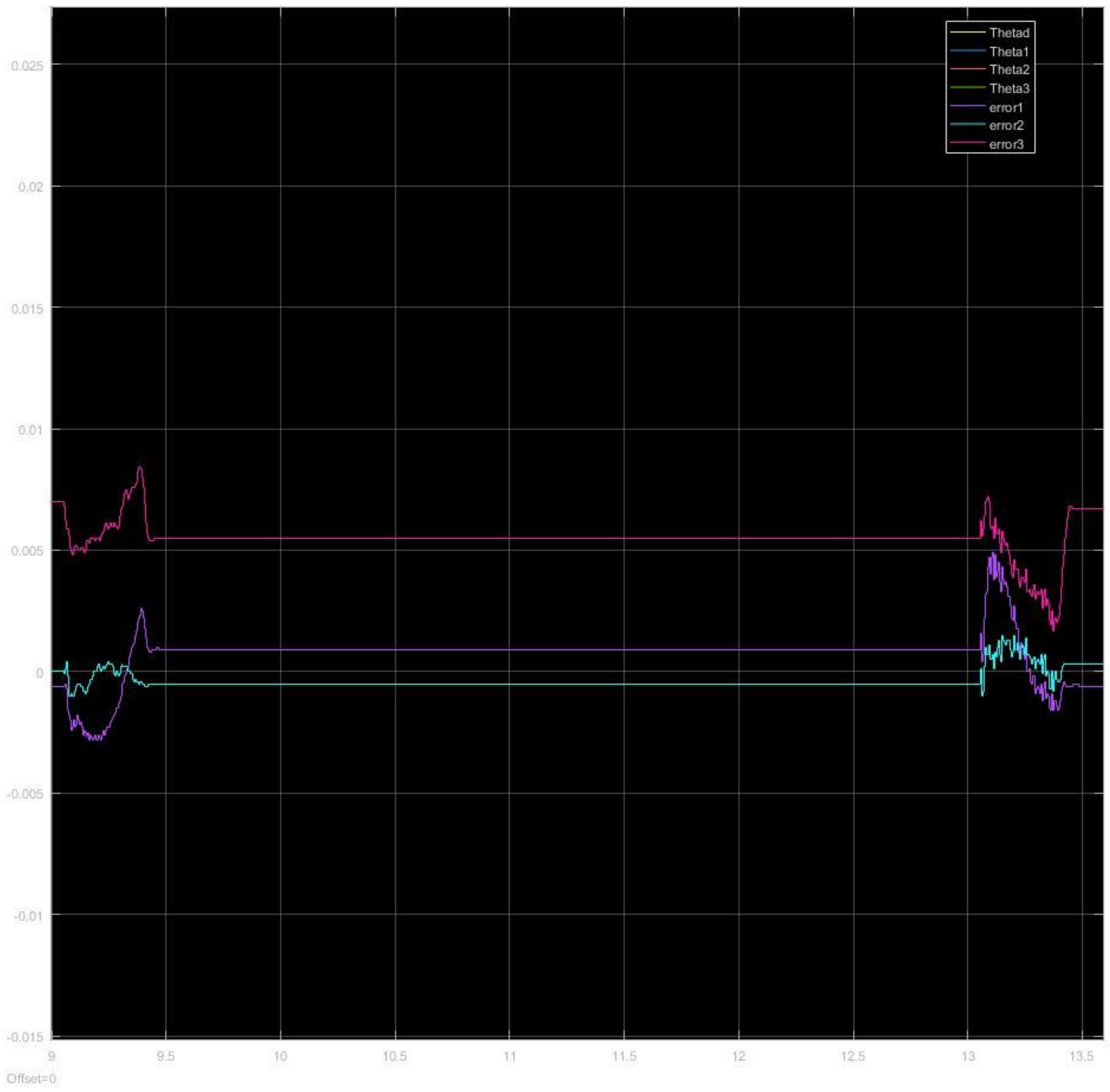


Figure 6. Theta Error under Inverse Dynamics Controller

All the errors are contained within the range of 0.01.

PD + Feedforward Controller

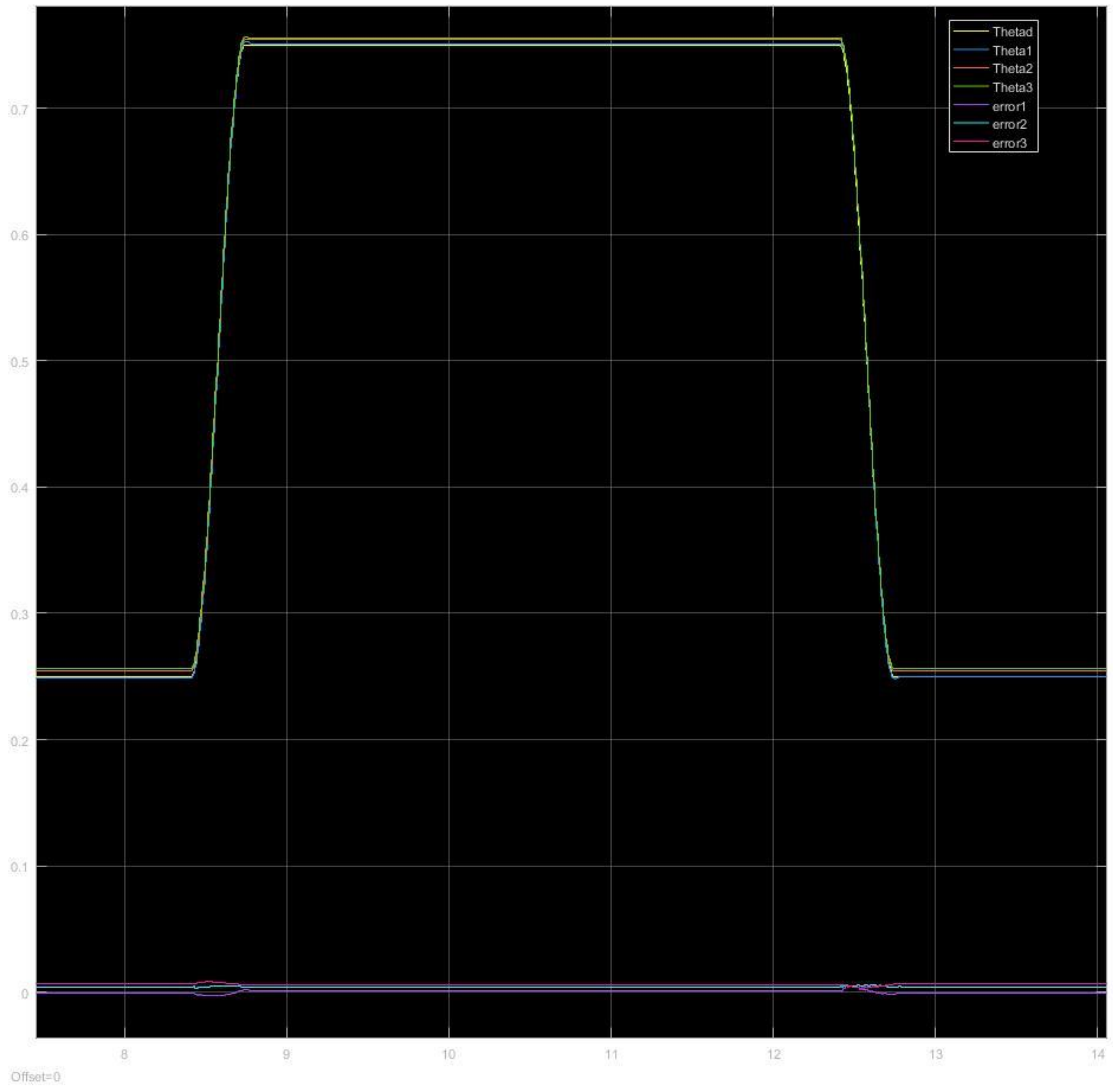


Figure 7. Theta under PD + Feedforward Controller

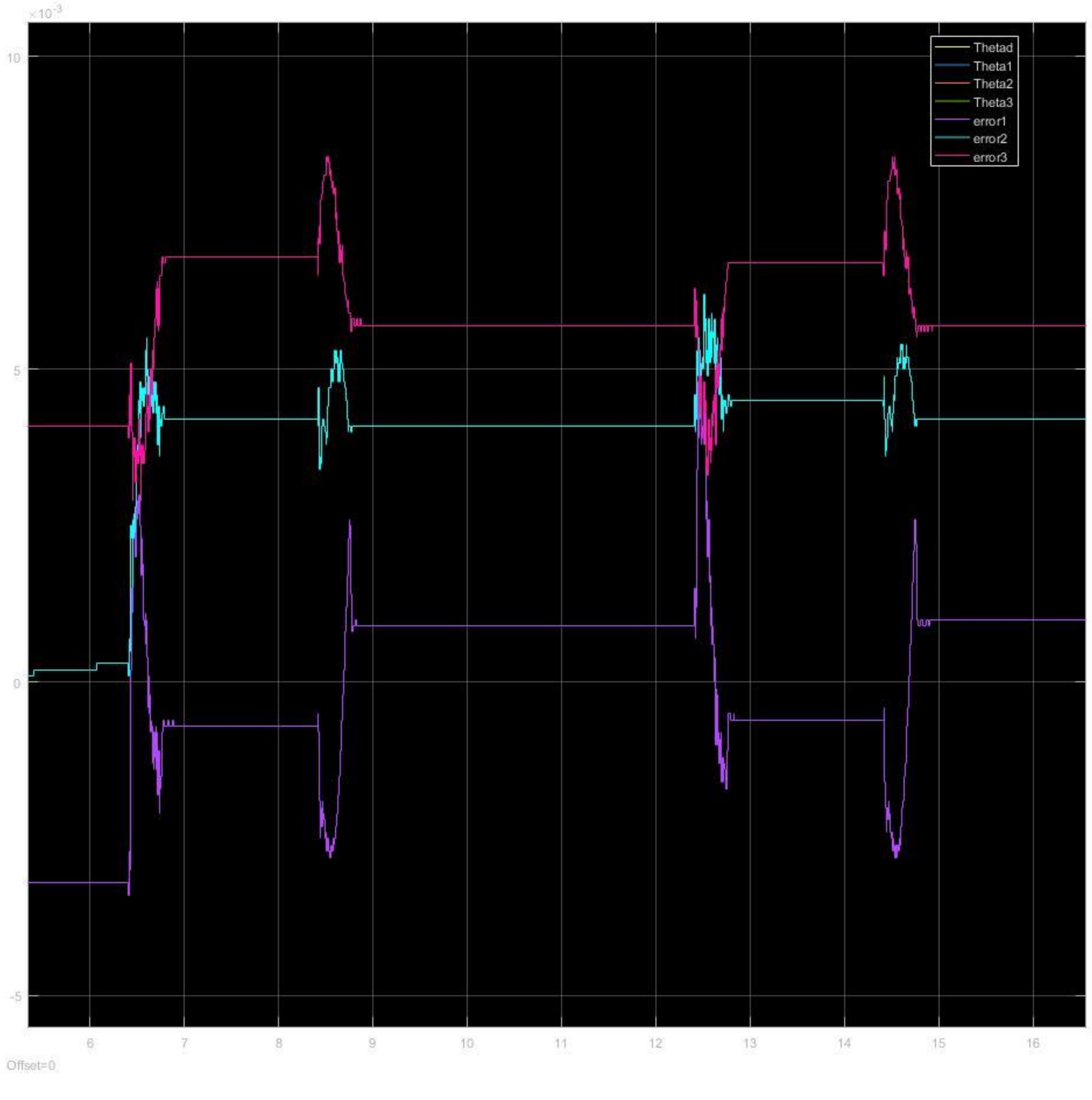


Figure 8. Error under PD + Feedforward Controller

All the errors are contained within the range of 0.01.

Robot Arm With Mass

Inverse Dynamics Controller

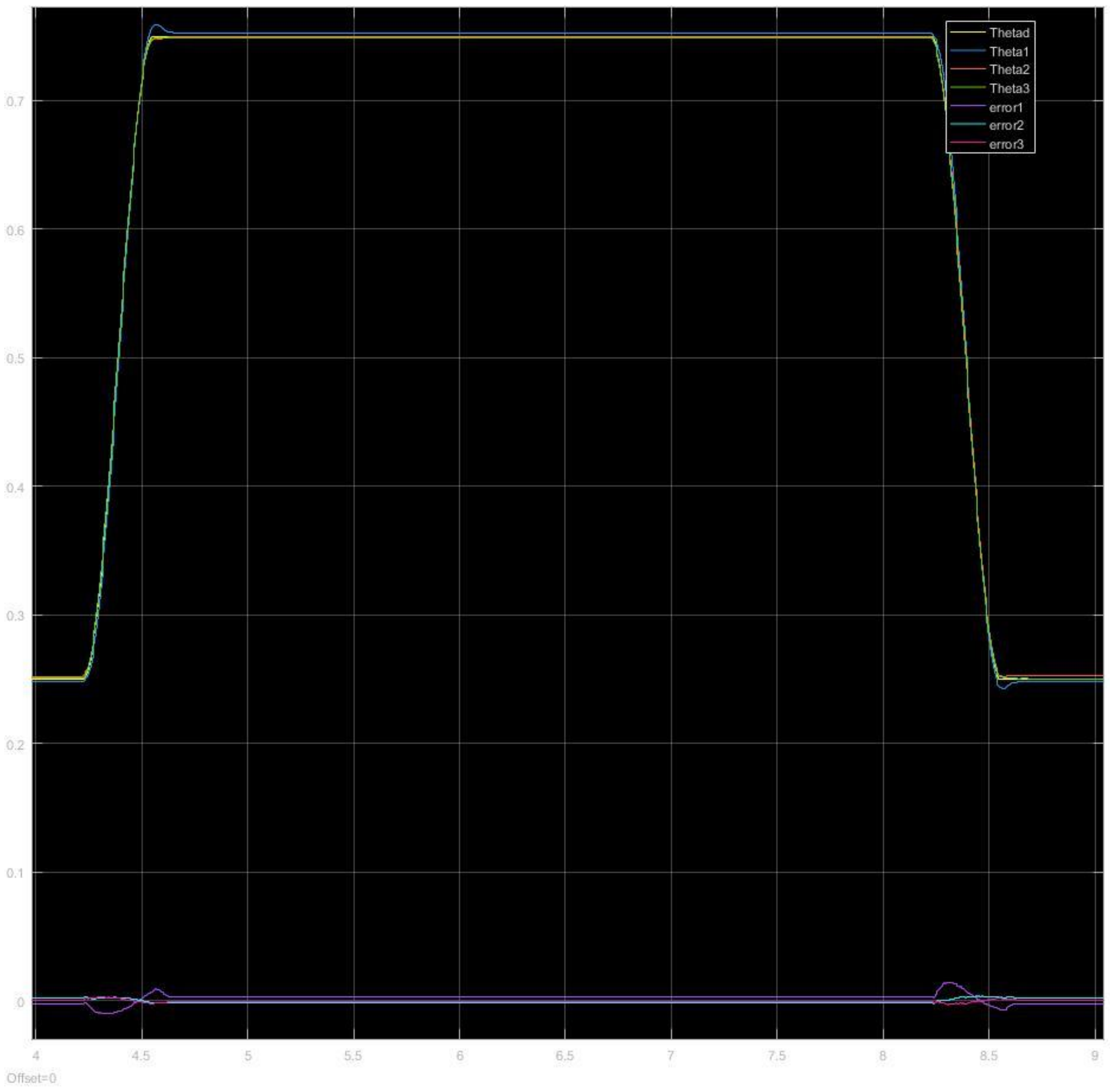


Figure 9. Theta under Inverse Dynamics Controller

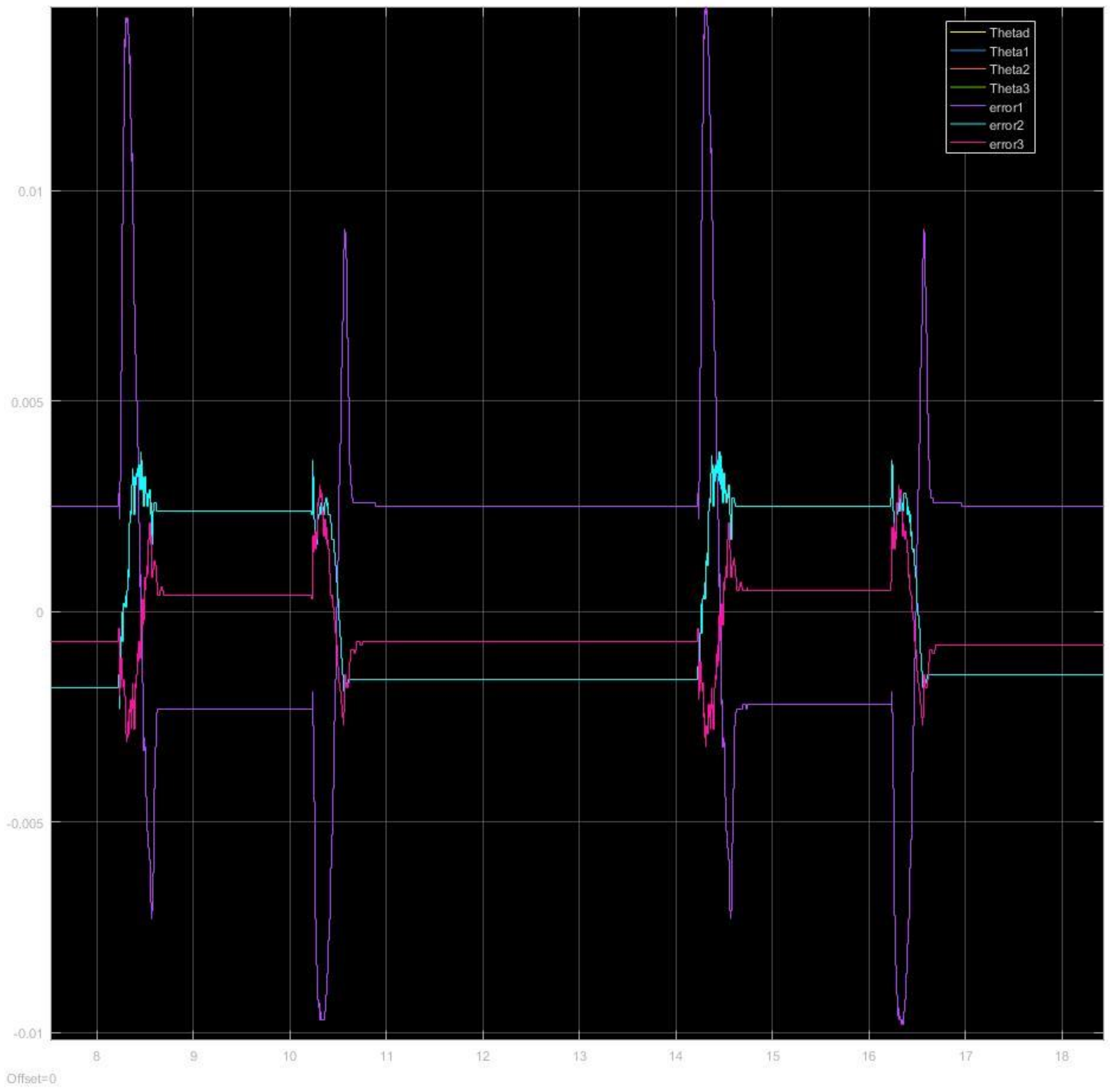


Figure 10. Error under Inverse Dynamics Controller

The largest error has reached to approximately 0.015.

PD + Feedforward Controller

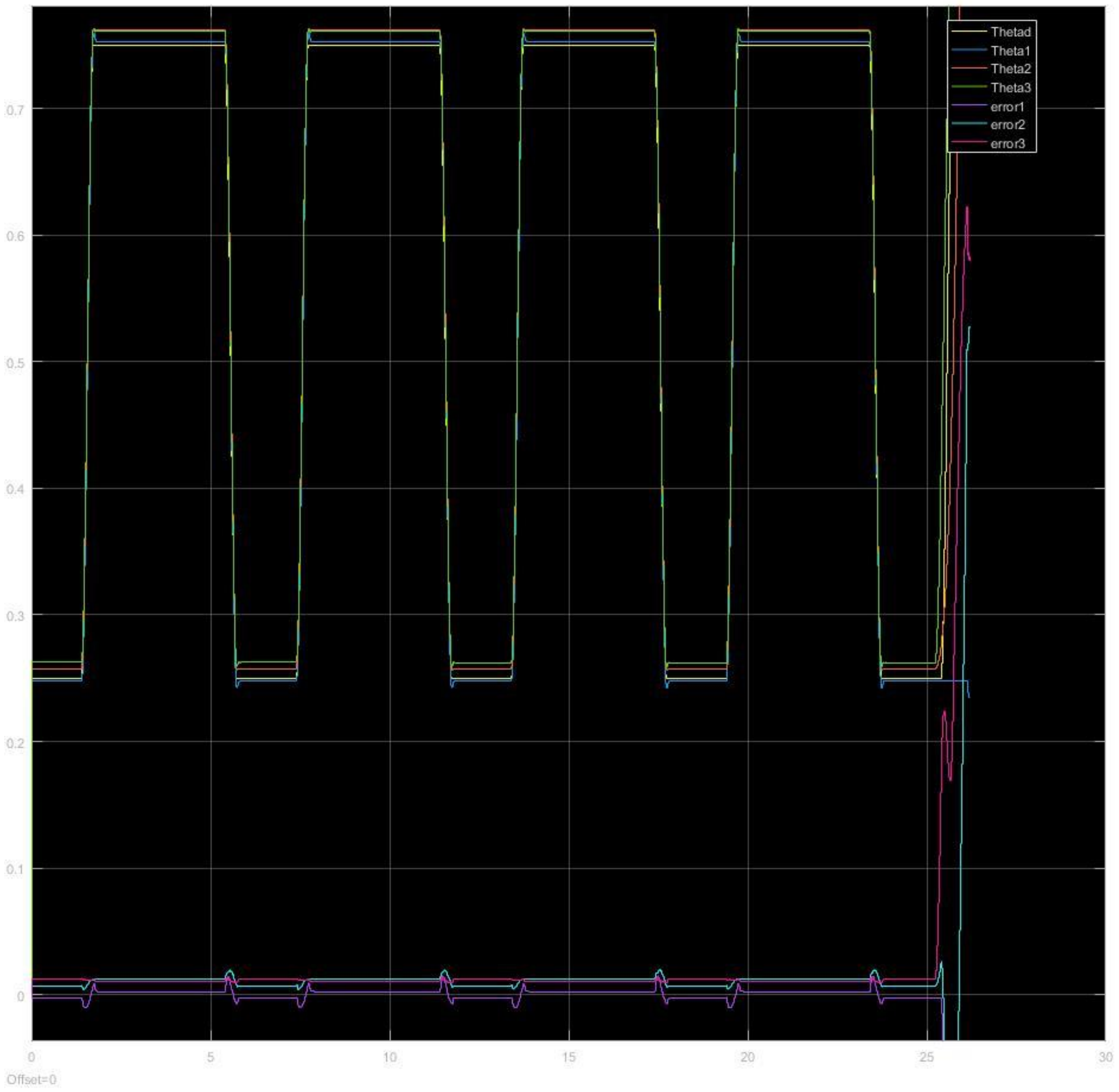


Figure 11. Theta Under PD + Feedforward Controller

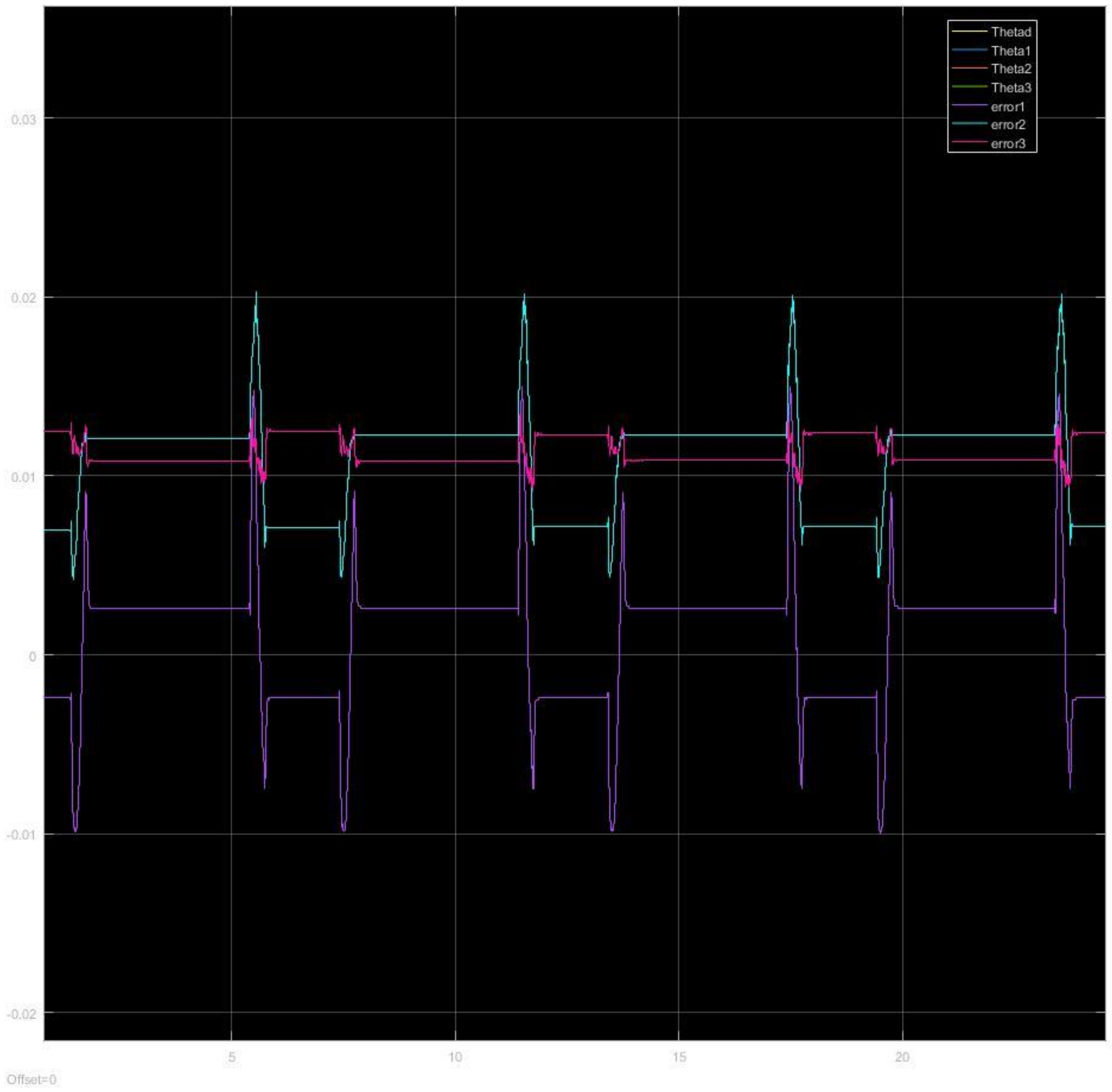


Figure 12. Error under PD + Feedforward Controller

The largest error has reached to approximately 0.021.

3.2 Comparison and Analysis

To compare the results obtained before, we could use the error measurement of theta1, since it uses the same control method all the time.

Robot Arm without Mass

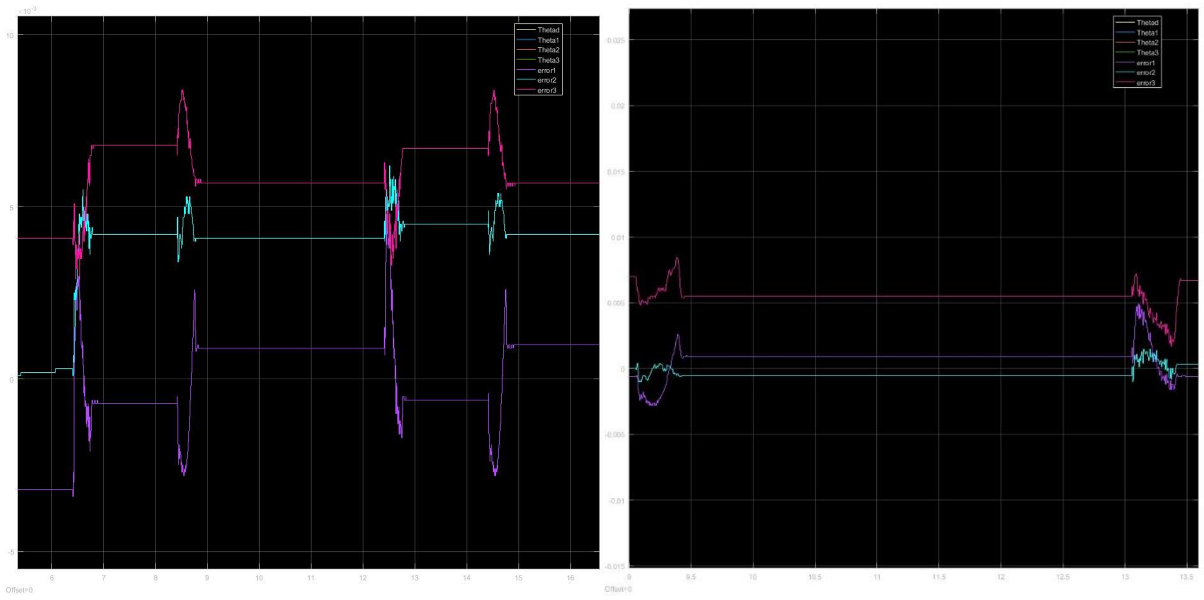


Figure 13. PD + Feedforward vs. Inverse Dynamics Control

The purple line in both sides of **Figure 13** is the error for theta 1. We could notice that the bright blue line that represent error of theta 2 is larger in left side and much smaller in right side; the pink line that represent error of theta 3 is almost the same in both left and right side of the figure. In this scenario, the inverse dynamics controller is slightly better than the PD + Feedforward controller, for the error on joint 2 is better than the latter one while the error on joint 3 is almost the same between the two. But all errors are relatively small, so two controllers have a good performance here.

Robot Arm with Mass

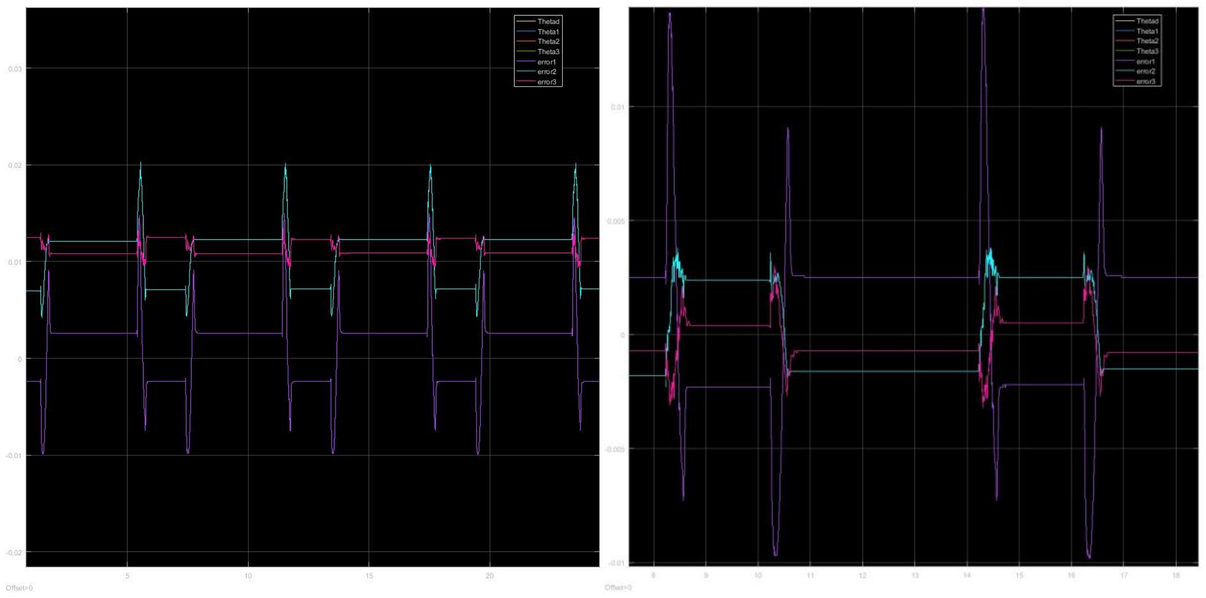


Figure 14. PD + Feedforward vs. Inverse Dynamics Control

Similarly, the purple line in both sides of **Figure 14** is the error for theta 1. In the left side for the PD + Feedforward Controller, the bright blue line that represent error of theta 2 has a steady state value around 0.013 and peak value around 0.021; while the pink line that represent error of theta 3 has a steady state value around 0.011 and peak value around 0.014. In the right side for the Inverse Dynamics Controller, the bright blue line that represent error of theta 2 has a steady state value around 0.003 and peak value around 0.004; while the pink line that represent error of theta 3 has a steady state value around 0.001 and peak value around 0.004. In this scenario, the Inverse Dynamics Controller is significantly better than the PD + Feedforward Controller. For joint 2 and joint 3, the error remains at the same level as if there are no mass. It is pretty clear that, without utilizing the dynamic equations of the robot arm, the simple PD + Feedforward Controller is having enlarged errors due to larger gravity applied to the system because of the extra mass added.

3.3 ID_CRS_withDisk.m

This file helps us to obtain the new parameters for our inverse dynamics equation. From what I observed, the file takes into consideration of the mass/disk, by approximating the mass to a regular plate disk with a certain diameter and weight. It took advantage of the linearity of the Inertia Tensor and uses the standard equations for a uniform plate. This are generally good assumptions, since the disk is almost a perfect uniform plate. By adding this mass to the end of the third link, the file could calculate the new vectors that should be used for the Inverse Dynamics equations.

3.4 Answer to the Questions

Can you say anything about the parameters we are using?

The parameters are quite accurate for the inverse dynamics, since we are able to get the errors to a relatively small level.

Does Inverse Dynamics perform better?

Yes, the Inverse Dynamics performs better than PD + Feedforward as we analyzed before. Clearly, it better compensates the gravity effect, by comparing the result obtained from the robot arm with mass.

Can any of the parameters be adjusted to improve the Inverse Dynamics control?

Absolutely, since we are guessing the parameter values for some of the Inverse Dynamics equations, we could always improve them by obtaining a more accurate model. Also for the parameters that needs to be tuned, we could find better parameters by tuning them systematically instead of just guessing around.

Appendix A. Implementation in Code Composer

```
#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 = -0.4238;
float offset_Enc3_rad = 0.2571;


// Your global varialbes.

long mycount = 0;

#pragma DATA_SECTION(whattoprint, ".my_vars") ///visible by matlab
float whattoprint = 0.0;

#pragma DATA_SECTION(theta1array, ".my_arrs")   //visible by matlab
float theta1array[100];

#pragma DATA_SECTION(theta2array, ".my_arrs")   //visible by matlab
float theta2array[100];
////////////////////// Lab 3 stuff
float Viscous_positive[3] = {0.17, 0.23, 0.17};

float Viscous_negative[3] = {0.2, 0.25, 0.2};

float Coulombs_positive[3] = {0.36, 0.4759, 0.5339};

float Coulombs_negative[3] = {-0.2948, -0.5031, -0.5190};

float slope_between_minimums[3] = {3.6, 3.6, 3.6};
//////////////////

float u_fric[3] = {0,0,0};

long arrayindex = 0;

float printtheta1motor = 0;
float printtheta2motor = 0;
float printtheta3motor = 0;
float printtheta1dh = 0;
float printtheta2dh = 0;
float printtheta3dh = 0;
float printendx = 0;
float printendy = 0;
float printendz = 0;
float printtheta1inv = 0;
float printtheta2inv = 0;
float printtheta3inv = 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;
float Theta1_old = 0;
float Omega1_raw = 0;
float Omega1_old1 = 0;
float Omega1_old2 = 0;
float Omega1 = 0;

float Theta2_old = 0;
float Omega2_raw = 0;
float Omega2_old1 = 0;
float Omega2_old2 = 0;
float Omega2 = 0;

float Theta3_old = 0;
float Omega3_raw = 0;
float Omega3_old1 = 0;
float Omega3_old2 = 0;
float Omega3 = 0;

float Ik1 = 0;
float Ik2= 0;
float Ik3 = 0;
float delt_IK1;
float delt_IK2;
float delt_IK3;


//////////D C g matrices
float D[2][2];
float C[2][2];
float g[2];
float gravity = 9.81;
float error2, error3;
float derror2, derror3;
//float dderror2, dderror3;

////////////////p array
//float p[5] = {0.03, 0.0128, 0.0076, 0.0753, 0.0298};        //p values
without disk
float p[5] = {0.0466, 0.0388, 0.0284, 0.1405, 0.1298};        //p values
with disk
///////////////////////////////// Change Kp Kd values here
int Kp2 = 2000;
int Kp3 = 4500;
/////////////////////////////////
int Kd2 = 200;
int Kd3 = 300;
/////////////////////////////////

int Kp1_pd = 90;
int Kp2_pd = 80;
int Kp3_pd = 85;

int Kd1_pd = 8;
int Kd2_pd = 8;
int Kd3_pd = 8;


int Ki1 = 1;
int Ki2 = 1;
int Ki3 = 1;
float J1 = 0.0167;
float J2 = 0.03;
float J3 = 0.0128;

float theta1d = 0;
float theta2d = 0;
float theta3d = 0;

float d_theta1d = 0;
float d_theta2d = 0;
float d_theta3d = 0;
float dd_theta1d = 0;
float dd_theta2d = 0;
float dd_theta3d = 0;

float a2 = 0;
float a3 = 0;


float coeff1 [4] = {-27.8265, 13.7741, 0, 0.25};
float coeff2 [4] = {2.782647410750677e+01, -3.476917939732969e+02,
1.445863594626050e+03, -2.000530017811884e+03};

void thetaval(float t)
{
    theta1d = (coeff1[0]*t*t*t + coeff1[1]*t*t + coeff1[2]*t +
coeff1[3])*(t>=0)*(t<=0.33)
            +0.75* (t>0.33)*(t<=4) + (coeff2[0]*t*t*t + coeff2[1]*t*t +
coeff2[2]*t + coeff2[3])*(t>4)*(t<=4.33) + 0.25*(t>4.33);
    theta2d = theta1d;
    theta3d = theta1d;

    d_theta1d = 0 +
(3*coeff1[0]*t*t+2*coeff1[1]*t+coeff1[2])*(t>=0)*(t<=0.33)
                 + (3*coeff2[0]*t*t+2*coeff2[1]*t+coeff2[2])*(t>4)*(t<=4.33);
    d_theta2d = d_theta1d;
    d_theta3d = d_theta1d;

    dd_theta1d = 0 + (6*coeff1[0]*t+2*coeff1[1])*(t>=0)*(t<=0.33) +
(6*coeff2[0]*t+2*coeff2[1])*(t>4)*(t<=4.33);
    dd_theta2d = dd_theta1d;
    dd_theta3d = dd_theta1d;
    return;
}
// This function is called every 1 ms
void lab(float theta1motor,float theta2motor,float theta3motor,float
*tau1,float *tau2,float *tau3, int error) {


    *tau1 = 0;
    *tau2 = 0;
    *tau3 = 0;

    //Motor torque limitation(Max: 5 Min: -5)
    // DH Frame Angle
    float theta1dh = theta1motor;
    float theta2dh = theta2motor - PI/2;
    float theta3dh = theta3motor-theta2motor+PI/2;

    // Forward Kinematics
    float endx = 25.4*cos(theta1dh)*(cos(theta2dh) +
cos(theta2dh+theta3dh));
    float endy = 25.4*sin(theta1dh)*(cos(theta2dh) +
cos(theta2dh+theta3dh));
    float endz = 25.4*(1-sin(theta2dh)-sin(theta2dh+theta3dh));

    // Forward Kinematics
    float theta1inv = atan2(endy,endx);
    float armlength = sqrt(endx*endx+endy*endy+(endz-25.4)*(endz-25.4));
    float theta2inv = - atan2(endz-25.4,sqrt(endx*endx+endy*endy)) -
acos(armlength*armlength/2/25.4/armlength);
    float theta3inv = 2 * acos(armlength*armlength/2/25.4/armlength);

    // save past states
    if ((mycount%50)==0) {

        theta1array[arrayindex] = theta1motor;
        theta2array[arrayindex] = theta2motor;

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

    }


    //////////////////////////////////////////////// Filtering of Theta
    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;
    ////////////////////////////////////////////////////////////////////

    float Omega[3] = {Omega1, Omega2, Omega3};
    float min_vel[3] = {0.1, 0.05, 0.05};
    int i;
    for(i = 0; i<3; ++i)
    {
        float joint_velocity = Omega[i];
        float minimum_velocity = min_vel[i];       //!!!!!!!!!!!!!!!!!!
Change min velocity HERE

        if (joint_velocity > minimum_velocity) {
            u_fric[i] = Viscous_positive[i]*joint_velocity +
Coulombs_positive[i] ;
            } else if (joint_velocity < -minimum_velocity) {
            u_fric[i] = Viscous_negative[i]*joint_velocity +
Coulombs_negative[i];
            } else {
            u_fric[i] = slope_between_minimums[i]*joint_velocity;
        }
    }

    thetaval( mycount%6000 / 1000.0);

    ////////////// PART II  //////////////////////

    a2 = dd_theta2d + Kp2*(theta2d-theta2motor) + Kd2*(d_theta2d-Omega2);
    a3 = dd_theta3d + Kp3*(theta3d-theta3motor) + Kd3*(d_theta3d-Omega3);

    //calculating D, C, g matrices

    D[0][0] = p[0];
    D[0][1] = -p[2]*sin(theta3motor-theta2motor);
    D[1][0] = D[0][1];
    D[1][1] = p[1];
    C[0][0] =0;
    C[0][1] = -p[2]*cos(theta3motor-theta2motor)*Omega3;
    C[1][0] = p[2]*cos(theta3motor-theta2motor)*Omega2;
    C[1][1] = 0;

    g[0] = -p[3]*gravity*sin(theta2motor);
    g[1] = -p[4]*gravity*cos(theta3motor);


    *tau1 = J1*dd_theta1d + Kp1_pd*(theta1d-theta1motor) +
Kd1_pd*(d_theta1d-Omega1) ;
    //*tau2 = J2*dd_theta2d + Kp2_pd*(theta2d-theta2motor) +
Kd2_pd*(d_theta2d-Omega2) ;
    //*tau3 = J3*dd_theta3d + Kp3_pd*(theta3d-theta3motor) +
Kd3_pd*(d_theta3d-Omega3) ;
    *tau2 = D[0][0]*a2+D[0][1]*a3+C[0][0]*Omega2+C[0][1]*Omega3+g[0];
    *tau3 = D[1][0]*a2+D[1][1]*a3+C[1][0]*Omega2+C[1][1]*Omega3+g[1];

    ///////////////////////////////

    // Friction Compensation
    *tau1 += u_fric[0];
    *tau2 += u_fric[1];
    *tau3 += u_fric[2];


    // calculating position error

    ////////////// Craps from lab 1
    if ((mycount%500)==0) {
        if (error != 0){
            serial_printf(&SerialA, "error position\n\r");
        }
        else{
        if (whattoprint > 0.5) {
            serial_printf(&SerialA, "I love robotics\n\r");
        } else {
            printtheta1motor = theta1motor;
            printtheta2motor = theta2motor;
            printtheta3motor = theta3motor;
            printtheta1dh = theta1dh;
            printtheta2dh = theta2dh;
            printtheta3dh = theta3dh;
```

```
          printendx = endx;
          printendy = endy;
          printendz = endz;
          printtheta1inv = theta1inv;
          printtheta2inv = theta2inv;
          printtheta3inv = theta3inv;
          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 = theta1d;
    Simulink_PlotVar2 = theta1motor;
    Simulink_PlotVar3 = theta2motor;
    Simulink_PlotVar4 = theta3motor;

    mycount++;
}

void printing(void){
    serial_printf(&SerialA, "Motor Angle: %.2f %.2f,%.2f
\n\r",printtheta1motor*180/PI,printtheta2motor*180/PI,printtheta3motor*180/
PI);
    //serial_printf(&SerialA, "Joint Angle: %.2f %.2f,%.2f
\n\r",printtheta1dh*180/PI,printtheta2dh*180/PI,printtheta3dh*180/PI);
    //serial_printf(&SerialA, "End Position: %.2f %.2f,%.2f
\n\r",printendx,printendy,printendz);
    //serial_printf(&SerialA, "Inverse Angle: %.2f %.2f,%.2f
\n\r",printtheta1inv*180/PI,printtheta2inv*180/PI,printtheta3inv*180/PI);
}
```