ME446 Lab 3: Inverse Dynamics Joint Control

Kevin Gim and Jasvir Virdi

April 19, 2019

Abstract

In this report, we implement an inverse dynamics based control algorithm and compare it's performance with a PD + feedforward controller for reference trajectory following of CRS robot arm. Friction of each joint has been considered by implementing friction compensation using a Static friction model. The trajectory tracking performance of the two controllers has been compared for two different trajectories. In addition, a different end-effector configuration with larger mass was used to explicitly observe the capability of the controllers. This work aims at understanding the Inverse Dynamics control law's ability to handle change in physical configuration using dynamics of the system.

1 Introduction

Figure 1: CAD of a CRS Robot Arm with an End-Effector(Left) and with a Weight(Right)

In previous labs, we have accomplished the kinematic and dynamic analysis, joint space PID position control, and task space PID position control using CRS Robot arm. Using the results from previous work, a more sophisticated approach, Inverse Dynamics Control, has been introduced for better position control performance.

Inverse Dynamics Control is a control technique for trajectory tracking that uses a nonlinear dynamic model of the robot arm by plugging in a second order linear input to the acceleration term. Since the Inverse Dynamics Control accounts for the dynamics of the robot arm, the control architecture provides better trajectory tracking performance as well as ability to manipulate with a heavier end-effector by adjusting the dynamic parameters.

For better joint torque control performance, friction compensation has been implemented to cancel the effect of friction of each joint. The joint friction force has been modeled using a modified static friction model. Instead of using the theoretical static friction model consists of a static Coulomb friction and velocity dependent Viscous friction, additional velocity dependent force is considered only for small velocity region in order to resolve discontinuity problem.

The trajectory tracking performance of the Inverse Dynamics Controller is compared with the PD + feedforward controller implemented in the previous lab. Moreover, tracking performance was tested with a different configuration of end-effector as shown in Figure 1. By installing the heavier end-effector that has much larger gravitational and inertial effect, we can more clearly differentiate trajectory tracking performance of the two controllers.

2 Friction Compensation

2.1 Friction Model

In order to reduce the effect of the friction of each joint, we implemented friction compensation. First, the friction model has been established. There are a number of methods to mathematically express friction force such as Static model, Dahl model, and Lugre model. Even though more complicated models gives more realistic model of friction force, it is difficult to find all parameters of the model from the experimental result. Hence, we approximate the friction force with the simplest Static friction model consists of Coulomb and Viscous friction force. Coulomb friction is the most basic friction in static case that is given by

$$F_c = \mu F_n sign(v)$$

where F_n is the normal force, μ is friction coefficient and v is the velocity of the object. The equation shows that the Coulomb friction is only proportional to the normal force independent to the magnitude of the velocity. There also is a velocity related term in static friction model, the Viscous friction. The Viscous friction is linear with respect to the velocity that is expressed as:

$$F_v(v) = \sigma_v v$$

The total friction force is calculated by summing the Coulomb and the Viscous friction forces. However, hardware implementation of the theoretical model is challenging since the Coulomb friction force need to be generated when the joint is not moving. Moreover, it is undesirable to have discontinuity in the friction force which may cause control problems. Therefore, we assume additional viscous friction with stiffer slope within a certain velocity threshold boundary. Since the velocity threshold values are set to ± 0.1, the additional viscous friction occurs in very small velocity in order to replicate the effect of the Coulomb friction. When the velocity exceeds the threshold, the fiction model follows the original Coulomb friction + Viscous friction model. Figure 2 presents the theoretical Static Friction model and modified Friction model.

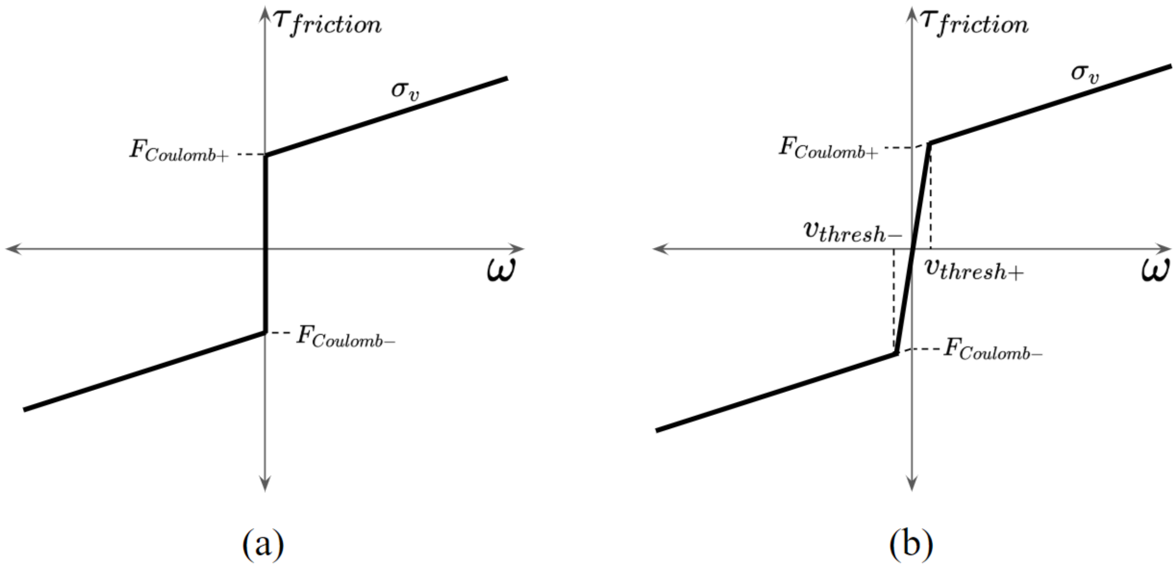

Figure 2: (a) Theoretical Static Friction Model (b) Empirical Friction Model

2.2 Hardware Implementation

The friction coefficients are given in the lab manual, nonetheless it is required to adjust the values for the specific robot we use. There are five parameters for each joint, positive viscous slope, negative viscous slope, positive coulomb force, negative coulomb force, and slope between the threshold. Note that coulomb force is a constant value as shown in Figure 2, (b). Using the given parameters, we have tuned the parameters through several iterations until each joint becomes "frictionless". Since there is a gap between our model and reality, the friction force on each joint cannot be eliminated thoroughly. Moreover, increasing the slope between the threshold makes the robot unstable in static case since it is dependent to the estimated velocity which has discrete value. The Friction force profile of the each joints are presented in Figure 3. The C code implementation of the friction compensation is included in main source code attached in Appendix.

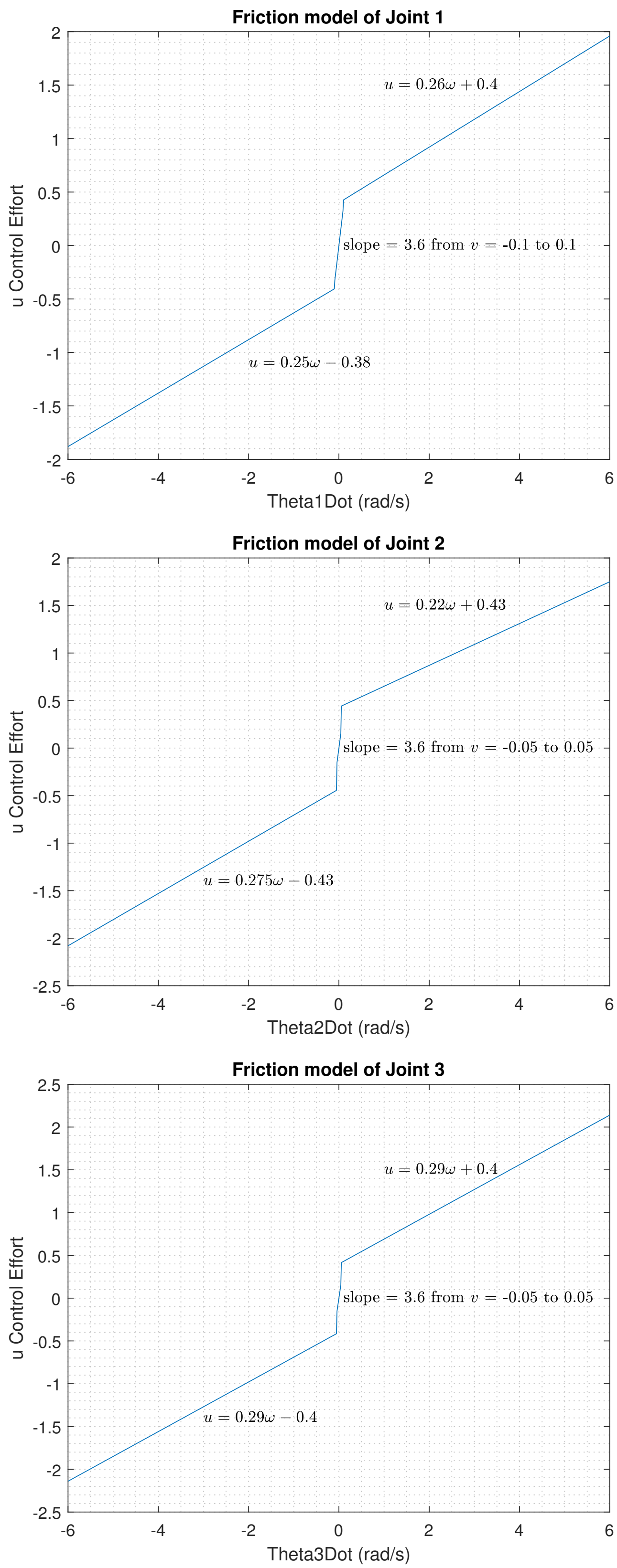

Figure 3: Friction model of each Joint with adjusted Parameters

3 Inverse Dynamics Controller

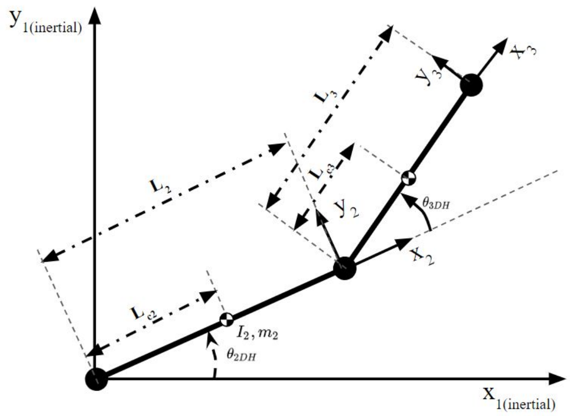

Figure 4: Diagram of a Two Link Planar Arm

In order to implement, a controller based on Inverse Dynamics, we need to know the dynamics behind our model to good accuracy. In lab 2, we already found the equations pertaining to motion of links 2 and 3 and they were then rearranged in a matrix form as follows:

$$D(\theta)\ddot{\theta} + C(\theta, \dot{\theta})\dot{\theta} + G(\theta) + friction = \tau$$

where

$$\theta = \begin{bmatrix} \theta_{2M} \\ \theta_{3M} \end{bmatrix}, \theta = \begin{bmatrix} \tau_{2M} \\ \tau_{3M} \end{bmatrix}, D(\theta) = \begin{bmatrix} p_1 & -p_3 \sin(\theta_{3M} - \theta_{2M}) \\ -p_3 \sin(\theta_{3M} - \theta_{2M}) & p_2 \end{bmatrix}$$

$$C(\theta, \dot{\theta}) = \begin{bmatrix} 0 & -p_3 \cos(\theta_{3M} - \theta_{2M})\dot{\theta}_{3M} \\ p_3 \cos(\theta_{3M} - \theta_{2M})\dot{\theta}_{2M} & 0 \end{bmatrix}, G(\theta) = \begin{bmatrix} p_4 g \sin(\theta_{2M}) \\ p_5 g \cos(\theta_{3M}) \end{bmatrix}$$

The parameters $p_1, p_2 ... p_5$ are defined as follows:

$$\begin{aligned} p_1 &= m_2 l_{c2}^2 + m_3 l_2^2 + I_{2zz} \\ p_2 &= m_3 l_{c3}^2 + I_{3zz} \\ p_3 &= m_2 l_2 l_{c3} \\ p_4 &= m_2 l_{c2} + m_3 l_2 \\ p_5 &= m_3 l_{c3} \end{aligned}$$

Here, for link 1 we still use the normal PD control. We start out with the same parameter values that we used in Lab 2 and model these control algorithms for the same setup. We change these accordingly if any changes to the setup are made like for example, mass being added at the end effector location etc. Now, the torque inputs for motor 2 and motor 3 are defined according to the following control policy:

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

where a_θ is the inner loop control input. If we substitute this expression into the main equation and simplify, we get a linearized ordinary differential equation.

$$a_\theta = \ddot{\theta}$$

Now, we can easily design an outer control law which ensures θ tracks θ_{des}, i.e, it follows the desired trajectory. Hence, a_θ is given as:

$$a_\theta = \ddot{\theta_{des}} + K_p(\theta_{des} - \theta) + K_d(\dot{\theta_{des}} - \dot{\theta})$$

Since, $a_\theta = \ddot{\theta}$, we put this in the above equation and get:

$$0 = \ddot{\theta_{des}} - \ddot{\theta} + K_p(\theta_{des} - \theta) + K_d(\dot{\theta_{des}} - \dot{\theta})$$
$$0 = \ddot{e} + K_d(\dot{e}) + K_p(e) \;\; where \;\; e = \theta_{des} - \theta$$

Now, for any $K_d > 0$ and $K_p > 0$, we get a stabilised system which makes e converge to 0. Hence we get $\theta = \theta_{des}$. The bigger K_d and K_p are, the faster convergence we get. However, in order for all this to work, we need really good estimation of $D(\theta)$, $C(\theta, \dot{\theta})$, $G(\theta)$ and friction.

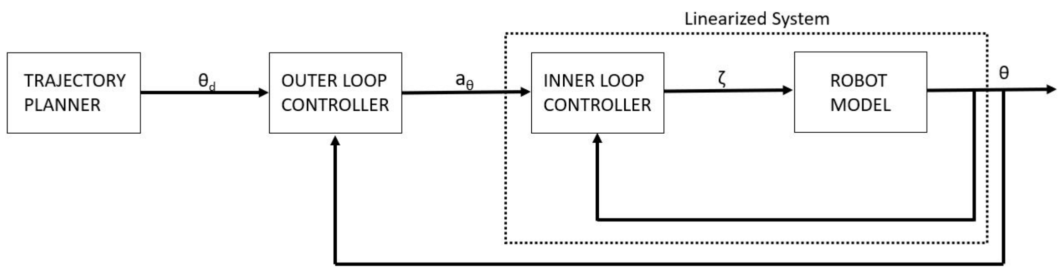

Figure 5: Inner/Outer Control Loop Architecture

The way this controller is developed can be easily seen through the inner/outer loop architecture diagram. The inner loop controller outputs torque which is directly fed into the robot model. This torque is based on the output of the outer loop controller a_θ, which in turn is dependent on the difference between desired and current motor angles.

In order to test the robustness of the controller, we make some modifications to the existing setup. We add a mass at the end effector position and our parameters namely $p_1, p_2 ... p_5$ are changed in accordance to our new setup. The main advantage of this type of controller is that once we have set the values K_p and K_d for the base system, we do not need to make any changes to it no matter how the current setup is changed. We only need to make changes to the robot dynamic model and the inner loop controller. Further details regarding the results and it's comparison with a standard PD + feedforward control are presented in the following sections.

4 Hardware Implementation Result

4.1 Old Cubic Trajectory

The same trajectory generated in the previous lab is used for initial implementation of the Inverse Dynamics Controller. The trajectory travels $0rad$ to $0.5rad$ for $2sec$ period and is shown in the following figure. Since the trajectory is defined as a time dependent cubic polynomial function, it is possible to define $\dot{\theta}_{des}$ and $\ddot{\theta}_{des}$ by taking a time derivative of the reference trajectory equation. Using the trajectory, we compared tracking performance of PD+Feedforward Controller and Inverse Dynamics Controller. Note that the first joint is controlled with the same PD+Feedforward Controller since we didn't establish dynamic model for the first joint.

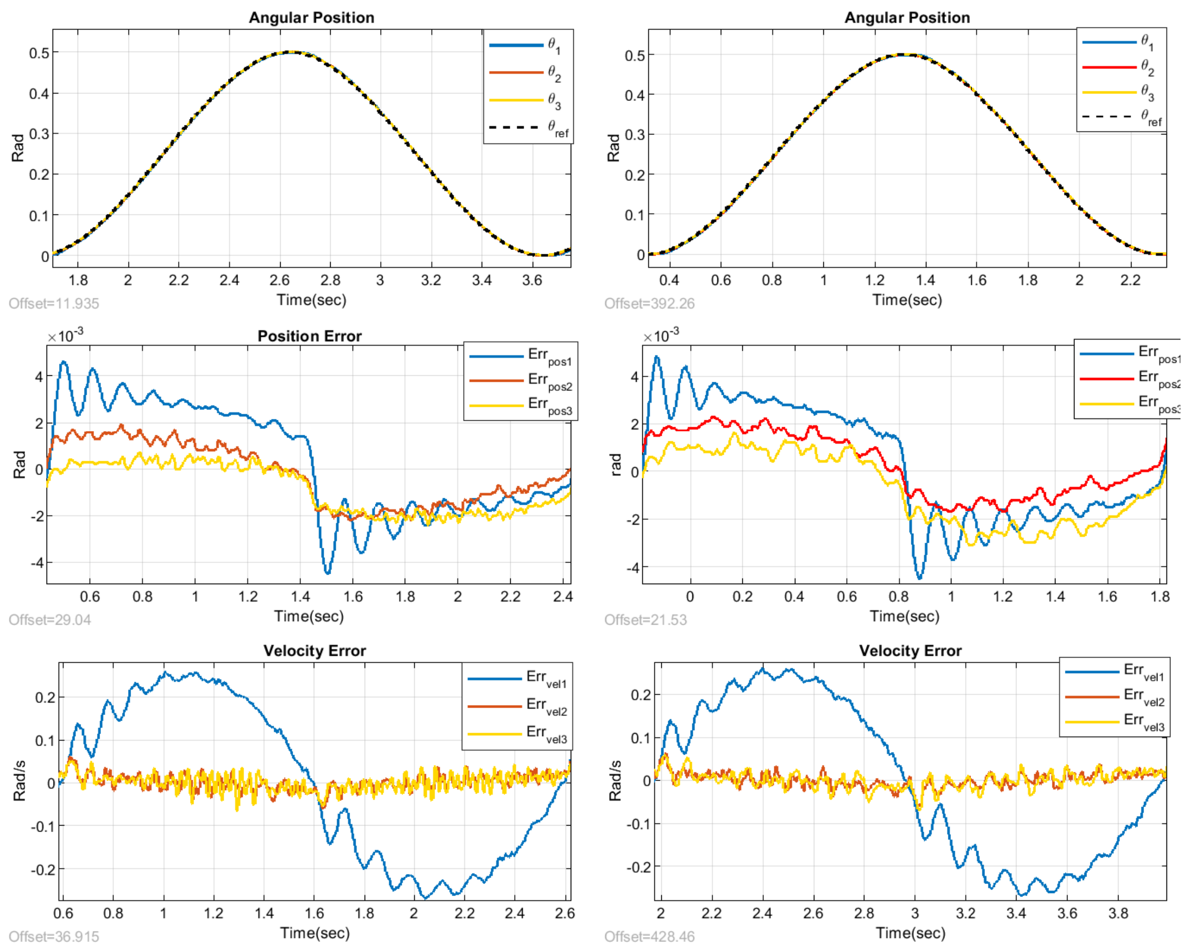

Figure 6: Trajectory Tracking performance of the PD+Feedforward Controller(Left) vs. Inverse Dynamics Controller(Right) for the old trajectory

Figure 6 presents the tracking result of the PD+feedforward and Inverse Dynamics Controller for following the old cubic trajectory. Plots in the first row are angular position of the three joints, second row are position error value and third row are velocity error value. we observed that there are no significant difference in tracking performance between the controllers. Both the position error and velocity error show very similar trends and magnitude as well. In both cases, the largest position error value is smaller than $0.005rad$, which equals to $0.2°$. Different from what was expected, both the methods show similar tracking performance, even though the Inverse Dynamics Controller is more sophisticated control method as it accounts the system dynamics. This is because the PD controller has been tuned well to minimize tracking error with a large gain so that it rejects tracking error very efficiently.

4.2 New Cubic Trajectory

Now, we design a new reference joint trajectory has more rapid movement to clearly observe the advantage of the Inverse Dynamics Controller. The trajectory starts with $0.25rad$ at $t = 0$ and increases to $0.75rad$ for $0.33sec$ following a cubic polynomial function. Then it stays for 4 seconds and go back to $0.25rad$ from $4sec$ to $4.33sec$. The trajectory stays at $0.25rad$ for another $4sec$ and repeats with period of $8sec$. The trajectory is shown in Figure 7 and the MATLAB script used to generate the trajectory shown below.

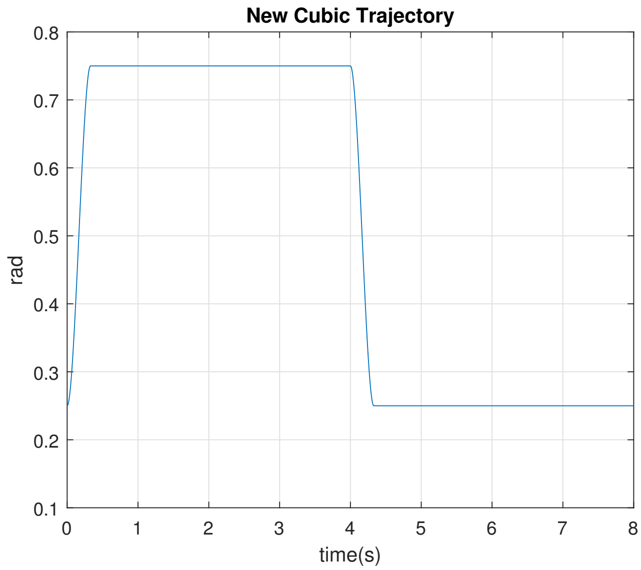

Figure 7: New Reference Cubic Trajectory

New Cubic Trajectory Generation MATLAB Script

```
t = linspace(0,8,1000);
for i = 1:length(t);

    if t(i)<0.330
        a = 27.8265;
        b = 13.7741;
        c = 0;
        d = 0.25;
        theta1_des(i) = a*t(i)^3 + b*t(i)^2 + c*t(i) + d;

    elseif 0.330< t(i) && t(i) <4
        d = 0.75;
        theta1_des(i) = d;

    elseif 4 < t(i) && t(i)< 4.330
        a = 27.8264741075056;
        b = 347.691793973283;
        c = 1445.86359462599;
        d = 2000.53001781181;
        theta1_des(i) = a*t(i)^3 + b*t(i)^2 + c*t(i) + d;

    else
          d = 0.25;
        theta1_des(i) = d;
    end
end

plot(t, theta1_des)
```

Since the new trajectory has faster velocity that is similar with a step function, it is more useful to compare the performance of the controllers. Figure 8 and 9 present the trajectory tracking result. Note that the Inverse Dynamics Controller is implemented without any adjustment yet control gains of the PD controller is tuned again for the new trajectory.

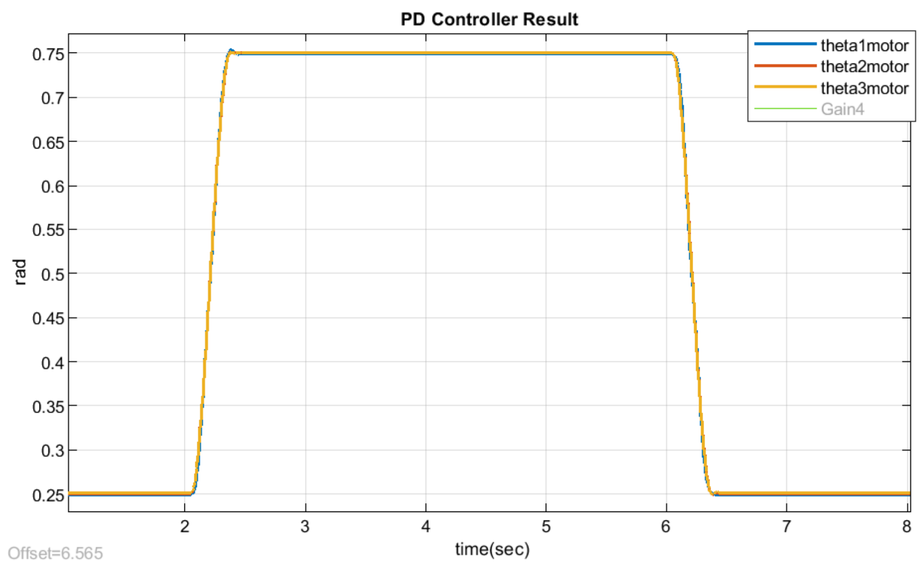

Figure 8: Trajectory Tracking Result with the PD Controller

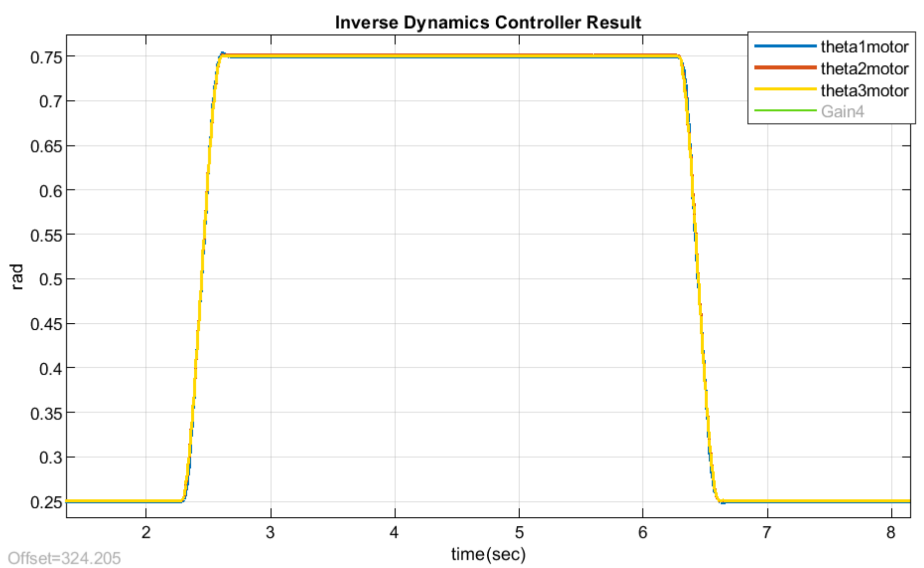

Figure 9: Trajectory Tracking Result with the Inverse Dynamics Controller

4.3 Heavier End-effector Configuration

In addition to introducing the new trajectory, a new end-effector configuration is tested with weight being addded as shown in the left figure of Figure 1. The parameters in the dynamics equation are adjusted based on the change in configuration including the feedforward parameters. The new parameters, p_i were found using attached MATLAB Script. In addition, the feedforward parameter J_3 was also recalculated for the new end-effector.

MABLAB Script for finding the new dynamic parameters

```
% guessing here due to unknown gear ratio
Imotor2 = 0.00375;
Imotor2 = Imotor2*20;
Imotor3 = 0.00375;
Imotor3 = Imotor3*10;  %guess

% center of mass of link one
lc2 = 0.125;
% Total moment of Inertia of Link one about its center of mass
Il2 = 0.0114;
ml2 = 1.3;
L2 = 10*.0254; %meters

% Total mass of link two
ml3 = 1.14;
% Center of mass of link two
lc3 = 0.157;
% Total moment of Inertia of Link Two about its center of mass
Il3 = 0.0111;

%Mass of disks
Mw = 1.54;
%Center of mass of disks
Lw = .32;
%Moment of Intertia of Disks
Iw = (1/12)*1.54*(3*(.0508)^2+(.0254)^2)

%Link 3 Center with disks attached
lc3T = (ml3*lc3 + Mw*Lw)/(ml3+Mw);

% generate the five parameters
pars(1,1) = (ml2*lc2^2 + (ml3+Mw)*L2^2 + Il2) + Imotor2;
pars(2,1) = ((ml3+Mw)*lc3T^2 + Il3 + ml3*(lc3T lc3)^2 + Iw + (Lw   lc3T)^2) + Imotor3;
pars(3,1) = ((ml3+Mw)*L2*lc3T);
pars(4,1) = (ml2*lc2 + (ml3+Mw)*L2);
pars(5,1) = (ml3*lc3T + Mw*Lw);  %Mike had (m3*lc3T + Mw*Lw)

% add the Torque Constant to the parameters
TorqueConst = 6.0; %N m/UnitIN
pars = pars/TorqueConst

%for FeedForward Control
J1 = 0.1/TorqueConst;
J2 = (Imotor2+ml2*lc2^2+ml3*L2^2+Il2)/TorqueConst;
J3 = (Imotor3+Il3+ml3*lc3T^2)/TorqueConst;
```

Note that the PD gains are kept at the same value to verify the advantage of the Inverse Dynamics Controller. Friction compensation is also added to the both controllers. The result of trajectory tracking is presented in following Figures.

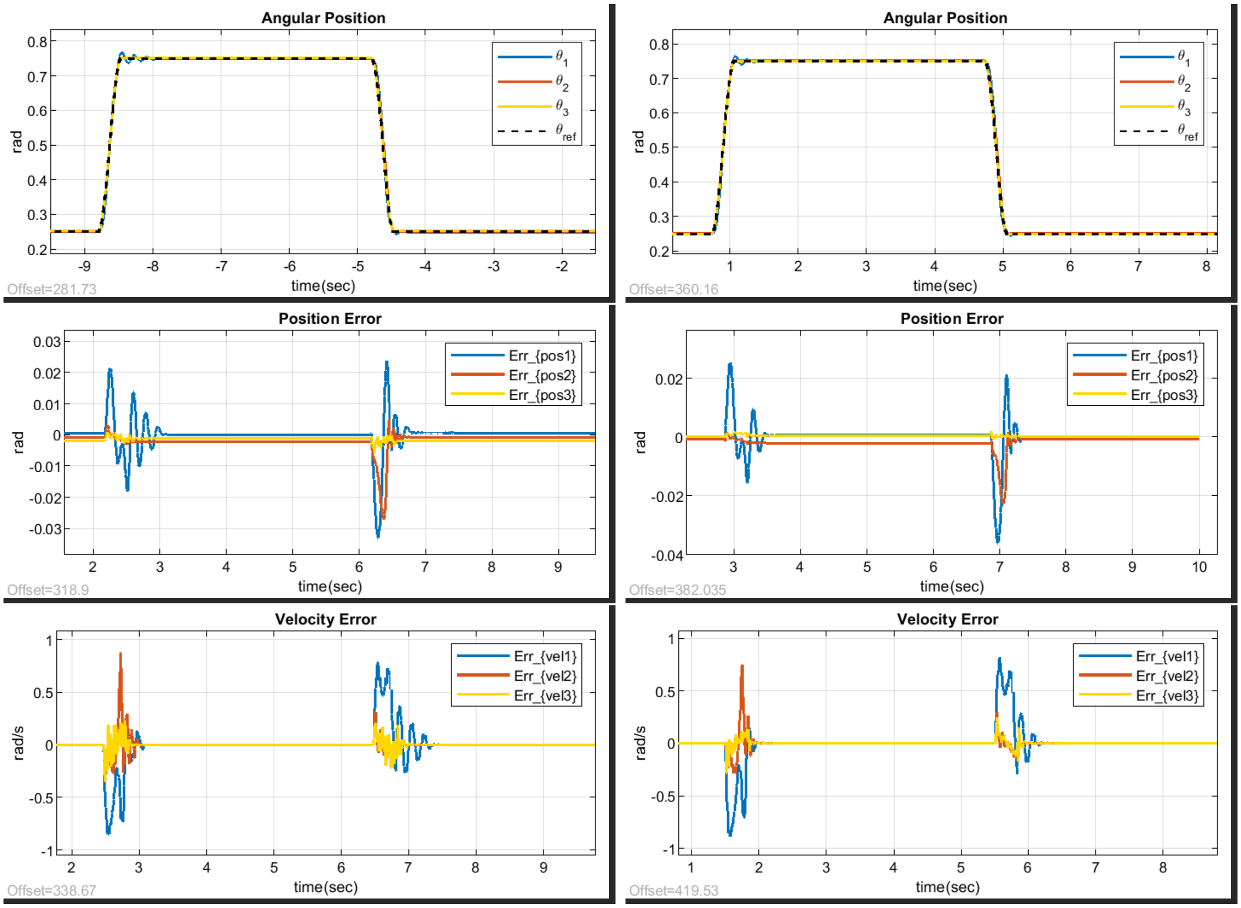

Figure 10: Trajectory Tracking performance of the PD+Feedforward Controller(Left) vs. Inverse Dynamics Controller(Right) for the new trajectory and the heavier end-effector

Similar with Figure 6, Figures in the first row are for angular position, second row is for position error and the third is for velocity error. The error plot shows that Inverse Dynamics Controller shows smaller peak error for θ_2 and θ_3. Steady state error for the joints is also smaller with the Inverse Dynamics Control as well. Moreover, it can be observed from the velocity error plot that the velocity errors are smaller for the Inverse Dynamics Controller.

Figure 11 and 12 gives a closer look with response analysis which includes rise time, overshoot etc. However, we observe that there is not a significant difference in either case. This may be due to the following reasons. First, the PD gains are large enough to successfully reject the system change. In addition, the adjusted feedforward parameters account for the system change that rejects the gravitational and inertial effect of the heavier end-effector.

Throughout the lab, we could verify that the Inverse Dynamics Controller has advantage that it doesn't requires laborious procedure of gain tuning even if there are changes in the physical system or trajectory. However, it is also verified that well-tuned PD+Feedforward controller is able to show similar performance with the Inverse Dynamics Controller.

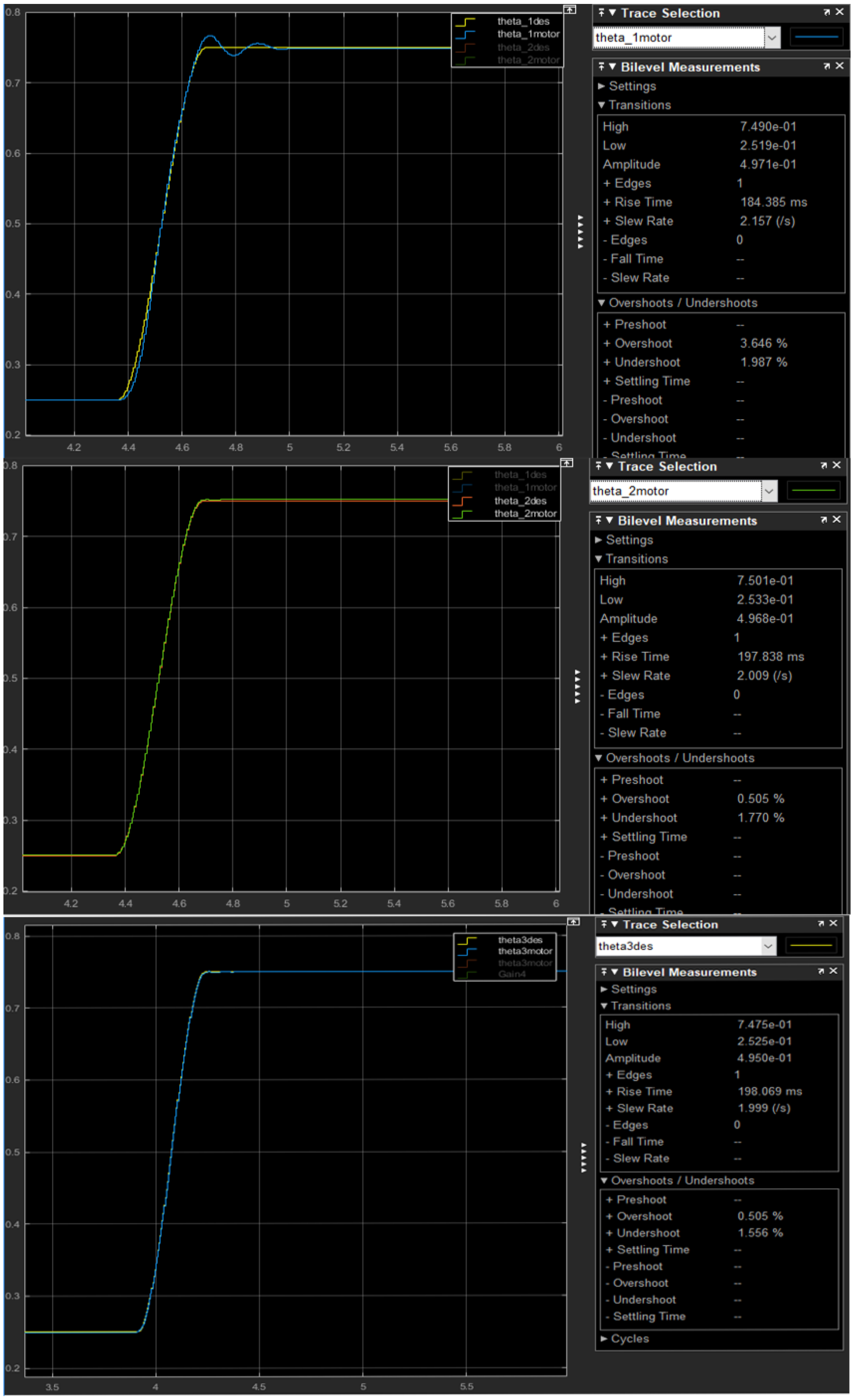

Figure 11: Trajectory Tracking Result with the Inverse Dynamics Controller

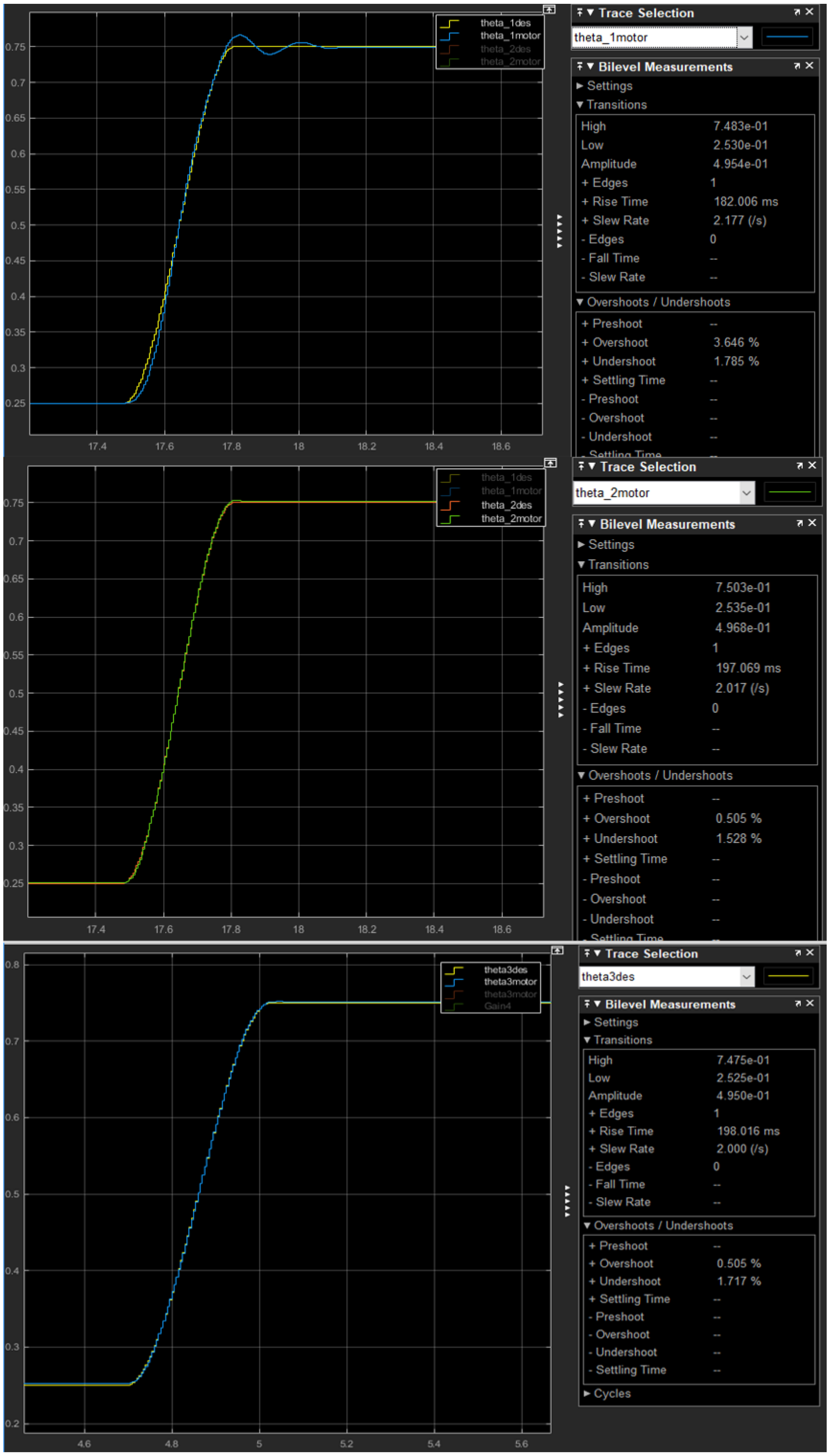

Figure 12: Trajectory Tracking Result with the Inverse Dynamics Controller

5 Appendix

Trajectory Tracking with Inverse Dynamics Controller C code

```
#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 = -0.36;
float offset_Enc3_rad = 0.27;

// Your global varialbes.
long mycount = 0;
#pragma DATA_SECTION(whattoprint, .my_vars)
float whattoprint = 0.0;
#pragma DATA_SECTION(theta1array, .my_arrs)
float theta1array[100];
long arrayindex = 0;
float printtheta1motor = 0;
float printtheta2motor = 0;
float printtheta3motor = 0;
float DHtheta1 = 0;
float DHtheta2 = 0;
float DHtheta3 = 0;
float x = 0;
float y = 0;
float z = 0;
float IKtheta1DH = 0;
float IKtheta2DH = 0;
float IKtheta3DH = 0;
float IKthetam1 = 0;
float IKthetam2 = 0;
float IKthetam3 = 0;
float r1 = 0;
float r2 = 0;
float l1 = 254;
float l2 = 254;
float l3 = 254;
// 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_old1 = 0;
float Omega1_old2 = 0;
float Omega1 = 0;

float Theta2_old = 0;
float Omega2_old1 = 0;
float Omega2_old2 = 0;
float Omega2 = 0;
float Theta3_old = 0;
```

```
float Omega3_old1 = 0;
float Omega3_old2 = 0;
float Omega3 = 0;
float theta1_des = 0;
float theta2_des = 0;
float theta3_des = 0;
float Omega1_des = 0;
float Omega2_des = 0;
float Omega3_des = 0;
float Omega1d_des = 0;
float Omega2d_des = 0;
float Omega3d_des = 0;
// PD gain with cubic reference signal
float KP_1 = 150;
float KP_2 = 500;
float KP_3 = 500;
float KD_1 = 0.7;
float KD_2 = 2.0;
float KD_3 = 1.3;
// PD gain for Inverse Dynamicswith cubic reference signal
float KP_2_inv = 10000;
float KP_3_inv = 15000;
float KD_2_inv = 80.0;
float KD_3_inv = 100.0;

float error_1 = 0.0;
float error_2 = 0.0;
float error_3 = 0.0;
float traperror_1 = 0.0;
float traperror_2 = 0.0;
float traperror_3 = 0.0;
float I_1 = 0.0;
float I_2 = 0.0;
float I_3 = 0.0;
float thresh_1 = 0.05;
float thresh_2 = 0.05;
float thresh_3 = 0.03;
float a = 0;
float b = 0;
float c = 0;
float d = 0;

float t = 0.0;
float x_f = 0.0;
float y_f = 0.0;
float z_f = 0.0;

// For friction compensation
// Provided coefficient
//float min_vel_1 = 0.1;
//float min_vel_2 = 0.05;
//float min_vel_3 = 0.05;
//
//float u_fric_1 = 0;
//float u_fric_2 = 0;
//float u_fric_3 = 0;
//float u_fric = 0;
```

```
//float vis_pos_1 = 0.2513;
//float vis_neg_1 = 0.2477;
//float vis_pos_2 = 0.2500;
//float vis_neg_2 = 0.2870;
//float vis_pos_3 = 0.1922;
//float vis_neg_3 = 0.2132;
//
//float cmb_pos_1 = 0.3637;
//float cmb_neg_1 = -.2948;
//float cmb_pos_2 = 0.4759;
//float cmb_neg_2 = -.5031;
//float cmb_pos_3 = 0.5339;
//float cmb_neg_3 = -.5190;
float u_fric_1 = 0;
float u_fric_2 = 0;
float u_fric_3 = 0;
float u_fric = 0;

float min_vel_1 = 0.09;
float vis_pos_1 = 0.26;
float vis_neg_1 = 0.25;
float cmb_pos_1 = 0.4;
float cmb_neg_1 = -.38;

float min_vel_2 = 0.049;
float vis_pos_2 = 0.22;
float vis_neg_2 = 0.275;
float cmb_pos_2 = 0.43;
float cmb_neg_2 = -.43;
float min_vel_3 = 0.049;
float vis_pos_3 = 0.29;
float vis_neg_3 = 0.29;
float cmb_pos_3 = 0.4;
float cmb_neg_3 = -.4;
float slope_1 = 3.6;
float slope_2 = 3.6;
float slope_3 = 3.6;
float p_1[5] = {0.0300, 0.0128, 0.0076, 0.0753, 0.0298};
float p_2[5] = {0.0466, 0.0388, 0.0284, 0.01405, 0.1298}; //with Weight
float J1 = 0.0167;
float J2 = 0.03;
float J3 = 0.0128;
float J3w = 0.02;
float a_m2 = 0;
float a_m3 = 0;
float g = 9.8;
int mode = 0;
int whattoplot = 0;
float fric_gain = 0.5;

float fric_comp(float Omega, float min_vel, float vis_pos, float cmb_pos, float
    vis_neg, float cmb_neg, float slope){
    if (Omega > min_vel) {
        u_fric = vis_pos*Omega + cmb_pos ;
    } else if (Omega < -min_vel) {
        u_fric = vis_neg*Omega + cmb_neg;
    } else {
```

```
        u_fric = slope*Omega;
    }
    return u_fric;
}


// This function is called every 1 ms
void lab(float theta1motor,float theta2motor,float theta3motor,float *tau1, float *tau2,float *tau3, int error) {

    % Defining sin and cos for saving DSP resource
    float sintheta2 = sin(theta2motor);
    float costheta2 = cos(theta2motor);
    float sintheta3 = sin(theta3motor);
    float costheta3 = cos(theta3motor);

    //Motor torque limitation(Max: 5 Min: -5)
    // save past states
    if ((mycount%50)==0) {
        theta1array[arrayindex] = theta1motor;
        if (arrayindex >= 100) {
            arrayindex = 0;
        } else {
            arrayindex++;
        }
    }

    if ((mycount%50)==0) {
        if (whattoprint > 0.5) {
            serial_printf(&SerialA, I love robotics);
        } else {
            printtheta1motor = theta1motor;
            printtheta2motor = theta2motor;
            printtheta3motor = theta3motor;
            DHtheta1 = theta1motor;
            DHtheta2 = theta2motor-PI*0.5;
            DHtheta3 = theta3motor-theta2motor+PI*0.5;
            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
    }

    //Forward Kinematics
    x_f = 508*cos(DHtheta1)*cos(DHtheta2+DHtheta3/2)*cos(DHtheta3/2);
    y_f = 508*sin(DHtheta1)*cos(DHtheta2+DHtheta3/2)*cos(DHtheta3/2);
    z_f = -254*(-1+sin(DHtheta2)+sin(DHtheta2+DHtheta3));

    //Inverse Kinematics
    IKtheta1DH = atan2(y,x);
    r1 = z-l1;
    r2 = sqrt(r1*r1 + x*x + y*y);
    IKtheta3DH = PI - acos((l2*l2+l3*l3-r2*r2)/(2*l2*l3));
    IKtheta2DH = -(IKtheta3DH)/2 - asin((r1/r2));
    IKthetam1= IKtheta1DH;
```

```
    IKthetam2=IKtheta2DH +(PI/2);
    IKthetam3=IKtheta3DH + IKtheta2DH;
    theta1_des = IKthetam1;
    theta2_des = IKthetam2;
    theta3_des = IKthetam3;

//     // Old Smooth motor trajectory (cubic)
//     t = (mycount%2000)/1000.;
//     float a_1 = -1;
//     float b_1 = 1.5;
//     float a_2 = 1;
//     float b_2 = -4.5;
//     float c_2 = 6;
//     float d_2 = -2;
//     if ((mycount%2000)<1000) {
//
//         theta1_des = a_1*t*t*t + b_1*t*t;
//         theta2_des = a_1*t*t*t + b_1*t*t;
//         theta3_des = a_1*t*t*t + b_1*t*t;
//
//         Omega1_des = 3*a_1*t*t + 2*b_1*t;
//         Omega2_des = 3*a_1*t*t + 2*b_1*t;
//         Omega3_des = 3*a_1*t*t + 2*b_1*t;
//
//         Omega1d_des = 6*a_1 + 2*b_1;
//         Omega2d_des = 6*a_1 + 2*b_1;
//         Omega3d_des = 6*a_1 + 2*b_1;
//
//     }
//     else {
//         theta1_des = a_2*t*t*t + b_2*t*t + c_2*t + d_2;
//         theta2_des = a_2*t*t*t + b_2*t*t + c_2*t + d_2;
//         theta3_des = a_2*t*t*t + b_2*t*t + c_2*t + d_2;
//
//         Omega1_des = 3*a_2*t*t + 2*b_2*t + c_2;
//         Omega2_des = 3*a_2*t*t + 2*b_2*t + c_2;
//         Omega3_des = 3*a_2*t*t + 2*b_2*t + c_2;
//
//         Omega1d_des = 6*a_2 + 2*b_2;
//         Omega2d_des = 6*a_2 + 2*b_2;
//         Omega3d_des = 6*a_2 + 2*b_2;
//     }
//
      // New Smooth motor trajectory (cubic) for Lab 3 (0.25, 0.75 rad)
        t = (mycount%8000)/1000.;
        if ((mycount%8000)<330) {
            a = -27.8265;
            b = 13.7741;
            c = 0;
            d = 0.25;
            theta1_des = a*t*t*t + b*t*t + c*t + d;
            theta2_des = a*t*t*t + b*t*t + c*t + d;
            theta3_des = a*t*t*t + b*t*t + c*t + d;
            Omega1_des = 3*a*t*t + 2*b*t + c;
            Omega2_des = 3*a*t*t + 2*b*t + c;
            Omega3_des = 3*a*t*t + 2*b*t + c;
            Omega1d_des = 6*a*t + 2*b;
```

```
            Omega2d_des = 6*a*t + 2*b;
            Omega3d_des = 6*a*t + 2*b;
            }
        else if ((mycount%8000)<4000 && 330<(mycount%8000)){
            a = 0;
            b = 0;
            c = 0;
            d = 0.75;
            theta1_des = a*t*t*t + b*t*t + c*t + d;
            theta2_des = a*t*t*t + b*t*t + c*t + d;
            theta3_des = a*t*t*t + b*t*t + c*t + d;
            Omega1_des = 3*a*t*t + 2*b*t + c;
            Omega2_des = 3*a*t*t + 2*b*t + c;
            Omega3_des = 3*a*t*t + 2*b*t + c;
            Omega1d_des = 6*a*t + 2*b;
            Omega2d_des = 6*a*t + 2*b;
            Omega3d_des = 6*a*t + 2*b;
        }
        else if (4000 < (mycount%8000) && (mycount%8000)<4330){
            a = 27.8264741075056;
            b = -347.691793973283;
            c = 1445.86359462599;
            d = -2000.53001781181;
            theta1_des = a*t*t*t + b*t*t + c*t + d;
            theta2_des = a*t*t*t + b*t*t + c*t + d;
            theta3_des = a*t*t*t + b*t*t + c*t + d;
            Omega1_des = 3*a*t*t + 2*b*t + c;
            Omega2_des = 3*a*t*t + 2*b*t + c;
            Omega3_des = 3*a*t*t + 2*b*t + c;
            Omega1d_des = 6*a*t + 2*b;
            Omega2d_des = 6*a*t + 2*b;
            Omega3d_des = 6*a*t + 2*b;
        }
        else{
              a = 0;
              b = 0;
              c = 0;
              d = 0.25;
              theta1_des = a*t*t*t + b*t*t + c*t + d;
              theta2_des = a*t*t*t + b*t*t + c*t + d;
              theta3_des = a*t*t*t + b*t*t + c*t + d;
              Omega1_des = 3*a*t*t + 2*b*t + c;
              Omega2_des = 3*a*t*t + 2*b*t + c;
              Omega3_des = 3*a*t*t + 2*b*t + c;
              Omega1d_des = 6*a*t + 2*b;
              Omega2d_des = 6*a*t + 2*b;
              Omega3d_des = 6*a*t + 2*b;
        }

    if (whattoplot ==0){
        Simulink_PlotVar1 = theta1motor;
        Simulink_PlotVar2 = theta2motor;
        Simulink_PlotVar3 = theta3motor;
        Simulink_PlotVar4 = theta1_des;
    }
    else if(whattoplot == 1){
        Simulink_PlotVar1 = error_1;
```

```
        Simulink_PlotVar2 = error_2;
        Simulink_PlotVar3 = error_3;
        Simulink_PlotVar4 = 0;
    }
    else if(whattoplot == 2){
           Simulink_PlotVar1 = Omega1_des - Omega1;
           Simulink_PlotVar2 = Omega2_des - Omega2;
           Simulink_PlotVar3 = Omega3_des - Omega3;
           Simulink_PlotVar4 = 0;
       }

    // Theta1 velocity
    Omega1 = (theta1motor - Theta1_old)/0.001;
    Omega1 = (Omega1 + Omega1_old1 + Omega1_old2)/3.0;
    Theta1_old = theta1motor;
    Omega1_old2 = Omega1_old1;
    Omega1_old1 = Omega1;
    Omega1 = (theta1motor - Theta1_old)/0.001;
    Omega1 = (Omega1 + Omega1_old1 + Omega1_old2)/3.0;
    // Theta2 velocity
    Omega2 = (theta2motor - Theta2_old)/0.001;
    Omega2 = (Omega2 + Omega2_old1 + Omega2_old2)/3.0;
    Theta2_old = theta2motor;
    Omega2_old2 = Omega2_old1;
    Omega2_old1 = Omega2;
    // Theta3 velocity
    Omega3 = (theta3motor - Theta3_old)/0.001;
    Omega3 = (Omega3 + Omega3_old1 + Omega3_old2)/3.0;
    Theta3_old = theta3motor;
    Omega3_old2 = Omega3_old1;
    Omega3_old1 = Omega3;


    error_1 = theta1_des - theta1motor;
    error_2 = theta2_des - theta2motor;
    error_3 = theta3_des - theta3motor;

    // Integral Control
    if (fabs(*tau1) > 5){
        traperror_1 = 0;}
    else{
        traperror_1 = error_1/2*0.001;
        I_1 += traperror_1;
    }
    if (fabs(*tau2) > 5){
        traperror_2 = 0;}
    else{
        traperror_2 = error_2/2*0.001;
        I_2 += traperror_2;
    }
    if (fabs(*tau3) > 5){
        traperror_3 = 0;}
    else{
        traperror_3 = error_3/2*0.001;
        I_3 += traperror_3;
    }

```

```
    if (fabs(error_1)>thresh_1){
        I_1 = 0;
    }
    if (fabs(error_2)>thresh_2){
        I_2 = 0;
    }
    if (fabs(error_3)>thresh_3){
        I_3 = 0;
    }

    // Part 1: Friction Compensation

    u_fric_1 = fric_comp(Omega1, min_vel_1, vis_pos_1, cmb_pos_1, vis_neg_1, cmb_neg_1, slope_1);
    u_fric_2 = fric_comp(Omega2, min_vel_2, vis_pos_2, cmb_pos_2, vis_neg_2, cmb_neg_2, slope_2);
    u_fric_3 = fric_comp(Omega3, min_vel_3, vis_pos_3, cmb_pos_3, vis_neg_3, cmb_neg_3, slope_3);

    // Part 2: Implement the Inverse Dynamics Control Law
    a_m2 = Omega2d_des + KP_2_inv * (error_2) + KD_2_inv * (Omega2_des - Omega2);
    a_m3 = Omega3d_des + KP_3_inv * (error_3) + KD_3_inv * (Omega3_des - Omega3);

    if (mode == 0){ // Inverse Dynamics controller without Mass
        *tau1 = KP_1 * (error_1) + KD_1 * (Omega1_des - Omega1) + J1*Omega1d_des;
        *tau2 = p_1[0]*a_m2 -p_1[2]*(sintheta3*costheta2-sintheta2*costheta3)*a_m3 + (-p_1[2]*(costheta2*costheta3+sintheta2*sintheta3))*Omega3 - p_1[3]*g*sintheta2+fric_gain*u_fric_2;
        *tau3 = -p_1[2]*(sintheta3*costheta2-sintheta2*costheta3)*a_m2 + p_1[1]*a_m3 + (p_1[2]*(costheta2*costheta3+sintheta2*sintheta3))*Omega2 -p_1[4]*g*costheta3+fric_gain*u_fric_3;
    }
    else if(mode == 1){ // PD Controller without Mass
        // PD + Feedforward
        *tau1 = KP_1 * (error_1) + KD_1 * (Omega1_des - Omega1) +  J1*Omega1d_des+fric_gain*u_fric_1;
        *tau2 = KP_2 * (error_2) + KD_2 * (Omega2_des - Omega2) +  J2*Omega2d_des+fric_gain*u_fric_2;
        *tau3 = KP_2 * (error_3) + KD_3 * (Omega3_des - Omega3) +  J3*Omega3d_des+fric_gain*u_fric_3;
    }
    else if(mode ==2){ // Inverse Dynamics Controller with Mass
        *tau1 = KP_1 * (error_1) + KD_1 * (Omega1_des - Omega1) + J1*Omega1d_des;
        *tau2 = p_2[0]*a_m2 -p_2[2]*(sintheta3*costheta2-sintheta2*costheta3)*a_m3 + (-p_2[2]*(costheta2*costheta3+sintheta2*sintheta3))*Omega3 - p_2[3]*g*sintheta2+fric_gain*u_fric_2;
        *tau3 = -p_2[2]*(sintheta3*costheta2-sintheta2*costheta3)*a_m2 + p_2[1]*a_m3 + (p_2[2]*(costheta2*costheta3+sintheta2*sintheta3))*Omega2 -p_2[4]*g*costheta3+fric_gain*u_fric_3;
    }
    else if(mode ==3){ // PD Controller with Mass
        *tau1 = KP_1 * (error_1) + KD_1 * (Omega1_des - Omega1) +  J1*Omega1d_des+fric_gain*u_fric_1;
```

```
        *tau2 = KP_2 * (error_2) + KD_2 * (Omega2_des - Omega2) +  J2*
            Omega2d_des+fric_gain*u_fric_2;
        *tau3 = KP_2 * (error_3) + KD_3 * (Omega3_des - Omega3) +  J3w*
            Omega3d_des+fric_gain*u_fric_3;
    }
    mycount++;
}
```