University of Illinois at Urbana Champaign

Friction Compensation and Inverse Dynamics Control

Lab Report III

Karan Chawla karangc2@illinois.edu
Zhaoer Li zli91@illinois.edu
Zhichao Zhang zhichao3@illinois.edu

Course Instructor: Prof. Daniel Block & Prof. Hae Won Park

April 10, 2017

1 Introduction

In this lab we add friction compensation for all three joints and investigate how well it works to cancel friction effects in the arm, design and implement an inverse dynamics control algorithm and compare its performance to PD plus feedforward control designed in lab 3, investigate how well the parameters of the CRS robot have been identified and see if better values for the same can be obtained.

2 Tuning Friction

In this lab, a torque to compensate friction inside the CRS robot joints is designed and added to control. Ideally, the friction of CRS robot joint would be modeled as a combination of both, coulomb and viscous friction, as shown in the figure below.

So, at the joints a counter torque equal to the torque generated by the friction is applied when the joint speed is 0. This is the coulomb friction and the counter torque that is proportional to the angular speed of the joint when its velocity is non-zero is due to viscous friction.

To simplify the model, the coulomb friction is modeled as a straight line with high slope instead of a vertical line in the region when the angular speed is less than a critical value. After the angular velocity crosses this critical speed, the friction lines becomes flatter as shown in Fig. 2.1. In order to better model the actual robot, a different slope will be adjusted for positive and negative velocity.

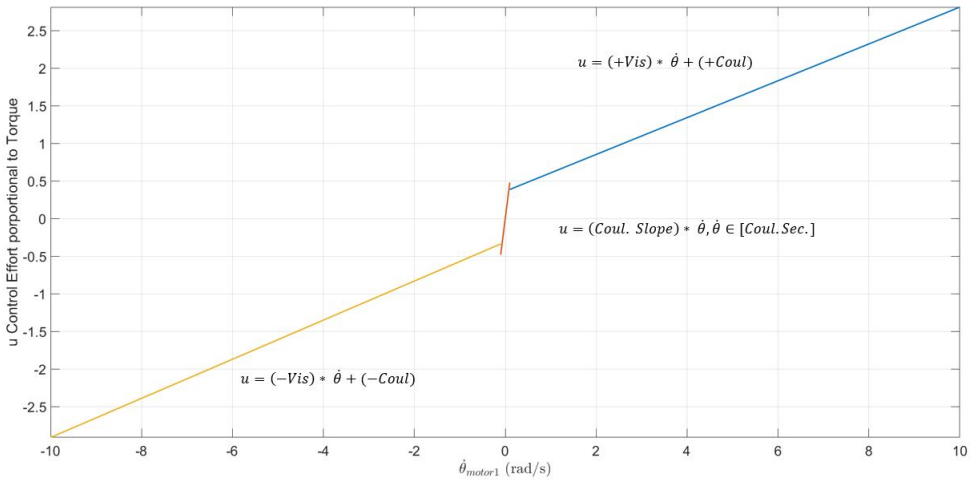

Figure 2.1: Friction Compensation Plot

The friction compensation plot includes three parts. When the angular velocity is less than the critical speed, the motor need to provide more torque to counter the friction. So the slope at low speed zone is large. In the high speed zone, the slope is small. Because of the gravity, the positive velocity and the negative velocity have different slopes and

different coulumb values. Applying additional torque on joint according to this plot, the joint can rotate like frictionless. In practice, by taking this plot into consideration, the final torque applied on the joint contains the torque that counters the friction and the torque we desired. Therefore, the final performance of the joint is extremely close to our desire.

3 Compensation Implementation

The program implementing the compensation has 6 constant, 3 for positive velocity and 3 for negative velocity. The C code uses the filtered velocity calculation from the previous lab and calculates the friction compensation torques.

```
float slopesJoint1[3] = {0.245,0.26,4.8};
float colJoint1[2] = {0.1,-0.1};
float intersectJoint1[2] = {0.3637, -0.31};

float slopesJoint2[3] = {0.25,0.287,3.6};
float colJoint2[2] = {0.05,-0.05};
float intersectJoint2[2] = {0.4759, -0.5031};
float tuneJoint2 = 0.5;

float slopesJoint3[3] = {0.35,0.2132,4.5};
float colJoint3[2] = {0.09,-0.09};
float intersectJoint3[2] = {0.195, -0.5190};

      //For joint 1
      if(joint=='1')
      {
         if(thetamotor_dot > colJoint1[0])
         {
            tau = tau + slopesJoint1[0]*thetamotor_dot + intersectJoint1[0] ;
         }
         else if(thetamotor_dot < colJoint1[1])
         {
            tau = tau + slopesJoint1[1]*thetamotor_dot + intersectJoint1[1];
         }
         else
         {
            tau = tau + slopesJoint1[2]*thetamotor_dot;
         }
      }
      //For joint 2
      else if(joint=='2')
      {
            if(thetamotor_dot > colJoint2[0])
            {
               tau = tau + tuneJoint2*( slopesJoint2[0]*thetamotor_dot +
                   intersectJoint2[0]) ;
```

```
        }
        else if(thetamotor_dot < colJoint2[1])
        {
           tau = tau + tuneJoint2*(slopesJoint2[1]*thetamotor_dot +
               intersectJoint2[1]);
        }
        else
        {
           tau = tau + tuneJoint2*(slopesJoint2[2]*thetamotor_dot);
        }
    }
    //For joint 3
    else if(joint=='3')
    {
            if(thetamotor_dot > colJoint3[0])
            {
               tau = tau + slopesJoint3[0]*thetamotor_dot +
                   intersectJoint3[0] ;
            }
            else if(thetamotor_dot < colJoint3[1])
            {
               tau = tau + slopesJoint3[1]*thetamotor_dot +
                   intersectJoint3[1];
            }
            else
            {
               tau = tau + slopesJoint3[2]*thetamotor_dot;
            }
    }
```

4 Compensation Constants Tuning

The most accurate way of tuning the constants is to set the robot in a configuration with the joints axis parallel to gravity(to avoid its interference) and tune them until finding the movement stays with constant velocity after a gentle push. This was done with joint one, but couldn't be done with the other two joints, that were tuned until seeing the characteristic free motion parabola when pushing them up.

Motor	+Vis	-Vis	+Coul	-Coul	Coul Sec.	Coul Slope
1	0.245	0.26	0.3637	-0.31	[-0.1,0.1]	4.8
2	0.25	0.287	0.4759	-0.5031	[-0.05,0.05]	3.6
3	0.35	0.2132	0.195	-0.5190	[-0.09,0.09]	4.5

Table 4.1: Tuned Gains for Inverse Dynamics Control

5 Inverse Dynamics Equation

In order to track the cubic polynomial reference trajectory, which is represented by the equation below

$$0.25 + 13.50t^2 - 27t^3 = q \tag{5.1}$$

Inverse dynamics control for joints two and three is implemented. The robot arm can be expressed by robot dynamics equations in matrix form as follows:

$$D(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = u \tag{5.2}$$

The idea of inverse dynamics controller is to seek a non-linear control law that cancels out the dynamics of the system.

$$u = f(q,\dot{q},t) \tag{5.3}$$

$$D(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = f(q,\dot{q},t) \tag{5.4}$$

The dynamics could be a linear closed loop system. Since D matrix is invertible,

$$\ddot{q} = D^{-1}(u - C(q,\dot{q})\dot{q} - g(q)) \tag{5.5}$$

and the system can be reduced to $\ddot{q} = a_q$.
Now we can input a_q to control a scalar linear system, so a_q can control a linear second order system($\ddot{q}$) it can be represented as follows:

$$a_q = -K_P q - K_D\dot{q} + r \tag{5.6}$$

$$\ddot{q} + K_P q + K_D\dot{q} = r \tag{5.7}$$

where r is the reference input as $r(t) = \ddot{q^d} + K_P q^d(t) + K_D\dot{q^d}(t)$. Then tracking the error $e(t) = q - q^d$.

$$\ddot{e}(t) + K_P e(t) + K_D e(t) = 0 \tag{5.8}$$

The acceleration $\ddot{q}$ of the manipulator can be solved for using Eq. 5.5 and $\ddot{q} = a_q(t)$ where $a_q(t)$ is the input acceleration vector. Using Eq. 5.6 and the definition of r, a_q can be written as:

$$a_q(t) = \ddot{q}^d + K_P(q^d - q) + K_D(\dot{q}^d - \dot{q}) \tag{5.9}$$

Using this equation we can control the torque input to the system.

$$\tau = D(\theta)a_\theta + C(\theta,\dot{\theta}) + g(\theta) \tag{5.10}$$

This is called the inner loop controller and the computation for a_θ is called the outer loop controller.

Joint 1 was controlled using a feedforward controller whereas joint 2 and joint 3 were controlled with the inverse dynamics control law thus defined.

$$a_{\theta_2}(t) = \ddot{\theta_2}^d + K_{P_2}(\theta_2^d - \theta_2) + K_{D_2}(\dot{\theta_2}^d - \dot{\theta_2}) \tag{5.11}$$

$$a_{\theta_3}(t) = \ddot{\theta_3}^d + K_{P_3}(\theta_3^d - \theta_3) + K_{D_3}(\dot{\theta_3}^d - \dot{\theta_3}) \tag{5.12}$$

6 Results

Here we present the comparison between the inverse dynamics controller and the PD control with feedforward that was implemented in the previous labs. To obtain a better control, the proportional and differential gains have been tuned. Since the motion of joint 2 and joint 3 is heavily dependent on one another, the gains were tuned simultaneously, in order to reach the performance criteria of a rise time of less than 300ms and 1% or less overshoot. The final tuned parameters for each joint can be seen in the table 6.1.

Motor	K_P	K_D
2	4000	200
3	4000	200

Table 6.1: Tuned Gains for Inverse Dynamics Control

6.1 Comparison between PD controller and Inverse Dynamics Controller

After implementing the previously described inverse dynamics controller, we inspect the performance of the robot arm tracking a cubic trajectory.

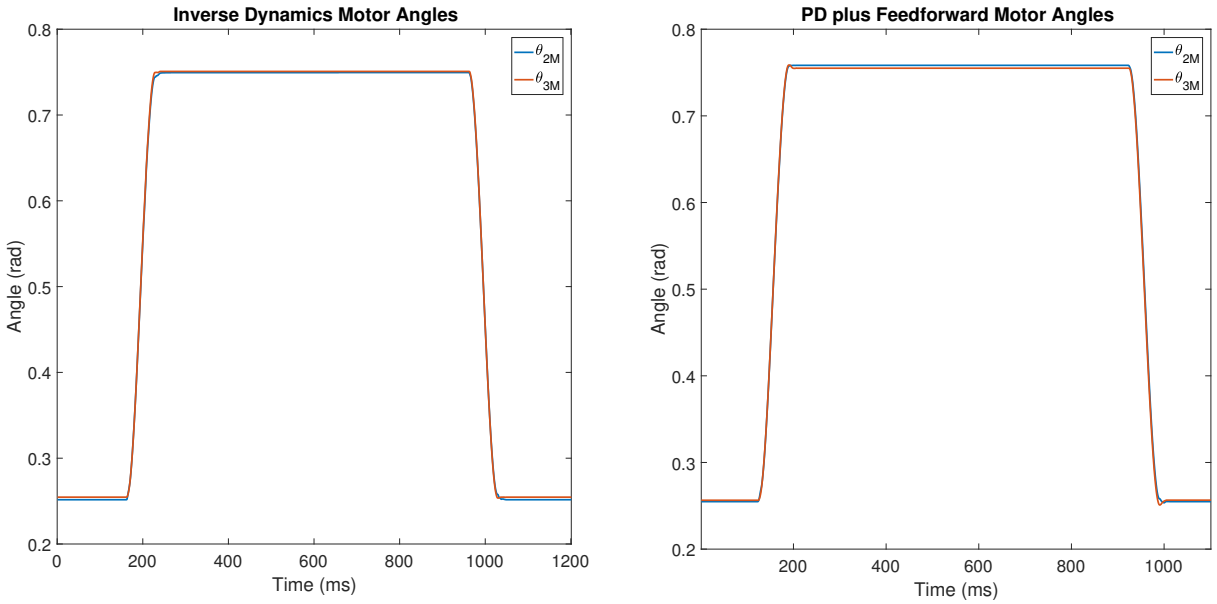

Figure 6.1: Comparison between Inverse Dynamics and PD Control on CRS Robot Arm: Trajectory

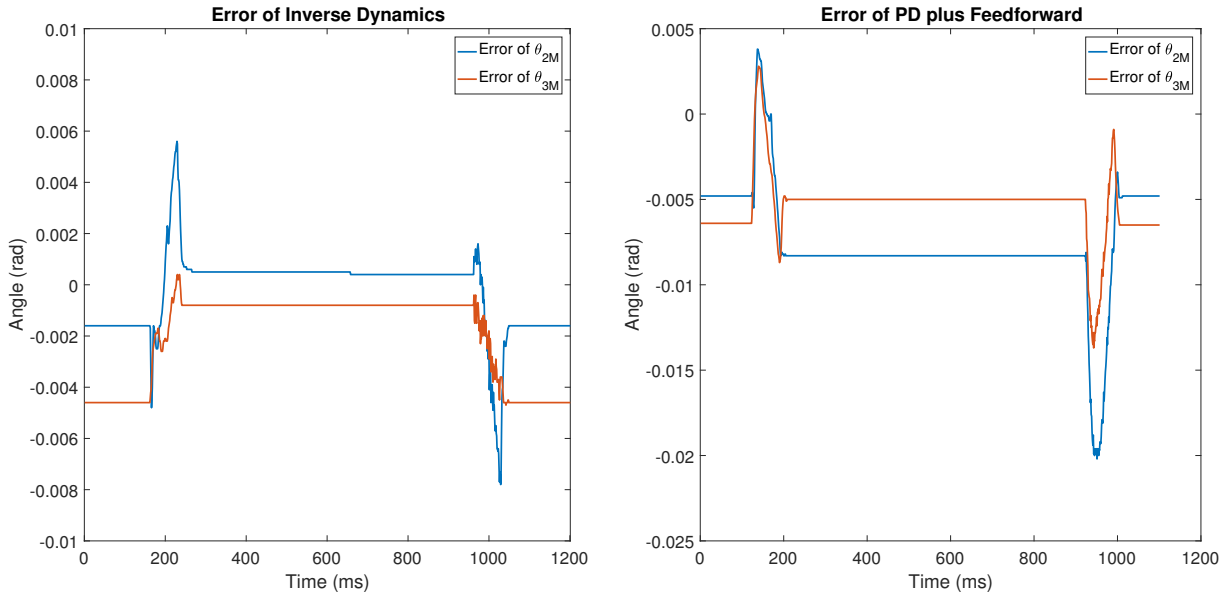

Figure 6.2: Comparison between Inverse Dynamics and PD Control on CRS Robot Arm: Error

In Fig. 6.1 and Fig. 6.2 the performance of the inverse dynamics controller is compared to that of a PD & feedforward control implemented in the previous lab. From the error plot we can tell that PD & Feedforward control performs much better than inverse dynamics control does in terms of steady state error and error peaks.

The following code explains the cubic trajectory that was implemented.

The cubic trajectory is a trajectory whose path between two points is a cubic polynomial with respect to time.

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

$$\dot{q}(t) = a_1 + 2a_2 t + 3a_3 t^2 := v(t)$$

$$\ddot{q}(t) = 2a_2 + 6a_3 t := \alpha(t)$$

The boundary condition is $q(t_0) = q_0$, $q(t_f) = q_f$, $v(t_0) = v_0$,$v(t_f) = v_f$. Therefore, the functions become

$$a_0 + a_1 t_0 + a_2 {t_0}^2 + a_3 {t_0}^3 = q_0$$

$$a_1 + 2a_2 t_0 + 3a_3 {t_0}^2 = v_0$$

$$a_0 + a_1 t_f + a_2 {t_f}^2 + a_3 {t_f}^3 = q_f$$

$$a_1 + 2a_2 t_f + 3a_3 {t_f}^2 = v_f$$

These functions can be written in the matrix form

$$\begin{bmatrix} 1 & t_0 & {t_0}^2 & {t_0}^3 \\ 0 & 1 & 2t_0 & 3{t_0}^2 \\ 1 & t_f & {t_f}^2 & {t_f}^3 \\ 0 & 1 & 2t_f & 3{t_f}^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} q_0 \\ v_0 \\ q_f \\ v_f \end{bmatrix}$$

For we know the t_0,t_f,q_0,q_f,v_0 and v_f, the values of the coefficients can be varified. In this lab, the boundary condition is $q_0 = 0$, $q_f = 0.75$, $v_0 = v_f = 0$. Thus, the values of the coefficients can be found by the following program

```
%For the first trajectory

t_0=0;
t_f=1/3;
q_0=0.25;
v_0=0;
q_f=0.75;
v_f=0;

T_mat=[1,t_0,t_0^2,t_0^3;0,1,2*t_0,3*t_0^2;
1,t_f,t_f^2,t_f^3;0,1,2*t_f,3*t_f^2];
qv=[q_0,v_0,q_f,v_f]';

a=inv(T_mat)*qv;

%For the second trajectory

t_0=4;
t_f=4+1/3;
q_0=0.75;
v_0=0;
q_f=0.25;
v_f=0;

T_mat=[1,t_0,t_0^2,t_0^3;0,1,2*t_0,3*t_0^2;
1,t_f,t_f^2,t_f^3;0,1,2*t_f,3*t_f^2];
qv=[q_0,v_0,q_f,v_f]';

a=inv(T_mat)*qv;
```

6.2 Comparison between PD controller and Inverse Dynamics Controller with External Mass

After investigation on both controllers with the original robot arm, an external mass is screwed at the end effector of the robot. A comparison of performance of the two controllers on the new system is made. New parameters for the system are applied using the parallel access theorem.

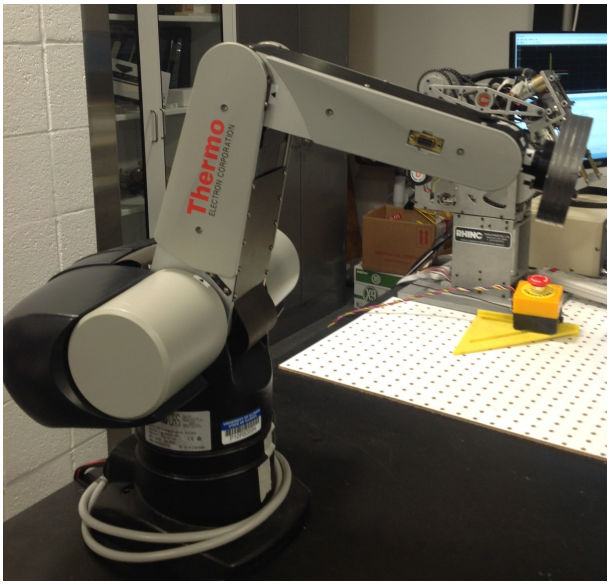

Figure 6.3: Experimental Setting of CRS Robot Arm Screwed with External Mass

The experimental setting is shown in Fig 6.3.

Original P	New P
0.0300	0.0466
0.0128	0.0388
0.0076	0.0284
0.0753	0.1405
0.0298	0.1298

Table 6.2: Original and New P parameters

From Table 6.2 we notice that each new P parameter is larger than its previous value because of a larger mass and inertia.

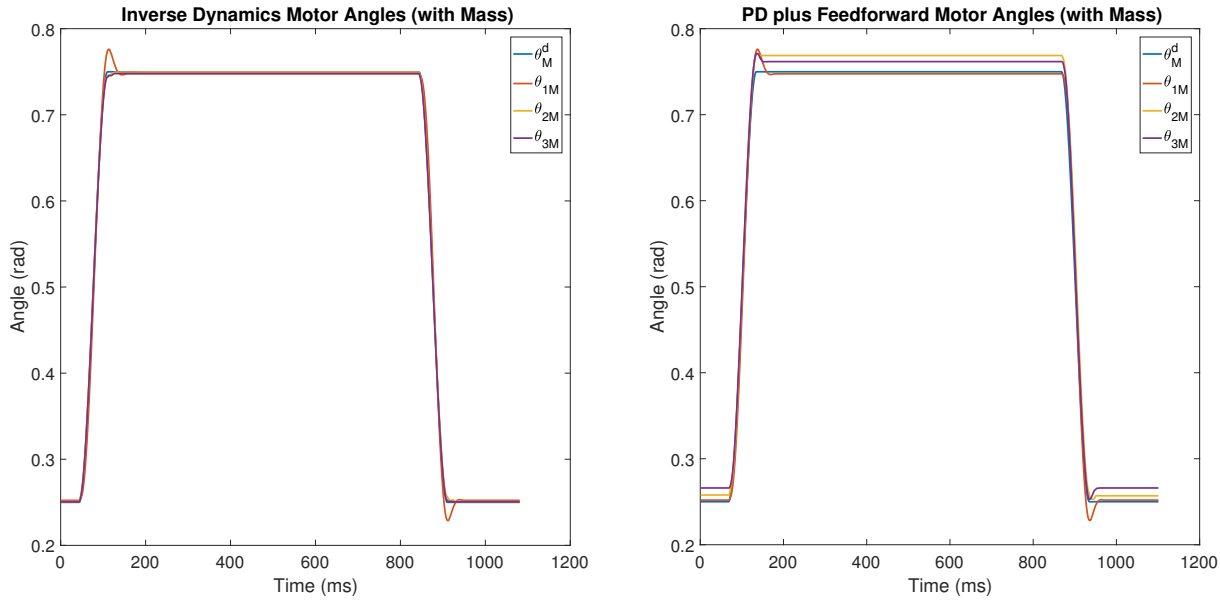

Figure 6.4: Comparison between Inverse Dynamics and PD Control on CRS Robot Arm with External Mass: Trajectory

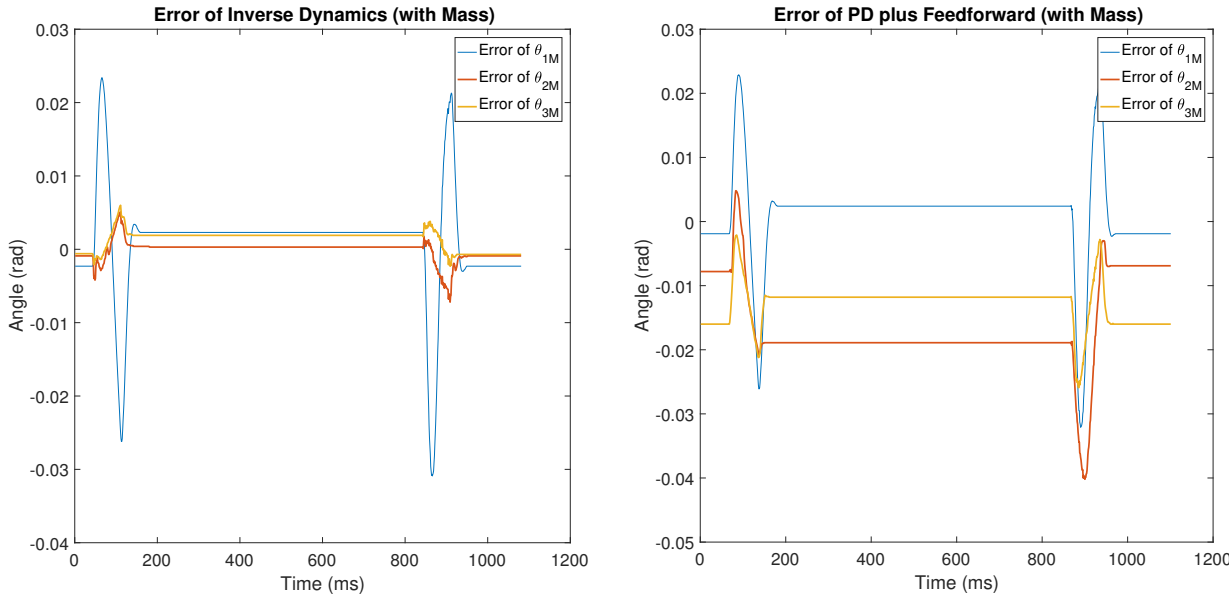

Figure 6.5: Comparison between Inverse Dynamics and PD Control on CRS Robot Arm with External Mass: Error

From Fig. 6.4 and Fig. 6.5, we see that the inverse dynamics controller works better

than PD & Feedforward in terms of steady state error and error peaks. Specifically, θ_{motor2} and θ_{motor3} with PD & Feedforward control has large steady state error and error peaks, especially in the negative direction, while with inverse dynamics control they have a noticeably smaller steady state error and error peaks, regardless of angle direction. Therefore, we can conclude that the gravity effect has definitely been improved with inverse dynamics control. Moreover, the outperforming of inverse dynamics control indicates that the new parameters applied for robot arm with external mass are well defined, or at least a good approximation of the actual system. Note that in both controllers, motor of joint 1 is controller by PD & Feedforward, thus performs similarly in both cases.

7 Appendix

lab.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 = -0.3834;
float offset_Enc3_rad = 0.2756;


// Your global varialbes.

long mycount = 0;
long newCount = 0;

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

//positions in the global coordinate frame
float x;
float y;
float z;
float thetas[3];

//PID Gains structure definition
```

```
typedef struct PID
{
   float Kp;
   float Kd;
   float Ki;
}PID;

typedef struct PDinv
{
   float Kp;
   float Kd;
}PDinv;

typedef struct vel_filter
{
   float theta_old;
   float omega_old1;
   float omega_old2;
   float omega;
}vel_filter;

//Friction Compensation Parameters

float slopesJoint1[3] = {0.245,0.26,4.8};
float colJoint1[2] = {0.1,-0.1};
float intersectJoint1[2] = {0.3637, -0.31};

float slopesJoint2[3] = {0.25,0.287,3.6};
float colJoint2[2] = {0.05,-0.05};
float intersectJoint2[2] = {0.4759, -0.5031};
float tuneJoint2 = 0.5;

float slopesJoint3[3] = {0.35,0.2132,4.5};
float colJoint3[2] = {0.09,-0.09};
float intersectJoint3[2] = {0.195, -0.5190};

//Global theta desired values
float theta1motor_des = 0.0;
float theta2motor_des = 0.0;
float theta3motor_des = 0.0;

//Declaration of PID objects for 3 joints
PID ctrl1 = {60,3,0.3};
PID ctrl2 = {60,3,0.3};
PID ctrl3 = {60,3,0.3};

PID control1 = {60,3,0.3};

//Initializing velocity struct
```

```
vel_filter filter1 = {0,0,0,0};
vel_filter filter2= {0,0,0,0};
vel_filter filter3= {0,0,0,0};

float prev_error1 = 0.0;
float prev_error2 = 0.0;
float prev_error3 = 0.0;

float curr_error1 = 0.0;
float curr_error2 = 0.0;
float curr_error3 = 0.0;

float total_error1 = 0.0;
float total_error2 = 0.0;
float total_error3 = 0.0;

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

//Feedforward constants
float J1 = 0.0167;
float J2 = 0.03;
float J3 = 0.0128;
float td1 = 0;
float td2 = 0;
float td3 = 0;

float td10 = 0;
float td20 = 0;
float td30 = 0;

float td1n = 0;
float td2n = 0;
float td3n = 0;

float td1dot = 0;
float td2dot = 0;
float td3dot = 0;

float td1ddot = 0;
float td2ddot = 0;
float td3ddot = 0;

float xr0 = 12;
float y0 = 0;
float zr0 = 8;
float Rr0 = 3;

static inline float velocityFilter(float thetamotor, vel_filter* vel_filter)
{
```

```
   vel_filter->omega = (thetamotor - vel_filter->theta_old)/0.001;
   vel_filter->omega = (vel_filter->omega + vel_filter->omega_old1 +
       vel_filter->omega_old2)/3.0;

   vel_filter->theta_old = thetamotor;

   vel_filter->omega_old2 = vel_filter->omega_old1;
   vel_filter->omega_old1 = vel_filter->omega;

   return vel_filter->omega;
}

//Inverse kinematics for the robot arm
void invKinematics(float x, float y, float z, float* thetas)
{
   float z0 = z - 10;
   float d = (x*x + y*y + z0*z0 - 200)/200;
   float q1inv = atan2(y, x);
   float q3inv = atan2(sqrt(1 - d*d), d);
   float q2inv = -atan2(z0, sqrt(x * x + y * y) ) - atan2(10 * sin(q3inv) , 10
       + 10 * cos(q3inv));

   float q1 = q1inv;
   float q2 = q2inv + PI/2;
   float q3 = q3inv + q2 - PI/2;

   thetas[0] = q1;
   thetas[1] = q2;
   thetas[2] = q3;
}

//Generate the cubic trajectory which was calculated in MATLAB
void trajectoryGenerator(float t)
{
   float Coeff1[4] = {0.25, 0.0, 13.5, -27.0};
   float Coeff2[4] = {-1.943250000000086e+03, 1.404000000000062e+03,
       -3.375000000000148e+02, 2.700000000000117e+01};
   /*
    * 0.12% steady state error
    */
   if(t <= 333)
   {
      t=t/1000;
      td1 = Coeff1[0]+Coeff1[1]*t+Coeff1[2]*t*t+Coeff1[3]*t*t*t;
      td1dot = Coeff1[1]+2*Coeff1[2]*t+3*Coeff1[3]*t*t;
      td1ddot = 2*Coeff1[2]+6*Coeff1[3]*t;

      td2 = Coeff1[0]+Coeff1[1]*t+Coeff1[2]*t*t+Coeff1[3]*t*t*t;
      td2dot = Coeff1[1]+2*Coeff1[2]*t+3*Coeff1[3]*t*t;
```

```
      td2ddot = 2*Coeff1[2]+6*Coeff1[3]*t;

      td3 = Coeff1[0]+Coeff1[1]*t+Coeff1[2]*t*t+Coeff1[3]*t*t*t;
      td3dot = Coeff1[1]+2*Coeff1[2]*t+3*Coeff1[3]*t*t;
      td3ddot = 2*Coeff1[2]+6*Coeff1[3]*t;
   }
   if( t < 4000 && t > 333)
   {
//    td1 = td1;
      td1dot = 0;
      td1ddot = 0;

//    td2 = 0t;
      td2dot = 0;
      td2ddot = 0;

//    td3 = 0.75;
      td3dot = 0;
      td3ddot = 0;
      }
   if( t >=4000 && t <= 4333)
   {
      t= t/1000;

      td1 = Coeff2[0]+Coeff2[1]*t+Coeff2[2]*t*t+Coeff2[3]*t*t*t;
      td1dot = Coeff2[1]+2*Coeff2[2]*t+3*Coeff2[3]*t*t;
      td1ddot = 2*Coeff2[2]+6*Coeff2[3]*t;

      td2 = Coeff2[0]+Coeff2[1]*t+Coeff2[2]*t*t+Coeff2[3]*t*t*t;
      td2dot = Coeff2[1]+2*Coeff2[2]*t+3*Coeff2[3]*t*t;
      td2ddot = 2*Coeff2[2]+6*Coeff2[3]*t;

      td3 = Coeff2[0]+Coeff2[1]*t+Coeff2[2]*t*t+Coeff2[3]*t*t*t;
      td3dot = Coeff2[1]+2*Coeff2[2]*t+3*Coeff2[3]*t*t;
      td3ddot = 2*Coeff2[2]+6*Coeff2[3]*t;
   }
   if( t > 4333)
   {
//    td1 = 0.25;
      td1dot = 0;
      td1ddot = 0;

//    td2 = 0.25;
      td2dot = 0;
      td2ddot = 0;

//    td3 = 0.25;
      td3dot = 0;
      td3ddot = 0;
```

```
   }
}


static inline float Controller(char joint , float thetamotor, float
    thetamotor_dot, PID *ctrl, float* curr_error, float* prev_error, float
    *total_error)
{
   //Use the PID controller close to desired angles
   float tau = 0;
   float int_error = 0;


   if(fabs(*curr_error) < 0.2)
   {
      //For joint 1
      if(joint=='1')
      {
         //Integration error
         *curr_error = (td1 - thetamotor);
         *total_error = *total_error + (*prev_error + *curr_error)*.001/2;
         int_error = *total_error;
         tau = ctrl->Kp*(td1-thetamotor) + ctrl->Kd*(td1dot -
             thetamotor_dot);// + ctrl->Ki*(int_error); //J2*td1ddot
         *prev_error = *curr_error;
         //tau = 0;
         if(thetamotor_dot > colJoint1[0])
         {
            tau = tau + slopesJoint1[0]*thetamotor_dot + intersectJoint1[0] ;
         }
         else if(thetamotor_dot < colJoint1[1])
         {
            tau = tau + slopesJoint1[1]*thetamotor_dot + intersectJoint1[1];
         }
         else
         {
            tau = tau + slopesJoint1[2]*thetamotor_dot;
         }
      // tau = 0;
      }
      //For joint 2
      else if(joint=='2')
      {
         //Integration error
         *curr_error = (td2 - thetamotor);
         *total_error = *total_error + (*prev_error + *curr_error)*.001/2;
         int_error = *total_error;
         tau = ctrl->Kp*(td2-thetamotor) + ctrl->Kd*(td2dot - thetamotor_dot);
             //+ ctrl->Ki*(int_error);
```

```
        *prev_error = *curr_error;
        //tau = 0;
           if(thetamotor_dot > colJoint2[0])
           {
              tau = tau + tuneJoint2*( slopesJoint2[0]*thetamotor_dot +
                  intersectJoint2[0]) ;

           }
           else if(thetamotor_dot < colJoint2[1])
           {
              tau = tau + tuneJoint2*(slopesJoint2[1]*thetamotor_dot +
                  intersectJoint2[1]);
           }
           else
           {
              tau = tau + tuneJoint2*(slopesJoint2[2]*thetamotor_dot);
           }
           //tau = 0;

     }
     //For joint 3
     else if(joint=='3')
     {
        //Integration error
        *curr_error = (td3 - thetamotor);
        *total_error = *total_error + (*prev_error + *curr_error)*.001/2;
        int_error = *total_error;
        tau = ctrl->Kp*(td3-thetamotor) + ctrl->Kd*(td3dot - thetamotor_dot);
            //+ ctrl->Ki*(int_error);
        *prev_error = *curr_error;
     // tau = 0;
           if(thetamotor_dot > colJoint3[0])
           {
              tau = tau + slopesJoint3[0]*thetamotor_dot +
                  intersectJoint3[0] ;
           }
           else if(thetamotor_dot < colJoint3[1])
           {
              tau = tau + slopesJoint3[1]*thetamotor_dot +
                  intersectJoint3[1];
           }
           else
           {
              tau = tau + slopesJoint3[2]*thetamotor_dot;
           }

     }
  }
  else
```

```
{
   //Use PD controller for everything else
   *total_error = 0;
   if(joint=='1')
   {
      tau = J1*td1ddot + ctrl->Kp*(td1-thetamotor) + ctrl->Kd*(td1dot -
          thetamotor_dot);
      //tau = 0;
      if(thetamotor_dot > colJoint1[0])
      {
         tau = tau + slopesJoint1[0]*thetamotor_dot + intersectJoint1[0] ;
      }
      else if(thetamotor_dot < colJoint1[1])
      {
         tau = tau + slopesJoint1[1]*thetamotor_dot + intersectJoint1[1];
      }
      else
      {
         tau = tau + slopesJoint1[2]*thetamotor_dot;
      }
      //tau = 0;
   }
   else if(joint=='2')
   {
      tau = J2*td2ddot + ctrl->Kp*(td2-thetamotor) + ctrl->Kd*(td2dot -
          thetamotor_dot);
      //tau = 0;
      if(thetamotor_dot > colJoint2[0])
      {
         tau = tau + tuneJoint2*(slopesJoint2[0]*thetamotor_dot +
             intersectJoint2[0]) ;

      }
      else if(thetamotor_dot < colJoint2[1])
      {
         tau = tau + tuneJoint2*(slopesJoint2[1]*thetamotor_dot +
             intersectJoint2[1]);
      }
      else
      {
         tau = tau + tuneJoint2*( slopesJoint2[2]*thetamotor_dot);
      }
      //tau = 0;

   }
   else if(joint=='3')
   {
      tau = J3*td3ddot + ctrl->Kp*(td3-thetamotor) + ctrl->Kd*(td3dot -
          thetamotor_dot);
```

```
        //tau = 0;
        if(thetamotor_dot > colJoint3[0])
        {
           tau = tau + slopesJoint3[0]*thetamotor_dot + intersectJoint3[0] ;
        }
        else if(thetamotor_dot < colJoint3[1])
        {
           tau = tau + slopesJoint3[1]*thetamotor_dot + intersectJoint3[1];
        }
        else
        {
           tau = tau + slopesJoint3[2]*thetamotor_dot;
        }

     }
  }

  //Saturate output torque
  if(tau>5)
     return 5.0;
  else if(tau < -5)
     return -5.0;
  else
     return tau;
}

//Very hacky change this later
float thetamotor_dot[3] = {0,0,0};
float thetamotor[3] = {0,0,0};

float Kp2 = 4000;
float Kd2 = 200;

float Kp3 = 4000;
float Kd3 = 200;

float sintheta2 = 0;
float costheta2 = 0;
float costheta3 = 0;
float sintheta32 = 0;
float costheta32 = 0;

float g = 9.81;

//Inverse Dynamics Control
static inline float invDynamicsController(char joint , float *thetamotor,
    float *thetamotor_dot, PID *control,float sintheta2,float costheta2,float
    costheta3,float sintheta32,float costheta32)
{
```

```
   //Use the inverse dynamics controller everywhere
   float tau = 0;

   //params for inv dynamics control law
   //float p[5] = {0.0300, 0.0128, 0.0076, 0.0753, 0.0298};
   float p[5] = { 0.0466, 0.0388, 0.0284, 0.1405, 0.1298};//with mass
   // Coeffs for inv dynamics control
   float a2 = td2ddot + Kp2*(td2 - thetamotor[1]) +
       Kd2*(td2dot-thetamotor_dot[1]);
   float a3 = td3ddot + Kp3*(td3 - thetamotor[2]) + Kd3*(td3dot -
       thetamotor_dot[2]);

   //Inv dynamics controller

   //For joint 1 it remains the same
   if(joint=='1')
   {
      tau = J1*td1ddot + control->Kp*(td1 - thetamotor[0]) +
          control->Kd*(td1dot - thetamotor_dot[0]);
      if(thetamotor_dot[0] > colJoint1[0])
      {
         tau = tau + slopesJoint1[0]*thetamotor_dot[0] + intersectJoint1[0] ;
      }
      else if(thetamotor_dot[0] < colJoint1[1])
      {
         tau = tau + slopesJoint1[1]*thetamotor_dot[0] + intersectJoint1[1];
      }
      else
      {
         tau = tau + slopesJoint1[2]*thetamotor_dot[0];
      }
   }
   else if(joint=='2')
   {
      tau = (p[0]*a2 - 2*p[2]*sintheta32*a3) + ( -
          p[2]*costheta32*thetamotor_dot[2]*thetamotor_dot[2]) + (
          -p[3]*g*sintheta2);


//    tau += J2*td2ddot;
      if(thetamotor_dot[1] > colJoint2[0])
      {
         tau = tau + tuneJoint2*(slopesJoint2[0]*thetamotor_dot[1] +
             intersectJoint2[0]) ;
         }
      else if(thetamotor_dot[1] < colJoint2[1])
      {
         tau = tau + tuneJoint2*(slopesJoint2[1]*thetamotor_dot[1] +
             intersectJoint2[1]);
```

```
      }
      else
      {
         tau = tau + tuneJoint2*( slopesJoint2[2]*thetamotor_dot[1]);
      }
   }
   else if(joint=='3')
   {
      tau = ( - p[2]*sintheta32)*a2 + p[1]*a3
           + ( p[2]*costheta32*thetamotor_dot[1] )*thetamotor_dot[1]
           + ( -p[4]*g*costheta3 );

//    tau += J3*td3ddot;

      if(thetamotor_dot[2] > colJoint3[0])
      {
         tau = tau + slopesJoint3[0]*thetamotor_dot[2] + intersectJoint3[0] ;
      }
      else if(thetamotor_dot[2] < colJoint3[1])
      {
         tau = tau + slopesJoint3[1]*thetamotor_dot[2] + intersectJoint3[1];
      }
      else
      {
         tau = tau + slopesJoint3[2]*thetamotor_dot[2];
      }
   }
   //Saturate output torque
   if(tau>5)
      return 5.0;
   else if(tau < -5)
      return -5.0;
   else
      return tau;
}


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

   float theta1motor_dot = velocityFilter(theta1motor, &filter1);
   float theta2motor_dot = velocityFilter(theta2motor, &filter2);
   float theta3motor_dot = velocityFilter(theta3motor, &filter3);

   sintheta2 = sin(theta2motor);
   costheta2 = cos(theta2motor);
   costheta3 = cos(theta3motor);
   sintheta32 = sin(theta3motor-theta2motor);
```

```
   costheta32 = cos(theta3motor-theta2motor);

   thetamotor[0] = theta1motor;
   thetamotor[1] = theta2motor;
   thetamotor[2] = theta3motor;

   thetamotor_dot[0] = theta1motor_dot;
   thetamotor_dot[1] = theta2motor_dot;
   thetamotor_dot[2] = theta3motor_dot;

   trajectoryGenerator(mycount%8000);

   //Motor torque limitation(Max: 5 Min: -5)

   //Feedforward control for the joints
   *tau1 = Controller('1',theta1motor, theta1motor_dot, &ctrl1, &curr_error1,
       &prev_error1, &total_error1);
   *tau2 = Controller('2',theta2motor, theta2motor_dot, &ctrl2, &curr_error2,
       &prev_error2, &total_error2);
   *tau3 = Controller('3',theta3motor, theta3motor_dot, &ctrl3, &curr_error3,
       &prev_error3, &total_error3);

//
// *tau1 = invDynamicsController('1',thetamotor,thetamotor_dot,
    &control1,sintheta2,costheta2,costheta3,sintheta32,costheta32);
// *tau2 = invDynamicsController('2',thetamotor,thetamotor_dot,
    &control1,sintheta2,costheta2,costheta3,sintheta32,costheta32);
// *tau3 = invDynamicsController('3',thetamotor,thetamotor_dot,
    &control1,sintheta2,costheta2,costheta3,sintheta32,costheta32);

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

      theta1array[arrayindex] = theta1motor;
      theta2array[arrayindex] = theta2motor;
      if (arrayindex >= 100) {
         arrayindex = 0;
      } else {
         arrayindex++;
      }

   }

   if ((mycount%1000)==0) {
      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 = td1;
   Simulink_PlotVar2 = theta1motor;
   Simulink_PlotVar3 = theta2motor;
   Simulink_PlotVar4 = theta3motor;

   mycount++;
   newCount++;

}

void printing(void){
   serial_printf(&SerialA, "%.2f %.2f %.2f :: %.2f %.2f %.2f :: %.2f %.2f %.2f
       \n\r",printtheta1motor*180/PI,printtheta2motor*180/PI,printtheta3motor*180/PI,x,y,z,thetas
}
```