Lab 3: Inverse Dynamics Joint Control

Caleb Albers & Karun Koppula

May 11, 2018

1 Introduction

The focus of this lab 3 is to implement friction compensation and inverse dynamics control laws. The friction compensation acted to reduce mechanical friction effects on all three joints. The inverse dynamics control served to look at the robustness of control methods for systems with varying payloads, which we tested by the addition of a heavy mass to the end-effector of our robot.

2 Friction Compensation

Newton's first Law of Motion states that any object in motion will stay in motion unless an external force acts upon it. One of the biggest forces that the CRS robot faces (other than gravity) is that of friction from the internal gears, chains, and motors. Although our previous control systems usually compensated somewhat for friction, it was never perfect. In all cases, friction caused more control effort to be extended. In order to allow our control systems to deal primarily with disturbances rather than friction forces, we implemented a friction compensation system. Ideally, this friction compensation would allow one to set a joint in motion and have the joint continue rotating forever (ignoring mechanical limits, gravity, and other external forces of course).

The mechanical friction present in each joint was accounted for by determining both the Coulomb (static) and viscous (dynamic) friction for each joint. In a further attempt to accurately characterize this, we measured these friction coefficients for joints moving in the positive direction and the negative direction. These coefficients are present in Equations 2.1 and 2.2.

The friction compensation was added independently for each joint and acted linearly with respect to joint velocity. Once a minimum velocity (0.05 m/s) was met. A slope of 3.6 was implemented between minimum velocities, as given in the lab manual. The implementation we had is present in Listing 1. This calculated compensation was then added to the τ control effort sent to each motor.

```
void friction_compensation() {
    int i = 0;
    for(i = 0; i < 3; i++) {
        if (Omega[i] > minimum_velocity[i]) {
            u_fric[i] = Viscous_positive[i]*Omega[i] + Coulomb_positive[i];
        } else if (Omega[i] < -minimum_velocity[i]) {
            u_fric[i] = Viscous_negative[i]*Omega[i] + Coulomb_negative[i];
        } else {
            u_fric[i] = slope_between_minimums[i]*Omega[i];
        }
    }
}
```

Listing 1: Friction Compensation Method

$$\begin{aligned} Viscous_1^+ &= 0.130 \qquad Viscous_1^- = 0.074 \\ Viscous_2^+ &= 0.250 \qquad Viscous_2^- = 0.210 \\ Viscous_3^+ &= 0.210 \qquad Viscous_3^- = 0.330 \end{aligned} \tag{2.1}$$

$$\begin{aligned} Coulomb_1^+ &= 0.300 \qquad Coulomb_1^- = -0.390 \\ Coulomb_2^+ &= 0.250 \qquad Coulomb_2^- = -0.600 \\ Coulomb_3^+ &= 0.400 \qquad Coulomb_3^- = -0.600 \end{aligned} \tag{2.2}$$

3 PD Plus Feedforward Control Law

3.1 Trajectory Tracking without Weight

Figure 1 shows the trajectory tracking response of the PD plus feedforward controller. The scale of the trajectory makes the error of the response difficult to see, with rise time at 330 milliseconds and the full trajectory at 4 seconds long.

Figure 2 shows a zoomed in picture of the response, at the peak, which gives a clearer idea of the errors.

Figure 3 shows the error over time of the feedforward trajectory tracking without weight. The response shows that the error tracking is different at the top and bottom of the trajectory response. While Joint 1 has no steady state error, the response takes a different

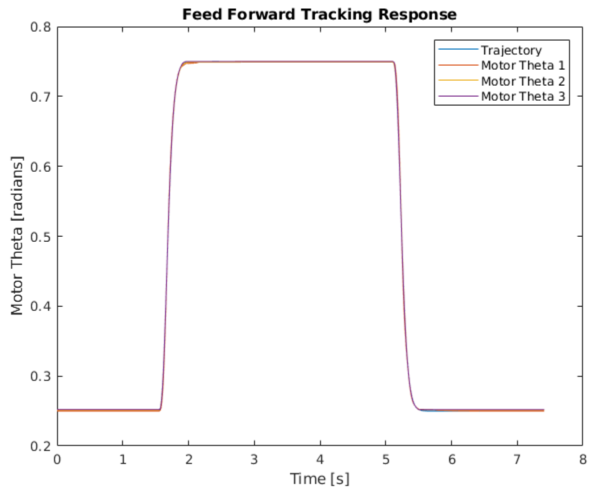

Figure 1: Feed Forward Trajectory Tracking without Weight

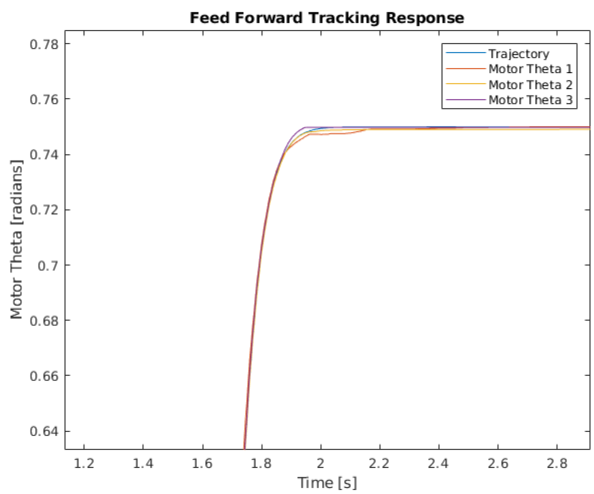

Figure 2: Close Up Feed Forward Trajectory Tracking without Weight

amount of time to settle to that state. The step response also shows a difference between the two movements, with the downwards movement from 0.75 radians to 0.25 radians having a greater error for Joint 1 than the upwards movement. This is not a gravitational effect, since the link actuated by Joint 1 is not acted upon by gravity in the axis of actuation. Joints 2 and 3 have steady state error on both sides of the trajectory, not reaching the desired point on either side. Without integral control, the control signal doesn't accumulate. The magnitude of the error is greater at the bottom of the response for both Joints 2 and 3.

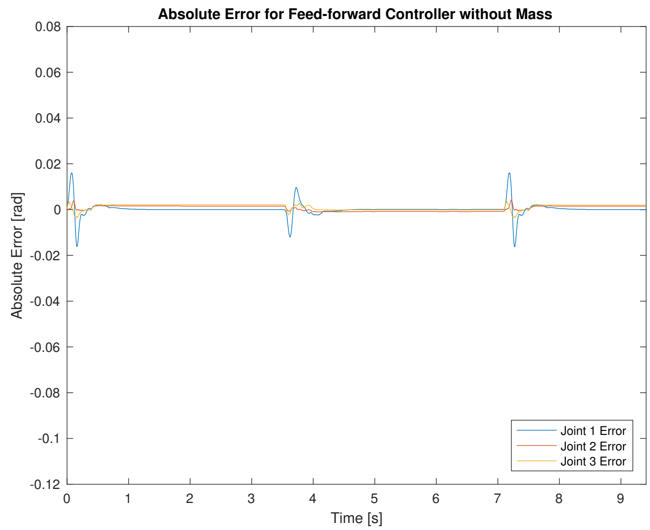

Figure 3: Error for Feed-Forward Controller without Mass Added

3.2 Trajectory Tracking with Weight

When the mass is added to the robot the performance of the controller response declines, although the steady state error decreases for all three joints. The full response for all three joints is given by Figure 4. For Joints 2 and 3, the error goes to zero at both sides of the response, while for Joint 1 it only goes to zero at the top of the response.

The response is still hard to see with the full step up and down, so a zoomed in view of the peak of the upwards movement is provided by 5

The error signal for all three joints is provided by Figure 6. Similar to the response without the added weight, the error for the trajectory tracking on the upwards movement is less for all three joints. On the upwards motion, the error response for all joints is approximately the same value as the error for Joint 1 without the mass, about 0.02 radians. On the downwards movement, the error is about five times larger, reaching a magnitude of 0.1 radians. This difference in response for Joints 2 and 3 can be explained by the force of gravity working on the mass. Gravity is against the motion of mass on the upwards motion and with the mass on the downwards motion. Joint 1 error can be partially explained by the inertia of the mass. The mass is moving faster on the downswing which causes the joint to move farther in that direction.

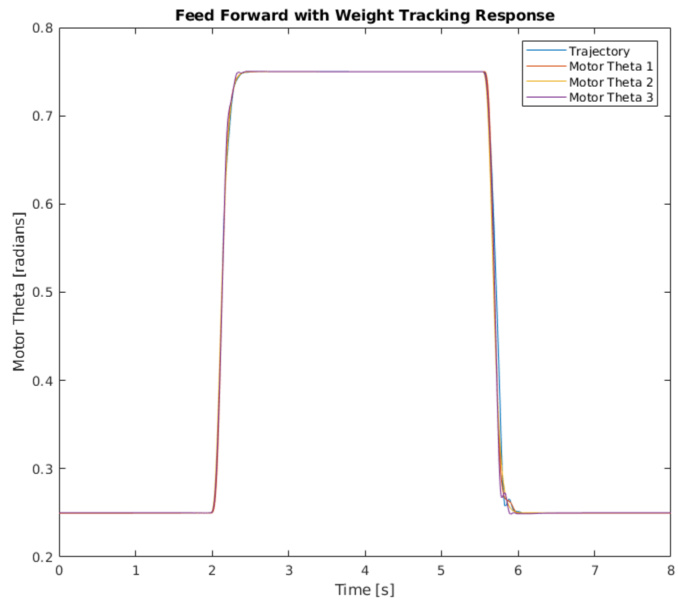

Figure 4: Feed Forward Trajectory Tracking with Weight

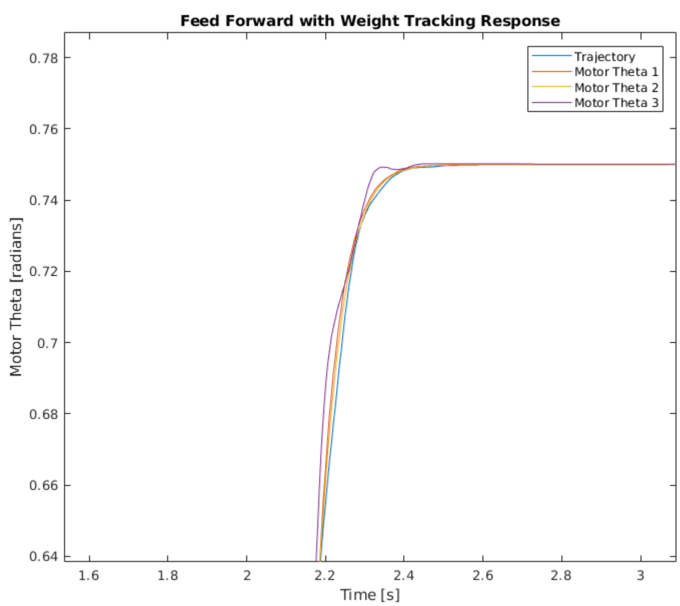

Figure 5: Close Up Feed Forward Trajectory Tracking with Weight

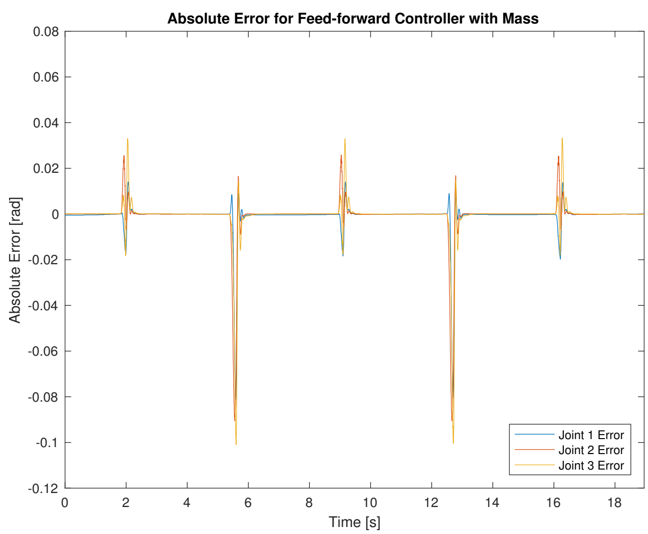

Figure 6: Error for Feed-Forward Controller with Mass Added

4 Inverse Dynamics Control Law

Implementing a control law based on inverse dynamics can be thought of as a cascaded, two-loop control system. The "inner loop" inverts the dynamics of the robot to linearize the system, while the "outer loop" implements a PD controller. To expand on this topic, outer loop acts as a PD controller on the acceleration $\ddot{\theta}$. The inner loop is based on the standard $\tau = D(\theta)\ddot{\theta} + C(\theta, \dot{\theta}) + g(\theta)$ dynamics equation, with the inertia matrix, Coriolis matrix, and gravity matrix respectively. However, inverting these terms to isolate $\ddot{\theta}$ means that a direct relationship between α_θ and $\ddot{\theta}$ can be found, as shown in Equation 4.3, with all other dynamics cancelled out. This α_θ (the control effort from the outer loop) is then able to be used in the dynamics Equation 4.2 in place of $\ddot{\theta}$ with respect to the inertia matrix $D(\theta)$. This allows us to utilize PD control while still taking into account the non-linearity of the actual robot's system response. As such, it leads to a more robust controller, as the dynamics that are inverted in the inner loop can be easily modified based on mass added to the system, while not requiring the PD control gains to be modified.

$$\begin{aligned}
p_1 &= m_2 l_{c2}^2 + m_2 l_2^2 + I_2 &&= 0.0300 \\
p_2 &= m_3 l_{c3}^2 + I_3 &&= 0.0128 \\
p_3 &= m_3 l_2 l_{c3} &&= 0.0076 \\
p_4 &= m_2 l_{c2} + m_3 l_2 &&= 0.0753 \\
p_5 &= m_3 l_{c3} &&= 0.0298
\end{aligned} \tag{4.1}$$

The parameters given in Equation 4.1 defined the inverse dynamics of the robot without any mass added to the end-effector. These parameters are utilized in the matrices shown as part of the inner loop control in Equation 4.2. The outer loop PD control is dictated by the α values calculated from Equation 4.4 and 4.5. The PD gains used in this calculation are shown in Equation 4.6. We implemented the inverse dynamics control on Joints 2 and 3, however kept feed-forward control on Joint 1 (as the z-axis rotation was reasonably linear). Figure 7 shows the program flow.

$$\tau = D(\theta)\alpha_\theta + C(\theta, \dot{\theta})\dot{\theta} + g(\theta) \tag{4.2}$$

$$\alpha_\theta = \ddot{\theta} \tag{4.3}$$

$$\alpha_{\theta_2} = \ddot{\theta}_2^d + K_{P2} \times (\theta_2^d - \theta_2) + K_{D2} \times (\dot{\theta}_2^d - \dot{\theta}_2) \tag{4.4}$$

$$\alpha_{\theta_3} = \ddot{\theta}_3^d + K_{P3} \times (\theta_3^d - \theta_3) + K_{D3} \times (\dot{\theta}_3^d - \dot{\theta}_3) \tag{4.5}$$

$$
\begin{aligned}
k_{p,1} &= 0 & k_{d,1} &= 0 \\
k_{p,2} &= 10000 & k_{d,2} &= 200 \\
k_{p,3} &= 10000 & k_{d,3} &= 200
\end{aligned}
\tag{4.6}
$$

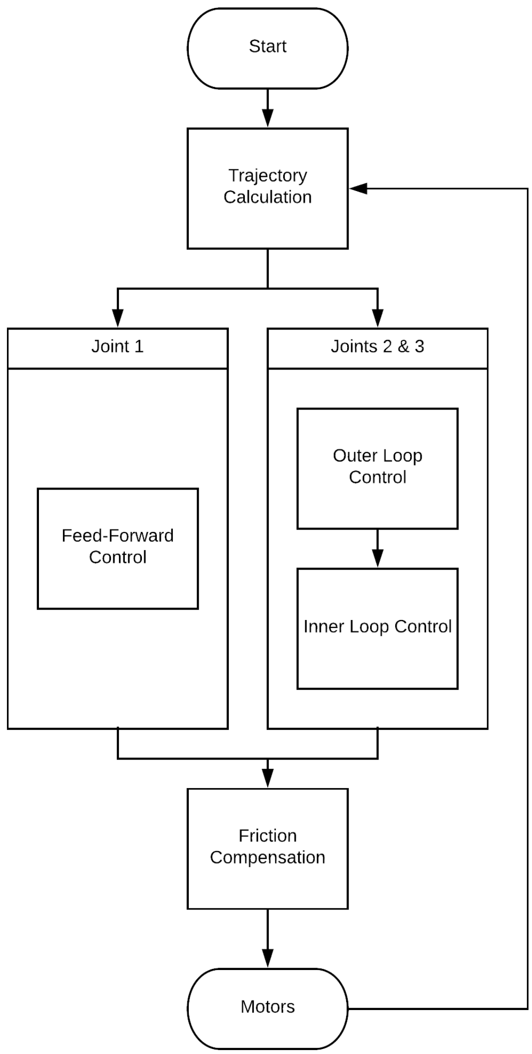

Figure 7: Program Execution Flowchart

4.1 Smooth Trajectory

Instead of utilizing the same cubic trajectory as in Lab 2, we utilized a discrete approximation of the transfer function shown in Equation 4.7 given by a Matlab function that was provided by the lab instructors. This trajectory was much smoother than the cubic we previously used. The position trajectory is shown in Figure 8, the velocity in Figure 9, and the acceleration in Figure 10.

$$\text{Transfer Function} = (\frac{30}{\text{s} + 30})^6 \tag{4.7}$$

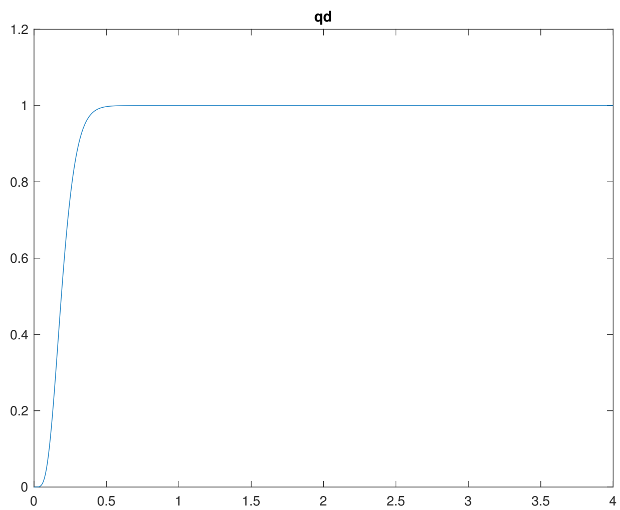

Figure 8: Desired Trajectory Position

4.2 Trajectory Tracking without Weight

The inverse dynamics step response with the smoothed trajectory is shown in Figure 11.

Since the motor response tracks very closely to the desired trajectory , it is helpful to see the zoomed in version of the response, at the peak of the response. Figure 12 gives this perspective with the scale given by the axes.

The inverse dynamics controller performs nearly identically to the PD plus feedforward controller without the mass attached. The magnitude of the maximum error response for both motions for Joints 2 and 3 were approximately the same for each controller. The values

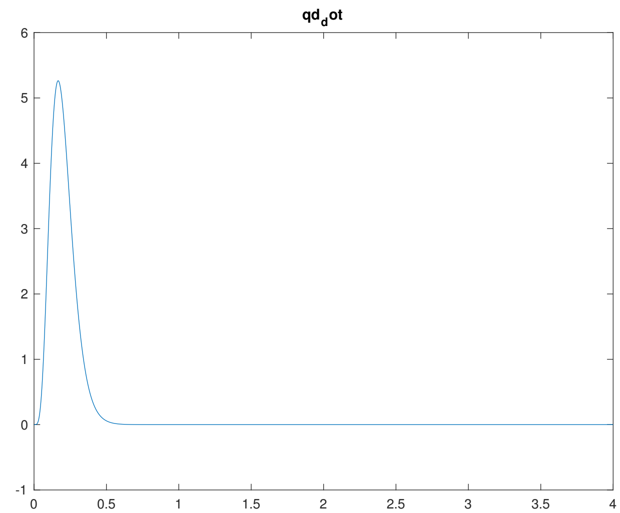

Figure 9: Desired Trajectory Velocity

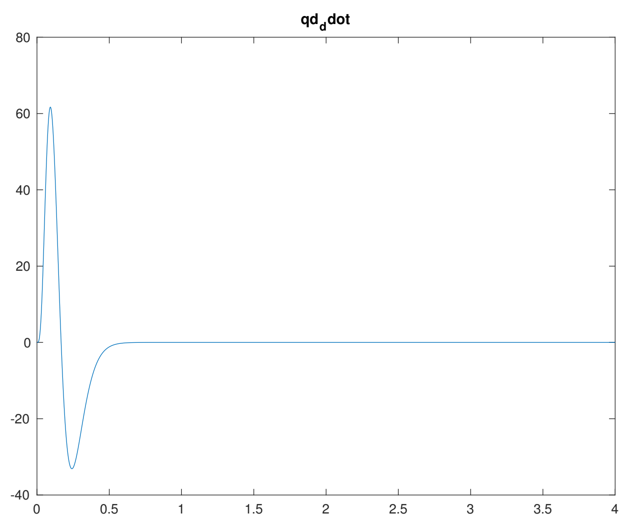

Figure 10: Desired Trajectory Acceleration

were all under 0.005 radians. The steady state error for both sides of the response were about the same as well, less than or equal to 0.002 radians.

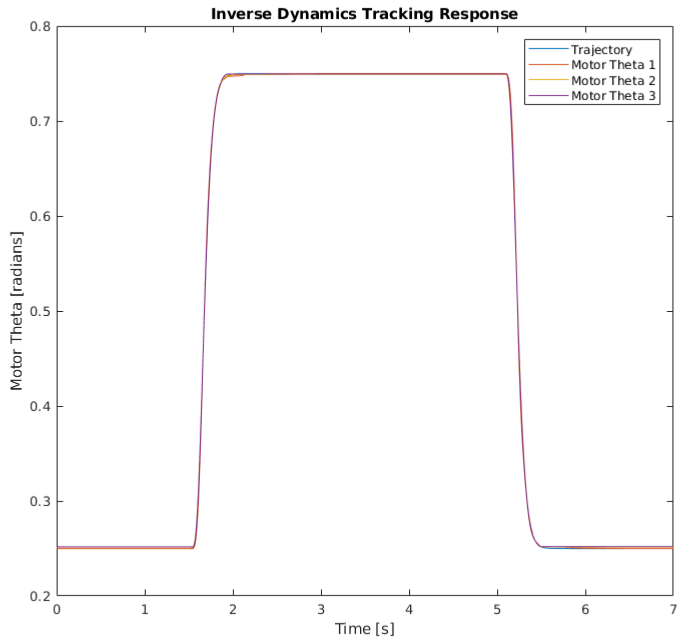

Figure 11: Inverse Dynamics Trajectory Tracking without Weight

4.3 Trajectory Tracking with Weight

$$
\begin{aligned}
p_1 &= 0.0466 \\
p_2 &= 0.0388 \\
p_3 &= 0.0284 \\
p_4 &= 0.1405 \\
p_5 &= 0.1298
\end{aligned} \tag{4.8}
$$

The inverse dynamics step response with the weight added to the end effector is shown by Figure 14. When the weight is added to the end effector, and the dynamical parameters are changed to account for the new equations of motion, the response is managed much better than the PD plus feedforward controller on the upwards motion. There is still significant error on the downwards motion for Joint 2. We tuned the controller response exclusively for the upwards motion response, so we did not account for this error. Joint 3 was tuned to the same parameters, but did not exhibit this behavior.

Since the motor response tracks very closely to the desired trajectory as compared to other controllers, it is helpful to see the Figure 15.

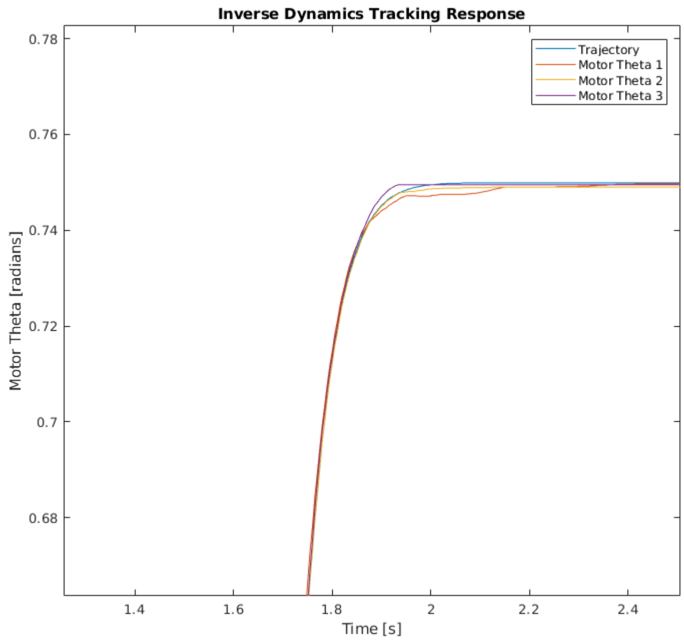

Figure 12: Close Up Inverse Dynamics Trajectory Response

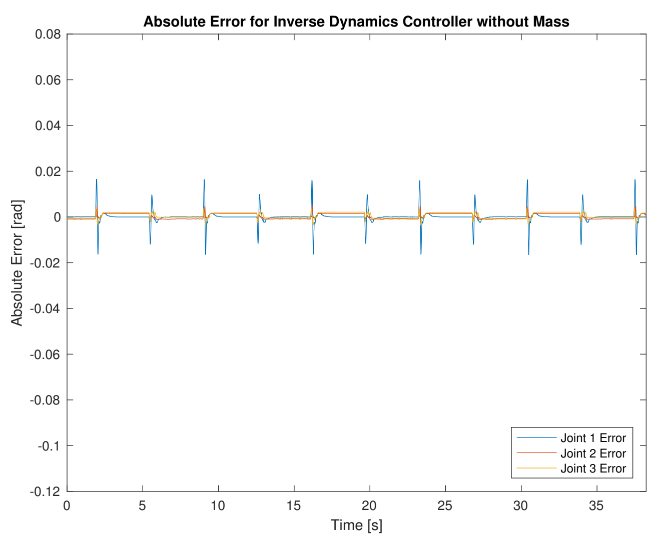

Figure 13: Error for Inverse Dynamics Controller with Mass Added

The inverse dynamics controller reduces the maximum error of the response for Joints 2 and 3 on the upwards motion by an order of magnitude as compared to the PD plus feedforward controller with the attached mass. There is, however, slight steady state error for the two

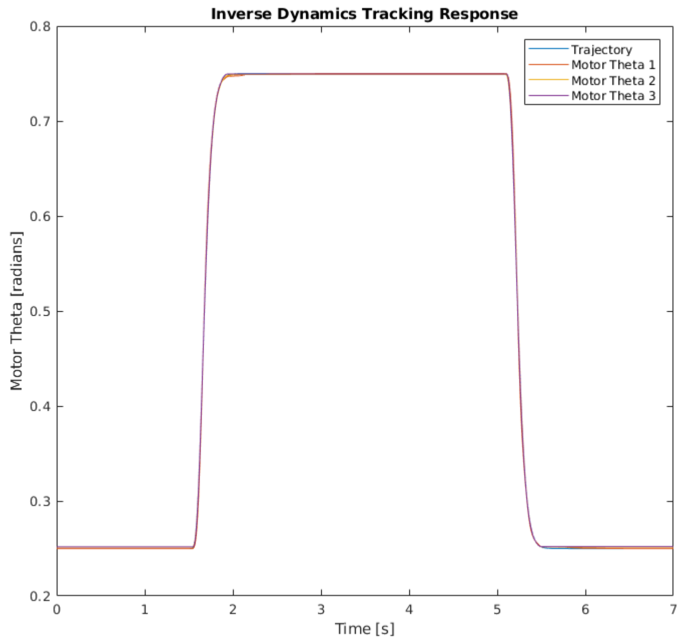

Figure 14: Inverse Dynamics Trajectory Tracking with Weight

joints, while the feedforward controller settled to zero. This response shows that the inverse dynamics controller has a better response, but does not account for all non-linearities of the system. There is also no integral control being used on this controller to remove steady state error.

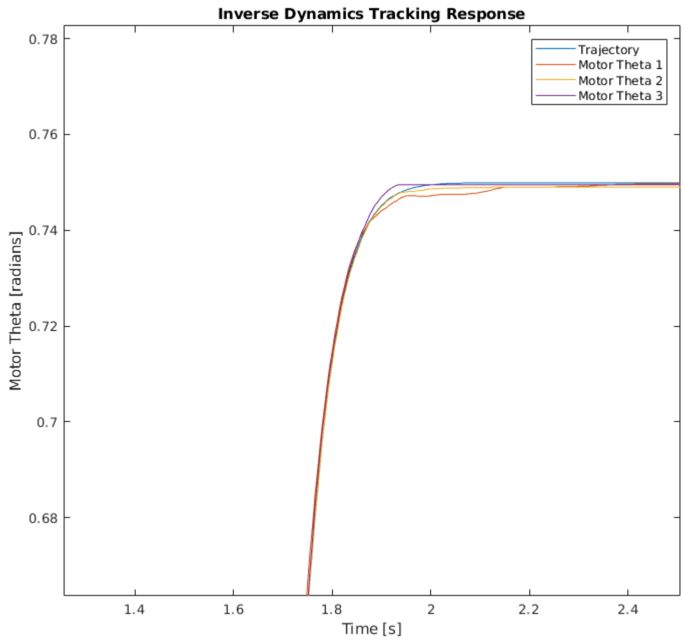

Figure 15: Close Up Inverse Dynamics Trajectory Response with Weight

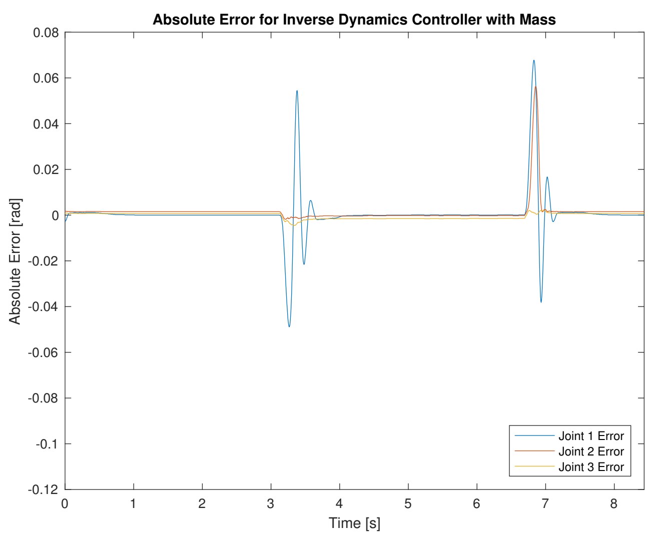

Figure 16: Error for Inverse Dynamics Controller without Mass Added

5 Inverse Dynamics vs. PD Control

The inverse dynamics controller was able to properly compensate for added mass, where the feed-forward was unable to achieve the same. This shows that the parameters given were able to accurately characterize the system and account for the added payload mass.

The inverse dynamics controller is more robust to changes in end-effector/payload mass. Both the feed-forward and inverse dynamics controllers saw a reduction in overshoot by about 50% when a large mass was added to the end-effector. This was to be expected, as the force due to gravity from the added mass dampened the overshoot effect. However, the inverse dynamics controller performed significantly better for undershoot. The feed-forward controller saw undershoot values increase almost 7-fold, however the inverse dynamics controller saw no discernible increase in overshoot, even with the added mass.

6 Conclusion

This lab included the implementation of friction compensation and inverse dynamics control. We explored the advantages of inverse dynamics control by adding mass to the end-effector of the robot (thereby changing the dynamics) and seeing how the inverse dynamics controller response differed with respect to a standard feed-forward controller. The superior robustness provided by inverse dynamics control methods was clearly seen in that regard.

A lab3.c code

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

/*
 * 1 for inverse dynamics
 * 2 for feed-forward only (all joints)
 */
int CONTROL_MODE = 2;

// 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 = -0.37;
//float offset_Enc3_rad = 0.27;

float offset_Enc2_rad = -0.427257;
float offset_Enc3_rad = 0.230558;


// Your global variables.

long mycount = 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];

#pragma DATA_SECTION(theta3array, ".my_arrs")
float theta3array[100];


long arrayindex = 0;

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

float x_pos = 0;
float y_pos = 0;
float z_pos = 0;

float fric_on = 0.0;

```

```
// inverse kinematics
float theta_1 = 0;
float theta_2 = 0;
float theta_3 = 0;

float motor_theta_1 = 0;
float motor_theta_2 = 0;
float motor_theta_3 = 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;

//Controller Parameters
//Proportional
float kp[3] = {110, 130, 55};

//Derivative
float kd[3] = {2,2,1.7};

//Intergral
float ki[3] = {550,600,185};

// velocity filtering
float Theta_old[3] = {0,0,0};
float Omega_old1[3] = {0,0,0};
float Omega_old2[3] = {0,0,0};
float Omega[3] = {0,0,0};

//Integral Estimation
float Ik[3] = {0,0,0};
float Ik_old[3] = {0,0,0};
float e_old[3] = {0,0,0};

float integral_threshold = 0.1;


// current positions
float theta_motor[3] = {0,0,0};

//current tau
float t[3] = {0,0,0};

//feedfoward control
float theta_d[3] = {0,0,0};
float theta_dot_d[3] = {0,0,0};
float theta_ddot_d[3] = {0,0,0};

float J[3] = {0.0167, 0.03, 0.0128};

```

```
// friction compensation
float minimum_velocity[3] = {0.05, 0.05, 0.05};
float u_fric[3] = {0,0,0};
float Viscous_positive[3] = {0.130,0.2500,0.21};
float Viscous_negative[3] = {0.074,0.21,0.33};
float Coulomb_positive[3] = {0.300,0.25,0.4};
float Coulomb_negative[3] = {-0.390,-0.6,-0.6};
float slope_between_minimums[3] = {3.6,3.6,3.6};

//Controller Parameters
//Proportional
float kp_inv[3] = {0, 10000, 10000};

//Derivative
float kd_inv[3] = {0,200,200};

//alpha values
float a_theta[3] = {0, 0, 0};

float mystep = 0.25;





//This function calculates the motor positions given a desired x,y,and z
    position
void inverse_kinematics(float x, float y, float z){

    //The DH angles are calculated using a geometric analysis of the possible
    configurations of the robot
    theta_1 = atan2(y,x);
      //The base DH angle is the inverse tangent between the x and y
    coordinates
    theta_3 = 3.14159 - acos(-(pow(x,2)+pow(y,2)+pow(z-10,2)-200)/200);
      //The elbow DH angle is calculated using the law of cosines
    theta_2 = -(atan2(z-10,sqrt(pow(x,2)+pow(y,2))) + theta_3/2);
      //The shoulder DH angle is calculated using Pythagoras' Theorem and the
    half angle formula

    //The DH angles must be converted to motor theta angles using appropriate
    transformation
    theta_d[0] = theta_1;
    theta_d[1] = theta_2 + PI/2;
    theta_d[2] = theta_3 + theta_2;
}

void filter_velocity() {
    int i;
    for(i = 0; i < 3; i++) {
        Omega[i] = (theta_motor[i] - Theta_old[i])/0.001;
        Omega[i] = (Omega[i] + Omega_old1[i] + Omega_old2[i])/3.0;
```

```
        Theta_old[i] = theta_motor[i];

        Omega_old2[i] = Omega_old1[i];
        Omega_old1[i] = Omega[i];
    }
}

void estimate_integral() {
    int i;
    for(i=0;i<3;i++){
        float e_k = theta_d[i] - theta_motor[i];
        Ik[i] = Ik_old[i] + (e_old[i] + e_k)/2*0.001;
        Ik_old[i] = Ik[i];
        e_old[i] = e_k;
    }
}



void pid_control() {
    int i = 0;
    for (i = 0; i < 3; i++){
        t[i] = kp[i]*(theta_d[i]-theta_motor[i]) - kd[i]*Omega[i];
        if (fabs(theta_d[i] - theta_motor[i]) < integral_threshold){
            estimate_integral();
            t[i] = t[i] + ki[i]*Ik[i];
        } else {
            Ik[i] = 0;
            Ik_old[i] = 0;
        }


        if (t[i] >= 5) {
            t[i] = 5;
            Ik[i] = Ik_old[i];
        }
        else if (t[i] <= -5) {
            t[i] = -5;
            Ik[i] = Ik_old[i];
        }
    }
}

//void trajectory(float time){
//    //lemniscate
//    float x_d = 5*sqrt(2)*cos(PI*time)*sin(PI*time)/(sin(PI*time)*sin(PI*time) + 1) + 14;
//    float y_d = 5*sqrt(2)*cos(PI*time)/(sin(PI*time)*sin(PI*time) + 1);
//    float z_d = 3*sin(PI*time)+ 10;
//
//
//    inverse_kinematics(x_d,y_d,z_d);
```

```
//}

//void get_trajectory(float time) {
//     float i = 0;
//     for (i = 0; i < 3; i++){
//         if((time >= 0) && (time < 1)) {
//             theta_d[i] = 1.5*pow(time,2) -pow(time,3);
//             theta_dot_d[i] = 3*time - 3*pow(time,2);
//             theta_ddot_d[i] = 3 - 6*time;
//         }
//         else if ((time >= 1) && (time <=2 )) {
//             theta_d[i] = -2 + 6*time - 4.5*pow(time,2) + pow(time,3);
//             theta_dot_d[i] = 6 - 9*time +3*pow(time,2);
//             theta_ddot_d[i] = -9 +6*time;
//         }
//         else {
//             theta_d[i] = 0;
//             theta_dot_d[i] = 0;
//             theta_ddot_d[i] = 0;
//         }
//     }
//
//}

void feedforward_control() {

    int joint_limit = 1;

    if(CONTROLMODE == 1) {
        joint_limit = 1;
    }
    else if(CONTROLMODE == 2) {
        joint_limit = 3;
    }

    int i = 0;
    for (i = 0; i < joint_limit; i++){ // ONLY OPERATE ON JOINT 1 WHILE USING INVERSE DYNAMICS CONTROL LOOPS
        t[i] = kp[i]*(theta_d[i]-theta_motor[i]) + kd[i]*(theta_dot_d[i] - Omega[i]) + J[i]*theta_ddot_d[i];
        if (fabs(theta_d[i] - theta_motor[i]) < 0.05){
            estimate_integral();
            t[i] = t[i] + ki[i]*Ik[i];
        } else {
            Ik[i] = 0;
            Ik_old[i] = 0;
        }
        if (t[i] >= 5) {
            t[i] = 5;
            Ik[i] = Ik_old[i];
        }
        else if (t[i] <= -5) {
```

```
            t[i] = -5;
            Ik[i] = Ik_old[i];
        }
    }

}

void friction_compensation() {
    int i = 0;
    for(i = 0; i < 3; i++) {
        if (Omega[i] > minimum_velocity[i]) {
            u_fric[i] = Viscous_positive[i]*Omega[i] + Coulomb_positive[i];
        } else if (Omega[i] < -minimum_velocity[i]) {
            u_fric[i] = Viscous_negative[i]*Omega[i] + Coulomb_negative[i];
        } else {
            u_fric[i] = slope_between_minimums[i]*Omega[i];
        }
    }
}


void inverse_dynamics_outer_loop() {
    int joint = 1; // only implement on Joint 2 and 3
    for(joint = 1; joint < 3; joint++) {
        a_theta[joint] = theta_ddot_d[joint]
                        + kp_inv[joint]*(theta_d[joint]-theta_motor[joint])
                        + kd_inv[joint]*(theta_dot_d[joint]-Omega[joint]);
    }
}

void inverse_dynamics_inner_loop() {
    // tau = D(theta)a_theta + C(theta, theta_dot)theta_dot + G(theta)

    float sintheta32 = sin(theta_motor[2] - theta_motor[1]);
    float costheta32 = cos(theta_motor[2] - theta_motor[1]);

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



//    t[1] = a_theta[1]*(p1-p3*sintheta32/*Omega[1]*/)
//            + Omega[1]*p3*costheta32*Omega[1]
//            - p4*9.81*sin(theta_motor[1]);
//
//    t[2] = a_theta[2]*(p2-p3*sintheta32)
//            - Omega[2]*p3*costheta32*Omega[2]
//            - p5*9.81*cos(theta_motor[2]);

```

```
        t[1] = p1*a_theta[1] - p3*sintheta32*a_theta[2]
                - Omega[2]*p3*costheta32*Omega[2]
                - p4*9.81*sin(theta_motor[1]);

        t[2] = -p3*sintheta32*a_theta[1] + p2*a_theta[2]
                + Omega[1]*p3*costheta32*Omega[1]
                - p5*9.81*cos(theta_motor[2]);

}



//This function calculates the forward kinematics of the manipulator given the
     motor positions

void forward_kinematics(float motor1, float motor2, float motor3){

    //The forward kinematics function uses the translational vector from the
    full DH matrix calculated in Robotica
    x_pos = 10*cos(motor1)*(cos(motor3)+sin(motor2));
    y_pos = 10*sin(motor1)*(cos(motor3)+sin(motor2));
    z_pos = 10*(1+cos(motor2)-sin(motor3));

}


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

typedef struct steptraj_s {
    long double b[7];
    long double a[7];
    long double xk[7];
    long double yk[7];
    float qd_old;
    float qddot_old;
    int size;
} steptraj_t;

steptraj_t trajectory = {1.0417207161648879e-11L,6.2503242969893271e-11L
    ,1.5625810742473319e-10L,2.0834414323297759e-10L,1.5625810742473319e-10L
    ,6.2503242969893271e-11L,1.0417207161648879e-11L,
                        1.0000000000000000e+00L,-5.8226600985221673e+00L
    ,1.4126404426217572e+01L,-1.8278500308471997e+01L,1.3303686800870629e+01L
    ,-5.1641897532443632e+00L,8.3525893381702754e-01L,
                        0,0,0,0,0,0,0,
                        0,0,0,0,0,0,0,
                        0,
                        0,
                        7};

// this function must be called every 1ms.
```

```
void implement_discrete_tf(steptraj_t *traj, float step, float *qd, float *qd_dot, float *qd_ddot) {
    int i = 0;

    traj->xk[0] = step;
    traj->yk[0] = traj->b[0]*traj->xk[0];
    for (i = 1;i<traj->size;i++) {
        traj->yk[0] = traj->yk[0] + traj->b[i]*traj->xk[i] - traj->a[i]*traj->yk[i];
    }

    for (i = (traj->size-1);i>0;i--) {
        traj->xk[i] = traj->xk[i-1];
        traj->yk[i] = traj->yk[i-1];
    }

    *qd = traj->yk[0];
    *qd_dot = (*qd - traj->qd_old)*1000;  //0.001 sample period
    *qd_ddot = (*qd_dot - traj->qddot_old)*1000;

    traj->qd_old = *qd;
    traj->qddot_old = *qd_dot;
}

// to call this function create a variable that steps to the new positions you want to go to, pass this var to step
// pass a reference to your qd variable your qd_dot variable and your qd_double_dot variable
// for example
//  implement_discrete_tf(&trajectory, mystep, &qd, &dot, &ddot);

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







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

//    theta_motor = {theta1motor, theta2motor, theta3motor};
    theta_motor[0] = theta1motor;
    theta_motor[1] = theta2motor;
    theta_motor[2] = theta3motor;

    filter_velocity();


//    position_d((mycount%2000)/1000.0);
```

```
//
//      feedforward_control();

//      float time = (mycount%2000)/1000.0;


    // 1) Calculate the desired trajectory
//      trajectory(time);

    if((mycount % 4000)==0) {
        mystep = 0.75;
    }

    if((mycount % 8000)==0) {
        mystep = 0.25;
    }

    float qd;
    float dot;
    float ddot;

    implement_discrete_tf(&trajectory, mystep, &qd, &dot, &ddot);

    int joint_idx = 0;
    for(joint_idx = 0; joint_idx < 3; joint_idx++) {
        theta_d[joint_idx] = qd;
        theta_dot_d[joint_idx] = dot;
        theta_ddot_d[joint_idx] = ddot;
    }

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


    // 3) Calculate the outer loop control to come up with values for a_theta_2 and a_theta_3
    inverse_dynamics_outer_loop();

    // 4) Calculate the inner loop control to find control effort to apply to joint 2 and 3
    inverse_dynamics_inner_loop();


    // 5) Calculate Lab 2 feed-forward control for joint 1 to find control effort to apply.
    feedforward_control(); // currently only active on joint 1

    *tau1 = t[0];
    *tau2 = t[1];
    *tau3 = t[2];

```

```
    // 6) Calculate friction compensation control effort given the velocities
    of joint 1,2, and 3
    friction_compensation();

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

    *tau1 = *tau1 + fric_on*u_fric[0];
    *tau2 = *tau2 + fric_on*u_fric[1];
    *tau3 = *tau3 + fric_on*u_fric[2];

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



    Simulink_PlotVar1 = qd;
    Simulink_PlotVar2 = theta_motor[0];
    Simulink_PlotVar3 = theta_motor[1];
    Simulink_PlotVar4 = theta_motor[2];


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

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

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

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

    }

    /*
     * Forward Kinematics
     *
     * based on motor angles
     */

    forward_kinematics(theta1motor, theta2motor, theta3motor);

    /*
     * Inverse Kinematics
     *
     * based on {x,y,z} pos calculated above
     */

    inverse_kinematics(x_pos, y_pos, z_pos);
```

```


    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
    }


    mycount++;
}

void printing(void){
//  serial_printf(&SerialA, "%.2f %.2f,%.2f   \n\r",printtheta1motor,printtheta2motor,printtheta3motor);
//  serial_printf(&SerialA, "x: %.2f,   y: %.2f,  z: %.2f   \n\r",x_pos,y_pos,z_pos);
//  //serial_printf(&SerialA, "Estimated IK solution: theta1: %.2f,   theta2: %.2f,  theta3: %.2f   \n\r",theta_1,theta_2,theta_3);
//    serial_printf(&SerialA, "Estimated IK solution: motor_theta1: %.2f, motor_theta2: %.2f,  motor_theta3: %.2f   \n\r",motor_theta_1,motor_theta_2,motor_theta_3);

//    serial_printf(&SerialA, "theta1motor:  %.2f \n\r",theta1motor);
//    serial_printf(&SerialA, "theta_motor[0]:  %.2f \n\r",theta_motor[0]);
}
```

Listing 2: lab3.c