ME 446 Lab 3: Inverse Dynamic Joint Control

Written By Tyler Smithline and Leo Jiang

ECE 489: Robot Dynamics and Control

Lab Section AB1

Introduction

The purpose of this lab is to implement friction compensation and inverse dynamics control on the CRS robot arm. By doing this, we saw how friction compensation can be used to negate the effects of friction and how inverse dynamics can provide improved control compared to other control algorithms, such as feedforward control. Finally, we investigated whether the inertial parameters of the CRS arm have been correctly identified, which is important for producing accurate inverse dynamic control. From this lab, we are now better equipped with the tools needed to succeed on the upcoming final project for this class.

Results

1. Include the final version of your C code.

The final C Code is included in the Appendix.

2. Include any MATLAB M-files you created (if any)

The transfer function we used for the smoothed trajectory is $\frac{30^4}{(s+30)^4}$. We plugged this into the provided filtersteptoC.m file to produce our desired smoothed step trajectory. No other MATLAB files were used for this lab.

3. In your own words, explain the inverse dynamics control algorithm.

Inverse dynamics joint control is a type of control algorithm that uses information on the mechanical properties of the robot arm system to determine the required torques needed to be applied to each joint of the robot arm for it to achieve a desired trajectory. Specifically, inverse dynamics control requires knowledge of the inertial properties of each joint and how these properties change with the joint configuration, θ. This is reflected in the inverse dynamics formula we use in the lab: $\tau = D(\theta)a_\theta + C\left(\theta, \dot{\theta}\right)\dot{\theta} + g(\theta)$. In this equation, $D(\theta)$ is the inertial matrix, which is multiplied by the PD feedforward control equation. This allows for the control algorithm to account for the inertial properties of the robotic arm system at the instantaneous joint configuration of the arm. An additional advantage of inverse dynamics is that it cancels the nonlinear parts of the dynamics, which linearizes the feedback from the arm and makes the system stable and easier to control than raw inverse dynamics.

4. Answer the questions found in the lab.

By comparing these two controllers can you say anything about the parameters we are using? Does Inverse Dynamics perform better? Have gravity effects been improved?

Figure 3 to 6 shows the comparison between the feedforward controller and inverse dynamics controller. The first half of the graphs are performance from feed-forward control and the second half of the graphs are performance from the inverse-dynamics control. We can see that both joint 2 and joint 3 have slightly lower steady-state error with inverse dynamics compared to the feed-forward control. This is largely because we incorporate gravity compensation with inverse dynamics. Additionally, we can also see smaller spikes in error when the arm starts moving from a standstill. Thus, we can conclude that inverse dynamic control performs better overall. The parameters $(D(\theta), C(\theta, \dot{\theta}), G(\theta))$ we used for the robot arms are estimations, so they don’t perfectly model the system but still provide an improvement compared to feed forward control. Better parameterization of the system would likely result in lower steady state error.

5. Trajectory Response and Error Plots

These plots can be found in the Appendix.

6. Your observations about the inverse dynamic controller and the system parameters used.

These observations are shared in the answers to the lab questions above.

Appendix

Trajectory Response and Error Plots:

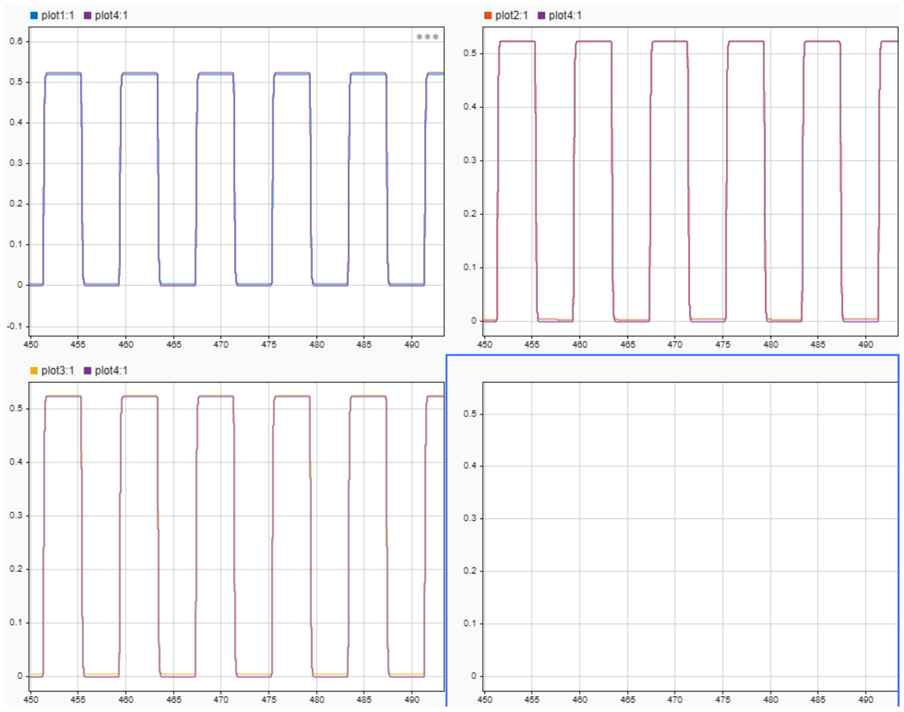

Figure 1. Smooth Step Response Trajectory; Theta Desired vs Actual for J1, J2, J3

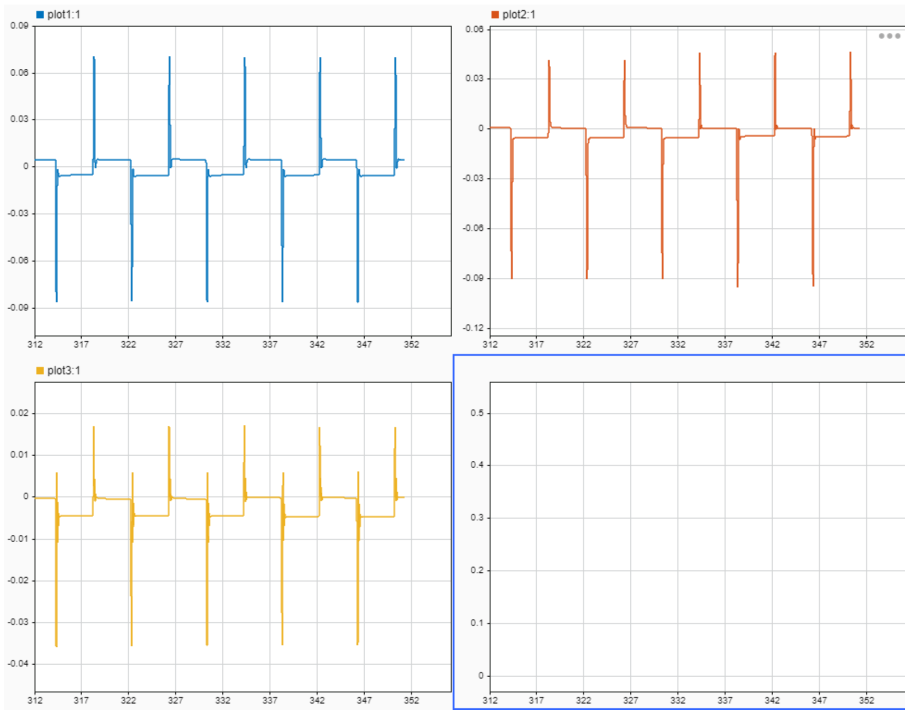

Figure 2. Errors for Smoothed Step Trajectory for J1, J2, J3

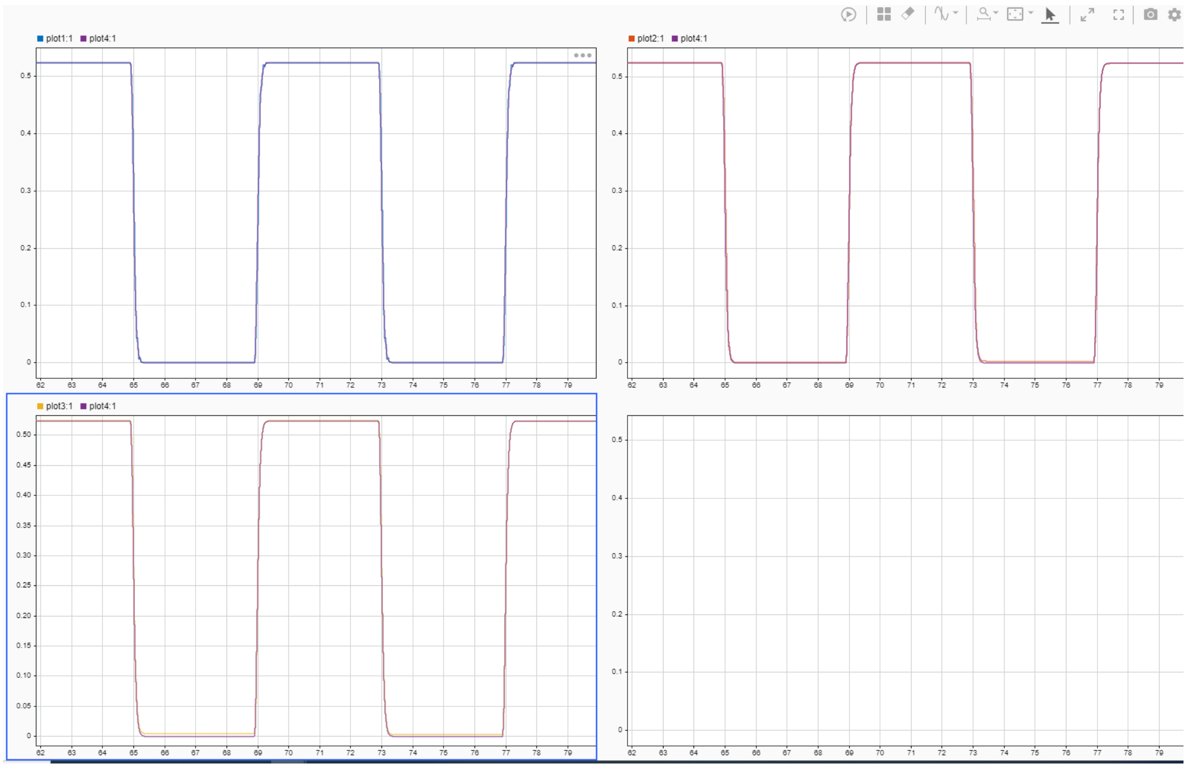

Figure 3. Smoothed Step Response Trajectory: Feedforward (First 10 seconds) vs Inverse dynamics (Remaining Time); Theta Desired vs Actual for J1, J2, J3

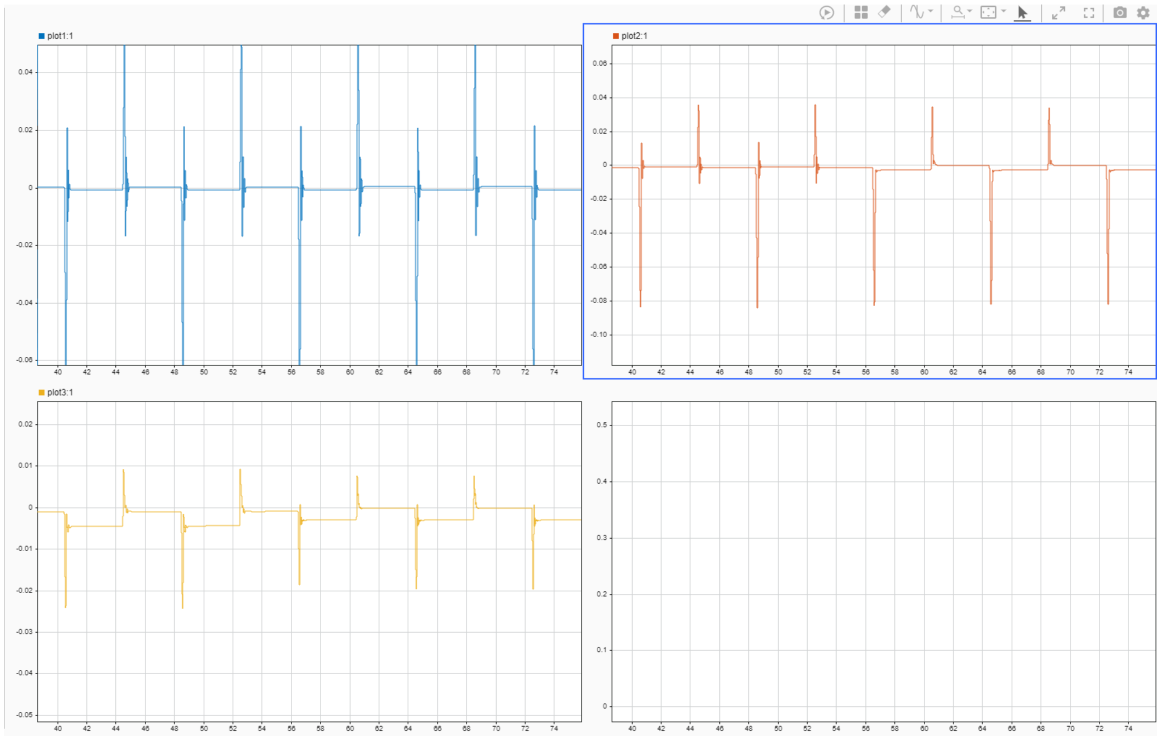

Figure 4. Errors Smoothed Step Trajectory: Feedforward (First 10 seconds) vs Inverse Dynamics (Remaining Time) for J1, J2, J3

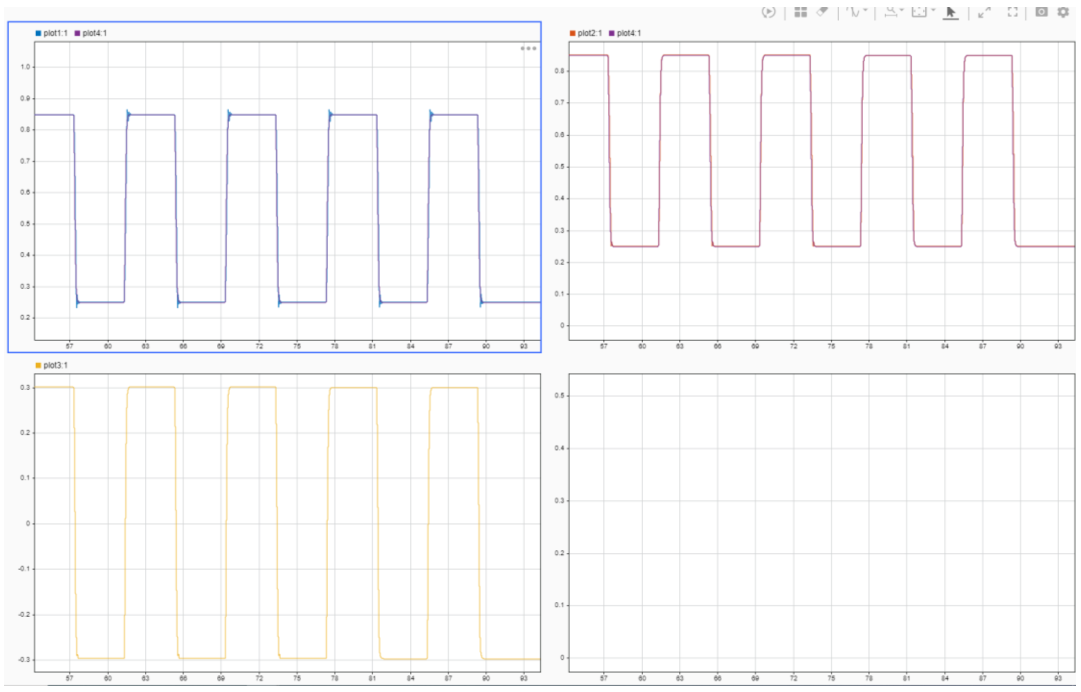

Figure 5. New Trajectory: Feedforward (First 10 sec) vs Inverse dynamics (Remaining Time); Theta Desired vs Actual for J1, J2, J3

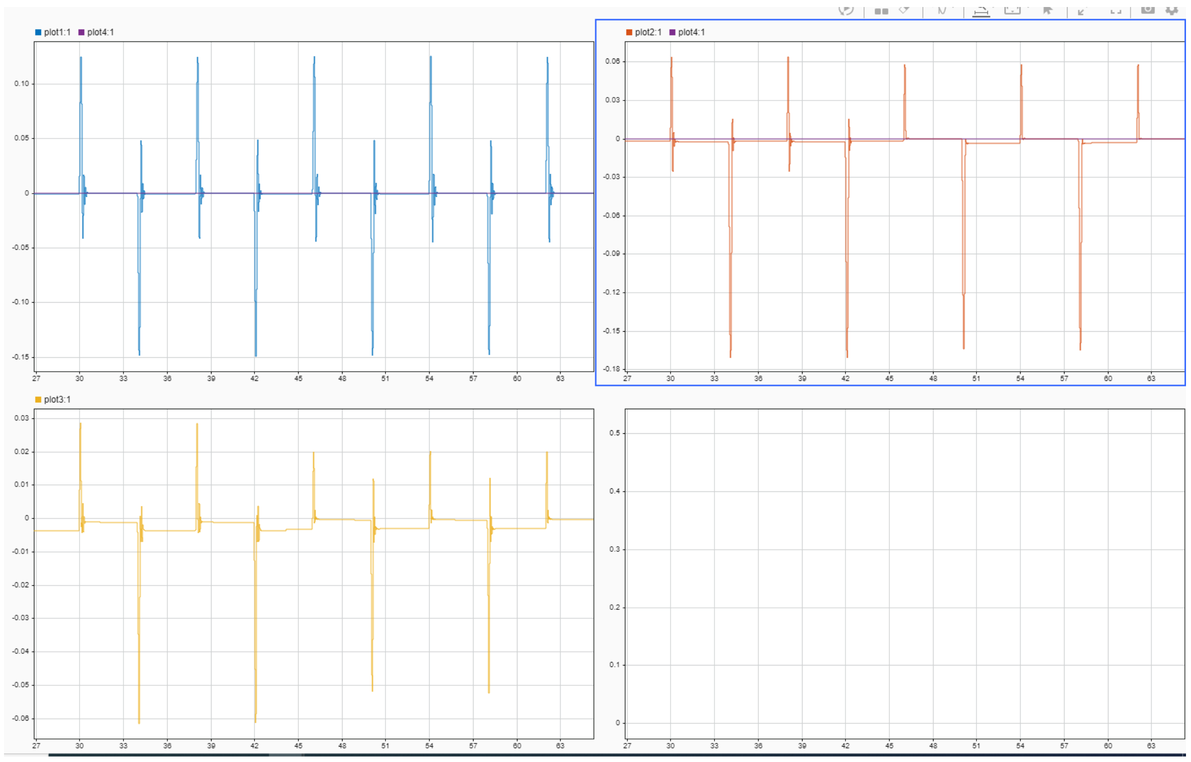

Figure 6. Errors for New Trajectory: Feedforward (First 15 sec) vs Inverse dynamics (Remaining Time) for J1, J2, J3

Final C Code:

```c
#include "math.h"
#include "F28335Serial.h"

#define PI          3.1415926535897932384626433832795
#define TWOPI       6.283185307179586476925286766559
#define HALFPI      1.5707963267948966192313216916398
#define GRAV        9.81

// 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 = -0.4454;
float offset_Enc3_rad = 0.2436;
//float offset_Enc2_rad = 0.0;
//float offset_Enc3_rad = 0.0;


// Your global varialbes.

long mycount = 0;

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

#pragma DATA_SECTION(theta1array, ".my_arrs")
float theta1array[100];
#pragma DATA_SECTION(theta2array, ".my_arrs")
float theta2array[100];


long arrayindex = 0;
int UARTprint = 0;

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

float motortheta1 = 0;
float motortheta2 = 0;
float motortheta3 = 0;

float theta1 = 0;
float theta2 = 0;
float theta3 = 0;

//float x = 0;
float y = 0;
float z = 0;

// Initialize values for all 3 joint motors
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 theta1d = 0;
float theta2d = 0;
float theta3d = 0;

float thetadot = 0.0;

float kp1 = 300;
float kp2 = 4000;
float kp3 = 18000;
float kd1 = 1.6;
float kd2 = 100;
float kd3 = 150;

float ptau1 = 0;
float ptau2 = 0;
float ptau3 = 0;

float err_old1 = 0;
float err_old2 = 0;
float err_old3 = 0;

float err1 = 0;
float err2 = 0;
float err3 = 0;

float err_dot1 = 0.0;
float err_dot2 = 0.0;
float err_dot3 = 0.0;

float Ikold1 = 0;
float Ikold2 = 0;
float Ikold3 = 0;

float Ik1 = 0;
float Ik2 = 0;
float Ik3 = 0;

float ki1 = 0;
float ki2 = 0;
float ki3 = 0;
//float ki2 = 0;
//float ki3 = 0;

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

float threshold1 = 0.01;
float threshold2 = 0.03;
float threshold3 = 0.02;

float thetaddot = 0;

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

float a2 = 0;
float a3 = 0;

float a0 = 0;
float a1 = 0;

float t = 0.0;
int home_traj = 0;

float x = 0.4;

float Viscous_positive1 = 0.22;
float Viscous_negative1 = 0.22;
float Coulomb_positive1 = 0.3637;
float Coulomb_negative1 = -0.2948;

float Viscous_positive2 = 0.2500;
float Viscous_negative2 = 0.287;
float Coulomb_positive2 = 0.45;
float Coulomb_negative2 = -0.47;

float Viscous_positive3 = 0.1922;
float Viscous_negative3 = 0.2132;
float Coulomb_positive3 = 0.440;
float Coulomb_negative3 = -0.440;

float u_fric1 = 0.0;
float u_fric2 = 0.0;
float u_fric3 = 0.0;

float v_co_1 = 3.6;
float v_co_2 = 3.6;
float v_co_3 = 3.3;

float p1 = 0.03;
float p2 = 0.0128;
float p3 = 0.0076;
float p4 = 0.0753;
float p5 = 0.0298;

float at2 = 0.0;
float at3 = 0.0;
float qd = 0;
float qddot = 0;
float qdddot = 0;

float sintheta1=0;
float sintheta32 = 0;
float costheta32 = 0;

float step1 = 0;
float step2 = 0;

float switch_control = 0;

float kp1f = 300;
float kp2f = 300;
float kp3f = 160;
float kd1f = 1.6;
float kd2f = 1.9;
float kd3f = 1.9;

float theta3dot = 0;
float theta3ddot = 0;

typedef struct steptraj_s {
    long double b[5];
```

```c
    long double a[5];
    long double xk[5];
    long double yk[5];
    float qd_old;
    float qddot_old;
    int size;
} steptraj_t;

steptraj_t trajectory = {4.7698076658265394e-08L,1.9079230663306158e-07L,2.8618845994959235e-07L,1.9079230663306158e-07L,4.7698076658265394e-08L,
                        1.0000000000000000e+00L,-3.8817733990147785e+00L,5.6505617704870286e+00L,-3.6557000616943993e+00L,8.8691245339137526e-01L,
                        0,0,0,0,0,
                        0,0,0,0,0,
                        0,
                        0,
                        5};
steptraj_t trajectory2 = {4.7698076658265394e-08L,1.9079230663306158e-07L,2.8618845994959235e-07L,1.9079230663306158e-07L,4.7698076658265394e-08L,
                        1.0000000000000000e+00L,-3.8817733990147785e+00L,5.6505617704870286e+00L,-3.6557000616943993e+00L,8.8691245339137526e-01L,
                        0,0,0,0,0,
                        0,0,0,0,0,
                        0,
                        0,
                        5};
// 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;
}

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

    //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++;
        }

    }

    if ((mycount%500)==0) {
        UARTprint = 1;
        GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1; // Blink LED on Control Card
        GpioDataRegs.GPBTOGGLE.bit.GPIO60 = 1; // Blink LED on Emergency Stop Box
    }

    // Print motor angles from encoders
    printtheta1motor = theta1motor;
    printtheta2motor = theta2motor;
    printtheta3motor = theta3motor;

//    x = (127.0*cos(theta1motor)*(cos(theta3motor) + sin(theta2motor)))/500.0;
//    y = (127.0*sin(theta1motor)*(cos(theta3motor) + sin(theta2motor)))/500.0;
//    z = (127.0*cos(theta2motor))/500.0 - (127.0*sin(theta3motor))/500.0 + 127.0/500.0;
    //t = (mycount%(6283*4))*0.001;
//    t = mycount*0.001;

//    x = 0.3;

    // Calculated motor thetas with inverse kinematics
    motortheta1 = atan2(y,x);
    motortheta2 = 1.570796 - 1.0*atan2(1.0*z - 0.254, sqrt(x*x + y*y)) - 1.0*atan2(2.0*sqrt(0.064516 - 0.25*x*x - 0.25*y*y - 0.25*(z - 0.254)*(z-0.254)), sqrt((z - 0.254)*(z - 0.254) + x*x + y*y));
    motortheta3 = 1.0*atan2(2.0*sqrt(0.064516 - 0.25*x*x - 0.25*y*y - 0.25*(z - 0.254)*(z-0.254)), sqrt((z - 0.254)*(z-0.254) + x*x + y*y)) - 1.0*atan2(1.0*z - 0.254, sqrt(x*x + y*y));

    // DH Thetas
    theta1 = motortheta1;
    theta2 = motortheta2 - PI/2;
    theta3 = motortheta3 - motortheta2 + PI/2;

//    theta1d = motortheta1;
//    theta2d = motortheta2;
//    theta3d = motortheta3;

    Omega1 = (theta1motor - Theta1_old)/0.001;
    Omega1 = (Omega1 + Omega1_old1 + Omega1_old2)/3.0;
    Theta1_old = theta1motor;
    //order matters here. Because we are saving the old value first before overriding it
    Omega1_old2 = Omega1_old1;
    Omega1_old1 = Omega1;

    Omega2 = (theta2motor - Theta2_old)/0.001;
    Omega2 = (Omega2 + Omega2_old1 + Omega2_old2)/3.0;
    Theta2_old = theta2motor;
    //order matters here. Because we are saving the old value first before overriding it
    Omega2_old2 = Omega2_old1;
    Omega2_old1 = Omega2;

    Omega3 = (theta3motor - Theta3_old)/0.001;
    Omega3 = (Omega3 + Omega3_old1 + Omega3_old2)/3.0;
    Theta3_old = theta3motor;
    //order matters here. Because we are saving the old value first before overriding it
    Omega3_old2 = Omega3_old1;
    Omega3_old1 = Omega3;


    t = (mycount%2000)*0.001;
//
```

```
//    y = 0.1*sin(3*t+PI/4);
//    z = 0.1*sin(4*t);
//    x = 0.2;

//    Step input trajectory
//    if((mycount%4000)==0) {
//        if(step1 > 0.1) {
//            step1 = 0;
//        } else {
//            step1 = PI/6;
//        }
//    }
//    new trajectory
    if((mycount%4000)==0) {
        if(step1 > 0.26) {
            step1 = 0.25;
        } else {
            step1 = 0.85;
        }
    }
    if((mycount%4000)==0) {
        if(step2 < 0.29) {
            step2 = 0.3;
        } else {
            step2 = -0.3;
        }
    }


    implement_discrete_tf(&trajectory, step1, &qd, &qddot, &qdddot);
    theta1d = qd;
    thetadot = qddot;
    thetaddot = qdddot;
    implement_discrete_tf(&trajectory2, step2, &qd, &qddot, &qdddot);
    theta3d = qd;
    theta3dot = qddot;
    theta3ddot = qdddot;

    // cubic trajectory oscillating between two points
//    if (t < 1) {
//        home_traj = 0;
//    } else {
//        home_traj = 1;
//    }
//
//    if(home_traj) {
//        a3 = 1;
//        a2 = -4.5;
//        a1 = 6;
//        a0 = -2;
//        theta1d = a0+a1*t+a2*pow(t,2)+a3*pow(t,3);
//        thetadot = a1+2*a2*t+3*a3*pow(t,2);
//        thetaddot = 2*a2+6*a3*t;
//    } else {
//        a2 = 1.5;
//        a3 = -1;
//        theta1d = a2*pow(t,2)+a3*pow(t,3);
//        thetadot = 2*a2*t+3*a3*pow(t,2);
//        thetaddot = 2*a2+6*a3*t;
//    }


//    if((mycount%1000)==0) {
//        if(theta1d > 0.1) {
//            theta1d = a2*t^2+a3*t.^3;;
//        } else {
//            theta1d = PI/6;
//        }
//    }
//
//    if((mycount%1000)==0) {
//        if(theta2d > 0.1) {
//            theta2d = 0;
//        } else {
//            theta2d = PI/6;
//        }
//    }
//
//    if((mycount%1000)==0) {
//        if(theta3d > 0.1) {
//            theta3d = 0;
//        } else {
//            theta3d = PI/6;
//        }
//    }

//    err1 = theta1d-theta1motor;
//    err2 = theta1d-theta2motor;
//    err3 = theta1d-theta3motor;
//    err_dot1 = thetadot - Omega1;
//    err_dot2 = thetadot - Omega2;
//    err_dot3 = thetadot - Omega3;

    err1 = theta1d-theta1motor;
    err2 = theta1d-theta2motor;
    err3 = theta3d-theta3motor;

    err_dot1 = thetadot - Omega1;
    err_dot2 = thetadot - Omega2;
    err_dot3 = theta3dot - Omega3;

    Ik1 = Ikold1+(err1-err_old1)/2.0*0.001;
    Ik2 = Ikold2+(err2-err_old2)/2.0*0.001;
    Ik3 = Ikold3+(err3-err_old3)/2.0*0.001;

    //friction
    if (Omega1 > 0.1) {
     u_fric1 = Viscous_positive1*Omega1 + Coulomb_positive1;
    } else if (Omega1 < -0.1) {
     u_fric1 = Viscous_negative1*Omega1 + Coulomb_negative1;
    } else {
     u_fric1 = 3.6*Omega1;
    }

    if (Omega2 > 0.05) {
     u_fric2 = Viscous_positive2*Omega2 + Coulomb_positive2;
    } else if (Omega2 < -0.05) {
     u_fric2 = Viscous_negative2*Omega2 + Coulomb_negative2;
    } else {
     u_fric2 = v_co_2*Omega2;
    }

    if (Omega3 > 0.05) {
     u_fric3 = Viscous_positive3*Omega3 + Coulomb_positive3;
    } else if (Omega3 < -0.05) {
     u_fric3 = Viscous_negative3*Omega3 + Coulomb_negative3;
    } else {
     u_fric3 = v_co_3*Omega3;
    }

```

```
//    *tau1 = u_fric1;
//    *tau2 = u_fric2;
//    *tau3 = u_fric3;


//    // feed forward
//    if (fabs(err1) < threshold1) {
//        ptau1 = kp1*(err1) - kd1*Omega1 + ki1*Ik1+thetaddot*J1;
//    }  else {
//        Ik1 = 0;
//        ptau1 = kp1*(err1) - kd1*Omega1+thetaddot*J1;
//    }
//
//    if (fabs(err2) < threshold2) {
//        ptau2 = kp2*(err2) - kd2*Omega2 + ki2*Ik2+thetaddot*J2;
//    }  else {
//        Ik2 = 0;
//        ptau2 = kp2*(err2) - kd2*Omega2+thetaddot*J2;
//    }
//
//    if (fabs(err3) < threshold3) {
//        ptau3 = kp3*(err3) - kd3*Omega3 + ki3*Ik3+thetaddot*J3;
//    }  else {
//        Ik3 = 0;
//        ptau3 = kp3*(err3) - kd3*Omega3+thetaddot*J3;
//    }


//    new traj
    if (switch_control == 1) {
        //inverse dynamic
        ptau1 = kp1*(err1) + kd1*err_dot1+thetaddot*J1;

        at2 = kp2*(err2) + kd2*err_dot2+thetaddot;

        at3 = kp3*(err3) + kd3*err_dot3+theta3ddot;

        sintheta32 = sin(theta3motor-theta2motor);
        costheta32 = cos(theta3motor-theta2motor);
        ptau2 = p1*at2-(p3*sintheta32)*at3 - p3*costheta32*Omega3*Omega3 - p4*9.8*sin(theta2motor);
        ptau3 = -p3*sintheta32*at2+p2*at3 - p3*costheta32*Omega2*Omega2 - p5*9.8*sin(theta3motor);
    } else {
        // feed forward
        ptau1 = kp1f*(err1) + kd1f*err_dot1+thetaddot*J1;
        ptau2 = kp2f*(err2) + kd2f*err_dot2+thetaddot*J2;
        ptau3 = kp3f*(err3) + kd3f*err_dot3+theta3ddot*J3;
    }
//    step traj
//    if (switch_control == 1) {
//        //inverse dynamic
//        ptau1 = kp1*(err1) + kd1*err_dot1+thetaddot*J1;
//
//        at2 = kp2*(err2) + kd2*err_dot2+thetaddot;
//
//        at3 = kp3*(err3) + kd3*err_dot3+thetaddot;
//
//        sintheta32 = sin(theta3motor-theta2motor);
//        costheta32 = cos(theta3motor-theta2motor);
//        ptau2 = p1*at2-(p3*sintheta32)*at3 - p3*costheta32*Omega3*Omega3 - p4*9.8*sin(theta2motor);
//        ptau3 = -p3*sintheta32*at2+p2*at3 - p3*costheta32*Omega2*Omega2 - p5*9.8*sin(theta3motor);
//    } else {
```

```
//        // feed forward
//        ptau1 = kp1f*(err1) + kd1f*err_dot1+thetaddot*J1;
//        ptau2 = kp2f*(err2) + kd2f*err_dot2+thetaddot*J2;
//        ptau3 = kp3f*(err3) + kd3f*err_dot3+thetaddot*J3;
//    }

    ////    fun
//    ptau1 = kp1*(err1) - kd1*Omega1;
//    ptau2 = kp2*(err2) - kd2*Omega2;
//    ptau3 = kp3*(err3) - kd3*Omega3;
    // Saturation of torque values
//    if (ptau1 > 5) {
//        ptau1 = 5;
//        Ik1 = Ikold1;
//    } else if (ptau1 < -5) {
//        ptau1 = -5;
//        Ik1 = Ikold1;
//    }
//
//    if (ptau2 > 5) {
//        ptau2 = 5;
//        Ik2 = Ikold2;
//    } else if (ptau2 < -5) {
//        ptau2 = -5;
//        Ik2 = Ikold2;
//    }
//
//    if (ptau3 > 5) {
//        ptau3 = 5;
//        Ik3 = Ikold3;
//    } else if (ptau3 < -5) {
//        ptau3 = -5;
//        Ik3 = Ikold3;
//    }

    *tau1 = ptau1;
    *tau2 = ptau2;
    *tau3 = ptau3;

    Ikold1 = Ik1;
    Ikold2 = Ik2;
    Ikold3 = Ik3;

    err_old1 = err1;
    err_old2 = err2;
    err_old3 = err3;

    // When we want to plot error
    Simulink_PlotVar1 = err1;
    Simulink_PlotVar2 = err2;
    Simulink_PlotVar3 = err3;
//    Simulink_PlotVar4 = theta1d;

    // When we want to plot actual vs desired angles
//    Simulink_PlotVar1 = theta1motor;
//    Simulink_PlotVar2 = theta2motor;
//    Simulink_PlotVar3 = theta3motor;
//    Simulink_PlotVar4 = theta1d;

    mycount++;

}

void printing(void){
```

```c
    serial_printf(&SerialA, "%.2f %.2f %.2f | %.2f %.2f %.2f | %.2f %.2f %.2f | %.2f %.2f %.2f \n\r",printtheta1motor*180/PI,printtheta2motor*180/PI,printtheta3motor*180/PI, x,y,z, motortheta1*180/PI, motortheta2*180/PI, motortheta3*180/PI, theta1*180/PI, theta2*180/PI, theta3*180/PI);
//    serial_printf(&SerialA, "tau2: %.2f tau3: %.2f \n\r", ptau2, ptau3);
}

```

MATLAB Code:

```matlab
function filtersteptoC(ctf)
% format filtersteptoC(ctf)  ctf is the continuous transfer this C code will approximate
% this function generates C code that implements a discrete transfer funtion.
% the C function created must be called every .001 seconds.
%dtf = c2d(ctf,.001);
dtf = c2d(ctf,.001,'tustin');
b = dtf.num{1};
a = dtf.den{1};
numcoeffs = length(a);
t = 0:.001:4;
[stepy] = step(dtf,t);
figure(1);
plot(t,stepy);
title('qd');
dy = filter([1 -1],[.001 0],stepy);
figure(2);
plot(t,dy);
title('qd_dot');
ddy = filter([1 -1],[.001 0],dy);
figure(3);
plot(t,ddy);
title('qd_ddot');
fprintf(1,'typedef struct steptraj_s {\n');
fprintf(1,'     long double b[%d];\n',numcoeffs);
fprintf(1,'     long double a[%d];\n',numcoeffs);
fprintf(1,'    long double xk[%d];\n',numcoeffs);
fprintf(1,'    long double yk[%d];\n',numcoeffs);
fprintf(1,'    float qd_old;\n');
fprintf(1,'    float qddot_old;\n');
fprintf(1,'    int size;\n')
fprintf(1,'} steptraj_t;\n');
fprintf(1,'\n');
fprintf(1,'steptraj_t trajectory = {');
for i=1:numcoeffs
   fprintf(1,'%10.16eL,',b(i));
end
fprintf(1,'\n\t\t\t\t\t\t');
for i=1:numcoeffs
   fprintf(1,'%10.16eL,',a(i));
end
fprintf(1,'\n\t\t\t\t\t\t');
for i=1:numcoeffs
   fprintf(1,'0,');
end
fprintf(1,'\n\t\t\t\t\t\t');
for i=1:numcoeffs
   fprintf(1,'0,');
end
fprintf(1,'\n\t\t\t\t\t\t');
fprintf(1,'0,\n\t\t\t\t\t\t');
fprintf(1,'0,\n\t\t\t\t\t\t');
fprintf(1,'%d};\n',numcoeffs);
fprintf(1,'\n');
fprintf(1,'// this function must be called every 1ms.\n');
fprintf(1,'void implement_discrete_tf(steptraj_t *traj, float step, float *qd, float *qd_dot, float *qd_ddot) {\n');
fprintf(1,'     int i = 0;\n');
fprintf(1,'\n');
fprintf(1,'     traj->xk[0] = step;\n');
fprintf(1,'     traj->yk[0] = traj->b[0]*traj->xk[0];\n');
fprintf(1,'     for (i = 1;i<traj->size;i++) {\n');
fprintf(1,'               traj->yk[0] = traj->yk[0] + traj->b[i]*traj->xk[i] - traj->a[i]*traj->yk[i];\n');
fprintf(1,'     }\n');
fprintf(1,'\n');
fprintf(1,'     for (i = (traj->size-1);i>0;i--) {\n');
fprintf(1,'               traj->xk[i] = traj->xk[i-1];\n');
fprintf(1,'               traj->yk[i] = traj->yk[i-1];\n');
fprintf(1,'     }\n');
fprintf(1,'\n');
fprintf(1,'     *qd = traj->yk[0];\n');
fprintf(1,'     *qd_dot = (*qd - traj->qd_old)*1000;  //0.001 sample period\n');
fprintf(1,'     *qd_ddot = (*qd_dot - traj->qddot_old)*1000;\n');
fprintf(1,'\n');
fprintf(1,'     traj->qd_old = *qd;\n');
fprintf(1,'     traj->qddot_old = *qd_dot;\n');
fprintf(1,'}\n');
fprintf(1,'\n');
fprintf(1,'// to call this function create a variable that steps to the new positions you want to go to, pass this var to step\n');
fprintf(1,'// pass a reference to your qd variable your qd_dot variable and your qd_double_dot variable\n');
fprintf(1,'// for example\n');
fprintf(1,'//   implement_discrete_tf(&trajectory, mystep, &qd, &dot, &ddot);\n');

```