Lab 4: Task Space Control

Caleb Albers & Karun Koppula

May 11, 2018

A lab4.c code

```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 = -0.37;
//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;

//This global variable is used to control whether or not friction compensation is used in the control effort.
float fric_on = 1.0;

// 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;


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

// velocity filtering
float Theta_old[3] = {0,0,0};
```

```c
float Omega_old1[3] = {0,0,0};
float Omega_old2[3] = {0,0,0};
float Omega[3] = {0,0,0};

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

// 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

float mystep = 0.25;


//Task Space Globals
float KP_task[3] = {2.0, 1.0, 2.0};
float KD_task[3] = {0.04, 0.025, 0.025};

float Velocity[3] = {0,0,0};
float Velocity_old1[3] = {0,0,0};
float Velocity_old2[3] = {0,0,0};

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

float Position[3] = {0,0,0};
float Position_old[3] = {0,0,0};

float Position_d[3] = {10,10,10};

//Force Coordinate Frame Rotation
float Rot[3] = {0,0,0};


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];
    }
}

```

```c

/* This function uses the non-linear model for friction to calculate the required control effort
 * needed to compensate at the current joint. It works by taking the current velocity Omega and
 * multiplying it by a Viscous friction gain, and then adding that to a Coulomb friction offset.
 * The Viscous gains and Coulomb offsets are unique for every joint, as well as unique for the
 * forward and reverse direction; They were calculated in Lab 3.
 *
 * The friction compensation can be visualized as follows:
 *
 *                              -----------
 *                             |       /   |
 *                             |      /    |
 * friction compensation:      |     |     |
 *                             |    /      |
 *                             |   /       |
 *                             |-----------|
 * If the velocity is between a certain threshold, a slope of 3.6 is used for the calculation.
 */

void friction_compensation() {
    int i = 0;
    // iterate over all three joints
    for(i = 0; i < 3; i++) {

        // is the velocity greater than the positive minimum velocity?
        if (Omega[i] > minimum_velocity[i]) {
            u_fric[i] = Viscous_positive[i]*Omega[i] + Coulomb_positive[i];
        } // Otherwise, is it lesser than the negative minimum velocity?
        else if (Omega[i] < -minimum_velocity[i]) {
            u_fric[i] = Viscous_negative[i]*Omega[i] + Coulomb_negative[i];
        } // Otherwise, it is between the minimums. Apply the default effort
        else {
            u_fric[i] = slope_between_minimums[i]*Omega[i];
        }
    }
}

//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
    Position[0] = 10*cos(motor1)*(cos(motor3)+sin(motor2));
    Position[1] = 10*sin(motor1)*(cos(motor3)+sin(motor2));
    Position[2] = 10*(1+cos(motor2)-sin(motor3));
```

```
}

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

float position_start[3] = {10, -2.5, 10};
float delta[3] = {5, 5, 0};
float t_total = 2.0;
float t_start = 0;

/*
 * This function is called every 1 ms in the main loop and generates a straight-line trajectory.
 * It works by taking a starting position (position_start) and then "stepping" every millisecond.
 * The distance vector that describes the entire 3D trajectory is given by delta. This means:
 * - the start position is position_start
 * - the end position is position_start + delta
 *
 * The variable t_total describes the total time the trajectory should take in seconds.
 * The variable t is a float that shows the current time in seconds (effectively giving fractions of a second)
 *
 * A float that describes the current progress between zero and t_total is generated by subtracting
 * the value t_start from t and then dividing that by t_total. The variable t_start is continually
 * updated to the current value t every t_start seconds.
 *
 * Once the fraction of time is calculated, the function calculates the new trajectory point by taking
 * the starting position and adding the delta vector multiplied by the fraction of time. This yields
 * an interpolated straight-line trajectory every millisecond.
 *
 * Once the trajectory has reached the ending position, the function sets t_start = t, sets the
 * new starting position to be the current position, and flips the sign of the delta vector.
 * This allows the function to calculate a trajectory going the opposite way back to
 * the initial starting position.
 *
 */
void task_space_trajectory(float t) {
    // calculate the fraction of total travel
    float time = (t-t_start)/t_total;
    int i = 0;
    // iterate over all three directions of travel
    for (i = 0; i < 3; i++) {
```

```
        // multiply the direction vector by the fraction of total fraction, and
        // add that to the starting position for this travel direction
        Position_d[i] = delta[i]*time + position_start[i];
    }

    // if the final position has been met, note the current time, flip the direction
    // of travel to go backward, and set the starting position as the current position.
    if(t-t_start == t_total) {
        t_start = t;

        int i = 0;
        for(i = 0; i < 3; i++) {
            delta[i] = -delta[i]; // reverse travel direction
            position_start[i] = Position_d[i]; // set new starting point to current position
        }
    }
}

/*
 * This function is called every millisecond. It calculates
 * the velocity in the task space by utilizing an infinite impulse response (IIR) filter.
 *
 * The implementation is the same as was described in Lab 2, where
 * prior filtered velocities are stored and averaged along with the current
 * calculated velocity in order to derive at a filtered velocity output.
 */
void filter_velocity_task() {
    int i;
    // Iterate over all three directions
    for(i = 0; i < 3; i++) {
        // calculate the descritized velocity by getting the difference
        // between the current and previous position and dividing by delta_t,
        // which is 0.001 since this function is called every millisecond.
        Velocity[i] = (Position[i] - Position_old[i])/0.001;

        // Grab the average of the descritized velocity and the two prior
        // filtered velocities
        Velocity[i] = (Velocity[i] + Velocity_old1[i] + Velocity_old2[i])/3.0;

        // Save the current position as the prior position, and save the
        // filtered velocities for later use next time the function is called.
        Position_old[i] = Position[i];
        Velocity_old2[i] = Velocity_old1[i];
        Velocity_old1[i] = Velocity[i]; // make sure to save the _filtered_ velocity
        // as that is what makes this an IIR filter!
    }
```

```
}



/*
 * The next section calculates the control effort prior to adding in friction compensation.
 * It is calculated by taking the transpose of the Jacobian and matrix multiplying it by the
 * vector F(x,y,z). To expand on this, the vector F describes the PD control effort
 * as a vector in the World frame, transformed into the N frame by multiplying R_WN.
 *
 * More detail as to the mathematics of our implementation is described below.
 */
void task_space_control() {

    // save the sin and cos calculation values in order to reduce computation time
    float sin_M1 = sin(theta_motor[0]);
    float cos_M1 = cos(theta_motor[0]);
    float sin_M2 = sin(theta_motor[1]);
    float cos_M2 = cos(theta_motor[1]);
    float sin_M3 = sin(theta_motor[2]);
    float cos_M3 = cos(theta_motor[2]);

    // Calculate the transpose of the Jacobian for the CRS robot
    float J_t[3][3] = {{-10*sin_M1*(cos_M3+sin_M2),  10*cos_M1*(cos_M3+sin_M2),  0},
                       { 10*cos_M1*(cos_M2-sin_M3),  10*sin_M1*(cos_M2-sin_M3),  -10*(cos_M3+sin_M2)},
                       {-10*cos_M1*sin_M3,           -10*sin_M1*sin_M3,          -10*cos_M3}};

    /*
     * Rot is a 3D vector specifying the direction that specifies the weak axis.
     * We are again saving sin and cos calculations to reduce compute time.
     */
    float sin_x = sin(Rot[0]);
    float cos_x = cos(Rot[0]);
    float sin_y = sin(Rot[1]);
    float cos_y = cos(Rot[1]);
    float sin_z = sin(Rot[2]);
    float cos_z = cos(Rot[2]);

    /*
     * A rotation matrix R_N^W describes the transformation from the world frame of the robot to the
     * N frame, which describes the aforementioned weak axis. It is a standard rotation matrix
```

```
     * calculated by first taking a rotation theta_z about the z-axis, then a rotation theta_x about
     * the x-axis, and finally a rotation theta_y about the y-axis.
     */

    float R_WN[3][3] = { {cos_z*cos_y - sin_z*sin_x*sin_y, -sin_z*cos_x, cos_z*sin_y+sin_z*sin_x*cos_y},
                         {sin_z*cos_y + cos_z*sin_x*sin_y, cos_z*cos_x, sin_z*sin_y-cos_z*sin_x*cos_y},
                         {-cos_x*sin_y, sin_x, cos_x*cos_y}};

    /*
     * The inverse of R_WN is R_NW, which is a rotation matrix describing the transformation from the
     * N frame back to the World frame W. Due to the property of matrices in SO(3), R^-1 = R^t (the
     * inverse of the matrix is equal to the transpose). Thus, R_NW is calculated by taking the
     * transpose of R_WN.
     */
    float R_NW[3][3] = {{R_WN[0][0], R_WN[1][0], R_WN[2][0]},
                        {R_WN[0][1], R_WN[1][1], R_WN[2][1]},
                        {R_WN[0][2], R_WN[1][2], R_WN[2][2]}};


    /*
     * In order to make the calculations easier, we split this up into two distinct efforts:
     *  -- f_p(x,y,z) describes the contribution from the proportional control law
     *  -- f_d(x,y,z) describes the contribution from the derivative control law
     * Each of these 3-vectors exist in the N frame by taking the Proportional and Derivative
     * gains, multiplying them by the difference between the desired and current positions, and the
     * desired and current velocities, respectively (all in the N frame).
     *
     * Effectively, f_p(x,y,z) is:
     *
     *                      |` KP_x,N * (x^d_N - x_n) `|
     * f_p = J^T * R_NW * |  KP_y,N * (y^d_N - y_n)  |
     *                      |_ KP_z,N * (z^d_N - z_n) _|
     *
     * Likewise, f_d(x,y,z) is:
     *
     *                      |` KD_x,N * (x_dot^d_N - x_dot_n) `|
     * f_d = J^T * R_NW * |  KD_y,N * (y_dot^d_N - y_dot_n)  |
     *                      |_ KD_z,N * (z_dot^d_N - z_dot_n) _|
     *
     * Where KP_N(x,y,z) is a vector holding the proportional gains in the x,y,z axes of the N frame,
```

```
 * KD_N(x,y,z) is a vector holding the derivative gains in the x,y,z axes of the N frame,
 * x^d_N, y^d_N, z^d_N describe the desired position in the N frame, and x_n,y_n,z_n is the current
 * position in the N frame. Likewise, x_dot^d_N, y_dot^d_N, z_dot^d_N are the desired velocities
 * in the N frame, and x_dot_n, y_dot_n, z_dot_n are the current velocities in the N frame.
 *
 * NOTE: the current velocities are calculated by using the velocity filtering function prior to
 * calling this function. Both the velocities and positions are given in the World frame initially,
 * and then transformed into the N frame by using the rotation matrix R_NW.
 *
 * These proportional and derivative control efforts are summed together into the vector F_N,
 * in the N frame.
 */

// Initialize the force components and vectors.
float f_p[3] = {0,0,0};
float f_d[3] = {0,0,0};
float F_N[3] = {0,0,0};
float F_W[3] = {0,0,0};

// Iterate through all three axes and calculate the proportional component, derivative component,
// and sum them together into the F vector in the N frame.
int i = 0;
for(i = 0; i < 3; i++) {
    f_p[i] = -KP_task[i]*R_NW[i][0]*(Position[0]-Position_d[0]) + -KP_task[i]*R_NW[i][1]*(Position[1]-Position_d[1]) + -KP_task[i]*R_NW[i][2]*(Position[2]-Position_d[2]);

    f_d[i] = -KD_task[i]*R_NW[i][0]*(Velocity[0]-Velocity_d[0]) + -KD_task[i]*R_NW[i][1]*(Velocity[1]-Velocity_d[1]) + -KD_task[i]*R_NW[i][2]*(Velocity[2]-Velocity_d[2]);

    F_N[i] = f_p[i] + f_d[i]; // sum the proportional and derivative components
}

/*
 * Calculate the F vector in the World frame by taking the force vector in the N frame and
 * transforming it via multiplication by R_WN, the rotation from the World frame to frame N.
 */
for(i = 0; i < 3; i++) {
    F_W[i] = R_WN[i][0]*F_N[0] + R_WN[i][1]*F_N[1] + R_WN[i][2]*F_N[2];
```

```
    }

    /*
     * Calculate control efforts (tau) by taking the F vector in the world frame and
     * multiplying it by the transpose of the Jacobian.
     */
    for(i = 0; i < 3; i++) {
        t[i] = J_t[i][0]*F_W[0] + J_t[i][1]*F_W[1] + J_t[i][2]*F_W[2];
    }
}



////////////////////
// The main function is called every 1 ms and performs the entire control loop using the timing variable mycount.
void lab(float theta1motor,float theta2motor,float theta3motor,float *tau1,float *tau2,float *tau3, int error) {

    //Storing the input motor angle into a global variable that we can monitor. It also allows us to use and manipulate these values in our other functions, like calculating the current position of the end effector in the task space.
    theta_motor[0] = theta1motor;
    theta_motor[1] = theta2motor;
    theta_motor[2] = theta3motor;

    // Convert the mycount variable that is incremented every millisecond into seconds to be used with the trajectory generated desired point.
    float time = (mycount)/1000.0;

    // 1)
    filter_velocity();

    // 2) Calculates and sets the desired position according to the linear trajectory form given by a point and a motion vector.
    task_space_trajectory(time);

    // 3) Calculate the position of the end effector using the forward kinematic equations of the robot manipulator and the current joint angles.
    forward_kinematics(theta_motor[0], theta_motor[1], theta_motor[2]);

    // 4) Use IIR filter to take a smooth estimate of the task space velocities of the end effector using the positions calculated by the forward kinematics.
    filter_velocity_task();

    // 5) Implement task space control laws to calculate the control effort t[0-3]
    task_space_control();

```

```c
    *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 5 above. Note fric_on is a Boolean allowing easy toggling of friction compensation.
    *tau1 = *tau1 + fric_on*u_fric[0];
    *tau2 = *tau2 + fric_on*u_fric[1];
    *tau3 = *tau3 + fric_on*u_fric[2];

    // 8) Send relevant variables to Simulink in order to tune the controller response.

    Simulink_PlotVar1 = Position_d[0];
    Simulink_PlotVar2 = Position[0];
    Simulink_PlotVar3 = Position[1];
    Simulink_PlotVar4 = Position[2];


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

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

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

    }

    mycount++;
}
```

Listing 1: lab4.c