```
1 #include <tistdtypes.h>
 2 #include <coecsl.h>
 3 #include "user includes.h"
 4 #include "math.h"
 6 // 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
 7 float offset Enc2 rad = -0.415; //-0.37;
 8 float offset_Enc3_rad = 0.233; //0.27;
10 // Your global variables.
11 float px = 0; // x coordinate of end effector
12 float py = 0; // y coordinate of end effector
13 float pz = 0; // z coordinate of end effector
14 float theta1m_IK = 0; // theta1 motor from inverse kinematics
15 float theta2m_IK = 0; // theta2 motor from inverse kinematics
16 float theta3m_IK = 0; // theta3 motor from inverse kinematics
17 long mycount = 0;
18 long counter = 0;
19 int cycle = 0;
20
21 // Omega calculations
22 float Theta1_old = 0;
23 float Omega1_old1 = 0;
24 float Omega1_old2 = 0;
25 float Omega1 = 0;
26 float e1 old = 0;
27 float integral1_old = 0;
28
29 float Theta2_old = 0;
30 float Omega2_old1 = 0;
31 float Omega2_old2 = 0;
32 float Omega2 = 0;
33 float e2_old = 0;
34 float integral2_old = 0;
36 float Theta3 old = 0;
37 float Omega3_old1 = 0;
38 float Omega3_old2 = 0;
39 float Omega3 = 0;
40 float e3_old = 0;
41 float integral3_old = 0;
43 // Velocity calculations
44 float x_old = 0;
45 float vx_old1 = 0;
46 float vx_old2 = 0;
47 | float vx = 0;
48 //float x_e_old = 0;
49 //float integral1_old = 0;
50
51 float y_old = 0;
52 float vy_old1 = 0;
53 float vy_old2 = 0;
54 | float vy = 0;
55 //float y_e_old = 0;
56 //float integral2_old = 0;
57
```

```
58 float z_old = 0;
 59 float vz_old1 = 0;
 60 float vz_old2 = 0;
 61 float vz = 0;
 62 //float z_e_old = 0;
 63 //float integral3_old = 0;
 64
 65 // Constants
 66 float dt = 0.001;
 67 float threshold1 = 0.01;
 68 float threshold2 = 0.06;
 69 float threshold3 = 0.02;
 70
 71 //Inverse Dynamics Gains
 72 float kp1 = 300.0;
 73 float kd1 = 4;
 74 float ki1 = 420;
 75
 76 float kp2 = 5000;
 77 float kd2 = 250;
 78 float ki2 = 280;
 79
 80 float kp3 = 7000;
 81 float kd3 = 300;
 82 float ki3 = 260;
 83
 84 //PD Gains
 85 float fkp1 = 300;
 86 float fkp2 = 350;
 87 float fkp3 = 300;
 88
 89 float fkd1 = 4;
 90 float fkd2 = 4;
 91 float fkd3 = 4;
 92
 93 float theta_dotdot = 0.0;
94
 95 float e1 = 0;
 96 float e2 = 0;
 97 float e3 = 0;
98
99 float x_desired = 0;
100 float y_desired = 0;
101 float z_desired = 0;
102 //Positive and negative viscous coefficients, and positive and negative coulomb
    (static) coefficients for each joint
103 float Viscouspos1=0.17, Viscouspos2=0.23, Viscouspos3=0.1922;
104 float Viscousneg1=0.17, Viscousneg2=0.287, Viscousneg3=0.2132;
105 float Coulombpos1=0.40, Coulombpos2=0.40, Coulombpos3=0.40;
106 float Coulombneg1=-0.35, Coulombneg2=-0.40, Coulombneg3=-0.50;
107
108 float slope1 = 3.6;
109 | float slope2 = 3.6;
110 float slope3 = 3.6;
111
112 //Without Mass
113 float p1 = 0.0300;
114 float p2 = 0.0128;
115 float p3 = 0.0076;
116 float p4 = 0.0753;
```

```
117 float p5 = 0.0298;
118
119 float a theta2;
120 float a_theta3;
122 float sintheta2;
123 float costheta2;
124 float sintheta3;
125 float costheta3;
126 float g = 9.81;
127
128
129 float J1 = 0.0167;
130 float J2 = 0.03;
131 float J3 = 0.0128;
132
133 float ff = 1;
134
135 float a0,a1,a2,a3;
136 float tau1cur, tau2cur, tau3cur;
137
138 float desired_1, desired_2, desired_3;
139 float desired_dot_1, desired_dot_2, desired_dot_3;
140 float desired_doubledot_1, desired_doubledot_2, desired_doubledot_3;
141
142 //Lab 4 Starter Code
143 float cosq1 = 0;
144 float sinq1 = 0;
145 float cosq2 = 0;
146 float sinq2 = 0;
147 float cosq3 = 0;
148 float sinq3 = 0;
149 float JT_11 = 0;
150 float JT_12 = 0;
151 float JT_13 = 0;
152 float JT_21 = 0;
153 float JT_22 = 0;
154 float JT_23 = 0;
155 float JT 31 = 0;
156 float JT_32 = 0;
157 float JT_33 = 0;
158 float cosz = 0;
159 float sinz = 0;
160 float cosx = 0;
161 float sinx = 0;
162 float cosy = 0;
163 float siny = 0;
164 //Rotational matrix
165 float R11 = 0;
166 float R12 = 0;
167 | float R13 = 0;
168 float R21 = 0;
169 float R22 = 0;
170 float R23 = 0;
171 float R31 = 0;
172 float R32 = 0;
173 | float R33 = 0;
174 //Transpose of the rotational matrix
175 float RT11 = 0;
176 float RT12 = 0;
```

```
177 float RT13 = 0;
178 float RT21 = 0;
179 float RT22 = 0;
180 float RT23 = 0;
181 float RT31 = 0;
182 float RT32 = 0;
183 float RT33 = 0;
184
185 float thetax = 0;
186 float thetay = 0;
187 float thetaz = 60;
188
189 float kpx = 0.5;
190 float kpy = 0.5;
191 float kpz = 0.5;
192 float kdx = 0.025;
193 float kdy = 0.025;
194 float kdz = 0.025;
195
196 //Friction Multiplication factor
197 float fzcmd = 8;
198 //Robots torque constant
199 float kt = 6;
200 //Gravity Compensation in Z direction
201 float grav_comp = 5;
202
203 // Straight line trajectory
204 float vel = 0.5;
205 float xa = 8;
206 float ya = 8;
207 float za = 10;
208 float xb = 25.32;
209 float yb = 18;
210 float zb = 10;
211 float t_start = 0;
212
213 // Gains in N-frame
214 float Kpx_n = 1.3;
215 float Kpy n = 1.3;
216 float Kpz_n = 1.3;
217 float Kdx n = 0.03;
218 float Kdy_n = 0.03;
219 float Kdz_n = 0.025;
220
221
222 #pragma DATA_SECTION(whattoprint, ".my_vars")
223 float whattoprint = 0.0;
225 #pragma DATA_SECTION(whatnottoprint, ".my_vars")
226 float whatnottoprint = 0.0;
228 #pragma DATA_SECTION(theta1array, ".my_arrs")
229 float theta1array[100];
230
231 #pragma DATA_SECTION(theta2array, ".my_arrs")
232 float theta2array[100];
233
234 long arrayindex = 0;
235
236 float printtheta1motor = 0;
```

```
237 float printtheta2motor = 0;
238 float printtheta3motor = 0;
239
240 // Assign these float to the values you would like to plot in Simulink
241 float Simulink_PlotVar1 = 0;
242 float Simulink_PlotVar2 = 0;
243 float Simulink_PlotVar3 = 0;
244 float Simulink_PlotVar4 = 0;
245
246
247 void inverseKinematics(float px, float py, float pz) {
        // Calculate DH angles from end effector position
248
249
        //Function evaluates inverse kinematics for the robot, inputs are px, py, pz
250
        float theta1 = atan2(py, px); // DH theta 1
        float z = pz - 10;
251
252
        float beta = sqrt(px*px + py*py);
253
        float L = sqrt(z*z + beta*beta);
        float theta3 = acos((L*L - 200)/200); // DH theta 3
254
255
        float theta2 = -theta3/2 - atan2(z, beta); // DH theta 2
256
257
258
        // Convert DH angles to motor angles
259
        theta1m_IK = theta1;
        theta2m_IK = (theta2 + PI/2);
260
261
        theta3m_IK = (theta3 + theta2m_IK - PI/2);
262 }
263
264 // Omega estimation
265 void omega(float theta1motor, float theta2motor, float theta3motor) {
266
        //Function updates omega values given theta1motor, theta2motor and theta3motor
267
        Omega1 = (theta1motor - Theta1_old)/0.001;
268
        Omega1 = (Omega1 + Omega1_old1 + Omega1_old2)/3.0;
269
270
        Theta1_old = theta1motor;
271
272
        Omega1_old2 = Omega1_old1;
273
        Omega1_old1 = Omega1;
274
275
        Omega2 = (theta2motor - Theta2 old)/0.001;
276
        Omega2 = (Omega2 + Omega2 \text{ old1} + Omega2 \text{ old2})/3.0;
277
278
        Theta2_old = theta2motor;
279
280
        Omega2_old2 = Omega2_old1;
281
        Omega2 old1 = Omega2;
282
283
        Omega3 = (theta3motor - Theta3_old)/0.001;
284
        Omega3 = (Omega3 + Omega3_old1 + Omega3_old2)/3.0;
285
286
        Theta3_old = theta3motor;
287
288
        Omega3_old2 = Omega3_old1;
289
        Omega3_old1 = Omega3;
290 }
291
292 //Velocity Updates
293 void velocity(float x, float y, float z) {
294
        //Function updates velocity given x, y, z
295
        vx = (x - x_old)/0.001;
296
       vx = (vx + vx_old1 + vx_old2)/3.0;
```

```
297
298
        x_old = x;
299
300
        vx old2 = vx old1;
        vx_old1 = vx;
301
302
        vy = (y - y_old)/0.001;
303
304
        vy = (vy + vy_old1 + vy_old2)/3.0;
305
306
        y_old = y;
307
308
        vy_old2 = vy_old1;
309
        vy_old1 = vy;
310
311
        vz = (z - z_old)/0.001;
312
        vz = (vz + vz_old1 + vz_old2)/3.0;
313
314
        z_old = z;
315
316
        vz_old2 = vz_old1;
        vz old1 = vz;
317
318 }
319
320
321 //Trajectory Generation
322 float cubic func(float a0, float a1, float a2, float a3, float t) {
323
        return a0*t*t*t + a1*t*t + a2*t + a3;
324 }
325 float dot(float a0, float a1, float a2, float t) {
        return 3*a0*t*t + 2*a1*t + a2;
326
327 }
328 float doubledot(float a0, float a1, float t) {
329
        return 6*a0*t + 2*a1;
330 }
331
332 void lab(float theta1motor,float theta2motor,float theta3motor,float *tau1,float
    *tau2,float *tau3, int error) {
333
334
        //Forward Kinematics
        float x = 10*\cos(\text{theta1motor})*(\cos(\text{theta3motor}) + \sin(\text{theta2motor}));
335
        float y = 10*sin(theta1motor)*(cos(theta3motor) + sin(theta2motor));
336
        float z = 10*\cos(\text{theta2motor}) - 10*\sin(\text{theta3motor}) + 10;
337
338
339
        float time = counter*0.001;
340
        //Coefficients for different time steps
341
        if (counter < 330){
342
            a0 = -27.826474107465835; a1 = 13.774104683195590; a2 = 0; a3 = 0.25;
343
        }
        if (counter >= 330 && counter < 4000){
344
345
            a0 = 0; a1 = 0; a2 = 0; a3 = 0.75;
346
347
        if (counter >= 4000 && counter < 4330){
348
            a0 = 27.826474107496760; a1 = -347.6917939731718; a2 = 1445.863594625530; a3
    = -2000.530017811164;
349
        if (counter >= 4330 && counter < 8000){
350
351
            a0 = 0; a1 = 0; a2 = 0; a3 = 0.25;
352
353
        //Desired Trajectories
354
        desired_1 = cubic_func(a0, a1, a2, a3, time);
```

```
355
        desired_dot_1 = dot(a0, a1, a2, time);
356
        desired_doubledot_1 = doubledot(a0, a1, time);
357
358
        desired_2 = cubic_func(a0, a1, a2, a3, time);
359
        desired_dot_2 = dot(a0, a1, a2, time);
360
        desired_doubledot_2 = doubledot(a0, a1, time);
361
362
        desired_3 = cubic_func(a0, a1, a2, a3, time);
        desired dot 3 = dot(a0, a1, a2, time);
363
        desired_doubledot_3 = doubledot(a0, a1, time);
364
365
        //Step Trajectory to 10, 10, 10 without velocity
366
367
        //Desired Position
368 //
          float xd = 10;
369 //
          float xd_dot = 0;
370 //
371 //
          float yd = 10;
372 //
          float yd_dot = 0;
373 //
374 //
          float zd = 10;
375 //
          float zd_dot = 0;
376
377
378 //Straight line Trajectory for Lab3
379 // set desired points
        float xd = xb;
380
381
        float yd = yb;
382
        float zd = zb;
383
        float xd_dot = 0;
384
        float yd_dot = 0;
385
        float zd dot = 0;
        float t_total = sqrt((xb-xa)*(xb-xa) + (yb-ya)*(yb-ya) + (zb-za)*(zb-za)) / vel;
386
387
388
        //Calculate desired x,y,z position as a linear function of time
389
        if (time >= t_start && time <= t_start + t_total) {</pre>
390
            xd = (xb-xa)*(time - t_start)/t_total + xa;
391
            yd = (yb-ya)*(time - t_start)/t_total + ya;
392
            zd = (zb-za)*(time - t_start)/t_total + za;
393
        }
394
395 // Calculate/Update omegas/velocities
396
        omega(theta1motor, theta2motor, theta3motor);
397
        velocity(x, y, z);
398
399 // position error
400
        float x_error = xd - x;
        float y_error = yd - y;
401
402
        float z_error = zd - z;
        float xd_error = xd_dot - vx;
403
404
        float yd_error = yd_dot - vy;
405
        float zd_error = zd_dot - vz;
406
407 //
        Desired theta - current theta
408
        float e1 = desired 1 - theta1motor;
        float e2 = desired_2 - theta2motor;
409
410
        float e3 = desired_3 - theta3motor;
411
412
        // Integral approximation
413
        float integral1 = integral1_old + (e1 + e1_old) * dt / 2;
414
        float integral2 = integral2 old + (e2 + e2 old) * dt / 2;
```

```
415
        float integral3 = integral3_old + (e3 + e3_old) * dt / 2;
416
417
        // Prevents integral windup
418
        if (fabs(e1) > threshold1) {
419
            integral1 = 0;
420
            integral1_old = 0;
421
        }
422
423
        if (fabs(e2) > threshold2) {
424
                integral2 = 0;
425
                integral2_old = 0;
        }
426
427
        if (fabs(e3) > threshold3) {
428
429
                integral3 = 0;
430
                integral3_old = 0;
431
        }
432
433
        //Mode = 0: Feed forward + acc;
        //Mode = 1: Feed forward + PD
434
435
        //Mode = 2: Task Space PD controller
        //Mode = 3: Simple Impedance Control
436
437
        int mode = 3;
438
        if (mode == 0){
439
            // acceleration of joint 2/3
440
                float D1 = p1;
                float D2 = -p3*sin(theta3motor-theta2motor);
441
442
                float D3 = -p3*sin(theta3motor-theta2motor);
443
                float D4 = p2;
444
                float C1 = 0;
445
                float C2 = -p3*cos(theta3motor-theta2motor)*Omega3;
                float C3 = p3*cos(theta3motor-theta2motor)*Omega2;
446
447
                float C4 = 0;
                float G1 = -p4*g*sin(theta2motor);
448
449
                float G2 = -p5*g*cos(theta3motor);
450
451
                a_theta2 = desired_doubledot_2 + kp2*(e2) + kd2*(desired_dot_2-Omega2);
452
                a theta3 = desired doubledot 2 + kp3*(e3) + kd3*(desired dot 3-Omega3);
453
454
455
                *tau1 = J1*desired_doubledot_2 + kp1*(e1)+kd1*(desired_dot_2-Omega1);
                *tau2 = (D1*a_theta2+D2*a_theta2)+(C1*Omega2+C2*Omega3)+G1;
456
457
                *tau3 = (D3*a_theta2+D4*a_theta3)+(C3*Omega2+C4*Omega3)+G2;
458
459
        else if(mode == 1){//Feed Forward + PD
            *tau1 = 0.0167 * desired_doubledot_1 + fkp1 * e1 + fkd1 * (desired_dot_1 -
460
    Omega1); // + ki1 * integral1;
461
            *tau2 = 0.03 * desired_doubledot_2 + fkp2 * e2 + fkd2 * (desired_dot_2 -
    Omega2); // + ki2 * integral2;
            *tau3 = 0.0128 * desired_doubledot_3 + fkp3 * e3 + fkd3 * (desired_dot_3 -
462
    Omega3);
463
        else if(mode == 2){//Task Space PD controller
464
465
            float fx = kpx*(xd - x) + kdx*(xd_dot - vx);
            float fy = kpy*(yd - y) + kdy*(yd_dot - vy);
466
            float fz = kpz*(zd - z) + kdz*(zd_dot - vz);
467
468
            //Compute Jacobian Transpose
469
470
            // Jacobian Transpose
471
            cosq1 = cos(theta1motor);
```

```
472
            sinq1 = sin(theta1motor);
473
            cosq2 = cos(theta2motor);
474
            sinq2 = sin(theta2motor);
475
            cosq3 = cos(theta3motor);
            sinq3 = sin(theta3motor);
476
477
            JT_11 = -10*sinq1*(cosq3 + sinq2);
478
            JT_12 = 10*cosq1*(cosq3 + sinq2);
479
            JT 13 = 0;
            JT_21 = 10*cosq1*(cosq2 - sinq3);
480
481
            JT_22 = 10*sinq1*(cosq2 - sinq3);
482
            JT_23 = -10*(cosq3 + sinq2);
            JT_31 = -10*cosq1*sinq3;
483
484
            JT_32 = -10*sinq1*sinq3;
485
            JT 33 = -10*cosq3;
            //Simple Impedance Control
486
487
            float x_grav_comp = 0;
488
            float y grav comp = 0.0254*JT 23*fzcmd/kt;
            float z_grav_comp = 0.0254*JT_33*(fzcmd + grav_comp)/kt;
489
490
            *tau1 = JT_11*fx + JT_12*fy + x_grav_comp;
            *tau2 = JT 21*fx + JT 22*fy + JT 23*fz + y grav comp;
491
            *tau3 = JT_31*fx + JT_32*fy + JT_33*fz + z_grav_comp;
492
493
494
        }
495
496
        else if(mode == 3){//Simple Impedance Control
            //Lab 4 = Impedence Controls, all the variables are used here, provided in
497
    the starter code
498
            //Compute Jacobian Transpose
499
            // Jacobian Transpose
500
501
            //The first section of code defines any terms used in lab 4, primarily the
   matrix components. Because the majority of the terms below are sinusoidal,
    calculating them once per loop and then calling the saved value is far more
    efficient than calling a cosine or sine every time a term is used. In the torque
    equations, the forward kinematics, jacobian, and rotation matrices are defined and
    stored to be called later.
            //Saves the cos and sin values of the three joint angles so they don't have
502
    to be recalculated each time they're called during the function's run.
503
            cosq1 = cos(theta1motor);
504
505
            sinq1 = sin(theta1motor);
            cosq2 = cos(theta2motor);
506
507
            sinq2 = sin(theta2motor);
508
            cosq3 = cos(theta3motor);
509
            sinq3 = sin(theta3motor);
510
            //Jacobian Transpose for the CRS robot, separated into its constituent parts
511
    from matrix form
            JT_11 = -10*sinq1*(cosq3 + sinq2);
512
513
            JT 12 = 10*\cos q1*(\cos q3 + \sin q2);
514
            JT 13 = 0;
            JT 21 = 10*cosq1*(cosq2 - sinq3);
515
            JT_22 = 10*sinq1*(cosq2 - sinq3);
516
517
            JT_23 = -10*(cosq3 + sinq2);
            JT 31 = -10*\cos q1*\sin q3;
518
519
            JT_32 = -10*sinq1*sinq3;
520
            JT 33 = -10*cosq3;
521
522
            //zxy Rotation and Transpose
```

```
523
                                                           //The values of cos and sin of the rotation about each axis are stored so
                   they don't have to be recalculated every time the function is called.
524
                                                           cosz = cos(thetaz);
525
                                                            sinz = sin(thetaz);
                                                           cosx = cos(thetax);
526
527
                                                           sinx = sin(thetax);
528
                                                           cosy = cos(thetay);
529
                                                           siny = sin(thetay);
530
531
                                                           //The given equations for the rotation matrix's transpose, divided into its
                   constituent parts
                                                           RT11 = R11 = cosz*cosy-sinz*sinx*siny;
532
533
                                                           RT21 = R12 = -\sin z * \cos x;
                                                           RT31 = R13 = cosz*siny+sinz*sinx*cosy;
534
                                                           RT12 = R21 = sinz*cosy+cosz*sinx*siny;
535
                                                           RT22 = R22 = cosz*cosx;
536
                                                           RT32 = R23 = sinz*siny-cosz*sinx*cosy;
537
538
                                                           RT13 = R31 = -cosx*siny;
539
                                                           RT23 = R32 = sinx;
540
                                                           RT33 = R33 = cosx*cosy;
541
542
                                                           ///The torque equations are calculated with the addition of a rotation
                   matrix in this section of code. The rotation matrix's purpose is to change the
                   direction in which a force can be applied to the end effector while keeping the
                   trajectory in the world frame. MATLAB was used to multiply the matrices by hand.
                                                             *tau1 = (JT 11*R11 + JT 12*R21 + JT 13*R31)*(Kdx n*R11*xd error +
543
                   Kpx_n*R11*x_error + Kdx_n*R21*yd_error + Kpx_n*R21*y_error + Kdx_n*R31*zd_error +
                   Kpx_n*R31*z_error) + (JT_11*R12 + JT_12*R22 + JT_13*R32)*(Kdy_n*R12*xd_error + JT_13*R32)*(Kdy_n*R12*xd_error) + (JT_11*R12 + JT_12*R22 + JT_13*R32)*(Kdy_n*R12*xd_error) + (JT_11*R12*xd_error) + (JT_11*Xd_error) + (
                   Kpy_n*R12*x_error + Kdy_n*R22*yd_error + Kpy_n*R22*y_error + Kdy_n*R32*zd_error +
                   Kpy_n*R32*z_error) + (JT_11*R13 + JT_12*R23 + JT_13*R33)*(Kdz_n*R13*xd_error + JT_13*R33)*(Kdz_n*R13*xd_error) + (JT_11*R13 + JT_12*R23 + JT_13*R33)*(Kdz_n*R13*xd_error) + (JT_11*R13 + JT_13*R33)*(Kdz_n*R13*xd_error) + (JT_11*R13 + JT_13*R33)*(Kdz_n*R13*xd_error) + (JT_11*R13*xd_error) + (JT_11*xd_error) + (JT_
                   Kpz n*R13*x error + Kdz n*R23*yd error + Kpz n*R23*y error + Kdz n*R33*zd error +
                   Kpz n*R33*z error);
544
                                                            *tau2 = (JT_21*R11 + JT_22*R21 + JT_23*R31)*(Kdx_n*R11*xd_error +
                   Kpx n*R11*x_error + Kdx_n*R21*yd_error + Kpx_n*R21*y_error + Kdx_n*R31*zd_error +
                   Kpx_n*R31*z_error) + (JT_21*R12 + JT_22*R22 + JT_23*R32)*(Kdy_n*R12*xd_error + JT_23*R32)*(Kdy_n*R12*xd_error) + (JT_21*R12 + JT_22*R22 + JT_23*R32)*(Kdy_n*R12*xd_error) + (JT_21*R12*xd_error) + (JT_21*xd_error) + (JT_21*xd_error) + (JT_21*xd_error) + (JT_21*xd
                   Kpy_n*R12*x_error + Kdy_n*R22*yd_error + Kpy_n*R22*y_error + Kdy_n*R32*zd_error + Kdy_n*R32
                   Kpy_n*R32*z_error) + (JT_21*R13 + JT_22*R23 + JT_23*R33)*(Kdz_n*R13*xd_error + JT_23*R33)*(Kdz_n*R13*xd_error) + (JT_21*R13 + JT_22*R23 + JT_23*R33)*(Kdz_n*R13*xd_error) + (JT_21*R13*xd_error) + (JT_21*R13*xd
                   Kpz_n*R13*x_error + Kdz_n*R23*yd_error + Kpz_n*R23*y_error + Kdz_n*R33*zd_error +
                   Kpz n*R33*z error);
545
                                                             *tau3 = (JT 31*R11 + JT 32*R21 + JT 33*R31)*(Kdx n*R11*xd error +
                   Kpx_n*R11*x_error + Kdx_n*R21*yd_error + Kpx_n*R21*y_error + Kdx_n*R31*zd_error +
                   Kpx_n*R31*z_error) + (JT_31*R12 + JT_32*R22 + JT_33*R32)*(Kdy_n*R12*xd_error +
                   Kpy_n*R12*x_error + Kdy_n*R22*yd_error + Kpy_n*R22*y_error + Kdy_n*R32*zd_error +
                   Kpy_n*R32*z_error) + (JT_31*R13 + JT_32*R23 + JT_33*R33)*(Kdz_n*R13*xd_error + JT_33*R33)*(Kdz_n*R13*xd_error) + (JT_31*R13 + JT_32*R23 + JT_33*R33)*(Kdz_n*R13*xd_error) + (JT_31*R13*xd_error) + (JT_31*xd_error) + (JT_31*xd_er
                   Kpz_n*R13*x_error + Kdz_n*R23*yd_error + Kpz_n*R23*y_error + Kdz_n*R33*zd_error +
                   Kpz_n*R33*z_error);
546
                                        }
547
548
                                           // Friction Compensation
549
                                        //If omega greater than 0.1 (Max Velocity) for joint 1
550
                                        if (Omega1 > 0.1) {
551
                                                             *tau1 = *tau1 + 0.6*(Viscouspos1 * Omega1 + Coulombpos1);
552
                                        //If omega greater than -0.1 (Min Velocity) for joint 1
553
554
                                        else if (Omega1 <-0.1) {
                                                            *tau1 = *tau1 +0.6*(Viscousneg1 * Omega1 + Coulombneg1);
555
556
                                        }
557
                                        //If omega between min and max velocity for joint 1
558
                                        else {
                                                             *tau1 = *tau1 + 0.6*(slope1*0mega1);
559
```

```
560
        }
561
562
        //If omega greater than 0.05(Max Velocity) for joint 2
563
        if (Omega2 > 0.05) {
            *tau2 = *tau2 + 0.6*(Viscouspos2 * Omega2 + Coulombpos2);
564
565
566
        //If omega greater than -0.05 (Min Velocity) for joint 2
567
        else if (Omega2 <-0.05) {
568
            *tau2 = *tau2 + 0.6*(Viscousneg2 * Omega2 + Coulombneg2);
569
570
        //If omega between min and max velocity for joint 2
571
        else {
572
            *tau2 = *tau2 + 0.6*(slope2*Omega2);
573
        }
574
        //If omega greater than -0.05 (Max Velocity) for joint 3
575
        if (Omega3 > 0.05) {
            *tau2 = *tau2 + 0.6*(Viscouspos3 * Omega3 + Coulombpos3);
576
        }
577
578
        //If omega greater than -0.05 (Min Velocity) for joint 3
579
        else if (Omega3 <-0.05) {
580
            *tau2 = *tau2 + 0.6*(Viscousneg3 * Omega3 + Coulombneg3);
581
582
        //If omega between min and max velocity for joint 3
583
        else {
584
            *tau2 = *tau2 + 0.6*(slope3*Omega3);
585
        }
586
587
        //Prevents integral windup (Max: 5 Min: -5)
588
        if (*tau1 >= 5) {
589
            *tau1 = 5;
590
            integral1 = integral1_old;
591
        } else if (*tau1 < -5) {</pre>
592
            *tau1 = -5;
593
            integral1 = integral1_old;
594
        }
595
596
        if (*tau2 >= 5) {
597
            *tau2 = 5;
598
            integral2 = integral2_old;
        } else if (*tau2 < -5) {</pre>
599
600
            *tau2 = -5;
601
            integral2 = integral2_old;
602
        }
603
604
        if (*tau3 >= 5) {
605
            *tau3 = 5;
606
            integral3 = integral3_old;
607
        } else if (*tau3 < -5) {</pre>
            *tau3 = -5;
608
609
            integral3 = integral3_old;
610
        }
611
612
        e1_old = e1;
613
        e2_old = e2;
        e3 	ext{ old } = e3;
614
        integral1_old = integral1;
615
616
        integral2_old = integral2;
617
        integral3_old = integral3;
618
619
```

```
620
        // save past states
621
        if ((mycount%50)==0) {
622
            theta1array[arrayindex] = theta1motor;
623
624
            theta2array[arrayindex] = theta2motor;
625
626
            if (arrayindex >= 100) {
627
                arrayindex = 0;
628
            } else {
629
                arrayindex++;
            }
630
631
632
        }
633
        if ((mycount%500)==0) {
634
            if (whattoprint > 0.5) {
635
                serial_printf(&SerialA, "I love robotics\n\r");
636
637
            } else {
638
                printtheta1motor = theta1motor;
639
                printtheta2motor = theta2motor;
                printtheta3motor = theta3motor;
640
                SWI_post(&SWI_printf); //Using a SWI to fix SPI issue from sending too
641
   many floats.
642
            }
643
            GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1; // Blink LED on Control Card
644
            GpioDataRegs.GPBTOGGLE.bit.GPIO60 = 1; // Blink LED on Emergency Stop Box
645
646
        }
647
648
649
        Simulink_PlotVar1 = desired_1; //yellow
650
651
        Simulink_PlotVar2 = theta1motor; //blue
        Simulink_PlotVar3 = theta2motor; //orange
652
        Simulink_PlotVar4 = theta3motor; //green
653
654
        mycount++;
655
        counter++;
656 }
657
658 void printing(void){
659
        // Printing (theta1, theta2, theta3, , py, pz) and then on a new line
    (theta1_IK, theta2_IK, theta3_IK)
       serial_printf(&SerialA, "px: %.2f, py: %.2f, pz: %.2f \n\r", px, py, pz);
660
661 }
662
663
```