#include "mbed.h"

#include "rtos.h"

#include "EthernetInterface.h"

#include "ExperimentServer.h"

#include "QEI.h"

#include "BezierCurve.h"

#include "MotorShield.h"

#include "HardwareSetup.h"

#include "Matrix.h"

#include "MatrixMath.h"

#define BEZIER\_ORDER\_FOOT    4

#define NUM\_INPUTS (18 + 2\*(BEZIER\_ORDER\_FOOT+1))

#define NUM\_OUTPUTS 37

#define PULSE\_TO\_RAD (2.0f\*3.14159f / 1200.0f)

// Initializations

Serial pc(USBTX, USBRX);    // USB Serial Terminal

ExperimentServer server;    // Object that lets us communicate with MATLAB

Timer t;                    // Timer to measure elapsed time of experiment

QEI encoderA(PE\_9,PE\_11, NC, 1200, QEI::X4\_ENCODING);  // MOTOR A ENCODER (no index, 1200 counts/rev, Quadrature encoding)

QEI encoderB(PA\_5, PB\_3, NC, 1200, QEI::X4\_ENCODING);  // MOTOR B ENCODER (no index, 1200 counts/rev, Quadrature encoding)

QEI encoderC(PC\_6, PC\_7, NC, 1200, QEI::X4\_ENCODING);  // MOTOR C ENCODER (no index, 1200 counts/rev, Quadrature encoding)

QEI encoderD(PD\_12, PD\_13, NC, 1200, QEI::X4\_ENCODING);// MOTOR D ENCODER (no index, 1200 counts/rev, Quadrature encoding)

MotorShield motorShield(24000); //initialize the motor shield with a period of 12000 ticks or ~20kHZ

Ticker currentLoop;

Matrix MassMatrix1(2,2);

Matrix Jacobian1(2,2);

Matrix JacobianT1(2,2);

Matrix InverseMassMatrix1(2,2);

Matrix temp\_product1(2,2);

Matrix Lambda1(2,2);

Matrix MassMatrix2(2,2);

Matrix Jacobian2(2,2);

Matrix JacobianT2(2,2);

Matrix InverseMassMatrix2(2,2);

Matrix temp\_product2(2,2);

Matrix Lambda2(2,2);

// Variables for q1

float current1;

float current\_des1 = 0;

float prev\_current\_des1 = 0;

float current\_int1 = 0;

float angle1;

float velocity1;

float duty\_cycle1;

float angle1\_init;

// Variables for q2

float current2;

float current\_des2 = 0;

float prev\_current\_des2 = 0;

float current\_int2 = 0;

float angle2;

float velocity2;

float duty\_cycle2;

float angle2\_init;

// Variables for q3

float current3;

float current\_des3 = 0;

float prev\_current\_des3 = 0;

float current\_int3 = 0;

float angle3;

float velocity3;

float duty\_cycle3;

float angle3\_init;

// Variables for q4

float current4;

float current\_des4 = 0;

float prev\_current\_des4 = 0;

float current\_int4 = 0;

float angle4;

float velocity4;

float duty\_cycle4;

float angle4\_init;

// Fixed kinematic parameters

const float l\_OA=.011;

const float l\_OB=.042;

const float l\_AC=.096;

const float l\_DE=.091;

const float m1 =.0393 + .2;

const float m2 =.0368;

const float m3 = .00783;

const float m4 = .0155;

const float I1 = 0.0000251;  //25.1 \* 10^-6;

const float I2 = 0.0000535;  //53.5 \* 10^-6;

const float I3 = 0.00000925; //9.25 \* 10^-6;

const float I4 = 0.0000222;  //22.176 \* 10^-6;

const float l\_O\_m1=0.032;

const float l\_B\_m2=0.0344;

const float l\_A\_m3=0.0622;

const float l\_C\_m4=0.0610;

const float N = 18.75;

const float Ir = 0.0035/pow(N,2);

// New Kinematic Parameters

const float mU = 0.03; //

const float lU = 0.2; // distance from wheel center to "user" CoM

const float I\_U = 1.0/12 \* mU \* pow(lU,2);

const float m5 = 0.0424 \* 6 / 2.205; // 6" length of 80/20 1010 profile

const float ma = 1.00; // 100 grams per wheel (6 oz. each) - weigh them!

const float mb = m5 + 2.0; // total guess; mass of motors plus mounting hardware

const float b = 0.267; // 12" 80/20

const float r = 0.05; // wheels radius

const float l\_cb = 0.1524; // 6" (doesn't account for amount hanging off past connection points)

const float I\_A = 0.5 \* ma \* pow(r,2); // thin solid disk

const float I\_B = 1 \* pow(10,-3); // truly a random guess. Need to do more calcs

// Timing parameters

float current\_control\_period\_us = 200.0f;     // 5kHz current control loop

float impedance\_control\_period\_us = 1000.0f;  // 1kHz impedance control loop

float start\_period, traj\_period, end\_period;

// Control parameters

float current\_Kp = 4.0f;

float current\_Ki = 0.4f;

float current\_int\_max = 3.0f;

float duty\_max;

float K\_xx;

float K\_yy;

float K\_xy;

float D\_xx;

float D\_xy;

float D\_yy;

float t\_swing;

float t\_stance;

float phase\_offset;

float ground\_penetration;

// Model parameters

float supply\_voltage = 12;     // motor supply voltage

float R = 2.0f;                // motor resistance

float k\_t = 0.18f;             // motor torque constant

float nu = 0.0005;             // motor viscous friction

// Current control interrupt function

void CurrentLoop()

{

    // This loop sets the motor voltage commands using PI current controllers with feedforward terms.

    //use the motor shield as follows:

    //motorShield.motorAWrite(DUTY CYCLE, DIRECTION), DIRECTION = 0 is forward, DIRECTION =1 is backwards.

    current1 = -(((float(motorShield.readCurrentA())/65536.0f)\*30.0f)-15.0f);           // measure current

    velocity1 = encoderA.getVelocity() \* PULSE\_TO\_RAD;                                  // measure velocity

    float err\_c1 = current\_des1 - current1;                                             // current errror

    current\_int1 += err\_c1;                                                             // integrate error

    current\_int1 = fmaxf( fminf(current\_int1, current\_int\_max), -current\_int\_max);      // anti-windup

    float ff1 = R\*current\_des1 + k\_t\*velocity1;                                         // feedforward terms

    duty\_cycle1 = (ff1 + current\_Kp\*err\_c1 + current\_Ki\*current\_int1)/supply\_voltage;   // PI current controller

    float absDuty1 = abs(duty\_cycle1);

    if (absDuty1 > duty\_max) {

        duty\_cycle1 \*= duty\_max / absDuty1;

        absDuty1 = duty\_max;

    }

    if (duty\_cycle1 < 0) { // backwards

        motorShield.motorAWrite(absDuty1, 1);

    } else { // forwards

        motorShield.motorAWrite(absDuty1, 0);

    }

    prev\_current\_des1 = current\_des1;

    current2     = -(((float(motorShield.readCurrentB())/65536.0f)\*30.0f)-15.0f);       // measure current

    velocity2 = encoderB.getVelocity() \* PULSE\_TO\_RAD;                                  // measure velocity

    float err\_c2 = current\_des2 - current2;                                             // current error

    current\_int2 += err\_c2;                                                             // integrate error

    current\_int2 = fmaxf( fminf(current\_int2, current\_int\_max), -current\_int\_max);      // anti-windup

    float ff2 = R\*current\_des2 + k\_t\*velocity2;                                         // feedforward terms

    duty\_cycle2 = (ff2 + current\_Kp\*err\_c2 + current\_Ki\*current\_int2)/supply\_voltage;   // PI current controller

    float absDuty2 = abs(duty\_cycle2);

    if (absDuty2 > duty\_max) {

        duty\_cycle2 \*= duty\_max / absDuty2;

        absDuty2 = duty\_max;

    }

    if (duty\_cycle2 < 0) { // backwards

        motorShield.motorBWrite(absDuty2, 1);

    } else { // forwards

        motorShield.motorBWrite(absDuty2, 0);

    }

    prev\_current\_des2 = current\_des2;

    current3     = -(((float(motorShield.readCurrentC())/65536.0f)\*30.0f)-15.0f);       // measure current

    velocity3 = encoderC.getVelocity() \* PULSE\_TO\_RAD;                                  // measure velocity

    float err\_c3 = current\_des3 - current3;                                             // current error

    current\_int3 += err\_c3;                                                             // integrate error

    current\_int3 = fmaxf( fminf(current\_int3, current\_int\_max), -current\_int\_max);      // anti-windup

    float ff3 = R\*current\_des3 + k\_t\*velocity3;                                         // feedforward terms

    duty\_cycle3 = (ff3 + current\_Kp\*err\_c3 + current\_Ki\*current\_int3)/supply\_voltage;   // PI current controller

    float absDuty3 = abs(duty\_cycle3);

    if (absDuty3 > duty\_max) {

        duty\_cycle3 \*= duty\_max / absDuty3;

        absDuty3 = duty\_max;

    }

    if (duty\_cycle3 < 0) { // backwards

        motorShield.motorCWrite(absDuty3, 1);

    } else { // forwards

        motorShield.motorCWrite(absDuty3, 0);

    }

    prev\_current\_des3 = current\_des3;

    current4     = -(((float(motorShield.readCurrentD())/65536.0f)\*30.0f)-15.0f);       // measure current

    velocity4 = encoderD.getVelocity() \* PULSE\_TO\_RAD;                                  // measure velocity

    float err\_c4 = current\_des4 - current4;                                             // current error

    current\_int4 += err\_c4;                                                             // integrate error

    current\_int4 = fmaxf( fminf(current\_int4, current\_int\_max), -current\_int\_max);      // anti-windup

    float ff4 = R\*current\_des4 + k\_t\*velocity4;                                         // feedforward terms

    duty\_cycle4 = (ff4 + current\_Kp\*err\_c4 + current\_Ki\*current\_int4)/supply\_voltage;   // PI current controller

    float absDuty4 = abs(duty\_cycle4);

    if (absDuty4 > duty\_max) {

        duty\_cycle4 \*= duty\_max / absDuty4;

        absDuty4 = duty\_max;

    }

    if (duty\_cycle4 < 0) { // backwards

        motorShield.motorDWrite(absDuty4, 1);

    } else { // forwards

        motorShield.motorDWrite(absDuty4, 0);

    }

    prev\_current\_des4 = current\_des4;

}

int main (void)

{

// Object for 7th order Cartesian foot trajectory

    BezierCurve rDesFoot\_bez(2,BEZIER\_ORDER\_FOOT);

    // Link the terminal with our server and start it up

    server.attachTerminal(pc);

    server.init();

    // Continually get input from MATLAB and run experiments

    float input\_params[NUM\_INPUTS];

    pc.printf("%f",input\_params[0]);

    while(1) {

        // If there are new inputs, this code will run

        if (server.getParams(input\_params,NUM\_INPUTS)) {

            // Get inputs from MATLAB

            start\_period                = input\_params[0];    // First buffer time, before trajectory

            traj\_period                 = input\_params[1];    // Trajectory time/length

            end\_period                  = input\_params[2];    // Second buffer time, after trajectory

            angle1\_init                 = input\_params[3];    // Initial angle for q1 (rad)

            angle2\_init                 = input\_params[4];    // Initial angle for q2 (rad)

            angle3\_init                 = input\_params[5];    // Initial angle for q1 (rad)

            angle4\_init                 = input\_params[6];    // Initial angle for q2 (rad)

            K\_xx                        = input\_params[7];    // Foot stiffness N/m

            K\_yy                        = input\_params[8];    // Foot stiffness N/m

            K\_xy                        = input\_params[9];    // Foot stiffness N/m

            D\_xx                        = input\_params[10];    // Foot damping N/(m/s)

            D\_yy                        = input\_params[11];    // Foot damping N/(m/s)

            D\_xy                        = input\_params[12];   // Foot damping N/(m/s)

            duty\_max                    = input\_params[13];   // Maximum duty factor

            t\_swing                     = input\_params[14]; // Add other inputs

            t\_stance                    = input\_params[15];

            phase\_offset                = input\_params[16]; //

            ground\_penetration          = input\_params[17];

            // Get foot trajectory points

            float foot\_pts[2\*(BEZIER\_ORDER\_FOOT+1)];

            for(int i = 0; i<2\*(BEZIER\_ORDER\_FOOT+1);i++) {

              foot\_pts[i] = input\_params[18+i];

            }

            rDesFoot\_bez.setPoints(foot\_pts);

            // Attach current loop interrupt

            currentLoop.attach\_us(CurrentLoop,current\_control\_period\_us);

            // Setup experiment

            t.reset();

            t.start();

            encoderA.reset();

            encoderB.reset();

            encoderC.reset();

            encoderD.reset();

            motorShield.motorAWrite(0, 0); //turn motor A off

            motorShield.motorBWrite(0, 0); //turn motor B off

            motorShield.motorCWrite(0, 0); //turn motor A off

            motorShield.motorDWrite(0, 0); //turn motor B off

            // Run experiment

            while( t.read() < start\_period + traj\_period + end\_period) {

                // Read encoders to get motor states

                angle1 = encoderA.getPulses() \*PULSE\_TO\_RAD + angle1\_init;

                velocity1 = encoderA.getVelocity() \* PULSE\_TO\_RAD;

                angle2 = encoderB.getPulses() \* PULSE\_TO\_RAD + angle2\_init;

                velocity2 = encoderB.getVelocity() \* PULSE\_TO\_RAD;

                angle3 = encoderC.getPulses() \* PULSE\_TO\_RAD + angle3\_init;

                velocity3 = encoderC.getVelocity() \* PULSE\_TO\_RAD;

                angle4 = encoderD.getPulses() \* PULSE\_TO\_RAD + angle4\_init;

                velocity4 = encoderD.getVelocity() \* PULSE\_TO\_RAD;

                const float th1 = angle1;

                const float th2 = angle2;

                const float th3 = angle3;

                const float th4 = angle4;

                const float dth1= velocity1;

                const float dth2= velocity2;

                const float dth3= velocity3;

                const float dth4= velocity4;

                // Calculate the Jacobian

 // Calculate the Jacobian

                float Jx\_th1 = l\_AC\*cos(th1+th2) + l\_DE\*cos(th1) + l\_OB\*cos(th1);

                float Jx\_th2 = l\_AC\*cos(th1+th2);

                float Jy\_th1 = l\_AC\*sin(th1+th2) + l\_DE\*sin(th1) +l\_OB\*sin(th1);

                float Jy\_th2 = l\_AC\*sin(th1+th2);

                float Jx\_th3 = l\_AC\*cos(th3+th4) + l\_DE\*cos(th3) + l\_OB\*cos(th3);

                float Jx\_th4 = l\_AC\*cos(th3+th4);

                float Jy\_th3 = l\_AC\*sin(th3+th4) + l\_DE\*sin(th3) +l\_OB\*sin(th3);

                float Jy\_th4 = l\_AC\*sin(th3+th4);

                // Calculate the forward kinematics (position and velocity)

                float xFoot1 = l\_OB\*sin(th1) + l\_AC\*sin(th1+th2) + l\_DE\*sin(th1);

                float yFoot1 = -l\_OB\*cos(th1) - l\_AC\*cos(th1+th2) - l\_DE\*cos(th1);

                float dxFoot1 = dth1\*(l\_AC\*cos(th1+th2) + l\_DE\*cos(th1) + l\_OB\*cos(th1)) + dth2\*l\_AC\*cos(th1 + th2);

                float dyFoot1 = dth1\*(l\_AC\*sin(th1+th2) + l\_DE\*sin(th1) + l\_OB\*sin(th1)) + dth2\*l\_AC\*sin(th1 + th2);

                float xFoot2 = l\_OB\*sin(th3) + l\_AC\*sin(th3+th4) + l\_DE\*sin(th3);

                float yFoot2 = -l\_OB\*cos(th3) - l\_AC\*cos(th3+th4) - l\_DE\*cos(th3);

                float dxFoot2 = dth3\*(l\_AC\*cos(th3+th4) + l\_DE\*cos(th3) + l\_OB\*cos(th3)) + dth2\*l\_AC\*cos(th3 + th4);

                float dyFoot2 = dth3\*(l\_AC\*sin(th3+th4) + l\_DE\*sin(th3) + l\_OB\*sin(th3)) + dth2\*l\_AC\*sin(th3 + th4);

                // Set gains based on buffer and traj times, then calculate desired x,y from Bezier trajectory at current time if necessary

                float teff1  = 0;

                float teff2 = 0;

                float vMult = 0;

                if( t < start\_period) {

                    if (K\_xx > 0 || K\_yy > 0) {

                        K\_xx = 100;

                        K\_yy = 100;

                        D\_xx = 5;

                        D\_yy = 5;

                        K\_xy = 0;

                        D\_xy = 0;

                    }

                    teff1 = 0;

                    teff2 = 0;

                }

                else if (t < start\_period + traj\_period)

                {

                    K\_xx = input\_params[5];  // Foot stiffness N/m

                    K\_yy = input\_params[6];  // Foot stiffness N/m

                    K\_xy = input\_params[7];  // Foot stiffness N/m

                    D\_xx = input\_params[8];  // Foot damping N/(m/s)

                    D\_yy = input\_params[9];  // Foot damping N/(m/s)

                    D\_xy = input\_params[10]; // Foot damping N/(m/s)

                    teff1 = fmod((t-start\_period),(t\_swing+t\_stance));

                    teff2 = fmod((t-start\_period),(t\_swing+t\_stance))-phase\_offset;

                    vMult = 1;

                }

                else

                {

                    teff1 = traj\_period;

                    teff2 = traj\_period;

                    vMult = 0;

                }

                // Get desired foot positions and velocities

                float rDesFoot1[2] , vDesFoot1[2];

                if (teff1 < t\_swing){

                    rDesFoot\_bez.evaluate(teff1/traj\_period,rDesFoot1);

                    rDesFoot\_bez.evaluateDerivative(teff1/t\_swing,vDesFoot1);

                    vDesFoot1[0]/= t\_swing;//traj\_period;

                    vDesFoot1[1]/= t\_swing;

                    vDesFoot1[0]\*=vMult;

                    vDesFoot1[1]\*=vMult;

                }

                else{

                    float w = 2\*3.1415/(t\_stance\*2);

                    rDesFoot1[1] = ground\_penetration\*sin(w\*teff1);

                    vDesFoot1[1] = ground\_penetration\*cos(w\*teff1);

                    // Desired x is the end of the bezier traj so don't change

                }

                float rDesFoot2[2] , vDesFoot2[2];

                if (teff2 < t\_swing){

                    rDesFoot\_bez.evaluate(teff2/traj\_period,rDesFoot2);

                    rDesFoot\_bez.evaluateDerivative(teff2/t\_swing,vDesFoot2);

                    vDesFoot2[0]/= t\_swing;//traj\_period;

                    vDesFoot2[1]/= t\_swing;

                    vDesFoot2[0]\*=vMult;

                    vDesFoot2[1]\*=vMult;

                }

                else{

                    float w = 2\*3.1415/(t\_stance\*2);

                    rDesFoot2[1] = ground\_penetration\*sin(w\*teff2);

                    vDesFoot2[1] = ground\_penetration\*cos(w\*teff2);

                }

                // Calculate the inverse kinematics (joint positions and velocities) for desired joint angles

                float xFoot\_inv1 = -rDesFoot1[0];

                float yFoot\_inv1 = rDesFoot1[1];

                float l\_OE1 = sqrt( (pow(xFoot\_inv1,2) + pow(yFoot\_inv1,2)) );

                float alpha1 = abs(acos( (pow(l\_OE1,2) - pow(l\_AC,2) - pow((l\_OB+l\_DE),2))/(-2.0f\*l\_AC\*(l\_OB+l\_DE)) ));

                float th2\_des = -(3.14159f - alpha1);

                float th1\_des = -((3.14159f/2.0f) + atan2(yFoot\_inv1,xFoot\_inv1) - abs(asin( (l\_AC/l\_OE1)\*sin(alpha1) )));

                float xFoot\_inv2 = -rDesFoot2[0];

                float yFoot\_inv2 = rDesFoot2[1];

                float l\_OE2 = sqrt( (pow(xFoot\_inv2,2) + pow(yFoot\_inv2,2)) );

                float alpha2 = abs(acos( (pow(l\_OE2,2) - pow(l\_AC,2) - pow((l\_OB+l\_DE),2))/(-2.0f\*l\_AC\*(l\_OB+l\_DE)) ));

                float th4\_des = -(3.14159f - alpha2);

                float th3\_des = -((3.14159f/2.0f) + atan2(yFoot\_inv2,xFoot\_inv2) - abs(asin( (l\_AC/l\_OE2)\*sin(alpha2) )));

                float dd1 = (Jx\_th1\*Jy\_th2 - Jx\_th2\*Jy\_th1);

                float dth1\_des = (1.0f/dd1) \* (  Jy\_th2\*vDesFoot1[0] - Jx\_th2\*vDesFoot1[1] );

                float dth2\_des = (1.0f/dd1) \* ( -Jy\_th1\*vDesFoot1[0] + Jx\_th1\*vDesFoot1[1] );

                float dd2 = (Jx\_th3\*Jy\_th4 - Jx\_th4\*Jy\_th3);

                float dth3\_des = (1.0f/dd2) \* (  Jy\_th4\*vDesFoot2[0] - Jx\_th4\*vDesFoot2[1] );

                float dth4\_des = (1.0f/dd2) \* ( -Jy\_th3\*vDesFoot2[0] + Jx\_th3\*vDesFoot2[1] );

                // Calculate error variables

                float e\_x1 = rDesFoot1[0] - xFoot1;

                float e\_y1 = rDesFoot1[1] - yFoot1;

                float de\_x1 = vDesFoot1[0] - dxFoot1;

                float de\_y1 = vDesFoot1[1] - dyFoot1;

                float e\_x2 = rDesFoot2[0] - xFoot2;

                float e\_y2 = rDesFoot2[1] - yFoot2;

                float de\_x2 = vDesFoot2[0] - dxFoot2;

                float de\_y2 = vDesFoot2[1] - dyFoot2;

                // Calculate virtual force on foot

                float fx1 = K\_xx\*e\_x1 + K\_xy\*e\_y1 + D\_xx\*de\_x1 + D\_xy\*de\_y1;

                float fy1 = K\_xy\*e\_x1 + K\_yy\*e\_y1 + D\_yy\*de\_y1 + D\_xy\*de\_x1;

                float fx2 = K\_xx\*e\_x2 + K\_xy\*e\_y2 + D\_xx\*de\_x2 + D\_xy\*de\_y2;

                float fy2 = K\_xy\*e\_x2 + K\_yy\*e\_y2 + D\_yy\*de\_y2 + D\_xy\*de\_x2;

                // Calculate mass matrix elements

                float M11\_1 = I1 + I2 + I3 + I4 + Ir + Ir\*pow(N,2) + pow(l\_AC,2)\*m4 + pow(l\_A\_m3,2)\*m3 + pow(l\_B\_m2,2)\*m2 + pow(l\_C\_m4,2)\*m4 + pow(l\_OA,2)\*m3 + pow(l\_OB,2)\*m2 + pow(l\_OA,2)\*m4 + pow(l\_O\_m1,2)\*m1 + 2\*l\_C\_m4\*l\_OA\*m4 + 2\*l\_AC\*l\_C\_m4\*m4\*cos(th2) + 2\*l\_AC\*l\_OA\*m4\*cos(th2) + 2\*l\_A\_m3\*l\_OA\*m3\*cos(th2) + 2\*l\_B\_m2\*l\_OB\*m2\*cos(th2);

                float M12\_1 = I2 + I3 + pow(l\_AC,2)\*m4 + pow(l\_A\_m3,2)\*m3 + pow(l\_B\_m2,2)\*m2 + Ir\*N + l\_AC\*l\_C\_m4\*m4\*cos(th2) + l\_AC\*l\_OA\*m4\*cos(th2) + l\_A\_m3\*l\_OA\*m3\*cos(th2) + l\_B\_m2\*l\_OB\*m2\*cos(th2);

                float M22\_1 = Ir\*pow(N,2) + m4\*pow(l\_AC,2) + m3\*pow(l\_A\_m3,2) + m2\*pow(l\_B\_m2,2) + I2 + I3; //Ir\*N^2 + m4\*l\_AC^2 + m3\*l\_A\_m3^2 + m2\*l\_B\_m2^2 + I2 + I3

                float M11\_2 = I1 + I2 + I3 + I4 + Ir + Ir\*pow(N,2) + pow(l\_AC,2)\*m4 + pow(l\_A\_m3,2)\*m3 + pow(l\_B\_m2,2)\*m2 + pow(l\_C\_m4,2)\*m4 + pow(l\_OA,2)\*m3 + pow(l\_OB,2)\*m2 + pow(l\_OA,2)\*m4 + pow(l\_O\_m1,2)\*m1 + 2\*l\_C\_m4\*l\_OA\*m4 + 2\*l\_AC\*l\_C\_m4\*m4\*cos(th4) + 2\*l\_AC\*l\_OA\*m4\*cos(th4) + 2\*l\_A\_m3\*l\_OA\*m3\*cos(th4) + 2\*l\_B\_m2\*l\_OB\*m2\*cos(th4);

                float M12\_2 = I2 + I3 + pow(l\_AC,2)\*m4 + pow(l\_A\_m3,2)\*m3 + pow(l\_B\_m2,2)\*m2 + Ir\*N + l\_AC\*l\_C\_m4\*m4\*cos(th4) + l\_AC\*l\_OA\*m4\*cos(th4) + l\_A\_m3\*l\_OA\*m3\*cos(th4) + l\_B\_m2\*l\_OB\*m2\*cos(th4);

                float M22\_2 = Ir\*pow(N,2) + m4\*pow(l\_AC,2) + m3\*pow(l\_A\_m3,2) + m2\*pow(l\_B\_m2,2) + I2 + I3; //Ir\*N^2 + m4\*l\_AC^2 + m3\*l\_A\_m3^2 + m2\*l\_B\_m2^2 + I2 + I3

                // Populate mass matrix

                MassMatrix1.Clear();

                MassMatrix1 << M11\_1 << M12\_1

                           << M12\_1 << M22\_1;

                MassMatrix2.Clear();

                MassMatrix2 << M11\_2 << M12\_2

                           << M12\_2 << M22\_2;

                // Populate Jacobian matrix

                Jacobian1.Clear();

                Jacobian1 << Jx\_th1 << Jx\_th2

                         << Jy\_th1 << Jy\_th2;

                Jacobian2.Clear();

                Jacobian2 << Jx\_th3 << Jx\_th4

                         << Jy\_th3 << Jy\_th4;

                // Once you have copied the elements of the mass matrix, uncomment the following section

                // Calculate Lambda matrix

                JacobianT1 = MatrixMath::Transpose(Jacobian1);

                InverseMassMatrix1 = MatrixMath::Inv(MassMatrix1);

                temp\_product1 = Jacobian1\*InverseMassMatrix1\*JacobianT1;

                Lambda1 = MatrixMath::Inv(temp\_product1);

                JacobianT2 = MatrixMath::Transpose(Jacobian2);

                InverseMassMatrix2 = MatrixMath::Inv(MassMatrix2);

                temp\_product2 = Jacobian2\*InverseMassMatrix2\*JacobianT2;

                Lambda2 = MatrixMath::Inv(temp\_product2);

                // Pull elements of Lambda matrix

                float L11\_1 = Lambda1.getNumber(1,1);

                float L12\_1 = Lambda1.getNumber(1,2);

                float L21\_1 = Lambda1.getNumber(2,1);

                float L22\_1 = Lambda1.getNumber(2,2);

                float L11\_2 = Lambda2.getNumber(1,1);

                float L12\_2 = Lambda2.getNumber(1,2);

                float L21\_2 = Lambda2.getNumber(2,1);

                float L22\_2 = Lambda2.getNumber(2,2);

                // Set desired currents

                current\_des1 = (Jx\_th1\*(L11\_1\*fx1+L12\_1\*fy1)+Jy\_th1\*(L12\_1\*fx1+L22\_1\*fy1))/k\_t;

                current\_des2 = (Jx\_th2\*(L11\_1\*fx1+L12\_1\*fy1)+Jy\_th2\*(L12\_1\*fx1+L22\_1\*fy1))/k\_t;

                current\_des3 = (Jx\_th3\*(L11\_2\*fx2+L12\_2\*fy2)+Jy\_th3\*(L12\_2\*fx2+L22\_2\*fy2))/k\_t;

                current\_des4 = (Jx\_th4\*(L11\_2\*fx2+L12\_2\*fy2)+Jy\_th4\*(L12\_2\*fx2+L22\_2\*fy2))/k\_t;

                // Form output to send to MATLAB

                float output\_data[NUM\_OUTPUTS];

                // current time

                output\_data[0] = t.read();

                // motor 1 state

                output\_data[1] = angle1;

                output\_data[2] = velocity1;

                output\_data[3] = current1;

                output\_data[4] = current\_des1;

                output\_data[5] = duty\_cycle1;

                // motor 2 state

                output\_data[6] = angle2;

                output\_data[7] = velocity2;

                output\_data[8] = current2;

                output\_data[9] = current\_des2;

                output\_data[10]= duty\_cycle2;

                // motor 3 state

                output\_data[11] = angle3;

                output\_data[12] = velocity3;

                output\_data[13] = current3;

                output\_data[14] = current\_des3;

                output\_data[15]= duty\_cycle3;

                // motor 4 state

                output\_data[16] = angle4;

                output\_data[17] = velocity4;

                output\_data[18] = current4;

                output\_data[19] = current\_des4;

                output\_data[20]= duty\_cycle4;

                // foot state

                output\_data[21] = xFoot1;

                output\_data[22] = yFoot1;

                output\_data[23] = dxFoot1;

                output\_data[24] = dyFoot1;

                output\_data[25] = rDesFoot1[0];

                output\_data[26] = rDesFoot1[1];

                output\_data[27] = vDesFoot1[0];

                output\_data[28] = vDesFoot1[1];

                output\_data[29] = xFoot2;

                output\_data[30] = yFoot2;

                output\_data[31] = dxFoot2;

                output\_data[32] = dyFoot2;

                output\_data[33] = rDesFoot2[0];

                output\_data[34] = rDesFoot2[1];

                output\_data[35] = vDesFoot2[0];

                output\_data[36] = vDesFoot2[1];

                // Send data to MATLAB

                server.sendData(output\_data,NUM\_OUTPUTS);

                wait\_us(impedance\_control\_period\_us);

            }

            // Cleanup after experiment

            server.setExperimentComplete();

            currentLoop.detach();

            motorShield.motorAWrite(0, 0); //turn motor A off

            motorShield.motorBWrite(0, 0); //turn motor B off

            motorShield.motorCWrite(0, 0); //turn motor C off

            motorShield.motorDWrite(0, 0); //turn motor D off

        } // end if

    } // end while

} // end main