A 6-DOF Stewart Platform

Implemented with an Arduino MEGA and standard RC Servo Motors

Submission Details

Final Deliverable – Report and Demonstration

Marion Garrett Sisk – msisk2@students.kennesaw.edu

Table of Contents

Introduction

A Stewart platform is a six degree-of-freedom (DOF) platform that can be positioned as desired using varying lengths of its support legs. These are often called “hexapods” when seen in the field, as they have six legs supporting an elevated platform. The original goal of this project was to produce a device that stabilize the top platform, given physical inputs from the bottom platform, however difficulties with the implementation of two different inertial measurement units (IMU’s), and challenges with the mathematical transformations involved in the operation of the platform have limited the scope.

Despite the limitation, the platform can be commanded to position itself relative to its home location. The platform is actuated by six servo motors commonly used in remote control applications. These motors use a PWM signal to set a specific location and require 5VDC power, drawing, at most, roughly 1.8A each. The motors are controlled via an Arduino MEGA 2560 microcontroller unit (MCU) which performs all of the linear algebra required to transform input coordinates into PWM signals for each of the servo motors. Additionally, there is an MPU6050 based IMU attached to the top platform. This device measures translational and rotational accelerations. This data can be integrated twice to provide positional data. The device communicates over the I2C bus, which is a simple serial protocol found in many embedded applications. In the current implementation, the data from the IMU is not used. It’s merely displayed via serial terminal.

The unit itself can accurately translate in the X, Y, and Z axes within the physical limits of the platform’s geometry. However there exists a number of problems with the implementation of the rotational matrix that have caused some issues getting rotational movement to accurately reflect the commanded position. A screen shot of the 3D model used in the design of the platform is given below alongside an electrical schematic showing how all of the devices are wired together.

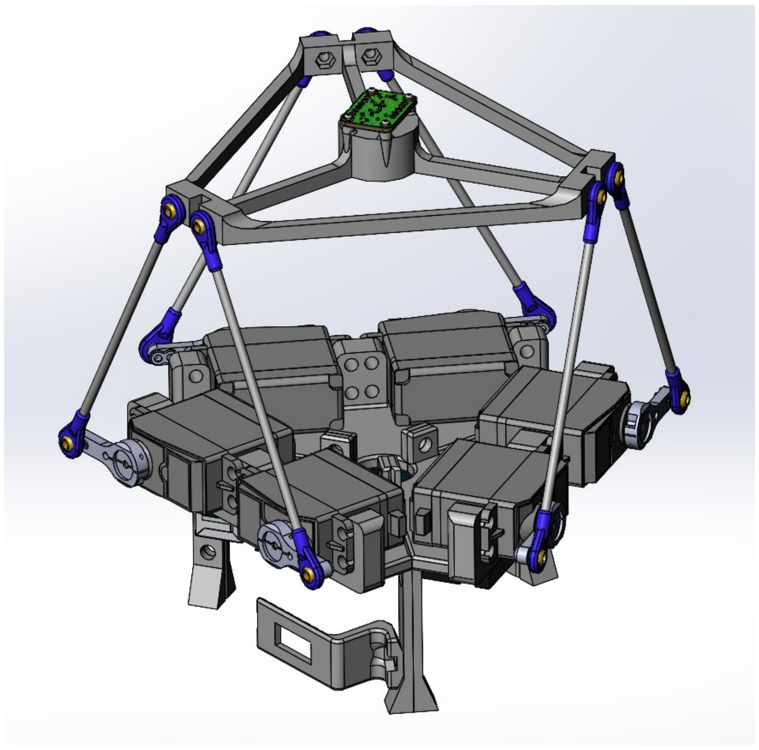

Figure 1: Screen shot of Stewart Platform 3D Model

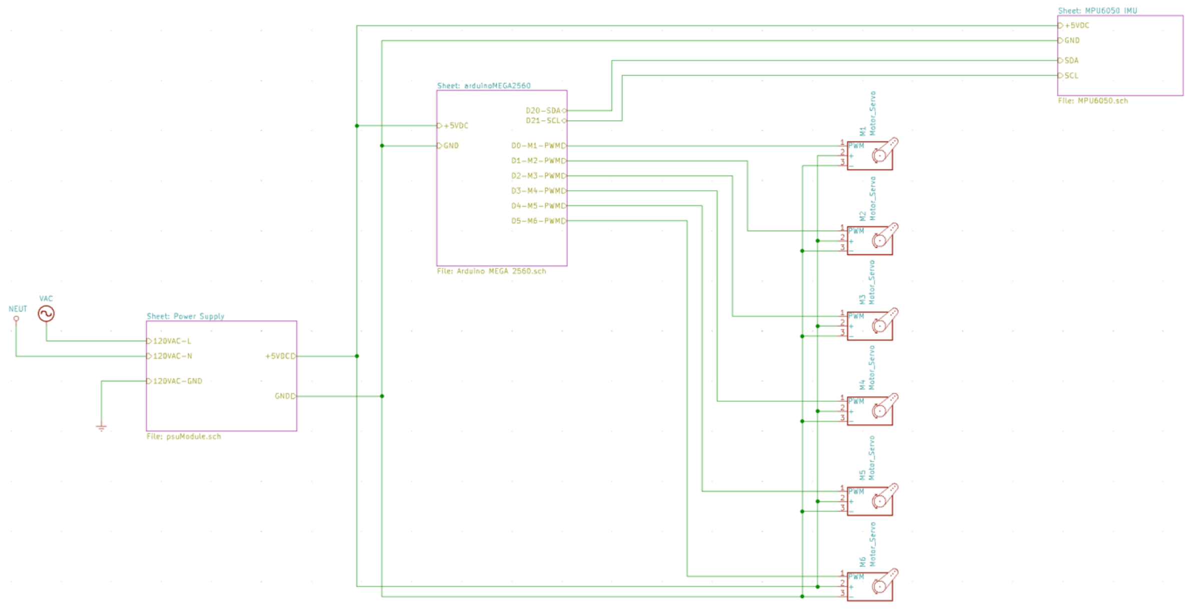

Figure 2: Electrical Schematic of Stewart Platform

Embedded Board Description

The embedded board used in this project is the Arduino MEGA 2560. This board is based on the ATmega2560 microcontroller unit (MCU). As part of the Microchip AVR series of products, his MCU is an 8-bit RISC architecture device produced by Microchip. The maximum clock speed is 16 MHz and can produce 16 MIPS (million instructions per second) at this speed given that most of the instructions are capable of executing in a single clock cycle.

The ATmega2560 also has non-volatile memory on board and can vary depending on the exact model. This particular model has 256kB of usable space for program storage. Additionally, the MCU has its own internal RAM for use in storing variables and other program artifacts during run-time. There is 8kB of SRAM (static random-access memory) available for this use.

The Input/Output capabilities of the ATmega2560 are much more expansive than what was truly needed in this project. The smaller ArduinoUNO would not have worked as the output pins required of the servo motors and the IMU conflicted on the limit number of pins available on the UNO. This MCU, the ATmega2560, has several serial ports available, has 16 PWM capable channels, a 16 channel, 10-bit ADC (analog/digital converter) for analog inputs, and supports both the SPI (serial peripheral interface) and I2C communications buses through dedicated serial ports. In this project, we use 6 of the PWM capable outputs to communicated with the servo motors, and the 2-wire I2C communications bus to poll the IMU and receive the data.

This is a very small processor, and as such does not run any particular operating system. The software is programmed using the Arduino IDE which is a custom implementation of C++ that hides many of the intricacies that exist in embedded coding. Things like assembly and pointers are obfuscated through the use of Arduino libraries. This makes programming easier, though can make debugging difficult. The Arduino IDE itself does not have very robust debugging capabilities.

Input Device Description

The input device used is the MPU6050 6-DOF accelerometer and gyroscope from AdaFruit. There was another generic MPU6050 break-out board that was attempted, however upon purchasing 3 of them, none of them seemed to work (there were several forum entries online complaining of this very issue, so the AdaFruit unit was purchased instead).

The MPU6050 communicates over the I2C bus. This communications bus is address-based. This means that all devices on the bus have a specific address. In this instance, the address of the IMU is 0x68. To read from the device, there is a specific library that is imported into the Arduino IDE. This library permits the creation of a software object for this specific device, and simplifies the communication to simple "getEvent()" calls. However, this method itself merely polls the registers on the MPU6050. The input parameters do, in fact, use the pointer qualifier to place the register values from the MPU6050 directly into the memory of the ATmega2560. This can be seen in the code via the "&" parameter prior to a variable name.

The data obtained from the device is then printed via the serial connection to the host PC. This connection is a standard RS232 connection over USB and uses the first serial port available in the MCU.

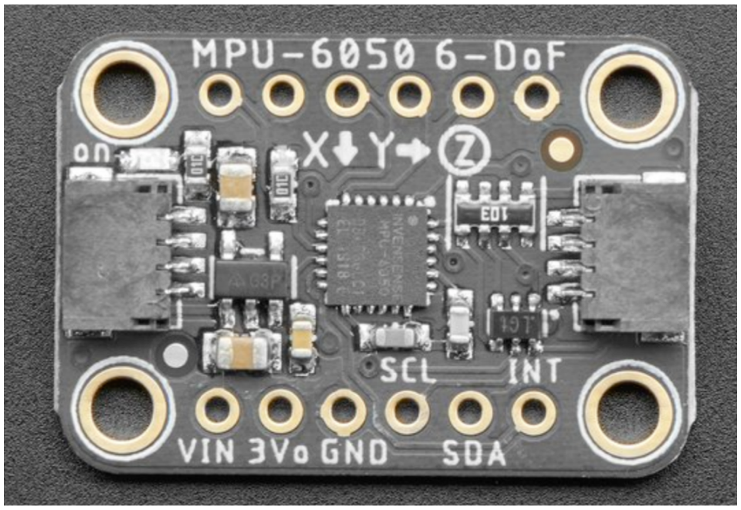

Figure 3: MPU6050 breakout board from AdaFruit. The "SCL" and "SDA" pins are the 2-wire I2C communications bus interface.

Output Device Description

The output device, or devices as is the case here, is a standard 5VDC servo motor. There are six of them, however they all operate the same way. Each motor has three leads going into it. There is a positive power lead, a negative power lead, and a signal lead. This signal lead is connected to one of the PWM capable output pins on the ATmega2560. Pulse-Width Modulation (PWM) is used to determine exactly where to position the servo arm. The servo itself has a motor, a gearbox, and a rotary encoder as well as built-in hardware that permits the motor to know the status of the output shaft. The PWM signal is given in microseconds, and the longer the signal, the larger the commanded angle is on the servo. For instance, the servos used are 270-degree servos. This means the arm can traverse 270 degrees of motion around the output shaft. A value of 1500 microseconds in the PWM signal would correspond to half this angle: 135 degrees. This is the servo's "Home" position. If a PWM signal longer than 1500 microseconds is provided to the servo, the motor is activated and the arm moves to a larger angle (up to 270, but in this project, it is limited to 225). Likewise, if a shorter PWM signal is sent, the arm moves to a smaller angle.

The way in which the software interfaces with the motors is again through the use of a library. There is a robust servo library available in the Arduino ecosystem. There is an option to use an explicit angle in degrees, however because of the physical arrangement of the servos, it becomes much easier to use the microsecond values when commanding the servo positions.

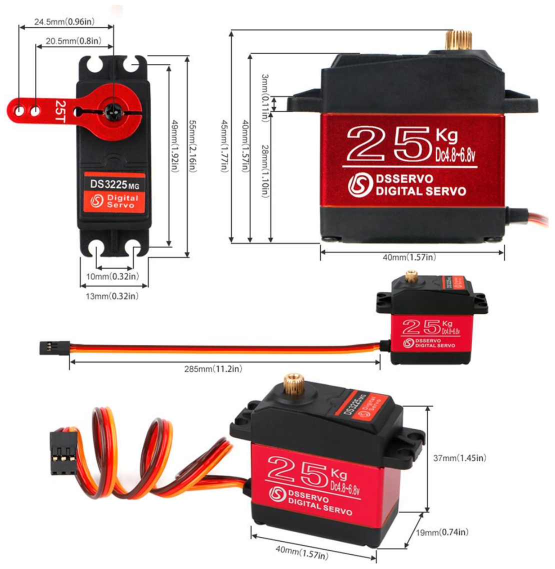

Figure 4: Representative Image of the servo motors used and their dimensions.

Programming Language

The programming language used is the Arduino programming language, which is largely based on C++ with specific elements added to make embedded systems programming a bit easier to attempt. The biggest advantage of this, for this project, is that the standard C math library can be used. The math library contains vital functions like trigonometric functions, exponents, and other operations. However, it did not provide a built-in means to facilitate linear algebra operations, so these were coded explicitly as functions within the software. This language was chosen over something like Python on a raspberry Pi because of its broad range of supported libraries and communities specifically geared towards electromechanical applications.

When compared to programming in assembly language, I would not even know where to begin to implement linear algebra functions like matrix multiplication in an assembly environment. Additionally, the specific protocols to interface with the I2C bus would become extremely difficult to keep track of. Certainly, programming using C++ is substantially easier than assembly.

Real-Time Constraints

As currently implemented, there are no specific real-time constraints present in the project. The device takes a commanded value as input, and subsequently moves to that position. However, there is a noticeable delay between when the command is given, and the actual movement occurs. This could be an issue however, for future work to implement a feedback control system using this platform, as a delay in processing can render the behavior unusable if immediate, or near immediate, feedback is needed. There are ample opportunities to optimize the code and simplify some of the calculations using some additional assumptions, which would reduce this probable delay, but no actual investigation into why the delay is noticeable has been done.

Security

This device doesn’t interface with anything other than a desktop PC. It is not inherently connected to a network and the only security risk would be exposure through the host PC.

However, should an attacker gain access to the host PC, there have been no efforts to harden or otherwise improve the security of the device itself. It will accept any 115200bps RS232 serial connection and accept commands.

Power Consumption

The power consumption was a major consideration for this device. Each servo motor has the capability to draw up to 1.8 Amperes of current. The limitation of the 5VDC out given in the Arduino cannot deliver more than 500mA worth of current. Certainly, this is not sufficient to properly drive a single servo, much less six of them. To fix this issue, an external 5VDC power supply was used. The power supply can provide up to 12A of current, however not at a 100% duty cycle. Fortunately, the current draw on each servo is proportional to the torque applied, and the typical torque experienced should not be an issue (the platform itself would break before any torque limit could be reached). In order to integrate this into the system, a custom Arduino “shield” was produced that permits the PWM signal from the MCU and the power from the PSU to flow to the servos. Additionally, the MCU itself is powered by the PSU alongside the servos.

Future Work

There is ample opportunity to further improve upon this device. In particular, I want to use it to implement a full closed-loop control system (the original scope of this project, that proved to be a bit too broad). Additionally, the platform could be used for positioning cameras or other devices (like a laser pointer and entertaining a feline companion). I would also like to explore how to implement this project in MATLAB, since MATLAB was used to help debug a lot of the mathematics issues without needing to compile and use a serial terminal. I also use Simulink every day at work for code generation and software models and it would be fun to explore designing and implementing an actual control system from the ground up like this in MATLAB. In any event, there’s lots of potential here for all sorts of fun uses.

Source Code

stewartPlatform.ino

```
/*
 *  Stewart Platform Firmware
 *
 *  This software is written to operate a 6-DOF Stewart Platform. It uses an MPU6050 IMU to provide sensor data.
 *  Full implementation of this data has yet to occur, however the results are displayed in a serial terminal.
 *
 *  This code uses derivations taken from the following sources:
 *
 *  https://content.instructables.com/ORIG/FFI/8ZXW/I55MMY14/FFI8ZXWI55MMY14.pdf
 *  https://www.xarg.org/paper/inverse-kinematics-of-a-stewart-platform/
 *
 *  For an in depth description of the mathematics, please see the above links. Note that the derivation given in
 *  source 1 (instructables link) has an error in the derivation that causes the formulas to fail.
 *
 *  When deriving the results for l^2 and s^2, the P vector values are used, and this is incorrect. They need to
 *  be the Q vector values (vector between the origin of the base and platform mounting point.) Another script
 *  has been written to more easily diagnose issues with the transformations involved. See mathTestScript.m in the
 *  parent directory. It must be run in MATLAB and mirrors the mathematics performed here.
 *
 *  Written by Garrett Sisk
 *  garrett@gsisk.com or msisk2@students.kennesaw.edu
 *
 */
#include "configuration.h"

#include <Arduino.h>
#include <Adafruit_MPU6050.h>
#include <Adafruit_Sensor.h>
#include <Wire.h>
#include <Servo.h>
#include <stdio.h>
#include <SPI.h>
#include <String.h>
#include <math.h>

// create variables/objects
double translation[3] = {0.0, 0.0, 0.0};
double rotation[3] = {0.0, 0.0, 0.0};
Adafruit_MPU6050 mpu;
Servo servoOne, servoTwo, servoThree, servoFour, servoFive, servoSix;
double L1, L2, L3, L4, L5, L6;
double alpha1, alpha2, alpha3, alpha4, alpha5, alpha6;

String inputString;


// instantiate matrix/vector variables
// Servo Output Shaft Position Vectors (in base frame)
double B_1[3][1] = {{X_B_1},{Y_B_1},{Z_B_1}};
double B_2[3][1] = {{X_B_2},{Y_B_2},{Z_B_2}};
double B_3[3][1] = {{X_B_3},{Y_B_3},{Z_B_3}};
double B_4[3][1] = {{X_B_4},{Y_B_4},{Z_B_4}};
double B_5[3][1] = {{X_B_5},{Y_B_5},{Z_B_5}};
double B_6[3][1] = {{X_B_6},{Y_B_6},{Z_B_6}};

// Platform Mounting Point Position Vectors (in platform frame)
double P_1[3][1] = {{X_P_1},{Y_P_1},{Z_P_1}};
double P_2[3][1] = {{X_P_2},{Y_P_2},{Z_P_2}};
double P_3[3][1] = {{X_P_3},{Y_P_3},{Z_P_3}};
double P_4[3][1] = {{X_P_4},{Y_P_4},{Z_P_4}};
double P_5[3][1] = {{X_P_5},{Y_P_5},{Z_P_5}};
double P_6[3][1] = {{X_P_6},{Y_P_6},{Z_P_6}};

// product matrices (intermittent results)
double RP_1[3][1] = {{0.0},{0.0},{0.0}};
double RP_2[3][1] = {{0.0},{0.0},{0.0}};
double RP_3[3][1] = {{0.0},{0.0},{0.0}};
double RP_4[3][1] = {{0.0},{0.0},{0.0}};
double RP_5[3][1] = {{0.0},{0.0},{0.0}};
double RP_6[3][1] = {{0.0},{0.0},{0.0}};

// summation matrices (intermittent results)
double Q_1[3][1] = {{0.0},{0.0},{0.0}};
double Q_2[3][1] = {{0.0},{0.0},{0.0}};
double Q_3[3][1] = {{0.0},{0.0},{0.0}};
double Q_4[3][1] = {{0.0},{0.0},{0.0}};
double Q_5[3][1] = {{0.0},{0.0},{0.0}};
double Q_6[3][1] = {{0.0},{0.0},{0.0}};

// arm length matrices (final results, prior to servo conversion)
double L_1[3][1] = {{0.0},{0.0},{0.0}};
double L_2[3][1] = {{0.0},{0.0},{0.0}};
double L_3[3][1] = {{0.0},{0.0},{0.0}};
double L_4[3][1] = {{0.0},{0.0},{0.0}};
double L_5[3][1] = {{0.0},{0.0},{0.0}};
double L_6[3][1] = {{0.0},{0.0},{0.0}};

void setup() {

  // configure the Inertial Measurement Unit (IMU)
  mpu.begin();
  mpu.setAccelerometerRange(MPU6050_RANGE_8_G);
  mpu.setGyroRange(MPU6050_RANGE_500_DEG);
  mpu.setFilterBandwidth(MPU6050_BAND_5_HZ);

  // set up the servo motors and set all of them to home position
  servoOne.attach(SERVO_ONE_PIN,SERVO_MIN_PWM, SERVO_MAX_PWM);
  servoOne.writeMicroseconds(SERVO_ZERO_PWM);

  servoTwo.attach(SERVO_TWO_PIN, SERVO_MIN_PWM, SERVO_MAX_PWM);
  servoTwo.writeMicroseconds(SERVO_ZERO_PWM);

  servoThree.attach(SERVO_THREE_PIN, SERVO_MIN_PWM, SERVO_MAX_PWM);
  servoThree.writeMicroseconds(SERVO_ZERO_PWM);

  servoFour.attach(SERVO_FOUR_PIN, SERVO_MIN_PWM, SERVO_MAX_PWM);
  servoFour.writeMicroseconds(SERVO_ZERO_PWM);

  servoFive.attach(SERVO_FIVE_PIN, SERVO_MIN_PWM, SERVO_MAX_PWM);
  servoFive.writeMicroseconds(SERVO_ZERO_PWM);

  servoSix.attach(SERVO_SIX_PIN, SERVO_MIN_PWM, SERVO_MAX_PWM);
  servoSix.writeMicroseconds(SERVO_ZERO_PWM);

  // Set up a serial port for debugging

Serial.begin(115200);// opens serial port, sets data rate to 115200 bps
if( !Serial ) {
  // wait for the serial to connect
}
Serial.println("Stewart Platform Initalizing....");
delay(1000);
}

void loop() {
  //----------------------------------------------------------------------------------------------------
  //
  //          Intertial Measurement Unit Inputs and Time Step
  //
  //----------------------------------------------------------------------------------------------------

  // get start time for measurement
  int startTime = millis();

  // measure the IMU status
  sensors_event_t a, g, temp;
  mpu.getEvent(&a, &g, &temp);

  // get X acceleration
  double xAccel = X_OFFSET + a.acceleration.x;
  // get Y acceleration
  double yAccel = Y_OFFSET + a.acceleration.y;
  // get Z acceleration
  double zAccel = Z_OFFSET + a.acceleration.z;
  // get roll acceleration (around X-axis)
  double xRoll = X_ROLL_OFFSET + g.gyro.x;
  // get pitch acceleration (around Y-axis)
  double yRoll = Y_ROLL_OFFSET + g.gyro.y;
  // get heading acceleration (around Z-axis)
  double zRoll = Z_ROLL_OFFSET + g.gyro.z;

  // account for gravity effect in Z-Axis
  zAccel = zAccel + 9.81; // m/s^2

  // get end time for measurement
  int endTime = millis();

  // calculate the time step
  int timeStep = endTime - startTime;

  //----------------------------------------------------------------------------------------------------
  //
  //          Control Equation Solution Based on IMU Inputs (Second Order Integral)
  //
  //----------------------------------------------------------------------------------------------------

  // translational conversions
  double xVelocity = V_0 + xAccel*timeStep;
  double xPos = S_0 + xVelocity*timeStep + (1/2)*xAccel*(timeStep^2);
  double yVelocity = V_0 + yAccel*timeStep;
  double yPos = S_0 + yVelocity*timeStep + (1/2)*yAccel*(timeStep^2);
  double zVelocity = V_0 + zAccel*timeStep;
  double zPos = S_0 + zVelocity*timeStep + (1/2)*zAccel*(timeStep^2);

  // rotational conversions
  double xOmega = OMEGA_0 + xRoll*timeStep;
  double thetaX = THETA_0 + xOmega*timeStep + (1/2)*xRoll*(timeStep^2);
  double yOmega = OMEGA_0 + yRoll*timeStep;
  double thetaY = THETA_0 + yOmega*timeStep + (1/2)*yRoll*(timeStep^2);
  double zOmega = OMEGA_0 + zRoll*timeStep;
  double thetaZ = THETA_0 + zOmega*timeStep + (1/2)*zRoll*(timeStep^2);

  //----------------------------------------------------------------------------------------------------
  //
  //          Translation of Raw Position Commands to PWM Signals for Servo Actuation
  //
  //----------------------------------------------------------------------------------------------------
  // set desired position
  if (Serial.available() > 0) {
      inputString = Serial.readStringUntil("\n");
  }

  //translation[1] = (double)inputString.toInt();
  rotation[1] = (double)inputString.toInt();

  // Set values in the
  double T[3][1] = {{translation[0]}, {translation[1]}, {translation[2]+H_0}};

  // Compute the rotational matrix
  double Phi = radians(rotation[0]);
  double Theta = radians(rotation[1]);
  double Psi = radians(rotation[2]);

  double R[3][3] = {
              {cos(Phi)*cos(Theta),   -sin(Phi)*cos(Psi)+cos(Phi)*sin(Theta)*sin(Psi),
sin(Phi)*sin(Psi)+cos(Phi)*sin(Theta)*cos(Psi)},
              {sin(Phi)*cos(Theta),   cos(Phi)*cos(Psi)+sin(Phi)*sin(Theta)*sin(Psi),   -
cos(Phi)*sin(Psi)+sin(Phi)*sin(Theta)*cos(Psi)},
              {-sin(Theta),           cos(Theta)*sin(Psi),                              cos(Theta)*cos(Psi)}
             };

  // Compute product of P_vector and rotational matrix (Note: function equal to P_R_b * P_n = P_R_b_P_n)
  multiplyMatrices(R, P_1, RP_1);
  multiplyMatrices(R, P_2, RP_2);
  multiplyMatrices(R, P_3, RP_3);
  multiplyMatrices(R, P_4, RP_4);
  multiplyMatrices(R, P_5, RP_5);
  multiplyMatrices(R, P_6, RP_6);

  // Add the input translational vector to each of the above matrices (Note: T_vector + P_R_b_P_n = T_P_R_b_P_n)
  addMatrices(T, RP_1, Q_1);
  addMatrices(T, RP_2, Q_2);
  addMatrices(T, RP_3, Q_3);
  addMatrices(T, RP_4, Q_4);
  addMatrices(T, RP_5, Q_5);
  addMatrices(T, RP_6, Q_6);

  // Subtrace the B vector from each of the above matrices (Note: T_P_R_b_P_n - B_n = L_n)
  subtractMatrices(Q_1, B_1, L_1);
  subtractMatrices(Q_2, B_2, L_2);
  subtractMatrices(Q_3, B_3, L_3);
  subtractMatrices(Q_4, B_4, L_4);
  subtractMatrices(Q_5, B_5, L_5);
  subtractMatrices(Q_6, B_6, L_6);

  // calculate the raw lengths of the arms
  L1 = vectorMagnitude(L_1);
  L2 = vectorMagnitude(L_2);
  L3 = vectorMagnitude(L_3);
  L4 = vectorMagnitude(L_4);
  L5 = vectorMagnitude(L_5);
  L6 = vectorMagnitude(L_6);

  // calculate alpha for each servo
  alpha1 = calculateServoAngle(Q_1, B_1, L1, BETA_ONE);
  alpha2 = calculateServoAngle(Q_2, B_2, L2, BETA_TWO);
  alpha3 = calculateServoAngle(Q_3, B_3, L3, BETA_THREE);
  alpha4 = calculateServoAngle(Q_4, B_4, L4, BETA_FOUR);
  alpha5 = calculateServoAngle(Q_5, B_5, L5, BETA_FIVE);
  alpha6 = calculateServoAngle(Q_6, B_6, L6, BETA_SIX);

  // move the actual servo
  servoOne.writeMicroseconds(convertAngleToPWM(alpha1, 1));
  servoTwo.writeMicroseconds(convertAngleToPWM(alpha2, 2));
  servoThree.writeMicroseconds(convertAngleToPWM(alpha3, 3));
  servoFour.writeMicroseconds(convertAngleToPWM(alpha4, 4));
  servoFive.writeMicroseconds(convertAngleToPWM(alpha5, 5));
  servoSix.writeMicroseconds(convertAngleToPWM(alpha6, 6));
}
```

```
//--------------------------------------------------------------------------------------------------------
//
//          Utility and Mathematic Functions
//
//--------------------------------------------------------------------------------------------------------

void multiplyMatrices(double A[3][3], double B[3][1], double C[3][1]) {
  // this method takes three matrices as inputs, performs the matrix multiplication A*B and adds results as
  // elements to C. Only a single loop is needed since the output is a 3x1 vector.
  for (int i = 0; i < 3; i++) {
    C[i][0] = (A[i][0]*B[i][0]) + (A[i][1]*B[i][0]) + (A[i][2]*B[i][0]);
    }
  }

void addMatrices(double A[3][1], double B[3][1], double C[3][1]) {
  // this method takes two column vectors of length 3 and adds them together
  for (int i = 0; i < 3; i++) {
    C[i][0] = A[i][0] + B[i][0];
  }
}

void subtractMatrices(double A[3][1], double B[3][1], double C[3][1]) {
  // this method takes two column vectors of length 3 and subtracts them.
  for (int i = 0; i < 3; i++) {
    C[i][0] = A[i][0] - B[i][0];
  }
}

double vectorMagnitude(double A[3][1]) {
  // this method returns the scalar magnitude of the vector provided
  double output = sqrt( pow(A[0][0],2) + pow(A[1][0],2) + pow(A[2][0],2) );
  return output;
}

double calculateServoAngle(double Q_vector[3][1], double B_vector[3][1], double inputL, double inputBeta) {
  // calculate "Big L"
  double bigL = pow(inputL,2) - ( pow(LINKAGE_ARM,2) - pow(SERVO_ARM,2) );
  // calculate "Big M"
  double bigM = 2 * SERVO_ARM * (Q_vector[2][0] - B_vector[2][0]);
  // calculate "Big N"
  double bigN = 2 * SERVO_ARM * (cos(radians(inputBeta))*(Q_vector[0][0] - B_vector[0][0]) +
sin(radians(inputBeta))*(Q_vector[1][0] - B_vector[1][0]));
  // calculate alpha
  return asin(bigL/sqrt(pow(bigM,2)+pow(bigN,2))) - atan(bigN/bigM);
}

double convertAngleToPWM(double inputAngle, int inputType) {
  // if the input type is 1, the servo motor is an odd entry. If the input type is 2, the servo motor is an even entry
  if (inputType == 2 || inputType == 4 || inputType == 6) { // if the type is even
    return SERVO_ZERO_PWM - (inputAngle - radians(ALPHA_0)) * SERVO_CONVERT;
  } else {
    return SERVO_ZERO_PWM + (inputAngle - radians(ALPHA_0)) * SERVO_CONVERT;
  }
}
```

configuration.h

```
// This header file sets all the physical parameters necessary for the Stewart Platform to operate.

#ifndef _CONFIGURATION_H
#define _CONFIGURATION_H

// Initial Kinematic Conditions
#define S_0                 0.0 // meters
#define V_0                 0.0 // meters/sec
#define A_0                 0.0 // meters/sec^2
#define THETA_0             0.0 // radians
#define OMEGA_0             0.0 // radians/sec
#define ALPHA_0             0.0 // radians/sec^2

// IMU Calibration Parameters
#define X_OFFSET            0.0
#define Y_OFFSET            0.0
#define Z_OFFSET            0.0
#define X_ROLL_OFFSET       0.0
#define Y_ROLL_OFFSET       0.0
#define Z_ROLL_OFFSET       0.0

// Servo Parameters
#define SERVO_ONE_PIN       15
#define SERVO_TWO_PIN       14
#define SERVO_THREE_PIN     2
#define SERVO_FOUR_PIN      3
#define SERVO_FIVE_PIN      4
#define SERVO_SIX_PIN       5
#define SERVO_ZERO_PWM      1500
#define SERVO_MAX_PWM       2000
#define SERVO_MIN_PWM       1000
#define SERVO_CONVERT       318.31


// Physical Dimension Parameters - all values given in millimeters. Set "B" is defined in the base reference frame.
// Set "P" is defined in the top reference frame. All values are given from their respective origins, and have been
taken
// from the Solidworks Assembly using the "Evaluate -> Measure" tool.

// NOTE: IT IS VITAL THAT THE PARAMETERS BELOW ARE CORRECT. COMPUTATIONAL ERRORS CAN OCCUR IF THE VALUES DO NOT
CORRESPOND TO
//        ACTUAL PHYSICAL MEASUREMENTS.

// NOTE: THE BASE AND PLATFORM COORDINATE SYSTEMS ARE COPLANAR IN X AND Y, BUT ARE SEPARATED BY A DISTANCE H_0 IN THE
Z-AXIS

// Linkage Length Parameters
#define SERVO_ARM           24.0
#define LINKAGE_ARM         122.5
#define H_0                 120.7606
#define ALPHA_0             10.7332 // degrees

// For Servo One
#define X_B_1               73.5
#define Y_B_1               -38.25
#define Z_B_1               0.0
#define X_P_1               75.0
#define Y_P_1               -7.43
#define Z_P_1               0.0
#define BETA_ONE            -90.0// degrees
// For Servo Two
#define X_B_2               73.5
#define Y_B_2               38.25
#define Z_B_2               0.0
#define X_P_2               75.0
#define Y_P_2               7.43
#define Z_P_2               0.0
#define BETA_TWO            90.0 // degrees
// For Servo Three
#define X_B_3               3.62
#define Y_B_3               82.78
#define Z_B_3               0.0
#define X_P_3               -31.06
#define Y_P_3               68.67
#define Z_P_3               0.0
#define BETA_THREE          30.0 // degrees
// For Servo Four
#define X_B_4               -69.88
#define Y_B_4               44.53
#define Z_B_4               0.0
#define X_P_4               -43.94
#define Y_P_4               61.24
#define Z_P_4               0.0
#define BETA_FOUR           30.0 // degrees
// For Servo Five
#define X_B_5               -69.88
```

```
#define Y_B_5             -44.53
#define Z_B_5             0.0
#define X_P_5             -43.94
#define Y_P_5             -61.24
#define Z_P_5             0.0
#define BETA_FIVE         -30.0 // degrees
// For Servo Six
#define X_B_6             3.62
#define Y_B_6             -82.78
#define Z_B_6             0.0
#define X_P_6             -31.06
#define Y_P_6             -68.67
#define Z_P_6             0.0
#define BETA_SIX          -30.0 // degrees

#endif // _CONFIGURATION_H
```