3-Axis Camera Gimbal Stabilizer
ME 6404: Final Project

Diwakar Singh, Isabella Hoskins, Pablo Aguilar

December 2, 2019

Abstract

The purpose of this project is to implement a feedback control system for 3-axis camera gimbal stabilizer. The goal is to build an active stabilizer that is lightweight and useful for stabilizing a camera. The system is based on pairing electric servo motors with a position sensor in order to compensate for unwanted movements. This makes sure that the camera stays level with the ground at all times and can give stabilized video footage. To simplify the problem, the platform motions are uncoupled in three directions: yaw (ψ), pitch (ϕ), and roll (θ). The stabilizer arms are 3D printed with PLC material and are paired with the servo motors to establish two axes that are perpendicular to each other. The entire device is hand held and is operated with two handles which makes for easy user operation. The final product performed relatively well with the ability to compensate motions around the pitch and roll axes. The system turned out quite bulky but lightweight enough to be controlled easily with both hands. The system could also crash occasionally, requiring a reset to resume operation. All in all, the system operated as desired and thus fulfilled its purpose. For further development, the primary area of improvement is the regulator to make the compensation smoother. Also, the system could be shrunk down significantly to reduce size and weight.

1 Introduction

1.1 Objective

The overall objective is to design a system that meet the following criteria. The first is to establish a clear statement of the system and objectives for its performance. The next is to develop a model of the system and have a fundamental understanding on the simplifications made on the actual system. The third is to design and implement a control system that meets the set performance specifications. Then, analysis on the robustness of the control system must be made in order to know the limitations of the system. Lastly, a hardware verification must be made to prove that the system executes as intended.

1.2 Motivation

The 3-axis gimbal stabilizer was chosen because it met all of the criteria and could be built and tested in a month. The goal for the stabilizer is to keep the top platform, which a camera can be attached, level to a set reference plane within a 3D space. The reference plane used to secure the camera in place is set during the calibration phase. After the calibration, the intended motion of the system is read by an IMU sensor. The microcontroller generates commands for motors along roll, pitch, and yaw to stabilize the platform for any intended and unintended motion. These adjustments are required to not overshoot and be fast enough to keep up with slow to moderate movements for the disturbances. Additionally, the system model and its dynamics were found through the use of the recursive least squares method. The system model could be used for future simulation and zero error tracking control.

1.3 Design Choices and Limitations

A few limitations were made due to time constraint, budget, and potential complexity of designing the gimbal stabilizer. Originally, the design included DC motors with encoders to actuate the motions along roll, pitch, and yaw axis with minimal constraints. Instead, servo motors were used as inexpensive, easy to use and no additional controllers were needed for implementation. However, this lead to the movements along the three axis as MG996R servo motor to be restrained up to 90° in both directions. Hence, the gimbal system can compensate for movements along the roll, pitch, and yaw axis for up to $\pm 90^\circ$. Another hardware restriction is that the operating speed of MG996R servo motor is $0.14s/60^\circ$, which cannot fast enough to mitigate large disturbances.

2 Experimental Details

2.1 Mechanical Construction

Figure 1 shows the CAD model of the 3-axis gimbal stabilizer designed for this project. The prototype was printed with PCA plastic and assembled with M3 bolts. Three MG996R servo motors are attached to the gimbal and are controlled with Texas Instrument's MSP432 microcontroller and powered by four AA batteries in series. On the small platform at a roll axis, a camera can be placed. The MPU9250 IMU sensor was placed on the handle to simplify the wires placement and to read only the intentional displacements.. An IMU sensor reads the angles yaw(ψ), pitch (ϕ), and roll(θ). To make the system as compact as possible, the mounting arms were designed to place the center of mass close to the midpoint of roll- and pitch- axis.

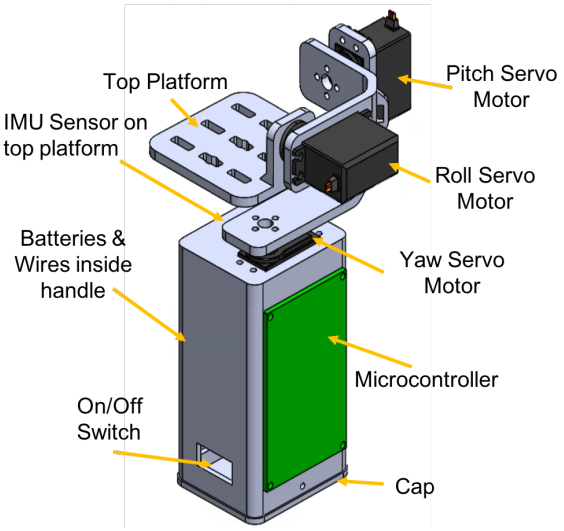


Figure 1: Three-axis gimbal system

2.2 Electronic Assembly

The full schematic for how the motor drivers were plugged in can be seen in Figure 2. The IMU sensor sends data to the master microcontroller which then processes the data continuously which in turns generates commands for the three servo motors as seen in the block diagram in Figure 3.

2.3 IMU Sensor

The IMU sensor used in this project was the MPU-9250 by Invensense. It consists of 3-axis accelerometer, a 3-axis gyroscope, and a 3-axis magnetometer. Moreover, six 16-bit analog-to-digital converters digitizes the sensor outputs. It uses the I2C communication with other devices and supports data transfer speeds up to $400kb/s$. Programmed into the chip is a digital processing unit (DMP) that performs computations of the raw data which allows for the position values to be collected by the MCU quicker. Additionally, different sensitivity can be set for the gyroscope and the accelerometer readings. For this project, default settings were used, i.e. $\pm 250°/s$ for the gyroscope and $\pm 8g$ for the accelerometer.

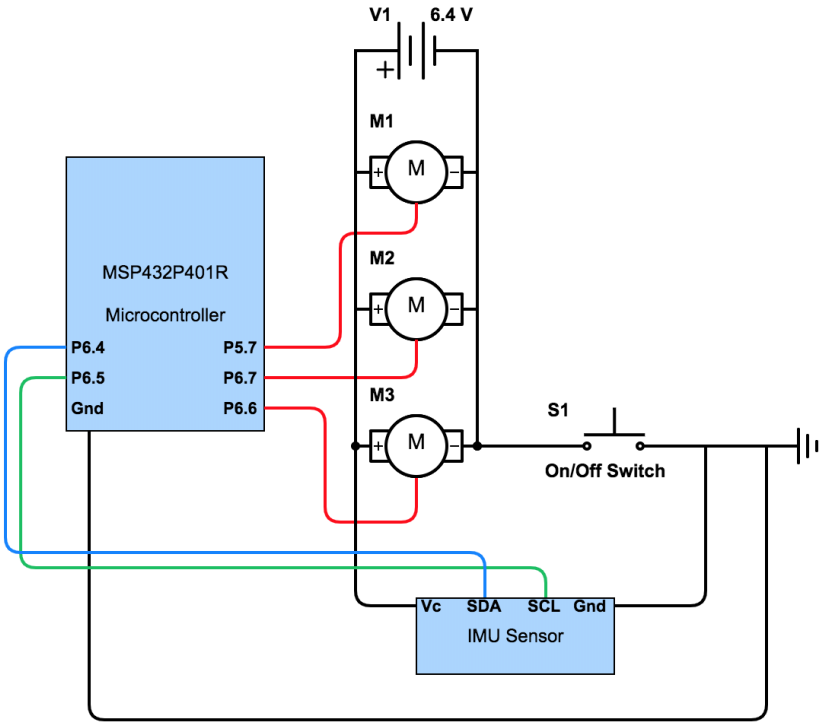


Figure 2: Schematic of circuit hardware setup

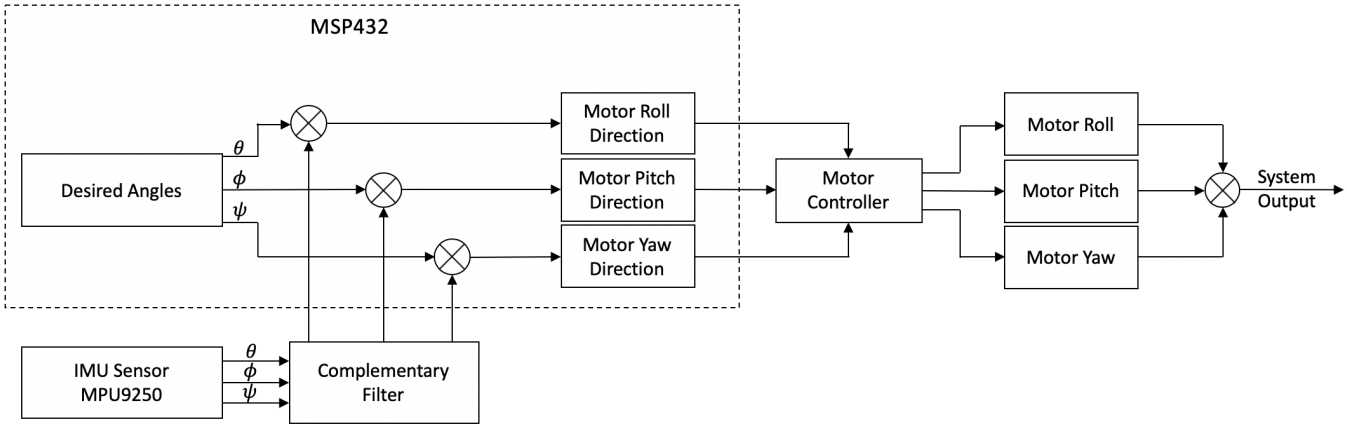


Figure 3: Block diagram of the complete system

2.4 Servo Motor

The servo motors used in the project is a metal gear MS996R servo motor with a maximum stall torque of $11kg/cm$. The motor rotates about $\pm 90°$ around its neutral position based on the duty cycle of the PWM wave supplied to its signal pin. For MG996R servo motors, the valid duty cycle is between 5% and 10% with $20ms$ time period. A duty cycle of 7.5% is to be applied for motor to stay at neutral position.

2.5 Programming

The programming was done in Code Composer Studio (CCS) 9.2.0, which is an integrated development environment (IDE) that supports Texas Instrument's Microcontroller. It includes an optimizing C/C++ compiler, source code editor, project build environment, debugger, profiler, and many other features. The programming language used is Embedded C and the microcontroller used is MSP432P401R.

3 Theory

3.1 The Active Stabilizer

The two major types of stabilizers available today are passive and the active stabilizers. While the passive stabilizer relies on remaining balanced with counterweights, the active stabilizer uses motors to keep the camera in the correct orientation and was chosen for this project. To counteract unwanted movements,the system needs to identify if the camera is rotated out of its reference plane. This is most often done with an inertial measurement unit (IMU) sensor that senses acceleration and angular velocity to determine the angle of the camera in the three dimensional space.

An intuitive way of presenting the angle is to separate it into three different axes, namely pitch, yaw, and roll. This definition is often used to describe rotation of aircraft but it translates well to the cameras movements as well. The orientation of the axes are shown in Figure 4. All three axes are perpendicular to each other and therefore all possible rotations can be decomposed to its roll, pitch and roll components.

For an active camera stabilization system, the pitch and roll axes are most important because unintentional movements around these axes make the recorded video seem shaky. Stabilization in the yaw axis can also be desirable in cases where the movement should be smooth and consistent, for example in panoramic shots.

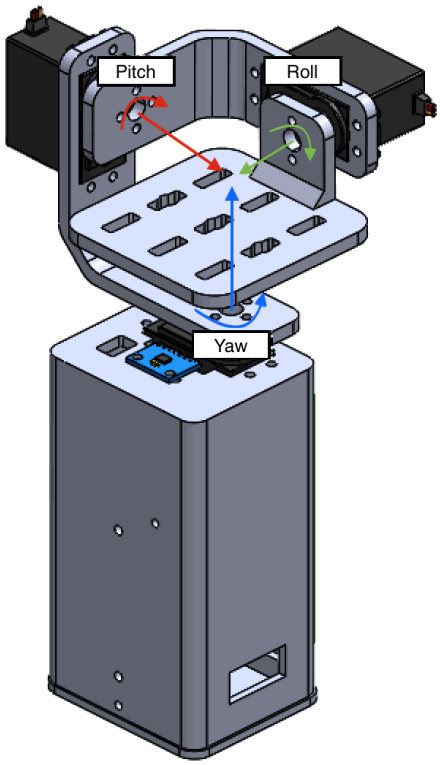


Figure 4: Roll, pitch, and yaw axes

3.2 IMU Sensor Placement

The placement of the IMU sensor significantly impacts the method of controlling the system. There are two primary options to place the sensor, either on the *camera side* or the *support side*.

If the IMU is placed on the camera platform, the camera and IMU share the same reference plane. This results in that the three dimensional angles of the camera platform are identical to the angles of the sensor. By knowing the exact orientation of the camera at all times, the position of the camera can be controlled with great precision. However, all angular compensations form the motors will move the IMU and therefore impact the sensor data. If the control system is insufficient, abrupt and jerky movements caused by the motors can lead to faulty sensor values and the camera platform will easily vibrate around its reference point.

Alternatively, the sensor can be placed on the support side of the system, which in this case is the handle. This positioning enables the angle of the platform and the sensor to rotate independently of each other. This approach greatly reduces the risk of vibrations on the camera side as the motors will not impact the sensor readings. Unfortunately, this sensor placement aggravates the ability to determine the camera angle. Because the camera and sensor do not share the same reference plane, the angle of the platform has to be estimated from the sensor data. This is done effectively and work well with proper motor control, but small estimation errors could lead to permanent angle offsets until the system is re-calibrated.

4 Orientation and Sensor Fusion

4.1 Coordinate System

In order to keep the camera platform stabilized in a fixed position it is important that the camera position can be measured relative to something that is fixed, namely a fixed reference frame (inertial frame). In this project, the North-East-Down (NED) reference frame has been used for orientation, where "down" is aligned with the direction of gravity. The sensor measurements, however, cannot always be measured directly in the inertial frame because the sensors measure quantities relative to their own frame of reference, which is not fixed in space, see Figure 5. This means that a conversion of measurement data has to be done to get an inertial frame representation. A way to do this is to use Euler angles as a representation of how the sensor frame is oriented relative to the inertial frame.

Figure 5: Inertial reference frame relative to unfixed sensor frame [1]

4.2 Euler Angles

Euler angles are used to describe the orientation of an inertial frame relative to a moving frame. The angles are denoted ψ, θ and ϕ, and represents the result of three rotations about different axes. The rotations are called yaw, pitch, and roll and can be represented as rotation matrices as seen in Equation 1 [2]. The lower and upper indices represent the starting and ending frame for the rotation. To get a rotation matrix representing all three rotations, all three rotation matrices have to multiplied in the order of yaw, pitch, and roll. This means that the the yaw rotation matrix rotates a vector from the inertial frame (I) to a so called the first intermediate frame ($In1$). The pitch rotation matrix rotates from $In1$ to the second intermediate frame ($In2$) and lastly the the roll rotation matrix rotates from $In2$ to the sensor frame (S). This is expressed mathematically in Equation 2 and the final rotation matrix for the rotation sequence can be seen in Equation 3, where c and s represents cosine and sine respectively.

$$\mathbf{R}_{I}^{In1}(\psi) = \begin{pmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{1a}$$

$$\mathbf{R}_{In1}^{In2}(\theta) = \begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{pmatrix} \tag{1b}$$

$$\mathbf{R}_{In2}^{S}(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & \sin\psi \\ 0 & -\sin\psi & \cos\psi \end{pmatrix} \tag{1c}$$

$$\mathbf{R}_I^S(\phi,\theta,\psi) = \mathbf{R}_{In2}^S(\phi)\mathbf{R}_{In1}^{In1}(\theta)\mathbf{R}_I^{In1}(\psi) \tag{2}$$

$$\mathbf{R}_I^S(\phi,\theta,\psi) = \begin{pmatrix} c(\theta)c(\psi) & c(\theta)s(\psi) & -s(\theta) \\ c(\psi)s(\theta)s(\phi) - c(\phi)s(\psi) & c(\psi)c(\psi) + s(\theta)s(\phi)s(\psi) & c(\theta)s(\phi) \\ c(\psi)c(\psi)s(\theta) + s(\phi)s(\psi) & c(\phi)s(\theta)s(\psi) - c(\psi)s(\phi) & c(\theta)c(\phi) \end{pmatrix} \tag{3}$$

Equation 3 can be used for estimating the pitch and roll angles from accelerometer readings. Gyro readings, however, cannot be represented in the inertial frame using Equation 3. For this the rotation matrix in Equation 4 is needed. The axes of the IMU-sensor frame are denoted X_s, Y_s and Z_s and when the camera is horizontal Zs is in the opposite direction of gravity. X_s is aligned with the pitch axis and Ys is aligned with the roll axis as shown in Figure 4. So, if the angular velocities in the sensor frame are denoted by ω_x, ω_y, and ω_z then the Euler angle rates can be calculated as in Equation 5.

$$\mathbf{D}(\phi,\theta,\psi) = \begin{pmatrix} 1 & sin(\phi)tan(\theta) & cos(\phi)tan(\theta) \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & \sin(\phi)/cos(\theta) & cos(\phi)/cos(\theta) \end{pmatrix} \tag{4}$$

$$\begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} = \begin{pmatrix} 1 & sin(\phi)tan(\theta) & cos(\phi)tan(\theta) \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & \sin(\phi)/cos(\theta) & cos(\phi)/cos(\theta) \end{pmatrix} \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} \tag{5}$$

By integrating Equation 5, it is possible to estimate all three angles with the gyroscope An important thing to notice about Equation 5 is that a pitch angle of 90° will cause some matrix elements to diverge towards infinity. This is a phenomenon called gimbal lock which sets a limitation for Euler angles.

4.3 Complementary Filter

Sensor fusion is used to combine measurements from several sensors in order to get more reliable information. When measuring orientation with gyroscopes and accelerometers each sensor has its disadvantages. The accelerometer does not provide reliable measurements when the attitude is changing and the gyroscope suffers from a time varying bias. A complementary filter is to combine the best part of each sensor and obtain an estimate that better corresponds to the real world. the frequency domain it can be described by Equation 6 [5].

$$\theta(s) = \frac{1}{1+\tau s}A(s) + \frac{\tau s}{1+\tau s}G(s) = \frac{A(s) + \tau G(s)}{1+\tau s} \tag{6}$$

where $A(s)$ and $G(s)$ are the frequency domain representation of accelerometer signal and the gyroscope signal, $\theta(s)$ the Euler angle and τ the time constant of each filter. So the idea with the complementary filter is to low-pass filter the data provided by the accelerometer, integrate and high-pass filter the data from the gyroscope and then combine both readings to give a more reliable estimate. The noisy accelerometers readings are filtered out and complemented with gyroscope data while the time-varying bias of the gyroscope is also filtered out and complemented by accelerometer data. Thus, the filter provides an estimate that is based on the best measurements of each sensor. Since Equation 6 is in the continuous time domain and cannot be directly implemented on a computer, a backward difference is used to discretize it. Equation 6 then becomes

$$\theta(k) = \alpha(\theta(k-1) + \omega(k)h) + (1-\alpha)a(k) \tag{7}$$

where $\alpha = \tau/(h+\tau)$. Equation 7 can easily be implemented on a computer, and requires little computational capacity due to its simplicity.

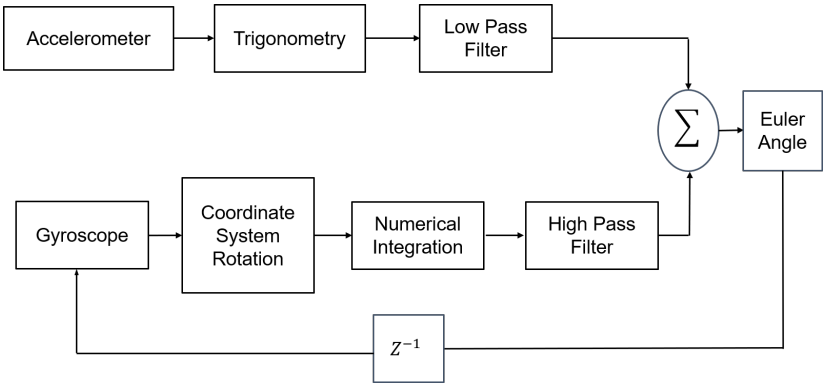


Figure 6: The schematic diagram of reading IMU sensor

In the hardware, the complementary filter has been implemented according to Equation 7 but the filter does not work directly with raw sensor data. Some processing has been done before the filtering, see Figure 6. In order to obtain an Euler angle from the accelerometer trigonometry has to be used. Equation 8 can be used to convert the acceleration readings a_x, a_y, and a_z from IMU sensor to pitch ϕ, and roll θ angles [3].

$$\theta = tan^{-1}(\frac{a_x}{\sqrt{a_y^2 + a_z^2}}) \tag{8}$$

$$\phi = tan^{-1}(\frac{-a_y}{\sqrt{a_x^2 + a_z^2}}) \tag{9}$$

It is not possible to estimate the ψ angle using an accelerometer as it does not change the static readings. The gyroscope data also has to be processed to get an earth frame representation, see Equation 5.

5 Trajectory Control

A trajectory is often set by the constraints on the process or actuators. Furthermore, trajectory generation can be used in both open- and closed- loop systems. Since, there is no feedback from the servo motors, trajectory control was implemented for the open loop control system only. Figure 7 shows the control block of the open-loop trajectory system. The trajectory chosen was constant yaw rotation to simulate a camera capturing a panoramic view. The disturbances were rejected either in the roll-axis or along the pitch-axis independently.

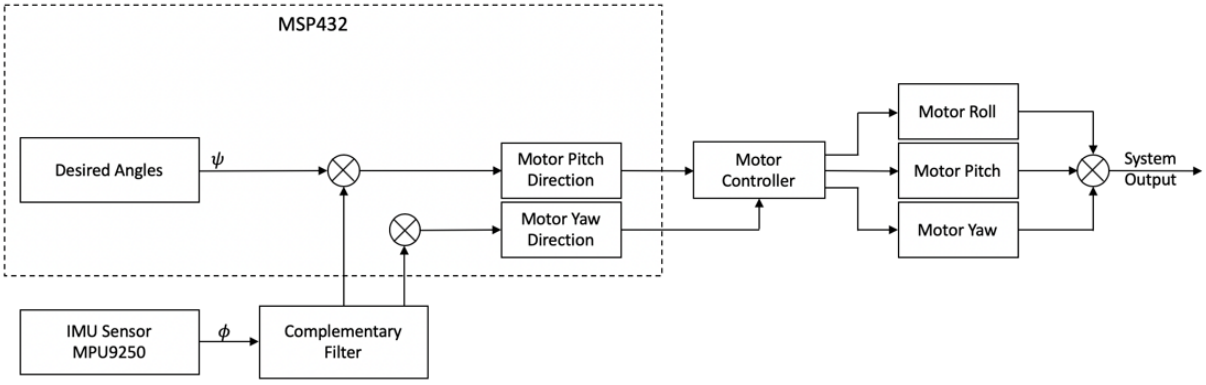


Figure 7: Control block of open loop trajectory system

6 Recursive Least Square

Past research shows that R/C servo motors have a third order response between motor rotation angle α and command voltage V [4]. The discretized relationship is given in Equation 10.

$$\frac{\alpha}{V} = \frac{gz^{-1} + hz^{-2} + iz^{-3}}{1 + az^{-1} + bz^{-2} + cz^{-3}} \tag{10}$$

where g, h, i, a, b, c are unknown parameters. By transforming Equation 10 to the time domain using a zero hold first order method, the current output α is approximated by Equation 11.

$$\alpha_{estimated} = gV(k-1)+hV(k-2)+iV(k-3)-a\alpha(k-1)-b\alpha(k-2)-c\alpha(k-3) = \beta_k^T \gamma_k \tag{11}$$

where $k-i$ is the reading i time steps before the k reading, β_k is a vector of previous α and V, and γ is the vector of unknown parameters. A recursive least square (RLS) method was implemented to estimate the parameters with the following equations.

$$\gamma_k = \gamma_{k-1} + P_k \beta_k E \tag{12}$$

where E is the error $\alpha - \alpha_{estimated}$, and P_k is the covariance matrix defined as

$$P_k = [P_{k-1} + P_{k-1}\beta_k(I + \beta_k^T P_{k-1}\beta_k)^{-1}\beta_k^T](1/\lambda) \tag{13}$$

where P_{k-1} is the previous covariance matrix and λ is the scalar factor that forces E to go to zero before P_k. A λ of 0.98 was chosen for implementation using trial and error. It is assumed that at low frequency disturbances, the motor angular displaces equal handle angular displacement in the opposite direction since the gimbal can stabilize slow movement. Thus, the IMU sensor readings can be used to estimate the motor rotation.

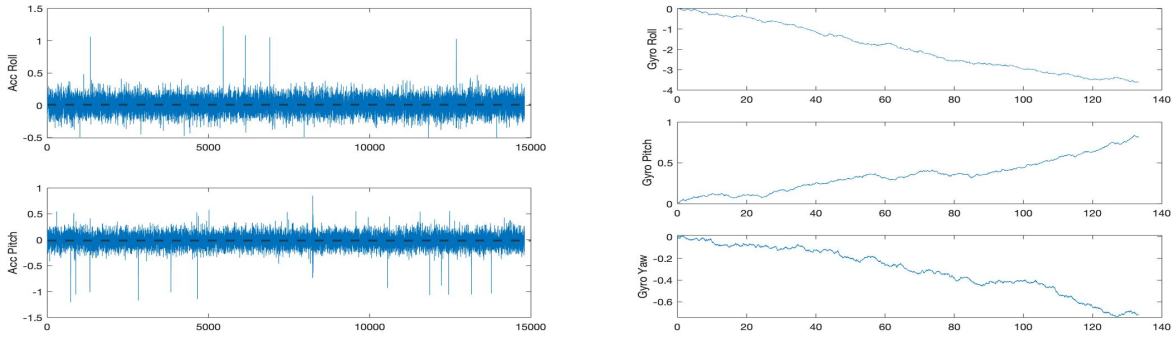
7 Results and Discussion

7.1 Prototype

A prototype has been built as shown in Figure 4. It has been tested and evaluated for active stabilization. The results have shown that it is able stabilized uncoupled motions in all the three axes to a certain extent. The motion along pitch axis is very smooth compared to the other two axis. For the pitch axis, the motion is smooth for small angles. When the pitch angle approaches zero, gimbal lock occurs as explained in Section 4.2. Also, the third axis being the yaw axis is not functioning the way it was intended during longer runs since the magnetometer component of the IMU sensor was not used in the final construction due to limitation of microcontroller computation capability.

7.2 IMU Sensor

The acquired raw data signal from the IMU sensor was processed and filtered to obtain a reliable position of the camera angle. The accelerometer data was very noisy and the gyroscope data drifts over time. To solve this problem a sensor-fusion algorithm was introduced by applying a complementary filter to the accelerometer and gyroscope data. Figure **??** highlights the noisy accelerometer data and the drift from the gyroscope data. The drift from the gyroscope data is caused by an integration error and occurs even though the bias is zero. The drift is a result of accumulating white noise of the integration reading.



(a) Accelerometer noise

(b) Gyroscope Drift

Figure 8: Accelerometer and Gyroscope angle from IMU sensor without complementary filter

By implementing the sensor fusion algorithm, the data from the gyroscope and accelerometer are merged into one signal by giving the signals different weights, according to Equation 6. For the complementary filter, the weight of the accelerometer data was set to 4% and the gyroscope was set to 90% to obtain a smooth and filtered signal as can be seen in Figure 9.

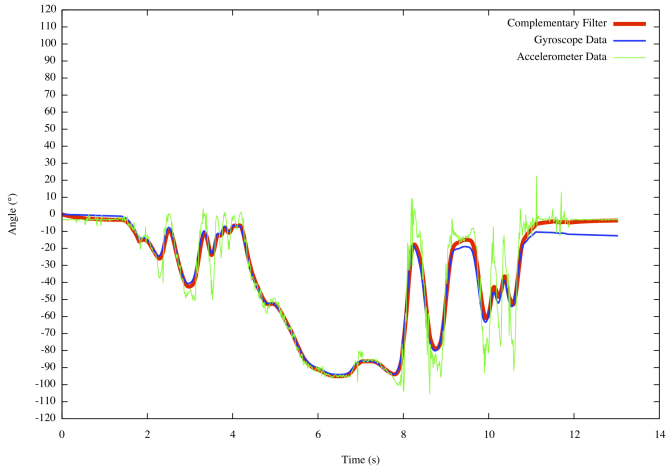


Figure 9: Filtered signal using complementary filter

7.3 Trajectory Control

It was observed that the controller stabilized the platform with low frequency disturbances in the pitch, roll, and yaw while performing the trajectory. However, the controller was unable to accurately stabilize the platform for disturbances for coupled motions. This limitation stems from the assumption that the controller considers the angle independently. Further-

more, inverse kinematics is needed to obtain the true position of the platform, but was not implemented due to the computational limitations of the MSP432 microcontroller.

7.4 Recursive Least Square

The results of the RLS implementation is found in Figure 10. All estimated parameters converge to a singular value while the error goes to zero. The final model for each motor is summarized in Table 1.

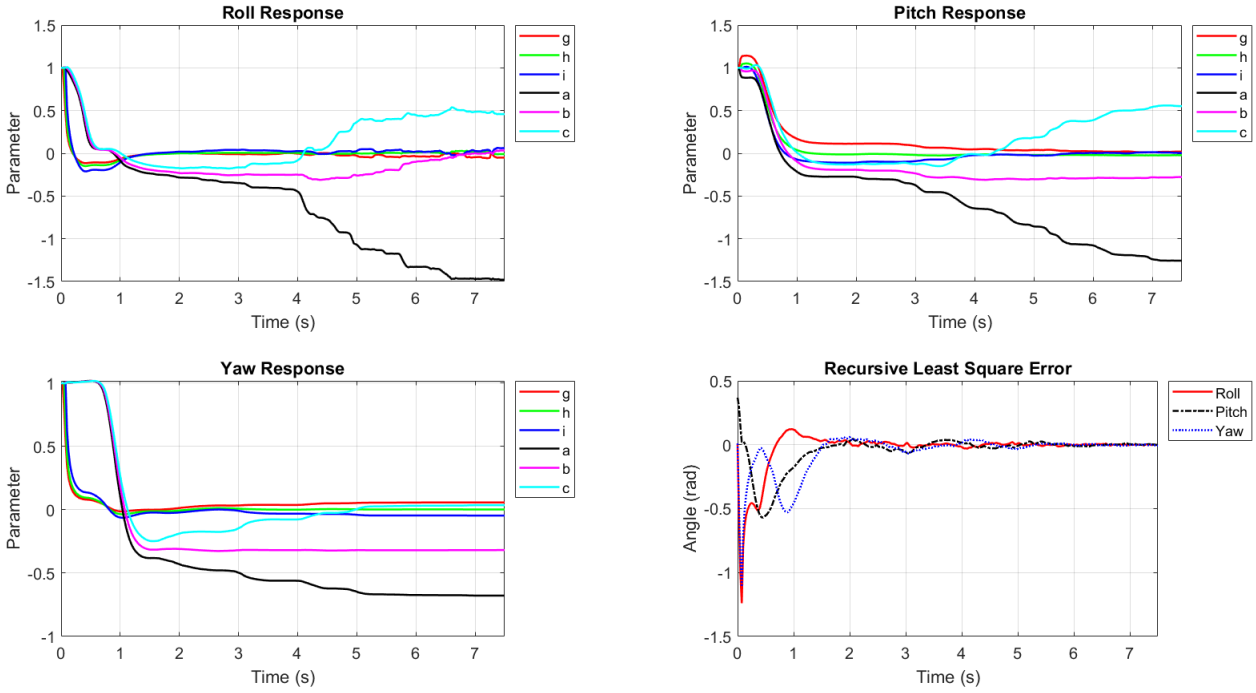


Figure 10: Recursive Least Square Results

Table 1: Estimated Parameters of Servo Motors

Motor	g	h	i	a	b	c
Roll	-0.0506	-0.0091	0.0602	-1.4785	0.0325	0.4587
Pitch	0.0205	-0.0218	0.0047	-1.2567	-0.2779	0.5522
Yaw	0.0584	0.0029	-0.0454	-0.6777	-0.3179	0.0369

Figure 11 compares the system response of the estimated model from RLS and the actual motor. The results shows that estimated model reasonably captures the dynamics of the motor. Models obtained for pitch and yaw motors display similar results.

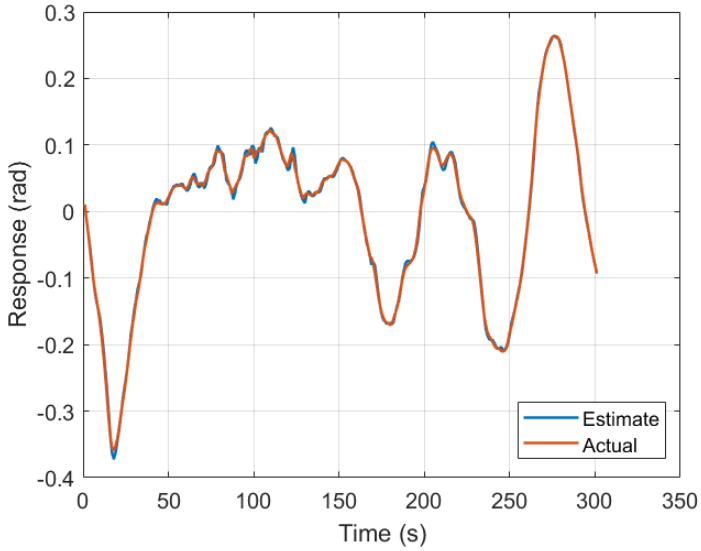


Figure 11: Estimate vs. Actual Response of Roll Motor

8 Conclusion and Future Work

The 3 axis gimbal satisfies the objectives outlined in the introduction. The primary goal of the gimbal is to attenuate disturbances introduced by the user by maintaining the top platform at the same reference plane. The system model was developed using RLS with the assumption that the relationship between the motor angle and input voltage is third order system. An additional assumption was made that the motor rotation is equal and opposite of the handle rotation when slowly moving the handle. A controller was then created to stabilize the top platform for low frequency disturbances while completing a trajectory. Experimentation shows that the control system for general usage is valid only at slow movements. The construction of the gimbal and the placement of the IMU sensor readings verifies that system executes as intended.

The next steps for the gimbal is to stabilize the platform for coupled motions. This can be achieved by replacing the MSP432 to a Raspberry Pi microcontroller, which has the computational power necessary to run inverse kinematics. Disturbance rejection can be improved by adding an IMU sensor on the top of the platform to obtain the true motor position. This secondary IMU sensor allows the implementation of a feedback controller that could deter large disturbances. The gains of this controller can be determined by implementing a linear quadratic regulator (LQR) since estimated model was obtained. A long term goal would be to place the 3 axis gimbal on a moving platform to mirror camera stabilizers used in film industry.

References

[1] Wilhelm Andrén and Ella Hjertberg. "Gyro Stabilization of a Positioning Unit". In: (2019).

[2] CHRobotics. "Understanding Euler Angles". In: (2012).

[3] Mark Pedley. "Tilt sensing using a three-axis accelerometer". In: *Freescale semiconductor application note* 1 (2013), pp. 2012–2013.

[4] Takashi Wada et al. "Practical modeling and system identification of R/C servo motors". In: *2009 IEEE Control Applications,(CCA) & Intelligent Control,(ISIC)*. IEEE. 2009, pp. 1378–1383.

[5] Ruoyu Zhi. "A drift eliminated attitude & position estimation algorithm in 3d". In: (2016).

Appendix

A. Most Important Feedback received this term

1. Condense experimental data to the least amount of figures as possible/show only relevant figures
2. Analyze each figure after introducing them in discussion
3. Display figures to clearly present information (eg. Increase font size and marker size)

B. CCS Code for General Usage and Trajectory Control

```
/* DriverLib Defines */
#include "driverlib.h"
#include <math.h>

/* Standard Defines */
#include <string.h>
#include <stdio.h>

/* Slave Address for I2C Slave */
#define SLAVE_ADDRESS      0x68
#define pi                 3.14159
#define Gyro_Sensitivity   131.0
#define Acc_Sensitivity    16384.0
#define Readings           5000

/* IMU sensor registers */
uint8_t accXReg[] = {0x3B, 0x3C};
uint8_t accYReg[] = {0x3D, 0x3E};
uint8_t accZReg[] = {0x3F, 0x40};
uint8_t gyrXReg[] = {0x43, 0x44};
uint8_t gyrYReg[] = {0x45, 0x46};
uint8_t gyrZReg[] = {0x47, 0x48};
uint8_t WhoAmI;

/* IMU Sensor Parameters */
volatile float accRoll, accPitch;          // Roll calculated from accelerometer
volatile float gyroRoll = 0, gyroPitch = 0;  // Roll calculated from gyroscope
volatile float roll=0, pitch=0, yaw = 0;
float desired_angleX, desired_angleY;

/* Calibration Parameters */
volatile float AccErrorX = 0.0, AccErrorY = 0.0;                  // Accelerometer reading offset
volatile float GyroErrorX = 0.0, GyroErrorY = 0.0, GyroErrorZ = 0.0; // Gyroscope reading
offset

/* Time and PWM signal Parameter */
volatile int time = 0;
```

```
volatile int currentTime = 0, previousTime;
volatile float dT = 0;   // in sec

/* Variables to save */
volatile int counter = 0;
//volatile float data_Vroll[Readings]   = {};
//volatile float data_Vpitch[Readings]  = {};
//volatile float data_Vyaw[Readings]    = {};
volatile float data_roll[Readings]   = {};
volatile float data_pitch[Readings]  = {};
volatile float data_yaw[Readings]    = {};

/* Declare functions */
void initialize_TIMER_A0(void);
void initialize_PWM(void);
void initializeI2C(void);
void TA0_0_IRQHandler(void);
int configRegRead(uint8_t *Register);
int configSingleByteRegRead(uint8_t Register);
void configRegWrite(uint8_t Register, uint8_t Data);
void calculate_IMU_error(void);
void angleCalculations(void);
int servoMotorRoll(float angle);
int servoMotorPitch(float angle);
int servoMotorYaw(float angle);

/* Timer_A UpMode Configuration Parameter */
const Timer_A_UpModeConfig upConfig =       // Configure counter in Up mode
{
 TIMER_A_CLOCKSOURCE_SMCLK,               // Tie Timer A to SMCLK
 TIMER_A_CLOCKSOURCE_DIVIDER_6,           // Increment every 1ms
 500,                                     // Period of Timer A (this value placed in TAxCCR0)
 TIMER_A_TAIE_INTERRUPT_DISABLE,          // Disable Timer A rollover interrupt
 TIMER_A_CCIE_CCR0_INTERRUPT_ENABLE,      // Enable Capture Compare interrupt
 TIMER_A_DO_CLEAR                         // Clear counter upon initialization
};

/* Timer_A PWM Configuration Parameter */
const Timer_A_PWMConfig pwmConfig1 =
{
 TIMER_A_CLOCKSOURCE_SMCLK,         // Tie Timer A to SMCLK
 TIMER_A_CLOCKSOURCE_DIVIDER_20,    // Increment counter every 20 clock cycles
 3000,                              // Timer Period (TAxCCR0)
 TIMER_A_CAPTURECOMPARE_REGISTER_1,  // Compare Register (y)
 TIMER_A_OUTPUTMODE_RESET_SET,      // Compare Output Mode (OUTMOD__7)
 225                                // Duty Cycle (TAxCCRy)
```

```
};
const Timer_A_PWMConfig pwmConfig2 =
{
 TIMER_A_CLOCKSOURCE_SMCLK,         // Tie Timer A to SMCLK
 TIMER_A_CLOCKSOURCE_DIVIDER_20,    // Increment counter every 20 clock cycles
 3000,                              // Timer Period (TAxCCR0)
 TIMER_A_CAPTURECOMPARE_REGISTER_4,  // Compare Register (y)
 TIMER_A_OUTPUTMODE_RESET_SET,      // Compare Output Mode (OUTMOD__7)
 215                                // Duty Cycle (TAxCCRy)
};
const Timer_A_PWMConfig pwmConfig3 =
{
 TIMER_A_CLOCKSOURCE_SMCLK,         // Tie Timer A to SMCLK
 TIMER_A_CLOCKSOURCE_DIVIDER_20,    // Increment counter every 20 clock cycles
 3000,                              // Timer Period (TAxCCR0)
 TIMER_A_CAPTURECOMPARE_REGISTER_3,  // Compare Register (y)
 TIMER_A_OUTPUTMODE_RESET_SET,      // Compare Output Mode (OUTMOD__7)
 238                                // Duty Cycle (TAxCCRy)
};

/* I2C Master Configuration Parameter */
const eUSCI_I2C_MasterConfig i2cConfig =
{
 EUSCI_B_I2C_CLOCKSOURCE_SMCLK,        // SMCLK Clock Source
 3000000,                              // SMCLK = 3MHz
 EUSCI_B_I2C_SET_DATA_RATE_400KBPS,    // Desired I2C Clock of 400kHz
 0,                                    // No byte counter threshold
 EUSCI_B_I2C_NO_AUTO_STOP              // No Autostop
};

void main(void)
{
    /* Halting the Watchdog */
    MAP_WDT_A_holdTimer();

    /* Configure clock */
    uint32_t dcoFrequency = 3E+6;          // 3MHz
    MAP_CS_setDCOFrequency(dcoFrequency);   // Set clock source to 3MHz
    MAP_CS_initClockSignal(CS_SMCLK, CS_DCOCLK_SELECT,
CS_CLOCK_DIVIDER_1);

    /** Configure GPIO pins **/
    /* Configuring P1.0 as output */
    MAP_GPIO_setAsOutputPin(GPIO_PORT_P1,GPIO_PIN0);
    MAP_GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN0);
```

```
    /* Configuring P2.0 as output */
    MAP_GPIO_setAsOutputPin(GPIO_PORT_P2, GPIO_PIN0 || GPIO_PIN1 || GPIO_PIN2);
    MAP_GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN0 || GPIO_PIN1 ||
GPIO_PIN2);

    /* Initializing Timer_A2 for PWM */
    initialize_PWM();

    /* Initialize I2C communication protocol */
    initializeI2C();

    /* Verify sensor communication */
    WhoAmI = configSingleByteRegRead(0x75); // Used to verify the identity of sensor, this
should read 0x71
    if (WhoAmI==0x71){
        printf("IMU sensor communication established. \r\n");
        MAP_GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN2);
    }
    else {
        MAP_GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN2);
    }

    /* Initializing Timer_A0 for interrupts */
    initialize_TIMER_A0();

    /* Starting calibration */
    calculate_IMU_error();
    MAP_GPIO_setOutputHighOnPin(GPIO_PORT_P1, GPIO_PIN0);

    double accX, accY, accZ;
    accX = configRegRead(accXReg) / Acc_Sensitivity;
    accY = configRegRead(accYReg) / Acc_Sensitivity;
    accZ = configRegRead(accZReg) / Acc_Sensitivity;

    /* Obtaining angle from accelerometer readings */
    desired_angleX = (atan2(accY,  sqrt(pow(accX,2) + pow(accZ,2))) * 180/pi) - AccErrorX;  //
in degrees
    desired_angleY = (atan2(-accX, sqrt(pow(accY,2) + pow(accZ,2))) * 180/pi) - AccErrorY;  //
in degrees

    gyroRoll = desired_angleX;
    gyroPitch = desired_angleY;
    MAP_GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN0);

    /* Enabling MASTER interrupts */
    MAP_Interrupt_enableMaster();
```

```
    while(1){
        float errorX, errorY, errorZ;

        /* Read sensor registers and calculate angles */
        angleCalculations();

        /* Calculate the error between current and desired angle */
        errorX = roll - desired_angleX;
        errorY = pitch - desired_angleY;
        errorZ = yaw;

        /* Turn servo motor */
        TA2CCR1 = servoMotorRoll(errorY) - 5;
        TA2CCR4 = servoMotorPitch(errorX) - 15;
        TA2CCR3 = servoMotorYaw(errorZ) + 8;
    }
}

void TA0_0_IRQHandler(void){
    /*  Clear the interrupt flag */
    MAP_Timer_A_clearCaptureCompareInterrupt(TIMER_A0_BASE,
TIMER_A_CAPTURECOMPARE_REGISTER_0);
    time++;
}

void initializeI2C(void){
    /* Select I2C function for I2C_SCL & I2C_SDA */
    GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_P6, GPIO_PIN5,
GPIO_PRIMARY_MODULE_FUNCTION); // I2C_SCL
    GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_P6, GPIO_PIN4,
GPIO_PRIMARY_MODULE_FUNCTION); // I2C_SDA

    /* Initializing I2C Master to SMCLK at 400kbs with no autostop */
    MAP_I2C_initMaster(EUSCI_B1_BASE, &i2cConfig);

    /* Enable I2C Module to start operations */
    MAP_I2C_enableModule(EUSCI_B1_BASE);

    /* Specify slave address */
    MAP_I2C_setSlaveAddress(EUSCI_B1_BASE, SLAVE_ADDRESS);

    /* Make reset, place a 0 into the 0x6B register */
    configRegWrite(0x6B, 0x00);
    configRegWrite(0x1D, 0x06);     // Acc Config: Programming digitally low pass filter
}
```

```
void initialize_TIMER_A0(void){
   // Configuring Timer_A0 for Up Mode using above strut
   MAP_Timer_A_configureUpMode(TIMER_A0_BASE, &upConfig);

   // Enabling interrupts and starting the timer
   MAP_Interrupt_enableInterrupt(INT_TA0_0);

   // Start counting timer
   MAP_Timer_A_startCounter(TIMER_A0_BASE, TIMER_A_UP_MODE);
}

void initialize_PWM(void){
   /* Set P5.6(T2.1), P5.7(2.2), and P6.6(T2.3) to be PWM functionality */
   MAP_GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_P5, GPIO_PIN6,
GPIO_PRIMARY_MODULE_FUNCTION);
   MAP_GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_P6, GPIO_PIN7,
GPIO_PRIMARY_MODULE_FUNCTION);
   MAP_GPIO_setAsPeripheralModuleFunctionOutputPin(GPIO_PORT_P6, GPIO_PIN6,
GPIO_PRIMARY_MODULE_FUNCTION);

   /* Configuring Timer_A to have a period of approximately 500ms and an initial duty cycle of
10% of that (3200 ticks) */
   MAP_Timer_A_generatePWM(TIMER_A2_BASE, &pwmConfig1);
   MAP_Timer_A_generatePWM(TIMER_A2_BASE, &pwmConfig2);
   MAP_Timer_A_generatePWM(TIMER_A2_BASE, &pwmConfig3);
}

int configRegRead(uint8_t *Register){
   int i = 0;
   uint8_t RXData[2];
   int16_t Data;

   for (i=0;i<2;i++){
      /* Set write mode. */
      MAP_I2C_setMode(EUSCI_B1_BASE, EUSCI_B_I2C_TRANSMIT_MODE);

      // Write START + address + register to the sensor.
      MAP_I2C_masterSendSingleByte(EUSCI_B1_BASE, Register[i]);

      /* Making sure that the last transaction has been completely sent */
      while (MAP_I2C_masterIsStopSent(EUSCI_B1_BASE)){}   // Indicates whether STOP got
sent.

      // Set read mode.
      MAP_I2C_setMode(EUSCI_B1_BASE, EUSCI_B_I2C_RECEIVE_MODE);
```

```
        // RESTART.
        MAP_I2C_masterReceiveStart(EUSCI_B1_BASE);

        // Read two bytes from the sensor, STOP.
        RXData[i] = MAP_I2C_masterReceiveMultiByteFinish(EUSCI_B1_BASE);
    }

    /* Parse the sensor's data. */
    Data = (int16_t)(RXData[0] << 8 | RXData[1]);
    return Data;
}

int configSingleByteRegRead(uint8_t Register){
    uint8_t RxData;

    /* Set write mode. */
    MAP_I2C_setMode(EUSCI_B1_BASE, EUSCI_B_I2C_TRANSMIT_MODE);

    // Write START + address + register to the sensor.
    MAP_I2C_masterSendSingleByte(EUSCI_B1_BASE, Register);

    /* Making sure that the last transaction has been completely sent */
    while (MAP_I2C_masterIsStopSent(EUSCI_B1_BASE)){}   // Indicates whether STOP got
sent.

    // Set read mode.
    MAP_I2C_setMode(EUSCI_B1_BASE, EUSCI_B_I2C_RECEIVE_MODE);

    // RESTART.
    MAP_I2C_masterReceiveStart(EUSCI_B1_BASE);

    // Read two bytes from the sensor, STOP.
    RxData = MAP_I2C_masterReceiveMultiByteFinish(EUSCI_B1_BASE);

    return RxData;
}

void configRegWrite(uint8_t Register, uint8_t Data){
    int i = 0 ;

    // Set write mode.
    MAP_I2C_setMode(EUSCI_B1_BASE, EUSCI_B_I2C_TRANSMIT_MODE);

    /* Send start bit and register */
    MAP_I2C_masterSendMultiByteStart(EUSCI_B1_BASE, Register);
```

```
    // Delay
    for(i = 0; i < 2000; i++){}

    /* Now write the data byte */
    MAP_I2C_masterSendMultiByteFinish(EUSCI_B1_BASE, Data);

    while (MAP_I2C_masterIsStopSent(EUSCI_B1_BASE)){}   // Indicates whether STOP got
sent.
}

void calculate_IMU_error(void){
    /* Note: Place the IMU flat in order to get the proper values.
     * This is the calibration process                          */

    int c = 0;
    float accX, accY, accZ;          // Store accelerometer register raw sensor reading
    float GyroX, GyroY, GyroZ;       // Store gyroscope register raw sensor reading

    // Read accelerometer and gyroscope values 200 times
    while (c < 200){
        GyroX = configRegRead(gyrXReg) / Gyro_Sensitivity;
        GyroY = configRegRead(gyrYReg) / Gyro_Sensitivity;
        GyroZ = configRegRead(gyrZReg) / Gyro_Sensitivity;

        accX = configRegRead(accXReg) / Acc_Sensitivity;
        accY = configRegRead(accYReg) / Acc_Sensitivity;
        accZ = configRegRead(accZReg) / Acc_Sensitivity;

        AccErrorX = AccErrorX + (atan2(accY, sqrt(pow(accX,2) + pow(accZ,2))) * 180/pi);
        AccErrorY = AccErrorY + (atan2(-accX, sqrt(pow(accY,2) + pow(accZ,2))) * 180/pi);

        GyroErrorX = GyroErrorX + GyroX;
        GyroErrorY = GyroErrorY + GyroY;
        GyroErrorZ = GyroErrorZ + GyroZ;
        c++;
    }

    /* Average 200 values to get the error value */
    AccErrorX = AccErrorX / 200.0;
    AccErrorY = AccErrorY / 200.0;
    GyroErrorX = GyroErrorX / 200.0;
    GyroErrorY = GyroErrorY / 200.0;
    GyroErrorZ = GyroErrorZ / 200.0;
}
```

```
void angleCalculations(void){
  double accX, accY, accZ;          // Store accelerometer register raw sensor reading
  float GyroX, GyroY, GyroZ;        // Store gyroscope register raw sensor reading

  /* Euler Angle transformation parameters */
  float GyroRateRoll, GyroRatePitch, GyroRateYaw;

  /* Accelerometer and gyroscope register readings */
  /*************************************************************
          Gyroscope Register Readings
   *************************************************************/
  GyroX = configRegRead(gyrXReg) / Gyro_Sensitivity - GyroErrorX;
  GyroY = configRegRead(gyrYReg) / Gyro_Sensitivity - GyroErrorY;
  GyroZ = configRegRead(gyrZReg) / Gyro_Sensitivity - GyroErrorZ;

  /*************************************************************
          Accelerometer Register Readings
   *************************************************************/
  accX = configRegRead(accXReg) / Acc_Sensitivity;
  accY = configRegRead(accYReg) / Acc_Sensitivity;
  accZ = configRegRead(accZReg) / Acc_Sensitivity;

  /* Obtaining angle from accelerometer readings */
  accRoll  = (atan2(accY, sqrt(pow(accX,2) + pow(accZ,2))) * 180/pi) - AccErrorX;  // in
degrees
  accPitch = (atan2(-accX, sqrt(pow(accY,2) + pow(accZ,2))) * 180/pi) - AccErrorY;  // in
degrees

  /* Converting body-frame angular rates to Euler angular rates */
  GyroRateRoll  = GyroX + sin(roll*pi/180.0)*tan(pitch*pi/180.0)*GyroY +
cos(roll*pi/180.0)*tan(pitch*pi/180.0)*GyroZ;
  GyroRatePitch = cos(roll*pi/180.0)*GyroY - sin(roll*pi/180.0)*GyroZ;
  GyroRateYaw   = sin(roll*pi/180.0)/cos(pitch*pi/180)*GyroY +
cos(roll*pi/180.0)/cos(pitch*pi/180.0)*GyroZ;

  /* Read time */
  previousTime = currentTime;             // Previous time is stored before the actual time read
  currentTime = time;                     // Current time actual time read
  dT = (currentTime - previousTime)/1000.0;   // in sec

  data_roll[counter] = roll;
  data_pitch[counter] = pitch;
  data_yaw[counter] = yaw;

  /* Integrating the angular velocity to get angles */
  gyroRoll = gyroRoll + GyroRateRoll * dT;
```

```
    gyroPitch = gyroPitch + GyroRatePitch * dT;
    yaw =  yaw + GyroRateYaw * dT;

    /* Complementary filter - combine accelerometer and gyro angle values */
    roll  = 0.94 * gyroRoll + 0.06 * accRoll;
    pitch = 0.94 * gyroPitch + 0.06 * accPitch;

    counter++;
}

int servoMotorRoll(float angle){
    int duty_cycle;
    duty_cycle = 230 - (5*angle)/3;
    if (duty_cycle >= 390){
        duty_cycle = 390;
    }
    else if (duty_cycle <= 60){
        duty_cycle = 60;
    }
    return duty_cycle;
}
int servoMotorPitch(float angle){
    int duty_cycle;
    duty_cycle = 230 + (5*angle)/3;
    if (duty_cycle >= 4000){
        duty_cycle = 400;
    }
    else if (duty_cycle <= 55){
        duty_cycle = 55;
    }
    return duty_cycle;
}
int servoMotorYaw(float angle){
    int duty_cycle;
    duty_cycle = 230 - (5*angle)/3;
    if (duty_cycle >= 400){
        duty_cycle = 400;
    }
    else if (duty_cycle <= 55){
        duty_cycle = 55;
    }
    return duty_cycle;
}
```

C. Matlab Code for RLS Implementation

```
clear; clc; close all;
```

```
angleArr{1} = xlsread('data_roll.csv');
angleArr{2} = xlsread('data_pitch.csv');
angleArr{3} = xlsread('data_yaw.csv');

L = [.97 0.98 0.99];
range = 300:600;
estimateParameters = cell(1,3);
estimateY = estimateParameters;
actualAngle = estimateY;
E = estimateY;

for i = 1:3
    num = angleArr{i};
    lambda = L(i);
    angleMotor = -num(range,1)*pi/180;
    DutyCycle = num(range,2);
    Voltage = (DutyCycle/3000)*6.4;
    VoltageOld = zeros(1,3); %1
    AngleOld = zeros(1,3); %3
    m = length(VoltageOld) + length(AngleOld);
    n = length(range);
    P_1 = eye(m);
    theta = ones(m,1);
    T = zeros(m,n);
    error = zeros(1,n);
    actual = error;
    estimate = error;
    theta_1 = theta;

    for k = 1:n
        y = angleMotor(k);
        Phi = [VoltageOld -AngleOld]';
        P= (P_1 - (P_1*Phi * Phi' * P_1) / (1 + Phi' * P_1 * Phi)) / lambda;
        T(:,k) = theta;
        error(k) = (y-(Phi'*theta_1));
        estimate(k) = Phi'*theta_1;
        theta = theta_1 + P*Phi*error(k);
        VoltageOld = [Voltage(k) VoltageOld(1:end-1)];
        AngleOld = [angleMotor(k) AngleOld(1:end-1)];
        P_1 = P;
        theta_1 = theta;
        actual(k) = y;
    end

    estimateParameters{i} = T;
```

```
    E{i} = error;

    y_est = angleMotor(1:length(AngleOld))';
    for k = (length(y_est) + 1):length(range)
        y_est = [y_est [Voltage(k-1) Voltage(k-2) Voltage(k-3) -actual(k-1) -actual(k-2) -actual(k-
3)]*theta];
    end

    estimateY{i} = y_est;

    figure

    subplot(2,2,1)
    for j = 1:length(theta)
        plot(1:length(T), T(j,:), '-', 'LineWidth', 1.5)
        hold on
    end
    grid on
    %legend('g', 'h', 'i', 'a', 'b', 'c')
    %title('Parameters')
    set(gca, 'FontSize', 12)
    xlabel('Number of Iterations')
    ylabel('Parameters')
    subplot(2,2,2)
    plot(1:length(error), error, 'LineWidth', 2)
    %title('Error')
    xlabel('Number of Iterations')
    ylabel('Model Error')
    set(gca, 'FontSize', 12)
    grid on
    figure
    % estimate = tf([theta(3) theta(2) theta(1)], [1 theta(4) theta(5) theta(6)], 0.025);
    % [y, t] = impulse(estimate);
    plot(1:length(y_est), y_est, '-', 'LineWidth', 1.5);
    hold on
    plot(1:length(actual), actual, '-', 'LineWidth', 1.5)
    legend('Estimate', 'Actual')
    set(gca, 'FontSize', 12)
    xlabel('Time (s)')
    ylabel('Response (rad)')
    grid on
    subplot(2,2,4)
    n = floor(length(theta)/2);
    plot(1:length(theta), [theta(n:end)' theta(1:n-1)'], '*', 'MarkerSize', 10)
    legend('Estimate')
    set(gca, 'FontSize', 12)
```

```
    %title('Actual vs. Estimate Parameters')
    xlabel('a b c g h i')
    ylabel('Parameter Value')
    grid on
%
end

figure
name{1} = 'Roll Response';
name{2} = 'Pitch Response';
name{3} = 'Yaw Response';
marker = {'r-', 'g-', 'b-', 'k-', 'm-', 'c-'};
for i = 1:3
    subplot(2,2,i)
    theta = estimateParameters{i};
    [m,n] = size(theta);
    for k = 1:m
        time = 0:0.025:0.025*(length(theta)-1);
        plot(time,theta(k,:), marker{k}, 'LineWidth', 1.5)
        hold on
    end
    xlim([0 time(end)])
    grid on
    xlabel('Time (s)')
    ylabel('Parameter')
    title(name{i})
    set(gca, 'FontSize', 12)
    legend('g', 'h', 'i', 'a', 'b', 'c', 'Location', 'northeastoutside')
end

subplot(2,2,4)
marker = {'r-', 'k-.', 'b:'};
for i = 1:3
    error = E{i};
    time = 0:0.025:0.025*(length(error)-1);
    plot(time, error, marker{i}, 'LineWidth', 1.5)
    hold on
end

xlim([0 time(end)])
grid on
xlabel('Time (s)')
ylabel('Angle (rad)')
title('Recursive Least Square Error')
set(gca, 'FontSize', 12)
legend('Roll', 'Pitch', 'Yaw', 'Location', 'northeastoutside')
```