ECE 167 Sensing and Sensor Technologies

Lab #4: Attitude Estimation

March 04, 2024

Commit ID: 8c7dee907e4ea971858e258d2eab29aff62168de

Niki Mobtaker, Jacob Gutierrez, Aleida Diaz-Roque

Group: Dits_and_Dahs

# I. Overview

# II. Wiring and Diagrams

# III. Project Design, Testing and Results

# IV. Conclusion

# I. Overview

In this lab on Attitude Estimation, we implemented attitude tracking through the use of a direction cosine matrix (DCM). Our exploration began with sensor calibration and a deep understanding of coordinate frames. We then moved on to the implementation of open-loop gyroscope integration, converting DCM to Euler angles, and working with both simulated and real data. The lab introduced closed-loop estimation techniques, incorporating valuable feedback from accelerometers and magnetometers to enhance accuracy. Additionally, we addressed the misalignment of accelerometer and magnetometer sensors, culminating in the completion of a full complementary attitude estimation by integrating magnetometer feedback. Our focus remained on the practical application of sensor fusion techniques.

# II. Wiring and Diagrams

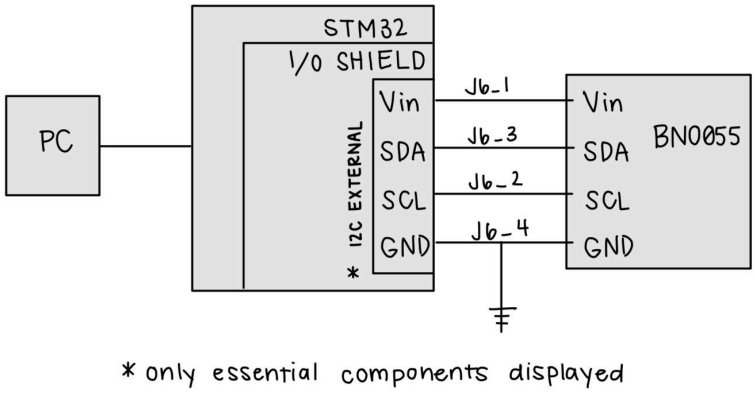



Figure 1. BNO055 Wiring to System

# III. Project Design, Testing and Results

## Part 2: Conversion from DCM to Euler Angles

### Project Design

The main purpose of this portion was to create a function which would take in a DCM in the '3, 2, 1' convention as seen in Fig. 2 and output the Euler angles.

$$\mathcal{R}_{[\phi,\theta,\psi]} = \begin{bmatrix} \cos\psi\cos\theta & \sin\psi\cos\theta & -\sin\theta \\ \cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi & \sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi & \cos\theta\sin\phi \\ \cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi & \sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi & \cos\theta\cos\phi \end{bmatrix}$$

Figure 2. Full rotation matrix from inertial to body frames

To avoid numerical instability, the function should detect the gimbal lock singularity and set the pitch angle accordingly to ±90 degrees and set the roll angle to 0 and choose an arbitrary value for the roll angle and calculate the yaw angle using the arctangent function. If there is no gimbal lock, it calculates the angles using the standard formulas defined in Fig 3. using the DCM elements.

$$\begin{aligned} \theta &= -\arcsin\mathcal{R}_{[1,3]} \\ \phi &= \operatorname{arctan2}(\mathcal{R}_{[2,3]}, \mathcal{R}_{[3,3]}) \\ \psi &= \operatorname{arctan2}(\mathcal{R}_{[1,2]}, \mathcal{R}_{[1,1]}) \end{aligned}$$

Figure 3. Euler angles extracted from DCM matrix

DCM to Euler angles ('dcm_to_euler' function): The function will take a pointer to a 3x3 rotation matrix (in floats) as input and return the Euler angles in degrees. The function will utilize inverse trigonometric function and handle cases of gimbal lock, where the pitch angle approaches ±90 degrees.

## Testing and Results

Test cases will cover two scenarios, normal cases and edge cases such as gimbal lock situations. The results will be compared against expected values based on known DCM inputs. In the first case the DCM matrix was computed using online software providing the euler angles as $(\phi, \theta, \psi) = (30, 15, 20)$. This yielded the DCM matrix in Fig. 4.

$$C = \begin{pmatrix} 0.908 & 0.418 & -0.0396 \\ -0.330 & 0.770 & 0.547 \\ 0.259 & -0.483 & 0.837 \end{pmatrix}$$

Figure 4. Generated euler

Once the matrix was passed into dcm_to_euler, it calculated the euler angles which matched our expected results.

```
DCM Matrix:                    Euler Angles:
0.908000 0.418000 -0.039600    Roll: -29.99 degrees
-0.330000 0.770000 0.547000    Pitch: -15.01 degrees
0.259000 -0.483000 0.837000    Yaw: -19.97 degrees
```

Figure 5. Euler angles derived from dcm_to_euler

# Part 3: Integration of Constant Body-Fixed Rates

## Project Design

The program written in this part of the lab is designed to integrate the rotation matrix of a gyroscope using the following two methods. Forward Integration and Matrix Exponential are used in the field of robotics and aerospace engineering for attitude propagation, which is the process of updating the orientation of an object (like a spacecraft or a robot) over time based on its angular velocity.

Forward Integration is a numerical method used to propagate the attitude of an object. It is based on the principle of Euler's method for solving ordinary differential equations. The idea is to use the current attitude and angular velocity to estimate the attitude at the next time step. In other words, we need to track the evolution of the rotation matrix that stores the DCM form of the attitude. The first equation that aides us in this process is the skew-symmetric matrix:

$$[\omega\times] = \begin{bmatrix} 0 & -r & q \\ r & 0 & -p \\ -q & p & 0 \end{bmatrix}$$

Figure 6. Skew Symmetric Matrix

The matrix [ω×] represents the cross product in its matrix form. The elements of the vector ω are directly obtained from the gyroscopes fixed to the body, and are denoted by the symbols [p, q, r]. Each of these symbols corresponds to the rate of rotation about their respective body axis unit vectors. For instance, p denotes the rate of rotation about the body's x-axis, q represents the y-axis, and r signifies the z-axis.

The other equation we need is the DCM differential equation, composed of 3 direct and 6 constraint differential equations. If we assume a constant rotation rate across the time step ΔT we arrive at the formulation of how the DCM evolves over time and integrate it numerically. So, given an initial DCM ($R_o$), we measure the body fixed rates, put them into a skew-symmetric matrix and use forward integration to update the DCM:

$$\mathcal{R}_{k+1} = \mathcal{R}_k - [\vec{\omega}\times]\mathcal{R}_k\Delta t$$

Figure 7. Forward Integration Method

The Matrix Exponential method, on the other hand, is a more accurate approach to attitude propagation. It is based on the Rodrigues' rotation formula, which provides an exact

solution for rotating a vector by a certain angle around an axis. In this method, the exponential of the skew-symmetric matrix of angular velocity is used to compute the rotation matrix that describes the rotation over a time step. This is a better approach because the results of the Forward Integration method are not rotation matrices since you do not do successive rotations by addition/subtraction but rather by multiplication. Below is the equation that describes the Matrix Exponential form method:

$$R_{k+1} = e^{-[\omega\times]\Delta t} R_k$$

Figure 8. Matrix Exponential Form

Since the matrix exponential form $R_{exp}(w\Delta t)$ is a rotation matrix itself and represents incremental rotation across Δt, we just need to implement a matrix multiplication function in our program to propagate the DCM across the time step.

So to integrate both of these methods we analyzed the matlab files provided and determined that the we need to write and implement the following functions:

1. Skew-Symmetric Matrix (‘skew_symmetric’ function): This function creates a skew-symmetric matrix from a 3D vector. The skew-symmetric matrix is used to represent the cross product operation in matrix form as shown in the figure above. The matrix is defined as follows for a vector$v = [v_x,\ v_y,\ v_z]$:

2. Matrix Multiplication (‘matrix_multiply’ function): This function performs standard matrix multiplication between two 3x3 matrices. It is used in various parts of the program to compute the product of matrices.

3. Open-Loop Integration (‘IntegrateOpenLoop’ function): This function integrates the rotation matrix using the open-loop method, which is based on Euler integration. The process involves the following steps:

- Compute the skew-symmetric matrix of the gyroscope input (angular rate vector) scaled by the time step (Δt).
- Multiply this matrix by the current rotation matrix to obtain the change in the rotation matrix (ΔR).
- Update the rotation matrix by adding the change (R = R + ΔR).

The integration is performed over a fixed number of steps (‘NUM_STEPS’), simulating the gyroscope's behavior over time.

4. Exponential Rodrigues Parameter Form (‘Rexp’ function): This function computes the exponential Rodrigues parameter form of the rotation matrix, which is used to update the rotation matrix in each step. The Rodrigues formula is given by:

$$R\_exp = I + sin(\omega\Delta t) * skew\_symmetric(w) + (1 - cos(\omega\Delta t)) * skew\_symmetric(w)^2$$

For small angles, a Taylor series expansion is used to approximate the sine and cosine functions to avoid numerical instability.

$$sinc(W) = \Delta t - (\Delta t\text{^}3 * \omega\text{^}2) / 6 + (\Delta t\text{^}5 * \omega\text{^}4) / 120$$

$$1 - Cos(W) = (\Delta t^2) / 2 - (\Delta t^4 * \omega^2) / 24 + (\Delta t^6 * \omega^4) / 720$$

5. Attitude Propagation (‘attitudePropagation’ function): This function performs attitude propagation using forward integration. It takes in the roll, pitch, and yaw rates (r, p, q), the current rotation matrix (Rk), and the time step (dT). It then calculates the skew-symmetric matrix

from the angular rates, multiplies it by the current rotation matrix to get the derivative of the rotation matrix (Rdot), and finally uses forward integration to update the rotation matrix to the next time step (Rk1).

6. Forward Integration (‘forwardIntegration’ function): This function updates the rotation matrix using Euler's method for integration. It takes in the current rotation matrix (Rk), the derivative of the rotation matrix (Rdot), the time step (dT), and outputs the updated rotation matrix (Rk1). It loops through each element of the matrix and updates it by adding the product of the derivative and the time step to the current value.

5. Main Function( Or ‘*Test Case’* rather): The main function sets up the initial conditions, including the time step, angular rate input, and initial rotation matrix. If the macro PART3_1 is defined, it then iterates over a fixed number of steps, updating the rotation matrix using the ‘*Rexp()*’ function and printing the result at each step. If instead the macro PART3_2 is defined then it begins the same process, but just calls the attitudePropagation() function which does the forward integration and sends out the new rotation matrix.

Once we got all these functions written and tested, we had to compare how fast the DCM drifts out of orthonormality for each method. To do this, we simply used the same initial conditions that were used in the ‘*OpenLoopIntegration_NoBias.m’*. This gave us two test cases in our main function, which we could parse through incrementally by implementing macro definitions ‘*#ifdef, #endif’*. Below is a set of pseudo-code to describe the way we did our attitude propagation:

```
// Define constants (Same used in matlab programs)

Initialize the rotation matrix as Identity Matrix

// PART 3.1: Attitude propagation using matrix exponential
#ifdef PART3_1
    For i from 0 to 40:
        Print "Step i:"

        // Calculate the rotation matrix using matrix exponential
        Define R_exp[3][3]
        Call Rexp(gyroInput, deltaTime, R_exp)

        // Update the rotation matrix
        Define R_new[3][3]
        Call matrix_multiply(R_exp, R, R_new)
        Copy R_new to R

        Print the updated rotation matrix

    End For
#endif

// PART 3.2: Attitude propagation using Forward Integration
#ifdef PART3_2
    For i from 0 to 40:
        Print "Step i:"

        // Update the rotation matrix using Forward Integration and
attitude_propagation
        Define R_new[3][3]
        Call attitudePropagation(r, p, q, R, deltaTime, R_new)
        Copy R_new to R

        Print the updated rotation matrix

#endif
```

## Testing and Results

To verify the validity of our results, we compared the output matrices generated from the test case using the Rexp() function in C to the results obtained using the same function in MatLab. For matrix exponential this can be seen in Fig. 9 and Fig. 10.

```
Step 0:
1.000000 0.000000 0.000000
0.000000 0.984808 0.173648
0.000000 -0.173648 0.984808
```

```
Step 40:
1.000000 0.000000 0.000000
0.000000 0.642787 0.766044
0.000000 -0.766044 0.642787
```

Figure 9. First and final matrices using matrix exponential

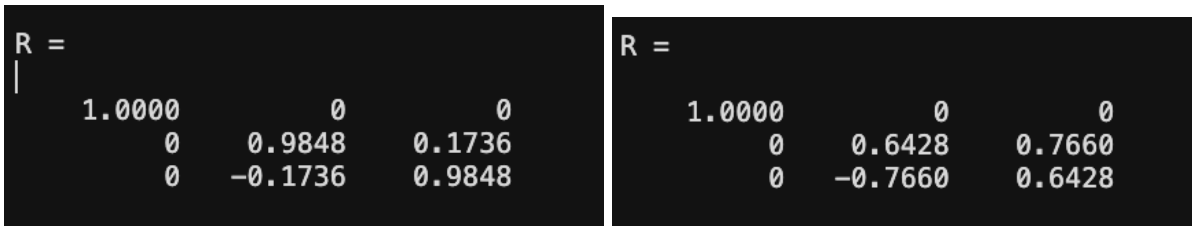



Figure 10. First and final matrices using openLoopIntegration_NoBias.m in MATLAB

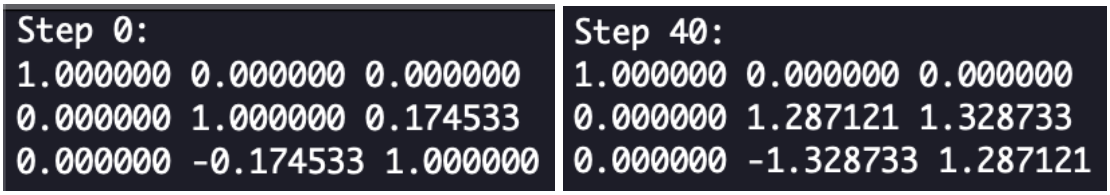



Figure 11. First and final matrices using Forward Integration and Attitude Propagation

# Part 4: Open Loop Gyroscope Integration

## 4.1 Open Loop Using Simulated Data

## Project Design

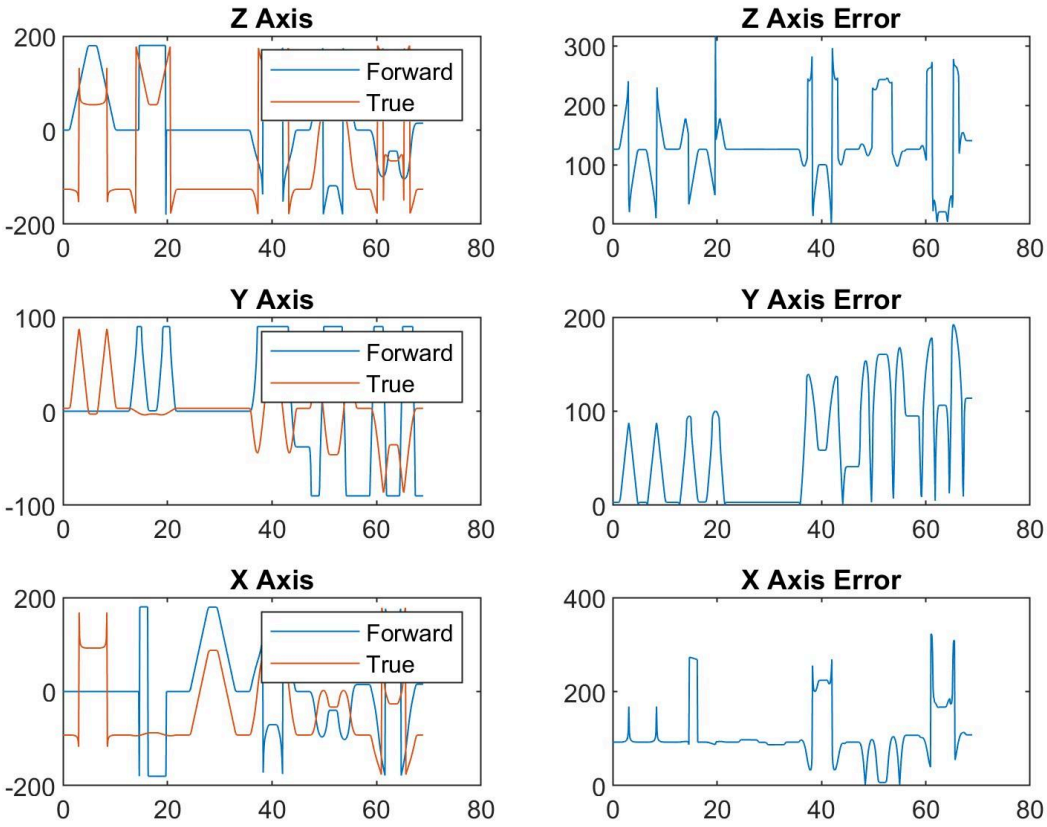
The purpose of this portion is to compare forward and exponential integration with Euler angles we extracted from the DCM to true Euler angles, and also compare with simulated data that is noisy and not noisy. To switch from forward and exponential integration, it required setting the flag to true or false in the IntegrateOpenLoop.m file when using IntegrateOpenLoop() function. We first created the simulated data without noise using delta as 0.1. We defined our phi, theta, and psi as the first, second, and third column of row one and converted them into rads. To get the bias of each axis, I averaged the simulated gyroscope data. Then we made our DCM matrix and created our identity matrix for the rotation matrix, defined as R_minus.

We implemented a for loop to iterate for the length of the true Euler matrix (same length as simulated data). I subtracted each gyroscope value by its proper bias and converted them into

rads. Dividing by 131 was required because of an hardware issue with the IMU. Next in my for loop, I called IntegrateOpenLoop() to integrate my data either forward or exponential. Then I used the Euler formulas to extract phi, psi, and theta and converted them into degrees. Then I added those values to a matrix for every iteration and set my R_minus to R_plus to make the function recursive. The rest of the code is devoted to plotting the true Euler and comparing it to the Euler angles we calculated. The comparison was subtracting both forms and plotting the difference.

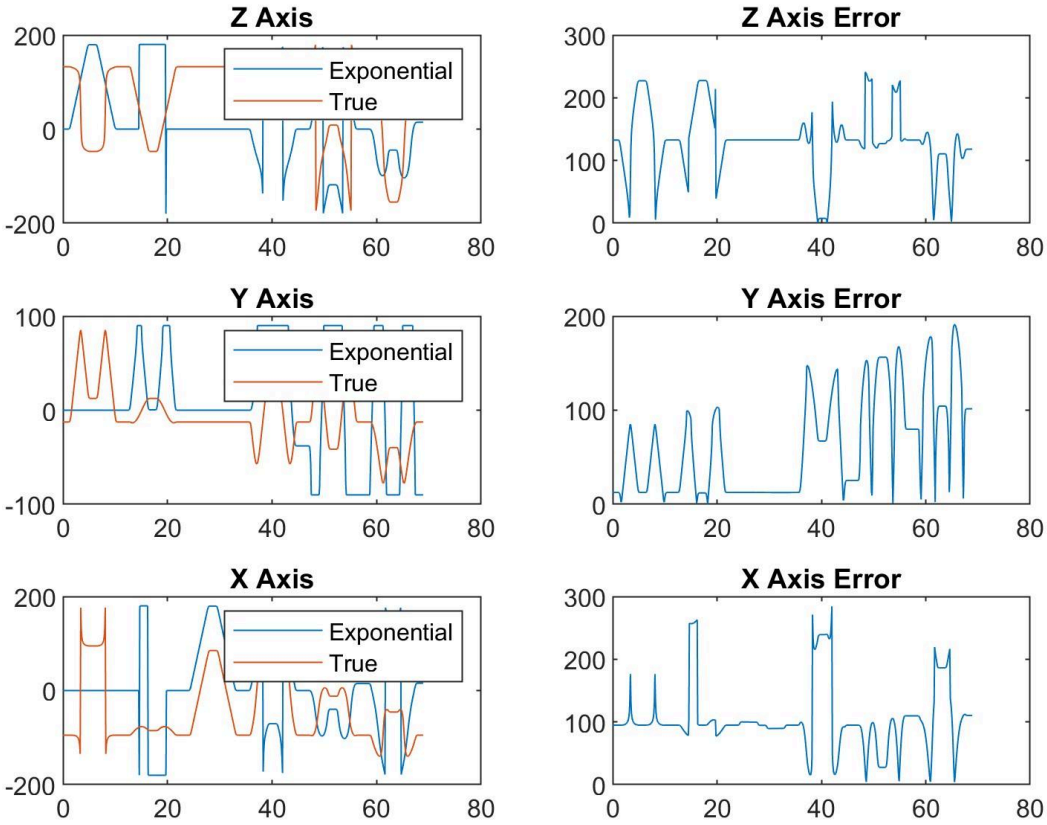
## Testing and Results

Plot 1 shows the non-noisy simulated data with forward integration.



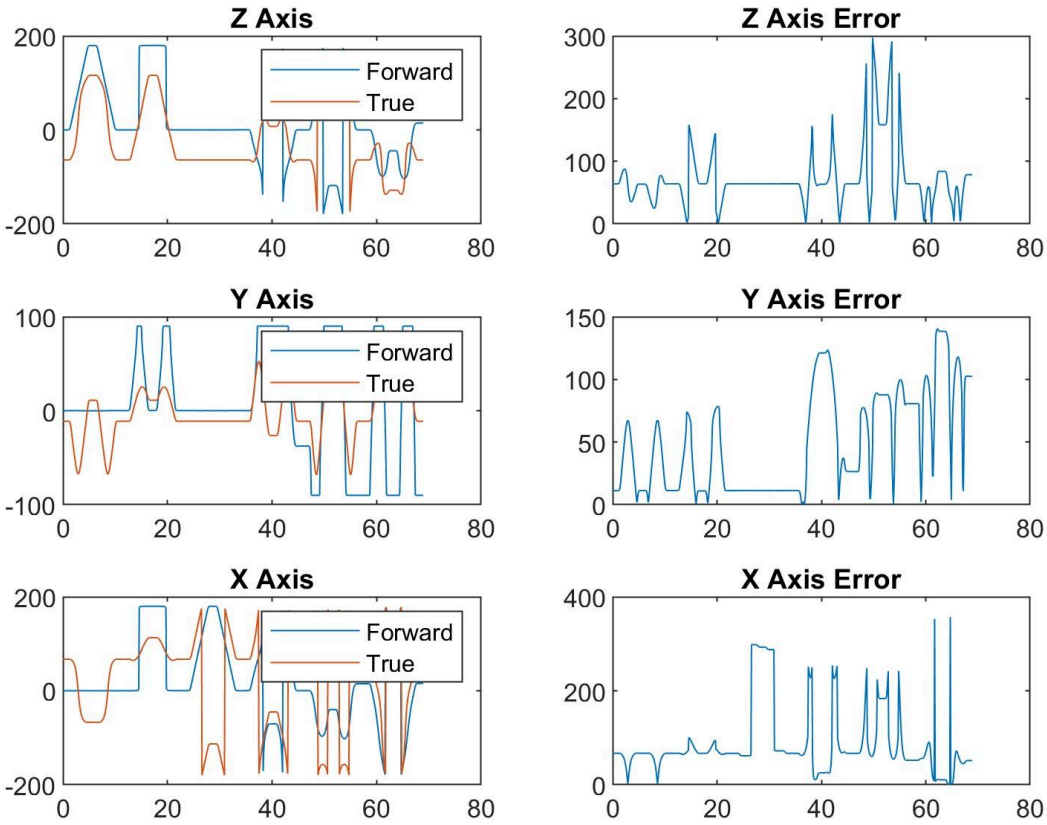

Plot 1

Plot 2 shows the non-noisy simulated data using exponential integration.



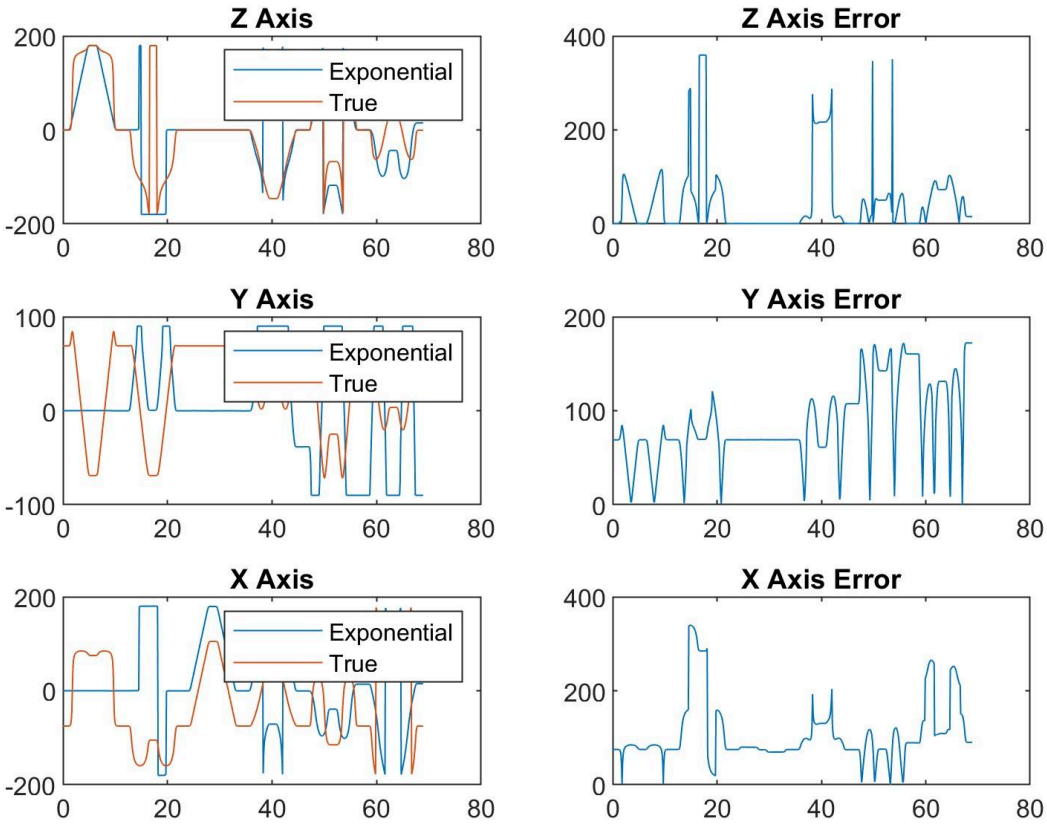

Plot 2

Plot 3 shows the noisy simulated data using forward integration.




Plot 3

Plot 4 shows the noisy simulated data using exponential integration.




Plot 4

With the error analysis we chose (to take the difference between true Euler and integration Euler), when the difference is closest to 0 is where the integration worked best. One phenomen I notice is that both integrations sometimes have a delay compared to the true Euler data, (x-axis is time created by time array mentioned in Project Design). This is apparent in Plot 1 and 2 of the y-axis. Also the integration isn't sensitive to big changes, it may be an offset issue; it's apparent in Plot 3 z-axis and y-axis, and Plot 4 x-axis. The true Euler doesn't usually start at zero where our integration always starts at 0. Observing all error plots, the integrations did not perform well compared to the true Euler angles. We can see how the integration follows the true Euler, but the

errors for all plots were across the whole spectrum and were reaching high differences many times.

## 4.2 Open Loop Using Real Data

## Project Design

This portion uses exponential integration for real data of the gyroscope. We took data for 10 seconds for the bias and 10 minutes for the gyroscope data. The MATLAB code is structured similarly to Part 4.1. We used my 10 minute data as the length of iteration for the for loop and subtracting by the bias values from the 10 second data.

To translate into C, we created formulas for p, q, and r by reading the current gyroscope reading, subtracting from the bias, converting to radians, and dividing 131. We put my values into a matrix and inserted the matrix, my delta value, and Rk (to store Euler angles) into the Rexp() function (created in Attitude_Estimate.c). To test, we used the dcm_to_euler() function and inputted my Rk matrix, and created variables to output to for pitch, roll, and yah.

## Testing and Results

The bias values of the gyroscope 10 second data are shown below.

$$Z_{bias} = 8.2164,\ X_{bias} = -14.5187, Y_{bias} = -24.4944$$

When we translated the code into C, the live values that our code generated were incorrect. They were below 5 degrees no matter the orientation. We tried evaluating the Rexp() function we wrote and couldn't find a solution. For completeness, we also tried taking out the divide by 131 for my p, r, q values and they were a little higher but not above 5 degrees.

# Part 5: Closed Loop Estimation

## Tuning the gains (Simulated Data)

In this part of the lab, we addressed the challenges of noisy gyroscopes with time-varying biases and the uncertainty of the initial sensor orientation. Straight integration of the gyroscopes leads to attitude errors, and simple integration does not align the estimated orientation with the "truth." To overcome these issues, we utilized closed-loop estimation, which incorporates feedback into the decision-making process. By measuring gravity and the Earth's magnetic field in the body frame and comparing them to their known values in the inertial frame, we generated a feedback signal to correct the attitude estimate.

We implemented a feedback mechanism using the cross product of the measured unit vector in body coordinates with the same quantity rotated into the body frame using the current attitude estimate $\widehat{(R)}$. In the equation below, $\overline{a_b}$ is in the body frame and $\overline{a_i}$ is in the inertial frame, and both have been converted to unit vectors before generating the error.

$$\omega_{meas_a} = \left[\vec{a}_b \times\right] (\hat{\mathcal{R}}\vec{a}_i)$$

Figure X. Correction Term

This error was used to adjust the gyroscope vector with proportional (Kp) and integral (Ki) gains for both the accelerometer and magnetometer corrections.

Our tasks involved tuning the gains on simulated data to achieve a balance between tracking accuracy and noise rejection. We explored different gain settings for the accelerometer and magnetometer separately, as well as in combination, to understand their impact on attitude tracking and bias estimation.

The gain settings were a portion of the lab where we struggled to see actual changes in the results. Despite continuous experimentation with the K values in the matlab code, we did not fully understand how it was changing the plots. Theoretically, it made sense but when it came to actually seeing the changes they were difficult to see. As a result we landed on the gain values that were given initially with the IntegrateClosedLoop.m file.

## Testing and Results

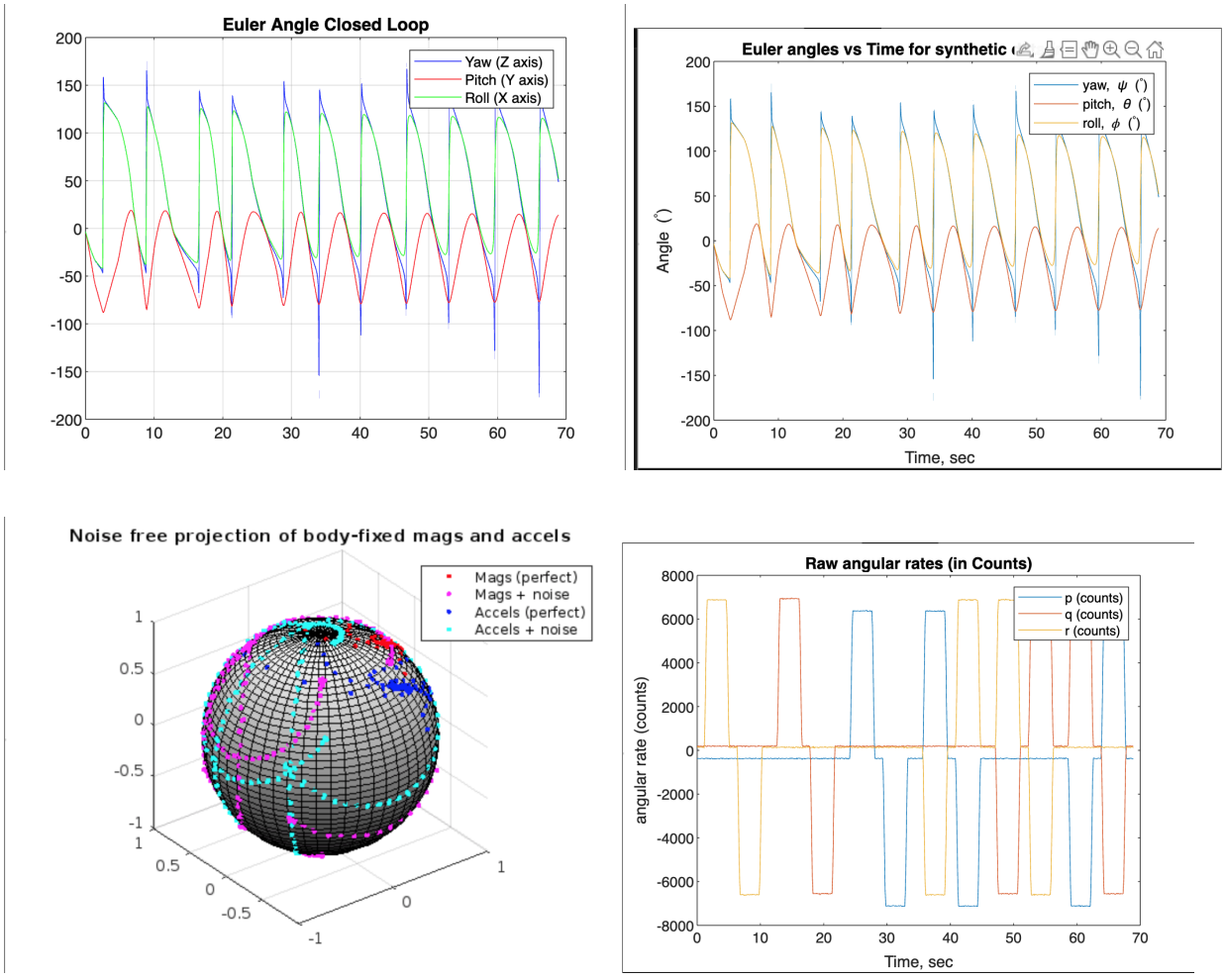



Figure 11. Closed Loop Estimation

## Part 6: Misalignment of Accelerometer and Magnetometer

### Project Design

In this portion, we generated the misalignment matrix using the MisalignmentTestCode.m file. I used my C code to generate the Euler angles using the dcm_to_euler function.

### Testing and Results

The misalignment matrix and Euler angles generated are shown below.

```
DCM Matrix:
0.016900 -0.032000 -0.001000
0.039200 0.015200 -0.013800
-0.004700 0.026900 -0.002100


 Euler Angles:
Roll: 94.46 degrees
Pitch: 0.27 degrees
Yaw: 66.68 degrees
```

## IV. Conclusion

In this lab, we implemented attitude tracking using a direction cosine matrix (DCM). We utilized open-loop gyroscope integration and DCM to Euler angles conversion with both simulated and real data. We delved into closed-loop estimation methods, leveraging feedback from accelerometers and magnetometers to improve accuracy. Addressing the misalignment of accelerometer and magnetometer sensors, we ultimately achieved a complementary attitude estimation by integrating magnetometer feedback.