Attitude Estimation

ECE-167/L: Sensing and Sensor Technology

University of California Santa Cruz

Purpose

The purpose of this lab is to get a handle on sensor fusion, and learn to integrate the gyros and other three-axis sensors in order to develop a useable attitude estimation algorithm driven by the actual sensors you have. You will learn how to use the accelerometer and magnetometer to estimate the gyro biases on the fly and to correct the attitude estimate. Attitude estimation, while somewhat esoteric, is a crucial component of all autonomous navigation and demonstrates sensor fusion in a readily accessible way. Having a functional attitude estimation system (both hardware and software) will definitely set you apart from most other students.

Hardware provided and intended use

Uno32, Sparkfun IMU Module, Earth's Gravity and Magnetic Field (provided free of charge).

1 Background

With the advent of smart phones, the price of integrated low-cost MEMs accelerometers, gyros, and magnetometers have dropped to an amazing degree. While these MEMs sensors are certainly not navigation (or even tactical) grade, they can be combined to give very good high bandwidth information about the orientation of the device in space (the *attitude* or *pose* of the device).

Like most MEMs sensors, however, these miniaturized devices have a plethora of error sources, including temperature drift, non-linearity, hysteresis, and some cross coupling between channels due to the manufacturing process. While more expensive sensors have much better spec's, these come at vastly higher cost ($100K+), and often with severe restrictions on use due to ITAR[1] restrictions.

In the previous lab you calibrated the scale factor and offset errors for the 3-axis sensors, and got a handle on the gyro bias and drift. In this lab, you will learn to solve the non-linear differential equation in order to track the attitude (in the form of the direction cosine matrix—DCM). Note that this is an excellent example of combining multiple flawed sensors in order to get a better answer. This is known as sensor fusion, and

[1] ITAR – International Traffic in Arms Regulations, passed in 1976, has severe restrictions on re-exportation of guidance devices. See: https://en.wikipedia.org/wiki/International_Traffic_in_Arms_Regulations

there are many techniques to do this. The specific one you will be implementing in this lab is known as complementary filtering. Other techniques such as Kalman Filtering (KF), Extended Kalman Filtering (EKF), and Muliplicative Update Extended Kalman Filtering (MEKF) are all beyond the scope of this class (but very interesting topics).

Also note that the 9DOF IMU moniker from both Invensense and Sparkfun is quite misleading. The 9DOF comes from the fact that there are 3 different 3-axis sensors on board (accelerometer, magnetometer, and gyro). Since each is measuring a different phenomenon, then there are 9 different measurements (10 with the temperature) available at any point in time, thus 9DOF. However, these sensors are combined through the process of sensor fusion and attitude estimation to estimate (not measure) the inertial displacement (x,y,z) and attitude (roll, pitch, yaw) of the device. Thus in any real sense, they can be used to get a 6DOF solution (which is all that exists in inertial space).

Further note that these specific devices are incapable of resolving the attitude to the accuracy required to remove gravity from the accelerometer output. That is to say that the residual error in gravity orientation will swamp the inertial acceleration of the device itself, and thus the inertial position (double integration of the acceleration) will be entirely unusable. Realistically, these devices are used to estimate attitude and not position. Technically, this is called an AHRS (Attitude Heading Reference System), not an IMU (Inertial Measurement Unit) or INS (Intertial Navigation System).

In this lab, you are going to implement an AHRS system from the raw outputs of the Sparkfun unit, and you are going to do it in C![2]

2 Familiarization with MATLAB Code

The instructor went over several things that you will need to implement in MATLAB in order to demonstrate how the attitude direction cosine matrix (DCM) works, and what was needed to integrate the gyros, feedback errors from auxiliary sensors, estimate the biases on the gyros and generally see how things worked in general.

Before you can imagine what is going on with the DCM, you need to get the concept of coordinate frames down. Specifically, we imagine a BODY frame with x out the nose, y out the right wing, and z down (to make a right handed coordinate frame). The BODY frame is rigidly attached to the body, such that it moves and rotates with the body. The other coordinate frame we use is the INERTIAL frame, which is a coordinate frame that is typically defined as X pointed North, Y pointed East, and Z pointed down (the so-called NED definition).[3]

In order to move from one coordinate frame to the other, we need a method to characterize that transformation. There are several ways to do this (known as attitude parameterizations), but the one we will use is the rotation or direction cosine matrix (DCM). The DCM is a 3×3 orthonormal matrix that carries all the information needed to move from one coordinate frame to another. The orthonomality property means that each column or row of the DCM has a unit norm. It also means that the inverse is the transpose (and it has a determinate of 1). For the math inclined among you, the DCM belongs to the special orthogonal 3 group ($\mathcal{SO}(3)$).

One of the key things to remember is that the DCM transforms vectors in the INERTIAL coordinate frame into the BODY frame. That is, if you want to know where the "nose" of the object is pointing in the INERTIAL frame, then you would calculate that as:

[2]Welcome to the big leagues. While the math behind attitude estimation is usually not taught until graduate level classes, we are going to have you implement an algorithm to do it (which we provide you with). Many of you will go on to do capstone design classes or other projects where attitude estimation is required, this is going to be something useful to you in the future, so pay attention.

[3]There are other inertial frames depending on the scale of motion you expect. For our needs, a flat earth model (the local tangent plane) works quite well.

$$\frac{\hat{e}_x}{I} = \left[\mathcal{R}\right]^T \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

.

$\frac{\hat{e}_x}{I}$ is the body-x unit vector expressed in the INERTIAL frame, which turns out to be the first column of $\left[\mathcal{R}\right]$. That is, take the transpose of the DCM, $\left[\mathcal{R}\right]^T$, to get the BODY to INERTIAL frame transformation, then multiply that by the cardinal unit vector in the x-direction. It is useful to keep track of what coordinate frame you are going from and which one you are going to.[4]

Spend some quality time with the MATLAB scripts that the instructor has made available to you. Understand what is going on inside them. You will be implementing these yourself in C on the micro, so you should be intimately familiar with how they work. More on this in Part 5.1.

3 Conversion from DCM to Euler Angles

Given that you cannot plot 3D coordinates on your micro very easily, you will need to be able to display something on the OLED to see what you are doing and if it makes sense. These are going to be the three Euler angles (from the 3-2-1 set) in degrees. These are defined to be: Ψ (yaw), Θ (pitch), Φ (roll). The conventions are that $+\Psi$ is a rotation from North towards East, $+\Theta$ is a rotation about the new body y-axis and is nose up, and $+\Phi$ is a rotation about the new (new) body x-axis and is right side down.

$$\left[\mathcal{R}\right] = \begin{bmatrix} \cos\Theta\cos\Psi & \cos\Theta\sin\Psi & -\sin\Theta \\ \sin\Phi\sin\Theta\cos\Psi - \cos\Phi\sin\Psi & \sin\Phi\sin\Theta\sin\Psi + \cos\Phi\cos\Psi & \sin\Phi\cos\Theta \\ \cos\Phi\sin\Theta\cos\Psi + \sin\Phi\sin\Psi & \cos\Phi\sin\Theta\sin\Psi - \sin\Phi\cos\Psi & \cos\Phi\cos\Theta \end{bmatrix}$$

You have been given in class the matrix form of the DCM from Euler Angles (above). You are going to write C code to extract the Euler angles from the DCM (by picking parts from the matrix and doing inverse trig on them). Note that you will need to get to degrees in order to make intuitive sense.[5]

Also, demonstrate that the above matrix is, in fact, orthonormal (each row and column has unit length, is perpendicular to the others, and that the transpose is the inverse). Include this in your lab report.

Make yourself a C function that you can call that returns the angles for a given rotation matrix. The input to this function should be a pointer to a 3×3 rotation matrix (stored in floats) and the return should be the angles *in degrees* (not radians). Be careful to define how you are going to store the 3×3 rotation matrix in memory.[6]

[4]The reason you want to keep switching coordinate frames is that some things are very easily measured in the body frame (e.g.: the actual sensors) but other things are more easily measured or known in the inertial frame (e.g.: gravity). Often you will go through several different frames to solve a problem.

[5]As much as we would like to say we can think and visualize radians, the truth is we are very conditioned to degrees.

[6]You should remember that you wrote an entire 3×3 matrix library as one of your labs in CMPE13 (now CSE13E). There is a method to the madness, and we try very hard to make your assignments relevant to real life.

4 Integration of Constant Body-Fixed Rates

Reviewing what we went over in Lecture, there are two equations that we need to track the evolution of the rotation matrix. The first is the skew symmetric (or cross product) matrix:

$$\left[\vec{\omega}\times\right] = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$

The $\left[\vec{\omega}\times\right]$ matrix is the cross product in matrix form such that $\left[\vec{\omega}\times\right]\vec{b} = \vec{\omega}\times\vec{b}$. The components of the $\vec{\omega}$ vector are directly measured by the body-fixed gyros and are given the names of $[p, q, r]$, and each is the *rotation rate* about their respective BODY axis unit vectors (e.g.: p is the rotation rate around BODY x-axis).

The second equation we need is the DCM differential equation:

$$\frac{\partial}{\partial t}\mathcal{R} \triangleq \dot{\mathcal{R}} = \left[\vec{\omega}\times\right]\mathcal{R}$$

which is a very compact way of writing 9 differential equations of which 3 are direct and 6 are constraints (imposed by the orthonormality of $\mathcal{R}$). This is the rather famous formulation for how the DCM evolves over time assuming a constant rotation rate across the time step (ΔT). We cannot solve it in closed form, so we must integrate it numerically.

Given a starting DCM ($\mathcal{R}_0$), measure the body fixed rates, put them into the skew symmetric matrix, and update the DCM:

$$\mathcal{R}^{+} = \mathcal{R}^{-} + \left[\vec{\omega}\times\right]\mathcal{R}^{-}\Delta t$$

The issues with this is that the matrix $\left[\vec{\omega}\times\right]\mathcal{R}^{-}\Delta t$ is not a proper rotation matrix, and even if it was, you don't do successive rotations by addition, you do them by multiplication. Thus, the DCM is going to slowly (or not so slowly) drift out of orthonormality.[7]

The other way we will use to integrate the DCM is to use the maxtrix exponential form. First remember that the solution of: $\dot{x} = ax$ is $x(t) = e^{at}x_0$. For a vector differential equation: $\dot{\vec{x}} = A\vec{x}$ there is an analogous solution: $\vec{x}(t) = e^{At}\vec{x}$ where A is now a matrix, and e^{At} is the matrix exponential. Without getting into the details, there is a closed form for the matrix exponential of the attitude prorogation (see the MATLAB `Rexp.m` function on the class website).

The important part to note is that the matrix exponential form, $\mathcal{R}_{exp}(\vec{\omega}\Delta t)$ is itself a rotation matrix, and represents the incremental rotation across the time step Δt. Thus, we can propagate the DCM across the time step using matrix multiplication:

$$\mathcal{R}^{+} = \mathcal{R}^{-}\mathcal{R}_{exp}(\vec{\omega}\Delta t)$$

In order to understand how the open loop integration of the DCM works, first play with the MATLAB code that does the open loop integration. Note that in a normal AHRS application, you would be integrating the gyros at a high rate to get high bandwidth information, but correct it with other sensor measurements at a much lower rate.

Warning: Be very careful with your gyro measurement units. You are going to get them from your sensor in signed 16-bit ints, and need to convert them to rad/s before you do any integration. Convince yourself that you have this conversion correct before proceeding.

[7]Remember that orthonormality is defined that the DCM inverse is equal to its transpose, that is: $[R]^{-1} = [R]^T$. Another way of stating that is that $[R][R]^T = I_{3\times3}$. You should test this and see how bad it gets as you proceed. There are techniques to re-orthonormalize a matrix, but they are beyond the scope of this lab.

What you are going to do is to set up the C code to do the integration given constant inputs on [p,q,r] and see if the DCM evolves in a way that makes sense to you.[8] It is pretty key here to familiarize yourself with the MATLAB code to see how things are being updated and to really understand the visualizations. You are going to do this in two ways: (1) using the simple forward integration; and (2) using the matrix exponential form. See how fast your DCM drifts out of orthonormality using both methods.

Note that you need a *sinc* function for the matrix exponential. Sin(x)/x is not going to cut it because it will give a divide by zero error when you have no rotation (which is often). You are going to have to do this on your own, an easy way to do this is the first few terms of a Taylor expansion near zero and switch to sin(x)/x farther away.[9]

5 Open Loop Gyro Integration

In order to track the attitude as you move the body around, you are going to integrate the gyros using the code that you developed in Part 4. You will also display the attitude (in the form of the Euler angles on the OLED) in real time. You should also save the gyro data (and the mag and accel data) so that you can run it through the integration on MATLAB in order to check your results.

5.1 Open Loop Using Simulated Data

To give you a clean place to start, we have created a file that will generate realistically noisy gyro, magnetometer, and accelerometer data that you can use to benchmark your algorithms on MATLAB before going over to C. The file `CreateTrajectoryData.m` takes as its input a time step (Δt) and a noise flag (either true or false). It outputs a $3 \times n$ vector for accelerometers, magnetometers, gyros and the true Euler angles. The accels and mags are in floating point numbers with unit norm, and the gyro outputs are in scaled ints (as you would get from the MPU), and the Euler angles are in degrees. The noise flag corrupts the outputs with realistic noise levels, however the Euler angles are always the true ones without noise. The function always starts from a random orientation.

You can inspect the function to see how the vectors are being generated (or not). There are a whole lot of tricks in there to generate rotation motion on one of the body axis at a time. The important part is that you will have a data set with a known attitude that you can play with to convince yourself that your algorithms are working.

Info: The output of the simulated data for the mags and accels are in *unit norm* numbers. That is, each is a (noisy) unit vector pointed in the direction that the mags and accels would measure *once you have applied your ellipsoidal correction that you generated in Lab 3.* When working with real data, it is very important to apply the ellipsoidal correction **before** feeding these data to the estimation algorithm.

Start with the noise flag set to false, and generate some data using the function `CreateTrajectoryData.m` in MATLAB. Set your time step to match what you expect to see on your real data (e.g.: approximately 50Hz).

[8]If you want to convert it to Euler angles, we have provided a function that will animate the motion of a "paper airplane" going through the sequence of Euler angles (see: `AnimateAttitude.m`). This is provided only as a visualization tool that was useful to the instructional staff when testing the functions. Use it only if it makes things clearer to you.

[9]Yes, we expect you to derive this and to show your work in the lab report, as well as the thresholds where you switch from the approximation to sin(x)/x.

Since this data is noise free, the only thing you need to worry about is the initial orientation which can be extracted from the Euler angles and cast into the DCM using the formula in Part 3. Integrate the gyros using the extracted initial DCM in MATLAB, using both forward integration and matrix exponential form.

Since you have the true Euler angles, extract Euler angles from your integrated DCM, and compare them (plot the errors). See how well you do with both methods. You might find that comparing statistics (means and standard deviations) of the errors is useful.[10] For exploration, see what happens if you don't initialize the DCM to the true one. Do you ever converge back to the true Euler angles?

Now, repeat the process with noisy data. Here, you are going to find the gyro biases by looking at the first second of simulated data. Get the average gyro biases over the first second, then subtract those biases from each subsequent gyro reading. Now integrate them again. How badly do things drift with the noise on the gyros. Again, compare the true vs calculated Euler angles to show how well you did. Again, try again with the DCM initialized to the identity matrix and see what happens.

5.2 Open Loop Using Real Data

Now that you have verified that your algorithms work (at least on simulated data in MATLAB), it is time to see how well they work on real data. You can record the data from your micro and run it through your MATLAB code that you wrote in Part 5.1; you won't have access to the true Euler angles, but you should see if you get reasonable motion.

Once you have convinced yourself that the open loop integration is working, then implement your open loop integration in C on the micro. Implement this using the matrix exponential form (and remember that you are taking the matrix exponential of $[p, q, r] * \Delta t$).

Before beginning the integration, remember to set the sensor on the bench and take 10 seconds of data or so in order to remove the (constant) bias from the gyros. We will get to removing time varying biases later.

6 Closed Loop Estimation

The gyros are noisy, and have a bias (and bias drift), thus straight integration of the gyros will lead to attitude errors. There is also the pesky problem that you never know what orientation the sensor is starting off in, and thus your $\mathcal{R}_0$ will be in error. Simple integration does not bring it into alignment with "truth."

Given that we measure both gravity (assuming small accelerations) and the Earth's magnetic field in the body frame, and we know what these are supposed to be in the inertial frame, we can compare them and use the error to generate a feedback signal to the estimator to force it to converge to the correct attitude (where they both match).

Specifically, we take the cross product of the measured unit vector (in body coordinates) with the same inertial quantity rotated into the body frame using the current attitude estimate ($\hat{\mathcal{R}}$) to form our error. That is:

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

where $\vec{\bullet}_b$ is in the body frame, and $\vec{\bullet}_i$ is in the inertial frame. Both vectors have been converted to unit vectors before generating the error. The cross product is the incremental rotation that would be required to bring the body measurement into alignment with the transformed inertial quantity.

[10]The MATLAB function `histfit.m` is good for visualizing this kind of thing.

Thus ω_{meas_a} can be viewed as an additional rate that needs to be integrated along with the gyros (scaled by a gain, K_{p_a}). The time varying bias on the gyros uses the same error to adjust the bias, but with an "integral" gain, K_{i_a}. There are corresponding errors for the magnetometer (ω_{meas_m}) with corresponding gains (K_{p_m} and K_{i_m}). The exact implementation can be viewed in the provided MATLAB file: `IntegrateClosedLoop.m`. The gains are inside the function, and can be altered for estimator tuning.

6.1 Tuning the Gains (simulated data)

Since the feedback terms (ω_{meas_a} and ω_{meas_m}) are giving you a quasi-rate to add to the gyros (much in the same way the bias is subtracted from the gyros), they are scaled via the gains and added into the gyro rate measurements. Since the rates are then integrated over Δt seconds, the roll of the proportional gains (K_{p_a} and K_{p_m}) is to set *how much* of the correction to apply at this time-step.

This is a trade-off, one the one hand, setting K_p high means that you take the full correction at that time-step, where setting K_p low means that you are slowly walking it in. This is going to depend on the relative noise of the aiding sensors (mags/accels) versus the gyros. If the aiding sensors are noisy, you want a small gain, if the aiding sensors are very accurate, you want a big gain. This is the classic *tracking vs noise amplification* trade-off. Good tracking means that you chase the noise on the aiding sensors; Good noise rejection means that you track much more slowly.

Proportional Gain Feedback: Using the simulated data (including noise), estimate the gyro bias from the first second of data, but leave the initial DCM as $I_{3\times3}$. Using only the accelerometer proportional feedback only (set K_{p_m}, K_{i_m}, and K_{i_a} to 0), play with the gain K_{p_a} to see how well you get the attitude to track truth (which you have from the simulation). The quantity $K_{p_a}\Delta t$ should be less than one (see how big you have to make it to make the estimate horrible). You should notice that you cannot seem to get yaw to track using only the accelerometer. Why is that?

Now do the same with only the magnetometer data. Play with the ranges of K_{p_m} to see how that changes compared with just the accelerometer. You are also going to have issues with an error you cannot get rid of, but it won't be neatly all in yaw for the magnetometer. Again, thing about why that is the case.[11]

And finally, use both the magnetometer and accelerometer together (still only proportional gain) to see how quickly you converge into the true attitude estimate. For all of the above, you have been subtracting the constant bias from the gyros that you measured in the first second of data.

Proportional Integral Gain Feedback: Next we turn our attention to the gyro bias. Now you set your initial bias ($\hat{b}$) to zeros, and let the integral gain (K_{i_a}) walk the bias estimate in. Again, details of the implementation are in the m-file. As before, do this just with the accelerometer, just with the magnetometer, and with both. As a general guideline, the integral gains should be a small fraction of the proportional gains (e.g.: K_{i_a}) = $K_{p_a}/10$). You can play with these to see how fast the bias converges (since you know the true value from the first second of the data). The hope is that by playing with the simulated data, you can gain some intuition as to how to tune your gains.

The key metrics for assessing how good are your gains: (1) Time to converge on true attitude, (2) Time to converge on true biases, and (3) Noise in the attitude estimate. Aggressive proportional gains will do well on (1) but badly on (3), likewise aggressive integral gains will do well on (2) but badly on (3). As we stated before, it is a trade-off.

[11] In order to get some insight into this, you have to look at the problem geometrically; You are, in effect, using a single line-of-sight vector as your error (either gravity or magnetic field). What kind of information do you get out of that?

7 Feedback Using Only the Accelerometer

As stated before, the open loop integration works well (and is what you use for the high bandwidth attitude tracking). However, it relies on the gyros having no bias (and no bias drift) and having the initial attitude correct. None of these assumptions are true. Note that you could get that pitch and roll parts of the DCM from the accelerometer measurements to initialize the DCM, but that you can only get yaw with the magnetometer.

In order to drive the solution into the correct DCM even with biases on the gyros and a wrong DCM to begin with, you are going to implement the closed loop attitude estimation using the accelerometer feedback (as you did with simulated data above). Note that you will be estimating the gyro biases and the DCM. The feedback is going to come in as an alteration of the [p,q,r] that you send into the matrix exponential function. Play with the gains, Kp and Ki and see if you can find the bounds of stability and where it works well.

The algorithm should be entirely in C running on your microcontroller. If you output the data, you can run it through your code developed in Part 6.1 to verify that your C code matches the MATLAB output.

You should note that using the accelerometer feedback means that you cannot fix the yaw since you have no information about it. This is due to the fact that gravity points straight down, thus there is no projection of the gravity vector into the inertial horizontal plane. Thus nothing to measure in the yaw axis.

8 Misalignment of Accelerometer and Magnetometer

We assume that the individual sensing elements are aligned well at the time of manufacture and that they share a consistent sensor axis. This might not be the case. Using the same tumble data, we can extract the misalignment of one sensor relative to the other.[12]

This is going to be done using some fancy linear algebra techniques that you don't really need to understand (see Prof. Elkaim's misalignment paper if you want to). In short, you are going to use a set of n Wahba's problem solutions to figure out the individual rotations for each measurement pair, and then use those rotations to generate a single set of points to solve for the misalignment between master and slave axis sets.

One of the sensors is set to "Master." In our case this is going to be the accelerometer. The magnetometer is going to be the "Slave" sensor. A misalignment matrix will be generated such that the magnetometer measurements can be put in a consistent axis set as the accelerometer. You will need the true magnetometer reading in inertial coordinates (from NOAA site). Or you can get it by setting the magnetometer carefully level and pointed north and averaging the measurements over a decent time.

Run the misalignment code in MATLAB and generate the misalignment matrix. Use this to rotate your magnetometer measurements into the accelerometer sensor frame. Does this make sense to you? If you convert the misalignment matrix to Euler angles, you can get a sense of how much rotation the misalignment is doing. It is small or big?

[12]Being pedantic, you are actually extracting the relative rotation of each sensors' *coordinate frame* relative to the other. The algorithms you are using work for both small and very large misalignments. This is very useful if you have some ambiguity about the sensor axes from the datasheet.

9 Full Complementary Attitude Estimation (Mag and Accel Feedback)

For the last part of the lab, you are now going to add in the magnetometer into the feedback. Thus you are going to have two Kp's and two Ki's, one for the accelerometer and one for the magnetometer.

This is using the "Full Monty" of calibration and aiding that you have available to you. The spherical calibration you did for your mags and accels, the bias removal of the gyro, and the misalignment calibration of the magnetometer to the accelerometer frame.

Output the results of your attitude estimate to the OLED in Euler Angles, and save them (along with the raw data) if you can. See if it all makes sense to you. Again, you can run the data through the MATLAB code to verify that you are getting correct computations.

Congratulations: this is not an easy thing to do, and many many people who attempt this get the math completely wrong. Keep this one in the "I know how to do this" file for later use during your careers. At the very least, it will impress your future employers even if you don't use it beyond this class.

10 Confirmation of Gyro Integration using Wahba's Problem Solution

Now that you have the calibrated and corrected magnetometer and accelerometer measurements (using the spherical calibration you did last lab) and the misalignment matrix you computed in Part 8 above, it is possible to calculate the DCM directly as a solution to Wahba's problem, given these two vector measurements.

You can do this using the Markley SVD solution (in MATLAB) or you can do this much more simply using the TRIAD algorithm (which is computationally very easy). In order to get TRIAD to work, you will need to be working with unit vectors (length of 1).

See https://en.wikipedia.org/wiki/Triad_method for the TRIAD method (especially Eq. 1-9). Be a little careful in that the $\hat{A}$ that comes from Eq. 9 might be the transpose of the DCM or might be the DCM itself (the code we give you is correct).

Compare the Wahba's solution (either from the Markley SVD or from the TRIAD) to what you got from the gyro integration (with feedback). You should be close. See if you can plot the errors and make some comments about what you see.

Appendix A: Work Flow for Attitude Estimation

This section is provided to give you a work flow to follow in your code, such that you don't skip any crucial steps. It assumes you have the ellipsoidal correction matrices from the tumble test, and the misalignment matrix between magnetometer and accelerometer triads.

```
Algorithm 1: Data Processing for Attitude Estimation

Require: $\tilde{A}_a, \tilde{B}_a, \tilde{A}_m, \tilde{B}_m$;                 // from tumble test
Require: $\mathcal{R}_{mis}$;                                   // from batch misalignment
Initialize: $\mathcal{R}_0$ and $\hat{b}$;                      // initial DCM and gyro bias
while (every $\Delta t$) do
    Read MPU Data (Int16) ;
    $\hat{a} \leftarrow \tilde{A}_a \vec{a}_{\texttt{int16}} + \tilde{B}_a$;              // convert accels to unit norm
    $\hat{m} \leftarrow \tilde{A}_m \vec{m}_{\texttt{int16}} + \tilde{B}_m$;              // convert mags to unit norm
    $\hat{m}_{aligned} \leftarrow \mathcal{R}_{mis} \hat{m}$;                // align magnetometer to accels
    convert gyro$_{int16}$ to [rad/s] ;
    $\omega =$gyro$_{rad/s} - \hat{b}$;                             // remove gyro bias
    Feedback: $\omega_{meas_a} = \left[\vec{a}_b \times\right] (\mathcal{R}^T \vec{a}_i)$ ;
    Feedback: $\omega_{meas_a} = \left[\vec{m}_b \times\right] (\mathcal{R}^T \vec{m}_i)$ ;
    $\omega_{total} \leftarrow \omega + K_{p_a} \omega_{meas_a} + K_{p_m} \omega_{meas_m}$ ;
    Integrate: $\mathcal{R} \leftarrow \mathcal{R} * \mathcal{R}_{exp}(\vec{\omega}_{total} \Delta t)$ ;
    $\dot{b} \leftarrow -K_{i_a} \omega_{meas_a} - K_{i_m} \omega_{meas_m}$ ;
    $\hat{b} \leftarrow \hat{b} + \dot{b} \Delta t$ ;
end
```

The initial DCM can be either set to $I_{3\times3}$ or estimated using your initial magnetometer and accelerometer readings. Either way it will converge quickly. The initial gyro bias estimate is set by taking an average of your initial gyro measurements while the sensor is still on the table.

The accels and mags come in as signed 16 bit integers that need to be converted to unit vectors (which is done using the ellsipsoid calibration from the tumble test). These give you your *body* measurements for the complementary filter correction.

The magnetometer unit vector is rotated into alignment using the misalignment matrix ($\mathcal{R}_{mis}$), to bring it into consistent coordinate frame with the accelerometer.

The gyros are first scaled and converted to rad/s, and then have the bias subtracted off of them. The feedback terms are computed using the cross product and the known inertial quantities (scaled to unit vectors).

The rotation rate from the gyros - biases are augmented using the scaled feedback terms, and then this is integrated using the matrix exponential form. The bias rate is computed using the same feedback terms, and the bias updated.

Appendix B: Useful MATLAB files for this lab

In this lab, you are asked to use MATLAB to confirm that everything is working, and then to replicate the MATLAB code in C on your micro, and again to re-confirm that you get the correct results (or at least results consistent with the MATLAB code).

To that end, here are some useful functions that we have written for you to use; they should all be available on the CANVAS site, and they need to be in the same folder to function in MATLAB.[13]

1. `[Rplus] = IntegrateOpenLoop(Rminus, gyros, deltaT)`
2. `[Rplus, Bplus] = IntegrateClosedLoop(Rminus, Bminus, gyros, mags, accels, magInertial, accelInertial, deltaT)`
3. `R_exp = Rexp(w)`
4. `rx = rcross(r)`
5. `[Acc,Mag,wGyro,Eul] = CreateTrajectoryData(dT,noiseFlag)`
6. `PrettyPlotAttitudeData(dT,Acc,Mag,wGyro,Eul)`
7. `R = DCMfromTrial(mags, accels, magInertial, accelInertial)`

and for the misalignment matrix:

1. `[R_est, Pbody] = whabaSVD(B_v,E_v,w)`
2. `[lambda,phi] = extractAxis(R)`
3. `[Rmis, Pbody] = AlignMasterSlave(Mb,Sbmeas,mi,si,Rmishat_0,i)`

You can access each of these functions help information by typing “help” and the function name at the MATLAB prompt. You can also look at the internal code to see how it was done and what special commands were used.

[13]Unless you want to be altering paths, but it is easier to just put them in the same folder.