Lab #4: Attitude Estimation

ECE-167: Sensing and Sensor Technology
University of California Santa Cruz

In the previous lab you calibrated the scale factor and offset errors for the 3-axis sensors, and got a handle on the gyroscope bias and drift. In this lab, you will learn to solve the nonlinear differential equation in order to track attitude via the direction cosine matrix (DCM). Attitude estimation is a crucial component of all autonomous navigation. This lab guides you through combining the outputs of multiple flawed sensors in order to get a better result than each sensor alone. This is known as sensor fusion. The specific sensor fusion technique you will be implementing in this lab is known as complementary filtering. There are also other techniques beyond the scope of this class, such as Kalman Filtering (KF), Extended Kalman Filtering (EKF), and Multiplicative Update Extended Kalman Filtering (MEKF).

Systems that use attitude are known as Attitude Heading Reference Systems (AHRS)[1]. In this lab, you are going to implement an AHRS system from the raw IMU outputs, and you are going to do it in C.

Hardware needed

uC32, BNO055 IMU Module

Part 1: Familiarization with MATLAB Code

Before you can understand what is going on with the DCM, you need to understand the concept of coordinate frames. There are two types: body, and inertial. Figure 2 shows a BODY frame that has a flying aircraft 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, shown in Figure 1, which is a coordinate frame that is typically defined as *X* pointed north, *Y* pointed east, and *Z* pointed down (the so-called North-East-Down, or NED, definition).[2]

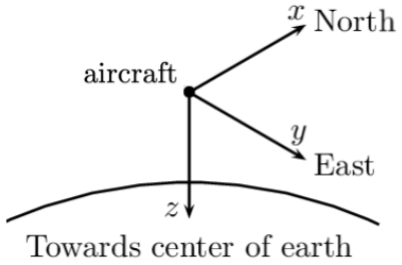

Figure 1: The NED coordinate system, with axes pointing north, east and down.

[1] Extremely sensitive IMUs can also be used to implement Inertial Navigation Systems (INS), which do in fact estimate position, but suffer from steadily growing error over time.
[2] 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.

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. We represent it as $[R]$. The orthonormality property means that each column or row of the DCM is orthogonal to the other rows/columns and, additionally, they all have a unit norm. It also means that the inverse is the transpose, and it also has a determinant of 1. Put succinctly, for a matrix Q, if it is orthonormal, $Q^TQ = QQ^T = I$.

One of the key things to remember is that **the DCM transforms vectors in the INERTIAL coordinate frame into the BODY frame**. If you want to do the opposite—a BODY to INERTIAL frame transformation—use the transpose of the DCM, $[R]^T$, then multiply that by the cardinal unit vector in the *x*-direction:

$$e_{x(I)} = [R]^T \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$e_{x(I)}$ is the body-*x* unit vector expressed in the INERTIAL frame, which turns out to be the first column of $[R]^T$. This will tell you where the the "nose[3]" of the object is pointing in the INERTIAL frame. It is useful to keep track of what coordinate frame you are going from and which one you are going to. 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.

Your task: Spend some quality time with the MATLAB scripts we have made available to you. Understand what is going on inside them. You will be implementing these yourself in C on the MCU.

Part 2: Conversion from DCM to Euler Angles

Given that you cannot plot 3D coordinates on your microcontroller 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[4] angles (in the '3-2-1' convention) in degrees. These are defined to be: Ψ (yaw), Θ (pitch), Φ (roll). The conventions, shown in Figure 2, are that +Ψ is a rotation from North towards East, +Θ is a rotation about the body y-axis that inclines the nose of the craft upwards, and +Φ is a rotation about the body x-axis that tilts the right side down. Notice that positive rotation in each axis is *clockwise.*

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

.

[3] Recall that the BODY x-axis goes out the nose of the plane, by convention

[4]Technically these are Tait-Bryan angles, not Euler angles, but it is very common to use Euler angles synonymously with pitch, roll and yaw.

Your task is to write a C function to extract the Euler angles from the DCM by picking parts from the matrix and doing inverse trigonometry on them. 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). The matrix form of the DCM from Euler Angles is detailed above.

Hint: You wrote a 3×3 matrix library as one of your labs in ECE13–refer to this for how to store the 3×3 rotation matrix in memory.

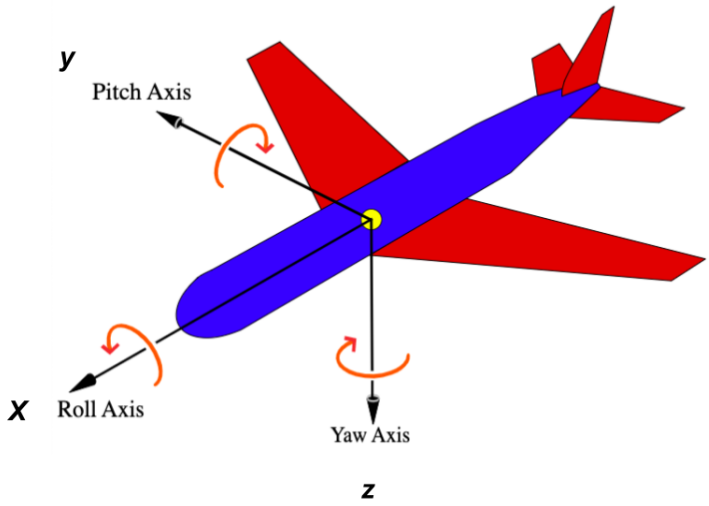

Figure 2: Pitch, yaw and roll from [1]

Part 3: Integration of Constant Body-Fixed Rates

Tracking attitude requires integrating rotation over time from a known starting point. This is known as *attitude propagation.* There are two equations that we need to track the evolution of the rotation matrix that stores the DCM form of the attitude. The first one is the skew-symmetric[5] (or cross product) matrix:

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

The [ω×] matrix is the cross product in matrix form such that [ω×] b = ω × b. The components of the ω vector are directly measured by the body-fixed gyroscopes 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, q is the y-axis and r is the z-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 R). This is the rather famous formulation for how the DCM evolves

[5] A skew-symmetric matrix has the property that A^T = -A and $a_{ij} = -a_{ji}$

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 (R_0), measure the body fixed rates, put them into the skew-symmetric matrix, and update the DCM using *forward integration*:

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

The issue with this is that the matrix $[\omega\times]R^{-}\Delta t$ is not a proper rotation matrix, and even if it were, you don't do successive rotations by subtraction/addition, you do them by multiplication. Thus, the DCM is going to slowly (or not so slowly) drift out of orthonormality.[6]

To properly integrate the DCM, we will use the *matrix exponential form*. First remember that the solution of $\dot{x} = ax$ is $x(t) = e^{at}x_0$. For a vector differential equation $\dot{x} = Ax$, there is an analogous solution, $x(t) = e^{At}x$, where A is now a matrix, and e^{At} is the matrix exponential. There is a closed form for the matrix exponential of attitude prorogation (see the MATLAB `Rexp.m` function in your git repo and the Attitude Cheat Sheet on Canvas for details).

The important part to note is that the matrix exponential form, $R_{exp}(\omega\Delta t)$ 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:

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

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 gyroscopes 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 gyroscope measurement units—make sure to convert them to rad/s before you do any integration. Convince yourself that you have this conversion correct before proceeding.

Your task is to write C code to do open-loop integration given constant inputs on [*p, q, r*]. You are going to do this in two ways: (1) using the simple forward integration; and (2) using the matrix exponential form. Compare how fast your DCM drifts out of orthonormality between the two methods.

Note that in the matrix exponential approach you end up having terms similar to a sinc function, $\sin(x\Delta T)/x$. This is problematic because it can cause a divide by zero error when x is zero, which happens whenever you have no rotation (which is often). An easy way to get around this is to use the first few terms of a Taylor expansion near zero[7] and switch to $\sin(x\Delta T)/x$ farther away. Derive this and show your work in the lab report, and also state as the thresholds where you switch from the approximation to $\sin(x\Delta T)/x$.

---

[6] Remember that orthonormality is defined as the inverse is equal to its transpose, that is: $[R]^{-1} = [R]^T$, or $[R][R]^T = I_{3\times3}$. You can 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.

[7] Technically, since you are expanding about zero, this is a Maclaurin series.

Hint 1: 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. Make sure the DCM evolves in a way that makes sense to you.[8]

Hint 2: The Attitude Estimation Cheat Sheet has detailed guidance on implementing the Taylor/Maclaurin expansion needed for the matrix exponential approach.

Part 4: Open Loop Gyroscope Integration

In order to track the attitude as you move the body around, you need to integrate the gyroscopes using the code that you developed in Part 3—except this time [*p, q, r*] won't be constant. You will also display the attitude (in the form of the Euler angles on the OLED) in real time. Also be sure to save the gyroscope data (and the magnetometer and accelerometer data) so that you can run it through the integration on MATLAB in order to check your results.

4.1 Open Loop Using Simulated Data

To give you a clean place to start, we have created a file that will generate realistically noisy gyroscope, 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). This file gives you a data set with a known attitude that you can play with to convince yourself that your algorithms are working.

It outputs a $3 \times n$ vector for the accelerometer, magnetometer, and gyroscope, as well as the true Euler angles. The accelerometer and magnetometer values are in floating point numbers with unit norm, and the gyroscope outputs are in scaled integers, 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. Note that the scaling for the simulated data isn't necessarily the same as your real hardware.

Warning: The output of the simulated data for the magnetometer and accelerometer are in unit norm numbers. That is, each is a (noisy) unit vector pointed in the direction that the magnetometer and accelerometer 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.

Your task is to integrate the simulated gyroscope output using the extracted initial DCM in MATLAB, using *both* forward integration and matrix exponential form. Start by generating some data using the function `CreateTrajectoryData.m` in MATLAB with the noise flag set to false. Set your time step

[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.

to match what you expect to see on your real data (e.g.: approximately 50Hz). 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 2.

Your next task is to extract Euler angles from your integrated DCM, and compare them against the true Euler angles. Plot the errors. See how well you do for both forward integration and matrix exponential form. You might find that comparing statistics (means and standard deviations) of the errors is useful.[9] What happens if you initialize the DCM to the identity matrix? Do you ever converge back to the true Euler angles?

Your final task for this section is to repeat the process above with noisy data by setting the noise flag in `CreateTrajectoryData.m` to true. Compare the true vs. calculated Euler angles to see how well you did. How badly did things drift with the noise added? Next find the gyroscope biases by looking at the first second of simulated data, then subtract those biases from each subsequent gyroscope reading. Now integrate and compare again—did things improve? What about when the DCM is initialized to the identity matrix?

4.2 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. Record real data from your microcontroller and run it through your MATLAB code that you wrote in the previous section. Though you won't have access to the true Euler angles, you should still see if you get reasonable results. Once you have convinced yourself that the open loop integration is working as intended, the next step is to do it in C.

Your task is to build off what you did in Part 2 and implement open loop integration in C, given *non-constant* inputs on [*p, q, r*]. This time you only need to implement the matrix exponential form. Before beginning testing your code, remember to set the sensor on the bench and collect 10 seconds or so of data in order to remove the (constant) bias from the gyroscopes. *Note:* You will learn how to remove time varying biases later.

Part 5: Closed Loop Estimation

The gyroscopes are noisy, and have a time-varying bias, thus straight integration of the gyroscopes 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 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). Systems that integrate feedback into decision making are known as *closed loop.*

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

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 ($\hat{R}$) to form our error:

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

where a_b is in the body frame, and a_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.

ω_{meas_a} is the accelerometer's correction term for the gyroscope vector, ω. It can be viewed as an additional rate that needs to be integrated along with the gyroscopes (scaled by a proportional gain, K_{p_a}). The time varying bias on the gyroscopes uses the same error to adjust the bias, but with an "integral" gain, K_{i_a}. Proportional and integral gains are the P and I of PID controllers. You can read more about them here. There's also another correction term 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 defined inside this function, and you will be setting sensible values.

Tuning the gains (simulated data)

Since the correction feedback terms ω_{meas_a} and ω_{meas_m} are giving you a quasi-rate to add to the gyroscopes (much in the same way the bias is subtracted from the gyroscopes), they are scaled via the gains and added into the gyroscope rate measurements. Since the rates are then integrated over Δt seconds, the role 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. On the one hand, setting K_p high means that you take the full correction at that time-step. On the other hand, 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 (magnetometer/accelerometer) versus the gyroscopes. 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.

Your task is to explore different gain settings on simulated data, using the instructions below. Note in your lab report any interesting things you observe, and what you settled on for your gains for this part.

Proportional gain feedback: Using the simulated data (including noise), estimate the gyroscope 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. 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, think about why that is the case.[10] 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 gyroscopes that you measured in the first second of data.

Proportional integral gain feedback: Next we turn our attention to the gyroscope bias. Now you set your initial bias (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.: $K_{i\_a} \cong K_{p\_a}/10$).

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 develop some intuition as to how to tune your gains.

Key metrics for assessing how good your gains are: (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.

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 gyroscopes having no bias (and no bias drift) and having the initial attitude correct. None of these assumptions are true. Note that you could get the 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. This means that when using only accelerometer feedback, you cannot fix the yaw since you have no information about it—since gravity points straight down, there is no projection of the gravity vector into the inertial horizontal plane. Thus nothing to measure in the yaw axis.

In order to drive the solution into the correct DCM even with biases on the gyroscopes 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 gyroscope 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, K_p and K_i and see if you can find the bounds of stability and where it works well.

Your task is to implement closed loop attitude estimation using the accelerometer feedback should be entirely in C running on your microcontroller. *Hint:* If you output the data, you can run it through your code developed in Part 5 to verify that your C code matches the MATLAB output.

[10] 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?

Part 6: 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.[11]

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 primary and secondary axis sets.

One of the sensors is set to primary. In our case this is going to be the accelerometer. The magnetometer is going to be the secondary 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.

Your task is to 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. Is it small or big?

Part 7: Full Complementary Attitude Estimation

Your task for the last part of the lab is to add in the magnetometer into the feedback in *addition* to the accelerometer. Thus you are going to have two K_p's and two K_i'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 magnetometer and accelerometer, the bias removal of the gyroscope, 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! You've made it to the end of the last lab.

[11] 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.

Part 8: Lab Report

There is no check-off for this lab.

We want a report of your methodology and results, such that any student who has taken this class would be able to reproduce your results (albeit with some effort). If there are any shortcomings in this lab manual, please bring it to our attention so that we may improve it for the next class.

Lab 4 Rubric

- 25% code quality (comments, readability)
- 25% code compiles
- 50% lab report
    - General format followed; all sections of lab addressed
    - Writing detailed enough for replication
    - Image quality
    - Writing quality (grammar, spelling, appropriate tone)

References

[1] https://en.wikipedia.org/wiki/Euler_angles#/media/File:Yaw_Axis_Corrected.svg

Appendix A: Workflow for Attitude Estimation

This section is provided to give you a workflow 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* Δt*)* **do**
  Read ICM 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] ;
  $\omega = \text{gyro}_{rad/s} - \hat{b}$; // remove gyro bias
  Feedback: $\omega_{meas_a} = \left[\vec{a}_b \times\right] (\mathcal{R}\vec{a}_i)$;
  Feedback: $\omega_{meas_a} = \left[\vec{m}_b \times\right] (\mathcal{R}\vec{m}_i)$;
  $\omega_{total} \leftarrow \omega + K_{p_a}\omega_{meas_a} + K_{p_m}\omega_{meas_m}$;
  Integrate: $\mathcal{R} \leftarrow \mathcal{R}_{exp}(\vec{\omega}_{total}, \Delta t) * \mathcal{R}$;
  $\dot{b} \leftarrow -K_{i_a}\omega_{meas_a} - K_{i_m}\omega_{meas_m}$;
  $\hat{b} \leftarrow \hat{b} + \dot{b}\Delta t$;
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 gyroscope bias estimate is set by taking an average of your initial gyroscope measurements while the sensor is still on the table. The accelerometer and magnetometer values come in as signed 16 bit integers that need to be converted to unit vectors (which is done using the ellipsoid 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 (Rmis), to bring it into a consistent coordinate frame with the accelerometer.

The gyroscopes 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 gyroscopes - 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 functions 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 microcontroller , 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 are all in your repo.

- `[Rplus] = IntegrateOpenLoop(Rminus, gyros, deltaT)`
- `[Rplus, Bplus] = IntegrateClosedLoop(Rminus, Bminus, gyros, mags, accels, magInertial, accelInertial, deltaT)`
- `R_exp = Rexp(w, deltaT)`
- `rx = rcross(r)`
- `[Acc,Mag,wGyro,Eul] = CreateTrajectoryData(dT,noiseFlag)`
- `PrettyPlotAttitudeData(dT,Acc,Mag,wGyro,Eul)`
- `R = DCMfromTriad(mags, accels, magInertial, accelInertial)`

and for the misalignment matrix:

- `[R_est, Pbody] = whabaSVD(B_v,E_v,w)`
- `[lambda,phi] = extractAxis(R)`
- `[Rmis,Pbody] = AlignPrimarySecondary(Pb,Sbmeas,pi,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.