Lab 0x05 - State Estimation

New Attempt

- Due Nov 30 by 11:59pm
- Points 20
- Submitting a file upload
- File Types pdf and py

Overview

The purpose of this assignment is twofold: first, to write a new hardware driver for your BNO055 inertial measurement unit (IMU), and second, to develop a state-estimation algorithm that allows you to combine data from your sensors with your model of Romi from HW 0x03 to better predict the true state of Romi.

This lab is nominally a 1-week lab, however depending on progress we may spill into a second week to finish the lab properly. Your instructor will talk with lab groups and determine if this is necessary based on student progress.

Background

Inertial Measurement Units (IMUs) are small electronic devices that measure orientation through indirect means. A standard IMU has either two or three sensors, each with three axes.

Accelerometer

To measure the direction of "down" on planet earth, a 3-axis accelerometer can be used, as it will read $1G$ of acceleration when at standstill. The IMU will use this information to establish a coordinate system in which the z-axis is aligned with earth's gravity.

Gyroscope

To measure and track orientation, most IMUs use a 3-axis gyroscope to measure angular velocity. The angular velocities are then integrated to track the 3D orientation of the IMU. Angular velocities can not be integrated directly because they are "pseudovelocities"; that is, they don't represent the derivative of physical displacements. To track orientation, the velocities must first be transformed into individual angular rates. Read more about Euler angles for additional detail.

Magnetometer

To track the heading of the IMU in absolute coordinates, many include a 3-axis magnetometer on top of the accelerometer and gyroscope. Without a magnetometer, there is no way to determine

whether the IMU is pointing north, south, east, or west; however, with the magnetometer acting as a compass, an absolute orientation can be determined.

Assignment

It's best to approach this assignment in two steps. First, write and test a driver class to get data from your IMU in a useful manner.

IMU Driver

By now you have written at least three hardware drivers (one for the motor driver, one for the encoders, and one for your line sensors) and should therefore be familiar with the coding structure for driver classes. You will need to create a Python class that facilitates interaction with the BNO055 IMU using I^2C communication.

Your driver must include the following features, some of which will only make sense after reading through the datasheet:

1. An initializer that takes in a **pyb.I2C** object preconfigured in **CONTROLLER** mode. Historically the term "master" has been used to refer to the primary device on an I2C bus, with the term "slave" used for other devices; however, in this course, we will adopt the words "controller" and "peripheral" instead. You may still see some of the old language in published documents like datasheets.

2. A method to change the operating mode of the IMU to one of the many "fusion" modes available from the BNO055.

3. A method to retrieve and parse the calibration status byte from the IMU.

4. A method to retrieve the calibration coefficients from the IMU as binary data.

5. A method to write calibration coefficients back to the IMU from pre-recorded binary data.

6. A method to read Euler angles from the IMU to use as measurements for feedback. You may also want a simpler method that returns individual Euler angles, such as the heading, as pitch and roll may not be very useful when Romi is driving on flat ground.

7. A method to read angular velocity from the IMU to use as measurements for feedback. As with the heading, you may want a method that returns individual angular velocities, such as the yaw rate

(similar to heading rate).

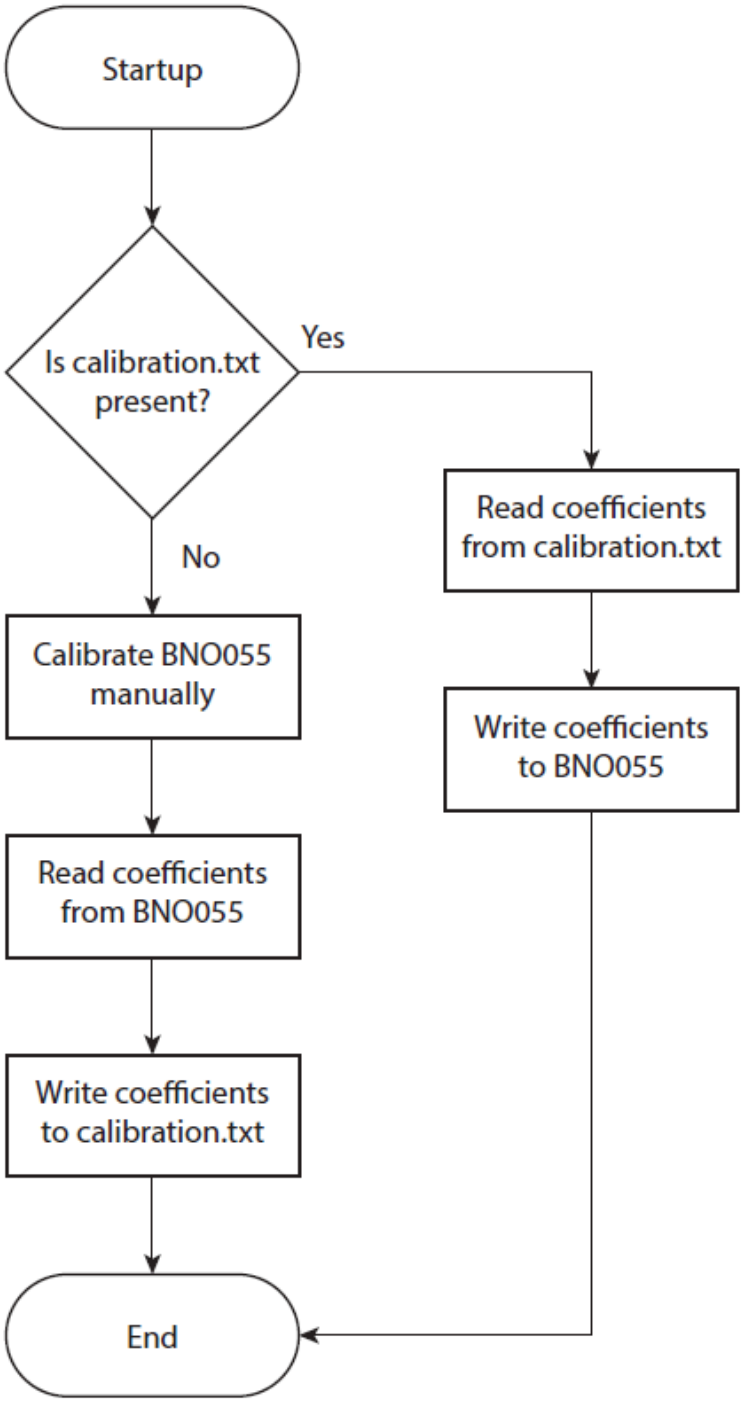
The BNO055 has other features that are not useful in our context; keep the code simple for now and add more features if you need them in the future.

Note: the calibration process for the BNO055 is tedious, but extremely beneficial. It is possible to use an uncalibrated BNO055, but the sensor readings will be unreliable and may jump around sporadically as the IMU continues to try and self calibrate internally.

Recommended Calibration Procedure

The BNO055 attempts to calibrate as soon as it is put into one of its operational modes (anything other than the configuration mode) and will continuously recalibrate during operation. While the sensor does output readings during initial calibration, these values cannot be properly trusted until calibration is complete as indicated by the calibration status byte. Once the calibration is complete, it is possible to retrieve the calibration coefficients from the IMU so that in the future the calibration process can be skipped.

The flowchart shown below depicts a routine to calibrate on startup when needed or to read calibration data from a text file on the Nucleo, when possible, to skip calibration. Without storing and retrieving calibration values from a text file it will be necessary to manually recalibrate each time the BNO055 is power-cycled which will added up to hours of wasted time by the end of the quarter.



State Estimation

The second goal of this assignment is to build a state estimator using a model of Romi along with input from sensors.

1. Refer back to your HW 0x03 submission and review your state equations. For this lab you will want to simplify the model to include only four states representing the core linear dynamics of Romi. The nonlinear dynamics required to determine Romi's specific location will be reincorporated later. For now, use the following state vector, input vector, and output vector. Note that the state vector will be

referred to by $\hat{\mathbf{x}}$ instead of the usual $\mathbf{x}$; the "hat" indicates that we are working with *estimated* states, not true states which are hidden within the system. Likewise, the estimated output will be $\hat{\mathbf{y}}$ instead of $\mathbf{y}$.

$$\hat{\mathbf{x}} = \begin{bmatrix} \Omega_L \\ \Omega_R \\ s \\ \psi \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} V_L \\ V_R \end{bmatrix}, \quad \hat{\mathbf{y}} = \begin{bmatrix} s_L \\ s_R \\ \psi \\ \dot{\psi} \end{bmatrix}$$

Notice that the selected output variables are the specific values that are measurable with Romi's sensors:

- The first two outputs, s_L and s_R, represent the translational displacement associated with each wheel, which are directly measured by the wheel encoders. These could be replaced by θ_L and θ_R if your group prefers to work in terms of angular displacement for each wheel instead of in terms of translational displacement.
- The next two outputs, ψ and $\dot{\psi}$, are the heading/yaw angle and the heading/yaw velocity, coming directly from your IMU; whether you use yaw or heading depends on which mode you've selected for the IMU and if absolute orientation is important to your group.

The state variables and input variables are the simplest needed to produce the measurable outputs and should be mostly consistent with your work on HW 0x03.

If needed, rearrange or rewrite your state and output equations according to the provided definitions for the state, input, and output variables above. Now that these equations are linear, you should use the LTI form for state and output equations.

$$\dot{\hat{\mathbf{x}}} = A\hat{\mathbf{x}} + B\mathbf{u}$$
$$\hat{\mathbf{y}} = C\hat{\mathbf{x}}$$

2. Now your group needs to decide whether to implement the RK4 solver or to work with a discretized model that does not require a solver. This distinction has been discussed in lecture, but the following recap may help you choose which option to pursue,

   The RK4 solver will allow direct implementation of the Continuous Time observer model but will run slower and may pose practical challenges. However, if you can get it working reliably it will be more convenient when it comes to tuning.

   The discretized model will not allow implementation of the Continuous Time observer model, but instead will require you to preprocess the model to convert it to a Discrete Time observer model. This will run faster in code but require more active work each time you want to retune the controller or make modifications.

   Your instructor recommends that you pursue the discretized option first, but also encourages you to

try both options.

3. If you want to pursue the RK4 solver, you can use these following steps.

    1. Now, refer back to your HW0x04 submission and how you implemented the RK4 solver. In that assignment the goal was to run a simulation with specific initial conditions over a specified duration of time. Now, for this lab, you will want to run your RK4 solver in a dedicated task and only run one iteration of the solver per iteration of your task. In this way you can co-simulate the state of the system due to the input vector $\mathbf{u}$.

        Use the `numpy` class in the `ulab` module built into the Micropython firmware to help implement the linear algebra needed for this assignment. Note that this implementation of numpy is limited in comparison to the full version available on a PC. For example, the overloaded matrix multiplication operator, `@`, cannot be used in Micropython; instead you will need to call the `numpy.dot()` function to perform matrix products. Otherwise the majority of your code from HW0x04 should port over directly.

        It may help with organization if you break this down into multiple functions. The following will be helpful:
        - A function that performs one "step" of the RK4 integration algorithm. This function should call the next two functions internally.
        - A function that implements the state equations, producing the derivative of the estimated state vector, $\dot{\hat{\mathbf{x}}}$.
        - A function that implements the output equations, producing the estimated output vector $\hat{\mathbf{y}}$.

        Once you build all this into a new task, route data from other pertinent tasks to this one through `Share` and `Queue` objects so that the task is aware of the applied motor voltages in $\mathbf{u}$ and each of the measurements in $\mathbf{y}$. You will not need to use $\mathbf{y}$ quite yet, but will need to shortly.

    2. At this point your code should be partially testable. That is, you should be able to test the functionality of your task along with the RK4 solver and your state and output equations. You should not expect the cosimulation to track the true behavior of Romi all that well as no feedback has been implemented yet. That is, while you can test the core functionality, the actual state estimation is not going to work well at this stage of development.

    3. After some initial debugging and testing of your code, modify your function that implements the state equations to incorporate feedback from the sensor measurements; you can of course write a new function if you prefer, but you shouldn't need both when you're done. The new function will implement the following modified form of your state equations that includes feedback from the

sensor measurements. The gain matrix L can be computed by pole placement or other means and is referred to as a the observer gain.

$$\dot{\hat{\mathbf{x}}} = A\hat{\mathbf{x}} + B\mathbf{u} + L\left(\mathbf{y} - \hat{\mathbf{y}}\right)$$

While this equation could be implemented directly, it will be simpler to implement and computationally less expensive to further manipulate the equation by plugging in $\hat{\mathbf{y}} = C\hat{\mathbf{x}}$ before implementing the equations.

$$\begin{aligned}
\dot{\hat{\mathbf{x}}} &= A\hat{\mathbf{x}} + B\mathbf{u} + L\left(\mathbf{y} - C\hat{\mathbf{x}}\right) \\
\dot{\hat{\mathbf{x}}} &= A\hat{\mathbf{x}} + B\mathbf{u} + L\mathbf{y} - LC\hat{\mathbf{x}} \\
\dot{\hat{\mathbf{x}}} &= \left(A - LC\right)\hat{\mathbf{x}} + B\mathbf{u} + L\mathbf{y} \\
\dot{\hat{\mathbf{x}}} &= \left(A - LC\right)\hat{\mathbf{x}} + \begin{bmatrix} B & L \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{y} \end{bmatrix}
\end{aligned}$$

Set up this way, you can preallocate the matrices $A_O = A - LC$ and $B_O = \begin{bmatrix} B & L \end{bmatrix}$ to avoid the expensive matrix operations during runtime.

4. If you want to pursue the discretized version of the observer model, follow these steps.

    1. Use the `numpy` class in the `ulab` module built into the Micropython firmware to help implement the linear algebra needed for this assignment. Note that this implementation of numpy is limited in comparison to the full version available on a PC. For example, the overloaded matrix multiplication operator, `@`, cannot be used in Micropython; instead you will need to call the `numpy.dot()` function to perform matrix products. Otherwise the majority of the syntax you used in HW0x04 should port over directly.

        Once you build all this into a new task, route data from other pertinent tasks to this one through Share and Queue objects so that the task is aware of the applied motor voltages in and each of the measurements in.

        Recall that the observer update equations, once discretized, can be represented as shown below.

        $$\hat{\mathbf{x}}_{k+1} = A_D\hat{\mathbf{x}}_k + B_D\mathbf{u}_k^*$$

        $$\hat{\mathbf{y}}_k = C\hat{\mathbf{x}}_k$$

        Where A_D and B_D are the results of the continuous to discrete conversion of the $A_O = A - LC$ and $B_O = \begin{bmatrix} B & L \end{bmatrix}$, respectively, and $\mathbf{u}_k^* = \begin{bmatrix} \mathbf{u}_k \\ \mathbf{y}_k \end{bmatrix}$ is the modified input vector consisting of the input to the physical system $\mathbf{u}_k$ concatenated with the measurement of the physical system output $\mathbf{y}_k$.

Make sure to use a sample time in your continuous to discrete conversion that matches with your task period. You can use tools like `c2d()` in MATLAB or `scipy.cont2discrete()` in Python. Note that the Micropython module `ulab` does not presently include functionality to do the continuous to discrete conversion on startup.

5. In theory, if everything has been implemented correctly and an appropriate gain matrix $\boldsymbol{L}$ has been utilized you should find good tracking between the estimated output $\hat{\mathbf{y}}$ and the measured output $\mathbf{y}$ and also good tracking between the estimated state $\hat{\mathbf{x}}$ and the true state $\mathbf{x}$.

   Do your best to tune and test and retune your observer until your state estimate tracks your measurements effectively. Once working properly this state estimation algorithm should provide more robust estimates of the true state of the system than would be achieved by using the measurements directly. That is, the observer uses both the system model predictions and the measurements together to come up with appropriate estimates of the states. A more aggressive gain matrix will favor the measurements more and the system model less.

6. When the state estimator is working reasonably well you can reincorporate the remaining dynamics that produces predictions of Romi's position. Unfortunately, Romi is not equipped with sensors to measure position directly, so the state estimator cannot correct the predictions of Romi's position with feedback as has been done for other states. That is, we must trust that the observer accurately determines Romi's heading and velocity, leading to accurate changes in Romi's location.

   For full transparency, accurate predictive tracking of Romi's location may or may not be possible without additional feedback from new sensors. One of the goals of this lab is to validate or invalidate the predictive tracking.

What can go wrong?

Consider the following list of potential "gotchas" that you may come across while testing.

- The units for each of your measurements must be consistent with your state-space model. That is, you should convert all variables to appropriate engineering units before using them in your state estimator.

- Make sure that the sign convention is correct for each variable without a shred of doubt. There may be some nuance to how the signs work as provided by the IMU, so make sure that your heading/yaw angles and velocities are consistent with your model.

- Observers only work if they are faster than the natural dynamics of the system being observed. That is, if you use pole placement, make sure to place the observer poles further to the left than the

natural system poles.

- Matrix math isn't cheap. Pay attention to the duration of your tasks and make sure that everything runs on time. If your RK4 solver doesn't run at the interval it is set for you will get bad drift in your observer predictions.

- It may be valuable to test the observer output under a variety of circumstances. What if Romi is still? What if it's moving in a straight line? What if it is driving in an arc? What about arbitrary motion? It may be possible to learn from different tests which of the gain values may need tweaking based on sensitivity to certain motion.

Deliverables

The final deliverables for this assignment will include a memorandum and several attachments.

Memo: Write a detailed but concise memo presenting your results from this assignment. How well did the state estimation work? In your response talk about the observed states in the estimated state vector $\hat{\mathbf{x}}$ and also about the predicted location of Romi, not handled directly by the observer.

Diagrams: Include in the body of your memo the task diagram your team has constructed along with state transition diagrams for each task implemented as a FSM.

If you feel that the images fit poorly within the body of the memo also include larger high-resolution images as attachments at the end of your memo that each fill up an entire landscape oriented page.

Source Code Include at the end of your memo a transcript of the source files you've written for this assignment. Do not include a transcript of the driver classes from previous labs unless you modified them significantly. Also include the source files themselves as attachments to the memo or as additional submissions on Canvas.