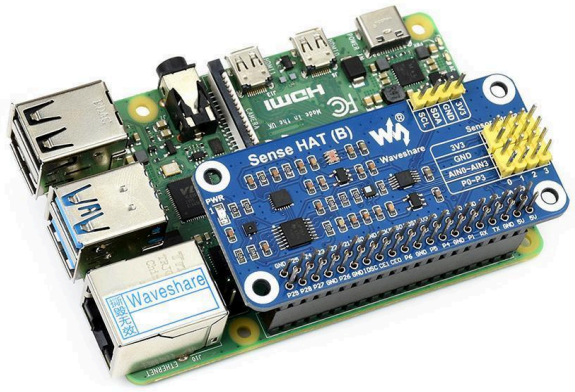
EE292S
Fall 2024

Project 1: Inertial Sensing

Introduction

In this project, you will develop a basic understanding of inertial measurement units, and the signal processing necessary to deal with noise impairments. The key goal is the development of a linear Kalman filter appropriate to the task of taking sensor inputs (accel/gyro/mag) and converting it into a spatial position. The process of using inertial sensors to determine position is called dead reckoning; for example, a car in a tunnel has no access to GPS signals, but the navigation guidance still has to be able to navigate. An in-vehicle navigation system usually is a mix of GPS (when available), with IMU-based odometry/positioning to "fill in" data when traveling through dead zones with no GPS signals.



A Raspberry Pi 4 (rpi4) equipped with a sensor "hat" module will be the development platform; the sense hat simply docks into the rpi4's 40 pin header. The sense hat offers additional functionality beyond just the IMU, including barometer, humidity, and raw A/D conversion (in case you might find any of those functions useful as additional input data streams). The inertial measurement unit on the sense hat is a TDK/Invensense ICM-20948; the reference datasheet for the part is attached. It is a 9DOF complete IMU, integrating a 3DOF accelerometer, 3DOF gyro, and a 3DOF magnetometer into a single MEMS package. Note that the ICM-20948 does not have an external crystal oscillator for size/cost/power reasons, so its internal timebase is subject to some drift (and by definition the Kalman filter has sensitivity to the sampling period; depending on how much performance you achieve, this fact may or may not play into your design decisions).

For all testing and performance optimization, assume that the IMU is only measuring human-being-level dynamics, i.e. you will test your code by manually moving the rpi4+sensehat, walking a few steps with it or even moving it by hand initially. We are assuming you will *not* be trying to measure acceleration/peak velocities while driving in a car, or sensing if the unit has been dropped. To be quantitative, assume that the maximum possible acceleration is 0.3g = 3 m/sec^2.

Practical Matters

Given the extensive support for the rpi4's general purpose I/O pins as libraries inside Python, we will use Python as the coding language for this (and all future) projects. The KalmanFilter class within the Python filterpy module will be extremely useful, especially as you start dealing with larger-dimensional state spaces. If you wish to code your own Kalman filter, please do not hesitate to do so, but it will add a considerable amount of work to debug and get right.

Demo Python code for dealing with the IMU is provided on the course website (along with demo code for the other sensors on the board). You can find this file in Canvas in ICM-20948.zip, under ICM-20948/Raspberry Pi/python/ICM20948.py. The IMU demo code defines a class that initializes the ICM-20948 ASIC, and provides class functions to read the sensor outputs. In case there is a need to change the IMU configuration, such as reducing the sensor bandwidth or peak acceleration range, the demo code illustrates how to write the on-chip register banks.

Note that several of the problems on homework #2 are intended to get you going on this project; you will have needed to have finished at least Part I in order to complete HW#2.

As a matter of practice, we normally capture data streams *with a known ground truth* from the sensor for whatever test cases are deemed important, and those traces are saved in a library. Some simple/obvious test cases:

- Sensor completely stationary
- Sensor starts at rest, accelerates in X, moves two feet, and then decelerates
- Sensor starting at rest, accelerates in one dimension, moves two feet, and reverses this to return to the original position .
- Sensor slowly rotating in place
- Sensor moving in a figure 8 in the X-Y plane (where "down", direction of gravity, is Z)
- All of the above in a warm room, or a cold room.

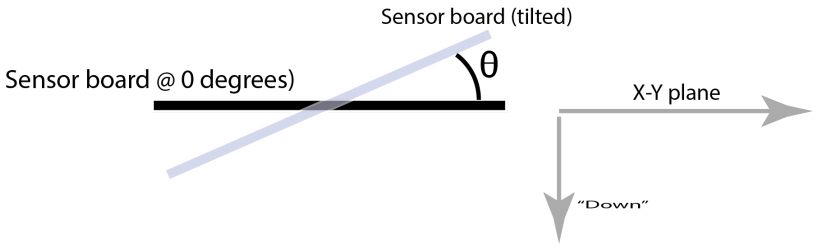
As the algorithm changes, we continuously test against the logged traces, versus testing live on the sensor every time we change something. In fact, you should begin this project by capturing traces, and then code/test your algorithm against those traces. Only after you feel you have something reasonable should you begin live testing with the actual sensor. You will find in industry that the data collection process is critical for (a) development efficiency, (b) gaining practical understanding of sensor behaviour, and (c) defining (quantitatively, as sensor data traces) all of the corner cases under which the sensor must function. Since ground truth is acquired/documented as part of each trace collection, the data collection process provides known references for assessing sensor error performance.

'ICM20948.py': After `icm20948_Gyro_Accel_Read()`, Accel and Gyro stores ADC values of readings. To convert from ADC values to real physical values (acceleration in m/s**2 and rotations in degrees/second):

- In ICM20938 datasheet (section 2 Features), Accelerometer and Gyroscope use 16-bit ADC.
- Search for `REG_ADD_GYRO_CONFIG_1` and `REG_ADD_ACCEL_CONFIG`, the full scale range of accelerometer and gyroscope can be found, e.g., ±500dps. The full scale range can be changed in ICM20948.py. Also you will need to map the ADC values to the full scale range to get readings in the right units.

Part I

- Bring up the sense hat on the rpi4. Write Python code that implements a simple gravity level; your code should plot, in real time, the angular deviation from level, where level is at 0°. The sensing function should be implemented in two ways - one using the gyro (pitch/roll/raw), and one with the accelerometer (x/y/z). Analyze the performance of the two methods (gyro vs. accel) - which is superior? Can you implement a means of fusing the two measurements to achieve greater accuracy than either of them alone? Your code should plot the highest accuracy signal (fused or otherwise) at a rate of 10 samples per second.



  Submit your code (with comments!), an output plot as you tilt the sensor at various known (ground truth) angles, as well as a brief analysis of how you optimized the fusion between the gyro data and the accelerometer data.
  Notes:
  - You can use your phone to get you ground truth angles (search "Angle Finder Apps"), or if you're feeling really retro use a protractor/string/weight (clinometer) to get your ground truth angles.
  - There are easy ways and complex ways to solve this part, achieving different levels of performance. If you are spending too much time to advance the performance, it's better to move on to later parts and then come back for superior performance.

- As discussed in lecture, implement code to measure and plot the Allan deviation for both the accelerometer and the gyro. Submit both code and plots. On the plots, indicate the breakpoints for the various 1/f, f, etc noise regimes (particularly for the gyro).
  Notes:
  - based on past runs, it usually takes at least 20mins of data collection to have the flat plateau in the Allan deviation plot.

Part II

- Using the accelerometer only, implement a 1-D Kalman filter to determine position based on acceleration measurements in a single axis (say, the X axis only). This will be similar in nature to the simple constant velocity example discussed in lecture, except that the measurement is in acceleration and not position. You will need to take out the bias in the accelerometer (you can use the other axis of the accelerometer / gyro to do this); you can either measure the bias and directly subtract it from the accelerometer output, or implement a calibration step in your code before you start the position tracking. Note

that you *will* have to take out whatever gravity component is leaking into the accelerometer measurement (and this is related to what you did in Part I).

To test, move your sensor along the single chosen axis for 6 feet, stop, and then move it back to its original position (you may need a longer USB cable for the testing). Plot sensor error as a function of distance traveled; your ultimate performance metric is error when the sensor is back at its original position (having traveled 6+6=12 feet). Submit code and sensor error plot. Analyze your error sources; document what you did in your code minimize to the error. Does how quickly you move impact your error?

Part III (Extra Credit)

- Extending this idea, create a Kalman filter that uses the X/Y axes of the accelerometer as well as the yaw sensor to determine position in an X/Y plane. The Kalman will naturally fuse all three pieces of data (two accels and one gyro). As above, move your sensor in a straight line away from its starting point for 6 feet and back; performance metric is deviation after the sensor has returned to the starting point. Submit code and plot of error vs distance. The plot should have multiple traces, corresponding to movement in various cardinal directions (does the sensor have a different error response if the unit is traveling north-south vs east-west?) Does this fusion with the gyro give a better accumulated error across the 12 feet traveled than the single-axis accelerometer by itself?

Some useful factoids

Python modules needed to access the GPIO pins on an rpi4:
(the following are shell commands necessary to get the needed Python modules; if you don't like apt-get, conda or another package manager is perfectly fine. Note that if your rpi4 is not connected to a network, these will obviously not work).

```
sudo apt-get update
sudo apt-get install python-pip3
sudo pip3 install RPi.GPIO
sudo pip3 install spidev
sudo apt-get install python3-smbus
```

I2C addresses on the board

(These are already embedded in the demo code for the board, but just in case)

ICM-20948: Inertial Measurement Unit: Device address: 0x68
ADS1015: A/D converter: Device address: 0x48
LPS22HB: Barometer: Device address: 0x5C
SHTC3: Temperature and humidity sensor: Device address: 0x70
TCS34725: Color recognition sensor: Device address: 0x29