Yaw Angle Estimation from Gyroscope and Magnetometer Based on Kalman Filter

Lavender Jiang
Advisor: Bobby Shen

May 3, 2018

1: Problem Statement

VOOF sensor uses IMU to measure the yaw angle, or the heading angle, of the IMR robot. A reliable and accurate yaw angle estimation is essential for both motion control and path planning. However, our current estimation, which integrates gyro reading over time, shows drifting error. To solve this problem, we propose to use Kalman filter to perform sensor integration for gyroscope and magnetometer. By doing so, we can obtain both the fast response of gyroscope and the long term reliability of magnetometer.

2: Modeling Gyro Error

The Model is based on the work of Neto, Mendes, et. al..[1] Let $\psi_g(t_k)$ be the gyro-estimated yaw angle (in degrees) at time t_k. Let $\omega(t)$ is the gyro z-axis reading. To synchronize the reading of magnetometer yaw and gyro yaw, set the initial gyro yaw as the initial compass yaw, that is $\psi_g(0) := \psi_m(0)$. For subsequent gyro yaw estimation, we integrate the angular velocity over time.

$$\psi_g(t_k) = \psi_g(0) + \int_0^{t_k} \omega(t)dt$$

However, there can be gyro errors in our yaw estimation. That is, let ψ_{true} be the true yaw angle; let the total gyroscope error at time t_k be $\delta\psi_g$, we have:

$$\psi_{\text{true}}(t_k) = \psi_g(t_k) - \delta\psi_g$$

The goal of our Kalman filter is to reliably find the gyroscope error $\delta\psi_g$, so that our estimation is close to the true yaw angle. The followings are explanation and justification of our model. For implementation, please see section 3. In the explanation, I would refer to specific equations in the implementation by writing **(Eq.n)**.

(2.1 Prediction)

Specifically, in gyro error, we might have scale factor error δk caused by temperature differences, and bias error b caused by integration drift. To model these errors, we assume that they all are zero-mean Gaussian white noises $u.$ and:

$$\frac{d}{dt}\delta\psi_g = \omega\delta k + b + u_\psi$$

$$\frac{d}{dt}\delta k = u_k$$

$$\frac{d}{dt}b = u_b$$

[1] Kalman filter-based yaw angle estimation by fusing inertial and magnetic sensing: a case study using low cost sensors. Emerald Group Publishing Limited. 2014.

To put this into matrix form, we have:

$$\frac{d}{dt}\begin{bmatrix}\delta\psi\\ \delta k\\ b\end{bmatrix} = \begin{bmatrix}0 & \omega & 1\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix}\begin{bmatrix}\delta\psi_g\\ \delta k\\ b\end{bmatrix} + \begin{bmatrix}u_\psi\\ u_k\\ u_b\end{bmatrix}$$

To obtain a discrete-time model, Neto and Mendes use inverse Laplace transform to find:

$$\begin{bmatrix}\delta\psi\\ \delta k\\ b\end{bmatrix} = \begin{bmatrix}1 & \Delta\psi_g & \Delta t\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}\delta\psi_g\\ \delta k\\ b\end{bmatrix} + \begin{bmatrix}S_\psi\\ S_k\\ S_b\end{bmatrix} \quad \textbf{(Eq.1)}$$

where S is the power spectral densities[2] of its corresponding Gaussian white noise u.

Let $\mathbf{x_{k+1|k}}$ be the predicted error of time $k+1$ based on information at time k, and let $\mathbf{x_{k|k}}$ be the predicted error of time k based on information at time k, we have:

$$\mathbf{x_{k+1|k}} = \mathbf{F}\mathbf{x_{k|k}} + \mathbf{w}$$

After we predict our state vector, we update the prediction matrix $\mathbf{P}$ with our transition matrix $\mathbf{F}$ and the covariance matrix $\mathbf{Q}$ associated with $\mathbf{w}$. **(Eq. 2)**

(2.2 Observation)

Our observation of the gyro error z is the difference between our gyro-estimated yaw angle ψ_g and magnetometer-estimated yaw angle ψ_m. Note that z is just a scalar, and let ψ_m be tilt-compensated yaw calculated from the data of accelerometer and magnetometer, we have:

$$z_k = \psi_g - \psi_m = (\psi_{\text{true}} + \delta\psi_g) - (\psi_{\text{true}} + v_m) = \delta\psi_g - v_m \quad \textbf{(Eq.3)}$$

To put above in the form of $z_k = H_k x_k + v_k$, we would have:

$$H_k = \begin{bmatrix}1 & 0 & 0\end{bmatrix}$$

Since the noise v_m is a scalar, the covariance matrix $\mathbf{R}$ of v_m is just the variance of magnetometer measurement error[3]. That is:

$$\mathbf{R} = \sigma_{\mathbf{m}}^{\mathbf{2}}$$

Then we are ready to calculate the Kalman gain, which intuitively is how much we trust the observation. **(Eq.4)**

(2.3 Update)

Finally, we update our prediction matrix and state vector. The first entry of the state vector $x_{k+1|k+1}$ is our best estimate of the current gyro error : $\delta\dot{\psi}_g$. We accumulate that discrete error to our culminate gyro error $\delta\psi_g$. By outputting $\psi_{\text{estimate}} = \psi_g - \delta\psi_g$, we should be making a reliable estimate.

For a high-level flow chart see section 3. More detailed explanations of steps can be found in Neto and Mende's paper[4] as well as Bzarg's blog post[5].

[2]The value of S can be found by experiments.

[3]The value of σ_m can be found by experiments.

[4]Kalman filter-based yaw angle estimation by fusing inertial and magnetic sensing: a case study using low cost sensors. Emerald Group Publishing Limited. 2014.

[5]How a Kalman filter works, in pictures. http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/#mjx-eqn-kalupdatefull

3: Implementation

Notation:

- A variable $\dot{y}$ is a discrete increment for the output variable y. That is $y = \sum_{\text{time}} \dot{y}$. A variable δy is the difference between y and the true parameter it describes. That is, δy describes the error. A variable Δy is the difference between current discrete measurement and the previous one. A variable $\hat{y}$ is the predicted value of y.
- Specifically, the input of our model is a scalar that describes the difference between our predicted yaw and magnetometer yaw : $\hat{\psi} - \psi_m$. When the filter first starts, we synchronize gyroYaw and magYaw by setting the starting point of angular velocity integration as the current reading of magYaw : ψ_{m_0}. Our transformation matrix F and covariance matrix Q, on the other hand, depends on the increment of gyroyaw : $\Delta\dot{\psi}_g$. The first entry of our state vector is the discrete error for our current measurement : $\delta\dot{\psi}_g$. And the first entry of our error vector is the total error for our measurement : $\delta\psi_g$. We obtain our predicted yaw by subtracting the gyro yaw (which has drift) with our predicted offset, that is $\hat{\psi} = \psi_g - \delta\psi$. It is important to understand the seemingly similar variables to correctly implement the yaw filter.
- Each entry of the state vector has a power spectral density (PSD) associated with it, which is used to calculate the covariance matrix Q. Another adjustable parameter is σ_m^2, which is used to calculate the Kalman Gain, which is, loosely speaking, how much we trust the observation. Generally, the less noisy our observation is, the more reliable it is.

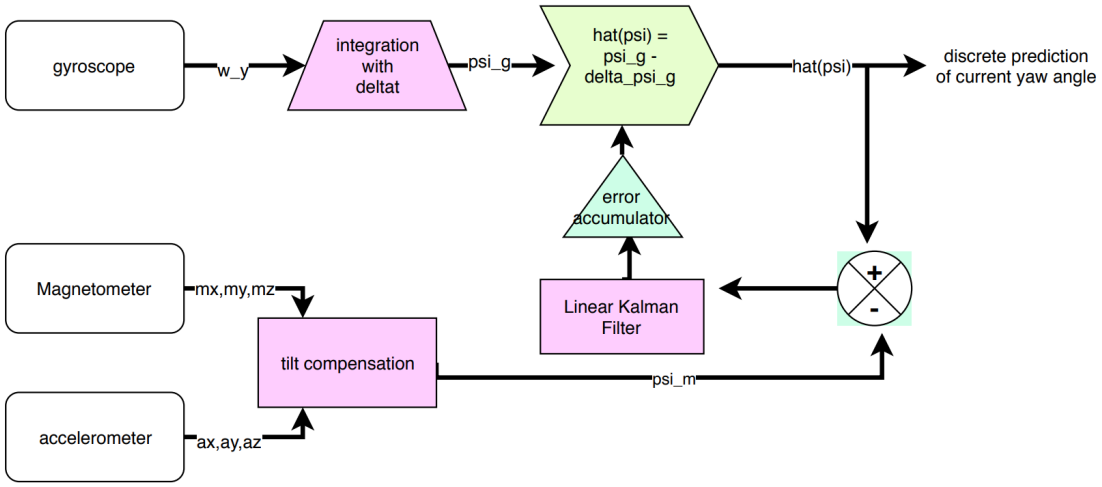
$$\text{Initialization} \begin{cases} \psi_g = \psi_{m_0} \quad \mathbf{x_{0|0}} = \begin{bmatrix} \delta\dot{\psi}_g \\ \delta\dot{k} \\ \dot{b} \end{bmatrix} = \mathbf{0} \quad \delta\mathbf{x_0} = \begin{bmatrix} \delta\psi_g \\ \delta k \\ b \end{bmatrix} = \mathbf{0} \\ \mathbf{P_{0|0}} = \mathbf{I_{3\times 3}} \quad \mathbf{F} = \begin{bmatrix} 1 & \Delta\psi_g & \Delta t \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ \mathbf{Q_k} = \mathbf{E}[\mathbf{w_k}\mathbf{w_k}^T] \quad \text{(Given by Neto and Mende's paper)} \\ \mathbf{H} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \quad \mathbf{R} = \sigma_\mathbf{m}^\mathbf{2} \end{cases}$$

$$\text{Prediction} \begin{cases} \mathbf{x_{k+1|k}} = \mathbf{F}\ \mathbf{x_{k|k}} - \delta\mathbf{x_k} \quad \mathbf{(Eq.1)} \\ \mathbf{P_{k+1|k}} = \mathbf{F}\ \mathbf{P_{k|k}}\ \mathbf{F}^T + \mathbf{Q_k} \quad \mathbf{(Eq.2)} \end{cases}$$

$$\text{Observation} \begin{cases} \mathbf{z_{k+1}} = \hat{\psi} - \psi_\mathbf{m} = \psi_\mathbf{g} - \delta\psi_\mathbf{g} - \psi_\mathbf{m} \quad \mathbf{(Eq.3)} \\ \mathbf{K_{k+1}} = \mathbf{P_{k+1|k}}\ \mathbf{H_k^T}\ (\mathbf{H_k P_{k+1|k} H_k^T} + \mathbf{R})^{-1} = (\mathbf{P_{k+1|k}}\ \mathbf{H_k^T})/(\mathbf{H_k P_{k+1|k} H_k^T} + \sigma_\mathbf{m}^\mathbf{2}) \quad \mathbf{(Eq.4)} \end{cases}$$

$$\text{Update} \begin{cases} \mathbf{x_{k+1|k+1}} = \mathbf{x_{k+1|k}} + \mathbf{K_{k+1}}(\mathbf{z_{k+1}} - \mathbf{H x_{k+1|k}}) \quad \mathbf{(Eq.5)} \\ \mathbf{P_{k+1|k+1}} = \mathbf{P_{k+1|k}} - \mathbf{K_{k+1}}\ \mathbf{H_k P_{k+1|k}} \quad \mathbf{(Eq.6)} \\ \delta\mathbf{x_{k+1}} = \delta\mathbf{x_k} + \mathbf{x_{k+1|k+1}} \\ \hat{\psi} = \psi_g - \delta\psi_g = \psi_g - \delta\mathbf{x_{k+1}}[\mathbf{0}] \end{cases}$$

Visually, here's the program flow for one iteration of Kalman filter:



3: Arduino Code

Disclaimer: Adapted from Simon Devy's open-sourced TinyEKF code. The codes are changed to work with linear model rather than Jacobian matrices.

(3.1) data structure

```
/*
* data structure for kalman filter
* include this file after using #define for Nsta and Mobs
*/

typedef struct {
    int n;
    int m;

    double x[Nsta];
    // state vector, size Nsta * 1
    double F[Nsta][Nsta];
    // state transition matrix, size Nsta * Nsta
    double Ft[Nsta][Nsta];
    // transpose of F, size Nsta * Nsta
    double dx[Nsta];
     // error accumulator, size Nsta * 1
    double P[Nsta][Nsta];
    // prediction covariance matrix, size Nsta * Nsta
    double Q[Nsta][Nsta];
     // covariance matrix associated with w, size Nsta * Nsta

    double z[Mobs];
    // observation vector, size Mobs * 1
```

```
    double v[Mobs];
    // obervation noise vector, size Mobs * 1
    double H[Mobs][Nsta];
    // observation transformation matrix, size Mobs * Nsta
    double Ht[Nsta][Mobs];
    // transpose of H, size Nsta * Mobs
    double R[Mobs][Mobs];
    // observation covarianc ematrix, size Mobs * Mobs
    double K[Nsta][Mobs];
    // Kalman gain, size Nsta * Mobs

    /* temporary storage */
    double tmp0[Nsta];
    // For Eq.1, size Nsta * 1
    double tmp1[Nsta][Nsta];
    // For storing P*Ft, size Nsta * Nsta
    double tmp2[Nsta][Nsta];
    // For storing Eq.2, size Nsta * Nsta
    double tmp3[Nsta][Mobs];
    // For storing P*Ht, size Nsta * Mobs
    double tmp4[Mobs][Mobs];
    // For storing H*P*Ht, size Mobs * Mobs
    double tmp5[Mobs][Mobs];
    // For storing inverse of H*P*Ht, size Mobs * Mobs
    double tmp6[Mobs];
    // For storing H*x, size Mobs * 1
    double tmp7[Mobs][Nsta];
    // For storing H*P, size Mobs * Nsta

} kalman_t;
```

(3.2) step function

```
/* discrete-time step functions; called in the loop */
int kalman_step(void* v, double* z)
{
   int * ptr = (int *)v;
  int n = *ptr;
  ptr++;
  int m = *ptr;

  kalman_t ftr;
  unpack(v, &ftr, n, m);

  //Eq.1
  mulmat(ftr.F, ftr.x, ftr.tmp0, n, n, 1);
  sub(ftr.tmp0, ftr.dx, ftr.x, n);
```

```
  //Eq.2
  transpose(ftr.F, ftr.Ft, n, n);
  mulmat(ftr.P, ftr.Ft, ftr.tmp1, n, n, n);
  mulmat(ftr.F, ftr.tmp1, ftr.tmp2, n, n, n);
  add(ftr.tmp2, ftr.Q, ftr.P, n*n);

  //Eq.3 Observation vector z is updated by the argument

  //Eq.4
  transpose(ftr.H, ftr.Ht, m, n);
  mulmat(ftr.P, ftr.Ht, ftr.tmp3, n, n, m);
  mulmat(ftr.H, ftr.tmp3, ftr.tmp4, m, n, m);
  add(ftr.tmp4,ftr.R,ftr.tmp5,m*m);
  if (inverse(ftr.tmp5)) return 1;
  mulmat(ftr.Ht, ftr.tmp5, ftr.tmp3, n, m, m);
  mulmat(ftr.P,ftr.tmp3,ftr.K, n, n, m); //Kalman gain

  //Eq.5
  mulmat(ftr.H, ftr.x, ftr.tmp6, m, n, 1);
  sub(z, ftr.tmp6, ftr.tmp6, m);
  mulmat(ftr.K, ftr.tmp6, ftr.tmp0, n, m, 1);
  accum(ftr.x, ftr.tmp0, n, 1);

  //Eq.6
  mulmat(ftr.H, ftr.P, ftr.tmp7, m, n, n);
  mulmat(ftr.K, ftr.tmp7, ftr.tmp1, n, m, n);
  sub(ftr.P,ftr.tmp1,ftr.P,n*n);

  // Eq. 7
  accum(ftr.dx,ftr.x, n, 1); // error updater
  return 0; //success
}
```

(3.3) Interface

```
/**
 * A header-only class for the Kalman Filter. User's implementing
    class should #define Nsta and Mob
 *  User will also need to implement a model() method for their
    application.
 */
class Kalman {

    private:

        kalman_t ftr; // struct that encapsulates all the necessary
           information, hidden from the user for safety purpose
```

```
protected:

    // The current state.
     double * x;

    // Initializes a filter object
     Kalman() {
         kalman_init(&this->ftr, Nsta, Mobs);
         this->x = this->ftr.x;
     }

    // Deallocates memory for a TinyEKF object.
     ~Kalman() { }

     // Implement this function for your linear Kalman model.
     virtual void model(double F[Nsta][Nsta], double H[Mobs][Nsta
        ], double w[Nsta], double v[Mobs]) = 0;

     // Change the value of the entry at the ith row, jth column
        of prediction error covariance
     void setP(int i, int j, double value)
     {
         this->ftr.P[i][j] = value;
     }

     void setR(int i, int j, double value)
     {
         this->ftr.R[i][j] = value;
     }

     void setF(int i, int j, double value)
     {
         this->ftr.F[i][j] = value;
     }

     void setQ(int i, int j, double value)
     {
         this->ftr.Q[i][j] = value;
     }

public:

      // Returns the state element at a given index.
     double getX(int i)
     {
         return this->ftr.x[i];
     }

     // Sets the state element at a given index.
```

```
        void setX(int i, double value)
        {
            this->ftr.x[i] = value;
        }

        // Performs one step of the prediction and update.
        bool step(double * z)
        {
            this->model(this->ftr.F, this->ftr.H, this->ftr.v);

            return kalman_step(&this->ftr, z) ? false : true;
            // return 1 indicates failed inversion
        }
};
```

Note: Generally, the transformation matrix F, the covariance matrices Q and R are invariant after the model has been established. Thus the codes would work with different linear model. However, the model we currently adapt for yaw estimation, which is based on the work of Neto, Mendes, et. al., uncommonly updates F, Q and R for each step. Thus, the actual implementation, as a special case, has to breach the above interface. For complete codes, please see BitBucket.

4: More Implementation Notes:

(4.1 Tilt Compensation)

In order to get the observation feedback, we need to calculate yaw angle from the input of accelerometer and magnetometer. This is done through tilt compensation. Here is the code snippet for the algorithm. This algorithm must first calculate roll and pitch angle before it calculates the yaw angle, because the former two have gravity as a reference. Here's a code snippet for the algorithm based on the technical documentation of Freescale[6] and ST electronics[7].

```
// use tilt compensation to calculate rpy
// all angles are in radians
// range : -pi/2 ~ pi/2
float tilt_roll = atan2(myIMU.ay,myIMU.az);
float az2 = myIMU.ay * sin(tilt_roll) + myIMU.az * cos(tilt_roll);
float tilt_pitch = atan2(-myIMU.ax,az2);
float my2 = myIMU.mz * sin(tilt_roll) - myIMU.my * cos(tilt_roll);
float mz2 = myIMU.my * sin(tilt_roll) + myIMU.mz * cos(tilt_roll);
float mx3 = myIMU.mx * cos(tilt_pitch) + mz2 * sin(tilt_pitch);
float tilt_yaw = (atan2(my2,mx3));
myIMU.tilt_roll= tilt_roll;
myIMU.tilt_pitch = tilt_pitch;
myIMU.tilt_yaw = tilt_yaw;
```

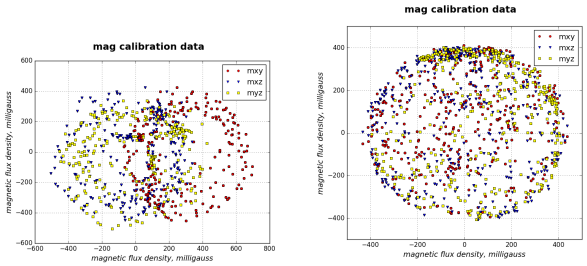
(4.2 Calibration)

[6] "Implementing a Tilt-Compensated eCompass using Accelerometer and Magnetometer Sensors"
[7] "Computing tilt measurement and tilt-compensated e-compass"

To ensure the correctness of our tilt-compensated yaw, we need to make sure both the accelerometer and magnetometer are well calibrated. For accelerometer, Kris Winer's MPU9250 Driver has a built-in offset calibration at startup. For magnetometer, however, we need to calibrate ourselves. For simplicity, we follow Winer's note: we collect magnetometer data for a few minutes while waving it in different directions. Then ideally, the data points formed by, for example, mx and my, should be a sphere centred at zero, otherwise there is an offset bias (hard-iron bias). Additionally, we can get three spheres by taking different pairs from xyz, and they should have same radius. Otherwise there is a scale bias (soft-iron bias). Here's a Python code snippet to help find the calibration value, note that we need to write in Arduino codes that the board sends to serial three values separated by tab for each interval:

```
import serial
import re
import numpy as np
from matplotlib import pyplot as plt
from time import time


duration = 50 # seconds to collect data
#open arduino serial communication
ser = serial.Serial('/dev/cu.usbmodem1988811', 9600)
start_time = time()
#initialize data holders
mxs = []
mys = []
mzs = []
mRes = 1.5
magmax = [-32767, -32767, -32767]
magmin = [ 32767, 32767, 32767]


while (time()-start_time < duration):
    # read serial input and parse into float
    str = ser.readline()
    stringList = re.split(r'\t+', str.rstrip('\n'))
    floatList = map(float,stringList)
    if (len(floatList)>=3):
        mx = (floatList[0])
        my = (floatList[1])
        mz = (floatList[2])
        print(mx,my,mz)
        for i in range(3):
            cur = floatList[i]
            if cur < magmin[i]: magmin[i] = cur
            if cur > magmax[i]: magmax[i] = cur
    #append to data holders for plotting
    mxs.append(mx)
    mys.append(my)
    mzs.append(mz)
    #print(ser.readline())
```

```
print("Done collecting data!")
print("magmin: ",magmin)
print("magmax: ",magmax)
#calculate calibration parameter
for i in range(3):
    magbias[i] = (magmax[i]+magmin[i])/2
    magscale[i] = (magmax[i]-magmin[i])/2
    print("magbias",i,magbias[i])

avgRad = (magscale[0]+magscale[1]+magscale[2])/3
for i in range(3):
    magscale[i] = avgRad/magscale[i]
    print("scale_correction",i,magscale[i])


#plot
fig1 = plt.figure()
fig1.suptitle('mag calibration data', fontsize='18', fontweight='
   bold')
plt.xlabel('magnetic flux density, milligauss', fontsize='14',
   fontstyle='italic')
plt.ylabel('magnetic flux density, milligauss', fontsize='14',
   fontstyle='italic')
plt.axes().grid(True)
line1, = plt.plot(mxs,mys,marker='o',markersize=4,linestyle='none',
   markerfacecolor='red') #xy
line2, = plt.plot(mxs,mzs,marker='v',markersize=4,linestyle='none',
   markerfacecolor='blue') #xz
line3, = plt.plot(mys,mzs,marker='s',markersize=4,linestyle='none',
   markerfacecolor='yellow') #yz
plt.legend([line1,line2,line3], ["mxy","mxz","myz"])
fig1.show()

ser.close()
```



Above is a comparison of data points before and after calibration.

(4.3 Debugging and Prototyping)

Since Arduino do not report run-time error, you might want to verify the correctness of your code before running it on Arduino. We can use a cpp program to simulate Arduino, make the Kalman

Filter library we wrote into a c library and use gcc to compile. If you find compiling c and c++ together tricky, here's a makefile that might help you. Note that use a cpp program that relies on user's C library to simulate the arduino, so every time we change the library, we need to remake the whole thing by typing "makinglib1 makinglib2 cppmain ./test":

```
CCBIN=gcc
#CCBIN=/usr/bin/gcc
CC=$(CCBIN) -Wall -Wextra -Wshadow -std=c99 -pedantic -g
CCD=$(CCBIN) -Wall -Wextra -Wshadow -std=c99 -pedantic -g -DDEBUG
C_LIBS=lib/xalloc.c

default: ftr ftr-d

makinglib1: cFuncs.h
    gcc -c -Wall  -fpic kalman.c

makinglib2:
    gcc -shared -o libkalman.so kalman.o

cppmain:
    g++ -L/Path_To_TheDirectoryOf_Your_CLib_Code -Wl,-rpath=/afs/
       andrew.cmu.edu/usr17/yaoj/private/Kalman -Wall -o test voof-
       simulate.cpp -lkalman

.PHONY: clean
clean:
    rm -Rf lib/*.o ftr ftr-d *.dSYM
```

It's also nice to test if the model works with some high level language before diving into the C codes: here's a Python program that help you verify your model:

```
TO BE CONTINUED
```

We can improve the estimation by using smoother input. If the reading is too noisy, we can consider using low pass filter. Here's a simple code snippet for it:

```
float alpha = 0.9; //adjustable parameter
 //infinite impulse response filter
myIMU.smooth_ax = (1- alpha) * myIMU.smooth_ax + alpha *myIMU.ax;
```

(4.4 Dealing with the discontinuity)

We used the discontinuous arctan function in tilt compensation. Hence, to avoid weird situation when the discontinuity happen, we keep the $\delta\psi$ in range $(\pi/2, -\pi/2)$:

```
if (delta_psi > 180) {delta_psi = delta_psi - 360;}
if (delta_psi < -180) {delta_psi = delta_psi + 360;}
```