ESE650 - Learning in Robotics
Homework 2

Vaibhav Thakkar
vaithak@seas.upenn.edu

Q1 (a) ε_k was sampled using np.random.normal(0,1) & similarly v_k using np.random.normal(0, $\sqrt{0.5}$)

→ A for loop was initialised from 1 to 100:
→ at each time step, sampling ε_k & v_k independently
→ updating x_k at each step using $x_k = -x_{k-1} + \varepsilon_k$

→ adding an observation $y_k = \sqrt{x_{k+1}^2} + v_k$ to the dataset.

(b) updating 'a' into the state itself

$$\Rightarrow \begin{bmatrix} x_{k+1} \\ a_{k+1} \end{bmatrix} = \begin{bmatrix} x_k a_k \\ a_k \end{bmatrix} + \begin{bmatrix} \varepsilon_k \\ 0 \end{bmatrix}$$

ε_k → noise vector

$\Rightarrow f(x)$ → state vector

$$y_k = \underbrace{\sqrt{x_k^2 + 1}}_{g(x)} + v_k$$

→ Linearizing both;
① for motion model; we get

$$A = \text{Jac}_{(f)} = \begin{bmatrix} a_k & x_k \\ 0 & 1 \end{bmatrix}$$

② $C = \text{Jac}_{(g)} = \begin{bmatrix} x_k / \sqrt{x_k^2 + 1} & 0 \end{bmatrix}$

→ for noise covariances

$$R = Cov\left(\begin{bmatrix} \varepsilon_k \\ 0 \end{bmatrix}\right) = \begin{bmatrix} \overset{\text{one}}{1} & 0 \\ 0 & 0 \end{bmatrix}$$

$$Q = Cov(\nu_k) = 0.5$$

$$\Rightarrow \quad \mu_{0|0} = \begin{bmatrix} 1 \\ \mu_{a_0} \end{bmatrix} \; ; \; \Sigma_{0|0} = \begin{bmatrix} 2 & 0 \\ 0 & \sigma^2_{a_0} \end{bmatrix}$$

~~This is assuming $a_0 \sim N(\mu_{a_0}, \sigma^2_{a_0})$~~

where $a_0 \sim N(\mu_{a_0}, \sigma^2_{a_0})$

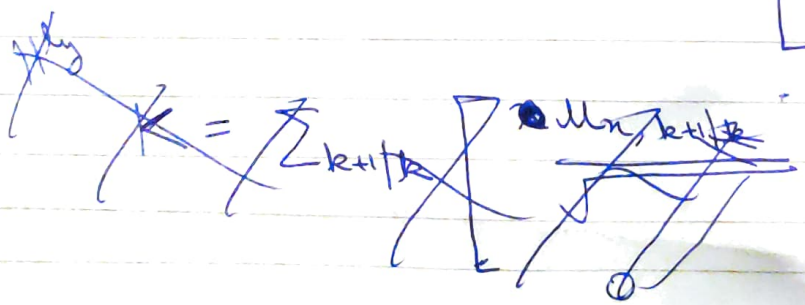
Equations for update

$$\mu_{k+1|k} = \begin{bmatrix} \mu_{a,k|k} * \mu_{x,k|k} \\ \mu_{a,k|k} \end{bmatrix}$$

$$\Sigma_{k+1|k} = A \Sigma_{k|k} A^T + R$$

$$= \begin{bmatrix} \mu_{a,k|k} & \mu_{x,k|k} \\ 0 & 1 \end{bmatrix} \Sigma_{k|k} \begin{bmatrix} \mu_{a,k|k} & 0 \\ \mu_{x,k|k} & 1 \end{bmatrix}$$

$$+ \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$



11th

$$C^T = \begin{bmatrix} \mu_{x,k+1|k} / \sqrt{\mu_{x,k+1|k}^2 + 1} \\ 0 \end{bmatrix}$$

$$K = \Sigma_{k+1|k} C^T \underbrace{(C \Sigma_{k+1|k} C^T + Q)^{-1}}_{\text{scalar term.}}$$

$$\mu_{k+1|k+1} = \mu_{k+1|k} + K (y_{k+1} - g(\mu_{k+1|k}))$$

$$\Sigma_{k+1|k+1} = (I - KC) \Sigma_{k+1|k}$$

Main implementation detail

$$a_0 \sim N(\mu_{a_0}, \sigma^2_{a_0})$$

→ the convergence of a_0 heavily depended on the value of μ_{a_0}

→ if μ_{a_0} was chosen as '0' wrong/ no state convergence happened because $[\mu_x, \mu_a]$ remained $[0, 0]$ for all iterations

→ for other μ_{a_0}, sometimes 'a' converged to (+1) & sometimes (-1)

→ Discussed more in (c)

c// for $\mu_{a_0} \neq 0$;

sometimes the system converged to (+1) & sometimes (-1) with more bias towards where it started from

→ i.e if $\mu_{a_0} > 0$
→ it converged to +1

else
→ it converged to -1

* Adding more data didn't help to flip this as this makes sense because what we are measuring is related $\boxed{x_k^2}$
this means the knowledge of sign (x_k) is lost in the observatn

→ the same data could be also generated for $a=1$ & $a=-1$.

Convergence of estimate of 'a' using EKF

Vaibhav Thakkar

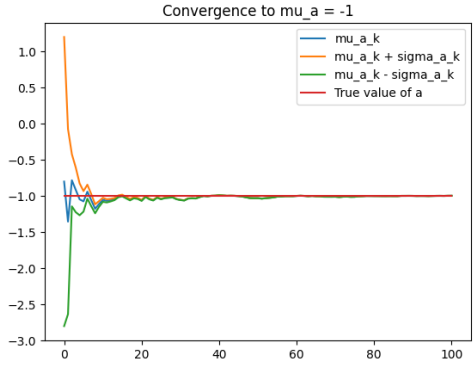


Figure 1: Convergence when the initial mean of a is negative

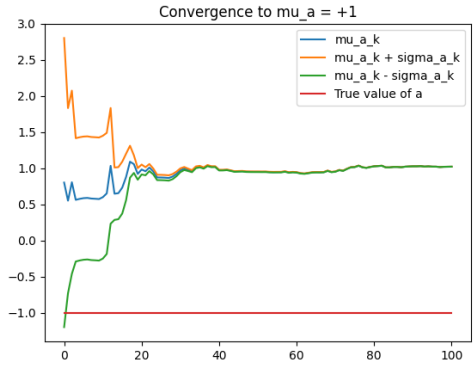


Figure 2: Convergence when the initial mean of a is positive

Vaibhav Thakkar
vaithak@seas.upenn.edu

Q2

(a) Calibrating the sensors

(i) Accelerometer

- I used the fact the only force acting on the system/drone is gravity ⇒ the IMU's accelerometer should have values corr. to gravity vector in body frame

$$\Rightarrow \begin{bmatrix} acc^n\ x \\ acc^n\ y \\ acc^n\ z \end{bmatrix} = R^T \begin{bmatrix} 0 \\ 0 \\ 9.8 \end{bmatrix}$$

where R was obtained using vicon-data

→ And rearranging the terms I got

$$\frac{1023}{3300}\alpha . value + \beta = raw$$

→ we have values from $R^T g$
→ raw reading from accelerometer

∴ This is a least squares problem of finding

$$\alpha^*, \beta^* = \arg\min_{\alpha,\beta} \| \alpha X + \beta - raw \|_2^2$$

→ after properly aligning the axis, I got the following values

$\beta_{acc^n} = [510.3, 500.2, 502.5]$
$\alpha_{acc^n} = [34.23, 33.9, 33.76]$

(ii) Gyroscope - for gyroscope, the data being noisy and taking derivatives of noisy signal made it more challenging.

(*) I observed by plotting the roll, pitch, yaw angles obtained from the rotation matrix of vicon data - the drone was at rest in the initial segment - all r, P, y angles = 0

(*) This means (raw - β) x 0 for this durat^n

⇒ β can be obtained by mean of raw values in this durat^n

⇒ I obtained $\beta_{gyro} = [373.65, 375.15, 369.75]$

(*) For α_{gyro}, I found the segments in graphs of euler angles where the slope of roll/pitch/yaw looked almost constant and fit a line using linear regression.

- Then the mean of 'α' values along this line were taken.

- I obtained the following values of α

$$\alpha_{gyro} = [201.56, 204.86, 203.21]$$

Plots used for calibration

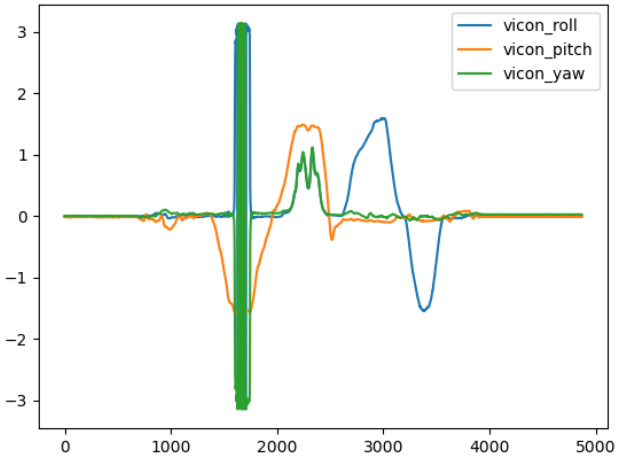
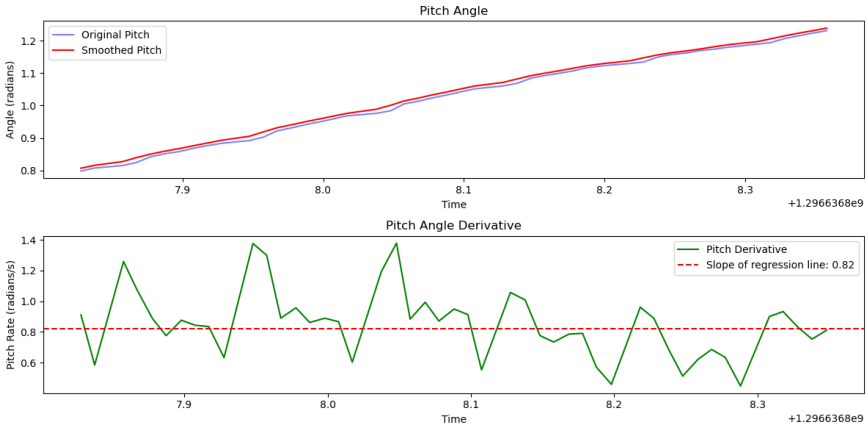
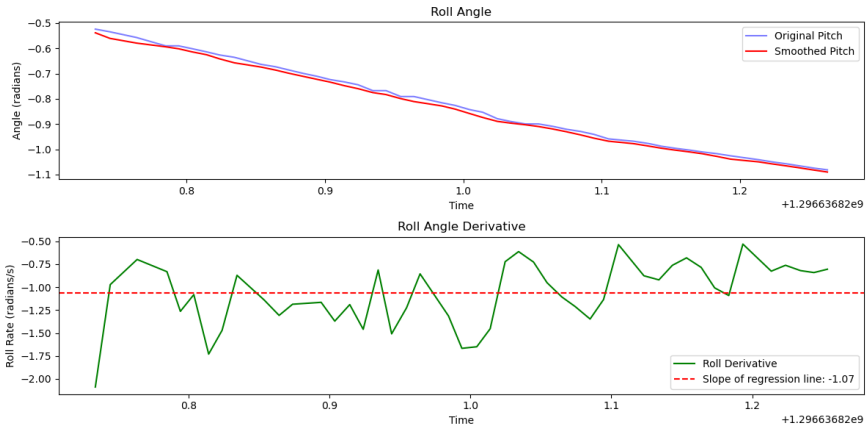


Figure 1: Euler angles plot



(a) Constant Slope Pitch Plot

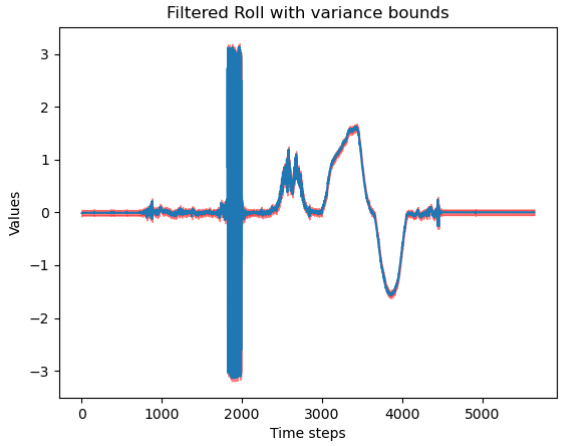


(b) Constant slope roll angle

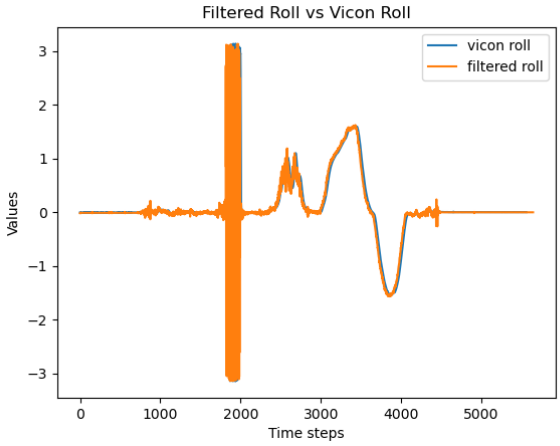
Figure 2: Regions of constant slope pitch and roll angles

Analysis and Debugging of UKF

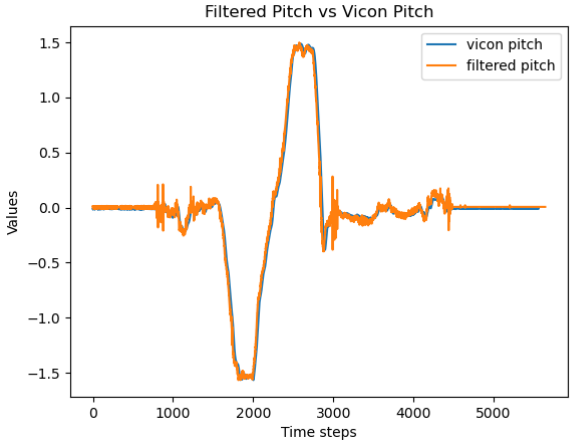
I have attached the plots of filtered Euler angles and angular velocities - mean + variance bands. Also attached are the filtered data comparisons with those obtained from Vicon data and gyro readings.



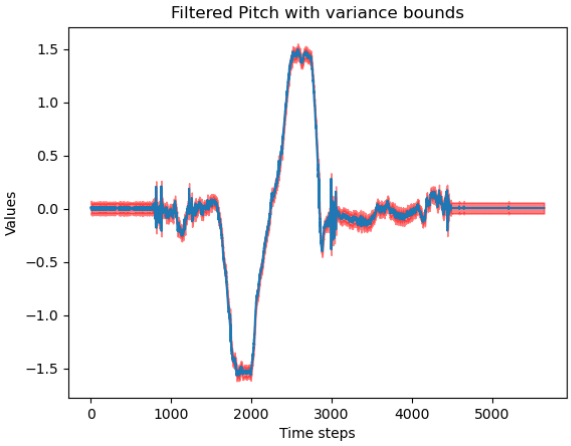
(a) Filtered Roll with Variance Bounds



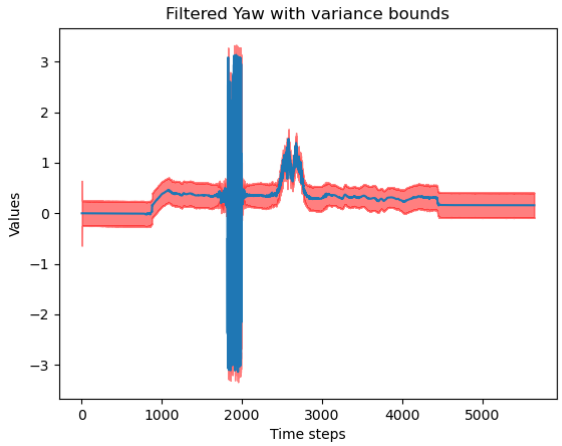
(b) Filtered Roll and comparison with Vicon



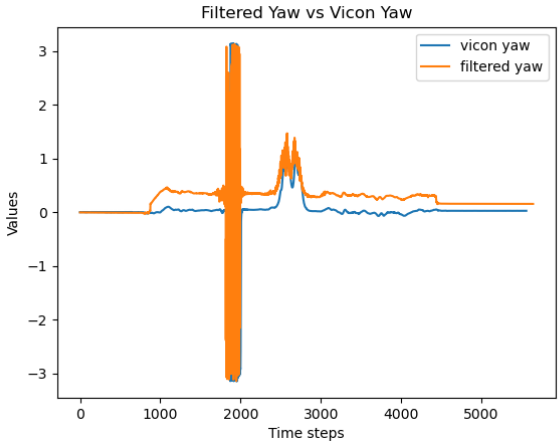
(c) Filtered Pitch and comparison with Vicon



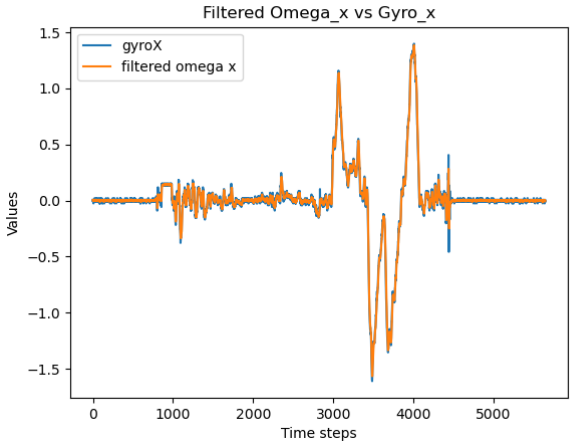
(d) Filtered Pitch with Variance Bounds



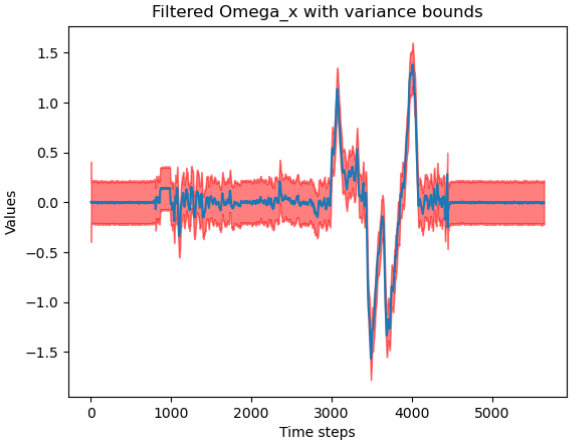
(e) Filtered Yaw with Variance Bounds



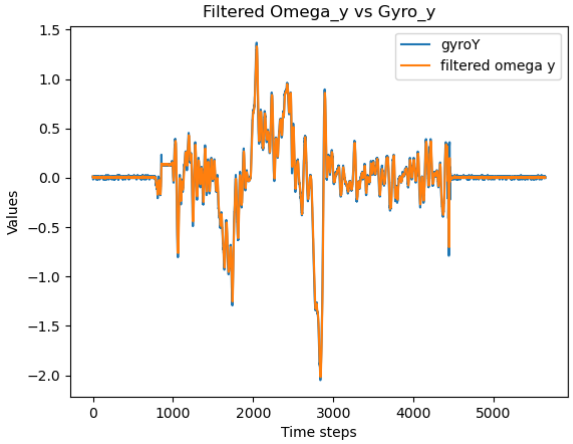
(f) Filtered Yaw and comparison with Vicon



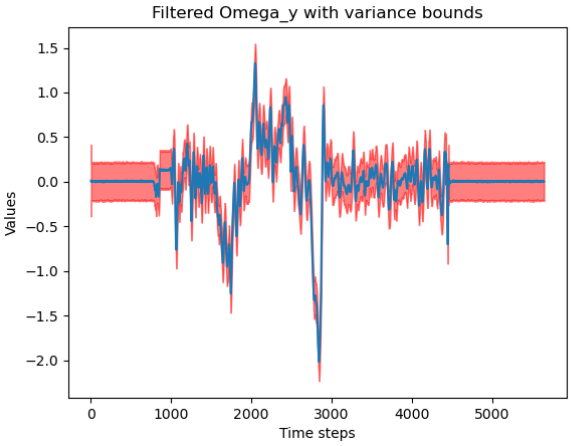
(g) Filtered OmegaX and it's comparison with gyro readings



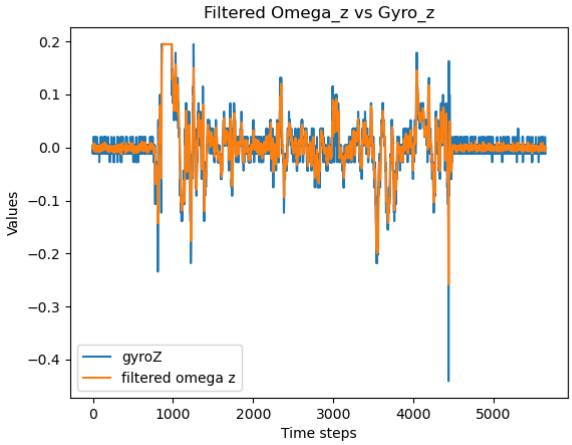
(h) Filtered OmegaX with Variance bounds



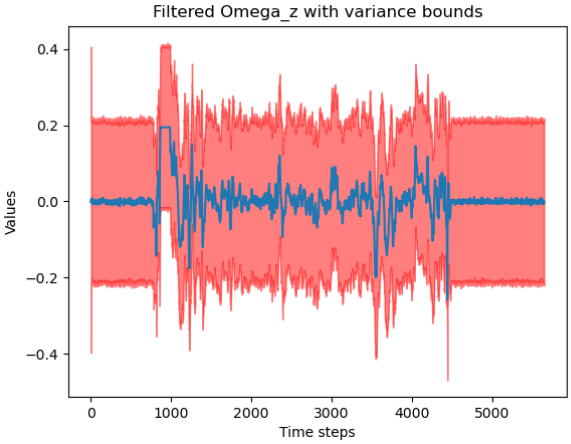
(i) Filtered OmegaY and it's comparison with gyro readings



(j) Filtered OmegaY with Variance bounds



(k) Filtered OmegaZ and it's comparison with gyro readings



(l) Filtered OmegaZ with Variance bounds

```python
import numpy as np
import math

class Quaternion:
    def __init__(self, scalar=1, vec=[0,0,0]):
        self.q = np.array([scalar, 0., 0., 0.])
        self.q[1:4] = vec

    def normalize(self):
        self.q = self.q/np.linalg.norm(self.q)

    def scalar(self):
        return self.q[0]

    def vec(self):
        return self.q[1:4]

    def axis_angle(self):
        theta = 2*math.acos(np.clip(self.q[0], -1.0, 1.0))
        vec = self.vec()
        if (np.linalg.norm(vec) == 0):
            return np.zeros(3)
        vec = vec/np.linalg.norm(vec)
        return vec*theta

    def euler_angles(self):
        phi = math.atan2(2*(self.q[0]*self.q[1]+self.q[2]*self.q[3]), \
                1 - 2*(self.q[1]**2 + self.q[2]**2))
        theta = math.asin(2*(self.q[0]*self.q[2] - self.q[3]*self.q[1]))
        psi = math.atan2(2*(self.q[0]*self.q[3]+self.q[1]*self.q[2]), \
                1 - 2*(self.q[2]**2 + self.q[3]**2))
        return np.array([phi, theta, psi])

    def from_axis_angle(self, a):
        angle = np.linalg.norm(a)
        if angle != 0:
            axis = a/angle
        else:
            axis = np.array([1,0,0])
        self.q[0] = math.cos(angle/2)
        self.q[1:4] = axis*math.sin(angle/2)
        #self.normalize()

    def from_rotm(self, R):
        theta = math.acos(np.clip((np.trace(R)-1)/2, -1.0, 1.0))
        if (np.abs(theta) < 1e-2):
            omega = np.array([0.0, 0.0, 0.0])
        else:
            omega_hat = (R - np.transpose(R))/(2*math.sin(theta))
            omega = np.array([omega_hat[2,1], -omega_hat[2,0], omega_hat[1,0]])
        self.q[0] = math.cos(theta/2)
        self.q[1:4] = omega*math.sin(theta/2)
        self.normalize()

    def inv(self):
        q_inv = Quaternion(self.scalar(), -self.vec())
        q_inv.normalize()
        return q_inv

```

```
    #Implement quaternion multiplication
    def __mul__(self, other):
        t0 = self.q[0]*other.q[0] - \
             self.q[1]*other.q[1] - \
             self.q[2]*other.q[2] - \
             self.q[3]*other.q[3]
        t1 = self.q[0]*other.q[1] + \
             self.q[1]*other.q[0] + \
             self.q[2]*other.q[3] - \
             self.q[3]*other.q[2]
        t2 = self.q[0]*other.q[2] - \
             self.q[1]*other.q[3] + \
             self.q[2]*other.q[0] + \
             self.q[3]*other.q[1]
        t3 = self.q[0]*other.q[3] + \
             self.q[1]*other.q[2] - \
             self.q[2]*other.q[1] + \
             self.q[3]*other.q[0]
        retval = Quaternion(t0, [t1, t2, t3])
        return retval

    def __str__(self):
        return str(self.scalar()) + ', ' + str(self.vec())


import math
import numpy as np

# from quaternion import Quaternion
from matplotlib import pyplot as plt
from scipy import io
from scipy.stats import linregress


def time_sync(imu_ts, vicon_ts, vicon_rots, accel, gyro):
    # Synchronize time stamps between vicon and IMU
    # For each timestamp in IMU, find the closest timestamp in Vicon
    vicon_idx = 0
    vicon_rots_synced = []
    accel_synced = []
    gyro_synced = []
    time_diff_threshold = 0.01
    sync_times = []
    for i in range(len(imu_ts)):
        while vicon_idx < len(vicon_ts) and vicon_ts[vicon_idx] < imu_ts[i]:
            vicon_idx += 1
        if vicon_idx < len(vicon_ts) and abs(vicon_ts[vicon_idx] - imu_ts[i]) < time_diff_threshold:
            vicon_rots_synced.append(vicon_rots[vicon_idx])
            accel_synced.append(accel[:,i])
            gyro_synced.append(gyro[:,i])
            sync_times.append(imu_ts[i])
    print(f"Number of synced time stamps: {len(sync_times)}")
    return np.array(vicon_rots_synced), np.array(accel_synced).T, np.array(gyro_synced).T, sync_times


def plot_and_log_data(vicon_rots, accel, gyro):
    print('vicon_rots.shape: ', vicon_rots.shape)
    print('accel.shape: ', accel.shape)
    print('gyro.shape: ', gyro.shape)
```

```
119
120     vicon_roll = np.zeros(vicon_rots.shape[0])
121     vicon_pitch = np.zeros(vicon_rots.shape[0])
122     vicon_yaw = np.zeros(vicon_rots.shape[0])
123     for i in range(len(vicon_rots)):
124         vicon_rot = vicon_rots[i]#.reshape((3,3))
125         q_vicon = Quaternion()
126         q_vicon.from_rotm(vicon_rot)
127         roll, pitch, yaw = q_vicon.euler_angles()
128         vicon_roll[i] = roll
129         vicon_pitch[i] = pitch
130         vicon_yaw[i] = yaw
131
132     # Plot vicon data
133     plt.plot(vicon_roll, label='vicon_roll')
134     plt.plot(vicon_pitch, label='vicon_pitch')
135     plt.plot(vicon_yaw, label='vicon_yaw')
136     plt.legend()
137
138
139 def calibrate_accel(vicon_rots, accel):
140     max_T = len(vicon_rots)
141     imu_ax_X = np.zeros((max_T, 2))
142     imu_ax_b = np.zeros(max_T)
143
144     imu_ay_X = np.zeros((max_T, 2))
145     imu_ay_b = np.zeros(max_T)
146
147     imu_az_X = np.zeros((max_T, 2))
148     imu_az_b = np.zeros(max_T)
149     for i in range(max_T):
150         # In world frame - accn = [0, 0, 9.81]
151         # Convert to body frame
152         accn_body_frame = np.dot(vicon_rots[i].reshape((3,3)).T, np.array([0,
    0, 9.81]))
153
154         # Convert body frame to IMU frame -> ax, and ay are reverse in IMU
    frame
155         imu_ax = -accn_body_frame[0]
156         imu_ay = -accn_body_frame[1]
157         imu_az = accn_body_frame[2]
158
159         # This will act as a value for IMU's accelerometer
160         imu_ax_X[i] = [(1023 * imu_ax) / (3300), 1]
161         imu_ax_b[i] = accel[0][i]
162
163         imu_ay_X[i] = [(1023 * imu_ay) / (3300), 1]
164         imu_ay_b[i] = accel[1][i]
165
166         imu_az_X[i] = [(1023 * imu_az) / (3300), 1]
167         imu_az_b[i] = accel[2][i]
168
169     # Least squares method
170     fit_ax = np.linalg.lstsq(imu_ax_X, imu_ax_b, rcond=None)
171     fit_ay = np.linalg.lstsq(imu_ay_X, imu_ay_b, rcond=None)
172     fit_az = np.linalg.lstsq(imu_az_X, imu_az_b, rcond=None)
173
174     accel_bias = np.array([fit_ax[0][1], fit_ay[0][1], fit_az[0][1]])
175     accel_sensitivity = np.array([fit_ax[0][0], fit_ay[0][0], fit_az[0][0]])
176     print(f"accel_bias: {np.round(accel_bias, 2)}")
177     print(f"accel_sensitivity: {np.round(accel_sensitivity, 2)}")
```

```
178
179
180 def compute_derivative(times, angles, window_length=3, name='pitch'):
181     """
182     Compute the slope of a time series of angles using a running mean filter
183     and linear regression.
184
185     Parameters:
186     times (array-like): Timestamps for each measurement
187     angles (array-like): Angles to compute slope for
188     window_length (int): Number of points to use in running mean
189
190     Returns:
191     computed_slope (float): Slope of the angle data
192     """
193     # Convert inputs to numpy arrays
194     times = np.array(times)
195     angles = np.array(angles)
196
197     # Ensure window_length is odd
198     if window_length % 2 == 0:
199         window_length += 1
200
201     smoothed_angles = np.convolve(angles, np.ones(window_length) /
    window_length, mode='valid')
202     angles = angles[:len(smoothed_angles)]
203     times = times[:len(smoothed_angles)]
204
205     # Compute pitch derivative
206     angles_derivative = np.diff(smoothed_angles) / np.diff(times)
207
208     # Compute slope using linear regression best fit line
209     slope, intercept, r_value, p_value, std_err = linregress(times,
    smoothed_angles)
210     computed_slope = slope
211
212     # Create visualization
213     plt.figure(figsize=(12, 6))
214     plt.subplot(2, 1, 1)
215     plt.plot(times, angles, 'b-', label='Original Pitch', alpha=0.5)
216     plt.plot(times, smoothed_angles, 'r-', label='Smoothed Pitch')
217     plt.legend()
218     plt.title(f"{name.capitalize()} Angle")
219     plt.xlabel('Time')
220     plt.ylabel('Angle (radians)')
221
222     plt.subplot(2, 1, 2)
223     plt.plot(times[:-1], angles_derivative, 'g-', label=f'{name.capitalize()}
    Derivative')
224     plt.axhline(y=computed_slope, color='r', linestyle='--',
225                 label=f'Slope of regression line: {computed_slope:.2f}')
226     plt.legend()
227     plt.title(f"{name.capitalize()} Angle Derivative")
228     plt.xlabel('Time')
229     plt.ylabel(f'{name.capitalize()} Rate (radians/s)')
230
231     plt.tight_layout()
232     return computed_slope
233
234 def compute_gyro_sensor_sensitivity(constant_angular_velocity,
    unbiased_readings):
```

```
    """
    Compute the sensitivity of a gyroscope sensor using a known constant angular velocity.

    Parameters:
    constant_angular_velocity (float): Known constant angular velocity in radians per second
    unbiased_readings (array-like): Unbiased gyroscope readings

    Returns:
    computed_sensitivity (float): Sensitivity of the gyroscope sensor
    """
    # Compute the mean of the unbiased gyroscope readings
    mean_gyro = np.mean(unbiased_readings)

    # Compute the sensitivity of the gyroscope sensor
    computed_sensitivity = (3300 * mean_gyro) / (1023 * constant_angular_velocity)
    return computed_sensitivity


def calibrate_gyro(vicon_euler_angles, gyro, timestamps):
    # For gyro bias, we can use the fact that initially the drone is at rest
    # So, the gyro readings should be zero
    gyro_bias = np.mean(gyro[:, 0:20], axis=1)
    print(f"gyro_bias (roll, pitch, yaw): {gyro_bias}")

    # Compute gyro unbiased readings
    gyro_unbiased = gyro - gyro_bias[:, np.newaxis]

    # For sensitivity, we will choose a time period where the drone is rotating at a constant rate
    constant_pitch_range = [2100, 2150]
    constant_pitch_times = timestamps[constant_pitch_range[0]:constant_pitch_range[1]]
    constant_pitch = vicon_euler_angles[constant_pitch_range[0]:constant_pitch_range[1], 1]
    pitch_rate = compute_derivative(constant_pitch_times, constant_pitch, name='pitch')
    pitch_sensitivity = compute_gyro_sensor_sensitivity(pitch_rate, gyro_unbiased[1, constant_pitch_range[0]:constant_pitch_range[1]])

    # Compute roll sensitivity
    constant_roll_range = [3250, 3300]
    constant_roll_times = timestamps[constant_roll_range[0]:constant_roll_range[1]]
    constant_roll = vicon_euler_angles[constant_roll_range[0]:constant_roll_range[1], 0]
    roll_rate = compute_derivative(constant_roll_times, constant_roll, name='roll')
    roll_sensitivity = compute_gyro_sensor_sensitivity(roll_rate, gyro_unbiased[0, constant_roll_range[0]:constant_roll_range[1]])

    # Compute yaw sensitivity
    # We don't have any good estimate of yaw rate, so we will use the average of roll and pitch sensitivities
    yaw_sensitivity = (roll_sensitivity + pitch_sensitivity) / 2

    print(f"Gyro sensitivities (mv/(rad/s)), (roll, pitch, yaw): {roll_sensitivity:.2f}, {pitch_sensitivity:.2f}, {yaw_sensitivity:.2f}")

```

```
282
283 def calibrate_sensors(data_num=1):
284     #load data
285     imu = io.loadmat('data/imu/imuRaw'+str(data_num)+'.mat')
286     vicon = io.loadmat('data/vicon/viconRot'+str(data_num)+'.mat')
287     vicon_ts = vicon['ts'][0]
288     vicon_rots = vicon['rots'].transpose((2,0,1))
289     accel = imu['vals'][0:3,:]
290     gyro = imu['vals'][3:6,:]
291     T = np.shape(imu['ts'])[1]
292
293     # Synchronize time stamps between vicon and IMU
294     imu_ts = imu['ts'][0]
295     vicon_ts = vicon['ts'][0]
296     vicon_rots, accel, gyro, sync_times = time_sync(imu_ts, vicon_ts,
    vicon_rots, accel, gyro)
297
298     # Plot and log data
299     plot_and_log_data(vicon_rots, accel, gyro)
300
301     # Calibrate accelerometer
302     print()
303     calibrate_accel(vicon_rots, accel)
304
305     # Fix ordering of gyro data 0 its given as Wz, Wx, Wy
306     gyro = np.array([gyro[1], gyro[2], gyro[0]])
307
308     # Vicon euler angles
309     euler_angles = []
310     for i in range(len(vicon_rots)):
311         q = Quaternion()
312         q.from_rotm(vicon_rots[i])
313         roll, pitch, yaw = q.euler_angles()
314         euler_angles.append([roll, pitch, yaw])
315     euler_angles = np.array(euler_angles)
316
317     # Calibrate gyroscope
318     print()
319     calibrate_gyro(euler_angles, gyro, sync_times)
320
321     plt.show()
322
323
324 if __name__ == '__main__':
325     calibrate_sensors(1)
326
327
328
329
330
331
```

(f)

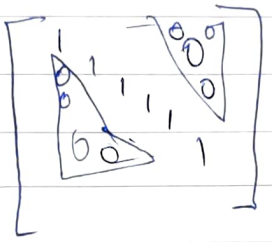
→ Choice of initial state and Covariance & noises

Initial state mean
= [1, 0, 0, 0, 0, 0, 0]

→ This was chosen as the drone was initially at rest ; this also means the initial covariance can be pretty low

Initial Covariance
= np.eye (6) = Identity matrix

=



Noise matrices

Process noise = 10 x np.eye (6)

this was chosen as we use Process noise * dt ; dt makes it pretty low itself

Measurement Noise
= 0.1 x np.eye (6)

⇒ this makes process noise * dt & measurement noise on the same scale.