```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

```

```python
    #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)
```

```python

    vicon_roll = np.zeros(vicon_rots.shape[0])
    vicon_pitch = np.zeros(vicon_rots.shape[0])
    vicon_yaw = np.zeros(vicon_rots.shape[0])
    for i in range(len(vicon_rots)):
        vicon_rot = vicon_rots[i]#.reshape((3,3))
        q_vicon = Quaternion()
        q_vicon.from_rotm(vicon_rot)
        roll, pitch, yaw = q_vicon.euler_angles()
        vicon_roll[i] = roll
        vicon_pitch[i] = pitch
        vicon_yaw[i] = yaw

    # Plot vicon data
    plt.plot(vicon_roll, label='vicon_roll')
    plt.plot(vicon_pitch, label='vicon_pitch')
    plt.plot(vicon_yaw, label='vicon_yaw')
    plt.legend()


def calibrate_accel(vicon_rots, accel):
    max_T = len(vicon_rots)
    imu_ax_X = np.zeros((max_T, 2))
    imu_ax_b = np.zeros(max_T)

    imu_ay_X = np.zeros((max_T, 2))
    imu_ay_b = np.zeros(max_T)

    imu_az_X = np.zeros((max_T, 2))
    imu_az_b = np.zeros(max_T)
    for i in range(max_T):
        # In world frame - accn = [0, 0, 9.81]
        # Convert to body frame
        accn_body_frame = np.dot(vicon_rots[i].reshape((3,3)).T, np.array([0, 0, 9.81]))

        # Convert body frame to IMU frame -> ax, and ay are reverse in IMU frame
        imu_ax = -accn_body_frame[0]
        imu_ay = -accn_body_frame[1]
        imu_az = accn_body_frame[2]

        # This will act as a value for IMU's accelerometer
        imu_ax_X[i] = [(1023 * imu_ax) / (3300), 1]
        imu_ax_b[i] = accel[0][i]

        imu_ay_X[i] = [(1023 * imu_ay) / (3300), 1]
        imu_ay_b[i] = accel[1][i]

        imu_az_X[i] = [(1023 * imu_az) / (3300), 1]
        imu_az_b[i] = accel[2][i]

    # Least squares method
    fit_ax = np.linalg.lstsq(imu_ax_X, imu_ax_b, rcond=None)
    fit_ay = np.linalg.lstsq(imu_ay_X, imu_ay_b, rcond=None)
    fit_az = np.linalg.lstsq(imu_az_X, imu_az_b, rcond=None)

    accel_bias = np.array([fit_ax[0][1], fit_ay[0][1], fit_az[0][1]])
    accel_sensitivity = np.array([fit_ax[0][0], fit_ay[0][0], fit_az[0][0]])
    print(f"accel_bias: {np.round(accel_bias, 2)}")
    print(f"accel_sensitivity: {np.round(accel_sensitivity, 2)}")
```

```python


def compute_derivative(times, angles, window_length=3, name='pitch'):
    """
    Compute the slope of a time series of angles using a running mean filter
    and linear regression.

    Parameters:
    times (array-like): Timestamps for each measurement
    angles (array-like): Angles to compute slope for
    window_length (int): Number of points to use in running mean

    Returns:
    computed_slope (float): Slope of the angle data
    """
    # Convert inputs to numpy arrays
    times = np.array(times)
    angles = np.array(angles)

    # Ensure window_length is odd
    if window_length % 2 == 0:
        window_length += 1

    smoothed_angles = np.convolve(angles, np.ones(window_length) / window_length, mode='valid')
    angles = angles[:len(smoothed_angles)]
    times = times[:len(smoothed_angles)]

    # Compute pitch derivative
    angles_derivative = np.diff(smoothed_angles) / np.diff(times)

    # Compute slope using linear regression best fit line
    slope, intercept, r_value, p_value, std_err = linregress(times, smoothed_angles)
    computed_slope = slope

    # Create visualization
    plt.figure(figsize=(12, 6))
    plt.subplot(2, 1, 1)
    plt.plot(times, angles, 'b-', label='Original Pitch', alpha=0.5)
    plt.plot(times, smoothed_angles, 'r-', label='Smoothed Pitch')
    plt.legend()
    plt.title(f"{name.capitalize()} Angle")
    plt.xlabel('Time')
    plt.ylabel('Angle (radians)')

    plt.subplot(2, 1, 2)
    plt.plot(times[:-1], angles_derivative, 'g-', label=f'{name.capitalize()} Derivative')
    plt.axhline(y=computed_slope, color='r', linestyle='--',
                label=f'Slope of regression line: {computed_slope:.2f}')
    plt.legend()
    plt.title(f"{name.capitalize()} Angle Derivative")
    plt.xlabel('Time')
    plt.ylabel(f'{name.capitalize()} Rate (radians/s)')

    plt.tight_layout()
    return computed_slope

def compute_gyro_sensor_sensitivity(constant_angular_velocity, unbiased_readings):
```

```python
    """
    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}")

```

```python

def calibrate_sensors(data_num=1):
    #load data
    imu = io.loadmat('data/imu/imuRaw'+str(data_num)+'.mat')
    vicon = io.loadmat('data/vicon/viconRot'+str(data_num)+'.mat')
    vicon_ts = vicon['ts'][0]
    vicon_rots = vicon['rots'].transpose((2,0,1))
    accel = imu['vals'][0:3,:]
    gyro = imu['vals'][3:6,:]
    T = np.shape(imu['ts'])[1]

    # Synchronize time stamps between vicon and IMU
    imu_ts = imu['ts'][0]
    vicon_ts = vicon['ts'][0]
    vicon_rots, accel, gyro, sync_times = time_sync(imu_ts, vicon_ts,
vicon_rots, accel, gyro)

    # Plot and log data
    plot_and_log_data(vicon_rots, accel, gyro)

    # Calibrate accelerometer
    print()
    calibrate_accel(vicon_rots, accel)

    # Fix ordering of gyro data 0 its given as Wz, Wx, Wy
    gyro = np.array([gyro[1], gyro[2], gyro[0]])

    # Vicon euler angles
    euler_angles = []
    for i in range(len(vicon_rots)):
        q = Quaternion()
        q.from_rotm(vicon_rots[i])
        roll, pitch, yaw = q.euler_angles()
        euler_angles.append([roll, pitch, yaw])
    euler_angles = np.array(euler_angles)

    # Calibrate gyroscope
    print()
    calibrate_gyro(euler_angles, gyro, sync_times)

    plt.show()


if __name__ == '__main__':
    calibrate_sensors(1)
```