Assignment 5 (INS/GNSS Integration) – Advanced Robot Navigation (RBE595 Spring 2024)

Rohin Siddhartha Palaniappan Venkateswaran

Task 1

In task 1, we have to derive the observation model for use in the non-linear Kalman filter (UKF) in this case, for both the feed-forward (error correction) and feed-back (bias modeling) versions. According to the state vectors provided in the assignment for both the feed-forward and feed-back models, we have derived the C matrices, which have therefore been used in the UKF code. The measurements provided are position (latitude, longitude, altitude) and velocities (VN, VE, VD).

$$\mathbf{C}_{ff} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

$$\mathbf{C}_{fb} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

Task 2

In task 2, according to the assignment, we have to implement the "Nonlinear Error State implementation", which is given by the state vector

$$x = \begin{bmatrix} L & \lambda & h & \phi & \theta & \psi & V_N & V_E & V_D & e_L & e_\lambda & e_h \end{bmatrix}$$

Let us now define the propagation model for this state vector.

Propagation Model

In this model, we first apply an attitude update, a velocity update, and finally a position update, in the same order.

Attitude Update

ω_E, is a constant which is defined as the rotation of the the earth in radians per second - $7.292 \times 10^{-5}\frac{rad}{s}$. We are required to first form a rotation matrix Ω_e^i as:

$$\Omega_e^i = \begin{bmatrix} 0 & -\omega_E & 0 \\ \omega_E & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{1}$$

We then calculate ω_e^n as:

$$\omega_e^n = \begin{bmatrix} \frac{v_e}{R_e(L)+h} \\ -\frac{v_n}{R_n(L)+h} \\ -\frac{v_E tan(L)}{R_e(L)+h} \end{bmatrix} \tag{2}$$

where:

$$v_e = R_i^e(v_i - \Omega_i r_i) \tag{3}$$

$$R_i^e = \begin{bmatrix} cos\omega_t & sin\omega_t & 0 \\ -sin\omega_t & cos\omega_t & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

$$R_E(L) = \frac{R_0}{\sqrt{1 - e^2 sin^2(L)}} \tag{5}$$

Now that we have ω_e^n, we create Ω_e^n and Ω_i^b following the screw form, but also find $R_{b,t}^n$ as:

$$R_{b,t}^n \approx R_{b,t-1}^n(I_3 + \Omega_i^b \delta t) - (\Omega_i^e + \Omega_e^n)R_{b,t-1}^n \delta t \tag{6}$$

Velocity Update

We can now begin our velocity update. We first calculate our resulting rotational velocity:

$$f_{n,t} \approx \frac{1}{2}(R^n_{b,t-1} + R^n_{b,t})f_{b,t} \tag{7}$$

$$v_{n,t} = v_{n,t-1} + \delta t(f_{n,t} + g(L_{t-1}, h_{t-1}) - (\Omega^n_{e,t-1} + 2\Omega^e_{i,t-1})v_{n,t-1}) \tag{8}$$

wherein $g(L, h)$ is the Somigliana gravity model, as provided in earth.py

Position Update

Finally, we can calculate our position update utilizing the equations and variables given as per the assignment and as calculated above respectively.

$$h_t = h_{t-1} + \frac{\delta t}{2}(v_{D,t-1} + v_{D,t}) \tag{9}$$

$$L_t = L_{t-1} + \frac{\delta t}{2}(\frac{v_{N,t-1}}{R_e(L_{t-1}) + h_{t-1}} + \frac{v_{N,t}}{R_e(L_{t-1}) + h_t}) \tag{10}$$

$$\lambda_t = \lambda_{t-1} + \frac{\delta t}{2}(\frac{v_{E,t-1}}{(R_E(L_{t-1}) + h_{t-1})cosL_{t-1}} + \frac{v_{E,t}}{(R_E(L_{t-1}) + h_t)cosL_t}) \tag{11}$$

This propagation model is coded up in python for use in our UKF implementation in main_ff.py as given below:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from scipy.spatial.transform import Rotation as R
from earth import *

def altitude_update(x_prior, wib, dt):
    '''
    Updates the attitude using the feedback architecture.
    Inputs:
        x_prior (np.ndarray): A 15x1 numpy array representing the prior state.
        wib (np.ndarray): A 3x1 numpy array representing gyro measurement.
        dt (float): Time step.
    Outputs:
        Prior Rotation Matrix and Updated Rotation Matrix.
    '''

    # Earths rate of rotation (rad/s)
    we = 7.2921157E-5

    # wie is the skew symmetric matrix of we
    omegaie = np.array([[0, we, 0], [-we, 0, 0], [0, 0, 0]])

    Rn, Re, Rp = principal_radii(x_prior[0,0], x_prior[2,0])

    wen = np.array([x_prior[7,0] / (Re + x_prior[2,0]), -x_prior[6,0] / (Rn +
                                                          x_prior[2,0]), -x_prior[7,0] * np.
                                                          tan(x_prior[0,0]) / (Re + x_prior[2,
                                                          0])]).reshape(-1,1)
    omegaen = np.array([[0, -wen[2,0], wen[1,0]], [wen[2,0], 0, -wen[0,0]], [-wen
                                             [1,0], wen[0,0], 0]])

    omegaib = np.array([[0, -wib[2,0], wib[1,0]], [wib[2,0], 0, -wib[0,0]], [-wib
                                             [1,0], wib[0,0], 0]])

    # Compute the prior rotation matrix using the angles from the prior state
    Rbtminusonen = R.from_euler('xyz', x_prior[3:6,0].T).as_matrix()

    # Compute the updated rotation matrix
    Rbtn = Rbtminusonen @ (np.eye(3) + omegaib * dt) - (omegaie + omegaen) @
                                             Rbtminusonen * dt

    return Rbtminusonen, Rbtn

def velocity_update(x_prior, R_updated, R_prior, fb, dt):
    '''
    Updates the velocity using the feedback architecture.
    Inputs:
        x_prior (np.ndarray): A 15x1 numpy array representing the prior state.
        R_updated (np.ndarray): A 3x3 numpy array representing the updated
                                             rotation matrix.
        R_prior (np.ndarray): A 3x3 numpy array representing the prior rotation
                                             matrix.
        fb (np.ndarray): A 3x1 numpy array representing the accelerometer
                                             measurement.
        dt (float): Time step.
    Outputs:
        Updated Velocity (np.ndarray): A 3x1 numpy array representing the updated
                                             velocity.
    '''

    fnt = 0.5 * (R_updated + R_prior) @ fb

    # Repeat some steps from altitude update
    # Earths rate of rotation (rad/s)
    we = 7.2921157E-5

    # wie is the skew symmetric matrix of we
    omegaie = np.array([[0, we, 0], [-we, 0, 0], [0, 0, 0]])

    Rn, Re, Rp = principal_radii(x_prior[0,0], x_prior[2,0])

    wen = np.array([x_prior[7,0] / (Re + x_prior[2,0]), -x_prior[6,0] / (Rn +
                                             x_prior[2,0]), -x_prior[7,0] * np.
                                             tan(x_prior[0,0]) / (Re + x_prior[2,
                                             0])]).reshape(-1,1)

    omegaen = np.array([[0, -wen[2,0], wen[1,0]], [wen[2,0], 0, -wen[0,0]], [-wen
                                             [1,0], wen[0,0], 0]])

    g_LH = gravity_n(x_prior[0,0], x_prior[2,0]).reshape(-1,1)

    vnt = x_prior[6:9,0].reshape(-1,1) + dt*(fnt + g_LH - (omegaen + 2*omegaie)
                                             @x_prior[6:9,0].reshape(-1,1))

    return vnt

def position_update(x_prior, vnt, dt):
    '''
    Updates the position using the feedback architecture.
    Inputs:
        x_prior (np.ndarray): A 15x1 numpy array representing the prior state.
        vnt (np.ndarray): A 3x1 numpy array representing the updated velocity.
    Outputs:
        (np.ndarray): A 3x1 numpy array representing the updated latitude,
                                        longitude, and altitude.
    '''

    # Compute the updated altitude
    ht = x_prior[2,0] - 0.5 * dt * (x_prior[8,0] + vnt[2,0])

    Rn, Re, Rp = principal_radii(x_prior[0,0], x_prior[2,0])

    Lat_updated = x_prior[0,0] + 0.5 * dt * (x_prior[6,0] / (Rn + x_prior[2,0]) +
                                            vnt[0,0] / (Rn + ht))

    Rn_updated, Re_updated, Rp_post = principal_radii(Lat_updated, ht)

    Lon_updated = x_prior[1,0] + 0.5 * dt * (x_prior[7,0] / ((Re + x_prior[2,0])*
                                           np.cos(np.deg2rad(x_prior[0,0]))) +
                                           vnt[1,0]/(Re_updated + ht)*np.cos(np
                                           .deg2rad(Lat_updated)))

    return Lat_updated, Lon_updated, ht


def feedforward_propagation_model(x_prior, gyro, acc, dt):
    '''
    Propagates the state using the feedback architecture.
    Inputs:
        x_prior (np.ndarray): A 15x1 numpy array representing the prior state.
        gyro (np.ndarray): A 3x1 numpy array representing gyro measurement.
        acc (np.ndarray): A 3x1 numpy array representing accelerometer
                                        measurement.
        dt (float): Time step.
    Outputs:
        (np.ndarray): A 15x1 numpy array representing the updated state.
    '''

    # Update the attitude
    Rbtminusonen, Rbtn = altitude_update(x_prior, gyro, dt)

    # Update the velocity
    vnt = velocity_update(x_prior, Rbtn, Rbtminusonen, acc, dt)

    # Update the position
    lat, lon, alt = position_update(x_prior, vnt, dt)

    #Get the roll, pitch and yaw angles
    qt = R.from_matrix(Rbtn).as_euler('xyz', degrees=True).reshape(-1,1)

    # The updated state vector is given by

    x_updated = np.vstack((lat, lon, alt, qt, vnt, x_prior[9:12,:]))

    return x_updated
```

Task 3

In task 3, according to the assignment, we have to implement the ”Nonlinear Full State implementation”, which is given by the state vector

$$x = \begin{bmatrix} L & \lambda & h & \phi & \theta & \psi & V_N & V_E & V_D & b_{a_x} & b_{a_y} & b_{a_z} & b_{g_x} & b_{g_y} & b_{g_z} \end{bmatrix}$$

The propagation model for this state vector remains the same as Task 2, except that we subtract the biases from the IMU in the altitude and velocity update.

Propagation Model

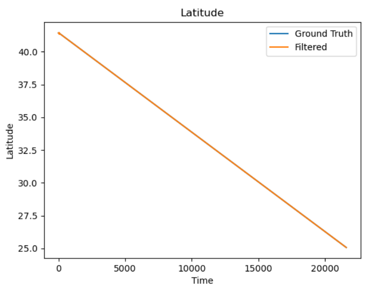
The coded up form of the propagation model for this feedback state implementation is given as follows

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from scipy.spatial.transform import Rotation as R
from earth import *

def altitude_update(x_prior, wib, dt):
    '''
    Updates the attitude using the feedback architecture.
    Inputs:
        x_prior (np.ndarray): A 15x1 numpy array representing the prior state.
        wib (np.ndarray): A 3x1 numpy array representing gyro measurement.
        dt (float): Time step.
    Outputs:
        Prior Rotation Matrix and Updated Rotation Matrix.
    '''

    # Subtract the bias from the gyro measurement
    wib = wib - x_prior[12:15,:]

    # Earths rate of rotation (rad/s)
    we = 7.2921157E-5

    # wie is the skew symmetric matrix of we
    omegaie = np.array([[0, we, 0], [-we, 0, 0], [0, 0, 0]])

    Rn, Re, Rp = principal_radii(x_prior[0,0], x_prior[2,0])

    wen = np.array([x_prior[7,0] / (Re + x_prior[2,0]), -x_prior[6,0] / (Rn +
                                        x_prior[2,0]), -x_prior[7,0] * np.
                                        tan(x_prior[0,0]) / (Re + x_prior[2,
                                        0])]).reshape(-1,1)

    omegaen = np.array([[0, -wen[2,0], wen[1,0]], [wen[2,0], 0, -wen[0,0]], [-wen
                                        [1,0], wen[0,0], 0]])

    omegaib = np.array([[0, -wib[2,0], wib[1,0]], [wib[2,0], 0, -wib[0,0]], [-wib
                                        [1,0], wib[0,0], 0]])

    # Compute the prior rotation matrix using the angles from the prior state
    Rbtminusonen = R.from_euler('xyz', x_prior[3:6,0].T).as_matrix()

    # Compute the updated rotation matrix
    Rbtn = Rbtminusonen @ (np.eye(3) + omegaib * dt) - (omegaie + omegaen) @
                                         Rbtminusonen * dt

    return Rbtminusonen, Rbtn

def velocity_update(x_prior, R_updated, R_prior, fb, dt):
    '''
    Updates the velocity using the feedback architecture.
    Inputs:
        x_prior (np.ndarray): A 15x1 numpy array representing the prior state.
        R_updated (np.ndarray): A 3x3 numpy array representing the updated
                                                    rotation matrix.
        R_prior (np.ndarray): A 3x3 numpy array representing the prior rotation
                                                    matrix.
        fb (np.ndarray): A 3x1 numpy array representing the accelerometer
                                                    measurement.
        dt (float): Time step.
    Outputs:
        Updated Velocity (np.ndarray): A 3x1 numpy array representing the updated
                                                     velocity.
    '''

    # Subtract the bias from the accelerometer measurement
    fb = fb - x_prior[9:12,:]

    fnt = 0.5 * (R_updated + R_prior) @ fb

    # Repeat some steps from altitude update
    # Earths rate of rotation (rad/s)
    we = 7.2921157E-5

    # wie is the skew symmetric matrix of we
    omegaie = np.array([[0, we, 0], [-we, 0, 0], [0, 0, 0]])

    Rn, Re, Rp = principal_radii(x_prior[0,0], x_prior[2,0])

    wen = np.array([x_prior[7,0] / (Re + x_prior[2,0]), -x_prior[6,0] / (Rn +
                                                x_prior[2,0]), -x_prior[7,0] * np.
                                                tan(x_prior[0,0]) / (Re + x_prior[2,
                                                0])]).reshape(-1,1)

    omegaen = np.array([[0, -wen[2,0], wen[1,0]], [wen[2,0], 0, -wen[0,0]], [-wen
                                                [1,0], wen[0,0], 0]])

    g_LH = gravity_n(x_prior[0,0], x_prior[2,0]).reshape(-1,1)

    vnt = x_prior[6:9,0].reshape(-1,1) + dt*(fnt + g_LH - (omegaen + 2*omegaie)
                                                @x_prior[6:9,0].reshape(-1,1))

    return vnt

def position_update(x_prior, vnt, dt):
    '''
    Updates the position using the feedback architecture.
    Inputs:
        x_prior (np.ndarray): A 15x1 numpy array representing the prior state.
        vnt (np.ndarray): A 3x1 numpy array representing the updated velocity.
    Outputs:
        (np.ndarray): A 3x1 numpy array representing the updated latitude,
                                                    longitude, and altitude.
    '''

    # Compute the updated altitude
    ht = x_prior[2,0] - 0.5 * dt * (x_prior[8,0] + vnt[2,0])

    Rn, Re, Rp = principal_radii(x_prior[0,0], x_prior[2,0])

    Lat_updated = x_prior[0,0] + 0.5 * dt * (x_prior[6,0] / (Rn + x_prior[2,0]) +
                                                 vnt[0,0] / (Rn + ht))
    Rn_updated, Re_updated, Rp_post = principal_radii(Lat_updated, ht)

    Lon_updated = x_prior[1,0] + 0.5 * dt * (x_prior[7,0] / ((Re + x_prior[2,0])*
                                            np.cos(np.deg2rad(x_prior[0,0]))) +
                                            vnt[1,0]/(Re_updated + ht)*np.cos(np
                                            .deg2rad(Lat_updated))

    return Lat_updated, Lon_updated, ht


def feedback_propagation_model(x_prior, gyro, acc, dt):
    '''
    Propagates the state using the feedback architecture.
    Inputs:
        x_prior (np.ndarray): A 15x1 numpy array representing the prior state.
        gyro (np.ndarray): A 3x1 numpy array representing gyro measurement.
        acc (np.ndarray): A 3x1 numpy array representing accelerometer
                                            measurement.
        dt (float): Time step.
    Outputs:
        (np.ndarray): A 15x1 numpy array representing the updated state.
    '''

    # Update the attitude
    Rbtminusonen, Rbtn = altitude_update(x_prior, gyro, dt)

    # Update the velocity
    vnt = velocity_update(x_prior, Rbtn, Rbtminusonen, acc, dt)

    # Update the position
    lat, lon, alt = position_update(x_prior, vnt, dt)

    # Update the bias
    acc_bias = x_prior[9:12,:]
    gyro_bias = x_prior[12:15,:]

    #Get the roll, pitch and yaw angles
    qt = R.from_matrix(Rbtn).as_euler('xyz', degrees=True).reshape(-1,1)

    x_updated = np.vstack([lat, lon, alt, qt, vnt, acc_bias, gyro_bias])

    return x_updated
```

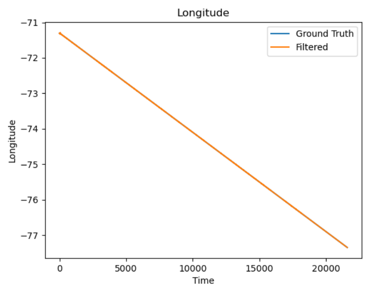
Task 4

Here, we will look at the results for both the feed-forward and feedback models.

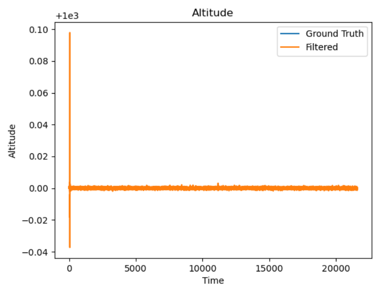
We want the UKF to perform the calculations throughout the dataset without running to matrix positive definite errors. To achieve this, a lot of noise values were tested and added to Q and R respecively to get the optimal performance as shown in the plots below.



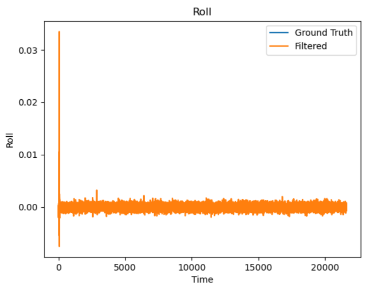
(a) Latitude



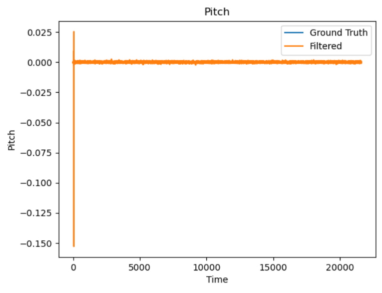
(b) Longitude



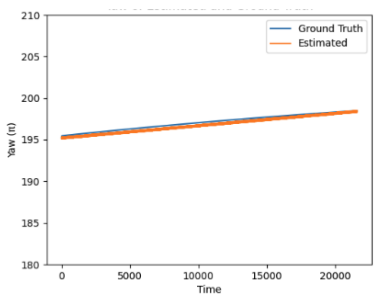
(c) Altitude



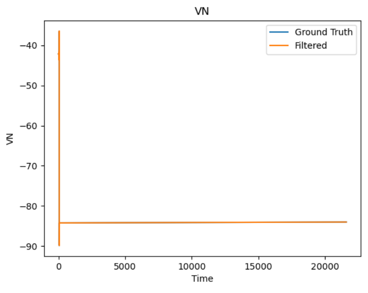
(d) Roll



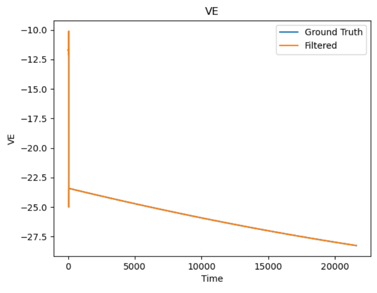
(e) Pitch



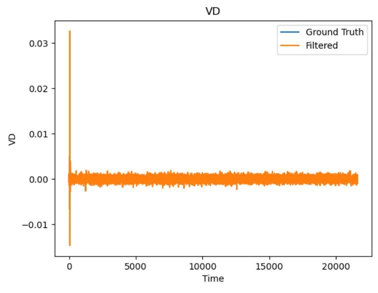
(f) Yaw



(g) VN



(h) VE



(i) VD

Figure 1: Tracking results for state variables from feedback model

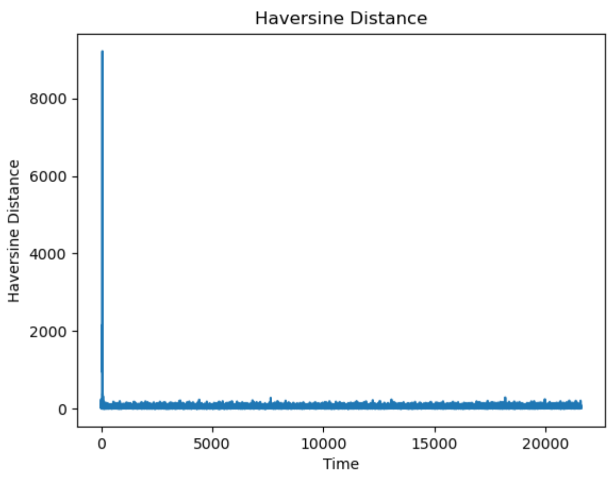
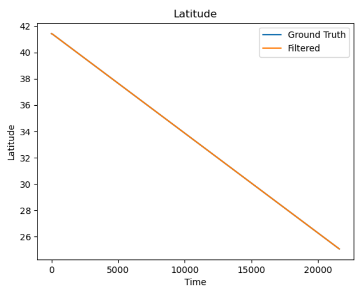
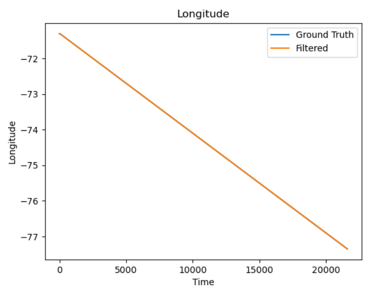


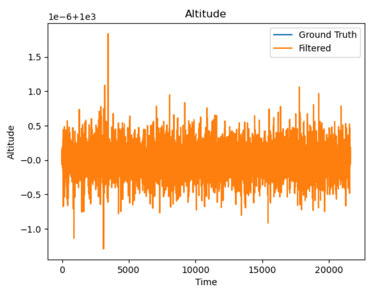
Figure 2: Haversine Distance plot for feedback model



(a) Latitude

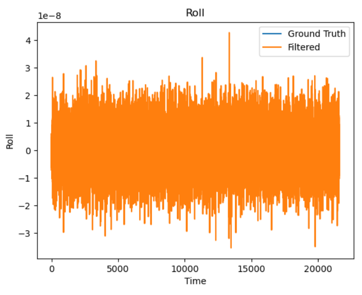


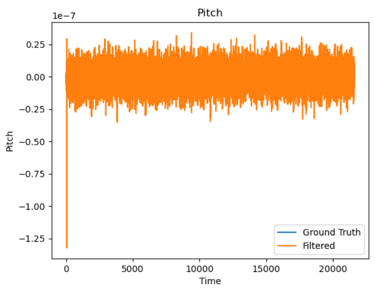
(b) Longitude



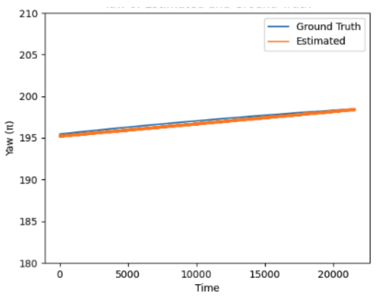
(c) Altitude

(d) Roll





(e) Pitch



(f) Yaw

Figure 3: Tracking results for state variables from feedforward model

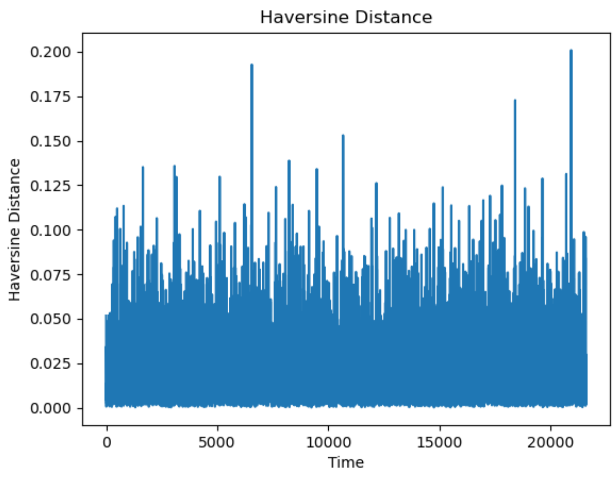


Figure 4: Haversine Distance plot for feedforward model

Discussion and Conclusion

Since our measurement model is precise, I have set the measurement noise lower than the process noise and thereby tuned the UKF for the feedback and feed-forward models. The latitude and longitude track very well.

In this implementation, the feedback model outperformed the feed-forward method. The use of Haversine distance to measure the error between the INS estimates and GNSS measurements has its disadvantages such as limited dimensions, sensitive to noise and could also potentially introduce bias especially over long distances.

Improving the system's accuracy beyond using GNSS measurements alone involves integrating additional sensors, implementing advanced filtering techniques, or combining multiple sources of information. Here are some methods:

1. **GNSS Augmentation Systems:** Incorporate data from augmentation systems like Satellite-Based Augmentation Systems (SBAS), Ground-Based Augmentation Systems (GBAS), or Real-Time Kinematic (RTK) positioning to improve GNSS accuracy.
2. **Deep Learning:** Implement Deep learning algorithms i.e neural networks to learn the mapping between sensor measurements and true states, thereby improving accuracy and robustness, especially in challenging environments.