Giorgio Mendoza

RBE595-S24-S01

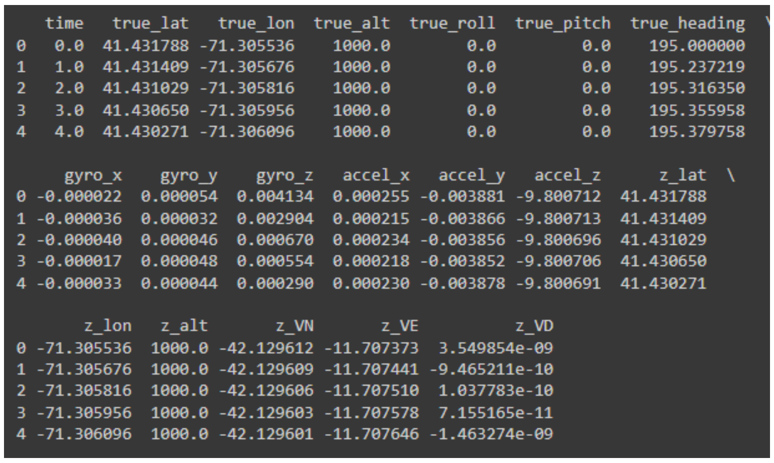
Dr. Brodovsky

INS/GNSS Integration Report

Task 1: Observation Model

In Task 1 of the project, the goal was to derive the observation model for implementing in a nonlinear Kalman Filter, specifically focusing on both feed-forward (error correction) and feedback (bias modeling) versions. The measurements provided for each trajectory include position (latitude, longitude, altitude) and velocities (northward, eastward, and downward). The task required formulating how these measurements interact with the state estimates within the Kalman Filter to improve the accuracy of position and velocity estimations in the INS/GNSS integration system.

To start the observation model, the data from the CSV file needed to be inspected and it had this format:



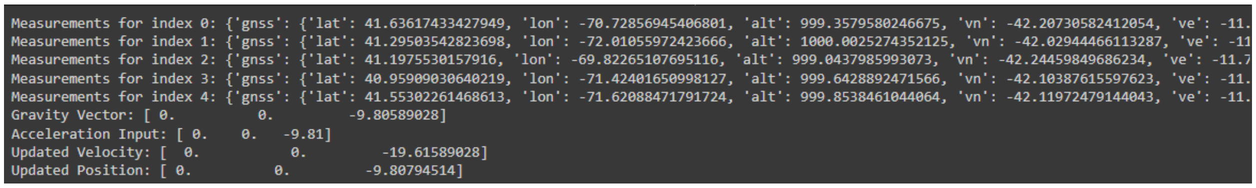
```
   time   true_lat   true_lon  true_alt  true_roll  true_pitch  true_heading  
0   0.0  41.431788 -71.305536    1000.0        0.0         0.0    195.000000
1   1.0  41.431409 -71.305676    1000.0        0.0         0.0    195.237219
2   2.0  41.431029 -71.305816    1000.0        0.0         0.0    195.316350
3   3.0  41.430650 -71.305956    1000.0        0.0         0.0    195.355958
4   4.0  41.430271 -71.306096    1000.0        0.0         0.0    195.379758

     gyro_x    gyro_y    gyro_z   accel_x   accel_y   accel_z      z_lat  \
0 -0.000022  0.000054  0.004134  0.000255 -0.003881 -9.800712  41.431788
1 -0.000036  0.000032  0.002904  0.000215 -0.003866 -9.800713  41.431409
2 -0.000040  0.000046  0.000670  0.000234 -0.003856 -9.800696  41.431029
3 -0.000017  0.000048  0.000554  0.000218 -0.003852 -9.800706  41.430650
4 -0.000033  0.000044  0.000290  0.000230 -0.003878 -9.800691  41.430271

       z_lon   z_alt       z_VN       z_VE          z_VD
0 -71.305536  1000.0 -42.129612 -11.707373  3.549854e-09
1 -71.305676  1000.0 -42.129609 -11.707441 -9.465211e-10
2 -71.305816  1000.0 -42.129606 -11.707510  1.037783e-10
3 -71.305956  1000.0 -42.129603 -11.707578  7.155165e-11
4 -71.306096  1000.0 -42.129601 -11.707646 -1.463274e-09
```

The observation model included calculations to consider the Earth's curvature and gravitational variations, ensuring accurate positioning and movement predictions. By combining these calculations with real-time sensor data, the model was capable of continuously updating and predicting the system's understanding of the object’s trajectory and orientation. This groundwork was crucial for the subsequent application of the Kalman Filter, aiming to improve navigation precision.

```python
import numpy as np

def get_sensor_measurements(data_df, index, add_noise=True):
    """
    Retrieves and optionally simulates sensor measurements for a given index from the data
DataFrame.

    Parameters:
        data_df (DataFrame): The DataFrame containing all sensor data.
        index (int): Index for which to retrieve data.
        add_noise (bool): Whether to add simulated noise to the measurements.

    Returns:
        dict: A dictionary containing simulated sensor measurements.
    """
    # Standard deviations for simulated noise (example values)
    pos_std = 0.5  # meters for position
    vel_std = 0.1  # m/s for velocity
    att_std = np.deg2rad(0.5)  # radians for attitude
    acc_std = 0.01  # m/s^2 for acceleration
    gyro_std = np.deg2rad(0.01)  # rad/s for gyro

    measurement = {
        "gnss": {
            "lat": data_df.at[index, 'z_lat'] + (np.random.randn() * pos_std if add_noise else
0),
            "lon": data_df.at[index, 'z_lon'] + (np.random.randn() * pos_std if add_noise else
0),
            "alt": data_df.at[index, 'z_alt'] + (np.random.randn() * pos_std if add_noise else
0),
            "vn": data_df.at[index, 'z_VN'] + (np.random.randn() * vel_std if add_noise else 0),
            "ve": data_df.at[index, 'z_VE'] + (np.random.randn() * vel_std if add_noise else 0),
            "vd": data_df.at[index, 'z_VD'] + (np.random.randn() * vel_std if add_noise else 0)
        },
        "imu": {
            "gyro_x": data_df.at[index, 'gyro_x'] + (np.random.randn() * gyro_std if add_noise
else 0),
            "gyro_y": data_df.at[index, 'gyro_y'] + (np.random.randn() * gyro_std if add_noise
else 0),
            "gyro_z": data_df.at[index, 'gyro_z'] + (np.random.randn() * gyro_std if add_noise
else 0),
            "accel_x": data_df.at[index, 'accel_x'] + (np.random.randn() * acc_std if add_noise
else 0),
            "accel_y": data_df.at[index, 'accel_y'] + (np.random.randn() * acc_std if add_noise
else 0),
            "accel_z": data_df.at[index, 'accel_z'] + (np.random.randn() * acc_std if add_noise
else 0)
        }
    }
    return measurement

def generate_noise_samples(num_samples, std_dev):
    """
    Generates multiple samples of noise for simulation purposes.

    Parameters:
        num_samples (int): Number of noise samples to generate.
        std_dev (float): Standard deviation of the noise.

    Returns:
        np.ndarray: Array of noise samples.
    """
    return np.random.normal(0, std_dev, num_samples)

# Example of generating multiple noise samples for position noise
num_samples = 1000
pos_noise_samples = generate_noise_samples(num_samples, 0.5)  # 0.5 meters std dev

# Assume data_df has been properly loaded and prepared
for index in range(5):  # Just an example to print first 5 entries
    current_measurement = get_sensor_measurements(data_df, index, add_noise=True)
    print(f"Measurements for index {index}: {current_measurement}")

def propagate_state(current_state, dt, imu_measurements, lat, alt):
    position = np.array(current_state['position'])
    velocity = np.array(current_state['velocity'])
    orientation = np.array(current_state['orientation'])

    acceleration = np.array(imu_measurements['acceleration'])
    gyro = np.array(imu_measurements['gyro'])

    # Assuming gravity needs to be negative in the z-direction
    gravity_vector = calculateGravityVector(lat, alt)
    gravity_vector[2] = -abs(gravity_vector[2])  # Ensure gravity is directed downward

    new_velocity = velocity + (acceleration + gravity_vector) * dt
    new_position = position + velocity * dt + 0.5 * (acceleration + gravity_vector) * dt**2
    new_orientation = orientation + gyro * dt
    print("Gravity Vector:", gravity_vector)
    print("Acceleration Input:", acceleration)
    print("Updated Velocity:", new_velocity)
    print("Updated Position:", new_position)

    return {'position': new_position, 'velocity': new_velocity, 'orientation': new_orientation}


# Example initial state
current_state = {
    'position': np.array([0, 0, 0]),
    'velocity': np.array([0, 0, 0]),
    'orientation': np.array([0, 0, 0])  # Assuming roll, pitch, yaw in radians
}

# Example IMU measurements
imu_measurements = {
    'acceleration': np.array([0, 0, -9.81]),  # Simple static example
    'gyro': np.array([0, 0, 0])
}

# Propagate state
updated_state = propagate_state(current_state, 1.0, imu_measurements, 45.0, 100)  # dt = 1
second, lat 45 deg, alt 100 m
```

The code above defines a method to fetch and optionally simulate sensor data from a dataset for navigation systems testing. It adjusts real measurements by adding random noise, reflecting real-world inaccuracies found in GNSS and IMU sensors. This simulation helps validate and tune navigation algorithms under varying conditions just as a test.

The function enhances the real sensor data with noise across various parameters, including position, velocity, and inertial measurements, to provide a comprehensive dataset for system evaluation.



```
Measurements for index 0: {'gnss': {'lat': 41.63617433427949, 'lon': -70.72856945406801, 'alt': 999.3579580246675, 'vn': -42.20730582412054, 've': -11.
Measurements for index 1: {'gnss': {'lat': 41.29503542823698, 'lon': -72.01055972423666, 'alt': 1000.0025274352125, 'vn': -42.02944466113287, 've': -11
Measurements for index 2: {'gnss': {'lat': 41.1975530157916, 'lon': -69.82265107695116, 'alt': 999.0437985993073, 'vn': -42.24459849686234, 've': -11.7
Measurements for index 3: {'gnss': {'lat': 40.95909030640219, 'lon': -71.42401650998127, 'alt': 999.6428892471566, 'vn': -42.10387615597623, 've': -11.
Measurements for index 4: {'gnss': {'lat': 41.55302261468613, 'lon': -71.62088471791724, 'alt': 999.8538461044064, 'vn': -42.11972479144043, 've': -11.
Gravity Vector: [ 0.          0.         -9.80589028]
Acceleration Input: [ 0.    0.   -9.81]
Updated Velocity: [  0.           0.         -19.61589028]
Updated Position: [ 0.          0.         -9.80794514]
```

The output displays simulated GNSS and IMU sensor data, including positions, velocities, and inertial measurements, each with random noise to simulate real-world sensor inaccuracies. The output also demonstrates gravity vector calculations and state propagation, needed for navigation tasks.

Task 2: Nonlinear Error State Implementation

This task involved the creation of a nonlinear Kalman filter using the nonlinear error state model, which includes variables for latitude (L), longitude (λ), altitude (h), Euler angles (θ, ϕ, ψ) for orientation, velocities (v_N, v_E, v_D), and error states (e_L, e_λ, e_h) to correct deviations from the base position. The error state model focuses on estimating the short-term errors in the navigation states, allowing the filter to correct the system state dynamically.

Task 3: Nonlinear Full State Implementation

The third task extended the model from Task 2 by incorporating biases in the IMU measurements, represented as b_a_x, b_a_y, b_a_z for accelerometers and b_g_x, b_g_y, b_g_z for gyroscopes. This full state model not only estimates the vehicle's position, orientation, and velocity but also adapts to long-term sensor errors through bias estimation. This approach is beneficial for systems where sensor biases are significant and can drift over time.

The Kalman Filter class can support both Feedback model and Feed Forward model. The Feedforward model focuses on predicting the next state based on current measurements without incorporating control inputs or previous states directly into the state estimation. The Feedback model incorporates feedback from the system's response, adjusting not only based on the measurements but also accommodating the system dynamics and any control inputs applied, including sensor biases.

Both tasks use the same functions predict_state and update_state to manage the propagation and update steps of the Kalman Filter. These functions handle the nonlinear dynamics and measurement integrations necessary for maintaining accurate state estimates under varying operational conditions. They also use sigma points to capture the posterior state estimate and uncertainty, which is needed for systems with nonlinear behaviors as this one.

```
import numpy as np
import pandas as pd  # Make sure to import pandas
import matplotlib.pyplot as plt


def predict_state(ukf, delta_t, fb, wb):
    sigma_points = ukf.computeSigmaPoints(ukf.mu, ukf.sigma)
    predicted_mu, predicted_sigma, _ = ukf.getPrediction(sigma_points, fb, wb, delta_t)
    predicted_sigma = ukf.fixCovariance(predicted_sigma)
    return predicted_mu, predicted_sigma

def update_state(ukf, predicted_mu, predicted_sigma, actual_measurements):
    sigma_points = ukf.computeSigmaPoints(predicted_mu, predicted_sigma)
    updated_mu, updated_sigma = ukf.updateState(actual_measurements, predicted_mu,
predicted_sigma, sigma_points)
    updated_sigma = ukf.fixCovariance(updated_sigma)
    return updated_mu, updated_sigma

# Initialize UKF with specified noise parameters
ukf = unscentedKalmanFilter(
    modelType='FEEDBACKMODEL',  # Feedback model type
    noiseScaleMeasurement=0.50,
    noiseScalePrediction=1e-4,
    k1=0,  # Zero or another appropriate value depending on your understanding
    a1=1e-3,
    b1=2
)

ukf.mu = np.zeros((ukf.stDim, 1))  # Initialize state estimate as zero
ukf.sigma = np.eye(ukf.stDim) * 0.1  # Initial covariance

# Load data
data_df = pd.DataFrame(data_list)  # Assume data_list is previously defined

# Lists to store trajectory data for plotting
lon_true, lat_true = [], []
lon_est, lat_est = [], []

# Processing a limited number of data points for demonstration
for index in range(min(10000, len(data_df))):  # Processing only first 500 points
    current_measurement = get_sensor_measurements(data_df, index, add_noise=True)

    fb = np.array([current_measurement['imu']['accel_x'], current_measurement['imu']['accel_y'],
current_measurement['imu']['accel_z']])
    wb = np.array([current_measurement['imu']['gyro_x'], current_measurement['imu']['gyro_y'],
current_measurement['imu']['gyro_z']])
    delta_t = 1.0  # Assuming 1 second time intervals for simplicity

    actual_measurements = np.array([
        current_measurement['gnss']['lat'],
        current_measurement['gnss']['lon'],
        current_measurement['gnss']['alt'],
        current_measurement['gnss']['vn'],
        current_measurement['gnss']['ve'],
        current_measurement['gnss']['vd']
    ]).reshape(6, 1)  # Reshape for consistency with UKF state dimensions

    predicted_mu, predicted_sigma = predict_state(ukf, delta_t, fb, wb)
    updated_mu, updated_sigma = update_state(ukf, predicted_mu, predicted_sigma,
actual_measurements)

    # Storing estimated and true positions for trajectory plotting
    lon_est.append(updated_mu[1][0])
    lat_est.append(updated_mu[0][0])
    lon_true.append(current_measurement['gnss']['lon'])
    lat_true.append(current_measurement['gnss']['lat'])
```

The plots below illustrate the results for Task2 and Task3 using about 20000 points.

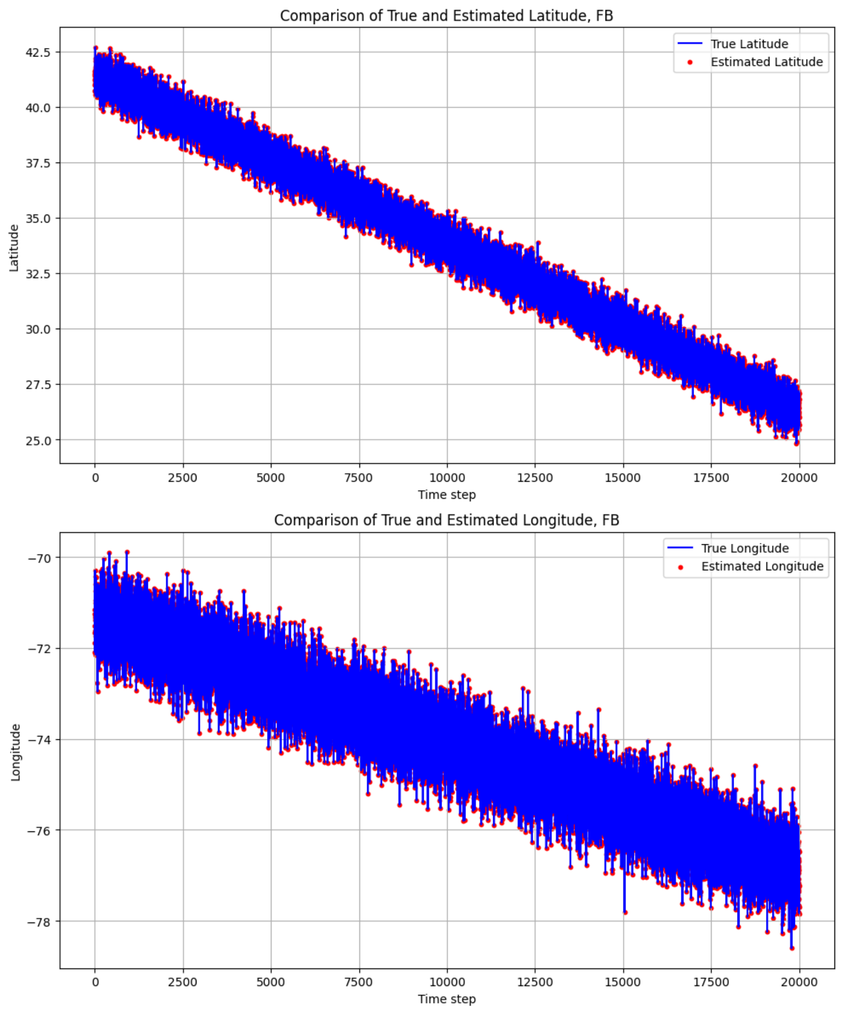


Figure 1: True vs Estimated Latitude using Feedback model

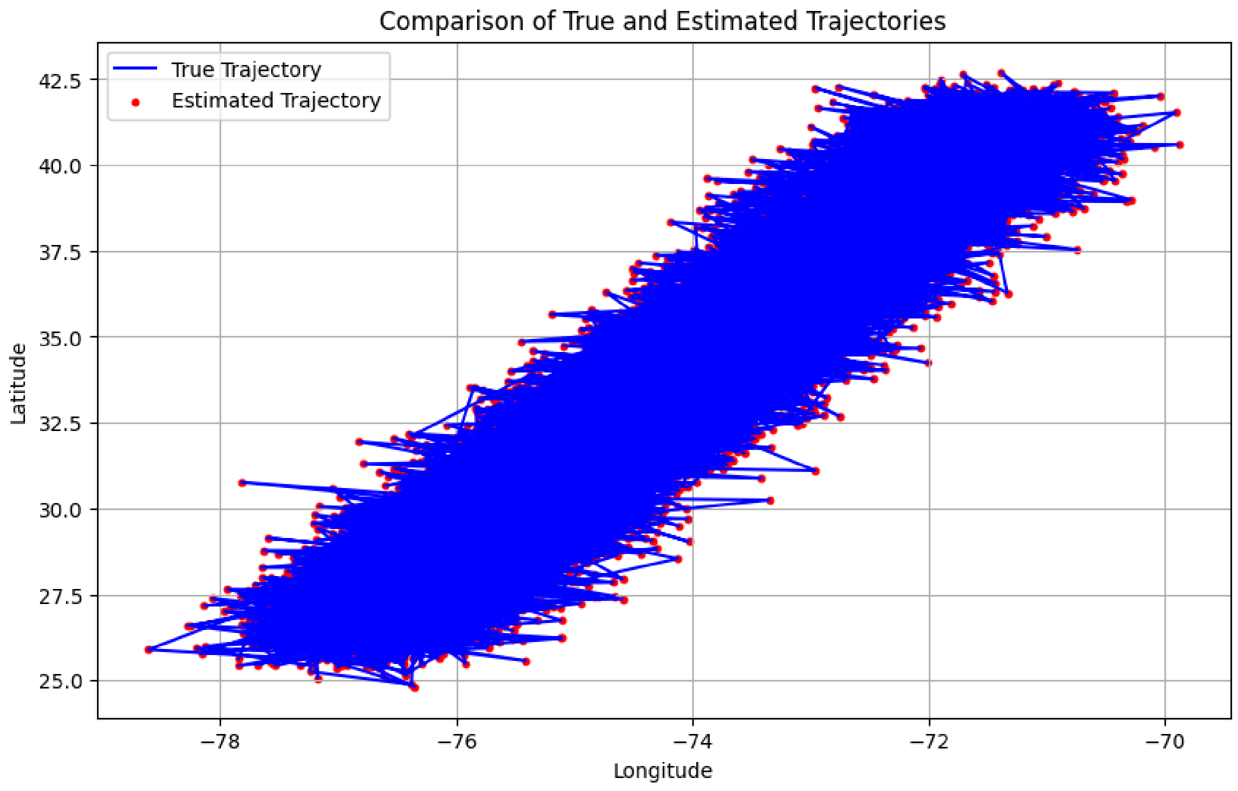


Figure 2: True vs Estimated Trajectories using Feedback model

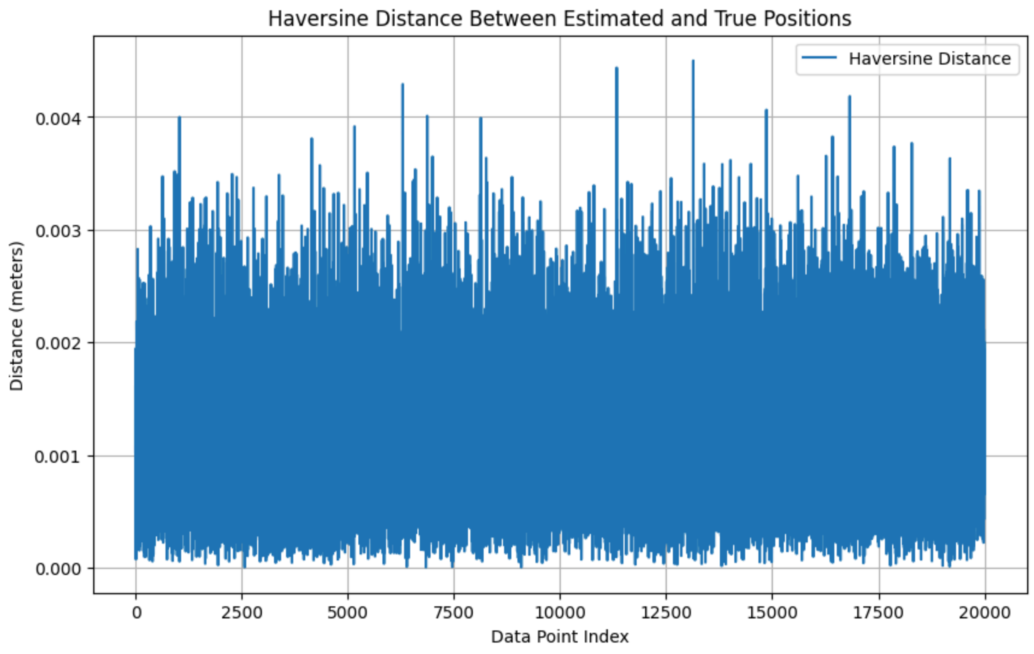


Figure 3: Haversine Distance using Feedback model

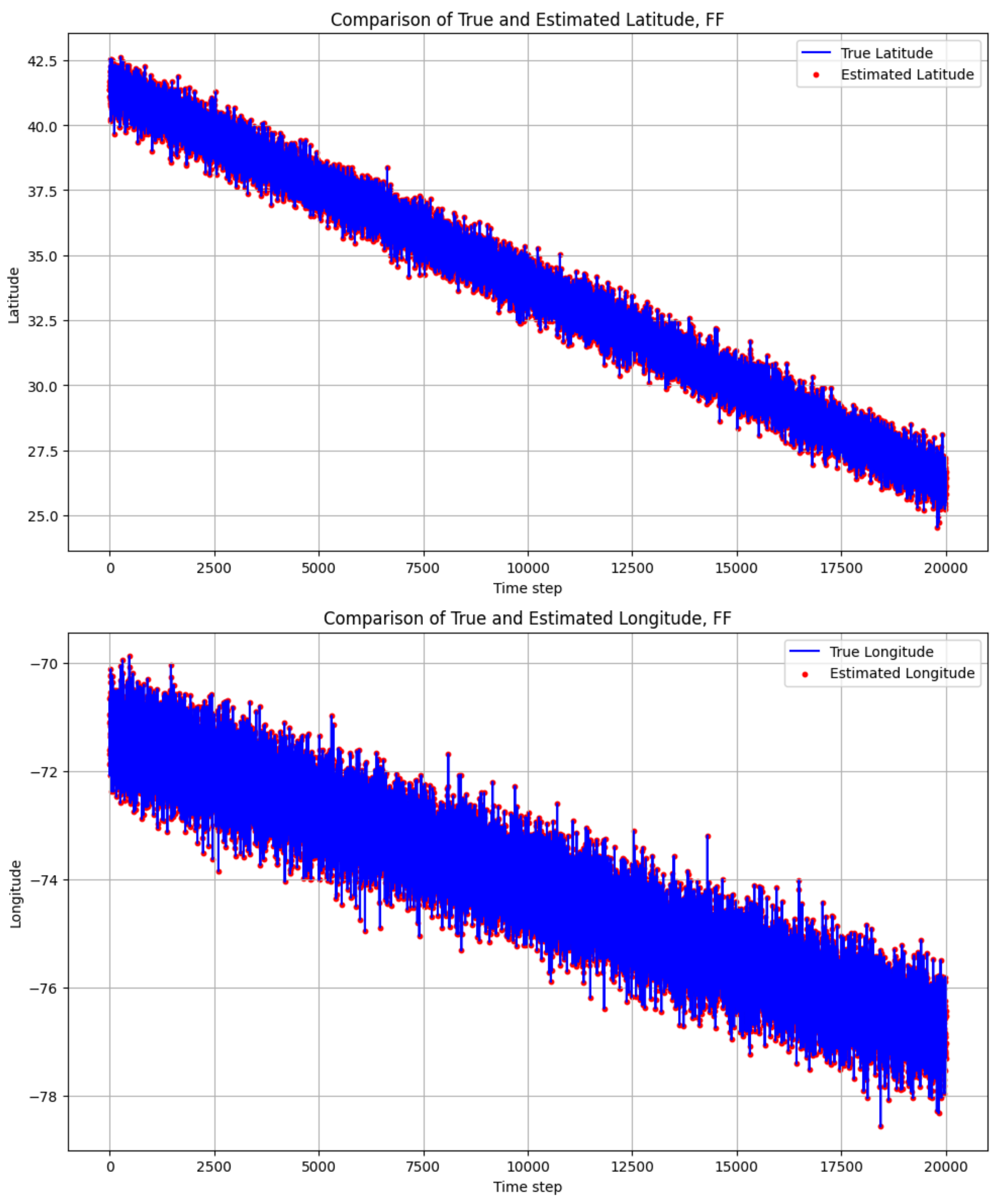


Figure 4: True vs Estimated Latitude using Feedforward model

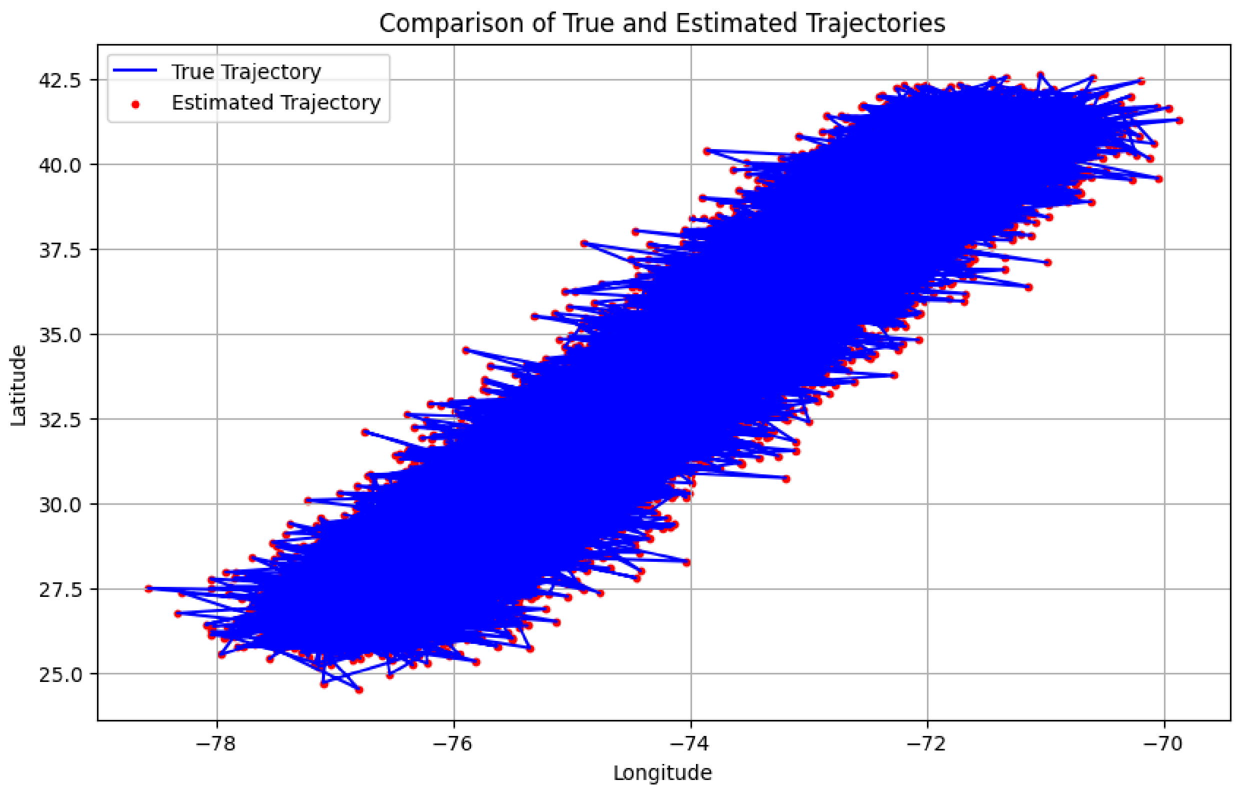


Figure 5: True vs Estimated Trajectories using Feedforward model

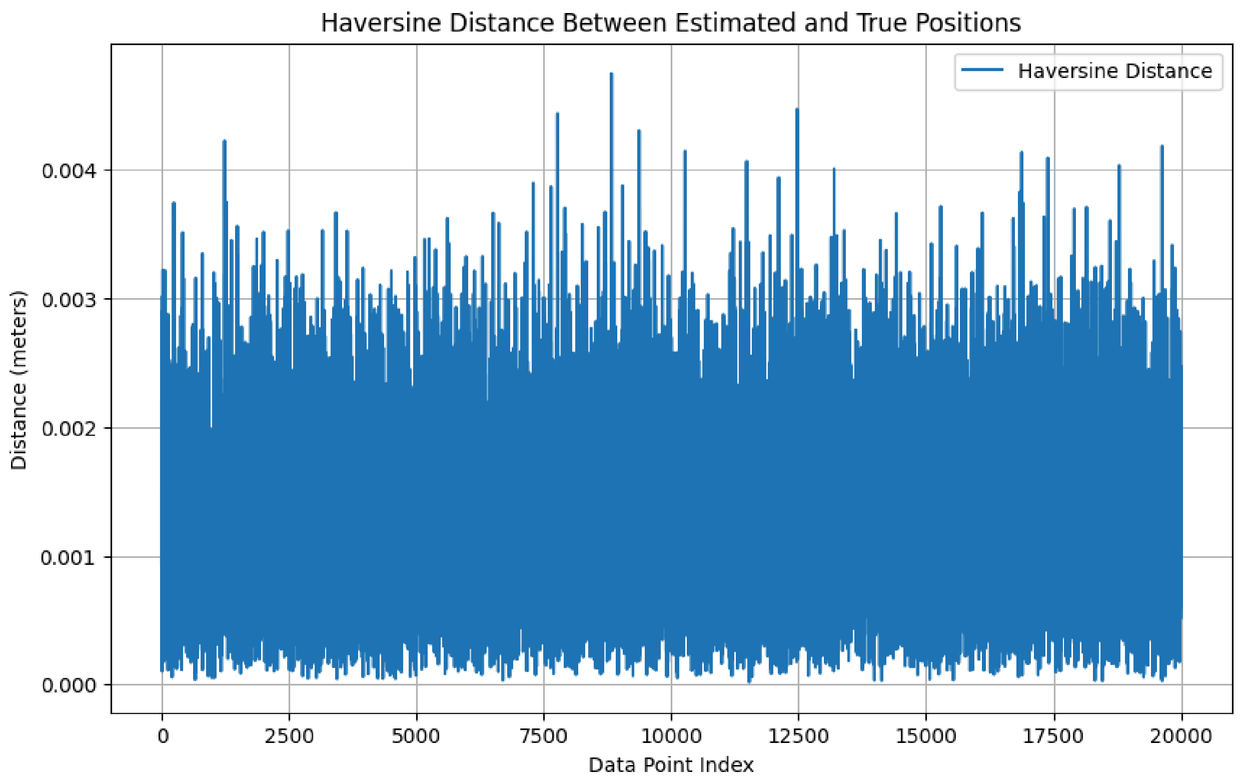


Figure 6: Haversine Distance using Feedforward model

Task 4: Discussion and performance analysis

Both the Feedback and Feedforward models demonstrate a reasonable performance in tracking the true trajectory. The latitude and longitude plots indicate that the estimated values closely follow the true values, though there is noticeable noise in the estimates, particularly in the longitude plot for the FF model.

The trajectory plots also show a close alignment between the estimated and true paths, with some deviations that appear more pronounced in specific sections of the route. These deviations suggest areas where tuning the Kalman filter parameters could potentially improve accuracy.

Lastly, the Haversine distance plots, which measure the error between the estimated positions and true positions, reveal that most errors are within a few meters, indicating a relatively accurate estimation. However, spikes in the errors may correspond to moments when the sensors' data was especially noisy or when there were abrupt changes in movement dynamics, such as turns or sudden accelerations/decelerations.

In terms of enhancements or improvements, refining the noise models for both the process and measurement updates, adjusting the Kalman filter's parameters, or incorporating additional sensor data if available could improve the model accuracy. Analyzing the specific points where large deviations occur could also provide insights into the causes of these errors and how to fix them.

Appendix:

Code for KF:

```python
from __future__ import annotations
from haversine import Unit, haversine
from typing import List, Tuple
from scipy.linalg import sqrtm
from scipy.spatial.transform import Rotation
import numpy as np

        #Initializes the Unscented Kalman Filter with specified parameters.
        #:param k1: Tuning parameter that affects the spread of sigma points.
        #:param a1: Scale parameter influencing the sigma point spread around the mean.
        #:param b1: Parameter to incorporate prior knowledge of the distribution (for Gaussian,
b1=2 is optimal).
        #:param modelType: Type of model ('FB' for feedback control, 'FF' for feedforward
control).
        #:param noiseScaleMeasurement: Variance of measurement noise.
       # :param noiseScalePrediction: Variance of process noise.

class unscentedKalmanFilter:
    def __init__(self, k1: float = 1.0, a1: float = 1.0, b1: float = 0.4, modelType: str =
"FEEDBACKMODEL", noiseScaleMeasurement: float = 5e-3, noiseScalePrediction: float = 1e-3,) ->
unscentedKalmanFilter:
        self.modelType = modelType
        stateDimensions = 12 if self.modelType == "FEEDFORWARDMODEL" else 15
        self.stDim = stateDimensions
        self.noiseScaleMeasurement = noiseScaleMeasurement
        self.noiseScalePrediction = noiseScalePrediction

        self.numberOfSigmaPoints = 2 * self.stDim + 1
        self.k1 = k1
        self.a1 = a1
        self.b1 = b1
        self.lambdaScaling = self.a1**2 * (self.stDim + self.k1) - self.stDim
        # Calculate weights for the mean and covariance based on lambda scaling
        weightsMean0 = self.lambdaScaling / (self.stDim + self.lambdaScaling)
        weightsCovariance0 = ((self.lambdaScaling / (self.stDim + self.lambdaScaling)) + (1 -
(self.a1**2)) + self.b1)

        weightI  = 1 / (2 * (self.stDim + self.lambdaScaling))
        # Initialize weight arrays for mean and covariance
        self.weightsMean  = np.zeros((self.numberOfSigmaPoints))
        self.weightsCovariance0 = np.zeros((self.numberOfSigmaPoints))
        self.weightsMean [0] = weightsMean0
        self.weightsMean [1:] = weightI
        self.weightsCovariance0[0] = weightsCovariance0
        self.weightsCovariance0[1:] = weightI

    def computeSigmaPoints(self, stateMean: np.ndarray, stateCovariance: np.ndarray) ->
np.ndarray:
        #Generates sigma points for use in the Unscented Kalman Filter update.
        if not np.allclose(stateCovariance, stateCovariance.T):
            raise ValueError("Covariance matrix is not symmetric.")
        if np.any(np.linalg.eigvalsh(stateCovariance) <= 0):
            raise ValueError("Covariance matrix is not positive semi-definite.")

        sigmaPointsArray = np.zeros((self.numberOfSigmaPoints, self.stDim, 1))
        sigmaPointsArray[0] = stateMean  #First sigma point is the mean itself
        scaleFactor = (self.stDim + self.k1) # Scaling factor for adjusting sigma points spread
        try:
            sqrtCovMatrix = sqrtm(scaleFactor * stateCovariance)
        except Exception as e:
            raise ValueError("Failed to compute the square root of the covariance matrix.") from
e
        # Generate sigma points by adding and subtracting scaled square root of covariance matrix
        for i in range(self.stDim):
            adjustmentVector = sqrtCovMatrix[:, i].reshape((self.stDim, 1))
            sigmaPointsArray[i + 1] = stateMean + adjustmentVector
            sigmaPointsArray[self.stDim + i + 1] = stateMean - adjustmentVector

        return sigmaPointsArray

    def propagationModel(self, state: np.ndarray, deltaTime: float, bodyFrameAcceleration:
np.ndarray, bodyFrameAngularRate: np.ndarray) -> np.ndarray:
        # Reshape body frame acceleration and angular rate for matrix operations
        bodyFrameAcceleration = np.array(bodyFrameAcceleration).reshape(3, 1)
        bodyFrameAngularRate = np.array(bodyFrameAngularRate).reshape(3, 1)

        # Extract state variables for easier handling
        latitude, longitude, altitude, roll, pitch, yaw, velocityNorth, velocityEast,
velocityDown = state[:9, 0]
        velocityVector = np.array([velocityNorth, velocityEast, velocityDown]).reshape(3, 1)

        # Compensate for feedback model corrections if applicable
        if self.modelType == "FEEDBACKMODEL":
            bodyFrameAcceleration -= state[9:12]
            bodyFrameAngularRate -= state[12:15]

        # Convert roll, pitch, yaw into a rotation matrix
        previousRotationMatrix = Rotation.from_euler("xyz", [roll, pitch, yaw],
degrees=True).as_matrix()

        # Define Earth's rotation rate and create the corresponding skew-symmetric matrix
        earthRate = earthRotationRate
        earthRateMatrix = np.array([[0, -earthRate, 0], [earthRate, 0, 0], [0, 0, 0]])

        # Calculate principal radii based on current latitude and altitude
        radiusNorth, radiusEast, radiusEastCosLatitude = calculatePrincipalRadii(latitude,
altitude)

        # Compute angular rates of change in navigation frame and create the corresponding
skew-symmetric matrix
        omegaNe = np.array([velocityEast / radiusEast, -velocityNorth / radiusNorth,
-(velocityEast * np.tan(np.deg2rad(latitude))) / radiusEast]).reshape(3, 1)
        skewOmegaNe = np.array([
            [0, -omegaNe[2, 0], omegaNe[1, 0]],
            [omegaNe[2, 0], 0, -omegaNe[0, 0]],
            [-omegaNe[1, 0], omegaNe[0, 0], 0],
        ])

        # Create a skew-symmetric matrix for body frame angular rates
        skewOmegaBi = np.array([
            [0, -bodyFrameAngularRate[2, 0], bodyFrameAngularRate[1, 0]],
            [bodyFrameAngularRate[2, 0], 0, -bodyFrameAngularRate[0, 0]],
            [-bodyFrameAngularRate[1, 0], bodyFrameAngularRate[0, 0], 0]
        ])

        # Update the rotation matrix with current angular movements
        updatedRotationMatrix = previousRotationMatrix @ (np.eye(3) + skewOmegaBi * deltaTime) -
((earthRateMatrix + skewOmegaNe) @ previousRotationMatrix * deltaTime)

        # Correct acceleration with updated rotation matrices
        correctedAcceleration = 0.5 * (previousRotationMatrix + updatedRotationMatrix) @
bodyFrameAcceleration
        updatedVelocity = velocityVector + deltaTime * (correctedAcceleration +
calculateGravity(latitude, altitude) - (skewOmegaNe + 2 * earthRateMatrix) @ velocityVector)

        # Calculate new altitude based on updated velocity
        newAltitude = altitude - deltaTime * 0.5 * (velocityDown + updatedVelocity[2, 0])
        radiusNorthNew, _, _ = calculatePrincipalRadii(latitude, newAltitude)
        newLatitude = latitude + deltaTime * 0.5 * (velocityNorth / radiusNorth +
updatedVelocity[0, 0] / radiusNorth)
        newLatitude += deltaTime * 0.5 * (velocityNorth / radiusNorthNew + updatedVelocity[0, 0]
/ radiusNorthNew)

        # Adjust longitude with updated eastward velocity
        _, _, radiusEastCosLatitudeNew = calculatePrincipalRadii(newLatitude, newAltitude)
        newLongitude = longitude + deltaTime * 0.5 * (velocityEast / radiusEastCosLatitude +
velocityEast / radiusEastCosLatitudeNew)

        # Derive new roll, pitch, and yaw from the updated rotation matrix
        newRoll, newPitch, newYaw = Rotation.from_matrix(updatedRotationMatrix).as_euler('xyz',
degrees=True)

        # Compile new state including unchanged state elements
        newState = np.zeros((self.stDim, 1))
        newState[:9, 0] = [newLatitude, newLongitude, newAltitude, newRoll, newPitch, newYaw,
updatedVelocity[0, 0], updatedVelocity[1, 0], updatedVelocity[2, 0]]
        newState[9:, 0] = state[9:, 0]

        return newState

    def updateState(self, measuredValues, stateMean, stateCovariance, sigmaPoints):
        # Define the size of the measurements and state
        measurementSize = measuredValues.shape[0]
        stateSize = self.stDim

        # Initialize the measurement matrix for mapping sigma points to measurements
        measurementMatrix = np.zeros((measurementSize, stateSize))
        measurementMatrix[0, 0] = 1
        measurementMatrix[1, 1] = 1
        measurementMatrix[2, 2] = 1
        measurementMatrix[3, 6] = 1
        measurementMatrix[4, 7] = 1
        measurementMatrix[5, 8] = 1

        # Transform sigma points into measurement space
        transformedSigmaPoints = np.dot(measurementMatrix, sigmaPoints.T).T

        # Calculate predicted measurements from transformed sigma points
        predictedMeasurements = np.zeros((measurementSize, 1))
        for i in range(self.numberOfSigmaPoints):
            predictedMeasurements += self.weightsMean[i] *
transformedSigmaPoints[i].reshape(measurementSize, 1)

        # Compute the innovation covariance matrix for the measurements
        innovationCovariance = np.zeros((measurementSize, measurementSize))
        for i in range(self.numberOfSigmaPoints):
            measurementDiff = transformedSigmaPoints[i].reshape(measurementSize, 1) -
predictedMeasurements
            innovationCovariance += self.weightsCovariance0[i] * np.outer(measurementDiff,
measurementDiff)
        measurementNoise = np.diag([0.1, 0.1, 1, 0.1, 0.1, 0.1])
        innovationCovariance += measurementNoise

        # Calculate cross-covariance between state and measurements
        crossCovariance = np.zeros((stateSize, measurementSize))
        for i in range(self.numberOfSigmaPoints):
            stateDiff = sigmaPoints[i].reshape(stateSize, 1) - stateMean
            measurementDiff = transformedSigmaPoints[i].reshape(measurementSize, 1) -
predictedMeasurements
            crossCovariance += self.weightsCovariance0[i] * np.dot(stateDiff, measurementDiff.T)

        # Calculate the Kalman gain
        kalmanGain = np.dot(crossCovariance, np.linalg.inv(innovationCovariance))

        # Update state mean and covariance using the Kalman gain and measurement residual
        measurementResidual = measuredValues - predictedMeasurements
        updatedStateMean = stateMean + np.dot(kalmanGain, measurementResidual)
        updatedStateCovariance = stateCovariance - np.dot(np.dot(kalmanGain,
innovationCovariance), kalmanGain.T)

        return updatedStateMean, updatedStateCovariance

    def fixCovariance(self, matrixCovariance: np.ndarray, jitterAmount: float = 1e-3):
        # Check if the matrix is symmetric and positive definite
        isSymmetric = np.allclose(matrixCovariance, matrixCovariance.T)
        isPositiveDefinite = True
        try:
            np.linalg.cholesky(matrixCovariance)
        except np.linalg.LinAlgError:
            isPositiveDefinite = False

        # If already symmetric and positive definite, return as is
        if isSymmetric and isPositiveDefinite:
            return matrixCovariance

        # Adjust the matrix to be symmetric
        matrixCovariance = (matrixCovariance + matrixCovariance.T) / 2

        # Adjust eigenvalues to ensure the matrix is positive definite
        eigenValues, eigenVectors = np.linalg.eig(matrixCovariance)
        eigenValues[eigenValues < 0] = 0
        eigenValues += jitterAmount

        # Reconstruct the matrix using adjusted eigenvalues
        matrixCovariance = eigenVectors.dot(np.diag(eigenValues)).dot(eigenVectors.T)

        return self.fixCovariance(matrixCovariance, jitterAmount=10 * jitterAmount)


    def getPrediction(self, sigmaPoints: np.ndarray, bodyFrameAcceleration: np.ndarray,
bodyFrameAngularRate: np.ndarray, deltaTime: float) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
        # Initialize array to hold transformed sigma points
        transitionedPoints = np.zeros_like(sigmaPoints)
        # Transform each sigma point through the motion model
        for i in range(sigmaPoints.shape[0]):
            transitionedPoints[i, :] = self.propagationModel(sigmaPoints[i], deltaTime,
bodyFrameAngularRate, bodyFrameAcceleration)
```

```
        # Calculate the predicted mean of the state
        predictedMean = np.zeros((self.stDim, 1))
        for i in range(self.numberOfSigmaPoints):
            predictedMean += self.weightsMean[i] * transitionedPoints[i]

        # Determine noise scale based on the model type
        if self.modelType == "FEEDFORWARDMODEL":
            noiseScale = 1e-3
        else:
            noiseScale = 5e-5

        # Compute the process noise covariance matrix
        processNoiseCovariance = np.random.normal(scale=self.noiseScalePrediction,
size=(self.stDim, self.stDim))
        # Calculate state differences for the covariance update
        stateDifferences = transitionedPoints - predictedMean
        predictedCovariance = np.zeros((self.stDim, self.stDim))
        # Compute the predicted covariance matrix by summing weighted outer products of state
differences
        for i in range(self.numberOfSigmaPoints):
            predictedCovariance += self.weightsCovariance0[i] * np.dot(stateDifferences[i],
stateDifferences[i].T)
        predictedCovariance += processNoiseCovariance

        return predictedMean, predictedCovariance, transitionedPoints

    def processMeasurements(self, data: List[SensorData]) -> Tuple[List[np.ndarray],
List[np.ndarray], List[float]]:
        groundTruth = [self.truePoseFromData(d) for d in data]
        filteredPositions = [self.truePoseFromData(data[0])]  # Initialize with the first state
        haversineDistances = []

        # Initialize state and time from the first data entry
        currentState = filteredPositions[0]
        previousStateTimestamp = data[0].time

        # Set initial process covariance matrix
        processCovarianceMatrix = np.eye(self.stDim) * 1e-3

        # Iterate over sensor data starting from the second entry
        for index in range(1, len(data)):
            read = data[index]
            deltaTime = read.time - previousStateTimestamp
            previousStateTimestamp = read.time

            # Extract IMU and GNSS data from the sensor readings
            imuData = self.imuFromData(read)
            gnssData = self.gnssFromData(read)
            bodyFrameAcceleration = imuData[0:3]
            bodyFrameAngularRate = imuData[3:6]

            # Generate sigma points, predict next state, and update
            sigmaPoints = self.computeSigmaPoints(currentState, processCovarianceMatrix)
            predictedMu, predictedSigma, transitionedPoints = self.getPrediction(sigmaPoints,
bodyFrameAcceleration, bodyFrameAngularRate, deltaTime)
            updatedMu, updatedSigma = self.update(gnssData, predictedMu, predictedSigma,
transitionedPoints)

            # Update the state and covariance matrices
            processCovarianceMatrix = updatedSigma
            currentState = updatedMu

            # Store the updated state and calculate distance to ground truth
            filteredPositions.append(currentState)
            haversineDistance = haversine(
                (currentState[0], currentState[1]),
                (groundTruth[index][0], groundTruth[index][1]),
                unit='degrees'
            )
            haversineDistances.append(haversineDistance)

        return filteredPositions, groundTruth, haversineDistances
```