ENGR 6412 Graduate Project: Neural Network Kalman Filters for Pose Estimation in 2D & 3D space

Zeyad Khalil
40270929
M.Eng Electrical and Computer Engineering
Montreal, Canada
zeyadkhalil471@gmail.com

***Abstract*—This project explores the integration of Neural Network Kalman Filtering (NNKF) into the domain of pose estimation in robotics, presenting a comparative analysis with the traditional Extended Kalman Filtering (EKF) methodology. Leveraging a meticulously designed 2D and 3D robot simulation environment, encompassing movement dynamics and sensor readings, the research aims to assess the accuracy of pose estimation and computational efficiency of both filters. The investigation entails the development of NNKF alongside EKF, offering insights into their limitations through comprehensive testing scenarios. Through this project, we endeavor to contribute valuable knowledge to the field of robotics, shedding light on the potential advancements offered by Neural Network Kalman Filtering in enhancing pose estimation accuracy.**

Keywords—NNKF, EFK, Pose Estimation, MSE

I. Introduction

In the realm of robotics, precise pose estimation is fundamental for achieving optimal performance in various applications, ranging from autonomous navigation to manipulation tasks. Traditional methods, such as Extended Kalman Filtering (EKF), have been employed for decades to estimate the state of a robotic system based on sensor observations. However, the inherent limitations of EKF, particularly in handling non-linear and complex systems, have prompted exploration into alternative methodologies.

This project introduces the integration of Neural Network Kalman Filtering (NNKF) as a novel approach to pose estimation in robotics. Neural networks, with their capacity to learn complex relationships, offer a promising avenue for addressing the shortcomings of traditional filtering methods. The project sets out to develop and implement both EKF and NNKF within a sophisticated robot simulation environment, which encompasses 2D and 3D scenarios, realistic movement dynamics, and sensor readings.

The primary objective is to rigorously assess and compare the accuracy of pose estimation achieved by EKF and NNKF, to scrutinize the computational efficiency of these methods.

II. Simulated Environment

A. 2D Simulation

In the 2D simulation environment, we've set up a world where our robot can roam around and sense its surroundings. The robot's initial pose is at the coordinates (0.0, 0.0), facing north (0 degrees). We've added a touch of realism by introducing motion noise to mimic the uncertainties in the robot's movements.

- Robot movement Models: The robot is equipped with a simple movement model that considers linear velocity and angular velocity. As the robot moves, it updates its pose by taking into account the current orientation and the time step. Motion noise is then incorporated to simulate the real-world unpredictabilities in the robot's motion.
- Sensor Characteristics: To make things interesting, we've implemented a sensor that mimics measurements you might get from, say, a distance sensor. These measurements include the range (distance to an object), bearing (orientation relative to the robot's heading), and the robot's current orientation (theta). We've spiced it up with a bit of measurement noise to keep it real.

B. 3D Simulation

In this scenario, let's add another dimension to the mix! In our 3D simulation environment, the robot not only moves along the XY plane but also explores the Z-axis. The initial pose includes the robot's position (x, y, z) and orientation (roll, pitch, yaw). We've thrown in some motion noise to make things lively.

- Robot movement Models: For 3D motion, the robot receives control inputs for linear velocity and angular velocities around the X, Y, and Z axes. The movement model updates the robot's position and orientation accordingly, factoring in the time step and, of course, a dash of motion noise.

- Sensor Characteristics: Our 3D sensor simulates measurements you might get from a spatial sensor suite. These include range (distance to an object), bearing (orientation in the XY plane), pitch (up or down tilt), and yaw (orientation around the Z-axis). Each measurement comes with a sprinkle of measurement noise for that authentic touch.

III. Extended Kalman Filter

A. 2D EKF

Our 2D EKF implementation revolves around the core principles of the Extended Kalman Filter for pose estimation. Here's the breakdown:

- Prediction Step: We predict the next state using the motion model, considering linear velocity (v) and angular velocity (omega). This involves updating the robot's pose and calculating the Jacobian matrix (G) of the motion model. The motion model accounts for the robot's movement in the x-y plane and its change in orientation (theta).
- Update Step: We define a measurement model that relates the robot's pose to sensor readings. In our case, the measurements include range, bearing, and theta. The Jacobian matrix (H) of the measurement model is calculated to link the state space to the measurement space. We introduce measurement noise (R) to account for uncertainties in sensor readings. The Kalman gain (K) is computed, and the state and covariance estimates are updated using the sensor readings.
- Localization Method: Our localize method integrates the prediction and update steps. Given control inputs (linear velocity and angular velocity) and sensor readings, it predicts the next state and updates the estimate.

B. 3D EKF

Our 3D EKF implementation takes pose estimation to the next dimension. Here's a breakdown of the key elements:

- Prediction Step: Similar to the 2D case, we predict the next state using a 3D motion model that considers linear velocity (v) and angular velocities around the X, Y, and Z axes. The Jacobian matrix (G) of the motion model is computed to update the robot's pose and orientation.
- Update Step: The measurement model relates the robot's pose to 3D sensor readings, including range, bearing, pitch, and yaw. The Jacobian matrix (H) is calculated to map the state space to the measurement space. Measurement noise (R) is introduced to account for uncertainties in sensor readings. The Kalman gain (K) and the state and covariance estimates are updated using the sensor readings.
- Localization Method: The localize method, similar to the 2D case, integrates the prediction and update steps. It takes control inputs (linear and angular velocities) and sensor readings to predict the next state and update the estimate.

IV. Neural Network Kalman Filter

Neural Network Kalman Filtering (NNKF) is a fusion of traditional Kalman Filtering techniques with the power of neural networks. It combines the recursive estimation capabilities of Kalman Filters with the ability of neural networks to learn complex mappings and patterns. The main idea is to replace the explicit modeling of the system dynamics and measurement models in Kalman Filters with neural networks, allowing the model to be learned from data. It has multiple steps covering how it works which are initialization, training, prediction (localization), integration with KF concepts, parameter tuning, and iterative process.

A. Initialization

- Initialize the neural network model with appropriate architecture and parameters. This includes defining the input and output dimensions, the number of hidden layers, and the activation functions.

B. Training

- During training, the neural network learns to map sensor readings to possess by adjusting its internal parameters (weights and biases) to minimize the difference between predicted poses and true poses.
- Train the neural network using historical data. The training data consists of pairs of sensor readings and corresponding true poses.

C. Prediction (Localization)

- In the prediction phase, when a new sensor reading is received, the trained neural network is used to predict the corresponding pose.
- The neural network takes the current sensor reading as input and produces an estimated pose as output.

D. Integration with KF Concepts

- NNKF retains the prediction and update steps from the Kalman Filtering framework but replaces the explicit motion and measurement models with neural networks.
- Prediction Step: The neural network predicts the next state (pose) based on the current state and control input. This is similar to the prediction step in the traditional Kalman Filter.
- Update Step: The neural network corrects the predicted state using the actual sensor measurement. This is analogous to the update step in the Kalman Filter.

E. Parameter Tuning

- The neural network's parameters are tuned during training to ensure that it provides accurate predictions. This involves adjusting weights and biases to minimize the difference between predicted and true poses.

F. Iterative Process

- The process is iterative: as new sensor readings become available, the neural network refines its understanding of the system dynamics.

V. Testing Parameters

After developing the EKF, NNKF, and robot simulation, testing began. Each test had a newly, randomly generated movement and readings and went as follows.

- Test with 50 steps while training the neural network at the same time in 2D space.
- Test with 50 steps with an already trained neural network in 2D space.
- Test with 100 steps while training the neural network at the same time in 2D space.
- Test with 100 steps with an already trained neural network in 2D space.
- Test with 1000 steps while training the neural network at the same time in 2D space.
- Test with 1000 steps with an already trained neural network in 2D space.
- Test with 50 steps while training the neural network at the same time in 3D space.
- Test with 50 steps with an already trained neural network in 3D space.
- Test with 100 steps while training the neural network at the same time in 3D space.
- Test with 100 steps with an already trained neural network in 3D space.
- Test with 1000 steps while training the neural network at the same time in 3D space.
- Test with 1000 steps with an already trained neural network in 3D space.

For all tests a plot was made covering EKF estimated pose vs True pose, NNKF estimated pose vs True pose, and EKF estimated pose vs NNKF estimated pose vs True pose. We also calculated the runtime for each filter, a time complexity, and an accuracy based on the mean squared error (MSE)

VI. Testing Results

A. 2D, 50 steps, untrained NN

The result shows a better estimation by the EKF and a better run time as well, this is probably due to the NNKF training the NN simultaneously as it estimates the location. The plot for this will be below.

- EKF Run Time: 0.000002947522 seconds
- NNKF Time: 0.03713560104370117 seconds
- EKF MSE: 0.24907969117725812
- NNKF MSE: 0.4775000837536986

B. 2D, 50 steps, trained NN

The result shows a better estimation by the NNKF, this is probably due to the NNKF being trained beforehand. The run time is still lower in the EKF and the errors seem to be due to unexpected outliers. Because of that I believe the EKF is still a more suitable choice than the NNKF. The plot for this will be below.

- EKF Run Time: 0.0000027563211 seconds
- NNKF Time: 0.0000039223432 seconds
- EKF MSE: 0.39635047554679215
- NNKF MSE: 0.012100267681284171

C. 2D, 100 steps, untrained NN

The result shows a better estimation by the EKF and a better run time as well, this is probably due to the NNKF training the NN simultaneously as it estimates the location. The plot for this will be below.

- EKF Run Time: 0.0000029154621 seconds
- NNKF Time: 0.037645578384399414 seconds
- EKF MSE: 0.9958291697394822
- NNKF MSE: 0.40542374774177165

D. 2D, 100 steps, trained NN

The result shows a better estimation by the NNKF, this is probably due to the NNKF being trained beforehand. The run time is still lower in the EKF and the errors seem to be due to unexpected outliers. Because of that I believe the EKF is still a more suitable choice than the NNKF. The plot for this will be below.

- EKF Run Time: 0.0000029521441 seconds
- NNKF Time: 0.00000741472 seconds
- EKF MSE: 1.0878410230199689
- NNKF MSE: 0.012608717485765501

E. 2D, 1000 steps, untrained NN

The result shows a better estimation by the EKF and a better run time as well, this is probably due to the NNKF training the NN simultaneously as it estimates the location. The plot for this will be below.

- EKF Time: 0.00000265413 seconds
- NNKF Time: 0.0372776985168457 seconds
- EKF MSE: 1.5699209387497033

- NNKF MSE: 4.634478263923316

F. 2D, 1000 steps, trained NN

The result shows a better estimation by the NNKF, this is probably due to the NNKF being trained beforehand. The run time is still lower in the EKF and the errors seem to be due to unexpected outliers. Because of that I believe the EKF is still a more suitable choice than the NNKF. The plot for this will be below.

- EKF Time: 0.0000042236511 seconds
- NNKF Time: 0.00000709772109 seconds
- EKF MSE: 4.168795938669634
- NNKF MSE: 0.01148896712423769

G. 3D, 50 steps, untrained NN

They both display a very poor estimation of the position. The NNKF is so busy training the NN that it cannot accurately predict the pose of the robot. The EKF is giving better results yet they are still unsatisfactory. The EKF will need more fine tuning to handle 3D environments. The plot for this will be below.

- EKF Time: 0.000245368741 seconds
- NNKF Time: 0.000050313711166381836 seconds
- EKF MSE: 11.212824624626336
- NNKF MSE: 0.7131749656375463

H. 3D, 50 steps, trained NN

Now that the NNKF does not need to train the NN, its run time is lower and it has a very good estimation that makes it a more logical choice. The EKF is still not tuned well enough and the runtime although lower, the estimation makes it a bad choice.

- EKF Time: 0.0009984970092773438 seconds
- NNKF Time: 0.00040404558181762695 seconds
- EKF MSE: 22.994592158726125
- NNKF MSE: 0.013013183409367293

I. 3D, 100 steps, untrained NN

They both display a very poor estimation of the position. The NNKF is so busy training the NN that it cannot accurately predict the pose of the robot. The EKF is giving better results yet they are still unsatisfactory. The EKF will need more fine tuning to handle 3D environments. The plot for this will be below.

- EKF Time: 0.000375486632154 seconds
- NNKF Time: 0.00007928848266601562 seconds
- EKF MSE: 25.138654839044435
- NNKF MSE: 0.771816823540115

J. 3D, 100 steps, trained NN

Now that the NNKF does not need to train the NN, its run time is lower and it has a very good estimation that makes it a more logical choice. The EKF is still not tuned well enough and the runtime although lower, the estimation although better than before still makes it a bad choice.

- EKF Time: 0.00084215487444123 seconds
- NNKF Time: 0.0007612395286560059 seconds
- EKF MSE: 2.9785704554555497
- NNKF MSE: 0.011449158068948842

K. 3D, 1000 steps, untrained NN

They both display a very poor estimation of the position. The NNKF is so busy training the NN that it cannot accurately predict the pose of the robot. The EKF is giving better results yet they are still unsatisfactory. The EKF will need more fine tuning to handle 3D environments. The plot for this will be below.

- EKF Time: 0.000542658745125 seconds
- NNKF Time: 0.00003961825370788574 seconds
- EKF MSE: 70.32156297316041
- NNKF MSE: 2.794905709671381

L. 3D, 1000 steps, trained NN

Now that the NNKF does not need to train the NN, its run time is lower and it has a very good estimation that makes it a more logical choice. The EKF is still not tuned well enough and the runtime although lower, the estimation makes it a bad choice.

- EKF Time: 0.0014556232354 seconds
- NNKF Time: 0.000385136604309082 seconds
- EKF MSE: 154.45381677500902
- NNKF MSE: 0.010929531227913958

Even with A very high number of steps (1000) the EKF seems to be the more logical choice as the run time is still way lower even though the NNKF is more accurate (in 2D). It would seem that the more steps the robot takes the more the accuracy of the EKF drops in comparison to the NNKF yet a test with far more steps should be implemented to further explore the concept.

Below are screenshots of the code and the result after running the code. Please follow the comments for optimal output and note that the robot , sensors, and movement are all simulated. The order of the screenshots is Python code followed by Plots of the tests in the order mentioned in ***V. Testing Parameters*** and ***VI. Testing Results***. The code also has multiple ways to show outputs ranging from just the simulated robot and movement to just the estimated poses in each filter to plotting them for comparison and measuring runtime and accuracy.

VII. Code Composition

The code is composed of two folders one titled 2D and the other 3D each one contains 4 files titled Robot.py, ekf_localizer.py, nnkf_localizer.py, and main.py.

A. 2D folder

- Robot.py: Simulates the robot, its movement, sensor measurements, and true pose in 2D space.
- Ekf_localizer.py: Uses the data provided from Robot.py excluding the true pose to estimate the position of the robot in 2D space using an extended kalman filter.
- Nnkf_localizer.py: Uses data provided from Robot.py excluding the true pose to estimate the position of the robot in 2D and train a neural network connecting the sensor readings with estimated pose and and another connecting motion with estimated pose.
- Main.py: Calls the other functions from Robot.py, ekf_localizer.py, and nnkf_localizer to get true and estimated poses to be plotted.

B. 3D folder

- Robot.py: Simulates the robot, its movement, sensor measurements, and true pose in 3D space.
- Ekf_localizer.py: Uses the data provided from Robot.py excluding the true pose to estimate the position of the robot in 3D space using an extended kalman filter.
- Nnkf_localizer.py: Uses data provided from Robot.py excluding the true pose to estimate the position of the robot in 3D and train a neural network connecting the sensor readings with estimated pose and and another connecting motion with estimated pose.
- Main.py: Calls the other functions from Robot.py, ekf_localizer.py, and nnkf_localizer to get true and estimated poses to be plotted.

Submission will include a zip file with the code included for your reference.

VIII. Conclusion

After testing and implementations I came to the realization that while extended kalman filters are very useful and versatile they fall short when more dimensions are added with more steps taken by the robot in comparison to the neural network kalman filter. Training the NN beforehand helps significantly reduce the run time of each step predicted almost by a factor of 10. (from 15ms to 0.13ms). Even in a 2D case a pre-trained NNKF even though it took more time was more accurate than a regular EKF yet the difference in this case is not significant enough to justify the use of NNKF over EKF.

- 2D EKF time Complexity: O(num_steps)
- 3D EKF time Complexity: O(num_steps)
- 2D NNKF Untrained time Complexity: O(num_steps * epochs * num_samples * num_neurons)
- 3D NNKF Untrained time Complexity: O(num_steps * epochs * num_samples * num_neurons)
- 2D NNKF Trained time Complexity: O(num_steps)
- 3D NNKF Trained time Complexity: O(num_steps)

IX. References

1. Ding, Y. R.; Liu, Y. C.; Hsiao, F. B. The Application of Extended Kalman Filtering to Autonomous Formation Flight of Small Uav System. International Journal of Intelligent Unmanned Systems 2013, 1 (2), 154–186 DOI: 10.1108/20496421311330074.
2. Kumar, K.; Bhaumik, S.; Date, P. Extended Kalman Filter Using Orthogonal Polynomials. Ieee Access 2021, 9 DOI: 10.1109/ACCESS.2021.3073289.
3. Arjas, A.; Alles, E. J.; Maneas, E.; Arridge, S.; Desjardins, A.; Sillanpaa, M. J.; Hauptmann, A. Neural Network Kalman Filtering for 3-D Object Tracking from Linear Array Ultrasound Data. Ieee Transactions on Ultrasonics, Ferroelectrics, and Frequency Control 2022, 69 (5) DOI: 10.1109/TUFFC.2022.3162097.
4. Revach, G.; Shlezinger, N.; Ni, X.; Escoriza, A. L.; van Sloun, R. J. G.; Eldar, Y. C. Kalmannet: Neural Network Aided Kalman Filtering for Partially Known Dynamics. Ieee Transactions on Signal Processing 2022, 70 DOI: 10.1109/TSP.2022.3158588.
5. Lin, D.; Wu, Y.-ming. Tracing and Implementation of Imm Kalman Filtering Feed-Forward Compensation Technology Based on Neural Network. Optik - International Journal for Light and Electron Optics 2020, 202 DOI: 10.1016/j.ijleo.2019.163574.

Appendix A: Python Code

Robot.py (2D)

Robot.py

2D > Robot.py > ...

Click here to ask Blackbox to help you code faster

```python
import time
import numpy as np
from scipy.stats import norm

# Simulation parameters
dt = 0.1  # Time step
num_steps = 100  # Number of steps

# Robot parameters
initial_pose = np.array([0.0, 0.0, 0.0])  # Initial pose [x, y, theta]
motion_noise = 0.2  # Motion noise
measurement_noise = 0.01  # Measurement noise

# Function to simulate robot motion
def move_robot(pose, u):
    x, y, theta = pose
    v, omega = u
    delta_theta = omega * dt
    new_pose = np.array([
        x + v * np.cos(theta) * dt,
        y + v * np.sin(theta) * dt,
        theta + delta_theta
    ])
    # Add motion noise
    new_pose += np.random.normal(0, motion_noise, 3)
    return new_pose

# Function to generate sensor readings
def generate_sensor_reading(true_pose):
    x, y, theta = true_pose
    # Simulate measurements with noise
    range_measurement = np.sqrt(x**2 + y**2) + np.random.normal(0, measurement_noise)
    bearing_measurement = np.arctan2(y, x) + np.random.normal(0, measurement_noise)
    theta_measurement = theta + np.random.normal(0, measurement_noise)
    return np.array([range_measurement, bearing_measurement, theta_measurement])

# Simulate robot motion and generate sensor readings
true_poses = []
sensor_readings = []

current_pose = initial_pose
for step in range(num_steps):
    # Simulate control input (linear velocity and angular velocity)
    control_input = np.array([0.1, 0.05])

```

```python
# Function to generate sensor readings
def generate_sensor_reading(true_pose):
    x, y, theta = true_pose
    # Simulate measurements with noise
    range_measurement = np.sqrt(x**2 + y**2) + np.random.normal(0, measurement_noise)
    bearing_measurement = np.arctan2(y, x) + np.random.normal(0, measurement_noise)
    theta_measurement = theta + np.random.normal(0, measurement_noise)
    return np.array([range_measurement, bearing_measurement, theta_measurement])

# Simulate robot motion and generate sensor readings
true_poses = []
sensor_readings = []

current_pose = initial_pose
for step in range(num_steps):
    # Simulate control input (linear velocity and angular velocity)
    control_input = np.array([0.1, 0.05])

    # Simulate robot motion
    current_pose = move_robot(current_pose, control_input)

    # Generate sensor reading
    sensor_reading = generate_sensor_reading(current_pose)

    # Save true pose and sensor reading
    true_poses.append(current_pose)
    sensor_readings.append(sensor_reading)

    # Print the current pose and sensor reading with timestamp
    timestamp = time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())
    print(f"Timestamp: {timestamp}, Step: {step + 1}, True Pose: {current_pose}, Sensor Reading: {sensor_reading}")
```

Ekf_localizer.py (2D)

ekf_localizer.py

2D > ekf_localizer.py > EKFLocalizer > __init__

Click here to ask Blackbox to help you code faster

```python
import numpy as np
from scipy.linalg import block_diag
from Robot import generate_sensor_reading, move_robot

class EKFLocalizer:
    def __init__(self, initial_pose, motion_noise, measurement_noise):
        self.mu = initial_pose  # Initial estimate of the robot pose [x, y, theta]
        self.sigma = np.zeros((3, 3))  # Initial covariance matrix
        self.motion_noise = motion_noise
        self.measurement_noise = measurement_noise

    def predict(self, u, dt):
        # Predict the next state using the motion model (constant velocity)
        v, omega = u
        delta_theta = omega * dt

        # Jacobian of the motion model
        G = np.array([
            [1, 0, -v * np.sin(self.mu[2]) * dt],
            [0, 1, v * np.cos(self.mu[2]) * dt],
            [0, 0, 1]
        ])

        # Motion model
        motion_model = np.array([
            v * np.cos(self.mu[2]) * dt,
            v * np.sin(self.mu[2]) * dt,
            delta_theta
        ])

        # Update the state and covariance prediction
        self.mu = self.mu + motion_model
        self.sigma = G @ self.sigma @ G.T + np.diag([self.motion_noise, self.motion_noise, 0])

    def update(self, z):
        # Measurement model
        h = np.array([
            np.sqrt(self.mu[0]**2 + self.mu[1]**2),
            np.arctan2(self.mu[1], self.mu[0]),
            self.mu[2]
        ])

```

```python
        # Jacobian of the measurement model
        H = np.array([
            [self.mu[0] / np.sqrt(self.mu[0]**2 + self.mu[1]**2), self.mu[1] / np.sqrt(self.mu[0]**2 + self.mu[1]**2), 0],
            [-self.mu[1] / (self.mu[0]**2 + self.mu[1]**2), self.mu[0] / (self.mu[0]**2 + self.mu[1]**2), 0],
            [0, 0, 1]
        ])

        # Measurement noise
        R = np.diag([self.measurement_noise, self.measurement_noise, self.measurement_noise])

        # Kalman gain
        K = self.sigma @ H.T @ np.linalg.inv(H @ self.sigma @ H.T + R)

        # Update the state and covariance estimate
        self.mu = self.mu + K @ (z - h)
        self.sigma = (np.eye(3) - K @ H) @ self.sigma

    # Modify the EKF localization method in ekf_localizer.py
    def localize(self, control_input, measurement, true_poses):
        dt = 0.1  # Time step (assuming the same time step as the motion model)

        # Prediction step
        self.predict(control_input, dt)

        # Update step
        self.update(measurement)

        # Return the estimated pose and true pose
        return self.mu, true_poses[-1]


if __name__ == "__main__":
    # Import necessary modules
    import time
    import numpy as np
    from scipy.stats import norm

    # Simulation parameters
    dt = 0.1  # Time step
    num_steps = 10000  # Number of steps

    # Robot parameters
    initial_pose = np.array([0.0, 0.0, 0.0])  # Initial pose [x, y, theta]
```

```python
    # Robot parameters
    initial_pose = np.array([0.0, 0.0, 0.0])  # Initial pose [x, y, theta]
    motion_noise = 0.2  # Motion noise
    measurement_noise = 0.01  # Measurement noise

    # Create an instance of the EKF localizer
    ekf_localizer = EKFLocalizer(initial_pose, motion_noise, measurement_noise)

    # Simulate robot motion and generate sensor readings
    true_poses = []
    sensor_readings = []
    estimated_poses = []

    current_pose = initial_pose
    for step in range(num_steps):
        # Simulate control input (linear velocity and angular velocity)
        control_input = np.array([0.1, 0.05])

        # Simulate robot motion
        current_pose = move_robot(current_pose, control_input)

        # Generate sensor reading
        sensor_reading = generate_sensor_reading(current_pose)

        # Save true pose and sensor reading
        true_poses.append(current_pose)
        sensor_readings.append(sensor_reading)

        # EKF localization
        estimated_pose, true_pose = ekf_localizer.localize(control_input, sensor_reading)
        estimated_poses.append(estimated_pose)

        # Print the current pose, sensor reading, estimated pose, and true pose with timestamp
        timestamp = time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())
        print(f"Timestamp: {timestamp}, Step: {step + 1}, True Pose: {current_pose}, "
              f"Sensor Reading: {sensor_reading}, Estimated Pose: {estimated_pose}")

    # Print the final estimated and true poses
    print("\nFinal Estimated Pose:", estimated_poses[-1])
    print("Final True Pose:", true_poses[-1])
```

Nnkf_localizer.py (2D)

nnkf_localizer.py 3

2D > nnkf_localizer.py > NNKFLocalizer > __init__

Click here to ask Blackbox to help you code faster

```python
import numpy as np
from Robot import generate_sensor_reading, move_robot
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from sklearn.preprocessing import StandardScaler

class NNKFLocalizer:
    def __init__(self, input_dim=3):
        # Define the neural network model
        self.model = self.build_model(input_dim)
        self.scaler = StandardScaler()

    def build_model(self, input_dim):
        model = Sequential([
            Dense(64, input_dim=input_dim, activation='relu'),
            Dropout(0.2),  # Adding dropout for regularization
            Dense(32, activation='relu'),
            Dense(3, activation='linear')
        ])
        optimizer = Adam(learning_rate=0.1)  # Adjust the learning rate if needed
        model.compile(optimizer=optimizer, loss='mean_squared_error')
        return model

    def train_model(self, input_data, target_data, epochs=10):
        input_data_scaled = self.scaler.fit_transform(input_data)
        self.model.fit(input_data_scaled, target_data, epochs=epochs, verbose=0)

    def localize(self, sensor_reading):
        scaled_sensor_reading = self.scaler.transform(np.array([sensor_reading]))
        # Predict the pose using the trained neural network
        estimated_pose = self.model.predict(scaled_sensor_reading)[0]
        return estimated_pose

if __name__ == "__main__":
    import time

    # Simulation parameters
    dt = 0.1  # Time step
    num_steps = 10000  # Number of steps

    # Robot parameters
    initial_pose = np.array([0.0, 0.0, 0.0])  # Initial pose [x, y, theta]
    motion_noise = 0.2  # Motion noise
    measurement_noise = 0.01  # Measurement noise

```

```python
    # Robot parameters
    initial_pose = np.array([0.0, 0.0, 0.0])  # Initial pose [x, y, theta]
    motion_noise = 0.2  # Motion noise
    measurement_noise = 0.01  # Measurement noise

    # Create an instance of the NNKF localizer
    nnkf_localizer = NNKFLocalizer(input_dim=3)

    # Simulate robot motion and generate sensor readings
    true_poses = []
    sensor_readings = []
    estimated_poses = []

    current_pose = initial_pose
    for step in range(num_steps):
        # Simulate control input (linear velocity and angular velocity)
        control_input = np.array([0.1, 0.05])

        # Simulate robot motion
        current_pose = move_robot(current_pose, control_input)

        # Generate sensor reading
        sensor_reading = generate_sensor_reading(current_pose)

        # Save true pose and sensor reading
        true_poses.append(current_pose)
        sensor_readings.append(sensor_reading)

        # NNKF localization
        nnkf_localizer.train_model(np.array(sensor_readings), np.array(true_poses), epochs=10)
        estimated_pose = nnkf_localizer.localize(sensor_reading)
        estimated_poses.append(estimated_pose)

        # Print the current pose, sensor reading, estimated pose, and true pose with timestamp
        timestamp = time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())
        print(f"Timestamp: {timestamp}, Step: {step + 1}, True Pose: {current_pose}, "
              f"Sensor Reading: {sensor_reading}, Estimated Pose: {estimated_pose}")

    # Print the final estimated and true poses
    print("\nFinal Estimated Pose:", estimated_poses[-1])
    print("Final True Pose:", true_poses[-1])
```

Main.py (2D)

main.py 2

2D > main.py > run_simulation

Click here to ask Blackbox to help you code faster

```python
import time
import numpy as np
import matplotlib.pyplot as plt
from Robot import generate_sensor_reading, move_robot
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.preprocessing import StandardScaler
from scipy.linalg import block_diag
from ekf_localizer import EKFLocalizer
from nnkf_localizer import NNKFLocalizer
from sklearn.metrics import mean_squared_error

def run_simulation(num_steps):
    # Robot parameters
    initial_pose = np.array([0.0, 0.0, 0.0])
    motion_noise = 0.2
    measurement_noise = 0.01

    # Create instances of the Localizers
    ekf_localizer = EKFLocalizer(initial_pose, motion_noise, measurement_noise)
    nnkf_localizer = NNKFLocalizer()

    # Lists to store results
    true_poses = []
    ekf_estimated_poses = []
    nnkf_estimated_poses = []

    # Simulate robot motion and generate sensor readings
    current_pose = initial_pose
    for step in range(num_steps):
        control_input = np.array([0.1, 0.05])

        # Simulate robot motion
        current_pose = move_robot(current_pose, control_input)

        # Generate sensor reading
        sensor_reading = generate_sensor_reading(current_pose)

        # Save true pose
        true_poses.append(current_pose)

        # EKF Localization
        start_time = time.time()
        estimated_pose, _ = ekf_localizer.localize(control_input, sensor_reading, true_poses)
        ekf_estimated_poses.append(estimated_pose)
        ekf_time = time.time() - start_time
```

```
        # NNKF localization
        # Ensure the scaler is fitted before calling localize
        nnkf_localizer.train_model(np.array([sensor_reading]), np.array([current_pose]), epochs=10)
        start_time = time.time()
        estimated_pose = nnkf_localizer.localize(sensor_reading)
        nnkf_estimated_poses.append(estimated_pose)
        nnkf_time = time.time() - start_time

    # Calculate accuracy using Mean Squared Error (MSE)
    mse_ekf = mean_squared_error(true_poses, ekf_estimated_poses)
    mse_nnkf = mean_squared_error(true_poses, nnkf_estimated_poses)

    return true_poses, ekf_estimated_poses, nnkf_estimated_poses, ekf_time, nnkf_time, mse_ekf, mse_nnkf

def plot_results(true_poses, ekf_estimated_poses, nnkf_estimated_poses):
    # Plot EKF estimate vs true pose
    plt.figure(figsize=(12, 4))
    plt.subplot(131)
    true_poses_array = np.array(true_poses)
    ekf_estimated_poses_array = np.array(ekf_estimated_poses)
    plt.plot(true_poses_array[:, 0], label='True X')
    plt.plot(ekf_estimated_poses_array[:, 0], label='EKF Estimated X')
    plt.legend()
    plt.title('EKF Estimate vs True Pose (X-axis)')

    # Plot NNKF estimate vs true pose
    plt.subplot(132)
    nnkf_estimated_poses_array = np.array(nnkf_estimated_poses)
    plt.plot(true_poses_array[:, 0], label='True X')
    plt.plot(nnkf_estimated_poses_array[:, 0], label='NNKF Estimated X')
    plt.legend()
    plt.title('NNKF Estimate vs True Pose (X-axis)')

    # Plot EKF estimate vs NNKF estimate vs true pose
    plt.subplot(133)
    plt.plot(true_poses_array[:, 0], label='True X')
    plt.plot(ekf_estimated_poses_array[:, 0], label='EKF Estimated X')
    plt.plot(nnkf_estimated_poses_array[:, 0], label='NNKF Estimated X')
    plt.legend()
    plt.title('EKF vs NNKF vs True Pose (X-axis)')

    plt.tight_layout()
    plt.show()
```

```
def print_results(ekf_time, nnkf_time, mse_ekf, mse_nnkf):
    print("\nResults:")
    print(f"EKF Time: {ekf_time} seconds")
    print(f"NNKF Time: {nnkf_time} seconds")

    print(f"\nEKF MSE: {mse_ekf}")
    print(f"NNKF MSE: {mse_nnkf}")

if __name__ == "__main__":
    num_steps = int(input("Enter the number of steps for simulation: "))

    true_poses, ekf_estimated_poses, nnkf_estimated_poses, ekf_time, nnkf_time, mse_ekf, mse_nnkf = run_simulation(num_steps)

    # Plot the results
    plot_results(true_poses, ekf_estimated_poses, nnkf_estimated_poses)

    # Print the results
    print_results(ekf_time, nnkf_time, mse_ekf, mse_nnkf)
```

Robot.py (3D)

Robot.py

3D > Robot.py > move_robot

Click here to ask Blackbox to help you code faster

```python
import time
import numpy as np
from scipy.stats import norm

# Simulation parameters
dt = 0.1  # Time step
num_steps = 100  # Number of steps

# Robot parameters
initial_pose = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])  # Initial pose [x, y, z, roll, pitch, yaw]
motion_noise = 0.2  # Motion noise
measurement_noise = 0.01  # Measurement noise

def move_robot(pose, u):
    x, y, z, roll, pitch, yaw = pose
    v, omega_x, omega_y, omega_z = u[:4]  # Take the first four values from the control input

    # Update roll, pitch, and yaw
    delta_roll = omega_x * dt
    delta_pitch = omega_y * dt
    delta_yaw = omega_z * dt

    # Update position
    new_pose = np.array([
        x + v * np.cos(yaw) * dt,
        y + v * np.sin(yaw) * dt,
        z,
        roll + delta_roll,
        pitch + delta_pitch,
        yaw + delta_yaw
    ])

    # Add motion noise
    new_pose += np.random.normal(0, motion_noise, 6)

    return new_pose


def generate_sensor_reading_3d(true_pose):
    x, y, z, _, _, yaw = true_pose
    # Simulate measurements with noise
    range_measurement = np.sqrt(x**2 + y**2 + z**2) + np.random.normal(0, measurement_noise)
    bearing_measurement = np.arctan2(y, x) + np.random.normal(0, measurement_noise)
    pitch_measurement = np.arcsin(z / np.sqrt(x**2 + y**2 + z**2)) + np.random.normal(0, measurement_noise)
    yaw_measurement = yaw + np.random.normal(0, measurement_noise)
    return np.array([range_measurement, bearing_measurement, pitch_measurement, 0.0, 0.0, yaw_measurement])

# Simulate robot motion and generate sensor readings
true_poses = []
sensor_readings = []

current_pose = initial_pose
for step in range(num_steps):
    # Simulate control input (linear velocity, and angular velocities around x, y, and z axes)
    control_input = np.array([0.1, 0.05, 0.02, 0.01])

    # Simulate robot motion
    current_pose = move_robot(current_pose, control_input)

    # Generate sensor reading
    sensor_reading = generate_sensor_reading_3d(current_pose)

    # Save true pose and sensor reading
    true_poses.append(current_pose)
    sensor_readings.append(sensor_reading)

    # Print the current pose and sensor reading with timestamp
    timestamp = time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())
    print(f"Timestamp: {timestamp}, Step: {step + 1}, True Pose: {current_pose}, Sensor Reading: {sensor_reading}")
```

Ekf_localizer.py (3D)

ekf_localizer.py

3D > ekf_localizer.py > EKFLocalizer > predict

Click here to ask Blackbox to help you code faster

```python
import numpy as np
from scipy.linalg import block_diag
from Robot import generate_sensor_reading_3d, move_robot

class EKFLocalizer:
    def __init__(self, initial_pose, motion_noise, measurement_noise):
        self.mu = initial_pose  # Initial estimate of the robot pose [x, y, z, roll, pitch, yaw]
        self.sigma = np.zeros((6, 6))  # Initial covariance matrix
        self.motion_noise = motion_noise
        self.measurement_noise = measurement_noise

    def predict(self, u, dt):
        # Predict the next state using the motion model (constant velocity)
        v, omega_x, omega_y, omega_z = u[:4]

        # Jacobian of the motion model
        G = np.array([
            [1, 0, 0, -v * np.sin(self.mu[5]) * np.cos(self.mu[4]) * dt, v * np.cos(self.mu[5]) * np.cos(self.mu[4]) * dt, 0],
            [0, 1, 0, v * np.cos(self.mu[5]) * np.cos(self.mu[4]) * dt, v * np.sin(self.mu[5]) * np.cos(self.mu[4]) * dt, 0],
            [0, 0, 1, -v * np.sin(self.mu[4]) * dt, 0, v * np.cos(self.mu[4]) * dt],
            [0, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 0, 1]
        ])

        # Motion model
        motion_model = np.array([
            v * np.cos(self.mu[5]) * np.cos(self.mu[4]) * dt,
            v * np.sin(self.mu[5]) * np.cos(self.mu[4]) * dt,
            v * np.sin(self.mu[4]) * dt,
            omega_x * dt,
            omega_y * dt,
            omega_z * dt
        ])

        # Update the state and covariance prediction
        self.mu = self.mu + motion_model
        self.sigma = G @ self.sigma @ G.T + np.diag([self.motion_noise, self.motion_noise, self.motion_noise, 0, 0, 0])

    def update(self, z):
        # Measurement model
        h = np.array([
            np.sqrt(self.mu[0]**2 + self.mu[1]**2 + self.mu[2]**2),
            np.arctan2(self.mu[1], self.mu[0]),
            np.arcsin(self.mu[2] / np.sqrt(self.mu[0]**2 + self.mu[1]**2 + self.mu[2]**2)),
```

```
    def update(self, z):
        # Measurement model
        h = np.array([
            np.sqrt(self.mu[0]**2 + self.mu[1]**2 + self.mu[2]**2),
            np.arctan2(self.mu[1], self.mu[0]),
            np.arcsin(self.mu[2] / np.sqrt(self.mu[0]**2 + self.mu[1]**2 + self.mu[2]**2)),
            self.mu[3],
            self.mu[4],
            self.mu[5]
])


        # Jacobian of the measurement model
        H = np.array([
            [self.mu[0] / np.sqrt(self.mu[0]**2 + self.mu[1]**2 + self.mu[2]**2),
             self.mu[1] / np.sqrt(self.mu[0]**2 + self.mu[1]**2 + self.mu[2]**2),
             self.mu[2] / np.sqrt(self.mu[0]**2 + self.mu[1]**2 + self.mu[2]**2), 0, 0, 0],
            [-self.mu[1] / (self.mu[0]**2 + self.mu[1]**2), self.mu[0] / (self.mu[0]**2 + self.mu[1]**2), 0, 0, 0, 0],
            [self.mu[0] * self.mu[2] / (self.mu[0]**2 + self.mu[1]**2 + self.mu[2]**2)**(3/2),
             self.mu[1] * self.mu[2] / (self.mu[0]**2 + self.mu[1]**2 + self.mu[2]**2)**(3/2),
             self.mu[2] / np.sqrt(self.mu[0]**2 + self.mu[1]**2 + self.mu[2]**2), 0, 0, 0],
            [0, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 0, 1]
        ])

        # Measurement noise
        R = np.diag([self.measurement_noise, self.measurement_noise, self.measurement_noise,
                     self.measurement_noise, self.measurement_noise, self.measurement_noise])

        # Kalman gain
        K = self.sigma @ H.T @ np.linalg.inv(H @ self.sigma @ H.T + R)

        # Update the state and covariance estimate
        self.mu = self.mu + K @ (z - h)
        self.sigma = (np.eye(6) - K @ H) @ self.sigma

    # Modify the EKF localization method
    def localize(self, control_input, measurement, true_poses):
        dt = 0.1  # Time step (assuming the same time step as the motion model)

        # Prediction step
        self.predict(control_input, dt)

        # Update step
        self.update(measurement)
```

```
        # Update step
        self.update(measurement)

        # Return the estimated pose and true pose
        return self.mu, true_poses[-1]



if __name__ == "__main__":
    # Import necessary modules
    import time
    import numpy as np
    from scipy.stats import norm

    # Simulation parameters
    dt = 0.1  # Time step
    num_steps = 100  # Number of steps

    # Robot parameters
    initial_pose = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])  # Initial pose [x, y, z, roll, pitch, yaw]
    motion_noise = 0.2  # Motion noise
    measurement_noise = 0.01  # Measurement noise

    # Create an instance of the EKF localizer
    ekf_localizer = EKFLocalizer(initial_pose, motion_noise, measurement_noise)

    # Simulate robot motion and generate sensor readings
    true_poses = []
    sensor_readings = []
    estimated_poses = []

    current_pose = initial_pose
    for step in range(num_steps):
        # Simulate control input (linear velocity, and angular velocities around x, y, and z axes)
        control_input = np.array([0.1, 0.05, 0.02, 0.01])

        # Simulate robot motion
        current_pose = move_robot(current_pose, control_input)

        # Generate sensor reading
        sensor_reading = generate_sensor_reading_3d(current_pose)

        # Save true pose and sensor reading
        true_poses.append(current_pose)
        sensor_readings.append(sensor_reading)
```

```
        # EKF localization
        estimated_pose, true_pose = ekf_localizer.localize(control_input, sensor_reading, true_poses)
        estimated_poses.append(estimated_pose)

        # Print the current pose, sensor reading, estimated pose, and true pose with timestamp
        timestamp = time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())
        print(f"Timestamp: {timestamp}, Step: {step + 1}, True Pose: {current_pose}, "
              f"Sensor Reading: {sensor_reading}, Estimated Pose: {estimated_pose}")

    # Print the final estimated and true poses
    print("\nFinal Estimated Pose:", estimated_poses[-1])
    print("Final True Pose:", true_poses[-1])
```

Nnkf_localizer.py (3D)

nnkf_localizer.py 3

3D > nnkf_localizer.py > NNKFLocalizer > build_model

Click here to ask Blackbox to help you code faster

```python
import numpy as np
from Robot import generate_sensor_reading_3d, move_robot
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from sklearn.preprocessing import StandardScaler

class NNKFLocalizer:
    def __init__(self, input_dim=6):  # Update input_dim to 6 for a 3D robot
        # Define the neural network model
        self.model = self.build_model(input_dim)
        self.scaler = StandardScaler()

    def build_model(self, input_dim):
        model = Sequential([
            Dense(64, input_dim=input_dim, activation='relu'),
            Dropout(0.2),  # Adding dropout for regularization
            Dense(32, activation='relu'),
            Dense(6, activation='linear')  # Output dimension is 6 for a 3D robot
        ])
        optimizer = Adam(learning_rate=0.1)  # Adjust the learning rate if needed
        model.compile(optimizer=optimizer, loss='mean_squared_error')
        return model

    def train_model(self, input_data, target_data, epochs=10):
        input_data_scaled = self.scaler.fit_transform(input_data)
        self.model.fit(input_data_scaled, target_data, epochs=epochs, verbose=0)

    def localize(self, sensor_reading):
        scaled_sensor_reading = self.scaler.transform(np.array([sensor_reading]))
        # Predict the pose using the trained neural network
        estimated_pose = self.model.predict(scaled_sensor_reading)[0]
        return estimated_pose

if __name__ == "__main__":
    import time

    # Simulation parameters
    dt = 0.1  # Time step
    num_steps = 100  # Number of steps

    # Robot parameters
    initial_pose = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])  # Initial pose [x, y, z, roll, pitch, yaw]
    motion_noise = 0.2  # Motion noise
    measurement_noise = 0.01  # Measurement noise
```

```python
    # Create an instance of the NNKF localizer
    nnkf_localizer = NNKFLocalizer(input_dim=6)  # Update input_dim to 6 for a 3D robot

    # Simulate robot motion and generate sensor readings
    true_poses = []
    sensor_readings = []
    estimated_poses = []

    current_pose = initial_pose
    for step in range(num_steps):
        # Simulate control input (linear velocity, and angular velocities around x, y, and z axes)
        control_input = np.array([0.1, 0.05, 0.02, 0.01])

        # Simulate robot motion
        current_pose = move_robot(current_pose, control_input)

        # Generate sensor reading
        sensor_reading = generate_sensor_reading_3d(current_pose)

        # Save true pose and sensor reading
        true_poses.append(current_pose)
        sensor_readings.append(sensor_reading)

        # NNKF localization
        nnkf_localizer.train_model(np.array(sensor_readings), np.array(true_poses), epochs=10)
        estimated_pose = nnkf_localizer.localize(sensor_reading)
        estimated_poses.append(estimated_pose)

        # Print the current pose, sensor reading, estimated pose, and true pose with timestamp
        timestamp = time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())
        print(f"Timestamp: {timestamp}, Step: {step + 1}, True Pose: {current_pose}, "
              f"Sensor Reading: {sensor_reading}, Estimated Pose: {estimated_pose}")

    # Print the final estimated and true poses
    print("\nFinal Estimated Pose:", estimated_poses[-1])
    print("Final True Pose:", true_poses[-1])
```

Main.py (3D)

main.py 2

3D > main.py > ...

Click here to ask Blackbox to help you code faster

```python
import time
import numpy as np
import matplotlib.pyplot as plt
from Robot import generate_sensor_reading_3d, move_robot
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.preprocessing import StandardScaler
from scipy.linalg import block_diag
from ekf_localizer import EKFLocalizer
from nnkf_localizer import NNKFLocalizer
from sklearn.metrics import mean_squared_error

def run_simulation(num_steps):
    # Robot parameters
    initial_pose = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0])  # Initial pose [x, y, z, roll, pitch, yaw]
    motion_noise = 0.2
    measurement_noise = 0.01

    # Create instances of the localizers
    ekf_localizer = EKFLocalizer(initial_pose, motion_noise, measurement_noise)
    nnkf_localizer = NNKFLocalizer(input_dim=6)  # Update input_dim to 6 for a 3D robot

    # Lists to store results
    true_poses = []
    ekf_estimated_poses = []
    nnkf_estimated_poses = []

    # Simulate robot motion and generate sensor readings
    current_pose = initial_pose
    for step in range(num_steps):
        # Simulate control input (linear velocity, and angular velocities around x, y, and z axes)
        control_input = np.array([0.1, 0.05, 0.02, 0.01, 0.005, 0.002])

        # Simulate robot motion
        current_pose = move_robot(current_pose, control_input)

        # Generate sensor reading
        sensor_reading = generate_sensor_reading_3d(current_pose)

        # Save true pose
        true_poses.append(current_pose)
```

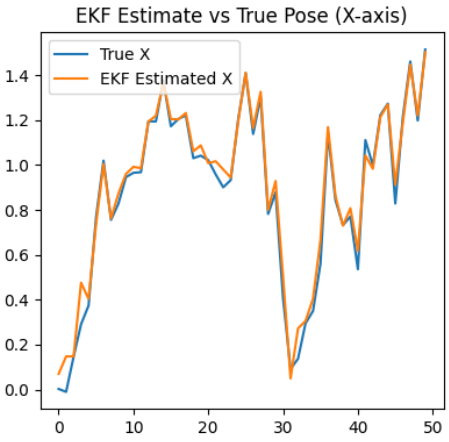
```python
        # EKF localization
        start_time = time.ti
        estimated_pose, _ = ekf_localizer.localize(control_input, sensor_reading, true_poses)
        ekf_estimated_poses.append(estimated_pose)
        ekf_time = time.time() - start_time

        # NNKF localization
        # Ensure the scaler is fitted before calling localize
        nnkf_localizer.train_model(np.array([sensor_reading]), np.array([current_pose]), epochs=10)
        start_time = time.time()
        estimated_pose_nn = nnkf_localizer.localize(sensor_reading)
        nnkf_estimated_poses.append(estimated_pose_nn)
        nnkf_time = time.time() - start_time

    # Calculate accuracy using Mean Squared Error (MSE)
    mse_ekf = mean_squared_error(true_poses, ekf_estimated_poses)
    mse_nnkf = mean_squared_error(true_poses, nnkf_estimated_poses)

    return true_poses, ekf_estimated_poses, nnkf_estimated_poses, ekf_time, nnkf_time, mse_ekf, mse_nnkf
```

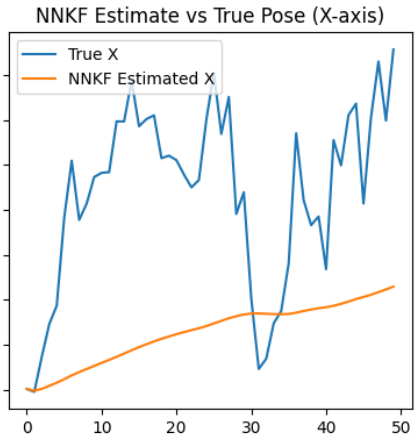
(variable) ekf_localizer: EKFLocalizer

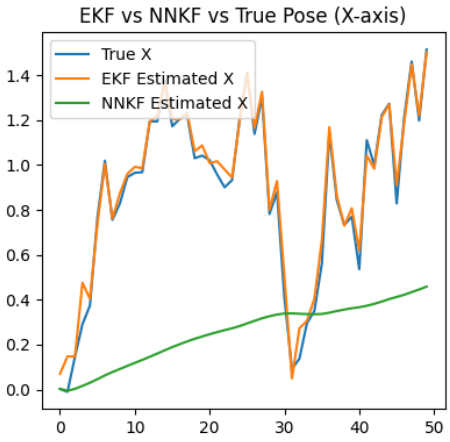
```python
if __name__ == "__main__":
    num_steps = int(input("Enter the number of steps for simulation: "))

    true_poses, ekf_estimated_poses, nnkf_estimated_poses, ekf_time, nnkf_time, mse_ekf, mse_nnkf = run_simulation(num_steps)

    # Plot the results
    plt.figure(figsize=(18, 6))
    plt.subplot(131)
    plt.title('EKF Estimate vs True Pose (X-axis)')
    plt.plot(np.array(true_poses)[:, 0], label='True X')
    plt.plot(np.array(ekf_estimated_poses)[:, 0], label='EKF Estimated X')
    plt.legend()

    plt.subplot(132)
    plt.title('NNKF Estimate vs True Pose (X-axis)')
    plt.plot(np.array(true_poses)[:, 0], label='True X')
    plt.plot(np.array(nnkf_estimated_poses)[:, 0], label='NNKF Estimated X')
    plt.legend()

    plt.subplot(133)
    plt.title('EKF vs NNKF vs True Pose (X-axis)')
    plt.plot(np.array(true_poses)[:, 0], label='True X')
    plt.plot(np.array(ekf_estimated_poses)[:, 0], label='EKF Estimated X')
    plt.plot(np.array(nnkf_estimated_poses)[:, 0], label='NNKF Estimated X')
    plt.legend()

    plt.tight_layout()
    plt.show()
    # Print the results
    print(f"\nEKF Time: {ekf_time} seconds")
    print(f"NNKF Time: {nnkf_time} seconds")

    print(f"\nEKF MSE: {mse_ekf}")
    print(f"NNKF MSE: {mse_nnkf}")
```

Appendix B: Estimation Plots

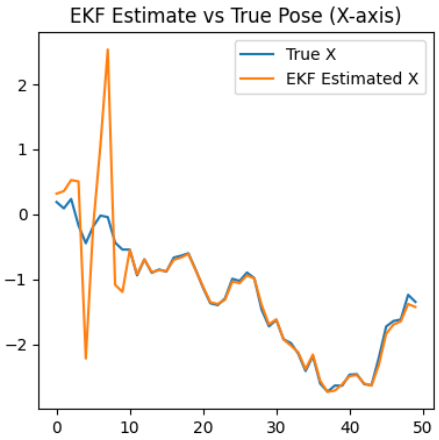
50 steps, 2D, Untrained

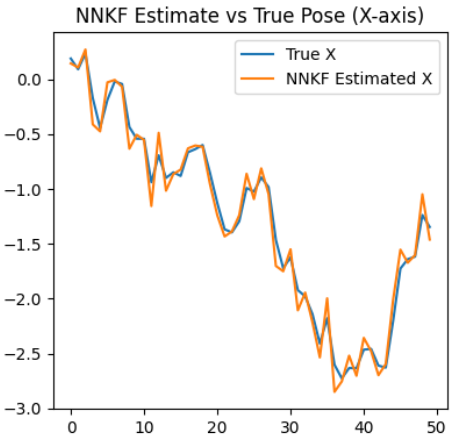


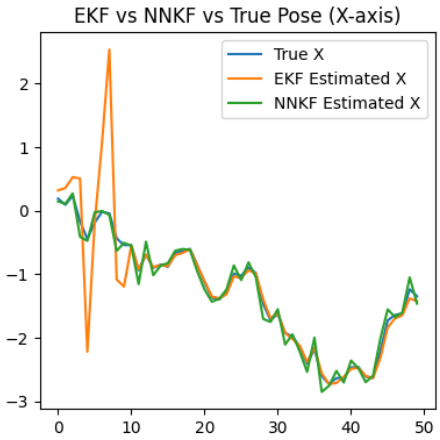




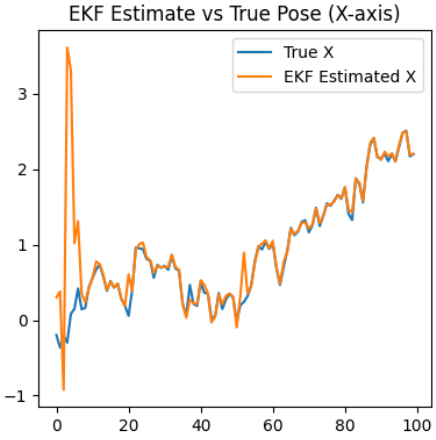
50 steps, 2D, Trained

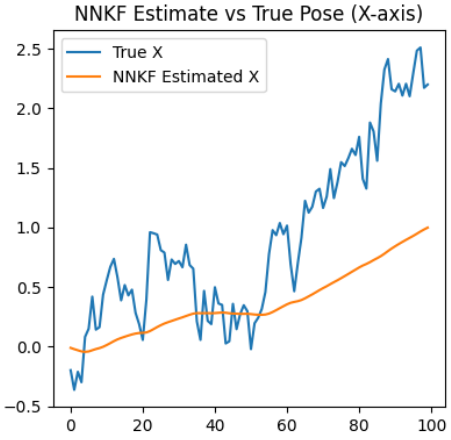


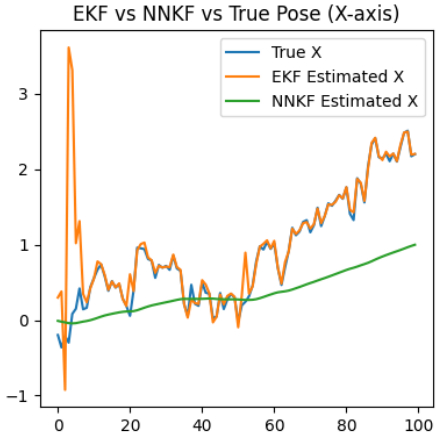




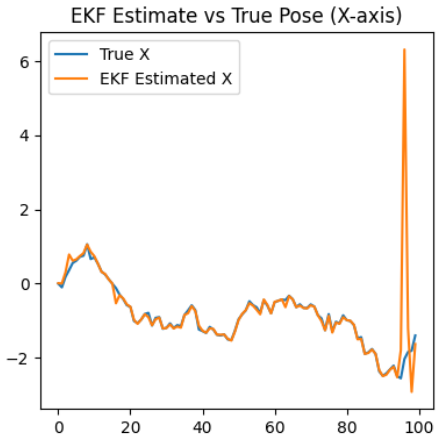
100 steps, 2D, Untrained

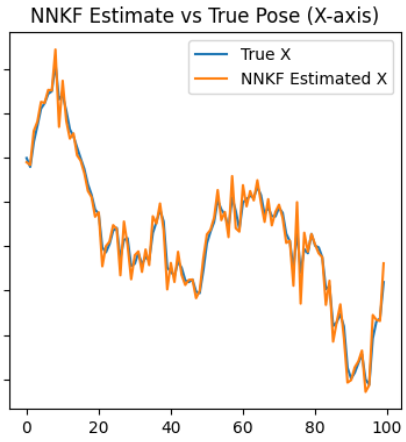


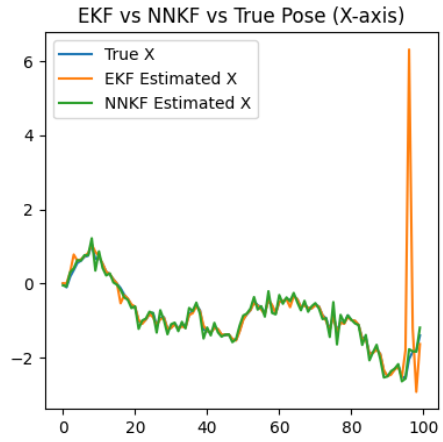




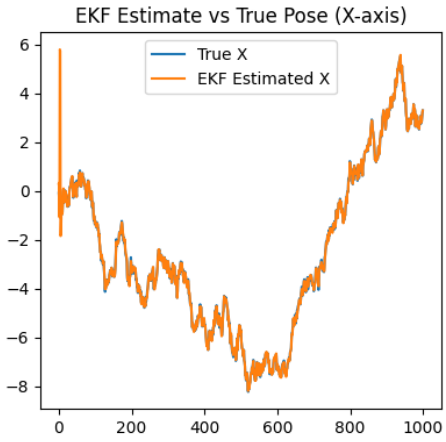
100 steps, 2D, Trained

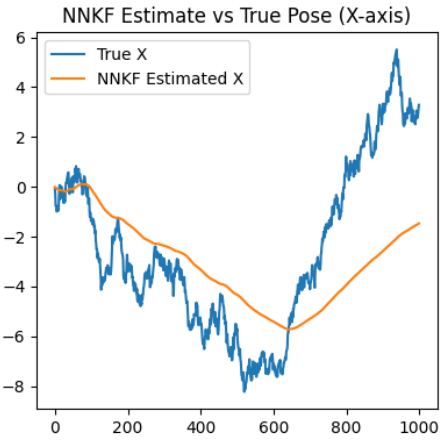


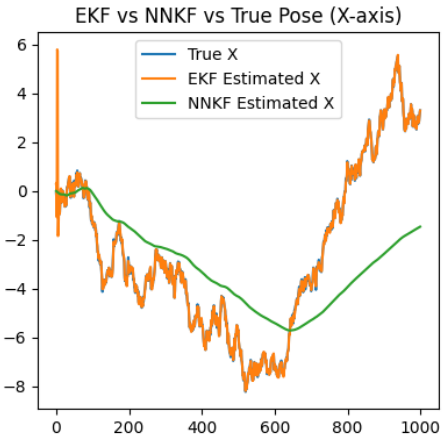




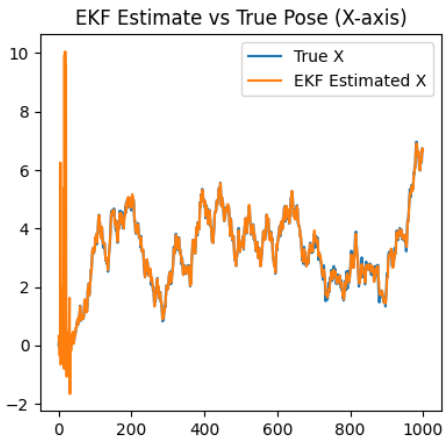
1000 steps, 2D, Untrained

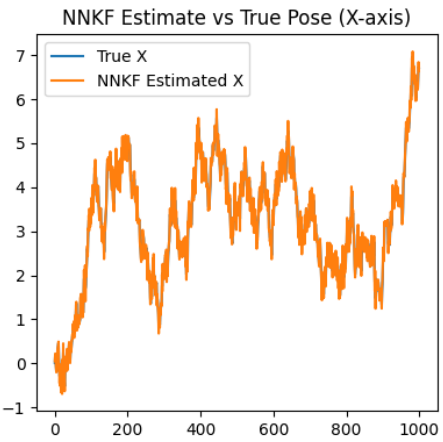


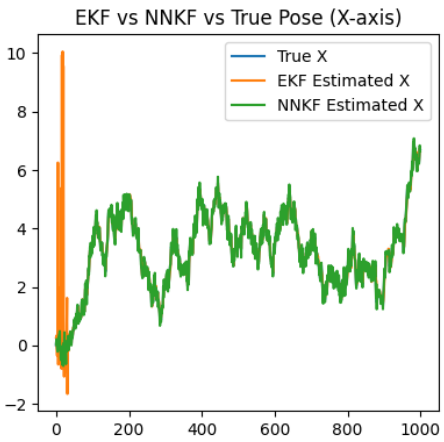




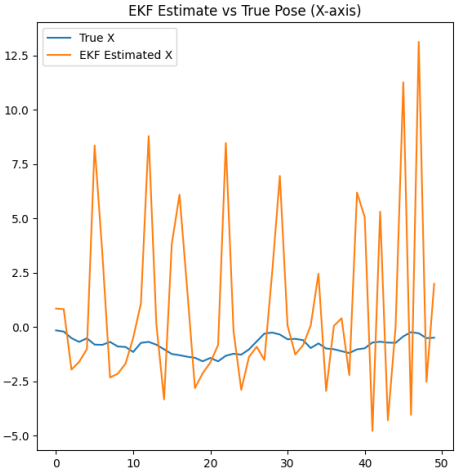
1000 steps, 2D, Trained

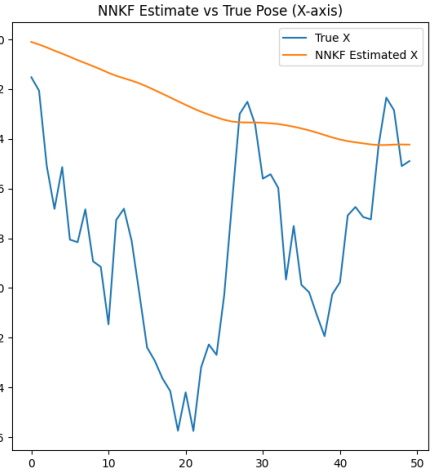


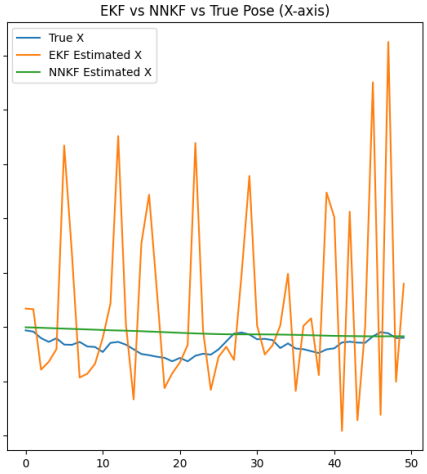




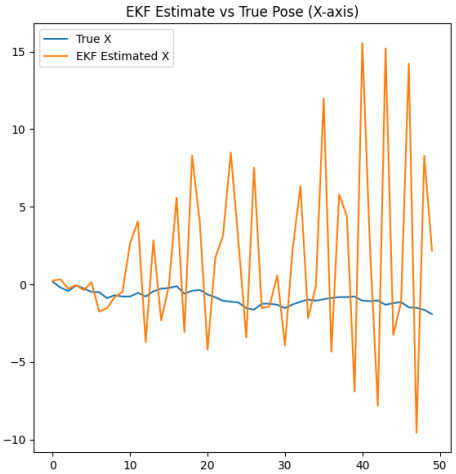
50 steps, 3D, Untrained

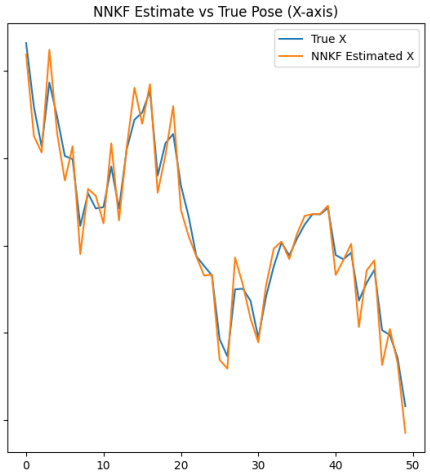


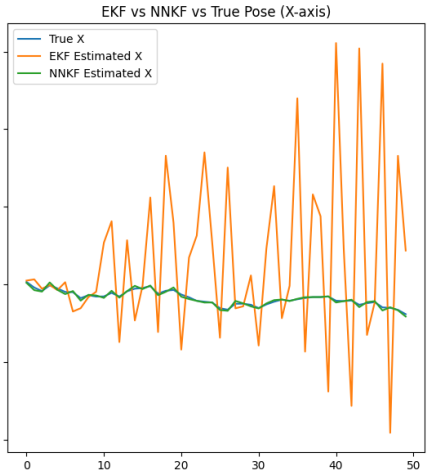




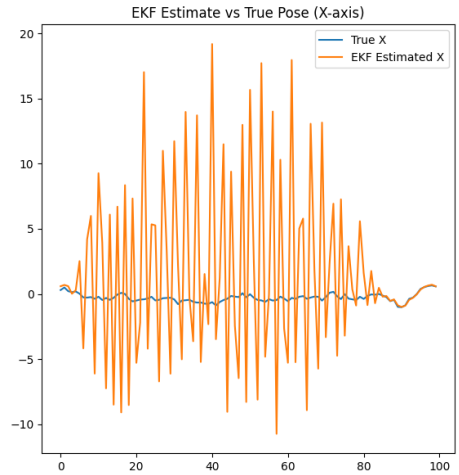
50 steps, 3D, Trained

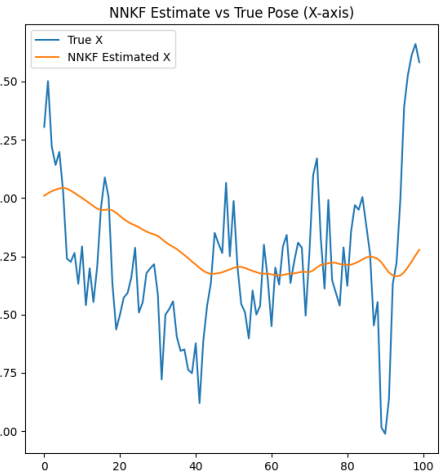


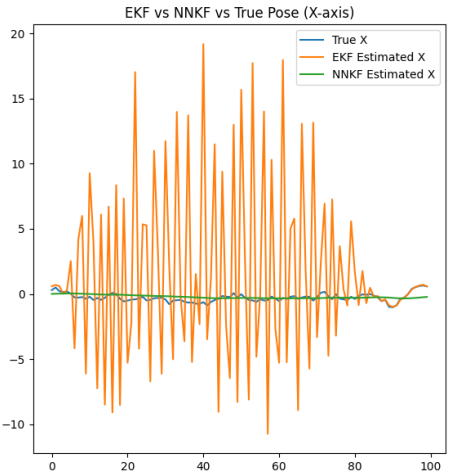




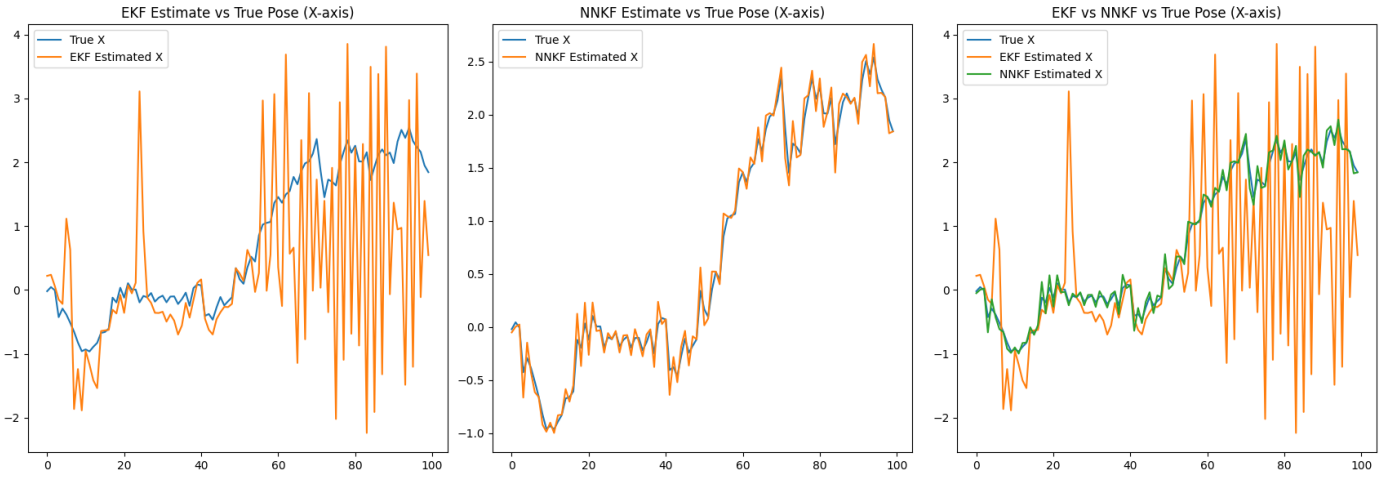
100 steps, 3D, Untrained



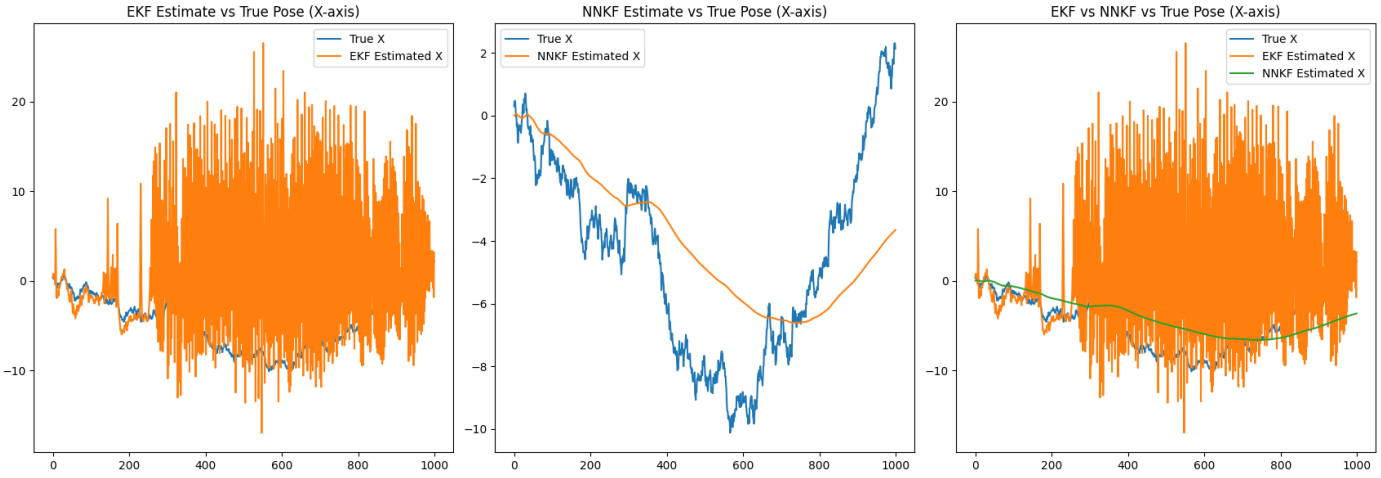




100 steps, 3D, Trained



1000 steps, 3D, Untrained



1000 steps, 3D, Trained