Assignment 4 (Particle Filter) – Advanced Robot Navigation (RBE595 Spring 2024)

Rohin Siddhartha Palaniappan Venkateswaran

Pose Estimation and Visualization

The goal of this task, as described in the assignment is to estimate the pose of the drone from the camera image at a certain timestamp. Further, we need to plot the estimated trajectory and the orientations and compare it with the ground truth values given in the **'vicon'** dataset. In order to perform this task, the algorithmic flow is as follows,

1. Read the data from the .mat files provided.
2. Correct for the inconsistencies in the data by performing certain operations.
3. Write a function that calculates the coordinates of all 4 corners of all the april tags as shown in the figure in the assignment.
4. Write a function that takes in the current dataset under consideration and the tag coordinates called "estimate pose"
5. Use the **'solvePNP'** function from openCV to estimate the pose of the drone in the 3 Dimensional space.
6. Apply the correct translational and rotational offsets as provided in the assignment to perform the transformations from the world to camera to drone frame.
7. Plot the estimated trajectory in the same plot as the ground truth trajectory

```python
import numpy as np
import cv2
import scipy.io
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import os

# Function to load data from .mat file
def load_data(filename):
    return scipy.io.loadmat(filename, simplify_cells=True)

def estimate_pose(data, tag_coordinates):

    # Extract 3D coordinates of AprilTag corners from the map layout
    map_corners_3d = []
    for data_id in data['id']:
        for id, corners in tag_coordinates:
            if data_id == id:
                for corner in corners:
                    corner_3d = np.array([corner[0], corner[1], 0])
                    map_corners_3d.append(corner_3d)

    map_corners_3d = np.array(map_corners_3d)

    # Extract 2D projections of AprilTag corners from the image data
    # The format of the p1 through p4 arrays are not[ [x1, y1], [x2, y2], [x3, y3
                                              ]] like I presumed but its actually
                                              [ [x1, x2, x3], [y1, y2, y3]]
```

```
data['p1'] = np.array(data['p1'])

if len(data['p1']) == 1:
    data['p1'] = data['p1'].reshape(-1, 1)

data['p1'] = data['p1'].T

data['p2'] = np.array(data['p2'])

if len(data['p2']) == 1:
    data['p2'] = data['p2'].reshape(-1, 1)

data['p2'] = data['p2'].T

data['p3'] = np.array(data['p3'])

if len(data['p3']) == 1:
    data['p3'] = data['p3'].reshape(-1, 1)

data['p3'] = data['p3'].T

data['p4'] = np.array(data['p4'])

if len(data['p4']) == 1:
    data['p4'] = data['p4'].reshape(-1, 1)

data['p4'] = data['p4'].T

image_corners_2d = []

for i in range(len(data['p1'])):
    image_corners_2d.append(data['p4'][i])
    image_corners_2d.append(data['p3'][i])
    image_corners_2d.append(data['p2'][i])
    image_corners_2d.append(data['p1'][i])

image_corners_2d = np.array(image_corners_2d)

# Camera intrinsic parameters and distortion coefficients (from parameters.
                                        txt)
camera_matrix = np.array([[314.1779, 0, 199.4848],
                          [0, 314.2218, 113.7838],
                          [0, 0, 1]])
dist_coeffs = np.array([-0.438607, 0.248625, 0.00072, -0.000476, -0.0911])

# Define the 3D coordinates of the camera with respect to the IMU
tvec_imu_camera = np.array([-0.04, 0.0, -0.03])  # Translation vector from
                                        IMU to camera

# Define the rotation of the camera with respect to the IMU (assuming yaw =
                                        pi/4)
yaw = np.pi / 4

# Write a rotation matrix to rotate about x by 180 degrees and z by 45
                                        degrees
rot_matrix_camera_imu = np.array([[1, 0, 0], [0, np.cos(np.pi), -np.sin(np.pi
                                        )], [0, np.sin(np.pi), np.cos(np.pi)
                                        ]]) @ np.array([[np.cos(np.pi/4), -
                                        np.sin(np.pi/4), 0], [np.sin(np.pi/4
                                        ), np.cos(np.pi/4), 0], [0, 0, 1]])

# Solve the PnP problem
```

```
    success, rvec, tvec = cv2.solvePnP(map_corners_3d, image_corners_2d,
                                        camera_matrix, dist_coeffs)

    if not success:
        raise RuntimeError("PnP solver failed to converge")

    # Convert rotation vector to rotation matrix
    rot_matrix, _ = cv2.Rodrigues(rvec)

    #Reshape tvec_imu_camera to (3, 1)
    tvec_imu_camera = tvec_imu_camera.reshape(3, 1)

    tvec = -rot_matrix.T @ tvec + -rot_matrix.T@ tvec_imu_camera

    rot_matrix = rot_matrix.T

    final_rot_matrix = rot_matrix@rot_matrix_camera_imu.T

    # Extract Euler angles from rotation matrix
    roll = np.arctan2(final_rot_matrix[2, 1], final_rot_matrix[2, 2])
    pitch = np.arctan2(-final_rot_matrix[2, 0], np.sqrt(final_rot_matrix[2, 1]**2
                                        + final_rot_matrix[2, 2]**2))
    yaw = np.arctan2(final_rot_matrix[1, 0], final_rot_matrix[0, 0])

    #return tvec_imu_camera.flatten(), np.array([roll, pitch, yaw])
    return tvec.reshape(3,), np.array([roll, pitch, yaw])
```

The program above does the following functions

1. **'world_corners'** function calculates the tag coordinates of all the april tags with all the specified distances between them as specified in the assignment.
2. **'estimate pose'** takes in the data and the tag coordinates and get the position and orientation from the 'solvePNP' function.
3. Uses the Rodriguez function to convert the orientation to a rotation matrix.
4. Perform the transformation from the camera to world to drone to world frame.
5. At the end of the function, return the position and orientation after all the transformations have been performed.
6. The plotting section is also embedded within this program itself where the estimated and ground truth positions as well as velocities have been plotted and stored in a new folder called **"observation_model_plots"**.

Process Model

In order to track the pose of the quadrotor, the Particle Filter has been implemented.

$$x = \begin{bmatrix} p \\ q \\ \dot{p} \\ b_g \\ b_a \end{bmatrix} \tag{1}$$

where the rotation matrix is defined as

$$R_q = \begin{bmatrix} \cos(q3)\cos(q2) - \sin(q1)\sin(q2)\sin(q3) & -\cos(q1)\sin(q3) & \cos(q3)\sin(q2) + \cos(q2)\sin(q1)\sin(q3) \\ \cos(q3)\sin(q1)\sin(q2) + \cos(q2)\sin(q3) & \cos(q1)\cos(q3) & \sin(q3)\sin(q2) - \cos(q3)\cos(q2)\sin(q1) \\ -\cos(q1)\sin(q2) & \sin(q1) & \cos(q1)\cos(q2) \end{bmatrix} \tag{2}$$

where $q = [\phi \, \theta \, \psi]$

Let us start by deriving the equations for process model first. According to the equation given in the assignment,

$$\dot{x} = f(x,u) = \begin{bmatrix} p_{\mathrm{dot}} \\ G_q^{-1} \cdot u_w \\ g + R_q \cdot u_a \\ n_{bg} \\ n_{ba} \end{bmatrix} \tag{3}$$

where the control inputs to our model is given by

$$u = \begin{bmatrix} wx \\ wy \\ wz \\ vx \\ vy \\ vz \end{bmatrix} \tag{4}$$

and $G(q)$ is defined as

$$G_q = \begin{bmatrix} \cos(q_2) & 0 & -\cos(q_1)\sin(q_2) \\ 0 & 1 & \sin(q_1) \\ \sin(q_2) & 0 & \cos(q_1)\cos(q_2) \end{bmatrix} \tag{5}$$

In order to write our function $f(x,u)$ i.e only in terms of x and u, we use the **sympy** library in python. After defining the variables in symbolic notation using **sympy**, we get the F function which is added to the previous state multiplied by dt, by simple Euler integration, we propagate the current state to find the next state.

$$X_{n+1} = X_n + f(x,u) \cdot \Delta t \tag{6}$$

$$X_{n+1} = F(x,u) \cdot \Delta t \tag{7}$$

Once, the process model is complete, we move to the observation model

Observation Model

Our observation model is relatively simple. We will be using the PnP based computer vision-based technique that measures the position and orientation: $\mathbf{z} = [\mathbf{p} \; \mathbf{q}] + \boldsymbol{\nu}$. We can write the state space equations as follows,

$$\mathbf{z} = \begin{bmatrix} \mathbf{I} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{I} & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ \mathbf{q} \\ \dot{\mathbf{p}} \\ \mathbf{b}_g \\ \mathbf{b}_a \end{bmatrix} + \mathcal{N}(0, \mathbf{R}) \tag{8}$$

And the code snippet is as follows

```
    def compute_observation_model_jacobian(self):
        # Compute the Jacobian of the observation model
        H = np.zeros((6, 15))
        H[:3, :3] = np.eye(3)
        H[3:, 3:6] = np.eye(3)
        return H
```

Since the observation model is linear, we need not compute the jacobians here for implementing Particle Filter.

Task 1

In order to implement the particle filter, we need to generate the particles first. The input to the particle filter algorithm is a numpy array of particles of size (particle count, 15). Here, particle count is defined by the user and the number of states is 15, as per the state space model. The range for the generated particles is set after examining the ranges for all the states and orientations in the provided datasets.

In the prediction step, each particle is propagated through the state space model as explained in the previous sections, also matrix multiplication is used for this purpose. Noise is added to the control inputs. Further, we calculate $\dot{x}$ as given in equation 3. Noise is added as per the defined covariance matrix Q and the next state is predicted as follows

$$X_{n+1} = X_n + (f(x, u) + noise) \cdot \Delta t \tag{9}$$

The noise covariance matrix Q is defined as:

The Q matrix is tuned accordingly such that equal weight is provided to position, angles and velocities. This eliminates the Nan values while doing weight updates as well as reduces RMSE, which is plotted and explained in detail in the next section.

For the update step, which involves weight calculation, we calculate the error between the measurement z and the predicted particles, and then projected to a gaussian to calculate the weights. As instructed in the assignment, we adjust the weights according to the highest index and then compute the weighted average. Finally, we get the estimates states, and store it in an array for plotting purposes.

In this step, we use the same covariance matrix as calculated in the last assignment.

```
R = np.array([
    [7.09701409e-03, 2.66809900e-05, 1.73906943e-03, 4.49014777e-04, 3.66195490e-
                                        03, 8.76154421e-04],
    [2.66809900e-05, 4.70388499e-03, -1.33432420e-03, -3.46505064e-03, 1.
                                        07454548e-03, -1.69184839e-04],
    [1.73906943e-03, -1.33432420e-03, 9.00885499e-03, 1.80220246e-03, 3.27846190e
                                        -03, -1.11786368e-03],
    [4.49014777e-04, -3.46505064e-03, 1.80220246e-03, 5.27060654e-03, 1.01361187e
                                        -03, -5.86487142e-04],
    [3.66195490e-03, 1.07454548e-03, 3.27846190e-03, 1.01361187e-03, 7.24994152e-
                                        03, -1.36454993e-03],
    [8.76154421e-04, -1.69184839e-04, -1.11786368e-03, -5.86487142e-04, -1.
                                        36454993e-03, 1.21162646e-03]
    ])
```

Finally, we get the results of tracking the pose for all the datasets, which is as follows,

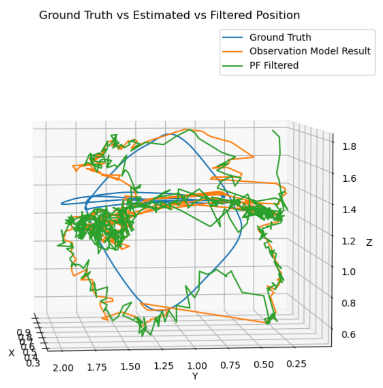


Figure 1: Dataset 1

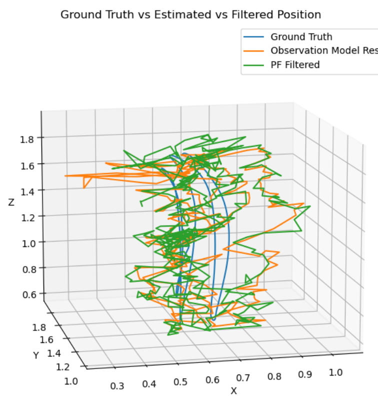


Figure 2: Dataset 2

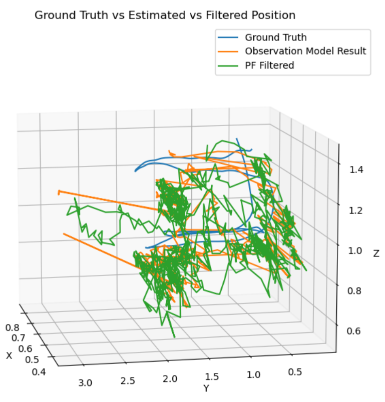


Figure 3: Dataset 3

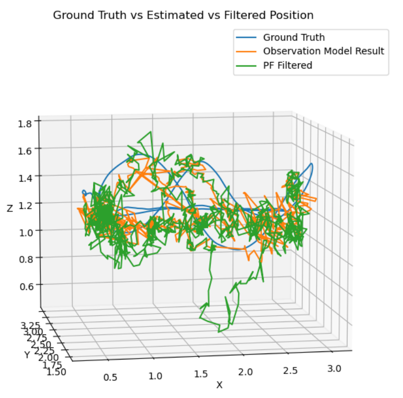


Figure 4: Dataset 4

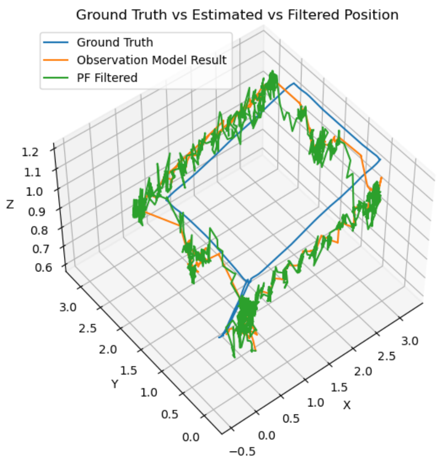


Figure 5: Dataset 5

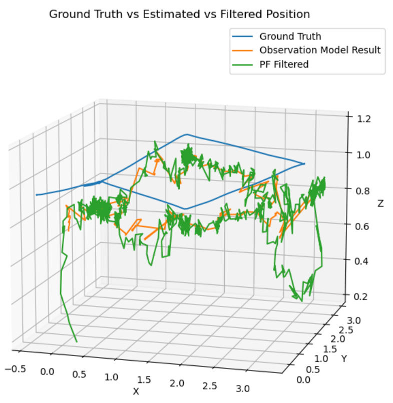


Figure 6: Dataset 6

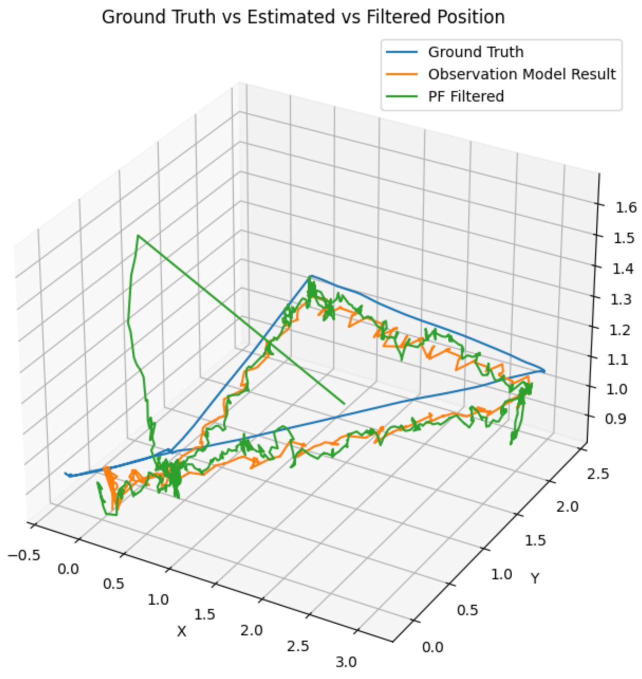


Figure 7: Dataset 7

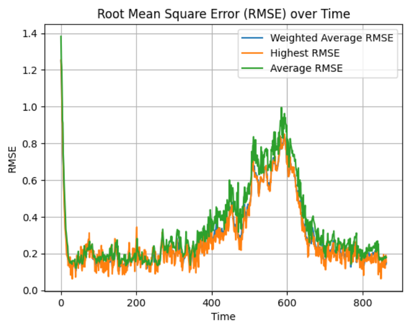
Task 2

In this task as instructed in the assignment, we have investigated the RMSE error for the 3 different weighted estimation methods : weighted average, highest, and average. This was tested over various particle sizes (250, 500, 750, 1000, 2000, 3000, 4000, 5000) for all the 7 datasets.

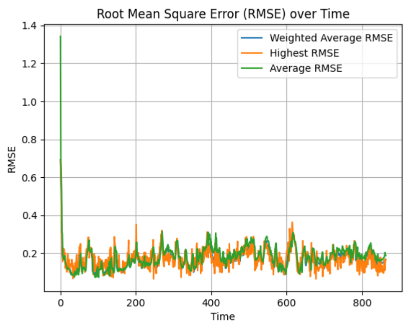
The key findings are listed below:

1. All the 3 methods showed a reduction in RMSE which indicates an improvement in the accuracy of the filter.
2. Implementing low variance resampling has contributed to the reduction of RMSE for all the datasets.
3. The RMSE values for all the sampling methods reached close to 0.15. This indicates that the particle filter more effectively tracks the position and orientation as the data processing progresses.

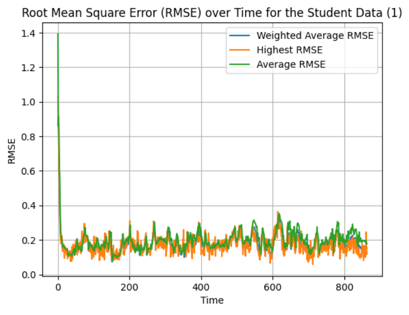
The RMSE values for all the datasets for all the particle sizes for the 3 weight estimation methods are shown below. The errors in a tabular format is also given in page 10.



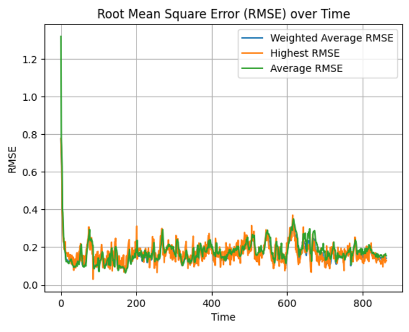
(a) 250 particles



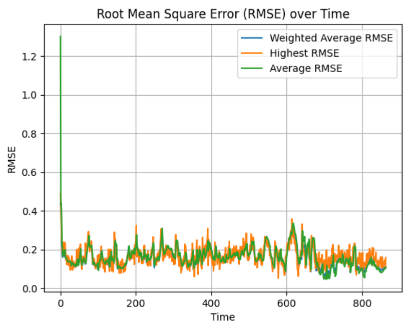
(b) 500 particles



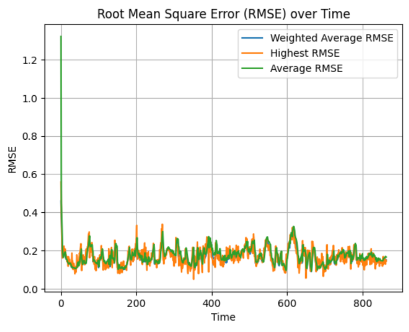
(c) 750 particles



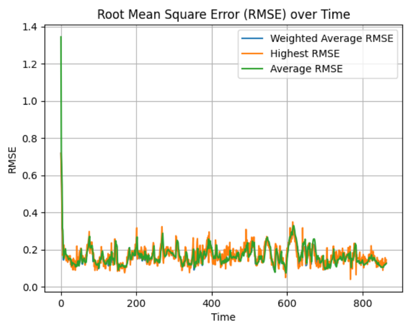
(d) 1000 particles



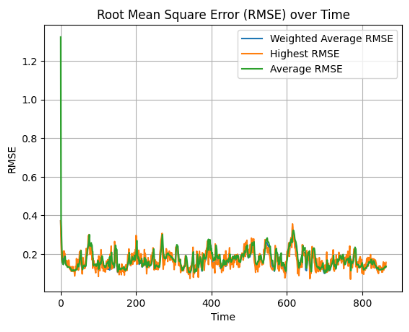
(e) 2000 particles



(f) 3000 particles

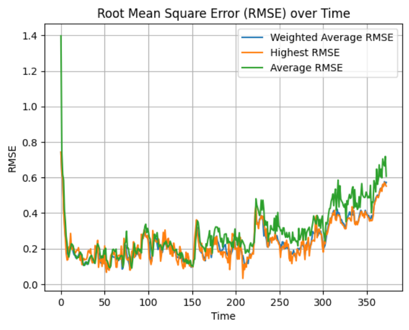


(g) 4000 particles

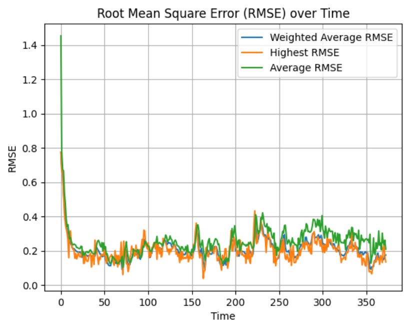


(h) 5000 particles

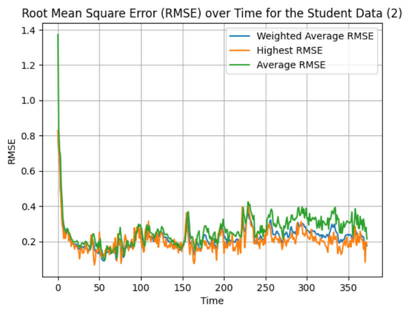
Figure 8: Dataset 1



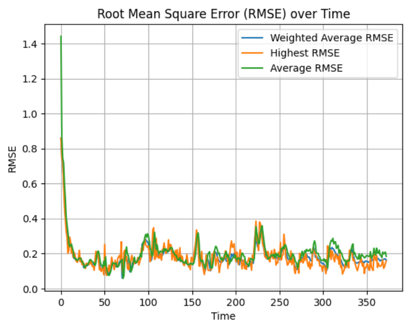
(a) 250 particles



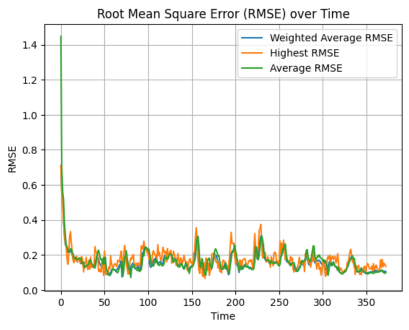
(b) 500 particles



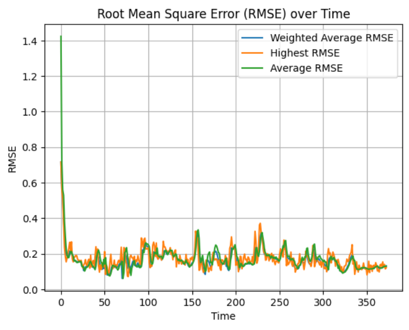
(c) 750 particles



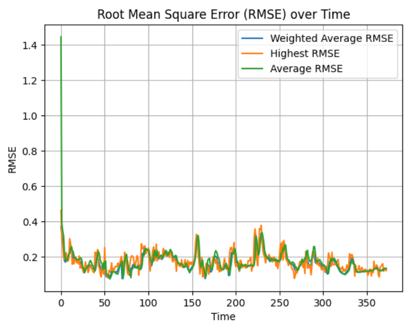
(d) 1000 particles



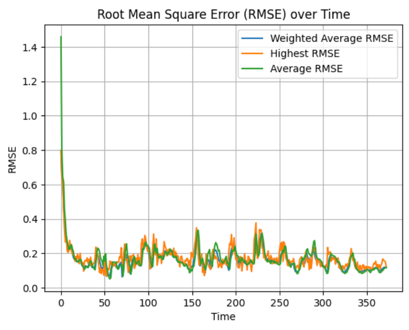
(e) 2000 particles



(f) 3000 particles

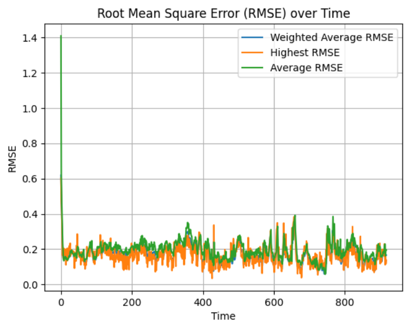


(g) 4000 particles

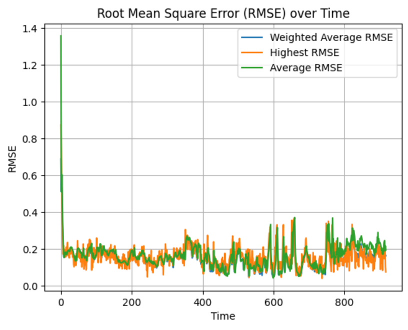


(h) 5000 particles

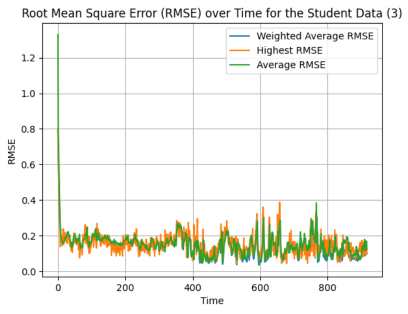
Figure 9: Dataset 2



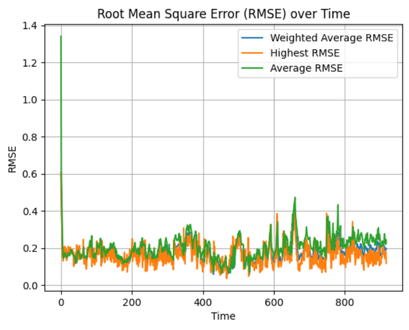
(a) 250 particles



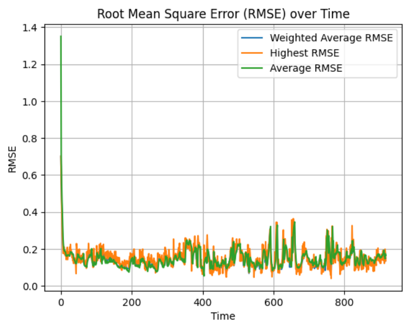
(b) 500 particles



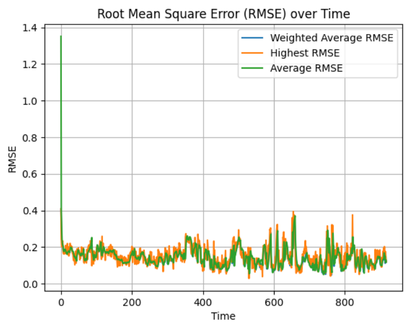
(c) 750 particles



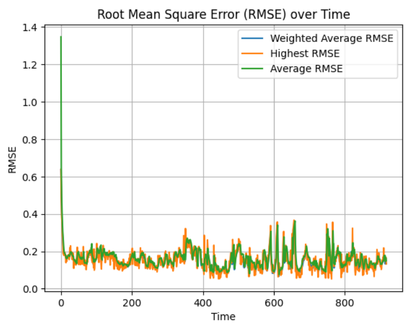
(d) 1000 particles



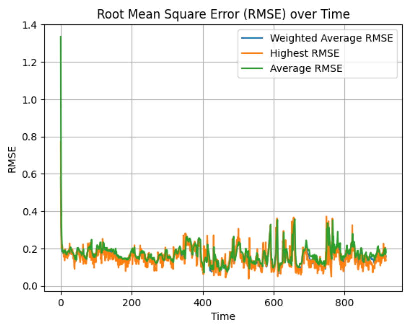
(e) 2000 particles



(f) 3000 particles

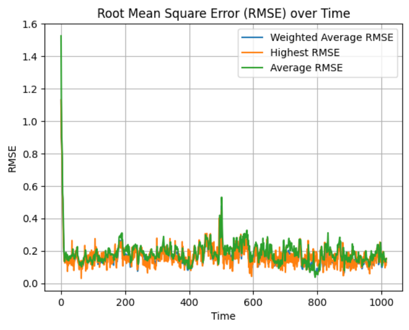


(g) 4000 particles

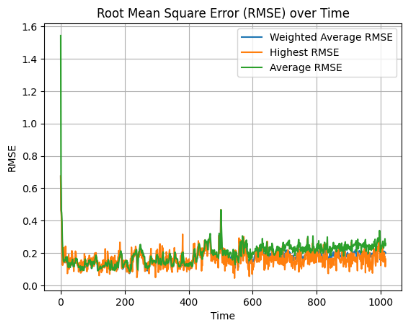


(h) 5000 particles

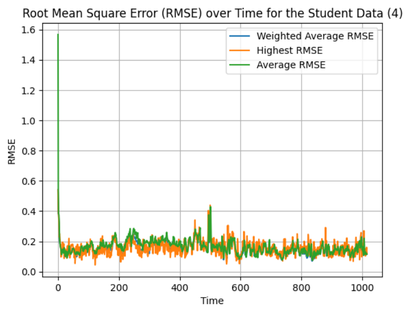
Figure 10: Dataset 3



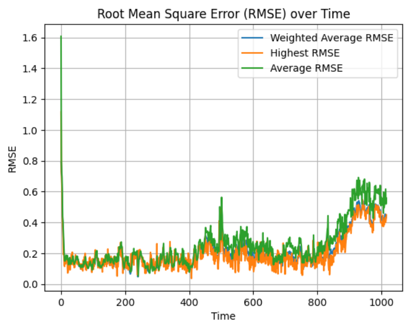
(a) 250 particles



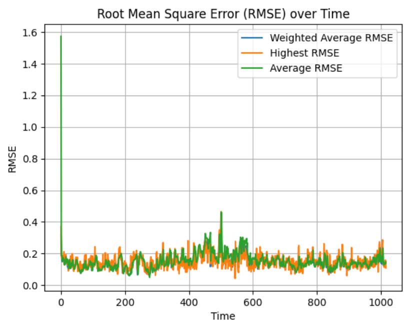
(b) 500 particles



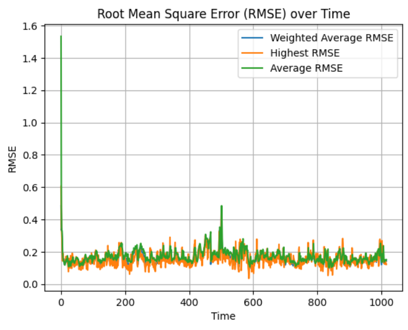
(c) 750 particles



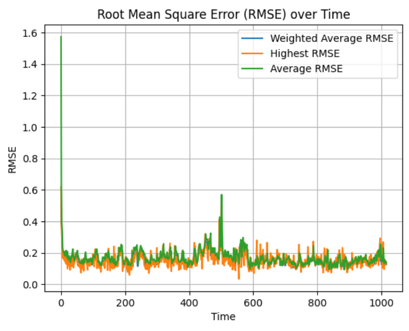
(d) 1000 particles



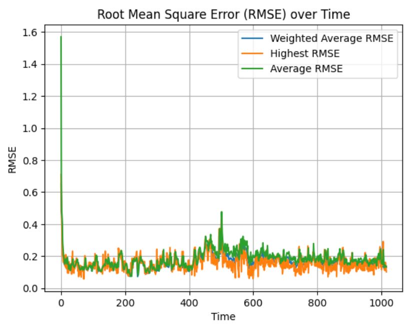
(e) 2000 particles



(f) 3000 particles

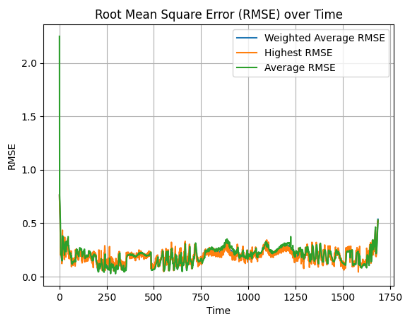


(g) 4000 particles

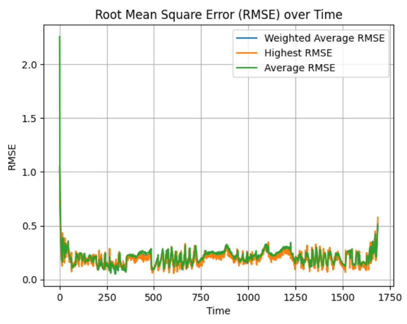


(h) 5000 particles

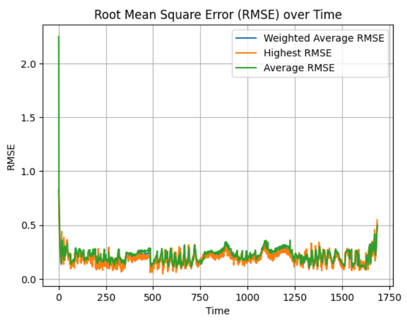
Figure 11: Dataset 4



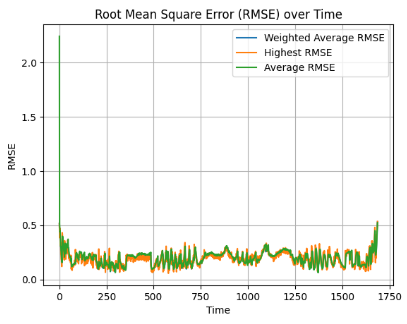
(a) 250 particles



(b) 500 particles



(c) 750 particles



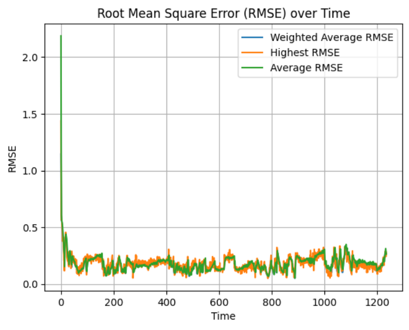
(d) 1000 particles

Figure 12: Dataset 5

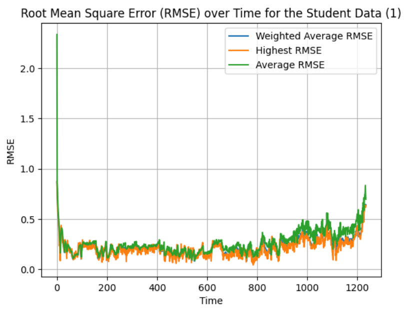
The decrease in RMSE across all the 3 weighted methods is observed when implemented along with low variance resampling. These results support the use of particle filters in applications demanding reliable state estimation and showcase their potential in various real-world scenarios.

Observations and Analysis

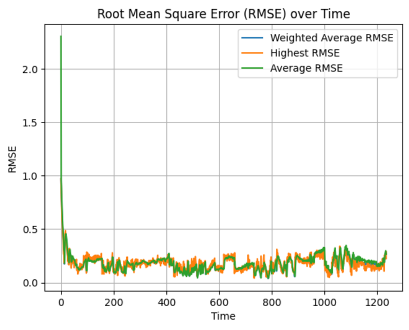
1. Increase in the particle size leads to a decrease in RMSE values.
2. Among the 3 sampling methods, weighted average method performs the best. The average and highest method perform slightly lower than the weighted average method. This may be due to the fact that averaging leads to smoothing of errors whereas selecting the highest value for the estimate may rely on other estimates which are not close to the ground truth states.



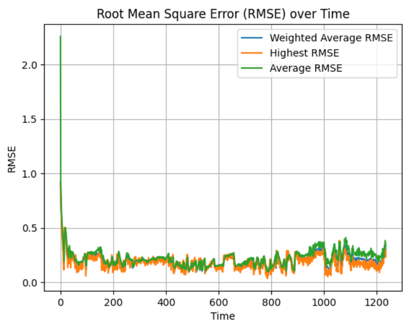
(a) 500 particles



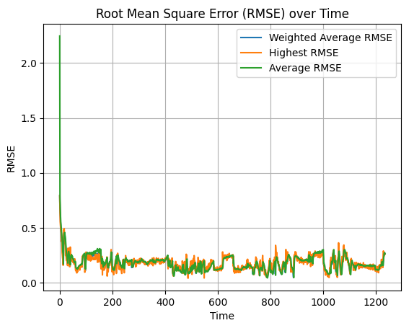
(b) 750 particles



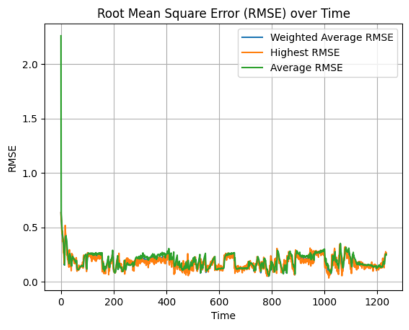
(c) 1000 particles



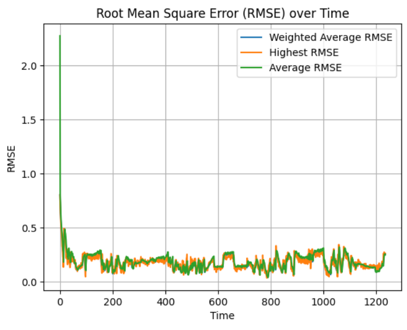
(d) 2000 particles



(e) 3000 particles

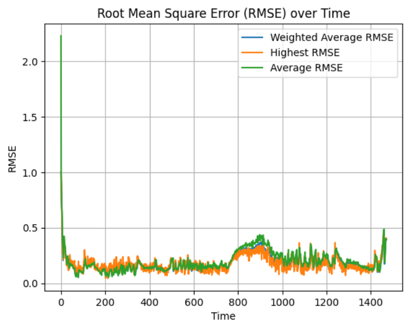


(f) 4000 particles

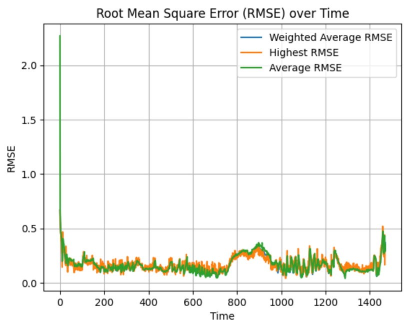


(g) 5000 particles

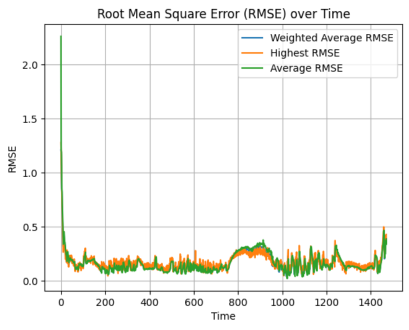
Figure 13: Dataset 6



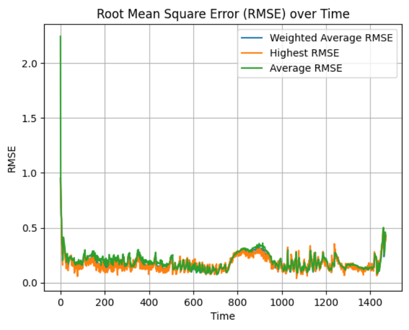
(a) 500 particles



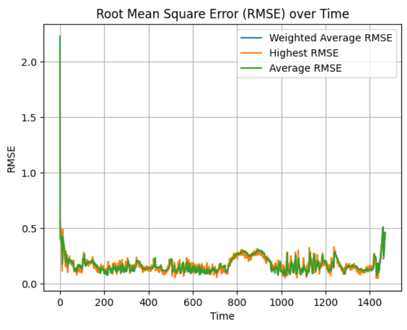
(b) 750 particles



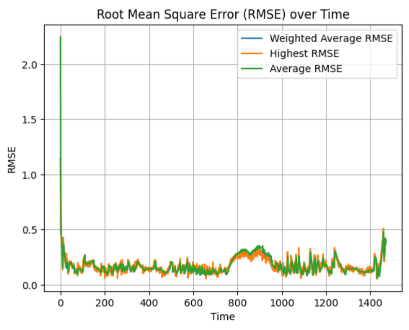
(c) 1000 particles



(d) 2000 particles



(e) 3000 particles



(f) 4000 particles

Figure 14: Dataset 7

Table 1: RMSE Values Table

Dataset Number	Number of Particles	Weighted Average Position RMSE	Weighted Average Orientation RMSE	Highest Position RMSE	Highest Orientation RMSE	Average Position RMSE	Average Orientation RMSE
1	200	0.181	0.151	0.176	0.15	0.207	0.157
1	500	0.149	0.119	0.156	0.115	0.158	0.126
1	750	0.146	0.091	0.155	0.09	0.155	0.101
1	1000	0.148	0.085	0.153	0.082	0.16	0.095
1	2000	0.152	0.085	0.153	0.084	0.161	0.094
1	3000	0.143	0.084	0.15	0.083	0.15	0.093
1	4000	0.151	0.082	0.156	0.083	0.159	0.091
1	5000	0.142	0.081	0.152	0.081	0.148	0.091
2	200	0.166	0.143	0.174	0.135	0.191	0.156
2	500	0.174	0.131	0.176	0.122	0.197	0.146
2	750	0.189	0.101	0.188	0.09	0.212	0.119
2	1000	0.154	0.1	0.165	0.089	0.17	0.119
2	2000	0.159	0.117	0.166	0.103	0.183	0.133
2	3000	0.158	0.108	0.162	0.098	0.179	0.124
2	4000	0.148	0.098	0.158	0.088	0.167	0.114
2	5000	0.144	0.092	0.155	0.08	0.16	0.11
3	200	0.218	0.154	0.219	0.151	0.25	0.168
3	500	0.14	0.097	0.143	0.1	0.157	0.108
3	750	0.136	0.086	0.137	0.09	0.152	0.096
3	1000	0.123	0.089	0.135	0.089	0.132	0.098
3	2000	0.136	0.085	0.136	0.088	0.152	0.095
3	3000	0.135	0.086	0.14	0.09	0.148	0.095
3	4000	0.128	0.083	0.135	0.087	0.139	0.092
3	5000	0.133	0.082	0.136	0.086	0.146	0.09
4	200	0.236	0.117	0.24	0.11	0.271	0.136
4	500	0.146	0.102	0.148	0.094	0.163	0.122
4	750	0.146	0.081	0.15	0.076	0.158	0.103
4	1000	0.161	0.079	0.159	0.075	0.178	0.1
4	2000	0.135	0.074	0.14	0.072	0.142	0.095
4	3000	0.143	0.075	0.146	0.07	0.152	0.096
4	4000	0.136	0.073	0.143	0.07	0.143	0.093
4	5000	0.136	0.073	0.144	0.069	0.141	0.094
5	200	0.171	0.119	0.165	0.124	0.195	0.121
5	500	0.18	0.145	0.178	0.145	0.197	0.148
5	750	0.159	0.116	0.16	0.123	0.173	0.117
5	1000	0.166	0.131	0.162	0.136	0.187	0.131
5	2000	0.163	0.118	0.162	0.125	0.178	0.118
5	3000	0.159	0.119	0.156	0.124	0.175	0.119
5	4000	0.162	0.119	0.159	0.125	0.178	0.119
5	5000	0.158	0.118	0.158	0.125	0.173	0.118
6	200	0.165	0.134	0.169	0.129	0.191	0.141
6	500	0.152	0.106	0.153	0.116	0.171	0.106
6	750	0.156	0.169	0.153	0.171	0.181	0.171
6	1000	0.146	0.122	0.144	0.124	0.168	0.124
6	2000	0.137	0.122	0.14	0.126	0.155	0.124
6	3000	0.147	0.156	0.146	0.158	0.17	0.158
6	4000	0.149	0.121	0.148	0.126	0.168	0.122
6	5000	0.141	0.12	0.145	0.122	0.158	0.121
7	200	0.155	0.111	0.152	0.113	0.181	0.116
7	500	0.156	0.102	0.153	0.109	0.178	0.104
7	750	0.153	0.112	0.151	0.12	0.173	0.113
7	1000	0.152	0.105	0.147	0.111	0.173	0.106
7	2000	0.147	0.104	0.148	0.109	0.163	0.106
7	3000	0.145	0.109	0.146	0.116	0.161	0.11
7	4000	0.142	0.104	0.143	0.112	0.159	0.105
7	5000	0.145	0.114	0.144	0.118	0.162	0.116

Task 3

Table 2: Position and Orientation RMSE for EKF

Student Number	Position RMSE	Orientation RMSE
1	0.155	0.094
2	0.120	0.070
3	0.185	0.113
4	0.193	0.116
5	0.150	0.091
6	0.089	0.073
7	0.118	0.072

Ease of Implementation

Comparing both the Extended Kalman Filter and the Particle Filter, the Particle Filter was easier to code due to its well defined mathematics and tuning parameters. The Extended Kalman Filter involved calculating the Jacobian which was quite complex to implement, and required more time to run. On the other hand, the particle filter involved randomly initialized particles, and then followed by the initialization, prediction, resampling and update steps. However, the Particle Filter was very sensitive to noise and the parameters had to be tuned carefully. In the case of EKF, only the process model covariance had to be tuned which was simpler compared to EKF. EKF needed a good initial guess otherwise the algorithm creates errors while tracking the trajectory, whereas the particle filter does not need initial guess of state as it just requires particles to start with.

Speed of Code

In terms of speed of code, the EKF runs faster than particle filter, especially as the number of particles increase from 750 to 2000 and further. Speed is a crucial factor in real-world tracking applications, even though particle filter offer a robust solution to nonlinearity, it requires require more computational resources, which can be a significant drawback in time-critical applications.

Accuracy of Results

The accuracy of the PF versus the EKF largely depends on the system's dynamics and the nature of observations. In our system I observed that in some data set as 2,5,6,7 EKF outperforms the PF slightly. And on the other datasets PFs tend to outperform EKFs in system.

Conclusion

According to my experience in both implementations, taking into consideration ease of implementation and time taken to run the code, I would prefer EKF as even though it is a bit cumbersome to compute the jacobians, it runs faster than a 2000 particle filter.