Project 4 - Particle Filters

Keith Chester

April 14, 2024

In this project we are provided with a subset of data (**studentdata*N*** where N is 0 through 7). The data is generated by flying a drone on an indoor course. The drone is recorded by a highly accurate motion capture system, which acts as our ground truth. The drone has an IMU (inertial measurement unit, consisting of an accelerometer and a gyroscope for acceleraton and rotational acceleration respectively), and an on board camera looking down. The floor of the course contains printed AprilTags of a set grid placement, allowing us to estimate a drone position from them alone.

The goal of this project is to demonstrate Particles Filters applied to a nonlinear system (our drone) and produce better estimates than determining the position through the camera alone. For this we will utilize the camera to get an estimated position, note the IMU, and attempt to utilize the filter to smooth the movements out to a realistic tracking path for the drone.

Task 1

Task 1 starts us off by instructing us to build the particle filter from which the rest of the assignment will experiment with. Our implementation can be found in **pf.py**, wherein we create a *ParticleFilter* class. The class is customized with the number of desired particles and other parameters such as noise range to be added during our process model. We attempt to, whenever possible, treat all particles and calculations as vectors and do our best to avoid iteration. This is done to optimize performance and minimize time spent in calculation per step.

For our application problem, we work with a 15 state space vector. When created, the particle filter creates a set of particles, consisting of N particles, each being a 15 element vector. We focus only on the first 6 elements of the vector, as the first three represent position and the second three represent orientation. The range of the position is set to be between 0 and 3 meters (save the z axis, which is limited to 1.5 meter) due to observation of the data being ran against - it seems the drone never goes beyond these bounds. The orientation is set to limit the drone in yaw, pitch, and roll in the $-\frac{\pi}{2}$ to $\frac{\pi}{2}$ range.

Once we have our initial particles and our initial position read, we begin iterating through a constant loop until we process all of the data. We calculate the Δt since our last read, and run the predict step on every particle.

The predict step attempts to solve xdot, which we know to be:

$$\dot{x} = \begin{bmatrix} \dot{p} \\ G(q)^{-1}\omega_w \\ g + R(q)\omega_a \\ n_g \\ n_a \end{bmatrix} \tag{1}$$

...wherein $\dot{p}$ is the derivative of our position, $G(q)$ is the rotation matrix from the world frame to the drone frame, ω_w is the force vector, $R(q)$ is the rotation matrix from the drone frame to the world frame, ω_a is the acceleration vector (gravity is negative due to frame orientation), n_g is the bias from the gyroscope, and n_a is the bias from the accelerometer.

We then apply the Δt to the $\dot{x}$ vector to get our new position:

$$x_{t+1} = x_t + \dot{x}\Delta t \tag{2}$$

When we do this calculation, we also generate noise based on a configurable hyperparameter for our class. We experimented with how to add this noise, landing on three possible routes:

- Add the noise directly to ω_w and ω_a before applying the Δt

- Add the noise universally across the $\dot{x}$ vector prior to applying the Δt
- Add the noise universally across the new x vector after adding $\dot{x}$.

Ultimately we decided to add the noise to the ω_w and ω_a vectors before applying the Δt. In a future section of this paper we will discuss how we narrowed down what noise values to use.

After the predict section, we estimate the pose of the drone from the image data through the use of the solvePnP function. We take this measurement and perform the update step, wherein we apply the diagonal of our calculated covariance matrix to our measured state. This matrix was calculated in a prior assignment, and is listed below as $C_{measurement}$:

$$\begin{bmatrix} 7.09701409e-03 & 2.66809900e-05 & 1.73906943e-03 & 4.49014777e-04 & 3.66195490e-03 & 8.76154421e-04 \\ 2.66809900e-05 & 4.70388499e-03 & -1.33432420e-03 & -3.46505064e-03 & 1.07454548e-03 & -1.69184839e-04 \\ 1.73906943e-03 & -1.33432420e-03 & 9.00885499e-03 & 1.80220246e-03 & 3.27846190e-03 & -1.11786368e-03 \\ 4.49014777e-04 & -3.46505064e-03 & 1.80220246e-03 & 5.27060654e-03 & 1.01361187e-03 & -5.86487142e-04 \\ 3.66195490e-03 & 1.07454548e-03 & 3.27846190e-03 & 1.01361187e-03 & 7.24994152e-03 & -1.36454993e-03 \\ 8.76154421e-04 & -1.69184839e-04 & -1.11786368e-03 & -5.86487142e-04 & -1.36454993e-03 & 1.21162646e-03 \end{bmatrix} \tag{3}$$

our update step modifies the estimated state x_t to x_u via:

$$x_u = I_{6x6}x_t + C_{measurement} \tag{4}$$

We then calculate the weights of the existing particles based on this measured prediction. Initially, the weights are set to $\frac{1}{N}$, but each future step they are:

$$W_i = \frac{1}{\sqrt{\Sigma(x_i - x_u)^2}} \tag{5}$$

We then normalize the weights to ensure they sum to 1, via:

$$W_i = \frac{W_i}{\Sigma W} \tag{6}$$

We then select the estimate from the weights and particles in one of several manners - by the highest weighted particle, the weighted average of all particles, or the flat average of all particles. We explore the performance of each later.

Finally, we resample each particle each step. This works by stochastically selecting a subset of the particles with a higher probability of tossing low weighted particles and keeping high weighted particles.

The process continues until we have created an estimate for each position across the entire dataset.

Noise Hyperparameter Search

Earlier we mentioned that we experimented with different noise methods; we also experimented with different noise values in order to find the "best performing" across all datasets for a singular generic value. We note that the first dataset, *studentdata0*, exhibited significant "teleportation" at parts due to the camera losing sight of AprilTags, resulting in that dataset performing particularly worse.

To do this, we created **noise_tester.py**, which runs the particle filter with a set number of particles (2000), but modifies the noise added to ω_a and ω_w. We ran these values across all datasets, calculating the RMSE and exporting to a CSV file. RMSE is defined as:

$$e = \sqrt{\frac{\Sigma(x_{true} - x_{est})^2}{N}} \tag{7}$$

A quick utilization of a spreadsheet client allowed us to quickly average the RMSE across specific noise combinations. Over time we expanded and then refined the search space of the noise values, finally landing on a set of values that worked across most datasets well, setting them as the defaults in the *ParticleFilter* class.

Dataset Demonstrations

Below are the results of various datasets and the default values of the particle filter, with 2000 particles:

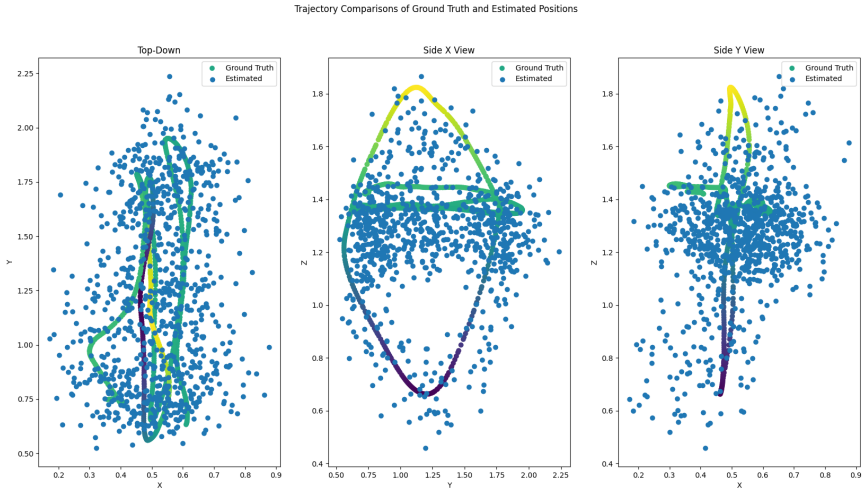

Figure 1: Dataset 1 Trajectories

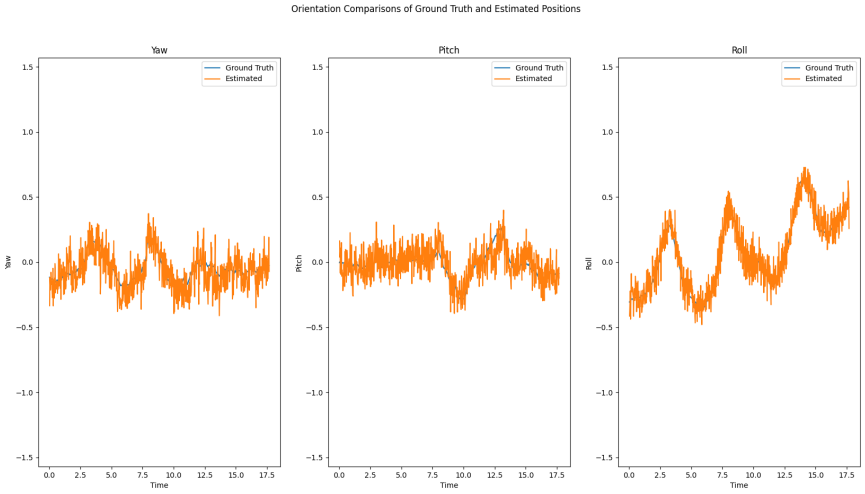

Figure 2: Dataset 1 Orientations

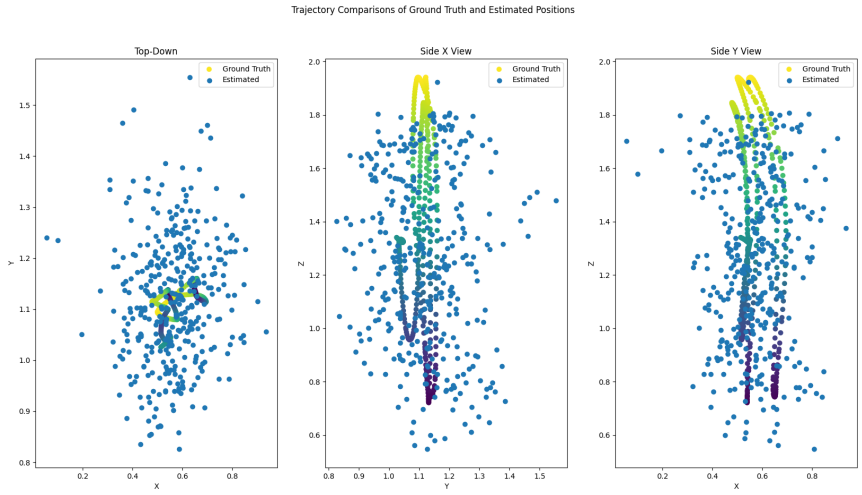

Figure 3: Dataset 2 Trajectories

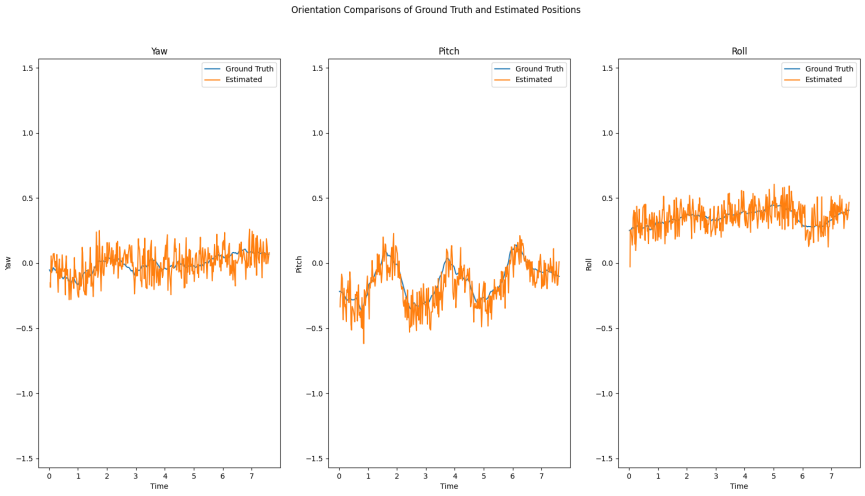

Figure 4: Dataset 2 Orientations

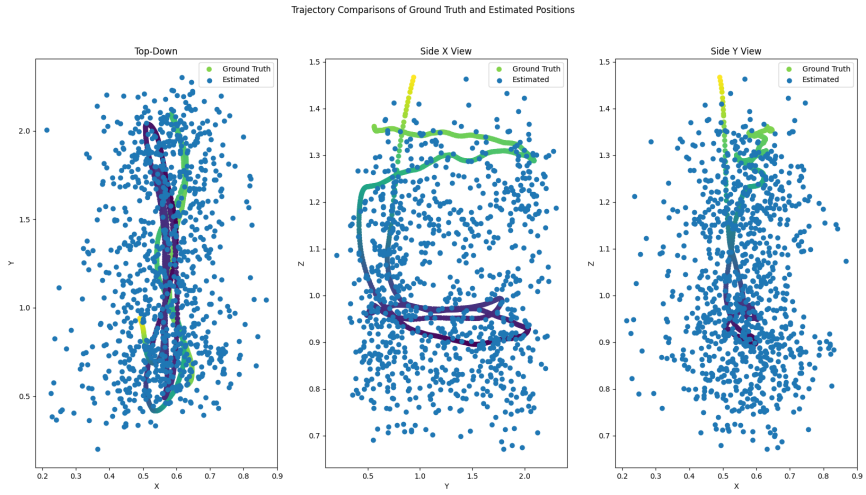

Figure 5: Dataset 3 Trajectories

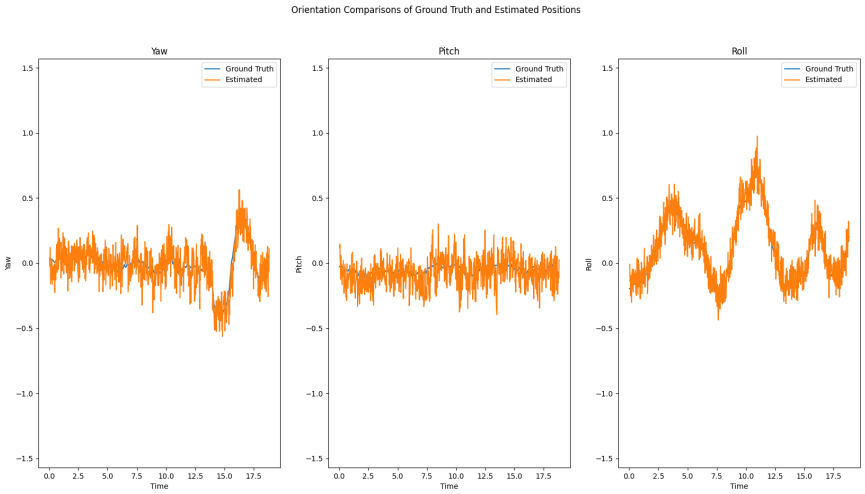

Figure 6: Dataset 3 Orientations

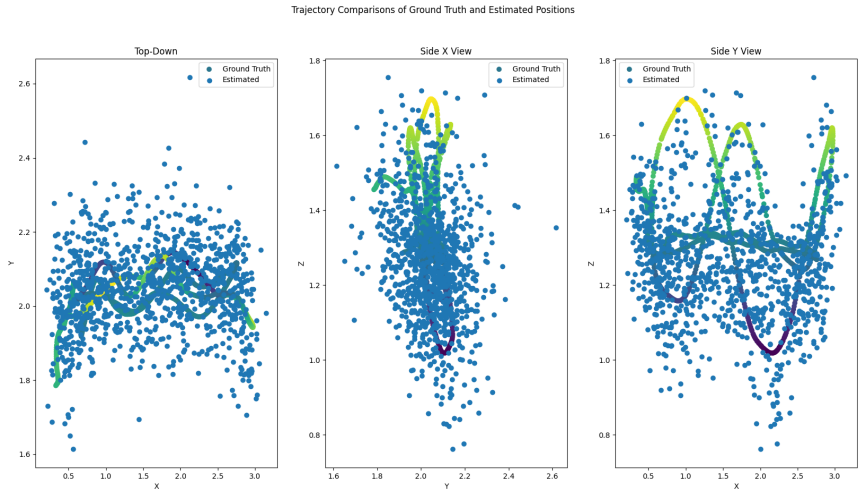

Figure 7: Dataset 4 Trajectories

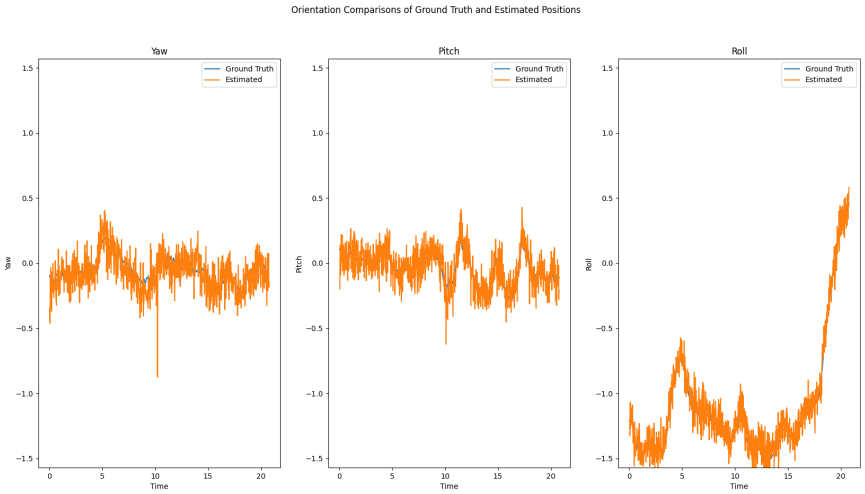

Figure 8: Dataset 4 Orientations

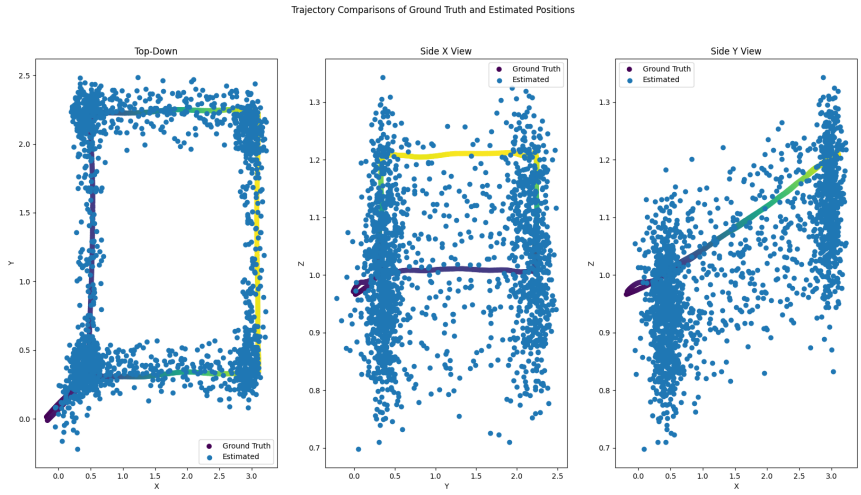

Figure 9: Dataset 5 Trajectories

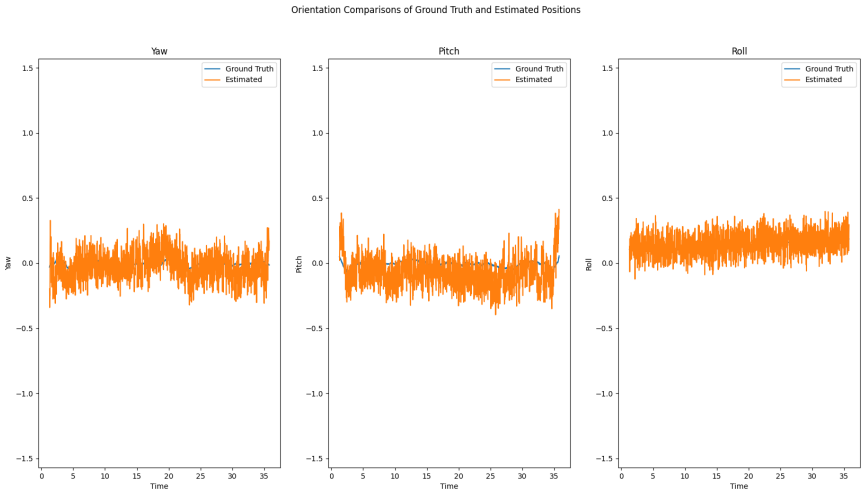

Figure 10: Dataset 5 Orientations

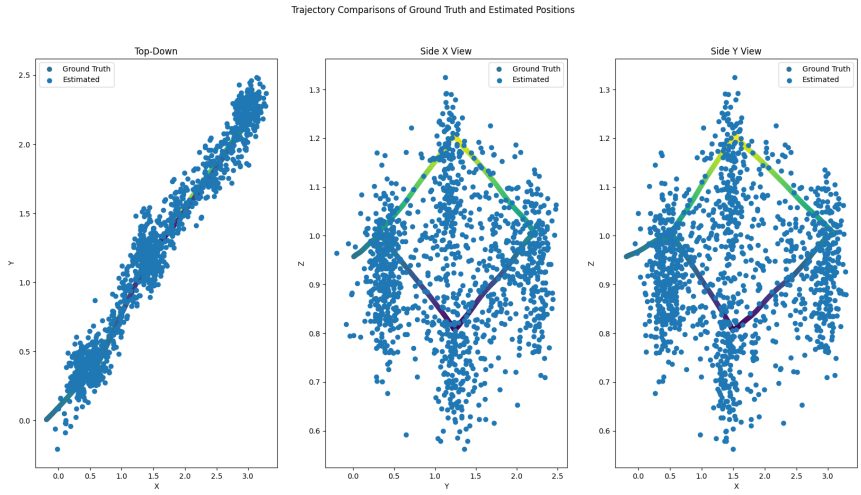

Figure 11: Dataset 6 Trajectories

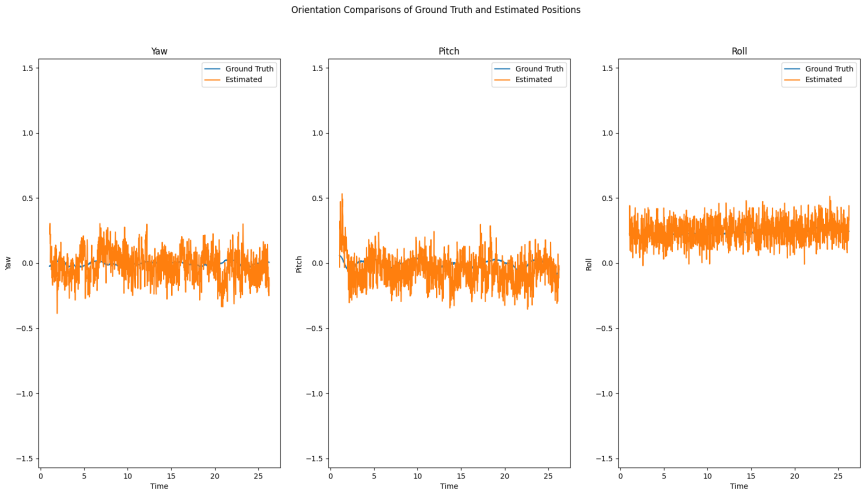

Figure 12: Dataset 6 Orientations

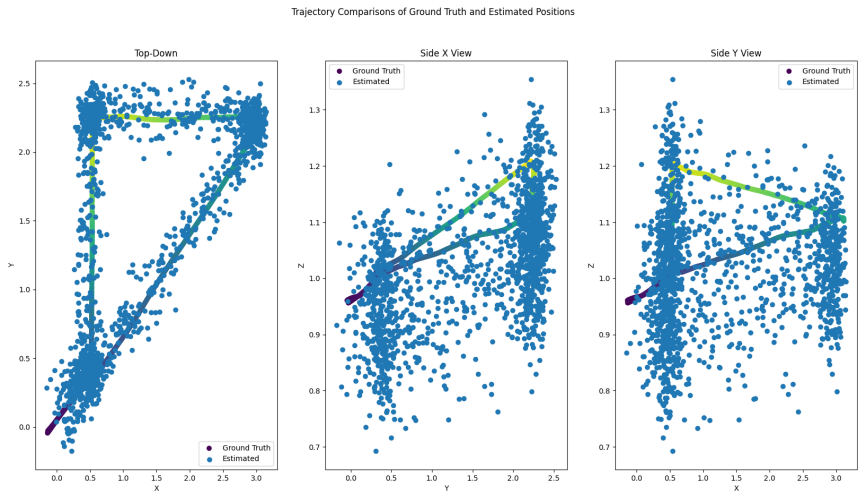

Figure 13: Dataset 7 Trajectories

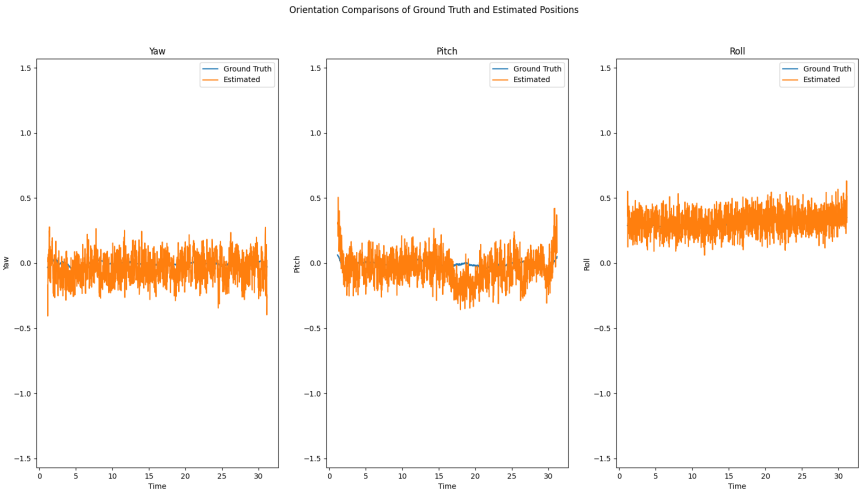

Figure 14: Dataset 7 Orientations

Task 2

In task 2 we are tasked with testing certain hyperparameters of our particle filter in order to observe performnace differences. Here we aim to test the sampling method - how we ascertain our estimate given our particles - and the effect of the number of particles on the performance of the filter.

Sampling Method

First, we will explore the sampling method. We have three to test:

- **Highest Weighted Particle** - We select the particle with the highest weight as our estimate.
- **Weighted Average** - We take the weighted average of all particles as our estimate.
- **Flat Average** - We take the flat average of all particles as our estimate, ignoring the weights.

This author predicted that weighted average would out perform the others. Below we have charts of the performance through RMSE of each dataset and method, with a steady 2000 particles and unchanged parameters otherwise for each. We then show a table of average RMSE performance across datasets.

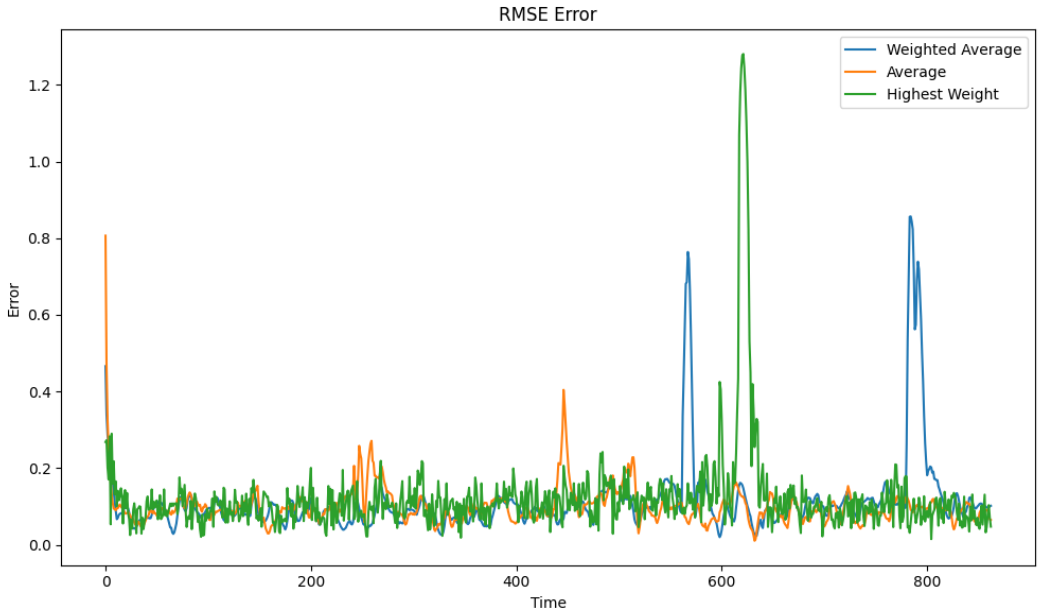

Figure 15: Dataset 1 Sampling Method Performance

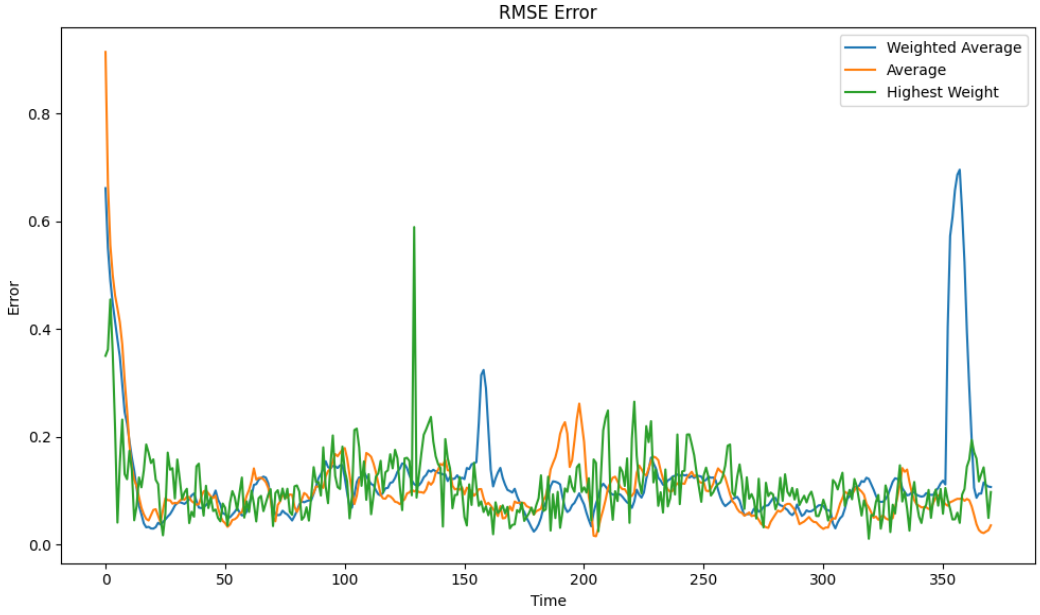

Figure 16: Dataset 2 Sampling Method Performance

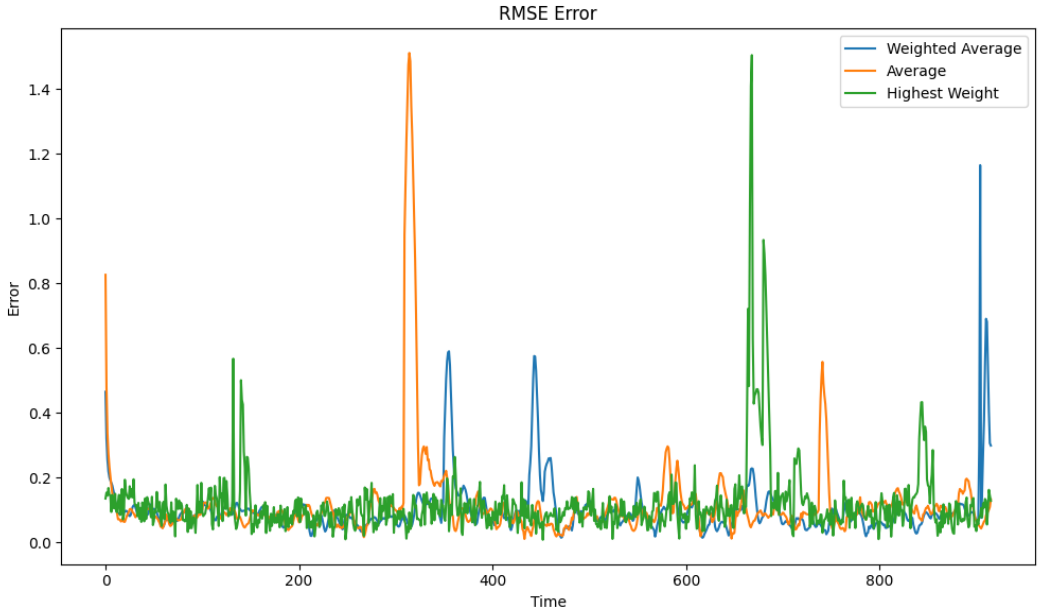

Figure 17: Dataset 3 Sampling Method Performance

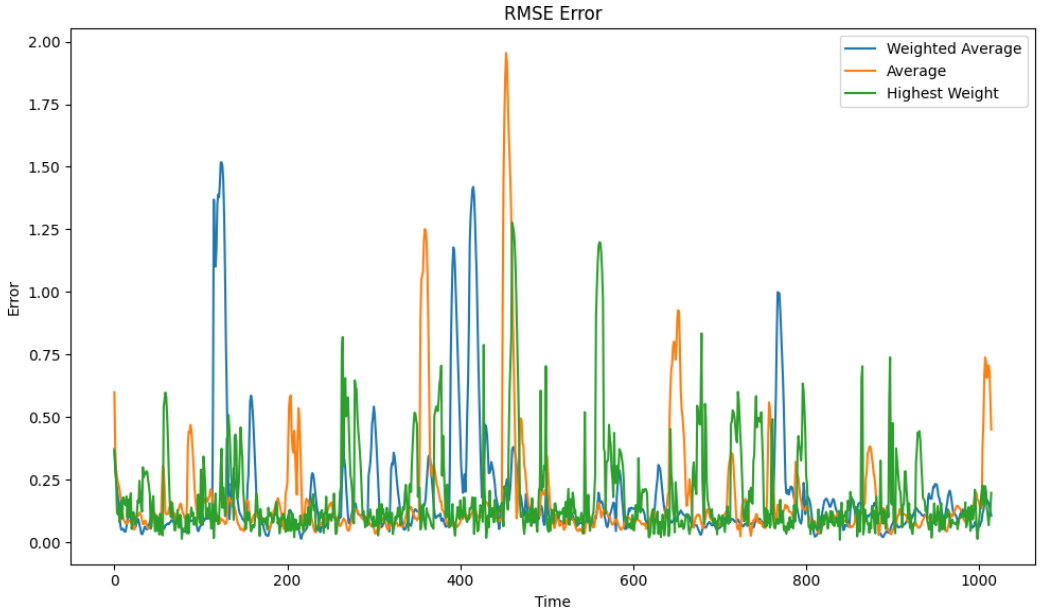

Figure 18: Dataset 4 Sampling Method Performance

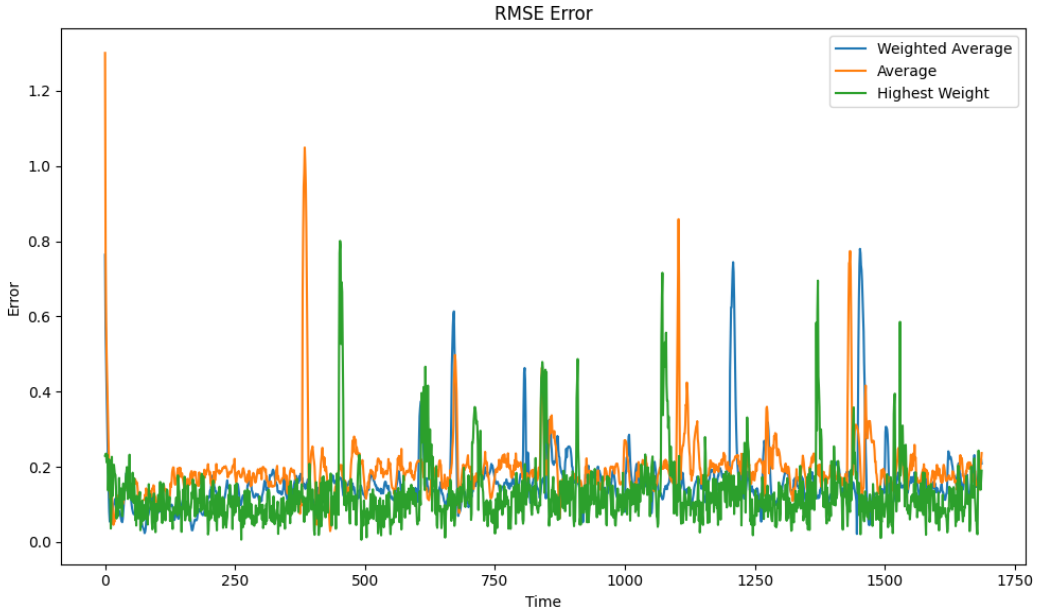

Figure 19: Dataset 5 Sampling Method Performance

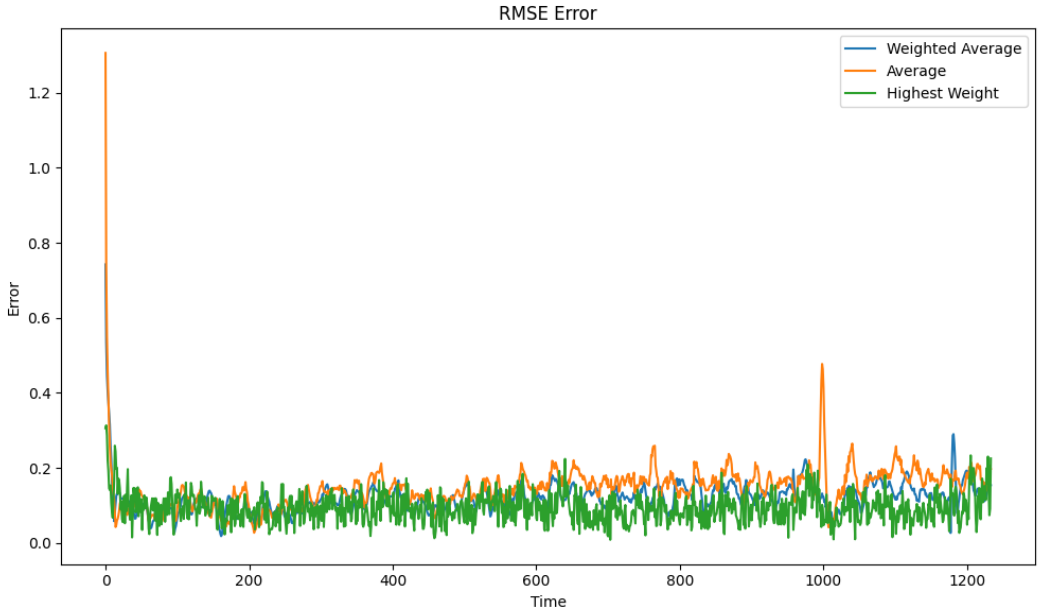

Figure 20: Dataset 6 Sampling Method Performance

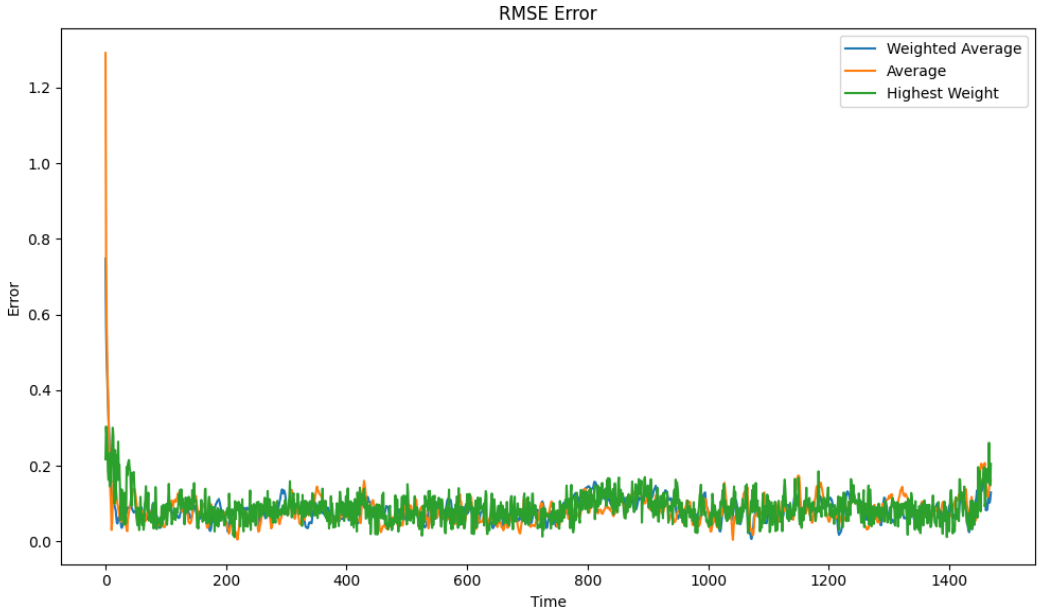

Figure 21: Dataset 7 Sampling Method Performance

And our performance numerically presented:

Dataset	Weighted Average	Averaged	Highest Weighted Particle
0	nan	1767.595	2070.701
1	0.113	0.100	0.121
2	0.117	0.103	0.108
3	0.100	0.117	0.117
4	0.175	0.170	0.183
5	0.156	0.192	0.127
6	0.120	0.147	0.093
7	0.084	0.083	0.088

Again, *studentdata0* is notably poor performance due to the camera losing sight of the AprilTags. We could get good performance from this dataset, but it required significantly higher noise. Since we are aiming to not tune the filter to a specific dataset, we keep the best performing across all other datasets.

To this author's surprised, the highest weighted particle often outperformed the other methods. That being said, it does seem to be slightly less "smoothed" in the charts, showing that it may present better results overall but has a higher likelihood of suffering from poor measurements.

For the remainder of this assignment, the highest weighted particle method will be used unless otherwise stated.

Particle Count

Now we seek to determine the effect of particle counts on the datasets. We are looking at the performance of 250, 500, 750, 1000, 2000, 3000, 4000, and 5000 particles.

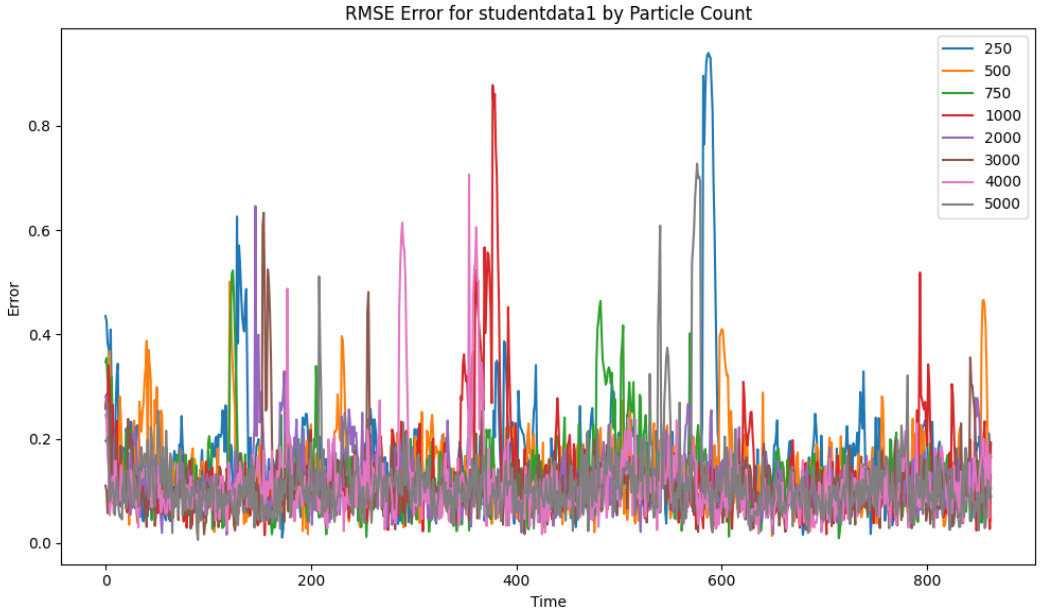

Figure 22: Dataset 1 Particle Count Performance

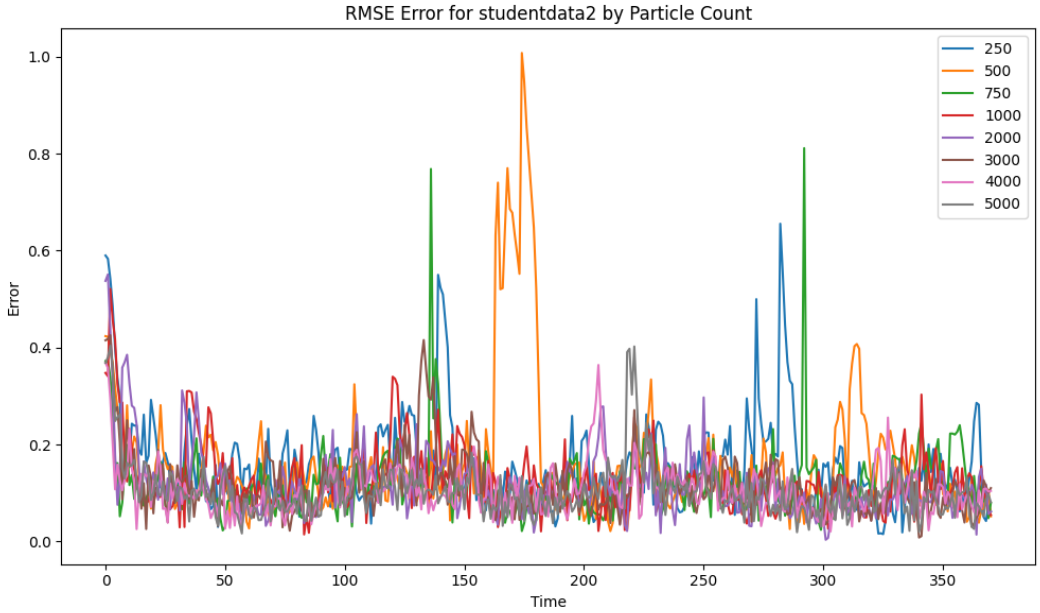

Figure 23: Dataset 2 Particle Count Performance

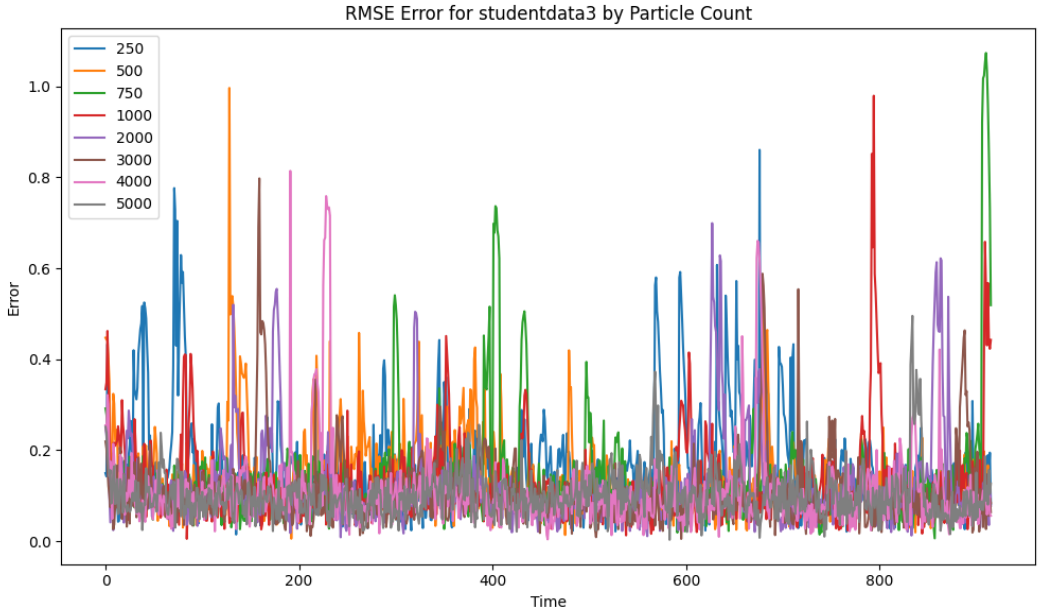

Figure 24: Dataset 3 Particle Count Performance

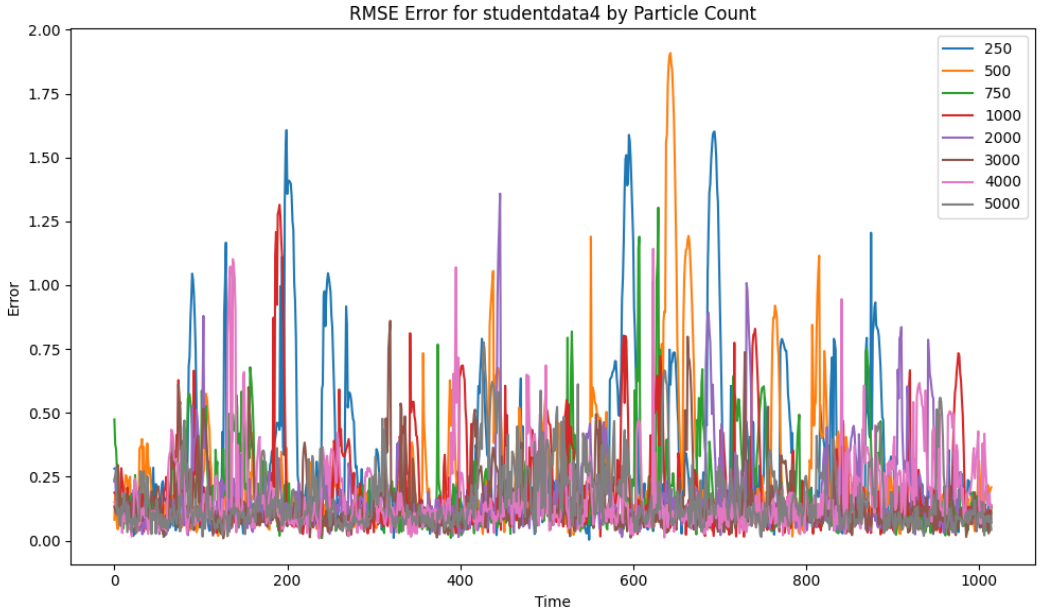

Figure 25: Dataset 4 Particle Count Performance

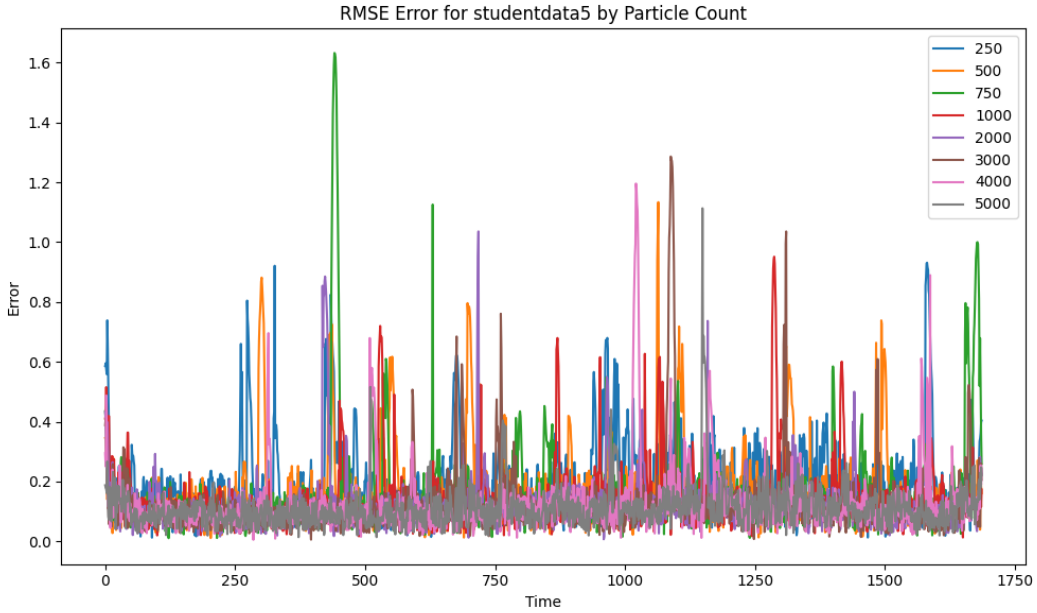

Figure 26: Dataset 5 Particle Count Performance

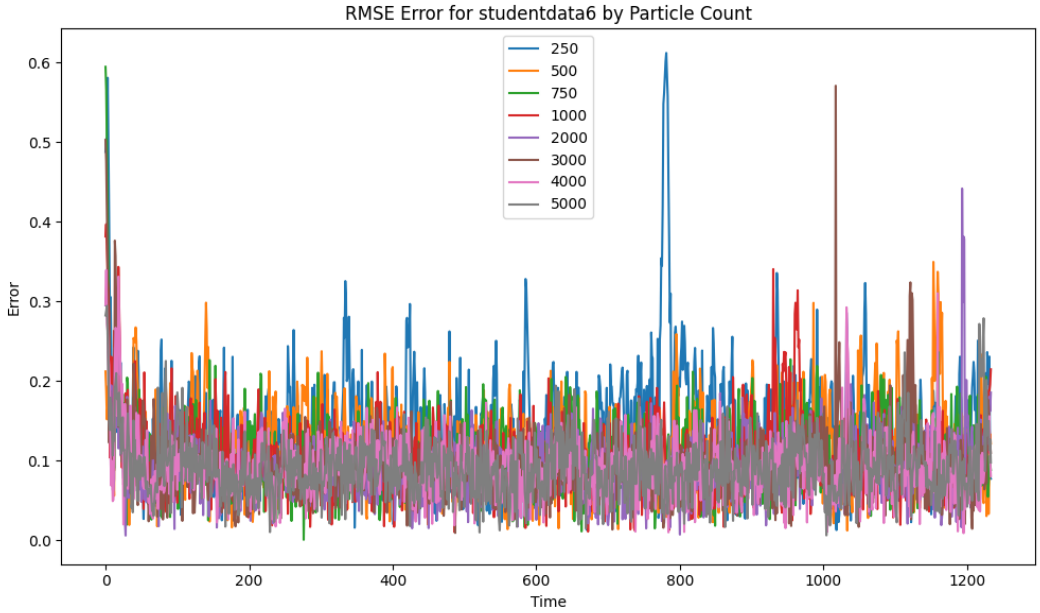

Figure 27: Dataset 6 Particle Count Performance

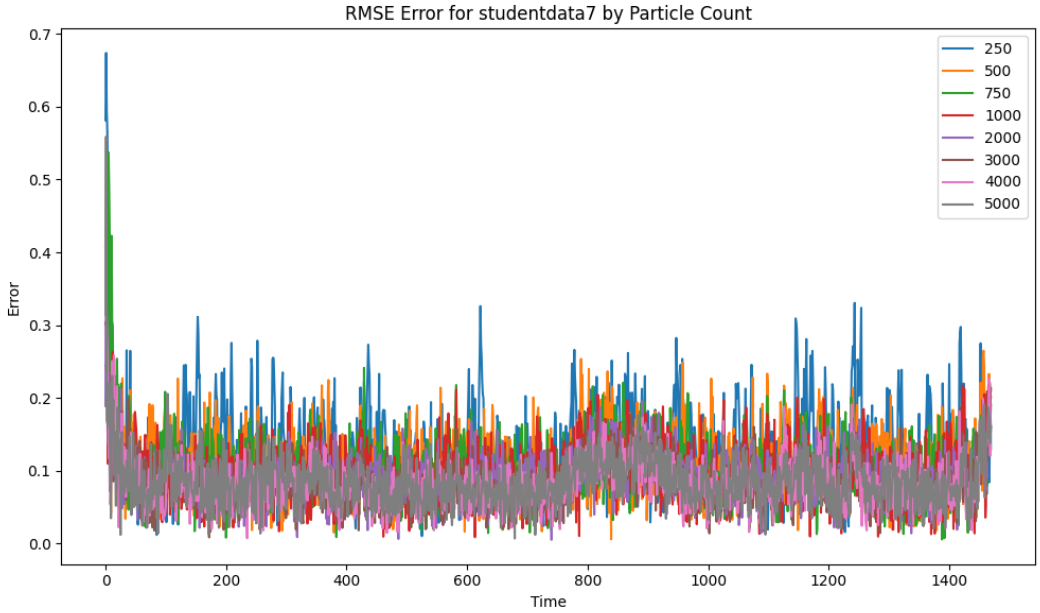

Figure 28: Dataset 7 Particle Count Performance

And our performance numerically presented to make do for the otherwise crowded charts:

Particles	**250**	**500**	**750**	**1000**	**2000**	**3000**	**4000**	**5000**
Dataset 1	0.169	0.145	0.139	0.112	0.108	0.110	0.111	0.114
Dataset 2	0.192	0.159	0.131	0.125	0.118	0.104	0.111	0.107
Dataset 3	0.187	0.143	0.146	0.134	0.119	0.106	0.102	0.096
Dataset 4	0.239	0.204	0.186	0.209	0.151	0.152	0.149	0.157
Dataset 5	0.197	0.159	0.152	0.149	0.142	0.133	0.113	0.122
Dataset 6	0.140	0.110	0.114	0.105	0.094	0.091	0.093	0.089
Dataset 7	0.132	0.109	0.102	0.097	0.089	0.085	0.085	0.083
Average	0.175	0.147	0.134	0.129	0.116	0.110	0.113	0.108

Here we clearly see that performance does improve with more particles - but this is only half the store. We also recorded time to perform across the entire dataset to demonstrate the effect increasing the particle count has on processing time:

Particles	**250**	**500**	**750**	**1000**	**2000**	**3000**	**4000**	**5000**
Dataset 1	4.39	8.04	12.69	16.81	27.85	45.15	57.45	68.63
Dataset 2	2.18	3.84	5.08	6.23	12.17	19.01	25.67	28.69
Dataset 3	5.24	9.29	13.15	16.96	31.00	46.10	59.97	73.82
Dataset 4	4.92	8.77	12.97	18.31	35.33	49.52	65.80	81.54
Dataset 5	8.62	16.04	21.19	29.19	55.45	84.19	107.83	138.57
Dataset 6	5.80	10.67	16.80	21.37	39.68	61.62	78.87	103.59
Dataset 7	8.22	14.92	21.13	24.46	53.17	74.71	98.45	123.14

Here we see, in seconds, the dramatically increasing time to process the datasets as particle counts increase. Further optimization of the code - such as fixing the few non vectorized operations, or abandoning Python entirely for an actually performant language, could help alleviate this. This is a crucial consideration for a particle filter - if the filter is expected to run in real time, we must consider the time performance of its calculations.

For the remainder of this assignment, we will utilize 2000 particles unless specified.

Task 3

In this section, we are tasked with comparing with the nonlinear Kalman filter produced in our prior assignment - for this author, a UKF filter. The filter has been ported into **ukf.py** and is not heavily changed from the prior assignment.

For this, we ran the UKF filter against the 2000 and 5000 particle filter runs utilizing the highest weighted particle method. We then compared the RMSE of the UKF filter against the particle filter.

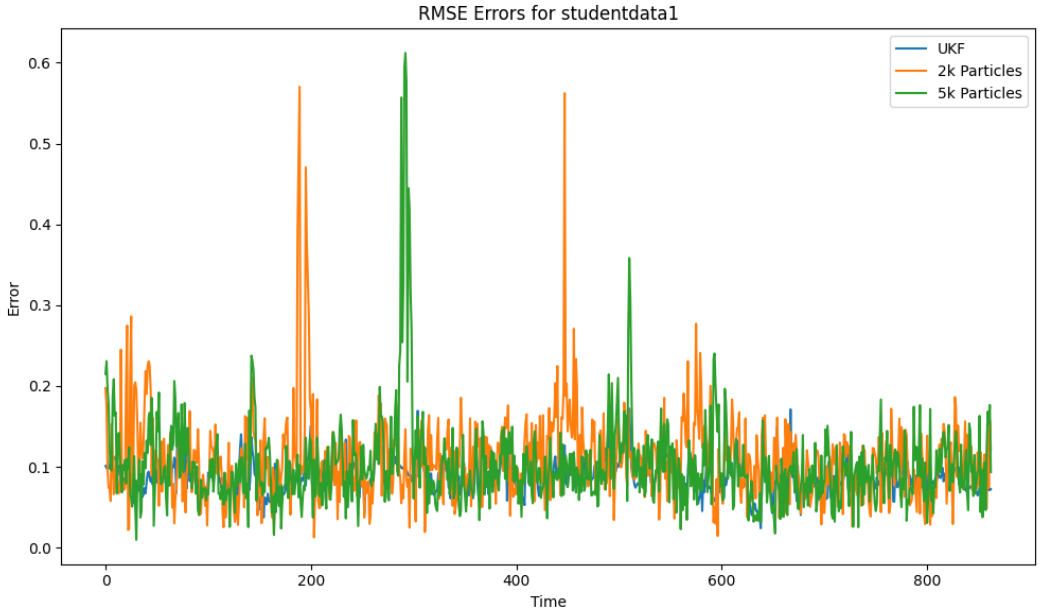

Figure 29: Dataset 1 RMSE Comparison

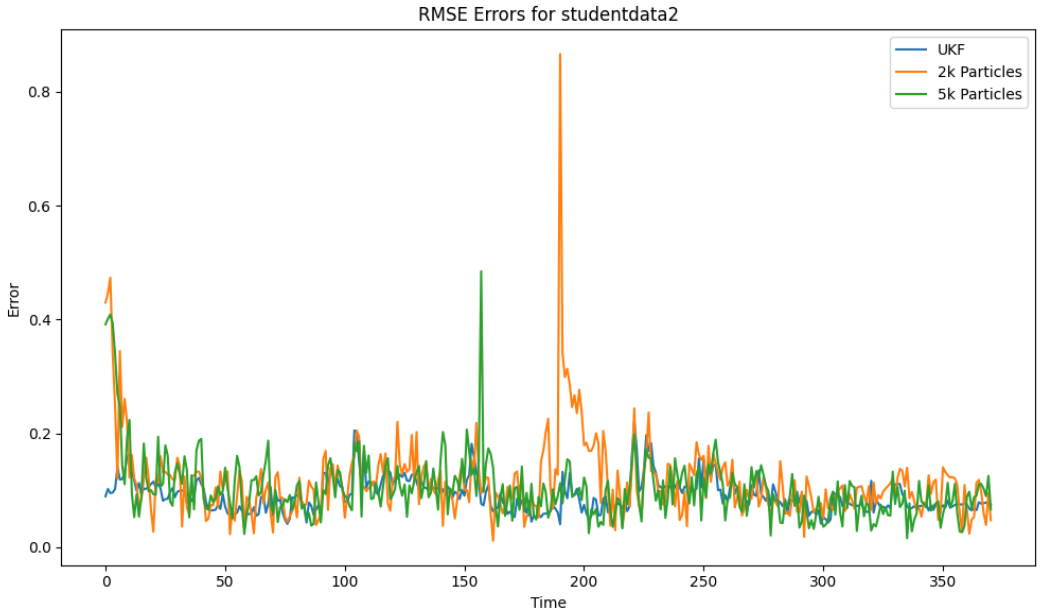

Figure 30: Dataset 2 RMSE Comparison

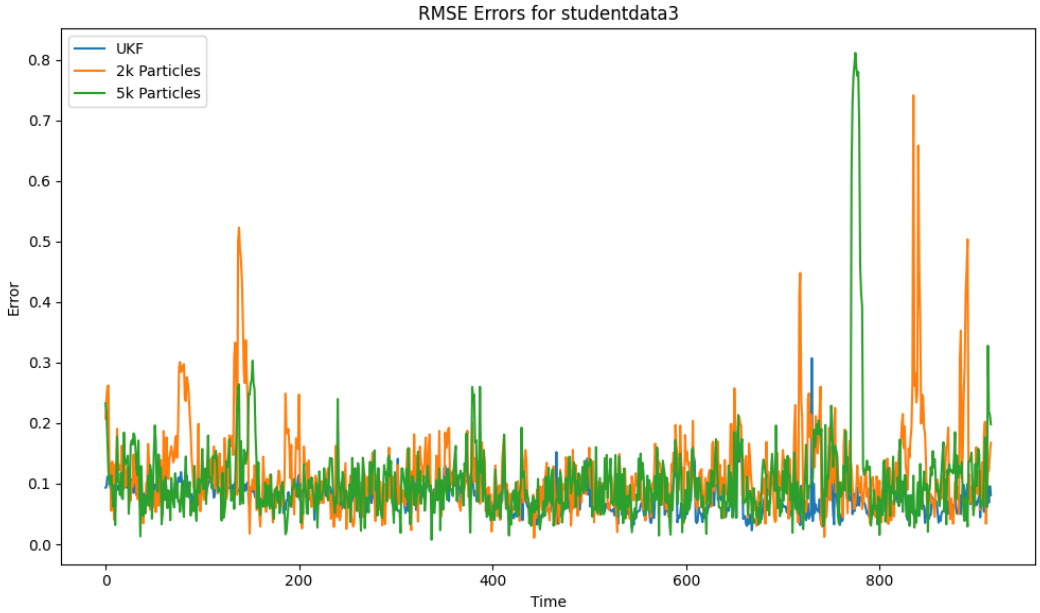

Figure 31: Dataset 3 RMSE Comparison

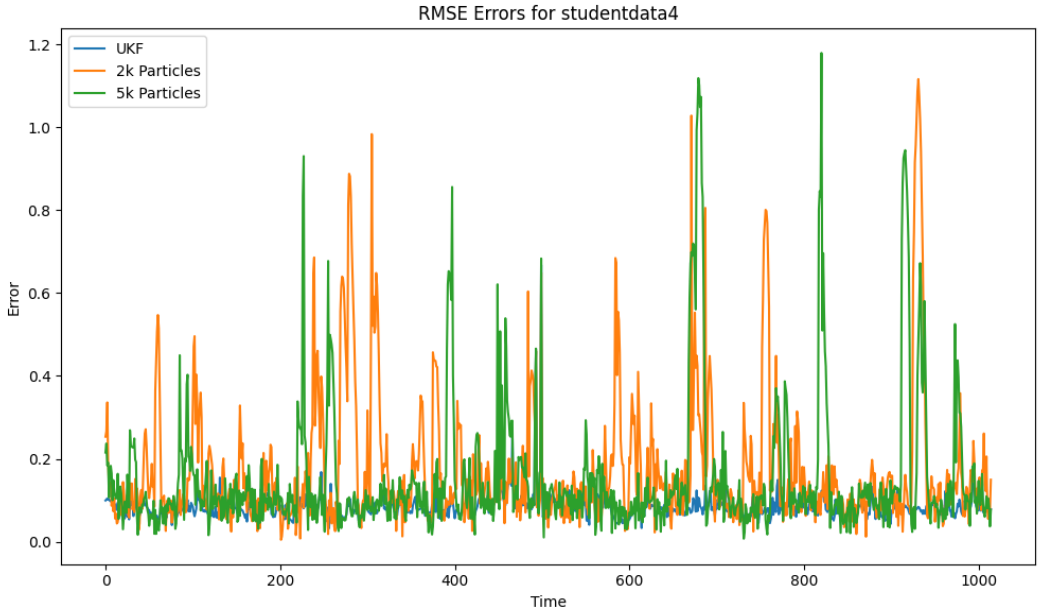

Figure 32: Dataset 4 RMSE Comparison

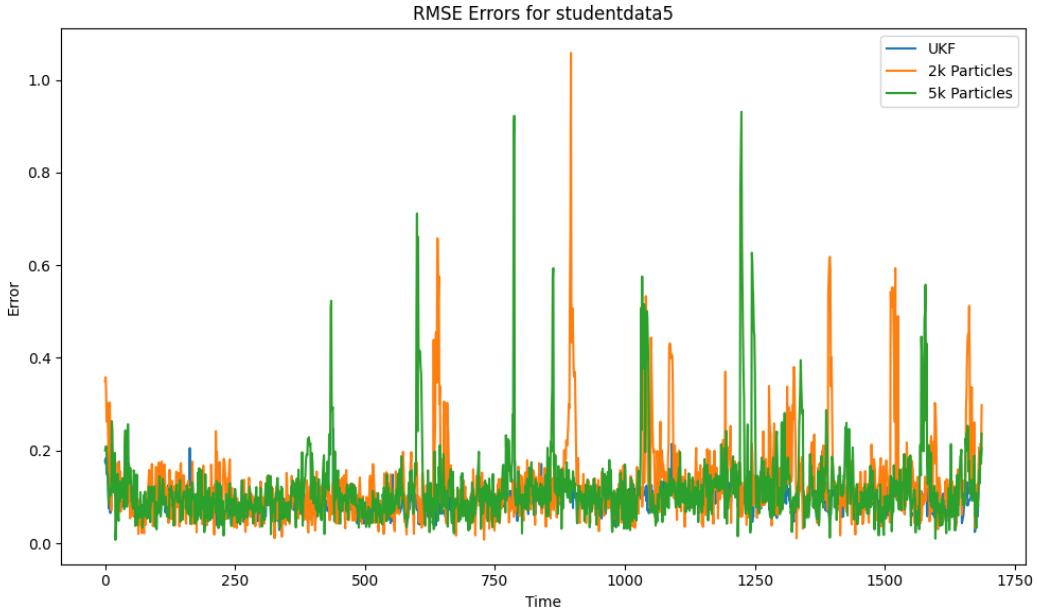

Figure 33: Dataset 5 RMSE Comparison

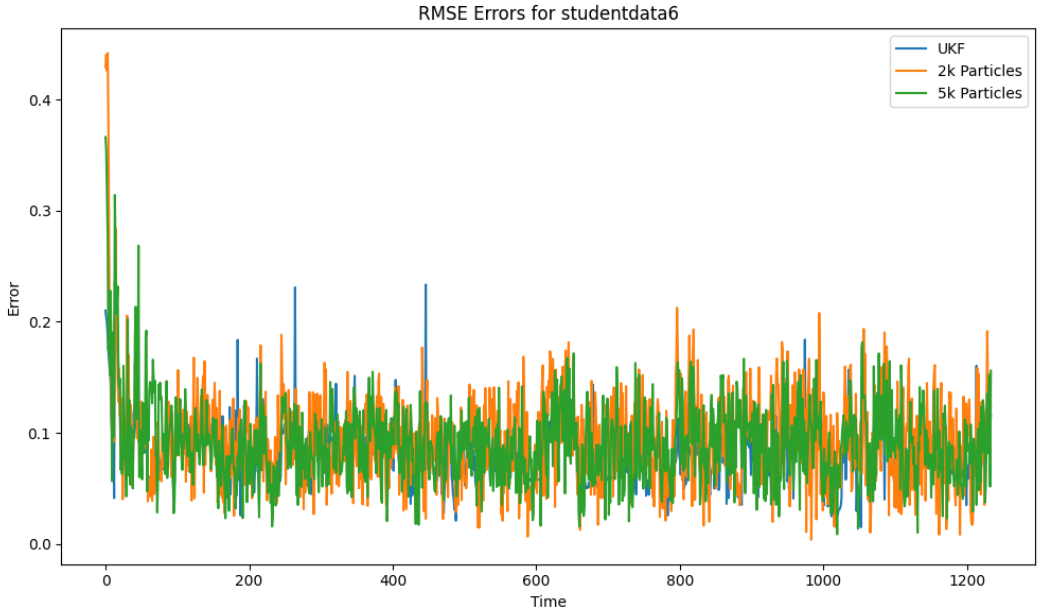

Figure 34: Dataset 6 RMSE Comparison

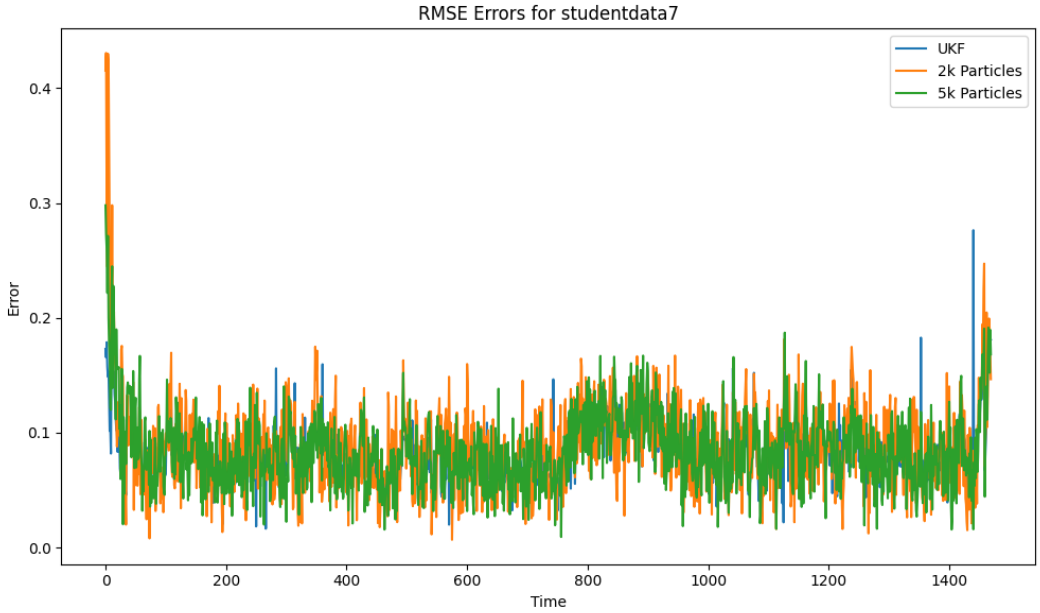

Figure 35: Dataset 7 RMSE Comparison

And our table of RMSE performance:

Dataset	UKF	2k	5k
1	0.087	0.109	0.102
2	0.091	0.116	0.103
3	0.076	0.115	0.105
4	0.084	0.175	0.153
5	0.090	0.125	0.119
6	0.080	0.094	0.090
7	0.082	0.088	0.084
Average	0.084	0.116	0.107

Here we see that UKF does out perform our particle filter, but with the trending downwards error as particles increase, it is likely that more particles would outperform the UKF filter.

Notable, however, is the execution time of the UKf filter versus the particle filter- it takes a few seconds - averaging 3.2 seconds across datasets - to complete a full dataset; an order of magnitude less than the 2000 particle filter, which in turn is twice as quick as the 5000 particle filter.

So when do we utilize one, or the other? First, we consider our state model and target - is it a highly nonlinear system? The greater the nonlinearity, the more likely that UKF will not be able to deal with the system. The particle filter is more resilient to this, and will be a more accurate filter in such a problem.

The next consideration is the time to process - if we are in a system with limited compute and a hard requirement for a high speed filter, we may not be able to utilize a particle filter. While compute continues to increase while size, power consumption, and cost continue to decrease, the number of particles required for a particle filter to perform in a given state space grows dramatically as you expand the state space. Thus a high dimensional state space problem may be computationally prohibitive even on modern hardware, denoting preference towards the UKF.

That being said, this author believes the particle filter was significantly easier to code for, as it did not require significant mathematical understanding or modeling of the system in order to implement. Similarly, the randomness of the particle filter led to much less tuning than our time with UKF, as it was robust through its randomness to the system's noise.