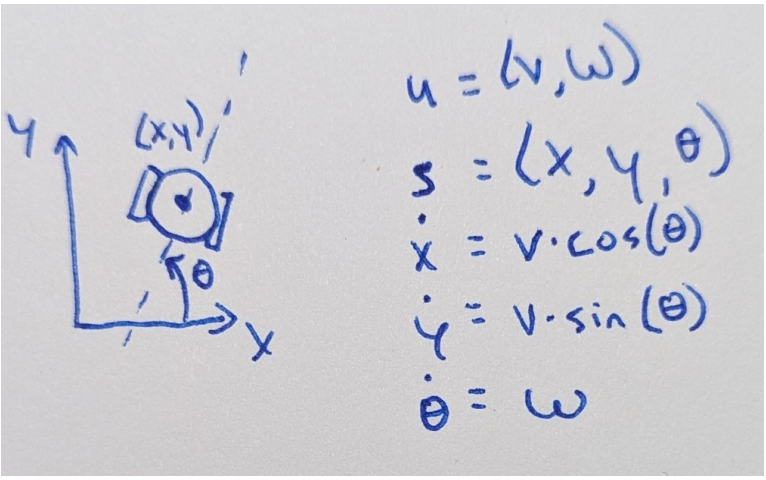
CS/ME 469: Machine Learning and Artificial Intelligence for Robotics
Assignment 0: Filtering Algorithms
Toby Buckley – DS0, Particle Filter

1. Design a motion model to estimate the positional (2-D location and heading) changes of a wheeled mobile robot in response to commanded speed (translational and rotational) controls. Clearly define the inputs, outputs and parameters of this model, and report the math. Explain your reasoning, as needed. Is this model linear in its inputs? Why or why not?

I assumed a simple unicycle model [5] (see image below). The inputs are the control in terms of the translational velocity (v) and angular velocity (w). The outputs are the state in terms of the position (x, y) and heading (θ). The model is not linear in its inputs because the linear velocities are a function of sin and cos, which are non-linear functions.



The gradient is computed from this model, and then a simple numerical integration takes place, where

$$x' = x + \Delta t * \nabla x$$

If a variance is supplied, then the model will also add on gaussian noise, scaled by the time-step, to the new state in the form of

$$x'' = x' + \Delta t * s$$

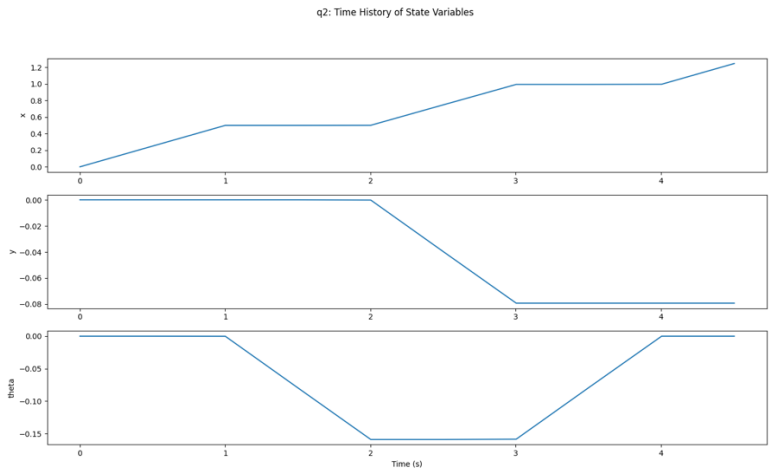
With s drawn from a gaussian with zero mean and covariance defined as a diagonal matrix constructed from the supplied variances for each state variable

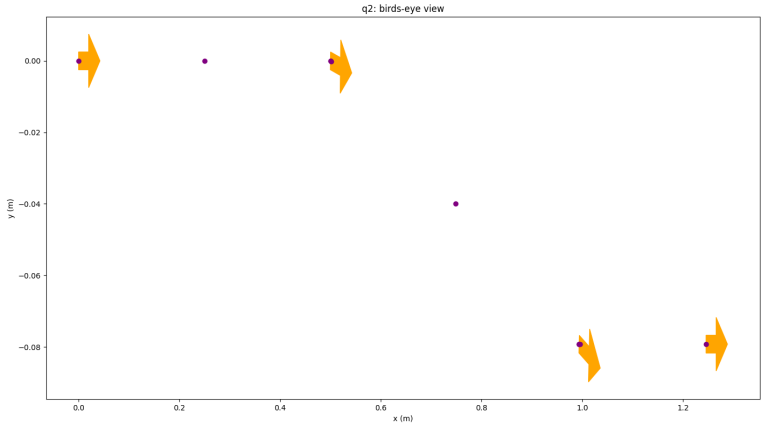
$$s \sim Gaus(0, cov)$$

The parameters are then the initial state, the time-step, and the variances (optional).

2. Implement your motion model in code, and observe the movements it generates for the following sequence of commands. Report the values of any parameters. (Below, ν is translational speed, ω is rotational speed, and t is the duration of the commands.) Report the resulting plot.

The only parameter set was the integration time-step, dt, which was 1e-3. No variance was added. The initial state was all zeros. The time history (below) shows how each state variable evolved. The birds-eye view shows how the robot moved in the physical space (heading denoted using an arrow).

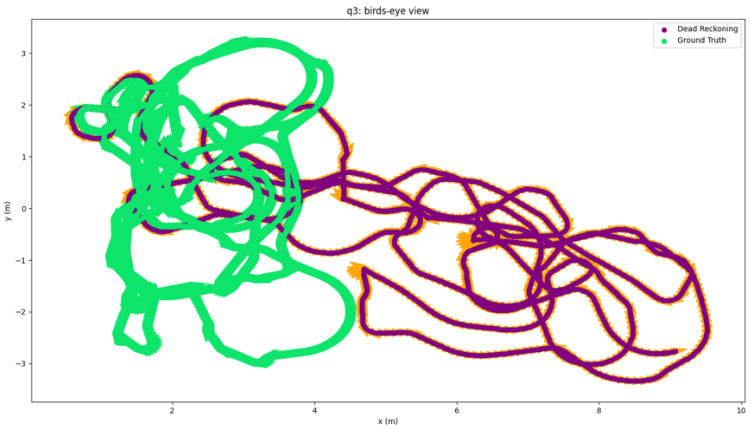




3. Test your motion model on the robot dataset. Issue the controls commands (in _Controls.dat) to your model, and compare this dead-reckoned path (i.e. controls propagation only) to the ground truth path. Report and discuss the resulting plot.

Parameters used: once again a dt of 1e-3 with no variance. The initial state was taken as the first datapoint from the ground truth values. From the time history, we see that the dead reckoning doesn't vary too terribly far from the ground truth, but of course drift due to physical slip of the robot or numerical integration error has led to very large deviations in the x, and y positions. Note that both the shape, as well as the offset of the x, and y history varies compared to ground truth. This implies the problem isn't simply due to constant-direction drift.

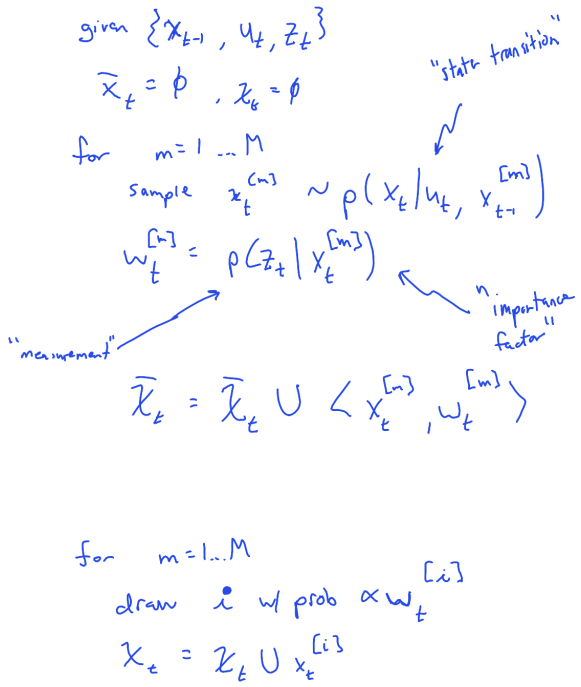
The birds-eye view shows just how grossly inaccurate dead reckoning was compared to ground truth.



4. Describe, concisely, the operation of your assigned filtering algorithm. Include any key insights, assumptions, requirements. Use equations in your explanation.

Particle filters work by modelling a discrete number of particles, each representing a concrete instantiation of the state.
Repeatedly, each particle is propagated forward based on a motion model, then used to estimate a measurement, then used to assess how accurate it is given a real measurement. Finally, each particle is re-assigned to another particle's state, based on the weights of all particles.

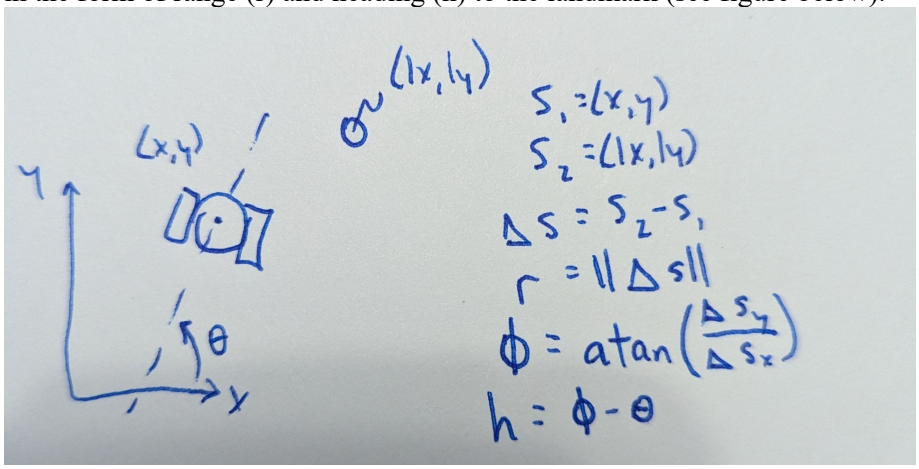
In algorithm form, it looks like this [6]:



Particle filters make no assumption about the form of the distribution, making them more general than Kalman filters, at the cost of additional required computations. They allow for arbitrary motion and measurement models, and naturally work with multi-modal spaces.

5. Design a measurement model for your filter. Report all maths and reasoning; include images for illustration when appropriate.

Given a landmark location (lx, ly) and a state (x, y, θ), the measurement model computes a predicted measurement in the form of range (r) and heading (h) to the landmark (see figure below).



Given this predicted measurement, and a real measurement, the model computes $p(z_t|x_t)$ using a multi-variate gaussian with mean equal to x_t and covariances as a diagonal matrix constructed from input variance parameters. Specifically, the probability density function (pdf), a method of the scipy multivariate gaussian class [7] is used to compute $p(z_t|x_t)$.

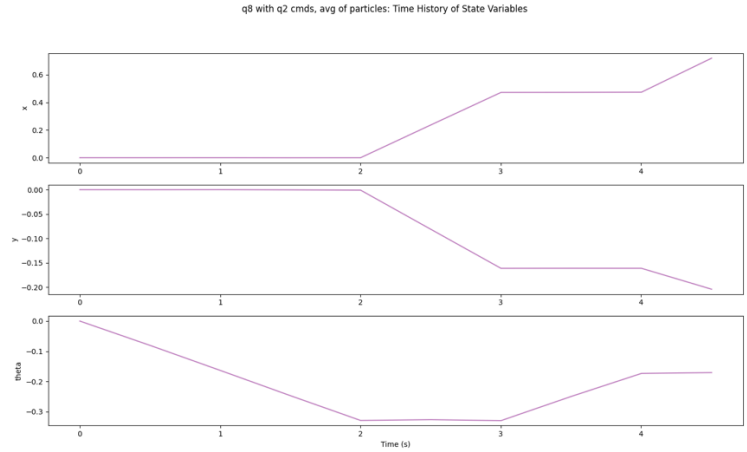
6. Test your measurement model on predicting the range and heading of the following landmarks when the robot is located at the associate positions. Report your results. (Remember, you have access to the ground truth landmark positions.)

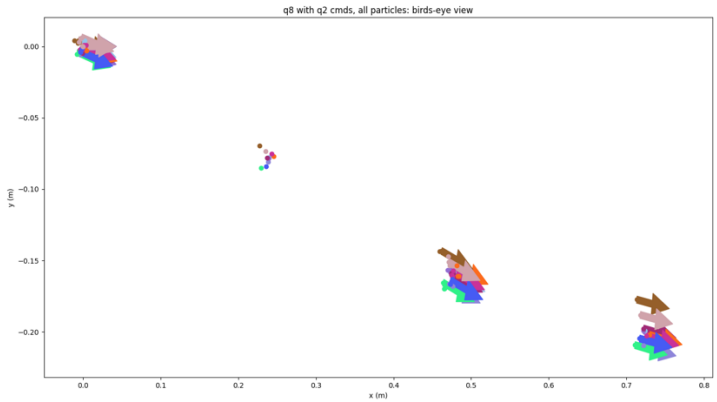
Given the exact state of the robot and the ground truth of the landmark's, it is relatively straight forward to estimate a range and heading from the robot to the landmark. My solution doesn't include measurement error, as that's taken into account in the particle filter when computing my measurement weights.

Landmark #6, with respect to the robot at 2 m, 3 m, 0 rad had an estimated range of 8.09 m and heading of -1.76 rad
Landmark #13, with respect to the robot at 0 m, 3 m, 0 rad had an estimated range of 2.57 m and heading of -1.21 rad
Landmark #17, with respect to the robot at 1 m, -2 m, 0 rad had an estimated range of 5.06 m and heading of 1.11 rad

7. Implement the full filter.
Please see the submitted code for full implementation details.

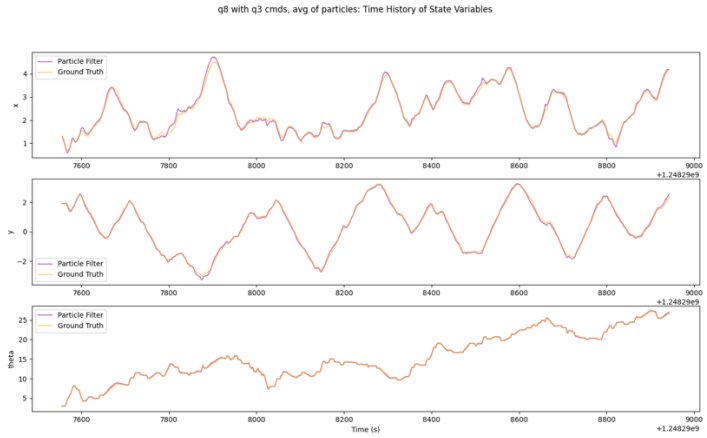
8. [*] Compare the performance of your motion model (i.e., dead reckoning) and your full filter on the sequence of commands from step 2, and also on the robot dataset (as in step 3). Report the results, explain any differences (what performs well? what performs poorly? under what conditions? and why?). Ground your explanations in the algorithm maths.
Parameter values used: motion variance: 0.01, measurement variance: 0.001, max dt: 0.001
When running the step 2 commands, no measurements were given so the feedback part of the filter cannot be used. Still, when visualizing each particle in the birds-eye view, it's clear that the various particles are moving along the assumed trajectory with minor deviations, due to the noise in the motion model.

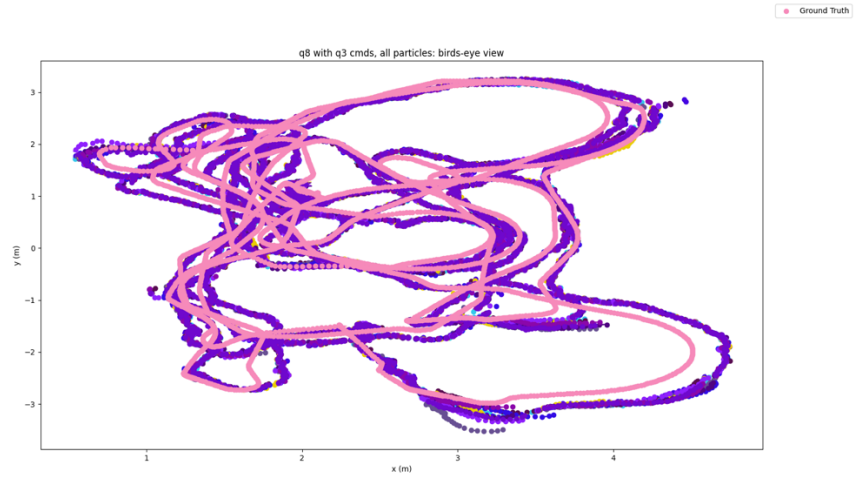




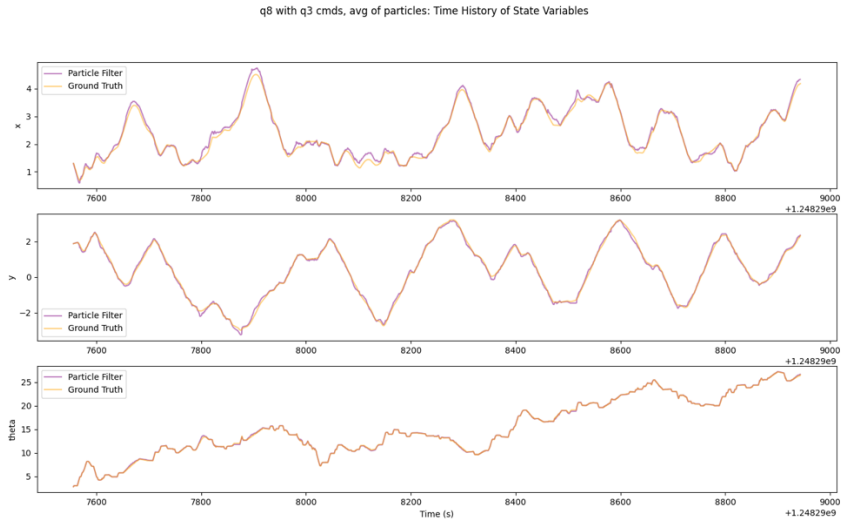
With the step 3 commands, we now have measurements to feed back into our filter
I limited the number of particles to 10 for the full trajectory since I have a weak laptop and the execution time was becoming untenable.

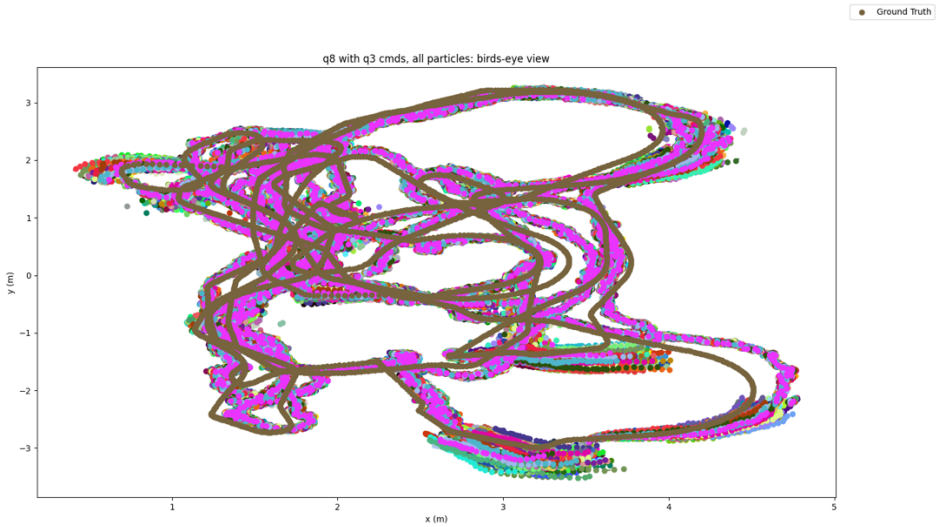
Parameter values used: motion variance: 0.01, measurement variance: 0.001, max dt: 1.0





A test with 200 particles was run:



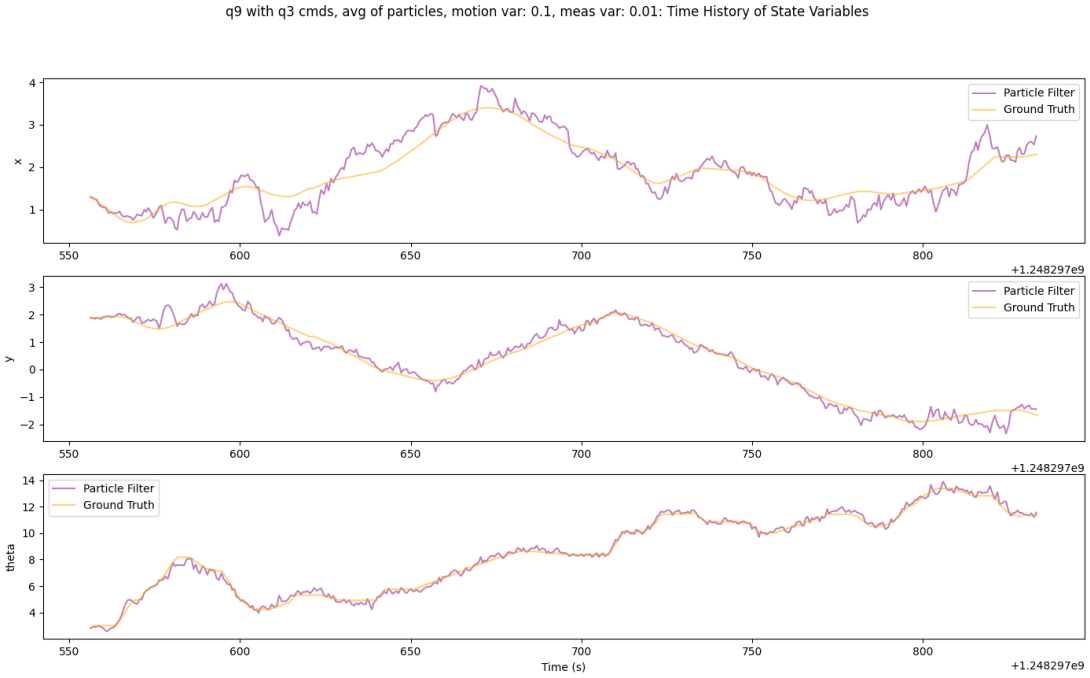


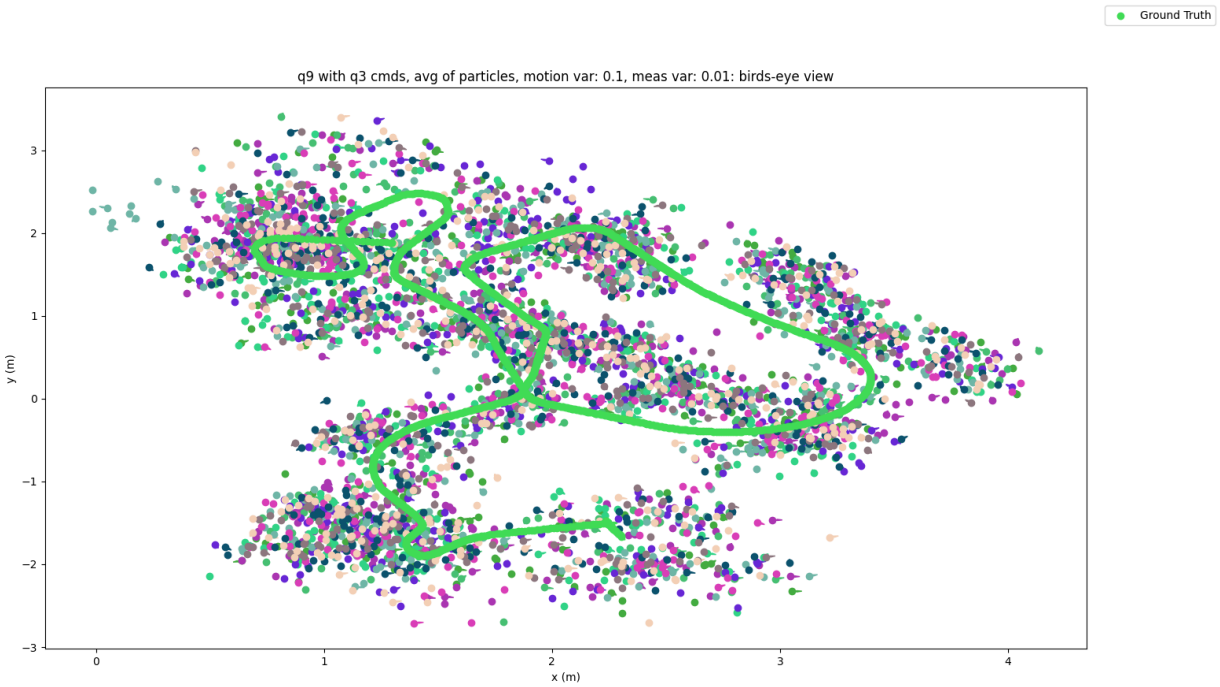
Surprisingly, number of particles didn't have a massive effect on my time-history error. It simply slowed down execution time significantly.

9. [*] Examine the effect (trends? limits?) of uncertainty in the algorithm (i.e., changing the noise parameters). Examine its performance at different parts of the execution (e.g., when missing an expected measurement reading). Report plots and/or tables, and provide explanations.

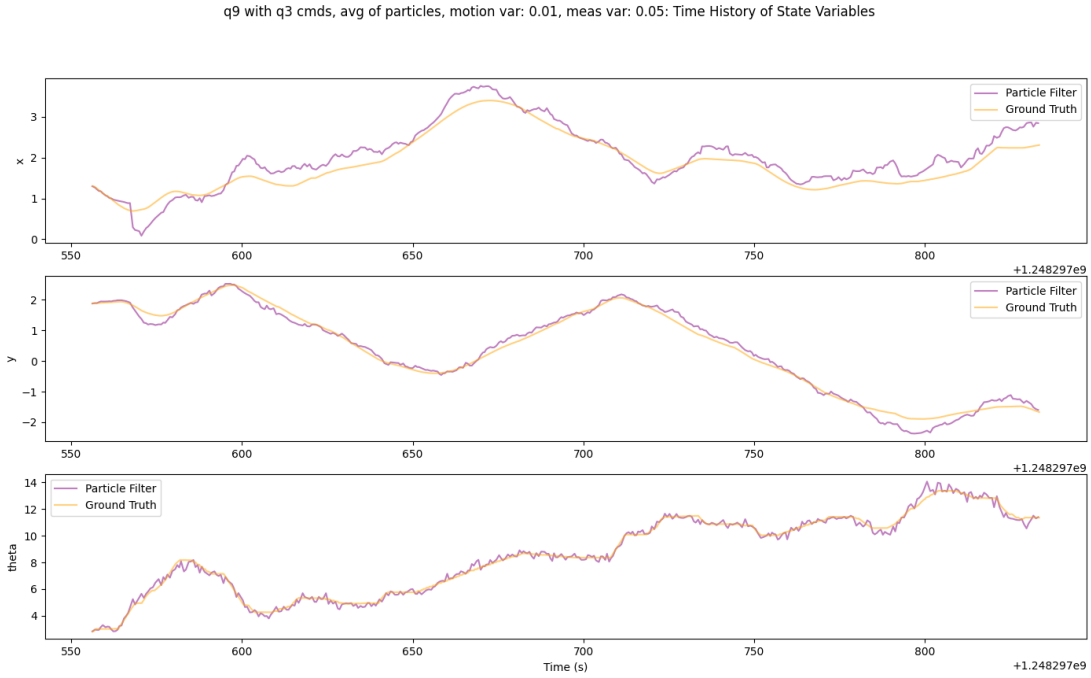
In order to execute the sweep with a reasonable execution time, only the first 20% of the trajectory is traversed, as well as only 10 particles are used.

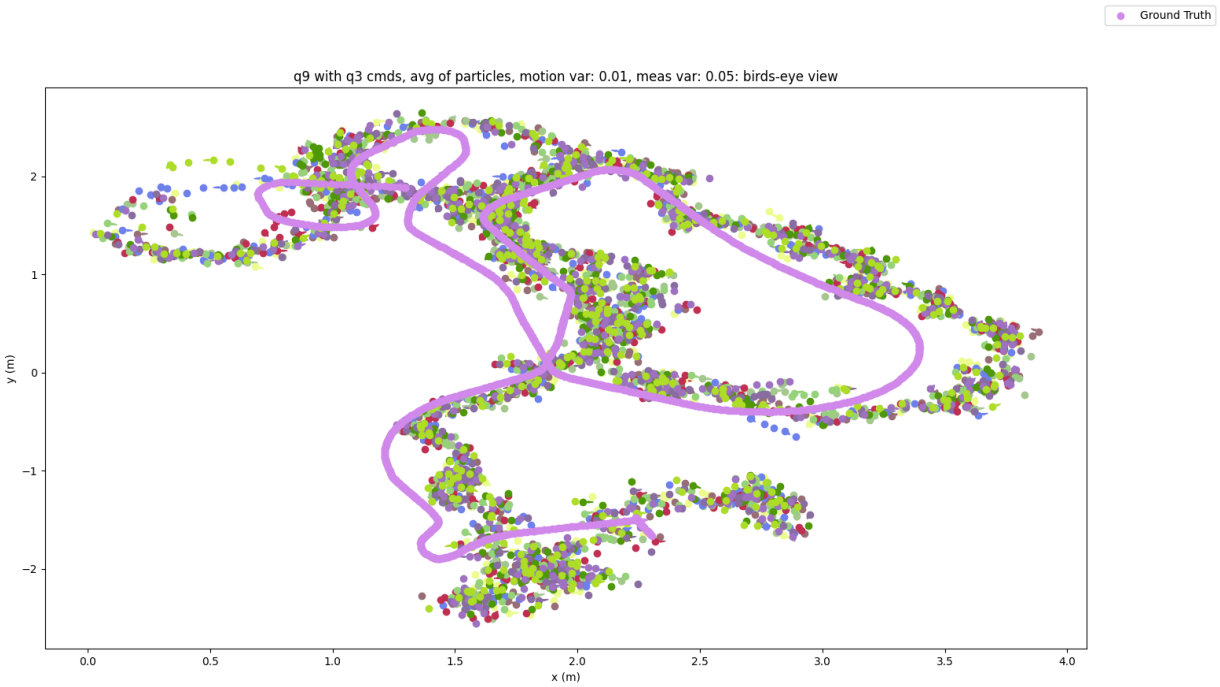
q9 run 1: motion var 0.1, meas var 0.01, max dt 1.0, number of particles: 10



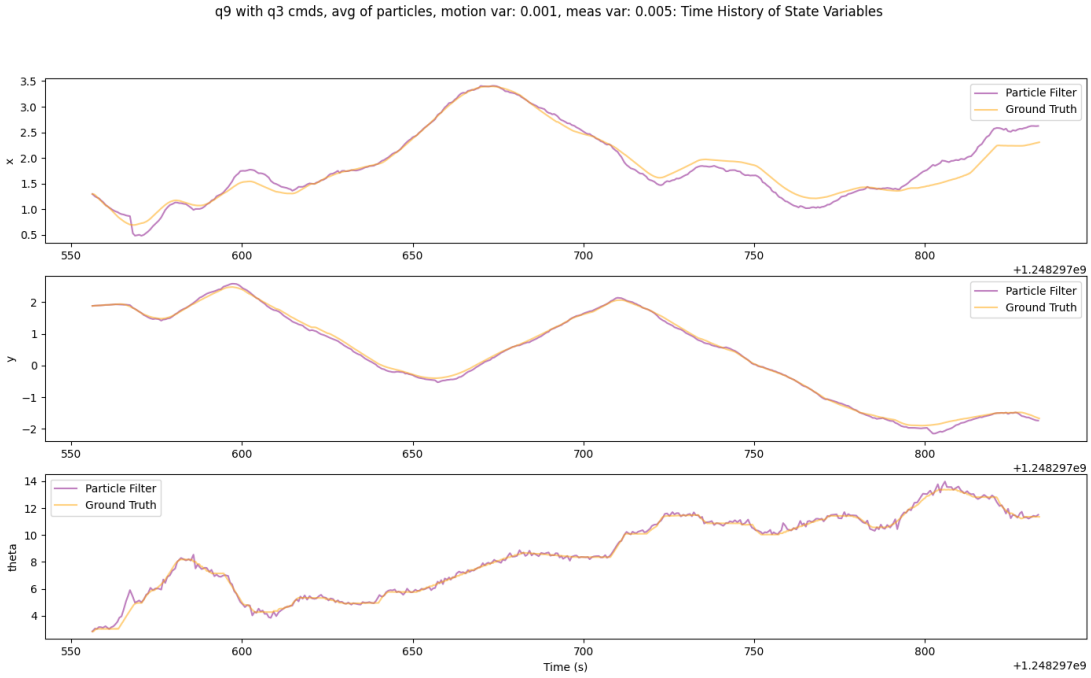


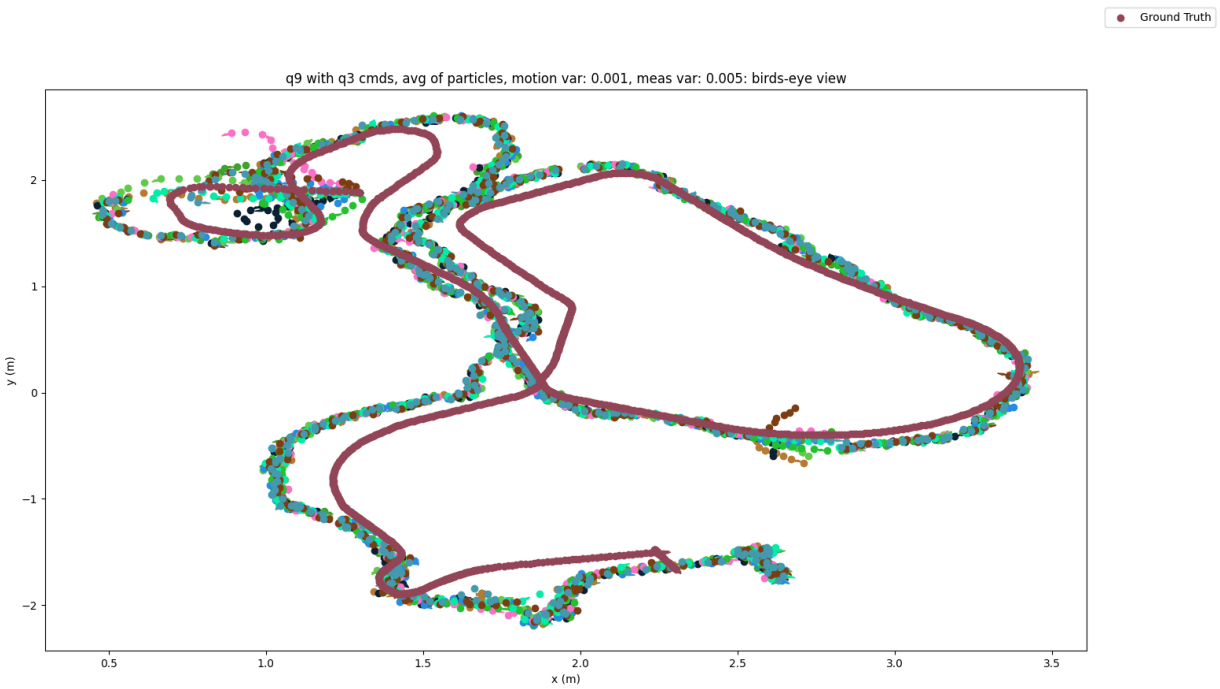
q9 run 2: motion var 0.01, meas var 0.05, max dt 1.0, number of particles: 10





q9 run 3: motion var 0.001, meas var 0.005, max dt 1.0, number of particles: 10





From this small sweep of variance values for the motion model and measurement model, we can make some insights as to the effect of high and low variance for both. When the motion model variance is high, then we see a huge distribution of states for each particle for each time step. When the motion model variance is too low, then the motion model loses its stochasticity and there's a higher likelihood that particles will collapse onto each other as the trajectory progresses.

When the measurement model variance is too high, then the weight for each particle becomes uniform, meaning that each particle has equal likelihood of being selected regardless of how accurate its predicted measurement is. Conversely, when the measurement model variance is too low then the weight for one particle will be very high, and all other particle's weight's will be very low. This can lead once again to all other particles collapsing onto the particle which is most similar to the measurement. This may seem like a good thing, but may give too much weight to any single measurement. And since each sensor from which the measurement comes from has noise, it's unwise to give any single measurement too high of weight.

Lastly, my particle filter only redraws when a new measurement is processed, so if measurements are missed we would expect the drift to worsen and for that part of the trajectory to behave more similarly to the dead reckoning trajectory.

References:

[1] https://stackoverflow.com/questions/15927755/opposite-of-numpy-unwrap
[2] https://numpy.org/doc/stable/reference/random/generated/numpy.random.choice.html
[3] https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html
[4] https://stats.stackexchange.com/questions/49160/particle-filtering-importance-weights
[5] https://msl.cs.uiuc.edu/planning/node660.html
[6] Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. Probabilistic Robotics. Cambridge, MA: MIT Press, 2006.
[7] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.multivariate_normal.html