Probabilistic Reasoning over time (Labb 2)

Kalle Josefsson (ka5532jo-s)

A short introduction to the problem: The task was to predict the position of a robot in a grid based on a sensor reading the robot after each step it took, this reading could also fail and we did not get a reading at all. The robot had two different sensors, one which had a constant probability of not getting a reading and the other constand probability of not getting a reading. Based on the reading the robot had different probabilities of being at certain points in the grid. The probability of the sensor being correct was 0.1 for both the sensors while the probabilities for being in any of the surrounding positions, and the positions surrounding those were set to a constant for the sensor with non uniform error hence changing the probability of error based on the position of the sensor. For the sensor with constant error the probabilities of being in the first and second outer layer depended on where the sensor reading was and the error probability was constant to 0.1. The other aspect that affected our guesstimates was the transition model for the robot. For each position in the grid the robot could be in 4 different states based on the heading of the robot, i.e the direction it was “looking” in. Given that the robot was not facing a wall the probability of it continuing straight forward was 0.7 and for going in another direction was 0.3 divided by the amount of possible directions it could go in without hitting a wall. Given that the robot was facing a wall the probability of going the other directions were set to 1.0 divided by the amount of possible paths, which was two or three depending if it was in a corner or just facing a wall o a side. Now I think we have all the basics put down so let us discuss how we solved the problem.

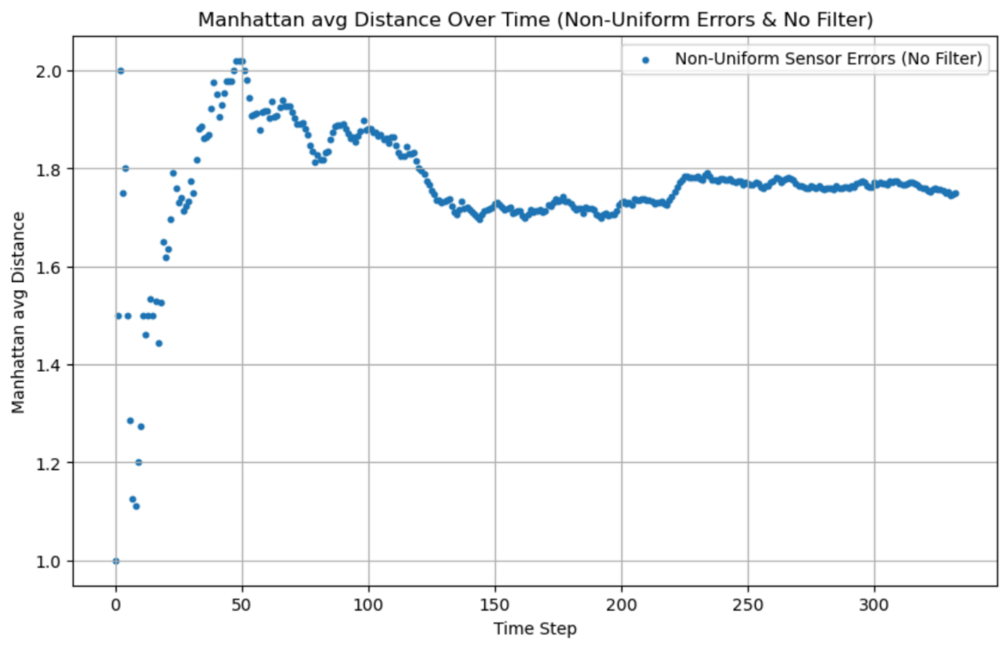
It took quite some time to understand what and how to do it, first I had to understand how the sensors and how the given code worked. Which I did after carefully analyzing it, discussing with a classmate Vidar Tobrand (vi5442to-s) and trying it out in the GUI which showed the different observation models based on different sensor readings as well as an image of the transition model. Now to the actual task, we were supposed to increase the accuracy of classifying the position of the robot based on sensor readings which only had a 10% chance of classifying it correctly. We were supposed to do this in two different ways for a number of different grid dimensions with the different sensors and compare the results. The first method was just to make a prediction on the current sensor reading and make a prediction. This was made by implementing a filter which returned an array with the probability for each possible position the robot could be in based on the observation model of that sensor reading, the transition model and the latest prediction. As you understand we progress forward in time and make predictions based on past states hence we need a base case, which in our case was an array with values that were equal, the probability of each state was the same since we did not have a previous array to base our prediction on. But when traversing through time we should get “none” readings and in that case our observation model just becomes the last observation model. After getting an array we just take the index of the max value in the array and we get our prediction. Now this model actually worked well but we can still improve it by continuing taking steps and then making a prediction since if we got a hum of where we are after a few steps it is easier to calculate where we were before. This is done by sending a backward message so we use information from both the past and the coming steps, this is called the forward backward with fixed lag and smoothing. The backward message is calculated using the bayesian theorem and matrix multiplication, using the observation models from, in our case 5 steps ahead, and traversing backwards to make our prediction. Thankfully this worked even better than normal forward filtering since normally more data means more accurate predictions.

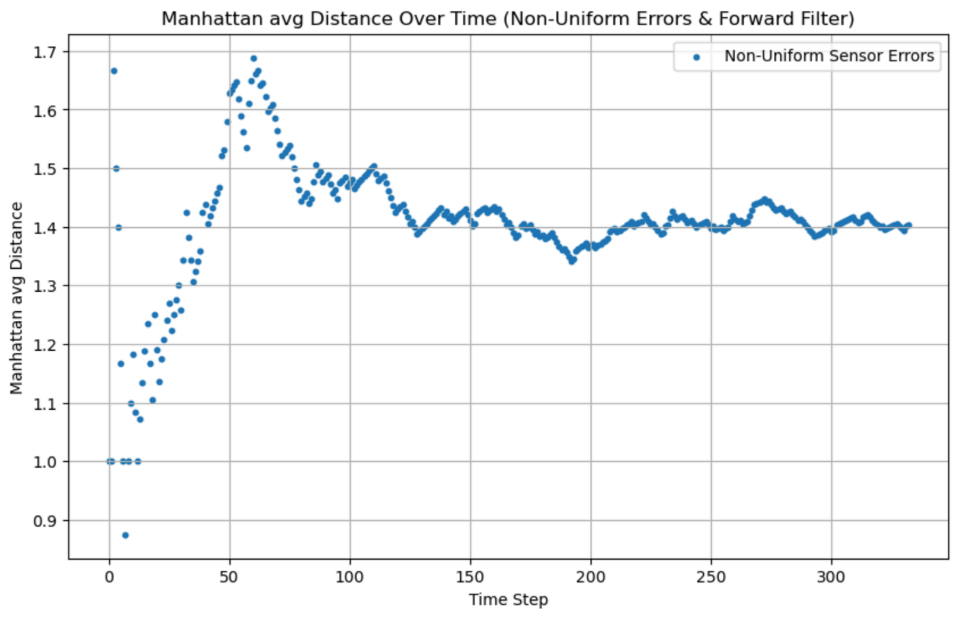
After making the implementation I did a peer review (el8726lj-s) and got my solution reviewed. We had done essentially the same theoretical implementation but the code was very different. He pointed out that I made my matrix operations in the forward awkward filter in the wrong order which was a careless error by me which I corrected and a happy surprise was that my results improved drastically. Then I gave him some input on how to present the results and what they meant. Instead of plotting the Manhattan distance for every step, plot the average Manhattan distance over time and see what it converges towards which is a better measure of how well the model is working.

Before going into the discussion of the results I would like to make one note for the code which is that since I implemented it in the way I did, you have to choose which task to run and insert in the localizer it is commented on in the code. The other thing I would like to point out is that it takes quite some time to run the 16x20 grid but it does eventually get there. There is also some small bug which does not affect the program per say but is that “numpy.ndarray” is printed out a lot of times when you run the notebook but it should be fine you just have to scroll down; I could not find where the bug was so unfortunately I had to leave it there. Sorry for the inconvenience.

Now to the interesting part, the results that were found. Below the plots to task one, two, three and four is presented. The robot walks the same path for each pair and for the last task the sensor readings are the same as well. In the first plot I do not

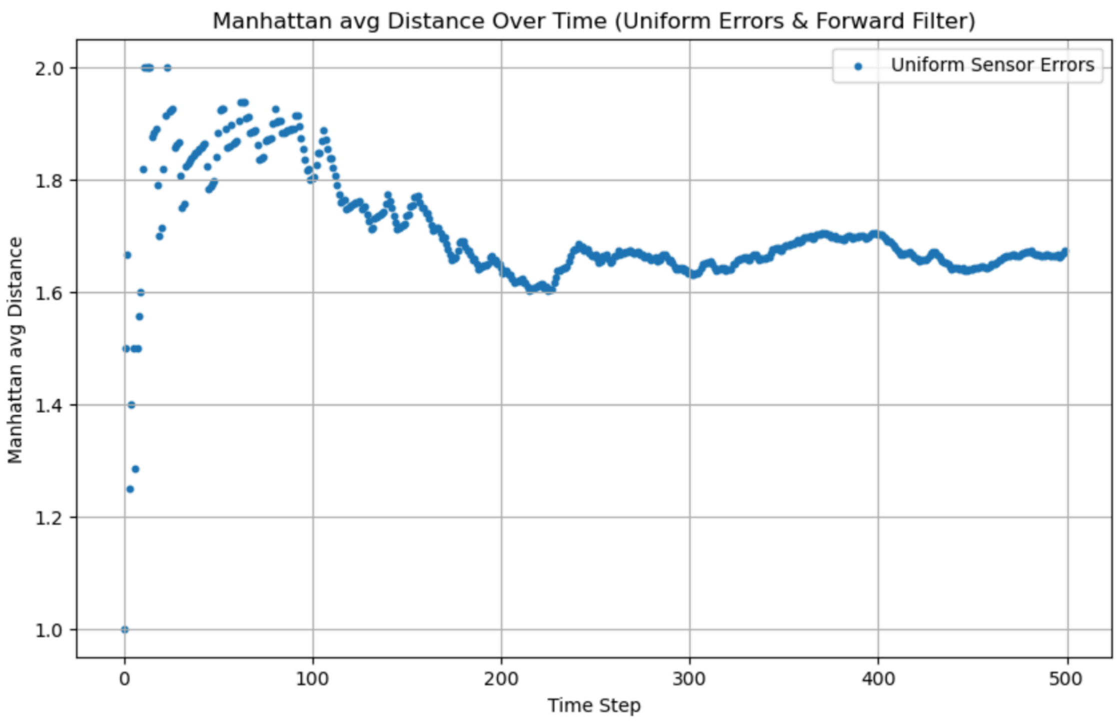
***Results task 1**: Without filter vs Forward Filter 8x8 grid (Non uniform error)*

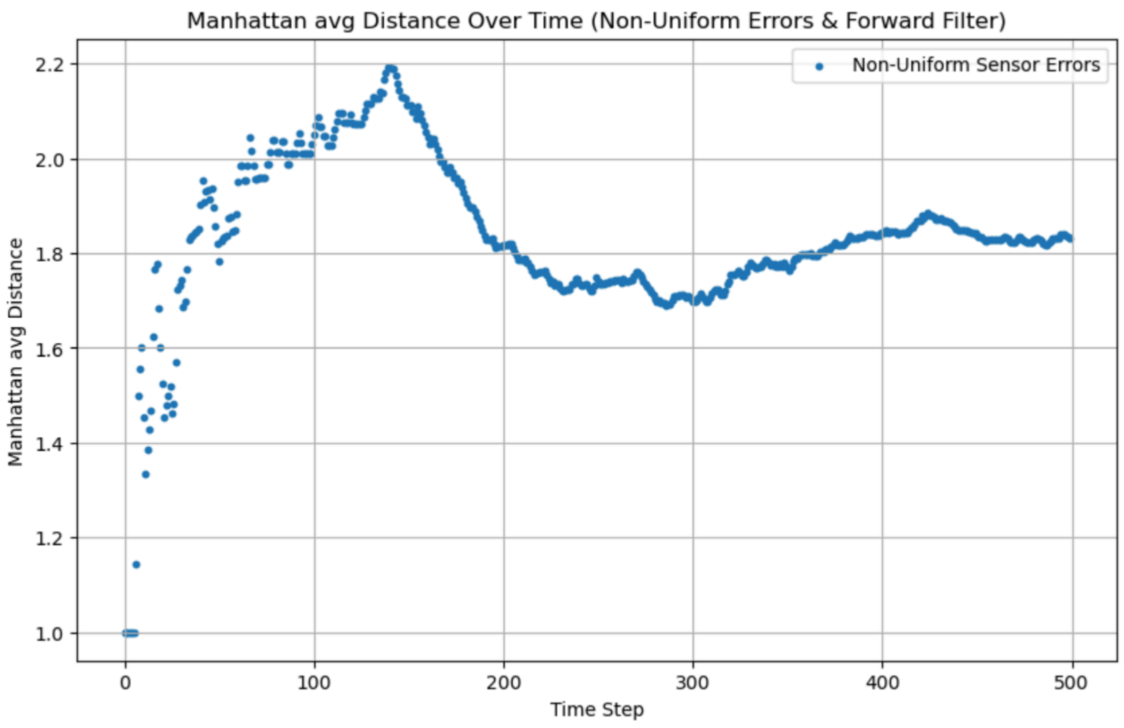




Total errors: 167/500
Forward filter: Accuracy: 0.3843843843843844 & Mean Manhattan : 1.416
Sensor No Filter: Accuracy: 0.12312 & Mean Manhattan: 1.7822

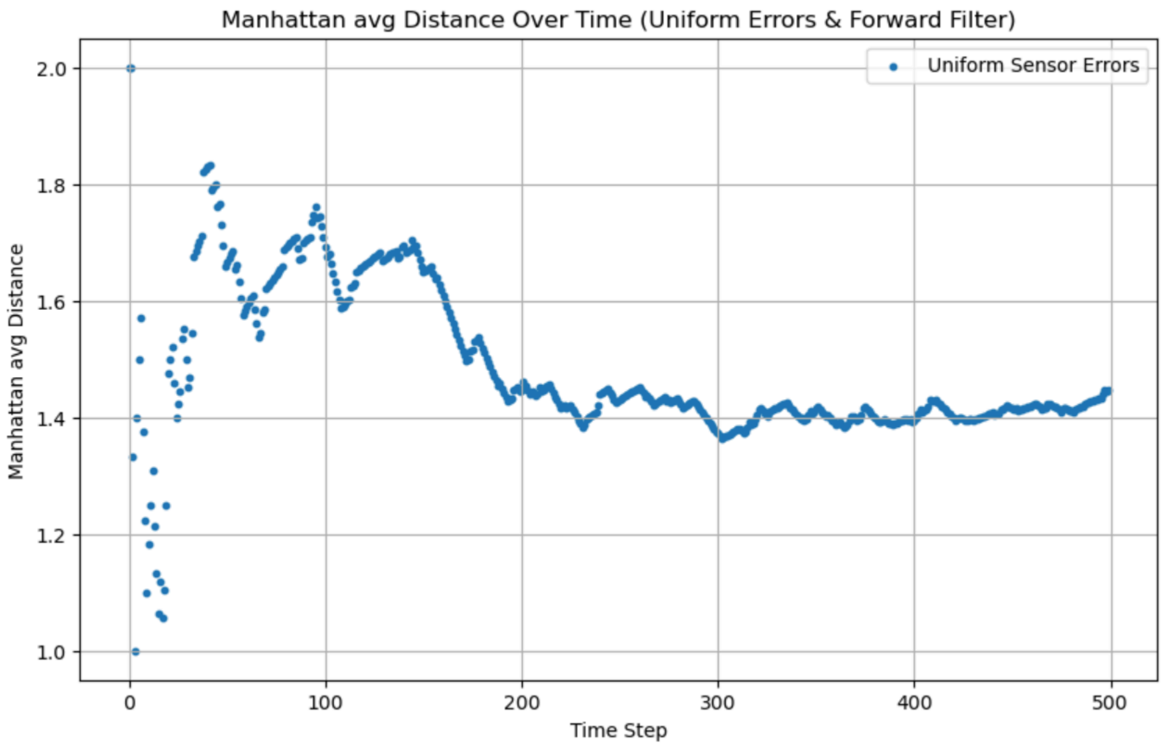
Results task 2*: Forward filter (Uniform error) vs Forward Filter (Non uniform error) 4x4 grid*

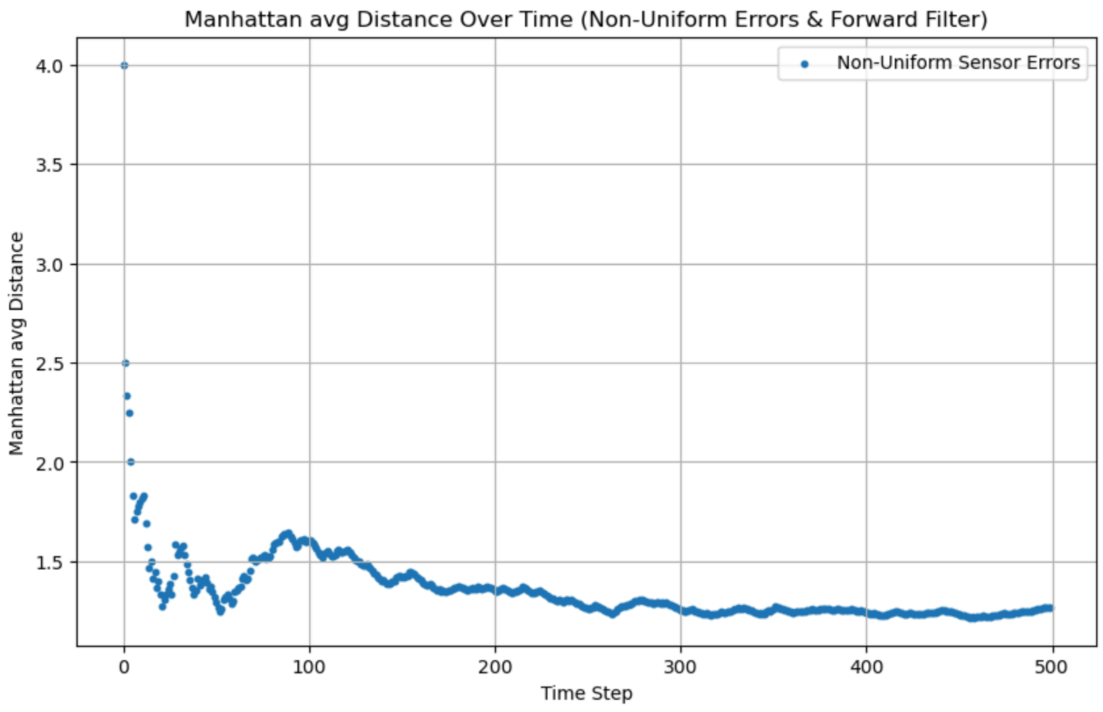




Forward Filter UF: Accuracy : 0.286 & Mean Manhattan: 1.672
Forward Filter NUF: Accuracy:0.304 & Mean Manhattan: 1.83

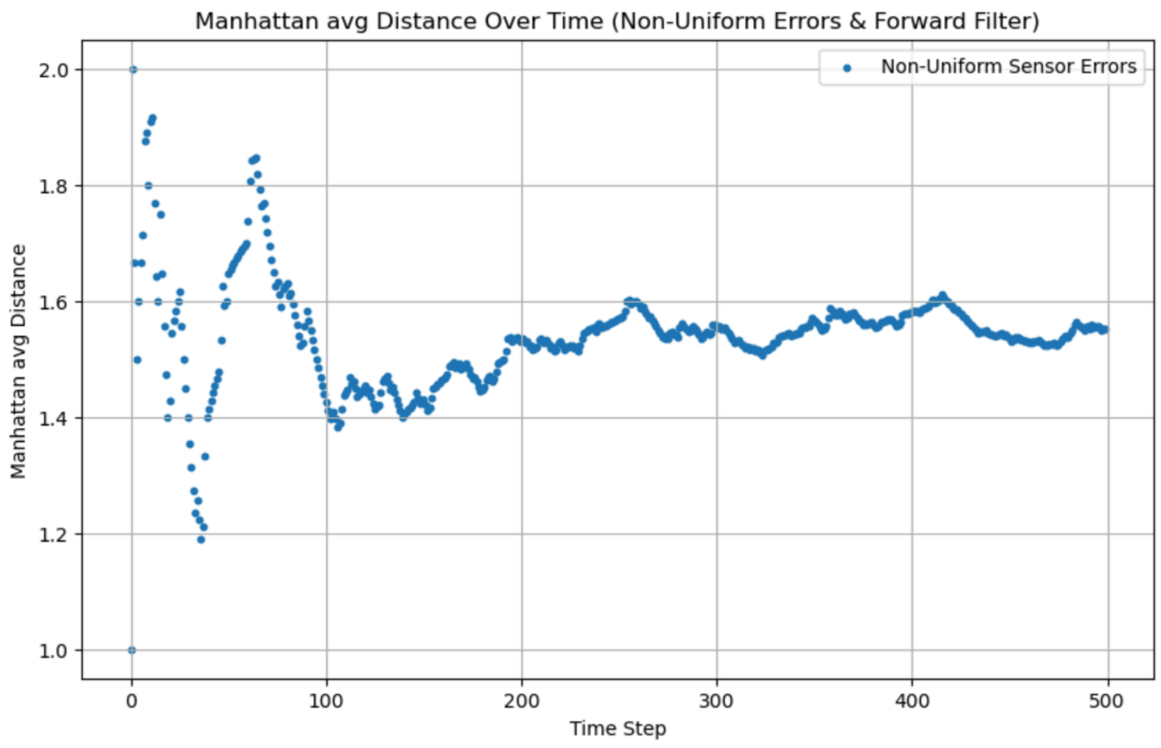
Results task 3: *Forward filter (Uniform error) vs Forward Filter (Non uniform error) 16x20 grid*

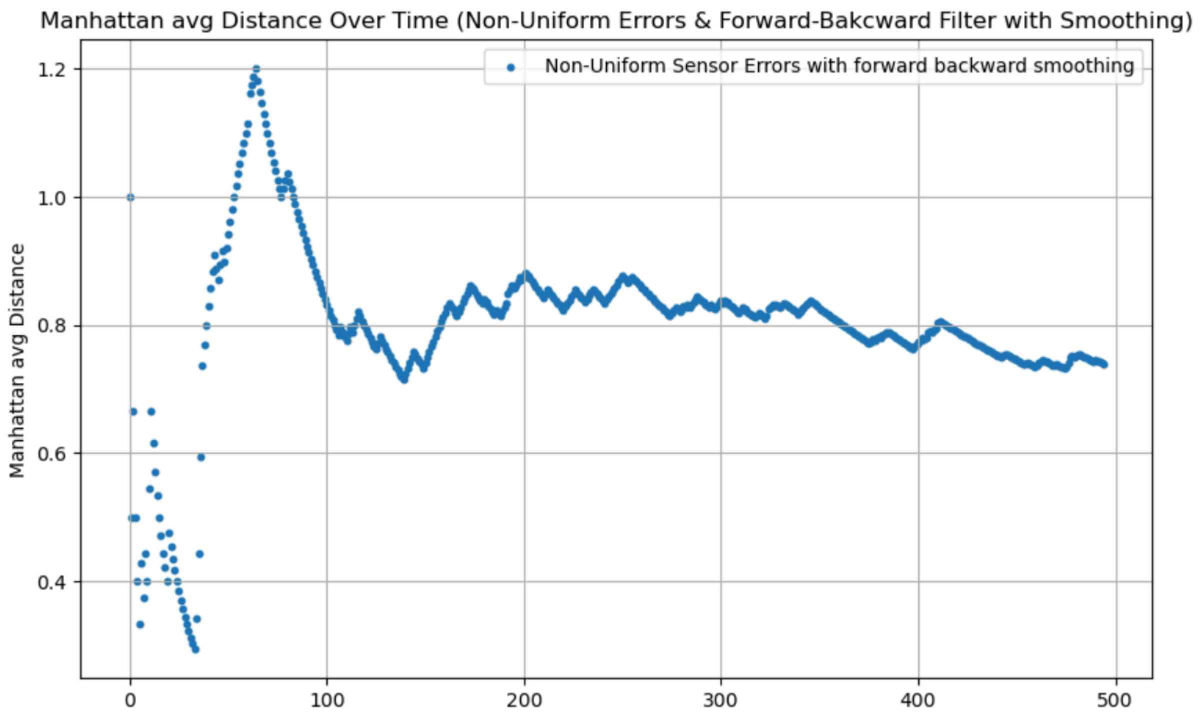




Forward Filter UF: Accuracy : 0.392 & Mean Manhattan: 1.446
Forward Filter NUF: Accuracy:0.456 & Mean Manhattan: 1.268

Results task 4: *Forward filter vs Forward Backward Filter Smoothing Fix Lag (Non uniform error) 10x10 grid*





Manhattan avg Distance Over Time (Non-Uniform Errors & No Filter)

Forward Filter: Accuracy: 0.386 & Mean Manhattan: 1.552
F-B Filter w Smoothing: Accuracy: 0.65050 & Mean Manhattan: 0.73939

As we can see from the results all models do better than the actual sensor reading which just has an accuracy of 10% and averaged a Manhattan distance of 2 across all measurements, given that a reading was made, which is still pretty good, a lot better than a straight guess since that would just be one divided by the number of available positions. If we look at the last two plots we can see that the smoothing filter gave an average Manhattan distance of less than one around 0.75, this was even lower in some runs. I think the lowest average distance I obtained was around 0.65 and an accuracy of upwards 0.75 which is really good compared to sensors.The best aspect of the forward backward filter is that it handles the failed sensor readings well since it can use the next steps it can deduce where it came from even if we did not have sensor reading in that timestep. However the computation time is pretty long especially for a 16x20 grid but I would still say it is worth it since it does actually work pretty well. Now if we compare using the same forward filter on the same run but with different sensors we can see that the results do not differ that much especially when we have a bigger grid since the risk of failing does not become as big for the non uniform failure sensor due to the fact that the probability of being in a corner is smaller the bigger the grid and therefore the accuracy and average distance is roughly the same, given that the robot is in a position which has no corner in the other two sections surrounding the sensor work exactly the same so for the bigger grids it should not matter which one we use. However for the smaller grids the non uniform failure has a much larger chance of sensing none and therefore the accuracy of it decreases and the uniform failure sensor works better. Hence the Uniform failure is better opted for smaller grids since the probability for failure is constant while the non uniform failure sensor is more opted for larger grids since the probability of failure decreases to 0.1 most of the time. This was clear when using the GUI on the 4x4 grid and testing between the two sensors and from the results I obtained when running the program a couple of times. Note that in the 4x4 grid the average Manhattan is smaller but accuracy is also lower this may be due to the fact that the probability of the sensor being in on of the position in the first outside layer of the sensor is higher than for the second layer for the non uniform while it is the same probability for the uniform, 0.4 divided by the number of available positions for both the first and second outer layer.

Now to discuss the relation between my implementation and the work described in the provided article. The HMM that I implemented for locating the robot works quite well in discrete environments but I think unlike the MCL it would not work very well in more dynamic or complicated spaces. My model assumes the behavior is Markov in which the future states only depend on the current state and action. This differs from the MCL method they described in the article, since it has better adaptability and is also more efficient, especially in the global localization problem with unknown initial positions. The MCL method is also faster, better and requires less computational force since it does not work on the entire environment, like mine does, only on the important ones which they call the high-probability areas. I think my implementation works well for more simple tasks where the state space and transitions are well defined as well as the environment being discrete while MCL can handle more complex ones and therefore the more versatile model.

Appendix:

Peer Review

From: el8726lj-s To: ka5532jo-s

Even though it might create an easy to navigate solution, it might have been a bad idea to replace the Localizer class for three separate Localizers for different settings (NUF,UF, etc), which in turn create and use three instances of filter classes (Filter_NUF, Filter_UF, etc). The biggest downside is that the execution time gets tripled, where only one third of the processing is actually used/presented, which leads to bad inefficiency. The positive side of this solution is however that it is much easier to navigate and edit code for the different settings. If I were to receive and try to understand the code, I would very much appreciate more code comments in some parts of your code. At a minimum, one above each function and class, but also for code within these that highlight and explain critical design choices. When we went through your code, we both realized that some functions/classes/etc often had more input parameters than they actually needed. It is not crucial for a correct implementation, but it might produce clearer code if you limit input parameters only to what they need.