Code walk-through:

In the initializer of pf.py, noise values are set for odometry rotation, translation, and drift. Rotation noise is applied to account for rotational uncertainty, translation noise is used in the robot's linear movements to measure the distance traveled along a direction, and drift noise calculates how much the robot deviates from its intended path (direction). These noise values are also used when setting the particle cloud, with each particle representing a potential location, incorporating error through a Gaussian distribution.

When updating the particle cloud, each particle is assigned a weight using the actual laser scan results compared to the particle's predicted laser scan data. The weight is proportional to how closely the predicted measurements align with the actual measurements. The weights are then normalized to create a cumulative probability distribution, preparing them for systematic resampling. In the case where the robot could be kidnapped, the particle cloud is replaced by newly generated particles with only the top 10% with the highest weight remaining. However, the challenge is how to decide if the robot has been kidnapped which would be discussed later.

To estimate the robot's final pose, I implemented K-means clustering on the particle cloud, which is a more robust approach than simply averaging all particle positions. K-means clustering helps identify the densest grouping of particles, which is most likely to represent the robot's actual location. This approach effectively discards outlier particles that may arise due to sensor noise or incorrect assumptions in the motion model. Outliers can distort the final pose estimation if included in a simple mean calculation, as their erroneous positions could pull the average away from the true location. By grouping particles into clusters, K-means allows the system to focus on the largest cluster, which represents the most probable region of the robot's location. This process thus minimizes the influence of particles that are far from the actual pose, enhancing the accuracy and stability of the robot's estimated position. Additionally, K-means clustering enables the system to handle multi-modal distributions, where multiple plausible locations might be represented in the particle cloud. By selecting the largest cluster as the final pose, the system can adapt to scenarios with more complex localization requirements.

Thought process on the kidnapped robot problem:

Attempt 1 by adding noise to particle cloud:

In this attempt, I tried to add noise when I update the particle cloud using resampling. More specifically, for every particle that's been re-selected, I added a Gaussian noise to their poses. However, after kidnapping the robot a few times, I realised this method is only effective when the robot is kidnapped for a short distance away relative to its last known position. It can be shown in rviz that the particle cloud does cover a wider range than not having noises for

selected new particles with each particle jittering around, and when I kidnapped the robot to a place that is within that range, most of the time the particle cloud will converge to that new position and the robot will relocate. However, when the robot is kidnapped out of the range where the particle cloud covers, it fails to do so. Then I thought about just adding a significant noise so that it would cover the whole map. However, went I ran rviz and tried to track the robot, the robot's location seemed to randomly jump around due to the particle cloud is picking up too much noise. Therefore, I realised just by adding noise to the particle cloud is not a good approach. Even so, I kept a small amount of noise for the robot to adjust when the distance of being kidnapped is not significant to reduce the computational power needed to insert new particles.

Attempt 2 keep adding random new particles all the time:

In this attempt, I tried to add different numbers of newly generated particles all the time at random locations in the defined map area no matter where the robot's location is. I thought this would be a good attempt form attempt 1 because the new particles would be spread out across the map instead of spreading around with the centre being the previous estimated robot's location. When I ran the visualisation first time, it did well in terms of tracking the robot's position. However, more and more particles were being built up as the robot moved further and further, which ended up with the particle cloud picking up too much noise. Therefore, I tried to reduce the number of new particles being added. But when the number of particles is too small, the algorithm seems to be just ignoring those newly added particles. Therefore, I realised I need to find a way to add enough particles but only when it's necessary so that it wouldn't be picked up as noise. I decided to set a condition for when the robot is being kidnapped then add new particles in.
As you could see in Figure 1, the robot's real position is in the room where there is a small cluster formed, due to the noise, a bigger cluster is formed outside the room and the estimated pose is determined in the larger cluster due to noise.

Figure 1

Attempt 3 (Final solution) adding particles only when the robot is determined to be kidnapped:

While this solution seems reasonable, the hard part is to determine when the robot is kidnapped. I first attempted to do this based on the average weight of the particle cloud. I did this by adding print statements in the node.py file and moving the robot around and sometimes kidnapping it as well. I tried to observe any pattern of the weights dropping when the robot is kidnapped. However, I realised both cases the average weight ranges from 2 to 10. This is because although when the robot is passing through the same areas on the map, the average weight would be higher when it is being tracked correctly than being lost. As shown in Figure 2 below, the top one which represents the robot is being kidnapped and the bottom one showing its true position, the weight has very minor differences due to being in different areas of the map. Thus, I decided there is no solution that would work for most parts of the map just by observing the weights.

To further determine the data pattern when the robot is kidnapped, I printed the standard deviation of the particle cloud together with the combination of weights. This time I noticed when the robot is being tracked properly, the standard deviation ranges between 2 and 4 and it drops below 2 when it's kidnapped. Therefore, I decided to use both the weight and the standard deviation to determine whether the robot is kidnapped. I started to use different threshold for the weight and standard deviation.

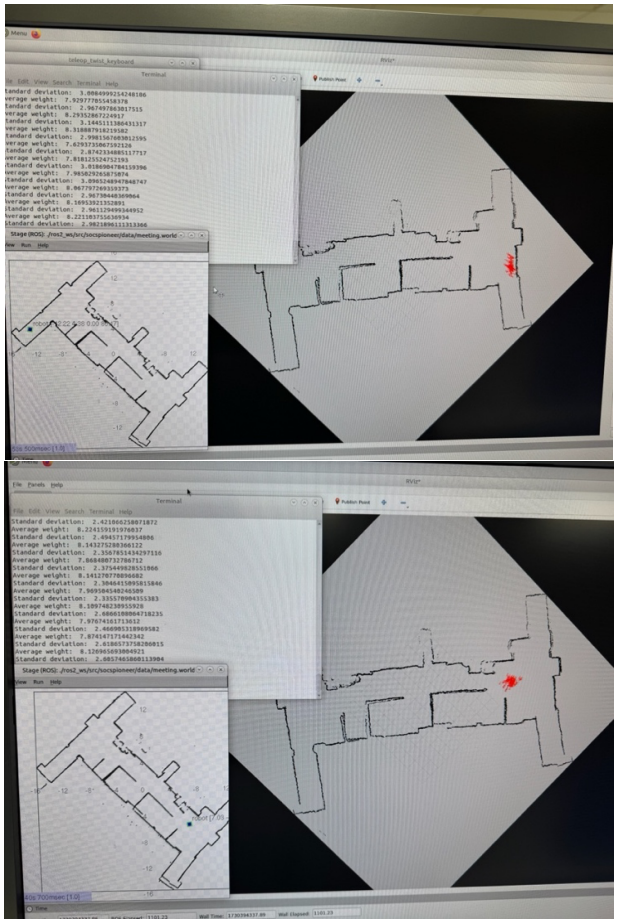

Figure 2

During the actual tuning of standard deviation and mean weight, I only used standard

deviation as a secondary reference because the weight is a direct indicator of the robot's belief while the standard deviation is used in some areas when the particle cloud's weight remains high even when the robot is kidnapped. Therefore, I only tried different value of the weights. I realised when I set the weight threshold too low, the particle cloud tend to cluster at the wrong area. However, when the weight threshold is too high, although it would deal with the kidnapped robot problem well, but it is not consistent when tracking the robot normally. I believe this is because when the robot moves and receives laser scan data, when it approaches some of the open spaces its confidence would drop, leading the robot mistakenly believe it is being kidnapped and try to find a new location. After some experiments, I set my weight threshold to be 3.15 as the mean weight of the particle cloud.
However, when I was trying to tune the weight threshold, I realised a lot of the times the particles cluster out of the actual room, which is inconvenient because when the particles do cluster together, the data pattern could look like the robot is not being kidnapped. As you could see in figure 3, the particle cloud clustered outside the map with the average weight and standard deviation all within the threshold. Therefore, I was trying to only generate particles in the room area. I then took the diagonal walls of the room and modelled them as straight parallel lines using y=mx+c, as the upper and lower bound of the room. However, this would imply that when the robot is kidnapped to the four corners sticking out in the room, it would not be able to re-localise it there. Instead, when I drive the robot back to the area where there are new particles forming it will re-localise pretty fast.

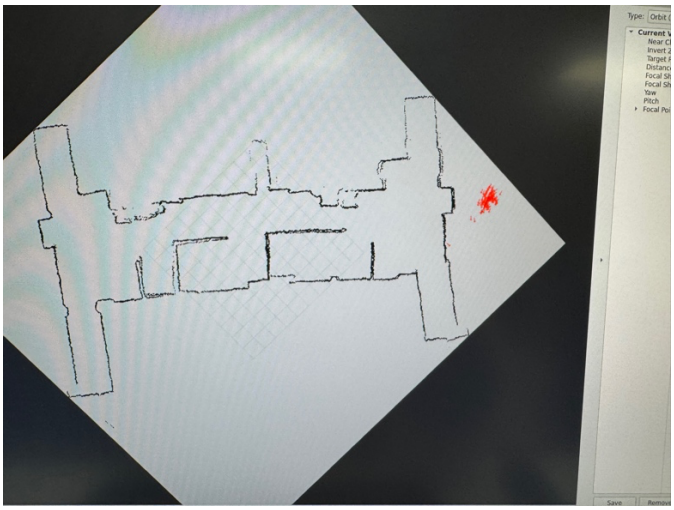

Figure 3

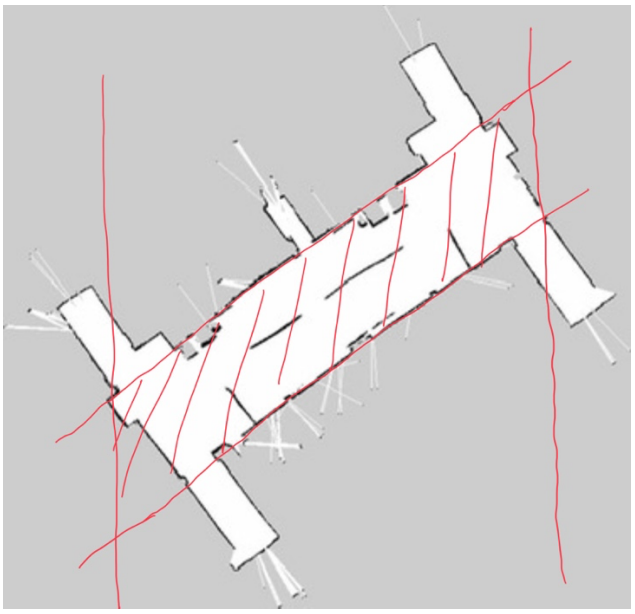

Figure 4

As shown in Figure 4, the shade area is where the random new particles are generated, however, this implies that if the robot is kidnapped to the areas sticking put in the corners, the robot can't recover quickly but this could be solved by moving the robot to the centre.

Now the final task left is to inject enough particles to the particle cloud as mentioned in the previous attempt. First off, I did the same as attempt 2 to just add random particles. However, I realised just by adding a small number of particles is not enough as the old particles who are not a good representation would have a more significant effect compared to the new particles while adding a lot of new particles could lead to noise being picked up in attempt 1. Therefore, I decided instead of just adding new particles, I replaced the majority particles in the cloud with newly generated particles but only keep those with the highest weights. I realised the more particles I replace, the faster the particle cloud converges to the true position of the robot. Therefore, I decided to only keep the top 7% of the original particle for reference when the robot is determined to be kidnapped. Replacing all particles introduces high randomness, which can cause the particle cloud to spread widely across the map. This can make re-localization slower, as the system needs to narrow down the search area again. Retaining reliable particles minimizes this noise and provides a more focused search for the true position.

The final implementation for addressing the "kidnapped robot" problem combines particle filtering with an adaptive particle replacement strategy. When the system detects signs of kidnapping—indicated by average particle weights less than 3.15 and a standard deviation less than 1.5—it replaces most particles with new, randomly generated ones across the map, while retaining a small percentage of high-weight particles. This method ensures that the particle cloud can quickly converge to the robot's true location without being overly disrupted by noise. By selectively preserving only the most reliable particles, this approach enables efficient re-localization after the robot is displaced, providing a responsive solution that maintains localization accuracy even in challenging scenarios.

Evaluation of performance:

With my final implementation, the robot won't mistakenly believe it has been kidnapped when I just move it around with my keyboard. However, it will redistribute the particle cloud around the main area of the room when it thinks it's been kidnapped then the particle cloud will cluster around the correct area most of the times. The exact performance depends on the area where the robot is kidnapped to. When the robot is kidnapped to one of the rooms, it will almost update its position instantly, very occasionally mistakenly ended up in the wrong room. However, when it is kidnapped to the relatively open spaces on the left- or right-hand side of the map, it can struggle to localise by itself. But when I move the robot around, especially by spinning the robot, it can almost re-localise itself instantly. Very occasionally, the robot believes itself in the wrong side of the map symmetrical to its true position. However, that was solved by moving the robot to the centre of the room because the robot will update its belief based on the odometry data received.