Monte Carlo Localization of Pioneer 3D-X robot

Group 14: Diogo Araújo, 93906, Gonçalo Mesquita, 94196, Paulo Cruz, 93150, Sebastião Melo, 93180

***Abstract*—The following report covers the problem of localization using a wheeled robot. To address it, the Monte-Carlo Localization algorithm, a.k.a. Particle Filter, was used, in conjunction with measurements taken from both an attached laser and the robot's own relative sensing.**

***Index Terms*—Introduction, Algorithm, Implementation, Results, Conclusion, Bibliography.**

I. Introduction

The objective of this project is to localize a robot within a known map, both in a micro-simulation and in real time, using the Monte-Carlo Localization algorithm.

II. Algorithm

A. Particle Filter

The localization problem consists of finding the position of a robot in a known environment. The robot is equipped with a laser that measures its distance to obstacles, and an odometry sensor that obtains a position by measuring its movement relative to the previous position. Both the distance and odometry sensors are noisy and introduce error. This will be kept in mind as we present the models used in the next section.

In the localization problem we are assuming that we have a known map of the surrounding environment and that, initially, there is equal probability of occupation in the whole map. The robot is placed in the environment and, by moving it around, it should be able to piece together its location in said environment's map based on what it has seen and on the estimate of how it moved through the environment. For this we implement the particle filter.

Particle filter methods are recursive Bayesian filters that have a variety of applications and essentially consist of three looped steps: prediction, update and re-sampling.

*1) **Predict**:* For this specific localization problem, it's initially assumed that there is equal probability of occupation and orientation by the robot throughout the map, so we can approximate the initial probability distribution by randomly creating a number of discrete poses in the map, i.e., uniformly randomizing the x and y coordinates as well as the orientation. Each of these poses is represented by a particle that is are stored in the filter as a possible estimate of the state.

While the robot is moving through the mapped environment with an approximate model for odometry, the robot movement is replicated by each particle. Basically we are predicting each of the particles movement if they moved in the same relative way as the robot. Particles will move in a slightly different way to one another due to the uncertainty introduced in the odometry model, representing its error.

*2) **Update**:* By creating an approximate model of the laser, it's possible to compare what each particle measures with its modeled laser, with the measurements of the real laser and determine how likely it is for each particle to represent the system state. We can then give different weights for each particle according to their likelihood of representing the real localization of the robot. After one iteration, the probability distribution is no longer uniform but is more concentrated around the "heaviest" particles. The laser will keep taking measurements in different places of the map and the particles more likely to be in the real position will have their weights increased. The weights are updated with equation 1.

$$w_t^{[m]} = P\left(Z_t | x_t^{[m]}\right) \tag{1}$$

Given that for each instance there are a lot of measurements, and considering those measurements independent we have,

$$\begin{aligned} P\left(Z_t | x_t^{[m]}\right) = P\left(z_t^{[1]} | x_t^{[m]}\right) \times P\left(z_t^{[2]} | x_t^{[m]}\right) \times ... \\ ... \times P\left(z_t^{[N]} | x_t^{[m]}\right) \end{aligned} \tag{2}$$

In which N is the number of measurements in each time instance.

*3) **Re-sample**:* We don't want to keep all the low weighted particles in the filter because it would waste computational resources on unlikely poses and could also cause the filter to never converge. Therefore, we are going to re-sample the particles according to this new probability distribution. This will place more particles in poses that are more likely and less particles everywhere else. With more iterations the filter should start to converge, which means that the high probability area should shrink and the next generation of particles should be even more concentrated, until we reach the target distribution.

III. Implementation

A. Models

*1) **Odometry mode**:* For the predict step it is necessary to have a motion model. In the predict phase of the algorithm the particles are going to be sampled in accordance with the motion model represented in equation 3

$$\begin{cases} x_{t+1} = x_t + \triangle D_t \cdot cos(\theta_t) \\ y_{t+1} = y_t + \triangle D_t \cdot sin(\theta_t) \\ \theta_{t+1} = \theta_t + \triangle \theta_t \end{cases} \tag{3}$$

In this motion model, (x_t, y_t) represents the current position of a given particle at time t. θ_t is the current orientation of a given particle at time t. $\triangle D_t$ and $\triangle \theta_t$ represent the actions done by the robot. $\triangle D_t$ is the distance travelled by the real robot at time t and $\triangle \theta_t$ is the rotation done by the robot at time t. The odometry model is completed when some Gaussian noise is added.

*2) **Measurement model**:* The measurement device used in this project was the Hokuyo URG-04LX-UG01. Consulting the datasheet, we know that it has a maximum range of 5.6 m, and that it captures an amplitude of 240º, starting at -120º and stopping at 120º.

For capturing the measurements for each particle, we produced a laser model using the parameters mentioned above to best represent the model. Each laser beam is represented by a line segment. The point (x_1, y_1) corresponds to the particle's current position, and the point (x_2, y_2) is computed using the formula represented in equation 4.

$$\begin{cases} x_2 = x_1 + z_{max} \cdot cos(\theta + \delta_k) \\ y_2 = y1 + z_{max} \cdot sin(\theta + \delta_k) \end{cases} \quad (4)$$

Here, θ represents the current orientation of the robot, δ_k is the increment of each laser beam and z_{max} is the maximum range of the laser. To be able to create several line segments, we increase δ_k, as seen in figure 1. The way the laser beam is able to measure a distance is by computing the mathematical interception with a wall, which is also a line segment. The simulated laser beam can intersect more than one wall at a time. However, there can only be one hit per laser beam. If the laser beam intersects more than one wall, then the right hit will be the one closest to the particle's current position. When there is a hit, a Gaussian noise is added, with mean of 0 and a standard deviation of $0.05m$ for the micro-simulation, and $0.1m$ when it comes to real data. For each laser beam it is necessary to check if it intersects with all the walls of the map. This slows down the algorithm significantly, meaning that the greater the number of measurements the slower the algorithm will be.

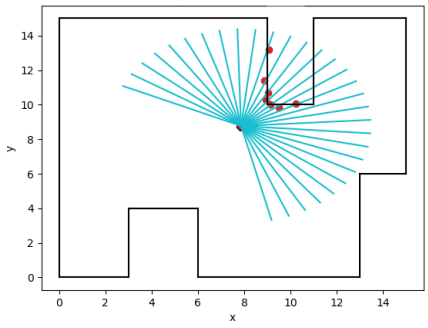

Figure 1: Laser model illustration

B. Particle Filter

*1) **Initialize the Particles and Weights**:* First we start by evenly distributing the particles all over the map, and initializing their weights as 1.

*2) **Algorithm only runs**:* If the robot moves, the algorithm runs. If the robot does not move, the algorithm does nothing.

*3) **Predict**:* Each particle is going to be sampled according to the motion model described in section III-A1.

*4) **Update**:* Given equations 1 and 2, the likelihoods are given by equation 5,

$$P\left(z_t^{[i]} | x_t^{[m]}\right) = w_1 \cdot \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(z_i - d_i)^2}{2\sigma^2}} + w_2 \cdot \frac{1}{z_{max}} \quad (5)$$
Subject to: $w_1 + w_2 = 1$

where z_{max} is the maximum laser range $(5.6m)$ and z_i and d_i represent real and estimated measurements, respectively. We applied this approach so that none of the particles' weights would be killed. For example, let's imagine a particle that is located near the robot, but is oriented in a way that makes the error of a certain measurement of this particle very high, making its likelihood very low (particle does not pass in the Gaussian like filter). This could bring the weight of that particle very close to zero, with the possibility of it being 'killed'. Since this particle was close to the robot's position, we want to prevent this from happening. With this in mind, our first approach was to increase the standard deviation. Consequently, this particle would not be 'killed' anymore. However, this created a big problem, because the filter would allow too much useless data to pass through, not re-sampling the particles that were more likely to be close to the robot's location.

Thus a different approach was attempted. As shown in equation 5, w_1 represents how much weight we should give to Gaussian distribution (our filter). The portion $w_2 \cdot \frac{1}{z_{max}}$ prevents the particles weights from going to zero, making sure that no particle 'dies prematurely'.

*5) **Kidnapping**:* Kidnapping happens when the robot is "picked up" and put in another position, not following the motion model. If the mean of the non-normalized likelihoods is lower than a given threshold twice in a row, it's very likely that kidnapping has occurred. Thus, the number of particles is increased ($M_{new} = M \cdot 1.2$), and then 80% of the particles are uniformly distributed throughout the map. This was the procedure chosen to avoid this type of problem, because there are situations in which the average of the non-normalized weights may drop abruptly, but there was no kidnapping (e.g., a person passing in front of the robot). Therefore kidnapping is only assumed when the average of the non-normalized weights falls below a given threshold consecutively. Only 80% of the particles are distributed for the same reason.

This distribution of particles is going to make our algorithm robust to kidnapping as well as allowing it to comeback from a wrong prediction. This will prove to be very advantageous

when the algorithm has a smaller number of particles.

6) ***Re-sampling****:* Here we used selective re-sampling, which means that if $n_{eff} < \frac{M}{2}$ (see right portion of equation 7) we use low variance re-sampling. Otherwise, we don't do re-sampling. If re-sampling is not done, the non-normalized weights are going to be updated as described by equation 6.

$$w_t^{[m]} = p(Z_t|x_t^{[m]})w_{t-1}^{[m]}Zt \tag{6}$$

This helps the performance (time wise) of our algorithm. The low-variance re-sampling algorithm re-samples periodically according to the left portion of equation 7, where r is the first sample drawn from a random location, M is the total number of samples and m is an integer number between 1 and M corresponding to each sample.

$$u = r + \frac{m-1}{M} \qquad n_{eff} = \frac{1}{\sum_{m=1}^{M}(w_t^{[m]})^2} \tag{7}$$

n_{eff} is the effective number of particles and w_t is the weight of each particle. If $n_{eff} > 0.85 \cdot M$ we reduce the number of particles to $M_{new} = M \cdot 0.9$, thus the algorithm performance, in terms of time, will increase over time. The number of particles is always above a threshold.

C. Synthetic data [Micro-simulator]

With everything mentioned before, and a few more features, we created a micro simulator. First, we start by designing a map with line segments. For example, two points were chosen, (0,0) and (3,0) and a line was drawn with those two as its vertices. This process is repeated until a map is created, such as the map shown in Figure 1. This was also the method used to produce maps for real time. In order to create artificial data we use the odometry and laser model explained previously. At the start of the simulation the robot is 'turned on' at a given point in space. At each iteration, the robot moves accordingly the odometry model. With this motion we can compute $\triangle D$ and $\triangle \theta$. The measurements of the robot are simulated using the laser model, thus creating the synthetic data.

D. ROS and Real data

In this section we explain how we used ROS to obtain real data, and to observe and compare our algorithm with a ground of truth.

1) ***Data****:* ROS is a very important tool which gives us the ability to record data obtained by the robot so that we can process it. In the localization problem we need to receive two essential types of data: position obtained by the odometry sensor and the distances measured by the laser. ROS makes this data available through topics. The essential topics we are subscribing to are the following: /pose topic, which gives us the position and orientation coordinates of the robot in each instant, relative to where it was first turned on, in meters and quaternions, respectively. And /scan topic, which gives us an array of distances measured by the Hokuyo laser in meters.

In order to use this data in our algorithm we created a subscriber script. This script subscribes to both of these topics and receives data at a certain rate defined by us. From the pose data we always keep two arrays. These two arrays are a new position array and a previous position array. They are essential in order to obtain $\triangle D$ and $\triangle \theta$. For the laser scan data we keep an array of only some positions of the received array because it would take too many computational resources to compare over 700 distances. Therefore, we reduced this array to 24 measurements because after testing with more measurements we concluded that this number of distances would constitute a perfect balance between speed and results.

2) ***Ground of truth****:* Before opting to create our own map with line segments we obtained a map with gmapping that is represented in figure 2.

Although this map wasn't used directly for our localization problem solution, it was an essential tool in visually validating our results. In order to obtain this visual validation of results we used a ROS package called amcl. This package contains its own algorithm of Adaptive Monte-Carlo Localization, which we used as a ground of truth. We created two launch files: tf_launch and amcl_launch. The first one is needed to publish the static transform between the laser and the robot (base_link-laser). The AMCL launch file, with our map, the transform base_link-laser, and the data received from the pose and scan topics, would create the transforms map-odom and odom-base_link, displaying the robot moving around the map, in Rviz, as we controlled it, with an estimated pose calculated by this ROS amcl algorithm. This launch file would also publish the estimated pose obtained by this algorithm to the topic /amcl_pose, which we also subscribed to in our subscriber script in order to have a ground of truth for our algorithm results.

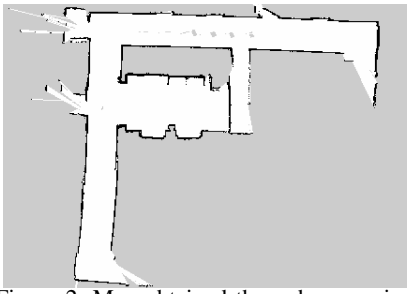

Figure 2: Map obtained through gmapping

IV. Results

To prove the effectiveness, consistency, and robustness of the algorithm, several tests using synthetic data and real data were performed.

A. Synthetic data

Keeping certain parameters constant throughout the tests performed with synthetic data, the following results were obtained.

*1) **First trajectory**:* Given the trajectory illustrated in figure 3, ten tests were performed for each of the following situations: number of particles equal to 300, 600, 900 (thirty tests in total). With this, we got the statistics illustrated in plot 4.

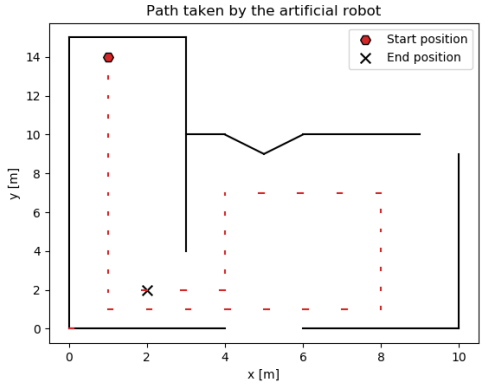

Figure 3: Trajectory of the robot using synthetic data

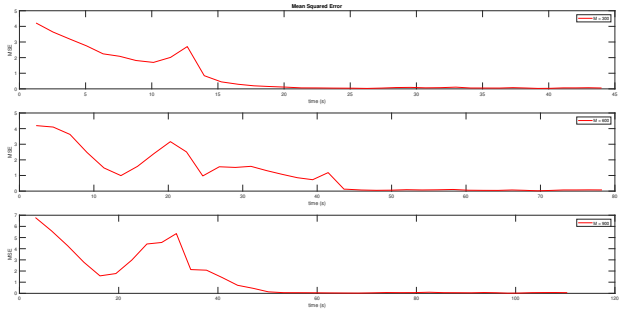

Figure 4: RMSE between robot position and algorithm prediction

From figure 4, it can be concluded that the algorithm is extremely effective and consistent, converging in all three situations, with an error in the order of the centimeters.

Looking more closely at the results for 300 and 600 particles, we can see the robustness of the algorithm.

As for the case of 300 particles, five times out of ten, the algorithm initially converged to the wrong position. However, eventually the average of the non-normalized likelihoods would reach (at least two consecutive times) a value below a given threshold, causing the number of particles to increase, and 80% of them to disperse evenly across the map (see section III-B5), giving the algorithm the possibility to recover and converge to the correct position.

It's also important to note that the smaller the number of particles, the faster the algorithm will converge, as can be seen by the results obtained. This makes sense, because the smaller the number of particles, the smaller the computational load on the algorithm (as will be verified later). Even considering that with fewer particles there is a higher probability of increasing and dispersing the particles, it still converges faster than if there were many particles. However, it will be shown that this no longer happens when using real data in real time.

Thus, the robustness to kidnapping also makes the algorithm more accurate for the situation of having a reduced number of particles.

*2) **Kidnapping**:*

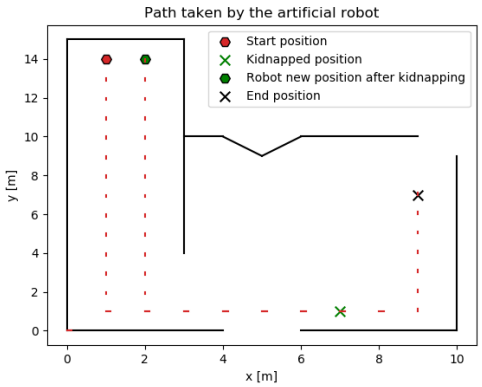

Figure 5: Trajectory of the robot using synthetic data

Given the trajectory illustrated in figure 5, ten tests were performed for the following situation: number of particles equal to 900. With this, we got the statistics illustrated in plot 6.

Analyzing the figure 6, we can see that the robot was kidnapped at $t = 64s$. This causes the error between the 'real location' and the prediction of the algorithm to rise abruptly from *0.02m* to approximately *14 m*. After the robot has been kidnapped the average of the non-normalized weights will drop significantly, causing the number of particles to increase, distributing 80% of them evenly across the map, as explained in section III-B5. This way, the algorithm quickly recovers and converges onto the correct position, making the error between the real and the estimated position in the order of the centimeters once again.

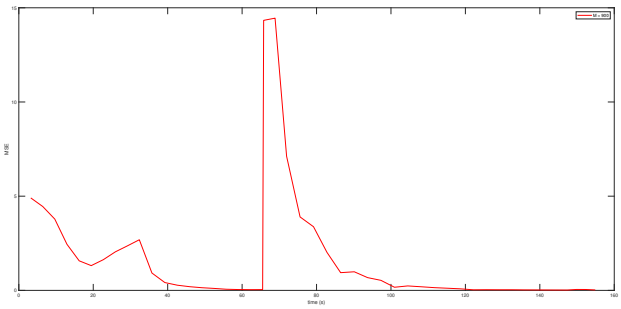

Figure 6: RMSE between robot position and algorithm prediction

B. Real data

From the micro-simulation results, and by doing several tests with real data in offline mode, i.e., the frequency at which the algorithm receives data was reduced (approximately to 0.15Hz), it was validated that the algorithm not only works, but it also performs extremely well, be it in terms of

accuracy, consistency, or robustness. Therefore, it was decided to perform statistics for real data in real time, in order to understand what are the limitations of the algorithm, and if it can still perform well under these circumstances.

In this section the following tests were performed: for two different trajectories, ten tests were performed for each of the situations: number of particles equal to 300, 600, 900 (sixty tests in total, thirty for each trajectory). There were certain changes in each case in order to prove that the modifications made to the algorithm were advantageous, making it much more consistent and robust.

The tests were performed keeping the following parameters unchanged throughout: standard deviation of the likelihoods equal to 1.5 m; uncertainty of the odometry model equal to 0.1 m (for both x and y coordinates); uncertainty for the laser model equal to 0.1 m; number of measures per instance equal to 24;

1) ***First trajectory****:* Given the trajectory illustrated in figure 7, the results obtained are illustrated in plots 8 and 9.

Comparing the results of 8 with 4, we quickly conclude that the algorithm has a dramatic drop in performance levels when tested with real data in approximately real time.

Particularly for the case of 300 particles, there is a sharp drop in accuracy (approximately two orders of magnitude) which can be easily explained. As stated in section IV-A1, with 300 particles, in ten times the algorithm converges five. What happens is that in the micro-simulation there was the time needed for the particles to spread out and for the algorithm to converge to the real location. However, in real-time this no longer happens. Even if the particles spread out over the map, since in the algorithm's perspective the robot makes very large jumps, it does not have enough time to converge again, because in a few iterations of the algorithm the robot's course will end.

However, we can see that as the number of particles increases the algorithm performs much better in terms of accuracy. In the case of 900 particles, we can see that the error in the last iteration of the algorithm is about 0.3 m, which is about an order of magnitude more than the values obtained with synthetic data. Considering that these tests were performed in real time, this is a very good result.

The divergence between the results from 300 to 900 particles comes from the simple fact that with 900 particles there is a greater chance of the algorithm being correct in one of the initial iterations, causing it to converge in an early stage of its process, obtaining a good accuracy.

Figure 9 shows the time per iteration of the algorithm for each case studied. As expected, the larger the number of particles, the longer the time per iteration. However, in this case, this does not prove to be very relevant, because even though the average time per iteration in the case of 300 particles is lower compared to the case of 900 particles, it is still much higher than the amount of data received per second from the rosbag.

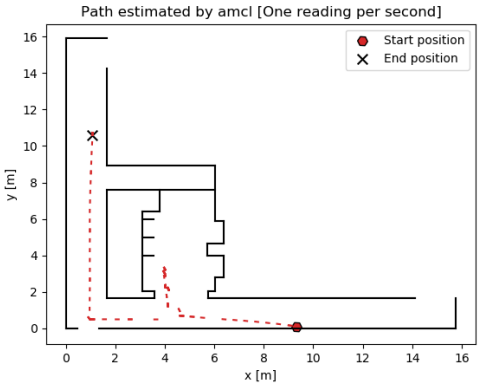

Figure 7: Amcl estimated trajectory

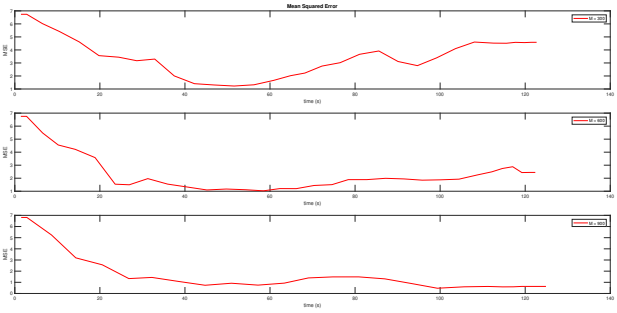

Figure 8: RMSE between amcl reference and algortihm prediction

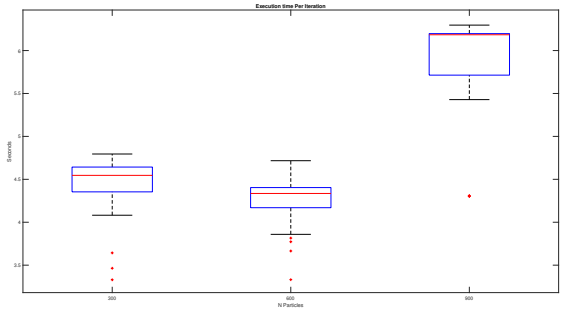

Figure 9: Time per iteration for the respective number of particles

2) ***Second trajectory****:*

Given the trajectory illustrated in image 10, the results obtained for the trajectory are represented in Figures 11 and 12. When comparing the results of Figure 11 with those of Figure 4, from the micro-simulation, we realize that we obtain the same conclusions as in the previous IV-B1. It is proven that in real data the algorithm has a worse performance than in the micro-simulation, which was already expected. Comparing the results of the first trajectory with the second, we can observe that there is a disparity. This disparity is due to the fact that we removed the portion of the code explained in section III-B5 in the second trajectory. That is, with 300 particles, the algorithm would no longer spread the particles if the average likelihoods were below a certain threshold. Consequently, for

a low number of particles the algorithm converges to the wrong position with no way of 'making a comeback', making the results worse compared to the previous section. Thus the implementation of the algorithm described in section III-B5 was a game changer.

Figure 12 shows the execution times of each iteration for the three studied cases. We can observe that our algorithm becomes faster and faster over time, due to the fact that we remove particles when $n_{eff} > 0.85 \cdot M$. However we have to be very careful when reducing the number of particles, especially when we have fewer particles, because we risk getting a small number of particles very quickly.

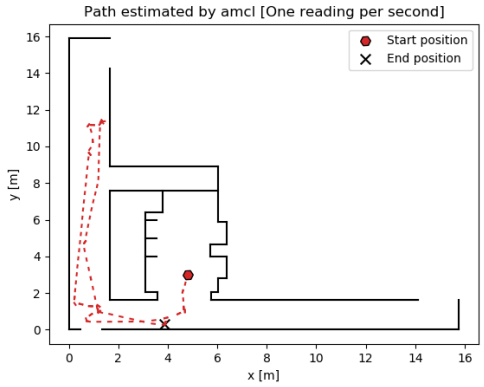

Figure 10: Amcl estimated trajectory

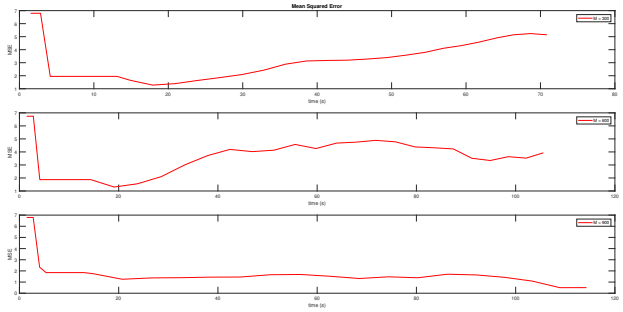

Figure 11: RMSE between amcl reference and algortihm prediction

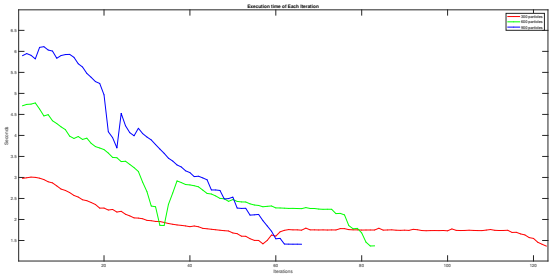

Figure 12: Time per algorithm iteration

V. Conclusions

When it comes to the micro-simulation and the algorithm offline with real data, the results were excellent, having the algorithm converge with very good accuracy, being consistent regardless of the type of map, the type of trajectory performed by the robot, or the number of particles, and being robust to kidnapping, as demonstrated in the previous sections. When it comes to real data, even though the results were also good, the error was more significant. This happens because of limitations we have had with the duration of each iteration, which is proportional to the number of particles. That is, the larger the number of particles, the slower the algorithm will be, so we cannot have the luxury of having too many particles. On the other hand, as has been shown from results with real data in real time, the number of particles cannot be too low either. So there is a trade-off between time per iteration and uniform particle filling of the map in question. One of the main aspects that makes our algorithm so slow was the laser model. In this case, more measurements imply more mathematical interceptions, therefore increasing the duration of the algorithm. There has to be a trade off between precision of the measurements and the time per iteration. Ways to resolve this problem would be greater computing power, different type of programming and more time to implement something more sophisticated.

However we still managed to use some techniques to reduce the duration of the algorithm quite significantly, such as reducing the number of particles if $n_{eff} > 0.85 \cdot M$. However, this has proven to be a risky move, and it is necessary to be very careful when this is applied in practice. Ultimately, robustness to kidnapping presented itself as the game changer in our algorithm. Not only by solving this problem, but also by making our algorithm much more accurate when using a reduced number of particles. Nevertheless we proved that the algorithm converges with artificial data and with real data in offline mode with excellent accuracy. In real time and using a number of particles around 900, we still obtain very good results, all things considered. In conclusion, the group has managed to develop a very consistent, robust algorithm that gives good results for the time given to do finish this project.

References

[1] Tutoriais em ROS, https://wiki.ros.org/ROS/Tutorials, 2022
[2] Tutorias em TF2, https://wiki.ros.org/tf2/Tutorials, 2022
[3] A. Vale, R. Ventura, A. Bernardino, M.Ekal, "Laboratory Guide"
[4] M.I.Ribeiro, P. Lima, R.Ventura, "Localization Fundamentals"
[5] M.I.Ribeiro, P. Lima, R.Ventura, "PROBABILISTIC LOCALIZATION - Monte Carlo Localization"
[6] S. Thrun, W. Burgard e D. Fox, "Probabilistic Robotics", 2005 MIT Press