FastSLAM on Turtlebot3 - SFT

Alexandre Reis 100123, Duarte Morais 100163, João Dias 100202 and Rodolfo Amorim 100260 - Group 25

***Abstract*—This paper presents the implementation of the FastSLAM algorithm on a Turtlebot3 robot using ArUco markers as landmarks. We created a micro-simulation environment for algorithm testing, developed an ArUco detection algorithm, and fine-tuned key parameters to optimize performance. As an additional contribution, an Occupancy Grid Mapping algorithm was integrated, as well as RViz visualization. To evaluate the algorithm, metrics were computed with regards to both the trajectory and the map for different environments and used to fine-tune and validate the algorithm. The obtained results were good and consistent over several runs of the same and similar ROSBags.**

I. Introduction And Motivation

SLAM or Simultaneous Localization and Mapping, is a problem where a mobile robot, put at an unknown location and environment, incrementally builds a map of the environment it is in, while estimating its position within this map, [1]. FastSLAM is an efficient SLAM algorithm, which decomposes this problem into a robot localization problem and a collection of landmark estimation problems, [2].

As such, the major goal of the project is to make an implementation of the FastSLAM algorithm with known correspondence on the *Turtlebot3* robot, using *ArUco* markers as landmarks (uniquely identifiable features). As a bonus goal, the group sought the development of an Occupancy Grid Mapping algorithm that uses FastSLAM's robot pose estimate.

II. Methods and Algorithms

A. FastSLAM Algorithm

FastSLAM with known data association is a particle filter based algorithm designed to solve the SLAM problem. In this approach, each particle represents a possible robot pose and an associated map, which is maintained by a set of Extended Kalman Filters (EKFs) for each landmarks. The algorithm's complexity is $O(MlogN)$, where M is the number of particles and N is the number of landmarks.

1) Particle Filter: The core of FastSLAM is the particle filter, which consists of multiple particles that help keep track of multiple hypotheses of the robot's pose and the environment's map, with associated uncertainty's.

2) Motion Model: The motion model predicts the robot's new position based on its previous position and the control inputs. Each particle updates its pose using the probabilistic motion model, $p(x_t|x_{t-1}, u_t)$, which captures the uncertainty in predicting x_t given x_{t-1} and u_t.

3) EKF: Within each particle, EKFs update the map. Each landmark has its own EKF, which updates its position and uncertainty based on new measurements. The measurement function h given the landmark position m_{ct} and the robot pose x_t^k is linearized using a Taylor expansion:

$$h(m_{ct}, x_t^k) \approx \hat{z}_t^k + H_t^k(m_{ct} - \mu_{ct,t-1}^k) \tag{1}$$

Here, $\hat{z}_t^k$ is the predicted measurement, $\mu_{ct,t-1}^k$ is the current estimate of the landmark position and H_t^k is the Jacobian of h.

Using this linear approximation, the EKF updates the landmark's position and uncertainty. The Kalman gain K_t^k is calculated to determine the influence of the new measurement:

$$K_t^k = \Sigma_{ct,t-1}^k (H_t^k)^T \left(H_t^k \Sigma_{ct,t-1}^k (H_t^k)^T + Q_t\right)^{-1} \tag{2}$$

Here, $\Sigma_{ct,t-1}^k$ is the covariance of the landmark estimate from the previous step, and Q_t is the measurement noise covariance.

The new mean $\mu_{ct,t}^k$ is updated with the new measurement z_t:

$$\mu_{ct,t}^k = \mu_{ct,t-1}^k + K_t^k(z_t - \hat{z}_t^k) \tag{3}$$

where z_t is the actual measurement observed by the robot, and $\hat{z}_t^k$ is the predicted measurement.

Finally, the covariance $\Sigma_{ct,t}^k$ of the landmark position is updated to reflect the new estimate:

$$\Sigma_{ct,t}^k = (I - K_t^k H_t^k)\Sigma_{ct,t-1}^k \tag{4}$$

where I is the identity matrix.

4) Resampling: Resampling is performed to focus computational resources on the most likely hypotheses. Particles with higher importance factors are more likely to be selected, while particles with lower weights may be discarded. While performing a resample, the variance between particles in the particle set decrease, but the "variance of the particle set as an estimator of the true belief increases over time" [3]. This means that with resampling, the particles will tend to better represent the possible states where the system can be.

In order to improve results, resampling should only be done when the particle's importance factor is too high on a small number of particles. In this case, the particle set does not represent well the state of the system.

B. Occupancy Grid Mapping - Bonus

Occupancy grid mapping is a Mapping algorithm whose output is a grid with cells assigned with probability values. This algorithm was implemented in this project to help visualize the FastSLAM's results.

The general Occupancy grid mapping problem involves the known pose of the robot $x_{1:t}$ and the real time sensor data $z_{1:t}$, in order to estimate the probability of occupancy for each cell: $p(m_i|z_{1:t}, x_{1:t})$.

This probability is calculated recursively in time with a Binary Bayes Filter. [3]

In the case of this algorithm, the probability of each cell is computed and updated with log odds $l_{t,i}$. But the outputted array should be converted back to actual probabilities.

To update each cell's probability, an *Inverse Sensor Model* is used. This converts the reading of a sensor into probability of each cell being occupied at that time.

$$l_{t,i} = l_{t-1,i} + \text{InvserseSensorModel}(m_i, x_{tzt}) - l_0 \quad (5)$$

In the frequent case of the sensor being a Lidar with range measurements, the *Inverse Sensor Model* uses Bresenham's Line Algorithm [4] to determine which grid cells are traversed by a sensor ray.

C. Kabsch Algorithm

The Kabsch algorithm is a method to align two different set of points in space. To accomplish this, a rotation matrix is computed by minimizing the root mean squared error between two trajectories. The algorithm has the following steps:

1) **Centering the trajectories:** computing and subtracting the centroid of each group of points.
2) **Compute the covariance matrix:** denoted H.
3) **Do singular value decomposition on the covariance matrix:** $H = USV^T$.
4) **Compute the optimal rotation matrix:** $R = UV^T$.

Once R has been computed, it is possible to rotate the reference frame of one of the trajectories to align with the other and translate it to the same origin, therefore having both group of points in the same reference frame.

III. Implementation

A. Aruco Detection Implementation

The group's implementation of the *ArUco* marker's detection was done using the *OpenCV Python* library.

In the beggining, an image is obtained from the camera feed, which is then transformed into gray scale. Having the image and the camera properties, it is possible to do the detection of an *ArUco* marker and make its identification, i.e., to extract its ID. With this detection, a translation vector between the camera and the center of the *ArUco* marker is computed, in the camera's coordinate frame. Then two transformations are applied to this translation vector: First a coordinate frame transformation, in order to express this vector in the coordinate frame of the *Turtlebot3* and second a transformation from Cartesian coordinates to Polar coordinates, (using the x and z values of the translation vector). This last transformation lets us obtain the range, according to the equation (6), and the bearing, in respect to (7).

$$r = \sqrt{x^2 + z^2} \quad (6)$$

$$\theta = atan2(\frac{x}{z}) \quad (7)$$

An example of the detection of an *ArUco* marker is presented in Fig. 1, where the range is given in meters and the bearing is given in degrees.

Fig. 1. *ArUco* marker detection from camera feed

B. Micro-simulator

With the objective of improving the time of implementation of the FastSLAM algorithm, the group created a micro-simulation environment that wouldn't rely on real data. Instead, synthetic odometry, range, and bearing values, that closely resembled real values, were generated for testing purposes.

Figure 2 shows the micro-simulation setup. On the left side of the window is the simulated environment where a user can control the robot's movement. On the right side is the FastSLAM output. It should be noted that the landmarks currently visible to the robot are highlighted with a larger circle, on the left side.

In the FastSLAM window, the displayed robot pose and map correspond to the particle with the highest importance factor. Other particles' pose are represented with red dots.

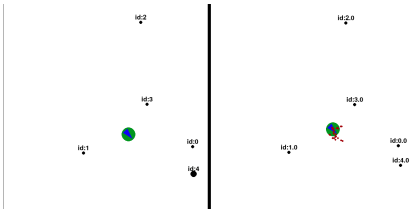

Fig. 2. Micro-simulation window

C. FastSLAM Algorithm Implementation

The FastSLAM implementation complies with these steps:

1) Particle Initialization: Particles are initialized with a null initial pose for the robot and an empty set of landmarks. The initial pose of the robot will correspond to that particle's reference frame's origin. Each particle maintains its own map of landmarks, which is updated as the robot observes new landmarks. Additionally, the initialization function sets up the particle's importance factor to 1.0 and the motion model parameters which are the standard deviations for motion noise.

2) Motion Model: The motion model updates each particle's pose based on odometry measurements. The motion update function computes the changes in position and orientation using the provided odometry data, adds Gaussian noise to these changes to simulate uncertainty in motion, and adjusts the particle's pose based on the noisy motion estimates.

The motion model is represented by the following equations, [5]:

$$\delta_{\text{rot1}} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta} \tag{8}$$

$$\delta_{\text{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^2} \tag{9}$$

$$\delta_{\text{rot2}} = \bar{\theta}' - \bar{\theta} - \delta_{\text{rot1}} \tag{10}$$

$$\hat{\delta}_{\text{rot1}} = \delta_{\text{rot1}} - \text{sample}(\alpha_1 \delta^2_{\text{rot1}} + \alpha_2 \delta^2_{\text{trans}}) \tag{11}$$

$$\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \text{sample}(\alpha_3 \delta^2_{\text{trans}} + \alpha_4 \delta^2_{\text{rot1}} + \alpha_4 \delta^2_{\text{rot2}}) \tag{12}$$

$$\hat{\delta}_{\text{rot2}} = \delta_{\text{rot2}} - \text{sample}(\alpha_1 \delta^2_{\text{rot2}} + \alpha_2 \delta^2_{\text{trans}}) \tag{13}$$

$$x' = x + \hat{\delta}_{\text{trans}} \cos(\theta + \hat{\delta}_{\text{rot1}}) \tag{14}$$

$$y' = y + \hat{\delta}_{\text{trans}} \sin(\theta + \hat{\delta}_{\text{rot1}}) \tag{15}$$

$$\theta' = \theta + \hat{\delta}_{\text{rot1}} + \hat{\delta}_{\text{rot2}} \tag{16}$$

$$x_t = (x', y', \theta')^T \tag{17}$$

In these equations, δ_{rot1} and δ_{rot2} represent rotations before after the translation, while δ_{trans} represents said translation. The turns and translation are noisy, therefore the values are subtracted by independent noise with zero mean and variance related to the the values of δ and α, the latter being robot-specific error parameters. The variables x', y', and θ' denote the updated position and orientation of the particle. Here, (x, y, θ) is the current pose of the particle.

3) Landmark Handling: Each particle updates its set of landmarks based on observations from the robot's sensors. The landmark update process involves detecting landmarks within the sensor's range and field of view, and then either updating an existing landmark using the Extended Kalman Filter (EKF) or creating a new landmark if it has not been observed before. For each landmark, the measurement model computes the expected range and bearing. The predicted measurement $\mathbf{z}_{pred}$ is given by:

$$\widehat{x}_{t|t-1} = \begin{bmatrix} \sqrt{(x_l - x)^2 + (y_l - y)^2} \\ \arctan(y_l - y, x_l - x) - \theta \end{bmatrix} = \begin{bmatrix} d \\ \phi \end{bmatrix} \tag{18}$$

where (x_l, y_l) is the estimated position of the landmark, and (x, y, θ) is the particle's pose. Here, d represents the Euclidean distance between the landmark and the particle, and ϕ represents the bearing angle to the landmark relative to the particle's orientation.

To update a landmark, we first predict the measurement using the current estimate of the landmark's position. The prediction includes both the distance and angle to the landmark. The Jacobian matrix J of the measurement function is computed as follows:

$$J = \begin{bmatrix} \frac{\partial d}{\partial x} & \frac{\partial d}{\partial y} & \frac{\partial d}{\partial \theta} \\ \frac{\partial \phi}{\partial x} & \frac{\partial \phi}{\partial y} & \frac{\partial \phi}{\partial \theta} \end{bmatrix} = \begin{bmatrix} \frac{\Delta x}{\sqrt{q}} & \frac{\Delta y}{\sqrt{q}} & 0 \\ -\frac{\Delta y}{q} & \frac{\Delta x}{q} & -1 \end{bmatrix} \tag{19}$$

where Δx and Δy are the differences between the robot's pose and the landmark's position, and $q = \Delta x^2 + \Delta y^2$.

The measurement noise covariance matrix Q is defined based on the sensor's noise characteristics. The measurement prediction covariance S is then calculated using the formula:

$$S = J \cdot \Sigma \cdot J^T + Q \tag{20}$$

where Σ is the covariance matrix of the landmark's position estimate.

The particle's weight is updated using the measurement likelihood, which is given by:

$$w = \frac{1}{\sqrt{2\pi\,|S|}} \exp\left(-\frac{1}{2}\mathbf{z}^T S^{-1} \mathbf{z}\right) \tag{21}$$

where $\mathbf{z}$ is the innovation (difference between the observed and predicted measurements), and $|S|$ is the determinant of the matrix S.

By continuously updating the weights and resampling the particles, the implementation maintains a set of particles that best represent the robot's pose and the map of the environment.

4) Resampling: In the case of the implemented algorithm, resampling is only done when $n_{eff} < \frac{M}{2}$. Where the number of effective particles is given by:

$$n_{eff} = \frac{1}{\sum_{m=1}^{M} (w_t^{(m)}))^2} \tag{22}$$

This threshold ensures that resampling is only done when the importance factor of certain particles is much higher than the others.

The resampling method used in this implementation is **low variance resampling**. This is a probabilistic method, where the probability of a particle being in the resampled set is proportional to it's importance factor, but ensuring that small importance factor particles have a chance of being sampled.

Note: The amount of particles is constant before and after resampling. If a low importance factor particle is not sampled, it means that there is at least one other particle that is sampled twice.

Fig. 3. FastSLAM implementation

D. Occupancy Grid Mapping - Bonus

In order to further develop the visualization of what the FastSLAM Algorithm is estimating, an Occupancy Grid Mapping algorithm was implemented.

The objective of this implementation was to enable a view of an occupancy grid map of the room, with a superimposed state of the SLAM algorithm's best particle.

So as to simplify this mapping problem, the FastSLAM's pose estimate was used as the true robot pose. Range measurements taken by the Turtlebot3's Lidar were also used.

An Occupancy Grid Mapping algorithm based on the Bresenham's Line Algorithm was used (section II-B). This algorithm is computed for every FastSLAM particle and uses the estimated pose of each particle at each time step. The published/shown Occupancy Grid Map is the one calculated by the Particle with the most importance factor, according to the FastSLAM algorithm. The resolution of the map's grid was chosen to be $0.1m/cell$ so that it wasn't too computational intensive.

The first results obtained from this implementation were very poor. The main reason for this is that the mapping algorithm assumes each particle's pose to be the true robot pose. In reality, the particle's pose deviates from the Ground Truth because of the particle filter's motion model (which adds noise to the position of each particle). The consequence of this is that, over time, Lidar measurements are used, which are not coherent with past measurements.

If there was a particle resampling based on the Occupancy Grid Map, this issue would not be noticeable in the long term, but in this implementation, it is.

To solve the aforementioned issue, two methods were implemented:

- Time based decrease in cell occupation probability, for occupied cells. This helps the algorithm give more importance to newer measurements than older ones.
- The algorithm will lose trust in what it knows about cells that are behind the current measurement of an obstacle. This uses a Bresenham's Line algorithm to estimate cells that are behind the occupied measured cell.
  If enough measurements (300) were made, where a specific cell is behind a measured obstacle, the algorithm changes that cell occupancy to unknown.

The two methods showed improvements mainly when the following scenario occurs: There is a rotation that was very much accentuated by the particle's motion model, in relation to the robot's true pose.

The downside of these two methods is that the Mapping algorithm becomes worse for big and complex maps, however these were not analysed in this project.

The end result is a Mapping algorithm that shows good results for short runs, but lacks robustness for longer time runs. Even though this is the case, the outcome of the map tends to always resemble the room in which the robot is in.

E. Fastslam node & RViz

In this project, *RViz* was used to display the occupancy grid map and the detected *ArUco* markers.

The primary topics used in our FastSLAM implementation and visualization in RViz include odometry information provided by the *'/odom'* topic, which includes the robot's position and orientation; laser scan data from the Turtlebot3's LIDAR sensor published on the '/scan' topic, which is crucial for the occupancy grid mapping; images captured from the Turtlebot3's camera on the */raspicam node/image/compressed* topic, used for ArUco marker detection; detected ArUco markers published on the '/aruco marker publisher/markers' topic, including their IDs and positions relative to the robot; and the occupancy grid map generated by the FastSLAM algorithm published on the *'/map'* topic.

IV. Experimental Results

A. ROSBags

To evaluate this FastSLAM implementation, several *ROSBags* were recorded (total of 14). For these *ROSBags*, 4 distinct maps, each with 5 landmarks were used. Within Maps 1 and 2, two different trajectories were used, a linear and a non-linear path. All of these trajectories were repeated three times. For each one of the remaining two maps, a non-linear trajectory was repeated three times.

B. Visualization of Results

A first evaluation of the results obtained by the FastSLAM implementation was done without computing any metrics. This was accomplished by using knowledge of the current map and comparing it with the camera feed and *RViz* representations, with and without occupancy grid.

In this context, we examined several key details: whether the bearing angle in the camera feed matched the simulation representation, if the final map accurately reflected the real-world locations and relative distances of the features, and whether the occupancy grid map in *RViz*, plotted with the features, resembled the layout of the 5th floor of IST's North Tower, where some of the ROSBags were collected. Figure 4 shows an example of this process.

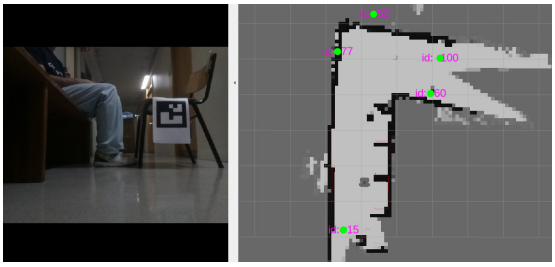

Fig. 4. RViz view of the camera feed, current known position of the features and occupancy grid map

C. Metrics & Ground-truth

The only way to evaluate the performance of the implemented FastSLAM is by using metrics. The chosen metrics are the following:

Mapping metric - SSE: The SSE (m^2) between the estimated pose of the landmarks and their ground-truth. In order to compare the 2 sets of points, they were rigidly transformed into the same reference frame, using the Kabsch Algorithm, section (II-C).

Trajectory metric - ATE: The ATE (m) represents the absolute deviation between a trajectory and its ground-truth. It's useful because is a concise way to evaluate the accuracy of the estimated trajectory produced by the implementation.

Trajectory metric - RPE: The RPE (m/m and rad/m) represents the relative deviation between consecutive poses of

a trajectory for both translation and rotation, when compared to the ground-truth. It is useful in the sense that gives a local scope of the accuracy of the trajectory, instead of a more global view, like ATE does.

For both types of metrics, the trajectories are rigidly transformed into the same reference frame using the Kabsh Algorithm, mentioned in section II-C.

The ground-truth used for the trajectory metrics was obtained by running the gmapping package. The trajectory was obtained by using the map and the transform from map to odometry frame published by this package.

As for the mapping metrics, the ground-truth position of the landmarks was measured in real life for each map.

D. Fine-tuning FastSLAM Parameters

The group's implementation of the FastSLAM requires the fine-tuning of some parameters, with the aim of improving its performance. The parameters to fine-tune are the following:

1) Alpha values (from the motion model equations)
2) Number of particles
3) Measurement Noise Covariance matrix (from the EKF filter)

To fine-tune these parameters, we used the metrics presented before and the *rosbags* from each map and trajectory.

First, let's denote from now on the Measurement Noise Covariance Matrix as Q, the Alpha values as α and the number of particles as N. To begin, let's fix the values of Q and N, to fine-tune the values of α. For the purpose of finding the best combination of values for these parameters, we used the SSE metric for the first two maps and trajectories. The values for α that lead to best results are in the table I and were fixed onwards.

TABLE I
VALUES FOR α

	α_1	α_2	α_3	α_4
Value	0.00008	0.00008	0.00001	0.00001

Moving on, the group will change the value of N, while fixing the values of α and Q, to see its effect on the SSE, for the different maps (for the trajectories with curves and loops). The variation of the value of N will be between 10 and 50, since after 50 particles the FastSLAM implementation became very slow, and bellow 10 particles the implementation would become more susceptible to noise. The effect of this variation on the value of the SSE can be seen in Fig. 5.

Considering Fig. 5, we can see that for a number of particles equal to 10 or 30, we got the smallest average SSE (taking into account all the maps). The group opted for an N equal to 30, due to the increase in robustness of the implementation, in comparison to a value of N equal to 10. In Fig. 5 the values of the SSE of the Maps 2, 3 and 4 were divided by 10, so that the values in the figure had the same scale. This increase in the SSE is due to the fact that in these Maps the landmarks were very distant from each other, whereas in Map 1 the landmarks were closer to each other.

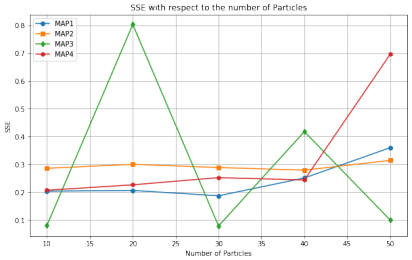

Fig. 5. SSE in respect to the the number of particles

Finally, having chosen the values for the parameters N and α, the group proceed to choose the values for the Measurement Noise Covariance matrix. We have that Q is a 1x2 matrix, accordingly we made the values of the elements equal to each other. This was due to the fact that we assumed that both measurements were equally noisy. (This Q matrix is in relation to the update of the position of a landmark, whereas the "Q" matrix for the creation of a landmark was kept at [0.1,0.1]). As such we began by looking for the best values of Q in terms of the SSE for the first trajectory in the first two maps. With the results present in Fig. 6.

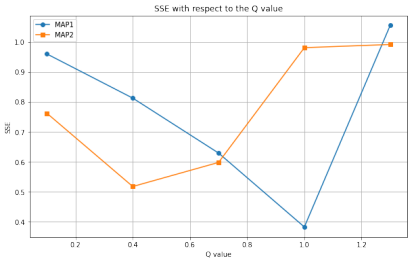

Fig. 6. SSE in respect to the value of Q

Due to the previous reason in Fig. 6, the value of the SSE for MAP2 was divided by 10. It is possible to see in this figure that the the value of Q that lead to the smallest average of SSE, between maps, was: $Q = [0.7, 0.7]$. Having this Q matrix, the group expected to see that this value would perform the best, e.g., for the RPE_{rot} in the map 3 and 4. The results for this metric is present in Fig. 7.

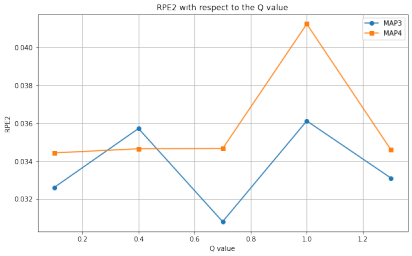

Fig. 7. RPE2 in respect to the value of Q

As before, we can see that the value of $Q = [0.7, 0.7]$ leads to the best performance in this metric. Since for the remaining metrics, except for the ATE, this values for the Q matrix lead

to almost the best performance, the group choose this values as the best.

E. FastSLAM Results

With the FastSLAM parameters fine-tuned we obtained the following results, which are the average of the three runs for each trajectory, for the previously mentioned metrics. These are shown in table II and in table III:

TABLE II
METRICS FOR MAP 1 AND 2

	SSE - Trajectory 1	***SSE - Trajectory 2***
MAP 1	0.6293	0.2323
MAP 2	5.9803	3.0944

TABLE III
METRICS FOR MAP 3 AND 4

	ATE	***RPE***$_{trans}$	***RPE***$_{rot}$	***SSE***
MAP 3	0.6232	0.0236	0.0308	0.7504
MAP 4	0.6440	0.0099	0.0344	2.2801

As mentioned before, maps 2,3 and 4 have landmarks very distanced between each other, which leads to a high SSE. Whereas in map 1 the landmarks are closer to each other, hence making the value of the SSE relatively small in comparison to other maps. Lastly, the values for the metrics: ATE, RPE$_{trans}$, RPE$_{rot}$ are also relatively small. All in all, we consider the values presented in both tables to be good for the problem in question. (We also took into account the fact that the result for the metrics of a given ROSBag would vary each time we would run it, however this variation came to about 4% of a given metric value, so we did not include this variation in the previous tables).

V. CONCLUSIONS

A. ArUco detection limitations

Regarding the detection of our landmarks, we found two significant limitations:

Firstly, the lightning of the environment our *Turtlebot3* was in, greatly affected the detection of the landmarks, making it almost impossible to detect any landmark under poor lightning conditions. Accordingly, the group made sure to have the best possible lightning conditions.

Secondly, the size of the *ArUco* markers also played a big role in their detection. For *ArUco* markers with small side lenghts, (usually 0.1 meters), whenever the robot was far away, (generally more than 1.7 meters), from the landmarks, the landmark detection algorithm was unable to detect these even though they were present in the image from the camera feed. For this reason, the group used *ArUco* markers with a side length of 0.25 meters, for all the recorded *rosbags*.

Concerning the value of the bearing given by the *ArUco* detection algorithm, the group initially had trouble in obtaining correct values for it, since, e.g., *ArUco* markers placed on a certain left part of an image corresponded to a value of almost zero for the bearing, when the value should have been instead quite different from zero. In spite of changing the code multiple times, nothing seemed to correct this inaccuracy, until we changed the value of the principal point present in the camera matrix, which came from the *Turtlebot3*. This has taught us to examine carefully any data that is used for our algorithms, even when it comes from reliable sources.

B. FastSLAM limitations

Regarding the FastSLAM limitations, it was noted how computational expensive running the algorithm with a high number of particles and using occupancy grid was. Running the occupancy grid implementation with, for example, 70 particles, led to breaks in the camera feed and time skips in the reading of the subscribed topics, therefore leading to loss of information and worse results. This could be fixed with code optimization. One idea is to store all the features' information on a matrix and updating the rows of the currently seen features, instead of doing so in a for loop. Another suggestion that would lead to better results is implementing the FastSLAM algorithm in C++.

This bad performance could be the explanation on the seemingly lack of relation between the number of particles and algorithm performance (as seen in 5) since it was expected generally more accurate results for a higher number of particles.

C. Final conclusion

To conclude, we believe the SLAM problem with known correspondence was solved with good results, as supported by the different maps, trajectories and metrics obtained. We are also happy with the progress done with the implementation of the occupancy grid bonus, believing it was a good middle-ground between an implementation with known and unknown correspondence.

In the future, we would like to work towards solving the limitations mentioned in this chapter, mainly the issues related to the performance of our code, as well as evolve to a full unknown correspondence FastSLAM implementation.

REFERENCES

[1] Durrant-Whyte, Hugh, and Tim Bailey. "Simultaneous localization and mapping: part I." IEEE robotics & automation magazine 13.2 (2006): 99-110.

[2] Montemerlo, Michael, et al. "FastSLAM: A factored solution to the simultaneous localization and mapping problem." Aaai/iaai 593598 (2002).

[3] Ribeiro, Maria Isabel, and Pedro Lima. "Autonomous Systems Lecture Slides." Autonomous Systems course, [IST], 2008.

[4] Saturn Cloud. "Bresenham Line Algorithm: A Powerful Tool for Efficient Line Drawing." Saturn Cloud Blog, 18 July 2023. https://saturncloud.io/blog/bresenham-line-algorithm-a-powerful-tool-for-efficient-line-drawing/.

[5] S. Thrun, W. Burgard, and D. Fox, Probabilistic Robotics. Cambridge, MA: The MIT Press, 2005.