# Markov Localization - Final Robotics Project

Michele Cattaneo
michele.cattaneo@usi.ch
Università della Svizzera italiana

Nicolai Hermann
nicolai.hermann@usi.ch
Università della Svizzera italiana

Oliver Tryding
oliver.tryding@usi.ch
Università della Svizzera italiana

CONTENTS



## I. INTRODUCTION

In this project, we implemented a Markov Localization algorithm in a simulated environment. This algorithm is used to probabilistically estimate the pose of a robot that is aware of its own environment but it is unaware of its current pose. For example, if the robot is re-positioned in its environment, it needs to understand its new pose. Markov localization explores a discrete representation of each possible pose of the robot within its environment. A probability distribution over these possible poses is kept track of and updated iteratively, given a movement model and sensor measurements. This distribution represents the likelihood that the robot is in a given pose. The state space, hence the space of all poses, needs to be properly discretized. Such a model allows for uncertainty to be introduced both in the sensor measurements and in the movement models, which makes it feasible to be applied in real world scenarios. In the last step, we implemented a version that uses a Monte Carlo Localization (or Particle Filter Localization) to overcome the downside of Markov Localisation that requires every possible pose to be associated with a probability value which has to be updated at each step.

## II. BACKGROUND

### A. Markov Localization

Markov Localization keeps track of a discretized probability distribution over the possible robot poses $p(l)$, where $l$ is a pose. A robot is equipped with sensors that provide measurements $i$ with a probability conditioned on a pose $l$ modeled as $p(i|l)$. This model allows to introduce uncertainty in the measurements, for example by modeling it as a normal distribution centered at the true measurement that would be produced if the robot was in pose $l$. Additionally, the robot has a movement model $p(l_t|l'_{t-1}, o_t)$ that models a distribution over poses $l_t$ at time $t$ given the old poses $l'_{t-1}$ and a command $o$ issued to the robot.

The beliefs about the poses are updated by iterating two computations that we can call `ACT` and `SEE`.

`ACT` uses a convolution over the old belief and the movement model for the issued command and can be expressed as follows:

$$p(l_t|o_t) = \int p(l_t|l'_{t-1}, o_t)p(l'_{t-1})dl'_{t-1}$$

`SEE` uses the update defined by the posterior distribution $p(l|i)$ given the likelihood of a sensor measurement given a pose and a prior on the pose, which is the current belief for the poses. Everything must be properly normalized by the evidence $p(i)$ which however is the same when updating any pose belief and can therefore be omitted during the computations. After all pose beliefs are updated, the values are normalized so that their sum is 1. The computations can be expressed as follows:

$$p(l|i) = \frac{p(i|l)p(l)}{p(i)} \propto p(i|l)p(l)$$

### B. Monte Carlo Localization

A clear downside of Markov Localization is the need of keeping track and updating a value for each possible pose at each time step. As the environment size grows quadratically, this task becomes unfeasible. A solution to this problem is the use of randomized sampling or particle filters, which

lead to the Monte Carlo approach for localization. In this approach, the belief state is approximated by keeping track of a subsample of all possible poses/states. The sampling process is weighted by the probability density function so that more samples are sampled around peaks of probabilities. Initially, $N$ samples are initialized randomly on a random pose $l$ associated with a uniform probability $p(l) = \frac{1}{N}$. Then the following iteration is performed:

- For each sensor reading $i$ (`SEE`): every sample has its probability updated as $p(l) = p(i|l)p(l)$ and normalized so that the total sums up to 1.
- For each issued command $o$ (`ACT`): For $N$ times, a sample $l'$ is sampled from the previous sample set with a likelihood given by $p(l')$ and a new sample at pose $l$ is generated according to $p(l|l', o)$. Every generated sample has a probability value of $p = \frac{1}{N}$.

A possible improvement is to draw some random samples at random poses to avoid the case in which the robot is completely lost.

## III. Project structure

The project assumes a grid world structure to simplify the setup and allow for easily interpretable visual results. The basic robot is positioned in a tile and its center of mass is placed in the center of a tile. There are 8 possible angles $\theta$ that the robot can take as an orientation. The allowed movements are forward and backward according to the current orientation. This means that the robot can move along an axis but also diagonally if its orientation is diagonal. The robot is equipped with a laser sensor of variable perceptive range. The laser interacts with the surrounding environment and whenever it intersects with an obstacle, it returns a measurement that can either be perfect or subject to noise. The project is structured so that it is possible to equip the robot with additional sensors or different kinds of sensors easily. The world is rectangular and polygonal objects (obstacles) can be placed around the world. The polygons can take any form but as soon as any part of the polygon occupies part of a tile, the whole tile is considered occupied and can not be walked on.

### A. Code and external libraries

The whole code base is written in Python. For computations and matrices handling we used Numpy. For convolutions and sampling from probability distributions, we used SciPy. For the GUI we used PyGlet which is a library for Python that provides an API for the creation of games and applications that require a GUI. PyGlet has the advantage of having no external dependencies and is straightforward to use for simple interfaces. Behind the scenes, every object is backed up by a geometric representation. This is done with the help of Shapely, which is yet another Python library that handles the manipulation and analysis of planar geometric objects. We mainly used this library to efficiently compute intersections between objects, such as a laser sensor represented by a line segment and an obstacle represented by a polygon. A requirements file containing the relevant packages will be available together with the code.

### B. Code Architecture

To adhere to the *open for extension, closed for modification* principle we aimed to make the code as modular as possible. To achieve this we used the Model View Controller pattern accompanied by the Observer pattern. This allowed us to reuse many parts of the code, for example, the continuous and discrete robots use the same View. Similarly, we introduced abstract classes to unify and abstract most implementation details. This enables us to implement new sensors, movement models, or environments without having to change existing code in any of the other components as different components only call abstract methods. All global parameters are collected in one file called `definitions.py` that can be used to select between different components and hyperparameters.

### C. Movement Model

The movement model for the robot can either be deterministic or subject to uncertainty.

*1) Deterministic Movement Model:* In the case of a deterministic movement model, we represent the movement as a simple matrix M. Such a matrix is used as a kernel for a convolution that is applied to the current belief and is appropriately rotated according to the possible orientations of the robot obtaining $M_{\text{up}}, M_{\text{up-right}}, M_{\text{right}}, M_{\text{down-right}}, M_{\text{down}}, M_{\text{down-left}}, M_{\text{left}}$ and $M_{\text{up-left}}$.

$$M_{\text{up}} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, M_{\text{up-right}} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \dots$$

All together these matrices can be stacked together creating a three-dimensional kernel that can be convolved over the three-dimensional matrix that contains the robot's belief. The convolution is however applied on each of the third dimensions independently. For each possible orientation $\theta$ the convolution is therefore defined as:

$$B_{a,b,\theta} = \sum_{i=0}^{w} \sum_{j=0}^{h} M_{i,j,\theta} \cdot B_{i+a,j+b,\theta}$$

where $B$ is the three-dimensional matrix containing the current belief and $M$ is the three-dimensional kernel. This formula omits the padding for simplicity.

*2) Movement Model subject to uncertainty:* An uncertain movement model is defined as the previous one, however, it is further parameterized by a vector of probabilities $p1, p2$, and $p3$. These three values represent the probability of executing the issued action, executing no action, and executing the opposite action, respectively. In such a setting the movement can be represented as follows:

$$M_{\text{up}} = \begin{bmatrix} 0 & p_1 & 0 \\ 0 & p_2 & 0 \\ 0 & p_3 & 0 \end{bmatrix} \text{ such that } \sum_{i=1}^{3} p_i = 1$$

The other matrices are defined analogously. This setting can be scaled to more complicated movement models, for example by increasing the kernel size, keeping the same overall logic.

### D. Kidnapping

A rather unconventional way to include uncertainty is to reposition the robot arbitrarily without telling him. Also, represent a stress test for our implementation. Issues that we encountered were that some probabilities were actually zero because of floating point imprecision. We fixed that by adding $2e-16$ to each belief after normalizing to make sure that no probability reaches zero. After making this required change the robot successfully recovered after the kidnapping. Through a mouse click the robot can be repositioned to any free tile and we successfully tested it in certain, uncertain, and Monte Carlo settings.

### E. Sensing Model

Analogously to the movement model, the sensing model can be either deterministic or subject to uncertainty. The sensing model is defined as $p(i|l)$ in the `SEE` step, hence the probability of getting a sensing input $i$ given a pose $l$.

*1) Deterministic Sensing Model:* In the case of a deterministic sensing model, this probabilities are defined as:

$$p(i|l) = \begin{cases} 1 \text{ if } |i - i^*| \leq \epsilon \\ 0 \text{ else} \end{cases}$$

for $i^*$ the true measurement that a perfect sensor would measure at pose $l$ and a small $\epsilon$ to allow for floating point numerical imprecision.

*2) Uncertain Sensing Model:* The uncertain sensing model was defined instead as Gaussian distributions centered at the true measurement $i^*$ and with a certain variance $\sigma^2$ which was exposed as a hyperparameter:

$$p(i|l) \sim N(i^*, \sigma^2)$$

### F. Probability Visualization

The Monte Carlo approach keeps track of a multidimensional matrix that contains probabilities for each possible robot's pose. Recall that in our case the robot has three degrees of freedom, hence $(x, y, \theta)$ which represents a position in the plane and a certain orientation. This led to the difficulty in deciding how to represent those probabilities in two dimensions (our GUI). The obvious solution is to somehow aggregate the probabilities along the orientation dimension. The easiest approach is to simply add them up along that dimension at the cost of losing the information about the orientation. This means that if a pose, represented by a tile, has a certain probability, we can't tell what orientation is contributing to that probability. A simple approach is to represent the probabilities by the tile color's opacity (the alpha channel) so that highly probable tiles have a more opaque color and lesser probable tiles have a more transparent color. Initially, we applied a linear scaling from probabilities $p \in [0, 1]$ to opacity intensities $\alpha \in [0, 255]$. This seemed to work well until we noticed that sometimes the tile onto which the robot was residing seemed to have a 0 probability, just to then recover later as time passed. Instead of being a bug, we realized that this issue was simply caused by our eye not being able to pick up the difference between opacity values below a certain range. For this reason, we replaced the linear scaling by a gamma correction to give a bigger range of opacities to small probabilities. Figure 1 shows some examples of gamma correction for a given choice of $\gamma$. In our case, the $x$-axis represents the probability and the $y$-axis represents opacity.

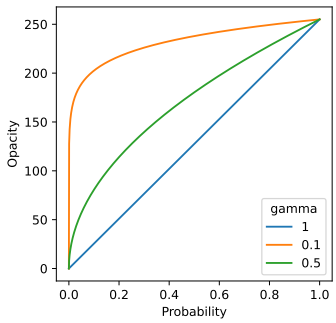



Fig. 1: Gamma correction examples.

After several trials, we realized that a good choice was $\gamma = 0.1$ in order to obtain a nice visualization. In section V (Results) we display some examples of those visualizations. A better approach would be to have $\gamma$ be the result of a function of $w$ and $h$, the width and height of the environment. This is because the larger the world, the smaller the probabilities will be as they are spread over a larger area.

Another idea was to use an HSV color space to allow the various probabilities obtained from the different orientations to be differentiated. Figure 2 shows an HSV color space. This 3-dimensional space is an alternative representation of the RGB color model. The idea was to map each of the possible robot's orientations to a certain color, hence a hue value. Then the saturation value could have been used proportionally to the probability. Mixing colors according to the various orientations results in some color that is not interpretable. Therefore only the most likely orientation could have been used and its color displayed. In the end, we opted not to use this approach because the grid's color would have still looked too hard to interpret due to the presence of many colors. After all, it is still hard for humans to make sense of and compare colors. Also, the environment was not colored smoothly as the most likely orientation frequently changed from tile to tile making it a mosaic pattern that was impossible to interpret for us.

## IV. Continuous World Representation

Transitioning from a grid-based to a continuous representation of the environment introduces new challenges and complexities in our localization efforts.

Increased computational complexity: In a continuous environment, the state space is infinite, which makes exact computation and representation impossible. As a result, we

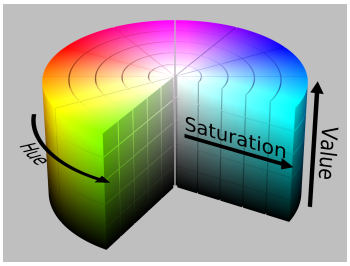



Fig. 2: HSV color space. Source: wikipedia.org/wiki/HSL_and_HSV

often need to use approximate methods such as particle filters, which increased computational complexity.

Handling Sensor Noise: The continuous representation of the world necessitates meticulous management of sensor noise. Given that the robot's position can be at any point within this continuous environment, minor inaccuracies in sensor readings can potentially lead to significant deviations in the estimated position, thereby compromising the localization process.

Data Association Challenges: In a continuous world, associating sensor observations with corresponding landmarks or features in the environment presents additional difficulties. This will be discussed further in the Monte Carlo Localization as it is highly relevant to the model.

### A. Monte Carlo Approach

We have implemented Monte Carlo Localization, also known as Particle Filter Localization, as a localization model for a continuous environment. Monte Carlo Localization is a method that overcomes the limitations of grid-based localization by using a set of random samples or "particles" to represent the robot's position and orientation within a continuous space. Each particle signifies a potential state of the robot, with the collective distribution of particles forming a probabilistic belief map. Given a known map, $M$, the goal of Monte Carlo Localization is to compute the posterior distribution of the robot's pose, $l_t$, at time $t$, given the history of control signals, $o_1 ... o_t$ and sensor measurements, $i_1 ... i_t$.

$$P(l_t | o_{1:t}, i_{1:t}, M)$$

This posterior distribution is approximated using a set of $N$ particles. The algorithm begins by dispersing these particles randomly throughout the environment. Each particle represents a hypothesis about the robot's pose, $l_t$, i.e. the position (x, y coordinates) and orientation (one of eight directions). Initial weights, $w_t$, are assigned to the particles in a uniform fashion as the exact location of the robot is not known at the outset.

As the robot moves, the particles are displaced according to a motion model which mimics the robot's movement. The motion model takes into account the control inputs and inherent uncertainties such as noise in the robot's movement. This step introduces a predicted belief about the robot's pose before incorporating sensor measurements. The control signals are incorporated through the motion model:

$$l_t = g(o_t, l_{t-1})$$

Where $g$ is the function that applies the control signals to the previous pose.

Upon receiving sensor measurements, the weights of the particles are updated according to a measurement model. The model evaluates how likely the robot would have perceived the current sensor readings if it were in the state represented by each particle. Particles that align closely with the sensor data receive higher weights. The weight of a particle is computed based on the measurement model:

$$w_t = P(i_t | l_t, M)$$

This forms the basis for resampling in the next phase.

Particles are resampled proportional to their weights. This results in the multiplication of particles with higher weights (those that better represent the robot's state) and the elimination of particles with lower weights. In mathematical terms, if we denote the new particle set by $L'_t$ and the old set by $L_t$, this can be written as:

$$L'_t = P(L_t | w_t)$$

The prediction-update-resampling loop repeats with every new motion and sensor measurement, gradually converging to a set of particles that closely approximates the true pose of the robot.

### B. Perception Aliasing

Perception aliasing is a significant challenge in localization, particularly in environments with similar or symmetric features. This phenomenon arises when different places in the world appear identical or nearly identical to the robot's sensors. Perception aliasing can lead to multiple peaks in the posterior probability distribution of the robot's pose, hindering effective localization.

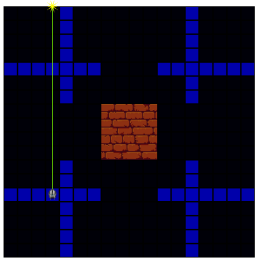
To mitigate the issue of perception aliasing and enhance the robustness of the localization, we incorporate two strategies: jittering and effective sample size thresholding.

Jittering is a technique that adds random noise to the particles' state during the resampling phase of Monte Carlo Localization. The introduction of noise, or jittering, diversifies the pool of particles, ensuring that the range of hypotheses about the robot's state remains broad and resilient to aliasing effects. Jittering can be formally described as:

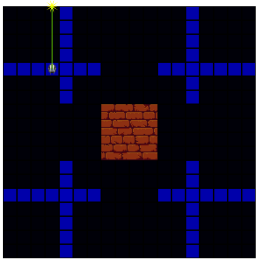
$$l'_t = l_t + N(0, \sigma)$$

where $N(0, \sigma)$ is a zero-mean Gaussian noise with standard deviation $\sigma$, and $l'_t$ represents the jittered state of the particle.

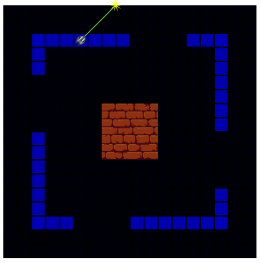
Effective sample size (ESS) thresholding is a measure of the number of particles that significantly contribute to the representation of the robot's pose distribution. When perception aliasing causes particles to concentrate around a few hypotheses, the ESS may decrease significantly. By monitoring the ESS, we can preemptively identify situations where perception



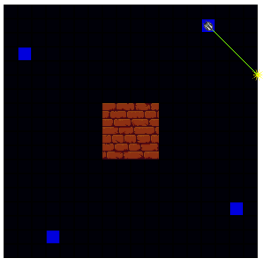
(a) Belief after 1 step.



(b) Belief after a few steps.

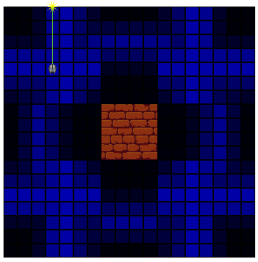


(c) Belief after more steps.

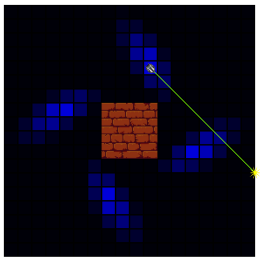


(d) Belief after a few steps and more rotations

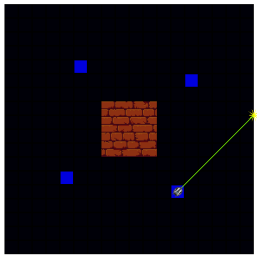
Fig. 3: Three stages of the robot's pose belief in a simple environment with certain sensing and certain movements.



(a) Belief after a few steps.



(b) Belief after more steps.



(c) Belief after rotating in place for some time.

Fig. 4: Three stages of the robot's pose belief in a simple environment with uncertainties.

aliasing is likely to cause localization failure. When ESS falls below a certain threshold, we can trigger a resampling of the particles. This can be represented mathematically as:

$$ESS = \frac{1}{\sum_j = 1^N (w_t)^2}$$

where $w_t$ is the weight of the particle.
If ESS $<$ threshold, then we resample.

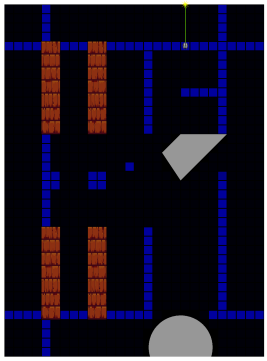
## V. Results

In this section, we will go through some of the obtained results by showing a few examples of environments and settings. To begin with, in Figure 3 we can see three stages of the robot's pose belief in a simple environment with certain movements and certain sensing. Because the environment is symmetric and it is only equipped with a single laser sensor, the robot can not figure out a single position but rather has 4 equally likely poses. In Figure 4 we show the results of a robot in the same environment but with an uncertain sensing and movement model. The overall probabilities are more spread because of those uncertainties. However, the robot is able to locate 4 equally likely poses if it starts rotating in place. With these rotating actions, the sensing model will slowly make the probabilities converge to 4 single tiles. We can note that having certain movements and certain sensing makes the probabilities converge extremely quickly. The only issue is the symmetry of the environment which makes it impossible to ultimately localize the robot.

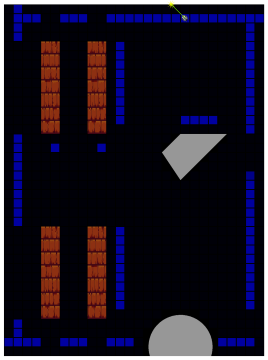
Figure 5 shows a few steps in a more complex environment with certain sensing and movements. Despite the more complex environment the robot quickly localizes itself after a few steps. Using certain sensing and movements is not very interesting and it should mostly be considered as proof that the code is working correctly because of the easily interpretable results.

We now move on to more interesting results considering uncertainties in the same more complex environment. Figure 6 shows a few stages of the localization. We can see that after a few steps, especially due to the high uncertainty in the sensing, the probabilities did not change much from a uniform distribution. It then takes a few more steps to obtain a bunch of clouds of probable poses.

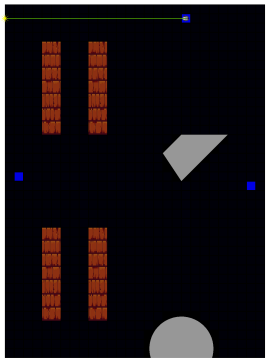
In the end, after even more steps the probabilities converge to a single cloud, and by performing rotations in place the cloud shrinks in size and slowly converges to the tiles just around the robot. Given more time and more rotations, the probabilities will all converge to the correct tile. Because of the uncertain movements, even after the probabilities fully converge to a single pose, when the robot starts to move around again, the cloud of probabilities will again grow in size as the uncertainty spreads probabilities to surrounding tiles. In Figure 7 we show yet another example of a few localization stages in a highly symmetrical environment. In the beginning, every tile is equally likely. After a few movements, the robot can be in any room, and after more steps, the probabilities converged to two possible rooms. With enough time, the belief will converge to the correct room because each of them contains a unique polygon which results in unique true measurements.



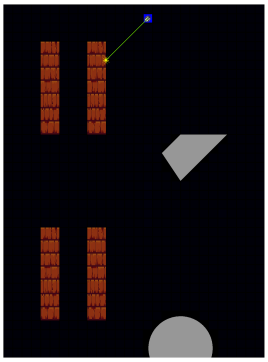
(a) Belief after a few steps.



(b) Belief after more steps.

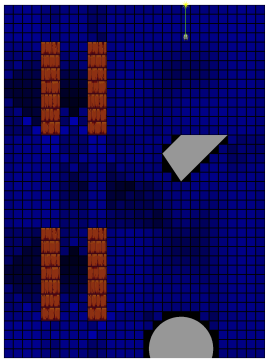


(c) Belief converged after another rotation.

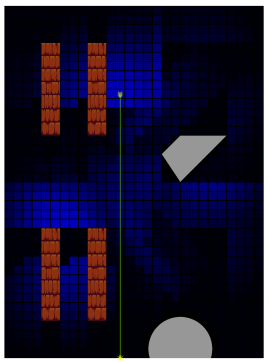


(d) Belief after sensing the brick wall.

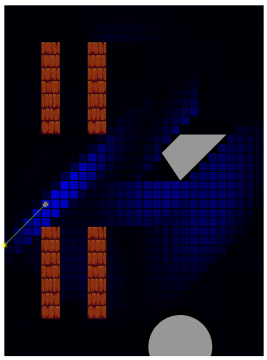
Fig. 5: Four stages of the robot's pose belief in a more complex environment with certain sensing and movement.



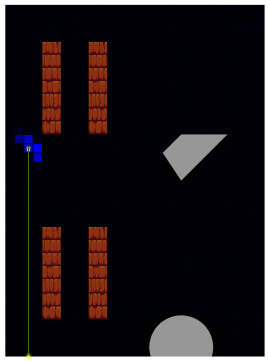
(a) Belief after the initial few steps.



(b) Belief after more steps.

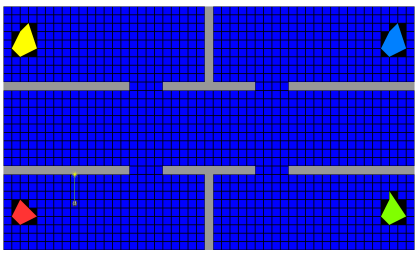


(c) Belief after further steps.

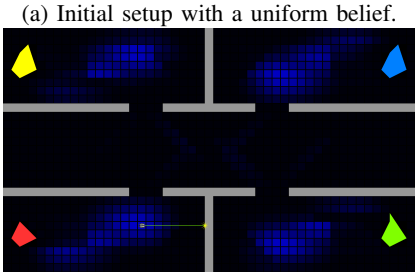


(d) Belief after many steps and rotating in place for some time.

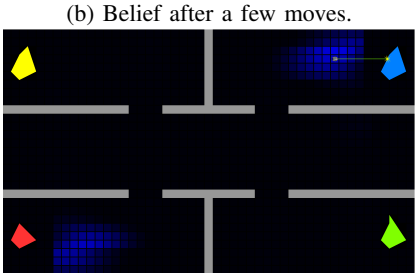
Fig. 6: Four stages of the robot's pose belief in a more complex environment with uncertain sensing and movement.



(a) Initial setup with a uniform belief.



(b) Belief after a few moves.



(c) Belief after further moves

Fig. 7: Three stages of the robot's pose belief at three time steps in a highly symmetric environment.

Figure 8 shows an example of the Monte Carlo Localization and the particles at different stages of the simulation.

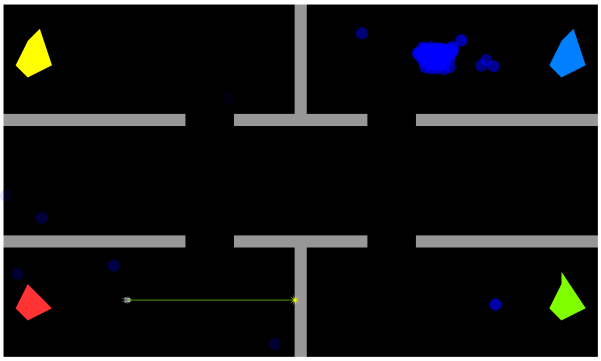
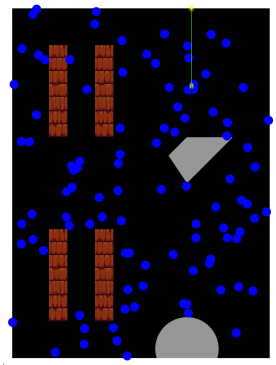
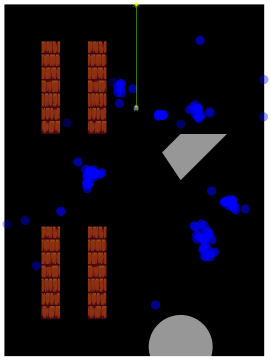


Fig. 9: An example of perception aliasing for Monte Carlo Localization.

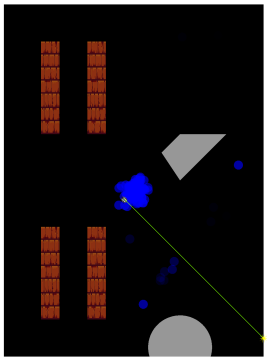
In 9 we see an example of perception aliasing in a highly symmetric environment.



(a) Initial particles around the environment.



(b) Particles about to converge.



(c) Particles converged

Fig. 8: Three stages of Monte Carlo Localization

## VI. Conclusions

In this project, we implemented two interesting approaches to the localization of a robot in a known environment. The first one is Markov Localization and the second one is Monte Carlo Localization (or Particle Filter). The first approach keeps track of a probability value for each possible pose that the robot can take. Such an approach is easily interpretable but quickly becomes prohibitive for large environments or large state spaces. The second approach overcomes this issue by only keeping track of a fixed-sized subsample of possible poses. This approach allows the state space to be continuous and in general requires less memory to execute. The tradeoff is a potentially slower convergence. The result we obtained is an application with an intuitive GUI that lets the user play around with different parameters settings and environments to visualize the strengths and weaknesses of these localization algorithms. This application can be used for educational purposes. Overall we are very happy about the results we obtained, especially since we managed to stick to our expectations and even added a few extra new features during the process.

## VII. Limitations and Outlook

In this section, we want to discuss the limitations of our implementation and future changes that could be made to further explore and understand the localization paradigm. The main constraint of our system is the limited number of rotations. First of all, this makes the set of possible measurements quite restricted and positions tend to be similar to each other due to the perception resolution of each position. There are only eight distances per position which makes it impossible to perceive for example unique objects further away as the laser can only hit them in one orientation. However, if they are unique enough the measurement might be unique which immediately solves the localization problem. This issue is solved by introducing uncertainty though. Another limitation is that only one action at a time can be taken and the robot is not able to move forward while turning. A possible improvement would therefore be to implement a better kinematics model such as differentiable drive kinematics. We made the choice of very simple kinematics due to time restrictions and a simple model allowed us to debug the code better. Also it simplified the visualizations a lot which was beneficial for us.

Besides continuous rotation, we would have been keen to explore different sorts and numbers of sensors. Just to name a few, it would have been interesting to include bumper sensors or all-or-nothing sensors that only signal if there is something within a given distance or not. Another possibility would have been to add a landmark sensor fixed at a certain location, that keeps sending distance measurements to the robot. Additionally, it would have been interesting to equip the robot with multiple lasers. The way our code is structured should allow this modification to be easily implemented.