This subsection describes the implementation and reasoning behind localizing the robot's position in the environment. It is assumed that the initial position is known

Goal:

- Localize the robot in the environment.

0.0.1 Reasoning

There are multiple ways of localizing a robot in an environment, and especially when both the map and the initial pose are assumed to be known. The first idea was to use feature extration and comapre what the robot saw through its lidar scanner to the features in the map. This method was tested and discarded. Instead the choice fell between Kalman filtering and particle filtering. The particle filter has been chosen as it is more flexible as the Kalman filter as it does not need to know the initial pose and it can work with an environment represented by an occupancy grid. To compare the particles and their position with the robot's position, raycasting, to determine what the particles see, will be used.

0.0.2 Implmentation

The particle filter implementation has been split up into two classes; Localization and laserscanner. The particle filter in itself is placed in the Localization class, while the laserscanner is a utility class.

Lasesrscanner

The laserscanner class provides an interface between the lidar and pose data from gazebo and the Localization class. The laserscanner class provides the functions to perform raycasting on the generated particles.
It also provides the three functions that needs to be updated in order for the particle filter to generate a new prediction on the robot's position. They are updateLidar(), that gets the robot's lidar data (angle and range of all 200 rays that is generated for the robot) and store them in two vectors, updatePos(), that updates the robot's position according to gazebo and lastely updateSpeed, which gets the robot's angle velocity and speed from gazebo. This has been done because all three sources of information has to be updated between running the particle filter, or else the particle filter might make its prediction based on old data but compare the estimated particles position with a new Lidar scanning and hence give a false negative. To ensure the particle filter does not make a new prediction unless new data has been recievced by laserscanner, each function sets a flag high, and the predict() function in the Localizition class will check if the flag is high or not.

The laserscanner class has three other function; rayCasting() performs the raycasting based on an (x,y) coordianteset and the angle[1] β while drawLines() adds the pixels coordinates of each genrated ray to a vector. 200 rays are generated by rayCasting per particle. The last function, calDistance(), calculates the distance of each ray based on the pixel values given vector from drawLines(). The vector either contains no black pixels and the length will be based on the starting coordinate and the end coordinate or if a black pixel is in

[1] the raycasting is based on the lidar max range and the angle incrementation of the lidar scanner.

the vector, the endpoint will be the black pixel instead of the last element in the vector. The drawLines() and calDistance() functions are based on Bresenham's line algorithm.

It is important to mention, that the lidar data provided by gazebo is given in meters. So a conversion from meters to pixels has to be performed in order for the conversion between the robot's rays and the particles rays to be calculated corretly. It is also worth noting that the robot sees its initial position as (0,0) allthough it starts in the middle of the map. This will be important later on.

Particle filter

The particle filter is made up of 5 functions with 3 of them being the main functions. A particle filter consists of a prediction function, that makes a prediction based on the former (x,y) coordinate and angle and the speed and angle velocity with which the robot moves, a update function and a resampling function.

The prediction function generates 200 particles[2], and each contains an estimate of where it thinks the robot is. To ensure that as many possible positions of the robot is covered, the estimates gets a random Gaussian noise added on the speed and angle velocity, see figure **??**.

$$x_{k+1} = x_k + (v_k + \delta v_k) \cdot \Delta T \cdot cos(\phi_k) \tag{0.1}$$

$$y_{k+1} = y_k + (v_k + \delta v_k) \cdot \Delta T \cdot sin(\phi_k) \tag{0.2}$$

$$\phi_{k+1} = \phi_k + (\omega_k + \delta \omega_k) \cdot \Delta T \tag{0.3}$$

Figure 1: *x_{k+1}, y_{k+1} and ϕ_{k+1} are the predicted next position and direction of the robot. x_k, y_k, ϕ_k is the current position in x, y and angle. $v_k and \omega_k$ is the current velocity and angle velocity of the robot. The $\delta\omega_k$ and δv_k is the added Gaussian noise.*

The first predict does, however, not occur in prediction(). It is generated in the init function, as the particles needs to be generated along with their weight, last known coordinates and so on, as the prediction function relies on this information to estimate the new position. As the initial position is known, the first particles position will be generated uniformly around this area, see listening **??**. This means that a conversion between the (0,0) coordinate provided by gazebo needs to be converted to the middle of the map, else the initial prediction will be way off - as it happened in the first iteration of the particle filter, see figure **??**.

Listing 1: *Codesippet of the initial prediction of the particles*

```
1 uniform_real_distribution<double> vel_variance(-2.5, 2.5);
2     uniform_real_distribution<double> angle_variance(-2.89, 2.89);
3     first_flag = true;
4
5     particle p;
6     for(int i = 0; i < N; i++)
7     {
```

[2]the higher the particle count, the more precise estimate of the location

```
8           r_x = (ls.robot_x + 42)/0.07;
9           r_y = ((-1)* ls.robot_x + 20)/0.07;
10          //gives all starting particles a random starting point in a 5x5 cube
11          double start_x = r_x + vel_variance(gen);
12          double start_y = r_y + vel_variance(gen);
13          double start_beta = ls.robot_angle + angle_variance(gen);
14          current_time = time(&timer);
15          p.last_time = current_time;
16          p.x = start_x;
17          p.y = start_y;
18          p.beta = start_beta;
19          p.weight = 1/200.0;
20          p.likelihood = 0.0;
21          // checks if any of the particles are out of the map, if so, it tries again
22          if(checkCoordinates(start_x, start_y))
23          {
24              //update the predicted values and last time.
25              samples.push_back(p);
26          }
27          else
28              i--;;
```

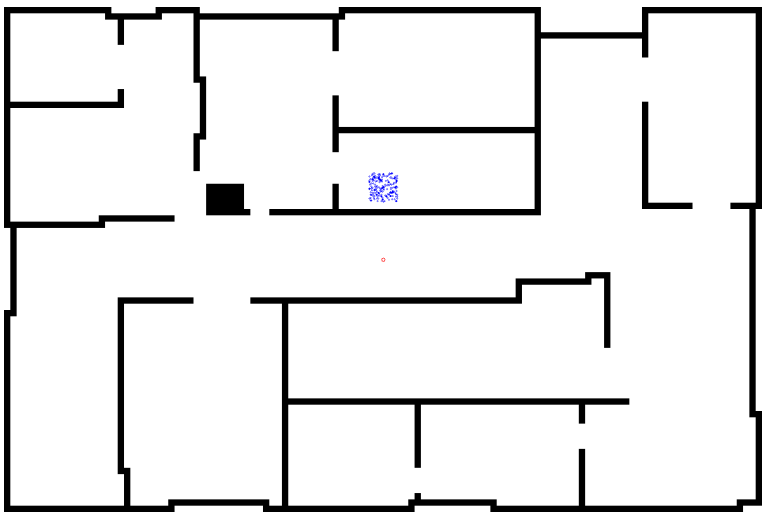

Figure 2: *The figure shows the initially generated particles, blue circles, and the initlal pose of the robot before the conversion mistake was fixed.*

After the prediction function has updated the estimates, it calls the updatePos()[3].

The updatePos() starts by updating a vector variable with the robot's rays lengths[4] as well as their relative angle seen as robot. Next it calls the rayCasting() from Laserscanner to generate the particle rays and get the length for all 200 particles. Then it calculats the likelihood of every particle being on or very close to the robot's actual location using maxium likelihood. The likelihood is based on the order and length of the particles rays, $\hat{d}_j$ and the robot's rays d_j. The more smaller difference between expected length the particles generate and the actual length the robot see, the more likely it is that the particle is in the right spot. The likelihood is computed using a Gaussian distribution equaltion as the sensor noise is assumed to be zero mean and variance σ^2:

[3]Not to be confused with updatePose from laserscanner
[4]obtained from gazebo

$$\frac{1}{\sigma \cdot \sqrt{2 \cdot \pi}} \cdot \exp - \frac{(\widehat{d_j} - d_j)^2}{2 \cdot \sigma^2} \tag{0.4}$$

Figure 3: *$\widehat{d_j}$ is the expected measurement obtained from the ray ccasting of each particle and the actual measurement taken from the robot's rays are given by d_j [**?**]*

The new weight of all particles are computed based on their likelihood. The particles weights are normalized afterwards, see listening **??**. UpdatePos() calls the updateMap() function before the last main function, resamping(). UpdateMap() is purely a help function to visualizes the particles and the robots actual postion, see figure **??**.

Listing 2: *Code of likelihood calculation and normliazing of aprticles weight*

```
1 //calculate the likelihood for a particle is around the robot for all particles.
2
3  double norm_const = 0.0;
4  vector<double> temp_weight;
5  const double extra = 1/(sigma*sqrt(2*pi));
6  double sum = 0;
7  for(int i = 0; i < N; i++) // samples size
8  {
9      double likelihood = 1.0;
10     for(int j = 0; j < N; j++) //nranges size - FIX
11     {
12         likelihood = likelihood * extra *exp(-(pow(samples[i].ray[j].distance-robot_rays[i]
13     }
14     //save the likelihood for the particle
15     samples[i].likelihood = likelihood;
16     //cal new weight based on the likelihood of the particles rays.
17     temp_weight.push_back(samples[i].weight * samples[i].likelihood);
18
19
20     // get the sum of all weights for normalizing.
21     norm_const = norm_const + temp_weight[i];
22 }
23 //normalize new weight of particles
24 for(int i = 0; i < N; i++)
25 {
26     samples[i].weight = temp_weight[i]/norm_const;
27     sum= sum + samples[i].weight;
28   //   cout << samples[i].weight << endl;
29
30 }
```

The function resampling is performed to avoid a small number of highly weighted particles to dominate the estimate of the robot location. It resamples all particles that are not over a certain threshhold and make the new particles based on the ones that are over the threshhold. All 200 particles will be resampled and get assigned an equal weight in accordance with the pseudo code from our textbook [**?**].

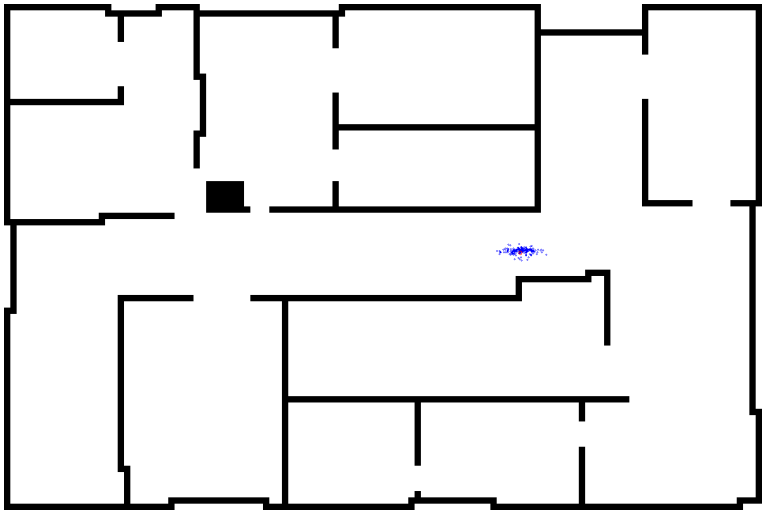

Figure 4: *The figure shows the 200 particles, blue circles, following very closely around the actual position of the robot, red circle.*

As can be observed on figure **??**, the particles are not spot on the actual position. They are however, fairly close to the actual position. The cluster of particles does not in themselves give an "actual" position of the robot, as it is merely a cluster of believed positions. Instead a method to obtaining the most likely postion would be taking the average of the the particles (x,y) and pick that as the best possible estiamte of the location. The poarticles generated in the initial prediction appears to be as close to the robot as one might expect. It does before much more closely clustered the closer to the hallway the robot gets.. It is suspected that this phenonenom occurs because of the uniform distribution used in the initial prediction, see listening **??**. If the position had been unknown from the start, the uniform distribution would have been a good choice, however not with the current bounds used, but as the position IS known, Gaussian would potentially been a better pick.

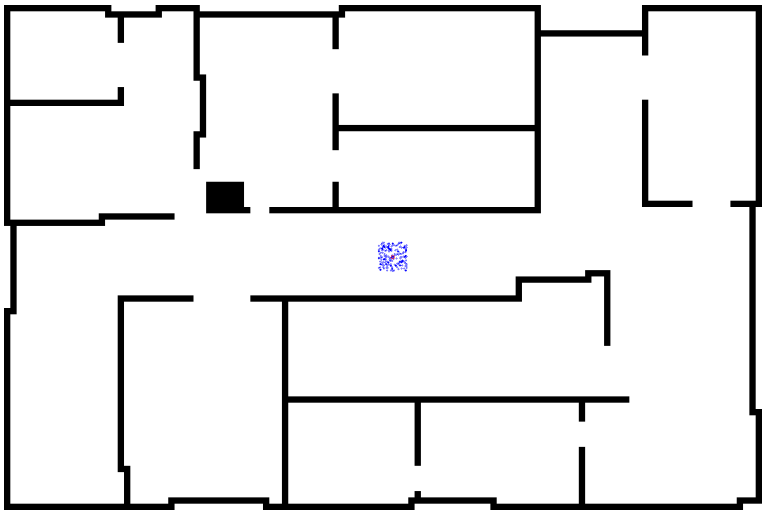

Figure 5: *The figure shows the initial position of the 200 particles, blue dots, and the robot, red dot - using an uniform distribution to generate the particles.*

One thing that became apparent was, that if the particles first believes in the wrong

particle, it can take quite a long time before it comes back. There is an issue with the particle filter, if it is used in the map when there is marbles present. This is because only the robot can see the marbles and the lidar scannings are affected as if there is a wall while the particles rays are generated from the provided map, which does not include the marbles. This means, that in a worst case scenario, where the robot is left standing looking at a marble close up, the particles might start migrating and completely miscalculate the location, see figure **??**. If the robot moves the particles will eventualy find their way back.

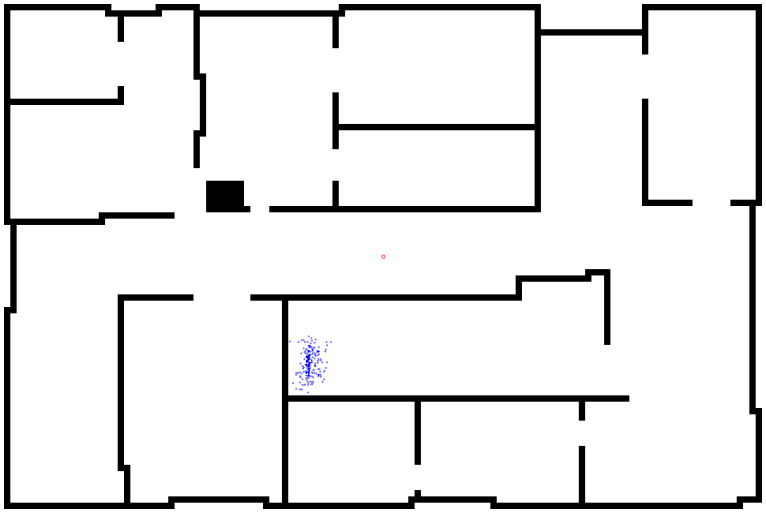

Figure 6: *The figure shows the initial position of the 200 particles, blue dots, and the robot, red dot, not reacting well to a difference in the what the robot sees and what the particles think it should see*

0.0.3 Conclussion

The particle filter works quite well, all though it would not yet be suited to actually be used in the map with marbles in it. It could maybe be better to switch out the uniform distribution for a Gaussian in the init() function, however it is not the biggest of issues. More turning using the sigma variable in the most likelihood calculation might solve how spread out the particles are in figure **??**.