# Death by Particles:
# An Efficient Brute Force Approach to Robot Localization

Nick Steelman
Benjamin Ziemann

October 12, 2018

The little robot that could, part 2: Find one's self

All code referenced in this document and more can be found in the project's git repo: https://github.com/zneb97/robot_localization

## 1 Robot Localization

### 1.1 Overview

Tasked with the problem of robot localization, we attempted to tackle the problem in the most computationally efficient way on a per-particle basis. This involved eschewing any large or complex computation for any given particle in favor of having more particles, 20,000 in our case. Our code flow is as follows:

```
//N = Number of Particles
//P = Pixels in Map
//M = Runtime of Nearest Neighbors ≈ O(Plog(P))
initialize_particles();
while predicting_position do
    check_distance_since_last_measurement(); //O(1)
    if distance > threshold then
        reset_last_measurement(); //O(1)
        move_particles_by_distance(); //O(N)
        update_likelihoods_of_particles(); //O(N)
        delete_bottom_percent_particles(); //O(Nlog(N)
        initialize_new_particles(); //O(N)
    end
    update_nearest_neighbor_map(); //O(M)
end
```

**Algorithm 1:** Our algorithm's workflow

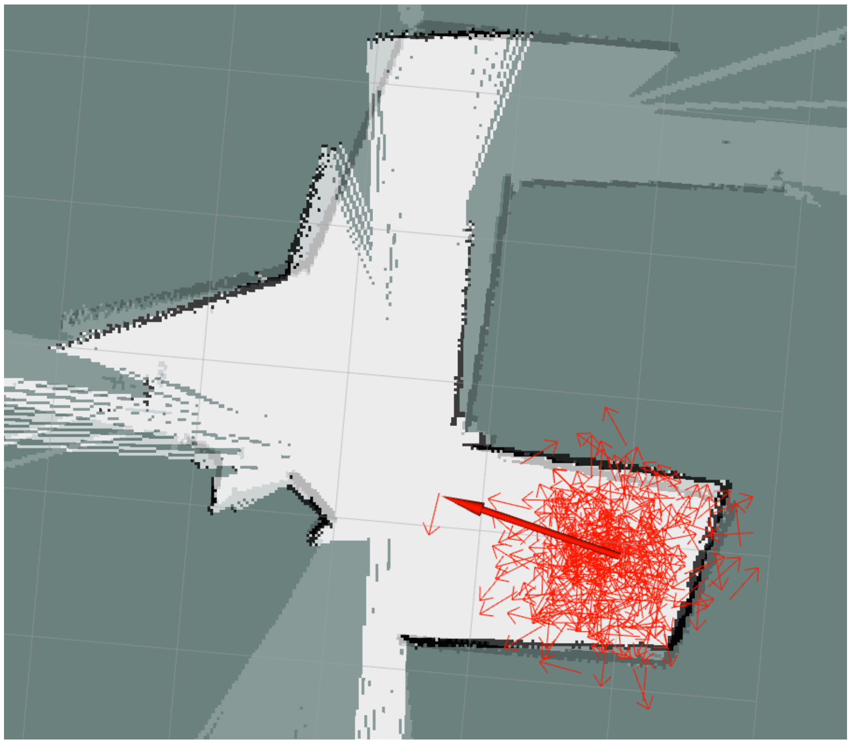
This report will detail (in order):

1. How we initialized the starting particles
2. Our method of checking distance and moving the particles in the map
3. Our method of weighting the particles by likelihood of being the true position of the robot
4. When we chose what particles to eliminate
5. How we chose to add in new particles

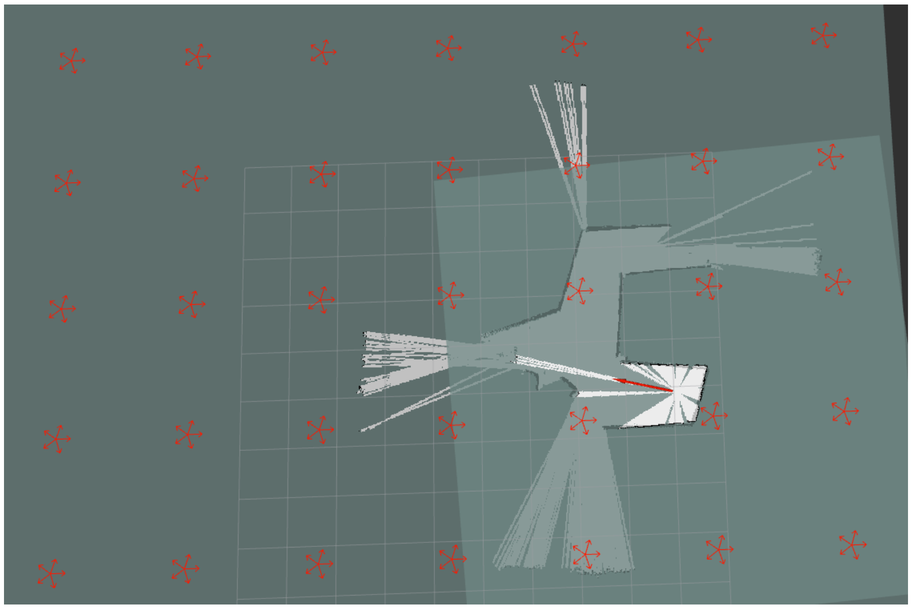
## 2 Initializing Particles

To trigger the initialization of our particles we wait for a 2D pose estimation of where the robot is from rviz (code provided). Note that with our algorithm this estimate does not need to be accurate in position or heading, it just has to be within the confines of the map. For initializing the location and heading of the particles we tried three different approaches.

For the first pass we normalized the distribution of the particles' location around the given 2D pose estimate. The headings were also normalized in full circle, with the pose estimate's heading being our center value. While this ended up working, it depended on having a decent estimate of the robot's actual position on the map ($\sim$ 2 meters). Heading didn't matter because of normalized yaw distribution. The reason we moved away from this was because we wanted to solve the "kidnapped robot" problem and the distribution of this was too small. While we could have initialized the spread to be larger, we instead chose to utilize more of the map.

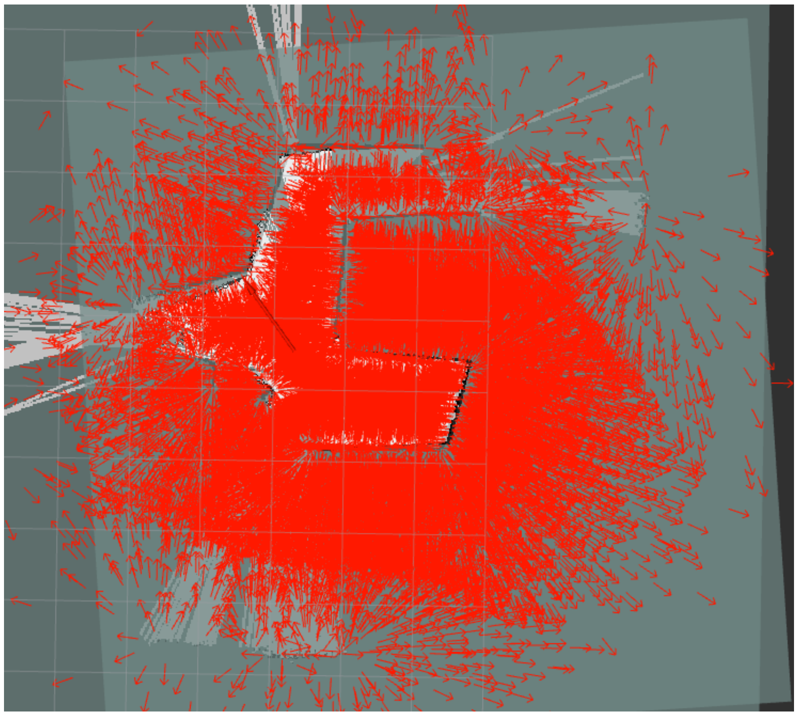

Normalized particle initialization

Another option we also explored was uniformly distributing our particles around the map. This made it much more akin to the kidnapped robot problem while greatly decreasing the number of particles that needed to be initialized. The main problem with this was the bounds of the entire map area (light blue in rviz) is much, much larger than the physical space the robot can traverse (white/grayish area). This meant most of the particles were initialized outside of reasonable places. While this could have been solved by creating a bounding box around the usable map, we ended up not wanting to dig into the map file.



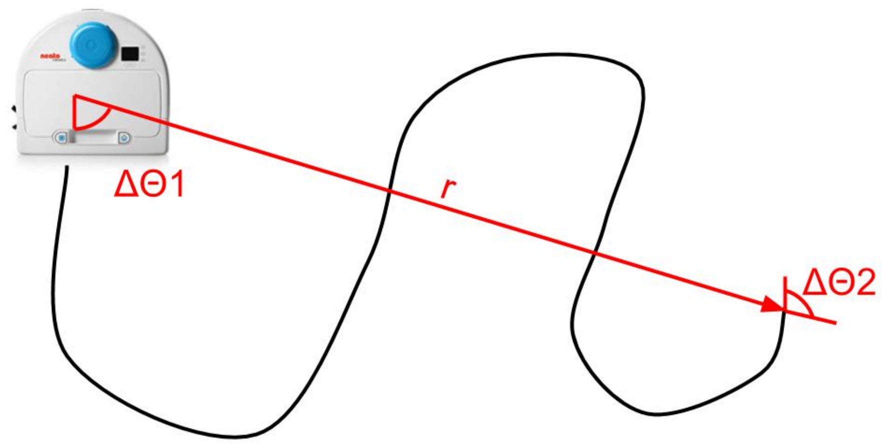
Uniform particle initialization

What we ended going instead ended up initializing the heading based on the nearest wall location and using a large enough spread to still run into a situation we'd be running into and solving the kidnapped robot problem. We did this by calculated the angle at which the closest obstacle from the robot is based on laser scans, then initialized the headings of the particles based on the angle. Because of this the headings start in the correct heading for the given nearest obstacle and allows the particles to collapse into more dense clouds faster. From there they are culled to the robot's actual position (see following sections). While it is more of brute force method, we believe the efficiency makes up for the shear amount of particles which results in a very smooth running regardless of the running rate.


Wall alignment particle initialization

## 3 Position Particles

In tracking the position of particles, a pitfall we encountered in the past is error converting between coordinate frames, since the coordinate frames of the sensor data, odometry, and map are all different. We found that for particle movement we can abstract away the problem of conversion since we only care about differences in particles positions and headings. For example, if we track the position and heading of the neato along a route shown below, we can summarize this movement as a change in heading ($\Delta\theta 0$), distance ($r$), than another change in heading ($\Delta\theta 1$). This translation will remain true for any particle in any coordinate frame that is only translated or rotated from the original, so we may efficiently apply this translation to all of our created particles. We initially implemented noise based on random distrubition to ($\Delta\theta 0$) but in the end because we initialize so many particles and they collapse so quickly we found this didn't account for much in the end.



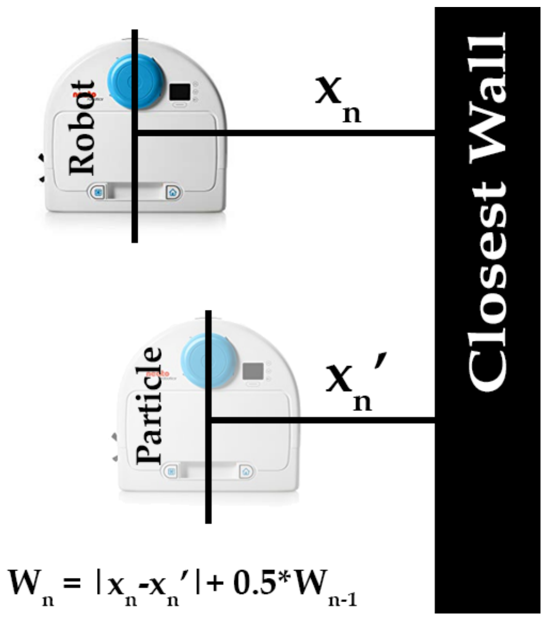
Our method of movement tracking

This representation also provide a convenient method of deciding when to update our particles. Given that our overall weighting scheme (section 3) relies only on distance to the nearest object and not position of the objects, we care only about translational differences. So, we define our update method as only occurring when the difference in position, r, id above a pre-defined threshold. Every time we reach this threshold we reset our origin, move all our particles the appropriate amount, and move on to weighting them as described in next section.

## 4 Weighting Particles

To set the weights of the particles we simply compared the distant to the closest of occupency (Paul's code) of the particle to the closest object based on the robot's laser scan. This means the closer the weight was to 0, the more likely it was the robot's position. Weights also partially carried over into the next iteration as we We chose this because it is a simple computation (O(1)) the occupancy field is pre-computed so we can use many more particles than if we were to check each particle's 'laser scan'.

We also played with giving the heading of the particles an effect on its weight by comparing it the robot's heading change. With the inclusion of the initialization to align with the wall of the map (see initialization section), this didn't really work out though.

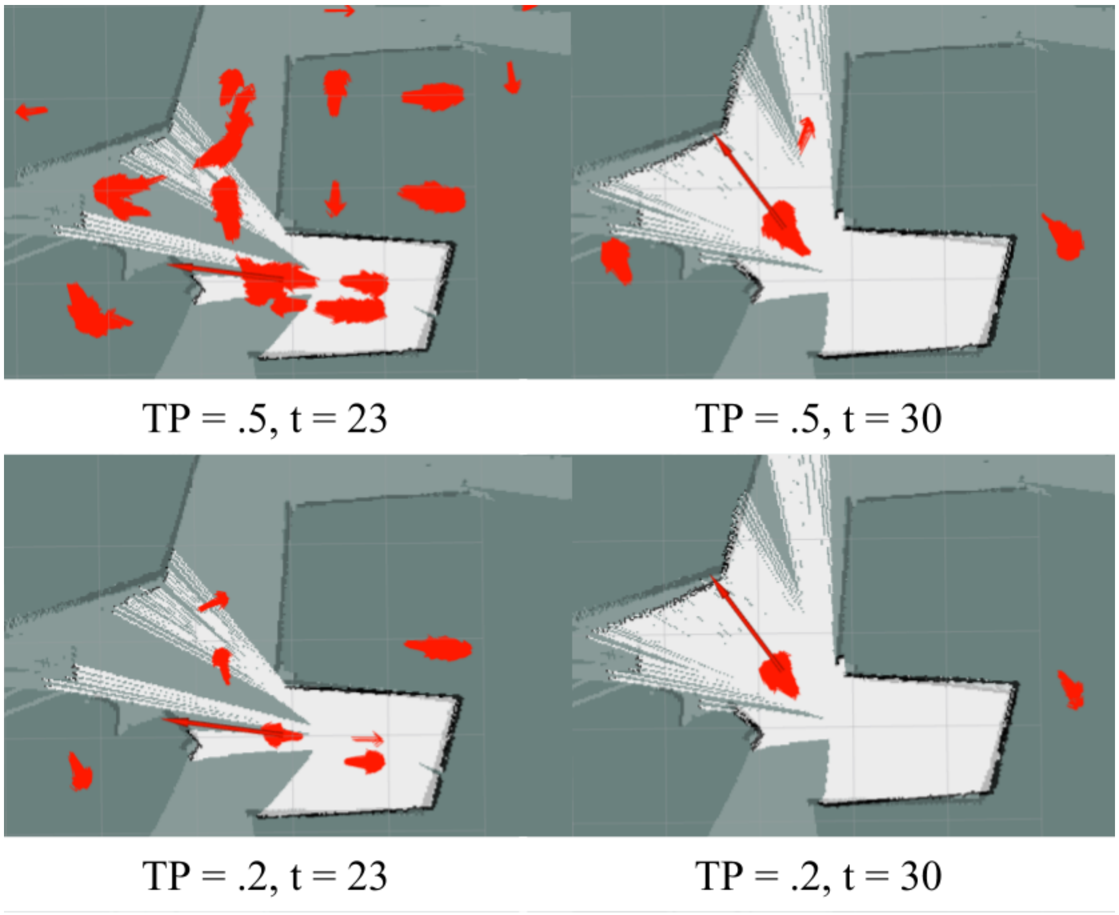


Weighting of the particles

## 5 Culling Particles

In line with our theme of simple, computationally efficient operations per particle, we decided to set a hard rule of always taking the top percentage, TP, of weighted particles. This rate can be adjusted to dictate the rate at which the algorithm culls particles, as shown in the figure below. For example, $TP = .2$ means 80 percent of particles are eliminated at every update and the model will quickly converge on some clusters. For our model, we found $TP = .5$ to be a convenient value as our input data is very small (1 value per particle per time step), so we need more of time to build up confidence in particles before quickly culling them.

**Run Time:** A definite drawback of this design choice is the $Nlog(N)$ run time to sort the array and take the max $K$ values. We could have instead used a max heap to reduce the run time to $Nlog(K)$ (Running K size max heap) or $N + Klog(N)$ (Build max heap and remove K values),but for $K = N/2$ we decided it the overhead was not worth the cost when we already had an efficient function to sort in Numpy.



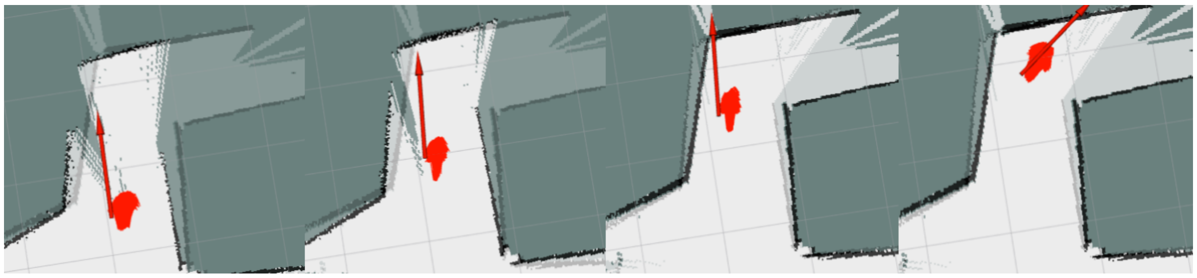
The culling effect of chosing smaller or larger values of $TP$.

## 6 Add Particles

Finally, we need to proliferate the particles left. For this approach we decided to equally distribute new particles around the existing ones. For each particle, we instantiated an appropriate amount of new particles each with the same x position, y position and heading plus some Gaussian noise for each value respectively. We found the standard deviation for translational, $\sigma_T$, and rotational, $\sigma_R$, noise was another parameter we could adjust that approximately translated to the "exploration" that a given particles does while moving in order to account for errors in particle initialization, movement information, and closest object information. So, a large standard deviation would lead a larger bubble of points around the true position that are more likely to encapsulate the true value but less exact about its location.

We tried to instantiate particles with a heading concurrent with the heading of the robot compared to the closest nearest object, but found that due to the noise in the laser scan data this method was prone to failure.

Since we focus on initializing massive amounts of particles, we were not worried about error in particles position, so we settled on $\sigma_T = .01$ Meters and $\sigma_R = \pi/2$ Radians. This doe achieve the effect we want of quickly honing in on the particles position, but also achieves the drawback we feared of noise from odometry and sensor data pushing the particle cloud off course as it moves. Although, even with the noise we observed the cloud gradually self correcting when in non-uniform areas.



Our particle cloud getting off track and righting itself.

## 7 Reflections

### 7.1 Challenges

On all of the sections there were minor difficulties in implementation, most of which were dealt with interfacing with ROS and other code. But, for a large part the difficulties we dealt with the most involved choosing parameter values. For a complicated problem like this, with so many parameters it can difficult to find a combination that makes sense. Consequently, we spent a good deal of time rerunning the simulation with different numbers of particles, noise of movements, noise of headings and methods of instantiating.

### 7.2 Improvements

Improvements we could make to this might be making or uniform distribution or distribution of points in generally better. Distributing over the map we could have tried to create a bounding box around the actually usable part of the map instead of the whole thing. Improving on our current distribution we could have made the spread and density of the initial particles a function of the size of the map.

Another improvement we considered was making how far new particles were added from the remaining particles a function of density. We wanted our algorithm to explore and this would have enabled that once the particles sufficiently collapsed they had reason to spread out to explore other routes but never too far so that the robot's actual location was lost.

### 7.3 Lessons Learned

Rviz, although tedious at times to setup and work with, was very helpful, especially for this kind of task that having repeatability is key, especially when tuning parameter values. Being able to predict what our code was going to do and then having a visual confirmation of it was also extremely reassuring and allowed us to experiment and explore a lot more than we probably would have otherwise.