RSS Lab 5 Report: Localization

RSS Team 9: SDWCH
William Chen, Henry Hu, Christina Jung, Silvia Knappe, Diego Mendieta

April 11, 2020

Contents

1 Introduction

Author: Christina

When designing an autonomous vehicle, one of the key tasks is to determine a robot's orientation and position in a known environment, also known as localization. In Lab 5, our goal was to have our robot successfully localize itself in a known environment, given odometry data, laser scans, and a map of the environment. In order to do so, we built two probabilistic models: a motion model and a sensor model. Then, we integrated the two models together using the particle filter algorithm.

2 Technical approach

2.1 Motion Model

Author: Silvia

The motion model of the robot uses odometry data combined with noise to generate probabilities that estimate the position of the robot. The data from the motion model integrated into the particle filter in order to obtain a final estimate of the robot's pose.

We use a previous pose $\begin{pmatrix} x \\ y \\ \theta \end{pmatrix}$ combined with odometry data that gives us change in position and angle and estimates a next pose $\begin{pmatrix} x^* \\ y^* \\ \theta^* \end{pmatrix}$. Given the previous and next pose we are able to extract motion parameters $\delta_{\text{trans}}, \delta_{\text{rot1}}, \delta_{\text{rot2}}$, as shown in the figure below.

Our algorithm differs slightly from the one shown in *Probabilistic Robotics* in that our δ_{rot2} encompasses the entire rotation, instead of sectioning it out into δ_{rot1} and δ_{rot2}. However, we still use δ_{rot1} to incorporate noise into our model, as shown in the section below. In order to calculate δ_{trans} and δ_{rot2} we used the equations below:

$$\delta_{\text{trans}} = \sqrt{(x^* - x)^2 + (y^* - y)^2}$$

$$\delta_{\text{rot2}} = \theta^* - \theta$$

In our simulated robot environment, all odometry data is perfectly accurate. However, this does not reflect real life, where drift and other phenomena affect

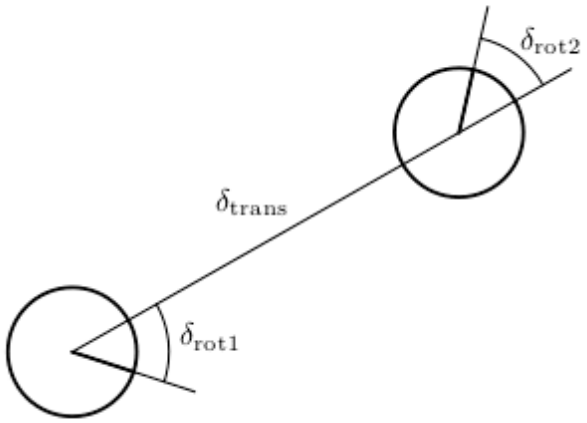

Figure 1: Visualization of motion parameters from *Probabilistic Robotics*

odometry data and make it inaccurate. In order to simulate real life odometry, we incorporated noise into our motion model in the form of scaled normal distributions added to each motion parameter. Our scaling factors after tuning were $\alpha_1 = .00075, \alpha_2 = \pi/32, \alpha_3 = \pi/32$. We applied the scaling factors and normal distributions to the motion parameters as shown, where $\mathcal{N}$ represents a normal distribution:

$$\delta_{\text{trans}} += \alpha_1 \mathcal{N}$$
$$\delta_{\text{rot1}} = \alpha_2 \mathcal{N}$$
$$\delta_{\text{rot2}} += \alpha_3 \mathcal{N}$$

We can now obtain a probability distribution over potential new poses given our previous pose and the motion parameters with noise added:

$$\begin{pmatrix} x^* \\ y^* \\ \theta^* \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} \delta_{\text{trans}} \cos(\theta + \delta_{\text{rot1}}) \\ \delta_{\text{trans}} \sin(\theta + \delta_{\text{rot1}}) \\ \delta_{\text{rot2}} \end{pmatrix}$$

This probability distribution is then passed into the particle filter.

2.2 Sensor Model

Author: Diego

The robot uses a probability model to process sensory information. The sensor model defines the probability of recording a sensor reading z_t from a hypothesis position x_t in a known static map m at time t. To determine this probability $p\left(z_t|x_t,m\right)$, we consider four different cases to be modeled.

The first case models the probability of the sensor detecting the object we expect it to. We model this probability with a Gaussian distribution centered around the ground truth distance z_t^*:

$$p_{hit}\left(z_t|x_t,m\right)=\begin{cases}\eta\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(z_t-z_t^*)^2}{2\sigma^2}\right) & \text{if} \quad 0\leq z_t\leq z_{\max}\\ 0 & \text{otherwise}\end{cases} \tag{1}$$

The second case considers the probability of getting a sensor reading of something than is closer than expected due unknown obstacles. We represent this as a linear function with a downwards slope as the distance from the robot increases:

$$p_{\text{short}}\left(z_t|x_t,m\right)=\frac{2}{z_t^*}\begin{cases}1-\frac{z_t}{z_t^*} & \text{if} \quad 0\leq z_t\leq z_t^*\\ 0 & \text{otherwise}\end{cases} \tag{2}$$

The third case is the probability of missing a measurement due to the beam not bouncing back to the sensor. This is modeled by a spike in the probability at the maximum distance value:

$$p_{\max}\left(z_t|x_t,m\right)=\begin{cases}1 & \text{if} \quad z_t=z_{\max}\\ 0 & \text{otherwise}\end{cases} \tag{3}$$

The final case accounts for random measurements. This is modeled by a uniform distribution with a small value ($\frac{1}{z_{\max}}$):

$$p_{\text{rand}}\left(z_t|x_t,m\right)=\begin{cases}\frac{1}{z_{\max}} & \text{if } 0\leq z_t<z_{\max}\\ 0 & \text{otherwise}\end{cases} \tag{4}$$

These four distributions are combined by a weighted average defined by a parameter α which is different for each case to obtain the net probability distribution $p\left(z_t|x_t,m\right)$ plotted in figure 2:

$$\begin{aligned}p\left(z_t|x_t,m\right)=\alpha_{hit}\cdot p_{hit}\left(z_t|x_t,m\right)+\alpha_{short}\cdot p_{short}\left(z_t|x_t,m\right)\\ +\alpha_{\max}\cdot p_{\max}\left(z_t|x_t,m\right)+\alpha_{rand}\cdot p_{rand}\left(z_t|x_t,m\right)\end{aligned} \tag{5}$$

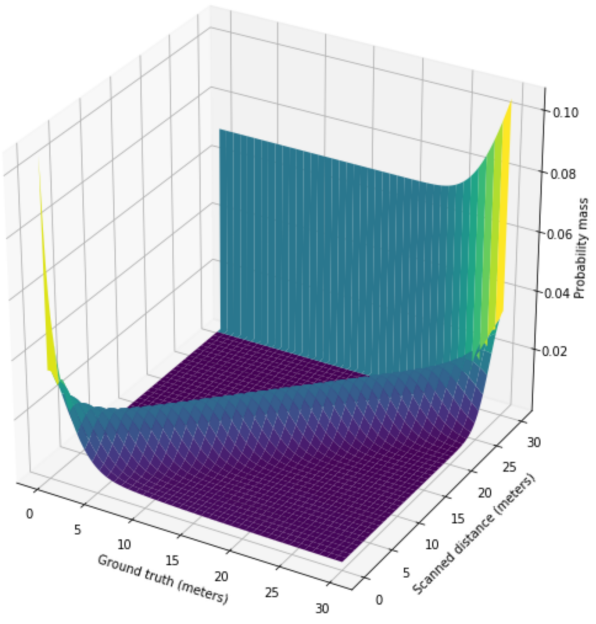

Figure 2: Probability distribution $p(z_t|x_t, m)$ according to ground truth distance and scanned distance

The sensor model requires a large amount of non-trivial operations that would prevent the computations from being performed properly in real-time; therefore, the probability distributions were calculated previously and stored on a lookup table indexed by z_t and z_t^* to preserve computation time. These values were then passed onto the particle filter for further computation.

2.3 Particle Filter

Author: William

Particle filter localization makes use of the previously-discussed motion and sensor models to estimate a robot's position and attitude. As its name suggests, the algorithm creates many particles, each one representing a guess of the robot's pose. The motion model is used to predict where the particles would move based on odometry data received from the robot. Then, the sensor model calculates updated probabilities for how likely a given particle's pose is the true pose of the robot, based off of LIDAR range measurements. Finally, a new set of particles is redrawn from the old ones, based on these computed probabilities. This overarching structure is shown in Figure 3. Regions of the map with large

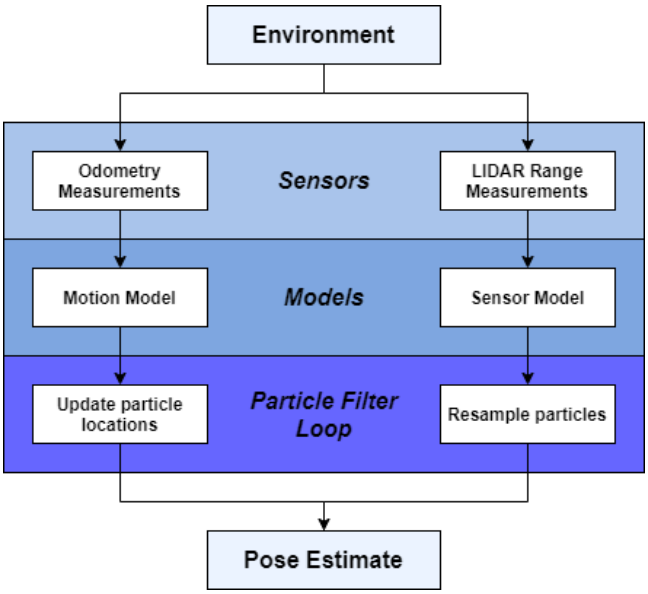

Figure 3: High-level architecture of the particle filter. The robot receives data from sensing and moving about its environment. The two models process this data, which then is used in predicting particle movement and resampling. These final two steps go towards calculating an estimate of the robot's pose.

congregations of particles with similar orientations indicate a high probability of the robot being in a nearby pose. When the aforementioned computational modules are repeatedly executed, this gives rise to the particle filter localization algorithm.

The method of particle initialization is vital to the performance of the particle filter, as without a method of determining the robot's initial pose, localization algorithms face the "kidnapped robot problem." If the particles' initialized poses do not largely agree with the robot's (due to the robot having been "kidnapped" and moved to an unknown new location), probabilistic localization methods such as the particle filter may not converge to a conclusive estimated pose. One potential solution is to initialize the particles all around the considered environment, which would represent a lack of a priori information on the robot's pose, so any given pose is reasonable and should be accounted for. However, doing this requires much more computation (as, to densely cover the environment, one would need more particles) while also generally taking more time to converge (as there could be multiple particle clusters that all represent likely robot poses). However, as the simulation we used effectively allowed one to choose the exact starting pose via Rviz's /initialpose message, we initialized all the particles in that pose to avoid the kidnapped robot problem altogether.

Once the particles are initialized, two additional ROS subscribers process odometry and LIDAR data for the two models. In doing so, particle movements and resamplings can occur independently, each time a corresponding message from either topic was published. Thus, the algorithm is never stuck "waiting" for an appropriate message, allowing it to run faster than if it instead required both an odometry message and a LIDAR scan message at each prediction-update-resample cycle. As the simulation (and most real robot platforms) publishes odometry at a faster rate than exteroceptive sensor data, this algorithmic architecture imparts the benefits of near-real-time localization estimations that the aforementioned stalling would prevent.

The odometry subscriber is responsible for converting angular and linear velocity information to differential movement information and then feeding it to the motion model. The /Odometry message in ROS provides said information as two 3D vectors. Given the robot's position as $\mathbf{r} = [x,\ y,\ 0]^T$ and orientation about the z-axis θ, these vectors have the following forms:

$$\mathbf{v}_r = \begin{bmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \\ 0 \end{bmatrix} \tag{6}$$

$$\omega_r = \begin{bmatrix} 0 \\ 0 \\ \frac{d\theta}{dt} \end{bmatrix} \tag{7}$$

Note that all quantities are in the robot's frame of reference, where the x-axis is straight forward, the y-axis is to the left, and positive angle θ represents a rotation counterclockwise when viewed from above. Moreover, since the robot in simulation uses Ackermann drive, it cannot laterally strafe, so $\frac{dy}{dt}$ will also be zero. To convert velocities to differential changes in their corresponding quantities per time-step, the following equations are used:

$$\Delta x = \frac{dx}{dt} * \Delta t \tag{8}$$

$$\Delta \theta = \frac{d\theta}{dt} * \Delta t \tag{9}$$

where Δt is the change in time between consecutive time-steps. /Odometry messages contain precise time-stamp data, so the subscriber can compute the value of Δt by subtracting the preceding time from the current one, each time it receives a message. This gives the motion model differential motions relative to the prior frame of the robot. That is, if the robot were at pose p_{t-1} at time-step $t-1$, in that pose's reference frame, it moves to pose $p_t = [\Delta x,\ 0, \Delta\theta]^T$ one time-step later. Using this data, the motion model computes poses for each of the particles if they had undergone the same measured motion as the robot, albeit with added noise, as discussed above.

The noise injected into the motion model compensates for deviations caused by proprioceptive sensor noise. When the robot moves or turns, the rate of

motion captured by odometry is generally inaccurate. Therefore, the ensemble of particles will drift away from the true path of the robot. To make up for this, additional randomness in the motion model causes the particles to spread out, providing a wider variance in position and orientation. This added pose diversity allows particles to probabilistically come close to the robot's ground truth pose, even if, on average, the particles move according to the odometry data. The sensor model is then able to determine which particles are likely to be close to the robot's true pose.

The sensor model gives each particle pose estimate a probability of being close to the true pose of the robot using LIDAR scan data. Using the methods mentioned in prior sections, it computes this likelihood by essentially comparing the robot's LIDAR scan with what the robot *would* sense if its pose were the same as that of each particle's. Then, the algorithm performs resampling, drawing N new particles from the set of old ones, with replacement (where N is also the number of old particles). The probability of each particle being drawn is the calculated likelihood from the sensor model. Thus, most of the resampled particles will be ones that would have measured similar scans to the actual robot's, resulting in both a correction for odometric drift and, ideally, a tighter and more accurate grouping of pose estimates for the robot.

Given the resampled particle poses, the estimated pose is calculated by taking the mean of each of the particles' pose parameters (with outliers more than a certain number of standard deviations away from the mean being excluded). The outlier rejection metric was implemented to prevent multi-modal particle pose ensembles from causing the average to be an unreasonable quantity. Say the ith particle has pose $[x_i,\ y_i,\ \theta_i]$. After removing outlier values, the estimated position is:

$$x_{\text{avg}} = \frac{1}{m}\sum_{i=1}^{m} x_i;\ \ y_{\text{avg}} = \frac{1}{n}\sum_{i=1}^{n} y_i; \tag{10}$$

assuming, after outlier paring, there are m and n x and y values remaining, respectively. Furthermore, to calculate the average attitude, the circular quantity average formula was used. If there are l non-outlier values, the average would be:

$$\theta_{\text{avg}} = \text{atan2}\left(\frac{1}{l}\sum_{i=1}^{l}\sin(\theta_i),\ \ \frac{1}{l}\sum_{i=1}^{l}\cos(\theta_i)\right) \tag{11}$$

The estimated pose is thus as follows:

$$\mathbf{p}_{\text{est}} = \begin{bmatrix} x_{\text{avg}} \\ y_{\text{avg}} \\ \theta_{\text{avg}} \end{bmatrix} \tag{12}$$

and is calculated after every update and resample, providing updated estimates after each LIDAR scan.

3 Experimental evaluation

Author: Henry Hu

We evaluated our localization algorithm's simulated performance using code instrumentation related to two major metrics: convergence of particles and absolute accuracy of the pose estimate. We took measurements during the first second of a simulated exploration of a hallway including rugged walls, on a map of building 31.

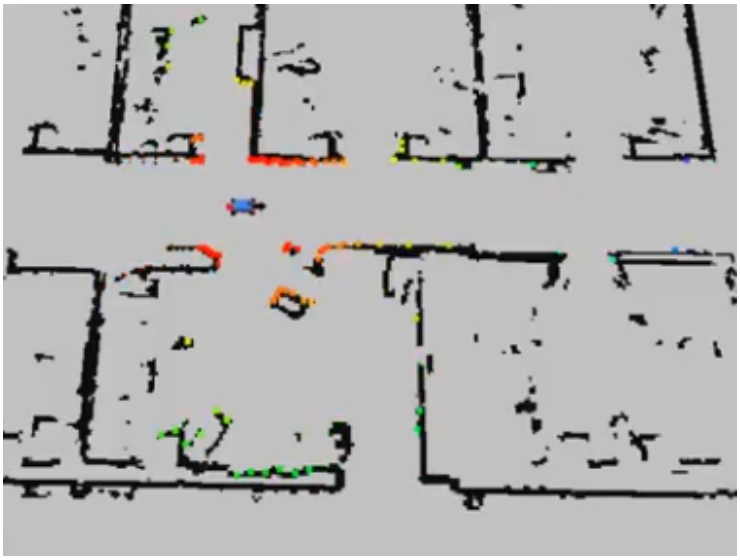

Figure 4: Evaluation environment, a hallway in building 31 including rugged walls

3.1 Particle Convergence

Particle convergence is a desirable attribute of a successful particle model because it ensures reliability of output. If particles represent the range of what the robot considers to be probable localizations, then a convergent set represents a growing precision of estimates over time.

To determine particle spread over time, we recorded the maximum pairwise distance (the range) between two particles at every pose estimate during a sample run. To measure distance, we use the Euclidean distance, dealing with position only. There is no reason to believe that our algorithm's angular convergence would be any worse than positional convergence, but the positional metric was more readily available.

In simulation, our algorithm shows the ability to converge to within 10 px (1.5 m) within 0.8 seconds. The precision immediately following sensor input converges to within 1 px (.15 m) within 0.8 seconds. The data are plotted in figure 5.

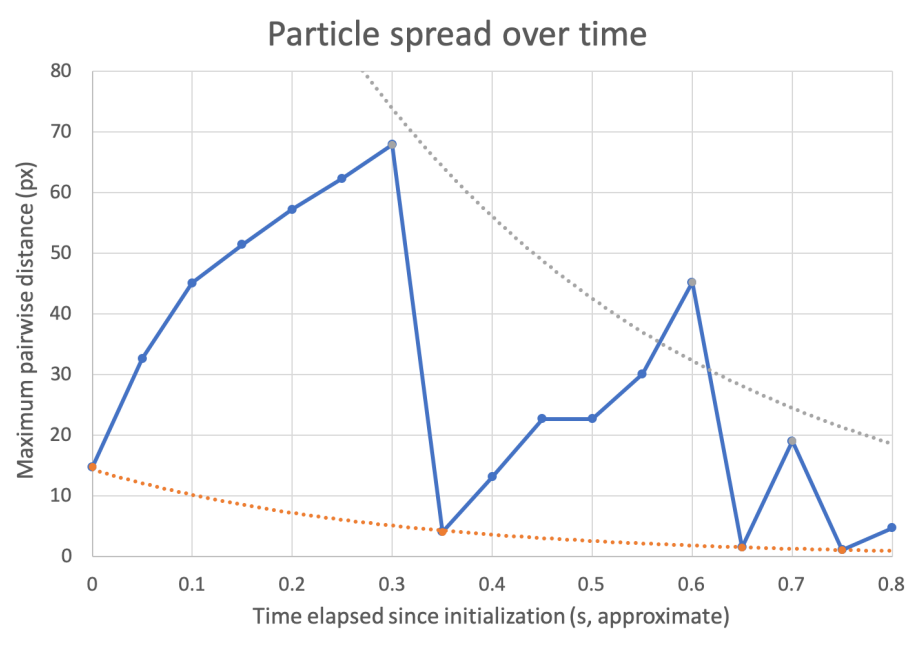

Figure 5: Visualization of particle convergence

Figure 5 shows that the algorithm does not fall into the failure mode of converging too quickly, eliminating possible options prematurely. Rather, at every sensory input, the spread of particles jumps downward, signifying a major rise in confidence, followed by a slow increase in particle spread according to the motion model.

As the gray and orange lines indicate, the high and low bounds for particle spread both become smaller over time. This trend is evidence that our particle model's estimates are correctly increasing in precision as the robot explores its environment.

3.2 Absolute Error

We were able to determine absolute error of our algorithm's pose estimates by comparing output to the true pose of the robot in simulation. The ultimate evaluation of our algorithm is this absolute error, which tells how accurate our localization is compared to the actual position of the robot.

The calculation that we performed to determine absolute error was the Euclidean distance between the position in our algorithm's output and the true position of the robot, given by the simulation. The positional metric was more readily available than the corresponding angular metric. Due to the time-dependence of the pose on both angle and position, as long as we monitor our positional accuracy long enough, it should be a valid proxy for the angular accuracy of the estimate as well.

In simulation, our algorithm's pose estimates settle to an accuracy of under 10 px (1.5 m) after two sensor measurements. The error converges to under 1 px (0.15 m) immediately after sensor measurements. The data are shown in figure 6.

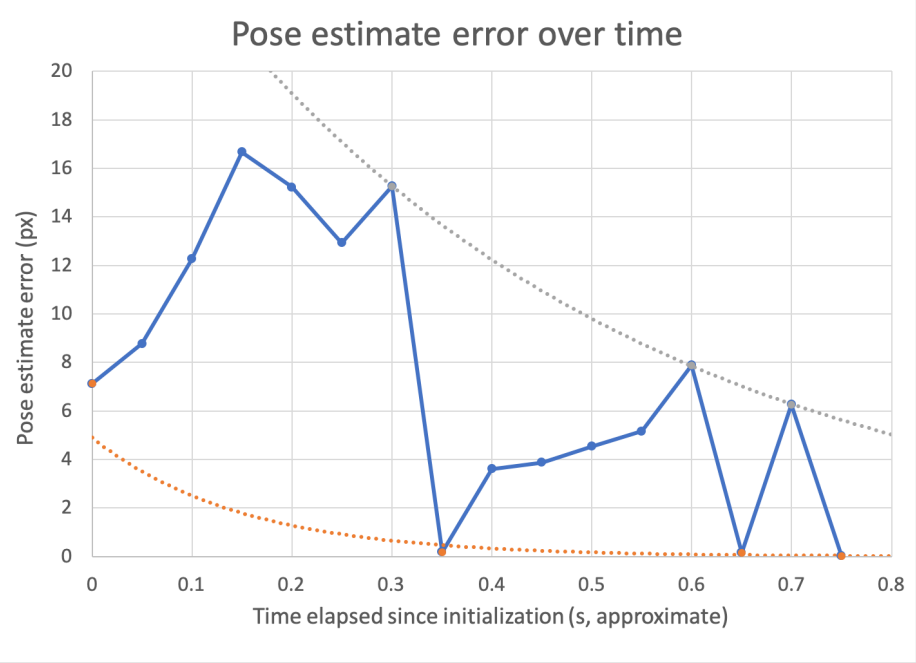

Figure 6: Visualization of absolute error

Figure 6 shows, similarly to the particle spread, that our algorithm's error decreases at every sensory input and increases during motion. As the trendlines indicate, our algorithm's estimates become more correct over time.

Additionally, our results are corroborated by the autograder results, which stated that our algorithm has an accuracy of less than .2 m from the ground truth pose, for a more complex path.

4 Conclusion

Author: Christina

In conclusion, we were able to demonstrate that our robot successfully localizes itself in a known environment. The motion model estimates the position of the robot, and the sensor model calculates the updated probabilities for how likely a given particle's pose is the true pose of the robot. The motion model and the sensor model are put together to construct the particle filter. Finally, we evaluated our localization algorithm in simulation using data on particle convergence and absolute error, which demonstrated the success of our algorithm.

5 Lessons Learned

William:

- Improved ability to create efficient and scalable vectorized functions, as running the filter in real time required this.
- Learned how to tune noise parameters for particle filters through simulated empirical data gathering.
- Learned that autograders for probabilisitic algorithms make me sad.

Henry:

- I enjoyed learning about the particle filter algorithm, similar to a probabilistic algorithm that we learned in 6.01
- Learned how to work and communicate remotely with the team.
- Learned (and still learning) how to focus in a lonely work environment

Christina:

- Learned (and still learning) how to adjust to online classes and to work remotely with the time difference
- Learned more about how motion and sensor models work

Silvia:

- Learned (and am still learning) how to adjust to the new virtual classes
- Learned how to implement the MCL algorithm
- Continued working on removing filler words from presentation.

Diego:

- Learned how to implement a sensor model and reduce runtime by pre-computiong certain variables
- Learned to work remotely with teammates