Introduction

Localization is the problem of determining the unknown location of the robot given a map of the environment. In this particular context, we assume a non-dynamic environment and represent the robot as a point/particle in simulation. In the beginning, we have x particles representing x possible positions of the robot. The process by which we prune these particles can be divided into 3 steps.

Firstly, a **motion model** determines how these particles move according to the odometry (velocity/steering commands). More specifically, the motion model takes the particle locations at time *t* and the odometry at that timestep, and returns the particle locations at time *t+1*.

Then, a **sensor model** determines which particles or estimates should stay. More specifically, the sensor model takes the particle positions at *t+1* (i.e the output of the motion model) and lidar data taken from the perspective of each particle, and determines the likelihood/probability of each particle respective to the map.

Finally, a **particle filter** combines both of these models into a single program and resamples the output of the sensor model to determine which particles move on to the next timestep. The idea is that if we take the more likely particles (according to the probability distribution in the sensor model) at every timestep, then the particles will eventually converge to the true location of the robot. This process is called Monte Carlo localization and the details are as follows.

1. Motion Model (Edward Rivera)

$$P_t = \begin{pmatrix} R & p \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix}$$

Figure 1 Pose Matrix Form (Middle), Pose Vector Form (right)

$$R = \begin{pmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{pmatrix}$$

Figure 2 Rotation Matrix

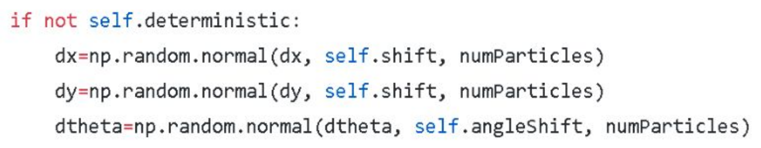
Locations in this lab were defined as poses. Poses in matrix form (Figure 1) include a position *p* component (*p* is a column vector [x,y]) and rotation *R* component (Figure 2). The decomposition of the pose in matrix form would yield three free parameters (x, y, θ), thus yielding a pose in vector notation (Figure 1). The equation in Figure 3 describes transforming one pose into another pose with a transformation pose *T*. The reason for *T* being on the right of P_t (as matrix multiplication is not commutative) is that P_t is from the time *t* frame to the

$$P_{t+1} = P_t \cdot T$$

Figure 3 Pose Transformation Equation

world reference frame and T is from the time *t+1* frame to the time *t* reference frame, hence this is the order to achieve a pose from the *t+1* frame to the world reference frame (analogous to matching dimensions in matrix multiplication). In the context of this problem, P_t are the particle poses at time *t*, P_{t+1} are the particle poses at time *t+1*, and *T* is the odometry for the particles (a single odometry pose in vector form is denoted as [dx, dy, dθ]).

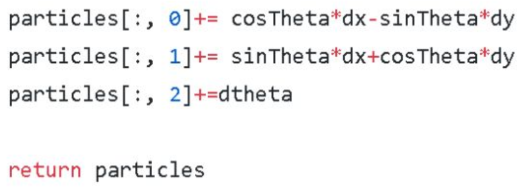
However, this relation is deterministic. If the odometry is currently the same for each particle, then each particle will end up in the same place. To allow the particles to spread out and represent the uncertainty of the robot pose, noise is inserted into the odometry. The code in Figure 4 taken directly from our implementation demonstrates the generation of gaussian noise for every component in the odometry for each particle. The noise for each odometry component is centered around the current odometry value and a standard deviation or shift was empirically verified. In other words, new odometry is generated from the ground truth odometry for each particle as noise taken from a normal distribution is inserted into each odometry component.



```
if not self.deterministic:
    dx=np.random.normal(dx, self.shift, numParticles)
    dy=np.random.normal(dy, self.shift, numParticles)
    dtheta=np.random.normal(dtheta, self.angleShift, numParticles)
```

Figure 4 Code Gaussian Noise

The shift or standard deviation of the noise in the x and y coordinates was initialized to 0.02 meters according to our intuition. The angleShift or standard deviation of the noise in the angle θ was initially set to 0.05 radian (3 degrees), but was later reduced to 0.01 radian (~0.5 degrees) for a faster convergence of the particles. The convergence was visualized by looking at the particle poses in RViz (a video can be found in our presentation or on our YouTube channel).



```
particles[:, 0]+= cosTheta*dx-sinTheta*dy
particles[:, 1]+= sinTheta*dx+cosTheta*dy
particles[:, 2]+=dtheta

return particles
```

Figure 5 Update Particle Poses

$$P_{t+1} = \begin{pmatrix} cos(\theta_t) & -sin(\theta_t) & x_t \\ . & . & . \\ . & . & . \end{pmatrix} \begin{pmatrix} . & . & dx \\ . & . & dy \\ . & . & 1 \end{pmatrix}$$

Figure 6 Matrix Multiplication Simplification - Position

$$P_{t+1} = \begin{pmatrix} cos(\theta_t) & -sin(\theta_t) & x_t \\ . & . & . \\ . & . & . \end{pmatrix} \begin{pmatrix} cos(d\theta) & . & . \\ sin(d\theta) & . & . \\ 0 & . & . \end{pmatrix}$$

Figure 7 Matrix Multiplication Simplification - Rotation

After generating the new odometry for each particle we need to use the transformation equation (Figure 3) to get the particle poses at *t+1*. Figure 5 (code taken directly from our implementation) represents a simplification of the matrix multiplication into 3 efficient addition steps, which allows the use of more particles. The first two addition steps can be explained by looking at the first addition step, which represents $x_{t+1}=x_t$ + some quantity and is verified by performing the matrix multiplication of the two columns in Figure 6. The las t addition step is a trigonometric identity: $\cos(\theta+d\theta)=\cos(\theta)*\cos(d\theta)-\sin(\theta)*\sin(d\theta)$. This equation is also verified by performing the matrix multiplication of the two columns in Figure 7.

After updating the particle poses with the noisy odometry, the particle poses are given to the sensor model.

2. Sensor Model (Kuan Wei)

The sensor model uses particle poses (position and orientation) generated by the motion model data to calculate the likelihood of each particle with a given LIDAR scan data. We then choose particles with the highest probabilities, allowing good hypotheses to have a higher chance of being sampled on the next iteration, and vice versa for bad hypotheses. The sensor model can be divided up into two main components – 1. Precomputing particle probabilities and storing them in a table, 2. Ray casting and evaluating particle probabilities.

2.1 Precomputing Particle Probabilities and Storing in a Table (Kuan Wei)

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

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

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

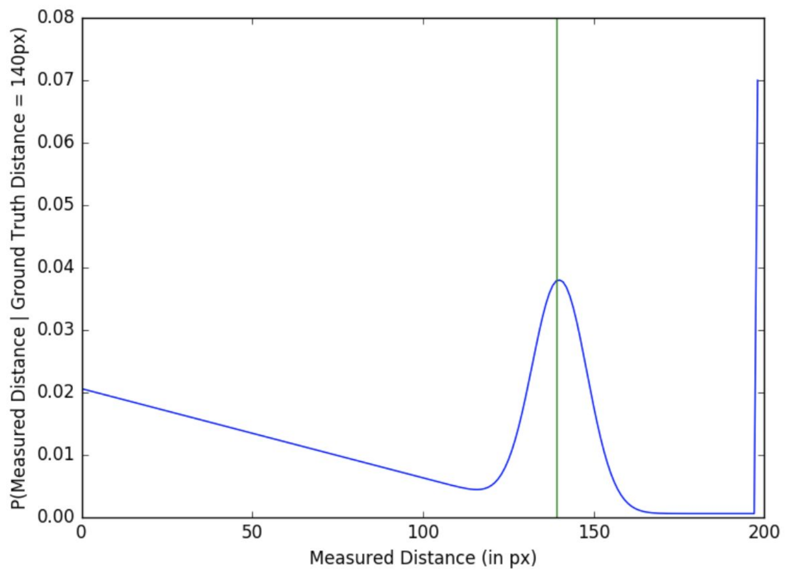
$$p_{rand}(z_t|x_t, m) = \begin{cases} \frac{1}{z_{max}} & \text{if} \quad 0 \leq z_t \leq z_{max} \\ 0 & \text{otherwise} \end{cases} \qquad \text{Equation 4}$$

$$p(z_t|x_t, m) = \alpha_{hit} \cdot p_{hit}(z_t|x_t, m) + \alpha_{short} \cdot p_{short}(z_t|x_t, m) + \alpha_{max} \cdot p_{max}(z_t|x_t, m) + \alpha_{rand} \cdot p_{rand}(z_t|x_t, m) \qquad \text{Equation 5}$$

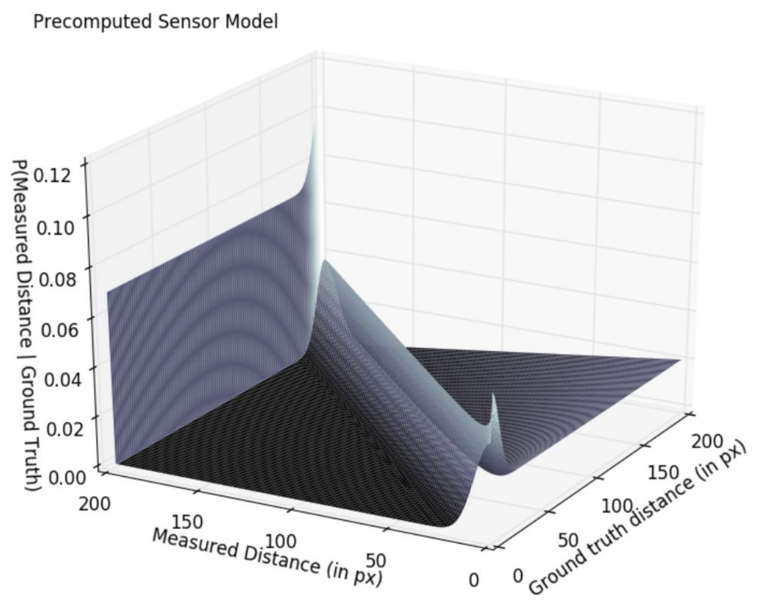
$$\alpha_{hit} + \alpha_{short} + \alpha_{max} + \alpha_{rand} = 1 \qquad \text{Equation 6}$$

The particle probabilities can be calculated with Equations 1-6 above. p_hit represents the probability of detecting a known obstacle in the map. p_short represents the probability of getting a short measurement, which could result from internal lidar reflections or hitting parts of the vehicle itself. p_max represents the probability of a large measurement which is usually from the lidar beam hitting an object with a strange reflective property not captured by the sensor. p_rand represents the probability of a random measurement like an earthquake. The η constant in the Equation 1 ensures that the probability distribution sums to 1.

The lookup table is a predetermined length 201x201, where the rows are the ground truth distances (z_t^*) and columns are the measured distances (z_t). By creating the table, we can quickly retrieve the probability given a z_t and z_t^* avoid duplicate calculations. The table is created by iterating z_t^* and z_t from 0 to the table length and using Equation 5 to compute a weighted sum of p_hit, p_short, p_max, and p_rand to obtain the final probability of each particle, where the weights are $\alpha_{hit} = 0.74$, $\alpha_{short} = 0.07$, $\alpha_{max} = 0.07$ and $\alpha_{rand} = 0.12$. After that, we normalize the particle probabilities such that the probabilities of each list sums up to 1. This resulted in the following graph.



If we plot the surface, we get the following graph.



2.2 Ray Casting and Evaluating Particle Probabilities (Aaron Garza)

The next steps in evaluating the sensor model involve performing ray casting on a set of particle poses and evaluating the associated probability that each particle represents the state of the car. In general, these steps assign probabilities to each particle in the set of poses based on the laser scan that a particle would see in the map and the observed laser scan. An outline of each step in this process is given below.

At a high level, the ray casting in this lab involves producing a ray or range from a given particle position to a point that the lidar on the car would detect. The particle poses passed to the ray casting function are in the form of an N row by 3 column matrix where N represents the number of particles and each column represents the x, y, and theta positions of each particle pose. The structure of this matrix is shown below.

$$\begin{bmatrix} x_1 & y_1 & \Theta_1 \\ x_2 & y_2 & \Theta_2 \\ & \dots & \\ x_N & y_N & \Theta_N \end{bmatrix}$$

Figure 10: Particle Pose matrix used for ray tracing

This produces an N row by m column matrix where m represents the number of lidar measurements in a single scan that the particle would see. These laser scans represent ground truth distances or the distances from the particle pose to points on an artificial laser scan, which are presented as $z_{t.}^{*}$ in this lab. Each scan per particle represents an artificial set of rays that a given particle would see from its perspective. A visualization of this is demonstrated below.

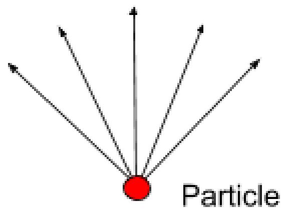


Figure 11: Visualization of ray casting for a particle where the red circle represents a particle, and each arrow represents a ray or ground truth distance

The next step in the sensor model involves extracting values from the pre calculated probability table using the ray casted ground truth distances, z_t^* , and a set of observed laser scan ranges, z_t represented as a vector of length m, where m is the number of lidar measurements in a single scan. In this lab, the generated probability tables are indexed by pixel values while the ground truth distance and observed scans matrices are in meters. In order to index the table, the two sets of data are divided by the map resolution to convert the entries in the matrices to pixels. The probability table is indexed according to each pair of z_t and z_t^* where z_t corresponds to a row and z_t^* corresponds to a column in the matrix. The output of this step is a matrix where each row contains a set of probabilities for each lidar measurement in the scan of each particle. In order to calculate the overall probability for each particle, an average of each row is taken by multiplying the probabilities in that row. This produces an N length vector, where N is the number of particles and each entry corresponds to a probability that that particle represents the true state of the robot in the map. This last step in the sensor model essentially compares the scan that a given particle would see in the map against the actual scan to produce the likelihood that the car is at the given particle.

3. Particle Filter (Kevin Sun)

The final step of our localization process is the resampling of particles, which gives the method its name and arguably differentiates this method from other probabilistic inference methods. Instead of attempting to calculate the probability that the robot is in a specific pose for every possible (x,y, theta), which is often computationally unfeasible, we represent some guesses for the robot’s pose as particles. We only keep track of the most likely guesses, and do not bother calculating probabilities for poses that are not deemed reasonable from previous timesteps. Each particle represents a likely guess of the robot's true pose at the current time. Without any noise, we would only need one particle with continually updated location to represent the probability distribution on poses. However, with noise, it is in our interest to keep several possible particles that are in different locations. Even if some individual particles do not reflect our most accurate estimate of the current pose, keeping more particles can allow us to wait for future information to refine our estimate.

There are multiple ways to refine the particles after obtaining their probabilities calculated from the sensor model. One way is to simply take a fixed fraction of these particles with highest probability. However, to ensure that the number of particles does not drop, we should introduce multiplicity so that the number of particles remains constant at each stage. Because of the noise introduced by the odometry measurements, these particles will disperse, and as long as one particle can remain close to the ground truth pose, there is a good chance that the model eventually finds the correct result. Another way, which is similar in spirit is to sample new particles from the old particle set, with probabilities weighted according to their relative probabilities in the sensor model. This is the method we picked as we observed that when the robot was quite certain about its pose, the poses returned were on average closer to the ground truth pose as a higher fraction of particles were generated closer to a more confident estimate.

We chose to use 400 particles as this was an amount that was computationally feasible to run with our expected update rate of at least 25Hz, and also produced accurate estimates of the robot's pose at all times. With low levels of noise, our robot was able to locate itself to an average error of 0.2 meters in simulation, and qualitatively the path of poses computed by our robot looked identical to the ground truth.

Conclusion (Aaron Makikalli)

Localization is an important aspect of any autonomous vehicle's navigation system. An effective localization algorithm enables the robot to avoid dangerous obstacles – including unpredictable environmental factors – and identify objects relevant to the robot's task. In this lab, we implemented a Monte Carlo Localization algorithm, which uses sensor data and a particle filter to determine the robots most likely pose.

We deconstructed the localization task into three subtasks: a motion model, a sensor model, and a particle filter. The motion model is responsible for applying the most current odometry data to the robot's last set of particles to return a new set of particles representing the robot's motion. The sensor model takes lidar data as an input and evaluates the set of particles to determine the robot's most likely pose based on a precomputed probability distribution model. The sensor model then resamples the particles. Finally, the particle filter combines the results of the motion model and sensor model to publish the transform of the average pose.

Overall, our localization algorithm was very effective at determining the robot's pose, as is evident by the rviz simulations and visualizations shown in this report. However, we currently only determine the average particle pose each time the robot receives new lidar data. One area for further refinement is to find the average particle pose each time the robot receives either new lidar data or new odometry data. Returning the average pose more frequently would increase the resolution with which we estimate the robot's pose, so this should be a more accurate method of localization.

The localization algorithm we developed in this lab will be applicable to both our final lab and the final challenge. Along with the safety controller from Lab 3 and the object detection algorithm from Lab 4, localization is a fundamental ability that keeps the robot safe and enables it to make informed decisions in environments characterized by uncertainty.

Lessons Learned

Technical

- Encourage early drafting of pseudocode in order to attend OH and rubberduck the staff which may prevent big picture misinterpretations of the task (Jack)
- It is crucial to understand what the code is doing and if that is EXACTLY what we want it to do. There could be problems even if the code passes the test cases. (Kuan Wei)
- Push updated code to GitHub more frequently. This enables others to debug more effectively and to understand how the different pieces of code will work together (Aaron Makikalli).
- It is important to start working on code early so that errors in testing that can prevent progress can be fixed early (Aaron Garza)
- Think about the code you write and don't blindly implement something you find without understanding it (Kevin Sun).

Communication

- Regular meetings are essential for establishing communication remotely in a team setting (Jack)
- Similar to what Jack said above, have meetings as early as we can to delegate work. Also be vocal about challenges in lab and or other classes and or life so we can help each other out. (Kuan Wei)
- Now that we are working virtually, it's especially important that we delegate tasks among ourselves early on. This prevents misunderstandings about who is responsible for each task and ensures that we work efficiently. (Aaron Makikalli)
- Communicating each other's schedules and availabilities can make it much easier to help each other and work efficiently, especially in a remote setting (Aaron Garza)
- Make sure members understand what responsibilities and tasks are required of the whole group. (Kevin Sun)