Lab # 5 Report: Localization

Team # 2

Isaac Lau
Lilly Papalia
Mario Peraza
Joshua Rapoport

6.141/16.405: RSS

April 17, 2021

1 Introduction

Author: Joshua Rapoport
We will describe the development and testing of a Robot Operating System (ROS) package that, given perfect map knowledge and periodic updates from position and range sensors, localizes a robot on a map.

The package in question was developed in Python 2 using ROSPy Melodic, and it can be configured to run in a 2D environment (visible in "rviz") or a 3D environment (visible in "TESSE" simulator).

We implement a **particle filter** using models for predicted motion and sensors in a Monte Carlo simulation, with the goal of generating a robust, stable estimation of the ground-truth robot pose.

Exploring localization will help our team tackle higher-level tasks in the future, as well as abstract the rules governing our previous, lower-level logic. The non-deterministic worldview of our solution is widely applicable to other localization methods, such as Simultaneous Localization and Mapping (SLAM), and it itself will serve as lower-level logic for more semantic processes, like motion planning.

2 Technical Approach

Author: Joshua Rapoport

Our particle filter creates $N = 100$ particles represented by 2D poses at some discrete time $k = 0$:

$$x_k^i = \begin{bmatrix} x_k^i \\ x_k^i \\ \theta_k^i \end{bmatrix} + \text{norm}(0, \epsilon) \text{ for all } i \in \{0, 1, 2, \ldots, N\} \text{ and } k \geq 0 \tag{1}$$

where ϵ^2 is the noise variance for position and heading, injected into all particles at initialization.

1. Generates a series of particles and diverge those particles using an odometry update plus some randomness,
2. Rate the likelihood of each particle using a known probability distribution function, and
3. Run a Monte Calro update to down sample and re-converge our particles by likelihood.

Our key technical tasks are segmented accordingly, with MotionModel handling odometry updates from the on-board IMU, SensorModel handling scan updates from a LIDAR sensor, and a Monte Carlo running down-sampling.

These submodules feed directly into a ROS node particle_filter , which is responsible to feeding data to the models, and publishing the resultant pose estimate. In ROS, nodes like particle_filter subscribe to topics like "/odom" and "/scan", and run a callback function whenever sensors (i.e. the IMU and LIDAR) publish a new message to those topics.

Likewise, particle_filter publishes its pose estimate as an Odometry message to a topic "/pf/pose/odom", to be subscribed-to by any other ROS package that requires the particle-filter estimate of the robot frame, " base_link_pf".

2.1 Motion Model

Author Joshua Rapoport

When the IMU publishes new odometry to the "/odom" topic, a callback method is run, executing the motion model as follows:

```
def imu_callback(odom_data):
    odometry = odom_data.twist * dt
    new_particles = motion_model.evaluate(particles, odometry)
```

Motion Model evaluates the series of N particles (from time $k-1$) for every new odometry, propagating those particles to N pose predictions at time k:

$$x_k^i = R_r^W(\theta_r^W) * \Delta x^r + x_{k-1}^i \tag{2}$$

where r is the predicted robot frame of a single particle i, and W is the world frame "/map".

If the provided particles are non-identical, and since odometry is given in the ground-truth robot frame, we generate a different rotation matrix R_r^W for each particle.

To encourage propagation of particles during the motion phase, Motion Model also injects noise into the Odometry, $\text{norm}(0, \epsilon)|_{x,y,\theta}$.

2.2 Sensor Model

Author: Isaac Lau

In the context of autonomous driving, the sensor model serves as a mathematical model that describes the relations between the measurements of our LIDAR sensor and the actual values of the desired parameters. In particular, we use the sensor model to define how likely it is to record a particular sensor reading from a hypothesis position given, *a priori*, a map of our driving environment.

Once we have developed a robust sensor model, we are able to use each particle's likelihood to comb through the motion model's particle guesses and discard the ones that are more likely to be wrong.

On a high level, our sensor model is comprised of the weighted average across 4 distinct probabilistic elements, represented as probability distribution functions:

1. P_{hit} gives the probability of detecting a known obstacle that has already been mapped in our *a priori* map of the driving environment. We represent this probability with a Gaussian PDF defined as:

$$p_{hit}(z_k^i|x_k, m) = \begin{cases} \eta \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z_k^i - d)^2}{2\sigma^2}\right) & \text{if } 0 \le z_k \le z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

2. P_{short} gives the probability of detecting short measurement likely a result of the LIDAR beams hitting some part of the car or unknown objects like a human or traffic cone.

$$p_{short}\left(z_k^i|x_k, m\right) = \frac{2}{d} \begin{cases} 1 - \frac{z_k^i}{d} & \text{if } 0 \le z_k^i < d \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

3. P_{max} gives the probability of a missed measurement meaning a LIDAR beam was sent out but did not return likely as a result of reflective distortions. In ROS, this is represented as a range value at the maximum range of the sensor, z_{max}.

$$p_{max}(z_k^i|x_k, m) = \begin{cases} \frac{1}{\epsilon} & \text{if} \quad z_{max} - \epsilon \leq z_k^i \leq z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

4. P_{rand} gives the probability of a random measurement to account for unforeseen noise.

$$p_{rand}(z_k^i|x_k, m) = \begin{cases} \frac{1}{z_{max}} & \text{if} \quad 0 \leq z_k^i \leq z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

With these 4 elements, we are then able compute the summation of each of the 4 parts weighted by the appropriate alpha constants, which for this lab were given to us. [Note that the equation and general techniques described in the sensor model are derived from the RSS localization worksheet that can be found at: https://github.mit.edu/rss2021-2/localization/blob/master/README.ipynb]

Optimization To reduce runtime and algorithmic complexity, we also precompute a Numpy table across the entire state space of possible sensor model probabilities. Our Numpy lookup table has X and Y axis representing the ground truth distance and the measured distance by the LIDAR, with each element within the array representing the likelihood of correctness. [See Figure 1]

Pseudo-code for the the laser callback function executing the sensor model:

```
def laser_callback(scan_data):
    observation = scan_data.ranges
    probs = sensor_model.evaluate(particles, observation)
```

Author: Lilly Papalia

Particles are composed of numerous range measurements, adding complexity to calculating the probability of a particle. As shown in Equation 7, to calculate the overall probability of a particle, the probability of each range measurement must be multiplied together.

$$p(z_k|x_k, m) = p(z_k^{(1)}, ..., z_k^{(n)}|x_k, m) = \prod_{i=1}^{n} p(z_k^{(i)}|x_k, m) \tag{7}$$

Calculating the probability of a given particle comes with significant computational cost. To reduce the computing power required, a table was precomputed

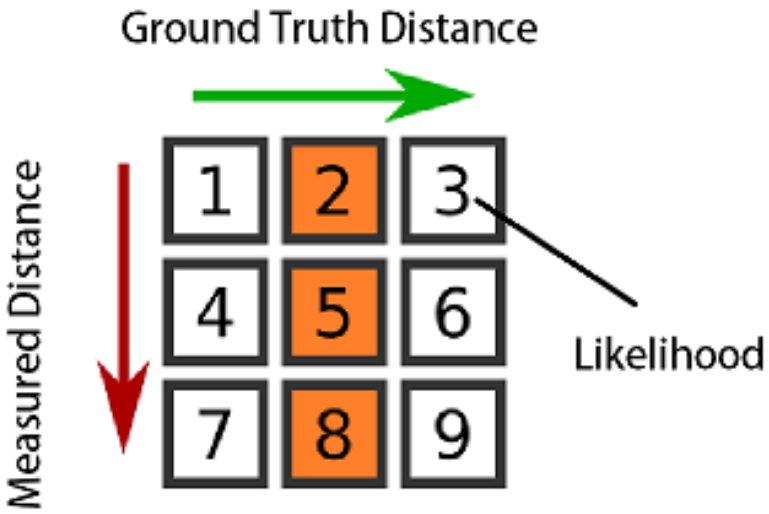

Figure 1: 2D Numpy array of pre-computed sensor model

containing probabilities. By precomputing the table once at the beginning, these values can be accessed quickly and more cost effectively.

Before the precomputed table can be used, there are a few steps to prepare the scan data. First ray tracing is performed on the particles. This essentially returns a stack of LIDAR messages. Next, the LIDAR scans are converted from meters to pixels. This is achieved by dividing by the map resolution and the LIDAR to map scale. The final step is clipping the scans to ensure all values are in the range from 0 to z_{max}. This ensures the probability will be included in the precomputed table, and now the range data is ready to use the table.

The probability of each range measurement is computed by accessing the precomputed table. Next, each range probability for a particle is multiplied together to calculate the overall probability of the particle. The sensor model now returns a vector containing the probability of each particle which is then used in the particle filter.

2.3 Particle Filter

Author: Isaac Lau

The particle filter module starts off with an initialization step in which we take the robot's initial pose and inject noise, producing an array of particles with small, random perturbations. With our starting array of noisy particles, we are then able to conduct the Monte Carlo Localization; in particular, we now combine the outputs of the individual motion and sensor models into our particle filter.

In terms of our particle filter architecture, we call our motion model whenever we receive data from our odometry subscriber, allowing us to update our array of particle positions / poses. In parallel, we call our precomputed sensor model whenever we get data from our laser subscriber and returns an array of computed particle probabilities. To implement this logic in Python, our particle filter utilizes thread safe code so that our system is able to handle disparate update frequencies between the laser subscriber and the odometry subscriber without interrupting each other's processes.

Once our particle filer has these two pieces of information, it then re-samples the motion model's potential particle positions / poses based on each particle's likelihood to be correct. In essence, through the re-sampling process, we are able to take the average of the particle poses weighted by the precomputed probabilities and return that as our estimated pose.

This estimated pose is ultimately the value that we report as the vehicle's location.
Monte Carlo simulation in particle_filter , as shown in the following pseudocode:

```
def monte_carlo():
    particle_downsample = choose(    particles,
                                     size = N,
                                     replace = True,
                                     p = probs)

    pose_estimate = average(particle_sample)
```

2.4 TESSE Integration

Author: Joshua Rapoport
While the particle filter itself is expected to function as long as it recieves the correct ROS messages from topics (e.g. Odometry, LaserScan), there are changes some ROS parameters that are necessary for conversion to the 3D TESSE environment.
For convenience and code clarity, topic parameter names specific to the 2D simulation were kept in a file "params.yaml":

```
map_topic: "/map"
laser_topic: "/scan"
odom_topic: "/odom"
```

However, we wanted to flexibly convert between a 2D and 3D simulation. We nested identical topic parameters for the map, the laser scan, the *ground truth* (GT) odometry, and observed/noisy odometry:

```
viz:
    map:          "/map"
    scan:        "/scan"
    odom_gt:      "/odom"
    odom:         "/odom" # See report Section 2.4.2
tesse:
    map:          "/map"
    scan:         "/tesse/front_lidar/scan"
    odom_gt:      "/tesse/odom"
    odom:         "/tesse/odom/noisy"
```

where ROS parameter "~viz /..." refers to 2D topics, and "~tesse /..." refers to 3D topics. This way, only one variable per file accessing these parameters needs to be changed. Consider a class variable ParticleFilter .ENV, which is set to either "viz" or "tesse". Then, to set up the subscriber in particle_filter .py:

```
import rospy

class ParticleFilter:
    ENV = "tesse"

    def __init__(self):
        # ...
        laser_topic = rospy.get_param("~%s/scan" % self.ENV)
        laser_sub =   rospy.Subscriber(scan_topic, ...)
        # ...

    # ...
```

This will get the ROS parameter at the path "~tesse/scan", which returns "/tesse/ front_lidar /scan".

Other changes are superficial, including the parameter "deterministic", which is false if you want to inject noise into the system. As odometry noise is fundamental to our implementation of particle filter, this would be false in TESSE. Another change is to the environment scale parameter, which is 1.0 for 2D, and 5.0 for 3D.

3 Experimental Evaluation

3.1 Motion Model

Author: Mario Peraza
When adding noise to the motion model, multiple trials were taken to determine which standard deviation produced best results when integrated into the particle filter. Preliminary tuning revealed that noise standard deviations up to $\epsilon < 0.01$ produces negligible difference compared to odometry with no noise added (see Figure 2).

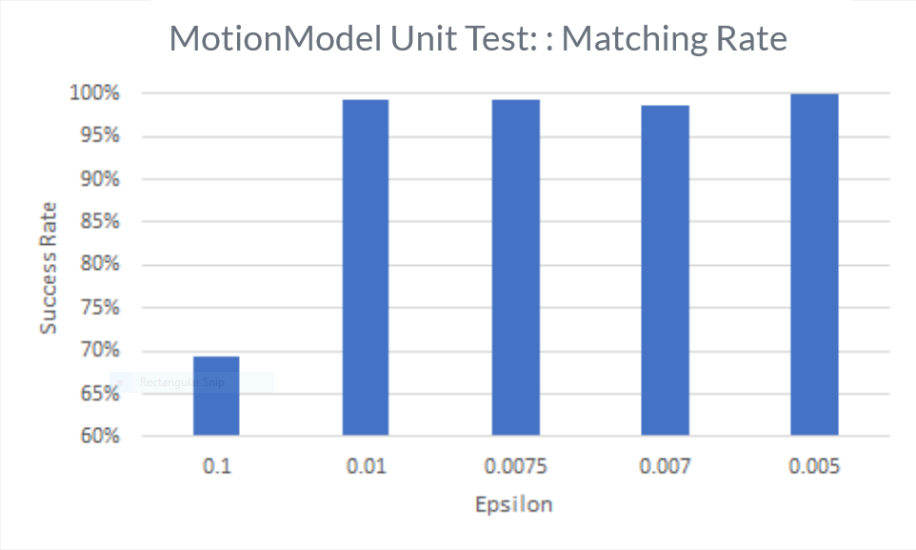

Figure 2: Standard Deviation of Odometry Noise vs. the rate of failure in a unit test designed for Ground Truth Odometry

Other implementations of noise were considered such as having a different standard deviation value for the heading of the particle, however our final implementation used the single value of 0.01. Most evaluation for the motion model was based on the particle filter evaluation and implementation.

3.2 Sensor Model

Author: Lilly Papalia

In order to visualize the precomputed table, a 3D graph was created displaying the ground truth distance, measured distance, and probability. Figure 3 displays our precomputed table, and it is very similar to the graph given in the readme presented for this lab confirming correctness.

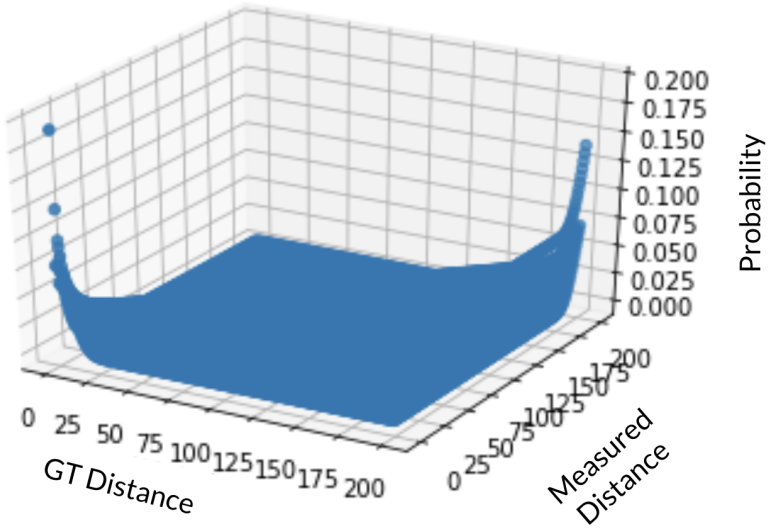

Figure 3: 3D graphic displaying precomputed table of probabilities. Similar shape to table presented in lab instructions, confirming our model is correct.

3.3 Particle Filter

Author: Lilly Papalia

The first method of analysis of the particle filter solution was through the use of a pose array, to display the particles, and a marker, to visualize the estimated pose. In Figure 4, the marker is the green box and the pose array is the red arrows. As shown in the image, the particles are clustered around the robot, and the estimated pose is directly on top of the robot model. This visualization helped gain confidence in the solution before diving into deeper analysis. A video of the visualization moving in rviz is available here.

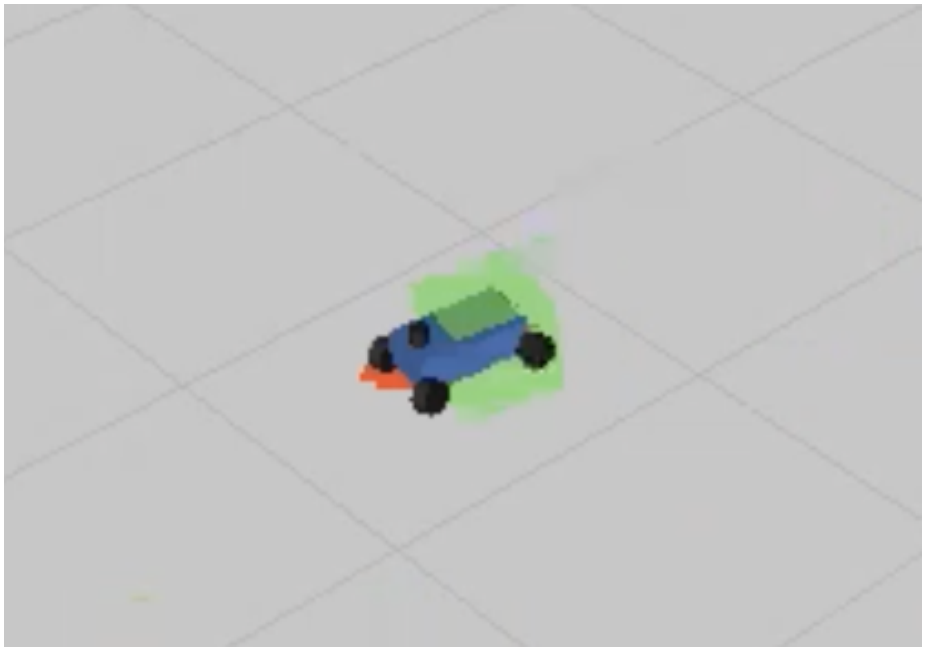

Figure 4: Visualization of Marker (green box) for particle filter pose estimate, and PoseArray (red arrows) for particles.

After visually seeing the particle filter working, deeper analysis was performed to confirm the accuracy of the MCL. First we plotted the estimated pose with the ground truth pose on the same plot using rqt_multiplot. As shown in Figure 5 the pose is following the correct trajectory.

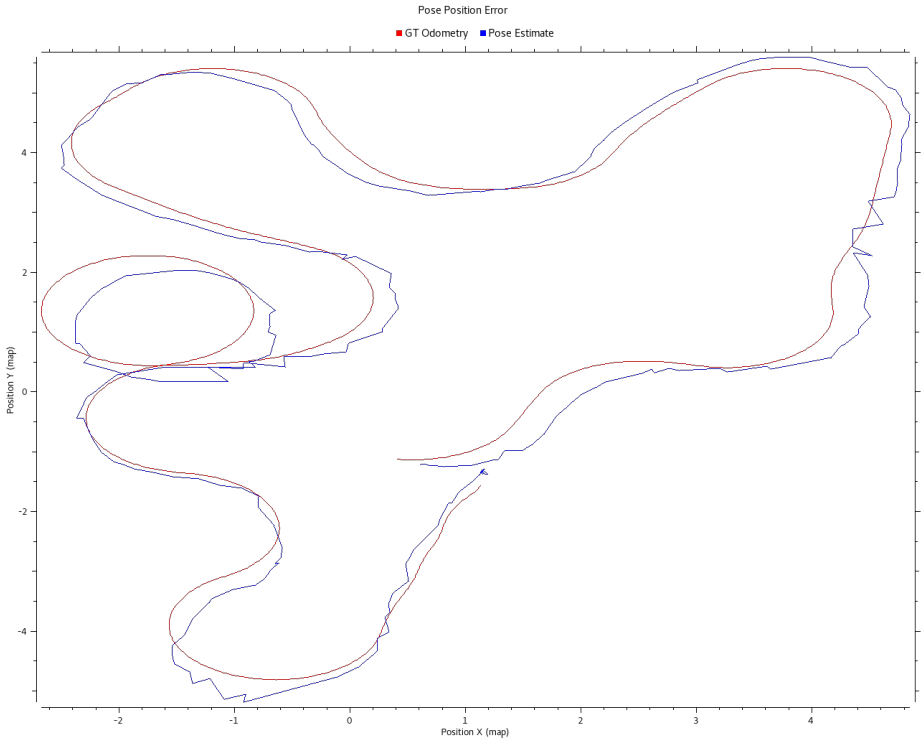

Figure 5: Graph of position of estimated pose and ground truth position in "/map". The results show show the error is small, indicating our estimated pose is accurate.

Another method used to evaluate the solution was through the use of the gradescope autograder. The autograder produced visuals of our estimated pose trajectory compared to the staff solution and ground truth position. These visuals helped uncover errors in our solution, including a missing transform between the particle filter frame and map frame. The visual for the test case with the most noise is displayed in Figure 6.

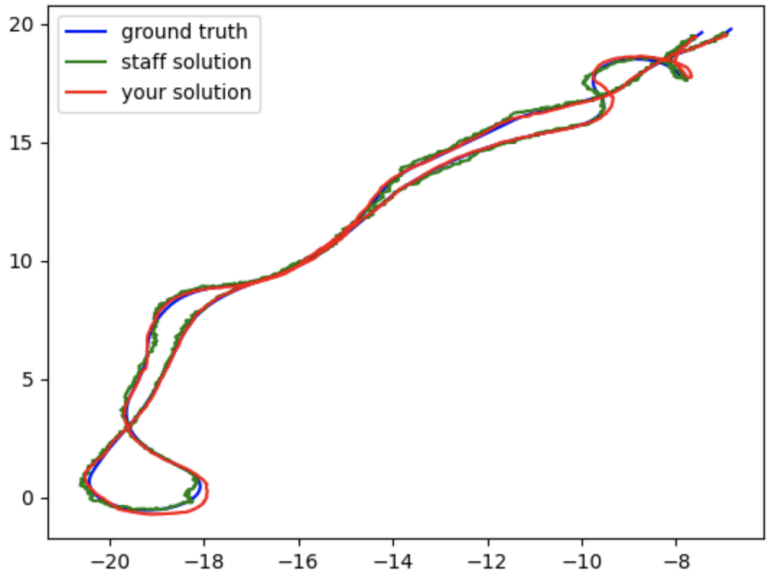

Figure 6: Visualization of our estimated pose trajectory compared to staff solution and ground truth location. This test cast included more noise than the other tests, and the solution was resilient and able to follow the trajectory, even with the extra noise.

For further analysis of the particle filter, the error between the ground truth and the estimated pose versus time was plotted. In order to create this plot, a rosbag file was recorded of the car in 2D simulation to play back and analyze the data. Matlab was then used to extract the data from the rosbag file. The error was calculated as the distance between the estimated pose and ground truth for each timestamp. The error plot is displayed in Figure 7.

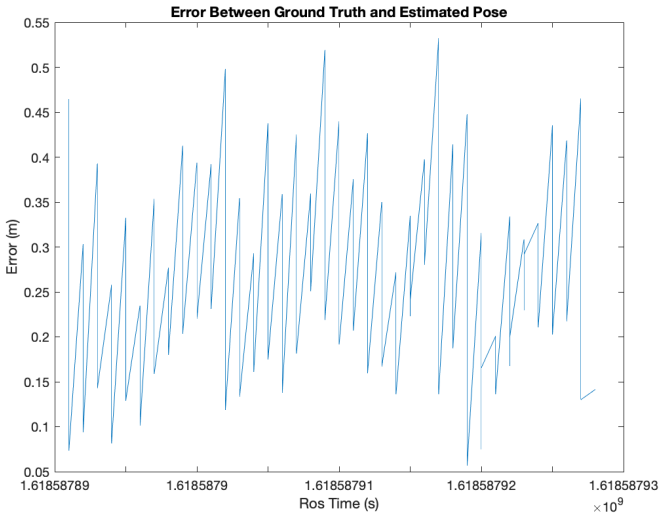

Figure 7: Graph of error over time between the estimated pose and ground truth position. As shown in the plot the error fluctuates, but is very small indicating our solution is successful and locating the robot

3.4 TESSE Implementation

Author: Lilly Papalia
When moving from the 2D simulation to the 3D TESSE environment, many issues with topics being published occured. For example, when checking the incoming LIDAR scan with rostopic hz, the scan was not being published. This lead to an unsuccessful implementation of TESSE. Working on optimizing our solution would help improve the TESSE implementation. Optimization is an important stage that needs to be incorporated in our solution going forward.

Hopefully using the virtual desktop interface (VDI) for lab 6 and the final project will help us better implement TESSE. One video was obtained in TESSE before encountering significant technical issues which can be accessed here.

4 Conclusion

Through the development and implementation of motion and sensor models, the Monte Carlo Localization method was successful when implemented in 2D allowing the simulated robot to orient and locate itself on the map. The 3D implementation of this localization method needs to be optimized to see improved results while working within the simulation, however our evaluation shows the model is accurate and efficient. Robotic localization is a difficult task to solve, Monte Carlo Localization is one of many solutions.

5 Lessons Learned

Presents individually authored self-reflections on technical, communication, and collaboration lessons you have learned in the course of this lab

5.1 Joshua

Overall, I think our team is consistently improving on a weekly basis. I attribute that to successfully targeting areas noted by professors and TA's. Working in pairs on parallelizable tasks, as opposed to individually, was far more effective in my eyes. Having feedback while writing code helped catch mistakes earlier, and I feel it strengthened my bond with my teammates. This was a major boon to collaboration.
I felt that sense of shared energy during implementation of SensorModel and Motion Model, but not during implementation of the particle filter. Live collaboration between 4 members has been difficult, and often left at most two members more engaged than the others.

5.2 Lilly

One major difference between this lab and past labs was working in pairs instead of alone on a section. This helped us keep each other accountable for completing tasks on time, and kept us from troubleshooting alone. This method was very helpful, and I see us continuing to work in pairs for lab 6 and the final project. While there were significant improvements this week, there is still work to be done. Ideally we would achieve tasks earlier, therefore we are able to present a briefing containing our final results.

5.3 Mario

During this lab we decided to work in pairs which proved to be very successful. There was better integration between modules and more communication between pairs. There were slight communication conflicts between pairs which led to slower complete integration that our team will need to work on in the future. It was very helpful having a partner who could review my work as we went though the lab. I believe the partner situation worked well and would

like to do this in the future. We will continue to work on communication and integration between pairs.

5.4 Isaac

While this lab was more challenging than any of the previous ones, I certainly appreciated that we implemented pair programming. This allowed us to check our partner's work and maximize efficiency. In terms of personal growth, I was also more willing to ask for help with this lab both within our team and on Piazza which helped me to debug and find errors / optimizations in my code. I am extremely pleased with the work we have done for this lab and I hope to continue practicing and ingraining the habits that worked for us this time so that, as a group, we can be more efficient in RSS and in our future careers.