Lab 5 Report: Localization

Team 17

Alicia Zang
Ari Grayzel
Robert Cato III
Sebastian Garcia
Shreya Gupta

RSS

April 5, 2022

1 Introduction

Section 1 by Shreya Gupta

In the field of robotics, autonomous navigation is an important problem space and area of research. Examples of autonomous navigation include the Roomba autonomous vacuum cleaner, autonomous underwater vehicles used by the military, and autonomous drones. In order to perform autonomous navigation, a robot needs to be able to know its orientation and position in its environment, which is called localization. Without localization, a robot can not plan its path and becomes reactionary. Not only does the robot need to know its position in its environment at its initial point, but throughout the path it takes.

We solve this general problem by using Monte Carlo Localization, which requires a ground truth map of the environment and provides the pose of the robot within this environment. Due to Monte Carlo Localization requiring a ground truth map, our robot is currently able to perform localization only in known environments and is unable to build a map.

In terms of the specific problem being addressed in this lab, we are trying to implement Monte Carlo Localization in both simulation and on the robot using a given set of ground truth maps that include the basement of the Stata Center. Our goal was to successfully be able to predict the pose of the robot as it moved along a path. Our solution involved the implementation of a motion model and a sensor model, which are combined in the particle filter for the final

Monte Carlo Localization algorithm.

The motion model takes the current pose of the robot and calculates the new pose by using the odometric values from the robot and a rotation matrix. The movement of the physical robot contains imperfections, which is why we also needed to inject noise into the model by adding noise to each odometry vector.

The sensor model calculates the probability that a sensor measurement corresponds to a predicted point on the ground truth map. We first precompute a matrix that combines probability distributions of various cases: the probability of detecting a known obstacle, the probability of a short measurement, the probability of a missed measurement, and the probability of a random measurement. This matrix is then queried using ground truth distances found using ray casting and lidar measurements received from the robot to get the probability that the observed sensor measurement corresponds to a predicted point.

The motion model and the sensor model are combined for the particle filter to estimate the pose of the robot with respect to the world(?) frame. The particle filter gets odometry data, and sends it to the motion model to generate predictions of the pose of the robot. Then the sensor model uses measurement data to estimate the likelihoods of each predicted pose being close to the real pose, and we are given importance factors or "weights" to each particle. Then the predictions are updated by resampling the particles that we currently have to represent a better distribution of the true pose. After updates to the particles from either model are made, the average pose is published as the estimated pose for the robot. This is used in both the simulation and the robot to perform localization.

2 Technical Approach

2.1 Particle Filter And Monte Carlo Localization

Section 2 by Sebastian Garcia

2.1.1 Problem

Before diving into the specifics of how our motion and sensor models work, it is important to understand the big picture, and why we need these models for a good localization algorithm. The particle filter's main usage is to predict and update potential poses that our robot could be in, and result with a single pose that best represents the possible state of the robot in the present.

2.1.2 Setup

In order to use Monte Carlo localization and our particle filter effectively, we first needed to initialize our models, including our noise provider. Using a

few numpy functions that provide distributions of random numbers, we could generate initial noise so that our starting particles are spread out, with equal importance factors, or weights. An importance factor is similar to a weight of a particle in that it measures the likeliness of a particle's pose being the correct overall pose of the robot. The importance factors of all particles should sum to one.

2.1.3 Approach and Implementation

The first step in the particle filter algorithm is to get the current state of the robot, which includes a control and odometry.We also initialize a set of particles to be displaced randomly at first around the robot.

Then, we can use our motion model and our current parameters to obtain a new set of particles based on the current odometry of the car. This is known as the prediction step in our algorithm, which precedes the important update step. If we were to use only the motion model for localization, there would be the problem of our particles deviating from the most correct trajectory due to drift and external noise. For this reason, we must include the sensor model to update the set of particles to better represent the distribution of potential states, and add internal noise in our motion model.

Next, in the update step, we feed the sensor model scan measurements along with our set of particles. In return we get probabilities corresponding to how close each particle is to matching the correct ground truth pose of the robot. In our case, we had our sensor model return probabilities that were not normalized, and so the normalization step was done in the particle filter for the purpose of resampling. By resampling, we mean generating a new set of particles based on the newly attained importance factors, or weights. It could be thought of as drawing random particles from a hat with replacement, and each having a different weight. That way, the most correct particles will appear in the updated set multiple times and the deviations from an average of these is smaller, so we can get a more accurate estimation.

Now, with properly working models, there should be a smooth trajectory generated by our pose estimation. However, just these two models alone will cause an issue known as particle death. In our resampling step, we converge to a smaller amount of unique particles after every time step, and eventually the particles will all be the same if we add the same odometry, and each has high importance factors. We introduced randomly distributed noise around a given mean into our motion model to combat this issue, By doing so, the particles never converge since a proportion of them would be distributed closely but not entirely similar.

The final step in estimating a pose is averaging the set of particle poses and publishing that pose to a topic that receives odometry messages. In our al-

gorithm, we averaged and published each time that the set of particles was updated, whether through our motion or sensor model. Since motion model generally ran faster, that was our main method of estimation, with the sensor model being used to correct drift. Additionally, since the models are able to modify and use the set of particles concurrently, we had to add thread locks to prevent any issues when using them.

2.2 Motion Model

Section 2.2 by Alicia and Robert

2.2.1 Problem

For the motion model, we are given the current state and the odometry of the robot, and use them to find the next state of the robot in the world.

2.2.2 Setup

Our motion model uses two matrices to update states for the particle filter: X_k and Δx. X_k is a matrix of the three state components for each of the predicted particles with respect to the world frame. Δx is a three dimensional vector for the measured odometry with respect to the robot's frame. The odometry vector is duplicated to match X_k's shape. For this model, we assume that the odometry values are independent from each other such that $\Delta\theta$ does not affect Δx or Δy during the k to $k+1$ time interval.

2.2.3 Approach and Implementation

Since the the position of the robot is in the world frame, we first convert the odometry to the world frame. After converting, we then add the odometry to the current state to get the next state.

If we flag the motion model to be non deterministic, Gaussian noise is added to every value in the odometry matrix. The noise on the Cartesian coordinates, Δx and Δy, uses a Gaussian distribution with zero mean and a standard deviation of 0.10. The noise applied to the $\Delta\theta$ coordinate is Gaussian with zero mean and a standard deviation of 10 degrees.

With these definitions and assumptions, we can calculate the state update model for every predicted pose at time $k+1$ as:

$$\begin{aligned} x_{k+1} &= \Delta x \cos(\theta_k) - \Delta y \sin(\theta_k) + x_k \\ y_{k+1} &= \Delta x \sin(\theta_k) + \Delta y \cos(\theta_k) + y_k \\ \theta_{k+1} &= \Delta\theta + \theta_k \end{aligned}$$

In matrix form, this is:

$$\mathrm{x}_{k+1} = \begin{bmatrix} \cos(\theta_k) & -\sin(\theta_k) & 0 \\ \sin(\theta_k) & \cos(\theta_k) & 0 \\ 0 & 0 & 1 \end{bmatrix} \Delta\mathrm{x} + \mathrm{x}_k$$

The pseudo-code below is how the equations were implemented:

```
motion_model evaluate(particles, odom):
    COSINES = cos(THETA_K_PREV)
    SINES = sin(THETA_K_PREV)

    if not DETERMINISTIC:
        new_odom = odom + NOISE_MATRIX

    X_K = X_K_PREV + DX * COSINES - DY * SINES
    Y_K = Y_K_PREV + DX * SINES + DY * COSINES
    THETA_K = THETA_K_PREV + DTHETA
```

2.3 Sensor Model

Section 2.3 by Shreya Gupta

2.3.1 Problem

The sensor model addresses the problem of uncertainty in the location of the robot corresponding to a certain laser scan measurement. The model should take in a ground truth distance d from a hypothesized pose x_k and a measured laser scan distance $z_k^{(i)}$ and output the probability that $z_k^{(i)}$ was measured from x_k. Essentially, we are trying to answer the question: where in the ground truth map was this LIDAR measurement taken from?

2.3.2 Approach and Implementation

The sensor model takes the form of a precomputed matrix with values of d being one "axis", values of $z_k^{(i)}$ being the other "axis", and the probabilities being the values inside the matrix itself. The d and $z_k^{(i)}$ axes start at 0 and end at z_max and use a set number of steps. This precomputation is done for performance reasons, as the computations involved are time consuming and performing these operations at each time step would slow down the program significantly. Also for performance reasons, no for loops were used in the code to build the matrix, only numpy operations.

The sensor model is ultimately a linear combination of four different probability distributions. Each of these probability distributions represents a different case relating to the measurement. Case (1) is the probability of detecting a known obstacle in the map. Case (2) is the probability of a short measurement. A short measurement could occur due to internal lidar reflections such as the laser

bouncing off a substance on the surface of the lidar or unknown objects such as people walking in front of the lidar. Case (3) is the portability of a very large (missed) measurement. This occurs when the lidar beam hits an object with strange reflective properties, resulting in them never bouncing back to the sensor. Case (4) is the probability of a completely random measurement. (case wording sourced from spec sheet)

The probability distributions for each of these cases are as follows:

$$case1 : p_{hit}(z_k^{(i)}|x_k, m) = \begin{cases} \eta \frac{1}{\sqrt{2\pi\sigma^2}} exp(\frac{-(z_k^{(i)}-d)^2}{2\sigma^2}) & 0 \leq z_k \leq z_{max} \\ 0 & otherwise \end{cases}$$

$$case2 : p_{short}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{2}{d} - \frac{2z_k^{(i)}}{d^2} & 0 \leq z_k \leq d \,\&\, d \neq 0 \\ 0 & otherwise \end{cases}$$

$$case3 : p_{max}(z_k^{(i)}|x_k, m) = \begin{cases} 1 & z_k^{(i)} = z_{max} \\ 0 & otherwise \end{cases}$$

$$case4 : p_{rand}(z_k^{(i)}|x_k, m) = \begin{cases} \frac{1}{z_{max}} & 0 \leq z_k \leq z_{max} \\ 0 & otherwise \end{cases}$$

These probability distributions are visually shown in the figures below:

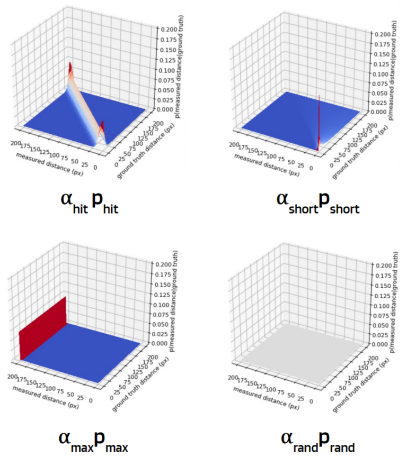


Figure 1: *Visual representation of the probability distributions of the 4 cases*

These probability distributions are combined using the following equation:

$$\begin{aligned} p(z_k^{(i)}|x_k, m) =& \alpha_{hit} p_{hit}(z_k^{(i)}|x_k, m) + \alpha_{short} p_{short}(z_k^{(i)}|x_k, m) \\ &+ \alpha_{max} p_{max}(z_k^{(i)}|x_k, m) + \alpha_{rand} p_{rand}(z_k^{(i)}|x_k, m) \end{aligned} \tag{1}$$

Where, $\alpha_{hit} + \alpha_{short} + \alpha_{max} + \alpha_{rand} = 1$. The visual representation of the final probability distribution used in the sensor model is shown below.

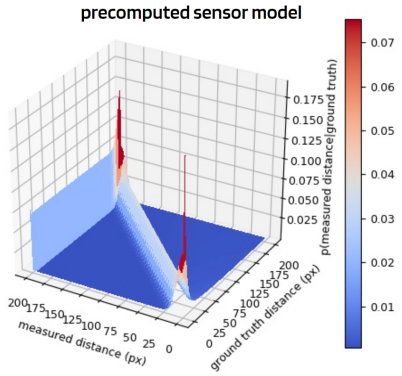


Figure 2: *Probability distribution of the pre-computed sensor model*

The alpha values used were $\alpha_{hit} = 0.74$, $\alpha_{short} = 0.07$, $\alpha_{max} = 0.07$, $\alpha_{rand} = 0.12$, $\sigma = 0.5$. These values were given to us in the handout and we chose not to tune them, since tuning 4 parameters in one model has the likelihood of becoming convoluted and we were already seeing changes in performance by tuning other parameters. The other main parameter involved in the sensor table was the size of the sensor table, which was 201 x 201. We also used the default values here for the same reasons as for the alpha values.

Once we had the precomputed sensor model, we evaluated the likelihood of a scan measurement $z_k^{(i)}$ being from a predicted point on the ground truth map x_k. As stated before, the precomputed sensor model's "inputs" are a lidar scan measurement $z_k^{(i)}$ and a distance on the ground truth map d. Using numpy functionality, we can input a matrix representing distances to obstacles on a ground truth map from various predicted points and a vector representing the lidar observation. We get the matrix of distances by using a ray tracing algorithm given to us. Using these two inputs to index into the probability matrix, we get the probability that a singular lidar beam measurement came from a predicted point. We then take the product of all these probabilities of single lidar bean measurements to get the probability that a lidar beam observation was made from a predicted point.

2.4 Physical Localization

Section 2.4 by Ari Grayzel

2.4.1 Problem

Once we had a working particle filter localizer in simulation, we had to transfer our algorithm to the physical platform. This process required minimal tweaks to our technical implementation, but still proved challenging in creating a resilient and reliable solution. An important detail of this is the requirement to localize the robot in the *real world*, as opposed to a simulated one.

2.4.2 Approach

The basic approach to transfer our model onto hardware was to update all necessary ROS topics and callbacks, and then test the platform and update any of our parameters to evaluate how they affected performance. We made no fundamental changes to how the algorithm works as the principles in simulation and on the physical platform are identical.

2.4.3 Implementation

Adapting our ROS implementation was fairly straightforward. Firstly, we adjusted all ROS topics as necessary. For example, instead of the /odom topic that our simulated particle filter subscribed to we instead subscribed to the /vesc/odom topic that provided odometry data from the variable electronic speed controller that controlled the robotic actuators. Additionally, instead of subscribing to a simulated LIDAR scan provided by our simulated robot pose, we subscribed to the actual data provided by our real LIDAR. The final step in transferring from simulation to real hardware was to re-tune algorithm parameters, primarily the amount of noise injected into our particles. This process is described in section 3.4.

3 Experimental Evaluation

3.1 Motion Model Noise Tuning

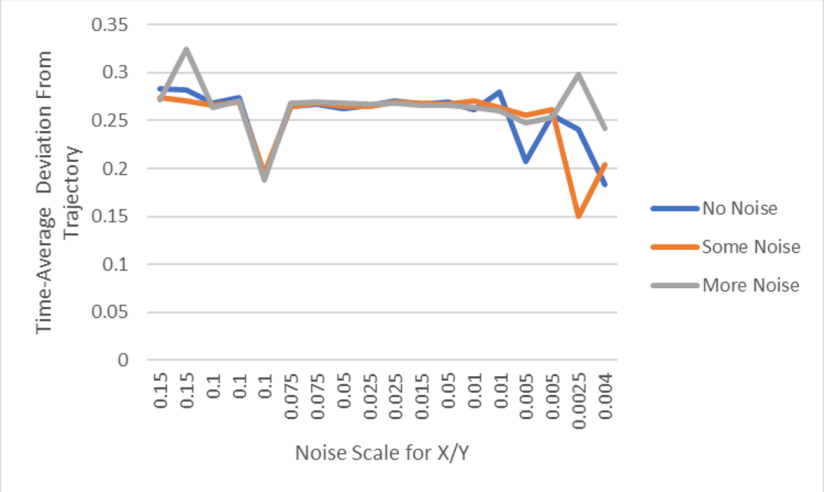
Section 3.1 by Sebastian Garcia
For our motion model, apart from the basic calculations of adding odometry into our predicted particles to predict where the robot is, we had to add noise into those predicted particles to prevent particle death.

Using numpy functions that generated a normal distribution based on a given mean and variance, we were able to generate random and unique noise for each particle in our set of particles. We added noise for each parameter in a given pose. We added noise from a distribution with a mean of zero for each. We adjusted to the variance from this mean to tune our noise. The variances for

x and y were kept the same, since they are on the same plane with a 1 to 1 scale. However, for the angle parameter, our noise had to be different, and have a variance in terms of radians from the mean of zero.

In order for our localization algorithm to work effectively both in simulation and on the real car, we had to tune the variances of the noise distributions in a series of trials. First for the simulation, which gave us a rough estimate of what the noise should revolve around, and later on the car due to differences between the map and the real world stata basement.

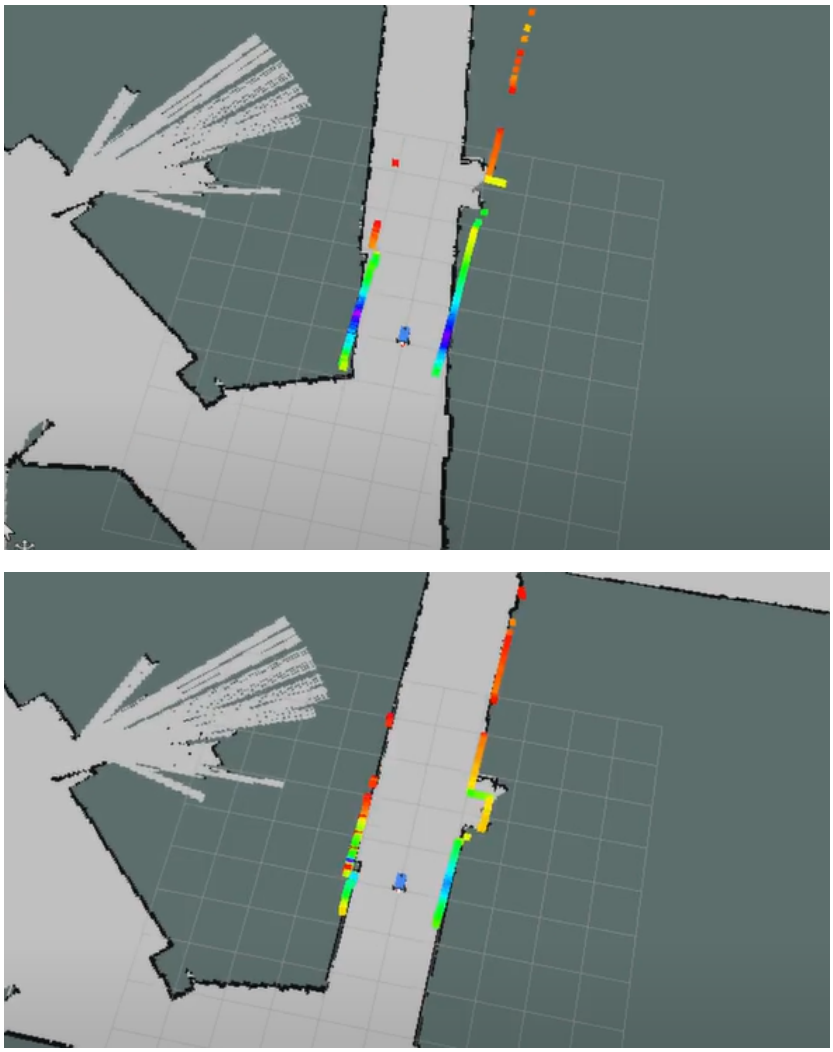
The way we did these trials are by measuring the time-average deviation of our localization algorithm's estimated trajectory and a ground truth trajectory. We started with noise parameters (variances) at 0.5 for x and y, and pi/18 for the angle. Seeing that these proved to give an incorrect trajectory and included messy patterns, we started to lower the variances. We kept the angle parameter constant at first, so that we only changed one value. We also kept the particle number at 5000. We ended up with a noise distribution variance for the cartesian parameters to be 0.05 for the most accuracy with an angle noise variance of $\frac{\pi}{18}$. Each time it was tested against trajectories which had no noise added to odometry, some noise, and a lot of noise. Here are the results of some of these trials.



If you look closely in the graph above, as we lowered the noise scales (variances), the error generally got smaller for all cases. The peaks shown at the ends of the graph are a good example of how even though our model is consistent a large amount of the time, sometimes due to chance it gets thrown off.

The trials performed above were for our simulated environment, and testing on the robot will be discussed in a later section. Another method of testing our algorithm in simulation was by obtaining a rosbag of our real car, and playing

it in simulation. We visually saw the errors as misalignment between the laser scans on the simulation map versus the actual walls and obstacles of the map. Additionally, in pure simulation we had found that $\frac{\pi}{18}$ worked well, but on our real car it was shrunk to a variance of $\frac{\pi}{180}$ for less volatility. Below is an example of our simulated car using our localization algorithm to correctly identify its pose with our final noise parameters. The first image is its starting pose, and the second is where it is after a few seconds.



Overall, our current parameters ended up being 0.5 for x and y noise variances, and $\frac{\pi}{180}$ for angle noise variance. It will be further tested in the future for greater accuracy.

3.2 Physical Testing

Section 3.2 by Ari Grayzel

The testing process on the physical robot was challenging to say the least. The requirement to localize the robot in a real map imposed constraints on our testing regime. Firstly, the only pre-made map of a location that we had access to was the Stata basement tunnels. Additionally, our position initialization became much more challenging. Compared to the simulation environment in which we were able to guarantee that the particles would be initialized to the same spot as the simulated robot, in the real world we had to do our best to situate the robot at the corresponding point in the real world as the initialized map position. Finally, perhaps the biggest challenge was the lack of a real "ground truth" to be able to compare our pose estimates to. This lead to a distinct lack of quantitative testing data. We instead relied heavily on subjective observations of performance.

The problems of only having an approximate initialization position became quite clear during our testing phase. In simulation, all particles were initialized at a single point. This was clearly optimal since we were sure that our particles were being initialized at the same location as the simulated robot. In contrast, initializing all particles at the same point functioned terribly in the real world, as any inaccuracy in our particle initialization relative to the real position of the robot would not be accounted for.

We tested various levels of cartesian and angular noise upon initialization. At high noise levels (the highest standard deviation in x and y we tested was 1m) particles would localize across too broad an area, including some which would localize inside of the wall. This created very large variation in particle positions and extremely unreliable localization. It is possible that larger number of particles could accommodate this as it is more likely that there would be sufficient particles sufficiently close to the actual robot position. We were of course limited in the number of particles we could use by the hardware and need to run at rates near 50HZ. We found our subjectively "best" initialization results with x and y standard deviation noise levels of 75cm and a heading standard deviation of 30 degrees. This is a relatively large spread, but ultimately the particles converged relatively quickly to the actual position of the robot, while not spreading out enough to trigger a possible false localization at a similar-looking spot.

Additionally, we found we needed to tune the noise injected in our motion model to different parameters than what worked optimally in simulation. Using a similar Gaussian distribution as our initialization routine, we determined experimentally that 10cm standard deviation of Cartesian noise and 1 degree heading yielded the best results on the physical platform.

4 Conclusion

Section 4 by Robert Cato III

In this lab, we successfully implemented Monte Carlo Localization in simulation. Our localization approach using a particle filter, motion model, and sensor model works robustly in the simulator where the initial location is precise and the ground truth map is static. However, the tuned simulation noise parameters did not extend well to the physical robot platform because it has a much higher degree of uncertainty in the initial pose as well as the ground "truth"; there are many sections of the Stata basement that are different than the provided map. We spent upwards of ten hours on debugging, tuning, and adjusting our localization package to work on the robot platform. We were unable to adjust the localization to run with decent accuracy by the end of this lab's time period. That being said, localization is critical to path planning so our team will need to spend additional time in the next week making our implementation work well on the robot.

5 Lessons Learned

5.1 Alicia Zang

One technical lesson I learned was that efficiency in computation matters a lot. Although I understood the reasons behind striving for efficiency, it didn't become apparent until this lab, where for loops through matrices are not as fast as specialized functions for matrix computation.

A communication lesson I learned was that giving updates on the progress of each part and letting other people know about other factors that might affect how much I could do on the lab. I also learned about making code easier for others to understand, like descriptive variable names and comments.

5.2 Ari Grayzel

Communication lessons I learned in this lab is how important it is to make sure that you communicate when you need help. There were several times in this lab I found myself working along and getting frustrated that there was nobody there to help me. What I should have done instead was be more proactive about communicating that the task required more people, and rather than frustrating myself and burning myself out I should have waited until I was able to get assistance with my problem.
My biggest technical lesson from this lab is more practice in efficient array slicing as well as refreshing my synchronous programming skills. I haven't done any sort of thread safety for about a year so it was good to get a chance to identify critical regions "in the wild".

5.3 Robert Cato III

The main communication lesson I learned from this lab was how to communicate with my team about my availability. I personally struggle with overworking

myself to the point of burn out and I learned through this lab that it's not a scary or painful process to let my team know that I need to work on something else (or eat a meal) for a little bit. I think this helped improve my contributions too.
The most important technical lesson I learned from this lab was how to better use numpy for fast computation. I use numpy modules in other classes but this lab really highlighted the importance and power of numpy's array indexing, broadcasting, and matrix multiplication.

5.4 Sebastian Garcia

The lessons related to communication that I learned in this lab involved needing to communicate that breaks were needed in order for more productive work sessions. I also learned that it's important to plan when to work with or around the team so that there can be support when needed. Technical lessons I learned included being able to optimize code using numpy, making code thread safe with thread locks, and structuring code in an organized way without just making a lot of functions.

5.5 Shreya Gupta

The main communication lesson I learned was being better about expressing when I was struggling to make time for 6.141 due to external factors. I think it's important to be transparent about your capacity to do work at any given moment, and I think I usually stress out a lot about taking a lighter workload. The main technical lesson I learned was how to write efficient and clean code, specifically in numpy to perform tasks that would otherwise require multiple for loops. I think in the past, I never had a project where speed was a large factor, so this was a fun and interesting challenge