Milestone 1: Obstacle Avoidance

Emily Sturman, [redacted], and [redacted]

1 Mathematical Computation

1.1 Minimum Braking Distance

The equation that we use for minimum braking distance is

$$d_{\texttt{braking}} = \frac{v^2}{2a} + v\ell$$

where ℓ is the latency, v is the current car velocity, and a is the deceleration rate.

1.2 Point Cloud

Our point cloud is defined by the set:

$$P = \{ \begin{bmatrix} r_i \cos(\theta_{\min} + i\Delta\theta) + x_{\text{laser}} \\ r_i \sin(\theta_{\min} + i\Delta\theta) + y_{\text{laser}} \end{bmatrix} \ : \ r_{\min} \leq r_i \leq r_{\max} \}$$

where r_i is the range value returned at index i of the LIDAR, $\theta_{\min}$ is the minimum angle of the LIDAR, and $\Delta\theta$ is the change in angle between each index in the LIDAR.

1.3 Free Path Length of a Straight Path

We have predefined/pre-measured values for $d_{\texttt{car\_width}}$, $d_{\texttt{safety\_margin}}$, and $d_{\texttt{base\_link\_to\_front}}$. The equation for the free path length ℓ on a straight path is:

$$\ell = \max(0, \min\{x - (d_{\texttt{base\_link\_to\_front}} + d_{\texttt{safety\_margin}}) : \ \begin{bmatrix} x \\ y \end{bmatrix} \in P \wedge x > 0$$
$$\wedge \|y\| \leq \frac{d_{\texttt{car\_width}}}{2} + d_{\texttt{safety\_margin}}\})$$

1.4 Free Path Length of a Curved Path

Let us define r_1 and r_2 as follows ($c = \begin{bmatrix}0\\ r\end{bmatrix}$):

$$r_1 = \|c\| - (\frac{d_{\texttt{car\_width}}}{2} + d_{\texttt{safety\_margin}})$$

$$r_2 = \sqrt{\left(\|c\| + \frac{d_{\texttt{car\_width}}}{2} + d_{\texttt{safety\_margin}}\right)^2 + (d_{\texttt{base\_link\_to\_front}} + d_{\texttt{safety\_margin}})^2}$$

Additionally, we will define a function adjust as:

$$\text{adjust}(x) = \begin{cases} x + 2\pi & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

We can thus write the full equation for our free path length ℓ as

$$\ell = \|c\| \,\text{adjust}(\min\{\text{atan2}(x, \|c\| - y) - \text{atan2}(d_{\texttt{base\_link\_to\_front}} + d_{\texttt{safety\_margin}}, r_1) \\ : \; p = \begin{bmatrix}x\\ y\end{bmatrix} \in P \wedge r_1 \leq \|p - c\| \leq r_2\})$$

1.5 Clearance on a Straight Path

We calculate the clearance of our vehicle on a straight path with the following equation:

$$d_{\texttt{clearance}} = \min\{|x| : \; \begin{bmatrix}x\\ y\end{bmatrix} \in P \wedge -d_{\texttt{base\_link\_to\_back}} \leq x \leq d_{\texttt{path\_length}} + d_{\texttt{base\_link\_to\_front}} + \\ d_{\texttt{safety\_margin}} \wedge \|y\| > \frac{d_{\texttt{car\_width}}}{2} + d_{\texttt{safety\_margin}}\}$$

1.6 Clearance on a Curved Path

Let us define the following function curvedClearance:

$$\text{curvedClearance}(p) = \begin{cases} \|p - c\| - r_2 & \text{if } \|p - c\| > r \\ r_1 - \|p - c\| & \text{otherwise} \end{cases}$$

We can calculate the clearance of our vehicle on a curved path with the following equation:

$$d_{\texttt{clearance}} = \min\{\text{curvedClearance}(p) : \; p \in P \wedge \|p - c\| < r_1 \vee \|p - c\| > r_2 \wedge \\ \text{atan2}(-d_{\texttt{base\_link\_to\_back}}, \frac{d_{\texttt{width}}}{2}) \leq \text{atan2}(x, \|c\| - y) \leq \frac{d_{\texttt{path\_length}}}{\|c\|}\}$$

1.7 Path Score

We calculate the path score with the following equation:

$$\texttt{score} = w_1 x_{\texttt{max}} + w_2 d_{\texttt{clearance}} + w_3 d_{\texttt{path\_length}}$$

where w_1, w_2, w_3 are nonzero weights (we set them to 2.0, 500, and 0.1 respectively through trial and error) and $x_{\texttt{max}}$ is the distance that the car can travel on the new path before reaching an obstacle. Note that $d_{\texttt{path\_length}}$ is not really considered in the score, since it will be equal for each path (as we do not have a goal point to travel to, and are just traveling infinitely forward).

2 Code Overview

We start off by filling the point cloud vector in `navigation_main.cc` and parsing the laser value by going through all the laser readings, and then reconstruct the point using Cartesian coordinate. We discard readings that are either too large or too small to be considered accurate, and then we finally apply a transformation to get the reading to correspond with the base link position.

At the center of our control logic, we iterate through each possible path (21 curvature options, from -1 to 1 curvature). For each option, we call ComputePath, which calculates a ComputePathResult (remaining distance, $x_{\texttt{max}}$, and clearance) from which a score can be calculated. We then take the path with the maximum score and set that as our current curvature, from which we can use our 1-D time-optimal algorithm to follow that path. We then recompute the curvature path at every iteration of our control loop.

3 Challenges Faced

The biggest challenge we faced for milestone 1 is figuring out a robust cost/reward function that can output scores that accurately represent the desired behaviors needed by the car to clear the obstacles. We know there are many different choices for the score function, including inverse quadratics, but we eventually settled with a simple linear combination of the previously described remaining distance, x_max, and clearance. However, tuning the weights proved to be very difficult, and due to the way clearance is constructed in our code, we found it hard to properly tune the car to take on a variety of scenarios smoothly. Furthermore, with our current scoring function, we are still having occasional edge cases where the program has successfully decided on the best path, but failed to execute on the real car due to the clearance setup.

Another challenge we have faced is the external factors, such as initial car positioning and obstacle setups. We were able to reconstruct a course similar to what was shown in the

demo video, however to test the more generalized ability of our car, we also made several other courses involving different kinds of obstacles. However, it was hard to determine what constitutes a "good" course, as we had multiple factors to consider when the car failed to navigate through the course. Could it be the course being too complicated for the scoring function? Or maybe the obstacles were placed too closely and the car did not have enough physical turning radius? Or could it be because the initial placement of the car creating a local maximum that trapped the car at the beginning of the course? It took a lot of effort, including visualization in simulations to rule out some of the problems, and even then we still couldn't say with certain if the fault is completely on the car or on the course.

4 Potential Improvements

One potential improvement that we'd like to make would be further adjusting our weights so the car can optimize its path better in certain corner cases as previously mentioned. For example, there are situations where the car can barely make pass an obstacle when chasing the maximized distance, and end up allowing the obstacle to enter the car's safety margin, thus causing the car to come to a stop. We are still uncertain whether this is due to our clearance setup or the weights tuning, so further investigations are needed.

Another area of improvement we can make is to construct a more powerful and generalized scoring function while incorporating other factors that can help steering the car through the obstacles. However this likely implies further and harder tuning will be required too.

5 Hardware Tuning

- Latency - as mentioned in the previous report, we found there to be a rather big discrepancy in the actual car latency versus the expected value (0.22-0.27s). This caused a lot of overshooting early on using the expected latency at 0.15s, and later using the method described in the lecture, by issuing the car an oscillating sine wave-like velocity, we can measure the phase shift between the expected wave and the recorded data, and the difference is then used as our real latency.

- ERPM Calibration - although we have gotten pretty good results from check point 0, we were still not satisfied with our accuracy with TOC distance driving. To get better results we also followed the instruction provided in the UT AUTOmata Reference Manual to calibrate the motor speed gain.

- Steering Calibration - obstacle detection relies a lot on whether the car's intended path aligns with the path it follows in reality, so we thought it would be a good idea to make sure that the steering was also calibrated correctly. For the offset, the methodology was to command the car to go at 1 m/s with zero curvature, and to see whether it tends

left or right, adjusting `steering_angle_to_servo_offset` in `vesc.lua` accordingly. It turned out that the initial value of `0.55` was pretty good. For the gain, the methodology was to command the car to go at 1 m/s with +1 curvature and -1 curvature, and adjusting `steering_angle_to_servo_gain` in `vesc.lua` to get the car to drive in a 1-meter radius. The previous value (approx. `-1.2`) was causing the car to turn too tightly, and we ended up lowering it to `-0.9`.

Milestone 2: Particle Filter

1 Mathematical Computation

1.1 Initializing Particles

When a pose is initialized at $(x_{\text{map}}, y_{\text{map}}, \theta_{\text{map}})$, we initialized each particle as (x, y, θ), where

$$
\begin{aligned}
x &\sim \mathcal{N}(x_{\text{map}}, \sigma_{\texttt{initialize\_xy}}) \\
y &\sim \mathcal{N}(y_{\text{map}}, \sigma_{\texttt{initialize\_xy}}) \\
\theta &\sim \mathcal{N}(\theta_{\text{map}}, \sigma_{\texttt{initialize\_theta}})
\end{aligned}
$$

1.2 Updating Particles

In `ParticleFilter::ObserveOdometry`, we update each particle location according to

$$
\begin{aligned}
p^{\text{robot},t+1} &= p^{\text{robot},t} + \Delta p \\
\Delta p &= \hat{\Delta p} + [\epsilon_x, \epsilon_y]^\top \\
\hat{\Delta p} &= \mathbf{R}(\theta^{\text{robot},t} - \theta^{\text{odom},t}) \times (p^{\text{odom},t+1} - p^{\text{odom},t}) \\
\epsilon_x &\sim \mathcal{N}(0, k_1 \left\| \hat{\Delta p} \right\| + k_2 \left| \theta^{\text{odom},t+1} - \theta^{\text{odom},t} \right|) \\
\epsilon_y &\sim \mathcal{N}(0, k_1 \left\| \hat{\Delta p} \right\| + k_2 \left| \theta^{\text{odom},t+1} - \theta^{\text{odom},t} \right|)
\end{aligned}
$$

When the line segment $p^{\text{robot},t}, p^{\text{robot},t+1}$ intersects with a wall in the map, we resample ϵ_x and ϵ_y. When the number of failed samplings exceeds a constant ($n_{\texttt{wall\_retry\_limit}}$), we set the particle equal to a particle chosen randomly from the existing particles, and repeat the sampling procedure up to $n_{\texttt{perturb\_retry\_limit}}$ times.

We update each particle angle according to $\theta^{\text{robot},t+1} = \theta^{\text{robot},t} + \Delta\theta$, where

$$
\Delta\theta \sim \mathcal{N}\left(\theta^{\text{odom},t+1} - \theta^{\text{odom},t}, k_3 \left\| \hat{\Delta p} \right\| + k_4 \left| \theta^{\text{odom},t+1} - \theta^{\text{odom},t} \right|\right)
$$

1.3 Computing Predicted Point Cloud

As input, we received the pose of the car in the map frame, (x, y, θ) as well as n (the number of laser beams, $\theta_{\min}$ and $\theta_{\max}$, the angular range of the LiDAR, and $r_{\min}$ and $r_{\max}$, the valid distances for each of the laser beams. We computed the position of the laser (in the map frame) as $\ell = (x, y) + R(\theta)(0.2\text{m}, 0.0)$ For each laser beam, we computed the endpoints of the beam as $\ell + R(\theta + \theta_i)(r_{\min}, 0)$ (near point) and $\ell + R(\theta + \theta_i)(r_{\min}, 0)$ (far point). Determining the nearest map intersection point for each laser beam was then done by iterating through `map_.lines`, computing the distance between the intersection point and the laser position, and taking the nearest intersection point for each beam.

1.4 Computing Particle Weights

We used the formula

$$p(s_t|x_t) \propto \prod_{i=1}^{i=n} p(s_t^i|x_t)$$

where

$$p(s_t^i|x_t) \propto \begin{cases} \exp\left(-\frac{d_{\text{short}}^2}{\sigma_s^2}\right) & \text{if } s_t^i < \hat{s}_t^i - d_{\text{short}} \\ \exp\left(-\frac{d_{\text{short}}^2}{\sigma_s^2}\right) & \text{if } s_t^i > \hat{s}_t^i + d_{\text{long}} \\ \exp\left(-\frac{(s_t^i - \hat{s}_t^i)^2}{\sigma_s^2}\right) & \text{otherwise} \end{cases}$$

Note that s_t is the LiDAR scan of the robot, and $\hat{s}_t$ is the predicted LiDAR scan of the robot.

1.5 Get estimated location

The estimated location is calcuated based on the weighted average of all particle's position and orientation. Specifically, given a set of particles, each with pose x_i, y_i, θ_i, and weight w_i we have:

$$\bar{x} = \frac{\sum_{i=1}^n w_i x_i}{\sum_{i=1}^n w_i}$$

$$\bar{y} = \frac{\sum_{i=1}^n w_i y_i}{\sum_{i=1}^n w_i}$$

$$\bar{\theta} = \text{atan2}\left(\sum_{i=1}^n w_i \sin\theta_i, \sum_{i=1}^n w_i \cos\theta_i\right)$$

1.6 Resampling the Point Cloud

The resampling procedure in our particle filter implementation is initiated through the `Resample` method. The core idea is to create a new set of particles from the existing set,

where each particle is chosen with a probability proportional to its weight. The process begins by constructing a cumulative weights array, `cumulative_weights`, where each entry is the sum of weights of all particles up to that point. This array serves as a means to select particles based on their weights. A new array `new_particles` is then created to hold the resampled particles. For each entry in `new_particles`, a random value is drawn uniformly from the range [0, total weight of all particles]. Utilizing the `lower_bound` function, the code identifies the index of the smallest value in the cumulative weights array that is greater or equal to the random value, thereby selecting a particle from the existing set. The selected particle is then copied to the `new_particles` array.

The Resample method proceeds to replace the old set of particles with the new resampled set by assigning `new_particles` to `particles_`. Additionally, it resets the `distance_since_last_resample_` to 0.0, as a measure to track the distance travelled since the last resampling. This resampling strategy ensures that particles with higher weights, which correspond better to the observed data, are more likely to be duplicated in the new set, while those with lower weights are likely to be discarded. This way, the particle set stays representative of the probable states of the system, which is crucial for the effective operation of the particle filter.

2 Tuned Parameters

We started with resampling disabled in order to make sure that our motion model and observation likelihood functions are robust enough that they will not immediately deviate from the actual translation and rotations of our real car, and thus resulting in even more false biases during our resampling process. One important parameter was the `KdShort`, which is the low clamping threshold for our observation likelihood function, i.e. obstacle tracking. Since unlike the maps, the actual environment is filled with obstacles like chairs and tables, and these noises would greatly affect the performance of the particle filter.

Once we achived an acceptable behavior, we reintroduced resampling. We tested our solution against the bag files initially released one week prior to the due date of the milestone, and we found that aside from `single_turn_reverse.bag`, the performance of our implementation was acceptable as-is. However once `basement.bag` was added to the testing pool, we found our code struggled to track the car once it started turning into the narrow hall-way before reversing happened. We concluded that it was due to the update standard deviation being too low , and thus did not have enough variance to allow more particles to spread out and performance wall-matchings. Therefore along with increasing the standard deviation, we also introduced more variances in the motion model and re-adjusted the k-values to account for both better translational and rotational performances.

The next challenge was the atrium hall, which due to its emptiness, we also had to increase the number of predicted races in order to get more data point for wall matching. Previously we are only picking every 54th ray for observation likelihood calculation, but we eventually increased this to using every 12th ray. We also found that by resampling more often we observed better

result with the atrium bag files, but we also further increased the standard deviation and variances, so that for other scenarios we don't risk having the particles collapsing into one singular point before having a chance to spread out and making better predictions.

- $\sigma_{\texttt{initialize\_xy}} = 0.12$, $\sigma_{\texttt{initialize\_theta}} = 0.07$
- $k_1 = 0.165, k_2 = 0.369, k_3 = 0.158, k_4 = 0.47$
- $d_{\text{short}} = 1.0, d_{\text{long}} = 6.0$
- $\sigma_{\text{short}} = 0.21, \sigma_{\text{long}} = 0.21$
- $d_{\text{resample}} = 0.12$
- $n_{\text{ray}} = 12$
- $n_{\text{wall\_retry\_limit}} = 5$, $n_{\text{perturb\_retry\_limit}} = 25$

3 Challenges Faced

The biggest challenge for this milestone is to properly tune the various parameters for our particle filters, from the motion values constants to resampling and update parameters. Since there are a lot of parameters that require adjustment, they often mask other underlying issues of our particle filter implementations. For instance, for the the previous checkpoint, we encountered a hardware related problem and thus was not able to tune our update functions properly, and resampling can exacerbate this problem as we would be resampling from inherently flawed probability calculations. Furthermore, the causes for some of the tracking errors were not intuitive, and their relationships with the parameters were also not immediately apparent. Adjusting the parameters and observing the results were our best strategy, but it also meant that it was not the most efficient way possible and thus time consuming. There were also occasions when we observe an improvement after some tuning to one variable, but upon modifying another parameter, our performance drops, with the only solution being adjusting the previous parameter again. One curious case was when we decreased the number of particles used by our particle filter, which resulted in significantly better tracking performance on the two Atrium bag files. Moreover, we also encountered challenges in generalizing our implementation for different environments. For example, we would tune parameters to work well with the GDC2 atrium rosbag files, but they would then not work for some of the more complicated GDC basement rosbag files. It took a lot of trial and error to find values that were sufficient for multiple maps.

Another challenge that we have encountered for Milestone2 was another hardware issue. After some unexplained lag in our remote terminal, we noticed that vim is no longer allowed to write the changes we made, with a warning on low disk space, despite system info telling us

otherwise. Then when we attempted to restart the car to clear out the VSCode server cache, we found that the car no longer boots into its operating system and thus could no longer be used for programming. Furthermore, despite Arnav's attempt at fixing the car for us, we wasn't able to get our code running before the milestone deadline. Even though this milestone is largely workable in simulation by comparing our particle performance with the given bag files for references, but we have also encountered certain issues that would be detected much quicker if we had the car to drive around. For instance, one problem we had with our original particle filter implementation was that instead of using `math_utils::AngleMod()` (which outputs in the range $[0, 2\pi)$), we called `std::fmod()` (which outputs in the range $(-2\pi, 2\pi)$). This can cause our particle filter to disperse and lose tracking instantly. Together, these problems cost us a lot of time during debugging and tuning.

4 Potential Improvements

Due to the nature of random particle spawning and likelihood function calculations, there are inevitable run to run variances that may result in our particle filter to perform poorly occasionally. One future improvement would be making the overall implementation more robust and less prone to the noises in the environment that could lead to further variances in our tracking accuracy. For instance, one area of improvement is better likelihood function calcuation. During testing with the bag files initially provided, we found that due to the lack of rear laser scan data of the LiDAR sensor, reversing presented us a lot of troubles, and we noticed that with the limited scan data could falsely correlate the obstacle objects in the real world with the map vectors, thus greatly affecting our car's translation tracking and resulting in errors too large to be corrected. One potential solution to this problem is to switch to the point-normal forms and more cases for our observation likelihood calculation introduced in the lecture, and was said to be able to fix this exact problem.

Other improvements include further parameter tuning, such as exploring with even fewer particles, less predicted scan calcuation, and resampling less often, which could all potentially increase the robustness of our implementation as well as the computational cost on our car. Currently we are noticing some lags in our particle filters as the car traverses through the environment, which the aforementioned optimizations could help. (Note that in our video linked below demonstrating our car running with particle filter, the lag in the latter half of the video is due to the transmitting issue as the car movesfurther away from the computer running visualization, which is unrelated to our implementation's actual performance).

Milestone 3: Navigation

1 Mathematical Computations

1.1 Search Algorithm

We implemented a search-based planner, in part due to time constraints and in part due to familiarity. The map is preprocessed ahead of time to generate a grid graph which represents the map topology, which is later traversed at runtime using a shortest-path algorithm. We also implemented parts of a lattice-based graph preprocessor, but ultimately stuck with the grid-based preprocessor in the interest of algorithmic complexity (our lattice-based graphs were orders of magnitude larger than the grid-based graph).

For the shortest-path algorithm, we first implemented BFS, then Dijkstra's algorithm, then $A^\star$, making incremental modifications. $A^\star$ search is similar to Dijkstra, as it uses a priority queue to prioritize lower-cost nodes. However, it also uses a heuristic (in this case, Euclidean distance), which is added to the cost in the priority queue so that nodes are also prioritized by how close they are to the destination node. This way, $A^\star$ is able to search fewer nodes than Dijkstra in order to find the shortest path from source to destination. We also implemented parts of a JPS algorthm, but ultimately stuck with $A^\star$.

1.2 Path-Following Algorithm

The algorithm that we used to follow the planned navigation path was based on the carrot navigation controller. First, we determine a "look-ahead point" by finding a point on our path that is a specified distance away (we used 1m) and setting that as our temporary navigation destination. Then, we run our previous obstacle avoidance navigation towards that point, and, once we get close enough, repeat the process to find the next look-ahead point on our path.

2 Adjustments to previous code

The main adjustment that we had to make to previous code was within `navigation.cc`. As detailed previously, we implemented a simple carrot navigation controller for our path-

following algorithm. To do this, we had to replace our previous navigation (which simply set its destination as the forward direction) with a function that would find the next look-ahead point in our planned path, which would then execute our obstacle avoidance code.

3 Challenges Faced

One challenge we faced was that our car stopped turning on after the file system was believed to corrupted, while the replacement car's LiDAR refused to connect to the main board. Because of this, we were only able to test our code in simulation originally.

For our initial implementation of the pure pursuit algorithm, we were able to have the car to follow the planned route in simulation with some good accuracy. However, we found the logic hard to integrate with what we had from Milestone 1, and after a rewrite with the integration in mind, we still ran into troubles similar to what we saw in Milestone 1 initially, where the car was not able to properly navigate through the obstacles and instead would drive up straight into the obstacle before stopping, as the obstacle enters the car's safety margin. Unlike Milestone 1, however, the tuning was not entirely straightforward as we need to balance between making sure the car follow the planned path while avoiding the obstacles. Furthermore, it was also crucial to determine when to replan so that when our car deviates too far from the planned path, it can still have a chance to reach the target.

In the end, we achieved a good performance and balance between path pursuit and obstacle avoidance by tuning the `score()` function and giving a heavy weight to the local free path length while maintaining good obstacle clearance so we don't push the car into a situation where obstacles could enter its safety margin and causing an emergency stop. By focusing on the free path length we were able to eliminate the issue where the car drove into obstacles and stopped.

4 Cool Results

The video contains two clips show casing the car's real world performance. In the first clip, the car traveled along a planned path until obstacle avoidance was required, and once it determined that it has cleared the obstacles, it continued to follow the obstacle. In the second clip, we showcased the car's ability to replan its path when it's placed in the opposited direction of the path. Due to the noisy environment (the 2d Lidar sensor struggled to read the chairs and desk legs properly), and being unable to take curvature small enough to quickly converge to the new path, it traveled around the TA station's desks twice before it was able to find a path that leads it back to the original target, which demonstrates that even in a tricky environment like the TA station, our car still had the ability to recognize the need for replanning and navigating through the diffcult environment.

SLAM using Iterative Closest Point

1 Description

Simultaneous localization and mapping, or SLAM, is a process that allows autonomous vehicles and other robots to map their surroundings without a preexisting map. It is used in a variety of applications, from industrial robots to home vacuum bots like the Roomba. Our goal was to implement SLAM on our F1/10 car and allow it to map out a small area based on LiDAR scans.

Ultimately, we were able to get SLAM working in simulation on real data in real time, but we were unable to get it working on the real car in real time (see Challenges section).

2 Theory

On a high level, LiDAR SLAM works by taking a set of LiDAR point cloud scans collected from small sections of the environment, and composing them together to create a point cloud representation that best represents the overall environment.

The main subroutine is the alignment of two point clouds, which involves finding the rotation and translation which minimizes the difference between each cloud. By collecting successive scans that are sufficiently similar to each other, and finding the optimal rotation and translation between each pair of adjacent scans, we can combine all scans into a map.

A first approach is to use the car's odometry data to approximate the rotation and translation between successive scans. However, odometry errors tend to accumulate, especially over longer distances.

2.1 Iterative Closest Point

We used the iterative closest point (ICP) algorithm to align successive scans. ICP involves using a predetermined error metric to measure the difference between two point clouds, and then then iteratively applying translations and rotations on the point cloud so as to minimize this error.

Our main implementation of ICP uses the point-to-point error metric, which is the sum of the squared distances between each point in the new cloud and its closest point in the old cloud. Formally, for every point in the new point cloud $p \in P$, we find (using a k-d tree) $q_p = \arg\max_{q \in Q} \|p - q\|$, and let the error be

$$E = \sum_{p \in P} \|p - q_p\|^2$$

We can then approximate the optimal translation by comparing the two point clouds' centers of mass, and the rotation R can be computed by multiplying the orthogonal matrices from the SVD of the cross-covariance matrix K of the two point clouds:

$$USV^\top = \text{SVD}(K)$$
$$R = UV^\top$$

This doesn't give the optimal translation in one step, but in most cases this process can be repeated to achieve arbitrarily small error values.

2.2 Outlier Rejection, Point Weighting

Consider the edge case where the car is driving from room A to room B. When the car collects a scan right before it enters room A, and collects the next scan right after it enters room A. These two scans may look very different from each other, which poses a challenge for alignment:

Figure 1: In orange, the first scan; in blue, the second scan; both shown in the laser frame.

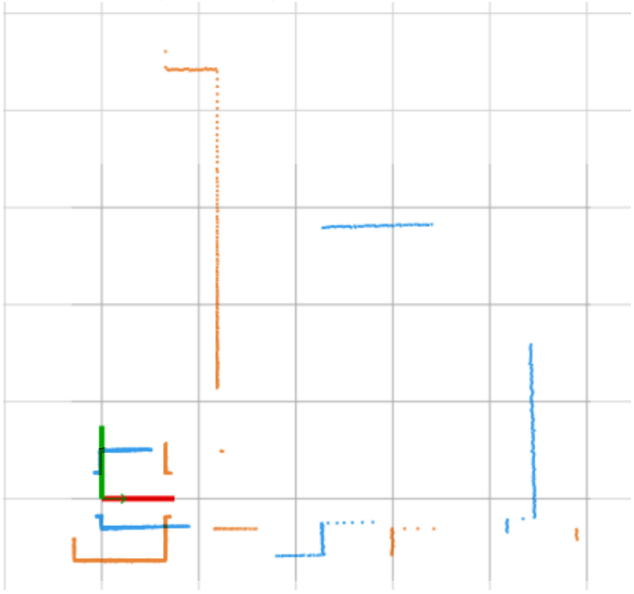
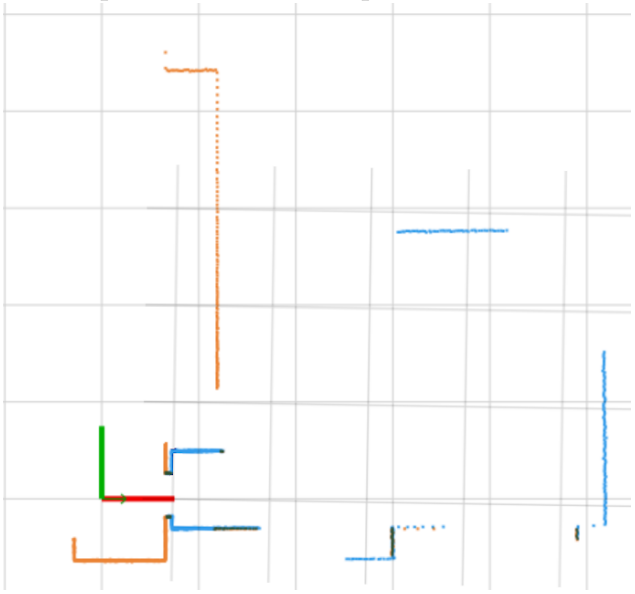
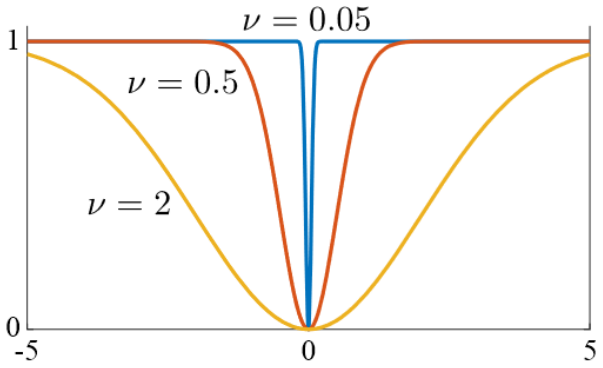


Figure 2: A ground-truth alignment of these two scans, shown in the frame of the first scan. Note there is very little overlap between the two point clouds.



An extra step can be introduced to help solve this problem, which involves giving less weight to so-called "outlier" points p which are not close to any point in the old scan. (In this context, "weight" can be thought of as a contribution to the error metric E discussed earlier. In other words, more weight is bad.) We implemented a partial version of Welsch's robust regression, based on the ideas in [FRICP]. The full version of Welsch's regression involves using a windowing function which changes with each iteration, to give more and more weight to points whose error residuals are close to optimal.

Figure 3: We used $\psi_\nu(x) = 1 - \exp(-x^2/2\nu^2)$, based on the paper. The error residual x is plotted on the horizontal axis, and the weight multiplier $\psi_\nu(x)$ is plotted on the vertical axis. As ν decreases, the outliers contribute more and more to E.



Our partial version uses a non-dynamic Welsch function, i.e. ν is a compile-time constant. In the ICP described in the paper, ν is updated in each iteration based on the median distance between each point and its 6 nearest neighbors, which is not an operation supported by Dr. Biswas's k-d tree by default. We considered implementing said operation, or designing our own protocol for iteratively decreasing ν, but were not able to get around to it.

3 Challenges Faced

3.1 Bugs while developing point-to-point

Point-to-point was generally quite painful to debug. Some issues included:

- Using `acos` to extract the angle from a rotation matrix. Several sources we found online were using combinations of `acos`, `asin`, and the determinant to recover the angle from the rotation matrix returned by ICP, so we were doing the same. `acos` only recovers the correct angle for half of the unit circle, so we were map misalignment that was very difficult to reproduce. After a stroke of luck, we switched to `atan2` and the problems disappeared.
- Normalizing the centers of masses of successive point clouds as a first guess, before starting ICP. This caused severe errors when outliers caused a huge shift in the center of mass of the scan, e.g. transitioning from room A to room B. The fix was to assume that the center of the new point cloud was equal to the map location of the previous scan, and to not normalize. Our results with this method were decent, so we did not investigate futrher. One potential improvement may be to factor odometry readings into the initial transformation.
- Various bugs with applying the correct transformations to align successive point clouds in the visualizer.

3.2 Point-to-Plane ICP

One challenge we faced was attempting to implement point-to-plane ICP. Many of the resources that we found only addressed a point-to-plane implementation in 3 dimensions, for which they were able to use the cross product. We spent a considerable amount of time debugging point-to-plane, but were unable to finish it in time.

3.3 Running on the Car

As described previously, we were unable to run our SLAM code on the car. When we tried, the car would repeatedly crash with the transformation matrices containing full of NaNs,

which we assume was due to the lower computing power and before one cycle of computation could be completed, the new lidar data was already received and might have caused race conditions that led to the transformation matrices being calcuated incorrectly (although it would crash even when we reduced the number of iterations to 3). Ultimately, we did not have time to fully debug this, as it was an issue that we didn't encounter until fairly late.

3.4 Pose graph optimization

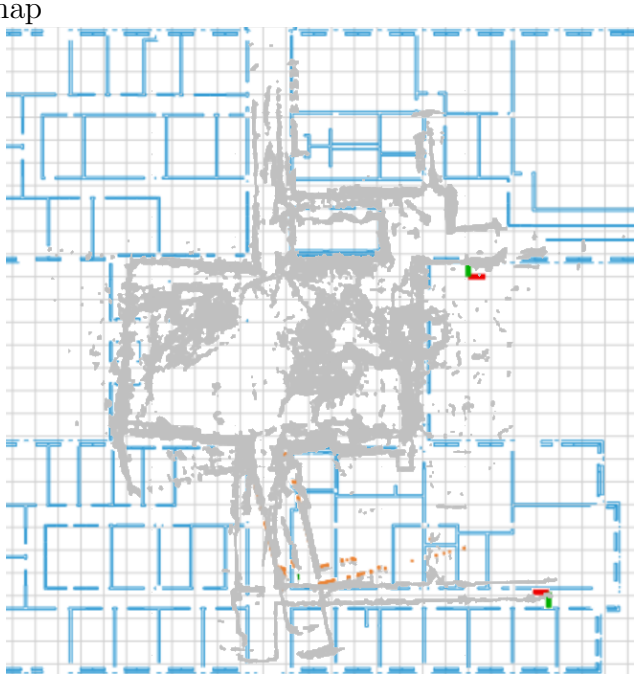
Pose graph optimization is a way to align not just successive point clouds, but to optimize the alignment globally. Maps constructed with global optimization tend to more accurately reflect the real environment, so this was something we were very interested in investigating. We played around a bit with the GTSAM library, but due to the more urgent problems described above, we did not have enough time to integrate it with our implementation.

3.5 Mysterious bug

Sometimes, our transformation matrix will output huge values for the translation, anywhere from kilometers to yoctometers to NaN. We checked to make sure all variables are initialized, but this still happens sometimes. Our current fix is to manually check for these values and terminate the SLAM program if they're unreasonably large, and then rerun.

4 Results

Figure 4: Comparison between our scan (generated from driving around in GDC2) and UT AUTOmata's GDC2 map



5 References

- CS 393R SLAM Assignment
- Dr. Biswas's k-d tree implementation
- ICP explainer with Python code
- Fast and Robust Iterative Closest Point (2020) (Github)
- Symmetric Point-to-Plane ICP (2019)
- Factor Graphs and GTSAM
- Efficient variants of the ICP algorithm (2001)