# Path Planning and Execution

Team 10

Jeremiah Budiman
Miranda Cai
Tiasa Kim
Bill Kuhl
Savva Morozov (editor)

MIT 6.141/16.405
Robotics: Science & Systems

April 15, 2022

## 1 Introduction

**Tiasa Kim**

Path planning, a.k.a. motion planning, is a crucial problem in robotics and autonomous systems because assuming that the robot or vehicle is dynamic, there is likely an end or goal destination in mind. For example, an autonomous mobile robot is often tasked with moving a storage rack from one aisle of a warehouse to another; a self-driving car may be tasked with transporting a passenger from their home to work, to the grocery store, then back home. Likewise, the goal for our racecar is to plan and execute a path in the environment (i.e. the Stata basement) (see Figure
Getting a racecar to move from A to B is challenging because there are multiple factors to consider and components required. Some of these constraints include identifying and avoiding obstacles in the environment, determining where a valid path exists. In the context of this lab and the final race, we are given a predefined map of the environment (i.e. an occupancy grid of the map). We can precompute paths based on the known dimensions, land points, and obstacles; however, in the real world, the environment is not always known nor do all obstacles remain static. Beyond finding a valid path, it's important for the racecar to be able to autonomously drive along the path. Therefore, motion planning is a dynamically complex process of learning, computing and executing.

When we consider path planning as an optimization problem, our goal is for the car to not just take any path to its destination, but the shortest path which forgoes any obstacles along the way. This criteria translates to our core values of speed and safety. Ensuring the shortest path means there is no other path the car could take to get to its goal location at a quicker rate. Likewise, a path hitting no obstacles means the physical racecar as well as any stakeholders are at minimal risk.

While a good amount of the work in this lab is an extension of previous lab work, Lab 6 focuses on planning (as the novel task) and control. Given our determined path, our task is to determine the racecar's pose within the world (i.e. via state estimation and localization from Lab 5). Once successfully doing so, we run command control inputs to follow the path using a path-tracking controller (i.e. pure pursuit from Lab 3).

In this report, we discuss two different algorithmic implementations of a path planner. The first is A*, a shortest-path search algorithm similar to Dijkstra's. In our approach, A* is paired with pure-pursuit to enact a planner-controller system. The second is Model Predictive Path Integral (MPPI) control, a

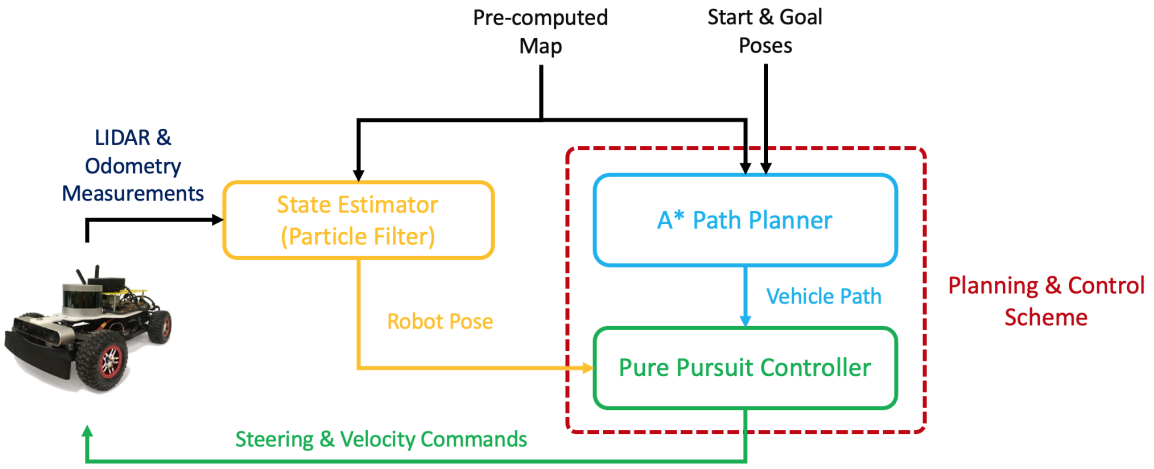



Figure 1: **High-level overview of the navigation framework.** Sensory measurements are aggregated by a particle filter to produce robot pose estimates. Upon user's request, the A* planner generates a path from the robot's location to the goal. Using current robot pose estimates, the pure pursuit controller then tracks this path by issuing control commands to the car.

sampling-based algorithm which optimizes path costs using stochastic sampling of trajectories.

In the following sections, we describe an overall framework of our system, the two comparable planning approaches we implemented and a complementary controller, an evaluation of our work in simulation and real-life, followed by concluding remarks and future work.

# 2 Technical Approach

## 2.1 Overview

**Savva Morozov**

In this section we will first briefly explain our overall navigation framework. We will then describe the process of pre-processing the environment map, as well as the two Planning and Control schemes that we implemented and tested in this lab.
Our A* navigation framework is shown in 1. First, the user provides a pre-computed map of an environment. We use a particle filter (yellow block, discussed in detail in Lab 5), which aggregates LIDAR and wheel odometry measurements, to produce an estimate of the robot pose. Upon user's request, the A* planner (blue block) generates a path between the initial and goal positions of the car. This path and the robot pose estimates are provided to the pure pursuit controller (green block), which issues control commands to the vehicle to follow the path. The only difference between MPPI and A* schemes is that MPPI is both a planner and a controller: at each timestep, it generates vehicle trajectories — control sequences — which are directly issued to the car. For completeness, the MPPI framework is shown in 7 in the Appendix.

## 2.2 Map Pre-Processing

**Jeremiah Budiman**

The goal of map pre-processing is to convert subscribed data into something that can be used in mapping. The fundamental priorities when pre-processing include quickness and safety — planning should not be computationally slow and should not route a path that could result in collisions. At a high level, pre-processing requires two steps: one is to reduce resolution of the map to make the search

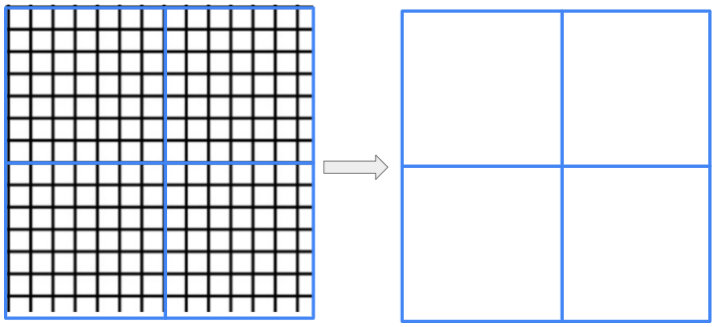


Figure 2: **The process of reducing the resolution of the original map.**

space smaller. The second requires creating a safety cost map to enable planning that will more likely avoid coming into close contact with obstacles.

#### 2.2.1 Reduce Resolution

In order to discretize the map properly, both obstacles and unknowns within the 2D grid will be treated as obstacles and the rest of the grid will be treated as free space. For the purposes of path planning, the map of the stata basement is significantly large. Specifically, the map includes over 2 million pixels, which leads to over 2 million cells in the converted discretized grid. Running A* on this grid would be computationally very slow.

In hopes of increasing the speed of the algorithm, the solution implemented was to reduce the size of the map by reducing the resolution. As shown in Fig. 2, this is done by taking every 7x7 block of cells and converting it into its own single cell in a newly created grid. The value of the new cell is the maximum value found within the 7x7 block (1 if there exists a free space and 0 otherwise). 7x7 was specifically chosen through qualitative testing — the size of the block would increase up until path-planning accuracy decreased. Running A* on this reduced-sized graph increases the performance of the overall algorithm without trading off accuracy.

#### 2.2.2 Safety Cost Map

In order to meet the goal of safety, a safety cost map must be implemented. Eroding the map did not sufficiently result in safely routed planned paths so instead, an additional heuristic was created where cost increases as distance from the robot to the wall decreases. The reasoning for this is that A* expands lower cost paths first and will therefore be incentivized to take routes farther from the wall in order to decrease cost.

To implement this, within the discretized grid, the euclidean distance between each free space and the nearest obstacle will be calculated. If a free space is a $\max_{\text{dist}}$ away from its nearest obstacle, the cell will have a cost of 0. Otherwise, the cell will have the following cost:

$$\frac{\max_{\text{dist}} - \text{euc}_{\text{dist}}}{\max_{\text{dist}}}$$

In implementation, the value of $\max_{\text{dist}}$ was tuned to be 4 through testing.

The eventual result might create a safety cost map as shown in Fig. 3, where costs are lower the further from the nearest obstacle free space is. When path planning, the values of these costs can easily be indexed into using the safety cost map.

### 2.3 A* and Pure Pursuit

#### 2.3.1 A* Planner

**Miranda Cai**

| | | | | | |
|---|---|---|---|---|---|
| wall | wall | wall | wall | wall | wall |
| High # | High # | wall | wall | High # | High # |
| High # | High # | High # | High # | High # | High # |
| Mid # | High # | High # | High # | High # | High # |
| Mid # | High # | High # | High # | High # | Mid # |
| Low # | Mid # | Mid # | Mid # | Mid # | Mid # |
| Low # | Low # | Low # | Low # | Low # | Low # |

Figure 3: **Example safety cost map with cost decreasing as the distance to the wall increases.**

To find the path that the racecar should follow to reach the goal point, our group decided to implement A* for our search algorithm mainly due to its compatibility with the given map and utilizing custom cost functions. As explained earlier in Section 2.2, we are given the map in the form of an array of pixel values, which after preprocessing we convert to become a discretized grid containing information on the locations of potential obstacles. Because our map has already been represented in this way, indexing into the discretized map during our search algorithm allows us to easily check whether continuing the path with a specific node would result in a collision. If the node's grid value is 1, then the map is stating that the space is occupied by an obstacle and cannot be driven through, in which case we ignore the node so that it can never be part of the racecar's path. Otherwise, the space is free and the car may use the node location in its path. Thus collision checking is quick and we do not spend too much time on this within our algorithm, while still ensuring that the path that we eventually set the racecar on is safe.

Secondly, by nature of A*, we are easily able to implement yet another layer of safety into the path we find. The way that A* works is that starting with the start node, A* will in increasing order of our self-defined cost function, append all allowable nodes and their neighbors into the path we are building until we have reached the goal node (or have discovered that none exists). The cost function is completely defined by the implementor, and in order to emphasize the importance of safety and speed our group defined the cost function *f(n)* of the current node $n$ in the following way:

$$f(n) = g(s, n) + d(n, t) + w(n)$$

$$g(s, n) = g(s, n-1) + 1, g(s, s) = 0$$

Where g(s,n) is the distance of the current path from the start node $s$ to $n$, $d(n, t)$ is the Euclidean distance from $n$ to the goal node $t$, and $w(n)$ is the cost of how close $n$ is to an obstacle by indexing into our safety cost map defined in Section 2.2.2. Because the grid is discretized into pixels, all of the nodes are a distance of 1 pixel away from each other, which is why the second equation defines path length to the current node as 1 + the path length to the previous node. As a reminder, the overall objective of having a search algorithm is to return a path for the racecar to follow that minimizes the total distance traveled so that the car can reach its destination faster. Thus, we incorporate $g(s, n)$ to directly implement this objective.

On the other hand, our addition of $d(n, t)$ and $w(n)$ are merely heuristics in order for us to discover this path both quicker and safer. Because A* builds the path node by node, we want to minimize the total number of nodes we have to explore before reaching the destination. Therefore by adding $d(n, t)$, which results in a greater cost the farther away we are from the goal node and a lower cost the closer we are, we make sure to prioritize the nodes that are closer to our destination in our exploration

queue. However, while we would ideally like to achieve the shortest path possible, we would also like to minimize the chance of the racecar crashing as it moves. This is where our second level of safety comes in, where we intentionally choose to avoid nodes that are too close to an obstacle by adding $w(n)$ to our cost. Unfortunately, in real life the car may not always be able to perfectly follow the planned path, and even small oscillations around a planned path that hugs the wall could be dangerous. Thus as a preventative measure for real-time error, we add a higher cost towards nodes that are closer to the wall such that A* will find and return a path away from walls if it exists before it finds a path next to the wall.

Another benefit of choosing A* is that it builds a path using only neighboring nodes on the map, such that the paths we build are high resolution, resulting in much smoother paths than had we implemented a randomized search algorithm where chosen nodes can be unevenly spaced such as in Rapidly-exploring Random Tree (RRT). By exploring all of the "best" possible coordinates using our cost function, A* guarantees both a shortest path within our constraints and a contiguous path. The biggest downside towards using A* is that running a comprehensive algorithm can take a long time depending on how many nodes we need to test before reaching the goal node. Refer to Section 2.2 on ways we counteracted this possible slowdown during preprocessing.

### 2.3.2 Pure Pursuit

**Bill Kuhl**, **Savva Morozov**

We decided to re-use the pure pursuit algorithm that was used in Lab 3: Wall Follower. We decided to use this as compared with other control techniques (PID) due to our experience with Pure Pursuit and its demonstrated robustness and modality to a variety of scenarios we may see when trying to follow the paths created by our A* algorithm. A brief description of the algorithm is given below.

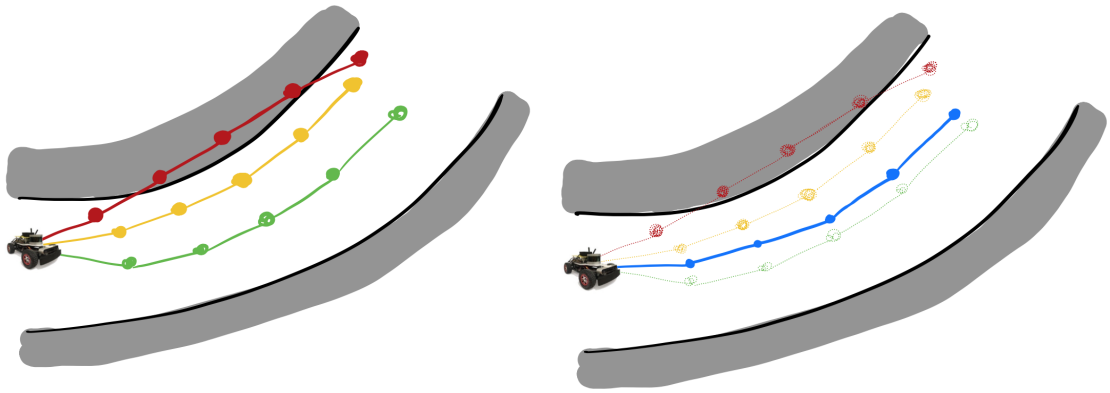
Pure pursuit is a path-tracking algorithm. It works by generating an arc between the racecar's current location and the desired point. This desired point is selected to be on the A* path, such that this point is the look-ahead distance away from the car. Pure pursuit then devices the steering command that enables the car to track the generated arc. We refer the reader to our Lab 3 report for more information.

## 2.4 Model Predictive Path Integral Control

**Savva Morozov**

In addition to A*, we also implemented Model Predictive Path Integral control (MPPI), as it was shown to perform safe, fast, and reliable maneuvers on Georgia Tech's AutoRally vehicle.
MPPI is a control strategy from a family Model Predictive Controllers (MPC). MPC is an iterative finite-horizon trajectory optimization algorithm. At each timestep, MPC receives an updated state estimate, replans a trajectory, executes the first control input from this trajectory, and does it all again at the next step. Two points ought to be emphasize here. First, MPC generates trajectories, not paths. By directly considering the vehicle dynamics over time, MPC is able to find high-speed trajectories that can be executed by the robot. Second, MPC generates local trajectories (over next $T$ steps) that are re-evaluated at each step. This allows MPC to safely handle uncertainty in the realization of the vehicle's dynamics.

Though most MPC formulations are optimization-based, MPPI is sampling-based. At each timestep, MPPI performs $J$ iterations of importance sampling from the distribution of control sequences. At each iteration, it samples a sequence of control inputs centered around the previous control trajectory (Fig. 4a). These samples are also known as rollouts: these control sequences can be translated into vehicle state trajectories using the vehicle's dynamics (we use the same nonlinear vehicle dynamics as in the simulator). Each rollout is given an importance weight based on a cost function that defines desired performance; here, we use the same cost function as in the A* planner. The result of the iteration is



(a) Sampled trajectory rollouts.

(b) Weighted mean of the rollouts.

Figure 4: **Single iteration of MPPI.** At each iteration, MPPI samples a sequence of control inputs (a). These control inputs are given importance weights (red trajectory is higher cost and lower weight, as it goes through the wall; green is lower cost, higher weight). A weighted mean of the control sequences is produces an updated control trajectory (blue, (b)).

a weighted average of these rollouts, which reprsents an updated control sequence (Fig. 4b). We refer the reader to [WDG+18] for a detailed description of the algorithm.

# 3 Experimental Evaluation

## 3.1 A* and Pure Pursuit

**Bill Kuhl**

### 3.1.1 Validation in Simulation

After creating our A* and pure pursuit packages, the next step was to validate in simulation. To use pure pursuit given a series of points, it was necessary to generate a series of line segments between the points on the paths. The nearest line segment to the car looked at to see if any point on it is at the lookahead distance. if not, the next line segment is selected until one comes up that is within the appropriate lookahead distance. By doing this a continuous path is able to be followed by the racecar.

After making sure the pure pursuit algorithm could process the path, experiments were conducted in simulation to ensure that the pure pursuit algorithm was tuned correctly so it would not oscillate or hit walls. Additionally the A* algorithm was tuned to create more distance away from walls to account for the uncertainty created from hardware testing and localization.

Tests were then conducted in RViz to ensure that the car could handle making and planning long paths, turning corners, navigating tight spaces, and operating at speed with the pure pursuit controller. The results show that the car performed exceptionally well at all speeds. Validating on gradescope, measuring the actual position of the racecar with the path, our implementation was able to achieve 98.97% accuracy at 10m/s, the top speed. This path can be seen in Fig. 5.

### 3.1.2 Hardware Validation

Four our hardware validation attempts, we had two goals. The first is to validate that our entire framework works in hardware and is not overfit to the simulation. The second is to ensure the performance is robust and repeatable. To achieve these goals we ran three tests of 15m including a sharp turn and multiple moving obstacles (other students conducting their tests on Tuesday night.

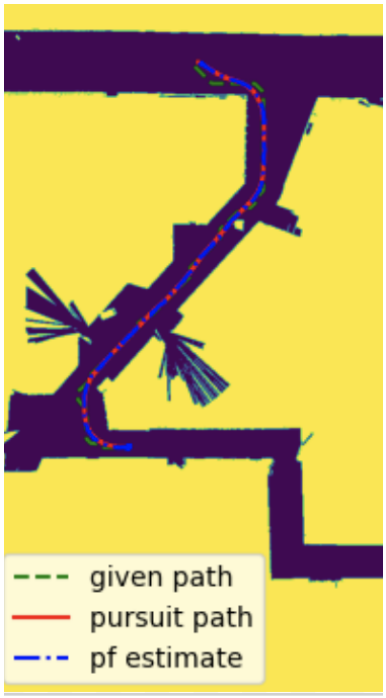


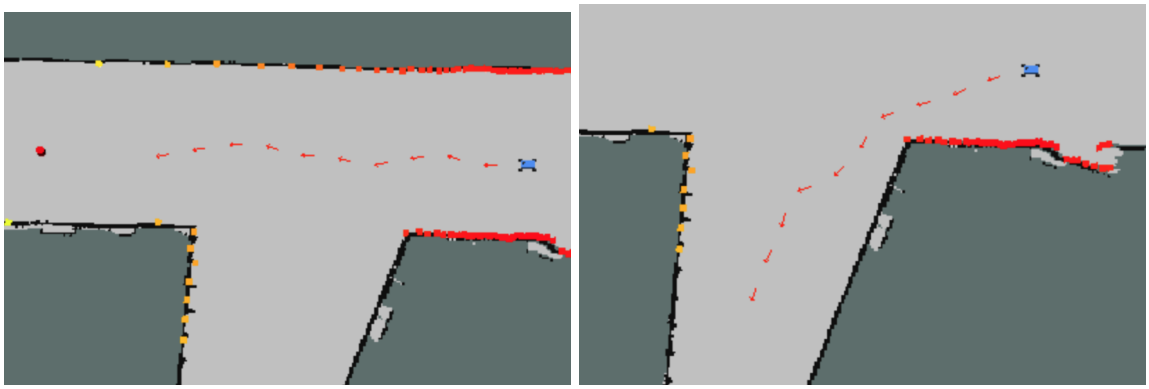

Figure 5: **Test of Pure Pursuit and A*.** Given a desired goal, our A* implementation generated a desired path, which was then followed the pure pursuit controller.

We were able to run the algorithms at 2m/s three times over this course with successful, safe, fast, and robust runs every time. A video of one of these tests can be found here:

Video Demo of 15m Hardware Test
From these tests we were able to validate that on hardware we achieved robust, repeatable, fast, and stable results. This aligns with our stated vision when we began writing the code for the lab.

## 3.2 MPPI Testing

**Savva Morozov**



(a) Straight line trajectory. (b) Trajectory with a turn.

Figure 6: **Examples of MPPI trajectories generated in simulation.** MPPI is ran with a time horizon of 10 steps, timestep of 0.2s, 10 iterations of 200 rollouts.

We have successfully implemented MPPI in simulation using Python. Examples of the kind of trajectories that we generated are presented in Fig. 6. Unfortunately (though not unexpectedly), the

Python implementation of the algorithm is nowhere near real-time: each iteration of 200 rollouts of 10 timesteps takes on average 2s to compute. Such an implementation cannot be practically executed either on the robot nor in simulation. Still, we consider these preliminary results to be promising, as the generated trajectories are effective in making progress towards the goal.

For the final stage, we intend to re-implement this algorithm in C++. Though [WDG+18] implements MPPI on the GPU, which allows them to run thousands of rollouts at each timestep, learning CUDA-programming is prohibitively time-consuming in our case. We will opt to run everything on the CPUs, make sure to parallelize the computation of individual rollouts, and possibly further simplify the vehicle's dynamics to achieve real-time performance.

# 4 Conclusion

**Tiasa Kim**

In this lab, we developed two path planners — A* and MPPI. Implementing separate planners allowed us to evaluate the robustness of the two in comparison and further optimize our racecar's performance in preparation for the final race. We tested both frameworks in simulation and successfully tested A* with pure pursuit on hardware. Overall, the results in our testing demonstrated qualitatively clear paths projected in RViz and correspondingly precise path-following by the racecar. Beyond this lab, our goal is to continue implementing MPPI (using C++ instead of Python for better speeds) and getting it to run reliably on hardware. We also intend to tune our particle filter and pure pursuit controllers to improve performance.

# 5 Appendix

## 5.1 A* Transformations

**Miranda Cai**

Note that because our discretized grid representation of the map used pixel coordinates rather than usable coordinates in the world frame, we had to transform between the two frames at the start and end of A*. Let $(p_x, p_y)$=x, y coordinates in pixel frame, $(w_x, w_y)$=x, y coordinates in world frame, $(x_i, y_i, \theta_i)$=the real world pose of the map's origin, and $r$=the discretized grid resolution. We then used the following formulas for the transformations,

$$\begin{bmatrix} w_x \\ w_y \end{bmatrix} = R_z \begin{bmatrix} p_x * r \\ p_y * r \end{bmatrix} + T$$

$$\begin{bmatrix} p_x \\ p_y \end{bmatrix} = \frac{1}{r} R_z^{-1} (\begin{bmatrix} w_x \\ w_y \end{bmatrix} - T)$$

$$R_z = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) \end{bmatrix}$$

$$T = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

Where $R_z$ is the rotation matrix about the z-axis and $T$ is the translation matrix. The first equation describes how to transform from pixel to world, and the second the reverse direction. We first had to convert the start and end poses from the world frame to the pixel frame such that indexing into our occupancy grid was possible. Then, at the end of A* after having completed calculating the path, we would have to convert all of the poses in the path from the pixel frame back into the world frame to be able to be used by pure pursuit.

## 5.2 MPPI Framework

**Savva Morozov**

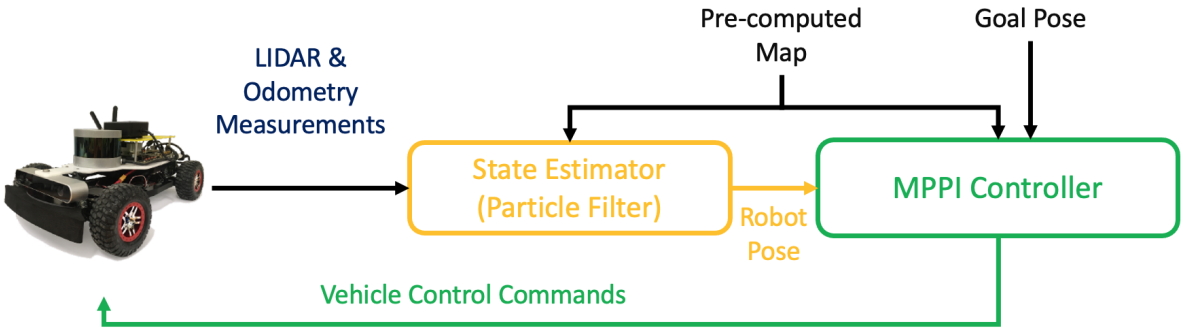



Figure 7: **High-level overview of the MPPI framework.** Sensory measurements are aggregated by a particle filter to produce robot pose estimates, which are fed to the MPPI planner. MPPI generates a local vehicle trajectory $T$ time-steps into the future, and then issues th first of these commands to the vehicle. MPPI replans these trajectories at run-time every time the vehicle location is updated.

# 6 Lessons Learned

**Jeremiah Budiman**

I learned a lot of technical aspects in this class. I had heard and learned about A* before this assignment so it was a great learning experience to actually get to implement the algorithm. I also learned that a lot of engineering is about fine-tuning variable values that might seem mindless and time-consuming but is absolutely necessary when trying to create a robust system. For example, determining what size of block would work when reducing resolution and choosing the max distance for the safety cost map required fine-tuning.
Additionally, as far as CI-M is concerned, our team did a great job communicating. Everyone knew what they were supposed to be doing at all times. Even with minimal time to prepare a presentation, all group members took the time to coordinate and ran a dry-run. We all gave each other very beneficial feedback and our presentation went well. It just goes to show how important coordination and peer review is when delivering a presentation.

**Miranda Cai**

Through this lab, I learned how to implement A* and most importantly how to come up with useful heuristics that can cut down the runtime of algorithms. For example, what sped up our A* the most from not even being able to complete to finishing in seconds was the idea of discretizing the map to a lower resolution. A 1730 x 1300 map is huge, and not efficient to check through when the size of the racecar was about 5 times the map resolution. It was really interesting also to come up with the heuristics for the A* cost function, since that was another thing completely up to us and something we tested a lot through trial and error.
In terms of CI-M, I think this was a very successful lab in which we actually seemed very comfortable with splitting up the work and collaborating asynchronously. We normally have a lot of git merge conflicts and things we don't get done by deadlines we've given ourselves, but this time around I think we've really improved on communication, and overall making sure everyone is doing ok on their part. Therefore, I learned the importance of frequent check-ins with the team on everybody's status, so that if there's any one part that is having issues instead of being blocked by this one issue, we can have multiple people try to fix it and to get by it quicker. Also almost everyone gave input on every part of this lab, which made brainstorming solutions and explaining the specific issue faster.

**Tiasa Kim**

A technical (but more theoretical / conceptual) lesson I learned in this lab was how A* works and the pros and cons of different sampling and search-based algorithms. Although I have worked with Dijkstra's and BFS/DFS in previous coursework, I had never come across A* until introduced in this

lecture / lab. One thing that fascinates me about the algorithm is the calculation of total cost of a path — not just based on the cost to get from the starting position to current position, but additionally using a heuristic of the hypothetical cost to go from each hypothetical current to the goal position. As a side note, I enjoyed this lab for the balance of structure from suggested algorithms and creative freedom to implement our planner according to our own design choices.

Regarding communication, I learned the effectiveness of including a visual storyboard and defining a design theme or value to center our report around. The two components, as advised by our CI-M Nora, helped me to approach the briefing and report with greater clarity and awareness of my message.

**Bill Kuhl**

There was a lot of debugging done technically on the pure pursuit algorithm, which I was in charge of writing. Importantly, it ended up being difficult to parse the paths when there were multiple points the lookahead distance from the car. I also could have used a better method of validating my solution. I ended up getting a solution that worked for me on sim, try it on gradescope, it breaks, then try an iteration. Problematically each test took aproximately 30 minutes to run which increased the iteration time.
Communications were better this time around. The team all had assigned roles, and we were responding dynamically and quickly while we were working together. I learned that in sructured communications it is important to have a set vision to scope the rest of our communication.

**Savva Morozov**

There was an important moment in our development when I wanted to continue working on MPPI and try "quickly" implementing it in C++, and use boost for parallelization. It was tempting and exciting to me; but instead, I opted to help my teammates with the A* implementation: redefined the cost function for the A* (by implementing the safety cost map, which I was already using in MPPI), adding visualization components, and helping test everything on hardware (there are twenty scripts that need to be run, and I've already had experience with this for MCL). We collected really good videos as a result. The lesson here was in prioritization: MPPI has been a stretch goal that I was really excited about — a low-chance high-reward idea that I loved tinkering with. Before the deadlines, it is often important to let go of this excitement and focus on what needs to be done to meet the deadline. Prioritization has been what I took out from this.
On the CI-M side, I focused on the advice Nora gave us: determining the specific values for the report (fast and safe), creating an outline for the entire report, and making good use of figures and visual information. Having been the editor of this report, gotta say, this took so much extra effort.

# References

[WDG+18] Grady Williams, Paul Drews, Brian Goldfain, James M Rehg, and Evangelos A Theodorou. Information-theoretic model predictive control: Theory and applications to autonomous driving. *IEEE Transactions on Robotics*, 34(6):1603–1622, 2018.