Software Stack for an Autonomous Vehicle

Evan Chow, Kush Hari, Praful Sigdel, Vikram Meyer

1 Introduction

We develop a software stack that enables a vehicle to drive autonomously in a simulation environment with the goal of picking up and dropping off passengers at different locations in the environment while avoiding collisions with other autonomous vehicles. There are 3 main challenges that must be addressed to enable this autonomous driving. First, our software must determine the location of the vehicle on the map using only the sensor measurements. We implement a localization module with an Extended Kalman Filter (EKF) to estimate the position of our vehicle on the map using measurements from a global positioning system (GPS) and inertial measurement unit (IMU). Second, our software should be able to estimate where other vehicles are on the map using only the camera measurements so that we can use the information to avoid collisions. We implement a perception module using multiple EKFs to track each vehicle as it enters and leaves the view of the cameras. Third, our software must be able to output vehicle controls that drive the vehicle in a collision-free manner to the pick-up and drop-off locations while ensuring traffic laws like speed limits and stop signs are obeyed. We implement a decision making module that is broken into two submodules to solve the challenge of controlling the vehicle to achieve our goals. The first submodule is the high-level route planning that finds the shortest road path from the current position to the pickup/dropoff location. The second submodule is a PID controller that outputs the target vehicle velocities and steering angles for the vehicle to follow the calculated route while adhering to collision-avoidance and road constraints. A diagram of our autonomy stack is shown in Figure 1.

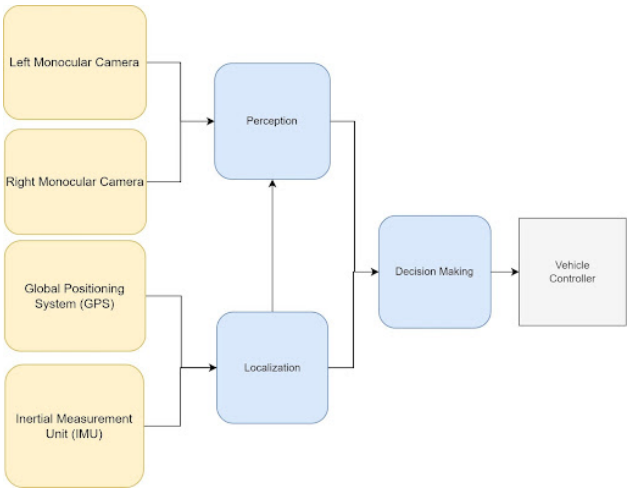

Figure 1: Autonomy Stack: The sensors are colored orange while othe modules we implemented are colored in blue. The gray vehicle controller is provided as part of the simulation.

2 Localization

2.1 Technical Background

Our localization module estimates x_k, the full-state of the ego vehicle at timestep k. The full vehicle state consists of the position, the linear velocity, orientation as a quaternion $[q_1, q_2, q_3, q_4]^T$, and angular velocity.

$$x_k = [p_1, p_2, p_3, v_1, v_2, v_3, q_1, q_2, q_3, q_4, \omega_1, \omega_2, \omega_3]^T$$

We use an Extended Kalman Filter (EKF) to estimate the vehicle's state using measurements from the GPS and IMU. Our measurement z_k alternates between being a GPS measurement and IMU measurement as they become available at different frequencies. The GPS measurements are $z_k = [p_1, p_2, \theta]^T$ where (p_1, p_2) represents the GPS sensor's position in the map frame and θ represents the angle of the GPS coordinate frame in the map frame. The IMU measurements are defined in the IMU's coordinate frame as $z_k = [v_1, v_2, v_3, \omega_1, \omega_2, \omega_3]^T$. The transformations that define the sensor's placement on the vehicle are used to convert between the sensor coordinate frames and the vehicle body coordinate frame.

Since z_k can be two types of measurements, we implement two separate measurement models $p(z_k|x_k)$ that define how we expect the measurement for each sensor to be given the vehicle's state. Our process model $p(x_k|x_{k-1})$ which gives the expected next state from the current state is implemented as a simplified version of the vehicle dynamics used in the simulation. Using the process model and measurement models, along with their jacobians, we implement an EKF to estimates $p(x_k|z_{1:k})$. The mean state estimate is then used in downstream tasks by the perception and decision making modules.

2.1.1 Extended Kalman Filter

The EKF linearizes the non-linear process and measurment models around the current state estimate and applies the Kalman Filter prediction and update steps to the mean and covariance of the state estimate to integrate sensor information to get a more accurate state estimate. We define the process model that the state evolves according to as the rigid body dynamics function provided with the simulator. We define two measurement models, one for the GPS measurements, another for the IMU measurements. Both measurement models are defined based off the code in the simulator that maps a vehicle state to a GPS and IMU measurment to be published. The jacobians of the models are calculated using Zygote.jl which performs reverse-mode automatic differentiation of the Julia functions that define the models. Q is the process noise matrix which we tune and set to a constant throughout the EKF run. R_k is the measurement noise matrix which depends on whichever sensor measurement is being used at timestep k. The following equations use the notation $\hat{x}_{n|m}$ to represent the estimate of x at time n given the observations up to and including time m.

The extended kalman filter is split into a prediction step and an update step. The prediction step uses the process model and its jacobian to propagate forward the state estimate and the covariance estimate.

$$\hat{x}_{k|k-1} = f(\hat{x}_{k-1|k-1})$$
$$P_{k|k-1} = \nabla_x f(\hat{x}_{k-1|k-1}) P_{k-1|k-1} \nabla_x f(\hat{x}_{k-1|k-1})^T + Q$$

The update step first calculates the difference between the observed measurements and the expected measurements at the current state.

$$y_k = z_k - h(\hat{x}_{k|k-1}$$
$$S_k = \nabla_x h(\hat{x}_{k|k-1}) P_{k|k-1} \nabla_x h(\hat{x}_{k|k-1})^T + R_k$$

Next, it uses computes the Kalman gain $K_k = P_{k|k-1} \nabla_x h(\hat{x}_{k|k-1})^T S_k^{-1}$ to update the predicted mean and covariance estimate for the state to account for this difference between the observed and expected measurements.

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k y_k$$
$$P_{k|k} = (I - K_k \nabla_x h(\hat{x}_{k|k-1})) P_{k|k-1}$$

2.2 Quantitative Results

The performance of the localization module was evaluated by comparing the accuracy of the estimate of the full state of the ego vehicle to the ground truth measurements being published by the simulation server. The most important features of the state estimate are the (x,y) position and orientation of the vehicle since those have the largest impact on the downstream tasks of perception and decision making. Therefore, during development, the primary metric we used to evaluate the performance of the localization module was the magnitude of the difference between the estimated and ground truth position and orientation.

State	$\|\|$**ground truth** $-$ **estimate**$\|\|$
(x,y)	4.663
(x,y,z)	31.123
orientation	0.304
(x,y) velocity	0.658
(x,y,z) velocity	0.658
angular velocity	0.162

Table 1: Average magnitude of error between ground truth and estimated state over a keyboard driving run with lots of turning and accelerating where estimates are sampled every 2.5 seconds.

Table 1 is gathered from keyboard driving that tries to mimic the behavior of the decision making module with the target velocities and a PID-like method of control to stay on the centerline of the lane. While the (x,y) position estimate error in table 1 is too large to successfully integrate localization with the existing decision making module, we can modify our keyboard driving to minimize acceleration, which is not incorporated into the process model meaning the EKF struggles to deal with it. Minimizing acceleration allows our localization estimate to maintain an error between the estimated and ground truth (x,y) position of < 1. Similarly, by maintaining a constant velocity for a few seconds after an acceleration change, the EKF is able to converge back to a more accurate position estimate. Thus, if we were to continue working on our autonomy stack, we would incorporate a minimal acceleration criteria into the decision making module as well as maintaining constant velocities for a fixed period after changing velocities, in order to allow the localization to converge to a more accurate state estimate again. Additionally, we would also incorporate the vehicle controls into the process model to better handle state estimates while accelerating.

Table 1 shows the estimated z position accumulating error over time, since there are no direct measurements of the z-position. In a more advanced simulator, this could perhaps be alleviated with an altimeter sensor. However, since the z-position can be safely assumed to be constant in the simulation and it does not play a critical role in the decision making module, we did not focus on reducing the accumulating error in

the z-position. The one thing we did to try to eliminate any initial error in the estimate of the z-position was to initial the z-position estimate to be the value of the body's z-position from the URDF and set the covariance value on the z-position to be 0.00001 to indicate our EKF should trust this initial in estimate much more than incoming measurements.

2.3 Qualitative Results

While standard GPS sensors do not measure heading, for this project, no other sensor measurements are available to directly measure the orientation of the vehicle, meaning it would otherwise have to be inferred from movement. Our initial tests with the the localization EKF where orientation is inferred from movement demonstrated difficulty with estimating the initial heading of the vehicle before any driving/movement has occurred. This initial heading is important for the decision making module to determine the correct initial driving commands. Therefore, before the simulator published direct orientation measurements, we instead calculated an initial orientation based on the driving direction of the road segment the GPS initially measured the vehicle as being on. However, despite setting the covariance on the initial estimate of the orientation to be low, the EKF would often predict the change in motion being due to an unmeasured change in orientation, when in fact the vehicle's orientation did not change. This led to an amount of error in the localization estimates that made it unusable for the decision making module since accurate orientation and position are critical for determine driving commands. There were two changes that decreased this orientation and position estimate error. The first was decreasing the process model's uncertainty from 0.01 to 0.00001, making the EKF trust the vehicle dynamic's model much more than the incoming measurements. The improvement from this fix was then compounded when the heading measurement was added to the GPS measurements, giving us direct measurements of the vehicle's orientation that prevent the estimated orientation accumulating error over time.

3 Perception

Building off the localization module, the perception module also implements an EKF (see 2.1.1 for details) for each perceived vehicle. To simplify the model, the other vehicle states are represented by their xy position, corresponding heading angle, velocity, and vehicle dimensions.

$$x_k = [p_1, p_2, \theta, v, \text{length}, \text{width}, \text{height}]^T$$

3.1 Technical Background

The perception algorithm itself is outlined in Figure 2. The two primary steps of the algorithm involve matching the current camera bounding boxes (BBs) to the previous vehicle state and performing the EKF step for each vehicle detected in frame.

It is essential to match the current measurements with the prior states properly in order to ensure the EKF will converge. This is done by first looping through all of the current measurements and finding the vehicle closest to the current BB based on the minimum norm distance between the current and previous BB. From there, if there are more BBs and previous vehicles, this means that additional vehicles have entered the frame this timestep, and some of the BBs are paired with the same vehicle. To solve this issue, n BB-vehicle pairs will be removed where n is the difference between the number of BBs and the number of previous vehicles. Finally, all unassigned BBs will be paired with a new prior to represent a new vehicle coming into frame. A challenge to this module is predicting the initial state of a vehicle when it comes into frame because there is no previous information about the vehicle. To address this challenge, the prior is assumed to be

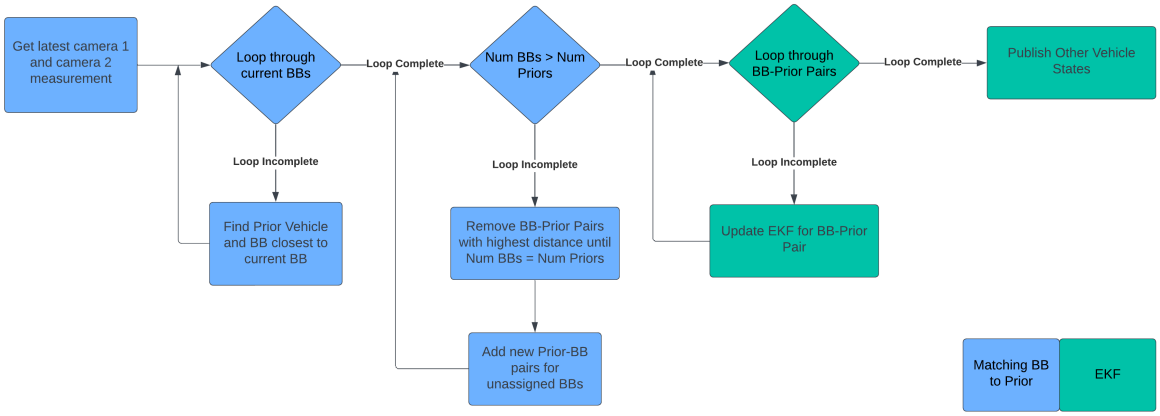

Figure 2: Perception Algorithm Flowchart

located directly in front of the vehicle driving at the same speed because this is the most probable location for a vehicle as seen in the below equation. This means that the other vehicle's theta, velocity, length, width and height can be the same as the ego vehicle, and the position was determined through the following linear interpolation. It was previously measured that a perceived vehicle's bounding box width was 11 pixels when it was 6.7 units ahead of the ego vehicle and 2 pixels when it was 40 units ahead of the vehicle. While a linear interpolation is not the most accurate way to generate an initial prediction, it is fairly reliable and an efficient calculation.

$$\text{Prior} = \begin{bmatrix} \text{ego.p1} + \text{interpolated depth} * cos(\text{ego.}\theta) \\ \text{ego.p2} + \text{interpolated depth} * sin(\text{ego.}\theta) \\ \text{ego.}\theta \\ \text{ego.v} \\ \text{ego.l} \\ \text{ego.w} \\ \text{ego.h} \end{bmatrix}$$

Once the measurements are properly matched with their priors, the EKF step can occur. As previously mentioned, each detected vehicle is running its own EKF to determine its state. The process model is defined below and is fairly similar to the basic dynamics model discussed in class.

$$f(x_{k-1}, \Delta t) = \begin{bmatrix} p1 + cos(\theta) * v * \Delta t \\ p2 + sin(\theta) * v * \Delta t \\ \theta \\ v \\ l \\ w \\ h \end{bmatrix}$$

The measurement model is a bit more complex. Each measurement is an 8x1 vector consisting of the top-left and bottom-right corner of the bounding boxes from camera 1 and camera 2. Overall, the model can be described in Figure 3. Given the vehicle state, the 3D bounding box corners can be calculated fairly easily. From there, the corners are transformed into camera frame. Then, each corner is converted to pixel space where the leftmost, rightmost, topmost, and bottommost values are saved to represent the bounding box.

To ensure these values are properly represented in the image, they are normalized with a convert-to-pixel method. While this model involves many steps, the most complicated part about it is finding its Jacobian for the EKF. Julia's automatic differentiation toolbox cannot work on this model because it uses min and max functions to find the corner pixels. Thus, each row of the Jacobian was calculated with its own modified measurement model that just solves for a single measurement dimension to prevent the use of min or max. From there, the EKF equations mentioned in the localization section are used to update the vehicle state.

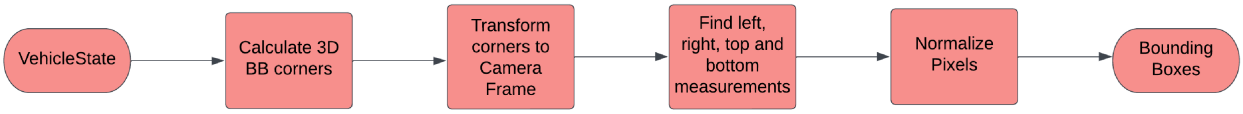

Figure 3: Measurement Model h(x) for EKF

3.2 Quantitative Results

To assess the efficacy of the perception module, the estimated vehicle states were compared to the ground truth vehicle states. Due to computational limitations, the simulation was unable to support more than 3 cars at a time for a given server. Thus, the only perception tests evaluated were the following: 1 vehicle in front of ego, 2 vehicles in front of ego, 1 vehicle leaving ego frame, 1 vehicle entering ego frame. The results are shown in the table below.

Test Case	**Normalized Error**
1 Vehicle in front of Ego	0.72
2 Vehicles in front of Ego	16 and 19
1 vehicle leaving Ego Frame	N/A
1 vehicle entering Ego Frame	6.24

Table 2: Perception Module Test Results Summary.

Overall, the EKFs converged for each model and were able to detect when a vehicle entered and exited camera frame. The "1 vehicle in front of Ego" performed extremely well, but this is most likely due to a very accurate prior, which was tuned for this exact scenario with a vehicle in front of the ego. Even though the "2 vehicles in front of Ego" converged, the errors were significantly worse. This error can most likely be mitigated by further tuning the Q and R matrices for the perception EKF to generate better models. While these initial tests are promising, perception is a complex problem with countless edge cases. Therefore, this module requires significantly more testing and tuning before it could reliably be used on an autonomous vehicle. These additional tests include testing on more vehicles, testing at intersections, and testing when vehicles move at different speeds.

3.3 Integration Plan

One of the biggest challenges of this project involved module integration. Even though each individual module is functional on its own, integrating the modules introduced new errors. For this reason, the final showcase algorithm did not have perception integrated into the main algorithm. Nonetheless, the team generated an integration plan for perception outlined in Figure 4. First, the main algorithm would loop through all perceived vehicle states and transform them into ego frame. From there, if the perceived vehicle's p2 position was not within 2 car widths of the ego vehicle, it would be skipped in the calculation since the vehicle dynamics constrain the vehicle from having significant movement along the p2 direction in a given timestep. If the vehicle is within 2 car widths of ego vehicle, it will be further transformed such that it is 1.5

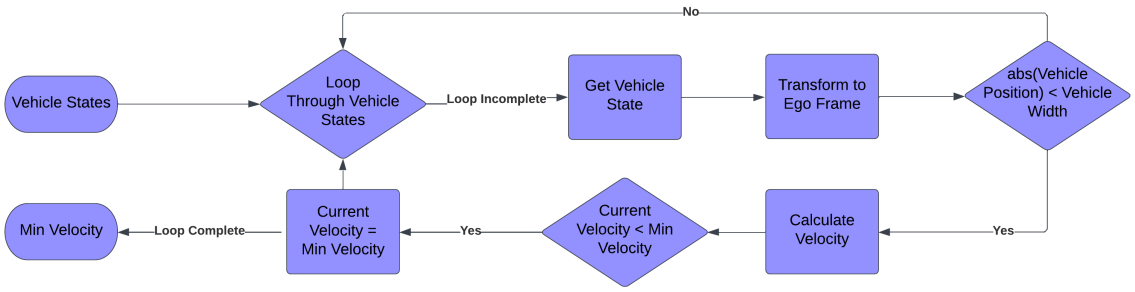

Figure 4: Perception Integration Algorithm.

car lengths closer to ego. This will help cover up some shortcomings of the perception module by making sure the ego vehicle will properly react to any vehicle at risk of collision. Finally, the same algorithm used to calculate the velocity when approaching a stop sign will be used here except the stop sign is the position of the other vehicle. This process will be repeated for every vehicle in front of ego, and the minimum velocity will be returned by the algorithm to minimize collision probability.

4 Decision Making

Our decision making module is broken into two sub-modules: high-level route planning and low-level steering. The route planning sub-module takes as input the map which is represented as a graph, a start node in the graph, and a target node in the graph. We use the localization estimate of the ego vehicle's position to calculate which node in the map graph will be the start. We implement Dijkstra's shortest-path algorithm to calculate the shortest route from the starting road segment to the target road segment.

Given the high-level route to follow, we then implement a simple steering algorithm to follow the route and take into account any vehicles the perception module detects. We split the current road segment the vehicle is on into a right half and a left half and determine which half the car is on using the localization state. Based off what side we are on, we increment or decrement the steering angle to follow the center of the lane. We maintain a constant, non-zero velocity, unless our perception module estimates the presence of a vehicle in front of us in the same road segment. In this case, we decrement our velocity to 0 and wait until the vehicle is no longer in front of us before moving incrementing our velocity back to the constant value.

4.1 Technical Background

When first loading into the simulation server, the ego vehicle is placed in a random position on the map and given a target loading zone ID. Thus, before any commands could be sent to the ego vehicle, the decision making module first determines the starting road segment of the ego vehicle and the path required to get to the target loading zone. The road segment corresponding to the current position of the ego vehicle is determined by comparing the position data from the localization module to each road segment in the map dictionary. With the boundaries of each road segment being defined with the beginning and end points of the road, each road segment is represented as a polygon. The entire map is then traversed to determine which polygon (road segment) the ego vehicle is currently in. We use the PolygonInbounds Julia package to for checking if our position point is in a road segment polygon. It employs a ray casting algorithm between a line drawn through the point of interest and the edges of the polygon.

Once the road segment position of the ego vehicle is established, Dijkstra's algorithm is used to find the shortest path in terms of road segments from the ego vehicle to the target loading zone. The road segments and their neighboring segments are represented in the map dictionary with an adjacency list, allowing us to implement Dijkstra's algorithm on top of the pre-existing data structure for storing the graph. To ensure efficiency of route planning, we use a h heap data structure to implement Dijkstra's algorithm.

Driving the ego vehicle involves sending a target velocity and steering angle to the simulator where a vehicle controller ensures the car follows the commands. To calculate the required target velocity of the vehicle for proper motion, we use a proportional controller, where the distance between the ego vehicle and another car or stop sign determines the set target velocity. The closer the ego vehicle is to another car or stop sign, the lower the target velocity is set. The steering angle is also calculated using a proportional controller; however, to keep the ego vehicle within the lane boundaries, more computations are done first. Using a similar strategy as the solution to finding the road segment the ego vehicle is on, each road segment is split into a series of 3 polygons representing the left, right, and middle of the road. The position of the ego vehicle is then used to determine which side of a specific road segment the ego vehicle is on. After determining which side of the road the ego vehicle lies, the steering angle is then adjusted with the proportional controller to ensure the vehicle stays within the bounds of the road.

4.2 Quantitative Results

To allow for stable motion of the ego vehicle, the maximum target velocity is set to 10 m/s and the proportional gain constant of the steering angle controller is set to 0.08. During testing, a maximum travel velocity of 10 m/s allowed for the ego vehicle to remain stable while the vehicle decelerated when encountering turns, stop signs, or loading zones. During testing, the proportional gain of 0.08 allowed for effective and stable steering angles for maintaining the position of the ego vehicle within lane boundaries, even while turning. We observed larger proportional gain values result in over-correcting the steering angle causing the ego vehicle to flip or spin out of control.

4.3 Qualitative Results

Both the route planning and steering sub-modules work successfully with ground truth position and orientation information. However, when switching out the ground truth state information with estimated state information from the localization module, the error in the position and orientation estimates prevents the decision making module from steering the car successfully. This happens because very accurate information is needed by the steering sub-module to determine which side of the road segment the car is on so that the proper steering corrections can be published. Since the error in the localization estimate's position is just large enough to throw off the proportional steering controller, the integrated modules have trouble following the calculated route.

Another limitation is that the decision making algorithms relied on the assumption that all roads, except turns, were completely horizontal or vertical in the map frame. To further develop the decision making module, full implementation of a PID controller may be beneficial as opposed to just a proportional controller, and new methods of determining where the ego vehicle is in relation to a current road segment should be explored. Transitioning to an optimization based decision making algorithm may also increase the ego vehicle's autonomous navigation success.

5 Contributions

Vikram implemented the localization module and setup the AV client for testing the modules in isolation and integrating them together. Evan implemented the routing and steering parts of the decision making module. Praful implemented the stop sign detection and velocity adjustment part of the decision making module. Kush implemented the perception module. Each person wrote the section of the report corresponding to their module. Vikram: 33%, Evan: 33%, Kush: 22%, Praful: 12%