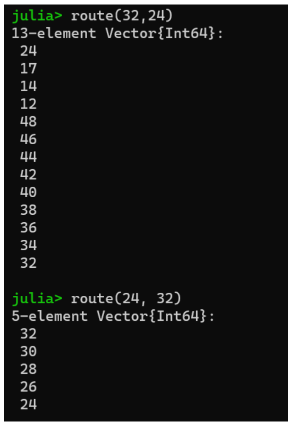
Introduction

The purpose of this project is to develop an autonomous taxi service system in a simulated environment that allows a vehicle to traverse and navigate the road in order to reach arbitrary payload locations. The virtual environment is modeled after real-world conditions, and we are given a digital model of Chevy SUV, along with two cameras, GPS, IMU, and LIDAR sensors in order to interact with the world and generate the command systems needed to control the steering and acceleration of our vehicle. In addition to the goal of transporting our vehicle to a specified payload destination, we must also respect various traffic laws and avoid collisions with other vehicles. We will receive a score based on our vehicle's ability to adeptly subscribe to these rules while reaching the destination. Proper project deployment would see the development of three realms of design architecture: Path Planning, Localization, and Perception. This code would be housed within a method called auto_client, which when called within the server will retrieve the target destination and proceed to generate vehicle commands to transport our vehicle to the loading zone at the specified location. The auto_client will do this by generating 4 threads, one dedicated to each realm of mentioned designed architecture - Path Planning, Localization, and Perception, and an additional thread that will be listening for measurement data and populating it within the relevant channels to be used by the other relevant threads. In this report we will go through the different realms that our project encompassed and elaborate on the design structure and implementation procedures and how it affected the structure and development of our project.

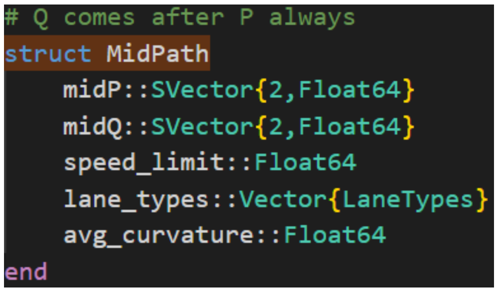
Path Finding

Given the starting RoadSegment and the destination RoadSegment, the A-star algorithm with a Manhattan distance heuristic returns a list of RoadSegments representing an efficient route for the car. From this generated list of RoadSegments, a list of MidPath objects is generated, representing a line the car should follow that is traced down the middle of all the routes.



```
julia> route(32,24)
13-element Vector{Int64}:
 24
 17
 14
 12
 48
 46
 44
 42
 40
 38
 36
 34
 32

julia> route(24, 32)
5-element Vector{Int64}:
 32
 30
 28
 26
 24
```

Given (destination RoadSegment ID, start RoadSegment ID), route uses A-star algorithm with Manhattan distance heuristic to return list of



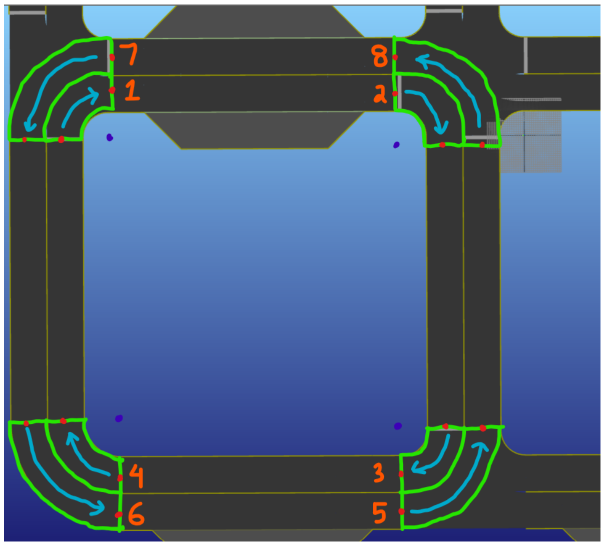
```
# Q comes after P always
struct MidPath
    midP::SVector{2,Float64}
    midQ::SVector{2,Float64}
    speed_limit::Float64
    lane_types::Vector{LaneTypes}
    avg_curvature::Float64
end
```

MidPath struct contains the two points, P and Q, at the beginning and end of the lane that are in the middle of the lane, respectively. For straight lanes, curvature will be infinite while curved lanes will

RoadSegment IDs representing path. have 1/radius curvature.

This traced path, known as MidPath in the code, was calculated by classifying RoadSegments into two categories: straight and curved. RoadSegments with infinite curvature are straight so the path was constructed by creating a line connecting the midpoints of the two LaneBoundary objects' pt_a and pt_b attributes. RoadSegments with finite curvature are quarter circles, which fall under the 8 different categories shown below, where $P(x_1, y_1)$ and $Q(x_2, y_2)$ represent the earlier and later midpoints in the MidPath, respectively.

	$x_{P, Q}$	$y_{P, Q}$	Curvature	Center
Case #1	$x_2 > x_1$	$y_2 > y_1$	negative	(x_2, y_1)
Case #2	$x_2 > x_1$	$y_2 < y_1$	negative	(x_1, y_2)
Case #3	$x_2 < x_1$	$y_2 < y_1$	negative	(x_2, y_1)
Case #4	$x_2 < x_1$	$y_2 > y_1$	negative	(x_1, y_2)
Case #5	$x_2 > x_1$	$y_2 > y_1$	positive	(x_1, y_2)
Case #6	$x_2 > x_1$	$y_2 < y_1$	positive	(x_2, y_1)
Case #7	$x_2 < x_1$	$y_2 < y_1$	positive	(x_1, y_2)
Case #8	$x_2 < x_1$	$y_2 > y_1$	positive	(x_2, y_1)

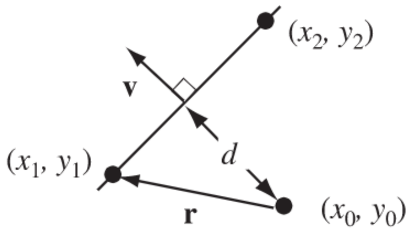


8 Different Cases of Curved Paths

When the car calculates its error during PID control, it will check whether its path is curved and if it is, the error is the distance of the car from the curved path which is calculated as the distance from the center of the circle minus the radius, which is 1/curvature. The error of the car on a straight path is simply its distance from the line. The error will be explained further in the PID control section.

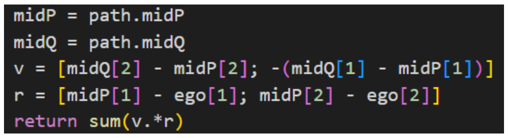
PID Control

A PID controller was used to control the movement of the car. As alluded to before, the path finding algorithm returned a list of MidPaths, which represent lines traced down the middle of the roads. With this list of MidPaths, the car's location was calculated and based on its quaternion orientation data, the point ⅔ the car length in front of the car was calculated. The proportional or cross track error (CTE) was calculated as the distance of this point from the current MidPath. At the start of the simulation, the current MidPath was calculated by looking at GPS location and iterating through all the lanes to see which lane contained the car's GPS location. Throughout the simulation, the current MidPath is changed when the point ⅔ the car length in front of the car is within 11 units of the next MidPath. At every iteration of the control loop, the car's location ⅔ car_length in front was calculated and if the current MidPath was a straight path, the distance from the point to this line was calculated as the CTE using the leftmost formula below. If the path was curved, the CTE function would find the center of the circle by identifying which of the 8 curved path cases the curved path fell under. It then calculated the center of the circle and used the second formula below to calculate the CTE.



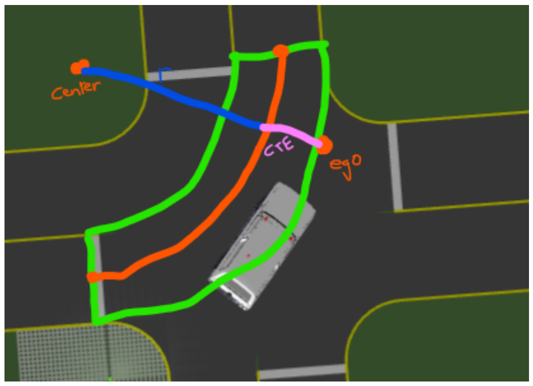
$$d = |\hat{\mathbf{v}} \cdot \mathbf{r}| = \frac{|(x_2 - x_1)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_1)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}.$$

Distance from point to line is $|v \cdot r|$ where v is the normal vector to the line and r is the vector from a point on the line to the point in question

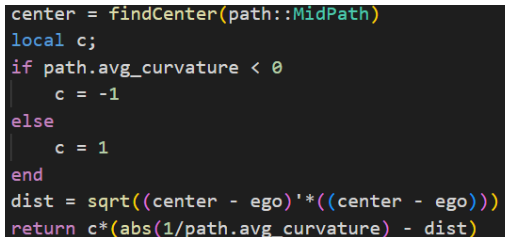


```
midP = path.midP
midQ = path.midQ
v = [midQ[2] - midP[2]; -(midQ[1] - midP[1])]
r = [midP[1] - ego[1]; midP[2] - ego[2]]
return sum(v.*r)
```

Implementation of formula for calculating CTE of straight MidPath



$$d = c(|ego - center| - radius)$$
where c=-1 for negative curvature and c=+1 for positive curvature



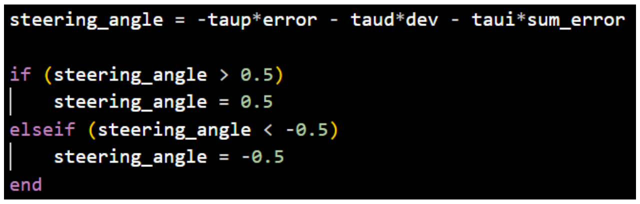
```
center = findCenter(path::MidPath)
local c;
if path.avg_curvature < 0
    c = -1
else
    c = 1
end
dist = sqrt((center - ego)'*((center - ego)))
return c*(abs(1/path.avg_curvature) - dist)
```

Implementation of formula for calculating CTE of curved MidPath

The algorithm for PID control consists of the weighted summation of three aggregated errors: Proportional Error, Integral Error, and Derivative Error respectively. The figures to the right demonstrate the basic outline of the algorithm and its effect on the motion plan of the vehicle. Proportional Error represents the error associated with the vehicle's distance from the MidPath line and corrects the steering angle to get closer to it. This type of course correction results in a sinusoidal motion curve as the vehicle oscillates between correcting towards the MidPath line then overcorrecting and recorrecting the error. In order to streamline the motion and eliminate the oscillations, Derivative Error was added. Derivative Error represents the difference between the current Proportional Error and the previous error. Its effect on motion planning is modeled by the PD control graph. After implementation it dampens how much the steering overcorrects itself and allows for tighter movements. There was a third error called Integral Error that represents the error of inherent bias in our systems steering

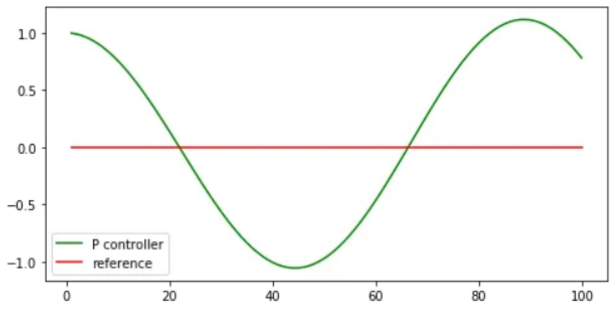
$$u(t) = K_p e(t) + K_i \int e(t)dt + K_p \frac{de}{dt}$$

$u(t)$ = PID control variable
K_p = proportional gain
$e(t)$ = error value
K_i = integral gain
de = change in error value
dt = change in time

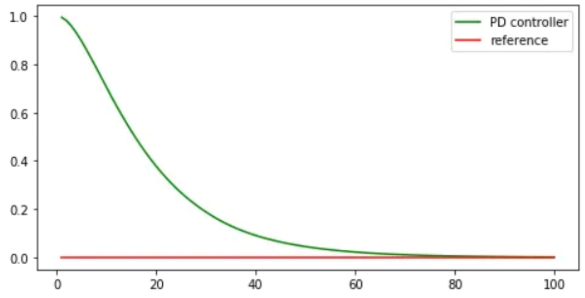


```
steering_angle = -taup*error - taud*dev - taui*sum_error

if (steering_angle > 0.5)
    steering_angle = 0.5
elseif (steering_angle < -0.5)
    steering_angle = -0.5
end
```

controls, but it was not significant in affecting the steering of our vehicle so its weight was set to 0. For the weights of Proportional and Derivative Error, they were determined via trial and error by observing the trajectory plan of the car and seeing how each error was contributing to the angle of the steering as it was pushed to the vehicle. Through this process, weights for the errors were found that streamlined the motion of the vehicle as much as possible and allowed for the greater range of motions that the vehicle needs in order to stay within the lane boundaries while moving and turning corners. One unique observation when determining the weights was that the Derivative Error was being trivialized by the sheer frequency of data packets from our measurements. It was unable to correctly mollify overcorrections because it received GPS coordinates so quickly that there was no distinction between the current position and previous and so a frequency dampener algorithm for the derivative error needed to be constructed in order to properly implement the Derivative Error. Additionally, the steering angle corrections would sometimes become too large and dramatically alter the trajectory of the vehicle so steering angle had to be limited in its ability to change in any arbitrary control loop. This value limit was calibrated to be 0.5 in order to ensure that vehicle was still able to make turns while also eliminating absurd steering angle corrections made by the PID control algorithm.



P Control

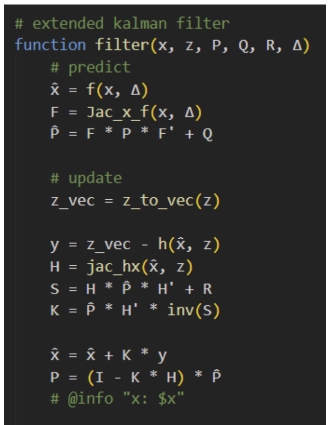


PD Control

Localization

The main goal of the localization segment of this autonomous vehicle project was to estimate the current state of the vehicle based on incoming GPS and IMU measurements from the simulated sensors, then packaging all of the information into a state with 13 variables to hold the position, orientation, angular velocity, and linear velocity. To accomplish this, we first take all of the raw GPS and IMU data and sort it by timestamp. We then take the earliest measurement and process it. We have chosen to employ the Extended Kalman Filter (EKF) methodology over the traditional Kalman Filter (KF) to accomplish this. This decision stems from the inherent nonlinearities present in the system dynamics and measurement models associated with the vehicle's motion and sensor data processing. Namely, the presence of quaternions, along with the use of trigonometric and transformation functions to process coordinates, indicates that the system requires a methodology that can support nonlinearities. The KF is more suitable for linear systems; thus, utilizing an EKF allows for the ability to handle these nonlinearities by linearizing the system dynamics and measurement models around the current state estimate.

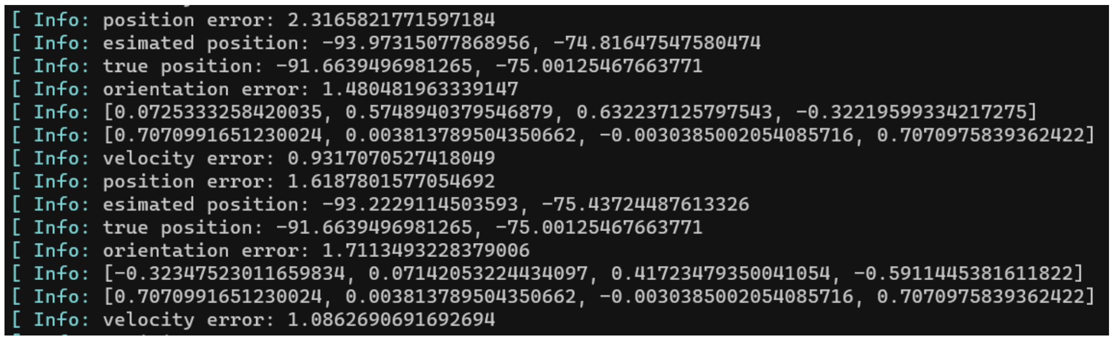
The EKF implementation used in this segment consists of two main steps: prediction and update. In the prediction step, the state is propagated forward in time using the process model f(x, Δ) and the Jacobian Jac_x_f(x, Δ). The state covariance matrix P is also updated in this step, incorporating the process noise Q. The prediction step helps in estimating the vehicle's state based on its previous state and the elapsed time. In the update step, the measurement model h(x, z) and


```
# extended kalman filter
function filter(x, z, P, Q, R, Δ)
    # predict
    x̂ = f(x, Δ)
    F = Jac_x_f(x, Δ)
    P̂ = F * P * F' + Q

    # update
    z_vec = z_to_vec(z)

    y = z_vec - h(x̂, z)
    H = jac_hx(x̂, z)
    S = H * P̂ * H' + R
    K = P̂ * H' * inv(S)

    x̂ = x̂ + K * y
    P = (I - K * H) * P̂
    # @info "x: $x"
```

its Jacobian jac_hx(x, z) are used. The measurement model relates the state variables to the measurements received from the sensors (GPS and IMU). The Jacobian of the measurement model represents the linearized version of the measurement model with respect to the state. The update step incorporates the sensor measurements into the state estimate, adjusting the state and covariance matrix accordingly. This step helps in correcting the state estimate based on the actual measurements from the sensors. Thus, processing the raw information from the GPS and IMU through the EKF allows us a good guess about the vehicle's current orientation, position, velocity, and angular velocity.

Localization Testing

To test the localization channel, we ran another async function as part of our client to compare the current estimate of the vehicle's state with the actual vehicle state from the ground truth channel. This function is run every couple of seconds to avoid filling the entire terminal window immediately. The error is recorded as the norm(estimated - true). Below is an example output from running this async function:



```
[ Info: position error: 2.3165821771597184
[ Info: esimated position: -93.97315077868956, -74.81647547580474
[ Info: true position: -91.6639496981265, -75.00125467663771
[ Info: orientation error: 1.480481963339147
[ Info: [0.0725333258420035, 0.5748940379546879, 0.632237125797543, -0.32219599334217275]
[ Info: [0.7070991651230024, 0.003813789504350662, -0.0030385002054085716, 0.7070975839362422]
[ Info: velocity error: 0.9317070527418049
[ Info: position error: 1.6187801577054692
[ Info: esimated position: -93.2229114503593, -75.43724487613326
[ Info: true position: -91.6639496981265, -75.00125467663771
[ Info: orientation error: 1.7113493228379006
[ Info: [-0.32347523011659834, 0.07142053224434097, 0.41723479350041054, -0.5911445381611822]
[ Info: [0.7070991651230024, 0.003813789504350662, -0.0030385002054085716, 0.7070975839362422]
[ Info: velocity error: 1.0862690691692694
```

From the 2 initial runs of the testing function shown in the picture above, we see that the position error was around 2.32, the orientation error was around 1.48, and the velocity error was around 1.086. The position and velocity values are decently on par with the expected ground truth results, but we see that there was substantially higher error in the orientation. This error might have been due to incorrect tuning of the covariance matrix P or the process noise matrix Q as we were unable to spend adequate time tuning and testing.

Perception

For perception, our goal was to estimate the state of other vehicles in the environment around our ego vehicle based on measurements given by the two cameras on our ego vehicle. The state of the other vehicles on the road that we were trying to estimate included position, heading, velocity, length, width, and height. The length, width, and height remained constant for each vehicle. The position, heading, and velocity of the other vehicles changed over time. Whenever another vehicle came into view of one of the cameras, we received a camera measurement. Each measurement consisted of the time it was received, the id of the camera the measurement came from, the focal length of the camera, the length of each pixel, the image width, the image height, and the bounding boxes.

Extended Kalman Filter

Although we received the bounding boxes directly from the camera measurements, we didn't know which bounding box corresponded to which vehicle. Our job was to run the camera measurements through an Extended Kalman Filter and generate a more accurate bounding box for each vehicle. In order to do this, we first had to come up with an initial state for a vehicle when it first came into camera view.

Since other cars were somewhere in front of us, the initial position of another vehicle was our current position plus a couple of units. The velocity and heading were arbitrary values. The length, width, and height were given. Then we had to come up with our process model, which included predicting the current state of the other vehicle given its previous state. We were able to use the same process model that we used from localization. We then had to come up with our measurement model, which predicted the current measurements of the other vehicles based on their predicted current state. This included generating our own bounding box for the other vehicles. Each vehicle is represented as a rectangle with 8 corners. After getting those 8 corners in world frame, we projected these corners into camera frame. We then found the pixel values of each of these corners projected into camera frame. We were able to determine which pixels represented the top-most, bottom-most, left-most, and right-most extents of our bounding boxes. We then got the coordinates for the top left and bottom right corner of the bounding box. Once the Extended Kalman Filter was complete for that time step and the bounding box had been generated, we placed the estimated state of the other vehicle onto the perception channel. The estimated states of the other vehicles were then passed to the path planning team to use in order to avoid collisions.

Quantitative/Qualitative Takeaways

When first developing the path finding algorithm, the MidPath along the curved roads was calculated as a straight line connecting the two points in the middle of the lane. We originally thought this would be sufficient for making turns; however, when testing, the car would veer off the road. In fact, increasing velocity beyond 1 m/s would always make the car go off track infinitely in the line's direction. After adjusting the PID control code to recognize curved paths and impose a quarter circle path over the curved path, the car was able to stay on track much more often and performance improved significantly, as the car could travel at max_speed on any curved path, while staying mostly in the lane.

We were planning to test our estimated states of other vehicles against the ground truth of those other vehicles. Since we weren't able to correctly generate the bounding boxes, we weren't able to successfully test the perception module. However, we were able to test the localization state channel locally - our finished project ended up working to a certain extent, but due to the slight error in all fields of the current state we were unable to achieve consistency. Presumably this was mainly due to the error in orientation, as the finished project was able to reach its goal on some occasions. When testing on the server hosted during the Immersion Showcase, the vehicle was able to achieve its goal with ground truths. However, without ground truths, the vehicle lost track of its current state quickly due to issues in syncing.

Team Contributions

For our project there were approximately three groups - Path Planning, Localization and Perception. Kev Koppa and Ryan Taylor were in this group with Kev handling the major algorithms for path planning and Ryan calibrating the PID control and organizing the design structure to incorporate Localization and Perception code after completion. Myles Robinson was on the Perception team and handled processing camera data and retrieving information about the external environment. Manas Malla was on the Localization team and handled processing of GPS and IMU data in order to determine where the car was in the environment and its current state. All members contributed equally to this project, so each member was responsible for approximately 25% of the code created.