# Path Planning Project

## Readme Explanation

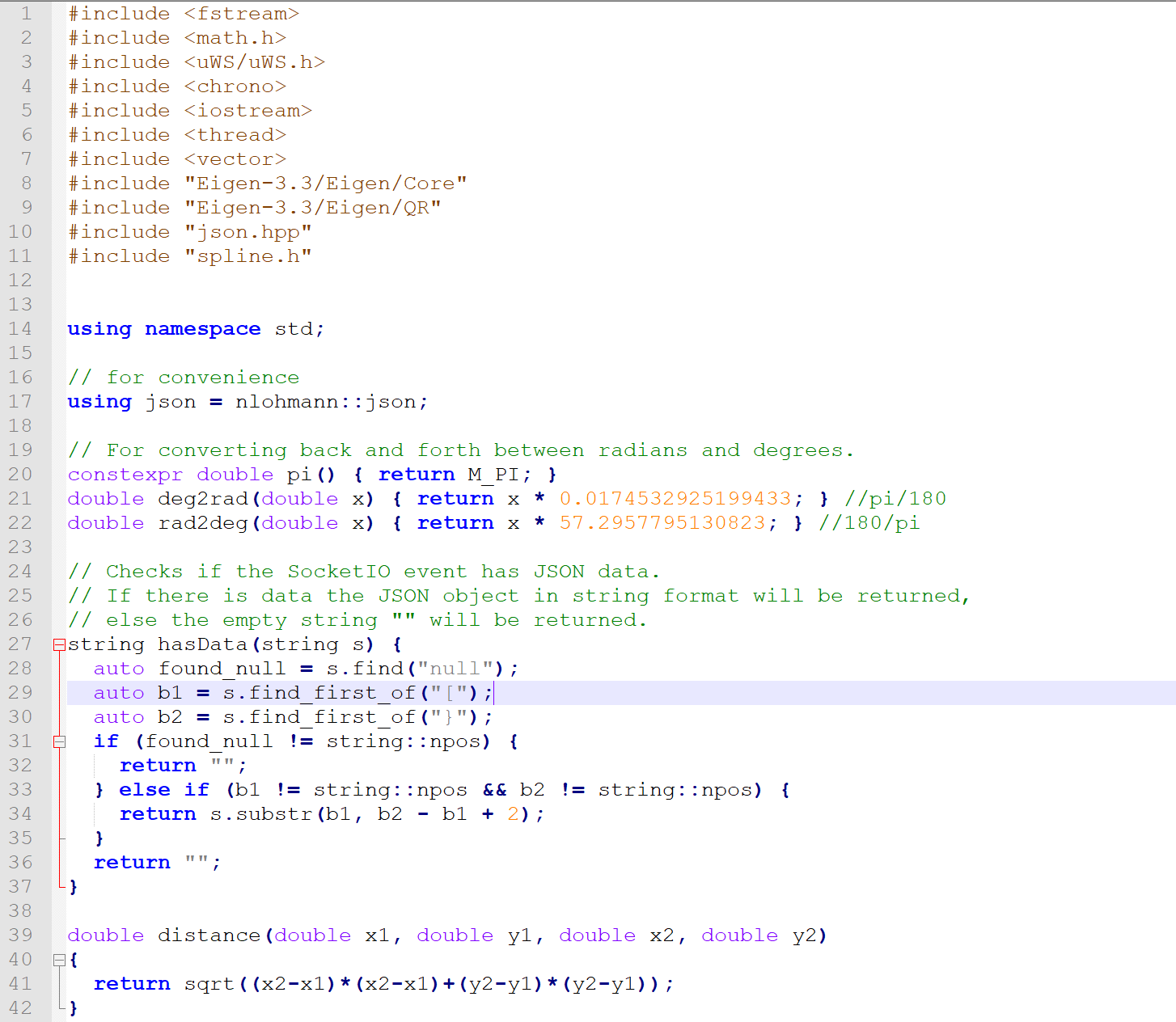
You’re reading it now!

## Project Explanation

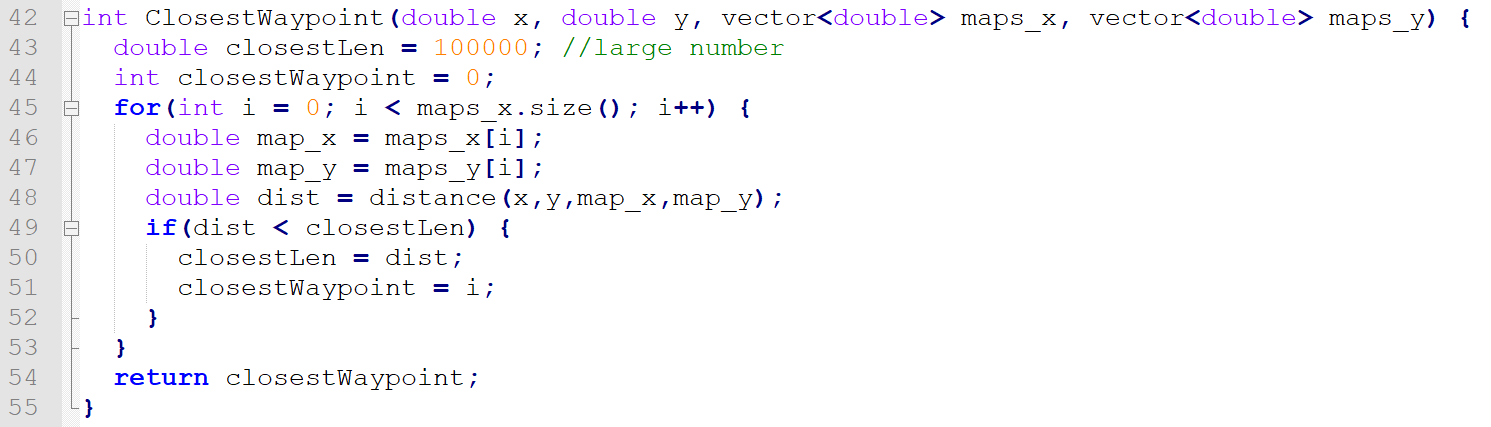
This project is to show a simple algorithm going around a track in a simulated world. Fused sensor inputs are taken in to the algorithm and used to define what lanes are safe, what reasonable speeds of the vehicle is, etc.

## Code documentation

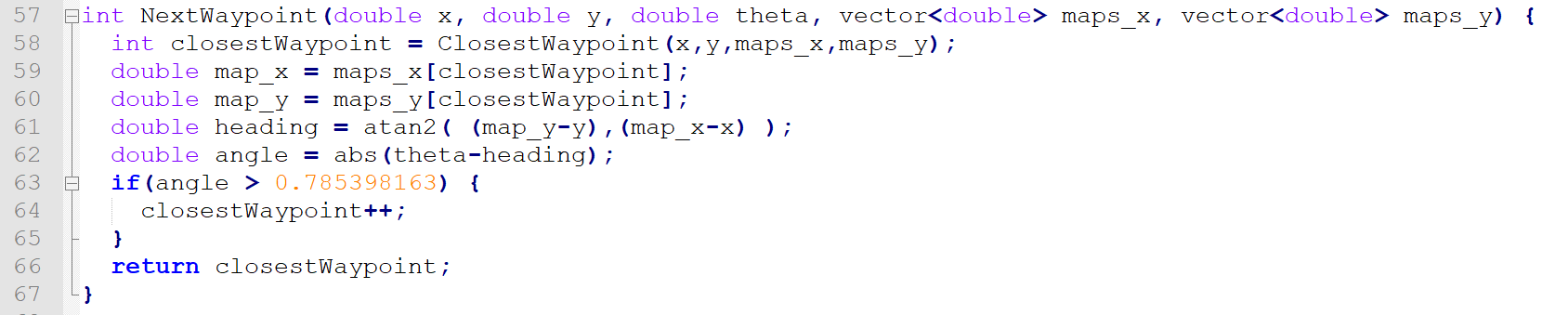
### Setup

Lines 1 through 42 are fairly straightforward, defining libraries, namespaces, and the json function. Lines 21 and 22 convert between degrees and radians. Lines 27 through 37 are a check to determine whether a string has data, and the function distance is defined.

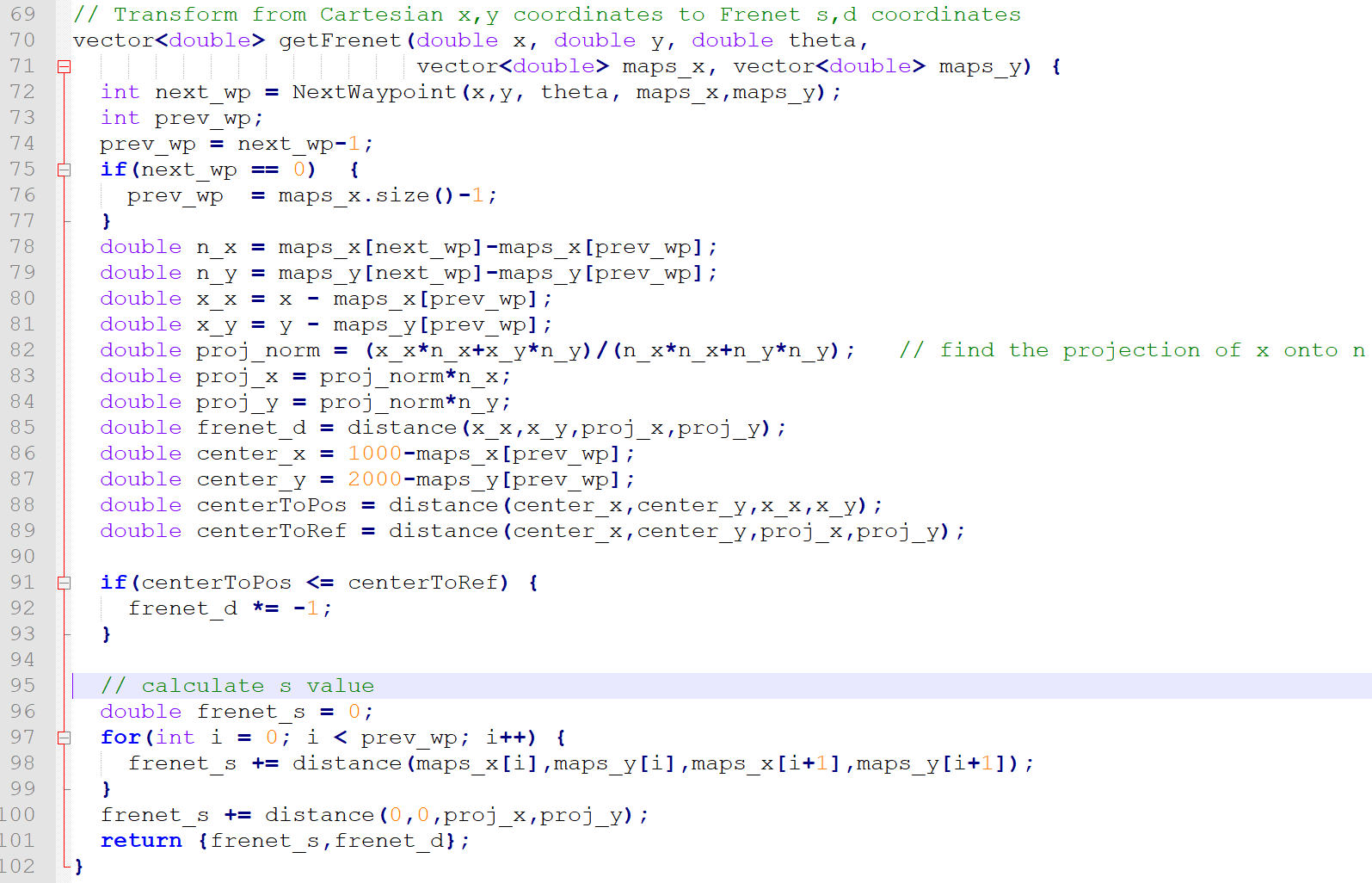
Lines 42 through 55 show a function ‘ClosestWaypoint’ which is fairly self explanatory, but for the sake of being verbose I’ll say it loops through the waypoints on the map and finds the waypoint that is closest to the x and y input into the function.



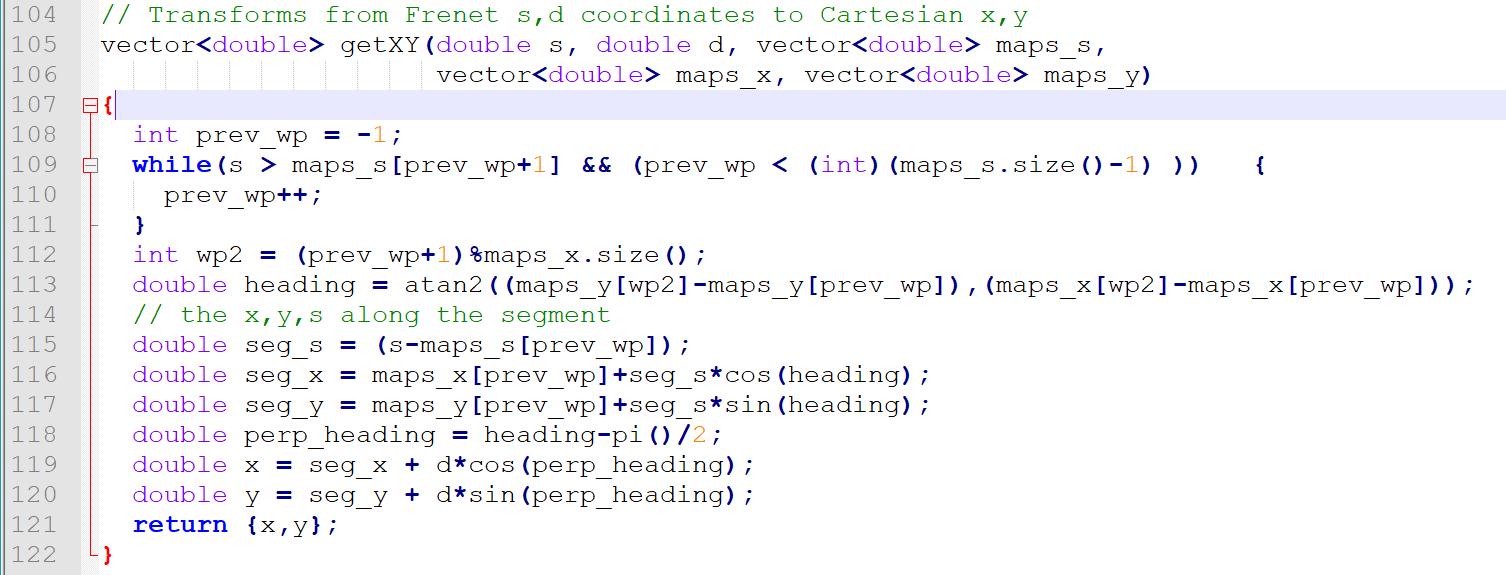
Lines 57 through 67 show the function ‘NextWaypoint’ which calls closest waypoint and then searches for the waypoint following that one.



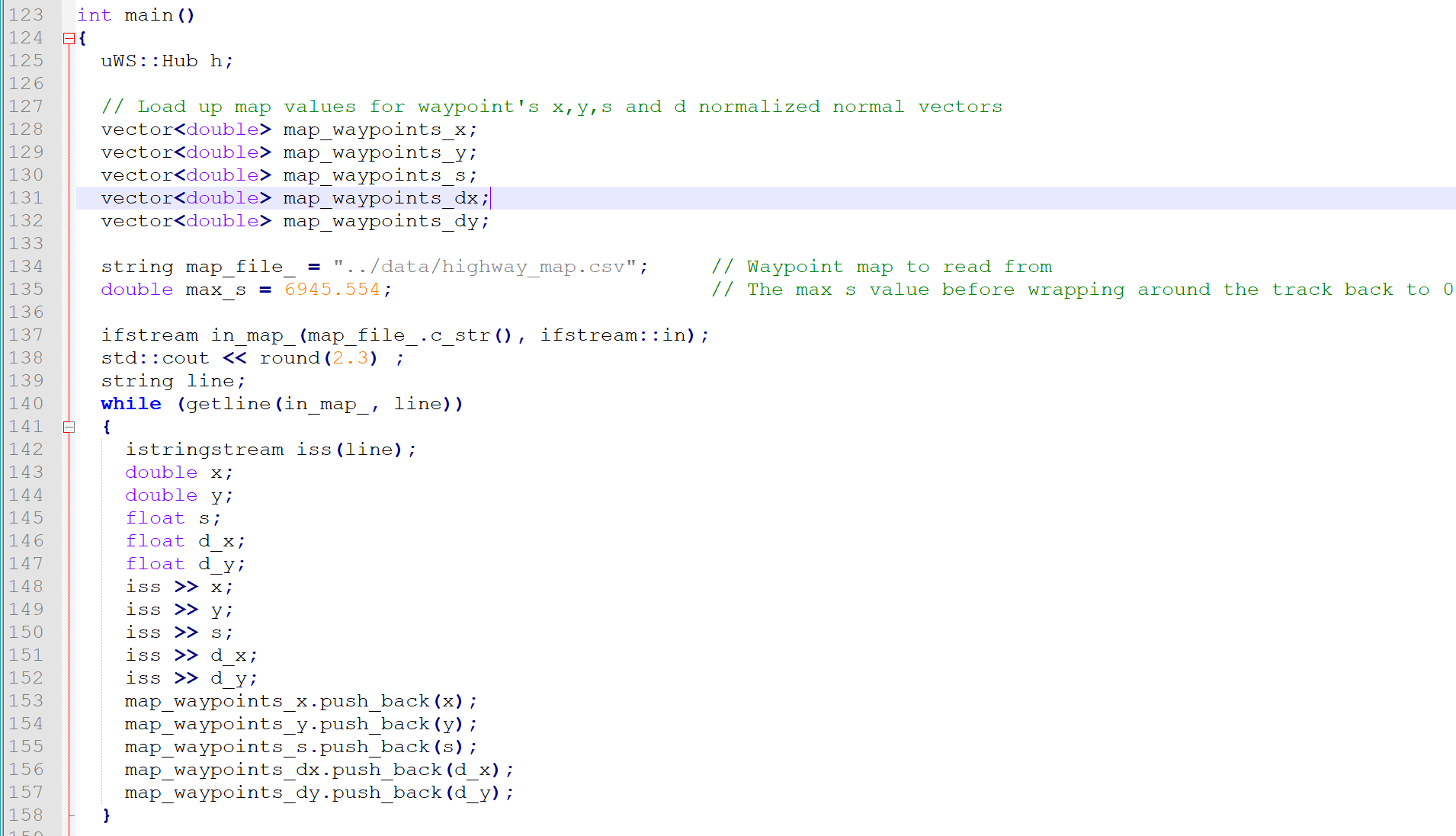
Lines 70 through 102 are the function getFrenet, which returns frenet s and d from inputs of x, y, theta, maps\_x and maps\_y.



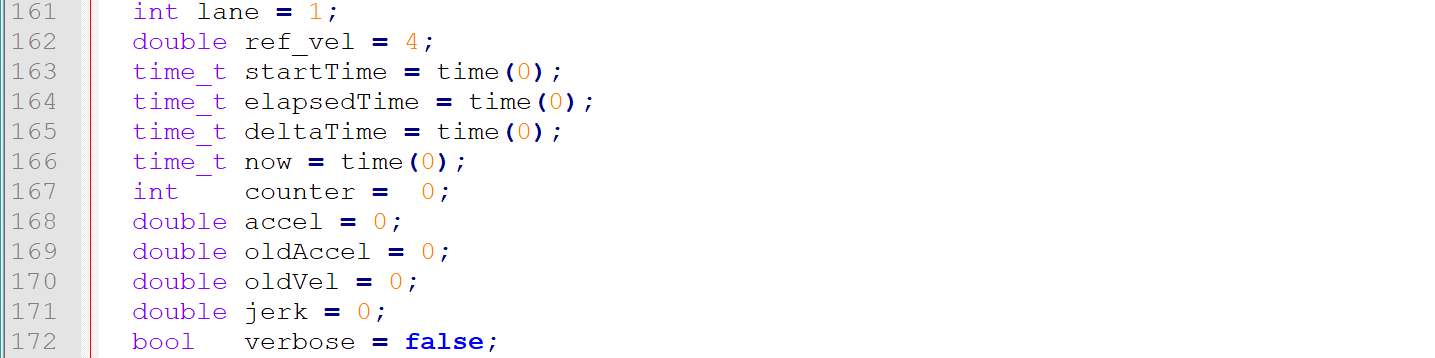
Lines 105 to 121 transform a set of coordinates from the Frenet s and d (distance along the path and distance lateral to the path) into the x and y Cartesian coordinates. It is the inverted function from the prior function.



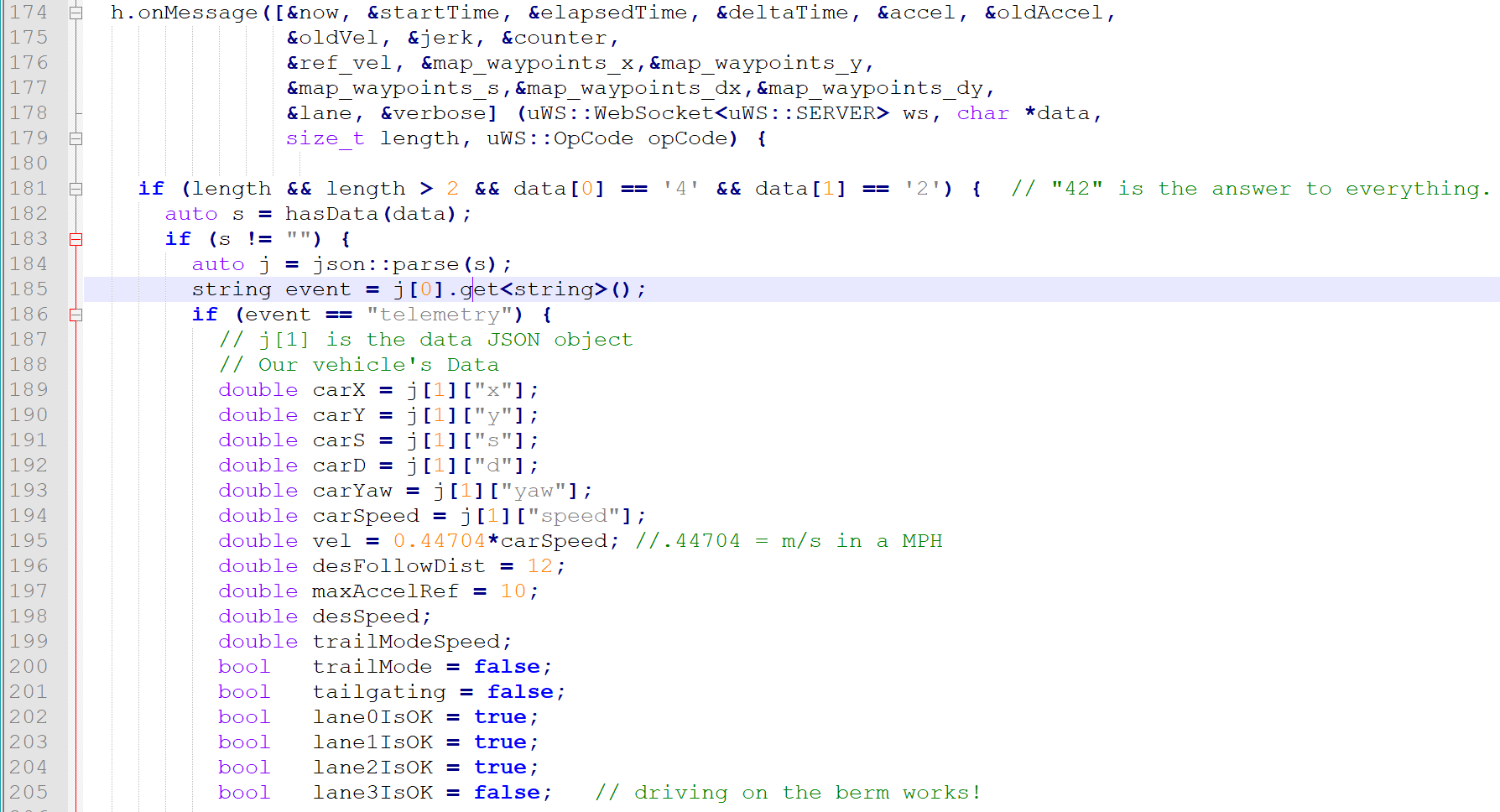
Lines 123 to 157 start setting up the main function, loading up the map, and starting to define functions and waypoints.



Lines 161 to 172 define some variables – lane, reference velocity, some times (useful to calculate acceleration and jerk, but less useful than one might think - the calculation I was doing for accel and jerk is not perfectly the same as the one done by the simulator).

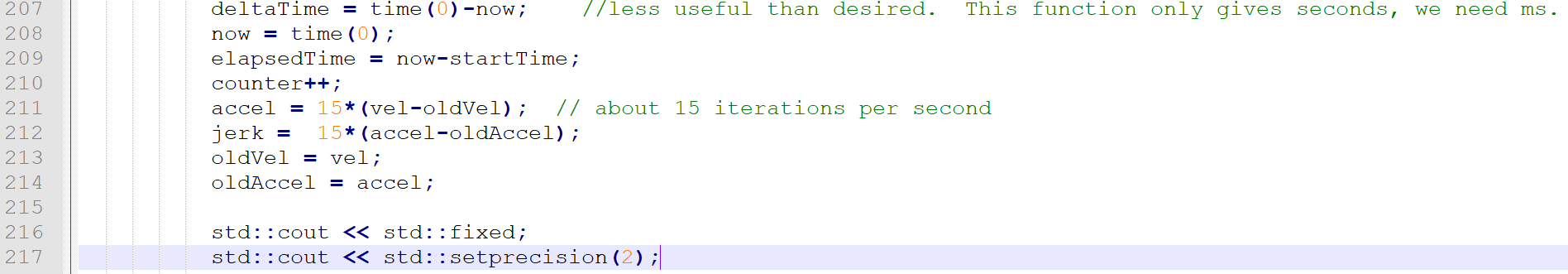


Lines 174 to 205 define the lambda function and variables passed to the lambda function, read in the first data message, and define some of the variables that we’ll be using, like the car’s X, Y , S, and D parameters, Yaw, speed, velocity, etc.

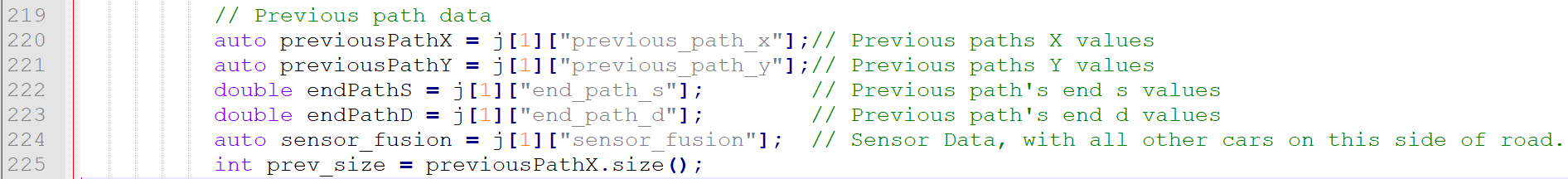


Please keep in mind that the car driving on the berm works!

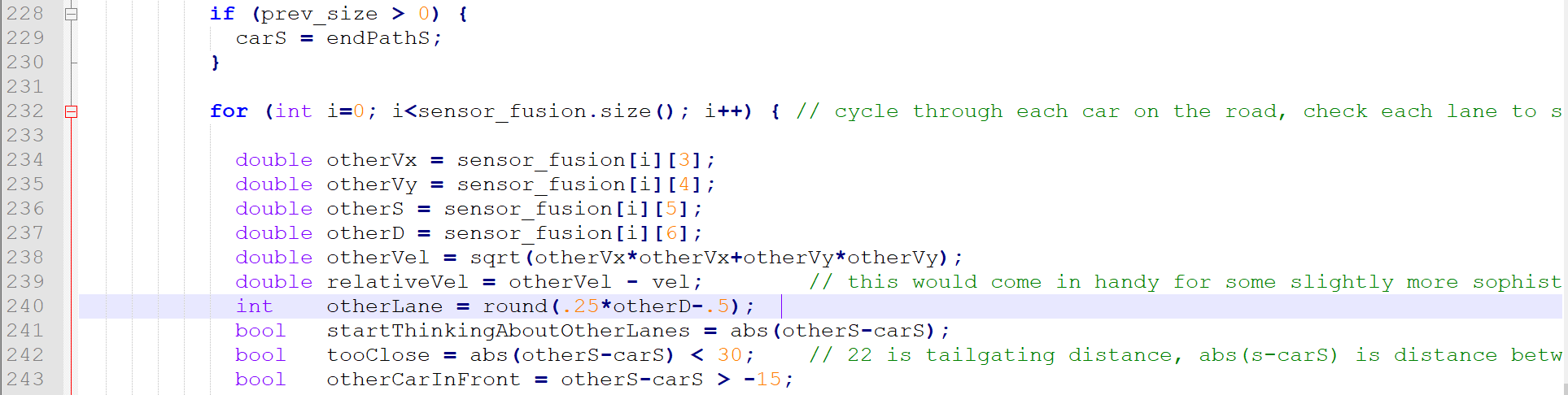
Lines 207 to 217 start calculating my own calculations fo jerk and acceleration. These are less useful than originally desired in part because the time function only gives information in seconds, which is not useful. Other functions could be used but even ignoring these problems the calculations weren’t lining up with the simulator – various scale values had to be used – so other techniques were used to avoid having problems with acceleration or velocity.



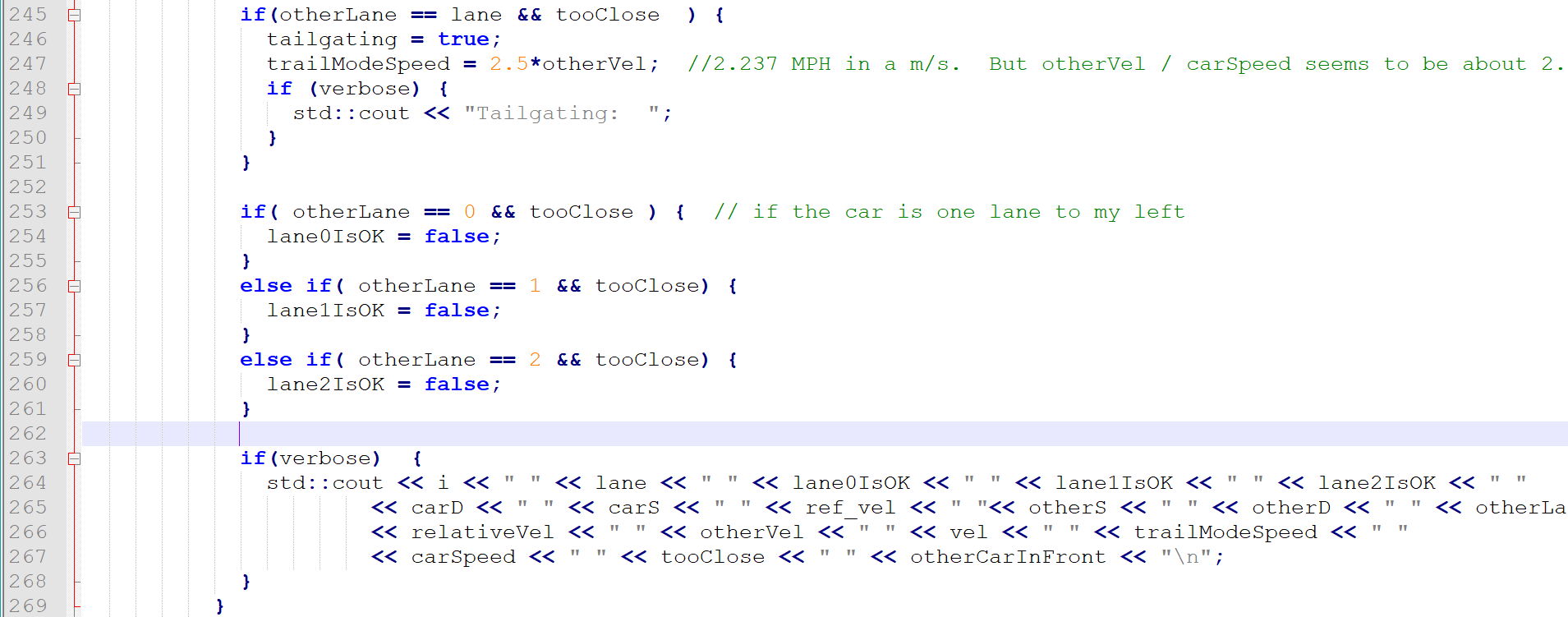
Lines 220 through 225 define some previous path data…



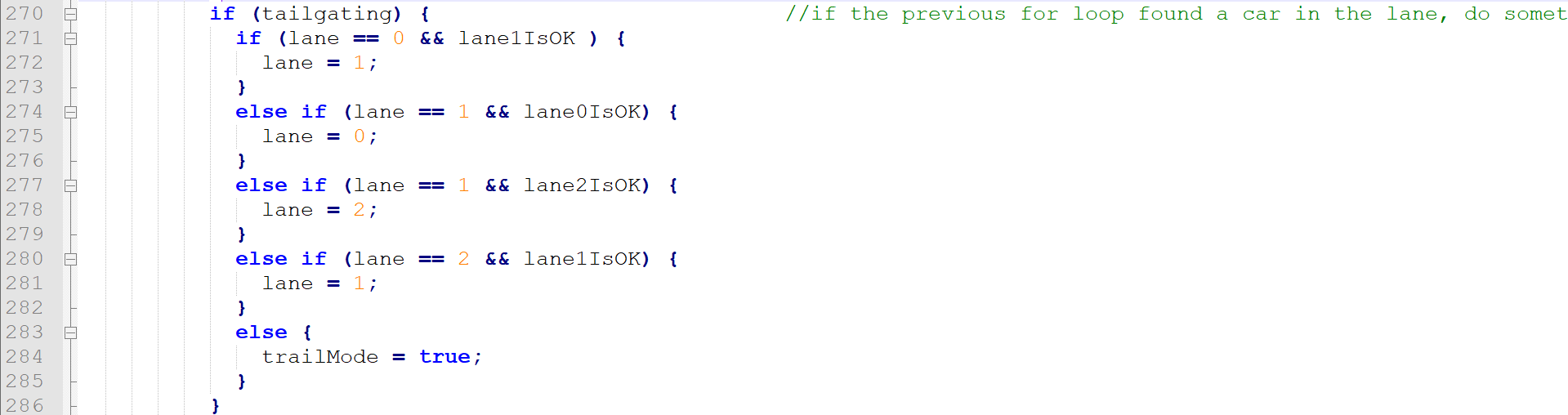
Lines 228 to 243 define carS, and then starts to loop through each sensor’s data of vehicle detection and starts to define the other cars variables – like x velocity, y velocity, s, d, etc.



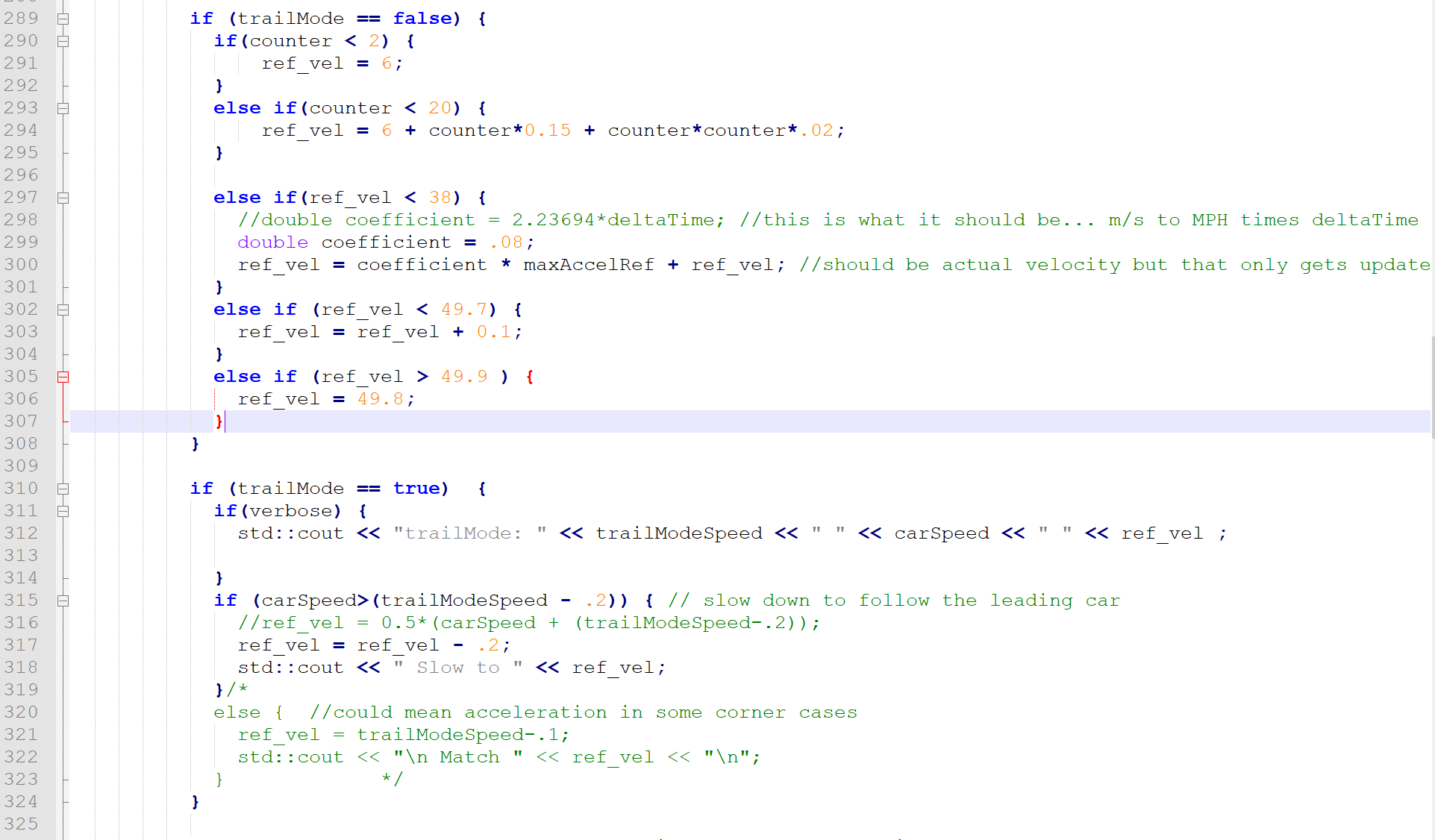
Lines 245 to 268 go through what happens if a car that we’re following is too close – first we set the flag that we’re tailgating, then we set whatever lane that vehicle is in as off limits. Finally we output some debugging data if the verbose flag is set to true.



Liens 270 to 286 ask, first, if we’re tailgating, and if we are, then goes through some simple logic to check what other lanes might be possible, and if any lanes are possible, move into one of them.



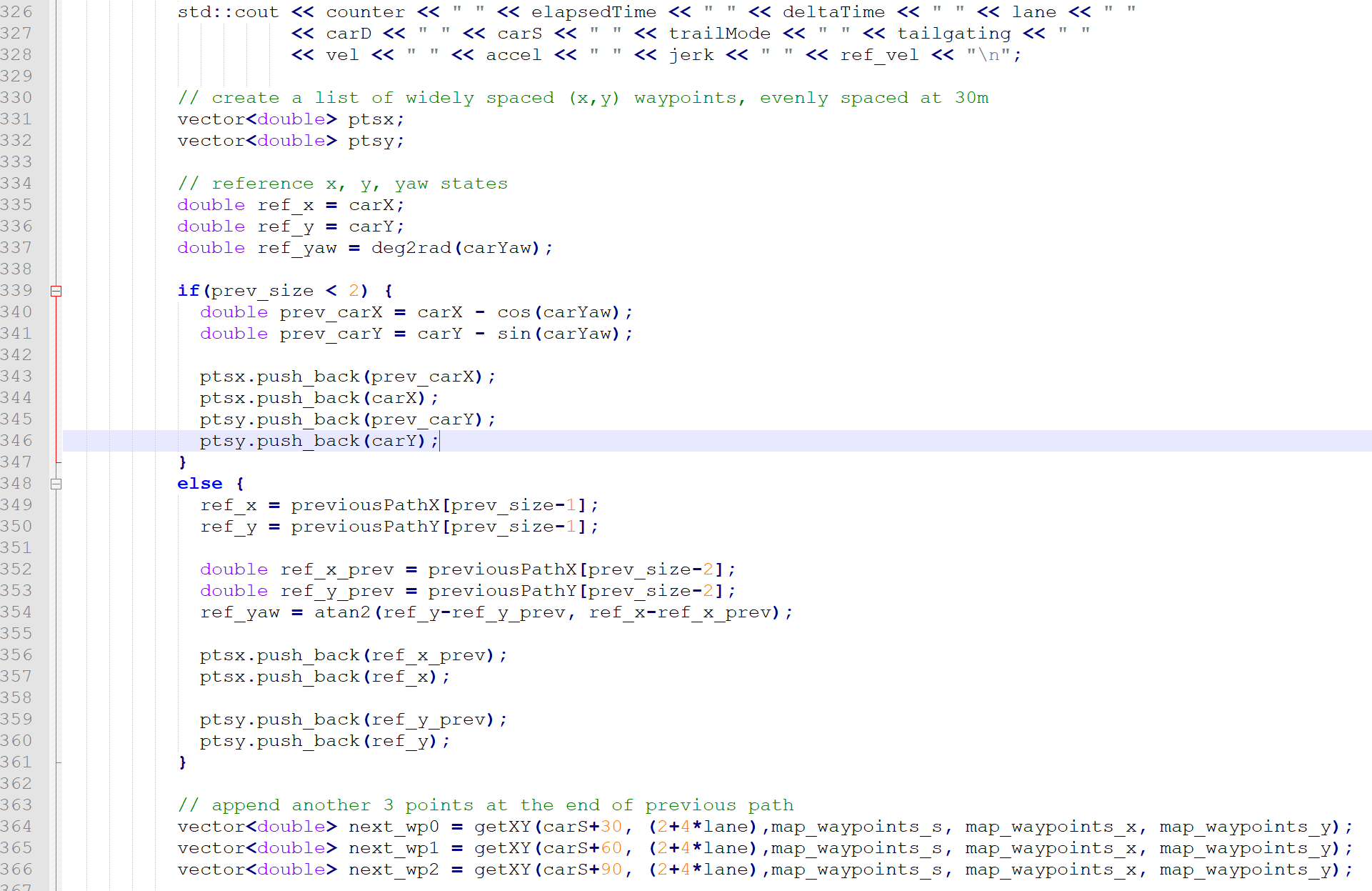
Lines 289 to 324 set the speed of the vehicle. In the first few iterations of the simulation, we sest the reference velocity to 6 and then add in some other terms. After 20 iterations, we go to a max acceleration scheme. As we get closer to the speed limit we slow down. Separately, if trailmode is active, we slow down.



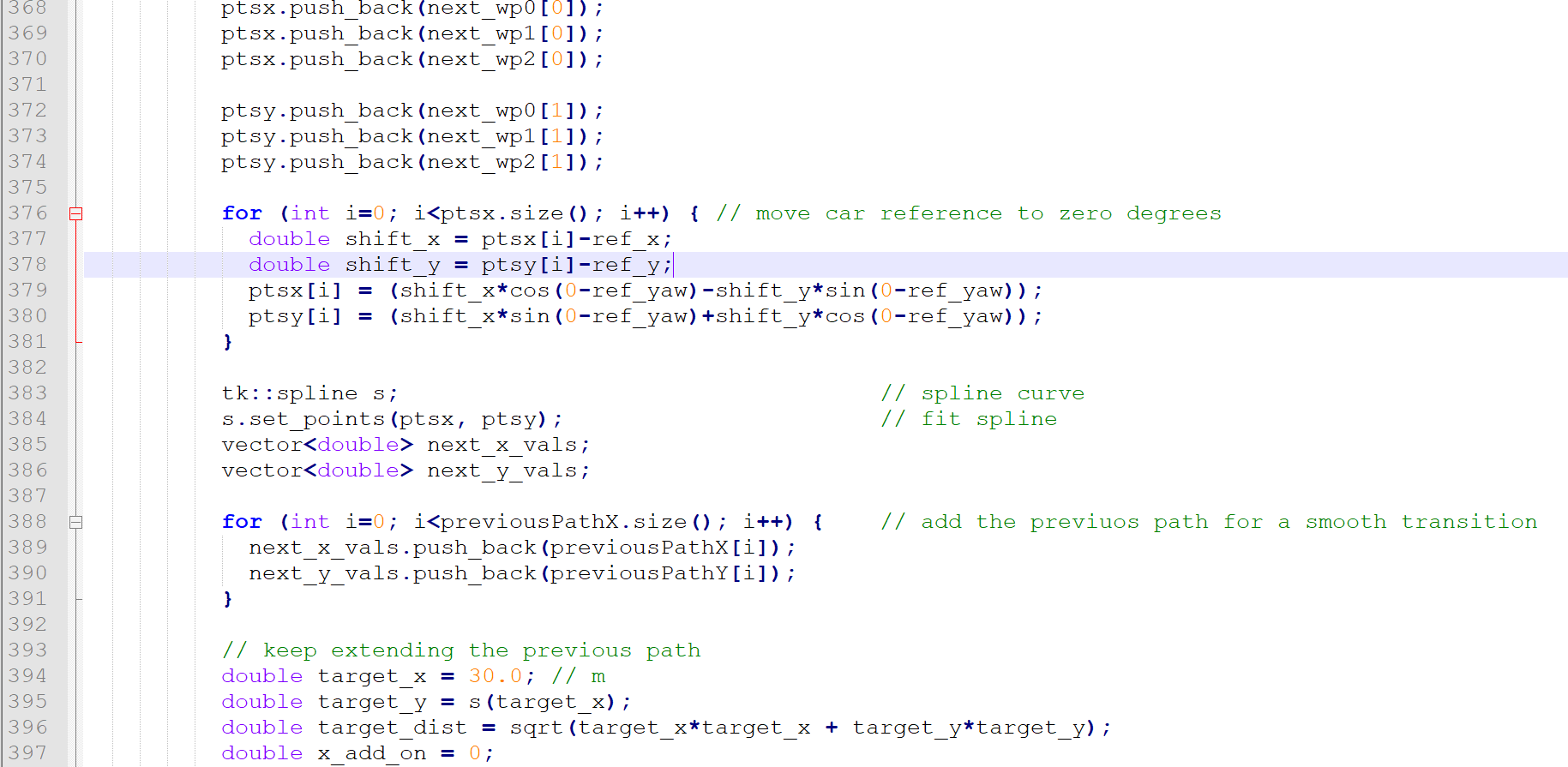
Lines 326 through 328 are primarily used to debug the algorithm, outputting several intermediate steps to the author.

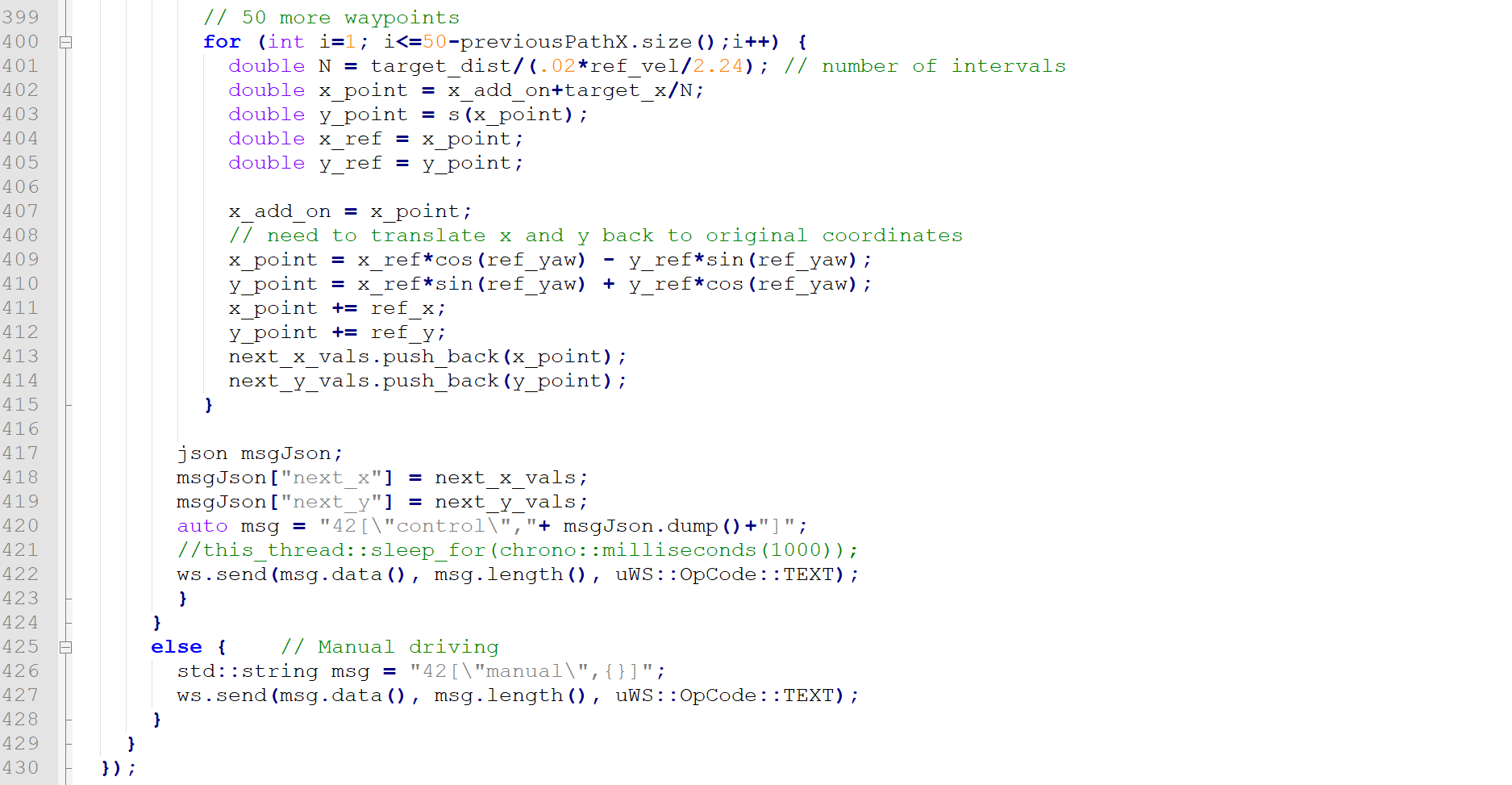
Lines 330 to 361 establish ptsx and ptsy, some of the referenc x, y, and zy, and start assembling the variables to send to the simulator.

Lines 364, 365, and 366 get the X and Y from the car S.



Lines 368 to 397 are equivalent to ones given in the video and start assembling the values to give back to the simulator.



Lines 400 to 430 finish up adding the points and send the message to the simulator. Then the lambda function is concluded (line 430). 

The remainder of the file is verbatim from examples given and is not included here.

## Conclusions and Further Thoughts

The car drives around the track.

There are some corner cases that could be improved. The acceleration allowed is not calculated for all conditions and instead is dealt with empirically in some cases. Finding the best lane is not done in all circumstances and in fact should be dealt with earlier as basically all cars’ position on the road is know early. Even further, we could predict some simple movements of other cars and then calculate out what we want to do from there.

All in all it works but it’s fairly ugly.