Model Documentation

** Highway Driving / Path Planning Project **

The Rubric Points of this project are the following:
* The car is able to drive at least 4.32 miles without incident.
* The car drives according to the speed limit.
* Max Acceleration and Jerk are not Exceeded.
* Car does not have collisions.
* The car stays in its lane, except for the time between changing lanes.
* The car is able to change lanes
* Reflection of the Path Planner - a Write Up called “Model Documentation”

Files Submitted & Code Quality

1. Submission includes all required files and can be used to run the simulator in autonomous mode

My project includes the following files:
* Write Up called “Model Documentation” (you’re reading it)
* main.cpp
* spline.h
* json.hpp
* helpers.h
* Eigen Library

2. Submission includes functional code

Using the Udacity provided simulator and the main.cpp file together with the other src Files from the GitHub Rep the car steers itself around the track in the simulator “Project 1: Path Planning” if you use the entry “./path_planning” in the console after going to the right path (using “cd /work/CarND-Path-Planning-Project/build”).

3. Submission code is usable and readable

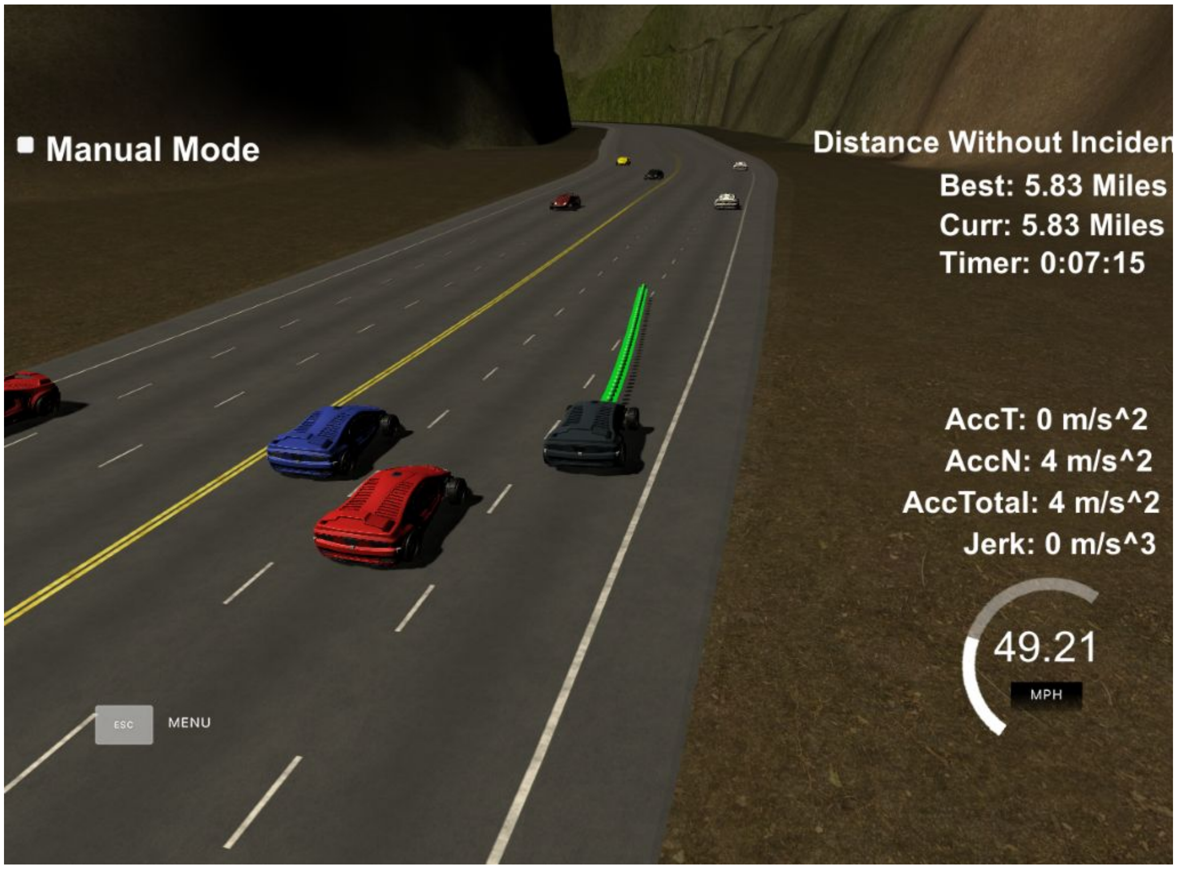
The main.cpp file contains the code that works together with the other src files and the added spline.h (as was given as an alternative to the method used in the lessons), which is an easy to use library calculating a spline, so the car drives on a smooth curve, so none of the rubric points concerning the Jerk, and Accelerations get violated and car drives in bends.
The code contains lots of descriptive comments and variables so the code is easy to understand.

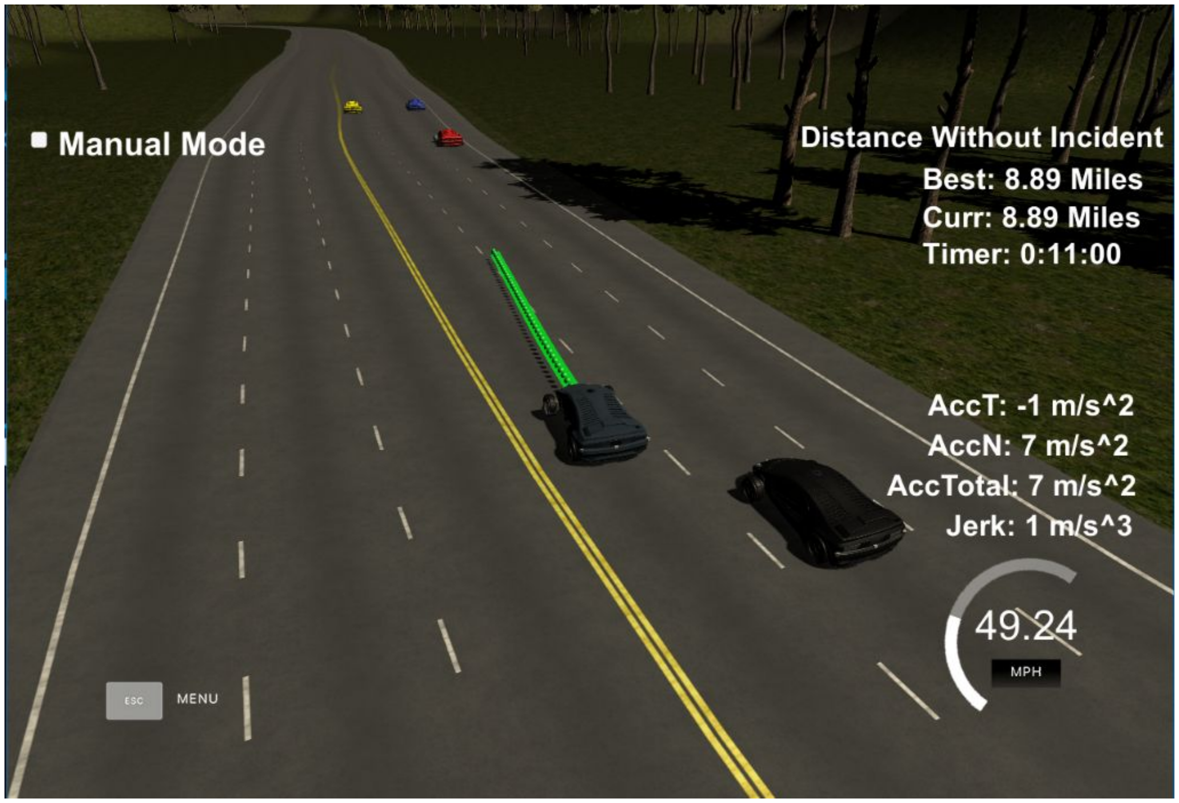
The Path Planning Strategy - Rubric Points

Here I will consider the rubric points (https://review.udacity.com/#!/rubrics/1971/view) individually and describe how I addressed each point in my implementation.

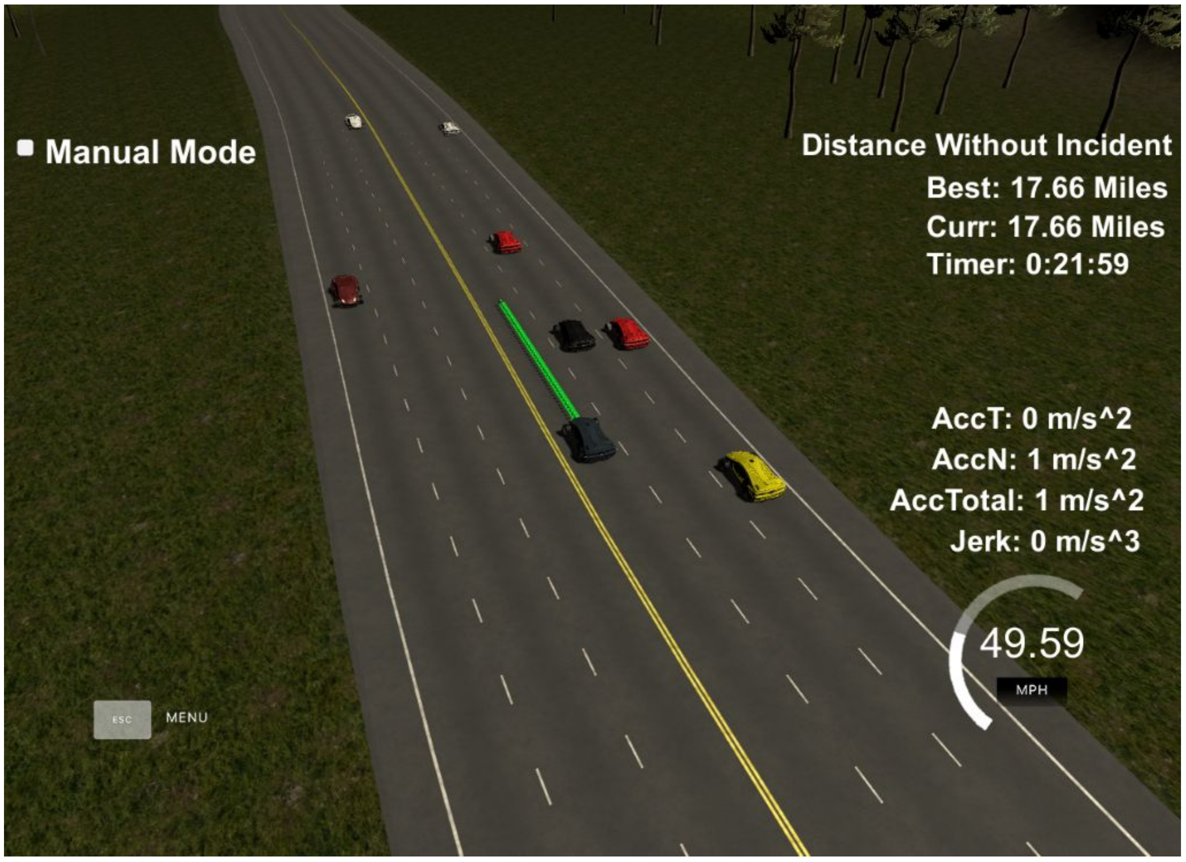
1. The car is able to drive at least 4.32 miles without incident.

Three screenshots here will show that the above has worked (and two display the ability to change lanes to make it more interesting):





(For the third trial I had optimized the speed average a little further. Rest stayed the same.)



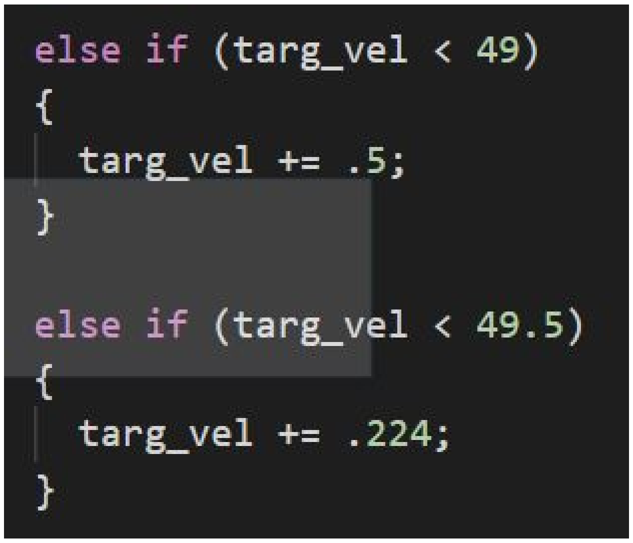
2. The car drives according to the speed limit.

The code does not contain a possibility for the car to drive faster than 50 mph (the speed limit). The code essentially accelerates at a rate of maximum .5 mph per 0.02 seconds, which is within the confinements of Acceleration and Jerk Maxima.
Most of the time (when the car is not obstructed by traffic) the car will drive just below the speed limit, in order to stay within the rubric and legal requirements but at the same time drives not much slower (as also required).
In order to achieve a little higher peak speed, I changed the code to accelerate to .5 mph acceleration till 49 mph and from there only .224 mph are added per step. The code doesn't go beyond 49.5 mph, however the measurement in the simulator is slightly higher but still below 50 mph. Like this the acceleration is fast in general, and the overall speed around the track is a little higher, than by only using .5 mph increments.

3. Max Acceleration and Jerk are not Exceeded.

The code essentially accelerates at a rate of maximum .5 mph per 0.02 seconds, which is within the confinements of Acceleration and Jerk Maxima. The car also breaks by the same amount, which again stays within the parameters which means the car should not experience total acceleration over 10 m/s^2 and jerk that is greater than 10 m/s^3.

Example code:



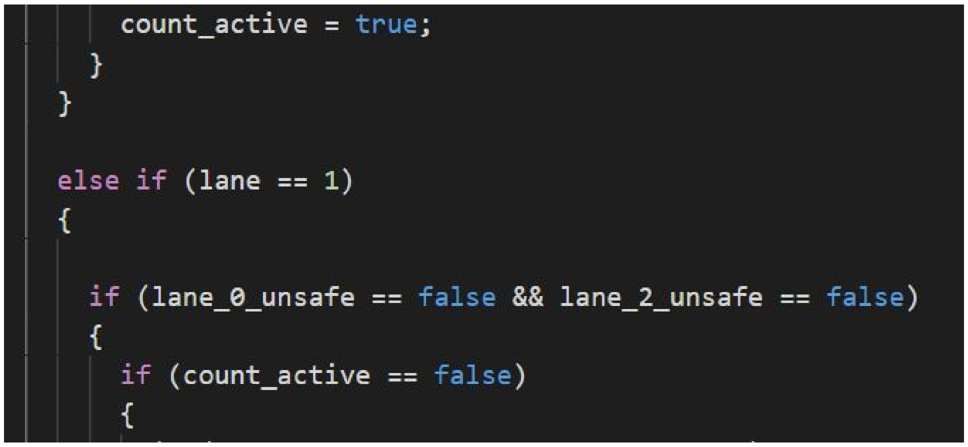
```
else if (targ_vel < 49)
{
  targ_vel += .5;
}

else if (targ_vel < 49.5)
{
  targ_vel += .224;
}
```

The amount of acceleration, speed and breaking is not sufficient to meet this criteria though in my case: I have programmed a path planner that will change lanes into a lane that is less advantageous in direct comparison with the lane the car is in, if that enables the car to get

unstuck and move onwards to a better lane on the other side, that makes it worth to change lanes in to the “less good” lane first in order to move on to the overall best one.
This results in a double lane change, that if that lane change is done to rapidly exceeds the Max Acceleration, due to the sudden change of direction in the calculated route (using the spline). This is why I made sure that 1 second passes before the next lane change is possible, so the car keeps a smooth and pleasant path, to enjoy for the theoretical passengers (and to meet the Rubric requirements).

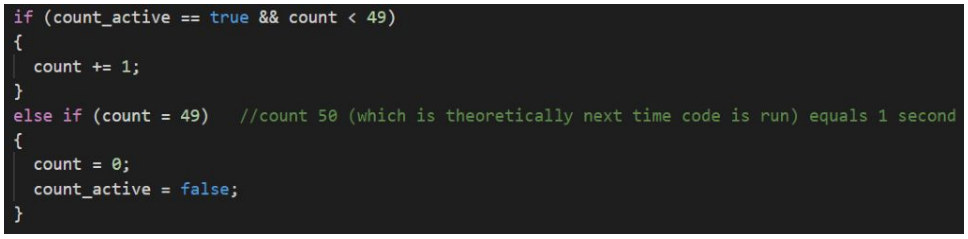
The same goes for a zig zag lane change, where the car moves into one lane to notice a better path opening up in the lane it came from. Here again is a slight delay build in, in order to keep the lane and hence the direction changes smooth enough to stay within the rubric points.

The code for keeping the transitions smooth:



```
            count_active = true;
        }
    }

    else if (lane == 1)
    {

        if (lane_0_unsafe == false && lane_2_unsafe == false)
        {
            if (count_active == false)
            {
```

In the first displayed line the count_active flag will be set to true, which in turn will start the count in the piece of code below. After the count is at 49 (so in the 50th loop it runs through the code again which should take about 1 second, if the code loops every 0.02 seconds) the count_active flag will be set to false again and the lane changing logic for lane 1 will be accessible for execution and result in an action according to the cost functions logic.

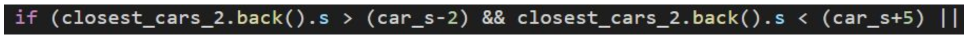


```
if (count_active == true && count < 49)
{
  count += 1;
}
else if (count = 49)   //count 50 (which is theoretically next time code is run) equals 1 second
{
  count = 0;
  count_active = false;
}
```

4. Car does not have collisions.

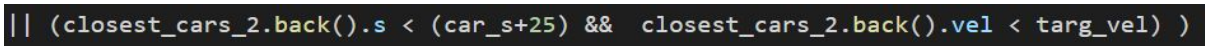
Here my decision was to not use a cost function, but a simple logic, that is: Either the lane to change into is safe or not.
If the other cars on the road are calculated to be far enough away and not too fast to bump into my car, the lane is considered safe for a potential lane change.
At this point I have collected the data necessary and rearranged them into matching vectors of cars in front of the car (or at the same height, all measured in frenet s coordinates) and cars driving behind the car.
The lane parts in front and in the back of my cars frenet s values are checked to be free of other cars within 5 meters to the front and to the back.

The logic for cars in front, in order to check if a lane is safe:



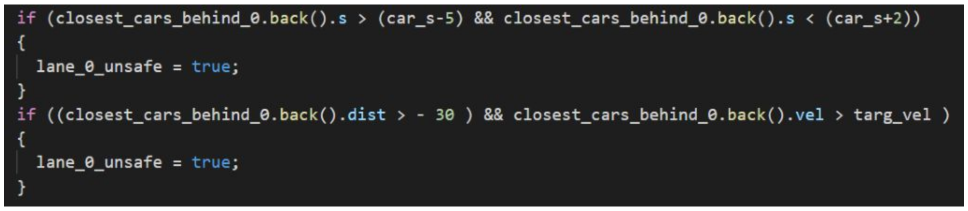
```
if (closest_cars_2.back().s > (car_s-2) && closest_cars_2.back().s < (car_s+5) ||
```

Second part below:



```
|| (closest_cars_2.back().s < (car_s+25) &&  closest_cars_2.back().vel < targ_vel) )
```

The cars distance to my car is checked, and if the threshold of up to 25 meters in front of my car is given the car will be able to change into the lane only if the car in front has a velocity equal or higher to my car. (This is necessary for slow moving traffic, where there is no other way but to slowly change lanes.) Otherwise I would have stayed with the original idea of mine, that would simply not allow a lane change at all as soon as a car was within 25 meters of my s frenet.
I have got the same logic vice versa for the cars behind me. The range within there is absolutely no movements allowed is next to the car and 5 meters in front and back of the car. This value is already the estimation into the future according to current position, speed and time. An additional layer of security is given by any car that is within 25 meters to the front or 30 meters to the back and is slower (in front) or faster (in the back) than my car also triggers the lane unsafe flag.



```
if (closest_cars_behind_0.back().s > (car_s-5) && closest_cars_behind_0.back().s < (car_s+2))
{
  lane_0_unsafe = true;
}
if ((closest_cars_behind_0.back().dist > - 30 ) && closest_cars_behind_0.back().vel > targ_vel )
{
  lane_0_unsafe = true;
}
```

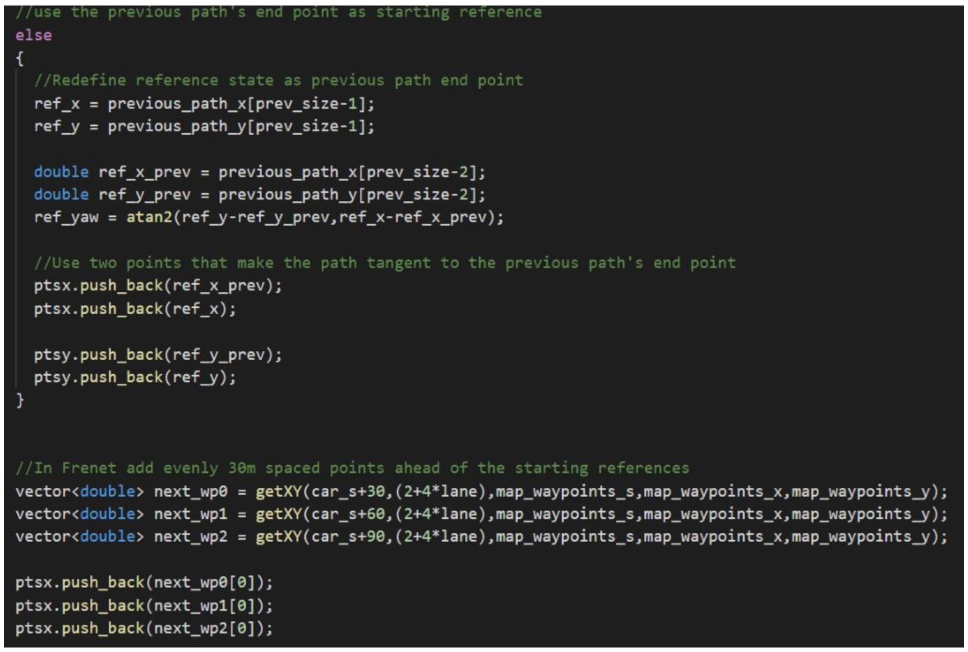
Additionaly the too_close flag introduced in the QnA is also used, to keep the car from bumping into a slower car, if there are no options to pass the car safely in time. In this scenario the speed will of course be reduced to avoid a collision.

5. The car stays in its lane, except for the time between changing lanes.

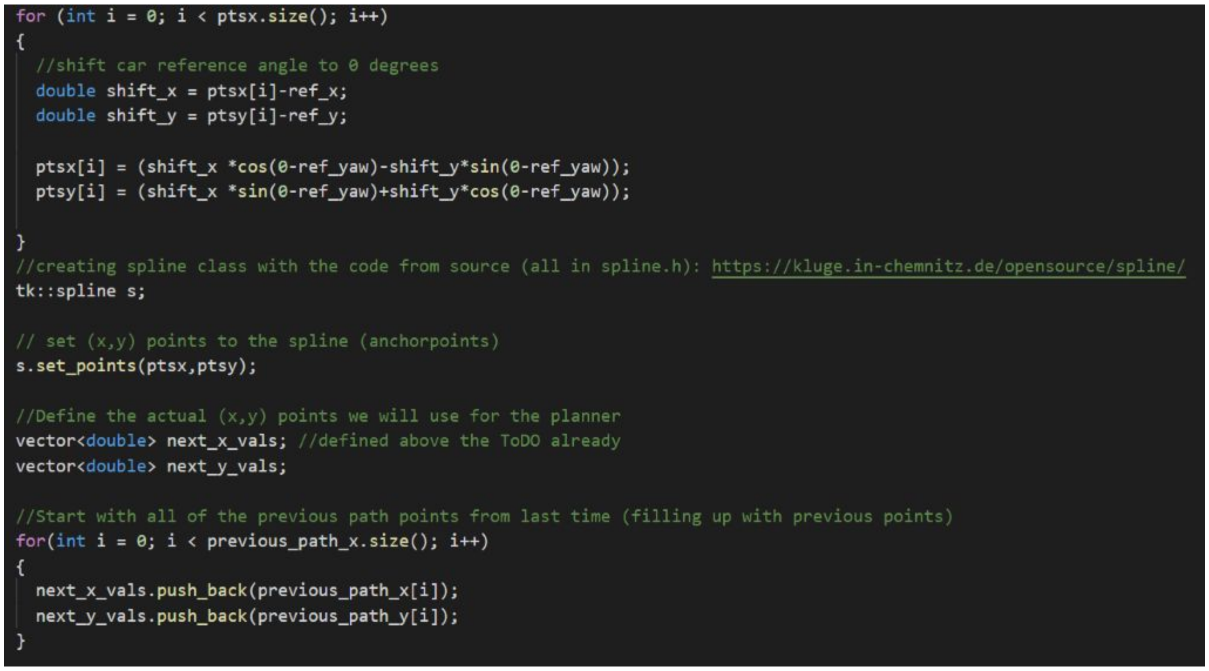
The cars route is calculated using the spline library and the Helpers function getXY relative to the middle of the lane. Since the lane width was given for the project the calculation for the centre line within each lane is calculated (each lane 4 meters width, 3 lanes in total makes the centres of these lanes at 2, 6 and 10 meters away from the centre).

In order to keep it smooth and within the above-mentioned Maxima the new part of the spline is calculated on top of the last given points of the previously calculated path. If there is no previously calculated path yet, I start with points in reference to the current position of the car as was suggested in the QnA part. The spline library makes the calculation of a smooth path following the bends of the road possible. An alternative would be a polynomial like in the lessons. Since the lane variable is part of the calculation of the spline, all I have to do in order to switch lanes, is to change the variables value and the spline will follow the new lane and a spline being a smooth curve at all times will also automatically calculate a smooth transition to the other lane.

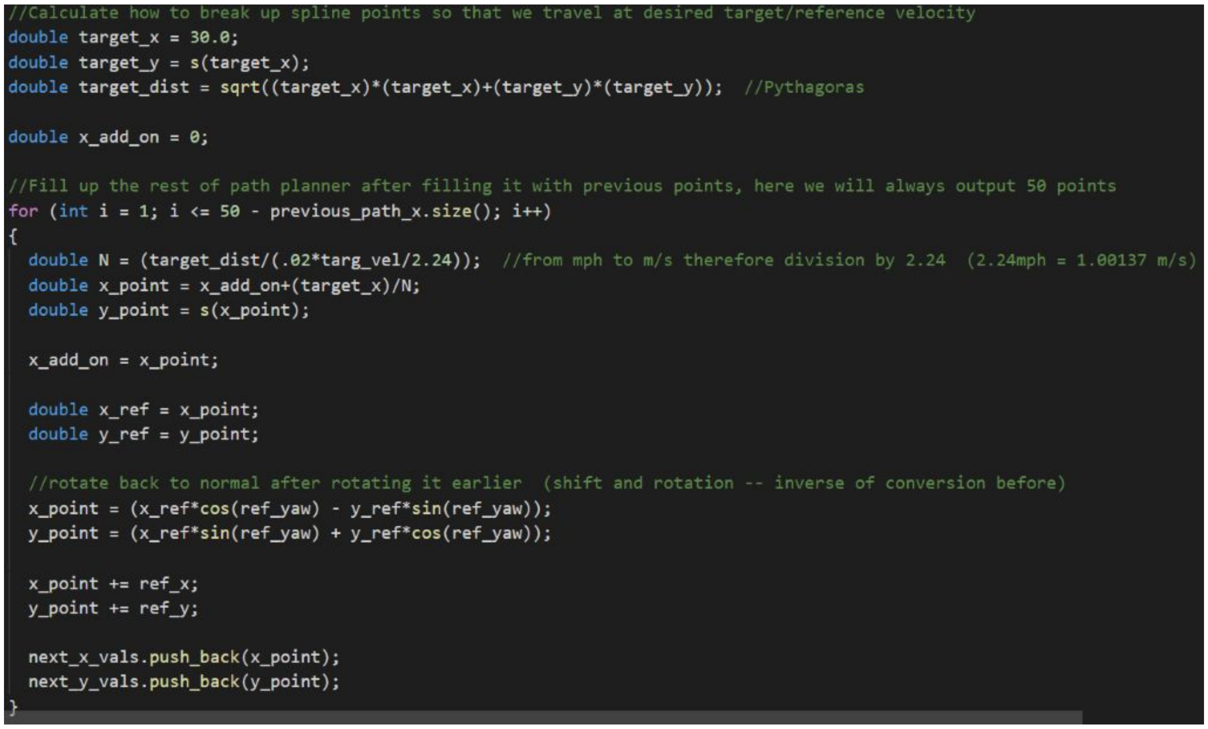
The main parts of the corresponding code along with plenty comments that point out what is happening.



```
//use the previous path's end point as starting reference
else
{
  //Redefine reference state as previous path end point
  ref_x = previous_path_x[prev_size-1];
  ref_y = previous_path_y[prev_size-1];

  double ref_x_prev = previous_path_x[prev_size-2];
  double ref_y_prev = previous_path_y[prev_size-2];
  ref_yaw = atan2(ref_y-ref_y_prev,ref_x-ref_x_prev);

  //Use two points that make the path tangent to the previous path's end point
  ptsx.push_back(ref_x_prev);
  ptsx.push_back(ref_x);

  ptsy.push_back(ref_y_prev);
  ptsy.push_back(ref_y);
}


//In Frenet add evenly 30m spaced points ahead of the starting references
vector<double> next_wp0 = getXY(car_s+30,(2+4*lane),map_waypoints_s,map_waypoints_x,map_waypoints_y);
vector<double> next_wp1 = getXY(car_s+60,(2+4*lane),map_waypoints_s,map_waypoints_x,map_waypoints_y);
vector<double> next_wp2 = getXY(car_s+90,(2+4*lane),map_waypoints_s,map_waypoints_x,map_waypoints_y);

ptsx.push_back(next_wp0[0]);
ptsx.push_back(next_wp1[0]);
ptsx.push_back(next_wp2[0]);
```



```
for (int i = 0; i < ptsx.size(); i++)
{
  //shift car reference angle to 0 degrees
  double shift_x = ptsx[i]-ref_x;
  double shift_y = ptsy[i]-ref_y;

  ptsx[i] = (shift_x *cos(0-ref_yaw)-shift_y*sin(0-ref_yaw));
  ptsy[i] = (shift_x *sin(0-ref_yaw)+shift_y*cos(0-ref_yaw));

}
//creating spline class with the code from source (all in spline.h): https://kluge.in-chemnitz.de/opensource/spline/
tk::spline s;

// set (x,y) points to the spline (anchorpoints)
s.set_points(ptsx,ptsy);

//Define the actual (x,y) points we will use for the planner
vector<double> next_x_vals; //defined above the ToDO already
vector<double> next_y_vals;

//Start with all of the previous path points from last time (filling up with previous points)
for(int i = 0; i < previous_path_x.size(); i++)
{
  next_x_vals.push_back(previous_path_x[i]);
  next_y_vals.push_back(previous_path_y[i]);
}
```



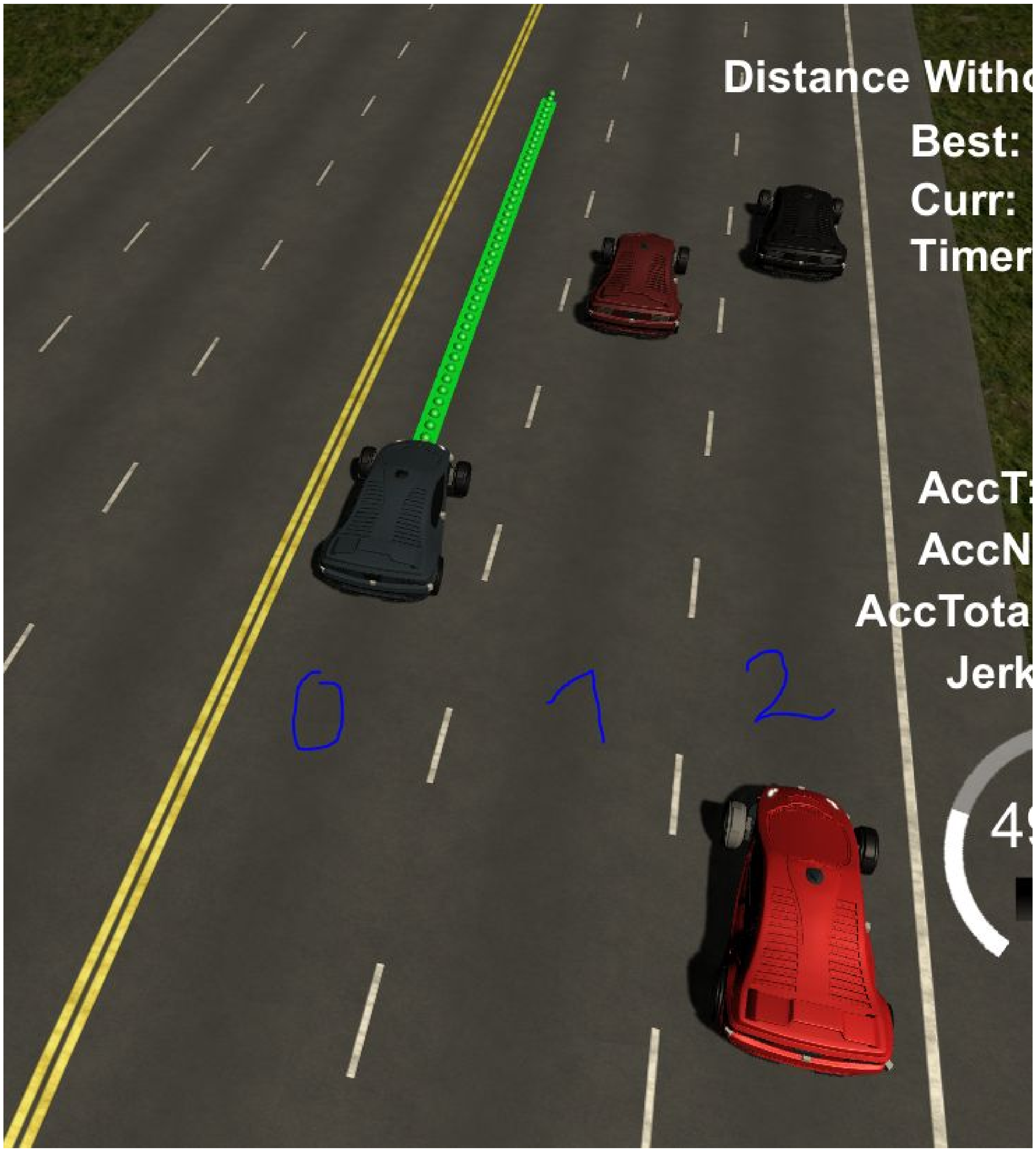
```
//Calculate how to break up spline points so that we travel at desired target/reference velocity
double target_x = 30.0;
double target_y = s(target_x);
double target_dist = sqrt((target_x)*(target_x)+(target_y)*(target_y));  //Pythagoras

double x_add_on = 0;

//Fill up the rest of path planner after filling it with previous points, here we will always output 50 points
for (int i = 1; i <= 50 - previous_path_x.size(); i++)
{
  double N = (target_dist/(.02*targ_vel/2.24));  //from mph to m/s therefore division by 2.24  (2.24mph = 1.00137 m/s)
  double x_point = x_add_on+(target_x)/N;
  double y_point = s(x_point);

  x_add_on = x_point;

  double x_ref = x_point;
  double y_ref = y_point;

  //rotate back to normal after rotating it earlier  (shift and rotation -- inverse of conversion before)
  x_point = (x_ref*cos(ref_yaw) - y_ref*sin(ref_yaw));
  y_point = (x_ref*sin(ref_yaw) + y_ref*cos(ref_yaw));

  x_point += ref_x;
  y_point += ref_y;

  next_x_vals.push_back(x_point);
  next_y_vals.push_back(y_point);
}
```

6. Explanation of Code structure in General (no specific rubric point):

We have the data of the own car and with sensor fusion measured data of the cars in its environment.

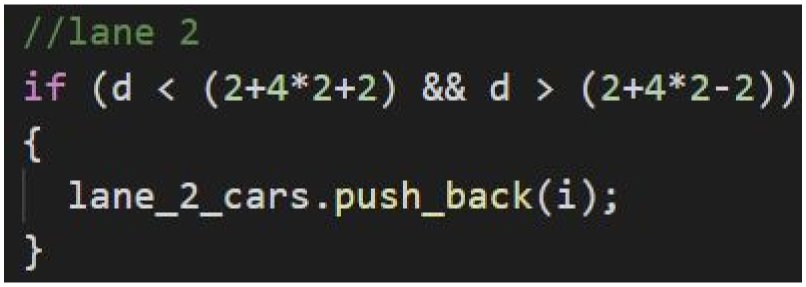
These data is accessed and processed in a few steps within my code:

The lane numbers drawn into the simulation:



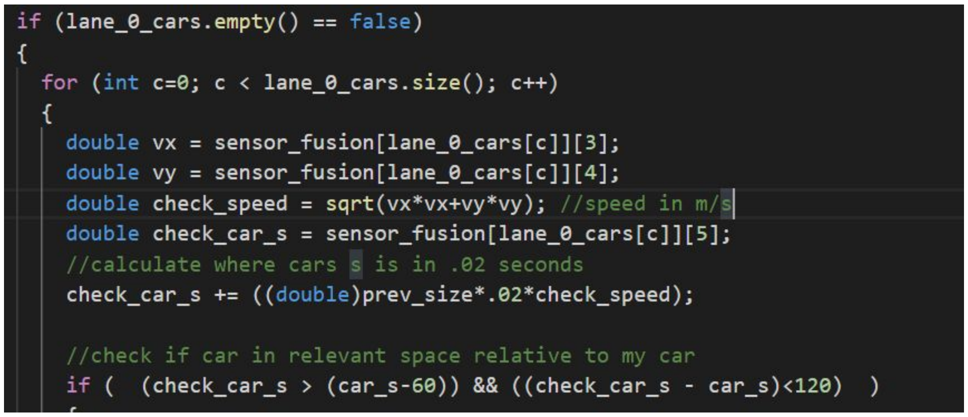
All of the cars are sorted according to the lanes they are driving in.

Sorting example below according to lanes using the cars d value relative to the road lane:



```
//lane 2
if (d < (2+4*2+2) && d > (2+4*2-2))
{
  lane_2_cars.push_back(i);
}
```

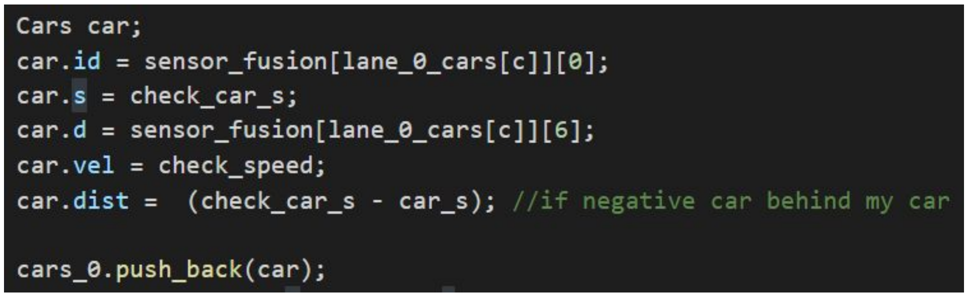
Only keeping the cars within the parameters of 120 meters to the front relative to my car and 60 meters to the back of my car:



```
if (lane_0_cars.empty() == false)
{
  for (int c=0; c < lane_0_cars.size(); c++)
  {
    double vx = sensor_fusion[lane_0_cars[c]][3];
    double vy = sensor_fusion[lane_0_cars[c]][4];
    double check_speed = sqrt(vx*vx+vy*vy); //speed in m/s
    double check_car_s = sensor_fusion[lane_0_cars[c]][5];
    //calculate where cars s is in .02 seconds
    check_car_s += ((double)prev_size*.02*check_speed);

    //check if car in relevant space relative to my car
    if (  (check_car_s > (car_s-60)) && ((check_car_s - car_s)<120)  )
```

This is done in a way that the future location of the current s is calculated using the given information from the sensor fusion data. The speed and time and cars s are all used to calculate the forecast s value in .02 seconds (the code will be translated into action and updates by the simulator every .02 seconds ideally).

Then I collect all the relevant cars for the respective lane along with the relevant information in a vector that can save the specific struct Cars:

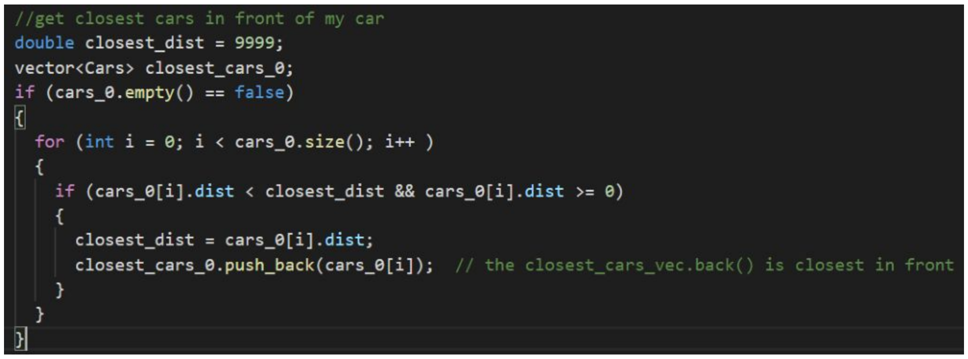


```
Cars car;
car.id = sensor_fusion[lane_0_cars[c]][0];
car.s = check_car_s;
car.d = sensor_fusion[lane_0_cars[c]][6];
car.vel = check_speed;
car.dist =  (check_car_s - car_s); //if negative car behind my car

cars_0.push_back(car);
```

Here the same logic for variable name giving applies throughout the code, to keep it simple and readable: lane_0 has cars_0 and cost_0, lane_1 has cars_1 and cost_1, etc.

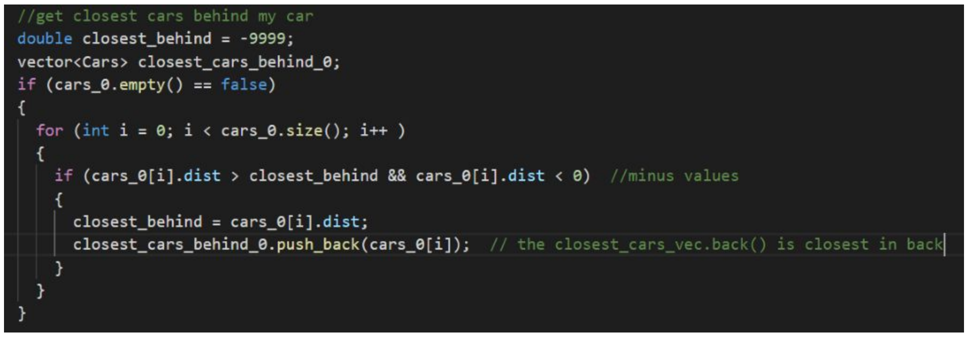
The code than gets a vector for cars in front of my car and a vector for cars in the back of my car and sorts them in a way that always the last element of that vector is the car closest to my car (measured in s and d frenet coordinates).

Example for front (which is very self explanatory):



```
//get closest cars in front of my car
double closest_dist = 9999;
vector<Cars> closest_cars_0;
if (cars_0.empty() == false)
{
  for (int i = 0; i < cars_0.size(); i++ )
  {
    if (cars_0[i].dist < closest_dist && cars_0[i].dist >= 0)
    {
      closest_dist = cars_0[i].dist;
      closest_cars_0.push_back(cars_0[i]);  // the closest_cars_vec.back() is closest in front
    }
  }
}
```

And an example for the cars behind my car (same logic vice versa, as the distance value is in my code now in the minus):



```
//get closest cars behind my car
double closest_behind = -9999;
vector<Cars> closest_cars_behind_0;
if (cars_0.empty() == false)
{
  for (int i = 0; i < cars_0.size(); i++ )
  {
    if (cars_0[i].dist > closest_behind && cars_0[i].dist < 0)  //minus values
    {
      closest_behind = cars_0[i].dist;
      closest_cars_behind_0.push_back(cars_0[i]);  // the closest_cars_vec.back() is closest in back
    }
  }
}
```

Again the variable name giving is consistent and simple: lane_0 has closest_cars_0 and closest_cars_behind_0, it's the same for the other lanes

7. The car is able to change lanes

The car has a logic for lane changes. It's a combination of two essential questions:
How to change lanes most effectively to avoid getting stuck and be efficiently travelling?
How to make sure that the car stays safe and within rubric requirements (safety and also regarding jerk and acceleration maximum values)? This second question has already been answered above. I have refrained from making it some kind of cost function, since to me it's either safe to change a lane or not, no "value" needed.

However in order to have an algorithm make a decision, that is sensible towards the other cars, I had to gather information about the environment and the cars therein in order to react accordingly.

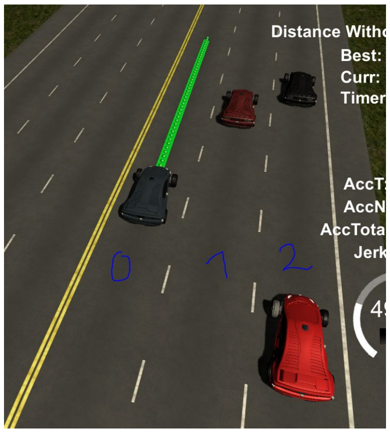
Essentially the car is staying in its lane if the lane has no cost, which in my case means that there is no car within the vicinity of 120 meters to the front.

As soon as there is a car within the 120 meters this gets flagged within the code:

```
car_ahead = true; //results in changed behavior
```

Now the code will execute a part that only gets activated if there is a car ahead. The lane change logic, which plans a path through the upcoming situation.

The logic is individualized for this specific projects' "world".



Essentially the lanes 0 and 2 which are the lanes, which are *not* the middle lane of the side of the road my car is driving on, have a different logic that works well in combination to the logic for lane 1, the lane between the two (see pic above showing which lane is which).

Essentially the cost functions are calculated for all 3 lanes simultaneously the entire time. Which means the car always has a cost value for each of the lanes, regardless where the car is on these 3 lanes.

cost_0 for lane_0, lane_1 will have cost_1, etc.

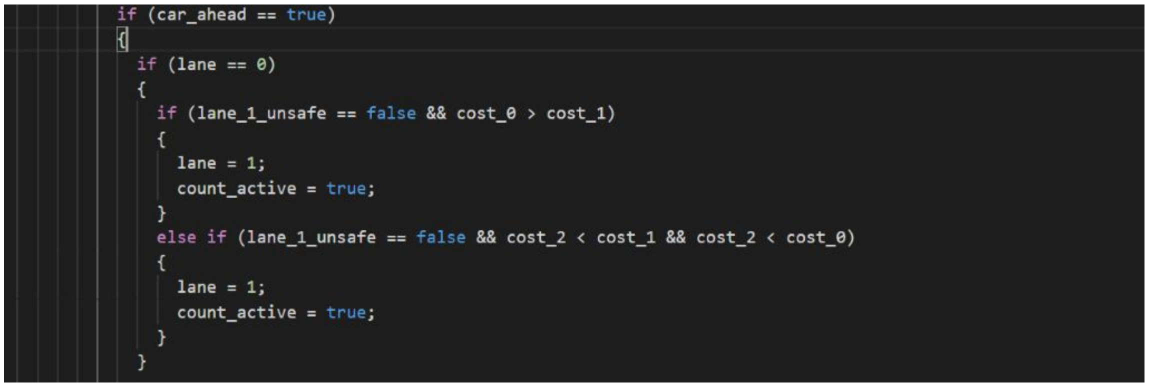
One of the costs calculations:

```
cost_0 = 120 - (closest_cars_0.back().s - car_s);
```

Essentially only cars within 120 meters to the front are considered further up in the code already. Within this range the distance between the closest car (in relation to my car) and my car will be subtracted from the range of 120. This way we have an exact cost value that adheres to the actual distance, the further away the car, the less cost, the better the lane is to be on as a rule of thumb.

The higher the cost, the more this lane with the higher costs will be avoided, but always only in relation to the other two lanes.

The entire lane changing logic for lane 0 is in principle the same for lane 2:



```
if (car_ahead == true)
{
  if (lane == 0)
  {
    if (lane_1_unsafe == false && cost_0 > cost_1)
    {
      lane = 1;
      count_active = true;
    }
    else if (lane_1_unsafe == false && cost_2 < cost_1 && cost_2 < cost_0)
    {
      lane = 1;
      count_active = true;
    }
  }
```

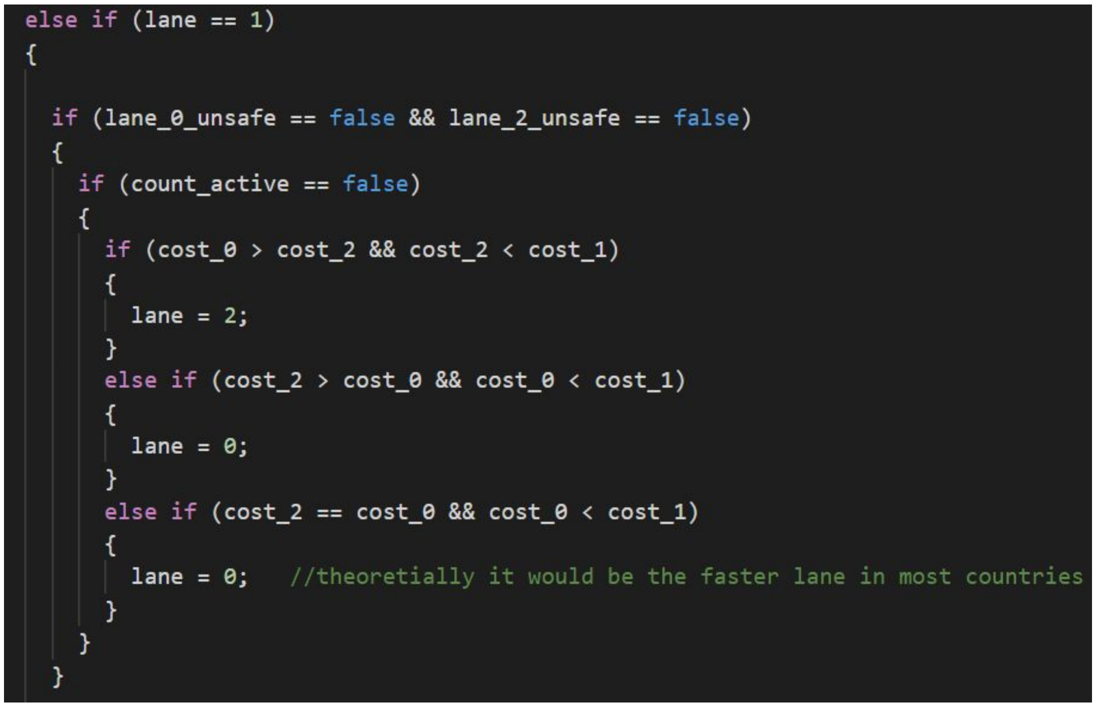
In words: If it is safe to change the lane and the cost for the lane next to the lane is lower drive there. (Since the car can only go to one lane from the outer lanes, that is the best option.) However if the cost of the other outmost lane is less than both other lanes, go into the direction of that lane, even if the lane in between has higher cost than the current lane. Always when a lane change is triggered from the outside lanes, the count starts in order to make sure the smoothness is adhered to, as described above.

The logic for the middle lane lane_1 is a bit more complex:

The cost function will decide which lane to change to or whether to stay in the current lane. If there happens to be equal costs on both other lanes (lane_0 and lane_2) I decided to always make a lane change to the lane_0, so the car is not stuck and because in most countries the left lane is faster than the right lane. Of course this also is only done if the lane the car is on has higher cost than the lane to be switched to.

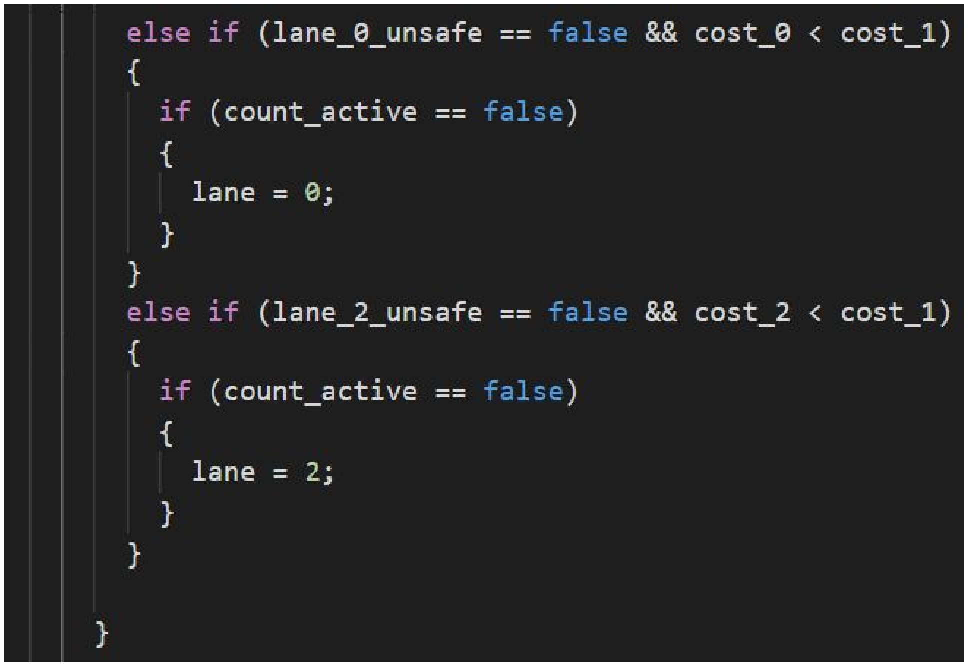
Lane changes are again only possible if they are safe. But they are also only allowed if no counting is going on / some time went by to keep the smooth transitions.

In the below part of the code we are looking at the case, that it is safe to change lanes to both sides.



```
else if (lane == 1)
{

  if (lane_0_unsafe == false && lane_2_unsafe == false)
  {
    if (count_active == false)
    {
      if (cost_0 > cost_2 && cost_2 < cost_1)
      {
        lane = 2;
      }
      else if (cost_2 > cost_0 && cost_0 < cost_1)
      {
        lane = 0;
      }
      else if (cost_2 == cost_0 && cost_0 < cost_1)
      {
        lane = 0;   //theoretially it would be the faster lane in most countries
      }
    }
  }
```

For the common case that one of the two lanes can be safely changed to, but the other not (and the count is inactive):



```
    else if (lane_0_unsafe == false && cost_0 < cost_1)
    {
      if (count_active == false)
      {
        lane = 0;
      }
    }
    else if (lane_2_unsafe == false && cost_2 < cost_1)
    {
      if (count_active == false)
      {
        lane = 2;
      }
    }

  }
```

Then the lane change is only initiated if it is safe to go to the lane with less cost than lane_1.