**Path Planning Project**

**Mandar Joshi**

1. **Code Layout & modifications:** Several detailed comments in the code. Summary below:
   1. Main.cpp/h – parses the information received from the system (ego vehicle information, sensor fusion data, previous path, etc.) saves it for processing by calling vehicle.cpp:update\_state()
   2. Road.cpp/h – Before the simulation can begin, an object of Road class is created. It holds 1) the waypoints for a given track 2) the ego vehicle and 3) all the neighboring vehicles.
   3. Map.cpp/h – The waypoints provided are spaced about 30 mts apart. In our approach we add about 50 points using the spline library (spline.h) between 2 consecutive points with the goal of improving the accuracy of localizing the ego vehicle. So in the 181 points become 90,000 points for the udacity track.
   4. Vehicle.cpp/h – implements all the logic to decide the next\_x and y values for the trajectory. At a high level Step1: Creates possible trajectories, Step2: executes the finite state machine with the possible next steps and picks the one with the lowest cost.
   5. Ptg.cpp/h – Implements the path generation as requested above. More details on the exact steps below.
   6. Constants.h – This header defines constants such as the SPEED\_LIMIT, PREFERRED\_BUFFER, etc.
   7. Debug.cpp/h – Implements a few debugging related functions and macros.
2. **Path generation**
   1. Our lane numbering starts with lane 1 as the left most and lane 3 as the right most lane on the road.
   2. The approach we have taken is to generate all possible trajectories for extreme left, left, straight, right and extreme right depending on the lane. For instance, for an ego vehicle in lane 1 we would generate straight, right and extreme right as the three trajectories. For an ego vehicle in lane 2, we would not generate an extreme left/right trajectory.
   3. Note that extreme right and left trajectories are not valid trajectories for the ego vehicle to take, but are only used in the cost function so as to be able to take into account favorable conditions in the extreme lane and eventually moving to the extreme lane through the LCR/LCL transitions.
   4. This approach for trajectory generation ensures the max acceleration and max jerk constraints are not exceeded. This approach is different from an approach where a large set trajectories are generated (using jerk minimizing trajectories (JMT) approach) and the best is chosen from that set.
   5. Shown below are the steps in generation of each trajectory - ptg.cpp: ptg\_get\_trajectory()
      1. Input:

void

ptg\_get\_trajectory(double target\_d,

int total\_points,

double ref\_s,

double ref\_v,

double car\_x,

double car\_y,

double car\_yaw,//in radians

int lane,

vector <double> \*previous\_path\_x\_ptr,

vector <double> \*previous\_path\_y\_ptr,

vector <double> \*next\_x\_vals\_ptr,

vector <double> \*next\_y\_vals\_ptr,

Waypoints \*waypoints\_ptr)

1. Target\_d: provides the lookahead distance for which the trajectory is to be generated. This depends on the vehicle in front and the “safe distance” from it.
2. Total\_points: Total number of points to be generated. The minimum points we expect are 30 (MIN\_NEXT\_XY\_VALS). Note that these are the total points – there might already be some points left over from the previous cycle that the car has not consumed yet.

(NOTE: Also in this approach it is important to note that the given set of points WILL be traversed by the car and all new points are added on top of the points that have already been generated in previous cycles. This is different than the approach where we could potentially provide a completely new set of points and ignore the previous points to handle newer developing scenarios. )

1. Ref\_s: This is the reference segment which is essentially the last segment of the ego vehicle before we received the json message from the system.
2. Ref\_v: This is the newly computed reference velocity for which the trajectory is to be generated. This new reference velocity is computed based on the vehicle (or open road) for a specific lane (vehicle.cpp: get\_new\_ref\_vel\_for\_lane())
3. Car\_x, car\_y : Provides the current location of the ego vehicle as received by the system.
4. Car\_yaw: Provides the yaw angle of the ego vehicle as received by the system.
5. Lane: This is the lane to which the trajectory is to be generated. This could be straight, left or right.
6. Previous\_path\_x, previous\_path\_y: The set of paths that have not yet been consumed by the ego vehicle.
7. Waypoints: This has the waypoints for a given track.
   * 1. Output: next\_x\_vals, next\_y\_vals: The function returns the next x and y path coordinates for the ego vehicle.
     2. The first step is to create a minimal set of points along which the trajectory is to be generated. To achieve this, the top 2 points in the previous path are picked as the starting 2 points. Note that using these last 2 points we are able to calculate the reference\_yaw of the direction the car is heading as per the previous trajectory we had provided it. This is used in converting the points to the vehicle coordinate system.
     3. In addition, 3 more points are added 30, 60 and 90 mts away. Note that if a lane change is intended, then the input parameter lane will ensure that these points are in the new lane. Also, note that this is not the final trajectory but only a set of points from which the right number of points will be picked up.
     4. These points are then converted to the vehicle coordinate system. This ensures that we have moved the origin to the first point we picked in (iii) above – ref\_x, and ref\_y. This way travel along the trajectory will have increasing x-coordinates of the trajectory – simplifying the computations. At this point we use the spline library that will be able to generate points for us along the trajectory i.e. it will give us the y coordinate when we provide the x coordinate (the x coordinate will essentially be the distance the car is to travel since we have converted to the vehicles coordinate system)
     5. Now, the number of points and the spacing between the points is to be computed so that the reference velocity can be achieved.
     6. Note that trajectory is visited every 0.02 seconds and the top (x, y) point in the trajectory is consumed i.e. the ego vehicle moves from its current location to next point in the trajectory every 0.02 seconds. Therefore we can compute the number of points based on the new reference velocity and the target\_d (input parameter) that we wish to travel with this trajectory.
        1. Note that the actual target distance is computed by projecting the target\_d (the input parameter) on the x axis so we eliminate the curvature of the spline.
     7. Finally, we convert the points back from vehicle coordinate space to the map coordinate space and return. Note that the new points have been added on top of the previous points.