Model Documentation

Code Model:

This part of the code uses the sensor fusion data to map out all objects(cars in this case) detected by the on board sensors. We use the sensors do determine the location (s and d in ferret coordinates) as well as velocities. Having prior knowledge of the road system(width of lane, number of lanes ,etc.) we use the 'd' parameter to pin point which lane the car is in. This helps us determine if the lane is occupied or free to switch to

```
for(int i = 0; i< sensor_fusion.size(); i++)
                    {
                        //get the d value of the car
                        float d = sensor_fusion[i][6];
                        double vx = sensor_fusion[i][3];
                        double vy = sensor_fusion[i][4];
                        double get_car_speed = sqrt(vx*vx+vy*vy);//vector sum fo     vx and vy
                        double get_car_s = sensor_fusion[i][5];
                        double get_car_lane = -1;
                        //car lane based on 'd' distance
                        if(d<=4.0)
                        {
                            get_car_lane = 0;
                        }
                        else if(d>4.0 && d<8.0)
                        {
                            get_car_lane = 1;
                        }
                        else
                        {
                            get_car_lane = 2;
                        }

                        // Estimate car s position after executing previous trajectory.
                        get_car_s += ((double)size_of_previous_path*0.02*get_car_speed);

                        //groundwork for state machine of state machine
                        if(get_car_lane == lane)
                        {
                            //if the car is in our lane, check if future position
      is viable or not
                            our_lane |= (get_car_s > car_s) && (get_car_s - car_s < safety_distance);
                        }else if ( get_car_lane - lane == -1 )
                        {
                            // Car is in the left lane w.r.t us
                            left_lane |= (car_s - safety_distance < get_car_s) && (car_s + safety_distance > get_car_s);
                        } else if ( get_car_lane - lane == 1 )
                        {
                            //car is in right lane w.r.t us
                            right_lane |= (car_s - safety_distance < get_car_s )&& (car_s + safety_distance > get_car_s);
                        }

                    }
```

This part of the code is the “state model”. We use our car velocity, availability of space in front of us, and presence of cars in other lanes to assess the best strategy for driving.

```
//creating rough state machine based on our speed and occupance of lanes
                    if(our_lane)
                    {
                        //If our lane is occupied
                        if(lane > 0 && !left_lane)
                        {
                            //if we arent on leftmost lane, and no car on left lane, then switch to left lane, as we pri
oritise left overtake
                            lane--;
                        }
                        else if(lane < 2 && !right_lane)
                        {
                            //if we arent on rightmost lane, and no car on right lane, then switch to right lane
                            lane++;
                        }
                        else
                        {
                            //slow down car because no other alternative
                            ref_vel-=max_acceleration;

                        }

                    } else
                    {
                        //our lane is free
                        //try switching to slow lane if possible
                        if(lane < 2 && !right_lane)
                        {
                            lane++;
                        }
                        if(ref_vel < max_vel)
                        {
                            ref_vel+=max_acceleration;
                        }

                    }
```

This section of the code involves generating a trajectory using splines. We use the present coordinates of the car, as well as potential future coordinates to generate a smooth 5th order spline, so as to minimize jerk. We transform coordinates to and from local and global coordinates.

```
                    // potential future points
                    vector<double> next_wp0 = getXY(car_s + safety_distance, 2 + 4*lane, map_waypoints_s, map_waypoints_
x, map_waypoints_y);
                    vector<double> next_wp1 = getXY(car_s + 2*safety_distance, 2 + 4*lane, map_waypoints_s, map_waypoint
s_x, map_waypoints_y);
                    vector<double> next_wp2 = getXY(car_s + 3*safety_distance, 2 + 4*lane, map_waypoints_s, map_waypoint
s_x, map_waypoints_y);

                    x_points.push_back(next_wp0[0]);
                    x_points.push_back(next_wp1[0]);
                    x_points.push_back(next_wp2[0]);

                    y_points.push_back(next_wp0[1]);
                    y_points.push_back(next_wp1[1]);
                    y_points.push_back(next_wp2[1]);

                    //Converting to car cordinates
                    for ( int i = 0; i < x_points.size(); i++ )
                    {
                        double shift_x = x_points[i] - ref_x;
                        double shift_y = y_points[i] - ref_y;

                        x_points[i] = shift_x * cos(0 - ref_yaw) - shift_y * sin(0 - ref_yaw);
                        y_points[i] = shift_x * sin(0 - ref_yaw) + shift_y * cos(0 - ref_yaw);
                    }

                    // Create the spline.
                    tk::spline s;
                    s.set_points(x_points, y_points);

                    // Output path points from previous path for continuity.
                    for ( int i = 0; i < size_of_previous_path; i++ )
                    {
                        next_x_vals.push_back(previous_path_x[i]);
                        next_y_vals.push_back(previous_path_y[i]);
                    }
                    // Calculate distance y position on 30 m ahead.
                    double target_x = 30.0;
                    double target_y = s(target_x);
                    double target_dist = sqrt(target_x*target_x + target_y*target_y);

                    double x_add_on = 0;

                    for( int i = 1; i < 50 - size_of_previous_path; i++ )
                    {

                        if ( ref_vel > max_vel )
                        {
                            ref_vel = max_vel;
                        }
                        double N = target_dist/(0.02*ref_vel/2.24);
                        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;

                        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);
                    }
```