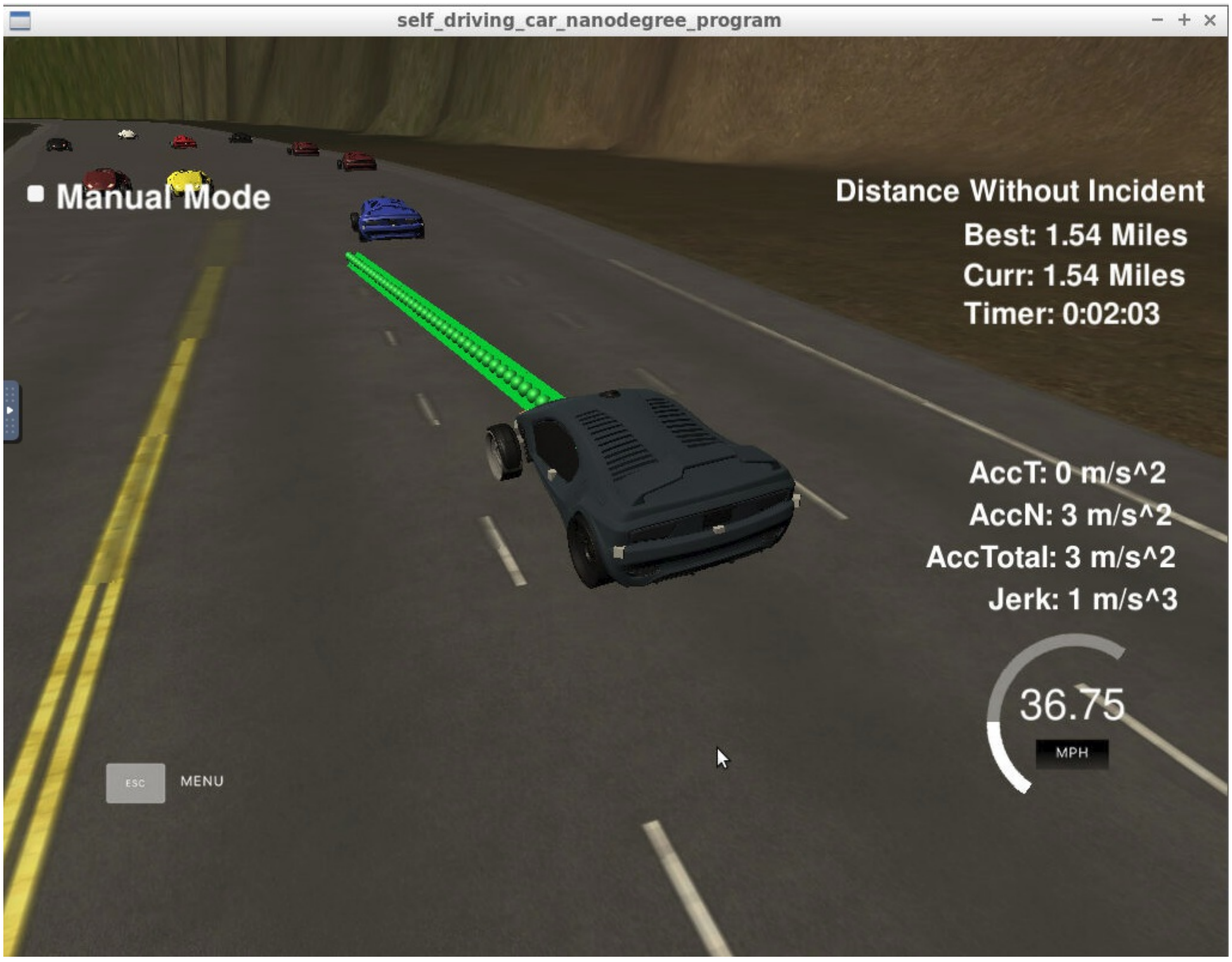
CarND Path Planning Project Report

Path Planning Project

The goals of this project are as follows:

1. Build a path planning algorithm that builds a trajectory for safely driving the car around the track.
2. Use map and sensor fusion data to build trajectory to make the car stay on the track as well as avoid collisions.

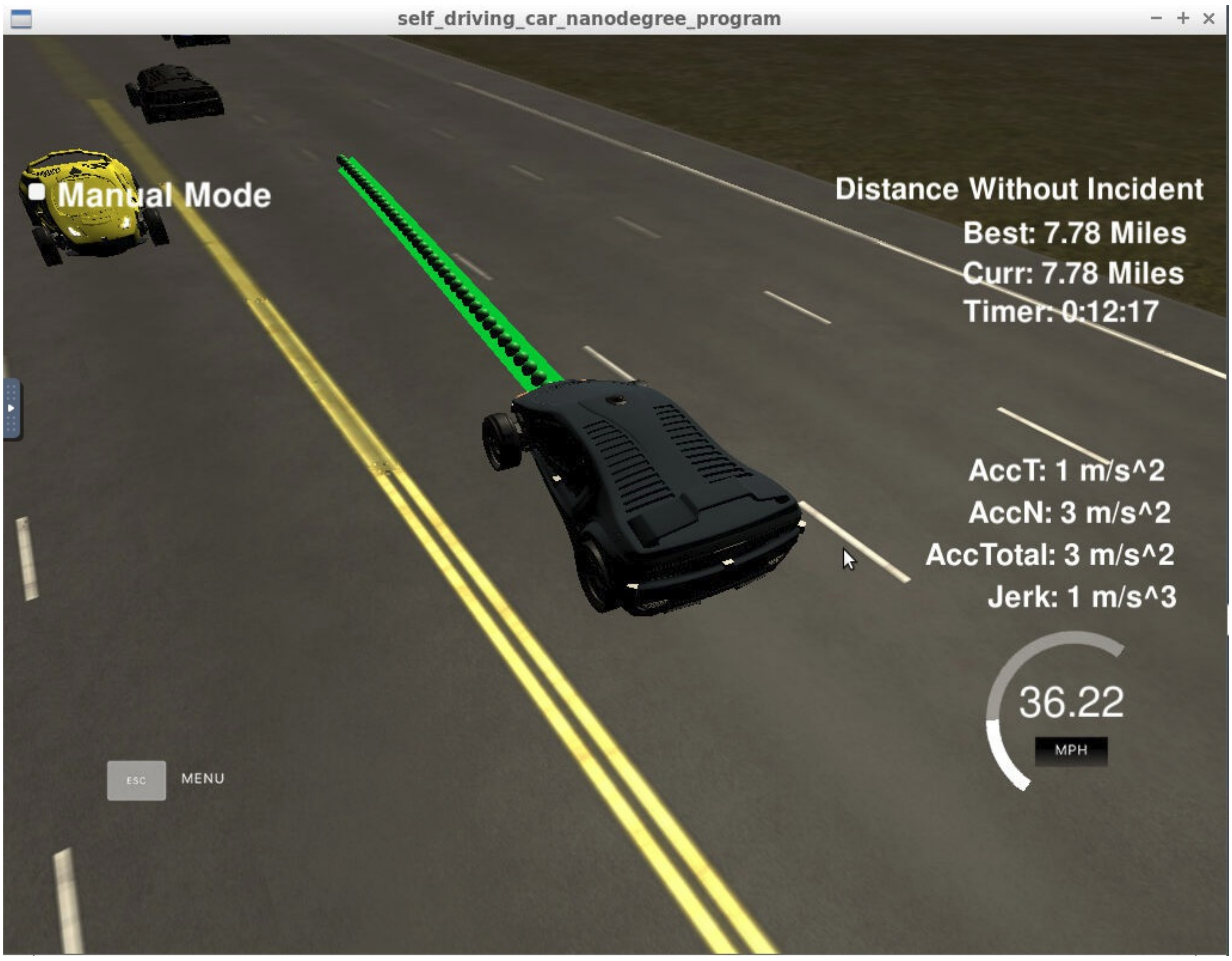


Rubric points

1. The code compiles correctly. ✓
   The code compiles with cmake and make without any warnings. Upon compilation it produces an executable named path_planning
2. The car is able to drive at least 4.32 miles without incident. ✓
3. The car drives according to the speed limit. ✓
4. Max Acceleration and Jerk are not Exceeded. ✓
5. Car does not have collisions. ✓
6. The car stays in its lane, except for the time between changing lanes. ✓
7. The car is able to change lanes ✓

8. There is a reflection on how to generate paths. ✓

Points 2 - 5 were verified in the simulator. Below is a screenshot of the algorithm running on the Udacity simulator for this project.



The workspace has a recording of the algorithm running saved in CarND-Path-Planning-Project/data/run2.mp4.

Model Documentation

The trajectory for the car is generated in two parts:

1. Scan the sensor_fusion input to check for any cars which are on the way and plan avoidance maneuvres. Any target vehicle in the same lane as the host and within 30 meters is
   detected and this triggers a slowing down as well as possible lane change (if a free lane is avaialble to change into). The desired lane thus identified is used in the next step.
2. Identify the intermediate way-points spaced widely at 30, 60 and 90m along the road on the desired lane as determined in step 1. Use spline library to create closely spaced points connecting these.
   The trajectory points thus calculated are such that two consecutive points are separated by a distance not more than what the car can travel in one simulation frame (20 ms) while staying within the speed limit (50 mph).

In main.cpp, lines 137-160 scan the sensor input and determine any target vehicles that are candidates for potential collision and if any of the left/ right lanes are available.
This code snippet is reproduced below:

```cpp
for(auto &tv: sensor_fusion) {
  double tv_d = tv[6];
  double tv_s = tv[5];
  double speed = distance(0, 0, tv[3], tv[4]);
  // project target vehicle location
  double proj = tv_s + speed * frame_period * path_size;
  if( (tv_d > (4*ego_lane)) && (tv_d < (4*(ego_lane + 1)))) {
    // Target vehicle is in the same lane as ego vehicle
    if( (proj > next_s) && ((proj - next_s) < 30) ) {
      // Collision detected
      too_close = true;
    }
  } else {
    if(isValidLane(ego_lane-1) && (getLane(tv_d) == ego_lane-1) && (abs(proj-
next_s) < 30) ) {
      // target vehicle blocks left lane. maybe checking forward direction
alone is sufficient ?
      left_lane_available = false;
    }
    if(isValidLane(ego_lane+1) && (getLane(tv_d) == ego_lane+1) && (abs(proj-
next_s) < 30) ) {
      // target vehicle blocks right lane. maybe checking forward direction
alone is sufficient ?
      right_lane_available = false;
    }
  }
}
```

Lines 161-169 then slow down the car and change the desired lane. Even when a vehicle is detected ahead and a lane is available, a lane change is performed only when it would be comfortable.
In case there is no vehicle directly ahead then the host vehicle speeds up and maintains the speed close to the speed limit.

Next, the code calculates the widely-spaced waypoints. First it copies over any points left from previous_path that are not yet travelled by the car and then adds new points at 30, 60 and 90 meters away measured along the road.
For these new points the d value is set based on the lane decided by the previous step. This set of widely spaced points are first transformed into car-local co-ordinates.
This is done for 2 reasons:

1. The spline produced in this way is very likely to closer to horizontal line. If spline section is vertical then it will have multiple y-values for the same x-value and this makes the calculations harder.
2. In the car-local co-ordinates, the car's next trajectory points can be approximately placed with uniform steps of x-co-ordinates.

The distance along the spline correspong to traveling 30 units along x-axis is approximated by the hypotenuse of the triangle connecting the end-points of the spline.
The number of steps needed to cover this distance is calculated ensuring the car stays below the speed limit (or a further reduced speed in case it is avoiding collision).
This is stored in `num_steps` (line 225) - the increment along x-axis for each step is 30 units divided by `num_steps`.

Now each point along the spline is determined by incrementing the x-coordinate by a step and using the spline function to determine the y-coordinate. This is then
transformed into the map co-ordinates. These points are then appended to the previous_path points and the combined list is returned as the trajectory. Below is reproduced the code-snippet that performs this.

```
// transform from car co-ordinates to world co-ordinates and add to next_x...
for(int i = path_size; i<50; ++i) {
  // upto a maximum of 50 points.
  delta_x += target_x/num_steps;
  auto newy = s(delta_x);
  // rotate and shift co-ordinates.
  double x_point = delta_x*cos(ref_yaw) - newy*sin(ref_yaw) + ref_x;
  double y_point = delta_x*sin(ref_yaw) + newy*cos(ref_yaw) + ref_y;
  next_x_vals.push_back(x_point);
  next_y_vals.push_back(y_point);
}
```