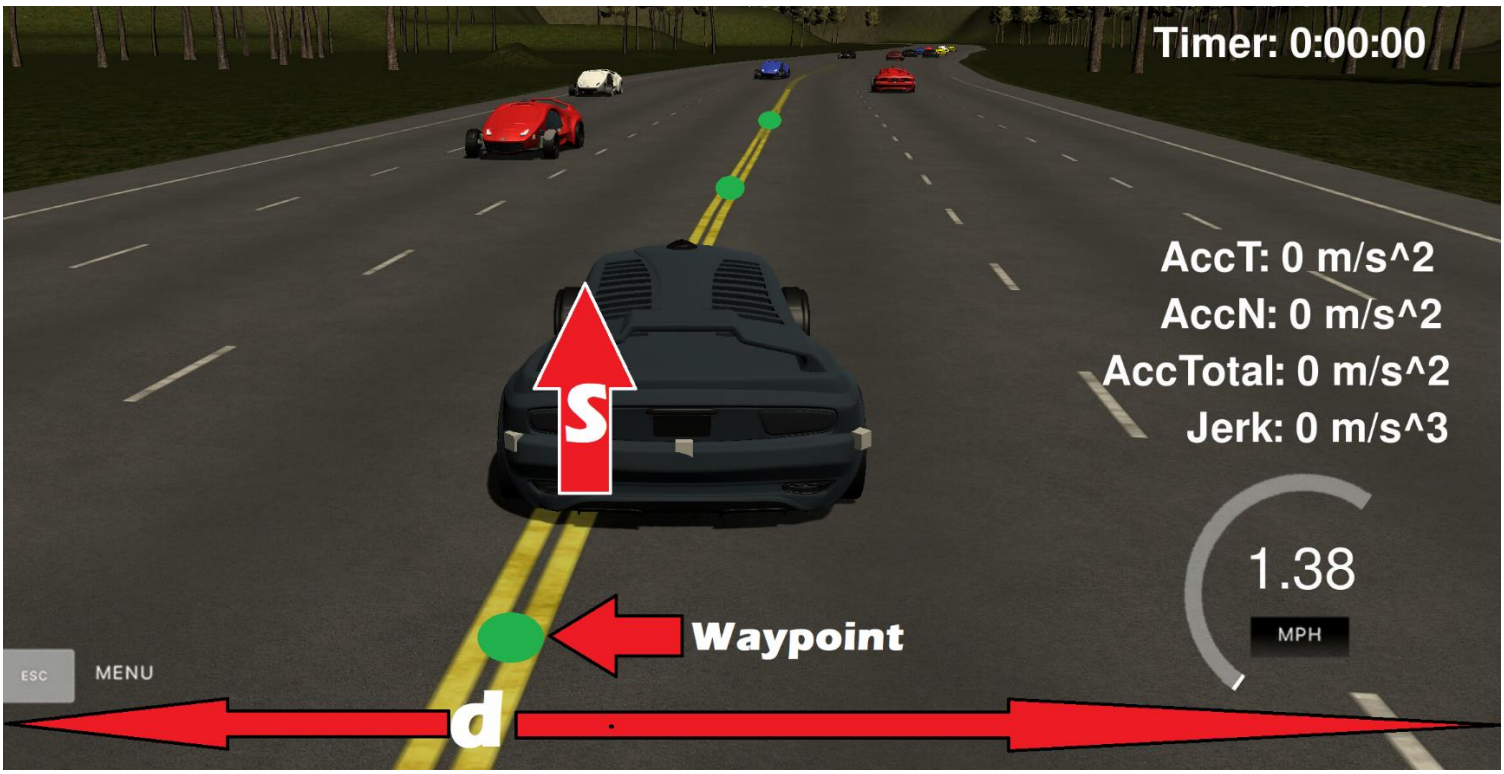
Path Planning Project



Report Prepared by Mwesigwa Musisi-Nkambwe for Udacity Self-Driving Car Nanodegree.

Objective:

Path-planning is vital to autonomous vehicle operation, it enables a vehicle to travel a safe, shortest, and optimal path between two points.

The goal was to build a path planner that creates smooth, safe (no life threating jerk/G-Forces, spike in acceleration) trajectories for the car to follow while safely navigating a virtual highway with traffic. Along with sensor fusion data generated from Lidar/Radar the car transmits its location. Sensor fusion is used to estimate the location of all the other cars on the same side of the road.

By the process of discrete path planning the path planner outputs a list of x and y global map coordinates. Each pair of x and y coordinates are a point; these points form a trajectory. A spline (a spline is a special function defined piecewise by polynomials) is used to ensure that our car moves in smooth curvature.

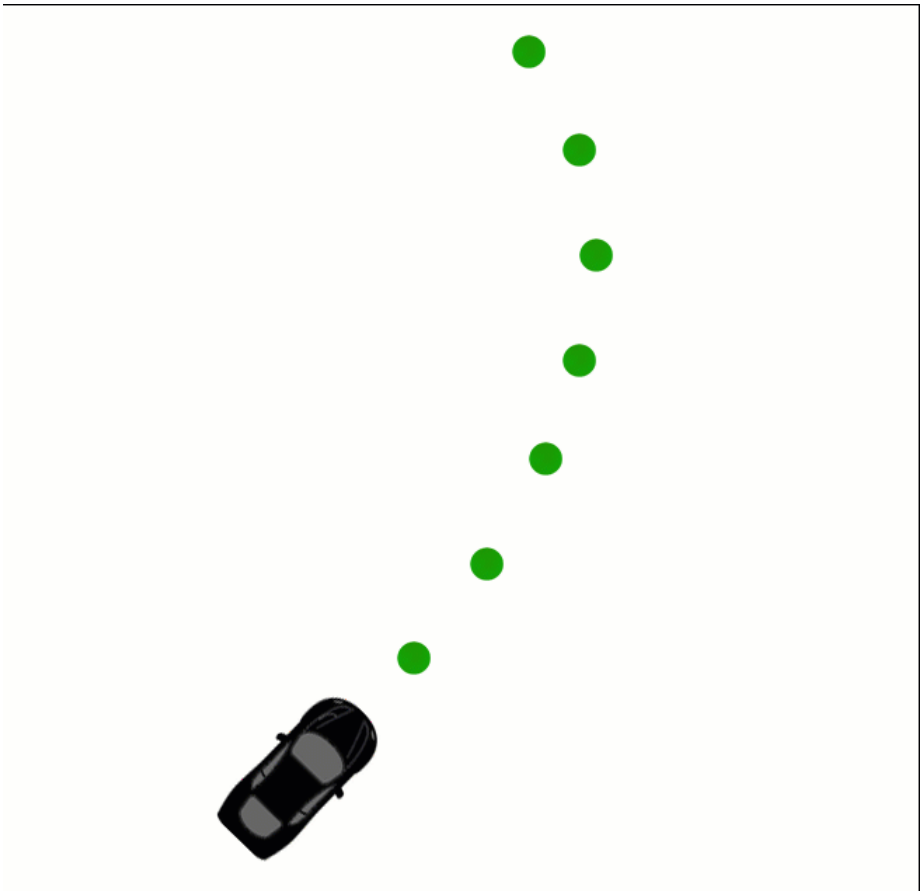


Figure 1 – Sample Trajectory Project Creates in Planning

The car moves from one point discrete to the next every 0.2 seconds.

The above image shows how the car moves from one point to the next. Acceleration is calculated by comparing the rate of change of average speed over 0.2 second intervals.

By simply increasing or decreasing point path spacing based on car_speed variable we can change the acceleration and diminish jerk.

Sensor Fusion

The **sensor_fusion** variable that is utilized throughout the project contains all the information about the cars on the right-hand side of the road. The data for each car is data found in this variable is:

Variable	Description
id	Unique identifier for each car
x	x values in global map coordinates
y	y values in global map coordinates
vx	x velocity component in global map reference
vy	y velocity component in global map reference
s	s frenet coordinates
d	d frenet coordinates

Table 1 – Table detailing variables in Sensor Fusion Data Structure

The frenet coordinate system is used coordinate system is constantly rotating as an observer moves along the curve.

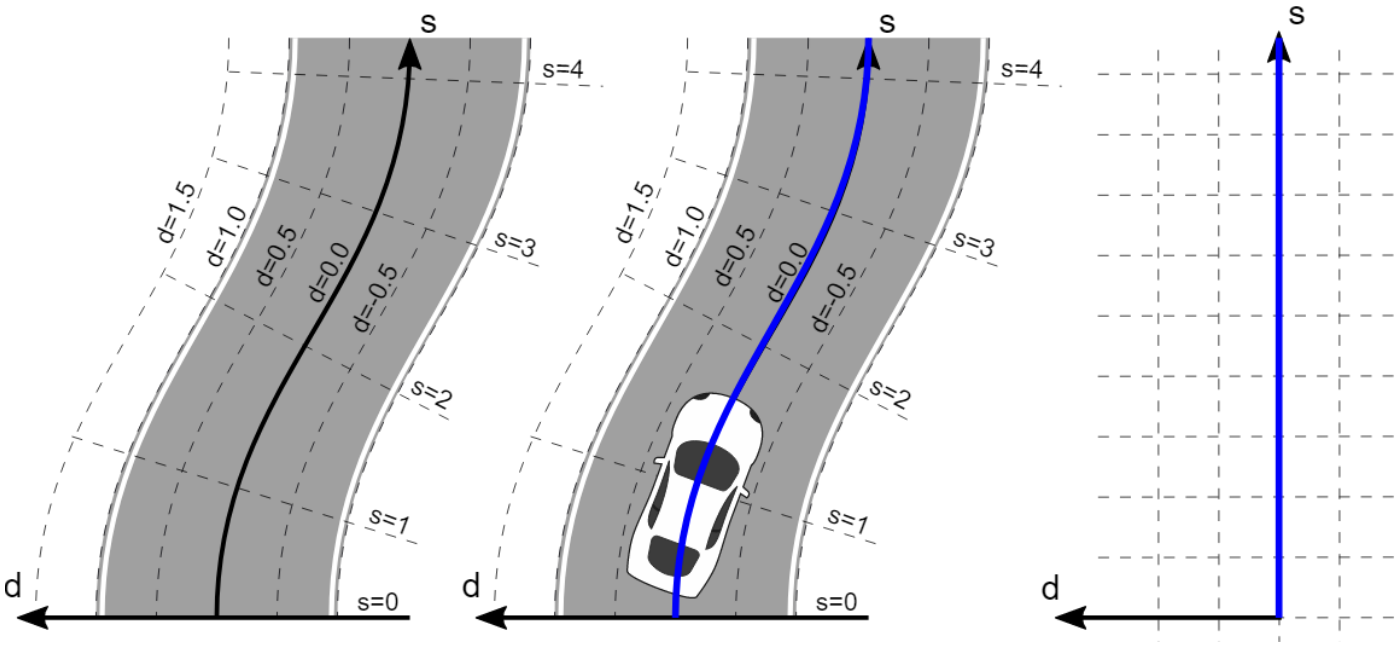


Figure 2 – Frenet Coordinate System used project.

(Image from https://fjp.at/posts/optimal-frenet/)

The file data/highway_map.csv is a list of waypoints found on the route we will following on the highway. The highway has a total of 181 waypoints. The waypoints are placed in the middle of the double-yellow line found in the center of the highway. A waypoint is made up of (x, y) global map position, with a Frenet s value and Frenet d unit normal vector (split up into the x component, and the y component).

The highway is made up of six 4m wide lanes (see image below), with 3 heading in each direction. Our car will be on the right-hand side. The car keeps its lane unless it is performing a lane change.

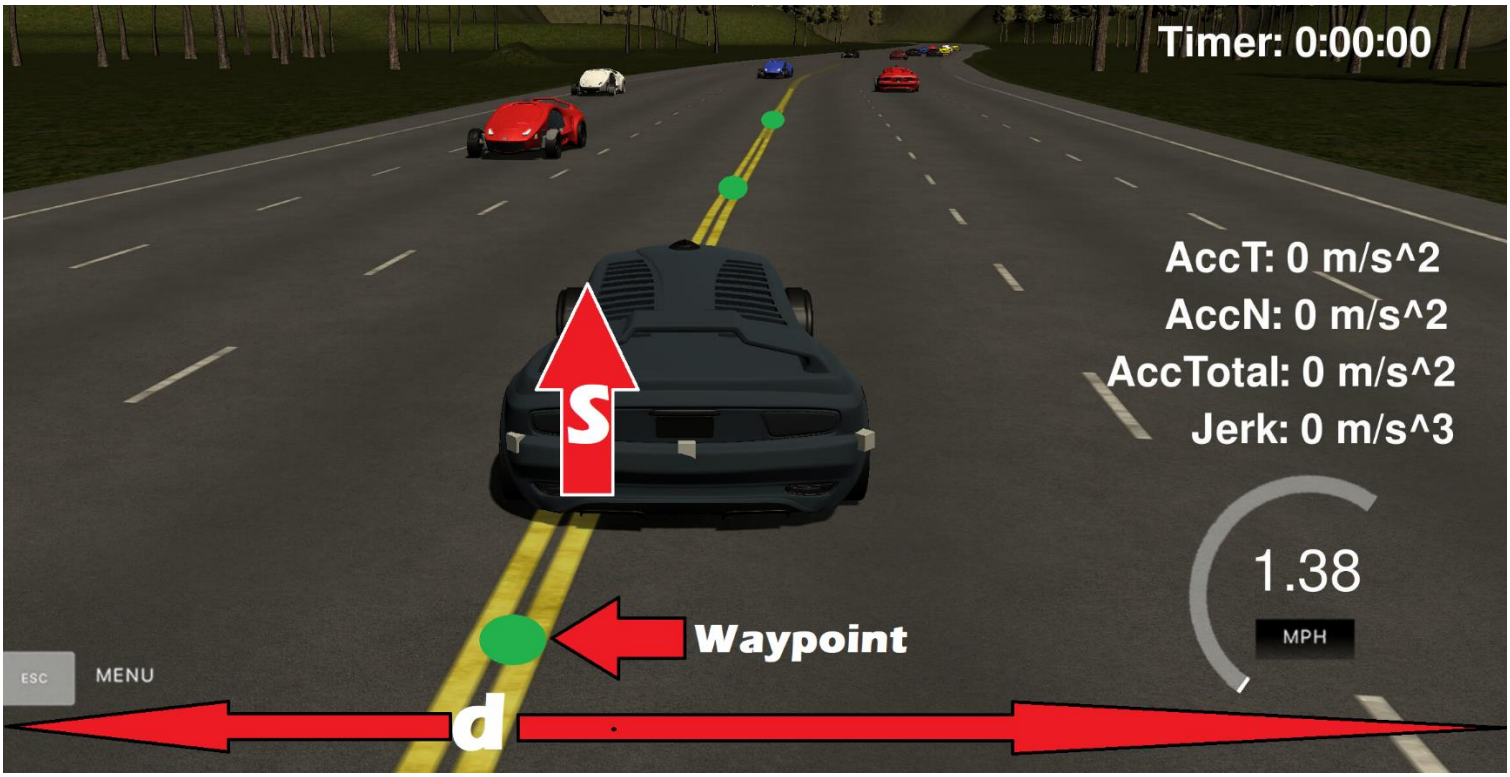


Figure 3 – Simulator screen showing how the frenet coordinate system is applied.

Prediction

Once the above data has been received, we preform prediction. we check to see if the previous path size is greater than zero.

I start of by declaring Boolean variables that are instrumental to this operation:

```
bool car_left= false;
bool car_right = false;
bool car_ahead = false;
```

Lanes are then checked using the frenet system as shown in figure 1.

For each vehicle in sensor_fusion, I check to see:

If (remember, each lane is 4m wide):
d is greater than **0** and smaller than **4**
Then the car is in the left most lane

If d is greater than **4** and smaller than **8**
Then the car is the middle lane

If d is greater than **8** and smaller than **12**
Then the car is the right most lane

```
//find ref_v to use
for(int i = 0; i < sensor_fusion.size(); i++)
{
  int check_lane;
  if(sensor_fusion[i][6] > 0 && sensor_fusion[i][6] < 4)
  {
    check_lane = 0;
  }
  else if(sensor_fusion[i][6] > 4 && sensor_fusion[i][6] < 8)
  {
    check_lane = 1;
  }
  else if(sensor_fusion[i][6] > 8 and sensor_fusion[i][6] < 12)
  {
    check_lane = 2;
  }
```

Figure 4 – C++ code used to identify what lane cars on the highways are in.

Once the above information has been acquired and stored (per vehicle, per greater for loop above) we enter an if statement within the for loop.

In the code below I take lane info acquired in Figure 2 and determine whether there is a vehicle ahead, to the left or to the right of the vehicle we wish to control.

Information collected in the code snippet above is them feed to the code below. This is used to make lane changing and acceleration/deceleration decisions. My “STATS FOR NERDS” feature outputs the processing done here to the console window.

```
//check s values greater than mine and s gap
if(check_lane == lane) // if in the same lane as ego car
{
  // Checking to see if there is a car within 30m ahead
  car_ahead = car_ahead | check_car_s > car_s && car_ahead | (check_car_s - car_s) < 30;
  std::cout << "Car # "<<i<<" is "<<(check_car_s - car_s)<<" ahead"<<"\n";
}
else if((check_lane - lane) == -1)
{
  // Checking to see if there is a car within 30m to the left
  car_left = car_left | (car_s+30) > check_car_s  && car_left | (car_s-30) < check_car_s;
  std::cout << "Car # "<<i<<" is to the left"<<"\n";
}
else if((check_lane - lane) == 1)
{
  // Checking to see if there is a car within 30m to the left
  car_right = car_right | (car_s+30) > check_car_s  && car_right | (car_s-30) < check_car_s;
  std::cout << "Car # "<<i<<" is to the right"<<"\n";
}
```

Figure 5 – Code used to determine where a vehicle is within 30 meters ahead of, to the left or to the right.

In the above code, **car_s** would be our car’s s frenet coordinate in the road.

Respectively **check_car_s** would be the corresponding s frenet coordinate for the current car (among others) on the highway.

Additionally, **lane** will always be the lane our car (that we are controlling) is in.

With this information (per car) put together with that that was collected in code snippet found in Figure 3; I first use **check_lane** to see if there is a car ahead of me. the lane we are checking is I methodically go thru

A **check_lane – lane** is then used to see if there is a car to the left or right of me.

This value (**check_lane – lane**) is –1 when there is a car to my left.

And (**check_lane – lane**) is 1 when there is a car to my right.

In all situations I am looking out 30m ahead, behind (negative s values) right and left.

My “STATS FOR NERDS” Feature shows this data being processed. This was a tremendous help when writing this part of the code, **hence I used the code to write the code.**

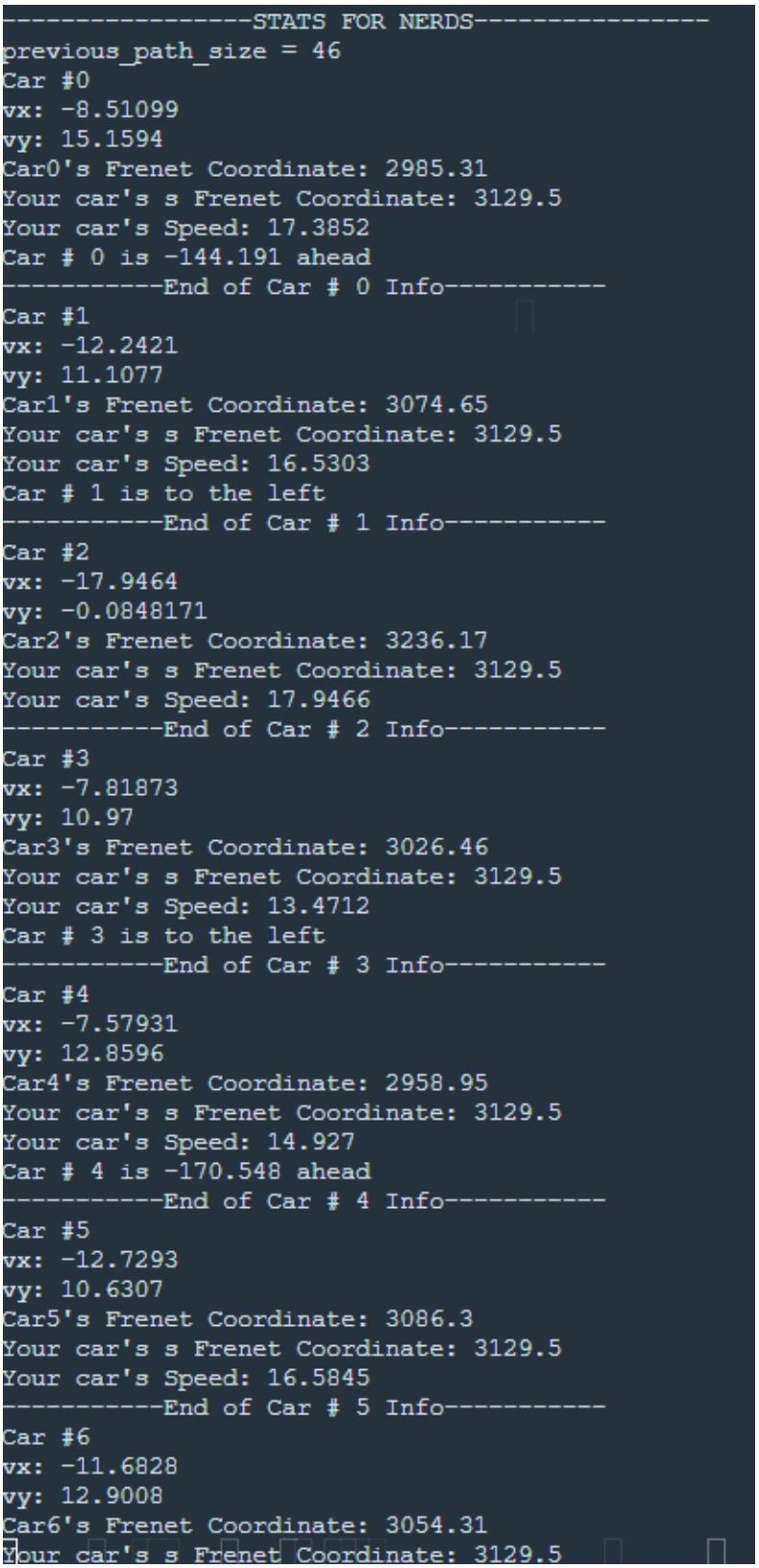


Figure 6 – Debug and bonus feature that shows info of all cars

Path Planning

Once the above logic has run thru, we are ready to decide what lane changing or acceleration/deceleration action we to take (or NOT).

A simple state machine that relies on the following logic showed below is implemented.

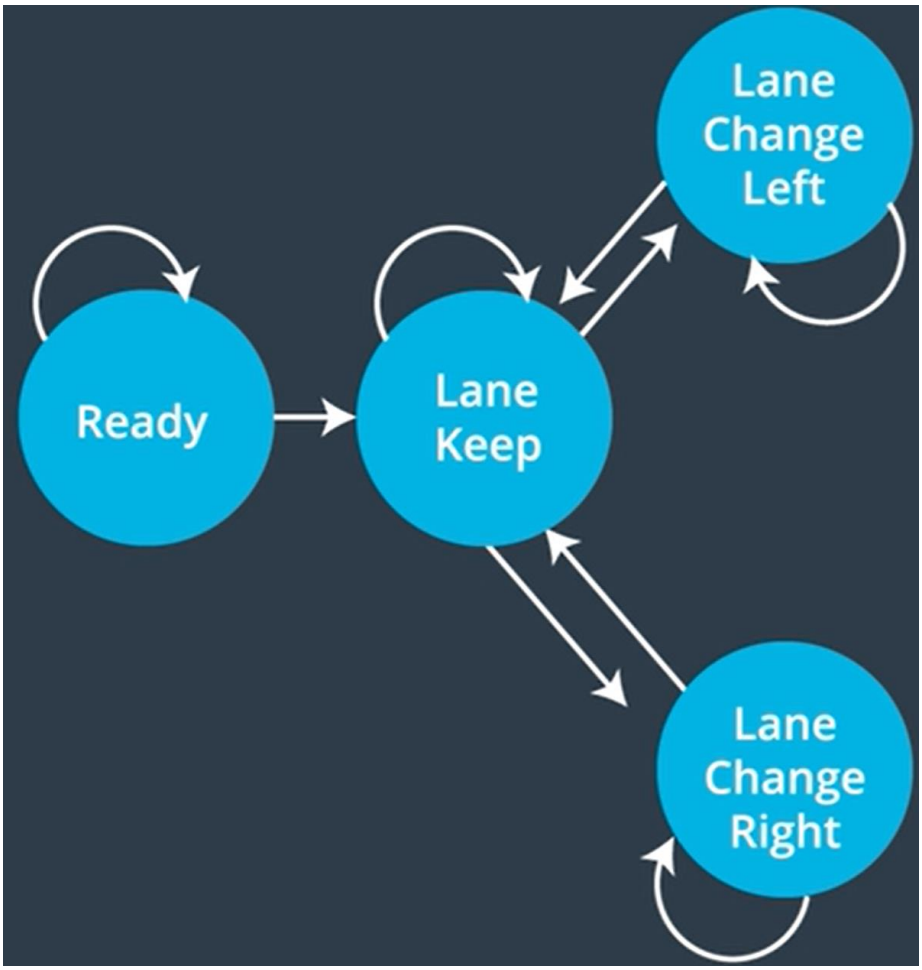


Figure 7 – State Machine showing decision making process when navigating traffic (Image courtesy of Udacity © 2011–2022)

The code used to implement this state machine can be found below:

```
if(car_ahead)
{
  if(!car_left && lane > 0)
  {
    lane--;
  }
  else if(!car_right && lane !=2)
  {
    lane++;
  }
  else if(!car_left && lane !=2)
  {
    lane++;
  }
  else
  {
    ref_vel -= .224;
  }
}
else if(ref_vel < 49.5)
{
  ref_vel += .224;
}
```

Figure 8 – Core functions of state machine in Figure 6 in code.

With all the above data the above code does the following

If there is a car ahead a decision to change lane or slow down needs to be made

We first cycle thru these options

If:

There is NO CAR in the LEFT LANE AND the LANE is not the LEFT MOST LANE move one lane to the LEFT

Else if:

There is NO CAR in the RIGHT LANE AND the LANE is not the RIGHT MOST LANE move one lane to the RIGHT

Else if:

There is NO CAR in the LEFT LANE AND the LANE is not the RIGHT MOST LANE move one lane to the RIGHT

If NONE of these are met, we decelerate by .224 mph (Slow deceleration with subsequent iterations that meets Jerk Requirement)

NOTE: Required Deceleration < 10 m/s^2 and Jerk < 10 m/s^3 (let's arrive alive).

Coming out of the nested If statement back to:

Else If there is no car ahead ← Corresponds to outer if statement

The speed is LESS THAN 49.5 (Meets Speed Limit Requirement), increase the speed by .224 mph (Slow acceleration with subsequent iterations that meets Jerk Requirement). With this new information the code then sets up the action that must be taken to safely navigate the highway.

The spline library is used to smooth out the disjoined path (dots above) the car travels. Implementing the Spline library enables the program to go thru all path planning points in a piecewise fashion.

The variables ref_x, ref_y and ref-yaw are used as a reference that would either be the starting point as to where the car is, or the previous paths end point.
The vectors ptsx & ptsy are x,y positions that we would fill up

```
double ref_x = car_x;
double ref_y = car_y;
double ref_yaw = deg2rad(car_yaw);

vector<double> ptsx;
vector<double> ptsy;
```

Figure 9 – Variables created for spline creation

The variable previous_path_size is used to see if there is a previous path that we can use as a starting point for our path forward. This is helpful in the transition from one pass thru from the server to the next. This is the last path that the path was following before it run thru this iteration.

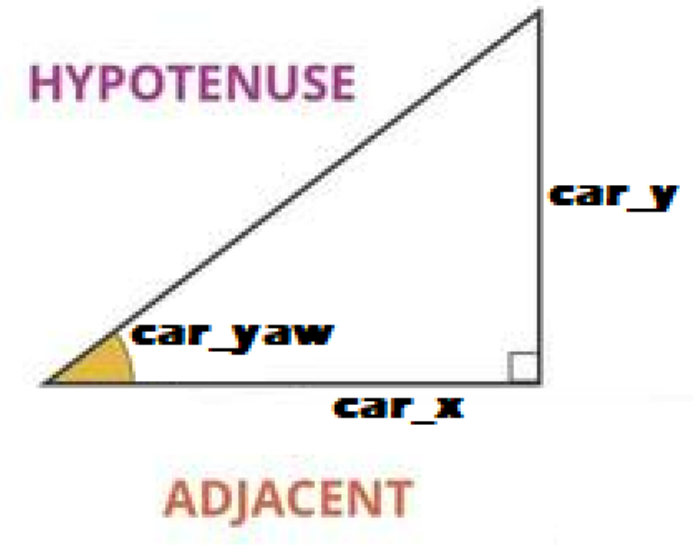
I first check to see if previous_path_size has points.

If previous_path_size does not have points or is close to empty (less than 2 points), I use the current car position with sin and consine to create a path tangent to where the car currently is.
I go “back in time” with 1 pseudo point (prev_car_x, prev_car_y) and 1 point of the current car position (car_x, car_y). The points that were collected & generated are then placed in vectors ptsx & ptsy giving us a heading direction.

If previous_path_size has more than 2 points. I use the current car position with sin and consine to create a path tangent to where the car currently is. I go “back in time” by 1 point (ref_x, ref_y) and even further, by 2 points (ref_x_prev, ref_y_prev). I get my angle ref_yaw and fill out ptsx & ptsy vector respectively, this give me 2 points. These two points placed in vectors ptsx & ptsy and ref_yaw give us a heading direction.

```
double ref_x = car_x;
double ref_y = car_y;
double ref_yaw = deg2rad(car_yaw);

vector<double> ptsx;
vector<double> ptsy;

// a close to empty previous path
if (previous_path_size<2)
{

  // Picking two previous points that are tangent to the car
 double prev_car_x = car_x - cos(car_yaw);
 double prev_car_y = car_y - sin(car_yaw);
 ptsx.push_back(prev_car_x);
 ptsx.push_back(car_x);

 ptsy.push_back(prev_car_y);
 ptsy.push_back(car_y);
}
else
{
 ref_x = previous_path_x[previous_path_size -1];
 ref_y = previous_path_y[previous_path_size -1];
 double ref_x_prev = previous_path_x[previous_path_size -2];
 double ref_y_prev = previous_path_y[previous_path_size -2];
 ref_yaw = atan2(ref_y - ref_y_prev, ref_x - ref_x_prev); //returns the angle θ between the ray to the point (x, y) and the positive x axis

 ptsx.push_back(ref_x_prev);
 ptsx.push_back(ref_x);

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

Figure 10 – code used to setup points and directions for initial spline positions.



$$\cos(car\_yaw) = \frac{car\_x}{hypotenuse}$$

Figure 11 - Concept behind "back in time" real and pseudo points.

With this starting reference I then proceed to map out the future points at 30m, 60m and 90m ahead using frenet. These points are then added to the ptsx & ptsy vectors. After this operation (below in code) the ptsx & ptsy respectively now have reference points and point at 30, 60 and 90 meters ahead.

```
double s_wp0 = car_s+30;
double d_wp0 = 2+4*lane;
double s_wp1 = car_s+60;
double d_wp1 = 2+4*lane;
double s_wp2 = car_s+90;
double d_wp2 = (2+4)*lane;

//Setup targets
vector<double> next_wp0 = getXY(s_wp0, d_wp0, map_waypoints_s, map_waypoints_x ,map_waypoints_y);
vector<double> next_wp1 = getXY(s_wp1, d_wp1, map_waypoints_s, map_waypoints_x ,map_waypoints_y);
vector<double> next_wp2 = getXY(s_wp2, d_wp2, 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]);

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

// tranformation to local car co-ordinates, car reference angle to 0 degrees
for(int i = 0; i < ptsx.size(); i++)
{

  //shift car refernece to 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));

}
```

Figure 12 – Code used to fill out points at 30m, 60m & 90m

A transformation is then made to the local car coordinates. The last point of the previous path is at (0, 0), (x, y) and the angle is at zero degrees simplifying future calculations.

```
// tranformation to local car co-ordinates, car reference angle to 0 degrees
for(int i = 0; i < ptsx.size(); i++)
{

  //shift car refernece to 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));

}
```

Figure 13 – Code used to switch metrics back to car's co-ordinate system (Starting point (0,0), 0 degrees.

Once all this data has been collected, we are ready to create out spline!

This is created with our ptsx & ptsy vectors (code below).

```
// Spline created!
tk::spline s;
// setting (x,y) points to spline
s.set_points(ptsx,ptsy);
```

Figure 14 – Code used to create spline once points have been realized.

I then proceed to fill out the actual points that the path planner will be using in vector variables next_x_vals & next_y_vals. I start with any points from the previous iteration, this helps with the transition; instead of recreating the path each time why not add points to it starting with what remains from the previous iteration.

```
vector<double> next_x_vals;
vector<double> next_y_vals;

for(int i = 0; i < previous_path_size; i++)
{
  next_x_vals.push_back(previous_path_x[i]);
  next_y_vals.push_back(previous_path_y[i]);
}
```

Figure 15 Code used to create initial points for Path Planner.

Please note the difference between what I will refer to now as **Anchors Points** that are created by the spline (pstx, ptsy, spline s). There are far spaced waypoints that make up the spline.

The **Future Path** (next_x_vals & next_y_vals) which would be our Path Planning points.

To meet the desirable speed, we then break up the spline (code below) by internally spacing out points in a deliberate way. This is done by Linearizing the spline; by looking out 30 meters (Figure 15) and using the s function that will give us a y with a value in x. I can call the distances to the target point along d and break up by path into N pieces in which N is travelled every .02 seconds.

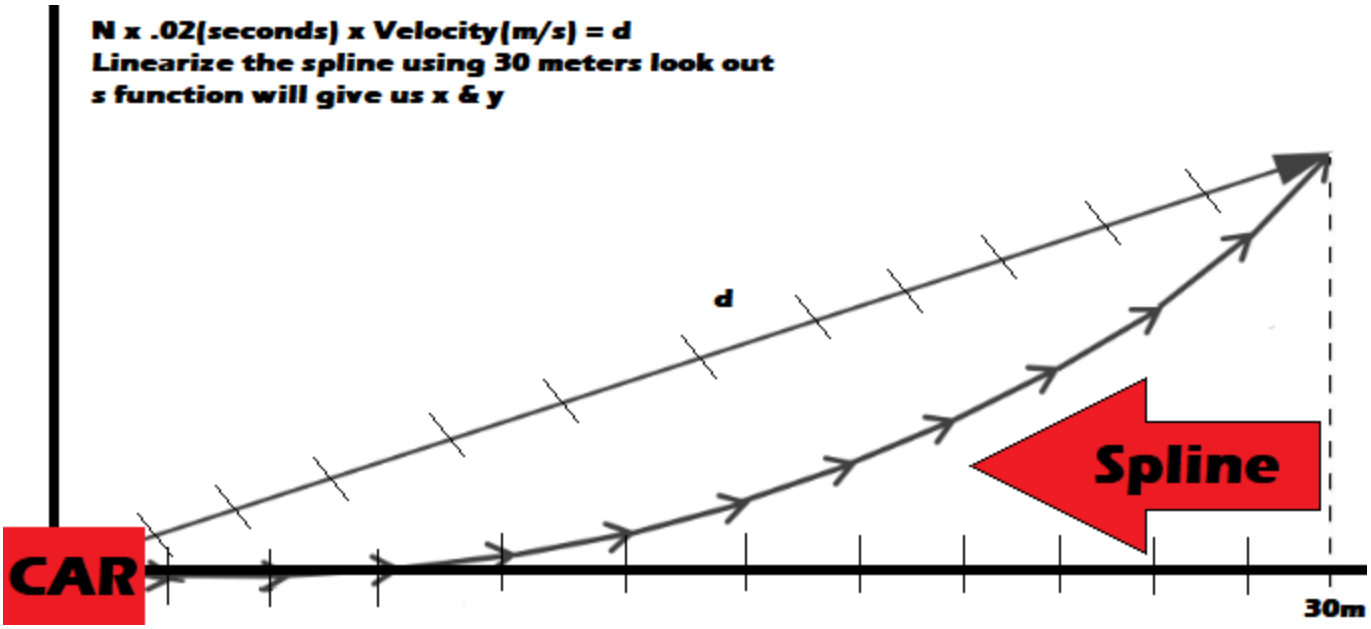


Figure 16 – Method used to space out points as to meet speed requirements.

In the code below calculations described above are processed and the co-ordinate system is switched from the local back to global one (a shift and a rotation). This is then pushed back into the next_x_vals and next_y_vals.

```
// Spline is broken up in order to meet desired speed
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;

double time_step = 0.02;

double dist_inc = 0.5;

for(int i = 1; i<= 50 - previous_path_size;i++)
{
  double next_s = car_s+(i+1)*dist_inc;
  double next_d = 6;


  double N = target_dist/(.02*ref_vel/2.24);
  double x_point = x_add_on+(target_x)/N;
  double y_point = s(x_point);

  vector<double> xy = getXY(next_s, next_d, map_waypoints_s, map_waypoints_x, map_waypoints_y);

  x_add_on = x_point;

  double x_ref = x_point;
  double y_ref = y_point;

   // rotating back to normal

  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);

}
```

Figure 17 – Code used to break up spline, rotate back to global co-ordinate system and insert values into path planning vector.