In this project, it is mainly to generate smooth paths without collision, and also to change lanes. The following two points are described:

1. Prevent collision from being too close to the vehicle in front
Calculate the distance between the vehicle and all vehicles. If there are vehicles within 30 meters in front of the lane, set the "too_close" sign to true; if there are vehicles within 30 meters in front of and behind the left lane, set the "turn_left" sign to true; if there are vehicles within 30 meters in front of and behind the right lane, set the "turn_right" sign to true.

```
bool too_close = false;
bool turn_left = false;
bool turn_right = false;

for (int i = 0; i < sensor_fusion.size(); i++){
 float d = sensor_fusion[i][6];
 double vx = sensor_fusion[i][3];
 double vy = sensor_fusion[i][4];
 double check_speed = sqrt(vx*vx + vy*vy);
  double check_car_s = sensor_fusion[i][5];
  check_car_s += ((double)prev_size*.02*check_speed);

  if (d<(4 * lane + 4) && d>(4 * lane) && (check_car_s > car_s) && ((chec
k_car_s - car_s) < 30)){
  too_close = true;
  }
  if (d<(4 * lane) && d>(4 * lane - 4) && abs(check_car_s - car_s) < 30){
  turn_left = true;
  }
  if (d<(4 * lane + 8) && d>(4 * lane + 4) && abs(check_car_s - car_s) <
30){
  turn_right = true;
  }
  }
```

When it is too close to the vehicle in front of the lane and there is no vehicle within 30 meters of the left lane, the vehicle changes to the left; when it is too close to the vehicle in front of the lane and there is no vehicle within 30 meters of the right lane, the vehicle changes to the right.

```
if (too_close && (!turn_left)&&(lane>0)){
```

```
 lane -= 1;
}
if (too_close && (!turn_right) && (lane<2)){
 lane += 1;
}
```

If the distance is too close, reduce the speed.

```
if (too_close){
 ref_vel -= .224;
}
else if (ref_vel<49.5){
 ref_vel += .224;
}
```

2. Produce smooth paths to prevent bumps

If the path is directly regenerated, the vehicle will generate a large bump during the process of switching between the last path and the current path. If the last navigation points of the last path can be used to join the current path, then the last path and the current path can smoothly transition.

Therefore, the first step is to extract the remaining navigation points of the last path. If there is no last navigation point, then use the front navigation points recorded by the simulator to fill in the new list.

```
if (prev_size<2){
 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[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);
```

```
19
20   ptsx.push_back(ref_x_prev);
21   ptsx.push_back(ref_x);
22
23   ptsy.push_back(ref_y_prev);
24   ptsy.push_back(ref_x);
25  }
```

The created waypoints with large interval and are filled in the new list as partial points of the new path.

```
1  vector<double> next_wp0 = getXY(car_s + 30, (2 + 4 * lane), map_waypoints
_s, map_waypoints_x, map_waypoints_y);
2  vector<double> next_wp1 = getXY(car_s + 60, (2 + 4 * lane), map_waypoints
_s, map_waypoints_x, map_waypoints_y);
3  vector<double> next_wp2 = getXY(car_s + 90, (2 + 4 * lane), map_waypoints
_s, map_waypoints_x, map_waypoints_y);
4
5  ptsx.push_back(next_wp0[0]);
6  ptsx.push_back(next_wp1[0]);
7  ptsx.push_back(next_wp2[0]);
8
9  ptsy.push_back(next_wp0[1]);
10  ptsy.push_back(next_wp1[1]);
11  ptsy.push_back(next_wp2[1]);
```

Translate and rotate the points in the new list to convert the coordinates of the car to the origin and the reference angle to 0 degrees.

```
1  for (int i = 0; i<ptsx.size(); i++){
2   double shift_x = ptsx[i] - ref_x;
3   double shift_y = ptsy[i] - ref_y;
4
5   ptsx[i] = (shift_x*cos(0 - ref_yaw) - shift_y*sin(0 - ref_yaw));
6   ptsy[i] = (shift_x*sin(0 - ref_yaw) + shift_y*cos(0 - ref_yaw));
7  }
```

In order to generate smooth paths, we can use the quintic polynomials in the course or spline curves. Because the generation of spline curves is relatively simple, we can use spline curve library to generate smooth paths.

```
1  tk::spline s;
2  s.set_points(ptsx, ptsy);
```

Breaks the point of the spline so that the planner output is 50 points.

```
1  double target_x = 30.0;
2  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 - previous_path_x.size(); i++){

 double N = (target_dist / (.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;
 }
```

Then translate and rotate to global coordinates, and output the waypoints to the controller.

```
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);
```