Apollo EM planner逻辑总结

* 基于3.0版本的总结

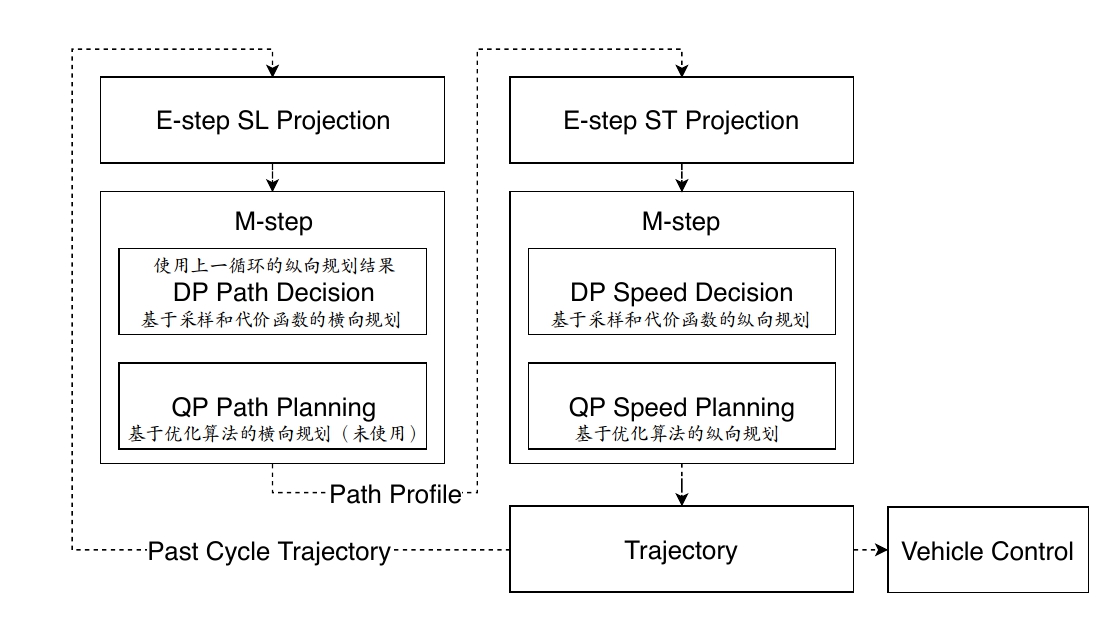
1. 概述

程序入口位于planner/em/em\_planner.cc中的EMPlanner::Plan函数。

对于每一条参考线（车道），都会按所有流程计算一遍cost，然后选出cost最低的车道；布尔量prioritize\_change\_lane（3.0为false）控制着是否给变道路径更高的优先级，如果设为true，且某条变道路径的cost小于阈值（10），则不再遍历其他的路径，直接返回此变道路径。

车道线内的规划逻辑在PlanOnReferenceLine函数中，需要次序执行5个任务，它们按顺序定义在planning\_config.pb.txt中，依次是 DP\_POLY\_PATH\_OPTIMIZER（横向规划）、PATH\_DECIDER（横向决策）、DP\_ST\_SPEED\_OPTIMIZER（基于采样的纵向粗略规划）、SPEED\_DECIDER（纵向决策）、QP\_SPLINE\_ST\_SPEED\_OPTIMIZER（基于优化的纵向细致规划）。

在一个规划周期中，执行顺序是先横向再纵向；但在横向规划中，利用了上一周期的纵向规划结果，因此实际上是横纵向规划交替进行、互相迭代。



1. 横向规划
2. DP规划

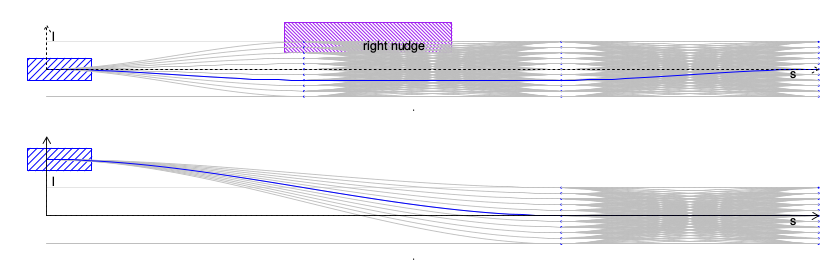
在此版本中，横向规划仅采用了基于DP（动态规划）的规划方法，而没用基于QP（二次规划）的方法。当时github上官方的回答是DP的结果已经足够好了，QP求解器又有时会产生无解的结果，因此跳过了QP步骤。但在Apollo的后续版本的PIECEWISE\_JERK\_PATH\_OPTIMIZER中，似乎又再次采用了基于QP的规划。（是否后续版本有改进？）

横向规划首先需要速度规划结果speed\_profile，这里基本直接使用了上一帧的速度规划结果，只是s和t要减去一截（两帧之间的行驶距离和时间）。如果speed\_profile为空，如激活AD之后的第一帧，则设为以当前速度（最低5m/s，最高31m/s）匀速行驶8s的一段speed\_profile。

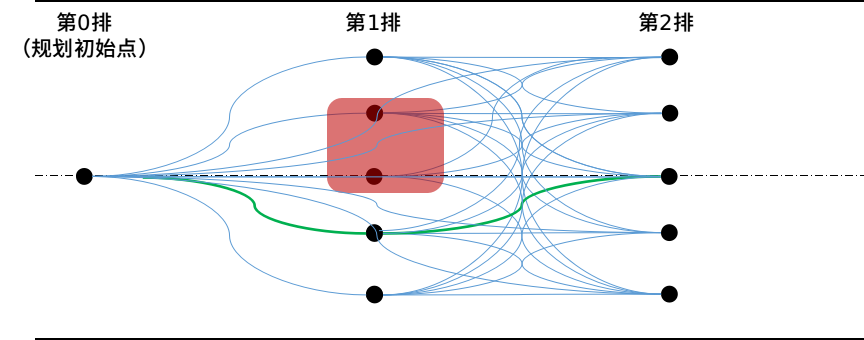
DP横向规划主逻辑在FindPathTunnel中，后者的主要逻辑在GenerateMinCostPath。

第一步是撒点SamplePathWaypoints， 纵向上，撒点总长度为40m和8s行进距离（按当前车速，下同）中的较大值；撒点的间距为4s行进距离，且限制在20m-40m之间，因此撒点至少；当车速较低时，撒点间距减半。横向上，撒点个数为7个，宽度介于1.2m和3.5m之间，且当前横向速度越高，撒点宽度也越宽。如果当前是pull\_over工况，则采样点只有一个，即为目标停车位置。

默认撒点区域位于车道内；若为变道工况，则横向撒点区域扩大到当前横向位置，即从当前的d开始，但目标车道的较远的boundary为保证撒点间隔和个数则相应缩紧。所有撒的点存入path\_waypoints数组中。



第二步是连线并计算cost，循环遍历每个撒下的点，将这个点与上一级的所有点分别用五次多项式连接，如果不是第一级，那么还要尝试和起始点直接连接。



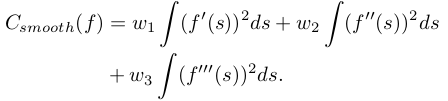
如上图所示，第二排的点不仅和第一排的相连，还要和初始点相连，对于之后各排也是如此。

第三步是计算每个node的cost，这里动态规划的思想体现在每个node的最小cost都设定为：与上一层所有node连线路径的cost，加上路径所连接的节点的最小cost；这样，每次向下一层推进时，就不必再遍历之前的每一层了，只需遍历上一层的每个点即可，上一层的节点已经包含了过去的所有信息。

每条路径的cost由三部分组成：一是路径代价；二是静态障碍物代价；三是动态障碍物代价。



路径代价实际是多项式代价，是Cost\_smooth和Cost\_guidance之和，但代码中实际没有dddl的cost，计算代码在CalculatePathCost中；



静态障碍物代价中考虑了静态障碍物和自行车、行人，计算过程在GetCostFromObsSL中。离障碍物横向小于0.5m的路径，HAS\_COLLISION会标记为true。此外，凡是横向最近点距离在1m之内的，且位于自车前方的障碍物，都要计算cost。计算的依据是自车和障碍物的中心点横向、纵向距离，并用Sigmiod进行归一化（为什么不考虑障碍物的大小？）。

动态障碍物的cost权重很低为1e-6，也即在路径规划中忽略动态障碍物，而是在速度规划中进行应对和决策：超车、让行、停车等。

在计算动态障碍物的cost过程中，因为是对每一段分段路径计算的，所以必须从0时刻开始循环，找到s处于这一段之间的时刻，每个时刻的s、v等都是来自于上一次纵向规划的结果。每个时刻计算ego与障碍物之间的box距离，具体而言，是所有顶点到另一个box的所有边之间的距离的最小值。根据此距离利用Sigmoid归一化作为cost。

最后一步就是从最后一排的所有node中找出cost最低的，然后找到其来自的node，依次进行，直到达到规划初始点；此cost最低的路线即为min\_cost\_path。

DP横向规划完成后，按config\_.path\_resolution()（1m）离散化后通过SetFrenetPath存入frenet\_path\_中，供下一环节使用。

1. 路径决策PathDecider
2. QP规划

由于DP规划是基于sampling的方式，肯定不是最优解，因此对path用QP进一步优化。

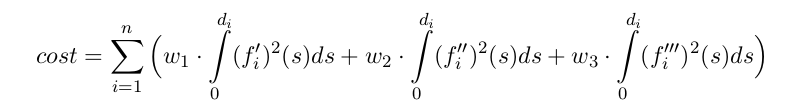
首先将整条path按最长20m的间距划分为若干段，每一段用5阶多项式表示，。



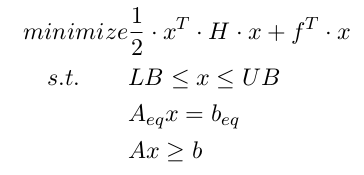
式中di为每一段的长度，多项式系数ai为优化的对象，优化含有多个目标，目标函数的定义在AddKernel()中：

* 优化目标函数

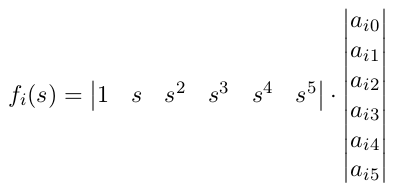
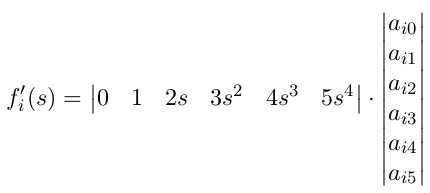
1. 优化后的曲线尽量接近参考线（道路中心线），AddReferenceLineKernelMatrix()，即最小化(fi(s) -g(s))^2；注意这里与第四项的实现不同，没有积分，因此只是考察控制点的距离；
2. 与上个循环的轨迹尽量接近，AddHistoryPathKernel()，这一项实现又调用了第一项cost的函数（参考线），区别是将上次的横向path作为参考线；
3. 正则化项，即ai0、ai1……ai5尽量小，AddRegularization()。
4. 曲线上各点的横向速度、加速度、jerk最小化，因此需要积分，此项优化目标为：



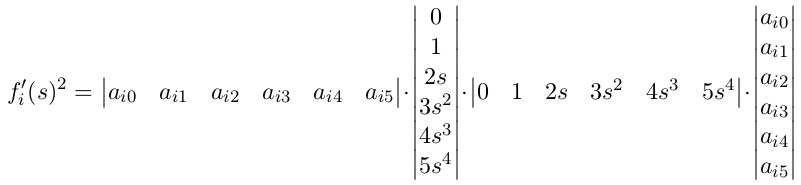
需要将cost转化为二次型标准形式：



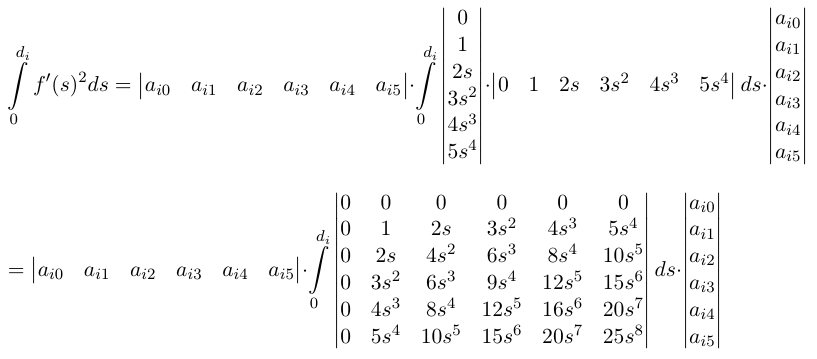
步骤如下：

，

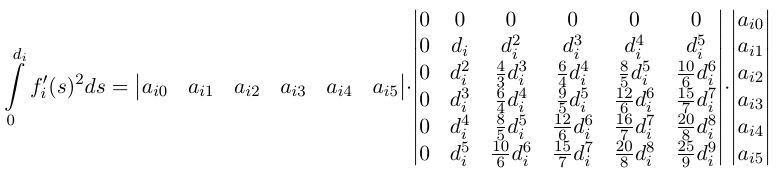
那么，



将与s无关的项提出，并积分：



最终得到：

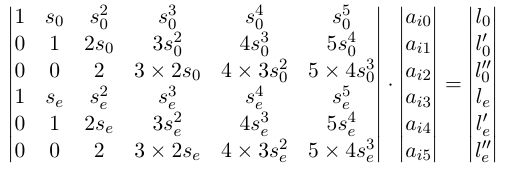


用类似的方法可以得到二阶导数和三阶导数的二次项形式，各阶导数的对应矩阵加起来就是最终的矩阵。

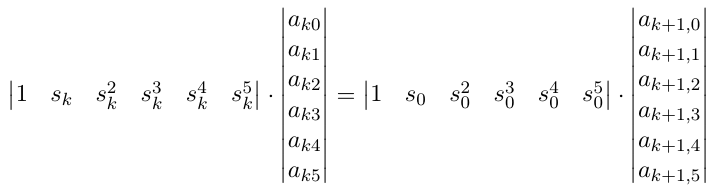
* 约束函数

逻辑在AddConstraint()函数中。

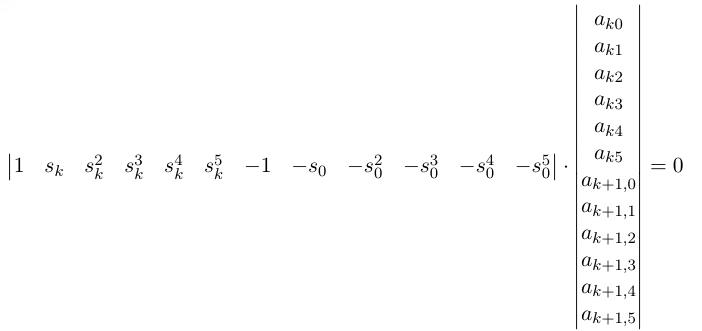
1. 整条轨迹起点、终点的约束（当前程序要求终点位于道路中心线上，横向偏差0.01以内，可以修改），可以用矩阵方便地表示为：



1. 控制点连续性约束，即：



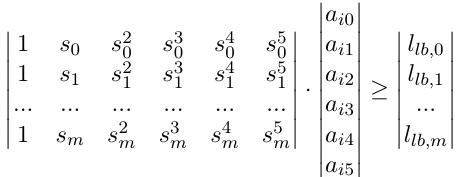
即：

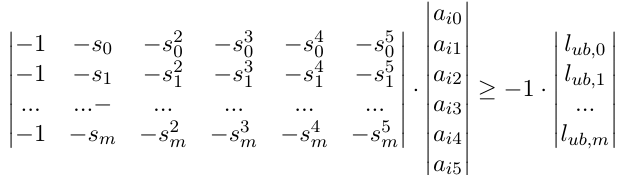


式中s0实际为0，代表线段的起点；类似的，有：



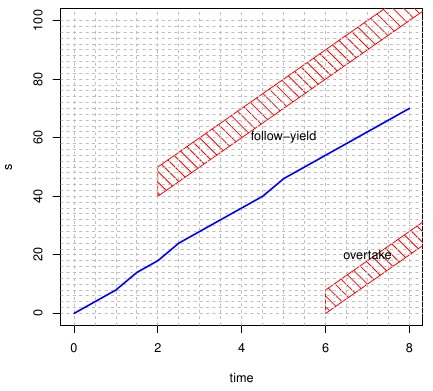
1. Theta（±0.1rad）、kappa（±0.2）、kappa\_d（±0.02）约束。
2. 边界约束；沿着path，按照s的最大间距max\_constraint\_interval（2m）采样，设采样点为m个，有：





1. 纵向规划
2. DP规划

DP速度规划的主要逻辑位于SearchStGraph()中，其总体逻辑为：将障碍物在s-t图上投影，然后用动态规划和cost的方法找出最优路径，其思路与横向DP规划相近。



1. QP规划