Apollo EM planner逻辑总结

* 基于3.0版本的总结

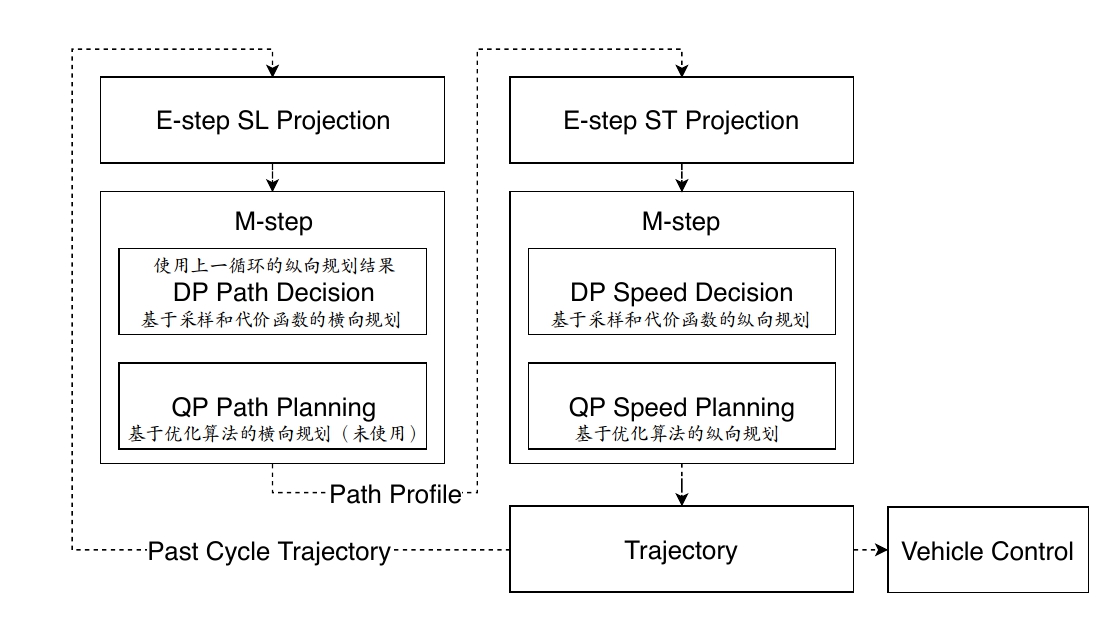
1. 概述

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

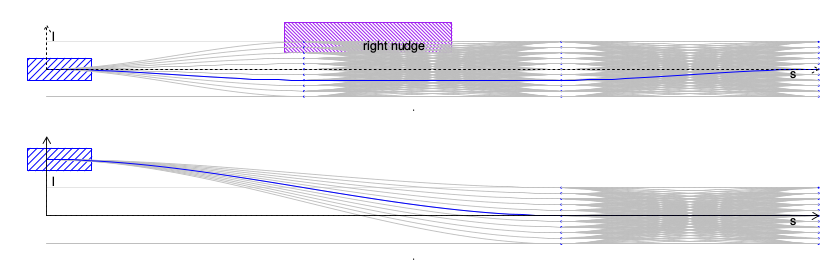
在此版本中，横向规划仅采用了基于DP（动态规划）的规划方法，而没用基于QP（二次规划）的方法。当时github上官方的回答是DP的结果已经足够好了，QP求解器又有时会产生无解的结果，因此跳过了QP步骤。但在Apollo的后续版本的PIECEWISE\_JERK\_PATH\_OPTIMIZER中，似乎又再次采用了基于QP的规划。（是否后续版本有改进？）

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。

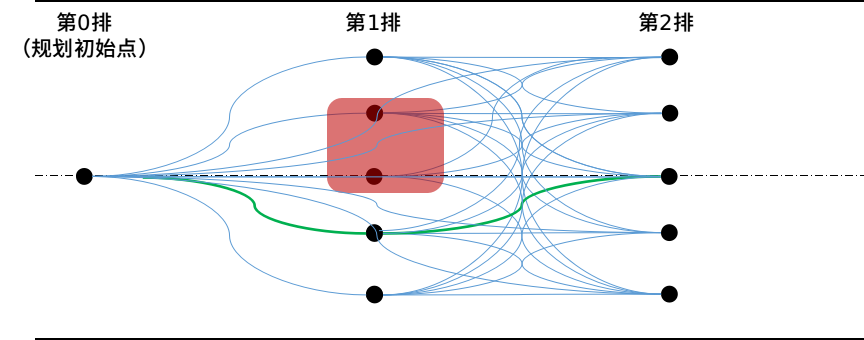
DP横向规划主逻辑在FindPathTunnel中，后者的主要逻辑在GenerateMinCostPath。

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。



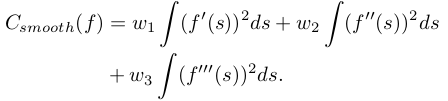
如上图所示，第二排的点不仅和第一排的相连，还要和初始点相连，对于之后各排也是如此。

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

路径决策只考虑静态障碍物和自行车、行人，而其他车辆等动态障碍物由纵向规划器进行变道超车、跟随、停车等决策。

对于障碍物，横向距离超过4m的忽略，小于0.5m的停车等待，介于之间的则进行“左侧绕过”或“右侧绕过”的决策，从而保证QP的优化求解在凸集中进行。

1. 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。QP求解可以进行2次，第一次如果无解，则对road\_boundary进行左右各0.5m放宽（车道保持路线，如为变道路线本来就放宽了）再进行第二次求解。

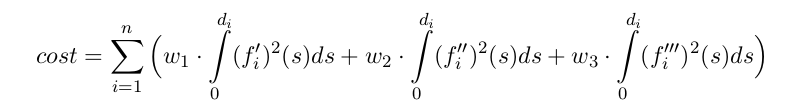
首先将整条path按最长20m的间距划分为若干段，每一段用5阶多项式表示：



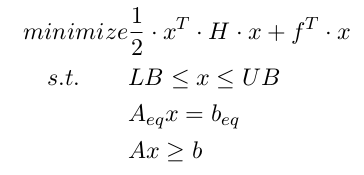
式中di为每一段的长度，多项式系数ai为优化的对象，优化含有多个目标，目标函数的定义在AddKernel()中：

* 优化目标函数

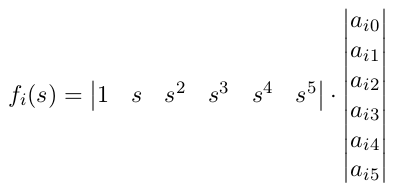
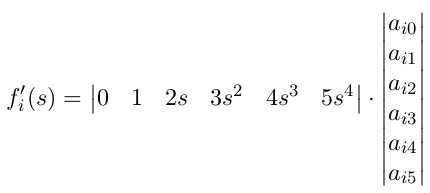
1. 优化后的曲线尽量接近参考线（道路中心线），AddReferenceLineKernelMatrix()，即最小化(fi(s) -g(s))^2；注意这里与第四项的实现不同，没有积分，因此只是考察分界点的距离；
2. 与上个循环的轨迹尽量接近，AddHistoryPathKernel()，这一项实现又调用了第一项cost的函数（参考线），区别是将上次的横向path作为参考线；
3. 正则化项，即ai0、ai1……ai5尽量小，AddRegularization()。
4. 曲线上各点的横向速度、加速度、jerk最小化，因此需要积分，此项优化目标为：



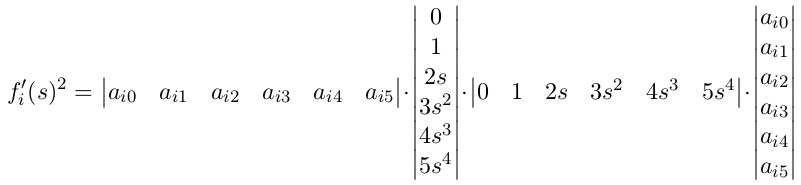
需要将cost转化为二次型标准形式：



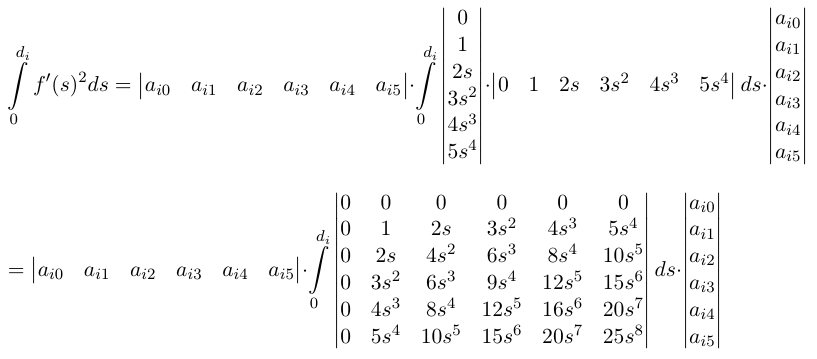
步骤如下：

，

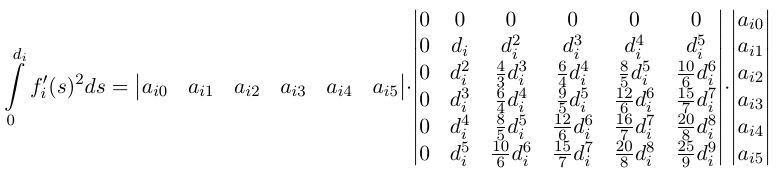
那么，



将与s无关的项提出，并积分：



最终得到：

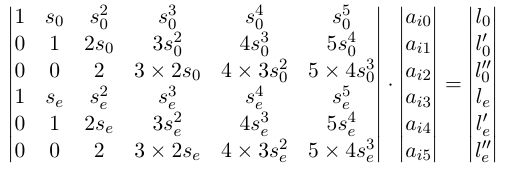


用类似的方法可以得到二阶导数和三阶导数的二次项形式，各阶导数的对应矩阵加起来就是最终的矩阵。

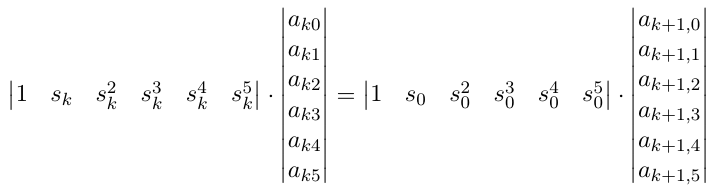
* 约束函数

逻辑在AddConstraint()函数中。

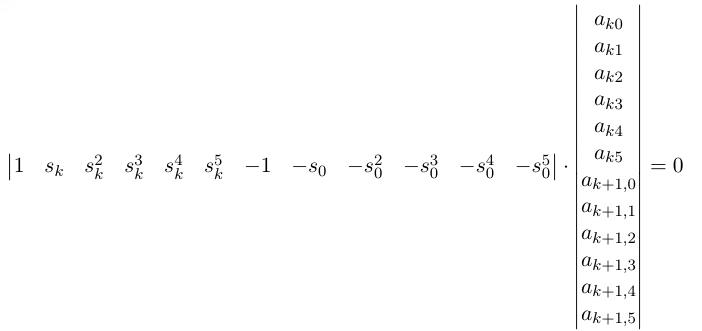
1. 整条轨迹起点、终点的约束（当前程序要求终点位于道路中心线上，横向偏差0.01以内，可以修改），可以用矩阵方便地表示为：



1. 分界点连续性约束，即：



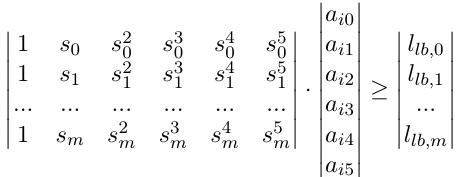
即：

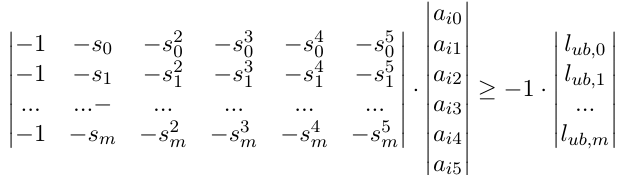


式中s0实际为0，代表线段的起点；类似的，有：



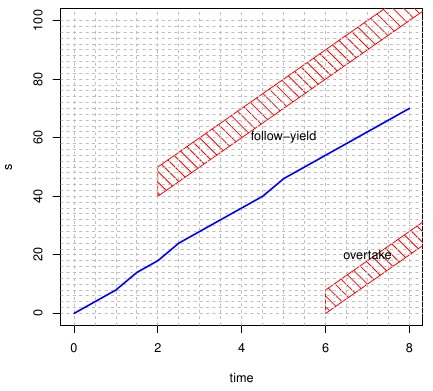
1. Theta（±0.1rad）、kappa（±0.2）、kappa\_d（±0.02）约束。
2. 边界约束；沿着path，按照s的最大间距max\_constraint\_interval（2m）采样，左右边界的boundary取决于上一步骤的RIGHT或LEFT\_NUDGE，设采样点为m个，有：





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，根据横向规划的路径（限速牌、kappa、kappa\_d等）计算限速信息，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。



获得s-t图之后，首先判断起始点(0, 0)是否处于障碍物的覆盖下，如果起始点都不可通行，则直接返回全0的speed\_profile；其次，判断障碍物boundary是不是空的，如果没有障碍物，则返回一条匀速1m/s的速度曲线。

如果不属于这两种情况，则如上图所示，在InitCostTable()函数中，初始化一张150m\*8s的s-t图，分辨率分别为1m和1s，每个点的cost初始化为+Inf，(0, 0)点cost为0。

接下来是速度规划的核心：CalculateTotalCost()函数，即计算上述s-t图上各点的cost；从起始点(0, 0)出发，按1s的间隔向后搜索s-t点。为了减小运算量，搜索s-t点时采用了剪枝策略；即根据上一时刻的速度和位置、考虑到车辆的最大加速度、v>=0的限制，则下一时刻可能的位置位于一定的s区间内，而不必遍历所有的s。例如，对于第3s，如最大s为20，速度为5m/s，加速度∈[-3， 3]，下一时刻只需要遍历(4, 0)到(4, 28)之间的点。

每个点都与上一时刻有可能到达当前点的点进行连线，每条线都有一个cost，包含以下几方面：

①在t时刻、s距离上，当前点是否处于障碍物的投影中，如有碰撞，则cost为+Inf；

②连线是否穿过障碍物的投影，如有则cost为+Inf；

③连线的acc是否超限值，acc = (S(k+1) + S(k-1) - 2\*S(k))/t^2；

④速度cost，超过限速或不达到限速均有cost，与速度差的绝对值成正比；

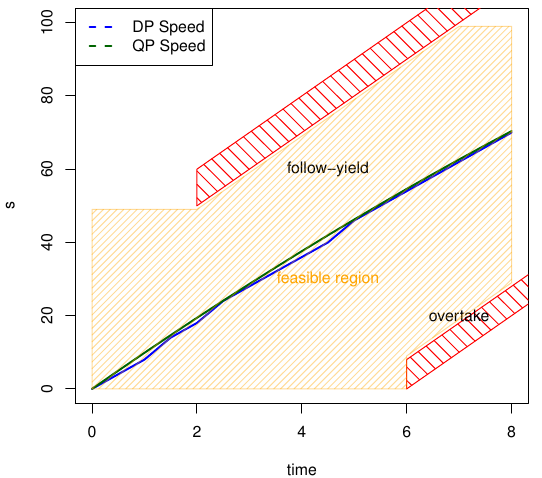
⑤加速度cost，与加速度平方成正比；

⑥jerk cost，与jerk平方成正比。

将上一时刻所有可能的点遍历过之后，最小的连线cost+对应的pre\_point cost之和，作为当前点的cost；相应的点作为当前点的pre\_point。

完成最后一列（所有时刻）的cost计算之后，找到最后一列和最大一行中cost最小的点，并依次将其的pre\_point添加进vector中，即为最后得到的speed profile。（最大一行表示车辆达到horizon最远范围，如果6s就到达150m，cost还小于8s到达的profile，则6s是更好的profile）。

1. 纵向决策



根据DP纵向规划的结果，对每个障碍物做出决策；若ego在st图上位于障碍物之上，决策为超车；如果ego在障碍物之下，决策为跟随或让行。

1. QP规划

DP规划的profile只精确到1s，并且只能保证s连续，不能保证v连续，因此，对DP优化出的速度需要QP再次细化；根据决策的信息，可以将优化的可行域缩小（超车或让行）成为凸集，从而使得QP变为可能。

和横向规划类似，纵向也有多次机会，第一次尝试利用acc∈[-3.3, 2.5]解决问题，若无解则放宽到acc∈[-4.0, 3.0]，第二次也无解则使用piecewise\_st\_graph；

纵向规划时，将总长7s的轨迹划分为4段，每一段用以t为自变量的5阶多项式表示。



※ 优化目标函数

①加速度最小化，AddSecondOrderDerivativeMatrix()

②jerk最小化，AddThirdOrderDerivativeMatrix()

这两项与QP Path的逻辑类似，对4段曲线分别求二阶、三阶倒数后求平方、积分；

③速度最快，AddCruiseReferenceLineKernel()

首先将曲线切分为30段，共有31个端点（时间点），将这些时间点进行①～③的目标优化，优化利用了QP Path中用过的AddReferenceLineKernelMatrix()，既无求导，也不对t积分，只是对端点进行优化；

每段的端点和最大距离250m的距离平方最小化，即车速最快目标；

④跟随，AddFollowReferenceLineKernel()

遍历30个时间点，每个时间点再遍历所有的obs\_boundaries，若有上一步骤中决策为Follow的障碍物，则这个t时间点的ref\_s为在t时刻，障碍物的尾部的s减去跟随距离。将这些端点与ref\_s距离的平方最小化，实现让ego跟随目标车辆；

若t时刻没有follow的障碍物，则不对t时刻作此项优化；

⑤让行，AddYieldReferenceLineKernel()

与跟随目标类似，遍历所有时间点，每个时间点遍历障碍物，若障碍物决策为Yield，则ref\_s为障碍物尾部的s减让行距离；

⑥与DP规划结果接近，AddDpStReferenceKernel()

DP规划结果的每个s-t点，与同一t时刻的s距离平方最小化；

⑦初始点jerk最小化；

⑧正则项，AddRegularization()

与QP Path类似，即ai0、ai1……ai5尽量小；

※ 约束函数

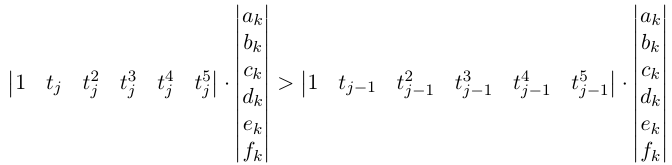
①初始位置、速度约束

AddPointConstraint(0.0, 0.0)，t=0时，s=0；AddPointDerivativeConstraint(0.0, init\_v)；t=0时，速度应当与实际速度相同；

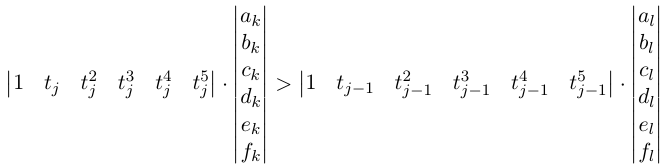
②单调性约束（不能倒车），AddMonotoneInequalityConstraint()

遍历30个时间点，比较其与前一个时间点的s值，应当大于；

首先要判断当前时间点与前一个时间点是否在同一段5阶曲线上，如果是同一段，那么有下式：



如果不是同一段，那么：

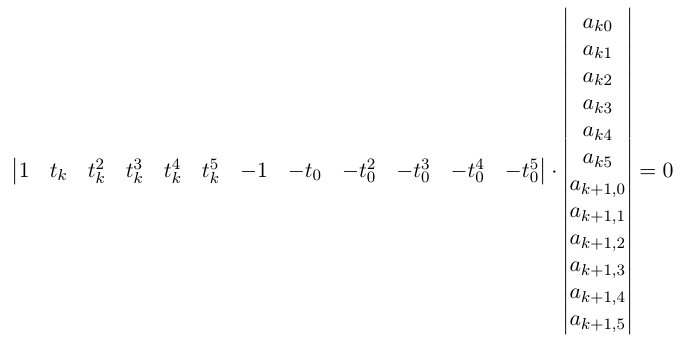


其中l是上一段的下标；

③分界点连续性约束，AddThirdDerivativeSmoothConstraint()



即有：

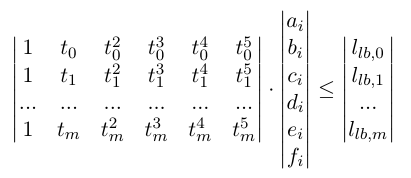
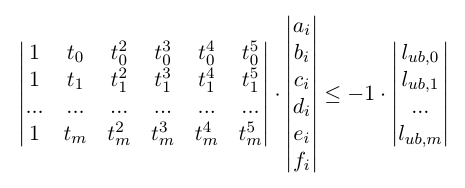


其中，t0 = 0，类似地，还有：



④边界约束（位置）

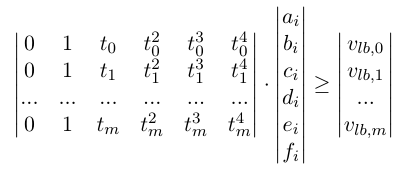
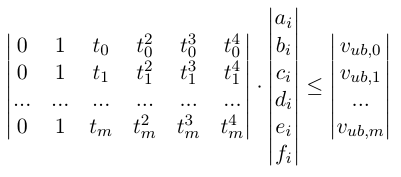
要求在s-t图上，ego不能和障碍物有重叠，该约束由函数包括GetSConstraintByTime()求出各个时刻的s上下界，然后通过AddBoundary()函数添加约束；约束作用在30个采样点上，速度和加速度约束也是如此：

，

⑤速度约束

速度约束的依据是根据Path的kappa、kappa\_d和地图信息计算出的速度限值，这些限值都以s为依据；因此，在s-t图上对时间t添加速度限值是不准确的，因为v和s是互相耦合的，即对于不同的v，时刻t对应的s也是不同的，因此，利用上一规划周期的速度规划结果，对t时刻的限速进行估计，结果应该会在几个循环之内收敛。

通过EstimateSpeedUpperBound()函数估计出速度的上下边界，利用AddDerivativeBoundary()添加约束，包括最高速度约束与最低速度（0）约束：

，

⑥加速度约束

如之前所述，第一次尝试求解时acc∈[-3.3, 2.5]，即为加速度限值。

与速度约束类似，只是再求一阶导数。

1. Case Study

