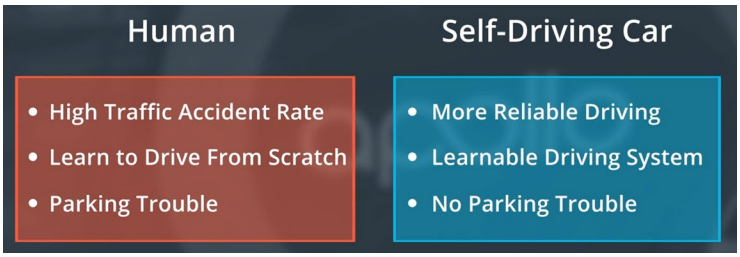
Self-Driving Cars

Apollo Note

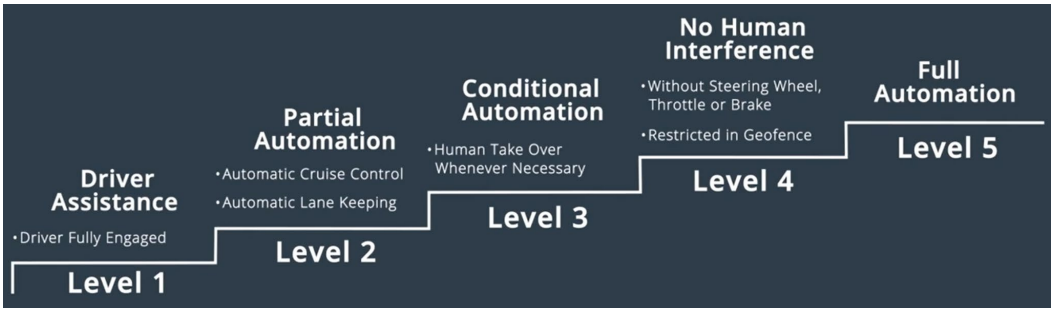
collated by decem17

Apollo 智能驾驶入门 https://apollo.auto/devcenter/coursetable_cn.html?target=1



不会喝醉，不会分心
可立即成为老司机
出租车 共享汽车

6 个等级 L0~L5
0 驾驶员是系统的唯一决策者
1 提供转向或加速支持 如巡航控制
2 自动巡航控制 车道保持 但驾驶员必须保持执行系统处理的任何功能
3 自动驾驶 但驾驶员必须准备在有必要时随时接管
4 地理围栏
5 在所有情况下应与人类驾驶员的水平一样或更高



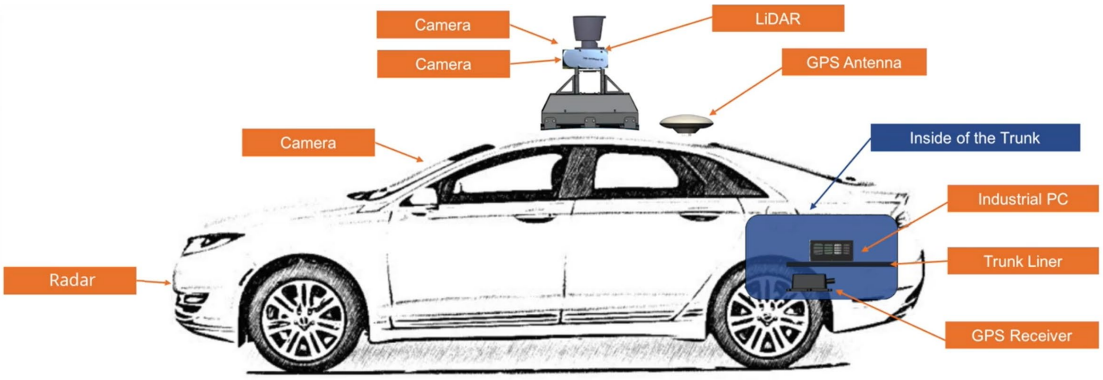
无人驾驶车辆由 专有计算机 和 传感器 组成，执行 感知 和 定位 等任务

无人驾驶车辆包含 5 个核心部分:

定位准确度 厘米级

硬件平台

线控驾驶车辆：控制器局域网络 CAN：车辆的内部通信网络，计算机系统通过 CAN 卡连接汽车内部网络．发送加速，制动和转向信号. Camera GPS IMU LiDAR IPC



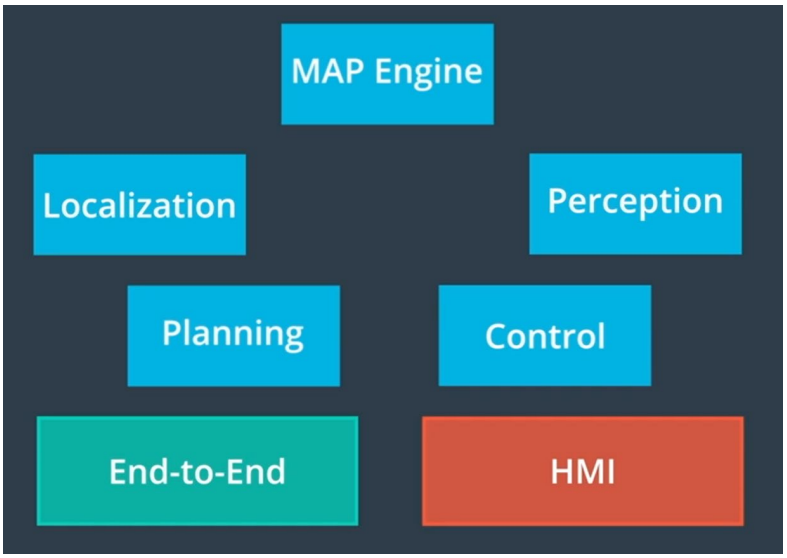
开源软件架构

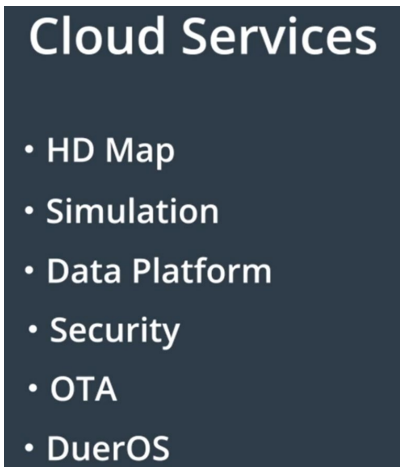
开放式软件层分为 3 个子层：实时操作系统 RTOS，运行时框架，应用程序模块层

实时操作系统 RTOS：确保在给定时间内完成特定任务(steering/brake/accelerator)，即在传感器收集到外界数据后的短时间内完成 实时性能确保系统稳定性和驾驶安全性的重要要求

运行时框架：是 Apollo 的操作环境，一个定制版的 ROS. ROS 根据功能将自治系统划分为多个模块. 每个模块负责接收，处理和发布自己的消息．由于这些模块相互独立，只能通过运行时框架进行通信. (定制：**共享内存** – 降低了需要访问不同模块时的数据复制需求，加快通信速度．**去中心化** – 解决了单点故障问题．现成的 ROS 由许多节点组成，需要由 ROS Master 节点管理. Apollo 将所有节点放在一个公共域中，域中每个节点都有关于域中其他节点的信息．公共域取代了原来的 Master 节点，消除了单点故障风险．**数据兼容性** – 用 protobuf 接口语言替代 message，解决后向兼容性的问题，有助于长期发展)

应用程序模块层：包括 MAP 引擎，定位，感知，规划，控制，端到端驾驶和人机接口．每个模块有自己的算法库.





Apollo 云服务包含高精度地图，仿真环境，数据平台，安全，空中软件升级，智能语音系统.

地图 – 定位，感知，规划

定位

高精度地图包含大量的驾驶辅助信息，最重要的是道路网的精确三维表征．还包含很多语义信息．

HD Map 最重要的特征之一是精度．厘米级

传感器查找地标，车辆将 HD 地图上的已知地标进行比较(预处理 – 消除了不准确或质量差的数据，坐标转换 – 将来自不同视角的数据转换为统一的坐标系，数据融合 – 将来自各种车辆和传感器的数据合并)

感知

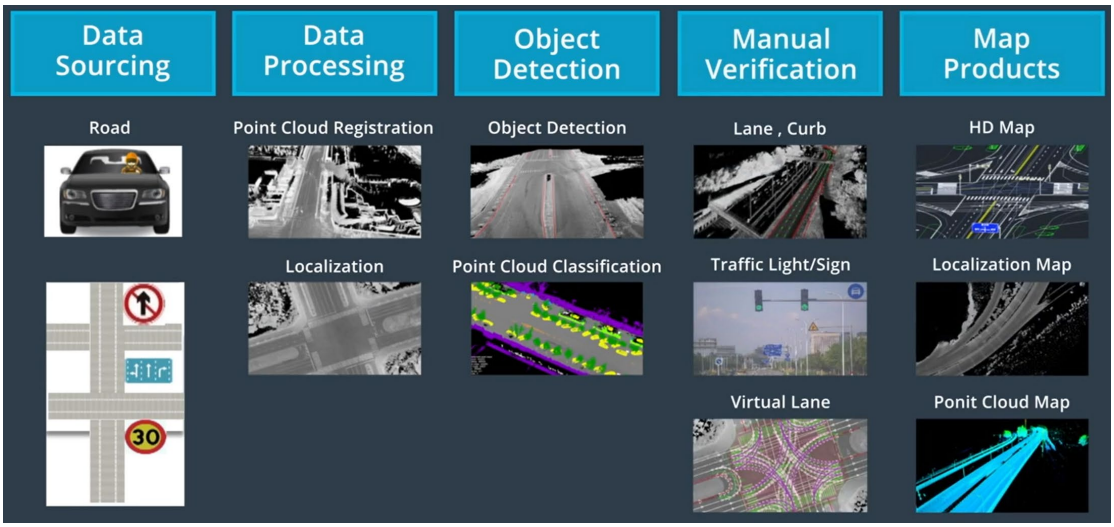
距离限制，恶劣天气，遮挡 限制识别障碍物

缩小检测范围，感兴趣区域 ROI(提高检测精确度和速度，节约计算资源)

规划

帮助找到合适的行车空间(识别车道中心线)，帮助规划器确定不同的路线选择(障碍物，减速带，人行横道)，帮助预测软件预测道路上其他车辆在将来的位置

HD Map 的构建包括：数据采集，数据处理，对象检测，手动验证和地图发布．

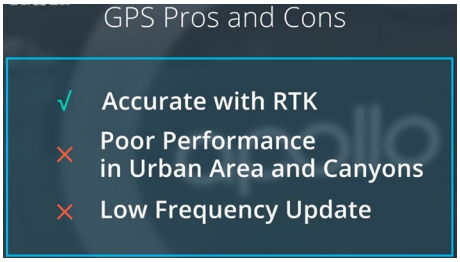


定位

二维地图 三角测量

GPS 至少 4 颗卫星

实时运动定位 RTK



惯性导航 IMU(加速度计 accelerometer, 陀螺仪 gyroscope), 以高频率更新, 可达 1000Hz, 因此可提供接近实时的位置信息. 缺点在于其运动误差随时间增加

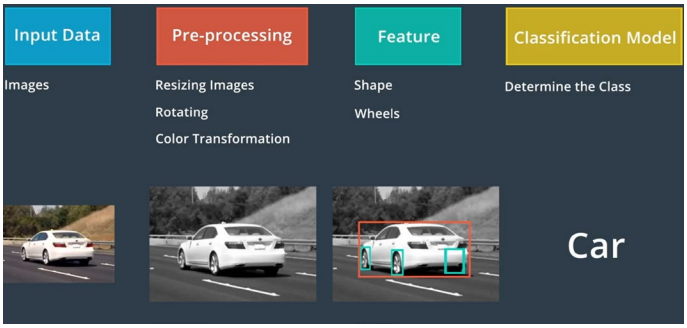
GPS+IMU: IMU 弥补了 GPS 更新频率较低的缺陷, GPS 纠正了 IMU 的运动误差.

LiDAR 通过点云匹配来定位. ICP 算法, 滤波算法(Histogram, Kalman 滤波). 优点在于其 robustness. 缺点在于构建 HD 地图并使其保持最新.

视觉定位通常与其他传感器数据相结合. 优点在于图像容易获得. 缺点是缺乏三维信息和对三维地图的依赖.

感知
检测(找出物体在世界中的位置) 分类(对象是什么) 跟踪(随时间推移观察移动物体, 如其他车辆, 行人, 自行车) 语义分割(将图像中的每个像素与语义类别进行匹配)

分类器分类的步骤



规划
通过结合 HD 地图, 定位和预测来构建车辆轨迹.
规划的第一步是路线导航, 侧重于如何从地图上的 A 前往 B. 将地图数据作为输入, 输出可行驶路径. 目标是生成免碰撞和舒适的可执行轨迹. 该轨迹由一系列点定义, 每个点都有一个关联速度和一个指示何时应抵达那个点的时间戳.

路线规划有 3 个输入. 地图, 当前位置, 目的地.

搜索路线时, Apollo 把地图数据重新格式化为“图形 graph”的数据结构. 该图形由节点 node 和边缘 edge 组成. 节点表示路段, 边缘代表这些路段之间的连接. 好处在于有许多在图形中查找路径的快速算法.

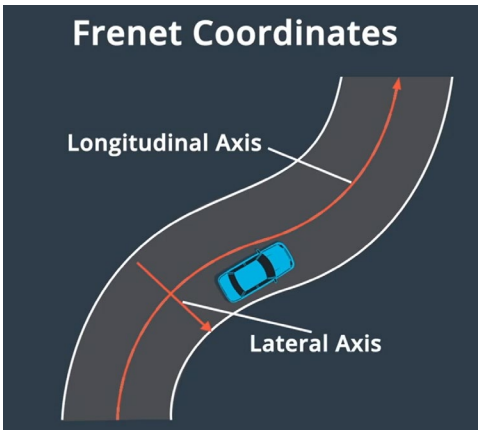
A*是经典的路径查找处理算法. g 值是从起始点到候选节点的 cost. h 值是从候选节点到目的地的估计 cost. f 值为两者相加. 取具有最低 f 值的候选节点.

高等级地图路线只是规划过程的一部分. 低等级轨迹需要处理不属于地图的物体, 如行人和其他车辆. 这些场景需要更低级别, 更高精确度的规划, 即轨迹生成.

轨迹生成的目标是生成由一系列路径点所定义的轨迹. 我们为每个路径点分配了一个时间戳和速度, 并用一条曲线与这些路径点拟合, 生成轨迹的几何表征. 由于移动的障碍物可能会暂时阻挡部分路段, 我们可将每个点上的时间戳与预测模块的输出相结合, 以确保在我们计划通过时轨迹上的每个路径点均未被占用. 每个点上的速度用于确保车辆按时到达每个路径点.
3D 轨迹 = 2D 位置 + 时间

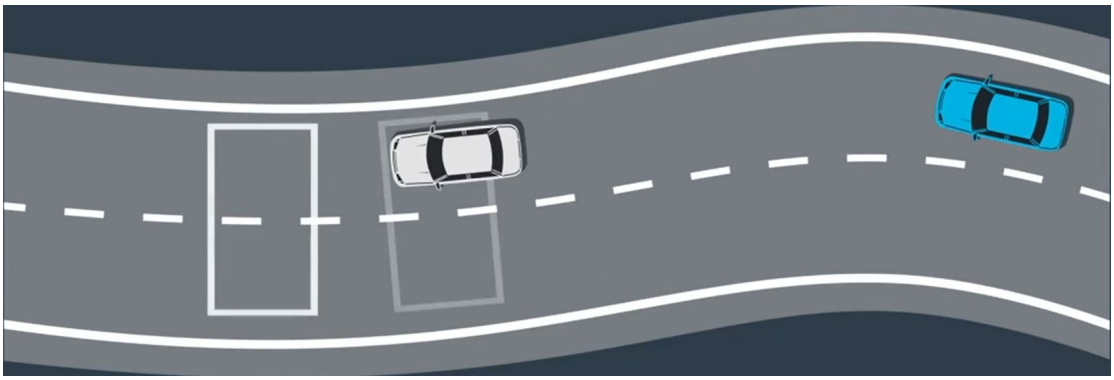
无碰撞 + 舒适(路径点的过渡以及速度的变化必须平滑) + 路径点对车辆实际可行 + 合法
最佳轨迹: 成本函数 由各种犯规处罚组成(deviation/collisions/speed limit/comfort) 不同场景下有不同的成本函数.

笛卡尔坐标系的替代方案是 Frenet 坐标系. 它描述了汽车相对于道路的位置. s 代表沿道路的距离, 是纵坐标, 表示在道路中的行驶距离. d 表示与纵向线的位移, 是横坐标, 表示汽车偏离中心线的距离.

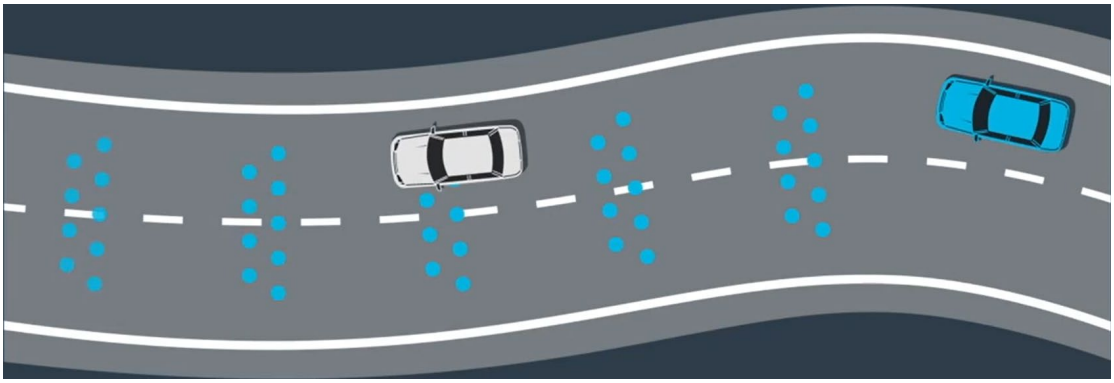


路径-速度解耦规划将 trajectory planning 分为两步: path planning 和 speed planning.
在 path planning 中生成候选曲线, 使用成本函数(平滑度, 安全性, 与车道中心的偏离)对每条路径进行评估, 确定 cost 最低的路径. 下一步是确定与路径点相关的一系列速度, 即“速度曲线”. 用“优化”功能来确定受到各种限制的良好速度曲线.

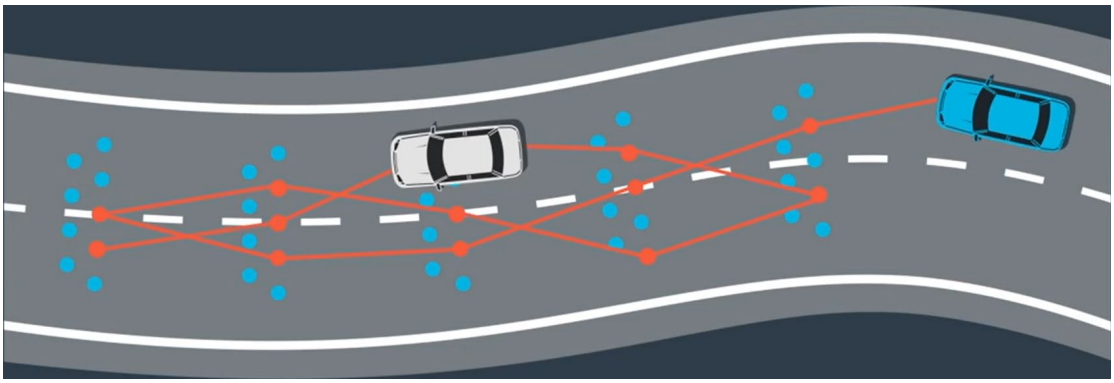
Path generation
1. 将路段分割成单元格



2. 对单元格中的点进行随机采样



3. 随机连接这些点，生成路径

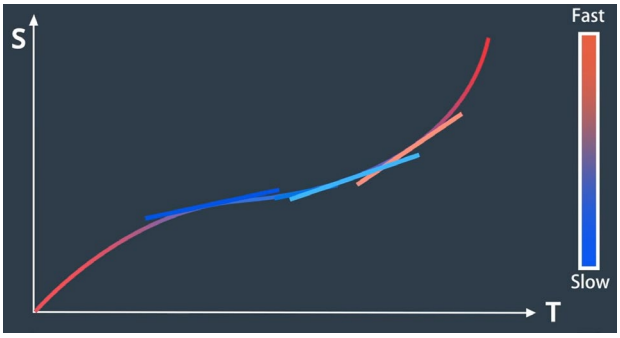


4. 用成本函数评估候选路径

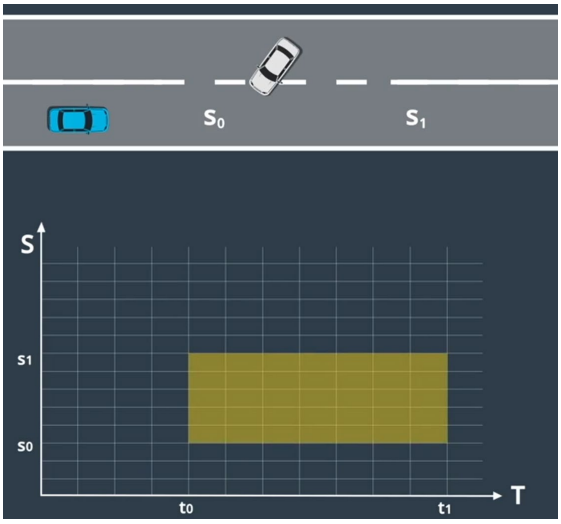
Cost function could take into account:

- Deviation from the center of the lane.
- Distance from obstacles.
- Changes in speed and curvature.
- Stress on the vehicle.

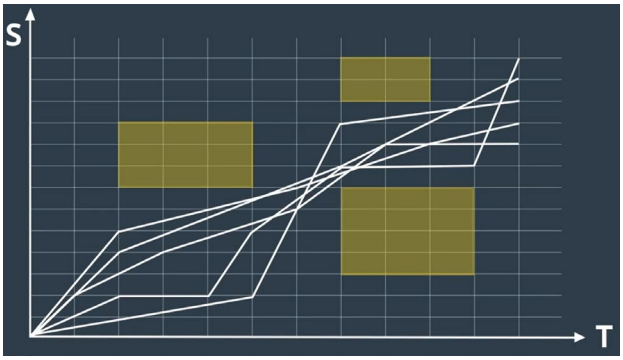
选择与选定路径相关联的速度曲线 – ST graph 可帮助我们设计和选择速度曲线
S 表示纵向位移, T 表示时间. 斜率则表示速度.



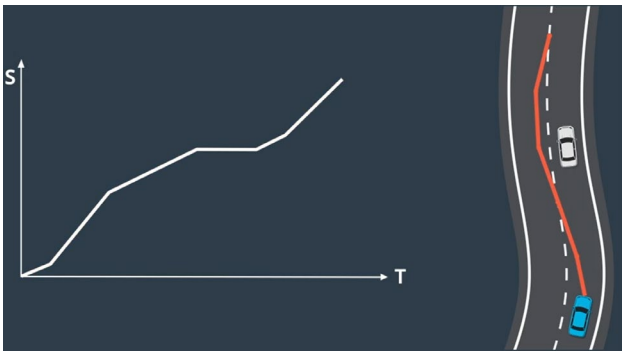
1. 为构建最佳速度曲线，需要将 ST 图离散为多个单元格. 每个单元格内速度保持不变(为了简化).
2. 预测模块判断障碍物在哪个时间段占据路线.



3. 使用优化引擎为该图选择最佳的速度曲线. 优化算法通过复杂的数学运算来搜索受到各种限制的低成本解决方案.

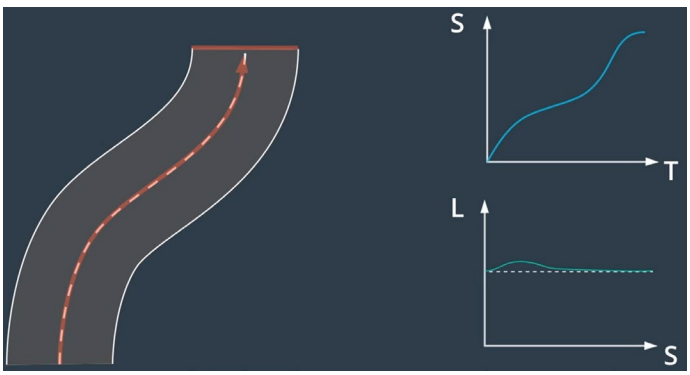


4. 得到不平滑的轨迹.

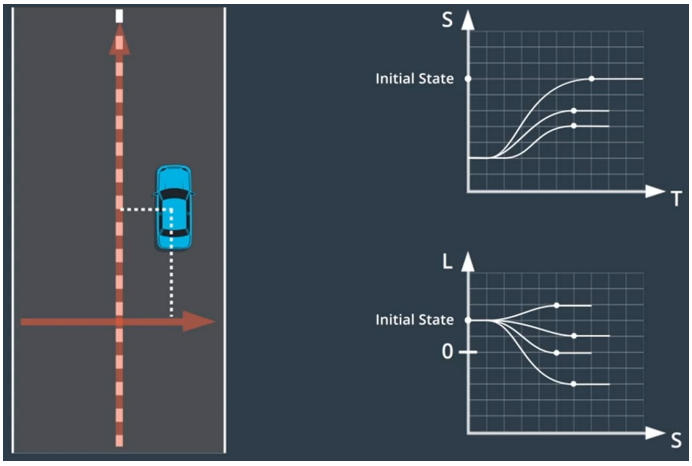


5. 使用二次规划 quadratic programming 技术. 将平滑的非线性曲线与这些分段式线段拟合.

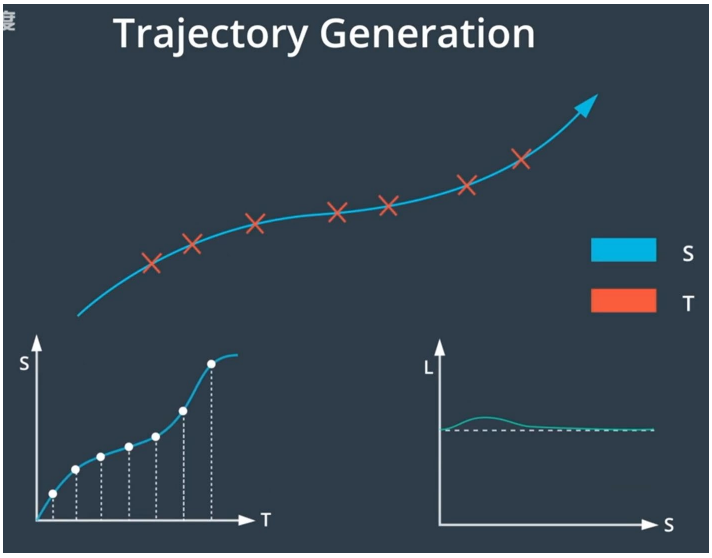
我们的目标是生成三维轨迹(纵向维度, 横向维度和时间维度). 分离轨迹的纵向和横向分量来把三维问题分解为 2 个单独的二维问题.



Lattice 规划分别建立 ST 和 SL 轨迹，然后合并.



ST 轨迹的终止状态:
车辆状态分为 3 组：巡航 cruising，跟随 following，停止 stopping
巡航：车辆将在完成规划步骤后定速行驶.
跟随：与前车保持安全距离，速度和加速度取决于跟随的车辆.
停止：速度和加速度被修正为 0.
SL 轨迹的终止状态：车辆应稳定地与车道中心线对齐. 我们想要的候选轨迹应以车辆与车道对齐并直线行驶而结束. 为达到这样的状态，车的朝向和位置的一阶和二阶导数都应为 0
一旦我们同时拥有了 ST 和 SL 轨迹，需要转换为笛卡尔坐标系(通过 ST 和 SL 中共有的 S 值进行匹配来合并轨迹)，然后构建由二维路径点和一维时间戳组成的三维轨迹.



控制是驱使车辆前行的策略. 最基本的控制输入为转向，加速和制动.
控制器使用一系列路径点来接收轨迹. 控制器必须准确，具备可行性，平稳度.
我们的目标是使用可行的控制输入，最大限度地降低与目标轨迹的偏差，最大限度地提高乘客的舒适度.
三种控制策略: PID 控制，线性二次调节器 LQR，模型预测控制 MPC.

控制器预计有 2 种输入：目标轨迹(来自规划模块)和车辆状态(通过本地化模块来计算，包括速度，转向和加速度).

PID 控制只需知道车辆与目标轨迹有多大的偏离.

P 控制器在车辆开始偏离时立即将其拉回目标轨迹. 比例控制意味着车辆偏离越远, 控制器就越难将其拉回. 问题在于它很容易超出参考轨迹.

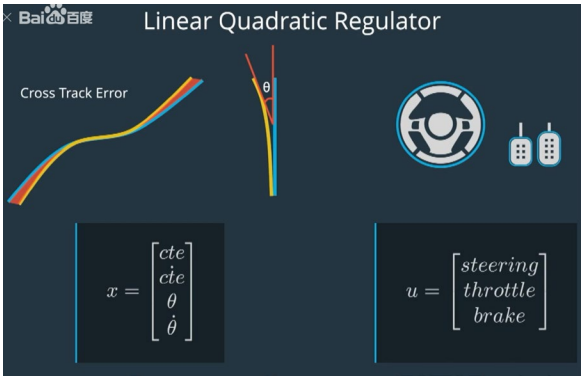
当车辆越来越接近目标轨迹时, 我们需要控制器更加稳定. D 控制器致力于使运动处于稳定状态, 即阻尼项, 可最大限度地减少控制器输出的变化速度.

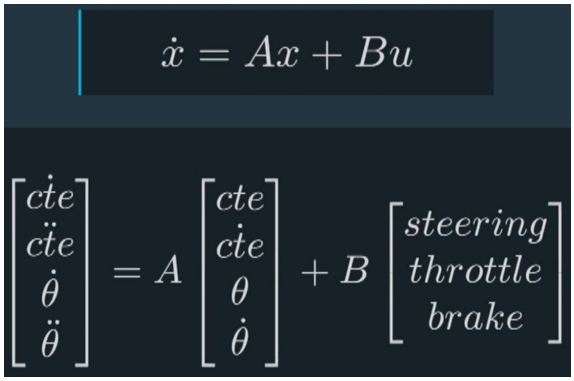
I 控制器负责纠正车辆的任何系统性偏差. 控制器会对系统的累积误差进行惩罚.

$$a = -K_p e - K_i \int e dt - K_d \frac{de}{dt}$$

但是 PID 只是一种线性算法, 对于复杂的系统而言是不够的. 很难将横向和纵向控制结合起来. 另一个问题在于 PID 依赖于实时误差测量, 意味着受到测量延迟限制时可能会失效.

LQR 是基于模型的控制器, 使用车辆的状态来使误差最小化. Apollo 使用 LQR 进行横向控制. 横向控制包含 4 个组件: 横向误差, 横向误差的变化率, 朝向误差, 朝向误差的变化率. 车辆有 3 个控制输入: 转向, 加速, 制动.





Q 是为了最小化误差. 但也希望尽可能少地使用控制输入, 由于会有成本(汽油, 电).

$$cost = \int_0^\infty (x^T Q x + u^T R u) dt$$

MPC 是一种更复杂的控制器, 非常依赖于数学优化. 三个步骤: 建立车辆模型, 使用优化引擎计算有限时间范围内的控制输入, 执行第一组控制输入.

MPC 是一个重复过程. 不断优化. 在每个时间步不断重新评估控制输入的最优序列.

MPC 考虑了车辆模型, 因此比 PID 更精确. 也适用于不同的成本函数, 因此可以在不同情况下优化不同的成本. 但是更复杂, 更缓慢, 更难以实现. MPC 是非常重要的无人驾驶车控制器.